UiO : Department of Informatics
University of Oslo

Developing Real-Time Collaborative Editing Using Formal Methods

Lars Tveito

Master's Thesis Autumn 2016

Developing Real-Time Collaborative Editing Using Formal Methods

Lars Tveito

August 2016

Abstract

Real-time collaboration allows multiple users to view and edit a document simultaneously over a network. In this thesis, we develop a new protocol, called Shared Buffer, which enables real-time collaboration in existing editors. Shared Buffer leverages a client-server architecture and minimizes the implementation effort of the client-side algorithm. It achieves this without degrading the responsiveness of the editor.

The greatest challenge of a real-time collaborative system is ensuring consistency between the distributed copies of the document. We chose eventual consistency as the consistency model, which essentially states that if all users stop typing, then eventually they will look at the same document.

We apply a formal verification technique called model checking, using it as a tool to validate the protocol. The behavior of the system is formally specified in Maude, a language based on equational and rewriting logic. Linear Temporal Logic (LTL) is used to formalize the consistency model. Using the Maude LTL model checker, we have verified that the system exhibits eventual consistency for a limited number of clients and operations.

A Shared Buffer server has been implemented in Clojure, a modern functional language with strong support for concurrency. Client implementations have been developed as an extension for Emacs, a widely used text editor, and as a library for the Python programming language.

Acknowledgements

When I was looking for a thesis project, I got to talk with Rudi. It did not take more than five minutes to realize that he would be a great supervisor. Throughout the process he has shown genuine interest in the project and given me both the freedom I wanted and the guidance I needed. Martin has been a tremendous help. He has given me the theoretical foundations that I sorely needed, both as my supervisor and as a lecturer. He is a bottomless source of knowledge, and I am convinced he has read every paper on theoretical computer science since the early 70's. I would like to convey my deepest gratitude to Rudi and Martin.

Throughout my years at the Department of Informatics, I have worked as a teaching assistant and been actively involved in the department's student council. These arenas are simply packed with fantastic people that I have been lucky enough to get to know. A thanks goes out to Joshi, who have played a significant part in encouraging me to pursue these roles. I want to thank the students who attended my lessons for letting me have their attention; it has been a wonderful experience for me, and hopefully for the students too. Furthermore, I would like to thank the many faculty members who actively work to make the Department of Informatics a fantastic place to study, and not shying from discussions with the students where improvement is due. I would like to thank my wonderful friends from the student council (FUI), who made my life at university a joy. There are simply too many to thank. This has made the university a great place to be.

Thanks to the nano-gang, who I started my time at the university with and still visit regularly on the fourth floor. I especially want to thank Jarle and Kai for having long discussions about the thesis.

For the last couple of years, I have spent a fair share of my time with Tore, Carl Martin and Sigurd, constantly having excited discussions of both a fruitful and completely useless nature. I've thoroughly enjoyed it, and would like to them to know that I kind-of love them.

Finally, I want to thank my wonderful family. My sisters, Lisa and Julie, for being the coolest kids I know, my father for being the one I can always turn to for sound advice, my mother for being the most loving and caring person in the world and finally, my brother, Vegard, for being my favorite person.

Thank you
Lars Tveito

Contents

List of Figures

List of Tables

Preface

In 2013 LispNYC[1], ClojureNYC[2] and Association of Lisp Users[3] hosted a programming competition called LISP In Summer Projects[4], awarding cash prizes for Lisp-related projects. They gathered some great finalist judges: Matthias Felleisen, Richard Gabriel, Rich Hickey, Peter Norvig, Christian Queinnec and Taiichi Yuasa.

Around the time the competition was announced, I attended a course on functional programming, in which we were to collaborate on programming assignments in small teams. Mostly, we programmed together while in the same room, and at any given moment the one with the most promising idea used the keyboard. Sometimes we all wanted to explore some idea, and we would have to type it out on separate computers, and synchronize our changes afterwards.

Seeing the aforementioned competition announcement, I immediately decided that implementing real-time collaboration in Emacs would make the perfect project. A program was developed using a trial and error approach, and by the end of the summer I had a rough prototype, consisting of an Emacs client and a server written in Common Lisp. It did not handle concurrent edits, which poses the greatest algorithmic challenge for a real-time collaborative editing system.

Still, the judges of the competition deemed the program worthy of second prize, awarding me $500 for my efforts. The project description submitted to the competition can be found in at the Lisp in Summer Projects website[5], and the source code for the project is hosted on Github[6].

This thesis is a continuation of the work I did on that project, but with a different focus. The aim is to find a way to gracefully handle concurrent edits, without adding complexity to the client side algorithm. Having experienced that detecting bugs in such a system was a challenging endeavor, lead to the decision of leveraging formal methods to ensure correctness for the algorithm.

[1] http://lispnyc.org
[2] http://www.meetup.com/Clojure-NYC/
[3] http://alu.org
[4] http://lispinsummerprojects.org/
[5] http://lispinsummerprojects.org/static/summer/231030-sharedbuffer.pdf
[6] https://github.com/larstvei/shared-buffer-lisp-in-summer-projects

Chapter 1

Introduction

A real-time collaborative editor enables multiple users, at (potentially) different locations, to work on the same document *simultaneously*. The idea has been around for a long time, and was demoed by Douglas Engelbart in the Mother of All Demos [12] already in 1968. In this thesis we leverage formal methods to aid the development of a real-time collaboration system.

First, let us make it clear what we will characterize as a real-time collaborative editor. An *editor* is a program that allows a user to manipulate a plain text document. The editor is considered *collaborative* if it has features that makes it easier for multiple users to form a document together. Furthermore, it is considered *real-time* if multiple users can edit the document simultaneously [10], and that changes are propagated to the other users within a reasonably short amount of time.

The first actual implementation of such an editor that was documented in the literature, came with GROVE (GROUP Outline Viewing Editor) [10], which introduced the concept of *Operational Transformation* (OT) which we will cover in Chapter 3. Previous systems [17, 14, 25] that provided collaborative capabilities were not considered *real-time* systems by [10], because users could not edit the same *section* of the document simultaneously.

The reader might be familiar with Google Docs[1], which is an example of a modern real-time collaborative document WYSIWYG (What You See Is What You Get) editor with over 240 million monthly active users [31]. It is largely based on OT [7], but has also initiated research in a technique called Differential Synchronization [16]. Through its integration with Google Drive[2], it offers a collaborative platform, where users can both store and manipulate their documents, as long as they have an internet connection and a (fairly modern) browser.

We present a tool for real-time collaborative editing called Shared Buffer, which is designed with developers in mind. What most developers have

[1] https://www.google.com/docs/about/
[2] https://www.google.com/drive/

in common is that they spend a lot of time manipulating *plain text*, yet they use a lot of different tools to do so [8]. We therefore aim at enabling real-time collaboration in *existing* text editors, as opposed to developing an editor with real-time collaborative features. As a means to this end, we develop a protocol which, ideally, should be portable to any text editor, or any program that embeds a text editor.

A client-server model is chosen, as opposed to a fully decentralized solution. We have tried to move complexity to the server whenever possible, if this simplifies the client-side algorithm. Furthermore, proving correctness for a fully decentralized solution has proven to be very difficult [19, 32, 18].

Our prototype client is for the text editor Emacs. The name Shared Buffer reflects a choice in design; in Emacs, text is stored in a *buffer*; when a file is opened, its contents is put inside a buffer which the user can manipulate. You may also have buffers that are not associated with any file. In Shared Buffer, there is no notion of a file, meaning there is no centrally stored copy of the document.

The server is written in a dialect of Lisp called Clojure[3], a modern functional programming language with strong concurrency semantics [11]. Being hosted on the JVM, Clojure offers full Java interoperability, meaning that we can leverage the vast collection of Java libraries.

1.1 The Naïve Algorithm

Let us now consider two cases that illustrates how a naïve implementation might work, and where it fails to produce a desirable result.

Say we have two users, u_0 and u_1 who are located in different countries; both are communicating with a server S. They each have a copy of a shared buffer. Both may either insert a character, or delete one from the buffer, and they may do so at any time. When a user performs an operation (meaning performing an insertion or deletion) on its local buffer, then this should be communicated to S. When S receives an operation, it should communicate this to the other user.

We represent scenarios that can occur in the system graphically by using a variation of message sequence charts. The diagrams are read from top to bottom with regards to time, where directed edges represents the transfer of a message.

Figure 1.1 describes a very simple scenario. Imagine that u_0 has an empty buffer which she precedes to insert an `a` into. Then u_1 inserts a `b` in front of the `a` that just showed up in her buffer. The `b` eventually reaches u_0, and the end result of the interaction is that they both will be looking at a buffer

[3] https://clojure.org/

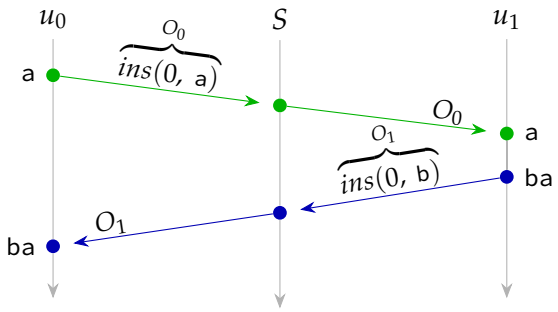

Figure 1.1: A conflict-free scenario with two clients.

containing `"ba"`. In this scenario the buffers ended up identical, so we say that we have reached a *consistent state*.

Simple scenarios like the one we saw, where only one message is "in flight" at any one time, would be gracefully handled even by the naïve approach. We can see that `a` was inserted prior to the `b` at both u_0 and u_1, hence they cannot have been applied concurrently. We will now demonstrate that the approach does not work when we introduce concurrent edits.

Let us return to the example from Figure 1.1, with a slight modification, visualized in Figure 1.2. The scenario is unchanged at u_0, where she first inserts an `a`, and later receives the `b` which leaves her with a buffer containing `"ba"`. Now say that u_1 inserts `b` *before* having received the `a`. When she has already typed a `b`, she receives a message saying that she should place an `a` at the first point in her buffer. The resulting buffer is `"ab"`. Now they are looking at different buffers, so we say we have reached an *inconsistent state*.

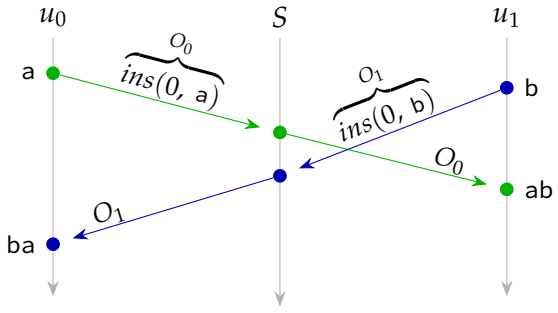

Figure 1.2: A minimal conflict with two clients.

In this thesis we introduce and discuss a new protocol which guarantees *eventual consistency* [40] between participating clients. Intuitively, this means that if all users stop typing at some point, then given enough time for traveling messages to reach their destination, they will all be looking at the same buffer.

1.2 Goals

The goal of this thesis is to create a protocol for real-time collaborative editing which can be characterized as:

1. **Portable** — easy to implement in an *existing* plain text editor,
2. **Responsive** — regular usage of the editor should not be degraded by enabling real-time collaboration,
3. **Robust** — gracefully handle all conflicting editing operations.

We would characterize the naïve algorithm as *portable* and *responsive*, but in no way *robust*. As we just demonstrated, it does not handle concurrent editing operations correctly.

The majority of existing *Operational Transformation* (OT) algorithms (presented in Chapter 3) can be considered *responsive* and *robust*, but not very *portable*, as all clients must implement control mechanisms to handle conflicts. It is fair to say that a fully distributed solution can be more *robust* than one with a centralized server, as this server is a single point of failure.

A fully synchronous solution (i.e. one where only one user may type at the time) would require some form of distributed locking scheme. It can be considered *portable*, given that the locking scheme is easy to implement. It is *robust* because conflicting editing operations are simply not allowed; however, it does have a single point of failure. It is not *responsive* as the user might have to wait before being able to type.

Our goal is to find the intersection these three characteristics, where we end up with an algorithm that is both *portable* like the naïve algorithm, *responsive* like the naïve or OT algorithms and *robust* like synchronous or OT algorithms. To the best of our knowledge, no such algorithm has been documented in the literature.

1.3 The Shared Buffer Algorithm

At the client side, the Shared Buffer algorithm is very similar to the naïve algorithm. The main difference is that messages sent to the server contains two numbers in addition to what editing operation was just performed. One number provides the server with sufficient information to resolve conflicts, while the other is a sequence number. The sequence number scheme (Section 5.5.1) allows a client to easily detect when to *reject* an incoming message. In a diagram, we can see that a message will be rejected if two lines between the server to a client *overlaps*.

To give a basic idea of how Shared Buffer handles conflicting editing operations, we will now revisit the example from Section 1.1, where the naïve algorithm failed to ensure consistency. The scenario is illustrated in Figure 1.3.

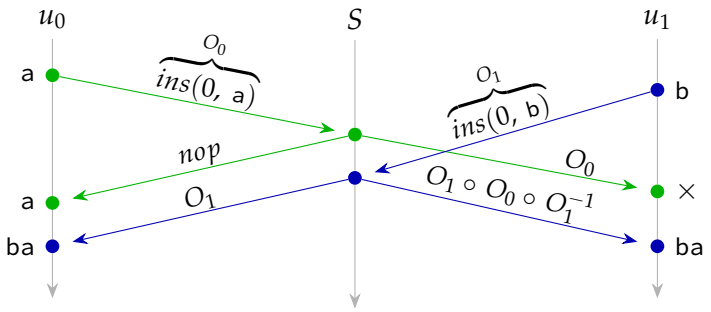

Figure 1.3: A minimal conflict resolved by Shared Buffer.

From the perspective of u_0, the situation has hardly changed. First, u_0 inserts an `a` at the first point in the buffer, and communicates this to the server S. Next, she receives a message, saying not to do anything. Finally, a message saying that a `b` should be inserted at the first point in the buffer is received, after which the resulting buffer is `"ba"`. The only difference here is that a message, saying not to do anything, was received.

There are two changes at u_1. First u_1 inserts a `b` at the first point in the buffer. Second, the message saying that an `a` should be inserted at the first point is *rejected*, due to a mismatch of sequence numbers (notice that the lines between S and u_1 overlaps in Figure 1.3). Finally, a new message is received, which says to first *undo* the `b` that was added, then insert an `a` at the first position in the buffer, and finally, insert a `b` at the first position of the buffer. This leaves u_1 with the buffer `"ba"`, which is the same as u_0, and we say we reached a *consistent* state.

More generally, the algorithm essentially consists of the server receiving operations, and adding them to a history that dictates the order of which operations must be applied. Then for every incoming message, it constructs an operation for every client that will make the client consistent with the history that has been constructed. The idea is simple, but as we will see in Chapter 5, it is not at all trivial to construct such a history, or make the clients consistent with the constructed history. In order to develop this algorithm, we have relied heavily on the use of *formal methods*, which has shed light on the many complexities that may arise.

1.4 Method

In this thesis we use a formal verification technique called *model checking* [3]. This technique requires us to obtain a formal model of the system we wish to validate. A model is represented as a set of states, and transitions between these states. We can think of model checking as a graph search, where the states acts as nodes, and edges represent the possibility of going from one state to another. If the graph is finite, we can prove that the model has a certain property by checking whether the property holds true in every

state. Furthermore, we want to use Linear Temporal Logic (LTL) to express properties over paths, which are sequences of states.

The model is an abstraction of a given system, where one carefully chooses what parts of the system is necessary to represent, in order to prove the properties that are of interest.

Moreover, we use the model as a way of driving the development process, or rather, solving the problem. When model checking a property, a counterexample is given if the property does not hold. By studying the example, we can change the model in hope of resolving the issue, and see the effects of the change. This resembles Test Driven Development (TDD), but instead of testing our actual system we perform tests on a model, and rather than testing a few selected scenarios, we check all possible scenarios.

The Maude System[4] is our chosen modeling language and verification tool. It provides an expressive language, that is well suited for modeling concurrent and distributed systems [5]. In addition, it provides an LTL Model Checker [9], which allows us to specify and verify LTL properties.

1.5 Contributions

The main contribution of this thesis is a protocol that enables real-time collaborative editing. Both a client- and server-side algorithm has been formally specified and implemented. The specification has been formally verified to guarantee eventual consistency for a limited number of clients and operations.

In the process we have:

- formally specified the system in Maude,
- validated the system via model checking using the Maude LTL model checker,
- provided a client-side implementation as an extension for Emacs,
- provided a prototype server-side implementation in Clojure.

In addition, the thesis demonstrates how formal specification and model checking can drive the development of an algorithm.

1.6 Chapter Overview

Chapter 2 introduces notation for operations which we will rely heavily on throughout the thesis. Our algorithm leverages both the composition of operations and the inverting of operations; we show some properties

[4] http://maude.cs.illinois.edu/

about operations and their composition to get a deeper understanding of the most fundamental structure this thesis is concerned with.

Chapter 3 covers related work. We introduce *Operational Transformation* (OT), and briefly discusses some OT-based algorithms. Some work has been done on formally verifying transformation functions, and we try to convey the results of this work. Lastly, we show how Shared Buffer compares to other solutions.

Chapter 4 contains a short introduction to Maude, and shows how we can use Maude to formally specify the behavior of the system. In particular, it shows how the *clients* in the system are specified.

Chapter 5 shows how the server in the system is specified. By walking through the most essential parts of the specification, we describe the server-side algorithm in detail and show *why* each control mechanism is put in place.

Chapter 6 shows how we express the chosen consistency model (eventual consistency) has been formalized as an LTL formula, and how we have leveraged the Maude LTL Model Checker.

Chapter 7 describes the general structure of the implementation, and how the protocol has been implemented. Here, the focus is on the remaining problems not fully covered specification.

Chapter 8 propose some interesting work that could be pursued in the future, and concludes the thesis by comparing our results with the goal we set for the thesis.

1.7 Project Source Code

All source code from the thesis can be found on Github[5].

[5] https://github.com/larstvei/master

Chapter 2

Formal Semantics of Editing Operations

A model of a given system is an abstraction of that system [22], which means only some aspects of the system are described. In our case, the fundamental capabilities of a text editor, namely the insertion and deletion of characters in a buffer, should be captured, along with the order in which they are performed. The time between operations is an example of something *not* represented in the model; as a result the model cannot be used to analyze the real time performance of the system. Other features of a text editor, like "search and replace", are also omitted, because such features can be represented as a series of deletions and insertions.

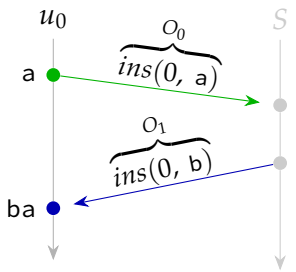

Figure 2.1: Our focus is on the operations of a single user.

In this chapter we introduce a formal definition of editing operations and their semantics. Operations are fundamental for all *Operational Transformation* (OT) algorithms, as well as the Shared Buffer algorithm. To get a deeper understanding of the operations we study their algebraic properties, which simplifies the process of both specifying and implementing them. Furthermore, the Shared Buffer heavily relies on the ability to take the inverse of an operation, which motivates Section 2.3.1.

The definitions are based on [10, 32]; we specify them formally and define the semantics of editing operations as a set of equations. This chapter is only concerned with events at a single client. We assume that every event is simply an operation being applied, and do not differentiate between an operation originating locally or remotely.

2.1 Operations and Buffers

The operations we are concerned with is the *insertion* and *deletion* of a character in a buffer.

Definition 2.1.1 (Operations)**.** The set $\mathcal{O}$ is inductively defined as the smallest set such that the following holds:

- $nop \in \mathcal{O}$,
- $ins(i,\ c) \in \mathcal{O}$ for any $i \in \mathbb{N}$ and $c \in Unicode$,
- $del(i) \in \mathcal{O}$ for any $i \in \mathbb{N}$,
- for any two $O_i, O_j \in \mathcal{O}$ then $O_j \circ O_i \in \mathcal{O}$. ⊣

The semantics of an operation is defined in terms of how it is applied to a buffer, where a buffer is simply defined as a 0-indexed string of UTF-8 encoded characters. The set $Unicode$ is our alphabet, which contains every character defined by the Unicode standard [6]. We let $\mathcal{B}$ constitute the set of all possible buffers — this set could also be expressed as $Unicode^*$.

An operation can be applied to a buffer, which in turn yields a new buffer. Consequently all $O \in \mathcal{O}$ are partial unary operations $O : \mathcal{B} \to \mathcal{B}$. The operations are partial because a given operation cannot necessarily be applied to all buffers; as an example, no delete operation can be applied to the empty buffer ϵ. We assume that no text-editor are able to produce an operation which is ill-defined on its current buffer.

Definition 2.1.2. (Semantics of Operations). Let $B \in \mathcal{B}$. The nop operation is the operation that does nothing, and applying it is defined as:

$$nop(B) = B$$

Let $i \in \mathbb{N}$, and both $c, c' \in Unicode$. We let a single space represent concatenation, where characters are treated like strings of length one. Applying an insertion is then defined as follows:

$$ins(0, c)(B) = c\ B$$
$$ins(i+1, c)(c'\ B) = c'\ ins(i, c)(B)$$

Similarly, applying a deletion is defined as:

$$del(0)(c\ B) = B$$
$$del(i+1)(c\ B) = c\ del(i)(B)$$

Let $O_i, O_j \in \mathcal{O}$, and let $O_j \circ O_i$ represent the *composition* of O_i and O_j. Applying a composed operation to a buffer is defined as:

$$O_j \circ O_i(B) = O_j(O_i(B))$$

⊣

Note that composition of operations is no different from regular function composition.

2.2 Scenarios Described in Terms of Operations and Their Application

In the previous section we formalized

- what operations are,
- how operations are applied to buffers, and
- how operations are combined.

Let us try to bridge the gap between the formal notion of an editing operation, and scenarios that involves a user typing on a keyboard. Imagine that a user types the word `"hello"` — this is modeled as a single operation:

$$ins(4, \texttt{o}) \circ ins(3, \texttt{l}) \circ ins(2, \texttt{l}) \circ ins(1, \texttt{e}) \circ ins(0, \texttt{h})$$

The result of applying the operation to the empty buffer ϵ evaluates to the buffer that only contains the word `"hello"`, and can be calculated as so:

$$\begin{aligned} ins(4, \texttt{o}) \circ ins(3, \texttt{l}) \circ ins(2, \texttt{l}) \circ ins(1, \texttt{e}) \circ ins(0, \texttt{h})(\epsilon) &= \\ ins(4, \texttt{o}) \circ ins(3, \texttt{l}) \circ ins(2, \texttt{l}) \circ ins(1, \texttt{e})(\texttt{"h"}) &= \\ ins(4, \texttt{o}) \circ ins(3, \texttt{l}) \circ ins(2, \texttt{l})(\texttt{"he"}) &= \\ ins(4, \texttt{o}) \circ ins(3, \texttt{l})(\texttt{"hel"}) &= \\ ins(4, \texttt{o})(\texttt{"hell"}) &= \texttt{"hello"} \end{aligned}$$

Now we will expand from the case where a single user types on a keyboard, and include operations that can be received from a server. In the scenario best described by Figure 1.1 (page 3), we saw two operations $ins(0, \texttt{a})$ and $ins(0, \texttt{b})$, named O_0 and O_1 respectively, being applied in the same order at two different locations.

From the perspective of u_0:

- O_0 is generated locally,
- O_1 is received from the server.

From the perspective of u_1:

- O_0 is received from the server,
- O_1 is generated locally.

Common to both u_0 and u_1 is their initial buffer (the empty buffer ϵ) and the operation they apply is $O_1 \circ O_0$. Because they perform the same operation to the same initial buffer, they must necessarily end up in a consistent state (i.e. end up with the same buffer).

