Transformation-Based Concurrency Control in Groupware Systems

by

Bradley M. Lushman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2002

Acknowledgements

Thanks to my supervisor, Gord Cormack, for his time and assistance, and to my readers, Charlie Clarke and Ken Salem, for taking the time to read my thesis and point out my errors. Thanks also to John Reppy, the inventor of Concurrent ML, for his assistance. Thanks to my parents, John and Pauline Lushman, my sister Jill Lushman, my grandparents, Gerry and Rita Larocque, and all of my family and friends. Research for this thesis was supported by a PGS A scholarship from the Natural Sciences and Engineering Research Council of Canada and by a Graduate Scholarship from the University of Waterloo.

Abstract

This thesis explores the problem of maintaining a consistent shared state in replication-based groupware systems. Whereas more traditional systems might maintain consistency via locking mechanisms, we consider a transformation-based approach that creates the illusion of a common execution history across all sites in the system. In this thesis, we develop a formal treatment of the theory of operation transforms, using techniques based on Ressel's[18] interaction models. We derive important results about the preconditions required for transformation-based algorithms to work, and we show equivalence between two existing transformation algorithms. We then use our results to build a provably correct generic framework for constructing transformation-based systems. We demonstrate the use of our framework by using it to implement a shared text buffer.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction and Motivation

This thesis is concerned with distributed collaboration on shared data. Specifically, we assume that a group of n collaborators (each of whom is located at some *site*) is working together to maintain a shared state. For efficiency, each collaborator has a local copy of the state, upon which he may perform updates. Updates performed locally are then transmitted to the other collaborators, who also apply them.

The canonical example of this scenario is distributed text editing. In this example, the shared state is a text buffer. Each collaborator edits his local copy of the state by issuing calls to the following procedures:

insert(p, s)—inserts the string s into the buffer at position p, $p \geq 1$;

delete(p, l)—deletes l consecutive characters from the buffer starting at position p, $p \geq 1$.

The insertions and deletions are transmitted to the other collaborators, who perform them as well.

1.1.1 Definition of Correctness

Conceptually, there is only one shared state in the system. The local copies are merely snapshots that represent each collaborator's knowledge of the "true" state (which itself is never materialized). Therefore, we would like the local copies to "agree" with each other in some sense. The sense in which the local copies of the shared state must agree with each other will form the basis for our definition of correctness in the system.

We cannot insist that at every instant, the local copies be identical at every site, since at any instant, there may be updates in transit. If an update u is in transit at time t then at time t, a site i may have received and applied u, while a site j has not. Hence, we cannot expect that i's copy of the state will be the same as j's. Further, in a large, active system with several sites, there may be updates in transit at *every* (or almost every) instant, and so it may be the case that no two sites ever have the same local copy of the state.

Ellis and Gibbs[6] give a two-part definition of correctness in groupware systems. A groupware system is considered correct if it satisfies the following two properties:

1) The Precedence Property: if update a causally succeeds update b (in the sense of Lamport's *happens before* relation[13]), then at all sites, a is applied after b;

2) The Convergence Property: all local copies of the state are identical at quiescence,

where *quiescence* is defined as a state in which all updates have been applied and no updates are in transit.

The Precedence Property states that no update is applied until it makes sense to apply it, i.e., all prerequisite updates have been applied. For example if the shared state is a text buffer, initially empty, and site 1 performs insert(1, "abc") followed by delete$(3, 1)$, then all sites must perform the insertion before the deletion; otherwise they may attempt to delete the third character of an empty buffer, which would cause an error.

The Convergence Property does not insist that local copies be identical at all times; instead it only insists that they be identical when the system is quiescent. Quiescence occurs, for example, after all sites have finished performing updates and all transmitted updates have been received and applied. Hence the Convergence Property implies that when the collaborative effort is "finished," each collaborator's local copy of the state will be the same, which is certainly a desirable property.

The problem with the Convergence Property is that it only applies to quiescent systems; it says nothing about the local copies of the state in a non-quiescent system. In particular, any system that never becomes quiescent satisfies the Convergence Property vacuously.

For example, suppose that a network partition has occurred. Such a system can never be quiescent, since updates can never cross the partition. Since the system can never be quiescent, it satisfies the Convergence Property. Hence, individual sites can literally do whatever they want and still satisfy the Convergence Property. Clearly, we need a stronger correctness criterion.

A better definition of convergence would be as follows: convergence holds if whenever sites i and j have performed the same set of updates, then i and j have identical local copies of the state. Before we can use this definition, we must define what we mean by "same":

Definition 1.1 *An* update *u is a triple $(O(u), S(u), T(u))$ that encapsulates an operation $O(u)$ to be performed, the site $S(u)$ at which u originated and a timestamp $T(u)$ that indicates the context in which u executed. Updates u_1 and u_2 are* the same *(or equal) if $O(u_1) = O(u_2)$, $S(u_1) = S(u_2)$, and $T(u_1) = T(u_2)$.*

We will be more precise about the meaning of "timestamp" later.

Definition 1.2 *Let S_1 and S_2 be sets of updates. S_1 and S_2 are* the same *if there is a bijective mapping $\pi : S_1 \rightarrow S_2$ such that for all $u \in S_1$, $\pi(u)$ is the same as u.*

Our modified definition of convergence does not require quiescence as a prerequisite; hence it is more widely applicable than Ellis and Gibbs' definition. However, even this definition is not universally applicable, as it presumes the existence of two sites with exactly the same execution history at a given instant. In a large, active system, this is not a realistic expectation either.

Instead, we define convergence as follows:

Definition 1.3 (Strong Convergence Property) *A groupware system is* convergent *if the local copy of the shared state at each site s is uniquely determined by the initial shared state and the set of updates that have been applied at s (and is not dependent upon s itself).*

Clearly this definition of convergence implies the previous two formulations; moreover, unlike the other formulations, this one can be applied at any instant in time. Hence, we will use the following definition of correctness:

Definition 1.4 *A groupware system is* correct *if it satisifies the Precedence Property and the Strong Convergence Property.*

1.1.2 Enforcing Strong Convergence

As we shall see later, it is easy to ensure (via suitable delay mechanisms) that the Precedence Property will hold. On the other hand, it is easy to conceive scenarios in which the Strong Convergence Property fails. For example, assume that the shared state is a text buffer, initially empty, and that there are two sites, 1 and 2. Site 1 executes insert(1, “a”) and transmits the update to Site 2. At the same time, Site 2 executes insert(1, “b”) and transmits the update to Site 1. As a result, Site 1 executes insert(1, “a”) followed by insert(1, “b”), while Site 2 executes insert(1, “b”) followed by insert(1, “a”). Hence the local copy of the state is “ba” at Site 1 and “ab” at Site 2. Since both sites executed the same set of updates with different results, the Strong Convergence Property fails. (Note that because neither of the two insertions causally preceded the other, the Precedence Property holds in this case.)

A traditional approach to enforcing the Strong Convergence Property would be to construct a locking mechanism in which at most one site may update the shared state at a time[10, 12]. In general, this approach is too strict and will cause unreasonable delays as sites wait for their turn to apply their respective updates. A more practical approach would be to slice the shared state into regions (for example, a shared text buffer representing a book could be split into chapters). Each region could then have its own lock. This scheme could allow more than one concurrent update on the shared state, so long as each site is updating a different region. However, even a finer degree of granularity such as this could lead to unnecessary delays.

Indeed, a locking system of any granularity has the potential to introduce unnecessary delays into the system. The very presence of a distributed lock implies that a site must contact a lock server before it may perform updates. If the network is slow, or down, then the site can do little or no work. Furthermore, distributed locking algorithms are very awkward to implement and prone to failure once in place. If not managed properly, distributed locking can lead to distributed deadlock. Further, if the lock server fails and the network becomes partitioned, sites might elect multiple lock servers and there would be no mutual exclusion at all.

Other concurrency control mechanisms include turn-taking[9], in which control of the document is passed from participant to participant in some (possibly pre-determined) order, and transactions[5], in which sequences of operations are guaranteed to execute as an atomic unit, or not at all. However, there are disadvantages to these mechanisms as well. Turn-taking is simply a restricted form of locking and all of the above-mentioned problems with locking apply to turn-taking. Transactions introduce the potential for forced rollbacks, if the execution history ceases to be serializable. Further, transactions are often implemented using distributed locks, and again, the problems associated with locks can manifest themselves.

A more promising approach to concurrency control, known as *operation transforms*, was first proposed by Ellis and Gibbs[6], and later revised by Cormack[3] and Ressel et al[18]. Instead of locking the shared state, all updates are allowed to proceed immediately (modulo unavoidable delays imposed by the Precedence Property). Central to the success of this approach is the realization that inconsistencies arise because transmitted updates are not always executed in the same context at each site. To correct this problem, updates are transformed before they are applied. The transformed updates have the property that, when applied, they create the illusion that all updates were applied in the intended execution context, and in the intended order. In this way, consistency is preserved.

1.1.3 Goals

The theory of operation transforms is not well studied. Indeed, Cormack[4] has shown that the algorithm of Ellis and Gibbs is incorrect, and furthermore, Hendrie[11] has shown that Cormack's algorithm is also incorrect. Ressel[18] never gave a complete proof of the correctness of his algorithm. This thesis will present a rigourous study of the theory of operation transforms, including a proof of the correctness of Ressel's algorithm, and a proof of equivalence between Ressel's algorithm and a corrected version of Cormack's algorithm. It will pay special attention to the preconditions that must hold in order for the approach to work.

The operation transforms approach to building groupware systems is sufficiently generic that it should be possible to construct a library of code to support the construction of transformation-based objects. Such a library could then be packaged with a programming language distribution, or form the basis for a toolkit for constructing distributed systems. It may also be useful as part of a toolkit for constructing operating systems in general. This thesis will discuss the construction of such a library in the programming language ML.

1.2 Outline

Chapter 2 will contain a summary of the algorithms and notations of Cormack and Ressel. Chapter 3 will contain a study of the theory of operation transforms. Chapter 4 will discuss a framework for constructing transformation-based objects in ML. Chapter 5 will discuss applications of the framework discussed in Chapter 4. Chapter 6 will summarize the results of this thesis and present avenues for future investigation.

Chapter 2

Related Work

In this chapter, we present a survey of work that has been done on operation transforms to date. The approach was first proposed by Ellis and Gibbs[6] in 1989. Their algorithm, called dOPT, was proved incorrect by Cormack[4] in 1995. Cormack presented an algorithm, called CCU[3], and intended as a correction to dOPT. A counterexample to CCU was found by Hendrie[11] in 1997. Independently of Cormack, Ressel et al proposed adOPTed[18] as a correction to dOPT in 1996.

This chapter summarizes the algorithms of Cormack and Ressel, highlighting the common features of the two algorithms, and introducing the notation that we will use for the remainder of this thesis.

2.1 Detecting Conflicting Updates

Both the CCU algorithm and the adOPTed algorithm use timestamps to detect conflicting updates. Timestamps are defined as follows:

Definition 2.1 *A timestamp is a tuple* $t = (x_1, x_2, \ldots, x_n)$ *where each* x_i *represents the number of updates that are known to have executed at site* i*. We denote by* $t[i]$ *the* i*-th component of the timestamp* t*.*

Definition 2.2 *Given a timestamp* $t = (x_1, \ldots, x_n)$*, the* norm of t*, denoted* $|t|$*, is defined by the sum* $x_1 + \cdots + x_n$*.*

Definition 2.3 *Given timestamps t_1 and t_2, we say that $t_1 \subset t_2$ (t_1 is* earlier *than t_2) if $t_1 \neq t_2$ and $t_1[i] \leq t_2[i]$ for each i. We say that $t_1 \subseteq t_2$ if $t_1 \subset t_2$ or $t_1 = t_2$.*

Definition 2.4 *Let u_1 and u_2 be updates. We say that u_1 and u_2 are* concurrent *(and write $u_1||u_2$), if $T(u_1) \not\subseteq T(u_2)$ and $T(u_2) \not\subseteq T(u_1)$.*

Definition 2.5 *Let t_1 and t_2 be timestamps. We define the* supremum *of t_1 and t_2, denoted* $\sup(t_1, t_2)$*, to be the smallest (in the sense of Definition 2.3) timestamp t such that $t_1 \subseteq t$ and $t_2 \subseteq t$. Similarly, we define the* infimum *of t_1 and t_2, denoted* $\inf(t_1, t_2)$ *to be the largest (in the sense of Definition 2.3) timestamp such that $t \subseteq t_1$ and $t \subseteq t_2$.*

It is not hard to see that if $t_1 = (a_1, \ldots, a_n)$ and $t_2 = (b_1, \ldots, b_n)$, then $\sup(t_1, t_2) = (\max(a_1, b_1), \ldots, \max(a_n, b_n))$ and $\inf(t_1, t_2) = (\min(a_1, b_1), \ldots, \min(a_n, b_n))$. Although we can extend Definition 2.5 in the obvious way to any finite set of timestamps, the above formulation will suffice for our purposes.

Definition 2.6 *Let u be an update. We define $T'(u)$ to be the timestamp such that $T'(u)[S(u)] = T(u)[S(u)] + 1$ and $T'(u)[i] = T(u)[i]$ if $i \neq S(u)$.*

The timestamp $T'(u)$ represents the timestamp at site $S(u)$ after u has been applied.

The relation $\subseteq$ establishes a partial order on timestamps. This partial order implements exactly Lamports's *happens-before* relation[13]. We will also define a total order on timestamps that extends $\subseteq$:

Definition 2.7 *Let $t = (t_1, t_2, \ldots, t_n)$ be a timestamp. Let $1 \leq i < j \leq n$. We denote by $t[i:j]$ the vector $(t_i, \ldots, t_j)$. For $1 \leq i \leq n$, we define $t[i:i] \equiv t[i]$. Define* $\dim(t) = n$*.*

Definition 2.8 *Define a relation $<$ on timestamps so that for timestamps t_1, t_2 with* $\dim(t_1) = \dim(t_2) = n$*, $t_1 < t_2$ if $t_1[1] < t_2[1]$ or ($t_1[1] = t_2[1]$ and $t_1[2:n] < t_2[2:n]$).*

Proposition 2.1 *Let t_1 and t_2 be timestamps with $t_1 \subseteq t_2$. Then $t_1 \leq t_2$.*

Proof Easy.

Having completed the definition of timestamps, we can reformulate the Precedence Property as follows:

Definition 2.9 *A groupware system is said to satisfy the Precedence Property if for all updates u_1 and u_2, if $T(u_1) \subset T(u_2)$, then u_1 is applied before u_2 at all sites.*

Definition 2.10 *Let u be an update issued in a groupware system. The* definition context *of u, denoted $DC(u)$*[1]*, is the set of updates that had been applied at site $S(u)$ when u was issued.*

For any groupware system that satisfies the Precedence Property, there is a one-to-one correspondence between the definition context of an update u and $T(u)$.

In the remaining sections we will introduce the CCU algorithm and the adOPTed algorithm.

2.2 The CCU Algorithm

2.2.1 Transformation Operators: /, \, and ˆ

Let X be the set of all possible values of the shared state. Suppose a groupware system supports a set $O \subseteq X^X$ of possible operations on X. If the shared state is a text buffer, then X is the set of all strings (over some alphabet), and O might equal $\{\text{insert}(s,p) | s \text{ a string}, p \in \mathbb{Z}, p \geq 1\}$, the set of all possible string insertions.

Suppose that, during the operation of the groupware system, site i issues an update u_i, while site j concurrently issues an update u_j. Then $u_i || u_j$. The updates u_i and u_j are broadcast to all sites. Since u_i and u_j are concurrent, $u_j \notin DC(u_i)$ and $u_i \notin DC(u_j)$. Site j receives the update u_i, but the set of updates that have been applied at site j does not match $DC(u_i)$ (since u_i had not been applied at site i when u_i was issued).

Instead of applying u_i directly, site j adjusts u_i to create a new update that includes u_j in its definition context. We capture this adjustment in a binary operator, $/ : O \times O \rightarrow O$. Intuitively, for operations o_1 and o_2, o_1/o_2 (read "o_1 after o_2") represents the operation with the semantics o_1 would have if its definition context had included o_2. In other words, o_1/o_2 is the operation that the issuer of o_1 would have transmitted, had he known that o_2 had been applied. See Definition 2.11 for a formal definition. Site j then applies $O(u_i)/O(u_j)$ to its copy of the local state.

Exactly how the / operator transforms operations is up to the application designer. Often, there will be a natural choice.

[1]This notation is due to Sun[21].

Example Suppose X is the set of all strings and O is the set of all string insertions, as above. Then one might define the / operator as follows:

$$\text{insert}(s_1, p_1)/\text{insert}(s_2, p_2) \quad = \quad \begin{cases} \text{insert}(s_1, p_1) & \text{if } p_1 < p_2 \\ \text{insert}(s_1, p_1 + |s_2|) & \text{if } p_1 \geq p_2 \end{cases}$$

Here, a call to insert(s_1, p_1) (issued, say, at site i) has been issued concurrently with a call to insert(s_2, p_2) (issued, say, at site j). When insert(s_1, p_1) is applied at site j, it should be transformed so that the point of insertion is the same as it was at site i. If $p_1 < p_2$, then s_2 was inserted to the right of where s_1 is to be inserted. Hence, the point of insertion is unchanged, and no transformation is needed. If $p_1 > p_2$, then s_2 was inserted to the left of where s_1 is to be inserted. Hence, the intended point of insertion has shifted to the right by an offset equal to the length of s_2, and we maintain the intended semantics of the insertion by transforming it to insert$(s_1, p_1 + |s_2|)$. If $p_1 = p_2$, then the two insertions occurred at the same point, and which comes first is an arbitrary choice. Here, we have chosen to let the previously-inserted string occur first.

At site i, the situation is similar. Update u_j arrives, but its definition context does not contain the update u_i, which has already been applied at site i. Thus, u_j must be adjusted to account for the already-applied u_i at site i. However, we cannot use the / operator. Continuing the above example, consider what would happen if $p_1 = p_2$. Then site j performs insert(p_1, s_2) followed by insert$(p_1 + |s_2|, s_1)$. If site i applies the / operator to the incoming $O(u_j)$, then site i performs insert(p_1, s_1) followed by insert$(p_1 + |s_1|, s_2)$. As a result, s_2 occurs directly after s_1 at site i and directly before s_1 at site j. The local copies of the state at sites i and j are now different and the Strong Convergence Property fails.

Instead, we introduce a second transformation operator, called \. The operators / and \ are defined as follows:

Definition 2.11 *Define operators* $/ : O \times O \to O$ *and* $\backslash : O \times O \to O$. *For operations* o_1 *and* o_2, o_1/o_2 *(read "*o_1 *after* o_2*") and* $o_1\backslash o_2$ *(read "*o_1 *before* o_2*") have the property that for all states* $x \in X$, $(o_1\backslash o_2)(o_2(x)) = (o_2/o_1)(o_1(x))$.

With / defined as above, we define \ for string insertions as follows:

$$\text{insert}(s_1, p_1)\backslash\text{insert}(p_2, s_2) \quad = \quad \begin{cases} \text{insert}(s_1, p_1) & \text{if } p_1 \leq p_2 \\ \text{insert}(s_1, p_1 + |s_2|) & \text{if } p_1 > p_2 \end{cases}$$

Note that this formulation of $\backslash$ only differs from our formulation of $/$ for string insertions when $p_1 = p_2$. In this case, the string s_1 is placed before s_2 instead of after it. Thus, if site j transforms the incoming $O(u_i)$ to $O(u_i)/O(u_j)$ and site i transforms the incoming $O(u_j)$ to $O(u_j)\backslash O(u_i)$, then the resulting state will be identical at sites i and j, and Strong Convergence still holds.

A major question that comes to mind when considering the operators $/$ and $\backslash$ is which of the two operators a given site should use to transform operations. The critical requirement is that if $u_1 || u_2$ then any site that performs u_1 first must transform $O(u_2)$ to $O(u_2)/O(u_1)$ and any site that performs u_2 first must transform $O(u_1)$ to $O(u_1)\backslash O(u_2)$ (or vice versa). In other words, if sites i and j apply u_1 and u_2 in different orders, then they must use different transformation operators.

One way to guarantee that this condition will hold is to use the timestamps of the conflicting updates: if $u_1 || u_2$ with $T(u_1) < T(u_2)$ then any site that applies u_2 first will use $/$ to transform u_1 and any site that applies u_1 first will use $\backslash$ to transform u_2. This behaviour is captured in a new operator, $\hat{}$, that is defined over updates:

Definition 2.12 *Let U be the set of all possible updates. Define a partial operator $\hat{} : U \times U \rightharpoonup U$ such that*

$$u_1 \hat{} u_2 \;=\; \begin{cases} (O(u_1)/O(u_2), S(u_1), T'(u_2)) & \textit{if } T'(u_1) > T'(u_2) \\ (O(u_1)\backslash O(u_2), S(u_1), T'(u_1)) & \textit{if } T'(u_1) < T'(u_2) \end{cases},$$

for all updates u_1, u_2 with $T(u_1) = T(u_2)$ and $S(u_1) \neq S(u_2)$. If $T(u_1) \neq T(u_2)$ or $S(u_1) = S(u_2)$, then $u_1 \hat{} u_2$ is undefined.

The operator $\hat{}$ allows us to characterize the behaviour of transformations as follows: for operations u_1 and u_2 with $T(u_1) = T(u_2)$, a site that performs u_1 first transforms u_2 to $u_2 \hat{} u_1$, and vice versa. However, there is another issue to address: what to do about more complex interaction scenarios. In a real groupware system we cannot expect that there will only be at most two updates in transit at any given time. We must be able to handle interaction scenarios of arbitrary complexity.

We begin by defining update sequences:

Definition 2.13 *Let u_1 and u_2 be updates with $T(u_2) \not\subseteq T(u_1)$. We denote by $u_1; u_2$ the* update sequence *consisting of u_1 followed by u_2. For updates $u_1, \ldots, u_n$ with $T(u_i) \not\subseteq T(u_j)$ for $1 \leq j < i \leq n$, we define the sequence $u_1; \ldots; u_n$ analogously.*

Definition 2.14 *Let $u_1; \ldots ; u_n$ be an update sequence, $x \in X$. The* effect *of $u_1; \ldots ; u_n$* on *x, denoted $(u_1; \ldots ; u_n)(x)$ is the value $O(u_n)(\cdots (O(u_1)(x)) \cdots)$.*

Definition 2.15 *We say that update sequences $u_1; \ldots ; u_n$ and $u'_1; \ldots ; u'_n$ are* equivalent *(and write $u_1; \ldots ; u_n \equiv u'_1; \ldots ; u'_n$) if $(u_1; \ldots ; u_n)(x) = (u'_1; \ldots ; u'_n)(x)$ for all $x \in X$.*

We use the notation U_1, U_2, etc., to denote update sequences.

Next we extend ˆ to work over sequences of updates:

Definition 2.16 *Let $U = u_1; \ldots ; u_n$ be an update sequence. U is called* connected *if $T(u_i) = T'(u_{i-1})$ for all $1 < i \leq n$. $T(u_1)$ is called the* origin *of U, denoted $T(U)$, and $T'(u_n)$ is called the* terminus *of U, denoted $T'(U)$.*

Proposition 2.2 *Let u_1 and u_2 be updates with $T(u_1) = T(u_2)$. Then $u_1; u_2\hat{}u_1$ and $u_2; u_1\hat{}u_2$ are connected.*

Proof Follows immediately from Definition 2.12.

Definition 2.17 *Let λ denote the empty update sequence. Then we define ˆ over update sequences as follows:*

$$U\hat{}\lambda = U \tag{2.1}$$
$$\lambda\hat{}U = \lambda \tag{2.2}$$
$$U_1\hat{}(U_2; U_3) = (U_1\hat{}U_2)\hat{}U_3 \tag{2.3}$$
$$(U_1; U_2)\hat{}U_3 = U_1\hat{}U_3; (U_2\hat{}(U_3\hat{}U_1)). \tag{2.4}$$

The first three components of this definition are straightforward; the fourth is not so intuitive. Rule (2.4) says that the sequence $U_1; U_2$ is correctly transformed to account for the sequence U_3 by first transforming U_1 with respect to U_3 (yielding $U_1\hat{}U_3$) and then transforming U_2. We cannot transform U_2 against U_3 directly because the definition context of updates in U_2 contains the updates in U_1, while the definition context of updates in U_3 does not. Instead, U_3 must be transformed to account for U_1 (yielding $U_3\hat{}U_1$) and then U_2 may be transformed against the result.

The following theorem about ˆ is proved in Cormack[3]:

Theorem 2.1 *Let U_1 and U_2 be update sequences. Then for all $x \in X$,*

$$(U_1; (U_2\hat{}\,U_1))(x) \quad = \quad (U_2; (U_1\hat{}\,U_2))(x).$$

Theorem 2.1 states that the analogue of the defining property of \ (presented in Definition 2.11) holds for sequences.

The extended definition of ˆ, combined with Theorem 2.1, provides a procedure for handling more complicated interaction scenarios: an update u arriving at site i is transformed against the sequence of updates that have been applied at site i, and with which u is concurrent.

2.2.2 Canonical Update Sequences: [] and |

Let W denote a set of updates. The CCU algorithm operates by arranging the elements of W into a sequence and then applying the updates. By the Precedence Property, only sequences that respect the causal order are admissible; however, within the causal order, any sequence of updates is a valid candidate. CCU always chooses a *canonical sequence*, defined as follows:

Definition 2.18 *Let W be a set of updates with $|W| = n$ and having distinct timestamps. The* canonical update sequence *for W, denoted $[W]$, is the sequence $u_1; \ldots; u_n$ with each $u_i \in W$ and $T(u_i) < T(u_{i+1})$ for $1 \leq i \leq n-1$.*

In the CCU algorithm, each site i maintains a set W_i of updates that have been applied at site i. When an update arrives, the algorithm augments W_i with the incoming update and the new canonical sequence $[W_i]$ is computed and applied to the initial state.

The algorithm for computing $[W]$ for a set W of updates is expressed via a binary function |, defined as follows:

Definition 2.19 *Let W be a set of updates, u_0 an update. Denote by $W^{<u_0}$ the set $\{u \in W | T'(u) < T'(u_0)\}$ and by $W^{\subset u_0}$ the set $\{u \in W | T'(u) \subset T'(u_0)\}$.*

Definition 2.20 *Define a binary operator | on sets of updates. Given sets W_1 and W_2 of updates, $W_1|W_2$ computes a sequence of updates, ordered according to $<$ that represents the updates in W_1, adjusted under the assumption that all of the updates in W_2 have already been applied. | is computed as follows:*

$$\begin{aligned} W|W &= \lambda \textit{ for all } W \\ W_1|W_2 &= \begin{cases} (W_1^{<u}|W_2); (u\hat{}\,(W^{<u}|W^{\subset u})) & \textit{if } u \notin W_2 \\ (W_1^{<u}|W_2^{<u})\hat{}\,(u\hat{}\,(W_2^{<u}|W^{\subset u})) & \textit{if } u \in W_2 \end{cases}, \end{aligned}$$

where $W = W_1 \cup W_2$ *and* $u \in W$ *with* $T'(v) \leq T'(u)$ *for all* $v \in W$.

Canonical update sequences are then computed as follows:

$$[W] \quad = \quad W|\{\}.$$

2.2.3 The Algorithm

The CCU algorithm at site s is as follows:

```
Initialization:
    $x_s \leftarrow x_0$              ; initial state
    $W_s \leftarrow \{\}$              ; update history
    $T_s \leftarrow (0, \ldots, 0)$   ; timestamp

Occurrence of local operation o:
    let $T = T_s$ with $s$th component incremented
    transmit $(o, s, T_s)$ to other sites
    $W_s \leftarrow W_s \cup \{(o, s, T)\}$
    $x_s \leftarrow o(x_s)$
    $T_s \leftarrow T$

Receipt of update $u$ from site $r$:
    if $T(u) \nsubseteq T_s$ then
        set $u$ aside and revisit when $T(u) \subseteq T_s$
    else
        $W_s \leftarrow W_s \cup \{(O(u), S(u), T'(u))\}$
        $x_s \leftarrow [W_s](x_0)$
        $T_s \leftarrow \sup(T_s, T'(u))$
```

The CCU algorithm, as presented above, has a serious inefficiency: upon the arrival of each new update u from the network, the entire canonical update sequence is recomputed and then reapplied to the initial state. Hence the time required to compute the new state grows with the number of updates that have been performed. We would prefer an algorithm that would permit us to apply a single transformed update to the current state. To address this problem, Cormack claims the following result about $|$:

Theorem 2.2 (CCU Theorem 2) *For sets W_1 and W_2 of updates,*

$$([W_2];(W_1|W_2))(x) \quad = \quad [W_1 \cup W_2](x)$$

for all $x \in X$.

Armed with Theorem 2.2, we can replace the lines

```
$W_s \leftarrow W_s \cup \{u\}$
$x_s \leftarrow [W_s](x_0)$
```

with

```
$x_s \leftarrow ((W_s \cup \{u\})|W_s)(x_s)$
$W_s \leftarrow W_s \cup \{u\}$.
```

Under this modification, an incoming update u is transformed against those already-applied updates with which it is concurrent, and then applied to x_s. Instead of applying the entire update sequence to the initial state with every incoming update, we simply transform and apply the new update to the current state.

The unmodified algorithm, by recomputing canonical update sequences at every step of the algorithm, guarantees that at every site, updates are always issued in canonical order. The modified algorithm does not have this property; it admits non-canonical update sequences, but ensures that these sequences have the same effect as the canonical sequence, when applied to the local state.