The scenario from Figure 1.2 (page 3) is almost identical to the scenario above, but the operations are applied in different orders.

From the perspective of u_0:

- O_0 is generated locally,
- O_1 is received from the server.

From the perspective of u_1:

- O_1 is generated locally,
- O_0 is received from the server.

u_0 and u_1 have the same initial buffer, but the composed operation of u_0 is $O_1 \circ O_0$ and the composed operation of u_1 is $O_0 \circ O_1$. By applying these operations to the empty buffer ϵ we show that u_0 and u_1 end up in an inconsistent state (i.e. end up with different buffers).

Operation applied by u_0:

$$\overbrace{ins(0,\ \texttt{b})}^{O_1} \circ \overbrace{ins(0,\ \texttt{a})}^{O_0}(\epsilon) = ins(0,\ \texttt{b})(\texttt{"a"}) = \texttt{"ba"}$$

Operation applied by u_1:

$$\overbrace{ins(0,\ \texttt{a})}^{O_0} \circ \overbrace{ins(0,\ \texttt{b})}^{O_1}(\epsilon) = ins(0,\ \texttt{a})(\texttt{"b"}) = \texttt{"ab"}$$

2.3 Algebraic Properties

An algebraic structure is a set along with one or more operations [1]. The set of operations $\mathcal{O}$ under composition $\circ : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$ forms an algebraic structure, denoted $\langle \mathcal{O}, \circ \rangle$.

The previous section contains a proof that $\circ$ is *not* commutative, meaning that $O_j \circ O_i = O_i \circ O_j$ is not the case for all $O_i, O_j \in \mathcal{O}$.

Proof. As examplified in the previous section:

$$ins(0, b) \circ ins(0, a) \neq ins(0, a) \circ ins(0, b)$$

□

The fact that $\circ$ is not commutative is precisely the problem with the naïve algorithm (Section 1.1); in other words, the naïve algorithm would guarantee eventual consistency if the order of which operations are applied does not affect the end result. In the next chapter we will introduce *Operational Transformation* which, at its core, is a technique for restoring commutativity for operations.

Furthermore, the structure $\langle \mathcal{O}, \circ \rangle$ is a *monoid* because it satisfies the following properties:

- nop is the identity element of $\circ$.

  Proof. Let $O \in \mathcal{O}$, then
  - $nop \circ O(B) = nop(O(B)) = O(B)$
  - $O \circ nop(B) = O(nop(B)) = O(B)$

  for any $B \in \mathcal{B}$. It follows that $nop \circ O = O \circ nop = O$. □

- $\circ$ is associative.

  Proof. Let $O_i, O_j, O_k \in \mathcal{O}$, then

- $((O_k \circ O_j) \circ O_i)(B) = (O_k \circ O_j)(O_i(B)) = O_k(O_j(O_i(B)))$
- $(O_k \circ (O_j \circ O_i))(B) = O_k((O_j \circ O_i)(B)) = O_k(O_j(O_i(B)))$

for any $B \in \mathcal{B}$. It follows that $O_k \circ (O_j \circ O_i) = (O_k \circ O_j) \circ O_i$. □

- $\circ$ is closed under $\mathcal{O}$.

Proof. By definition. □

There are two main reasons for noting these algebraic properties; one is that it is helpful when writing a formal specification in Maude, because Maude is an *algebraic* specification language; the other is that it helps when translating the structure to a given programming language, by making sure the selected representation preserves the properties of a monoid.

2.3.1 Invertibility

A *group* can be described as a monoid with *invertibility*, meaning every element in $\mathcal{O}$ has an inverse. More formally, for $\langle \mathcal{O}, \circ \rangle$ to be a group, it must satisfy that for any $O_i \in \mathcal{O}$ there exists a $O_j \in \mathcal{O}$ such that:

$$O_j \circ O_i = O_i \circ O_j = nop$$

The inverse of an operation $O \in \mathcal{O}$ is denoted O^{-1}, and so the equation can be restated as:

$$O \circ O^{-1} = O^{-1} \circ O = nop$$

Undo is a common feature in text editors, and should guide us in constructing an inverse function for $\mathcal{O}$. Intuitively it seems to satisfy the equation, in the sense that adding a character to a buffer, followed by an undo, is the same as having done nothing at all.

Guided by this intuition, the inverse of $ins(0, \texttt{a})$ should be $del(0)$, because applying $del(0) \circ ins(0, \texttt{a})$ to a buffer will always yield the same buffer. We can make the exact same argument for $ins(0, \texttt{b})$; its inverse should be $del(0)$. What should then be the inverse of $del(0)$? It cannot be both $ins(0, \texttt{a})$ and $ins(0, \texttt{b})$. If we were to choose one arbitrarily, then a user could suddenly experience that the character the inserted morphed in to another.

The problem is solved by extending the delete operations with what character is deleted, and so we redefine delete operations as so:

- $del(i, c) \in \mathcal{O}$ for any $i \in \mathbb{N}$ and $c \in Unicode$.

With the information of what character was deleted in the operation, we disambiguate what the inverse of a deletion should be. The inverse of $ins(0, \texttt{a})$ should be $del(0, \texttt{a})$, and the inverse of $ins(0, \texttt{b})$ should be $del(0, \texttt{b})$, where the inverse of each deletion should be $ins(0, \texttt{a})$ and $ins(0, \texttt{b})$ respectively.

Inverting composed operations is analogous with undoing multiple steps. Say a user types an `a` followed by a `b`, then undoing it would be to first delete the `b`, then delete the `a`. So for instance, the inverse of $ins(1,\ \texttt{b}) \circ ins(0,\ \texttt{a})$ should be $del(0,\ \texttt{a}) \circ del(1,\ \texttt{b})$.

Definition 2.3.1 (Inverse of an Operation)**.** The inverse of the *nop* element is the *nop* element itself:

$$nop^{-1} = nop$$

The inverse of an insertion of a character $c \in Unicode$ at position $i \in \mathbb{N}$, is the deletion of that character at that position:

$$ins(i,\ c)^{-1} = del(i,\ c)$$

Similarly for deletions:

$$del(i,\ c)^{-1} = ins(i,\ c)$$

For a composed operation $O_j \circ O_i \in \mathcal{O}$, the order of the operations is reversed, and the operations are inverted:

$$(O_j \circ O_i)^{-1} = O_i^{-1} \circ O_j^{-1}$$

$\dashv$

Now that we have defined an inverse for all operations, we can check if invertibillity holds. Say we have the operation $ins(0,\ \texttt{a})$, then its inverse is $del(0,\ \texttt{a})$. We apply $del(0,\ \texttt{a}) \circ ins(0,\ \texttt{a})$ to the empty buffer ϵ:

$$\begin{aligned} del(0,\ \texttt{a}) \circ ins(0,\ \texttt{a})(\epsilon) = \\ del(0,\ \texttt{a})(\texttt{"a"}) = \epsilon \end{aligned}$$

In order to satisfy the invertibility axiom, the reverse should be true as well. It is not because applying $ins(0,\ \texttt{a}) \circ del(0,\ \texttt{a})$ on the empty buffer ϵ, because it is not well defined. Consequently, the invertibility axiom does not hold, and so $\langle \mathcal{O}, \circ \rangle$ is not a group.

Inverting operations is an essential part of the Shared Buffer algorithm, and we rely on the definition above even though the invertibillity axiom does not hold. Notice that the counterexample $ins(0,\ \texttt{a}) \circ del(0,\ \texttt{a})(\epsilon)$ expresses that a deletion is applied to the empty buffer and then undone. It seems fair to question if that situation could really occur, because there is no reasonably defined way for an editor to perform the deletion in the first place.

We have to ensure that the algorithm never construct an operation that cannot be applied to a given client's buffer. We rely on the model checker to provide a counterexample, if we were to construct such an operation.

Chapter 3

Related Work

In this chapter we present some of the work of Ellis and Gibbs [10], the pioneers of *Operational Transformation* (OT) and the very interesting work of Imine et al. on proving correctness for transformation functions using formal verification techniques [20, 32, 19]. The chapter should sufficiently convey the basic idea of OT and how it works, without going into the finer details. We use notation established in Chapter 2 to describe the workings of OT.

3.1 Basics of Operational Transformation

Ellis and Gibbs introduced the dOPT (Distributed Operational Transformation) algorithm [10], and with it, *Operational Transformation* (OT), which tries to solve the problem of diverging copies of a buffer, in a fully distributed setting. The main idea is to construct a *transformation function* where remote operations are transformed with regards to conflicting local operations in a way that guarantees consistency.

In order to achieve this, an additional parameter, *priority*, is added to insertions and deletions; the priority is a unique identifier for a given client, represented as a number, and is used in order to break ties. The transformation function $T : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$, proposed by Ellis and Gibbs, is restated in Figure 3.1.

Let us again consider the example described in Figure 1.2 (page 3), where two operations $O_0 = ins(0, \texttt{a}, 0)$ and $O_1 = ins(0, \texttt{b}, 1)$ are performed concurrently, leading to an inconsistent state. Rather than applying operations directly, remote operations are transformed with regards to (potential) concurrent local operations, before they are applied. Communication is done directly between clients (as opposed to going via a server).

$$T(ins(p1, c1, pr1), ins(p2, c2, pr2)) = \begin{cases} ins(p1, c1, pr1) & \text{if } p1 < p2 \\ ins(p1 + 1, c1, pr1) & \text{else if } p1 > p2 \\ nop & \text{else if } c1 = c2 \\ ins(p1 + 1, c1, pr1) & \text{else if } pr1 > pr2 \\ ins(p1, c1, pr1) & \text{otherwise} \end{cases}$$

$$T(ins(p1, c1, pr1), del(p2, pr2)) = \begin{cases} ins(p1, c1, pr1) & \text{if } p1 < p2 \\ ins(p1 - 1, c1, pr1) & \text{otherwise} \end{cases}$$

$$T(del(p1, pr1), ins(p2, c2, pr2)) = \begin{cases} del(p1, pr1) & \text{if } p1 < p2 \\ del(p1 + 1, pr1) & \text{otherwise} \end{cases}$$

$$T(del(p1, pr1), del(p2, pr2)) = \begin{cases} del(p1, pr1) & \text{if } p1 < p2 \\ del(p1 - 1, pr1) & \text{else if } p1 > p2 \\ nop & \text{otherwise} \end{cases}$$

Figure 3.1: The transformation function from [10].

From the perspective of u_0:

- O_0 is generated locally,
- O_1 is received from u_1, $T(O_1, O_0)$ is applied.

From the perspective of u_1:

- O_1 is generated locally,
- O_0 is received from u_0, $T(O_0, O_1)$ is applied.

The scenario is illustrated in Figure 3.2. By composing the operations at each user and applying that operation to the empty buffer ϵ, the resulting buffer is found.

Operation applied by u_0:

$$T(O_1, O_0) \circ ins(0, \texttt{a}, 0)(\epsilon) =$$
$$T(ins(0, \texttt{b}, 1), ins(0, \texttt{a}, 0))(\epsilon) =$$
$$ins(1, \texttt{b}, 1)(\texttt{"a"}) = \texttt{"ab"}$$

Operation applied by u_1:

$$T(O_0, O_1) \circ ins(0, \texttt{b}, 1)(\epsilon) =$$
$$T(ins(0, \texttt{a}, 0), ins(0, \texttt{b}, 1))(\epsilon) =$$
$$ins(0, \texttt{a}, 0)(\texttt{"b"}) = \texttt{"ab"}$$

Note that $T(O_1, O_0) \circ O_0 \neq T(O_0, O_1) \circ O_1$ (i.e. the operations are not *equal*), but they are *equivalent* in the sense that applying them to the same buffer yields the same result, denoted:

$$T(O_1, O_0) \circ O_0 \equiv T(O_0, O_1) \circ O_1$$

As shown, the transformation function T can be used to resolve a conflict. However, the algorithm should be able to handle any number of concurrent

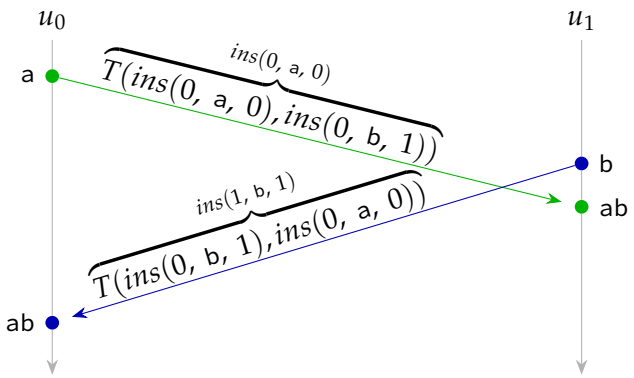

Figure 3.2: Conflict resolved using T.

operations, from an arbitrary number of clients, which may lead to conflicts of great complexity — it is not given that the transformation function can resolve every conflict that can arise.

3.2 Discussing Consistency in Operational Transformation

This section introduces some consistency models that have been used to describe correctness of OT algorithms, and achievements in trying to verify these algorithms. In OT it is common to refer to a *site* as a uniquely identified object with a data segment (for example a document) which a user can manipulate. When no messages are "in flight", the system is said to be *quiescence*.

The consistency model of [10], is defined by the following two properties:

- **Causality**[1]: If O_i was executed before O_j at one site, then O_i must be executed before O_j on all sites.
- **Convergence**: At quiescence, all copies are identical.

Sun et al. [39] expanded the consistency model of [10] with:

- **Intention preservation**: If an operation O_i has been transformed to O_i', then the effects of applying O_i' must be equivalent of that of applying O_i.

dOPT is a fully distributed algorithm, where determining temporal relationships between events (i.e. generation and reception of operations) is a more challenging task than when leveraging a centralized server. It uses a *state vector* (also referred to as a *vector clock*) which is essentially an extension of Lamport clocks [23], yielding a partial order of events. An

[1] Referred to as the *Precedence Property* in [10].

ordering being partial means that there exists events where neither event precedes the other, which means the events are *concurrent*.

The dOPT algorithm ensures that operations are applied according to the partial order of events, where an event is either the generation of an operation or the reception of one. This ensures causality, but not convergence. Because the order is partial there are events that are concurrent; instead of trying to order these events *totally* (i.e. ensure that for any two events, one will precede the other) a transformation function is used. Given two concurrent operations O_i, O_j, where O_i has already been applied, O_j must be transformed with regards to O_i before it is applied.

A transformation function T must satisfy:

$$T(O_j, O_i) \circ O_i \equiv T(O_i, O_j) \circ O_j \qquad (C_1)$$

for all $O_i, O_j \in \mathcal{O}$ in order to guarantee convergence; this is a necessary, but not a sufficient condition [10]. The transformation function T from Figure 3.1 does not satisfy the condition, which has been shown by [20].

We have been able to reproduce the result by model checking our Maude specification. A minimal counterexample, as shown in Figure 3.3, involves two operations, $O_0 = ins(0, \texttt{b})$ and $O_1 = del(0)$, applied to an initially non-empty buffer. The priority parameter is omitted in this example, because it has no effect on the outcome. Assume that both u_0 and u_1 initially has a buffer containing `"a"`.

From the perspective of u_0:

- $ins(0, \texttt{b})$ is generated locally,
- O_1 is received from u_1, $T(O_1, O_0)$ is applied.

From the perspective of u_1:

- $del(0)$ is generated locally,
- O_0 is received from u_0, $T(O_0, O_1)$ is applied.

Again, the resulting buffer can be calculated by applying the respective operations to the buffer `"a"`.

Operation applied by u_0:

$$\begin{aligned} T(O_1, O_0) \circ ins(0, \texttt{b})(\texttt{"a"}) &= \\ T(del(0), ins(0, \texttt{b}))(\texttt{"ba"}) &= \\ del(1)(\texttt{"ba"}) &= \texttt{"b"} \end{aligned}$$

Operation applied by u_1:

$$\begin{aligned} T(O_0, O_1) \circ del(0)(\texttt{"a"}) &= \\ T(ins(0, \texttt{b}), del(0))(\epsilon) &= \\ ins(-1, \texttt{b})(\epsilon) &= \textit{error} \end{aligned}$$

Here we demonstrate two problems with the transformation function. One is that the buffers diverged, seeing that u_0 and u_1 does not end up in the same final state. Secondly, the transformation function returns $ins(-1, \texttt{b})$ which is not well defined, and is not in the set of operations $\mathcal{O}$. The problem is manifested in the second equation from Figure 3.1, where $<$ must be replaced with $\leq$. The equation is restated correctly for completeness:

$$T(ins(p1, c1, pr1), del(p2, pr2)) = \begin{cases} ins(p1, c1, pr1) & \text{if } p1 \leq p2 \\ ins(p1 - 1, c1, pr1) & \text{otherwise} \end{cases}$$

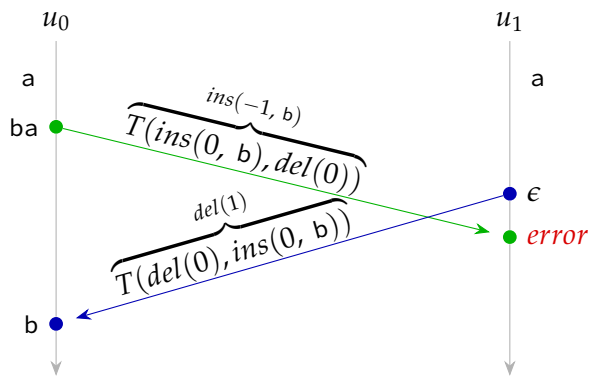

Figure 3.3: Disproving C_1.

From what we can tell, the bug went unnoticed for many years[2], which shows the subtleness of the bug — uncovering bugs like this is hard, and is part of our motivation for using formal methods.

The corrected version of T satisfies C_1, but this is not sufficient for guaranteeing convergence in a fully distributed setting. It handles all conflicts where only two operations are involved; in order to handle any number of concurrent operations, being executed in an arbitrary order, T must also satisfy:

$$T(T(O_k, O_i), O'_j) = T(T(O_k, O_j), O'_i) \tag{C_2}$$

where $O'_j = T(O_j, O_i)$ and $O'_i = T(O_i, O_j)$ for all $O_i, O_j, O_k \in \mathcal{O}$. It has been proved that a transformation function T that satisfies C_1 and C_2 is sufficient in order to guarantee convergence [35, 26].

In [20], Imine et al. show, using a theorem prover, that neither the corrected version of T from [10], or any of transformation functions from [33, 39, 36] satisfies C_1 and C_2. Furthermore, they propose a transformation function of their own, which was proved correct by their theorem prover. Yet, in [32], they prove this transformation function wrong. This paper also shows that there does not exist a transformation function that satisfies C_1 and C_2 without adding additional parameters to the operations.

To the best of our knowledge, there has not been found a transformation function that satisfies the consistency model of Sun et al. [39].

3.3 Operational Transformation with a Client-Server Architecture

In [27], Nichols et al. introduces a simplified algorithm for OT in the Jupiter Collaboration System, based on GROVE [10], leveraging a centralized

[2] In [20] Imine et al. credits the finding to [39, 35, 33], but we have not been able to confirm that any of them uncovered that T does not satisfy C_1.

server. It is a symmetric algorithm, in the sense that the core algorithm is the same at the client and the server. As in GROVE, the document is replicated at every client, but the server keeps an additional copy of the document. Each client synchronizes its changes with the server, yielding a two-way synchronization protocol, as opposed to the fully distributed N-way synchronization of GROVE.

When a client generates an operation, it is communicated to the server. On reception of an operation, the server applies it to its document, transforming it if necessary, and sends the transformed operation to the other clients. When a client receives an operation from the server, the operation is transformed if necessary, and then applied to the client's local copy.

Having a server guarantees that causality violation never occurs [38], without the need for maintaining a state vector. This is because all communication is done via the server, so a client will receive all remotely generated operations (i.e. operations it did not generate itself) in the order the server received them.

In the Jupiter Collaboration System each client-server pair must store the operations sent until they are acknowledged by their counterpart. This is because new operations might need to be transformed with regards to non-acknowledged operations.

The Google Wave protocol is largely based on the Jupiter Collaboration System [7]. In contrast to the Jupiter Collaboration System, the Google Wave protocol requires that clients wait for acknowledgement before sending new operations. Clients may still apply changes to their local copy, but need to queue the operations, and send them on reception of an acknowledgement. This reduces the memory consumption at the server, because the server will only need to keep one history of operations.

3.4 Existing Implementations

There are a wide range of existing implementations that supports collaborative editing. Providing a comprehensive survey on all existing solutions is not a goal of this thesis, so we will only briefly discuss a select few.

One particularly interesting implementation is Gobby[3], which runs on the Obby protocol and is open. Through browsing the source code, we believe it is based on the Jupiter Collaborative System algorithm to ensure consistency between clients[4]. It is a feature-full protocol, supporting things like group undo (only undo locally generated operations), group chat and more.

[3] https://gobby.github.io/

[4] We have skimmed the source code and found files with "jupiter" in the name, which indicates that the Jupiter algorithm is used. However, we have not found this explicitly stated in the documentation or done a comprehensive study of the source code.

There exists an Emacs extension called Rudel[5], which supports the Obby protocol. It does however seem abandoned, and we have not been able to make it run on a modern Emacs distribution. In addition, the protocol seems non-trivial to implement, seeing that the extension is over 5000 lines of code.

Another interesting implementation is Floobits[6], as it supports a wide range of editors, leveraging a centralized server. It is, however, not open and it can be used with a limited amount of "workspaces" in exchange for a monthly fee. Because the protocol is closed, we do not know how it resolves conflicts between participating clients.

3.5 Shared Buffer in Comparison

The Shared Buffer algorithm leverages a centralized server, similarly to the Jupiter Collaborative System. The main problem that Shared Buffer solves which, to the best of our knowledge is not covered in the literature, is to maintain consistency between clients without requiring complicated control mechanisms on the client side. It does this in an optimistic way, meaning that there are put no restrictions to when, or in what section of the buffer the clients may perform editing operations.

When leveraging a centralized server, the transformation function used only needs to satisfy C_1 in order to guarantee convergence [38]. C_2 essentially states that the transformation function is not dependent on the order of which a sequence of operations are transformed [20]. As the server is the only entity that will perform transformations, there is only one order of which the operations will be transformed.

There are complicating factors introduced by relieving the clients of having to applying control mechanisms themselves. One is that the client is the only entity that has a fully updated view of what operations it has performed. To deal with this, the algorithm must ensure that clients never perform operations received from the server, if the client is in conflict with the server. This is ensured by using a sequence number scheme (Section 5.5.1), which is very simple on the client side.

Further complications arise when a client simply rejects outdated messages; in the Jupiter Collaboration System, the client would accept the operation, and transform it if necessary, which would leave it consistent with the server. Transformation functions only work for operations that are generated from the same state [37]; when giving the client the opportunity to generate operations from an inconsistent state, then transformation functions alone cannot be used to ensure consistency. To resolve this, the new operations from the client must be reset to a consistent state; to achieve this

[5] https://sourceforge.net/projects/rudel/
[6] https://floobits.com/

we leverage *exclusion* transformations [39], which can be seen as the dual of the transformation functions we have seen so far.

Shared Buffer does not rely heavily on transformation functions, and instead try to order operations in a way that preserves the users intentions. This has proved successful for the majority of cases, but transformation functions are needed to deal with some edge cases; these are typically the cases where multiple operations are performed at the same position in the buffer.

The Shared Buffer algorithm borrows ideas from both fully distributed algorithms and the Jupiter Collaborative System. It has similarities to the GOT algorithm [39], in that it uses both inclusion and exclusion transformation functions. The GOT algorithm also uses a undo/do/redo-scheme which is similar to Shared Buffer. A main distinction is that Shared Buffer constructs operations that will undo operations on the clients behalf.

The consistency model used for Shared Buffer is a weak consistency model, namely *eventual consistency* [40]. This was chosen because it can naturally be seen as a minimum requirement for a real-time collaborative system. In addition, it is a property which is easy to express in LTL (Linear Temporal Logic), which allows us to use the Maude LTL model checker for verification. Requirements with regards to preserving user intent is not formally specified, but measures are taken to preserve user intent in the majority of cases.

The system has been formally specified and model checked, and has been verified correct for a small number of clients and operations (see Section 6.4). This means that it handles *all possible conflicts* that can occur within the set bounds. Our specification is *broader* than that of Imine et al. in the sense that we model the clients, the server and the flow of messages, where as the work of Imine et al. focuses on properties of the transformation functions. This gives us confidence that the algorithm is correct in its entirety.

Chapter 4

Client-side Specification

The Shared Buffer System is formally modeled using The Maude System. Modeling a system is in essence capturing what can occur in the system in a precise manner, at a suitable level of abstraction. For instance, it is important to model that clients can send messages concurrently, and that there is no way to a priori determine the order of which they are received by the server. On the other hand, we merely assume that messages between a given client and the server are delivered in order, undamaged and without duplication, and make no attempt to model how this is achieved.

Earlier work on formal verification of *Operational Transformation* (OT) algorithms has been focused on verifying properties of the *transformation functions* (as discussed in Section 3.2), which is an essential part of all OT algorithms. However, there are other aspects of the algorithms, that are left unverified, leaning on analytical proofs by the original authors. Instead of writing analytical proofs we leverage formal methods to ensure robustness of the system.

We aim at modeling the clients, the server and the communication between these, but restrict ourselves to editing sessions where all the clients have the same initial buffer and a constant number of connected clients. The algorithm is a part of the specification, and therefore also subject for verification. To the best of our knowledge, formal verification techniques has not been applied on a *complete* real-time collaboration algorithm in the literature before.

The verification technique we have chosen to apply is model checking. Proving properties of a finite-state system using model checking is *decidable*, but if the system has an infinite number of reachable states, there is no guarantee that the model checker will terminate. Our system is infinite as there can be an arbitrary number of operations, an arbitrary number of clients, with an infinite number of different initial buffers. If we were to model the system without limitations, the model checker would not necessarily terminate. In order to deal with this the system is modeled as a finite system, where the number of operations, the number of clients

and an initial buffer are given as parameters when model checking.

This chapter starts of by giving a short introduction to the Maude language and how we specify the system. We will then go on to specify the behavior of the clients. The equations in this chapter are translations of the Maude specification. For rewrite rules (that we will introduce shortly) we use Maude syntax in favor of mathematical notation.

4.1 A Short Introduction to Maude

The Maude System consists of a programming and modeling language, as well as tools for exploring the state space of the model. In essence, Maude is an implementation of *rewriting logic* which has proved useful for modeling distributed systems [28].

A Maude specification consists of *sorts*, *signatures*, *variables*, *equations* and *rewrite rules*. A sort is simply a label that is associated with some value. A signature defines a function symbol, along with its domain and codomain, which constitutes the values of a sort (where a constant is represented as a function symbol with arity zero). The values of a specification are constructed by applying function symbols with respect to their domains, and yields what is called a *ground term*. Variables must be of a given sort and is essentially a placeholder for a ground term; a term (i.e. "non-ground") can contain variables, which is what separates it from a *ground* term.

An equation is a relation that takes a left-hand and a right-hand term; it symbolizes that the terms are considered equivalent. Rewriting rules are similar to equations, but the terms does not need to be equivalent. Rather, a rewriting rule symbolizes that the left-hand term *may evolve* to the right-hand term, and is strictly read left to right. The equations of the system represents the *static* part of the system, and the rewrite rules represent the *dynamic* part of the system.

Two fundamental commands in The Maude System helps to shed light on how equations and rewriting rules operate. The *reduce* command takes a term and if the term (or a subterm) matches the left-hand term of an equation, it is rewritten to the right hand term, and the process continues until the reduced term does not match any equation in the specification. The *rewrite* command can be given an argument deciding how many rewrites it performs. It takes a term which it applies to an arbitrary rewrite rule that matches the left-hand term of the rule, followed by reducing the resulting term. The process is repeated until it has reached the specified number of rewrites, or if no more rewrites can be applied. It can be useful to think of an equation as special case of a rewrite rule which is always applied immediately.

In our specification, we use rewrite rules to describe nondeterministic changes in the system, like a user inserting or deleting a character from

its buffer. The equations are used to describe the system's reaction to the changes in the system, which is deterministic.

4.2 Client-side Specification

In Chapter 2 we defined operations and how they are applied. This was stated as a set of equations which has been translated to Maude. The Maude representation is almost identical with the aforementioned definitions, but operation application is syntactically different and have not modeled the entire *Unicode* set, but rather chosen a small set of characters.

A client consists of a user label, buffer, sequence number, state token, along with an incoming and outgoing message queue. The following Maude term, where capital letters are variables of the appropriate sort, will match any user:

```
< U | buffer : B, seqno : N, token : T, out-queue : Q, in-queue: Q' >
```

An example of a ground term that will match the term above can look like so:

```
< user 0 | buffer : nil, seqno : 0, token : 0,
           in-queue : nil, out-queue : nil >
```

Note that `nil` is used to represent both an empty buffer and an empty queue. A user may nondeterministically insert a character to its buffer, which is represented using a rewrite rule, where `=>` symbolizes the rewrite relation [1]:

```
rl [user-inserts] :
    < U | buffer : (B B'), seqno : S, token : T, out-queue : Q >
  =>
    < U | buffer : (B C B'), seqno : s S, token : T,
          out-queue : (msg(ins(size(B), C), T, S) Q) > .
```

The buffer is represented as `(B B')`, where both `B` and `B'` are any two buffers that matches the client's buffer when concatenated (where concatenating buffers is analogous to concatenation of strings). Note that the buffers may be empty. Assuming we have a character `C` (where a character is treated as a string of length one), an insertion is simply placing a character in between the two buffers, which yields a new buffer `(B C B')`. This rule enables `C` to be inserted at any point in the buffer, because we have not put any restriction on `B` and `B'` besides that they together form the complete buffer.

The client must communicate its change to the server, which is modeled as putting a message on in its outgoing queue. A message, denoted `msg`, consists of an operation, a state token and a sequence number. The size of `B` determines the position of the insertion, and so `ins(size(B), C)`

[1] The rules stated here are slightly simplified from those in the actual Maude specification.

represents that `C` was inserted at position `size(B)`. The current state token `T` and sequence number `S` is added to the message. Lastly, the client must increment its sequence number, using the successor function `s`. The rule for deletion is almost identical, with the difference of removing a character from the buffer, rather than inserting one, and labeling the operation `del`.

The server can put messages on a client's incoming queue, which the client in turn (eventually) reads from. In order to model the latency of messages traveling, it reads from the queue in an nondeterministic manner. The following rewrite rule may be applied if the sequence number of the client is equal to the sequence number of the message at the end of its incoming queue:

```
rl [user-receive] :
    < U | buffer : B, seqno : S, token : T,
          in-queue : (Q msg(O,T',S)) >
  =>
    < U | buffer : apply O on B, seqno : s S, token : T',
          in-queue : Q > .
```

Notice that the variable `S` is used both as the client sequence number and the sequence number of the message to ensure that the sequence numbers are equal. On reception of a message the operation is applied to the client's local buffer, the sequence number is incremented and the state token of the message replaces the current state token of the client. Note that `apply O on B` is a term that will match an equation which applies the operation `O` to `B`. It is syntactically different from $O(B)$, which we have seen in previous chapters, but semantically identical.

Similarly, there is a rule for rejecting a message, which may be applied if an incoming message has a sequence number that is *not* equal to the sequence number of the client. In that case the message is removed from the queue, and the sequence number is incremented; nothing else changes.

At this point we can try to perform rewrites on a term containing multiple users, and they will insert and delete characters and add messages to their outgoing queue. Seeing that we have not specified a server yet, they will not receive any messages and their outgoing queue will grow monotonically. Their respective buffers are likely to diverge. Here is an example of two users starting with an empty buffer after three rewrites:

```
< user 0 | buffer : nil, seqno : 2, token : 0, in-queue : nil,
           out-queue : (msg(del(0, a), 0, 1) msg(ins(0, a), 0, 0)) >

< user 1 | buffer : b, seqno : 1, token : 0, in-queue : nil,
           out-queue : msg(ins(0, b), 0, 0) >
```

From the resulting term we can read that u_0 (i.e. `user 0`) has inserted an `a`, and deleted it afterwards, while u_1 has inserted a `b`.

4.2.1 Measures to Reduce the State Space

The *state explosion problem* is considered the main obstacle for model checking [4]. When performing model checking, an initial state must be given; in Maude this corresponds to a term. If the term matches a rewrite rule, it may be applied which in turn yields a new state. A term can match multiple rewrite rules, so the number of reachable states is given by the sum of how many rewrite rules the initial term matches, and how many rewrite rules each of the resulting terms matches and so on. There may be an infinite number of reachable states.

Our focus has been on writing a specification which accurately describes the system, rather than optimizing the specification for verification; nevertheless, some measures has been taken to reduce the state space and are documented here.

In reality, a user could decide to type any arbitrary character at any given time; in the model, all insertions are done alphabetically, meaning the first insertion is always an `a`, the second is a `b` and so on. This is necessary in order to keep the state space at a manageable size. What character the user types is of no importance to the algorithm, so there is no need to check what would happen if, say, `b` where typed before `a`.

Another restriction is put on which user performs an operation at a given time. Users are labeled because the server needs information about each individual user; however, if we were to swap all u_0 labels with u_1, it would have no effect on the scenario, as long as *all* labels are swapped. It can be helpful to look at a visualized scenario and permute the labels at the top to see that the scenarios are symmetric, and checking both is redundant.

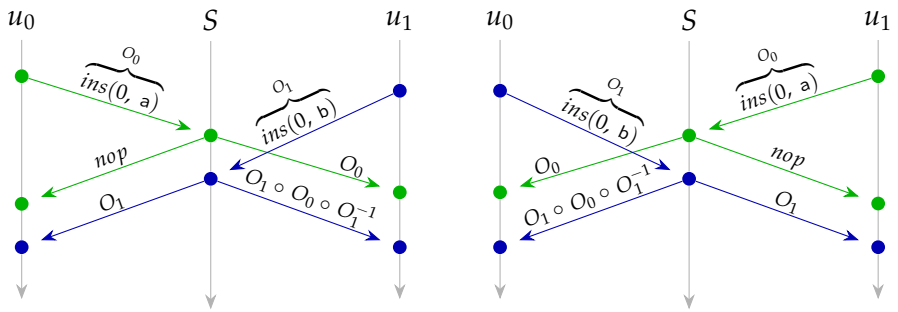

Figure 4.1: Permuting labels.

A scheme is imposed where u_0 always performs the first operation, u_0 or u_1 performs the second, u_0, u_1 or u_2 performs the third, and so on. This corresponds to allowing the scenario to the left from Figure 4.1 to occur, but not the scenario on the right.

In the example at the end of the last section, three rewrites were made. Without the restriction of u_0 performing the first operation, there would be 49 reachable states. When imposing the scheme, only 25 different states can

be reached within three rewrites, reducing the state space by a factor of 2. As the number of users grow, the reduction of states increases.

Table 4.1: Reachable states after three rewrites with and without restriction.

Number of users	2 users	3 users	4 users	5 users	6 users
Non-restricted	49	106	193	316	481
Restricted	25	29	29	29	29
Reduction by a factor of	1.96	3.66	6.66	10.9	16.59

Note that the server only reacts to incoming messages, and never acts on its own accord. This is why the only measures to reduce the state space is done by imposing restrictions on the clients.

Chapter 5

Server-side Specification

At this point the behavior of clients has been specified. The more interesting part of the system is the server, as it handles most of the complexity in the algorithm. This chapter fully describes the server-side algorithm.

The server maintains a *state token*, a *history* and a mapping from users to a list of possibly rejected operations associated with a token. For every operation received, the server increments its state token, which is one initially. The history determines the order of which operations should be applied. The list of possibly rejected operations (associated with a token) is used to construct an operation which will make the client consistent with the current history. When receiving a message from a client, we assume that there exists a way of uniquely identifying the client — in the model this is a user identifier.

5.1 Events

An essential part of the server algorithm is maintaining a history, where the entries in a history are *events*. There is only one type of event at the system, namely the reception of a message, containing an operation, a token and a sequence number.

Definition 5.1.1 (Events)**.** The events on a server S communicating with a set of clients, identified by users in $\mathcal{U}$, is formally defined as the smallest set of four-tuples $\mathbb{E}$, where every $\langle O, t, m, u \rangle \in \mathbb{E}$ satisfies the following:

- $O \in \mathcal{O}$ and O is not a composition of operations
- $t \in \mathbb{N}$
- $m \in \mathbb{N}$
- $u \in \mathcal{U}$ ⊣

Given a message $msg(O, t, s)$ sent from $u \in \mathcal{U}$ and received by the server at time m, the event $\langle O, t, m, u \rangle$ is constructed. Remember that the token t is only updated at the client when it successfully receives a message from the server (i.e. does not reject); it implies that the latest message the client has received was when the server's state token was t. Furthermore, we rely on the client to have executed all operations with a time stamp smaller than t.

The history of events dictates an order of which operations should be applied. In the case where there are no concurrent events, the arrival time m is used to determine what event should precede the other. But to decide an order, we first need to be able to detect concurrent operations. Intuitively, two operations are concurrent if they were generated independently from each other.

Definition 5.1.2 (Concurrent Events)**.** Two events $E_i = \langle O_i, t_i, m_i, u_i \rangle$ and $E_j = \langle O_j, t_j, m_j, u_j \rangle$, where $E_i, E_j \in \mathbb{E}$, are said to be concurrent if:

$$u_i \neq u_j \wedge ((t_i \leq t_j \wedge m_i \geq t_j) \vee (t_j \leq t_i \wedge m_j \geq t_i))$$

E_i is concurrent with E_j is denoted $E_i \parallel E_j$. ⊣

A user cannot produce concurrent events; The first criteria of the definition $u_i \neq u_j$ ensures that events performed by the same user cannot be considered concurrent. Note that an event is not concurrent with itself by this definition; the case is ignored because there is never a need to examine the relationship between an event and itself. There are no duplicate events in the system, seeing that the time stamp is guaranteed to be unique.

Let us now consider $t_i \leq t_j \wedge m_i \geq t_j$. It is helpful to read it as: "O_i was generated at same time or before O_j was generated, but O_i arrived at the server at the same time or after O_j was generated". If $t_i \leq t_j$ then it cannot be the case that O_i was generated after having applied O_j. Similarly, if $t_j \leq m_i$ then it cannot be the case that O_j was generated after having applied O_i. When the operations were generated independently from each other, we say they are concurrent. By the same reasoning, E_j is concurrent with E_i if $t_j \leq t_i \wedge m_j \geq t_i$, assuring symmetry.

Given two non-concurrent events, one must have *happened before* the other. Note that this *happened before*-relation is defined in terms of a state token and a time stamp, and does not necessarily represent which event happened before the other in *real* time.

Definition 5.1.3 (Happened Before)**.** An event $E_i = \langle O_i, t_i, m_i, U_i \rangle$ *happened before* $E_j = \langle O_j, t_j, m_j, U_j \rangle$, where $E_i, E_j \in \mathbb{E}$, if:

$$E_i \nparallel E_j \wedge m_i < m_j$$

E_i *happened before* E_j is denoted $E_i \longrightarrow E_j$. ⊣

If the events are not concurrent, then the arrival time is used to determine which event happened before the other.

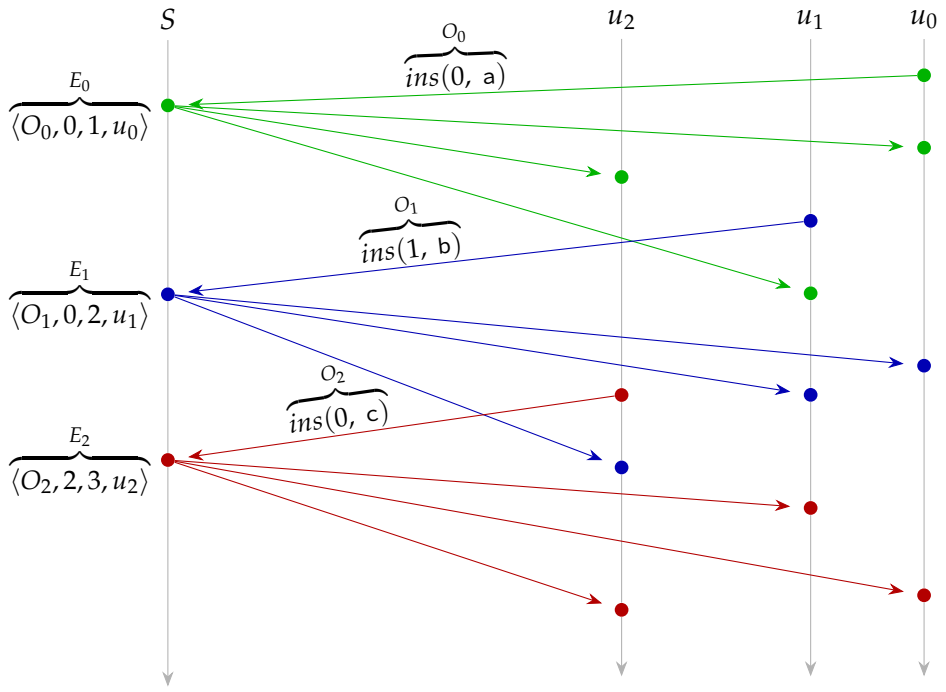

Figure 5.1: A non-trivial example.

Let us look at a non-trivial example of a scenario and examine the relationships between events. The scenario is visualized in Figure 5.1. We have the following three events:

$$E_0 = \langle O_0, t_0, m_0, u_0 \rangle = \langle O_0, 0, 1, u_0 \rangle$$
$$E_1 = \langle O_1, t_1, m_1, u_1 \rangle = \langle O_1, 0, 2, u_1 \rangle$$
$$E_2 = \langle O_2, t_2, m_2, u_2 \rangle = \langle O_2, 2, 3, u_2 \rangle$$

Their relationships are as follows:

- $E_0 \parallel E_1$. The events were generated with the same state token, and the state token of E_0 is smaller than the arrival time of E_1. More precisely, they are concurrent because the operations are performed by different users and $t_0 \leq t_1 \wedge m_0 \geq t_1$.
- $E_0 \longrightarrow E_2$. This is because u_2 received O_0 before generating O_2. More precisely, they are not concurrent because $t_0 \leq t_2 \wedge m_0 \not\geq t_2$ and $t_2 \not\leq t_0 \wedge m_2 \geq t_0$, and $E_0 \longrightarrow E_2$ because $m_0 < m_2$.
- $E_1 \parallel E_2$. The state token of E_1 is smaller than that of E_2, but the arrival time of E_1 the equal to the state token of E_2, meaning O_2 was generated before O_1 was received. More precisely, the events are performed by different users and $t_1 \leq t_2 \wedge m_1 \geq t_2$.

It can be helpful to notice that there is a correspondence between overlapping lines in the diagram and events being concurrent.

5.2 An Ordering of Events

The history maintained on the server should respect an ordering $\prec$. This ordering must respect the *happened before*-relation (Definition 5.1.3), meaning that for any two events $E_i, E_j \in \mathbb{E}$ where if $E_i \longrightarrow E_j$ then $E_i \prec E_j$. The question that remains is how to order concurrent events.

Let us first take a step back to make a useful observation. Say a character is inserted at point i in a given buffer. The characters that were located at positions 0 to i (exclusive) remain at the same position — the characters that were located at i or higher are shifted one step to the right. Similarly for deletions, characters that were located at position 0 to i (exclusive) stay where they were, but characters located at point $i + 1$ or higher are shifted to the left (the character that were at point i is deleted).

Say we have two operations $ins(2, \texttt{x})$ and $ins(4, \texttt{y})$ and both are independently applied to the buffer `"abcdef"`.

$$ins(2, \texttt{x})(\texttt{"abcdef"}) = \texttt{"abxcdef"}$$
$$ins(4, \texttt{y})(\texttt{"abcdef"}) = \texttt{"abcdyef"}$$

Two ways of combining the operations are $ins(4, \texttt{y}) \circ ins(2, \texttt{x})$ and $ins(2, \texttt{x}) \circ ins(4, \texttt{y})$. Applying them to the buffer `"abcdef"` yields different results.

$$ins(4, \texttt{y}) \circ ins(2, \texttt{x})(\texttt{"abcdef"}) = ins(4, \texttt{y})(\texttt{"abxcdef"}) = \texttt{"abxcydef"}$$
$$ins(2, \texttt{x}) \circ ins(4, \texttt{y})(\texttt{"abcdef"}) = ins(2, \texttt{x})(\texttt{"abcdyef"}) = \texttt{"abxcdyef"}$$

Notice that in both cases `x` was placed between `b` and `c`. On the other hand `y` was placed between `c` and `d` in the first case, and between `d` and `e` in the second. Originally `y` was placed between `d` and `e`, so we can assume that was the *intention* of the user. We make the general rule, that when the positions of two operations differ, the operation with the highest position should always be performed first.

Given two operations that operates on the same position, then deletions should always be performed before insertions. If the insertion is done first, then the delete operation would simply remove the character which was just inserted, which does not seem to satisfy either user. If the deletion is done first, the correct character is deleted and the insertion is placed between the characters it wanted, with the exception of the character immediately in front of it. Finally, if the operations are of the same type and operate on the same position, then the arrival time is used as a tiebreaker.

The following defines when an event is said to precede another. Two accessor functions for operations are used, where *pos* returns the position argument of the operation, and *type* returns *ins* or *del*, depending on the operation being an insertion or deletion.

Definition 5.2.1 (Precedence)**.** Two events $E_i = \langle O_i, t_i, m_i, U_i \rangle$ and $E_j = \langle O_j, t_j, m_j, U_j \rangle$ where $E_i, E_j \in \mathbb{E}$ are given. Let $p_i = pos(O_i)$, $p_j = pos(O_j)$, $type_i = type(O_i)$ and $type_j = type(O_j)$. An event E_i *precedes* E_j if:

$$E_i \prec E_j = \begin{cases} p_i > p_j \\ \text{or } (p_i = p_j \wedge type_i \neq type_j \wedge type_i = del) \\ \text{or } (p_i = p_j \wedge type_i = type_j \wedge m_i < m_j) & \text{if } E_i \parallel E_j \\ m_i < m_j & \text{otherwise} \end{cases}$$

E_i *precedes* E_j is denoted $E_i \prec E_j$. ⊣

Let us look back at the scenario from Figure 5.1. There were three events:

$$E_0 = \langle O_0, t_0, m_0, u_0 \rangle = \langle ins(0, \texttt{a}), 0, 1, u_0 \rangle$$
$$E_1 = \langle O_1, t_1, m_1, u_1 \rangle = \langle ins(1, \texttt{b}), 0, 2, u_1 \rangle$$
$$E_2 = \langle O_2, t_2, m_2, u_2 \rangle = \langle ins(0, \texttt{c}), 2, 3, u_2 \rangle$$

where $E_0 \parallel E_1$, $E_0 \longrightarrow E_2$ and $E_1 \parallel E_2$. We have the following relations:

- $E_1 \prec E_0$ because they are concurrent and $pos(O_1) > pos(O_0)$.
- $E_0 \prec E_2$ because they are not concurrent and $m_0 < m_2$.
- $E_1 \prec E_2$ because they are concurrent and $pos(O_1) > pos(O_2)$.