2.2.4 The Hendrie Counterexample

Theorem 2.2 transforms the original CCU algorithm from a backtracking algorithm to one that always moves forward (i.e. previous work is never undone). However, Hendrie[11] showed that Theorem 2.2 is actually false and gave the following example that causes the modified CCU algorithm to fail:

Example (Hendrie) Let $X = \{\text{rock}, \text{paper}, \text{scissors}\}$. Let $O = \{\text{Rock}, \text{Paper}, \text{Scissors}\}$, where $\text{Rock}(x) = \text{rock}$, $\text{Paper}(x) = \text{paper}$, and $\text{Scissors}(x) = \text{scissors}$ for all $x \in X$. Define / so that

$$\text{Rock}/\text{Paper} = \text{Paper}/\text{Rock} = \text{Paper}$$

$$\text{Paper}/\text{Scissors} = \text{Scissors}/\text{Paper} = \text{Scissors}$$

$$\text{Scissors}/\text{Rock} = \text{Rock}/\text{Scissors} = \text{Rock}.$$

With / defined in this way, the following definition of \ is consistent with Definition 2.11:

$$\begin{aligned}
&\text{Rock}\backslash\text{Paper} = \text{Paper}\backslash\text{Rock} = \text{Paper}\\
&\text{Paper}\backslash\text{Scissors} = \text{Scissors}\backslash\text{Paper} = \text{Scissors}\\
&\text{Scissors}\backslash\text{Rock} = \text{Rock}\backslash\text{Scissors} = \text{Rock}.
\end{aligned}$$

For example,

$$\begin{aligned}
&(\text{Paper}; \text{Rock}/\text{Paper})(x) = (\text{Paper}; \text{Paper})(x) = \text{paper}\\
&(\text{Rock}; \text{Paper}\backslash\text{Rock})(x) = (\text{Rock}; \text{Paper})(x) = \text{paper}
\end{aligned}$$

for all $x \in X$. Now suppose that there are three sites and that updates $u_1 = (\text{Scissors}, 1, (0, 0, 0))$, $u_2 = (\text{Paper}, 2, (0, 0, 0))$, and $u_3 = (\text{Rock}, 3, (0, 0, 0))$ are issued concurrently at sites 1, 2, and 3, respectively. The choice of initial state is arbitrary, as the behaviour of the updates in O is not dependent on the value of the state. Timestamps are incremented by CCU when states are processed, so these updates are stored as $u'_1 = (\text{Scissors}, 1, (1, 0, 0))$, $u'_2 = (\text{Paper}, 2, (0, 1, 0))$, and $u'_3 = (\text{Rock}, 3, (0, 0, 1))$. Since $(0, 0, 1) < (0, 1, 0) < (1, 0, 0)$, the canonical order for these updates is $u'_3; u'_2; u'_1$. Thus, for $W = \{u'_1, u'_2, u'_3\}$, we have

$$\begin{aligned}
[W](x) = [\{u'_1, u'_2, u'_3\}](x) &= (u'_3; u'_2\hat{\ }u'_3; (u'_1\hat{\ }u'_3)\hat{\ }(u'_2\hat{\ }u'_3))(x)\\
&= O((u'_1\hat{\ }u'_3)\hat{\ }(u'_2\hat{\ }u'_3))(O(u'_2\hat{\ }u'_3)(O(u'_3)(x)))\\
&= ((\text{Scissors}/\text{Rock})/(\text{Paper}/\text{Rock}))((\text{Paper}/\text{Rock})(\text{Rock}(x)))\\
&= \text{Paper}(\text{Paper}(\text{Rock}(x)))\\
&= \text{paper}.
\end{aligned}$$

According to Theorem 2.2, the following calculation should produce the same result:

$$\begin{aligned}
([\{u'_1, u'_2\}]; (\{u'_3\}|\{u'_1, u'_2\}))(x) &= ((u'_2; u'_1\hat{\ }u'_2); (u'_3\hat{\ }u'_2)\hat{\ }(u'_1\hat{\ }u'_2))(x)\\
&= O((u'_3\hat{\ }u'_2)\hat{\ }(u'_1\hat{\ }u'_2))(O(u'_1\hat{\ }u'_2)(O(u'_2)(x)))\\
&= ((\text{Rock}\backslash\text{Paper})\backslash(\text{Scissors}/\text{Paper}))((\text{Scissors}/\text{Paper})(\text{Paper}(x)))\\
&= \text{Scissors}(\text{Scissors}(\text{Paper}(x)))\\
&= \text{scissors}.
\end{aligned}$$

Since the second calculation results in scissors and not paper, we have a counterexample to Theorem 2.2. This counterexample translates directly into a counterexample to the modified CCU algorithm. The first calculation represents the transformations that take place if the updates arrive at a site in canonical order (i.e. $u_3; u_2; u_1$). The second calculation represents the transformations that take place if the updates arrive at a site in the order $u_2; u_1; u_3$. Hence if one site receives the updates in the order $u_3; u_2; u_1$ and another site receives them in the order $u_2; u_1; u_3$, then the former site's state will be paper and the latter site's state will be scissors. Since the two sites will have applied the same set of updates with different results, the Strong Convergence Property does not hold.

As noted in Hendrie's paper[11], the proof of Theorem 2.2 in Cormack's paper[3] relies on the following observation: if U_1 and U_2 are update sequences with $U_1 \equiv U_2$, then for any update sequence U_3, $U_1\hat{}U_3 \equiv U_2\hat{}U_3$. However, this property does not follow from the definition of $\hat{}$, and in particular it does not hold for Hendrie's counterexample.

The Hendrie counterexample is, of course, somewhat contrived. However, it does suggest that some extra hypotheses are needed before the modified CCU algorithm will work. We will investigate the nature of these hypotheses in Chapter 3.

Note also that the Hendrie counterexample is only a refutation of the modified CCU algorithm. In the original CCU algorithm, all sites recompute the canonical update sequence every time a remote update arrives. Therefore, every site processes updates in canonical order and it is easy to see that Strong Convergence follows.

2.3 The adOPTed Algorithm

The adOPTed algorithm [18] is very similar in spirit to the modified CCU algorithm. Its major features are its transformation function, *tf*, and its preconditions, known as TP1, the Symmetry Property, and TP2.

2.3.1 The Transformation Function, *tf*

Let U denote the set of all possible updates. The transformation function, denoted *tf*, is a partial function from $U \times U$ into $U \times U$, defined for all $(u_1, u_2) \in U \times U$ such that $T(u_1) = T(u_2)$, but $S(u_1) \neq S(u_2)$. For such a pair (u_1, u_2), if $tf(u_1, u_2) = (u_1', u_2')$, then $S(u_1') = S(u_1)$, $S(u_2') = S(u_2)$, $T(u_1') = T'(u_2)$, and $T(u_2') = T'(u_1)$. Hence, $T'(u_1') = T'(u_2')$. The exact formulation of $O(u_1')$ and $O(u_2')$ is left to the application designer, and reflects the designer's desired semantics for transformations.

The transformation function is used as follows: let u_1 and u_2 be updates with $T(u_1) = T(u_2)$ (hence $u_1 || u_2$). Suppose that $S(u_1) = i$ and $S(u_2) = j$ with $i \neq j$, and $tf(u_1, u_2) = (u_1', u_2')$. Then site i executes the sequence $u_1; u_2'$ and site j executes the sequence $u_2; u_1'$.

Example As before, let X, the set of all application states, be the set of all strings (over some alphabet). Let O, the set of all supported operations, be the set $\{\text{insert}(s, p) \mid s \text{ a string}, p \in \mathbb{Z}, p \geq 1\}$ of string insertions. Let $u_1 = (\text{insert}(p_1, s_1), 1, (0, 0))$ and $u_2 = (\text{insert}(p_2, s_2), 2, (0, 0))$. Then $tf(u_1, u_2) = ((\text{insert}(p_1', s_1'), 1, (0, 1)), (\text{insert}(p_2', s_2'), 2, (1, 0))$, where p_i' and s_i' might be defined as follows:

$$\begin{aligned} s_1' &= s_1 \\ s_2' &= s_2 \\ p_1' &= \begin{cases} p_1 & \text{if } p_1 < p_2 \\ p_1 + |s_2| & \text{if } p_1 \geq p_2 \end{cases} \\ p_2' &= \begin{cases} p_1 & \text{if } p_2 \leq p_1 \\ p_1 + |s_2| & \text{if } p_2 > p_1 \end{cases} . \end{aligned}$$

We also define functions that compute the components of tf in isolation:

Definition 2.21 *Define partial functions $tf_1, tf_2 : U \times U \rightharpoonup U \times U$, as follows: if $u_1, u_2 \in U$ with $tf(u_1, u_2) = (u_1', u_2')$, then $tf_1(u_1, u_2) = u_1'$ and $tf_2(u_1, u_2) = u_2'$. If $tf(u_1, u_2)$ is undefined, then so are $tf_1(u_1, u_2)$ and $tf_2(u_1, u_2)$.*

2.3.2 TP1, The Symmetry Property, and TP2

In order to ensure that the system remains consistent under application of tf, the adOPTed algorithm assumes that tf satisfies the following three conditions:

Definition 2.22 (TP1) *tf is said to satisfy TP1 if for all updates u_1 and u_2 for which $tf(u_1, u_2)$ is defined, if $tf(u_1, u_2) = (u_1', u_2')$, then $(u_1; u_2')(x) = (u_2; u_1')(x)$ for all $x \in X$.*

Definition 2.23 (Symmetry Property) *tf is said to satisfy the Symmetry Property if for all updates u_1 and u_2 for which $tf(u_1, u_2)$ is defined, if $tf(u_1, u_2) = (u_1', u_2')$, then $tf(u_2, u_1) = (u_2', u_1')$. (Note that if $tf(u_1, u_2)$ is defined, then it follows immediately from the definition of tf that $tf(u_2, u_1)$ is defined.)*

Definition 2.24 (TP2) *tf is said to satisfy TP2 if for all updates u_1, u_2, and u_3 that are pairwise in the domain of tf, if $tf(u_2, u_3) = (u_2', u_3')$, then $tf_1(tf_1(u_1, u_2), u_3') = tf_1(tf_1(u_1, u_3), u_2')$.*

TP1 is essentially an assertion that the Strong Convergence Property must hold for pairs of updates, much like Definition 2.11. The Symmetry Property simply states that the behaviour of *tf* is not dependent upon the order in which it receives its arguments. TP2 is not so intuitive. We will explore its origin and its implications in detail in Chapter 3.

2.3.3 The Algorithm

The adOPTed algorithm is as follows[2]:

```
Main:
    $X \leftarrow X_0$            ; initial state
    $L \leftarrow \emptyset$
    $Q \leftarrow \emptyset$
    $t \leftarrow (0, \ldots, 0)$      ; initial timestamp
    $s \leftarrow$ local site ID
    while not aborted
        if there is an input $o$
            $u \leftarrow (o, s, t)$
            $Q \leftarrow Q + u$
            $L \leftarrow L + u$
            broadcast $u$ to other sites
        else
            if there is an update $u$ from network
                $Q \leftarrow Q + u$
                $L \leftarrow L + u$
        Execute Update

Execute Update:
    if $\exists u \in Q$ with $T(u) \leq t$ then
```

[2]The adOPTed algorithm actually does some memoization in the the body of Translate Update for increased efficiency. In the interest of clarity and simplicity, the memoization has been removed.

```
        choose one such $u = (o_j, j, t_j)$
        $Q \leftarrow Q - u$
        $u'' \leftarrow$ Translate Update$(u, t)$
        apply operation $O(u'')$ as user $S(u)$
            to state $X$
        increment $S(u)$-th component of $v$

Translate Update $(u, t)$:
    $(o_j, j, t_j) \leftarrow u$
    if $t_j = t$ then return $u$
    else
        let $i$ be such that Reachable?(Decr$(t, i)$)
            and $t_j[i] \leq t[i] - 1$
        $t' \leftarrow$ Decr$(t, i)$
        $u_i \leftarrow$ Update$(i, t[i])$
        $u'_i \leftarrow$ Translate Update$(u_i, t')$
        $u' \leftarrow$ Translate Update$(u, t')$
        $(u'', u''_i) \leftarrow tf(u', u'_i)$
        return $u''$

Reachable?$(t)$:
    for every $i$ in $\{1, \ldots, n\}$:
        $t[i] = 0$ or $T'($Update$(i, t[i])) \leq t$

Decr$(t, i)$:
    copy of $t$ with $i$-th component decremented

Update$(i, j)$:
    $j$-th update from site $i$ in $L$
```

Although the adOPTed algorithm and the modified CCU algorithm are worded differently, they perform essentially the same sequence of steps. Local updates are applied and broadcast to other sites. Remote updates are set aside until all of their prerequisites have been applied, and then transformed against the execution history and applied. The difference between the two algorithms lies in the way that transformations are computed. In Chapter 3, we will show that these two approaches to computing transformations are actually equivalent.

2.4 Other Transformation-Based Systems

In this section we describe some other transformation-based groupware systems. Although we will not be considering these systems in our analysis, integrating them with the theory we develop may provide interesting opportunities for further investigation.

2.4.1 Jupiter

Jupiter[15] is a transformation-based, distributed collaboration system from Xerox PARC. It supports several different types of shared objects, including text documents and a shared whiteboard. The major features of Jupiter are as follows:

- users can share and unshare their documents dynamically;
- users can enter and leave the system at will;
- users can introduce new shared objects to the system.

The principal difference between Jupiter's transformation algorithm and the CCU and adOPTed algorithms is that Jupiter relies on a central server to act as an arbiter when updates conflict. The central server decides which transformations to perform, and since all documents are physically stored on the server, clients get a consistent view of the shared state. CCU and adOPTed do not rely on the existence of any distinguished site; each site is responsible for computing its own state.

2.4.2 REDUCE

The REDUCE system[20] is a descendant of the GROVE[6] system of Ellis and Gibbs. A major feature of REDUCE is that it contains a mechanism to detect when portions of a site's execution history are no longer needed and discard them. REDUCE also plays particular attention to the intended effect of a particular update. While adOPTed only strives to ensure that all sites eventually converge to a common state, REDUCE requires that the common final state be, in some sense, what the collaborators "intended" it to be (note that the CCU algorithm also makes such considerations, although we do not discuss them here). These considerations of intention constrain the ways in which we may transform updates to transformations that "make sense."

The transformation algorithm of REDUCE differs from adOPTed and CCU in that when an update arrives out of order, updates in the history log that should succeed the incoming update are undone. Then the incoming update is transformed (if necessary) and applied. Finally the undone updates are transformed (if necessary) and reapplied. The adOPTed and modified CCU algorithms do not undo previous work. The original CCU algorithm does recompute and reapply update sequences, but it does not explicitly undo any update.

2.4.3 GOT

The GOT[19] algorithm is similar in appearance to the CCU algorithm. Like the / and \ operators of CCU, GOT introduces two dual transformation operators, IT and ET. However, whereas / and \ both augment the definition context of an update with respect to particular ordering assumptions, IT and ET augment and reduce the definition context, respectively. In particular, IT transforms an update to account for additional updates that have been applied at a site, while ET transforms an update to account for updates that were expected, but have not yet been applied. In a sense, IT and ET are inverse operators.

GOT is unable to resolve all conflicts among updates using IT and ET alone. Thus, on occasion, GOT, like REDUCE, must resort to undoing and redoing previously applied updates.

2.4.4 GOTO

The GOTO algorithm[21] of Sun and Ellis is a descendant of GOT and adOPTed. Sun and Ellis claim that by assuming Ressel's TP2 as an additional precondition, GOT can be transformed into a strictly forward-moving algorithm (i.e. no undoing of previously applied updates). As a result, among the alternative transformation-based systems presented here, GOTO is the most similar to CCU and adOPTed. Although an attempt to unify the theory underlying GOTO with the theory behind CCU and adOPTed would be a particularly interesting topic for investigation, we will not make any such attempt here.

Chapter 3

Theory of Operation Transforms

In this chapter, we establish some theoretical results about operation transforms. In particular, we will show equivalence between the CCU and adOPTed algorithms, and explore the nature of TP2.

3.1 Interaction Models

The concept of an interaction model was first defined by Ressel[18] and provides a convenient setting in which to reason about operation transforms. Our definition of an interaction model will be as follows:

Definition 3.1 *Let W be a set of updates in a groupware system consisting of n sites, and x_0 an initial state. An* interaction model *for W is a directed, edge-labelled, vertex-labelled graph $G = (E, V)$ in n-dimensional space. Vertices represent timestamps and are labelled with sets of application states. Edges are labelled with sets of updates (either original or transformed). Given a vertex v, $L(v)$ denotes the associated label. Similarly, $L(e)$ denotes the label of an edge e.*

The edges and vertices, together with their labels, that comprise the interaction model are given by the following rules:

$$v_0 = (0, \ldots, 0) \in V,\ L(v_0) = \{X_0\}; \tag{3.1}$$

$$\text{if } u \in W,\ T(u) \in V,\ \text{and } X \in L(T(u)), \text{ then } T'(u) \in V \text{ and } (T(u), T'(u)) \in E,\ \text{with} \tag{3.2}$$

$u \in L(T(u), T'(u))$;

if $v \in V,\ X \in L(v),\ (v, w) \in E,$ and $u \in L(v, w),$ then $O(u)(X) \in L(w)$; (3.3)

if $(v, w_1), (v, w_2) \in E,\ u_1 \in L(v, w_1),$ and $u_2 \in L(v, w_2),$ then $(w_1, w), (w_2, w) \in E,$ (3.4)

with $tf_2(u_1, u_2) \in L(w_1, w)$ and $tf_1(u_1, u_2) \in L(w_2, w)$, where $w = \sup(w_1, w_2)$.

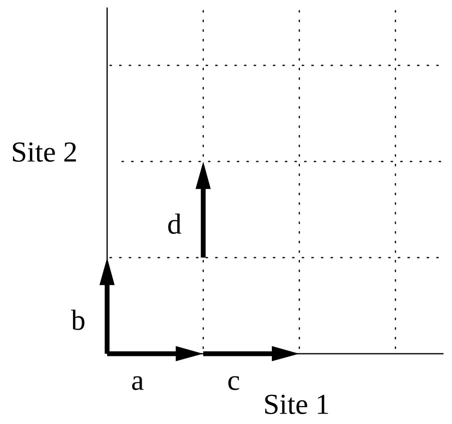
Note that there is a one-to-one correspondence between paths in the interaction model from initial state to current state and total orderings of the updates in W consistent with the causal order $\subseteq$.

Definition 3.2 *Let W be a set of updates and $G = (V, E)$ an interaction model. G* faithfully represents *W if for all $u \in W$, $(T(u), T'(u)) \in E$.*

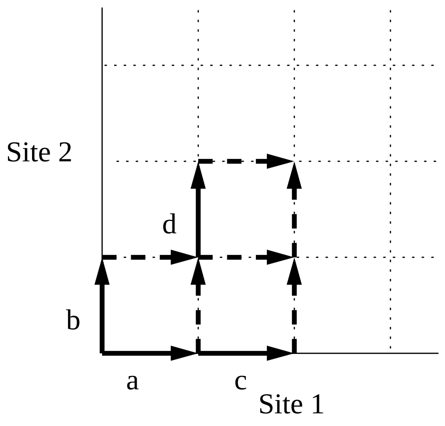
Definition 3.3 *Let W be a set of updates. W is said to satisfy the Precedence Property if for all $u \in W$, and for all timestamps T with $T < T(u)$, there exists an update $u' \in W$ with $T(u') = T$.*

Proposition 3.1 *Let M be a set of updates and G an interaction model for M. If M satisfies the Precedence Property, then G faithfully represents M.*

Example Assume there are two sites in the system, and let $W = \{u_1 = (a, 1, (0, 0)), u_2 = (b, 2, (0, 0)), u_3 = (c, 1, (1, 0)), u_4 = (d, 2, (1, 1))\}$. We can represent each update $u \in W$ as a unit vector in the plane from $T(u)$ to $T'(u)$. Then W can be represented graphically as follows:



The interaction model G for W is the following graph in the plane:



Notice that W satisfies the Precedence Property and that G faithfully represents W, since there is an edge in G for each update in W.

Proposition 3.2 *Let G be an interaction model containing an edge e. Let $u, v \in L(e)$. Then $T(u) = T(v)$ and $S(u) = S(v)$ (hence also $T'(u) = T'(v)$).*

Proof The proof is by induction on the structure of G, and is straightforward.

Because of Proposition 3.2 we may associate timestamps and sites with edges: $T(e)$ is $T(u)$ for $u \in L(e)$, and similarly $S(e)$ is $S(u)$ for $u \in L(e)$ (by a simple inductive argument, we can show that no edge has an empty label).

Interaction models are always connected digraphs with a single source at timestamp $(0, \ldots, 0)$ and a single sink at the largest timestamp (in the sense of $\subseteq$) in the model. They provide a powerful tool for reasoning about the behaviour of the adOPTed algorithm (and, as we shall see, the CCU algorithm as well). We now explore how the Strong Convergence Property may be phrased in terms of interaction models.

Definition 3.4 *Let W be a set of updates and $G = (V, E)$ an interaction model for W. G is said to be* edge-convergent *if for every edge $e \in E$, $|L(e)| = 1$.*

Definition 3.5 *Let W be a set of updates and $G = (V, E)$ an interaction model for W. G is said to be* vertex-convergent *if for every vertex $v \in V$, $|L(v)| = 1$.*

Recall that the Strong Convergence Property requires that, given an initial state, the local state at any site be completely determined by the set of updates that have been applied at that site. By the Precedence Property, a site's current notion of the system's timestamp uniquely determines the set of updates that have been applied and vice versa. Furthermore, for a given vertex (timestamp) in the interaction model, its label represents the set of possible states a site with that timestamp may have. Since the Strong Convergence Property requires that the timestamp uniquely determine the local state, we see that the Strong Convergence Property is equivalent to vertex-convergence of the interaction model.

3.2 Equivalence of CCU and adOPTed

In this section, we prove equivalence between the transformations performed by the CCU algorithm and those performed by the adOPTed algorithm. We begin by showing the relationship between $\hat{}$ and *tf*, and then show that the behaviour of the CCU algorithm can be predicted by interaction models.

3.2.1 $\hat{}$ and *tf*

There are technical difficulties involved in attempting to prove a relationship between $\hat{}$ and *tf*. The major issue is that the definitions of $\hat{}$ and *tf* are not self-contained; they are dependent upon transformation rules imposed externally by the application designer. Therefore, in order to prove anything, we need to assume that $\hat{}$ and *tf* have been defined based on transformations with the same semantics. We can formalize this idea as follows:

Definition 3.6 *Let t_1 and t_2 be binary transformation operators on pairs of updates (where $t_i(a, b)$ is read "a transformed by t_i with respect to b"). t_1 and t_2 are said to be defined based on the same semantics if, for all updates a and b with $a||b$, we have $a; t_1(b, a) \equiv a; t_2(b, a)$ and $b; t_1(a, b) \equiv b; t_2(a, b)$.*

Intuitively, two transformation operators are based on the same semantics if, given the same execution history and the same incoming update, the two operators transform the incoming update in the same way. This is the notion of "based on the same semantics" that we shall use in the discussion that follows.

Let $u_1 = (o_1, s_1, t)$ and $u_2 = (o_2, s_2, t)$ be updates. By TP1,

$$u_1; tf_2(u_1, u_2) \equiv u_2; tf_1(u_1, u_2).$$

By contrast, definition 2.11 gives us

$$\begin{aligned} o_2/o_1(o_1(x)) &= o_1\backslash o_2(o_2(x)) \quad \text{for all } x \in X, \text{ if } T'(u_1) < T'(u_2) \ ; \\ o_2\backslash o_1(o_1(x)) &= o_1/o_2(o_2(x)) \quad \text{for all } x \in X, \text{ if } T'(u_1) > T'(u_2) \ , \end{aligned}$$

and so

$$\begin{aligned} u_1; (o_2/o_1, s_2, T'(u_1)) &\equiv u_2; (o_1\backslash o_2, s_1, T'(u_2)) \quad \text{if } T'(u_1) < T'(u_2); \\ u_1; (o_2\backslash o_1, s_2, T'(u_1)) &\equiv u_2; (o_1/o_2, s_1, T'(u_2)) \quad \text{if } T'(u_1) > T'(u_2). \end{aligned}$$

If tf and ˆ are to have the same semantics, then we need

$$\begin{aligned} tf_1(u_1, u_2) &= \begin{cases} (o_1\backslash o_2, s_1, T'(u_2)) & \text{if } T'(u_1) < T'(u_2) \\ (o_1/o_2, s_1, T'(u_2)) & \text{if } T'(u_1) > T'(u_2) \end{cases} \\ tf_2(u_1, u_2) &= \begin{cases} (o_2/o_1, s_2, T'(u_1)) & \text{if } T'(u_1) < T'(u_2) \\ (o_2\backslash o_1, s_2, T'(u_1)) & \text{if } T'(u_1) > T'(u_2) \end{cases} , \end{aligned}$$

i.e.

$$tf(u_1, u_2) = \begin{cases} ((o_1\backslash o_2, s_1, T'(u_2)), (o_2/o_1, s_2, T'(u_1))) & \text{if } T'(u_1) < T'(u_2) \\ ((o_1/o_2, s_1, T'(u_2)), (o_2\backslash o_1, s_2, T'(u_1))) & \text{if } T'(u_1) < T'(u_2) \end{cases} .$$

According to Definition 2.12,

$$u_1\hat{\ }u_2 = \begin{cases} (o_1/o_2, s_1, T'(u_2)) & \text{if } T'(u_1) > T'(u_2) \\ (o_1\backslash o_2, s_1, T'(u_2)) & \text{if } T'(u_1) < T'(u_2) \end{cases} .$$

By comparison, we conclude that

$$tf(u_1, u_2) = (u_1\hat{\ }u_2, u_2\hat{\ }u_1),$$

and we have proved the following result:

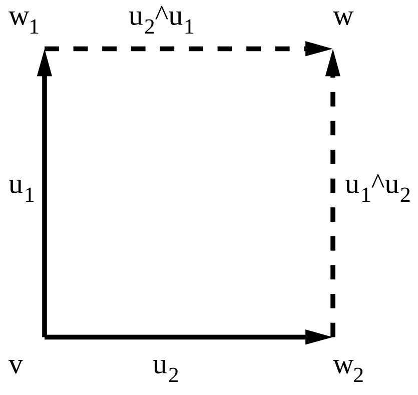


Figure 3.1: Modified interaction model

Theorem 3.1 *Let u_1 and u_2 be updates. Assume that ˆ and tf are defined based on the same transformation semantics. Then $tf(u_1, u_2) = (u_1\hat{\ }u_2, u_2\hat{\ }u_1)$. Equivalently, $u_1\hat{\ }u_2 = tf_1(u_1, u_2)$.*

With an equivalence between ˆ and tf_1 established, we can reformulate (3.4) as follows:

$$\text{if } (v, w_1), (v, w_2) \in E,\ u_1 \in L(v, w_1),\ \text{and } u_2 \in L(v, w_2),\ \text{then } (w_1, w), (w_2, w) \in E, \quad (3.5)$$
$$\text{with } u_2\hat{\ }u_1 \in L(w_1, w) \text{ and } u_1\hat{\ }u_2 \in L(w_2, w), \text{ where } w = \sup(w_1, w_2).$$

The reformulated rule is illustrated in Figure 3.1.

We now consider the extended definition of ˆ for sequences. We show that its behaviour can be predicted by an interaction model.

Definition 3.7 *Let $U = u_1; \ldots; u_n$ be an update sequence corresponding to a path $e_1, \ldots, e_n$ (each e_i is an edge (v_i, w_i)) in an interaction model G, and $t \in \mathbb{Z}^n$ a vector. The* translation of U by t *is the sequence $e'_1, \ldots, e'_n$ in G, where the edge e'_i is the edge $(v_i + t, w_i + t)$.*

Lemma 3.1 *Let u be an update and U a connected update sequence with origin equal to $T(u)$. Let t_1 and t_2 represent the origin and terminus of U, respectively. Then, viewing u as a singleton update sequence, we have $u\hat{\ }U$ is the vector in the interaction model obtained by translating u by $t_2 - t_1$.*

Proof The proof is by induction on the length of U. If U has length 0, then $U = \lambda$ and $t_2 - t_1 = 0$. Then we have $u\hat{\ }U = u$ by definition, which is the result of translating u by 0 in the interaction model. If U has length 1, then U is a singleton update u_1, and the result follows from Proposition 2.2. If U has length $n > 1$, then we can decompose U as $U_1; U_2$ where U_1 and U_2 have length less than n. By Definition 2.12, $u\hat{}(U_1; U_2) = (u\hat{}U_1)\hat{}U_2$. Let t_3 denote the terminus of U_1. Since U is connected, t_3 is also the origin of U_2. By induction, $u\hat{}U_1$ is the vector in the interaction model obtained by translating u by $t_3 - t_1$. Also by induction, $(u\hat{}U_1)\hat{}U_2$ is the vector in the interaction model obtained by translating $u\hat{}U_1$ by $t_2 - t_3$. But this is just u translated by $(t_3 - t_1) + (t_2 - t_3) = t_2 - t_1$. Thus, $u\hat{}U$ is the vector obtained by translating u by $t_2 - t_1$ and the result now follows by induction.

Theorem 3.2 *Let U_1 and U_2 be connected update sequences with $T(U_1) = T(U_2)$. Let $t_1 = T(U_1)$ and $t_2 = T'(U_1)$. Then $U_2\hat{}U_1$ is the path in the interaction model obtained by translating U_2 by $t_2 - t_1$.*

Proof The proof is by induction on the total length of U_1 and U_2, which we call n. For $n = 0$ or $n = 1$, the result is trivial, as at least one of U_1 and U_2 must be empty. For $n = 2$, if neither of U_1 and U_2 is empty, then U_1 and U_2 are both singleton updates and the result follows from Proposition 2.2. Assume $n > 2$. If U_2 is empty, then the the result is trivial, as $U_2\hat{}U_1 = \lambda$. If U_2 has length 1, then the result follows by Lemma 3.1. So assume that U_2 has length greater than 1. Then we can decompose U_2 as $U_3; U_4$, where U_3 and U_4 each have length at least 1. By Definition 2.12, $(U_3; U_4)\hat{}U_1 = (U_3\hat{}U_1); (U_4\hat{}(U_1\hat{}U_3))$. Now, U_3 and U_1 have total length less than n, and so by induction, $U_3\hat{}U_1$ is the path obtained by translating U_3 by the vector $t_2 - t_1$. Let $t'_1 (= t_1)$ and t'_2 denote the origin and terminus of U_3, respectively. By induction, $U_1\hat{}U_3$ is the path obtained by translating U_1 by the vector $t'_2 - t'_1$. Since U_2 is connected, U_4 has origin t'_2. Hence, U_4 and $U_1\hat{}U_3$ have the same origin. Since $U_1\hat{}U_3$ is just a translation of U_1, $U_1\hat{}U_3$ is a connected sequence with the same length as U_1. Thus, U_4 and $U_1\hat{}U_3$ have total length less than n, and so by induction, $U_4\hat{}(U_1\hat{}U_3)$ is a translation of U_4 by $t_2 - t_1$. Hence, $(U_3; U_4)\hat{}U_1$ is a translation of U_3 by $t_2 - t_1$ followed by a translation of U_4 by $t_2 - t_1$, but this is nothing but a translation of U_2 by $t_2 - t_1$. The result now follows by induction.

3.2.2 $|/[]$ and Interaction Models

We now consider how the behaviour of the operators $|$ and $[]$ can be expressed via interaction models. We first need to distinguish between two types of updates:

Definition 3.8 *Let u be an update in an interaction model G. u is called an* original *update if u was issued by a site (that is, if u is in G because of (3.2)). Otherwise, u was generated by a transformation of some other update (that is, u is in G because of (3.5)), and u is called* transformed.

Definition 3.9 *Let u be an update in an interaction model G. We denote by $or(u)$ the original update that was transformed (perhaps via several applications of (3.5)) to produce u. If u is an original update, then we define $or(u) = u$. Otherwise, $u = v\hat{}w$ for some updates v and w, and $or(u) = or(v)$.*

Proposition 3.3 *Let u and v be updates. Let $w = u\hat{}v$. Then $T(w)[S(u)] = T(u)[S(u)]$ and $T'(w)[S(u)] = T'(u)[S(u)]$ (recall that $S(w) = S(u)$).*

Proof Obvious.

Proposition 3.4 *Let u and v be updates. Then $T(u) \subset T(u\hat{}v)$.*

Proof Obvious.