The only possible ordering of these three events is:

$$E_1 \prec E_0 \prec E_2$$

This scenario can only occur if the initial buffer was non-empty, seeing that $ins(1, \texttt{b})$ was applied to the initial buffer; let us assume the initial buffer was `"f"`. From the perspective of each user:

Perspective of u_0:	Perspective of u_1:	Perspective of u_2:
$ins(0, \texttt{a})(\texttt{"f"}) = \texttt{"af"}$	$ins(1, \texttt{b})(\texttt{"f"}) = \texttt{"fb"}$	$ins(0, \texttt{c})(\texttt{"af"}) = \texttt{"caf"}$

Note that the buffer of u_2 is different from the other two users, seeing that $ins(0, \texttt{a})$ had been applied before $ins(0, \texttt{c})$ was generated. By composing the operations from the events, according to the ordering, we would get the operation $ins(0, \texttt{c}) \circ ins(0, \texttt{a}) \circ ins(1, \texttt{b})$. The result of applying the composed operation to the initial buffer is:

$$ins(0, \texttt{c}) \circ ins(0, \texttt{a}) \circ ins(1, \texttt{b})(\texttt{"f"}) =$$
$$ins(0, \texttt{c}) \circ ins(0, \texttt{a})(\texttt{"fb"}) =$$
$$ins(0, \texttt{c})(\texttt{"afb"}) = \texttt{"cafb"}$$

It seems to satisfy the intent of every user; u_0 placed an `a` in front of the `f`, u_1 placed a `b` ahead of the `f` and u_2 placed a `c` in front of the `a`.

5.2.1 Events Under Precedence is not a Total Order

A total order under a relation requires the relation to be antisymmetric, total and transitive. Antisymmetric, meaning that if E_i precedes E_j then E_j cannot precede E_i and total, meaning that any two events are comparable under the precedence relation (Definition 5.2.1); we have found no counterexample to either of these properties. However, the precedence relation is not transitive, and so an ordering under $\prec$ is not a total order, nor is it a partial order.

These following three events is enough to show that $\prec$ is not transitive:

$$E_0 = \langle ins(0,\ \texttt{a}), 0, 1, u_0 \rangle$$
$$E_1 = \langle ins(1,\ \texttt{b}), 0, 2, u_0 \rangle$$
$$E_2 = \langle del(0,\ \texttt{f}), 0, 3, u_1 \rangle$$

The session was initiated with a buffer `"f"`. The first user (u_0) typed an `a` followed by a `b`, resulting in the buffer `"abf"`. The other user (u_1) deleted the only character in the buffer, resulting in the empty buffer ϵ.

We have that $E_0 \longrightarrow E_1$, $E_0 \parallel E_2$ and $E_1 \parallel E_2$. Because u_0 performed both the operations from E_0 and E_1 we have that $E_0 \prec E_1$. u_1 performed a deletion at the same point as u_0 performed an insertion, so $E_2 \prec E_0$. However, $E_1 \prec E_2$ because the operation in E_2 has a higher position. This means we have two plausible orderings:

$$E_0 \prec E_1 \prec E_2$$
$$E_2 \prec E_0 \prec E_1$$

Neither satisfies transitivity, as $E_0 \not\prec E_2$ and $E_2 \not\prec E_1$. The problem is related to how events by the same user are totally ordered (i.e. always compared by a unique time stamp), but this information is not embedded in the event itself.

This has two implications that we want to note. One is that we cannot *sort* events based on $\prec$, nor use a standard ordered data structure, due to its lack of transitivity. The other is that there exists multiple plausible orderings of a given set of events, meaning that there are multiple orders where the precedence relation is satisfied between each consecutive pair of events.

A possible resolution to this is discussed in Future Work, Section 8.2.2. Instead we build a history that relies on the given precedence relation, presenting a scheme that takes the lack of transitivity into consideration.

5.3 Building a History of Events

A history of events is maintained on the server, and it dictates the order of which operations should be applied by every participant. It is built in an

iterative manner, meaning that for every incoming message the new event is placed at some point in the history.

The *happened before* (Definition 5.1.3) relation yields a partial order of events, leaving some events unordered, due to them being concurrent; the precedence relation (Definition 5.2.1) preservers the ordering provided by the happened before relation, while trying to order the concurrent events in a way that preserves the users intentions.

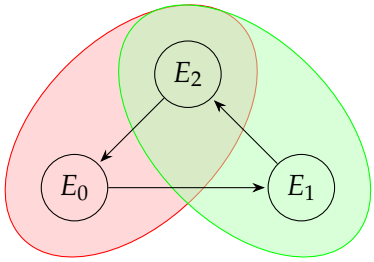

Figure 5.2: Precedence relation.

Figure 5.2 illustrates the example from the last section. The precedence relation is visualized by edges between the nodes; events that are concurrent are grouped. Notice that the precedence relation shows a cycle. The problem that needs to be solved, is choosing a path that visits every node exactly once, such that the precedence relation is satisfied between each pair of consecutive nodes. As discussed in the previous section, there are two possibilities in this example.

The history is a list of events, where the head of the list is the most recent event, according to the precedence relation $\prec$. In other words it is a list where the successor relation $\succ$ holds between each pair of consecutive events (where $\succ$ is defined as the inverse relation of $\prec$). The main reason for ordering the history by the successor (rather than the precedence) relation is performance. If there are no conflicts, every new event will be inserted at the head of the list. Assuming a fairly fast internet connection and the (comparably) slow pace of human typing, this is by far the most likely scenario. Adding events by the precedence relation would give linear time in the most likely scenario, but using the successor relation we can often avoid traversing the entire history.

When adding an event E to a history H, we could just add E to the first position where E succeeds the event to its right (if there is no conflict this would be the head of the list). However, in the example visualized in Figure 5.2, this approach would break user intent. We have not found a way to completely avoid breaking user intent, but we have found a way to make it less frequent. The idea is to find the first event (i.e. the most recent) that is not concurrent with the event that is being added, and let the event skip past events until it precedes the event to its left.

A function *collect* collects all events until it finds one that is not concurrent

with E; its dual, *drop*, skips all events until it finds one that is not concurrent and returns the remaining history (including the event it found). A function *put*, and a helper function put', are defined in order to place the event at a suitable position in the history. It is defined here; note that *nil* represents an empty history and white space is used to represent concatenation of lists and events (where events are treated as singleton lists under concatenation).

$$
\begin{aligned}
put(E, H) &= put'(E, collect(E, H))\ drop(E, H) \\
put'(E, nil) &= E \\
put'(E, H\ E') &= \begin{cases} H\ E'\ E & \text{if } E \prec E' \\ put'(E, H)\ E' & \text{otherwise} \end{cases}
\end{aligned}
$$

We can now look at an example where we build a history with the three events E_0, E_1 and E_2 from the last section.

$$
\begin{aligned}
put(E_0, nil) &= put'(E_0, nil) &&= E_0 \\
put(E_1, E_0) &= put'(E_1, nil)\ E_0 &&= E_1\ E_0 \\
put(E_2, E_1\ E_0) &= put'(E_2, E_1\ E_0) &&= E_1\ E_0\ E_2
\end{aligned}
$$

In the first equation the history is empty, and so there is no choice where to put E_0. When adding E_1 there is only one element in the history, namely E_0, which is performed by the same user — *collect* returns *nil* and *drop* returns a singleton list containing E_0. In the last equation *collect* collects the entire history, and E_2 is first compared with E_0 which it precedes, and is therefore added to the end of the history.

5.4 Transform the History

Thus far we have found a way to construct a history of events such that every operation is represented in the history and greatly reduces the number of inconsistencies that can arise. Our hope was that this approach would be sufficient to handle all conflicts, but it turns out that there can still arise inconsistencies between clients. Using Maude to analyze the system uncovered that 2.2% of states in the system, given a buffer of size two and 3 operations are inconsistent. To deal with the remaining portion of inconsistent states we apply transformation functions (discussed in Chapter 3).

The following example consists of a set of events where the history does not provide a correct result.

$$
\begin{aligned}
E_0 &= \langle O_0, t_0, m_0, u_0 \rangle = \langle ins(0,\ \texttt{a}), 0, 1, u_0 \rangle \\
E_1 &= \langle O_1, t_1, m_1, u_0 \rangle = \langle ins(2,\ \texttt{b}), 0, 2, u_0 \rangle \\
E_2 &= \langle O_2, t_2, m_2, u_1 \rangle = \langle del(0,\ \texttt{f}), 0, 3, u_1 \rangle
\end{aligned}
$$

The ordering decided by the algorithm from the last section is:

$$E_2 \prec E_0 \prec E_1$$

The problem is that executing the operation $O_2 \circ O_0 \circ O_1$, obtained from the ordering, is not possible to execute on the initial buffer `"f"`. The `f` is deleted, then an `a` at the beginning of the buffer, and finally a `b` is attempted to be inserted at position 2, which is beyond the bounds of the buffer. The reason that this occurs is that the first deletion *shrinks* the buffer, but no measure is taken to make sure that the position of O_2 is decremented accordingly. This problem can be resolved by transforming the operations.

We use two types of transformation functions, namely inclusion and exclusion; an inclusion transformation function is the kind we discussed in Chapter 3. Given two operations $O_i, O_j \in \mathcal{O}$, then an inclusion transformation of O_i with regards to O_j can be understood as "O_i as if O_j had already been applied". Exclusion transformation is the reverse, and transforming O_i with regards to O_j can be understood as "O_i as if O_j had not been applied".

Due to our ordering, we only need to transform against deletions to guarantee eventual consistency. It is however possible that including the complete set of transformations would better preserve user intent — due to time restrictions we have not been able to verify this.

We define two functions *it* and *et*, where *it* is the inclusion transformation function and *et* is the exclusion transformation function. *it* is derived from the (corrected) transformation from Figure 3.1 (page 16), whereas *et* is derived from [37]. Furthermore, we define both *it* and *et* for composed operations, which is derived from [19].

One alteration has been made. When applying an inclusion transformation on two deletions with the same position, the result is *nop*. To reverse this effect, *et* must be able to retrieve the deletion that was omitted (i.e. transformed to *nop*). Using an exclusion transformation on *nop* cannot be done, because it contains no information about what character was deleted. In [37] this is handled by keeping a lookup table. Instead, we store the information in the *nop* object itself, by letting *nop* (optionally) contain an operation; its semantics is not changed, meaning applying *nop* has no effect regardless.

Note that the following equations, defining *it* and *et*, uses the word *otherwise* as it is used in Maude. It means that if none of the above equations covers a particular case, then it is covered by the equation tagged with otherwise. This is important for the last case, which would match all

operations if not for this use of otherwise. *it* is defined as follows:

$$it(ins(p1,\ \texttt{c1}), del(p2,\ \texttt{c2})) = \begin{cases} ins(p1,\ \texttt{c1}) & \text{if } p1 \leq p2 \\ ins(p1-1,\ \texttt{c1}) & \text{otherwise} \end{cases}$$

$$it(\overbrace{del(p1,\ \texttt{c1})}^{O_i}, \overbrace{del(p2,\ \texttt{c2})}^{O_j}) = \begin{cases} del(p1,\ \texttt{c1}) & \text{if } p1 < p2 \\ del(p1-1,\ \texttt{c1}) & \text{else if } p1 > p2 \\ nop(O_i \circ O_j) & \text{otherwise} \end{cases}$$

$$it(O_i, O_j \circ O_k) = it(it(O_i, O_k), O_j) \quad \text{if } O_j \text{ is not composed}$$

$$it(O_i, O_j) = O_i \quad \text{otherwise}$$

The exclusion transformation is defined in a similar manner, where a position is incremented to reverse the effect of an inclusion transformation. In the cases where an inclusion transformation would omit an operation, the exclusion transformation retrieves the operation.

$$et(ins(p1,\ \texttt{c1}), del(p2,\ \texttt{c2})) = \begin{cases} ins(p1,\ \texttt{c1}) & \text{if } p1 \leq p2 \\ ins(p1+1,\ \texttt{c1}) & \text{otherwise} \end{cases}$$

$$et(del(p1,\ \texttt{c1}), del(p2,\ \texttt{c2})) = \begin{cases} del(p1,\ \texttt{c1}) & \text{if } p1 < p2 \\ del(p1+1,\ \texttt{c1}) & \text{otherwise} \end{cases}$$

$$et(nop(O_i \circ O_j), O_j) = O_i$$

$$et(O_i, O_j \circ O_k) = et(et(O_i, O_j), O_k) \quad \text{if } O_j \text{ is not composed}$$

$$et(O_i, O_j) = O_i \quad \text{otherwise}$$

These transformation functions are used to "fix up" the history after a new event has been added. The inclusion transformations assumes that two operations were generated from the same state (i.e. the same buffer) [37], but this is not always the case. To deal with these cases, exclusion transformations is used to "reset" the operation to a agreed upon state, then perform inclusion transformations on this operation, and finally, we perform the inclusion transformations for the effects that were excluded.

Given an event $E = \langle O, t, m, u \rangle$ the events that will have effected O can be found. These are the events with a smaller time stamp than m, or have the same user u that occur at an earlier point in the history. The event must be "reset" to an agreed upon state, which is decided by the event with the smallest token which is concurrent with E. In order to use transformations on multiple events, the operations of a history can be composed using the *compose* function. It is defined as:

$$\begin{aligned} compose(nil) &= nop \\ compose(\langle O, t, m, u \rangle\ H) &= O \circ compose(H) \end{aligned}$$

We define a function fix which takes a history and returns a "fixed up" version of the history:

$$
\begin{aligned}
&fix(nil) = nil \\
&fix(\langle O, t, m, u\rangle\ H) = \langle O_{fixed}, t, m, u\rangle\ H' \\
&\text{where } H' \quad = fix(H) \\
&\qquad E \quad = \langle O, t, m, u\rangle \\
&\qquad H_c \quad = \text{filter } H', \text{ only keep events that are concurrent with } E \\
&\qquad O_e \quad = \text{compose operations that have effected } O \\
&\qquad O_{fixed} \quad = it(it(et(O, O_e), compose(H_c)), O_e)
\end{aligned}
$$

By making sure the fix function is applied to the history after each insertion, we ensure that composing the entire history can always be safely applied to the initial buffer. What remains is to ensure that each client eventually will apply the operations of the history.

5.5 Ensuring Consistency

After having gone into great depths on how to construct a history, the question of getting all clients to conform to this history still remains. This section concludes the specification of the server-side algorithm, and shows how the history, and properties of editing operations, are leveraged to guarantee eventual consistency. The only nondeterministic behavior at the server is the reception of a message; this section describes the server's *reaction* to incoming messages.

Messages from the server are on exactly the same form as messages from a client, namely a $msg(O, t, s)$ where O is an operation, t is a token and s is a sequence number. When a client receives a message, the operation is only applied if the sequence number of the message s is equal to the client's local sequence number. If the client applies the operation, then it is guaranteed to be consistent with the history at time t. Being consistent with the history at time t is defined as: Compose the history as it was when the server's state token was t; if this operation is applied to the initial buffer and the resulting buffer is equal to the client's buffer, then the client is consistent with the history at time t.

On each reception the server sends a response to every connected client. The server constructs two operations, one to the client who sent the message and another to every other client. The token is the same in every message. The sequence number may vary for each client.

5.5.1 Sequence Number Scheme

Remember that a client always increments its sequence number after it performs an operation or receives a message from the server. Say we have

a set of users U who identify each connected client. The server keeps a sequence number $s_u \in \mathbb{N}$ for each participant $u \in U$, initialized at zero.

When a message is received from a client identified by $u \in U$, the server sends messages to every connected client, each of which contains a sequence number. For each client identified by the users in $u' \in U \setminus \{u\}$, the stored sequence number $s_{u'}$ is used, and incremented after the message is sent. It is incremented because the client will increment its sequence number at reception of the message, and will therefore expect a higher sequence number the next time it receives a message.

When a message $msg(O, t, s)$ is received from a client identified by u, then the response to this client will contain a sequence number determined by the function $nextSeq$:

$$nextSeq(s_u, s) = s_u + 1 + (s_u - s)$$

If there is no conflict then $s = s_u$, and so the response will just be the sequence number of the message incremented by one. It is incremented by one because sending the messages will have caused the client to increment its sequence number. If there is a conflict, then u must have performed one or more operations between the time the server sent the last message and the client received it. The difference $s_u - s$ is the precise number of operations that u has performed in this time span. Because each of these operations has caused the client's sequence number to increase, the difference is added in order to match the sequence number of the client.

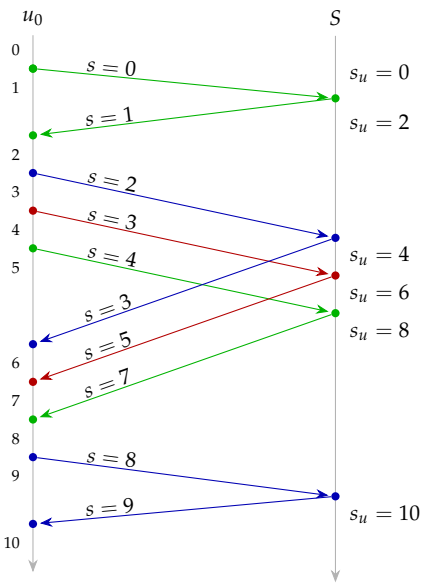

Figure 5.3: Sequence number scheme.

After the server has sent a response to the client, s_u is set to $s_u = nextSeq(s_u, s) + 1$. The extra one is added because the reception of the message will (again) cause the client's sequence number to be incremented.

The main benefits of this scheme are that it remains very simple on the client side (seeing that it only increments after every send or receive) and that it guarantees that clients do not perform operations from the server which are based on outdated information. Instead, the client rejects messages for the duration of a conflict with the server. The disadvantage is that the system is not responsive during long lasting conflicts, which may arise if a user, for instance, holds down a button on her keyboard for a long time. However, it seems reasonable to assume that such situations are unlikely to occur, and even less likely to cause problems in practical uses of the system.

5.5.2 Constructing Operations

When the server receives a message $msg(O, t, s)$ two operations are constructed; one for the client who sent the message, and one for all the other clients. We will discuss the message that is sent to all other clients first.

If the message was sent by a client identified by $u \in U$ and the message is received at time m, then it is assumed that all clients identified by some $u' \in U \setminus \{u\}$ is consistent with the history at time m. The event $\langle O, t, m, u \rangle$ is added to the history H, constructing a new history H'. A new message is constructed on the basis of H and H'.

If the new event is added to the beginning of the history (i.e. the most recent), then the constructed operation will always be O itself. Let us say O is in conflict with another operation O_i, with a lower position, then O must applied before the operation with the lower position is applied. Since it is assumed that the clients have already applied O_i, this must be undone before. When this is done, O may be applied followed by O_i.

A function *until* takes a history and a token, finds the earliest event that arrived at time greater or equal to the given token. It is defined as follows:

$$until(nil, t) = nil$$
$$until(H\ \langle O, t', m, u \rangle, t) = \begin{cases} H\ \langle O, t', m, u \rangle & \text{if } m \geq t \\ until(H, t) & \text{otherwise} \end{cases}$$

Now we can define a function *makeOp*, which takes two histories and a token. It is presumably called with the history H (as it was at time m, before the new event was added), and the history with the new event added H', along with the token t. It is defined as follows:

$$makeOp(H', H, t) = compose(until(H', t)) \circ compose(until(H, t))^{-1}$$

This operation will first undo operations that occurred at time t or later, followed by the operations from the updated history (at time t or later). As we discussed in 2.3.1, redundant operations on the form $O_i^{-1} \circ O_i$ can be omitted, which should be done (but is not strictly necessary) before the operation is sent.

The message $msg(makeOp(H', H, t), m + 1, s_{u'})$ is sent to all clients identified by some $u' \in U \setminus \{u\}$, where $s_{u'}$ may vary depending on the recipient. For each of these clients, the server keeps a mapping from u' to a list of pairs, containing an operation and a token [1]. The operation and token is added to the head of this list after sending the message.

[1] In the Maude specification, each user maps to a list of messages. It contains the message that was sent, but where the sequence number $s_{u'}$ is incremented by one. This is because the sequence number $s_{u'} + 1$ has to be stored somewhere, as discussed in 5.5.1, and a list of messages was already present in the specification, in order to model message queues.

The second case to consider is what to send to the client, identified by u, that originally sent the message $msg(O, t, s)$. If there is no conflict, then the constructed operation is always nop. It is important to note that if a conflict has arisen, it can only be because the client has performed operations in the time between the server last sent it a message, and the time the client received it. The token t indicates that the client was consistent with the history at time t.

Because every operation (along with a token indicating what time it was sent) that is sent to the client is stored, the operations the client have rejected is retrievable. It has rejected all operations with a corresponding token that is strictly greater than t. These operations have to be represented in the response.

The idea is to always make the client undo the operation O, then perform all the operations that it has rejected, and finally perform the operation that is sent to every other client (the operation constructed by $makeOp$). If there is no conflict, then the operation constructed by $makeOp$ is simply O and there are no operations stored operations to compose; the resulting operation is $O \circ O^{-1}$ which is equal to nop.

A simple helper function *rejected* is used. It takes a list of pairs (where each pair contains an operation and a token) R and a token t. The function returns the pairs with token strictly greater than t. Composing the resulting list is analogous to composing a history. The operation sent to the client who sent the message $msg(O, t, s)$ is given by a function *makeResponse* that takes four arguments: The received operation O, the operation $makeOp(H', H, t)$, the list that contains rejected operations (along with the corresponding tokens) and the token t as argument.

$$makeResponse(O, O', R, t) = O' \circ compose(rejected(R, t)) \circ O^{-1}$$

This operation can safely be applied by the client who sent the message $msg(O, t, s)$, and will make the client consistent with the history at time $m + 1$.

The last remaining detail is regarding the list of rejected operations (along with the corresponding token). The operation from *makeResponse* ensures that previously rejected operations are performed; it is very important not to perform these rejected operations again if a new conflict arises. To solve this problem, the list is replaced with a singleton list which only contains the operation constructed by *makeResponse* and the token $m + 1$.

A Summarizing Example

Let us revisit the example from Figure 5.4, where we include the operations that are sent to each client. The events received on the server is:

$$E_0 = \langle O_0, t_0, m_0, u_0 \rangle = \langle ins(0,\ \mathtt{a}), 0, 1, u_0 \rangle$$
$$E_1 = \langle O_1, t_1, m_1, u_1 \rangle = \langle ins(1,\ \mathtt{b}), 0, 2, u_1 \rangle$$
$$E_2 = \langle O_2, t_2, m_2, u_2 \rangle = \langle ins(0,\ \mathtt{c}), 2, 3, u_2 \rangle$$

The server decides on an ordering $E_1 \prec E_0 \prec E_2$, which means that all clients must perform the operation $O_2 \circ O_0 \circ O_1$. By composing all operations sent to a given client, we can calculate what operation they perform.

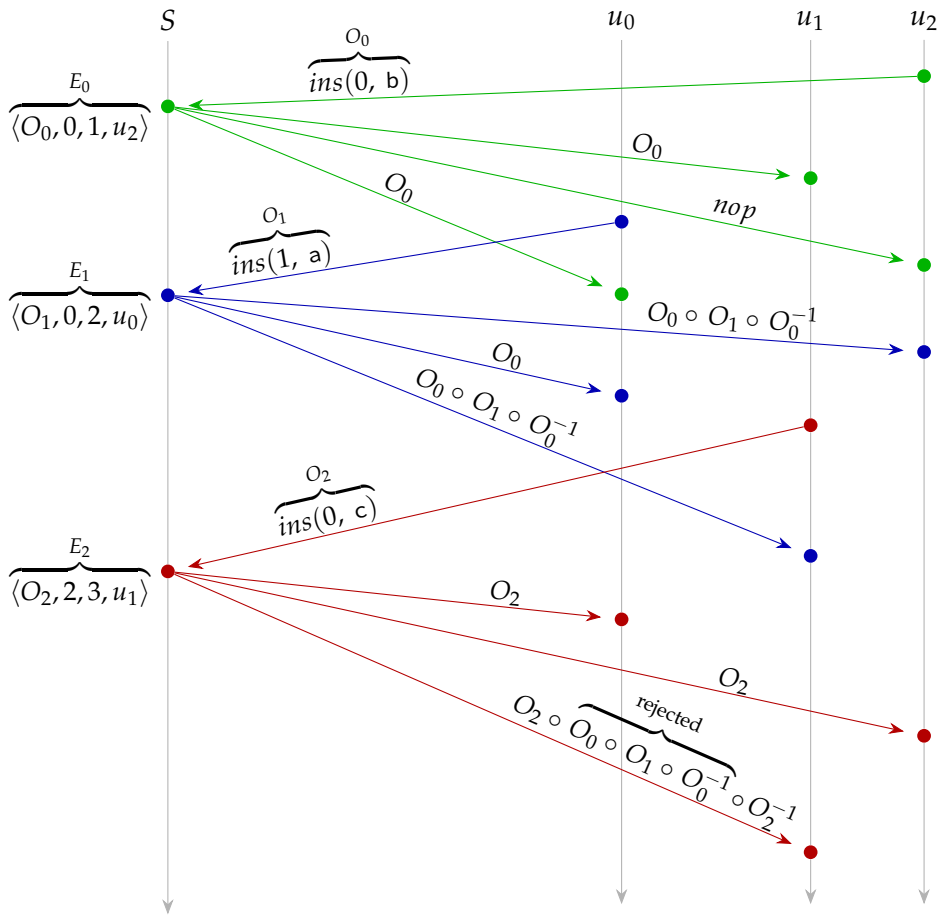

Figure 5.4: The events are ordered $E_1 \prec E_0 \prec E_2$.

From the perspective of u_0:

- O_1 is generated,
- O_0 is received,
- O_2 is received.

The composition of these operations yields the operation $O_2 \circ O_0 \circ O_1$.

From the perspective of u_1:

- O_0 is received,
- O_2 is generated,
- $O_0 \circ O_1 \circ O_0^{-1}$ is received, but rejected,
- $O_2 \circ O_0 \circ O_1 \circ O_0^{-1} \circ O_2^{-1}$ is received.

The composition of the operations (excluding the rejected operation) yields:

$$O_2 \circ O_0 \circ O_1 \circ O_0^{-1} \circ \overbrace{O_2^{-1} \circ O_2}^{nop} \circ O_0 =$$
$$O_2 \circ O_0 \circ O_1 \circ \overbrace{O_0^{-1} \circ O_0}^{nop} = O_2 \circ O_0 \circ O_1$$

Lastly, from the perspective of u_2:

- O_0 is generated,
- $O_0 \circ O_1 \circ O_0^{-1}$ is received,
- O_2 is received.

The composition of the operations yields:

$$O_2 \circ O_0 \circ O_1 \circ \overbrace{O_0^{-1} \circ O_0}^{nop} = O_2 \circ O_0 \circ O_1$$

5.6 Summary

This chapter is in essence a comprehensive walk through of the equations of the Maude specification that are related to the server-side of the specification. The reception of a message is the only nondeterministic behavior at the server; this is expressed as a single rewriting rule in Maude. This rule calls the functions we have described in the preceding sections.

The rewriting rule in Figure 5.5 concisely summarizes this chapter, as it describes the *reaction* to the reception of a message. The reception of a message is modeled as the server nondeterministically picking a message off of the end of the message outgoing queue of a client. It adds a response message to the incoming queue to the client. The expression `(send O' to US)` similarly adds a message containing the operation `O'` on all the other clients incoming queues with the correct sequence number.

The `sites` field is a mapping from users to the information the server stores about each client. Each site consists of a list of messages, which contains messages that might be rejected by the client; note that the sequence

```
crl [server-receive] :
    < U | out-queue : (Q msg(O, T, N)),
          in-queue  : Q' >
    < server | history : H, state : M, sites : (U |-> Q'', US) >
  =>
    < U | out-queue : Q,
          in-queue  : (msg(O'', s M, S) Q') >
    < server | history : H', state : s M, sites : US' >
    (send O' to US)
  if  H'  := fix(put(O T M U, H)) /\
      O'  := makeOp(H', H, T) /\
      S   := nextSeq(Q'', N) /\
      O'' := makeResponse(O, O', Q'', T) /\
      US' := (U |-> msg(O'', s M, s S), US) .
```

Figure 5.5: The rule expresses the reaction to the reception of a message.

number is incremented by one in order to comply with the sequence number scheme (Section 5.5.1). Also note that the rule is expressed as a *conditional* rule `crl` in order to achieve the effects of a *let*-expression, commonly found in functional programming languages.

In this chapter we have seen how the server orders events in a way that ensures that causality violation does not occur, and minimizes the need for transformation functions. We have demonstrated that this is not sufficient, and show how inclusion and exclusion transformation functions can be applied to produce a history which can dictate a safe order of which operations can be applied. Lastly, we have seen how consistency is ensured by constructing operations which can be applied by clients, leaving them consistent with the constructed history.

Chapter 6

Model Checking the Specification

The main advantage of writing a *formal* specification of a system is the possibility of analyzing properties of the system in an automated manner. We are convinced that using such tools have led us to detect problems with the algorithm that we otherwise would not, and provided useful output to further aid our understanding of the problem. In addition, it has provided us with confidence the algorithm is robust.

For this thesis, we have chosen to apply a well known technique for formal verification, called model checking[1]. Maude has good support for performing model checking, via its `search` command [5], and an LTL (Linear Temporal Logic) model checker [9]. We have taken a practical approach to model checking, using it as a tool to aid the development of the algorithm. In this chapter we will present how we have used it, and how we have been able to use the results of the model checker to enhance the algorithm.

In previous chapters, we have seen numerous examples of scenarios which could lead to inconsistencies unless certain measures are taken. All of these examples have been found using the Maude LTL model checker.

6.1 Always Eventually Consistent

Our goal for the algorithm is that it must guarantee *eventual consistency* [40]. It is typically used in the context of data replication. In the context of real-time collaborative editing, eventual consistency means that "if all users stop typing, then eventually they will all be looking at the same buffer". It is a weak consistency model, in the sense that it allows for an *inconsistency window*, meaning that clients can be out of sync for a period of time.

[1] For a good introduction to model checking, see [3].

Our motivation for using such a weak consistency model is that it is a reasonable minimum requirement for a real-time collaborative system. However, it does not provide any guarantees with regards to preserving *user intent*. If eventual consistency was the only requirement for our system, then our system could simply tell every client to delete the contents of their buffer at every change; this is obviously not a satisfactory solution.

We have not formalized any requirements with regards to user intent, and rather taken a more intuitive approach. As we demonstrated in Chapter 5, ensuring that conflicting operations with a higher position are applied before operations with a lower position preserves the users intentions. We choose to assume that this is by far the most likely scenario, as it seems hard to imagine that users will find it fruitful to persistently try to edit the exact same point of the buffer. This does not mean that we completely ignore the case where this situation would occur. The two requirements we set for such situations is that they do not break consistency, and do not result in instructing the clients to perform an operation which operates outside the bounds of their buffer.

We use LTL (Linear Temporal Logic) to formally express the property *always eventually consistent*. A temporal logic is a tool for reasoning about time, where, in LTL, time is simply represented as a sequence of states. The words *always* and *eventually* expresses temporal properties over sequences of states, while *consistent* is a property which states that all buffers in the system are equal.

6.2 Expressing Consistency in Maude

The Maude model checker module [9] provides a sort `Prop`; a term of the sort `Prop` is a property. We define a function symbol `consistent` which is of the sort `Prop` (and does not take any arguments). The *semantics* of a property is defined by equations using the operator `|=`, which takes a state and a property. A state is simply a term which represents all the components of the system, including the clients and the server.

A state being `consistent` is defined as "all buffers in the system are equal to each other". This is expressed by using a function that gathers all buffers in a given state, adding them to a set, and checking if the size of this set is (less than or) equal to one. We allow for an empty set, because a system with no clients is considered consistent. The property is formally specified as:

```
eq C |= consistent = | buffers(C) | <= 1 .
eq C |= consistent = false [owise] .
```

The equations state that a state `C` is a model for the property `consistent` if the size of a set containing all the buffers in the state is less than or equal to one. In all other cases, `C` is not a model for the property `consistent`.

6.3 Model Checking in Maude

Model checking is performed by supplying an initial state and a property. An initial state of our system is defined by a function `init` that takes three optional arguments; the initial buffer, the number of clients and the number of operations that can occur during a session. Called with no arguments it returns a state with an initially empty buffer, two clients and three operations, and reduces to the following term:

```
< server | history : nil,
           state : 1,
           sites : (user 0 |-> msg(nop, 0, 0),
                    user 1 |-> msg(nop, 0, 0)) >

< user 0 | buffer : nil,
           seqno : 0,
           token : 0,
           in-queue : nil,
           out-queue : nil >

< user 1 | buffer : nil,
           seqno : 0,
           token : 0,
           in-queue : nil,
           out-queue : nil >
```

It contains terms that represent server and two clients. The server consists of a history, a state token and a mapping from users to the server's information about the clients; the information it keeps is a list of rejected messages. We can perform model checking on this initial state, by calling the `modelCheck` function. The following expression immediately returns *true*:

```
reduce modelCheck(init, consistent) .
```

The reason for this is that the clients' buffers are identical, and so the initial state is consistent; there is no need to check any other states, because no temporal connectives has been added. We have leveraged two temporal connectives, namely *always*, symbolized by `[]`, and *eventually*, symbolized by `<>`.

The connective *always* takes a property, and its semantics is (informally) defined as "the property holds in this state, and all future states". Similarly, *eventually* takes a property, and its semantics is (informally) defined as "the property holds in this state, or in some future state".

Our system should be *eventually consistent*, but just checking if the initial term models the property `<> consistent` is not sufficient. This is because, initially, all buffers are equal, and so the property holds "in this, or some future state". The property `[]<> consistent` will ensure that the system

will *from every state* reach a consistent state at some point in time.

The vast majority of the experiments we have conducted has been on the form:

```
reduce modelCheck(init B N M, []<> consistent) .
```

where `B`, `N` and `M` determines the buffer, number of clients and number of operation. The experiment can reduce to `true`, indicating that the system is eventually consistent within the given bounds. If the property does not hold, the model checker provides a counterexample which shows a sequence of states that lead to an inconsistency. Lastly, the computation may not terminate within a reasonable amount of time consume or too much memory, which leaves us with an inconclusive result.

Some other experiments have been conducted using the `search` command, which preforms a breadth-first search of the state space from an initial state, until it matches a given pattern and satisfies an (optional) condition. The pattern is just a term with variables, and the condition may refer to these variables. This has been chosen in favor of using the LTL model checker where the property could easily be expressed as a pattern, and where we knew what to look for. Stated differently, we utilized the `search` command like one might utilize a debugging tool, trying to narrow down what caused a bug detected by the LTL model checker.

6.4 Experiments

The number of reachable states from an initial state depends on the size of the initial buffer, the number of clients and the number of operations. It grows exponentially with regards to the number of characters in the buffer; allowing insertions makes the buffer grow further, and so, increasing the number of operations causes the state space to grow drastically. We do not consider our endeavor especially successful with regards to verifying the system *effectively*.

Table 6.1: Model checking has verified eventual consistency with the following bounds.

Initial buffer size	Clients	Operations	CPU time
0	2	3	0.3 seconds
1	2	3	2.4 seconds
1	3	3	22.8 seconds
2	3	3	2 minutes 25 seconds
1	4	3	2 minutes 58 seconds
1	2	4	10 minutes 20 seconds
3	3	3	12 minutes 7 seconds
3	4	3	3 hours 8 minutes 18 seconds
2	2	4	3 hours 41 minutes 25 seconds

The experiments from Table 6.1, have been run on The Abel computer cluster[2], allowing us to run multiple experiments simultaneously. The computations are memory intensive jobs, which is why we decided on getting access to the cluster. Note that Maude is single threaded, so we were not able to leverage the many cores available in the cluster for single computations.

We have naturally tried to increase the bounds found in Table 6.1, but the experiments have not terminated within a week of computation, or exceeded the 16GB memory limit. Though a non-terminating computation does not prove anything, but it means that no counterexample was found within reasonable time.

Lastly, we want to emphasize that we consider the experiments that produced counterexamples, which are in a vast majority, to have been of great value to the development of the algorithm. Sadly, we cannot document these to the same degree as the ones that did not produce a counterexample. This is because that, for a long time, the specification was under rapid development, and counterexamples were produced within a short amount of time on a local computer. It was first when the specification started showing real promise that we needed to access more computing power in order to produce counterexamples. We have been able to correct all errors that have been found by the model checker, and have not been able to produce further counterexamples.

6.5 Processing Counterexamples

Model checking provides a counterexample if the system does not conform to a given property; these counterexamples have been extremely valuable for understanding what caused some particular problem. A counterexample is represented as a sequence of states, starting from a given initial state and ending in some inconsistent state.

By default, all Maude terms are separated by a single space, which for large terms quickly becomes difficult to read. The user can specify a format, which can greatly improve the readability of terms. The initial state showed in Section 6.3 exemplifies how we have chosen to represent a state in the system. In the representation of a state, all lines contain some piece of information we were interested in.

The counterexamples provided by the model checker typically span hundreds of lines, which makes them difficult to process. It is very useful to only look at *changes* between two given states.

Because each line in a state represents some specific part of the system, a simple program has been written to extract the difference between consecutive states. We found this program immensely useful, as it

[2] http://www.uio.no/english/services/it/research/hpc/abel/

generally reduces the size of the output by a factor between 2 and 3, and contained much less visual noise.

Our experience is that spending some time optimizing the visual representation of the output from the model checker greatly improves its usefulness, from a purely practical point of view.

Chapter 7

Implementation of Shared Buffer

Shared Buffer has been implemented by first writing a formal specification, and then writing a program that conforms to that specification. Care has been taken to write the specification in a way that eases the process of translating the specification to a programming language. This has mainly been done through minimizing the use of features specific to Maude, and specifying the algorithm at a fairly low level of abstraction. The act of implementing the system can be viewed as a test to see how well we have achieved this.

We provide a prototype implementation of a real-time collaborative system, where the server is written in Clojure and a client is developed as an extension for the text editor Emacs. The full source code of the Emacs extension is found in Appendix A. As for the server source code, the core is found in Appendix B, but not the complete implementation. The parts that are direct translations of functions covered thoroughly in Chapter 5 are not included, as well as some utility functions.

Note that the implementation is in a prototype stage, where some control mechanisms covered in Chapter 5 have not yet been implemented. This is due to time limitations, as there has been made changes to the specification close to the submission of the thesis.

This chapter attempts to bridge the gap between the specification and the implementation, and present solutions to problems that are not present in the abstract model of the system.

7.1 Editing Sessions

The server can handle multiple independent editing sessions, all of which are identified by a key. A client can either join a session by providing a key, or the server will create a new session with a random generated key. If

the client specifies a key which is not associated with any existing editing session, a new session is created which is identified by the specified key. The editing session exists as long as there are connected clients associated with the session.

For a client to join an editing session it must first establish a connection to the server. After a connection is established, the client sends a `connect-request` message, optionally specifying a session key. The client may not perform any operations until it receives the first message from the server. If no session key was provided, or the session did not already exist, then this is a `connect-response` message, containing a random generated session key. If the client joins an ongoing session, it receives one or more operations which will make it consistent with the other clients. After the first message is received, the client is considered initialized, and it may start to perform operations.

Because the server does not keep a copy of the buffer, the buffer must be fetched from an initialized client. This means that a client needs to be able to send its entire buffer on the server's request. Conflicts can arise during this process, which is discussed in Section 7.4. This is a design choice made in order to lower the memory consumption on the server. If we would later regret this decision, it would not be difficult to implement a "silent client" on the server (i.e. one that receives, but never generates operations), and always fetch the buffer from this particular client.

The server can handle changes in multiple editing sessions concurrently, because they are completely independent. Changes in the *same* editing session must be handled completely synchronously on the server.

7.2 Wire Protocol

Communication between a client and the server is done through WebSockets, which is a two-way communication protocol based on TCP [13]. It is chosen in order to achieve interoperability with modern browsers, as well as many programming languages. As WebSockets are TCP-based, it guarantees that data is not damaged, delivered in order and without duplication [30].

Messages are sent using the JSON object encoding format [2] as UTF-8-encoded strings. A JSON object is a collection of key/value pairs, where a key is a string, and a value can be a string, a number, a JSON object, an array, `true`, `false` or `null`.

Every message between clients and the server has a key `type`; its associated value determines the type of the message. A message may have several additional keys, depending on the message type. We first describe what each message of a given message type contains; this is specified in Tables 7.1 – 7.6. Some of the message types refers to an operation (of type object), which is specified in Table 7.7.

Table 7.1: Specification for messages of type `connect-request`.

Key	Type	Description
`session`	string	A string identifying a session (optional)

Table 7.2: Specification for messages of type `connect-response`.

Key	Type	Description
`session`	string	A string identifying a session

Table 7.3: Specification for messages of type `operation`.

Key	Type	Value description
`operation`	object	An operation
`session`	string	A string identifying a session
`seqno`	number	The sequence number
`token`	number	The state token number

Table 7.4: Specification for messages of type `operations`.

Key	Type	Value description
`operations`	array	An array of operations
`session`	string	A string identifying a session
`seqno`	number	The sequence number
`token`	number	The state token number

Table 7.5: Specification for messages of type `buffer-request`.

Key	Type	Value description
`session`	string	A string identifying a session

Table 7.6: Specification for messages of type `buffer-response`.

Key	Type	Value description
`operation`	object	An operation
`session`	string	A string identifying a session
`token`	number	The state token number

Table 7.7: Specification of operation objects.

Key	Type	Description
`ins`	string	The inserted string, present if `del` is not
`del`	string	The deleted string, present if `ins` is not
`pos`	number	The position of the operation

The diagram in Figure 7.1 describes the flow of messages from a client's perspective. It captures how a client must send a `connect-request` to get out of an uninitialized state, and must wait until it receives a message from the server, before it is ready to generate and receive operations. States that are labeled "Process", are there to show how most events (like a user typing or a message is received) require some sort of reaction, and that this must be completed before the client is free to continue typing or receiving messages.

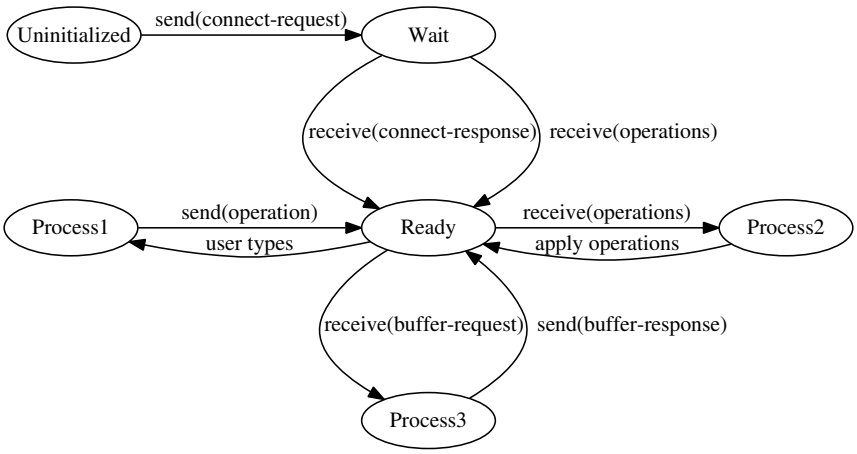

Figure 7.1: Client message flow diagram.

A similar diagram is provided in Figure 7.2 to show how different messages trigger different responses from the server.

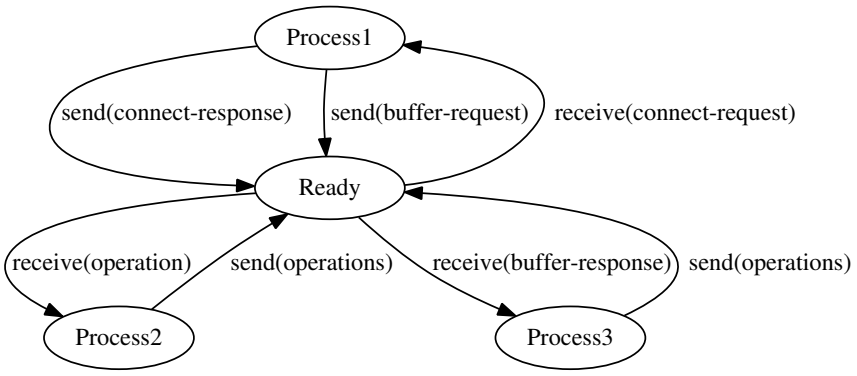

Figure 7.2: Server message flow diagram.

7.3 Minimal Implementation in Python

For testing purposes, a minimal implementation of the protocol has been implemented in Python. It connects to an editing session, and types randomly for a given number of seconds, and finally writes its buffer to file. The buffer is simply a Python string.

The function `on_open` is called when a connection to the sever has been established (how a connection is established is not covered here, as it is specific for what WebSocket library one chooses to use). `on_open` is given a socket and key as argument.

```
def on_open(ws, key):
    ws.send(json.dumps({'type' : 'connect-request', 'session' : key}))
```

We assume that certain global variables are defined, namely a key, a token, a sequence number and a buffer. When a message is received, a function `on_message` is triggered, and is given a socket and the message as arguments. The message must be converted to a Python dictionary, using a JSON library.

If the message type is `connect-response`, then the key is set. If it is a `buffer-request`, then a response is sent over the socket containing an operation which contains the entire buffer. Lastly, if the message type is `operations`, then we call a function which will be described shortly.

```
def on_message(ws, message):
    global key
    global token
    global buf

    msg = json.loads(message)

    if msg['type'] == 'connect-response':
        key = msg['session']
    elif msg['type'] == 'buffer-request':
        response = {'type':'buffer-response',
                    'session':key,
                    'token':token,
                    'operation' : {'pos':0, 'ins':buf}}
        ws.send(json.dumps(response))
    elif msg['type'] == 'operations':
        on_operation(msg)
```

When an operation is received, the following function is applied. If the sequence number matches the sequence number of the client, then the token is set, and the operations are applied. The sequence number is incremented regardless.

```
def on_operation(msg):
    global seqno
    global token

    if msg['seqno'] == seqno:
        token = msg['token']
        applyops(msg['operations'])
    seqno += 1
```

The function `applyops` traverses a list of operations, and inserts or deletes from the buffer at the given position.

```
def applyops(operations):
    global buf

    for op in operations:
        pos = op['pos']
        if op.has_key('ins'):
            buf = buf[0:pos] + op['ins'] + buf[pos:]
        elif op.has_key('del'):
            buf = buf[0:pos] + buf[pos + len(op['del']):]
```

When a (random) operation has been generated, this is applied, then sent over the socket. Finally the sequence number is incremented.

```
def on_change(ws, string, pos, msgtype):
    global key
    global seqno
    global token

    op = {msgtype:string, 'pos':pos }
    msg = {'type':'operation', 'session':key,
           'seqno':seqno, 'token':token, 'operation':op}

    applyops([op])
    ws.send(json.dumps(msg))
    seqno += 1
```

Together, these functions summarize a minimal client-side implementation of the Shared Buffer protocol.

7.4 Conflicts During Client Initialization

In the specification we assumed that the connected clients stayed constant throughout an editing session. The system needs to support clients connecting during an ongoing editing session, which needs to be handled in the implementation. Because the server does not have its own copy of the buffer, the buffer must be fetched from a client. This can happen during a conflict, which poses a problem.

When a client connects to an editing session, it is important that it waits until it receives the first message from the server; this is in order to avoid conflicts between the uninformed client and the server. This is a reasonable restriction because if the client connects to an ongoing editing session, we deem it unlikely that the client can provide useful contributions to the buffer before having received the buffer. Before the server sends this first message to the client, the client is said to be *uninitialized*.