Proposition 3.5 *Let u be an update in an interaction model G. u is an original update if and only if for every update v in G with $T(v)[S(u)] = T(u)[S(u)]$ and $T'(v)[S(u)] = T'(u)[S(u)]$, we have $T(u) \subseteq T(v)$.*

Proof Suppose u is an original update and let $v \neq u$ be such that $T(v)[S(u)] = T(u)[S(u)]$ and $T'(v)[S(u)] = T'(u)[S(u)]$. Note that $S(v)$ must equal $S(u)$. If v is an original update, then v is the $T(u)[S(u)]$-th update from site $S(u)$. But this is impossible, since u is the $T(u)[S(u)]$-th update from site $S(u)$. So $v = v_1\hat{}v_2$ for some updates v_1 and v_2. Then by Proposition 3.3, $T(v_1)[S(u)] = T(u)[S(u)]$ and $T'(v_1)[S(u)] = T'(u)[S(u)]$ and by Proposition 3.4, $T(v_1) \subset T(v)$. The same argument applies to v_1 as to v. As timestamps are bounded below, this sequence of arguments must terminate with an update v_0 that is not the result of an application of $\hat{}$, i.e., v_0 is an original update. By previous arguments, $T(v_0)[S(u)] = T(u)[S(u)]$, $T'(v_0)[S(u)] = T'(u)[S(u)]$, and $T(v_0) \subset T(v)$. Then v_0 is the $T(u)[S(u)]$-th update from site $S(u)$, i.e., $v_0 = u$. Thus, $T(u) \subseteq T(v)$. Conversely, suppose that u is an update in G such that for every update v in G with $T(v)[S(u)] = T(u)[S(u)]$ and $T'(v)[S(u)] = T'(u)[S(u)]$, we have $T(u) \subseteq T(v)$. If u is a transformed update, then $u = u_1\hat{}u_2$ for some updates u_1 and u_2. Then $T(u_1)[S(u)] = T(u)[S(u)]$, $T'(u_1)[S(u)] = T'(u)[S(u)]$ (Proposition 3.3), and $T(u_1) \subset T(u)$ (Proposition 3.4). But this is a contradiction, and so u cannot be transformed. Hence u is original, and the result follows.

Proposition 3.6 *Let u be an update in an interaction model G. Then $or(u)$ is the update v in G with $T(v)[S(u)] = T(u)[S(u)]$, $T'(v)[S(u)] = T'(u)[S(u)]$, and $T(v)$ minimal (in the sense of $\subseteq$).*

Proof By Proposition 3.3 and straightforward induction, we get $T(or(u))[S(u)] = T(u)[S(u)]$ and $T'(or(u))[S(u)] = T'(u)[S(u)]$. Minimality comes from Proposition 3.5 and the fact that $or(u)$ must be original. Uniqueness comes from the fact that the label set of any edge introduced in (3.2) is a singleton set.

Propositions 3.5 and 3.6 show that the concepts of original and transformed updates, and the operator *or* are well-defined. We may now use them freely in the demonstrations that follow.

Definition 3.10 *Let v_1 and v_2 be vertices in an interaction model G with $v_1 \subseteq v_2$. The* canonical path *from v_1 to v_2 is the path from v_1 to v_2 in G that corresponds to the canonical ordering of the updates in $DC(v_2) \setminus DC(v_1)$.*

Recall that there is a one-to-one correspondence between paths in the interaction model and total orders consistent with the partial order. Hence, canonical paths are well-defined.

The choice of which path is canonical is dependent upon our choice of total ordering of events. For the total order we chose in Definition 2.8, the canonical path is determined as follows: at each vertex along the way, if there is more than one possible next step, then the step along the axis corresponding to the largest site id is chosen. A canonical path is illustrated in Figure 3.2.

We shall also need the following lemma:

Lemma 3.2 *Let W_1 and W_2 be sets of updates such that W_2 satisfies the Precedence Property and $W_1 \subseteq W_2$. Then $W_1|W_2 = \lambda$.*

Proof The proof is by induction on $n = |W_2|$. If $n = 0$, then $W_1 = W_2 = \lambda$ and $W_1|W_2 = \lambda$ by definition. For $n > 0$, let u be the update in W_2 for which $T'(u)$ is maximal. Since $W_1 \subseteq W_2$, we have $u \in W_2$. Hence, by Definition 2.20, $W_1|W_2 = (W_1^{<u}|W_2^{<u})\hat{}(u\hat{}(W_2^{<u}|W^{\subset u}))$. Since $T'(u)$ is maximal, we have $W_1^{<u} = W_1 \setminus \{u\}$ and $W_2^{<u} = W_2 \setminus \{u\}$. Hence, $W_1^{<u} \subseteq W_2^{<u}$ and $|W_2^{<u}| = n - 1$. So by induction, $W_1^{<u}|W_2^{<u} = \lambda$. Hence, $W_1|W_2 = \lambda\hat{}(u\hat{}(W_2^{<u}|W^{\subset u})) = \lambda$, and the result now follows by induction.

With these definitions and results in hand, we now have the following relationship between $|$ and interaction models:

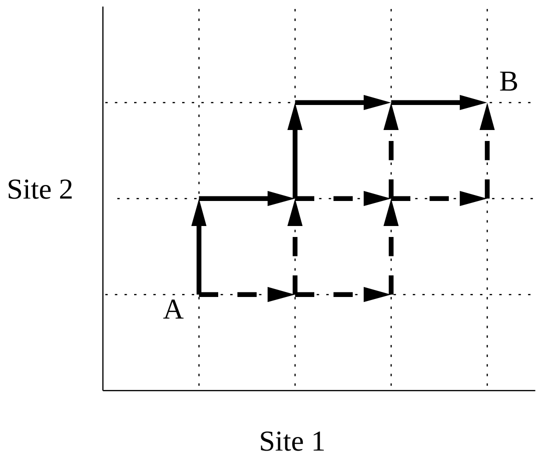


Figure 3.2: A Canonical Path. The solid arrows indicate the canonical path from A to B.

Theorem 3.3 *Let W_1 and W_2 be sets of updates such that both W_2 and $W_1 \cup W_2$ satisfy the Precedence Property. Let W_2 have interaction model G_1 with sink vertex v_1. Let $W_1 \cup W_2$ have interaction model G_2 with sink vertex v_2. Then $W_1|W_2$ is the canonical path in G_2 from v_1 to v_2.*

Proof Let $W = W_1 \cup W_2$. The proof is by induction on $n = |W|$. We have $n \geq |W_2|$. For $n = |W_2|$, we have $W_1 \subseteq W_2$, and so by Lemma 3.2, $W_1|W_2 = \lambda$. Since $W_2 = W_1 \cup W_2$, the corresponding interaction models for these two sets have the same sink vertex, and so the canonical path between the two sinks is λ and the result follows. Assume $n > |W_2|$. Let $u \in W$ with $T'(u)$ maximal. There are two cases to consider:
Case 1: $u \notin W_2$. Then $W_1|W_2 = (W_1^{<u}|W_2); (u\hat{}(W^{<u}|W^{\subset u}))$. Since $T'(u)$ is maximal, $u \notin W_1^{<u} \cup W_2$, and so $|W_1^{<u} \cup W_2| < n$. Also, W_2 and $W_1^{<u} \cup W_2$ satisfy the Precedence Property. Therefore, by induction, $W_1^{<u}|W_2$ is the canonical path from v_1 to v_2', where v_2' is the sink vertex of the interaction model G_2' corresponding to $W_1^{<u}|W_2$. Also, $W^{\subset u} \subseteq W^{<u} \subset W$, and both $W^{<u}$ and $W^{\subset u}$ satisfy the Precedence Property. Therefore, by induction, $W^{<u}|W^{\subset u}$ is an update sequence U whose corresponding path in the interaction model is the canonical path from the sink of $W^{\subset u}$ to the sink of $W^{<u}$. Since $W^{\subset u}$ is the set of causal prerequisites of u, we have $DC(u) = W^{\subset u}$, and so $T(u) = T(U)$. By Theorem 3.2, $T(u\hat{}(W^{<u}|W^{\subset u})) = T'(U)$, and $T'(U)$ is the sink of $W^{<u}$. Further, $W^{<u} = W_1^{<u} \cup W_2$ (since $u \notin W_2$). Hence, $W_1|W_2 = (W_1^{<u}|W_2); (u\hat{}(W^{<u}|W^{\subset u}))$ is a connected sequence, hence a connected path, and it remains to show that this path is canonical.

By induction, we have that $W_1^{<u}|W_2$ is canonical in the interaction model for $W^{<u}$. If $W_1^{<u}|W_2$ is still canonical in the interaction model for W, then it follows immediately that $W_1|W_2$ is canonical. So suppose that $W_1^{<u}|W_2$ is not canonical in the interaction model for W. Then there exists an edge (v, w_1) in $W_1^{<u}|W_2$ that would not be chosen in a canonical path from v_1 to v_2. Instead, an edge (v, w_2) would have been chosen. Since $W_1^{<u}|W_2$ is canonical in $W^{<u}$, (v, w_2) is not in the interaction model for $W^{<u}$. Let $u' \in L(v, w_2)$. If $or(u')$ is not u, then (v, w_2) would have been in the interaction model for $W^{<u}$. Therefore, $or(u') = u$, and so $S(u') = S(u)$. Furthermore, by Proposition 3.3, $w_2 = T'(u)[S(u)]$. At the point where an update in $L(v, w_2)$ has been applied, the timestamp of the system is w_2. Note that $w_2 \not\subseteq w_1$ and $w_1 \not\subseteq w_2$. Hence, by (3.5), $\sup(w_1, w_2)$ is a vertex in G_2, and $w_2 \subset \sup(w_1, w_2)$. Hence, an update whose original update was u was not applied last, and this contradicts the definition of $|$. Thus, there is no such edge (v, w_2), and so $W_1^{<u}|W_2$ is canonical in W. The result now follows by induction.
Case 2: $u \in W_2$. Then $W_1|W_2 = (W_1^{<u}|W_2^{<u})\hat{}(u\hat{}(W_2^{<u}|W^{\subset u}))$. Since $T(u)$ is maximal, $u \notin W_1^{<u} \cup W_2^{<u}$, and so $|W_1^{<u} \cup W_2^{<u}| < n$. Also, $W_1^{<u}$ and $W_2^{<u}$ satisfy the Precedence Property. Therefore, by induction, $W_1^{<u}|W_2^{<u}$ is the canonical path from the sink of $W_2^{<u}$ to the sink of $W_1^{<u}$. Also, $W_2^{<u} \cup W^{\subset u} \subset W$, and both $W_2^{<u}$ and $W^{\subset u}$ satisfy the Precedence Property. Therefore, by induction, $W_2^{<u}|W^{\subset u}$ is a sequence U whose corresponding path in the interaction model is the canonical path from the sink of $W^{\subset u}$ to the sink of $W_2^{<u}$. Since $W^{\subset u}$ is the set of causal prerequisites of u, we have $DC(u) = W^{\subset u}$, and so $T(u) = T(U)$. By Theorem 3.2, $T(u\hat{}(W_2^{<u}|W^{\subset u})) = T'(U)$, and $T'(U)$ is the sink of $W_2^{<u}$. Hence, by Theorem 3.2, $W_1|W_2$ is a translation of the sequence $(W_1^{<u}|W_2^{<u})$ by the vector $T'(u\hat{}U) - T(u\hat{}U)$. By induction, $(W_1^{<u}|W_2^{<u})$ is a canonical path in $W^{<u}$, and $W_1|W_2$ is just a translation of this sequence, and hence is still canonical in W. The result now follows by induction.

Corollary 1 *Let W be a set of updates that satisfies the Precedence Property. Let W have interaction model G with sink vertex v. Then $[W]$ is the canonical path from the origin of G to v.*

Proof Follows immediately from the definition of $[W]$ as $W|\{\}$. Note that $\{\}$ has both source and sink equal to $(0, \ldots, 0)$.

From Theorem 3.3, the following result is clear:

Theorem 3.4 *Let W_i be the set of updates stored at site i at some instant during the execution of the CCU algorithm on n sites. Let $u_1; \ldots; u_n = [W_i]$. Let G be the interaction model for W_i with* *sink vertex v. Then in the canonical path $e_1, \ldots, e_n$ from $(0, \ldots, 0)$ to v in G, we have $u_j \in L(e_j)$ for $1 \leq j \leq n$. That is, the sequence of updates performed by the CCU algorithm is among the possible update sequences that can be obtained by tracing the canonical path from $(0, \ldots, 0)$ to v in G.*

Notice that Theorem 3.4 does not claim that $[W_i]$ is the only update sequence obtainable by tracing the canonical path from $(0, \ldots, 0)$ to v in G. This issue is much easier to address once we have considered the role of edge-convergence in correctness of groupware systems. We will discuss edge-convergence in detail in the next section and will therefore defer the remainder of the equivalence argument until then.

3.3 Edge-Convergence, TP2, and Correctness

We have already dicussed the Precedence Property, TP1, and the Symmetry Property in detail. Indeed, nearly all of the theory we have developed is built upon the assumption that these properties hold. Our goal in this section is to establish a proof of correctness of the adOPTed algorithm. As we work towards a proof of correctness, we will see that Ressel's final hypothesis, TP2, also plays a key role in the argument.

3.3.1 Edge-Convergence and TP2

As we established in Section 3.1, correctness of the adOPTed algorithm is equivalent to vertex-convergence of the interaction model. However, TP2 comes from considerations related to edge-convergence, which we discuss first.

Recall that an interaction model is called edge-convergent if the label of every edge in the model is a singleton set. In a two-site system, we get edge-convergence for free:

Theorem 3.5 *Let M be a set of updates in a two-site system such that M satisfies the Precedence Property. Let G be an interaction model for M. Then G is edge-convergent.*

Proof We prove the following result by induction: Let $e = (v, w)$ be an edge in G. Then $|L(e)| = 1$. Our proof will be by induction on $n = |v|$. If $n = 0$, then $v = (0, \ldots, 0)$, the initial timestamp, and so e must be in the interaction model by (3.2). Hence any update $u \in L(e)$ must be original. Since there can be only one original update u with $T(u) = v$ and $T'(u) = w$, u must be the only update in $L(e)$, and so $|L(e)| = 1$. Assume $n > 0$. If e is in the interaction model by

(3.2), then $L(e)$ contains only original updates, and as we have seen, this gives $|L(e)| = 1$ immediately. So assume that e is in the interaction model by (3.5). Then by (3.5), for any $u \in L(e)$ there exist edges $e_1 = (v', v)$ and $e_2 = (v', w')$ in G such that there exist updates $u_1 \in L(e_1)$ and $u_2 \in L(e_2)$ such that $u = u_2\hat{\ }u_1$. Notice that e_1 and e do not originate at the same site; otherwise $w' = v$, which gives $e_1 = e_2$ and then (3.5) does not apply (recall that $\hat{\ }$ is not defined over updates that occur at the same site). So e_1 and e_2 originate at different sites. Suppose that for $u' \in L(e)$, $u' \neq u$, there also sites $e_3 = (v'', v)$ and $e_4 = (v'', w'')$ in G with $u' = u_4\hat{\ }u_3$ for some $u_3 \in L(e_3)$ and $u_4 \in L(e_4)$. By the same argument as above, e_3 and e originate from different sites. Since there are only two sites in the system, e_3 and e_1 must originate at the same site. Hence $v' = v''$. Similarly, e_2 and e_4 must originate at the same site, and so $w' = w''$. Thus, $e_3 = e_1$ and $e_4 = e_2$, i.e., e_1 and e_2 are uniquely determined independently of u. Note that $|v'| = |v| - 1$, and both e_1 and e_2 originate at v'. Therefore, by induction, $|L(e_1)| = |L(e_2)| = 1$, and so $L(e_1) = \{u_1\}$ and $L(e_2) = \{u_2\}$ for some updates u_1 and u_2. Let $u \in L(e)$. Since the edges e_1 and e_2 that generate e are uniquely determined, and their respective labels are the singleton sets $\{u_1\}$ and $\{u_2\}$, the only value of u admitted by (3.5) is $u_2\hat{\ }u_1$. Hence, $L(e) = \{u_2\hat{\ }u_1\}$, and in particular $|L(e)| = 1$. Edge-convergence now follows by induction.

When we consider systems with more than two sites, the situation becomes more interesting. Edge-convergence is no longer guaranteed (also recall that Hendrie's counterexample to the modified CCU algorithm assumed a system with at least three sites). As an example, consider a system with three sites. Assume that the system has quiesced at a vertex $v = (a, b, c)$ in the interaction model. Then at timestamp v, site 1 issues update u_1, site 2 issues update u_2, and site 3 issues update u_3, with u_1, u_2, and u_3 pairwise concurrent. This situation is illustrated in Figure 3.3. By (3.2), the interaction model contains the edges $e_1 = (v, (a+1, b, c))$, $e_2 = (v, (a, b+1, c))$, and $e_3 = (v, (a, b, c+1))$ with labels $\{u_1\}$, $\{u_2\}$, and $\{u_3\}$, respectively. By three applications of (3.5), the interaction model also contains the edges e_{12}, e_{21}, e_{13}, e_{31}, e_{23}, and e_{32}, with labels as illustrated in Figure 3.4. Now consider the edge $e_0 = ((a+1, b, c+1), (a+1, b+1, c+1))$ (Figure 3.5). By (3.5) applied to edges e_{21} and e_{31}, e_0 is in the interaction model and $L(e_0)$ contains $(u_2\hat{\ }u_1)\hat{\ }(u_3\hat{\ }u_1)$. However, by (3.5) applied to edges e_{23} and e_{13}, $L(e_0)$ contains $(u_2\hat{\ }u_3)\hat{\ }(u_1\hat{\ }u_3)$. Thus, in a three-site system, an edge may be admitted to the interaction model via two different applications of (3.5), and each of these leads to a (potentially) different expression for the element of the label set. Similar constructions are also possible in larger systems. Hence, edge-convergence is not guaranteed when there are more than two sites.

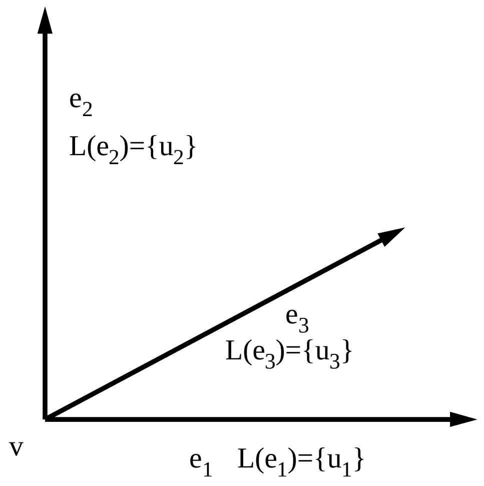


Figure 3.3: Three concurrent updates.

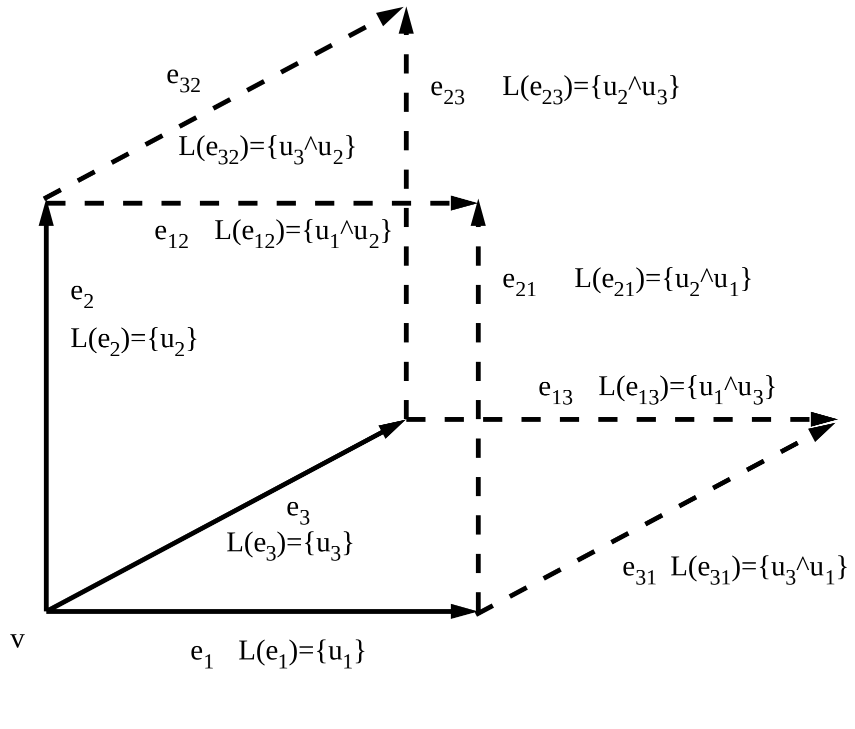


Figure 3.4: Three concurrent updates—three applications of (3.5).

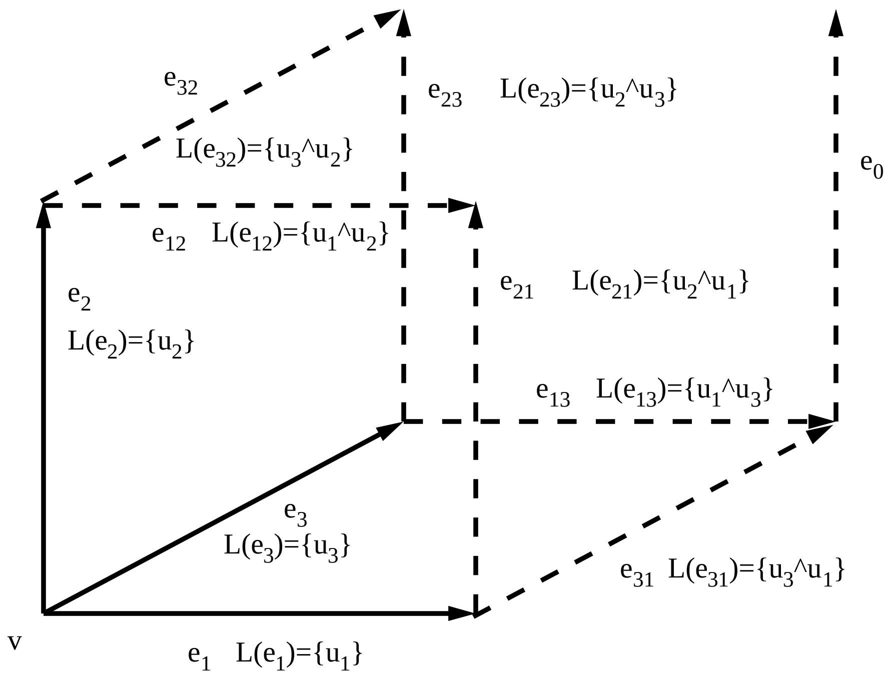

e_{32}
$L(e_{32})=\{u_3 \wedge u_2\}$
e_{23}
$L(e_{23})=\{u_2 \wedge u_3\}$
e_0
e_{12}
$L(e_{12})=\{u_1 \wedge u_2\}$
e_{21}
$L(e_{21})=\{u_2 \wedge u_1\}$
e_2
$L(e_2)=\{u_2\}$
e_{13}
$L(e_{13})=\{u_1 \wedge u_3\}$
e_3
$L(e_3)=\{u_3\}$
e_{31}
$L(e_{31})=\{u_3 \wedge u_1\}$
v
e_1
$L(e_1)=\{u_1\}$

Figure 3.5: Three concurrent updates—four applications of (3.5).

In order to preserve edge-convergence in the three-site case, we need an additional hypothesis. In the above construction, edge-convergence requires $(u_2\hat{}u_3)\hat{}(u_1\hat{}u_3) = (u_2\hat{}u_1)\hat{}(u_3\hat{}u_1)$. In general, for updates a, b, and c, we require $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$. This condition is precisely TP2:

$$\begin{aligned}
\text{TP2:} \qquad & tf_1(tf_1(a,b),c') = tf_1(tf_1(a,c),b'),\ \text{where } tf(b,c) = (b',c') \\
\Leftrightarrow \quad & (tf_1(a,b))\hat{}c' = (tf_1(a,c))\hat{}b' \\
\Leftrightarrow \quad & (a\hat{}b)\hat{}c' = (a\hat{}c)\hat{}b',\ \text{with } b' = b\hat{}c,\ c' = c\hat{}b \\
\Leftrightarrow \quad & (a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)
\end{aligned}$$

Thus, TP2 is a statement about edge-convergence in the three-site case. In the sections to come, we will explore the effect of TP2 in n-site systems, and the overall role of edge-convergence in correctness.

3.3.2 TP2 in n-Site Systems

Based on the developments in the previous section, it is reasonably clear that TP2 is sufficient to guarantee edge-convergence in the three-site case. There can be at most three pairwise concurrent updates at any instant in a three-site system, and TP2 applied to those three updates will avoid the generation of multiple labels that we observed previously. The more interesting question is whether TP2, a condition on triples of updates, is sufficient to guarantee edge-convergence in the n-site case, or whether a stronger condition, or perhaps a family of conditions, is required.

In fact, in the presence of our previous assumptions, TP2 is enough, as we will see in the next theorem. First we need a few preliminary results:

Lemma 3.3 *Let (v_1, w) and (v_2, w) (with $u_1 \in L(v_1, w)$ and $u_2 \in L(v_2, w)$) be edges in interaction model G with $v_1 \neq v_2$. Assume that G is edge-convergent. Then u_1 and u_2 are not original updates, i.e. they are both transformed updates.*

Proof Let $t = (a_1, a_2, \ldots, a_n)$. Without loss of generality, assume that u_1 operates in the first coordinate of the interaction model and that u_2 operates in the second (that is, u_1 and u_2 execute at sites 1 and 2, respectively). Then $v_1 = (a_1 - 1, a_2, \ldots, a_n)$ and $v_2 = (a_1, a_2 - 1, \ldots, a_n)$. If u_1 is an original update, then by the timestamp, it is the a_1-st original update from site 1. By the state vectors, u_1 executes under the assumption that a_2 updates from site 2 have already been applied. If u_2 is also an original update, then we have that u_1 executes under the assumption that u_2 has already executed, i.e. u_1 causally precedes u_2. Then by symmetry, we also have u_2 causally precedes u_1, and this is a contradiction. So u_2 must be transformed. Let $u_0 = or(u_2)$. Then u_0 is the a_2-nd original update from site 2. Let W represent the set of original updates against which u_0 is transformed to produce u_2. Clearly, $or(u_1)$ cannot causally precede any one of these updates, by the existence of the edge (v_1, w) (i.e. state w can be reached by applying u_1 last). Hence, we can transform u_0 with respect to the set of requests $W' = W \setminus \{or(u_1)\}$ without violating the causal ordering. Doing so will produce an update vector whose head is v_1 and whose tail is $(a_1 - 1, a_2 - 1, a_3, \ldots, a_n) = \inf(v_1, v_2)$. Hence $\inf(v_1, v_2)$ is a vertex in G. Let $v = \inf(v_1, v_2)$. At v, neither $or(u_1)$ nor $or(u_2)$ (appropriately transformed) has been applied. Since these updates are concurrent, we can apply them in any order. In particular, we can apply $or(u_1)$ (appropriately transformed), producing the edge (v, v_2). So this edge is in G. Assume that the label of edge (v, v_1) contains the update r and that the label of edge (v, v_2) contains the update s. Then by edge-convergence, we have $u_1 = s\hat{\ }r$ and $u_2 = r\hat{\ }s$. Hence u_1 is not an original request. A similar argument applies to u_2, and the result follows.

From the proof of Lemma 3.3, we can immediately establish the following two results:

Lemma 3.4 *Let (v_1, w) and (v_2, w) be edges in interaction model G. Assume that G is edge-convergent and let $v = \inf(v_1, v_2)$. Then (v, v_1) and (v, v_2) are edges in G.*

Lemma 3.5 *Let $e_1 = (v_1, w)$ and $e_2 = (v_2, w)$ be edges in G. Assume that G is edge-convergent. Then there exist updates r and s such that $L(e_1) = \{r\hat{\ }s\}$ and $L(e_2) = \{s\hat{\ }r\}$.*

Definition 3.11 *Let $G = (V, E)$ be an interaction model, n a nonnegative integer. We denote by $G^{<n}$ the graph $G' = (V', E')$ with $V' = \{v \in V \mid |v| < n\}$ and $E' = \{(v_1, v_2) \in E \mid v_1 \in V', v_2 \in V'\}$.*

We are now ready to prove the main result:

Theorem 3.6 *Let G be an interaction model with updates satisfying the Precedence Property, TP1, the Symmetry Property, and TP2. Then G is edge-convergent.*

Proof We prove the following result by induction on n: Let $e = (v_1, v_2)$ be an edge in G with $|v_1| = n$. Then $|L(e)| = 1$.

If $|v_1| = 0$, then $v_1 = (0, \ldots, 0)$ and so $e = (v_1, v_2)$ represents an original update, having been applied to the initial state. Hence $L(e)$ contains only that original update.

Suppose now that the result holds for all $0 \leq n < k$ and consider the case $n = k$. Let $e = (v_1, v_2)$ be an edge in G with $|v_1| = k$ and with $u \in L(E)$. If e represents an original update, then $L(e) = \{u\}$ and we are done. So assume that e is a transformed update. Then G contains edges $f = (u_1, v_1)$, $g = (u_1, u_2)$, with $L(f) = \{q\}$ and $L(g) = \{p\}$ (by induction $G^{<k}$ is edge-convergent, so $|L(f)| = |L(g)| = 1$) such that $u = p\hat{\ }q$. If only one such pair f and g exists, then $L(e) = \{u\}$ and we are done. So assume that there also exist edges $f' = (u_1', v_1)$, $g' = (u_1', u_2')$, with $L(f') = \{s\}$ and $L(g') = \{r\}$ so that $r\hat{\ }s \in L(e)$ as well. Note that by construction, f' and g' must be parallel to e in G. This implies that $u_1 \neq u_1'$ (otherwise g and g' would coincide) and further, $f \neq f'$. Hence f and f' are perpendicular. Since $f = (u_1, v_1)$ and $f' = (u_1', v_1)$, then by Lemma 3.5, q and s can be expressed as $a\hat{\ }b$ and $b\hat{\ }a$ respectively, for some updates a and b. Let $u_0 = \inf(u_1, u_1')$. By Lemma 3.4 and edge-convergence of $G^{<k}$, edges (u_0, u_1) and (u_0, u_1') are in G and labelled $\{b\}$ and $\{a\}$, respectively. Note that by definition, $or(u) = or(p) = or(r) = o$ for some original update o. Let W be the set of original updates against which o is transformed to produce u. Clearly, $or(p)$ and $or(r)$ are both in W, and neither of these updates causally precedes any other update in W. So we can consider the set of updates $W' = W \setminus \{or(p), or(r)\}$ and transform o with respect to W'. This produces an edge $E = (u_0, u_0 + v_2 - v_1)$ in G, labelled, say, with $\{c\}$ (by edge-convergence of $G^{<k}$). Then we have $p = c\hat{\ }b$ and $r = c\hat{\ }a$. Thus, $L(e)$ contains

$$\begin{aligned} p\hat{\ }q &= (c\hat{\ }b)\hat{\ }(a\hat{\ }b) \\ r\hat{\ }s &= (c\hat{\ }a)\hat{\ }(b\hat{\ }a) \end{aligned}$$

and these two expressions are equal by TP2. Hence both ways by which we constructed e yield the same element of $L(e)$, and therefore $|L(e)| = 1$. The result now follows by induction.

3.3.3 Edge-Convergence and Correctness

From Theorem 3.6, we see that TP2 is sufficient to guarantee edge-convergence in a system of any size. We now explore the connection between edge-convergence and vertex-convergence. The main result is as follows:

Theorem 3.7 *Let G be an edge-convergent interaction model with updates satisfying the Precedence Property, TP1, and the Symmetry Property. Then G is vertex-convergent.*

Proof We prove the following result by induction on n: Let v be a vertex in G with $|v| = n$. Then $|L(v)| = 1$.