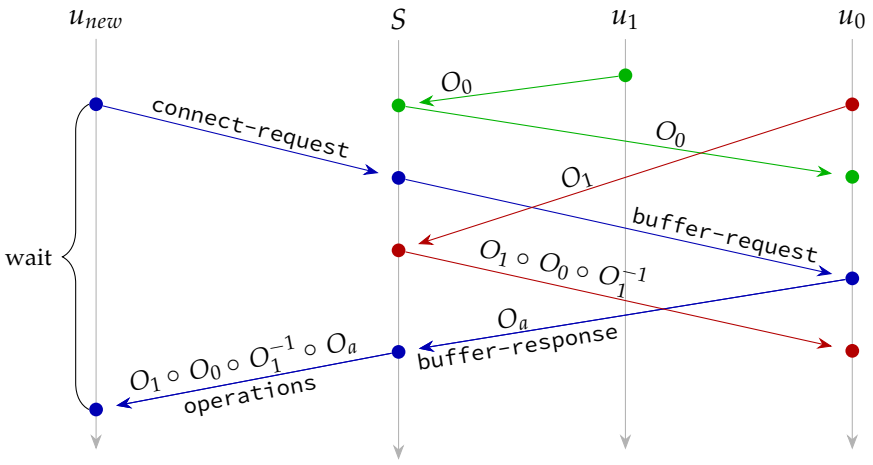

Figure 7.3: Client initialization.

The server arbitrarily chooses an initialized client, and sends it a message of type `send-buffer`. On reception, the client immediately responds with a message of type `buffer`, containing the `session`, the current `token` and an operation O_a which is an insertion at position zero with the entire contents of the buffer. It is not, however, guaranteed that the buffer is sent from a consistent state.

Remember from Section 5.5.2, that the server keeps a list of (possibly) rejected operations, each corresponding to a token for every client. When the server receives the `buffer` message, containing O_a, it constructs an operation which contains O_a and the operations needed to make the new client consistent with the current state. These operations are gathered by calling the *rejected* function with the list of rejected operations associated with the client that sent the `buffer` message, along with the token from the message. A function *makeInitialOp*, which is similar to *makeResponse* from Section 5.5.2, is defined as follows:

$$makeInitialOp(O_a, R, t) = compose(rejected(R, t)) \circ O_a$$

Where O_a is the operation that inserts the entire buffer, R is the list of operations (with corresponding tokens), and t is the token from the `buffer` message (i.e. the token at the time the `buffer` message was sent). A message containing a constructed operation (using *makeInitialOp*) is sent along with the server's current token and sequence number 0 to the new client, and it is set to initialized. Such a scenario is visualized in Figure 7.3 (note that responses to u_1 is omitted).

This solution is present in the prototype implementation.

7.5 Sequence and Token Numbers

There are two important considerations with regards to the sequence number scheme. One is that a sequence number should only be incremented on the client side when a sequence number is included in the message, i.e. when it generates or receives operations. This means that the sequence should for instance *not* be incremented at the client when it sends a message with type `buffer`.

The other consideration is with regards to that many languages keep an upper bound to the size of an integer. A number in JSON can be arbitrarily large, so it is feasible to let sequence and token numbers to grow arbitrarily large. Clojure supports arbitrarily large integers, so this is a possibility on the server side. But because we do not know what languages will implement the protocol we cannot assume that this is easily available for all implementation languages.

For instance, Emacs Lisp integers have an upper bound of $2^{29} - 1$ (on 32-bit platforms)[24]. Assuming that there is only one participant who types at 80 words per minute [29] (which is higher than the average typist) and words are of length six on average, then this user could type continuously for over a year before a problem occurs. On a 64-bit platform, they may type continuously for (roughly) as long as the sun has existed. The time it takes will decrease as more users connect to the editing session, but it still seems like a low-priority issue, and is therefore not solved in the prototype implementation.

For sequence numbers, the problem could be easily solved by using modulo arithmetic and some chosen n (for instance 2^{16}). The only relation used with the sequence number is the equality relation, and so, by always applying modulo n after calculating a sequence number then equality is ensured, assuming no conflicts are of size $2^{16} - 1$ or greater. This would have to be done at both the server and the clients.

The token numbers are compared with the less than relation, which is not preserved simply by using modulo arithmetic. In TCP this is solved by using the following definition for less than: "If s and t are timestamp values, $s < t$ if $0 < (t - s) < 2^{31}$, computed in unsigned 32-bit arithmetic"[21]. A similar scheme could be included in Shared Buffer, but choosing some lower bound in order to support Emacs (and potentially other languages). This can be achieved without adding any additional complexity on the client side, because the client only sets the token it receives from the server.

7.6 Implementing Shared Buffer for Emacs

A minimal client is written in *Emacs Lisp*, which is a programming language embedded in the text editor *Emacs*. Having a Lisp interpreter as a part of

the running program makes Emacs uniquely extendable, as code can be evaluated at run time, changing the behavior of Emacs without needing to recompile, or even restart the program. The implementation of Shared Buffer is at a very manageable size around 250 lines of Emacs Lisp code (where 100+ lines are documentation). This section covers the main considerations in writing a client for Emacs, and can be seen as an example on how to implement the protocol in an existing editor.

In Emacs, every piece of text is located in a buffer (often associated with some file). It is important that Shared Buffer is enabled for a single buffer at the time, if not, the changes made in other buffers would be communicated to the server, which would quickly break consistency. The idea of a buffer-local variable is important, as it allows for a single variable to have different values depending of the current buffer.

Shared Buffer stores its state in four buffer-local variables: A key that identifies an editing session, a reference to the socket for communicating with the server, a sequence number and a token number. The key is only set when starting a new editing session, or if a `connect` message is received from the server. The socket is only set when starting a new editing session. The sequence and token number is set according to the specification (Section 4).

For communication with the server, two libraries are utilized, namely WebSockets[1] and JSON[2] (which is built in). The WebSockets library provides a function for opening a connection to a given host, which can take functions as argument which are called on different events on the socket. The JSON library provides functions for encoding a native Emacs Lisp data structure to a JSON string, and similarly read JSON strings, which returns an Emacs Lisp data structure. Together, these libraries provide a very simple interface for communicating with the server.

The main challenge is to detect all changes in the buffer. In Emacs, a *hook* refers to a variable that contains a list of functions that are triggered by certain events. Emacs provides two hooks that are called before and after every change. A function for sending insertions is added to the `after-change-hook` and a function for sending deletions is added to the `before-change-hook`. Insertions must be detected *after* it has been performed, in order to retrieve the text that was inserted, and similarly, deletions must be detected *before* the change in order to retrieve the text that was deleted.

The documentation for change hooks clearly state that "These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local)"[34]. This is not completely true, due to the existence of a variable `inhibit-modification-hooks`. If this variable is set to a true value, then the change hooks are not executed, which poses a problem for Shared Buffer because *all* changes must be de-

[1] https://elpa.gnu.org/packages/websocket.html
[2] http://git.savannah.gnu.org/cgit/emacs.git/tree/lisp/json.el

tected. Some functions that binds this variable have caused problems with the current implementation of Shared Buffer.

We contacted the emacs-devel mailing list[3] asking whether this was intended, because it breaks the guarantee that the change hooks should provide. Stefan Monnier, a seasoned Emacs developer gave the following reply:

> Making "real" modifications to the buffer while binding inhibit-modification-hooks to a non-nil value sounds like a bug.

This gives the impression that using these hooks should be a viable option, and so we have not made efforts to get around this problem. The variable is toggled around 50 times in the Emacs source code, so submitting bug reports where the usage causes problems seems like a manageable task. It is worth noting that Emacs has a large ecosystem of external packages, and exploring all of these seems like a daunting task. It is possible to keep a list of functions that are known to set the variable in a way that interferes with Shared Buffer, and disable these during a shared editing session — this is assuming that the number functions is reasonably small.

We have two suggestions to alternative approaches. One is to rely on the undo-history that is stored in Emacs, in addition to keeping track of which changes have been communicated to the server. Another is to take regular "snapshots" of the buffer, and breaking the difference between a snapshot and the next into operations, which can then be communicated to the server. Bot of these may be viable alternatives for other editors that might not provide something similar to change hooks.

It is worth noting that Emacs runs in a single thread; this excludes many edge-cases, like what would happen if a message is received at the same time as the user generates an operation. This is something that is worth considering if one were to implement the protocol in a multithreaded environment.

A small set of functions are added to create an interface for the user; these functions are called *interactive* functions, which can be added to a key binding and called using `M-x`, which is Emacs' interface to all interactively callable functions. A function `sb-share-buffer` may be called to share the current buffer; a session key is randomly generated by the server and sent back in a `connect` message; the session key is added to the clipboard. To connect to an editing session, a function `sb-join-session` is provided, where the user must provide a session key. A function `sb-disconnect` simply disconnects the buffer to the shared editing session (but leaving the buffer as it was before disconnecting). Lastly, `sb-add-key-to-kill-ring` is a function to add the current session key to the clipboard ("kill-ring" is Emacs jargon for the clipboard), making it easier to share to others. Note that every function is prefixed with the abbreviation `sb`, which is an established code convention in Emacs, due to it having a single name space

[3] http://lists.gnu.org/archive/html/emacs-devel/2016-07/msg00721.html

for all global variables and functions.

7.7 Implementing Shared Buffer in Clojure

The main task of implementing Shared Buffer in Clojure is simply translating Maude equations to Clojure functions, most of which have been described in Chapter 5. This is mostly a straight forward task, due to the both being declarative languages.

Two libraries are leveraged to ease communication with the clients, namely HTTP Kit[4] and data.json[5]. Together they form a very similar interface to the Emacs Lisp equivalents (Section 7.6). The main difference is that messages on a WebSocket is received on a *channel* (i.e. a blocking queue), and messages from multiple channels can be processed concurrently.

7.7.1 State Management

Clojure separates *identity* and *state*, where an identity is "a stable logical entity associated with a series of different values over time"[6]. Though mutating variables directly is possible, it is greatly discouraged. There are three encouraged ways of dealing with mutating state which can be summarized as [15]:

- **Refs** — use for synchronous and coordinated state change,
- **Atoms** — use for synchronous and uncoordinated state change,
- **Agents** — use for asynchronous and uncoordinated state change.

Shared Buffer follows a common Clojure idiom, which is to keep the entire application state inside a single *atom*. Every change to an atom is *atomic*, meaning that it never has an intermediate state; either a change has happened or it has not. This generally yields a maintainable code base. Changes to atoms are mainly done by applying a function to the current state of the atom. It is important that the function is *pure* (i.e. no side effects), because it may be called multiple times.

The state is a nested map, keeping track of the sessions and the connected clients, along with a function for shutting down the server. Changes made to the state are limited to functions that *inherently* has side-effects, namely the functions that either sends or receives messages. Note that there is a correspondence between when we allow state changes in the system, and when we used rewrite rules in Maude. If an error occurs in a purely functional program, then a stack trace will give a very precise location of where the error occurred; either the function is erroneous or it is called with

[4] http://www.http-kit.org/
[5] https://github.com/clojure/data.json
[6] http://clojure.org/about/state

a bad argument; limiting the places state change occurs greatly reduces the potential locations of the error.

To get an idea of how the state is structured, we provide an example of a state. The state is a nested map, which maps keys to values, and is represented by surrounding the pairs with curly braces `{}`. The keys can be any value, but often keywords are preferred; they are symbols prefixed with `:`, and can be used as functions that look up values in a map. Lists are denoted as parenthesized expressions and sets are denoted with curly braces prefixed with a hash tag `#{}`. This is an actual state of the system, but some values has been replaced with variables for readability:

```
{:sessions {:key1 {:tokens {c1 4, c2 2},
                   :clients #{c2 c1},
                   :token 5,
                   :lock obj1,
                   :history h}},
 :clients {c1 {:id u1,
               :seqno 6,
               :session :key1,
               :initialized true,
               :ops r1},
           c2 {:id u2,
               :seqno 7,
               :session :key1,
               :initialized true,
               :ops r2}},
 :server shut-down-function}
```

The changes that happen within a single session mush be handled in a coordinated way; it is important a change in a session is based on updated information. On every receive, a lock associated with a session is held until the change is complete. This allows for independent sessions to be handled concurrently, with the small exception of making changes to the atom. We have tried to minimize the number of changes done to the atom, but think that there is still room for improvement here.

7.7.2 Operations

Maude has great support for defining algebraic data-types, which made defining operations very simple. In Clojure, most data is built from the language primitives, and is largely based on collections. As we showed in Section 2.3, operations under composition is a *monoid*. Basing the representation on a monoidal structure gives a guarantee that some key properties are preserved.

Lists under concatenation is a monoid[7]. This motivates the choice of using

[7] https://en.wikibooks.org/wiki/Haskell/Monoids

lists as the underlying representation of operations. A singleton operation (i.e. insertion or deletion) is a singleton list, containing a map with the keys `:ins` or `:del` (exclusively) and `:pos`. The composition function simply calls the concatenation function `concat`.

The empty list represents a *nop* element. A map which contains a key `:nop` is also treated as the *nop* element[8]; this is because an exclusion transformation can generate a *nop* element, in which case we need to store the excluded operation (as discussed in Section 5.4). Such an operation will only be stored in an event, and a client will never observe its existence, because it will be omitted before the operation is sent.

The following function defines the inverse of an operation (Definition 2.3.1) in Clojure:

```
(defn inv
  "Returns the inverse of x."
  [x]
  (if (seq? x)
    (map inv (reverse x))
    (rename-keys x {:ins :del, :del :ins})))
```

It reverses the list, and replaces all `:ins` with `:del` and vise versa.

Operations constructed by the Clojure-equivalents of *makeOp* and *makeResponse* can often be simplified by removing adjacent pairs of operations that are inverses of each other. In Maude this is easily achievable by stating an equation. This has to be done explicitly in Clojure.

The following Clojure function simplifies a given operation in linear time[9]. It looks at one element of the list (i.e. a map which represents a deletion or insertion) at the time and adds it to the stack *unless* it is a *nop* element or the inverse of the top of the stack; in that case, both the top of the stack and the element will be omitted. It can be seen as a special case of peephole optimiaztion[10].

```
(defn simplify [op]
  "Reduces op it to a minimal form. It removes nop elements and adjacent
  operations that are inverses of each other."
  (-> (fn [stack o]
        (cond (nop? o) stack
              (empty? stack) (conj stack o)
              (inverses? (peek stack) o) (pop stack)
```

[8] Note that a solution where a map containing `:nop` would serve as the sole identity element would work just as well; however it seemed redundant to add such an element only to later remove it in the simplification step. This is why the empty list is treated as an "alternative" identity element.

[9] This solution was found after noticing that the problem is almost identical to a programming challenge we happened to have solved. The exercise can be found here: https://open.kattis.com/problems/evenup.

[10] https://en.wikipedia.org/wiki/Peephole_optimization

```
            :else (conj stack o)))
    (reduce [] op) seq))
```

The operations that are sent to the clients are always simplified before they are sent. The list is reversed and before sending it to a client. This is because it is more natural to traverse a list (or array) form left to right at the client, applying operations while traversing. Lastly, the list of maps is converted to an array of JSON objects.

7.8 Remaining Work

The implementation is in an prototype stage. Most importantly, not all control mechanisms from Chapter 5 has yet been implemented. As translating the Maude specification has been quite straight forward, we do not foresee any particularly challenging problems with completing this, and is why we have prioritized a correct specification before a correct implementation.

By analyzing the specification, and removing the control mechanisms that are not yet implemented, then we can give some estimate at how well the implementation performs, with regards to consistency. Given two clients and three operations working on a buffer of initial size two, then eventual consistency is breached in roughly two percent of the states. This is assuming that all states are equally likely to be reached. We would argue that these situations are even less likely to occur in reality, because it would require users to simultaneously perform operations on the same position of the buffer.

There are also a lot of features that would be natural to add to a real-time collaborative editor that have not been discussed. For instance, it could be very useful to display the cursor of other participants; in addition, displaying highlighted regions simplifies communication about the buffer. Yet, we want to keep the system minimalistic and not include features that could be better solved with a separate tool; an example would be a chat feature. Features of this kind is discussed, as they are of little (or no) relevance to the main problem, namely ensuring eventual consistency.

7.8.1 String-wise Operations

The protocol allows for inserting and deleting *strings*, yet the specification only deals with single characters. We suggest a way of dealing with this, which allows us to remain faithful to the specification, in the sense that the history will consist of events where each operation contains a single character.

Given an event $\langle O, t, m, u \rangle \in \mathbb{E}$ where O is an operation containing a string, rather than a character, we can split the event into one event per character.

If we assume i is a position, `c` is a character and `s` is a string, and let `c s` be a string with `c` as the first character, then the idea can be expressed with this equation:

$$
\begin{aligned}
split(\langle ins(i,\ \texttt{c}), t, m, u\rangle) &= \langle ins(i,\ \texttt{c}), t, m, u\rangle \\
split(\langle ins(i,\ \texttt{c s}), t, m, u\rangle) &= \langle ins(i,\ \texttt{c}), t, m, u\rangle\ split(\langle ins(i+1,\ \texttt{s}), t, m, u\rangle)
\end{aligned}
$$

$$
\begin{aligned}
split(\langle del(i,\ \texttt{c}), t, m, u\rangle) &= \langle del(i,\ \texttt{c}), t, m, u\rangle \\
split(\langle del(i,\ \texttt{c s}), t, m, u\rangle) &= \langle del(i,\ \texttt{c}), t, m, u\rangle\ split(\langle del(i,\ \texttt{s}), t, m, u\rangle)
\end{aligned}
$$

The list of events can be treated one by one, emulating that the client has performed the operations in a character-wise fashion. In the specification we made sure that m is always unique, but this is in order to always have a tie breaker and decide whether an event happened before (Definition 5.1.3) another. This is not necessary when all the events are performed by the user, because an event by a user can never precede an event (that is already present in the history) performed by the same user.

We implemented this as a proof of concept. Though we are very confident that it preserves eventual consistency, it greatly degraded the performance of the system, to such a degree that we don't consider it a satisfactory solution. It might be possible to optimize this, but instead we want to rely on existing solutions to the problem.

In [39] Sun et al. introduced both exclusion and inclusion transformation functions for strings. This is a much more performant solution which should be preferred in the implementation. Most Operational Transformation algorithms work independently from the data that is transformed, as long as they satisfies the properties C_1 and C_2 (which were introduced in Section 3.2). We see no reason this should not be true for Shared Buffer as well. Even though C_2 has been shown not to hold for the transformation function of Sun et al., but this only effects fully distributed solutions, and should therefore not be a problem for Shared Buffer.

7.8.2 Agents in Favor of an Atom

In Section 7.7.1 we described how the state of the program is contained in a single atom. An alternative structure, which would relieve the need of a lock, is using one agent per session. Every agent keeps a queue of functions that will eventually be applied to its current state. A problem with this approach is that these agents would need to be placed in some piece of mutable state. Generally, reference types (i.e. refs, atoms and agents) do not nest well, as computations might be run multiple times. Having not been able to find a safe solution to this, we have decided to stay with the single atom solution.

7.8.3 Acknowledgement Messages

No acknowledgement messages are needed in Shared Buffer to ensure consistency. Yet, without acknowledgement messages, both the history and list of rejected messages can continue to grow indefinitely. If all clients have successfully received a token n or higher, then they have all executed the operations with arrival time n or smaller. This means they cannot operate independently from any of these operations, and so, no new event will be concurrent with an arrival time smaller than n. This allows us to safely remove these entries from the history.

If one client stays silent during a session, there is no way for the server to safely remove entries from the history. If we were to introduce regular acknowledgement messages from the clients, this would yield substantially better performance at the server, as there are multiple linear time functions applied to the history every time a message is received.

As of now, the history is trimmed on the server, but no acknowledgement messages are sent. Every message received from a client is an implicit acknowledgement, which yields good performance if all the clients are active. Some consideration should go into the frequency of the acknowledgement messages. A suggestion would be to send an acknowledgement when the token number has been raised n times, while the client has stayed silent. The n should be chosen on the basis of how much overhead sending acknowledgement messages causes, and how large the n needs to be before performance on the server is noticeably degraded.

Chapter 8

Conclusions and Future Work

In this concluding chapter, we start by summarizing how the use of formal methods has influenced the development of the algorithm. We present some ideas that we did not have time to pursue, but consider interesting candidates for future work. Lastly, we show what we have achieved with regards to the goals that we set for this thesis.

8.1 Leveraging Formal Methods

We are completely convinced that writing a formal specification, and using model checking to analyze it, have been essential to the development of the resulting algorithm of this thesis. Using Maude to specify the program required us to find a representation for every component in the program; this gave us a precise way of thinking, talking and writing about the program, which in itself has been immensely useful.

Using a model checker is comparable with having a comprehensive test suite for a system. A major advantage is that it will automatically test *every possible scenario*, and relieve the programmer of the responsibility of determining what scenarios are most likely to produce an error. Furthermore, testing a distributed system can be extremely difficult, as the correctness of the system is dependent on results that span over multiple computers. Furthermore, it is notoriously hard to write system-wide tests that are *reproducible*.

We have experienced two downsides of using formal methods. One is that writing a formal specification takes time, and the tools can be somewhat crude[1]. The other is that there will always be a gap between the abstract model of the program one is specifying, and the actual running program.

One major mistake was made in the specification that model checking could not detect. For a long time, a client would insert a character at the

[1] For instance, we experienced problems with commenting out sections of the Maude specification. Maude seem to parse comments, and sometimes mistaking it for code.

end of its buffer, if the position was outside the bounds of the buffer. This should yield an error; the server should never send an operation to a client which exceeds the bounds of that client's buffer.

After correcting the mistake, we redid the experiments and found that the algorithm did not guarantee eventual consistency after all. The problem was manifested in the fix function from Section 5.4. A challenge here was that it was hard to produce a counterexample, as it took about 12 hours of computation. We were able to correct this, but we suspect we could have found a more elegant solution if we had caught it earlier. After it was corrected we have not been able to produce further counterexamples and have redone all experiments we have previously executed.

This somewhat sobering experience taught us that no matter the tool, there is always room for human error.

8.2 Future Work

There is remaining implementation work, as not all control mechanisms are currently present in the implementation. We considered a correct specification to be far more important for the thesis. The prototype implementation gives us confidence that the ideas from the specification work in practice.

Here we summarize some ideas for future work.

8.2.1 Undo

The adOPTed-algorithm [33] tackles the problem of undo in a real-time collaborative editor; their idea is only to undo operations that are generated locally. The Shared Buffer system does not provide any special mechanisms for undo, but specific client implementations may provide solutions to this problem; the only requirement is that all changes (including undos) are reported to the server.

This problem would be best handled on the client side, as undo is an editor-specific feature. If we were to handle this on the server side, a message would need to include information revealing that an operation was caused by an undo. Then the server could only add undone operations which were originally performed by the same user, and ensure that operations generated by other users would be restored. Though it might provide a correct end result, it would be a confusing experience for the user, as it would need to perform global undos, and trust that remote operations would be restored.

As of now, the Emacs client ignores this problem, which only allows for global undo (i.e. undoing both local and remote operations). In Emacs, a user can mark a region, and only undo in this region. This means that

a user can solve this problem manually by marking region they want to undo. However, it would be preferable to do this automatically.

We are not sure if this is achievable without altering the existing undo functions in Emacs. Because Emacs supports undoing within a region, there must also exist mechanisms for undoing a change which is not the *latest* change; it seems plausible to use these mechanisms to "skip" undo entries that were generated remotely and achieve a good local undo.

8.2.2 Constructing a Total Order

In Section 5.2.1 we showed that the precedence relation $\prec$ (Definition 5.2.1) is not transitive, and thus not a total order. We have an idea that we think could restore transitivity, but have not had to include it in the specification; this is mainly due to a change in the specification would require us to redo all experiments.

The example showed the three events:

$$E_0 = \langle O_0, t_0, m_0, u_0 \rangle = \langle ins(0, \texttt{a}), 0, 1, u_0 \rangle$$
$$E_1 = \langle O_1, t_1, m_1, u_0 \rangle = \langle ins(1, \texttt{b}), 0, 2, u_0 \rangle$$
$$E_2 = \langle O_2, t_2, m_2, u_1 \rangle = \langle del(0, \texttt{f}), 0, 3, u_1 \rangle$$

The three events can be ordered in two different ways, none of which satisfy transitivity.

$$E_0 \prec E_1 \prec E_2$$
$$E_2 \prec E_0 \prec E_1$$

One possible way of resolving this could be to use *exclusion transformation* to reset the operations to the latest consistent state before comparing them. In this example this would mean to exclude O_0 from O_1, showing O_1 *as if* O_0 had not been executed first. Then we would have $E_2 \prec E_1$ because $\langle et(O_1, O_0), t_1, m_1, u_0 \rangle$ would give $\langle ins(0, \texttt{b}), t_1, m_1, u_0 \rangle$, and a deletion should precede an insertion if the positions are the same.

The observation was found too late to be able to implement it and properly verify it. If our intuition serves us correct, this could yield a better result with regards to preserving user intent, and could guarantee a transitive ordering.

8.2.3 Preserve User Intent Fully

The eventual consistency model does not formalize the notion of what a user intended. As a first step, we could enforce that a delete operation must always delete the correct character. As of now, the wrong character will occasionally be deleted, which is a violation of user intent, and is not considered a graceful handling of a conflict. Given a buffer of initial size

two, three clients and three operations, this will occur in 0.5% of states, assuming that all states are equally likely to be reached.

Formally verifying the specification using the consistency model described in Section 3.2 could be better suited for ensuring that user intention is preserved. Furthermore, it would be interesting to see if the suggestion from the last section would be enough to fully preserve user intent.

8.2.4 Community

When the prototype implementation is in a more production ready stage, it will be published with a free software license. Seeing that the aim of the developed protocol is to enable real-time collaboration in *existing editors*, the protocol would ideally be ported to a wide range of editors in the future. The developers who are best fit to implement this are probably the developers who are an active part of the community around each editor. We view fostering a community around the project as the most promising way of spreading our real-time collaboration protocol to a wide range of editors.

8.3 Conclusion

The goal of this thesis was producing a protocol for real-time collaboration which could be characterized as:

1. **Portable** — easy to implement in an *existing* plain text editor,
2. **Responsive** — regular usage of the editor should not be degraded by enabling real-time collaboration,
3. **Robust** — gracefully handle all conflicting editing operations.

Portability has been ensured by designing a client-side algorithm that is easy to implement. Compared to the naïve algorithm, Shared Buffer adds little implementation overhead. We consider this goal to have been fully achieved.

The system is *responsive*, in the sense that local editing operations are never prohibited, and sending a message after changes cause, at worst, a negligible delay. Existing *Operational Transformation* (OT) algorithms where the client performs transformations themselves are arguably more responsive, in the sense that intermediate results are displayed during a conflict. This is a necessary trade-off to achieve our wanted level of portability, and we consider the goal achieved.

Aided by formal methods, we have shown that no conflict can break eventual consistency under certain bounds. Occasionally a wrong character will be deleted, but we do not consider this a serious issue because this only

occurs when multiple concurrent operations are performed at the same position, and it does not affect eventual consistency. The system has not been proven to guarantee eventual consistency for an arbitrary number of clients and operations, yet we still think the system can rightfully be characterized as *robust*.

Though we see room for improvement, we think the protocol fits nicely in the intersection of portability, responsiveness and robustness.

Appendix A

Shared Buffer for Emacs — Source Code

```
;;; shared-buffer.el --- Collaberative editing in Emacs. -*- lexical-binding: t -*-

;; Copyright (C) 2016 Lars Tveito.

;; Author: Lars Tveito <larstvei@ifi.uio.no>
;; Version: 0.2.1
;; Package: shared-buffer
;; Package-Requires: ((cl-lib "1.0") (websocket "1.3"))

;; Contains code from GNU Emacs <https://www.gnu.org/software/emacs/>,
;; released under the GNU General Public License version 3 or later.

;; Shared buffer is free software; you can redistribute it and/or modify it
;; under the terms of the GNU General Public License as published by
;; the Free Software Foundation; either version 3, or (at your option)
;; any later version.
;;
;; Shared buffer is distributed in the hope that it will be useful, but
;; WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General
;; Public License for more details.
;;
;; You should have received a copy of the GNU General Public License
;; along with Shared buffer.  If not, see <http://www.gnu.org/licenses/>.

;;; Code:

;;; Requirements

(require 'websocket)
(require 'cl-lib)
```

```
(require 'json)

;;; Variables

(defvar-local sb-key nil
  "A key that identifies the editing session.

For convenience `sb-add-key-to-kill-ring' copies the key to the
kill ring, so that you may share it.")

(defvar-local sb-socket nil
  "The socket used for communicating with the server.

The variable is set by `sb-connect-to-server'. ")

(defvar-local sb-seqno 0
  "A sequence number that is incremented at every interaction with the server.")

(defvar-local sb-token 0
  "A number that represents the last consistent state.

It is only updated when an operation is received.")

;; Changing major-mode should not affect Shared Buffer.
(dolist (var '(sb-key
               sb-socket
               sb-seqno
               sb-token
               shared-buffer-mode
               after-change-functions
               before-change-functions))
  (put var 'permanent-local t))

(defalias 'sb-substr 'buffer-substring-no-properties)

;;; Socket communication

(defun sb-connect-to-server (host)
  "Establishes connection with HOST, and binds `sb-socket'."
  (let ((host (concat "ws://" (or host "localhost") ":3705")))
    (setq sb-socket (websocket-open host
                                    :on-open    #'sb-on-open
                                    :on-message #'sb-receive
                                    :on-close   #'sb-on-close))
    (when sb-socket
      (setf (process-buffer (websocket-conn sb-socket))
            (current-buffer)))))
```

```
(defun sb-on-open (websocket)
  "This function is called when a connection to WEBSOCKET is established.

It sends an a request to connect to a session, specified by
‘sb-key’, and waits for a reply."
  (with-current-buffer (process-buffer (websocket-conn websocket))
    (sb-send (list :type 'connect-request :session sb-key))
    (with-local-quit
      ;; Block until first message is received
      (accept-process-output (websocket-conn websocket)))))

(defun sb-on-close (websocket)
  "Called when a connection to WEBSOCKET is closed."
  (with-current-buffer (process-buffer (websocket-conn websocket))
    (message "Shared Buffer: Connection closed!")
    (sb-quit)))

;;; Send

(defun sb-send (data)
  "Encode DATA to a JSON string and send it over ‘sb-socket’."
  (websocket-send-text sb-socket (json-encode data)))

(defun sb-send-buffer ()
  "Send an insertion that includes the entire buffer."
  (sb-send (list :type 'buffer-response
                 :session sb-key
                 :operation
                 (list
                  :pos (1- (point-min))
                  :ins (sb-substr (point-min) (point-max)))
                 :token sb-token)))

(defun sb-send-operation (type beg end)
  "Send an operation.

TYPE specifies if it is a deletion or insertion, whilst BEG and
END specifies what part of the buffer is sent."
  (sb-send (list :type 'operation
                 :session sb-key
                 :operation
                 (list
                  type (sb-substr beg end)
                  :pos (1- beg))
                 :token sb-token
                 :seqno (1- (cl-incf sb-seqno)))))

(defun sb-send-insertion (beg end len)
```

```
  "Send an insertion.

Called after every change, where BEG and END specifies the part
of the buffer which is changed. If LEN is zero then the change
was a deletion, and nothing is sent."
  (when (zerop len) (sb-send-operation :ins beg end)))

(defun sb-send-deletion (beg end)
  "Send an insertion.

Called before every change, where BEG and END specifies the part
of the buffer which is about to be changed. If BEG is equal to
END, then the change was a deletion, and nothing is sent."
  (when (not (= beg end)) (sb-send-operation :del beg end)))

;;; Receive

(defun sb-receive (websocket frame)
  "Receive a message on WEBSOCKET.

The message is inside the FRAME. Every message has a type, which
is either \"connect\", \"send-buffer\" or \"operations\"."
  (with-current-buffer (process-buffer (websocket-conn websocket))
    (let* ((payload (websocket-frame-payload frame))
           (json-object-type 'plist)
           (json-array-type 'list)
           (data (json-read-from-string payload))
           (type (plist-get data :type)))
      (cond ((string= type "connect-response")
             (sb-set-session (plist-get data :session)))
            ((string= type "buffer-request")
             (sb-send-buffer))
            ((string= type "operations")
             (sb-apply-operations data))
            (t (error "Shared Buffer: Error in protocol"))))))

(defun sb-set-session (session)
  "Change the editing SESSION.

The editing session is only set if the connection was established
using `sb-share-buffer'."
  (setq sb-key session)
  (sb-add-key-to-kill-ring)
  (message "Shared Buffer: \
Connected to session: %s, the key was added to the kill ring." session))

;;; Buffer modification
```

```
(defun sb-apply-operations (data)
  "Apply operations received from the server.

‘sb-token’ is set, and ‘sb-seqno’ is incremented.
Argument DATA is a plist with fields :seqno, :operations
and :token."
  (when (= sb-seqno (plist-get data :seqno))
    (cl-mapc 'sb-apply-operation (plist-get data :operations))
    (setq sb-token (plist-get data :token)))
  (cl-incf sb-seqno))

(defun sb-apply-operation (operation)
  "Apply a given OPERATION.

The operation is either an insertion or a deletion at a given
position."
  (save-excursion
    (let ((inhibit-modification-hooks t)
          (point     (1+ (plist-get operation :pos)))
          (insertion (plist-get operation :ins))
          (deletion  (plist-get operation :del)))
      (when (and insertion deletion)
        (error "Shared Buffer: Error in protocol"))
      (goto-char point)
      (when insertion (insert insertion))
      (when deletion  (delete-char (length deletion))))))

;;; Interactive functions

(defun sb-share-buffer (host &optional buffer)
  "Share the current buffer to enable real-time collaboration.

HOST specifies the host address of the server. A key will be
generated by the server, and added to the kill ring when
received. This key can be shared so that other can join your
editing session. Use ‘sb-join-session’ to connect to an existing
editing session.

If called non-interactively, a BUFFER may be specified."
  (interactive "sHost: ")
  (with-current-buffer (or buffer (current-buffer))
    (sb-connect-to-server (if (string= "" host) nil host))
    (if (not sb-socket)
        (message "Shared Buffer: Connection failed.")
      (shared-buffer-mode 1))))

(defun sb-join-session (host session)
  "Connect to HOST to join SESSION.
```

```
Like ‘sb-share-buffer’, except that a new buffer with the name
SESSION is created with the contents of the shared buffer."
  (interactive "sHost: \nsSession: ")
  (let ((buffer (generate-new-buffer session)))
    (switch-to-buffer buffer)
    (setq sb-key session)
    (sb-share-buffer host buffer)))

(defun sb-disconnect (&optional buffer)
  "Disconnect the current shared buffer session.

If called non-interactively a BUFFER may be specified."
  (interactive)
  (when shared-buffer-mode
    (with-current-buffer (or buffer (current-buffer))
      (websocket-close sb-socket))
    (sb-quit)))

(defun sb-add-key-to-kill-ring ()
  "Add the session key to the kill ring, so you may share it."
  (interactive)
  (kill-new sb-key))

;;; Minor mode

(defun sb-quit ()
  "Disable ‘shared-buffer-mode’ and clean up."
  (mapc #’kill-local-variable
        ’(sb-key sb-host sb-socket sb-seqno sb-token))
  (shared-buffer-mode 0)
  (remove-hook ’before-change-functions ’sb-send-deletion)
  (remove-hook ’after-change-functions ’sb-send-insertion))

(define-minor-mode shared-buffer-mode
  "A minor mode for Shared Buffer."
  nil " SB" nil
  ;; Make toggling shared-buffer-mode call sb-share-this-buffer, or
  ;; sb-disconnect if the mode is enabled.
  (add-hook ’before-change-functions ’sb-send-deletion t t)
  (add-hook ’after-change-functions  ’sb-send-insertion t t))

(provide ’shared-buffer)

;;; shared-buffer.el ends here
```

Appendix B

Shared Buffer Server — Source Code

```clojure
(ns shared-buffer-server.core
  (:gen-class)
  (:use org.httpkit.server)
  (:require [shared-buffer-server.app :refer :all]
            [shared-buffer-server.utils :refer :all]
            [shared-buffer-server.history :refer :all]
            [clojure.data.json :as json]))

(defonce state (atom {:sessions {} :clients {}}))

(defn clients-in-session [state key]
  "Given a session key, return the number of clients the session."
  (-> state :sessions key :clients count))

(defn empty-session? [state key]
  "Given a key, return true if there are no clients in the session."
  (zero? (clients-in-session state key)))

(defn initialized? [state client]
  "Returns true if the given client is initialized."
  (get-in state [:clients client :initialized]))

(defn get-initialized-client [state key]
  "Get an initialized client from a session identified by key."
  (->> state :sessions key :clients
       (filter (partial initialized? state)) rand-nth))

(defn get-uninitialized-clients [state key]
  "Get all uninitialized clients from a session identified by key."
  (->> state :sessions key :clients
       (remove (partial initialized? state))))
```

```
(defn min-token [session]
  "Get the smallest state token among clients in the session."
  (->> session :tokens vals (apply min)))

(defn next-seq [n m]
  "Given the a stored sequence number for a client and the sequence number
  of a message, calculate the next sequence number for the client."
  (+ (inc n) (- m n)))

(defn initialize-client
  "Adds a given client to the state."
  [state client]
  (-> state
      (assoc-in [:clients client :id] (hash client))
      (assoc-in [:clients client :seqno] 0)))

(defn join-session
  "Add a client to the session unconditionally."
  [state client key]
  (-> state
      (assoc-in  [:clients client :session] key)
      (assoc-in  [:clients client :initialized] (empty-session? state key))
      (assoc-in  [:sessions key :tokens client] 0)
      (update-in [:sessions key :clients] (fnil conj #{}) client)
      (update-in [:sessions key :token] (fnil identity 1))
      (update-in [:sessions key :lock] (fnil identity (Object.)))))

(defn dissolve-client
  "Remove a given client from the state. If this is the only client in the
  session, then dissolve the session."
  [state client status]
  (let [key   (get-in state [:clients client :session])
        state (dissoc-in state [:clients client])]
    (if (= 1 (count (get-in state [:sessions key :clients])))
      (dissoc-in state [:sessions key])
      (update-in state [:sessions key :clients] disj client))))

(defn update-client [state client seqno op f]
  "Update the sequence number and list of (possibly) rejected operations of
  a given client. Function f is either conj or a function that replaces the
  list."
  (-> state
      (assoc-in  [:clients client :seqno] seqno)
      (update-in [:clients client :ops] f op)))

(defn update-session [state key history]
  "Update a session with a new history, and increment it's state token."
```

```
  (-> state
      (assoc-in  [:sessions key :history] history)
      (update-in [:sessions key :token] inc)))

(defn next-state [state client token seqno op history]
  "Updates the session and the client, and trims the history."
  (let [key (get-in state [:clients client :session])
        t   (-> state :sessions key min-token)]
    (-> state
        (update-client client seqno op (fn [_ x] (list x)))
        (assoc-in [:sessions key :tokens client] token)
        (update-session key (trim-history history t)))))

;;; Send

(defn make-msg [key op seqno token]
  "Given a operation, sequence number and token, generate a message."
  {:type :operations
   :session key
   :operations (reverse op)
   :seqno seqno
   :token token})

(defn send-op!
  "Sends an operation to a set of clients."
  [key op token clients]
  (doseq [c clients]
    (let [site (get-in @state [:clients c])
          seqno (:seqno site)
          msg (make-msg key op seqno token)]
      (when (get-in @state [:clients c :initialized])
        (send! c (json/write-str msg)))
      (swap! state update-client c (inc seqno) [op token] conj))))

(defn send-buffer-request [state key]
  "Send a buffer request to some initialized client."
  (->> {:type :buffer-request :session key}
       (json/write-str)
       (send! (get-initialized-client state key))))

;;; Receive

(defmulti receive
  "Receive is dispatched on the message type."
  (comp keyword :type))

(defmethod receive :buffer-response [msg client]
  "Given a buffer, send an operation to all uninitialized clients, making
```

```
  them consistent with the current history."
  (locking (get-in @state [:sessions (keyword (:session msg)) :lock])
    (let [key (keyword (:session msg))
          op  (list (-> msg :operation))
          ops (get-in @state [:clients client :ops])
          op2 (make-initial-op op ops (:token msg))
          msg (make-msg key op2 0 (-> @state :sessions key :token))]
      (doseq [c (get-uninitialized-clients @state key)]
        (send! c (json/write-str msg))
        (swap! state assoc-in  [:clients c :initialized] true)
        (swap! state update-in [:clients c :seqno] inc)))))

(defmethod receive :connect-request [msg client]
  "Given a connect-request, add the client to the session specified in message,
  or generate a new session. If the session is ongoing, then fetch the
  buffer from a client."
  (let [key (or (:session msg) (generate-key))]
    (swap! state join-session client (keyword key))
    (when (or (not (:session msg))
              (= 1 (clients-in-session @state (keyword key))))
      (->> {:type :connect-response :session key}
           (json/write-str)
           (send! client)))
    (when-not (initialized? @state client)
      (send-buffer-request @state (keyword key)))))

(defmethod receive :operation [msg client]
  "Given a new operation, add the operation to the history, and send operations
  to all clients, making them consistent with the current history."
  (locking (get-in @state [:sessions (keyword (:session msg)) :lock])
    (let [key     (keyword (:session msg))
          session (get-in @state [:sessions key])
          site    (get-in @state [:clients client])
          op      (list (-> msg :operation))
          seqno   (next-seq (:seqno msg) (:seqno site))
          token   (:token msg)
          time    (inc (:token session))
          event   [op token (:token session) (:id site)]
          hist    (add-event (:history session) event)
          op1     (make-op (:history session) hist token)
          op2     (make-response op op1 (:ops site) token)
          reply   (make-msg key op2 seqno time)
          rest    (disj (:clients session) client)]
      (send-op! key op1 time rest)
      (send! client (json/write-str reply))
      (swap! state next-state client token (inc seqno) [op2 time] hist))))

(defmethod receive :default [msg client]
```

```
  "If none of the above, then print the message for debugging purposes."
  (println "undefined message type" msg))

;;; Handle requests

(defn receiver [client]
  "Returns a function that, that reads the json string and calls receive."
  (fn [data]
    (-> data
        (json/read-str :key-fn keyword)
        (receive client))))

(defn handler
  "The handler for incoming requests."
  [req]
  (with-channel req channel
    (if (websocket? channel)
      (swap! state initialize-client channel)
      (send! channel (app req)))
    (on-close channel (partial swap! state dissolve-client channel))
    (on-receive channel (receiver channel))))

;;; Server management

(defn stop-server
  "Stops the server, and resets all variables to their initial value."
  []
  (when-not (nil? (:server @state))
    ((:server @state) :timeout 100)
    (reset! state {:sessions {} :clients {}})))

(defn -main
  "The main function for Shared Buffer. It simply starts the server."
  [& args]
  (swap! state assoc :server (run-server #'handler {:port 3705})))
```