If $|v| = 0$, then $v = (0, \dots, 0)$ and the result is clear, since $L(0, \dots, 0) = \{x_0\}$, where x_0 is the initial state.

Suppose now that the result holds for all $0 \leq n < k$ and consider the case $n = k$. Let v be a vertex in G with $|v| = k$. Since $k > 0$, there exists an edge $e = (u_1, v)$ leading to v. By the above argument, $|L(e)| = 1$. If there is only one such e, then $L(v) = \{O(u)(X)\}$ where $L(e) = \{u\}$ and $L(u_1) = \{X\}$, and we are done ($|L(u_1)| = 1$ follows from vertex-convergence of $G^{<k}$). Suppose there are two such edges, $e_1 = (u_1, v)$ and $e_2 = (u_2, v)$. Again, by previous arguments, $|L(e_1)| = |L(e_2)| = 1$. Let $u = \inf(u_1, u_2)$. By Lemma 2, the edges (u, u_1) and (u, u_2) are in G. By induction, $L(u) = \{s\}$ for some state s and $L(u, u_1) = \{a\}$ and $L(u, u_2) = \{b\}$ for some requests a and b. Also by induction, $|L(u_1)| = |L(u_2)| = 1$, so by construction, $L(u_1) = \{O(a)(s)\}$ and $L(u_2) = \{O(b)(s)\}$, respectively. By edge-convergence, (u_1, v) and (u_2, v) are labelled $\{b\hat{}a\}$ and $\{a\hat{}b\}$, respectively. So the two possible labellings of v are $\{O(b\hat{}a)(a(s))\}$ and $\{O(a\hat{}b)(b(s))\}$. But by TP1, these expressions must yield the same result. Thus, $|L(v)| = 1$ and we are done.

The next result now follows immediately:

Theorem 3.8 *Let tf be defined so as to satisfy TP1, the Symmetry Property, and TP2. Then the adOPTed algorithm satisfies the Strong Convergence Property, and is therefore correct.*

Proof Edge-convergence follows from TP2, vertex-convergence follows from edge-convergence, and Strong Convergence follows from vertex-convergence. Since interaction models predict the behaviour of the adOPTed algorithm (see Ressel[18]), correctness follows.

Edge-convergence also allows us to finish our proof of equivalence between the CCU algorithm and the adOPTed algorithm:

Theorem 3.9 *Let x be an initial state and W a set of updates that satisfies the Precedence Property. Assume that $\hat{}$ and tf are defined with the same transformation semantics, and that TP1, the Symmetry Property and TP2 hold. Then the CCU algorithm and the adOPTed algorithm will compute the same final state after transforming and applying all of the updates in W.*

Proof By Theorem 3.4, the label set of each edge in the canonical path from initial state to final state in the interaction model contains an update that the CCU algorithm would apply. By edge-convergence, all edges are labelled with singleton sets, and so there is only one sequence of updates obtainable by tracing the canonical path from initial state to final state. If the adOPTed algorithm (which is non-deterministic) follows the canonical path, then it must compute exactly the same sequence of updates as the CCU algorithm. By edge-convergence, we have vertex-convergence, and so the adOPTed algorithm and the CCU algorithm must compute the same state at the sink vertex of the interaction model, i.e., the same final state.

We now consider the modified CCU algorithm. As we saw in Chapter 2, Hendrie[11] showed by example that the modified CCU algorithm is incorrect. However, the updates defined in Hendrie's example do not satisfy TP2. We would like to show that if we assume TP2, then the modified algorithm works. We now supply a proof of Theorem 2.2 with TP2 as an additional hypothesis:

Theorem 3.10 (Modified CCU Theorem 2) *Let W_1 and W_2 be sets of updates such that W_2 and $W_1 \cup W_2$ satisfy the Precedence Property. Assume that $\hat{}$ is defined so as to satisfy TP2. Then*

$$[W_1 \cup W_2] \quad \equiv \quad [W_2]; (W_1|W_2)$$

Proof Let G_2 be the interaction model for W_2 and G_1 be the interaction model for $W_1 \cup W_2$. Let G_2 have sink vertex v_2, and G_1 have sink vertex v_1. By Theorem 3.3, $[W_1 \cup W_2]$ is the canonical path in G_1 from $(0, \ldots, 0)$ to v_1, $[W_2]$ is the canonical path in G_2 from $(0, \ldots, 0)$ to v_1, and $W_1|W_2$ is the canonical path in G_1 from v_2 to v_1. Thus, both $[W_1 \cup W_2]$ and $[W_2]; (W_1|W_2)$ are connected paths in G_1 from $(0, \ldots, 0)$ to v_1. By TP2, we have edge convergence, which implies vertex convergence. Hence both paths must compute the same state at v_1, and the result follows.

Corollary 1 *With TP2 as an additional hypothesis, the modified CCU algorithm is correct.*

Proof Because the modified CCU algorithm does not process an update until all of the update's causal prerequisites are met, the set W_i of updates in consideration at site i is guaranteed to satisfy the Precedence Property. Hence in any application of $|$ to sets W_1 and W_2, W_2 and $W_1 \cup W_2$ will satisfy the Precedence Property, and the modified CCU Theorem 2 will apply. By the other theorems in Cormack[3], correctness follows.

3.3.4 Necessity of TP2

In the last section, we saw that, in the presence of our previous assumptions (the Precedence Property, TP1, and the Symmetry Property), TP2 is a sufficient condition to guarantee Strong Convergence. However, TP2 is a very strong condition, and it puts very strict constraints on the behaviour of ˆ. Indeed, we have not yet demonstrated that any useful set of operations satisfies TP2. Thus, our goal in this section will be to explore the question of whether TP2 is a necessary condition.

Our first observation is immediate:

Proposition 3.7 *TP2 is not necessary to guarantee correctness in two-site systems.*

Proof Two-dimensional interaction models are always edge-convergent (Theorem 3.5), and correctness follows from edge-convergence.

Indeed, correctness follows without extra hypotheses in the two-site case. However, Hendrie's counterexample shows that the general n-site algorithm does require additional preconditions. The question is whether TP2 is required, or whether we may assume some weaker condition in its place.

We first note that TP2 is a necessary and sufficient condition for edge-convergence:

Proposition 3.8 *Let G be an interaction model for a system with updates satisfying TP1 and the Symmetry Property. Then updates in G satisfy TP2 if and only if G is edge-convergent.*

Proof The forward implication follows from Theorem 3.6. The reverse implication follows from the construction in Section 3.3.1.

Hence the question of necessity of TP2 reduces to a question of necessity of edge-convergence. This question remains open. However, we can demonstrate necessity in several restricted cases:

Lemma 3.6 *For any groupware system with $n > 3$ sites whose behaviour is predicted by an interaction model, and whose updates satisfy TP1 and the Symmetry Property, the following condition is neccessary for Strong Convergence:*

$$(a; (b\hat{}a); ((c\hat{}a)\hat{}(b\hat{}a))) \quad \equiv \quad (b; (a\hat{}b); ((c\hat{}b)\hat{}(a\hat{}b))) \tag{3.6}$$

for all updates a, b, and c.

Proof Assume that the system is at timestamp $t = (0, \ldots, 0)$ with initial state $x \in X$. Consider again the construction outlined in Figures 3.3, 3.4, and 3.5. Sites 1, 2, and 3 issue updates u_1, u_2, and u_3, respectively, with u_1, u_2, and u_3 pairwise concurrent. We can reach the sink vertex of the interaction model via several paths, two of which are $(e_1; e_{21}; e_0)$, and $(e_2; e_{12}; e_0)$. The first path computes the state $(u_1; u_2\hat{}u_1; (u_3\hat{}u_1)\hat{}(u_2\hat{}u_1))(x)$. The second path computes the state $(u_2; u_1\hat{}u_2; (u_3\hat{}u_2)\hat{}(u_1\hat{}u_2))(x)$. Since we require vertex convergence, these two states must be identical:

$$\begin{aligned}(u_1; u_2\hat{}u_1; (u_3\hat{}u_1)\hat{}(u_2\hat{}u_1))(x) &= (u_2; u_1\hat{}u_2; (u_3\hat{}u_2)\hat{}(u_1\hat{}u_2))(x) \text{ for all } x \in X \\ (u_1; u_2\hat{}u_1; (u_3\hat{}u_1)\hat{}(u_2\hat{}u_1)) &\equiv (u_2; u_1\hat{}u_2; (u_3\hat{}u_2)\hat{}(u_1\hat{}u_2))\end{aligned}$$

for all updates u_1, u_2, and u_3. The result follows with $u_1 = a$, $u_2 = b$, and $u_3 = c$.

Notice that (3.6) follows from TP2, and is therefore a potentially weaker precondition. However, under certain circumstances, TP2 follows from (3.6):

Theorem 3.11 *Suppose that the set O of supported operations has the property that every operation $o \in O$ is surjective. Then TP2 is necessary for Strong Convergence.*

Proof Let a, b, and c be updates with $O(a)$, $O(b)$, and $O(c)$ in O. By Lemma 3.6, we have

$$(a; b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a)) \equiv (b; a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))$$

Let $x \in X$. By surjectivity, there is some $y \in X$ such that $(a; b\hat{}a)(y) = (b; a\hat{}b)(y) = x$. We then have

$$\begin{aligned}(a; b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a))(y) &= (b; a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))(y) \\ (O((c\hat{}a)\hat{}(b\hat{}a)))((a; b\hat{}a)(y)) &= (O((c\hat{}b)\hat{}(a\hat{}b)))((b; a\hat{}b)(y)) \\ (O((c\hat{}a)\hat{}(b\hat{}a)))(x) &= (O((c\hat{}b)\hat{}(a\hat{}b)))(x).\end{aligned}$$

Thus $(O((c\hat{}a)\hat{}(b\hat{}a)))(x) = (O((c\hat{}b)\hat{}(a\hat{}b)))(x)$ for all $x \in X$. Since we are modeling operations as functions on the state space X, we thus have $O((c\hat{}a)\hat{}(b\hat{}a)) = O((c\hat{}b)\hat{}(a\hat{}b))$. Since we also have $S((c\hat{}a)\hat{}(b\hat{}a)) = S((c\hat{}b)\hat{}(a\hat{}b))$ and $T((c\hat{}a)\hat{}(b\hat{}a)) = T((c\hat{}b)\hat{}(a\hat{}b))$, we have $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$, and we see that TP2 is necessary for Strong Convergence.

Theorem 3.12 *Suppose that the set O of supported operations has the property that for all $f, g \in O$, and for all $x \in X$, $f(g(x)) = f(x)$ (i.e. operations in O* mask *each other). Then TP2 is necessary for Strong Convergence.*

Proof Let a, b, and c be updates with $O(a)$, $O(b)$, and $O(c)$ in O. By Lemma 3.6, we have

$$(a; b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a)) \equiv (b; a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))$$

Let $x \in X$. Then we have

$$\begin{aligned}
(a; b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a))(x) &= (b; a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))(x) \\
(b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a))(O(a)(x)) &= (a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))(O(b)(x)) \\
(b\hat{}a; (c\hat{}a)\hat{}(b\hat{}a))(x) &= (a\hat{}b; (c\hat{}b)\hat{}(a\hat{}b))(x) \\
O((c\hat{}a)\hat{}(b\hat{}a))(O(b\hat{}a)(x)) &= O((c\hat{}b)\hat{}(a\hat{}b))(O(b\hat{}a)(x)) \\
O((c\hat{}a)\hat{}(b\hat{}a))(x) &= O((c\hat{}b)\hat{}(a\hat{}b))(x).
\end{aligned}$$

Thus $(O((c\hat{}a)\hat{}(b\hat{}a)))(x) = (O((c\hat{}b)\hat{}(a\hat{}b)))(x)$ for all $x \in X$. Hence, as in the proof of Theorem 3.11, TP2 is necessary for Strong Convergence.

Notice that the operators Rock, Paper, and Scissors from Hendrie's example satisfy the hypotheses of Theorem 3.12. Hence any transformation scheme we define on these operators must satisfy TP2; otherwise Strong Convergence will fail.

Text buffer insertions and deletions are neither masking operations nor surjective. However, in some sense, they are "almost" surjective. For example, although not every text buffer can be obtained by applying insert(3, "a") to some state x, every text buffer with "a" at position 3 can. Thus, we would expect TP2 to be necessary for insertions and deletions, although we cannot conclude necessity from Theorem 3.11.

The following result gives another condition that necessitates TP2:

Theorem 3.13 *Suppose the set O of supported operations has the property that for all $f, g \in O$ and for all $x \in X$, if $f \neq g$ then $f(x) \neq g(x)$. Then TP2 is necessary for Strong Convergence.*

Proof Suppose that TP2 is not necessary for Strong Convergence. Then there exists an interaction model G which is vertex-convergent, but not edge-convergent. Let $e = (v_1, v_2)$ be an edge in G with $|L(e)| > 1$. Then there exist updates $u_1, u_2 \in L(e)$ with $O(u_1) = f$ for some $f \in O$, $O(u_2) = g$ for some $g \in O$, and $f \neq g$. Since G is vertex-convergent, $L(v_1) = \{x_0\}$ for some state $x_0 \in X$. By (3.3), $L(v_2)$ contains $f(x_0)$ and $g(x_0)$. But by vertex-convegence, $|L(v_2)| = 1$, and so $f(x_0) = g(x_0)$. Hence, if for all $f, g \in O$, $x \in X$ we have $f(x) \neq g(x)$, TP2 must be necessary for Strong Convergence, and the result follows.

Now consider the set of text buffer insertions and deletions from the point of view of Theorem 3.13. Let x be a text buffer and f and g be text buffer operations with $f \neq g$. Then it is easy to check via case analysis that $f(x) \neq g(x)$, and so we have the following corollary:

Corollary 1 *TP2 is necessary when O is the set of text buffer insertions and deletions.*

The question of whether TP2 is truly necessary in all cases is still open. However, we have demonstrated that in several important cases, it is indeed necessary. Therefore, we make the following conjecture:

Conjecture 3.1 *TP2 is a necessary condition for Strong Convergence.*

3.3.5 Verifying TP2

In the previous section, we saw that TP2 is a necessary condition for Strong Convergence in many cases, and we conjectured that it is always necessary. Thus, as designers of groupware systems, we would like to be able to verify whether our formulations of ˆ or *tf* really do satisfy TP2. Verifying TP2 turns out to be a very time-consuming problem; since it is a condition on triples of updates, the verification process tends to involve cubic effort. In this section, we explore a few opportunities to reduce the effort somewhat, but the general problem remains difficult.

Our first observation is based on Ressel's NOOP Property[18]:

Definition 3.12 *Let X be a set of states. Denote by NOOP the operation $f \in X^X$ such that $f(x) = x$ for all $x \in X$. Also denote by NOOP any update u for which $O(u) =$ NOOP (when there is no possibility of ambiguity).*

Definition 3.13 *A groupware system is said to satisfy the NOOP Property if for all operations $f \in O$, $NOOP\hat{}f = NOOP$ and $f\hat{}NOOP = f$.*

Proposition 3.9 *If a groupware system has $NOOP \in O$ and satisifies the NOOP Property, then for all updates a, b, c, if one of a, b, and c is NOOP, then TP2 holds for all permutations of a, b, and c.*

Proof We will verify the following formulation of TP2:

$$(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c).$$

The other permutations can be obtained by renaming a, b, and c. If $a = \text{NOOP}$, then

$$\begin{aligned}(a\hat{\ }b)\hat{\ }(c\hat{\ }b) &= (\text{NOOP}\hat{\ }b)\hat{\ }(c\hat{\ }b)\\ &= \text{NOOP}\hat{\ }(c\hat{\ }b)\\ &= \text{NOOP}\end{aligned}$$

and

$$\begin{aligned}(a\hat{\ }c)\hat{\ }(b\hat{\ }c) &= (\text{NOOP}\hat{\ }c)\hat{\ }(b\hat{\ }c)\\ &= \text{NOOP}\hat{\ }(b\hat{\ }c)\\ &= \text{NOOP}.\end{aligned}$$

If $b = \text{NOOP}$, then

$$\begin{aligned}(a\hat{\ }b)\hat{\ }(c\hat{\ }b) &= (a\hat{\ }\text{NOOP})\hat{\ }(c\hat{\ }\text{NOOP})\\ &= a\hat{\ }c\end{aligned}$$

and

$$\begin{aligned}(a\hat{\ }c)\hat{\ }(b\hat{\ }c) &= (a\hat{\ }c)\hat{\ }(\text{NOOP}\hat{\ }c)\\ &= (a\hat{\ }c)\hat{\ }\text{NOOP}\\ &= a\hat{\ }c.\end{aligned}$$

The case that $c = \text{NOOP}$ is identical to the case that $b = \text{NOOP}$.

Thus, when verifying TP2, as long as our groupware system satisfies the NOOP Property, we do not need to consider the case that one of the updates is NOOP.

One might wonder about the value of Proposition 3.9, as it is difficult to imagine that an application designer would create a system that supported an operation that does nothing. However, NOOP can find its way into a groupware system in a very natural manner: the transformation operators $/$ and $\backslash$ are defined as binary operators on the set O of supported operations. Therefore, by definition of an operator, O must be closed under $/$ and $\backslash$. It is often the case that the result of applying / or \ to two operations will result in NOOP. For example, one might define insert(3, "a")/delete$(2, 4)$ to be NOOP, as the region into which the "a" was to be inserted was deleted. Thus, the requirement that O be closed under / and \ can very naturally introduce NOOP into the system.

Our next observation is based on the idea of commuting updates:

Definition 3.14 *Operations f and g are said to* commute *with each other if for all states $x \in X$, $f(g(x)) = g(f(x))$. Updates u and v are said to commute with each other if $T(u) = T(v)$, $S(u) \neq S(v)$, and $O(u)$ and $O(v)$ commute with each other.*

Definition 3.15 *A groupware system is said to have the Commuting Property if for all updates u and v, if u and v commute with each other, then $u\hat{}v = u$ and $v\hat{}u = v$.*

Proposition 3.10 *Let a, b, and c be updates that commute pairwise in a groupware system that has the Commuting Property. Then TP2 holds for any permutation of a, b, and c.*

Proof We will check the following formulation of TP2:

$$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c).$$

Other permutations can be obtained by renaming a, b, and c. We have

$$\begin{aligned}(a\hat{}b)\hat{}(c\hat{}b) &= a\hat{}c \\ &= a\end{aligned}$$

and

$$\begin{aligned}(a\hat{}c)\hat{}(b\hat{}c) &= a\hat{}b \\ &= a.\end{aligned}$$

The result now follows.

Proposition 3.11 *Let a be an update that commutes with all other updates in a groupware system that has the Commuting Property. Then for all updates b and c such that a, b, and c are pairwise concurrent, TP2 holds for any permutation of a, b, and c.*

Proof Since

$$\begin{aligned}(a\hat{}b)\hat{}(c\hat{}b) &= a\hat{}(c\hat{}b)\\ &= a\end{aligned}$$

and

$$\begin{aligned}(a\hat{}c)\hat{}(b\hat{}c) &= a\hat{}(b\hat{}c)\\ &= a,\end{aligned}$$

we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$. Also, since

$$(b\hat{}a)\hat{}(c\hat{}a) = b\hat{}c$$

and

$$\begin{aligned}(b\hat{}c)\hat{}(a\hat{}c) &= (b\hat{}c)\hat{}a\\ &= b\hat{}c,\end{aligned}$$

we have $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$. The verification that $(c\hat{}b)\hat{}(a\hat{}b) = (c\hat{}a)\hat{}(b\hat{}a)$ is identical.

Thus, by Proposition 3.10, we can ignore any triple of updates that commute pairwise, and by Proposition 3.11, we can ignore any operation that commutes with all others. Indeed, the proof of Proposition 3.11 shows that a need not commute with all other updates; it is sufficient for a to commute with the $\hat{}$-closure of $\{b, c\}$.

Our final observation is based on the idea that some updates may be expressible in terms of others:

Theorem 3.14 *Let O and O' be sets of operations with transformations defined so that operations in O' satisfy TP1 and TP2. Suppose that every operation in O can be expressed as a combination of operations in O' such that the transformation rules in O agree with those in O'. Then Strong Convergence is guaranteed in O. Further, if TP2 is known to be necessary, then TP2 is guaranteed in O.*

Proof Let u be an update with $O(u) \in O$. Then there exist operations $o_1, \ldots, o_m \in O'$ such that $O(u) = o_m \circ \cdots \circ o_1$. Let S_1 be a groupware system based on O and S_2 be a groupware system based on O'. The update u_1 issued by site i at time t corresponds to an edge e in the interaction model G_1 for S_1. If, in S_2, at time t, site i issues a sequence of updates $(u_1; \ldots ; u_m)$ with $O(u_j) = o_j$ for $1 \leq j \leq n$, then the interaction model G_2 for S_2 contains a sequence of edges that, laid end-to-end, create a single large vector that correspond to the edge e in G_1. Thus, we can superimpose G_2 onto G_1 and we discover that G_1 is, in fact, a subgraph of G_2[1]. Since TP1 and TP2 hold in S_2, G_2 is vertex-convergent, and since G_1 is a subgraph of G_2, G_1 is vertex-convergent as well. If TP2 is necessary for Strong Convergence in S_1, then edge-convergence follows from vertex-convergence, and TP2 follows from edge-convergence. The result follows.

Theorem 3.14 makes it possible to verify TP2 on a set of operations by breaking them down as compositions on a smaller set of operations and verifying TP2 on the smaller set. We can also take an existing set of operations on which TP2 is known to hold and build more complex operations from them without having to verify TP2. For example, if a set of transformation rules on text buffer insertions and deletions is known to satisfy TP2, then we can implement a "move" operation as a deletion followed by an insertion without having to check TP2 on the new operation.

In this chapter, we showed equivalence between the CCU and adOPTed algorithms. We showed that, in the presence of TP2, the Strong Convergence Property holds, and the modified CCU algorithm is correct. We have seen that TP2 is often a necessary condition for correctness, and although the general question of necessity remains open, we expect that it is indeed a necessary condition. Finally, we illustrated a few techniques to aid in the process of verifying TP2. In the next chapter, we will see how to construct a general toolkit for constructing distributed objects based on the modified CCU algorithm.

[1]Strictly, speaking, G_1 is not a subgraph of G_2 because when exterior vertices are removed from G_2 to produce G_1, the exterior edges they connect are coalesced rather than deleted. Interior edges and vertices, however, are deleted. We use the term "subgraph" for lack of a better one.

Chapter 4

A Library for Operation Transforms in ML

In this chapter, we discuss the implementation of a framework for constructing distributed objects based on the modified CCU algorithm. We begin with a summary of the implementation language, Concurrent ML, and then describe the details of the implementation.

4.1 Summary of Concurrent ML

Concurrent ML (CML) was designed by John Reppy[17] in 1990. It is an extension of SML[14, 22] that allows for concurrent programming and is included in the more recent distributions of Standard ML of New Jersey (SML/NJ).

CML was written entirely in ML, without any modifications to the compiler. Because SML/NJ is internally represented using continuation passing style[1] and provides first-class continuations, a thread can be represented simply as a continuation, and can thus be created very cheaply. Preemption is accomplished via SML/NJ's signal-handling mechanism. Threads are automatically garbage-collected when they finish their computation.

4.1.1 Thread Management

Threads are created in CML via the `spawn` function in the `CML` structure. There are two variants of `spawn`[1]:

```
type thread_id
val spawn : (unit -> unit) -> thread_id
val spawnc : ('a -> unit) -> 'a -> thread_id
```

`spawn` takes a function `f`, of type `unit -> unit`, and creates a new thread to execute `f`. The new thread's ID is returned to the caller. `spawnc` is similar, except that it allows the caller to pass a parameter to the function `f`. Notice that, in both cases, the function `f` returns `unit`. This restriction is forced upon the design of CML by the fact that continuations do not return values. Thus, since the return value of the function will be ignored anyway, it is assumed to be `unit`. As a result, if a thread needs to communicate a computed value to its parent, it needs to use some mechanism other than the return value of the function. We will discuss communication among threads in the next section.

A thread can give up control of the CPU using the `yield` function, and can terminate itself via the `exit` function:

```
val yield : unit -> unit
val exit : unit -> 'a
```

The return type of `exit` is `'a` because `exit` never returns.

4.1.2 Communication Among Threads

Although it is possible for CML threads to communicate via shared memory (a mutable value may reside in the environment of multiple threads), this approach to communication is decidedly non-functional and the CML design discourages its use (for example, CML contains no built-in support for mutex locks).

[1]In the syntax of ML type expressions, `->` is the type constructor for functions and `*` is the type constructor for tuples. The type `unit` is the degenerate type with a single element, denoted `()`. The type `exn` is the type shared by all exceptions. Quoted identifiers, such as `'a` or `'b`, represent type variables, and act as placeholders for any valid type.

Instead, CML threads communicate via message-passing. Messages are passed over *channels*, which are created via the `channel` function in the `CML` structure:

```
type 'a chan
val channel : unit -> 'a chan
```

A value of type `'a chan` is a channel capable of carrying messages of type `'a`. A channel is essentially a queue of values. Channels are not associated with a particular sender or receiver; arbitrary threads may send and/or receive on a particular channel. Also note that the type `'a chan` is parametrically polymorphic, and that channels are first-class values. In other words, given any type, we can create a channel capable of carrying values of that type. Thus channels can carry functions, or even other channels. Furthermore, we may parameterize functions over channels or store channels in data structures. This degree of flexibility provides considerable freedom in designing communication schemes.

Communication over channels is done using the functions `send` and `recv`:

```
val send : ('a chan * 'a) -> unit
val recv : 'a chan -> 'a
```

`send` and `recv` are blocking operations. In particular, `send` blocks until a receiver picks up the message that was sent, and `recv` blocks until a message is available for receipt. Note that it is a trivial matter to implement a non-blocking send operation; we simply spawn a thread that calls the provided `send` function and then terminates. However, CML provides a separate mechanism, *mailboxes*, for asynchronous communication:

```
type 'a mbox
val mailbox : unit -> 'a mbox
val send : ('a mbox * 'a) -> unit
val recv : 'a mbox -> 'a
```

Mailboxes are used in the same way as channels, except that `send` for mailboxes is non-blocking. When there is no clear reason to prefer synchronous or asynchronous communication, Reppy advocates synchronous communication, as reasoning about synchronous communication is generally easier[17].

4.1.3 Synchronization and First-Class Events

A hallmark feature of CML is its treatment of synchronization. CML distinguishes the act of synchronization from the event upon which synchronization occurs. The event is abstracted into a concrete value and given first-class status in the language. As a result, events may be stored in data structures, or threads may exchange events with each other over channels. Thus, it is possible for a thread to synchronize on another thread's events.

Events are created by calling one of several functions in the `CML` structure:

```
type 'a event
val joinEvt : thread_id -> unit event
val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event
val timeOutEvt : Time.time -> unit event
val atTimeEvt : Time.time -> unit event
```

A value of type `'a event` is an event that produces a value of type `'a` upon synchronization. The function `joinEvt` takes the ID of a particular thread and produces an event that synchronizes on the termination of that thread. `sendEvt` and `recvEvt` return events that synchronize on the completion of a `send` or `recv` operation. `timeOutEvt` produces an event that synchronizes after the specified amount of time has passed. `atTimeEvt` produces an event that synchronizes after a specified clock time is reached.

Note that none of the above-mentioned functions block; they do not perform synchronizations, but instead return the objects upon which synchronizations may be performed. Synchronization is performed via another function, `sync`:

```
val sync : 'a event -> 'a
```

Invoking `sync` on an event causes the calling thread to block until the event's synchronization condition is met. `sync` then returns the value of type `'a` carried by the event. For example, joining on a particular thread is performed as follows:

```
val _ = sync (joinEvt TID)
```

where `TID` is the ID of the thread upon which we wish to join.

Often, a thread may need to synchronize on one of several events. For example, a server may need to poll several channels and choose one on which a message is waiting (or block, if all channels are empty). CML provides two operations, `select` and `choose`, for this purpose:

```
val select : 'a event list -> 'a
val choose : 'a event list -> 'a event
```

`choose` takes a list of events and non-deterministically returns a ready event from the list. If none of the events are ready, `choose` blocks until one or more of the events become ready, and then returns one of these. `select` chooses a ready event from an event list and synchronizes on it; it is semantically equivalent to `sync o choose`, but has a more efficient implementation.

CML also supports *event combinators*, operations that act on events and produce new events:

```
val wrap : ('a event * ('a -> 'b)) -> 'b event
val wrapHandler : ('a event * (exn -> 'a event)) -> 'a event
val alwaysEvt : 'a -> 'a event
```

`wrap` attaches a post-synchronization action to an event. When the event is synchronized, the post-synchronization function is invoked and the result returned to the caller. `wrapHandler` is similar, except that it wraps a post-synchronization exception handler around an event. `alwaysEvt` returns an event that is always ready. Given a value `x`, `alwaysEvt x` returns an event that is always ready, and returns `x` upon synchronization. `alwaysEvt` is useful for specifying default actions in event lists; if, during a `select` or `choose`, no other events in the event list are ready, an event that is always ready can be used to trigger some alternative action (although it is possible for the always-ready event to be selected even when other events are ready).

4.1.4 Multicast

A message sent over a channel or mailbox is picked up by a single recipient. Often, we would like to send a message to several recipients, and the exact set of recipients may not be computable at design time. In such situations, we can use multicast channels, which are defined in the `Multicast` structure:

```
type 'a mchan
type 'a port
val mChannel : unit -> 'a mchan
val port : 'a mchan -> 'a port
val recv : 'a port -> 'a
val recvEvt : 'a port -> 'a event
val multicast : ('a mchan * 'a) -> unit
```

A value of type `'a mchan` is a multicast channel capable of carrying messages of type `'a`. Multicast channels are created with `mChannel`. Messages are sent over multicast channels using the function `multicast`. Note that, by necessity, `multicast` is non-blocking, as the set of recipients is not of fixed size, and may even be empty.

A thread that needs to retrieve values from a multicast channel creates a *port* on that channel. A port on a `'a mchan` is a value of type `'a port`. A port is created via a call to `port`. Each port on a particular multicast channel receives all of the messages sent on that channel since the port's creation. A port does not receive messages that were sent before it was created; otherwise, the garbage collector could never collect old messages. A thread receives messages from a port via the `recv` function. `recv` is blocking; `recvEvt` returns the corresponding event.

4.1.5 A Sample CML Program

CML objects cannot be manipulated at the top level; all CML code must reside inside of structures. CML programs are launched by invoking `RunCML.doit` with the name of a "main" function that runs the application.

An example "hello world" program is illustrated below:

```
structure Hello = struct
    val ch: string CML.chan = CML.channel()

    fun receiver () =
       let
          val s = CML.recv ch
```

```
      in
         TextIO.print s
      end

   fun sender () =
      let
         val _ = CML.spawn receiver
      in
         CML.send(ch, "Hello world!\n")
      end

   fun main' () =
      let
         val tid = CML.spawn sender
      in
         CML.sync (CML.joinEvt tid)
      end

   fun main () = ignore (RunCML.doit(main', SOME(Time.fromMilliseconds 5)))
end
```

The program is launched by calling `Hello.main()`. `main` calls the auxiliary function `main'`, which spawns a thread to execute the function `sender`. `sender` spawns a thread to execute the function `receiver`. `sender` then sends `receiver` the string "Hello world!\n" over a channel and `receiver` prints the string. `main'` joins on `sender` and then finishes. The optional second argument to `RunCML.doit` specifies the time-slicing quantum.