Appendix C

Maude Specification — Source Code

```
load model-checker

--- Redefine CONFIGURATION for format
mod CONFIGURATION is
    sorts Attribute AttributeSet .
    subsort Attribute < AttributeSet .
    op none : -> AttributeSet  [ctor] .
    op _,_ : AttributeSet AttributeSet -> AttributeSet
        [ctor assoc comm id: none format (d d ntsss d)] .

    --- Removed Msg
    sorts Oid Cid Object Portal Configuration .
    subsort Object Portal < Configuration .
    op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
    op none : -> Configuration [ctor] .
    op __ : Configuration Configuration -> Configuration
        [ctor config assoc comm id: none format (d nn d)] .
    op <> : -> Portal [ctor] .
endm

--- Likewise for MAP
fmod MAP{X :: TRIV, Y :: TRIV} is
    protecting BOOL .
    sorts Entry{X,Y} Map{X,Y} .
    subsort Entry{X,Y} < Map{X,Y} .

    op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
    op empty : -> Map{X,Y} [ctor] .
    op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
        [ctor assoc comm id: empty prec 121 format (d r onttssss d)] .
    op undefined : -> [Y$Elt] [ctor] .
```

```
    var D : X$Elt .
    vars R R' : Y$Elt .
    var M : Map{X,Y} .

    op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
    eq insert(D, R, (M, D |-> R')) =
      if $hasMapping(M, D) then insert(D, R, M)
        else (M, D |-> R)
        fi .
    eq insert(D, R, M) = (M, D |-> R) [owise] .

    op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 23] .
    eq (M, D |-> R)[D] =
      if $hasMapping(M, D) then undefined
        else R
        fi .
    eq M[D] = undefined [owise] .

    op $hasMapping : Map{X,Y} X$Elt -> Bool .
    eq $hasMapping((M, D |-> R), D) = true .
    eq $hasMapping(M, D) = false [owise] .
endfm

fmod STR is
    sort Str .
    ops a b c d e f g h i : -> Str .
endfm

view Str from TRIV to STR is
    sort Elt to Str .
endv

fmod BUFFER is
    including LIST{Str} .
    including STR .

    sort Buffer .
    subsort List{Str} < Buffer .
endfm

view Buffer from TRIV to BUFFER is
    sort Elt to Buffer .
endv

fmod ERROR is
    including BUFFER .
```

```
    sort Error .
    subsort Buffer < Error .

    op error : -> Error [format (r o)] .

    var S : Str .

    eq S error = error .
    eq error S = error .
endfm

mod UNIQUE is
    including CONFIGURATION .
    including LIST{Str} .

    sort Unique .

    subsort Unique < Configuration .

    op [ _ ] : List{Str} -> Unique .
endm

fmod USER is
    including NAT .

    sort User .
    op user_ : Nat -> User .
endfm

view User from TRIV to USER is
    sort Elt to User .
endv

mod ACTIONS is
    including CONFIGURATION .
    including USER .
    including SET{User} .
    including NAT .

    sorts Actions .
    subsort Set{User} < Actions < Configuration .

    op empty : -> Actions [ctor] .
    op _-_ : Actions Actions -> Actions [ctor assoc id: empty] .
endm

fmod ABSTRACT-OPERATION is
```

```
    sort Op .

    op nop : -> Op [ctor] .
    op nop : Op -> Op [ctor] .

    vars O O' : Op .

    op inv : Op -> Op .
    eq inv(nop) = nop .
    eq inv(inv(O)) = O .
    eq inv(O o O') = inv(O') o inv(O) .

    op _o_ : Op Op -> Op [assoc] .
    eq O o nop = O .
    eq nop o O = O .
    eq O o inv(O) = nop .
endfm

view Op from TRIV to ABSTRACT-OPERATION is
    sort Elt to Op .
endv

fmod INSERT-DELETE is
    including ABSTRACT-OPERATION .
    including NAT .
    including STR .

    sort Type .

    ops ins del comp : -> Type .
    op ins : Nat Str -> Op [ctor format (g o)] .
    op del : Nat Str -> Op [ctor format (r o)] .

    var N : Nat .
    var S : Str .
    vars O O' : Op .

    op pos : Op -> Nat .
    eq pos(ins(N, S)) = N .
    eq pos(del(N, S)) = N .

    op type : Op -> Type .
    eq type(ins(N,S)) = ins .
    eq type(del(N,S)) = del .
    eq type(O o O') = comp .
endfm

fmod TRANSFORMATION is
```

```
    including INSERT-DELETE .

    vars N N' : Nat .
    vars S S' : Str .
    vars O O' O'' : Op .

    op  it : Op Op -> Op .
    ceq it(ins(N, S), del(N', S')) = ins(N, S) if N <= N' .
    eq  it(ins(N, S), del(N', S')) = ins(sd(N, 1), S) [owise] .

    ceq it(del(N, S), del(N', S')) = del(N, S) if N < N' .
    ceq it(del(N, S), del(N', S')) = del(sd(N, 1), S) if N > N' .
    eq  it(del(N, S), del(N', S')) = nop(del(N, S) o del(N', S')) [owise] .

    ceq it(O, O' o O'') = it(it(O, O''), O') if type(O') =/= comp .
    eq  it(O, O') = O [owise] .

    op  et : Op Op -> Op .
    ceq et(ins(N, S), del(N', S')) = ins(N, S) if N <= N' .
    eq  et(ins(N, S), del(N', S')) = ins(s N, S) [owise] .

    ceq et(del(N, S), del(N', S')) = del(N, S) if N < N' .
    eq  et(del(N, S), del(N', S')) = del(s N, S) [owise] .

    eq et(nop(O o O'), O') = O .

    ceq et(O, O' o O'') = et(et(O, O'), O'') if type(O') =/= comp .
    eq  et(O, O') = O [owise] .
endfm

fmod APPLY is
    including ERROR .
    including INSERT-DELETE .

    vars Oi Oj : Op .
    vars S T : Str .
    var B : Buffer .
    var I : Nat .

    op apply_on_ : Op Buffer -> Buffer .
    eq apply nop on B = B .

    eq apply inv(ins(I,S)) on B = apply del(I,S) on B .
    eq apply inv(del(I,S)) on B = apply ins(I,S) on B .

    eq apply ins(0, S) on B = S B .
    eq apply ins(s I, S) on T B = T (apply ins(I, S) on B) .
    eq apply ins(s I, S) on nil = error .
```

```
    eq apply del(0, S) on T B = B .
    eq apply del(s I, S) on T B = T (apply del(I, S) on B) .
    eq apply del(I, S) on B = error [owise] .

    eq apply Oi o Oj on B = apply Oi on (apply Oj on B) .

    eq apply Oi on error = error .
endfm

fmod MSG is
    including NAT .
    including ABSTRACT-OPERATION .

    sort Msg .

    op msg : Op Nat Nat -> Msg .
endfm

view Msg from TRIV to MSG is
    sort Elt to Msg .
endv

mod SITE is
    including CONFIGURATION .
    including BUFFER .
    including LIST{Msg} .
    including USER .

    sorts Site Queue .
    subsort Site < Configuration  .
    subsort List{Msg} < Queue .

    op <_|_> : User AttributeSet -> Object [ctor object] .
    op buffer :_ : Buffer -> Attribute .
    op seqno :_ : Nat -> Attribute .
    op token :_ : Nat -> Attribute .
    op in-queue :_ : Queue -> Attribute .
    op out-queue :_ : Queue -> Attribute .

    vars T T' N N' : Nat .
    vars Q Q'      : Queue .
    var  O         : Op .

    op compose : Queue -> Op .
    eq compose(nil) = nop .
    eq compose(Q msg(O, T, N)) = compose(Q) o O [owise] .
```

```
    op rejected : Queue Nat -> Queue .
    eq rejected(Q msg(O, T, N) Q', T) = Q .
    eq rejected(Q, T) = Q [owise] .

    op nextSeq : Queue Nat -> Nat .
    eq nextSeq(msg(O, T, N') Q, N) = s N + sd(N, N') .
endm

view Queue from TRIV to SITE is
    sort Elt to Queue .
endv

mod SITE-RULES is
    including ACTIONS .
    including APPLY .
    including UNIQUE .
    including SITE .

    vars B B' B'' : Buffer .
    vars T T' N M : Nat .
    vars U U' : Set{User} .
    var R : Actions .
    var Q : Queue .
    var O : Op .
    var S : Str .
    var A : AttributeSet .

    rl [user-inserts] :
        ((U, U') - R)
        [ S B'' ]
        < U | buffer : (B B'), seqno : N, token : T, out-queue : Q, A >
      =>
        R
        [ B'' ]
        < U | buffer : (B S B'), seqno : s N, token : T,
              out-queue : (msg(ins(size(B), S), T, N) Q), A > .

    rl [user-deletes] :
        ((U, U') - R)
        < U | buffer : (B S B'), seqno : N, token : T, out-queue : Q, A >
      =>
        R
        < U | buffer : (B B'), seqno : s N, token : T,
              out-queue : (msg(del(size(B), S), T, N) Q), A > .

    crl [user-receive] :
        < U | buffer : B, seqno : N, token : T, in-queue : (Q msg(O,T',N)), A >
      =>
```

```
        < U | buffer : B', seqno : s N, token : T', in-queue : Q, A >
      if B' := apply O on B .

    crl [user-receive-reject] :
        < U | buffer : B, seqno : N, in-queue : (Q msg(O,T',M)), A >
      =>
        < U | buffer : B, seqno : s N, in-queue : Q, A >
      if N =/= M .
endm

fmod ORDERING is
    including INSERT-DELETE .
    including SET{User} .

    sort Event .

    op ____ : Op Nat Nat Set{User} -> Event .

    vars T T' M M' : Nat .
    vars E E' : Event .
    vars O O' : Op .
    vars U U' : Set{User} .

    op concurrent : Event Event -> Bool .
    eq concurrent(O T M U, O' T' M' U') =
        intersection(U, U') == empty and
        ((T <= T' and T' <= M) or (T' <= T and T <= M')) .

    op _<_ : Event Event -> Bool .
    ceq O T M U < O' T' M' U' = pos(O) > pos(O')
        or (pos(O) == pos(O') and type(O) =/= type(O') and type(O) == del)
        or (pos(O) == pos(O') and type(O) == type(O') and M < M')
      if concurrent(O T M U, O' T' M' U') .
    eq O T M U < O' T' M' U' = M < M' [owise] .

    op _>_ : Event Event -> Bool .
    eq E > E' = E' < E .
endfm

view Event from TRIV to ORDERING is
    sort Elt to Event .
endv

fmod HISTORY is
    including LIST{Event} .
    including TRANSFORMATION .

    sort History .
```

```
subsort List{Event} < History .

vars M N T M' N' T' : Nat .
vars O O' O'' O''' : Op .
vars H H' H'' : History .
vars U U' : User .
vars E E' : Event .

op  collect : Event History -> History .
eq  collect(E, nil) = nil .
ceq collect(E, E' H) = E' collect(E, H) if concurrent(E, E') .
eq  collect(E, E' H) = nil [owise] .

op  drop : Event History -> History .
eq  drop(E, nil) = nil .
ceq drop(E, E' H) = drop(E, H) if concurrent(E, E') .
eq  drop(E, E' H) = E' H [owise] .

op  put : Event History -> History .
eq  put(E, H) = put'(E, collect(E, H)) drop(E, H) .

op  put' : Event History -> History .
eq  put'(E, nil) = E .
ceq put'(E, H E') = H E' E if E < E' .
eq  put'(E, H E') = put'(E, H) E' [owise] .

op compose : History -> Op .
eq compose(nil) = nop .
eq compose((nop(O) T M U) H) = compose(H) .
eq compose((O T M U) H) = O o compose(H) [owise] .

op  until : History Nat -> History .
eq  until(nil, T') = nil .
ceq until(H (O T M U), T') = H (O T M U) if M >= T' .
eq  until(H (O T M U), T') = until(H, T') [owise] .

op mintoken : History -> Nat .
eq mintoken(O T M U) = T .
eq mintoken((O T M U) H) = min(T, mintoken(H)) [owise] .

op  fix : History -> History .
eq  fix(nil) = nil .
ceq fix((O T M U) H) = (O' T M U) H'
  if H'   := fix(H) /\
     H''  := filter(O T M U, H') /\
     O''' := compose(opsIn(U, T, mintoken(H'')), H')) /\
     O''  := et(O, O''') /\
     O'   := it(it(O'', compose(H'')), O''') .
```

```
    op  filter : Event History -> History .
    eq  filter(E, nil) = nil .
    ceq filter(E, E' H) = E' filter(E, H) if concurrent(E, E') .
    eq  filter(E, (O T M U') H) = filter(E, H) [owise] .

    op  opsIn : User Nat Nat History -> History .
    eq  opsIn(U, T, T', nil) = nil .
    ceq opsIn(U, T, T', (O N M U') H) = (O N M U') opsIn(U', T, T', H)
      if (M < T and M >= T') or (U == U' and T == N) .
    eq  opsIn(U, T, T', E H) = opsIn(U, T, T', H) [owise] .
endfm

mod SERVER is
    including MAP{User,Queue} .
    including HISTORY .

    op < server |_> : AttributeSet -> Object [ctor object] .
    op history :_ : History -> Attribute .
    op state :_ : Nat -> Attribute .
    op sites :_ : Map{User,Queue} -> Attribute .
endm

mod SEND is
    including SERVER .

    sort Send .
    subsort Send < Configuration .

    op send_to_ : Op Map{User,Queue} -> Send .

    vars A A' : AttributeSet .
    vars O O' : Op .
    vars T T' N : Nat .
    vars US US' : Map{User,Queue} .
    vars Q Q' : Queue .
    var U : User .

    eq [send] : send O to empty = none .
    eq [send] :
        < U | in-queue : Q, A >
        < server | state : T, sites : (U |-> msg(O', T', N) Q', US'), A' >
        send O to (U |-> msg(O', T', N) Q', US)
      =
        < U | in-queue : (msg(O, T, N) Q), A >
        < server | state : T, sites : (U |-> msg(O, T, s N) msg(O', T', N) Q', US'
        send O to US .
endm
```

```
mod SERVER-RULES is
    including SEND .

    vars A A' : AttributeSet .
    vars Q Q' Q'' : Queue .
    vars O O' O'' : Op .
    vars T M S S' : Nat .
    var U  : User .
    var E : Event .
    vars H H' : History .
    vars US US' : Map{User,Queue} .

    op makeOp : History History Nat -> Op .
    eq makeOp(H', H, T) = compose(until(H', T)) o inv(compose(until(H, T))) .

    op makeResponse : Op Op Queue Nat -> Op .
    eq makeResponse(O, O', Q, T) = O' o compose(rejected(Q, T)) o inv(O) .

    crl [server-receive] :
        < U | out-queue : (Q msg(O, T, S)), in-queue : Q', A >
        < server | history : H, state : M, sites : (U |-> Q'', US), A' >
      =>
        < U | out-queue : Q,
              in-queue  : (msg(O'', s M, S') Q'), A >
        < server | history : H', state : s M, sites : US', A' >
        (send O' to US)
      if  H'  := fix(put(O T M U, H)) /\
          O'  := makeOp(H', H, T) /\
          S'  := nextSeq(Q'', S) /\
          O'' := makeResponse(O, O', Q'', T) /\
          US' := (U |-> msg(O'', s M, s S'), US) .
endm

mod PREDICATES is
    including APPLY .
    including LTL-SIMPLIFIER .
    including MODEL-CHECKER .
    including SATISFACTION .
    including SERVER .
    including SET{Buffer} .

    subsort Configuration < State .

    ops consistent : -> Prop .

    var A : AttributeSet .
    var B : Buffer .
```

```
    var U : User .
    var C : Configuration .
    var H : History .

    op buffers : Configuration -> Set{Buffer} .
    eq buffers(< U | buffer : B, A > C) = B, buffers(C) .
    eq buffers(C) = empty [owise] .

    eq C |= consistent = | buffers(C) | <= 1 .
    eq C |= consistent = false [owise] .

    op legal : Buffer -> Prop .
    eq < server | history : H, A > C |= legal(B) =
        apply compose(H) on B =/= error .
    eq C |= legal(B) = false [owise] .
endm

mod TEST is
    including ACTIONS .
    including APPLY .
    including BUFFER .
    including CONFIGURATION .
    including ERROR .
    including MSG .
    including ABSTRACT-OPERATION .
    including PREDICATES .
    including SERVER .
    including SERVER-RULES .
    including SITE .
    including SITE-RULES .
    including STR .
    including SET{Buffer} .
    including HISTORY .
    including TRANSFORMATION .
    including UNIQUE .
    including USER .

    var C : Configuration .
    vars T T' N N' I : Nat .
    vars O O' : Op .
    vars Q Q' : Queue .
    var U : User .
    var US : Map{User,Queue} .
    vars A A' : AttributeSet .
    vars B B' B'' : Buffer .
    vars Oi Oj : Op .
    var S : Str .
    vars H H' : History .
```

```
    op init  : -> Configuration .
    eq init = init nil 2 3 .

    op init_ : Buffer -> Configuration .
    eq init B = init B 2 3 .

    op init__ : Buffer Nat -> Configuration .
    eq init B N = init B N 3 .

    op init___ : Buffer Nat Nat -> Configuration .
    eq init B N N' =
        [ a b c d e ]
        (actions 0 N N')
        < server | history : nil, state : 1,
                   sites : sites N >
        clients B N .

    op clients__ : Buffer Nat -> Configuration .
    eq clients B 0 = none .
    eq clients B s N =
        < user N | buffer : B, seqno : 0, token : 0, out-queue : nil, in-queue : nil >
        clients B N .

    op sites_ : Nat -> Map{User,Queue} .
    eq sites 0 = empty .
    eq sites s N = user N |-> msg(nop, 0, 0), sites N .

    op actions___ : Nat Nat Nat -> Actions .
    eq actions N I N = empty .
    eq actions N I N' = users(min(s N, I)) - actions s N I N' [owise] .

    op users : Nat -> Set{User} .
    eq users(0) = empty .
    eq users(s N) = user N, users(N) .
endm
```

Bibliography

[1] R. Antonsen. *Logiske metoder: kunsten å tenke abstrakt og matematisk*. Universitetsforlaget, 2014. ISBN: 9788215022741. URL: https://books.google.no/books?id=MdH5rQEACAAJ.

[2] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. http://www.rfc-editor.org/rfc/rfc7159.txt. RFC Editor, Mar. 2014. URL: http://www.rfc-editor.org/rfc/rfc7159.txt.

[3] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[4] Edmund M. Clarke et al. "Model Checking and the State Explosion Problem." In: *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Vol. 7682. Lecture Notes in Computer Science. Springer, 2011, pp. 1–30. ISBN: 978-3-642-35745-9. DOI: 10.1007/978-3-642-35746-6_1. URL: http://dx.doi.org/10.1007/978-3-642-35746-6_1.

[5] Manuel Clavel et al., eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3.

[6] The Unicode Consortium. *The Unicode Standard, Version 5.0 (5th Edition)*. Addison-Wesley Professional, 2006. ISBN: 0321480910. URL: http://www.amazon.com/Unicode-Standard-Version-5-0-5th/dp/0321480910?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0321480910.

[7] Soren Lassen David Wang Alex Mah. *Google Wave Operational Transformation*. 2010. URL: http://web.archive.org/web/20160309220831/http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html (visited on 03/01/2016).

[8] *Developer Survey Results 2016*. 2016. URL: http://stackoverflow.com/research/developer-survey-2016 (visited on 05/27/2016).

[9] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. "The Maude {LTL} Model Checker." In: *Electronic Notes in Theoretical Computer Science* 71 (2004). {WRLA} 2002, Rewriting Logic and Its ApplicationsFabio Gadducci, Ugo Montanari, pp. 162–187. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/S1571-0661(05)82534-4. URL: http://www.sciencedirect.com/science/article/pii/S1571066105825344.

[10] Clarence A Ellis and Simon J Gibbs. "Concurrency control in groupware systems." In: *Acm Sigmod Record*. Vol. 18. 2. ACM. 1989, pp. 399–407.

[11] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure programming*. Beijing; Sebastopol, Calif.: O'Reilly, 2012. ISBN: 9781449394707 1449394701. URL: http://www.amazon.com/Clojure-Programming-Chas-Emerick/dp/1449394701/ref=sr_1_1?s=books&ie=UTF8&qid=1366804017&sr=1-1&keywords=Clojure.

[12] Douglas C Engelbart. "The mother of all demos." In: (1968).

[13] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. http://www.rfc-editor.org/rfc/rfc6455.txt. RFC Editor, Dec. 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt.

[14] Robert S. Fish, Robert E. Kraut, and Mary D. P. Leland. "Quilt: A Collaborative Tool for Cooperative Writing." In: *Proceedings of the ACM SIGOIS and IEEECS TC-OA 1988 Conference on Office Information Systems*. COCS '88. Palo Alto, California, USA: ACM, 1988, pp. 30–37. ISBN: 0-89791-261-6. DOI: 10.1145/45410.45414. URL: http://doi.acm.org/10.1145/45410.45414.

[15] Michael Fogus and Chris Houser. *The Joy of Clojure: Thinking the Clojure Way*. 1st. Greenwich, CT, USA: Manning Publications Co., 2011. ISBN: 1935182641, 9781935182641.

[16] Neil Fraser. "Differential Synchronization." In: *Proceedings of the 9th ACM Symposium on Document Engineering*. DocEng '09. Munich, Germany: ACM, 2009, pp. 13–20. ISBN: 978-1-60558-575-8. DOI: 10.1145/1600193.1600198. URL: http://doi.acm.org/10.1145/1600193.1600198.

[17] Irene Greif, Robert Seliger, and William E. Weihl. "Atomic Data Abstractions in a Distributed Collaborative Editing System." In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: ACM, 1986, pp. 160–172. DOI: 10.1145/512644.512659. URL: http://doi.acm.org/10.1145/512644.512659.

[18] Abdessamad Imine et al. "ECSCW 2003: Proceedings of the Eighth European Conference on Computer Supported Cooperative Work 14–18 September 2003, Helsinki, Finland." In: ed. by Kari Kuutti et al. Dordrecht: Springer Netherlands, 2003. Chap. Proving Correctness of Transformation Functions in Real-Time Groupware, pp. 277–293.

ISBN: 978-94-010-0068-0. DOI: 10.1007/978-94-010-0068-0_15. URL: http://dx.doi.org/10.1007/978-94-010-0068-0_15.

[19] Abdessamad Imine et al. "Formal design and verification of operational transformation algorithms for copies convergence." In: *Theoretical Computer Science* 351.2 (2006). Algebraic Methodology and Software TechnologyThe 10th International Conference on Algebraic Methodology and Software Technology 2004, pp. 167–183. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2005.09.066. URL: http://www.sciencedirect.com/science/article/pii/S030439750500616X.

[20] Abdessamad Imine et al. "Proving Correctness of Transformation Functions Functions in Real-Time Groupware." In: *Proceedings of the Eighth European Conference on Computer Supported Cooperative Work, 14-18 September 2003, Helsinki, Finland*. Ed. by Kari Kuutti et al. Springer, 2003, pp. 277–293. URL: http://www.ecscw.org/2003/015Imine_ecscw03.pdf.

[21] Van Jacobson, Bob Braden, and Dave Borman. *TCP Extensions for High Performance*. RFC 1323. http://www.rfc-editor.org/rfc/rfc1323.txt. RFC Editor, May 1992. URL: http://www.rfc-editor.org/rfc/rfc1323.txt.

[22] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[23] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[24] Bil Lewis, Daniel LaLiberte, Richard Stallman, et al. *GNU Emacs Lisp Reference Manual*. Free Software Foundation Cambridge, MA, 2014. URL: https://www.gnu.org/software/emacs/manual/elisp.html.

[25] Brian T. Lewis and Jeffrey D. Hodges. "Shared Books: Collaborative Publication Management for an Office Information System." In: *Proceedings of the ACM SIGOIS and IEEECS TC-OA 1988 Conference on Office Information Systems*. COCS '88. Palo Alto, California, USA: ACM, 1988, pp. 197–204. ISBN: 0-89791-261-6. DOI: 10.1145/45410.45431. URL: http://doi.acm.org/10.1145/45410.45431.

[26] Brad Lushman and Gordon V. Cormack. "Proof of correctness of Ressel's adOPTed algorithm." In: *Information Processing Letters* 86.6 (2003), pp. 303–310. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/S0020-0190(03)00227-8. URL: http://www.sciencedirect.com/science/article/pii/S0020019003002278.

[27] David A. Nichols et al. "High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System." In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 111–120. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215706. URL: http://doi.acm.org/10.1145/215585.215706.

[28] Peter Csaba Ölveczky. *Formal Modeling and Analysis of Distributed Systems*. Springer, 2015. ISBN: 978-1-4471-6687-0.

[29] Teresia R Ostrach. "Typing speed: How fast is average: 4,000 typing scores statistically analyzed and interpreted." In: *Orlando, FL: Five Star Staffing* (1997).

[30] J. Postel. *Transmission Control Protocol*. RFC 793 (Standard). Updated by RFCs 1122, 3168. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[31] Emil Protalinski. *Google announces 10% price cut for all Compute Engine instances, Google Drive has passed 240M active users*. 2014. URL: http://thenextweb.com/google/2014/10/01/google-announces-10-price-cut-compute-engine-instances-google-drive-passed-240m-active-users/ (visited on 05/27/2016).

[32] Aurel Randolph et al. "On Consistency of Operational Transformation Approach." In: *Proceedings 14th International Workshop on Verification of Infinite-State Systems, Infinity 2012, Paris, France, 27th August 2012*. Ed. by Mohamed Faouzi Atig and Ahmed Rezine. Vol. 107. EPTCS. 2012, pp. 45–59. DOI: 10.4204/EPTCS.107.5. URL: http://dx.doi.org/10.4204/EPTCS.107.5.

[33] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors." In: *CSCW '96, Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work, Boston, MA, USA, November 16-20, 1996*. Ed. by Mark S. Ackerman, Gary M. Olson, and Judith S. Olson. ACM, 1996, pp. 288–297. ISBN: 0-89791-765-0. DOI: 10.1145/240080.240305. URL: http://doi.acm.org/10.1145/240080.240305.

[34] Richard M Stallman. *Gnu Emacs Manual: For Version 24.5*. Free Software Foundation, 2015.

[35] Maher Suleiman, Michèle Cart, and Jean Ferrié. "Concurrent Operations in a Distributed and Mobile Collaborative Environment." In: *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*. Ed. by Susan Darling Urban and Elisa Bertino. IEEE Computer Society, 1998, pp. 36–45. ISBN: 0-8186-8289-2. DOI: 10.1109/ICDE.1998.655755. URL: http://dx.doi.org/10.1109/ICDE.1998.655755.

[36] Maher Suleiman, Michèle Cart, and Jean Ferrié. "Serialization of concurrent operations in a distributed collaborative environment." In: *Proceedings of GROUP'97, International Conference on Supporting Group Work: The Integration Challenge, November 16-19, 1997, Embassy Suites Hotel, Phoenix, Arizona, USA*. ACM, 1997, pp. 435–445. DOI: 10.1145/266838.267369. URL: http://doi.acm.org/10.1145/266838.267369.

[37] Chengzheng Sun, David Chen, and Xiaohua Jia. "Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems." In: Citeseer.

[38] Chengzheng Sun and Clarence Ellis. "Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements." In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW '98. Seattle, Washington, USA: ACM, 1998, pp. 59–68. ISBN: 1-58113-009-0. DOI: 10.1145/289444.289469. URL: http://doi.acm.org/10.1145/289444.289469.

[39] Chengzheng Sun et al. "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems." In: *ACM Trans. Comput.-Hum. Interact.* 5.1 (1998), pp. 63–108. DOI: 10.1145/274444.274447. URL: http://doi.acm.org/10.1145/274444.274447.

[40] Werner Vogels. "Eventually Consistent." In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432. URL: http://doi.acm.org/10.1145/1435417.1435432.