4.2 Implementation of CCU in CML

In this section, we outline the construction of a generic CCU module in CML. We begin with a description of the overall system architecture and then describe the CCU module itself.

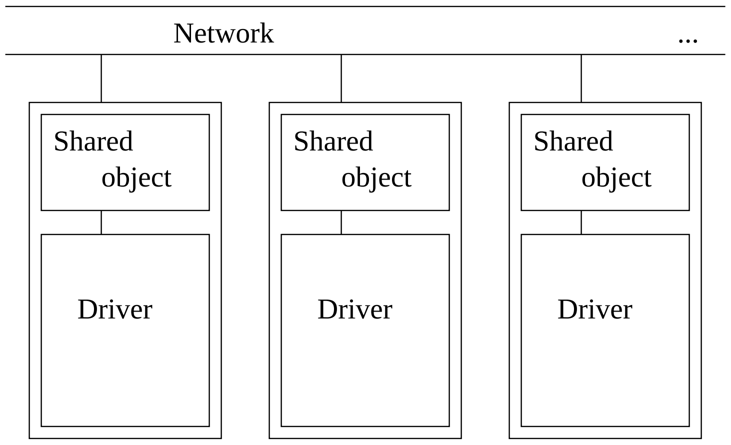


Figure 4.1: System architecture

4.2.1 System Architecture

The overall architecture of our system is based on the GROVE system of Ellis[7], and is illustrated in Figure 4.1. Each of the large boxes represents a site. Sites are connected to one another via some communications network. The logic at each site is broken into two components: the shared object and the driver. The shared object maintains the local copy of the shared state and performs all of the communication with other sites necessary to implement the CCU algorithm. The driver is the application making use of the shared object. It encapulates the user interface and any application logic, and is responsible for issuing updates to the shared object.

The shared object provides a set of interface functions to the driver that allow the driver to issue updates and query the shared state. While the shared object will have the same implementation from site to site, sites may use different drivers. Thus, for example, two sites may be sharing a text buffer, but using different text editor “front ends” to issue updates to the buffer.

Our system is a prototype system running on a simulated network. A separate module provides the network abstraction; a real network may be substituted by supplying an alternate implementation of this abstraction. Providing the actual communications layer and handling the associated deployment issues are problems left to future investigation.

Our implementation will provide a facility for generating implementations of the shared object component for each site. The implementation of drivers is left to the application designer, but we will provide a sample driver for a shared text buffer in Chapter 5.

4.2.2 The Network Module

The dOPT, CCU and adOPTed algorithms all assume an underlying communications framework in which all messages are eventually delivered, although not necessarily in order, and with the potential for arbitrarily long delays. A broadcast network is modeled nicely by a multicast channel, with facilities for random delay and reordering of messages. Thus our network signature is simply the multicast signature:

```
signature NETWORK = MULTICAST
```

The `Network` structure provides most of the functionality of the original `Multicast` structure, with two main modifications. First, the function `multicast` now spawns a thread that delays a random amount of time before sending its message on the multicast channel. Second, the function `port` now installs a server and a mailbox on any port that it creates. The server picks up messages from the port, and then spawns a thread for each message that delays a random amount of time and sends the message to the mailbox. `recv` now receives its messages from the mailbox instead of directly from the port. These two modifications introduce delays into the system, and break the FIFO ordering of events that would come from simply using a multicast channel to model the network.

4.2.3 The Timestamp Module

The `Timestamp` structure implements the vector timestamps that are used by both the CCU and adOPTed algorithms to detect conflicting updates. Its signature is as follows:

```
signature TIMESTAMP = sig
   exception Incompatible
   exception Range

   eqtype timestamp

   val mktimestamp : int -> timestamp
   val size : timestamp -> int
   val inc : (timestamp * int) -> timestamp
   val causalLT : (timestamp * timestamp) -> bool
   val totalLT : (timestamp * timestamp) -> bool
   val sup : (timestamp * timestamp) -> timestamp
   val inf : (timestamp * timestamp) -> timestamp
   val toString : timestamp -> string
end
```

The `Timestamp` structure exports a type `timestamp` and a set of interface functions for manipulating timestamps. `mktimestamp` creates the timestamp $(0, \ldots, 0)$ of a specified length. `size` returns the number of components of a particular timestamp. `inc` increments a specified component of the given timestamp. `causalLT` and `totalLT` compare two timestamps according to the causal order $\subseteq$ and the total order $\leq$, respectively. `sup` and `inf` compute the supremum and infimum, respectively, of a pair of timestamps. `toString` returns a printable representation of a particular timestamp. `causalLT`, `totalLT`, `sup`, and `inf` raise the exception `Incompatible` if invoked on timestamps of different sizes. `inc` raises the exception `Range` if an invalid component is specified.

4.2.4 The CCU Module

The CCU module implements the modified CCU algorithm. We wish to allow the application designer to specify the nature of the shared state and the operations on that state. We will then take these specifications and construct the shared objects. SML has a natural mechanism for creating abstractions in this way: functors. Thus, our CCU module will be a functor.

The input to the functor must contain a specification of the shared state, the set of supported operations on the state, and the rules for transforming operations against one another. Here our model from previous chapters breaks down. We have been modelling operations as functions on the state space X, and thus the operators $/$ and $\backslash$ are actually functions operating on functions. However, equality of functions is undecideable in general, and so any attempt to define $/$ and $\backslash$ as operators on functions will fail.

Instead, we take a more abstract approach to modelling states and operations. We will represent both of these entities as types, and then $/$ and $\backslash$ can be modelled as functions on the operation type. In addition, we need to supply an additional function, `apply`, that provides the semantics of the supported operations. This approach reduces the problem of determining equality of functions to testing equality of values of the operation type (of course, it is possible for two different values of the operation type to have the same semantics; / and \ need to be aware of such "equivalence classes").

The input signature, `CCUOBJ`, is as follows:

```
signature CCUOBJ = sig
    type state
    val stateToString : state -> string

    eqtype operation
    val operationToString : operation -> string
    val apply : (operation * state) -> state
    val / : (operation * operation) -> operation
    val \ : (operation * operation) -> operation
end
```

The functions `stateToString` and `operationToString` return printable representations of states and operations.

The output from the functor is a structure that provides interface functions for drivers. A driver needs to be able to issue updates to the shared state and to query the shared state for presentation to the end user. In addition, the structure must provide interface functions for creating and initializing instances of the shared object. These requirements comprise the output signature, `CCUAPI`, which is as follows:

```
signature CCUAPI = sig
    type state
    eqtype operation
    type siteid

    structure T : TIMESTAMP

    (* For communication with the driver. *)
    datatype message = MSG of operation * T.timestamp
                     | QUIT of T.timestamp
    (* For communication with peers. *)
    datatype netmessage = NMSG of operation * siteid * T.timestamp
                        | NQUIT of siteid * T.timestamp
    type ccuobject
    type commtoken

    (* Blocking *)
    val update : (commtoken * message) -> T.timestamp
    val query : commtoken -> (state * T.timestamp)
    exception Done

    (* Non-blocking -- use with caution. *)
    val send : (commtoken * message) -> unit

    structure M : MULTICAST
    val create : (state * netmessage M.mchan * netmessage M.port
                        * int * siteid) -> ccuobject
    val start : ccuobject -> commtoken
    val numSites : ccuobject -> int
end
```

The types `state` and `operation` are the same as those in the input signature, `CCUOBJ`, but are repeated here because they are used to specify the types of some of the elements of `CCUAPI`. The type `siteid` is an abstraction of the type used to specify and distinguish sites (currently, `siteid` is simply a synonym for `int`). The datatype `message` is the type used by drivers to send messages to the shared objects. A message can either be an update (characterized by an operation and a timestamp; the site ID is already known by the shared object) or a directive to quit. The datatype `netmessage` is the type used by shared objects to communicate with each other. Again, a message can either be an update or a quit directive.

The type `ccuobject` is the return type of the `create` function and the input type of the `start` function. It is simply a placeholder for an initialized object between a call to `create` and a call to `start`. Drivers do not use the type `ccuobject` directly; instead, they use the type `commtoken` which encapsulates the communications pipeline between the driver and its shared object, much like a file descriptor.

Communication between a driver and its instance of the shared object occurs via the two interface functions `update` and `query`. `update` issues an update to the shared object and returns a timestamp, indicating the instance's knowledge of the state of the system. `query` queries the shared object for its current state. `query` returns the current state and the current timestamp. `update` and `query` are blocking operations. A non-blocking `send` for `commtoken`s is also provided for convenience, but one must be careful that the sender does not flood the `commtoken` with messages.

Instances of the shared object are created by calling `create`. `create` takes as input an initial state, a simulated network channel (see Section 4.2.2), a port on that channel, the number of sites in the system, and a site ID, and produces as output a `ccuobject` encapsulating this information. This object will write to the provided channel and read from the provided port when communicating with its peers.

An instance of the shared object is activated by calling `start`. `start` takes as input a `ccuobject` and creates a `commtoken` for use by the driver. It then spawns a thread that polls the driver and the network for incoming updates and applies them to the shared state (after suitable transformation). `start` then returns the `commtoken`.

4.2.5 Initialization

Initialization of shared objects and drivers is a somewhat involved process. To create a system consisting of n sites, we proceed in several steps. First, we create a multicast channel for delivering messages among peers. Second, we create n ports on the multicast channel. Next, we create n `ccuobject`s, each initialized with a different port. Next, we create n drivers, each initialized to communicate with a different instance of the shared object. Finally, we start each of the drivers.

The order in which we perform these tasks is important. For example, we must not start any driver before we have created all of the ports; otherwise, the driver may start to send messages and the ports that do not exist yet would miss them. The entire process of initialization is sufficiently tedious and error-prone that we would like to automate it somewhat.

To facilitate the initialization process, we introduce another functor, `InitFn`. `InitFn` takes as parameters the shared object structure (i.e. the output of the `CCU` functor) and the driver structure and produces the necessary initialization code. The output of `InitFn` is a structure whose signature consists of a single function:

```
val init : (int * Obj.state) -> unit
```

`init` takes as parameters the number of sites and the initial state (the structure `Obj` represents the output of the `CCU` functor), and returns no result. When invoked, `init` initializes the drivers and shared object, and runs the system.

Drivers passed as arguments to `InitFn` must conform to the following signature:

```
signature DRIVER = sig
   type driver
   type commtoken
   type init
   val initialize : unit -> unit
   val initData : unit -> init
   val mkdriver : (unit -> commtoken) -> Timestamp.timestamp ->
                  init -> driver
   val main : driver -> unit
end
```

The type `driver` is the return type of `mkdriver` and the input type of `main`. It is primarily an encapsulation of the data passed to `mkdriver`. The type `commtoken` is the same as the type `commtoken` from the output of the `CCU` functor, and is included simply for the purpose of expressing types.

The type `init` allows for possible initialization of drivers. For example, it could contain configuration information, such as user interface settings. The details are up to the application designer. The function `initialize` performs any required initialization on data in the structure, and is called my `InitFn` before it creates any drivers. The function `initData` generates the initialization data for the drivers. The k-th invocation of `initData` produces the initialization data for the k-th driver.

The function `mkdriver` creates a driver. It takes as parameters a function that starts the driver's instance of the shared object (and produces its `commtoken`), an initial timestamp, and its configuration values (contained in the type `init`). `mkdriver` returns a value of type `driver`. The function `main` takes a `driver` and starts it.

One disadvantage of using the `InitFn` functor is that, because the type `driver` appears as part of the `DRIVER` signature, we fall short of our design goal of being able to have a different

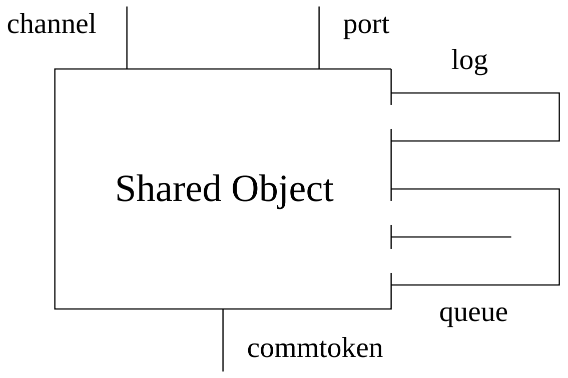


Figure 4.2: Shared object architecture

driver at each site. The best we can do is make the type `driver` a disjoint union type and allow `mkdriver` to choose one of the alternative implementations. We cannot, for example, create two driver structures and have some instances of the shared object associated with one driver and the rest with the other. If we need this level of flexibility, then we must write the initialization code by hand.

4.2.6 Shared Object Architecture

The structure of a shared object is illustrated in Figure 4.2. A shared object communicates with the broadcast network via a multicast channel and a port on that channel. The object writes to the channel and reads from the port. Communication with the driver occurs via a set of channels encapsulated by a `commtoken`. The shared object also maintains a log of previously-applied updates and a queue of pending updates. The log maintains the history of appied updates against which new updates are transformed. The queue holds updates that have not yet been applied because their causal prerequisites have not yet arrived.

The shared object runs in a polling loop. On each iteration, the shared object accepts an update from either the driver or the network, or removes an update from the front of the queue. The update is then processed if its causal prerequisites have been met; otherwise it is placed at the end of the queue. Once an update has been processed (i.e. transformed and applied), it is added to the log.

Notice that this design actually deviates somewhat from the actual CCU algorithm; in the CCU algorithm, local updates are applied immediatetely, but in our design, a shared object may service an update from the network before a local update (the CML `select` and `choose` operators have no notion of priority). However, it is quite easy to show that even under these relaxed conditions, we still get Strong Convergence in the presence of TP2.

4.2.7 Constructing Transformation-Based Systems Using the CCU Library

With the CCU library in place, constructing transformation-based groupware systems is a five step process:

1. Specify the operations, state, and transformations by creating a structure conforming to the `CCUOBJ` signature.
2. Invoke the `CCU` functor on this structure to produce a structure with signature `CCUAPI`.
3. Write a driver structure conforming to the `DRIVER` signature.
4. Invoke the `InitFn` functor on the output of the `CCU` functor and the driver structure.
5. Write a mainline structure to invoke `RunCML.doit` with a function that invokes `init` with the desired number of sites.

In Chapter 5, we will illustrate the use of the CCU library by implementing a shared text buffer.

Chapter 5

Applications

In this chapter, we will demonstrate how to use the framework developed in Chapter 4 to build transformation-based shared objects. In Section 1, we will present the transformation rules for a shared text buffer that supports insertions and deletions, and we will build a shared text buffer using our CCU library. In Section 2, we will consider other examples of transformation-based systems.

5.1 A Shared Text Buffer

Our running example of a shared object throughout this thesis has been that of a shared text buffer supporting insertions and deletions. In this section, we complete our treatement of this example by giving the full set of transformation rules and providing an implementation of a shared text buffer using our CCU library.

5.1.1 Transformation Rules

Recall from Chapter 1 that the shared text buffer supports the following operations:

insert(p, s)—inserts the string s into the buffer at position p, $p \geq 1$;

delete(p, l)—deletes l consecutive characters from the buffer starting at position p, $p \geq 1$.

$$
\begin{aligned}
\text{insert}(p_1, s_1)/\text{insert}(p_2, s_2) &= \begin{cases} \text{insert}(p_1, s_1) & (p_1 < p_2) \\ \text{insert}(p_1 + |s_2|, s_1) & (p_2 \leq p_1) \end{cases} \\
\text{insert}(p_1, s_1)\backslash\text{insert}(p_2, s_2) &= \begin{cases} \text{insert}(p_1, s_1) & (p_1 \leq p_2) \\ \text{insert}(p_1 + |s_2|, s_1) & (p_2 < p_1) \end{cases} \\
\text{delete}(p_1, l_1)/\text{delete}(p_2, l_2) &= \begin{cases} \text{delete}(p_1, l_1) & (p_1 + l_1 \leq p_2) \\ \text{delete}(p_1, p_2 - p_1) & (p_1 \leq p_2 \leq p_1 + l_1 \leq p_2 + l_2) \\ \text{delete}(p_1, l_1 - l_2) & (p_1 \leq p_2 \leq p_2 + l_2 \leq p_1 + l_1) \\ \text{delete}(p_1, 0) & (p_2 \leq p_1 \leq p_1 + l_1 \leq p_2 + l_2) \\ \text{delete}(p_2, p_1 + l_1 - p_2 - l_2) & (p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1) \\ \text{delete}(p_1 - l_2, l_1) & (p_2 + l_2 \leq p_1) \end{cases} \\
\text{delete}(p_1, l_1)\backslash\text{delete}(p_2, l_2) &= \text{delete}(p_1, l_1)/\text{delete}(p_2, l_2) \\
\text{delete}(p_1, l_1)/\text{insert}(p_2, s_2) &= \begin{cases} \text{delete}(p_1, l_1) & (p_1 + l_1 \leq p_2) \\ \text{delete}(p_1, l_1 + |s_2|) & (p_1 \leq p_2 < p_1 + l_1) \\ \text{delete}(p_1 + |s_2|, l_1) & (p_2 < p_1) \end{cases} \\
\text{delete}(p_1, l_1)\backslash\text{insert}(p_2, s_2) &= \text{delete}(p_1, l_1)/\text{insert}(p_2, s_2) \\
\text{insert}(p_1, s_1)/\text{delete}(p_2, l_2) &= \begin{cases} \text{insert}(p_1, s_1) & (p_1 < p_2) \\ \text{insert}(p_1, \text{``''}) & (p_2 \leq p_1 < p_2 + l_2) \\ \text{insert}(p_1 - l_2, s_1) & (p_2 + l_2 \leq p_1) \end{cases} \\
\text{insert}(p_1, s_1)\backslash\text{delete}(p_2, l_2) &= \text{insert}(p_1, s_1)/\text{delete}(p_2, l_2)
\end{aligned}
$$

Figure 5.1: Operation Transforms for Text Buffer Operations

The transformation rules for these operations are given in Figure 5.1[1]. The behaviour of / and \ over insertions is as discussed in Chapter 2. For a deletion d_1 transformed against another deletion d_2, there are three major cases. If d_1 occurs strictly to the left of d_2, it is left unchanged. If d_1 occurs strictly to the right of d_2, then it is translated to the left by the number of characters deleted by d_2. Otherwise the two deletions overlap, and d_1 is transformed so as to delete only those characters not already deleted by d_2.

For a deletion d transformed over an insertion i, if d occurs to the left of i, then d is unchanged.

[1]The transformation rules for the text buffer operations are given in Cormack[3]. We have modified these rules slightly in the interest of compliance with TP2.

If d occurs to the right of i, then d is shifted to the right by the number of characters inserted by i. If the insertion point of i lies within the region to be deleted by d, then the region is expanded to include the inserted characters.

For an insertion i transformed over a deletion d, if i occurs to the left of d, then i is unchanged. If i occurs to the right of d, then the point of insertion of i is shifted to the left by the number of characters deleted by d. If the point of insertion of i occurs within the region deleted by d, then no characters are inserted.

5.1.2 Validating the Text Buffer Transforms

Notice that both insert(p, “”) and delete($p, 0$) are synonyms for NOOP and are therefore the same operation. Thus, we need to be sure that they have the same transformation semantics. First consider insert(p, “”) transformed against insertions. The result is either insert(p, “”) or insert($p+k$, “”) for some k. Either way, the result is NOOP. For an insertion insert(p, s) transformed against insert(p_2, “”), the result is either insert(p, s) or insert($p + |$“”$|, s$) = insert(p, s). For insert(p, “”) transformed against a deletion, the resulting insertion always inserts the string “”, and thus the result is NOOP. For a deletion transformed against insert(p, “”), the result is either the original deletion, or the original deletion with one of the parameters augmented by |“”|, which is equal to 0. Hence, the deletion is unchanged.

Next consider delete($p, 0$) transformed over an insertion. From Figure 5.1, the relevant rule is

$$\text{delete}(p_1, l_1)/\text{insert}(p_2, s_2) = \begin{cases} \text{delete}(p_1, l_1) & (p_1 + l_1 \leq p_2) \\ \text{delete}(p_1, l_1 + |s_2|) & (p_1 \leq p_2 < p_1 + l_1) \\ \text{delete}(p_1 + |s_2|, l_1) & (p_2 < p_1) \end{cases}$$

Since $l_1 = 0$, we see that the condition $p_1 \leq p_2 < p_1 + l_1$ cannot hold, and so the resulting deletion always deletes l_1 (i.e., 0) characters. Thus, the result is NOOP. For an insertion transformed against delete($p, 0$), the relevant rule from Figure 5.1 is

$$\text{insert}(p_1, s_1)/\text{delete}(p_2, l_2) = \begin{cases} \text{insert}(p_1, s_1) & (p_1 < p_2) \\ \text{insert}(p_1, \text{“”}) & (p_2 \leq p_1 < p_2 + l_2) \\ \text{insert}(p_1 - l_2, s_1) & (p_2 + l_2 \leq p_1) \end{cases}$$

Since $l_2 = 0$, the condition $p_2 \leq p_1 < p_2 + l_2$ cannot hold. Hence the resulting insertion is identical to the original. For delete($p, 0$) transformed against a deletion, the relevant rule from Figure 5.1

is

$$\text{delete}(p_1, l_1)/\text{delete}(p_2, l_2) \quad = \quad \begin{cases} \text{delete}(p_1, l_1) & (p_1 + l_1 \leq p_2) \\ \text{delete}(p_1, p_2 - p_1) & (p_1 \leq p_2 \leq p_1 + l_1 \leq p_2 + l_2) \\ \text{delete}(p_1, l_1 - l_2) & (p_1 \leq p_2 \leq p_2 + l_2 \leq p_1 + l_1) \\ \text{delete}(p_1, 0) & (p_2 \leq p_1 \leq p_1 + l_1 \leq p_2 + l_2) \\ \text{delete}(p_2, p_1 + l_1 - p_2 - l_2) & (p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1) \\ \text{delete}(p_1 - l_2, l_1) & (p_2 + l_2 \leq p_1) \end{cases}$$

Under the assumption that $l_1 = 0$, the condition $p_1 \leq p_2 \leq p_1 + l_1 \leq p_2 + l_2$ reduces to $p_1 = p_2$, $p_1 \leq p_2 \leq p_2 + l_2 \leq p_1 + l_1$ reduces to $l_2 = 0$, and $p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1$ reduces to $p_2 + l_2 = p_1$. Thus, we see that in all cases the resulting operation is delete$(k, 0)$ for some k, and this is equivalent to NOOP. For a deletion transformed over delete$(p, 0)$, the relevant rule from Figure 5.1 is the same. Under the assumption that $l_2 = 0$, $p_1 \leq p_2 \leq p_1 + l_1 \leq p_2 + l_2$ reduces to $p_2 = p_1 + l_1$, $p_2 \leq p_1 \leq p_1 + l_1 \leq p_2 + l_2$ reduces to $l_1 = 0$, and $p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1$ reduces to $p_1 = p_2$. Thus, the result is always the original operation.

Thus, insert$(p$, “”$)$ and delete$(p, 0)$ have the same transformation sematics, which is what we set out to verify. Furthermore, we have proved the following theorem:

Theorem 5.1 *The operations in Figure 5.1 satisfy the NOOP Property.*

See Section 3.3.5 for a description of the NOOP Property.

We are now justified in using the representations insert$(p$, “”$)$, delete$(p, 0)$, and NOOP interchangeably. Since we know from Corollary 1 of Theorem 3.13 that TP2 is necessary for the set of text buffer operations, we would also like to know whether the transforms in Figure 5.1 satisfy TP2. In fact they do:

Theorem 5.2 *The operations in Figure 5.1 satisfy TP2.*

Proof The proof is omitted as it is quite long and adds little to the discussion. The interested reader is referred to Appendix A.

5.1.3 CCU Object Specification

We now begin our construction of a shared text buffer application. We will follow the procedure outlined in Section 4.2.7. As noted in this procedure, our first task is to create a structure that outlines the characteristics of the shared state and operations, and defines the transformation semantics. Our implementation of this structure is as follows:

```
structure CCUTextBuf = struct
    type state = char list
    val stateToString = implode

    datatype operation = Insert of int * string | Delete of int * int

    fun operationToString (Insert(n, s)) =
           "Insert(" ^ Int.toString n ^ ",\"" ^ s ^ "\")"
    |   operationToString (Delete(m, n)) =
           "Delete(" ^ Int.toString m ^ "," ^ Int.toString n ^ ")"

    exception BadUpdate

    fun apply(Insert(1,b), x) = (explode b) @ x
    |   apply(Insert(a,b), x::xs) = if a <= 0 then raise BadUpdate
                                    else x :: apply(Insert(a-1,b), xs)
    |   apply(Delete(1,0), x) = x
    |   apply(Delete(1,b), _::xs) = if b < 0 then raise BadUpdate
                                    else apply(Delete(1, b-1), xs)
    |   apply(Delete(a,b), x::xs) = if a <= 0 then raise BadUpdate
                                    else x :: apply(Delete(a-1,b), xs)
    |   apply _ = raise BadUpdate

    nonfix /
    fun / (i as Insert(a, b), Insert(c, d)) =
        if a < c then i else Insert(a + size d, b)
    |   / (Delete(a,b), Delete(c,d)) =
        if a + b <= c then
            Delete(a, b)
        else if a <= c andalso c <= a + b andalso a + b <= c + d then
            Delete(a, c - a)
        else if a <= c andalso c <= c + d andalso c + d <= a + b then
            Delete(a, d - b)
        else if c <= a andalso a <= a + b andalso a + b <= c + d then
            Delete(a, 0)
        else if c <= a andalso a <= c + d andalso c + d <= a + b then
            Delete(c, a + b - c - d)
        else
            Delete(a - d, b)
    |   / (Delete(a,b), Insert(c,d)) =
        if a + b <= c then
            Delete(a, b)
        else if a <= c andalso c < a + b then
            Delete(a, b + size d)
        else
            Delete(a + size d, b)
    |   / (Insert(a,b), Delete(c,d)) =
        if a < c then
            Insert(a, b)
        else if c <= a andalso a < c + d then
            Insert(c, "")
        else
            Insert(a - d, b)

    fun \ (i as Insert(a, b), Insert(c, d)) =
        if a <= c then i else Insert(a + size d, b)
    |   \ x = / x
end
```

Notice that this structure conforms to the signature `CCUOBJ`, specified in Section 4.2.4. We represent the application state as a list of characters. The function `implode` from the ML Standard Basis converts character lists to strings, so we use this function as our definition of `stateToString`. Our operation type is a tagged pair representing either an insertion or a deletion. The function `operationToString` converts values of type `operation` to strings, and is straightforward. The function `apply` provides the semantics of values of type `operation`; it defines what it means to apply an insertion or a deletion to a list of characters. Finally, the functions `/` and `\` implement the transformation rules on operations.

We have not indicated conformance to `CCUOBJ` in the definition of our `CCUTextBuf` structure (that is, we did not write `structure CCUTextBuf : CCUOBJ = struct` . . .). This omission was intentional; we want to export the constructors `Insert` and `Delete`, which are not contained in `CCUOBJ`.

Our structure also illustrates a benefit of using a type to represent operations. By using ML's `datatype` construction, we can make our operations resemble actual functions, even though they are simply tagged pairs. Furthermore, we can use ML's pattern-matching facilities when defining / and \.

5.1.4 Driver

Next we will write a driver specification for our shared text buffer. Recall that the purpose of the driver is to act as an interface between the user at a site and that site's instance of the shared object. However, since we are running on a simulated network at a single site, our options for implementing communication between drivers and users are somewhat limited. Consequently, we will write drivers that issue updates based on script files.

Our driver will consist of two structures: a structure `Driver` that contains the interface to the shared object, and a structure `GetCommands` that reads commands out of a script file. We will present only `Driver` here; the interested reader can find the source for `GetCommands` in Appendix B.

The `Driver` structure is as follows:

```
structure Driver: DRIVER = struct
   structure TS = Timestamp
   structure C = GetCommands
   structure TB = TextBuf

   type operation = TB.operation
   datatype message = datatype TB.message
   type commtoken = TB.commtoken
   exception Done of TS.timestamp

   type init = string
```

```
(* counter to properly assign input files *)
val x = ref 1

fun initialize () = x := 1

fun initData () = ("dr" ^ Int.toString(!x) ^ ".cmd") before x := !x + 1

abstype driver = Driver of (unit->commtoken) * TS.timestamp * string
with
    fun mkdriver f t s  = Driver (f, t, s)
    fun main (Driver (f, t, s)) =
       let
          val comStream = C.mkCommandStream s
          val commToken = f ()

          fun mainLoop ts =
             let
                val c = C.getCommand comStream
             in
                TextIO.print (s ^ "\n");
                case c of
                   C.Delay x => (
                     CML.sync (
                        CML.timeOutEvt (
                           Time.fromMicroseconds (Int32.fromInt x)
                        )
                     );
                     mainLoop ts
                   )
                |  C.Operation x => mainLoop (
                        TB.update(commToken, MSG(x, ts))
                        handle _ => raise Done(ts)
```

```
                    )
                 |  C.Quit => raise Done ts
               end
               handle C.EOF => raise Done ts
           in
              mainLoop t
              handle Done ts => (TB.send(commToken, QUIT ts);
                                 C.closeCommandStream comStream)
           end
   end
end
```

The driver recognizes three types of commands: operations (i.e. insertions and deletions), delay directives, and quit directives. These types of commands are encapsulated in the type `command`, which is defined in the structure `GetCommands` as follows:

```
datatype command = Operation of operation | Delay of int | Quit
```

`GetCommands` also defines a type `commandstream` to represent a source of commands from the script file, as well as functions to intialize, close, and retrieve commands from a `commandstream`.

The driver has been written to conform to the `DRIVER` signature, so that we may use the `InitFn` functor (see Section 4.2.5). The type `driver` is simply an encapsulation of the three parameters of `mkdriver`: a function to start the shared object instance, an initial timestamp, and an auxiliary initialization value of type `init`. In our implementation, `init` is a synonym for `string`; we are initializing the driver with the name of the script file. The k-th instance of the driver will read from the file `dr`k`.cmd`, and the function `initData` will return this string upon its k-th invocation (the functor `InitFn` will generate the code to invoke `initData` once for each driver we wish to create).

The function `main` starts the driver. It initializes the `commandstream` and starts the shared objects. It then invokes `mainloop` which recurses on itself until it process a quit directive or reaches the end of the script file. At this point, it raises the exception `Done`, and `main` signals the shared object to quit and closes the `commandstream`.

During the operation of `mainloop`, the driver repeatedly gets the next command and processes it. If the command is a delay directive, then the the driver synchronizes on a timeout event that becomes ready after the specified delay interval has expired. If the command is an operation, then the driver issues the update, retrieves the new timestamp, and recurses. If the command is a quit directive, then the driver raises the exception `Done`.

Notice that the `Driver` structure refers to a structure called `TextBuf`, which we have not yet defined. This structure is the shared object instance, and has signature `CCUAPI`. We create this structure next.

5.1.5 Completing the Implementation

To complete our implementation of a transformation-based shared text buffer, we need three more structures. We first create the actual shared object instance structure from our `CCUTextBuf` structure. We do this by invoking our `CCU` functor:

```
structure TextBuf = CCU(structure ccuobj = CCUTextBuf)
```

Next, we create the initialization code for our system, by invoking our `InitFn` functor:

```
structure Init = InitFn(structure Obj = TextBuf; structure D = Driver)
```

Finally, we create a mainline structure to set everything in motion:

```
structure Mainline: sig
   val main : (int * string) -> unit
end
= struct
    fun main (n, initState) = ignore (
        RunCML.doit(fn () => Init.init(n,explode initState), NONE)
    )
end
```

The structure `Mainline` exports a single function, called `main`. `main` takes as arguments an integer representing the number of sites and a string representing the initial state of the text buffer. `main` then starts up the CML environment and invokes the initialization code in `Init`, which in turn starts the system. `main` returns when (if) the entire system terminates.

5.1.6 A Sample Run

In this section, we will illustrate our shared text buffer's behaviour by running it on some sample input. The input data takes the form of script files for the drivers. We will run our system with three sites; therefore, we will need three script files, called `dr1.cmd`, `dr2.cmd`, and `dr3.cmd`. `dr1.cmd` is as follows:

```
i 1 a
q
```

This script instructs site 1 to insert the string "a" at position 1, and then quit. `dr2.cmd` and `dr3.cmd` are similar, except that site 2 is instructed to insert "b" at position 1 and site 3 is instructed to insert "c" at position 1.

Having set up our script files, we now start our system as follows:

```
Mainline.main(3,"");
```

This invocation of `main` initializes our text buffer with three sites and the empty string as the initial state. Once `main` has been called, the system runs until all sites have quit.

The CCU library contains a debugging module, which, if enabled, allows us to examine the progress and outcome of the shared computation. With debugging enabled, each shared object writes its actions to the screen and to a log file. Each site has its own log file; site n uses the file `site`n`.log` as its log file. Figure 5.2 contains a sample of the messages that are printed to the screen. For the sake of clarity, only messages from Site 2 have been reproduced here; in the real system, messages from Sites 1 and 3 would be interleaved.

Each log file ends with a line of the form `Final state is x`. Thus, to see that our system has performed correctly, we simply execute

```
grep Final *.log
```

Our call to `grep` produces the following as output:

```
site1.log:Final state is cba.
site2.log:Final state is cba.
site3.log:Final state is cba.
```

Here we see that the three sites have converged on a common state, which is the desired outcome.

```
Creating site 2 with initial state
Received local update Insert(1,"b") with timestamp (0,0,0).
Transformed to Insert(1,"b").
New state is b.
Received local QUIT message with timestamp (0,1,0).
Site 2 received NQUIT message from site 2 with timestamp (0,1,0).
Site 2 received message Insert(1,"a") from site 1 with timestamp (0,0,0).
Transformed to Insert(2,"a").
New timestamp is (1,1,0).
New state is ba.
Site 2 received NQUIT message from site 1 with timestamp (1,0,0).
Site 2 received message Insert(1,"c") from site 3 with timestamp (0,0,0).
Transformed to Insert(1,"c").
New timestamp is (1,1,1).
New state is cba.
Site 2 received NQUIT message from site 3 with timestamp (0,0,1).
Site 2 terminating.
Final state is cba.
```

Figure 5.2: Sample output from the shared text buffer.

5.2 Other Transformation-Based Objects

In this section, we will consider other kinds of shared objects, upon which we could build a transformation-based system.

Our first observation is that, although we represented the shared text buffer as a list of characters, nothing in our analysis or our implementation fundamentally relies on the fact that the elements of the list are characters. Therefore, we can use the transformation rules we currently have for any list-based state.

For example, we could represent a shared presentation as a list of slides. Insertion and deletion of slides can be governed by our current set of insertion and deletion rules. Since we already know that TP2 holds on these rules (see Appendix A), we do not have to do any of our own verification.

Concurrent edits on a single slide could be handled in several ways. The simplest approach is to treat each slide as an atomic entity and disallow concurrent edits. Slides would be created at the driver level and then uploaded to the shared state when they are complete. This approach most closely models the shared text buffer example, in which characters (which can only be inserted or deleted, but not edited) take the place of slides; however, it offers the least opportunity for collaboration. A more reasonable approach is to model each slide as a sequence of design elements (e.g. text boxes, drawings, bulletted lists). Then we can again use the standard insertion and deletion techniques, but this time at the level of a single slide. Furthermore, if desired, we could allow concurrent editing of a single design element within a slide, e.g. a text box. Again, we can use the standard insertion and deletion rules at this finer level of detail. We can continue pushing the insertion and deletion rules down to finer and finer granularities, until we achieve the flexibility we feel we need. Verification of TP2 on this system should be straightforward.

Next, we observe that the shared text buffer itself could be used in more abstract ways. Many real-world objects are easily modeled by text. Using markup languages like XML[8], we can encode anything from purchase orders to building plans as structured text. Complex edits on these documents can be expressed as sequences of insertions and deletions on the underlying text. Because of Theorem 3.14, we can get TP2 for free.

Text buffers and abstract lists are good for modelling state that is linear in nature. For non-linear data, we may desire some other underlying representation. For example, for bitmap images, we might like some state representation capable of representing two dimensions. One obvious approach to representing two-dimensional data is to use a list of lists. A more direct approach was suggested by Palmer and Cormack[16], who give a set of operations and transforms for a shared spreadsheet. Compliance of these operations with TP2 is still an open question, and is left to future investigation.

Chapter 6

Conclusions and Future Work

In this chapter, we provide concluding remarks and indicate possible topics of further investigation.

6.1 Summary

This thesis explores the problem of maintaining a consistent shared state in groupware systems. Rather than employ traditional locking mechanisms, we consider a transformation-based approach that creates the illusion of a common execution history across all sites in the system. Despite the fact that this technique has been around since 1989[6], the underlying theory has not received much attention. In this thesis, we develop a formal treatment of operation transforms based on Ressel's interaction models. We then used this theory to build a provably correct framework for constructing transformation-based systems.

Our specific contributions in this thesis are as follows:

- We define correctness criteria for groupware systems that are more generally applicable than the original criteria of Ellis and Gibbs.
- We give a formal treatment of the theory of operation transforms. Using Ressel's interaction models as a basis for our reasoning, we show equivalence between the CCU algorithm and the adOPTed algorithm. Further, we show that with Ressel's TP2 as an additional precondition, Hendrie's counterexample to the modified CCU algorithm disappears and the modified CCU algorithm becomes correct.

- We show that TP2 is a necessary condition for correctness in several important cases and conjecture that it is necessary in general. Finally, we give a few techniques for simplifying the task of verifying TP2 on a set of transforms.
- We show that the canonical text buffer operations satisfy TP2.
- We construct a framework for building transformation-based groupware systems. This framework is an implementation of the theory developed in Chapter 3 and as such, is provably correct.
- We demonstrate the use of our framework by implementing a shared text buffer.

6.2 Future Work

In the sections that follow, we outline some areas for further research that would augment our current study.

6.2.1 Necessity of TP2

In Section 3.3.4, we showed that, in several instances, TP2 is a necessary condition for Strong Convergence, and conjectured that it is a necessary condition in general. To complete the theory of operation transforms, we would like to settle this issue, with either a proof of necessity or an example of a strongly-convergent system on more than two sites that does not satisfy TP2. Based on the necessity results that we already have, we expect that a counterexample, if one exists, would be highly unintuitive.

6.2.2 Deployment on a Real Network

To complete our CCU library, we need to remove the simulated network module and deploy the system on an actual network. Before we can deploy to a real network we need to address two issues. First, sockets in SML/NJ currently cannot carry arbitrary data types; they can only carry byte vectors. Thus, in order to relay messages over sites, we need to marshall them. Marshalling site IDs and timestamps is not difficult, but we must rely on the application designer to supply the marshalling and unmarshalling routines for operations. Thus, we must extend the `CCUOBJ` signature with two functions, `marshall` and `unmarshall` of type `operation -> string` and `string -> operation`, respectively.

The second issue a "real" implementation must address is deployment. The entire system compiles on a single machine, but each instance of the shared state is supposed to run at a different physical location. A straightforward approach to the deployment problem might be to make SML/NJ generate a separate heap image for each site. We could then manually install each instance at its respective site. Clearly, we would prefer a solution that required less human intervention. To circumvent the manual installation problem would probably require installing a "listener" process at each potential site that can receive heap images over a network and install them. Under this scenario, we would still have to install the listener manually, but the same listener process could then be used to install multiple groupware systems.

6.2.3 A Dynamic Set of Participants

All of the theory and implementation in this thesis has relied on the assumption that the set of participants in the system is static. In reality, we would like to allow participants to enter and exit the collaboration process throughout its lifetime.

The assumption of a dynamic set of participants has no real effect on the theory. From a theoretical point of view, a participant who quits is no different from a participant who remains on line but ceases to issue updates. Similarly, a new participant is no different from a participant who was always there but has only just begun to issue updates. Thus, a correct system with a static set of participants will continue to be correct if we make the set of participants dynamic.

However, a proper *implementation* of a system with a dynamic set of participants does cause some difficulty. A major problem is the creation of unambiguous timestamps. Suppose we provide a primitive called `fork` that a site can issue to create a new instance of the shared object at a specified location. Now suppose, in an n-site system, site k calls `fork`. The new site k' needs its own component of the timestamp in which to operate, and so we need to expand all of our timestamps by one component. We then assign the $(n+1)$-st component to the new site. However, suppose that site l has concurrently called `fork`. The new site l' would also be assigned component $n+1$. Thus, sites hearing about site k' first would assign it component $n+1$ and site l' component $n+2$, while sites hearing about site l' first would do the reverse. Further, sites k' and l' themselves will both think that the $(n+1)$-st component is theirs.

A possible solution to this problem might be to have an independent arbiter assign timestamp components. However, this solution poses all of the problems that we noted in Chapter 1 with respect to lock servers. We believe a more promising solution would be to restructure the timestamps. Instead of representing timestamps as tuples of integers, we might represent them as trees of integers. Then, when sites k and l create new sites k' and l', respectively, site k''s component will be a child of the k-th component of the original timestamp, while site l''s component will be a child of the l-th component. In this way, we can assign timestamps unambiguously.

6.2.4 Checkpoints

In both the CCU and adOPTed algorithms, each site maintains a log of previously-applied updates. These logs provide the execution history against which incoming updates are transformed. As the collaborative effort progresses, the update logs grow. Enforcing a checkpoint protocol, under which we could draw periodic conclusions about the progress of the entire system would allow us to remove old entries from the log. If it is somehow known that at some instant during the collaborative process, each site has a timestamp equal to at least $(a_1, \ldots, a_n)$, then any newly issued update will causally succeed the first a_i updates from site i, for each i. Hence, the newly issued update cannot be concurrent with any of these updates and will never be transformed against them. Therefore, we can flush these old updates from the log. Note that knowledge about the minimum timestamp across all sites in the system implies knowledge of the complete set of participants. Thus, reconciling the need for checkpoints with the need for a dynamic set of participants may prove difficult.

6.2.5 Time and Space Complexity

The CCU algorithm computes transforms via the $|$ operator, whose definition is doubly recursive. Thus, the cost of computing transforms under CCU is potentially exponential in the number of updates in the log. Ressel addresses this issue in his adOPTed algorithm. The adOPTed algorithm, as it appears in Ressel[18], memoizes the result every time it computes a transform. In effect, adOPTed realizes the interaction model of the system as a data structure in memory at each site. Thus, previously computed transforms are not recomputed, as they are in CCU. However, the size of the interaction model is, in the worst case, on the order of m^n, where m is the number of updates in the log and n is the number of sites. In practice, we do not believe that the worst case is likely to materialize unless all sites are feverishly active at once, but further study is required to test our intuituion.

With a checkpoint system in place, we would be able to discard older portions of the interaction model, just as we could discard older log entries. We believe that, with the resulting space savings, computing transforms would become tractable for large numbers of sites, but a proper analysis of the complexity of the system under memoization and a study of what constitutes “typical” activity of groupware users are needed before we can draw any real conclusions. While it may turn out that the number of sites that a transformation-based system can handle before the exponential nature of computing transforms renders the system impractically slow is limited, we believe that if this limit exists, it is probably high enough to support most real-world demands.

Appendix A

Verification of TP2 for Text Buffer Operations

In this appendix, we prove Theorem 5.2; that is, we show that the text buffer operations defined in Figure 5.1 (Section 5.1.1) satisfy TP2. In Corollary 1 of Theorem 3.13, we showed that for the text buffer operations, TP2 is a necessary condition for Strong Convergence. Recall that the text buffer operations take the following form:

insert(p, s)—inserts the string s into the buffer at position p, $p \geq 1$;
delete(p, l)—deletes l consecutive characters from the buffer starting at position p, $p \geq 1$.

We can rephrase these operations as compositions of the following operations:

ins(p, c)—inserts the character c into the buffer at position p, $p \geq 1$;
del(p, l)—deletes l consecutive characters from the buffer starting at position p, $p \geq 1$, $l \geq 1$;
NOOP—does nothing[1].

Since TP2 is known to be necessary on the set of text buffer operations, by Theorem 3.14, we can verify TP2 on the original operations by verifying TP2 on this new set, which is somewhat simpler (insertions are all of length 1). Notice that we cannot restrict deletions to single

[1]Our replacement of delete$(p, 0)$ with NOOP is justified by Theorem 5.1. However, for convenience, we will allow del$(p, 0)$ to act as a synonym for NOOP.

characters. If a deletion occurs at the same posistion as an insertion, then the deletion must be transformed to remove both the character to be deleted and the character just inserted. Therefore, the set of single-character insertions and deletions would not be closed under ˆ.

This adjustment of the set of operations leaves the transformation rules largely unchanged. The major difference is that all references to the length of the string parameter of insertions become 1 (except in the case of insert(p, “”), which becomes NOOP). Thus, for example, we now have

$$\begin{aligned}
\text{ins}(p_1, c_1)/\text{ins}(p_2, c_2) &= \begin{cases} \text{ins}(p_1, c_1) & (p_1 < p_2) \\ \text{ins}(p_1+1, c_1) & (p_2 \le p_1) \end{cases} \\
\text{ins}(p_1, c_1)\backslash\text{ins}(p_2, c_2) &= \begin{cases} \text{ins}(p_1, c_1) & (p_1 \le p_2) \\ \text{ins}(p_1+1, c_1) & (p_2 < p_1) \end{cases}
\end{aligned}$$

We now proceed with our verification of TP2. Let a, b, and c be updates[2] with $T(a) < T(b) < T(c)$. There are eight cases to consider:

Case 1 $a = \mathit{ins}(p_1, c_1)$*,* $b = \mathit{ins}(p_2, c_2)$*,* $c = \mathit{ins}(p_3, c_3)$*.*

We must show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$*:* Based on the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash c)\backslash(b\backslash c)$. We then have

$$\begin{aligned}
(a\backslash b)\backslash(c/b) &= (\text{ins}(p_1, c_1)\backslash\text{ins}(p_2, c_2))\backslash(\text{ins}(p_3, c_3)/\text{ins}(p_2, c_2)) \\
&= \begin{pmatrix} \text{ins}(p_1, c_1) & (p_1 \le p_2) \\ \text{ins}(p_1+1, c_1) & (p_2 < p_1) \end{pmatrix} \Bigg\backslash \begin{pmatrix} \text{ins}(p_3, c_3) & (p_3 < p_2) \\ \text{ins}(p_3+1, c_3) & (p_2 \le p_3) \end{pmatrix},
\end{aligned}$$

which evaluates as follows:

	$p_3 < p_2$	$p_3 \ge p_2$
$p_1 \le p_2$	$\text{ins}(p_1, c_1)\backslash\text{ins}(p_3, c_3)$	$\text{ins}(p_1, c_1)\backslash\text{ins}(p_3+1, c_3)$
$p_1 > p_2$	$\text{ins}(p_1+1, c_1)\backslash\text{ins}(p_3, c_3)$	$\text{ins}(p_1+1, c_1)\backslash\text{ins}(p_3+1, c_3)$

[2] In order to avoid the confusion of excess notation, we will relax the distinction between updates and operations. In particular, we will allow statements like $a = \text{ins}(p_1, c_1)$ that equate an update with an operation. Further, we will allow / and \ to be applied to updates, even though they were defined over operations.

$=$

	$p_3 < p_2$	$p_3 \geq p_2$
$p_1 \leq p_2$	$\mathrm{ins}(p_1, c_1) \quad (p_1 \leq p_3)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_3)$	$\mathrm{ins}(p_1, c_1) \quad (p_1 \leq p_3+1)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_3+1)$
$p_1 > p_2$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1+1 < p_3)$ $\mathrm{ins}(p_1+2, c_1) \quad (p_1+1 \geq p_3)$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1+1 \leq p_3+1)$ $\mathrm{ins}(p_1+2, c_1) \quad (p_1+1 > p_3+1)$

$=$

	$p_3 < p_2$	$p_3 \geq p_2$
$p_1 \leq p_2$	$\mathrm{ins}(p_1, c_1) \quad (p1 \leq p_3)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_3)$	$\mathrm{ins}(p_1, c_1)$
$p_1 > p_2$	$\mathrm{ins}(p_1+2, c_1)$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1 \leq p_3)$ $\mathrm{ins}(p_1+2, c_1) \quad (p_1 > p_3)$

.

Also, we have

$$\begin{aligned}(a\backslash c)\backslash(b\backslash c) &= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3))\backslash(\mathrm{ins}(p_2,c_2)\backslash\mathrm{ins}(p_3,c_3))\\ &= \begin{pmatrix}\mathrm{ins}(p_1,c_1) & (p_1 \leq p_3)\\ \mathrm{ins}(p_1+1,c_1) & (p_3 < p_1)\end{pmatrix}\Bigg\backslash\begin{pmatrix}\mathrm{ins}(p_2,c_2) & (p_2 \leq p_3)\\ \mathrm{ins}(p_2+1,c_2) & (p_3 < p_2)\end{pmatrix},\end{aligned}$$

which evaluates as follows:

	$p_2 \leq p_3$	$p_3 < p_2$
$p_1 \leq p_3$	$\mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_2, c_2)$	$\mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_2+1, c_2)$
$p_3 < p_1$	$\mathrm{ins}(p_1+1, c_1)\backslash\mathrm{ins}(p_2, c_2)$	$\mathrm{ins}(p_1+1, c_1)\backslash\mathrm{ins}(p_2+1, c_2)$

$=$

	$p_2 \leq p_3$	$p_3 < p_2$
$p_1 \leq p_3$	$\mathrm{ins}(p_1, c_1) \quad (p_1 \leq p_2)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_2)$	$\mathrm{ins}(p_1, c_1) \quad (p_1 \leq p_2+1)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_2+1)$
$p_3 < p_1$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1+1 \leq p_2)$ $\mathrm{ins}(p_1+2, c_1) \quad (p_1+1 > p_2)$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1+1 \leq p_2+1)$ $\mathrm{ins}(p_1+2, c_1) \quad (p_1+1 > p_2+1)$

$=$

	$p_2 \leq p_3$	$p_3 < p_2$
$p_1 \leq p_3$	$\mathrm{ins}(p_1, c_1) \quad (p_1 \leq p_2)$ $\mathrm{ins}(p_1+1, c_1) \quad (p_1 > p_2)$	$\mathrm{ins}(p_1, c_1)$
$p_3 < p_1$	$\mathrm{ins}(p_1+2, c_1)$	$\mathrm{ins}(p_1+1, c_1) \quad (p_1 \leq p_2)$ $\mathrm{insert}(p_1+2, c_1) \quad (p_1 > p_2)$

.

Comparing this table with the previous one, we see that they are equivalent. Hence $(a\backslash b)\backslash(c/b) = (a\backslash c)\backslash(b\backslash c)$, i.e. $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$: By the total ordering of a, b, and c, we have $(b\hat{}a)\hat{}(c\hat{}a) = (b/a)\backslash(c/a)$ and $(b\hat{}c)\hat{}(a\hat{}c) = (b\backslash c)/(a\backslash c)$. We then have

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{ins}(p_2,c_2)/\mathrm{ins}(p_1,c_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1)) \\ &= \left(\begin{array}{ll}\mathrm{ins}(p_2,c_2) & (p_2 < p_1) \\ \mathrm{ins}(p_2+1,c_2) & (p_2 \geq p_1)\end{array}\right)\Big\backslash\left(\begin{array}{ll}\mathrm{ins}(p_3,c_3) & (p_3 < p_1) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_1)\end{array}\right),\end{aligned}$$

which evaluates as follows:

	$p_3 < p_1$	$p_3 \geq p_1$
$p_2 < p_1$	$\mathrm{ins}(p_2,c_2)\backslash\mathrm{ins}(p_3,c_3)$	$\mathrm{ins}(p_2,c_2)\backslash\mathrm{ins}(p_3+1,c_3)$
$p_2 \geq p_1$	$\mathrm{ins}(p_2+1,c_2)\backslash\mathrm{ins}(p_3,c_3)$	$\mathrm{ins}(p_2+1,c_2)\backslash\mathrm{ins}(p_3+1,c_3)$

$=$

	$p_3 < p_1$	$p_3 \geq p_1$
$p_2 < p_1$	$\mathrm{ins}(p_2,c_2) \quad (p_2 \leq p_3)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 > p_3)$	$\mathrm{ins}(p_2,c_2) \quad (p_2 \leq p_3+1)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 > p_3+1)$
$p_2 \geq p_1$	$\mathrm{ins}(p_2+1,c_2) \quad (p_2+1 \leq p_3)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2+1 > p_3)$	$\mathrm{ins}(p_1+1,c_1) \quad (p_2+1 \leq p_3+1)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2+1 > p_3+1)$

$=$

	$p_3 < p_1$	$p_3 \geq p_1$
$p_2 < p_1$	$\mathrm{ins}(p_2,c_2) \quad (p_2 \leq p_3)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 > p_3)$	$\mathrm{ins}(p_2,c_2)$
$p_2 \geq p_1$	$\mathrm{ins}(p_2+2,c_2)$	$\mathrm{ins}(p_1+1,c_2) \quad (p_2 \leq p_3)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2 > p_3)$

.

Also, we have

$$\begin{aligned}(b\backslash c)/(a\backslash c) &= (\mathrm{ins}(p_2,c_2)\backslash\mathrm{ins}(p_3,c_3))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3)) \\ &= \left(\begin{array}{ll}\mathrm{ins}(p_2,c_2) & (p_2 \leq p_3) \\ \mathrm{ins}(p_2+1,c_2) & (p_2 > p_3)\end{array}\right)\Big/\left(\begin{array}{ll}\mathrm{ins}(p_1,c_1) & (p_1 \leq p_3) \\ \mathrm{ins}(p_1+1,c_1) & (p_1 > p_3)\end{array}\right),\end{aligned}$$

which evaluates as follows:

	$p_1 \leq p_3$	$p_1 > p_3$
$p_2 \leq p_3$	$\mathrm{ins}(p_2,c_2)/\mathrm{ins}(p_1,c_1)$	$\mathrm{ins}(p_2,c_2)/\mathrm{ins}(p_1+1,c_1)$
$p_2 > p_3$	$\mathrm{ins}(p_2+1,c_2)/\mathrm{ins}(p_1,c_1)$	$\mathrm{ins}(p_2+1,c_2)/\mathrm{ins}(p_1+1,c_1)$

$=$

	$p_1 \leq p_3$	$p_1 > p_3$
$p_2 \leq p_3$	$\mathrm{ins}(p_2,c_2) \quad (p_2 < p_1)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 \geq p_1)$	$\mathrm{ins}(p_2,c_2) \quad (p_2 < p_1+1)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 \geq p_1+1)$
$p_2 > p_3$	$\mathrm{ins}(p_2+1,c_2) \quad (p_2+1 < p_1)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2+1 \geq p_1)$	$\mathrm{ins}(p_2+1,s_2) \quad (p_2+1 < p_1+1)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2+1 \geq p_1+1)$

$=$

	$p_1 \leq p_3$	$p_1 > p_3$
$p_2 \leq p_3$	$\mathrm{ins}(p_2,c_2) \quad (p_2 < p_1)$ $\mathrm{ins}(p_2+1,c_2) \quad (p_2 \geq p_1)$	$\mathrm{ins}(p_2,c_2)$
$p_2 > p_3$	$\mathrm{ins}(p_2+2,c_2)$	$\mathrm{ins}(p_2+1,c_2) \quad (p_2 < p_1)$ $\mathrm{ins}(p_2+2,c_2) \quad (p_2 \geq p_1)$

.

Comparing this table with the previous one, we see that they are equivalent. Hence, $(b/a)\backslash(c/a) = (b\backslash c)/(c\backslash a)$, i.e. $(b\hat{\ }a)\hat{\ }(c\hat{\ }a)$.

$(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$*:* By the total ordering of a, b, and c, we have $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c/a)/(b/a)$ and $(c\hat{\ }b)\hat{\ }(a\hat{\ }b) = (c/b)/(a\backslash b)$. We then have

$$\begin{aligned}(c/a)/(b/a) &= (\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1))/(\mathrm{ins}(p_2,c_2)/\mathrm{ins}(p_1,c_1)) \\ &= \begin{pmatrix}\mathrm{ins}(p_3,c_3) & (p_3 < p_1) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_1)\end{pmatrix} \Big/ \begin{pmatrix}\mathrm{ins}(p_2,c_2) & (p_2 < p_1) \\ \mathrm{ins}(p_2+1,c_2) & (p_2 \geq p_1)\end{pmatrix},\end{aligned}$$

which evaluates as follows:

	$p_2 < p_1$	$p_2 \geq p_1$
$p_3 < p_1$	$\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_2,c_2)$	$\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_2+1,c_2)$
$p_3 \geq p_1$	$\mathrm{ins}(p_3+1,c_3)/\mathrm{ins}(p_2,c_2)$	$\mathrm{ins}(p_3+1,c_3)/\mathrm{ins}(p_2+1,c_2)$

=

	$p_2 < p_1$	$p_2 \geq p_1$
$p_3 < p_1$	$\begin{matrix}\mathrm{ins}(p_3,c_3) & (p_3 < p_2) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_2)\end{matrix}$	$\begin{matrix}\mathrm{ins}(p_3,c_3) & (p_3 < p_2+1) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_2+1)\end{matrix}$
$p_3 \geq p_1$	$\begin{matrix}\mathrm{ins}(p_3+1,c_3) & (p_3+1 < p_2) \\ \mathrm{ins}(p_3+2,c_3) & (p_3+1 \geq p_2)\end{matrix}$	$\begin{matrix}\mathrm{ins}(p_3+1,c_3) & (p_3+1 < p_2+1) \\ \mathrm{ins}(p_3+2,s_3) & (p_3+1 \geq p_2+1)\end{matrix}$

=

	$p_2 < p_1$	$p_2 \geq p_1$
$p_3 < p_1$	$\begin{matrix}\mathrm{ins}(p_3,c_3) & (p_3 < p_2) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_2)\end{matrix}$	$\mathrm{ins}(p_3,c_3)$
$p_3 \geq p_1$	$\mathrm{ins}(p_3+2,c_3)$	$\begin{matrix}\mathrm{ins}(p_3+1,c_3) & (p_3 < p_2) \\ \mathrm{ins}(p_3+2,c_3) & (p_3 \geq p_2)\end{matrix}$

.

Also, we have

$$\begin{aligned}(c/b)/(a\backslash b) &= (\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_2,c_2))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_2,c_2)) \\ &= \begin{pmatrix}\mathrm{ins}(p_3,c_3) & (p_3 < p_2) \\ \mathrm{ins}(p_3+1,c_3) & (p_3 \geq p_2)\end{pmatrix} \Big/ \begin{pmatrix}\mathrm{ins}(p_1,c_1) & (p_1 \leq p_2) \\ \mathrm{ins}(p_1+1,c_1) & (p_1 > p_2)\end{pmatrix},\end{aligned}$$

which evaluates as follows:

	$p_1 \le p_2$	$p_1 > p_2$
$p_3 < p_2$	$\text{ins}(p_3, c_3)/\text{ins}(p_1, c_1)$	$\text{ins}(p_3, c_3)/\text{ins}(p_1+1, c_1)$
$p_3 \ge p_2$	$\text{ins}(p_3+1, c_3)/\text{ins}(p_1, c_1)$	$\text{ins}(p_3+1, c_3)/\text{ins}(p_1+1, c_1)$

=

	$p_1 \le p_2$	$p_1 > p_2$
$p_3 < p_2$	$\text{ins}(p_3, c_3) \quad (p_3 < p_1)$ $\text{ins}(p_3+1, c_3) \quad (p_3 \ge p_1)$	$\text{ins}(p_3, c_3) \quad (p_3 < p_1+1)$ $\text{ins}(p_3+1, c_3) \quad (p_3 \ge p_1+1)$
$p_3 \ge p_2$	$\text{ins}(p_3+1, c_3) \quad (p_3+1 < p_1)$ $\text{ins}(p_3+2, c_3) \quad (p_3+1 \ge p_1)$	$\text{ins}(p_3+1, c_3) \quad (p_3+1 < p_1+1)$ $\text{ins}(p_3+2, c_3) \quad (p_3+1 \ge p_1+1)$

=

	$p_1 \le p_2$	$p_1 > p_2$
$p_3 < p_2$	$\text{ins}(p_3, c_3) \quad (p_3 < p_1)$ $\text{ins}(p_3+1, c_3) \quad (p_3 \ge p_1)$	$\text{ins}(p_3, c_3)$
$p_3 \ge p_2$	$\text{ins}(p_3+2, c_3)$	$\text{ins}(p_3+1, c_3) \quad (p_3 < p_1)$ $\text{ins}(p_3+2, c_3) \quad (p_3 \ge p_1)$

.

Comparing this table with the previous one, we see that they are equivalent. Hence, $(c/a)/(b/a) = (c/b)/(a\backslash b)$, i.e., $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$. This completes Case 1.

Case 2 *$a = del(p_1, l_1)$, $b = ins(p_2, c_2)$, $c = ins(p_3, c_3)$.*

For this case, and the next two, we will treat the case that the insertions operate at the same position as separate from the general case. The advantage of this approach is that, in the general case, / and \ over two insertions can be treated as identical transforms (since they differ only when the insertion points are the same, which we assume is not the case). The result is greater opportunity to reuse some of our work. We must show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$: By the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash c)\backslash(b\backslash c)$. We have

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\text{del}(p_1, l_1)\backslash \text{ins}(p_2, c_2))\backslash(\text{ins}(p_3, c_3)/\text{ins}(p_2, c_2)),\\ (a\backslash c)\backslash(b\backslash c) &= (\text{del}(p_1, l_1)\backslash \text{ins}(p_3, c_3))\backslash(\text{ins}(p_2, c_2)\backslash \text{ins}(p_3, c_3)).\end{aligned}$$

Let us first assume that $p_2 = p_3$. Then we have

$$\begin{aligned}
(a\backslash b)\backslash(c/b) &= \left(\begin{array}{c} \mathrm{del}(p_1,l_1) \quad (p_1+l_1 \le p_2) \\ \mathrm{del}(p_1,l_1+1) \quad (p_1 \le p_2 < p_1+l_1) \\ \mathrm{del}(p_1+1,l_1) \quad (p_2 < p_1) \end{array}\right) \Big\backslash \mathrm{ins}(p_3+1,c_3) \\
&= \begin{cases} \mathrm{del}(p_1,l_1)\backslash\mathrm{ins}(p_3+1,c_3) & (p_1+l_1 \le p_2) \\ \mathrm{del}(p_1,l_1+1)\backslash\mathrm{ins}(p_3+1,c_3) & (p_1 \le p_2 < p_1+l_1) \\ \mathrm{del}(p_1+1,l_1)\backslash\mathrm{ins}(p_3+1,c_3) & (p_2 < p_1) \end{cases} \\
&= \begin{cases} \mathrm{del}(p_1,l_1) & (p_1+l_1 \le p_2) \\ \mathrm{del}(p_1,l_1+2) & (p_1 \le p_2 < p_1+l_1) \\ \mathrm{del}(p_1+2,l_1) & (p_2 < p_1) \end{cases} .
\end{aligned}$$

Also,

$$\begin{aligned}
(a\backslash c)\backslash(b\backslash c) &= \left(\begin{array}{c} \mathrm{del}(p_1,l_1) \quad (p_1+l_1 \le p_3) \\ \mathrm{del}(p_1,l_1+1) \quad (p_1 \le p_3 < p_1+l_1) \\ \mathrm{del}(p_1+1,l_1) \quad (p_3 < p_1) \end{array}\right) \Big\backslash \mathrm{ins}(p_2,c_2) \\
&= \begin{cases} \mathrm{del}(p_1,l_1)\backslash\mathrm{ins}(p_2,c_2) & (p_1+l_1 \le p_3) \\ \mathrm{del}(p_1,l_1+1)\backslash\mathrm{ins}(p_2,c_2) & (p_1 \le p_3 < p_1+l_1) \\ \mathrm{del}(p_1+1,l_1)\backslash\mathrm{ins}(p_2,c_2) & (p_3 < p_1) \end{cases} \\
&= \begin{cases} \mathrm{del}(p_1,l_1) & (p_1+l_1 \le p_3) \\ \mathrm{del}(p_1,l_1+2) & (p_1 \le p_3 < p_1+l_1) \\ \mathrm{del}(p_1+2,l_1) & (p_3 < p_1) \end{cases} ,
\end{aligned}$$

and we see that $(a\backslash b)\backslash(c/b) = (a\backslash c)\backslash(b\backslash c)$. Now suppose that $p_2 \ne p_3$. Then we have

$$(a\backslash b)\backslash(c/b) = \left(\begin{array}{c} \mathrm{del}(p_1,l_1) \quad (p_1+l_1 \le p_2) \\ \mathrm{del}(p_1,l_1+1) \quad (p_1 \le p_2 < p_1+l_1) \\ \mathrm{del}(p_1+1,l_1) \quad (p_2 < p_1) \end{array}\right) \Big\backslash \left(\begin{array}{c} \mathrm{ins}(p_3,c_3) \quad (p_3 < p_2) \\ \mathrm{ins}(p_3+1,c_3) \quad (p_3 > p_2) \end{array}\right),$$

which evaluates as follows:

	$p_3 < p_2$	$p_3 > p_2$
$p_1 + l_1 \le p_2$	$\mathrm{del}(p_1,l_1)\backslash\mathrm{ins}(p_3,c_3)$	$\mathrm{del}(p_1,l_1)\backslash\mathrm{ins}(p_3+1,c_3)$
$p_1 \le p_2 < p_1 + l_1$	$\mathrm{del}(p_1,l_1+1)\backslash\mathrm{ins}(p_3,c_3)$	$\mathrm{del}(p_1,l_1+1)\backslash\mathrm{ins}(p_3+1,c_3)$
$p_2 < p_1$	$\mathrm{del}(p_1+1,l_1)\backslash\mathrm{ins}(p_3,c_3)$	$\mathrm{del}(p_1+1,l_1)\backslash\mathrm{ins}(p_3+1,c_3)$

=

	$p_3 < p_2$	$p_3 > p_2$
$p_1 + l_1 \le p_2$	$\text{del}(p_1, l_1)$ $(p_1 + l_1 \le p_3)$ $\text{del}(p_1, l_1 + 1)$ $(p_1 \le p_3 < p_1 + l_1)$ $\text{del}(p_1 + 1, l_1)$ $(p_3 < p_1)$	$\text{del}(p_1, l_1)$ $(p_1 + l_1 \le p_3 + 1)$ $\text{del}(p_1, l_1 + 1)$ $(p_1 \le p_3 + 1 < p_1 + l_1)$ $\text{del}(p_1 + 1, l_1)$ $(p_3 + 1 < p_1)$
$p_1 \le p_2 < p_1 + l_1$	$\text{del}(p_1, l_1 + 1)$ $(p_1 + l_1 + 1 \le p_3)$ $\text{del}(p_1, l_1 + 2)$ $(p_1 \le p_3 < p_1 + l_1 + 1)$ $\text{del}(p_1 + 1, l_1 + 1)$ $(p_3 < p_1)$	$\text{del}(p_1, l_1 + 1)$ $(p_1 + l_1 + 1 \le p_3 + 1)$ $\text{del}(p_1, l_1 + 2)$ $(p_1 \le p_3 + 1 < p_1 + l_1 + 1)$ $\text{del}(p_1 + 1, l_1 + 1)$ $(p_3 + 1 < p_1)$
$p_2 < p_1$	$\text{del}(p_1 + 1, l_1)$ $(p_1 + l_1 + 1 \le p_3)$ $\text{del}(p_1 + 1, l_1 + 1)$ $(p_1 + 1 \le p_3 < p_1 + l_1 + 1)$ $\text{del}(p_1 + 2, l_1)$ $(p_3 < p_1 + 1)$	$\text{del}(p_1 + 1, l_1)$ $(p_1 + l_1 + 1 \le p_3 + 1)$ $\text{del}(p_1 + 1, l_1 + 1)$ $(p_1 + 1 \le p_3 + 1 < p_1 + l_1 + 1)$ $\text{del}(p_1 + 2, l_1)$ $(p_3 + 1 < p_1 + 1)$

=

	$p_3 < p_2$	$p_3 > p_2$
$p_1 + l_1 \le p_2$	$\text{del}(p_1, l_1) \quad (p_1 + l_1 \le p_3)$ $\text{del}(p_1, l_1 + 1) \quad (p_1 \le p_3 < p_1 + l_1)$ $\text{del}(p_1 + 1, l_1) \quad (p_3 < p_1)$	$\text{del}(p_1, l_1)$
$p_1 \le p_2 < p_1 + l_1$	$\text{del}(p_1, l_1 + 2) \quad (p_1 \le p_3)$ $\text{del}(p_1 + 1, l_1 + 1) \quad (p_3 < p_1)$	$\text{del}(p_1, l_1 + 1) \quad (p_1 + l_1 \le p_3)$ $\text{del}(p_1, l_1 + 2) \quad (p_3 < p_1 + l_1)$
$p_2 < p_1$	$\text{del}(p_1 + 2, l_1)$	$\text{del}(p_1 + 1, l_1) \quad (p_1 + l_1 \le p_3)$ $\text{del}(p_1 + 1, l_1 + 1) \quad (p_1 \le p_3 < p_1 + l_1)$ $\text{del}(p_1 + 2, l_1) \quad (p_3 < p_1)$

.

Also, we have

$$(a\backslash c)\backslash(b\backslash c) \quad = \quad \begin{pmatrix} \text{del}(p_1, l_1) \quad (p_1 + l_1 \le p_3) \\ \text{del}(p_1, l_1 + 1) \quad (p_1 \le p_3 < p_1 + l_1) \\ \text{del}(p_1 + 1, l_1) \quad (p_3 < p_1) \end{pmatrix} \Big\backslash \begin{pmatrix} \text{ins}(p_2, c_2) \quad (p_2 < p_3) \\ \text{ins}(p_2 + 1, c_2) \quad (p_2 > p_3) \end{pmatrix}.$$

As this is just the expression for $(a\backslash b)\backslash(c/b)$ with subscripts 2 and 3 interchanged (made possible by our elimination of the case $p_2 = p_3$), the resulting table is the previous table with subscripts 2 and 3 interchanged:

	$p_2 < p_3$	$p_2 > p_3$
$p_1 + l_1 \le p_3$	$\text{del}(p_1, l_1) \quad (p_1 + l_1 \le p_2)$ $\text{del}(p_1, l_1 + 1) \quad (p_1 \le p_2 < p_1 + l_1)$ $\text{del}(p_1 + 1, l_1) \quad (p_2 < p_1)$	$\text{del}(p_1, l_1)$
$p_1 \le p_3 < p_1 + l_1$	$\text{del}(p_1, l_1 + 2) \quad (p_1 \le p_2)$ $\text{del}(p_1 + 1, l_1 + 1) \quad (p_2 < p_1)$	$\text{del}(p_1, l_1 + 1) \quad (p_1 + l_1 \le p_2)$ $\text{del}(p_1, l_1 + 2) \quad (p_2 < p_1 + l_1)$
$p_3 < p_1$	$\text{del}(p_1 + 2, l_1)$	$\text{del}(p_1 + 1, l_1) \quad (p_1 + l_1 \le p_2)$ $\text{del}(p_1 + 1, l_1 + 1) \quad (p_1 \le p_2 < p_1 + l_1)$ $\text{del}(p_1 + 2, l_1) \quad (p_2 < p_1)$

.

Comparing this table with the previous one, we see that they are equivalent. Hence $(a\backslash b)\backslash(c/b) = (a\backslash c)\backslash(b\backslash c)$, i.e. $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$*:* By the total ordering of a, b, and c, we have $(b\hat{}a)\hat{}(c\hat{}a) = (b/a)\backslash(c/a)$ and $(b\hat{}c)\hat{}(a\hat{}c) = (b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned}(b/a)\backslash(c/a) &= (\text{ins}(p_2,c_2)/\text{del}(p_1,l_1))\backslash(\text{ins}(p_3,c_3)/\text{del}(p_1,l_1)),\\ (b\backslash c)/(a\backslash c) &= (\text{ins}(p_2,c_2)\backslash\text{ins}(p_3,c_3))/(\text{del}(p_1,l_1)\backslash\text{ins}(p_3,c_3)).\end{aligned}$$

Let us first assume that $p_2 = p_3$. Then we have

$$\begin{aligned}(b/a)\backslash(c/a) &= \left(\begin{array}{c}\text{ins}(p_2,c_2) \quad (p_2 < p_1)\\ \text{NOOP} \quad (p_1 \le p_2 < p_1 + l_1)\\ \text{ins}(p_2 - l_1, c_2) \quad (p_1 + l_1 \le p_2)\end{array}\right)\Bigg\backslash\left(\begin{array}{c}\text{ins}(p_3,c_3) \quad (p_3 < p_1)\\ \text{NOOP} \quad (p_1 \le p_3 < p_1 + l_1)\\ \text{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \le p_3)\end{array}\right)\\ &= \begin{cases}\text{ins}(p_2,c_2)\backslash\text{ins}(p_3,c_3) & (p_2 < p_1)\\ \text{NOOP}\backslash\text{NOOP} & (p_1 \le p_2 < p_1 + l_1)\\ \text{ins}(p_2 - l_1, c_2)\backslash\text{ins}(p_3 - l_1, c_3) & (p_1 + l_1 \le p_2)\end{cases}\\ &= \begin{cases}\text{ins}(p_2,c_2) & (p_2 < p_1)\\ \text{NOOP} & (p_1 \le p_2 < p_1 + l_1)\\ \text{ins}(p_2 - l_1, c_2) & (p_1 + l_1 \le p_2)\end{cases}.\end{aligned}$$

Also,

$$\begin{aligned}(b\backslash c)/(a\backslash c) &= \text{ins}(p_2,c_2)\Bigg/\left(\begin{array}{c}\text{del}(p_1,l_1) \quad (p_1 + l_1 \le p_3)\\ \text{del}(p_1, l_1 + 1) \quad (p_1 \le p_3 < p_1 + l_1)\\ \text{del}(p_1 + 1, l_1) \quad (p_3 < p_1)\end{array}\right)\\ &= \begin{cases}\text{ins}(p_2,c_2)/\text{del}(p_1,l_1) & (p_1 + l_1 \le p_3)\\ \text{ins}(p_2,c_2)/\text{del}(p_1,l_1+1) & (p_1 \le p_3 < p_1 + l_1)\\ \text{ins}(p_2,c_2)/\text{del}(p_1+1,l_1) & (p_3 < p_1)\end{cases}\\ &= \begin{cases}\text{ins}(p_2 - l_1, c_2) & (p_1 + l_1 \le p_3)\\ \text{NOOP} & (p_1 \le p_3 < p_1 + l_1)\\ \text{ins}(p_2,c_2) & (p_3 < p_1)\end{cases},\end{aligned}$$

and we see that $(b/a)\backslash(c/a) = (b\backslash c)/(a\backslash c)$. Now suppose that $p_2 \neq p_3$. Then we have

$$(b/a)\backslash(c/a) = \left(\begin{array}{c}\text{ins}(p_2,c_2) \quad (p_2 < p_1)\\ \text{NOOP} \quad (p_1 \le p_2 < p_1 + l_1)\\ \text{ins}(p_2 - l_1, c_2) \quad (p_1 + l_1 \le p_2)\end{array}\right)\Bigg\backslash\left(\begin{array}{c}\text{ins}(p_3,c_3) \quad (p_3 < p_1)\\ \text{NOOP} \quad (p_1 \le p_3 < p_1 + l_1)\\ \text{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \le p_3)\end{array}\right),$$

which evaluates as follows:

	$p_3 < p_1$	$p_1 \leq p_3$, $p_3 < p_1 + l_1$	$p_1 + l_1 \leq p_3$
$p_2 < p_1$	$\text{ins}(p_2, c_2)\backslash\text{ins}(p_3, c_3)$	$\text{ins}(p_2, c_2)\backslash\text{NOOP}$	$\text{ins}(p_2, c_2)\backslash\text{ins}(p_3 - l_1, c_3)$
$p_1 \leq p_2$, $p_2 < p_1 + l_1$	$\text{NOOP}\backslash\text{ins}(p_3, c_3)$	$\text{NOOP}\backslash\text{NOOP}$	$\text{NOOP}\backslash\text{ins}(p_3 - l_1, c_3)$
$p_1 + l_1 \leq p_2$	$\text{ins}(p_2 - l_1, c_2)\backslash\text{ins}(p_3, c_3)$	$\text{ins}(p_2 - l_1, c_2)\backslash\text{NOOP}$	$\text{ins}(p_2 - l_1, c_2)\backslash\text{ins}(p_3 - l_1, c_3)$

$=$

	$p_3 < p_1$	$p_1 \leq p_3$, $p_3 < p_1 + l_1$	$p_1 + l_1 \leq p_3$
$p_2 < p_1$	$\text{ins}(p_2, c_2)$ $(p_2 < p_3)$ $\text{ins}(p_2 + 1, c_2)$ $(p_3 < p_2)$	$\text{ins}(p_2, c_2)$	$\text{ins}(p_2, c_2)$ $(p_2 \leq p_3 - l_1)$ $\text{ins}(p_2 + 1, c_2)$ $(p_3 - l_1 < p_2)$
$p_1 \leq p_2$, $p_2 < p_1 + l_1$	NOOP	NOOP	NOOP
$p_1 + l_1 \leq p_2$	$\text{ins}(p_2 - l_1, c_2)$ $(p_2 - l_1 \leq p_3)$ $\text{ins}(p_2 - l_1 + 1, c_2)$ $(p_3 < p_2 - l_1)$	$\text{ins}(p_2 - l_1, c_2)$	$\text{ins}(p_2 - l_1, c_2)$ $(p_2 - l_1 < p_3 - l_1)$ $\text{ins}(p_2 - l_1 + 1, c_2)$ $(p_3 - l_1 < p_2 - l_1)$

$=$

	$p_3 < p_1$	$p_1 \leq p_3$, $p_3 < p_1 + l_1$	$p_1 + l_1 \leq p_3$
$p_2 < p_1$	$\text{ins}(p_2, c_2) \quad (p_2 < p_3)$ $\text{ins}(p_2 + 1, c_2) \quad (p_3 < p_2)$	$\text{ins}(p_2, c_2)$	$\text{ins}(p_2, c_2)$
$p_1 \leq p_2$, $p_2 < p_1 + l_1$	NOOP	NOOP	NOOP
$p_1 + l_1 \leq p_2$	$\text{ins}(p_2 - l_1 + 1, c_2)$	$\text{ins}(p_2 - l_1, c_2)$	$\text{ins}(p_2 - l_1, c_2) \quad (p_2 < p_3)$ $\text{ins}(p_2 - l_1 + 1, c_2) \quad (p_3 < p_2)$

.

Also,

$$(b\backslash c)/(a\backslash c) \quad = \quad \left(\begin{array}{c} \text{ins}(p_2, c_2) \quad (p_2 < p_3) \\ \text{ins}(p_2 + 1, c_2) \quad (p_3 < p_2) \end{array} \right) \Big/ \left(\begin{array}{c} \text{del}(p_1, l_1) \quad (p_1 + l_1 \leq p_3) \\ \text{del}(p_1, l_1 + 1) \quad (p_1 \leq p_3 < p_1 + l_1) \\ \text{del}(p_1 + 1, l_1) \quad (p_3 < p_1) \end{array} \right),$$

which evaluates as follows:

	$p_2 < p_3$	$p_3 < p_2$
$p_1 + l_1 \leq p_3$	$\text{ins}(p_2, c_2)/\text{del}(p_1, l_1)$	$\text{ins}(p_2 + 1, c_2)/\text{del}(p_1, l_1)$
$p_1 \leq p_3 < p_1 + l_1$	$\text{ins}(p_2, c_2)/\text{del}(p_1, l_1 + 1)$	$\text{ins}(p_2 + 1, c_2)/\text{del}(p_1, l_1 + 1)$
$p_3 < p_1$	$\text{ins}(p_2, c_2)/\text{del}(p_1 + 1, l_1)$	$\text{ins}(p_2 + 1, c_2)/\text{del}(p_1 + 1, l_1)$

=

	$p_2 < p_3$	$p_3 < p_2$
$p_1 + l_1 \leq p_3$	$\text{ins}(p_2, c_2)$ $(p_2 < p_1)$ NOOP $(p_1 \leq p_2 < p_1 + l_1)$ $\text{ins}(p_2 - l_1, c_2)$ $(p_1 + l_1 \leq p_2)$	$\text{ins}(p_2 + 1, c_2)$ $(p_2 + 1 < p_1)$ NOOP $(p_1 \leq p_2 + 1 < p_1 + l_1)$ $\text{ins}(p_2 - l_1 + 1, c_2)$ $(p_1 + l_1 \leq p_2)$
$p_1 \leq p_3$, $p_3 < p_1 + l_1$	$\text{ins}(p_2, c_2)$ $(p_2 < p_1)$ NOOP $(p_1 \leq p_2 < p_1 + l_1 + 1)$ $\text{ins}(p_2 - l_1 - 1, c_2)$ $(p_1 + l_1 + 1 \leq p_2)$	$\text{ins}(p_2 + 1, c_2)$ $(p_2 + 1 < p_1)$ NOOP $(p_1 \leq p_2 + 1 < p_1 + l_1 + 1)$ $\text{ins}(p_2 - l_1, c_2)$ $(p_1 + l_1 + 1 \leq p_2 + 1)$
$p_3 < p_1$	$\text{ins}(p_2, c_2)$ $(p_2 < p_1 + 1)$ NOOP $(p_1 + 1 \leq p_2 < p_2 + l_1 + 1)$ $\text{ins}(p_2 - l_1, c_2)$ $(p_1 + l_1 + 1 \leq p_2)$	$\text{ins}(p_2 + 1, c_2)$ $(p_1 + 1 < p_2 + 1)$ NOOP $(p_2 + 1 \leq p_1 + 1 < p_1 + l_1 + 1)$ $\text{ins}(p_2 - l_1 + 1, c_2)$ $(p_1 + l_1 + 1 \leq p_2 + 1)$

=

	$p_2 < p_3$	$p_3 < p_2$
$p_1 + l_1 \leq p_3$	$\text{ins}(p_2, c_2)$ $(p_2 < p_1)$ NOOP $(p_1 \leq p_2 < p_1 + l_1)$ $\text{ins}(p_2 - l_1, c_2)$ $(p_1 + l_1 \leq p_2)$	$\text{ins}(p_2 - l_1 + 1, c_2)$
$p_1 \leq p_3$, $p_3 < p_1 + l_1$	$\text{ins}(p_2, c_2)$ $(p_2 < p_1)$ NOOP $(p_1 \leq p_2)$	NOOP $(p_2 < p_1 + l_1)$ $\text{ins}(p_2 - l_1, c_2)$ $(p_1 + l_1 \leq p_2)$
$p_3 < p_1$	$\text{ins}(p_2, c_2)$	$\text{ins}(p_2 + 1, c_2)$ $(p_1 < p_2)$ NOOP $(p_2 \leq p_1 < p_1 + l_1)$ $\text{ins}(p_2 - l_1 + 1, c_2)$ $(p_1 + l_1 \leq p_2)$

.

Comparing this table with the previous one, we see that they are equivalent. Thus, $(b/a)\backslash(c/a) = (b\backslash c)/(a\backslash c)$, i.e. $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$.

$(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$: By the total ordering of a, b, and c, we have $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c/a)/(b/a)$ and $(c\hat{\ }b)\hat{\ }(a\hat{\ }b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned}(c/a)/(b/a) &= (\text{ins}(p_3, c_3)/\text{del}(p_1, l_1))/(\text{ins}(p_2, c_2)/\text{del}(p_1, l_1)),\\ (c/b)/(a\backslash b) &= (\text{ins}(p_3, c_3)/\text{ins}(p_2, c_2))/(\text{del}(p_1, l_1)\backslash\text{ins}(p_2, c_2)).\end{aligned}$$

First assume that $p_2 = p_3$. Then we have

$$
\begin{aligned}
(c/a)/(b/a) &= \left(\begin{array}{c} \mathrm{ins}(p_3, c_3) \quad (p_3 < p_1) \\ \mathrm{NOOP} \quad (p_1 \le p_3 < p_1 + l_1) \\ \mathrm{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \le p_3) \end{array}\right) \Big/ \left(\begin{array}{c} \mathrm{ins}(p_2, c_2) \quad (p_2 < p_1) \\ \mathrm{NOOP} \quad (p_1 \le p_2 < p_1 + l_1) \\ \mathrm{ins}(p_2 - l_1, c_2) \quad (p_1 + l_1 \le p_2) \end{array}\right) \\
&= \begin{cases} \mathrm{ins}(p_3, c_3)/\mathrm{ins}(p_2, c_2) & (p_3 < p_1) \\ \mathrm{NOOP}/\mathrm{NOOP} & (p_1 \le p_3 < p_1 + l_1) \\ \mathrm{ins}(p_3 - l_1, c_3)/\mathrm{ins}(p_2 - l_1, c_2) & (p_1 + l_1 \le p_3) \end{cases} \\
&= \begin{cases} \mathrm{ins}(p_3 + 1, c_3) & (p_3 < p_1) \\ \mathrm{NOOP} & (p_1 \le p_3 < p_1 + l_1) \\ \mathrm{ins}(p_3 - l_1 + 1, c_3) & (p_1 + l_1 \le p_3) \end{cases} .
\end{aligned}
$$

Also,

$$
\begin{aligned}
(c/b)/(a\backslash b) &= \mathrm{ins}(p_3 + 1, c_3) \Big/ \left(\begin{array}{c} \mathrm{del}(p_1, l_1) \quad (p_1 + l_1 \le p_2) \\ \mathrm{del}(p_1, l_1 + 1) \quad (p_1 \le p_2 < p_1 + l_1) \\ \mathrm{del}(p_1 + 1, l_1) \quad (p_2 < p_1) \end{array}\right) \\
&= \begin{cases} \mathrm{ins}(p_3 + 1, c_3)/\mathrm{del}(p_1, l_1) & (p_1 + l_1 \le p_2) \\ \mathrm{ins}(p_3 + 1, c_3)/\mathrm{del}(p_1, l_1 + 1) & (p_1 \le p_2 < p_1 + l_1) \\ \mathrm{ins}(p_3 + 1, c_3)/\mathrm{del}(p_1 + 1, l_1) & (p_2 < p_1) \end{cases} \\
&= \begin{cases} \mathrm{ins}(p_3 - l_1 + 1, c_3) & (p_1 + l_1 \le p_2) \\ \mathrm{NOOP} & (p_1 \le p_2 < p_1 + l_1) \\ \mathrm{ins}(p_3 + 1, c_3) & (p_2 < p_1) \end{cases} ,
\end{aligned}
$$

and we see that $(c/a)/(b/a) = (c/b)/(a\backslash b)$. Now suppose that $p_2 \ne p_3$. Then we have

$$
(c/a)/(b/a) = \left(\begin{array}{c} \mathrm{ins}(p_3, c_3) \quad (p_3 < p_1) \\ \mathrm{NOOP} \quad (p_1 \le p_3 < p_1 + l_1) \\ \mathrm{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \le p_3) \end{array}\right) \Big/ \left(\begin{array}{c} \mathrm{ins}(p_2, c_2) \quad (p_2 < p_1) \\ \mathrm{NOOP} \quad (p_1 \le p_2 < p_1 + l_1) \\ \mathrm{ins}(p_2 - l_1, c_2) \quad (p_1 + l_1 \le p_2) \end{array}\right),
$$

which is exactly the expression for $(b/a)\backslash(c/a)$, with subscripts 2 and 3 interchanged. Thus, $(c/a)/(b/a)$ expands to the following table:

	$p_2 < p_1$	$p_1 \le p_2$, $p_2 < p_1 + l_1$	$p_1 + l_1 \le p_2$
$p_3 < p_1$	$\mathrm{ins}(p_3, c_3)$ $(p_3 < p_2)$ $\mathrm{ins}(p_3 + 1, c_2)$ $(p_2 < p_3)$	$\mathrm{ins}(p_3, c_3)$	$\mathrm{ins}(p_3, c_3)$
$p_1 \le p_3$, $p_3 < p_1 + l_1$	NOOP	NOOP	NOOP
$p_1 + l_1 \le p_3$	$\mathrm{ins}(p_3 - l_1 + 1, c_3)$	$\mathrm{ins}(p_3 - l_1, c_3)$	$\mathrm{ins}(p_3 - l_1, c_3)$ $(p_3 < p_2)$ $\mathrm{ins}(p_3 - l_1 + 1, c_3)$ $(p_2 < p_3)$

.

Also,

$$(c/b)/(a\backslash b) \quad = \quad \left(\begin{array}{c} \mathrm{ins}(p_3, c_3) \quad (p_3 < p_2) \\ \mathrm{ins}(p_3+1, c_3) \quad (p_2 < p_3) \end{array}\right) \Big/ \left(\begin{array}{c} \mathrm{del}(p_1, l_1) \quad (p_1 + l_1 \leq p_2) \\ \mathrm{del}(p_1, l_1+1) \quad (p_1 \leq p_2 < p_1 + l_1) \\ \mathrm{del}(p_1+1, l_1) \quad (p_2 < p_1) \end{array}\right),$$

which is exactly the expression for $(b\backslash c)/(a\backslash c)$, with subscripts 2 and 3 interchanged. Thus, $(c/b)/(a\backslash b)$ expands to the following table:

	$p_3 < p_2$	$p_2 < p_3$
$p_1 + l_1 \leq p_2$	$\mathrm{ins}(p_3, c_3) \quad (p_3 < p_1)$ $\mathrm{NOOP} \quad (p_1 \leq p_3 < p_1 + l_1)$ $\mathrm{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \leq p_3)$	$\mathrm{ins}(p_3 - l_1 + 1, c_3)$
$p_1 \leq p_2,$ $p_2 < p_1 + l_1$	$\mathrm{ins}(p_3, c_3) \quad (p_3 < p_1)$ $\mathrm{NOOP} \quad (p_1 \leq p_3)$	$\mathrm{NOOP} \quad (p_3 < p_1 + l_1)$ $\mathrm{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \leq p_3)$
$p_2 < p_1$	$\mathrm{ins}(p_3, c_3)$	$\mathrm{ins}(p_3 + 1, c_3) \quad (p_1 < p_3)$ $\mathrm{NOOP} \quad (p_3 \leq p_1 < p_1 + l_1)$ $\mathrm{ins}(p_3 - l_1 + 1, c_3) \quad (p_1 + l_1 \leq p_3)$

Comparing this table with the previous one, we see that they are equivalent. Hence $(c/a)/(b/a) = (c/b)/(a\backslash b)$, i.e. $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$. This completes Case 2.

Case 3 $a = \mathit{ins}(p_1, c_1)$, $b = \mathit{del}(p_2, l_2)$, $c = \mathit{ins}(p_3, c_3)$.

We must show that $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$, $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$, and $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$.

$(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$: By the total ordering of a, b, and c, we have $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{\ }c)\hat{\ }(b\hat{\ }c) = (a\backslash c)\backslash(b\backslash c)$. We have

$$\begin{aligned} (a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_2, l_2))\backslash(\mathrm{ins}(p_3, c_3)\backslash\mathrm{del}(p_2, l_2)), \\ (a\backslash c)\backslash(b\backslash c) &= (\mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_3, c_3))\backslash(\mathrm{del}(p_2, l_2)/\mathrm{ins}(p_3, c_3)). \end{aligned}$$

Let us first assume that $p_1 = p_3$. Then we have

$$\begin{aligned} (a\backslash b)\backslash(c/b) &= \left(\begin{array}{c} \mathrm{ins}(p_1, c_1) \quad (p_1 < p_2) \\ \mathrm{NOOP} \quad (p_2 \leq p_1 < p_2 + l_2) \\ \mathrm{ins}(p_1 - l_2, c_1) \quad (p_2 + l_2 \leq p_1) \end{array}\right) \Big\backslash \left(\begin{array}{c} \mathrm{ins}(p_3, c_3) \quad (p_3 < p_2) \\ \mathrm{NOOP} \quad (p_2 \leq p_3 < p_2 + l_2) \\ \mathrm{ins}(p_3 - l_2, c_3) \quad (p_2 + l_2 \leq p_3) \end{array}\right) \\ &= \left\{\begin{array}{ll} \mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_3, c_3) & (p_1 < p_2) \\ \mathrm{NOOP}\backslash\mathrm{NOOP} & (p_2 \leq p_1 < p_2 + l_2) \\ \mathrm{ins}(p_1 - l_2, c_1)\backslash\mathrm{ins}(p_3 - l_2, c_3) & (p_2 + l_2 \leq p_1) \end{array}\right. \\ &= \left\{\begin{array}{ll} \mathrm{ins}(p_1, c_1) & (p_1 < p_2) \\ \mathrm{NOOP} & (p_2 \leq p_1 < p_2 + l_2) \\ \mathrm{ins}(p_1 - l_2, c_1) & (p_2 + l_2 \leq p_1) \end{array}\right. . \end{aligned}$$

Also,

$$\begin{aligned}
(a\backslash c)\backslash(b\backslash c) &= \mathrm{ins}(p_1,c_1)\Big\backslash\begin{pmatrix} \mathrm{del}(p_2,l_2) & (p_2+l_2\le p_3) \\ \mathrm{del}(p_2,l_2+1) & (p_2\le p_3<p_2+l_2) \\ \mathrm{del}(p_2+1,l_2) & (p_3<p_2)\end{pmatrix} \\
&= \begin{cases} \mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2) & (p_2+l_2\le p_3) \\ \mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2+1) & (p_2\le p_3<p_2+l_2) \\ \mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2+1,l_2) & (p_3<p_2)\end{cases} \\
&= \begin{cases} \mathrm{ins}(p_1-l_2,c_1) & (p_2+l_2\le p_3) \\ \mathrm{NOOP} & (p_2\le p_3<p_2+l_2) \\ \mathrm{ins}(p_1,c_1) & (p_3<p_2)\end{cases},
\end{aligned}$$

and we see that $(a\backslash b)\backslash(c/b)=(a\backslash c)\backslash(b\backslash c)$. Now suppose that $p_1\ne p_3$. Then we have

$$\begin{aligned}
(a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{del}(p_2,l_2)) \\
&= (\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_2,l_2))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{del}(p_2,l_2)),
\end{aligned}$$

which by Case 2 (interchange subscripts 1 and 2 and compare with $(b/a)\backslash(c/a)=(b\backslash c)/(a\backslash c)$ in Case 2) is equal to $(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3))/(\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3))$. Hence

$$\begin{aligned}
(a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3))/(\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3)) \\
&= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3))\backslash(\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3)) \\
&= (a\backslash c)\backslash(b\backslash c).
\end{aligned}$$

Hence, $(a\hat{}b)\hat{}(c\hat{}b)=(a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a)=(b\hat{}c)\hat{}(a\hat{}c)$: By the total ordering of a, b, and c, we have $(b\hat{}a)\hat{}(c\hat{}a)=(b/a)\backslash(c/a)$ and $(b\hat{}c)\hat{}(a\hat{}c)=(b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned}
(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1)), \\
(b\backslash c)/(a\backslash c) &= (\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3)).
\end{aligned}$$

First assume that $p_1=p_3$. Then we have

$$\begin{aligned}
(b/a)\backslash(c/a) &= \begin{pmatrix} \mathrm{del}(p_2,l_2) & (p_2+l_2\le p_1) \\ \mathrm{del}(p_2,l_2+1) & (p_2\le p_1<p_2+l_2) \\ \mathrm{del}(p_2+1,l_2) & (p_1<p_2)\end{pmatrix}\Big\backslash\mathrm{ins}(p_3+1,c_3) \\
&= \begin{cases} \mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3+1,c_3) & (p_2+l_2\le p_1) \\ \mathrm{del}(p_2,l_2+1)\backslash\mathrm{ins}(p_3+1,c_3) & (p_2\le p_1<p_2+l_2) \\ \mathrm{del}(p_2+1,l_2)\backslash\mathrm{ins}(p_3+1,c_3) & (p_1<p_2)\end{cases} \\
&= \begin{cases} \mathrm{del}(p_2,l_2) & (p_2+l_2\le p_1) \\ \mathrm{del}(p_2,l_2+2) & (p_2\le p_1<p_2+l_2) \\ \mathrm{del}(p_2+2,l_2) & (p_1<p_2)\end{cases}.
\end{aligned}$$

Also,

$$\begin{aligned}
(b\backslash c)/(a\backslash c) &= \left(\begin{array}{c} \mathrm{del}(p_2,l_2) \quad (p_2+l_2 \le p_3) \\ \mathrm{del}(p_2,l_2+1) \quad (p_2 \le p_3 < p_2+l_2) \\ \mathrm{del}(p_2+1,l_2) \quad (p_3 < p_2) \end{array}\right) \Big/ \mathrm{ins}(p_1,c_1) \\
&= \begin{cases} \mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1) & (p_2+l_2 \le p_3) \\ \mathrm{del}(p_2,l_2+1)/\mathrm{ins}(p_1,c_1) & (p_2 \le p_3 < p_2+l_2) \\ \mathrm{del}(p_2+1,l_2)/\mathrm{ins}(p_1,c_1) & (p_3 < p_2) \end{cases} \\
&= \begin{cases} \mathrm{del}(p_2,l_2) & (p_2+l_2 \le p_3) \\ \mathrm{del}(p_2,l_2+2) & (p_2 \le p_3 < p_2+l_2) \\ \mathrm{del}(p_2+2,l_2) & (p_3 < p_2) \end{cases},
\end{aligned}$$

and we see that $(b/a)\backslash(c/a) = (b\backslash c)/(a\backslash c)$. Now suppose that $p_1 \ne p_3$. Then we have

$$\begin{aligned}
(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1)) \\
&= (\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_1,c_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1)),
\end{aligned}$$

which by Case 2 (interchange subscripts 1 and 2 and compare with $(a\backslash b)\backslash(c/b) = (a\backslash c)\backslash(b\backslash c)$ in Case 2) is equal to $(\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3))\backslash(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3))$. Hence

$$\begin{aligned}
(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3))\backslash(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3)) \\
&= (\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_3,c_3))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{ins}(p_3,c_3)) \\
&= (b\backslash c)/(a\backslash c).
\end{aligned}$$

Hence, $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$.

$(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$: By the total ordering of a, b, and c, we have $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c/a)/(b/a)$ and $(c\hat{\ }b)\hat{\ }(a\hat{\ }b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned}
(c/a)/(b/a) &= (\mathrm{ins}(p_3,c_3)/\mathrm{ins}(p_1,c_1))/(\mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1)), \\
(c/b)/(a\backslash b) &= (\mathrm{ins}(p_3,c_3)/\mathrm{del}(p_2,l_2))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2)).
\end{aligned}$$

First assume that $p_1 = p_3$. Then we have

$$\begin{aligned}
(c/a)/(b/a) &= \mathrm{ins}(p_3+1,c_3) \Big/ \left(\begin{array}{c} \mathrm{del}(p_2,l_2) \quad (p_2+l_2 \le p_1) \\ \mathrm{del}(p_2,l_2+1) \quad (p_2 \le p_1 < p_2+l_2) \\ \mathrm{del}(p_2+1,l_2) \quad (p_1 < p_2) \end{array}\right) \\
&= \begin{cases} \mathrm{ins}(p_3+1,c_3)/\mathrm{del}(p_2,l_2) & (p_2+l_2 \le p_1) \\ \mathrm{ins}(p_3+1,c_3)/\mathrm{del}(p_2,l_2+1) & (p_2 \le p_1 < p_2+l_2) \\ \mathrm{ins}(p_3+1,c_3)/\mathrm{del}(p_2+1,l_2) & (p_1 < p_2) \end{cases} \\
&= \begin{cases} \mathrm{ins}(p_3-l_2+1,c_3) & (p_2+l_2 \le p_1) \\ \mathrm{NOOP} & (p_2 \le p_1 < p_2+l_2) \\ \mathrm{ins}(p_3+1,c_3) & (p_1 < p_2) \end{cases}.
\end{aligned}$$

Also,

$$\begin{aligned}
(c/b)/(a\backslash b) &= \left(\begin{array}{c} \text{ins}(p_3,c_3) \quad (p_3 < p_2) \\ \text{NOOP} \quad (p_2 \le p_3 < p_2+l_2) \\ \text{ins}(p_3-l_2,c_3) \quad (p_2+l_2 \le p_3) \end{array}\right) \Bigg/ \left(\begin{array}{c} \text{ins}(p_1,c_1) \quad (p_1 < p_2) \\ \text{NOOP} \quad (p_2 \le p_1 < p_2+l_2) \\ \text{ins}(p_1-l_2,c_1) \quad (p_2+l_2 \le p_1) \end{array}\right) \\
&= \begin{cases} \text{ins}(p_3,c_3)/\text{ins}(p_1,c_1) & (p_1 < p_2) \\ \text{NOOP}/\text{NOOP} & (p_2 \le p_1 < p_2+l_2) \\ \text{ins}(p_3-l_2,c_3)/\text{ins}(p_1-l_2,c_1) & (p_2+l_2 \le p_1) \end{cases} \\
&= \begin{cases} \text{ins}(p_3+1,c_3) & (p_1 < p_2) \\ \text{NOOP} & (p_2 \le p_1 < p_2+l_2) \\ \text{ins}(p_3-l_2+1,c_3) & (p_2+l_2 \le p_1) \end{cases},
\end{aligned}$$

and we see that $(c/a)/(b/a) = (c/b)/(a\backslash b)$. Now suppose $p_1 \ne p_3$. Then

$$\begin{aligned}
(c/a)/(b/a) &= (\text{ins}(p_3,c_3)/\text{ins}(p_1,c_1))/(\text{del}(p_2,l_2)/\text{ins}(p_1,c_1)) \\
&= (\text{ins}(p_3,c_3)/\text{ins}(p_1,c_1))/(\text{del}(p_2,l_2)\backslash\text{ins}(p_1,c_1)),
\end{aligned}$$

which by Case 2 (interchange subscripts 1 and 2 and compare with $(c/b)/(a\backslash b) = (c/a)/(b/a)$ in Case 2) is equal to $(\text{ins}(p_3,c_3)/\text{del}(p_2,l_2))/(\text{ins}(p_1,c_1)/\text{del}(p_2,l_2))$. Hence

$$\begin{aligned}
(c/a)/(b/a) &= (\text{ins}(p_3,c_3)/\text{del}(p_2,l_2))/(\text{ins}(p_1,c_1)/\text{del}(p_2,l_2)) \\
&= (\text{ins}(p_3,c_3)/\text{del}(p_2,l_2))/(\text{ins}(p_1,c_1)\backslash\text{del}(p_2,l_2)) \\
&= (c/b)/(a\backslash b).
\end{aligned}$$

Hence, $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$. This completes Case 3.

Case 4 $a = \mathit{ins}(p_1,c_1)$, $b = \mathit{ins}(p_2,c_2)$, $c = \mathit{del}(p_3,l_3)$.

We must show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$*:* By the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash c)\backslash(b\backslash c)$. We have

$$\begin{aligned}
(a\backslash b)\backslash(c/b) &= (\text{ins}(p_1,c_1)\backslash\text{ins}(p_2,c_2))\backslash(\text{del}(p_3,l_3)/\text{ins}(p_2,c_2)), \\
(a\backslash c)\backslash(b\backslash c) &= (\text{ins}(p_1,c_1)\backslash\text{del}(p_3,l_3))\backslash(\text{ins}(p_2,c_2)\backslash\text{del}(p_3,l_3)).
\end{aligned}$$

First assume that $p_1 = p_2$. Then we have

$$(a\backslash b)\backslash(c/b) = \text{ins}(p_1,c_1)\Bigg\backslash\left(\begin{array}{c} \text{del}(p_3,l_3) \quad (p_3+l_3 \le p_2) \\ \text{del}(p_3,l_3+1) \quad (p_3 \le p_2 < p_3+l_3) \\ \text{del}(p_3+1,l_3) \quad (p_2 < p_3) \end{array}\right)$$

$$= \begin{cases} \mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_3, l_3) & (p_3 + l_3 \le p_2) \\ \mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_3, l_3 + 1) & (p_3 \le p_2 < p_3 + l_3) \\ \mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_3 + 1, l_3) & (p_2 < p_3) \end{cases}$$

$$= \begin{cases} \mathrm{ins}(p_1 - l_3, c_1) & (p_3 + l_3 \le p_2) \\ \mathrm{NOOP} & (p_3 \le p_2 < p_3 + l_3) \\ \mathrm{ins}(p_1, c_1) & (p_2 < p_3) \end{cases} .$$

Also,

$$\begin{aligned} (a\backslash c)\backslash(b\backslash c) &= \begin{pmatrix} \mathrm{ins}(p_1, c_1) \quad (p_1 < p_3) \\ \mathrm{NOOP} \quad (p_3 \le p_1 < p_3 + l_3) \\ \mathrm{ins}(p_1 - l_3, c_1) \quad (p_3 + l_3 \le p_1) \end{pmatrix} \Big\backslash \begin{pmatrix} \mathrm{ins}(p_2, c_2) \quad (p_2 < p_3) \\ \mathrm{NOOP} \quad (p_3 \le p_2 < p_3 + l_3) \\ \mathrm{ins}(p_2 - l_3, c_2) \quad (p_3 + l_3 \le p_2) \end{pmatrix} \\ &= \begin{cases} \mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_2, c_2) & (p_1 < p_3) \\ \mathrm{NOOP}\backslash\mathrm{NOOP} & (p_3 \le p_1 < p_3 + l_3) \\ \mathrm{ins}(p_1 - l_3, c_1)\backslash\mathrm{ins}(p_2 - l_3, c_2) & (p_3 + l_3 \le p_1) \end{cases} \\ &= \begin{cases} \mathrm{ins}(p_1, c_1) & (p_1 < p_3) \\ \mathrm{NOOP} & (p_3 \le p_1 < p_3 + l_3) \\ \mathrm{ins}(p_1 - l_3, c_1) & (p_3 + l_3 \le p_1) \end{cases} , \end{aligned}$$

and we see that $(a\backslash b)\backslash(c/b) = (a\backslash c)\backslash(b\backslash c)$. Now suppose that $p_1 \ne p_2$. Then we have

$$\begin{aligned} (a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1, c_1)\backslash\mathrm{ins}(p_2, c_2))\backslash(\mathrm{del}(p_3, l_3)/\mathrm{ins}(p_2, c_2)) \\ &= (\mathrm{ins}(p_1, c_1)/\mathrm{ins}(p_2, c_2))/(\mathrm{del}(p_3, l_3)\backslash\mathrm{ins}(p_2, c_2)), \end{aligned}$$

which by Case 2 (interchange subscripts 1 and 3 and compare with $(c/b)/(a\backslash b) = (c/a)/(b/a)$ in Case 2) is equal to $(\mathrm{ins}(p_1, c_1)/\mathrm{del}(p_3, l_3))/(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3))$. Hence,

$$\begin{aligned} (a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1, c_1)/\mathrm{del}(p_3, l_3))/(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3)) \\ &= (\mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_3, l_3))\backslash(\mathrm{ins}(p_2, c_2)\backslash\mathrm{del}(p_3, l_3)) \\ &= (a\backslash c)\backslash(b\backslash c). \end{aligned}$$

Thus, $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$.

$(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$: By the total ordering of a, b, and c, we have $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b/a)\backslash(c/a)$ and $(b\hat{\ }c)\hat{\ }(a\hat{\ }c) = (b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned} (b/a)\backslash(c/a) &= (\mathrm{ins}(p_2, c_2)/\mathrm{ins}(p_1, c_1))\backslash(\mathrm{del}(p_3, l_3)/\mathrm{ins}(p_1, c_1)), \\ (b\backslash c)/(a\backslash c) &= (\mathrm{ins}(p_2, c_2)\backslash\mathrm{del}(p_3, l_3))/(\mathrm{ins}(p_1, c_1)\backslash\mathrm{del}(p_3, l_3)). \end{aligned}$$

First assume that $p_1 = p_2$. Then we have

$$(b/a)\backslash(c/a) = \mathrm{ins}(p_2 + 1, c_2)\Big\backslash \begin{pmatrix} \mathrm{del}(p_3, l_3) \quad (p_3 + l_3 \le p_1) \\ \mathrm{del}(p_3, l_3 + 1) \quad (p_3 \le p_1 < p_3 + l_3) \\ \mathrm{del}(p_3 + 1, l_3) \quad (p_1 < p_3) \end{pmatrix}$$

$$= \begin{cases} \text{ins}(p_2+1,c_2)\backslash\text{del}(p_3,l_3) & (p_3+l_3 \le p_1) \\ \text{ins}(p_2+1,c_2)\backslash\text{del}(p_3,l_3+1) & (p_3 \le p_1 < p_3+l_3) \\ \text{ins}(p_2+1,c_2)\backslash\text{del}(p_3+1,l_3) & (p_1 < p_3) \end{cases}$$

$$= \begin{cases} \text{ins}(p_2-l_3+1,c_2) & (p_3+l_3 \le p_1) \\ \text{NOOP} & (p_3 \le p_1 < p_3+l_3) \\ \text{ins}(p_2+1,c_2) & (p_1 < p_3) \end{cases}.$$

Also,

$$\begin{aligned} (b\backslash c)/(a\backslash c) &= \left(\begin{array}{cc} \text{ins}(p_2,c_2) & (p_2 < p_3) \\ \text{NOOP} & (p_3 \le p_2 < p_3+l_3) \\ \text{ins}(p_2-l_3,c_2) & (p_3+l_3 \le p_2) \end{array}\right) \Big/ \left(\begin{array}{cc} \text{ins}(p_1,c_1) & (p_1 < p_3) \\ \text{NOOP} & (p_3 \le p_1 < p_3+l_3) \\ \text{ins}(p_1-l_3,c_1) & (p_3+l_3 \le p_1) \end{array}\right) \\ &= \begin{cases} \text{ins}(p_2,c_2)/\text{ins}(p_1,c_1) & (p_2 < p_3) \\ \text{NOOP}/\text{NOOP} & (p_3 \le p_2 < p_3+l_3) \\ \text{ins}(p_2-l_3,c_2)/\text{ins}(p_1-l_3,c_1) & (p_3+l_3 \le p_2) \end{cases} \\ &= \begin{cases} \text{ins}(p_2+1,c_2) & (p_2 < p_3) \\ \text{NOOP} & (p_3 \le p_2 < p_3+l_3) \\ \text{ins}(p_2-l_3+1,c_2) & (p_3+l_3 \le p_2) \end{cases}, \end{aligned}$$

and we see that $(b/a)\backslash(c/a) = (b\backslash c)/(a\backslash c)$. Now suppose that $p_1 \ne p_2$. Then

$$\begin{aligned} (b/a)\backslash(c/a) &= (\text{ins}(p_2,c_2)/\text{ins}(p_1,c_1))\backslash(\text{del}(p_3,l_3)/\text{ins}(p_1,c_1)) \\ &= (\text{ins}(p_2,c_2)\backslash\text{ins}(p_1,c_1))/(\text{del}(p_3,l_3)\backslash\text{ins}(p_1,c_1)), \end{aligned}$$

which by Case 2 (interchange subscripts 1 and 3 and compare with $(b\backslash c)/(a\backslash c) = (b/a)\backslash(c/a)$ in Case 2) is equal to $(\text{ins}(p_2,c_2)/\text{del}(p_3,l_3))\backslash(\text{ins}(p_1,c_1)/\text{del}(p_3,l_3))$. Hence,

$$\begin{aligned} (b/a)\backslash(c/a) &= (\text{ins}(p_2,c_2)/\text{del}(p_3,l_3))\backslash(\text{ins}(p_1,c_1)/\text{del}(p_3,l_3)) \\ &= (\text{ins}(p_2,c_2)\backslash\text{del}(p_3,l_3))/(\text{ins}(p_1,c_1)\backslash\text{del}(p_3,l_3)) \\ &= (b\backslash c)/(a\backslash c) \end{aligned}$$

Thus, $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$.

$(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$: By the total ordering of a, b, and c, we have $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c/a)/(b/a)$ and $(c\hat{\ }b)\hat{\ }(a\hat{\ }b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned} (c/a)/(b/a) &= (\text{del}(p_3,l_3)/\text{ins}(p_1,c_1))/(\text{ins}(p_2,c_2)/\text{ins}(p_1,c_1)), \\ (c/b)/(a\backslash b) &= (\text{del}(p_3,l_3)/\text{ins}(p_2,c_2))/(\text{ins}(p_1,c_1)\backslash\text{ins}(p_2,c_2)). \end{aligned}$$

First assume that $p_1 = p_2$. Then we have

$$(c/a)/(b/a) = \left(\begin{array}{cc} \text{del}(p_3,l_3) & (p_3+l_3 \le p_1) \\ \text{del}(p_3,l_3+1) & (p_3 \le p_1 < p_3+l_3) \\ \text{del}(p_3+1,l_3) & (p_1 < p_3) \end{array}\right) \Big/ \text{ins}(p_2+1,c_2)$$

$$
\begin{aligned}
&= \begin{cases} \text{del}(p_3, l_3)/\text{ins}(p_2+1, c_2) & (p_3+l_3 \le p_1) \\ \text{del}(p_3, l_3+1)/\text{ins}(p_2+1, c_2) & (p_3 \le p_1 < p_3+l_3) \\ \text{del}(p_3+1, l_3)/\text{ins}(p_2+1, c_2) & (p_1 < p_3) \end{cases} \\
&= \begin{cases} \text{del}(p_3, l_3) & (p_3+l_3 \le p_1) \\ \text{del}(p_3, l_3+2) & (p_3 \le p_1 < p_3+l_3) \\ \text{del}(p_3+2, l_3) & (p_1 < p_3) \end{cases} .
\end{aligned}
$$

Also,

$$
\begin{aligned}
(c/b)/(a\backslash b) &= \left(\begin{array}{cc} \text{del}(p_3, l_3) & (p_3+l_3 \le p_2) \\ \text{del}(p_3, l_3+1) & (p_3 \le p_2 < p_3+l_3) \\ \text{del}(p_3+1, l_3) & (p_2 < p_3) \end{array} \right) \Big/ \text{ins}(p_1, c_1) \\
&= \begin{cases} \text{del}(p_3, l_3)/\text{ins}(p_1, c_1) & (p_3+l_3 \le p_2) \\ \text{del}(p_3, l_3+1)/\text{ins}(p_1, c_1) & (p_3 \le p_2 < p_3+l_3) \\ \text{del}(p_3+1, l_3)/\text{ins}(p_1, c_1) & (p_2 < p_3) \end{cases} \\
&= \begin{cases} \text{del}(p_3, l_3) & (p_3+l_3 \le p_2) \\ \text{del}(p_3, l_3+2) & (p_3 \le p_2 < p_3+l_3) \\ \text{del}(p_3+2, l_3) & (p_2 < p_3) \end{cases} ,
\end{aligned}
$$

and we see that $(c/a)/(b/a) = (c/b)/(a\backslash b)$. Now suppose that $p_1 \neq p_2$. Then we have

$$
\begin{aligned}
(c/a)/(b/a) &= (\text{del}(p_3, l_3)/\text{ins}(p_1, c_1))/(\text{ins}(p_2, c_2)/\text{ins}(p_1, c_1)) \\
&= (\text{del}(p_3, l_3)\backslash\text{ins}(p_1, c_1))\backslash(\text{ins}(p_2, c_2)\backslash\text{ins}(p_1, c_1)),
\end{aligned}
$$

which by Case 2 (interchange subscriptes 1 and 3 and compare with $(a\backslash c)\backslash(b\backslash c) = (a\backslash b)\backslash(c/b)$ in Case 2) is equal to $(\text{del}(p_3, l_3)\backslash\text{ins}(p_2, c_2))\backslash(\text{ins}(p_1, c_1)/\text{ins}(p_2, c_2)$. Hence,

$$
\begin{aligned}
(c/a)/(b/a) &= (\text{del}(p_3, l_3)\backslash\text{ins}(p_2, c_2))\backslash(\text{ins}(p_1, c_1)/\text{ins}(p_2, c_2) \\
&= (\text{del}(p_3, l_3)/\text{ins}(p_2, c_2))/(\text{ins}(p_1, c_1)\backslash\text{ins}(p_2, c_2) \\
&= (c/b)/(a\backslash b).
\end{aligned}
$$

Thus, $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$. This completes Case 4.

Case 5 $a = \mathit{del}(p_1, l_1)$, $b = \mathit{del}(p_2, l_2)$, $c = \mathit{ins}(p_3, c_3)$.

We must show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$: By the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash b)\backslash(c\backslash b)$. We have

$$
(a\backslash b)\backslash(c/b) = (\text{del}(p_1, l_1)\backslash\text{del}(p_2, l_2))\backslash(\text{ins}(p_3, c_3)/\text{del}(p_2, l_2))
$$

$$
= \left(\begin{array}{l}
\text{del}(p_1, l_1) \\
\qquad (p_1 + l_1 \leq p_2) \\
\text{del}(p_1, p_2 - p_1) \\
\qquad (p_1 \leq p_2 \leq p_1 + l_1 \leq p_2 + l_2) \\
\text{del}(p_1, l_1 - l_2) \\
\qquad (p_1 \leq p_2 \leq p_2 + l_2 \leq p_1 + l_1) \\
\text{NOOP} \\
\qquad (p_2 \leq p_1 \leq p_1 + l_1 \leq p_2 + l_2) \\
\text{del}(p_2, p_1 + l_1 - p_2 - l_2) \\
\qquad (p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1) \\
\text{del}(p_1 - l_2, l_1) \\
\qquad (p_2 + l_2 \leq p_1)
\end{array} \right) \Bigg\backslash \left(\begin{array}{c}
\text{ins}(p_3, c_3) \quad (p_3 < p_2) \\
\text{NOOP} \quad (p_2 \leq p_3 < p_2 + l_2) \\
\text{ins}(p_3 - l_2, c_3) \quad (p_3 \leq p_2 + l_2)
\end{array} \right),
$$

which evaluates as follows:

	$p_3 < p_2$	$p_2 \leq p_3 < p_2 + l_2$	$p_2 + l_2 \leq p_3$
$p_1 + l_1 \leq p_2$	$\text{del}(p_1, l_1)\backslash\text{ins}(p_3, c_3)$	$\text{del}(p_1, l_1)\backslash\text{NOOP}$	$\text{del}(p_1, l_1)$ $\backslash\text{ins}(p_3 - l_2, c_3)$
$p_1 \leq p_2 \leq$ $p_1 + l_1 \leq p_2 + l_2$	$\text{del}(p_1, p_2 - p_1)\backslash\text{ins}(p_3, c_3)$	$\text{del}(p_1, p_2 - p_1)\backslash\text{NOOP}$	$\text{del}(p_1, p_2 - p_1)$ $\backslash\text{ins}(p_3 - l_2, c_3)$
$p_1 \leq p_2 \leq$ $p_2 + l_2 \leq p_1 + l_1$	$\text{del}(p_1, l_1 - l_2)\backslash\text{ins}(p_3, c_3)$	$\text{del}(p_1, l_1 - l_2)\backslash\text{NOOP}$	$\text{del}(p_1, l_1 - l_2)$ $\backslash\text{ins}(p_3 - l_2, c_3)$
$p_2 \leq p_1 \leq$ $p_1 + l_1 \leq p_2 + l_2$	$\text{NOOP}\backslash\text{ins}(p_3, c_3)$	$\text{NOOP}\backslash\text{NOOP}$	$\text{NOOP}\backslash\text{ins}(p_3 - l_2, c_3)$
$p_2 \leq p_1 \leq$ $p_1 + l_1 \leq p_2 + l_2$	$\text{del}(p_2, p_1 + l_1 - p_2$ $-l_2)\backslash\text{ins}(p_3, c_3)$	$\text{del}(p_2, p_1 + l_1 - p_2$ $-l_2)\backslash\text{NOOP}$	$\text{del}(p_2, p_1 + l_1 - p_2$ $-l_2)\backslash\text{ins}(p_3 - l_2, c_3)$
$p_2 + l_2 \leq p_1$	$\text{del}(p_1 - l_2, l_1)\backslash\text{ins}(p_3, c_3)$	$\text{del}(p_1 - l_2, l_1)\backslash\text{NOOP}$	$\text{del}(p_1 - l_2, l_1)$ $\backslash\text{ins}(p_3 - l_2, c_3)$

$=$

	$p_3 < p_2$	$p_2 \le p_3$, $p_3 < p_2 + l_2$	$p_2 + l_2 \le p_3$
$p_1 + l_1 \le p_2$	del(p_1, l_1) $(p_1 + l_1 \le p_3)$ del$(p_1, l_1 + 1)$ $(p_1 \le p_3 < p_1 + l_1)$ del$(p_1 + 1, l_1)$ $(p_3 < p_1)$	del(p_1, l_1)	del(p_1, l_1) $(p_1 + l_1 \le p_3 - l_2)$ del$(p_1, l_1 + 1)$ $(p_1 \le p_3 - l_2 < p_1 + l_1)$ del$(p_1 + 1, l_1)$ $(p_3 - l_2 < p_1)$
$p_1 \le p_2 \le$ $p_1 + l_1 \le p_2 + l_2$	del$(p_1, p_2 - p_1)$ $(p_2 \le p_3)$ del$(p_1, p_2 - p_1 + 1)$ $(p_1 \le p_3 < p_2)$ del$(p_1 + 1, p_2 - p_1)$ $(p_3 < p_1)$	del$(p_1, p_2 - p_1)$	del$(p_1, p_2 - p_1)$ $(p_2 \le p_3 - l_2)$ del$(p_1, p_2 - p_1 + 1)$ $(p_1 \le p_3 - l_2 < p_2)$ del$(p_1 + 1, p_2 - p_1)$ $(p_3 - l_2 < p_1)$
$p_1 \le p_2 \le$ $p_2 + l_2 \le p_1 + l_1$	del$(p_1, l_1 - l_2)$ $(p_1 + l_1 - l_2 \le p_3)$ del$(p_1, l_1 - l_2 + 1)$ $(p_1 \le p_3 <$ $p_1 + l_1 - l_2)$ del$(p_1 + 1, l_1 - l_2)$ $(p_3 < p_1)$	del$(p_1, l_1 - l_2)$	del$(p_1, l_1 - l_2)$ $(p_1 + l_1 - l_2 \le p_3 - l_2)$ del$(p_1, l_1 - l_2 + 1)$ $(p_1 \le p_3 - l_2 <$ $p_1 + l_1 - l_2)$ del$(p_1 + 1, l_1 - l_2)$ $(p_3 - l_2 < p_1)$
$p_2 \le p_1 \le$ $p_1 + l_1 \le p_2 + l_2$	NOOP	NOOP	NOOP
$p_2 \le p_1 \le$ $p_1 + l_1 \le p_2 + l_2$	del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_1 + l_1 - l_2 \le p_3)$ del$(p_2, p_1 + l_1 - p_2$ $-l_2 + 1)$ $(p_2 \le p_3 <$ $p_1 + l_1 - l_2)$ del$(p_2 + 1, p_1 + l_1$ $-p_2 - l_2)$ $(p_3 < p_2)$	del$(p_2, p_1 + l_1$ $-p_2 - l_2)$	del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_1 + l_1 - l_2 \le p_3 - l_2)$ del$(p_2, p_1 + l_1 - p_2 - l_2 + 1)$ $(p_2 \le p_3 - l_2 <$ $p_1 + l_1 - l_2)$ del$(p_2 + 1, p_1 + l_1 - p_2 - l_2)$ $(p_3 - l_2 < p_2)$
$p_2 + l_2 \le p_1$	del$(p_1 - l_2, l_1)$ $(p_1 - l_2 + l_1 \le p_3)$ del$(p_1 - l_2, l_1 + 1)$ $(p_1 - l_2 \le p_3 <$ $p_1 - l_2 + l_1)$ del$(p_1 - l_2 + 1, l_1)$ $(p_3 < p_1 - l_2)$	del$(p_1 - l_2, l_1)$	del$(p_1 - l_2, l_1)$ $(p_1 - l_2 + l_1 \le p_3 - l_2)$ del$(p_1 - l_2, l_1 + 1)$ $(p_1 - l_2 \le p_3 - l_2 <$ $p_1 - l_2 + l_1)$ del$(p_1 - l_2 + 1, l_1)$ $(p_3 - l_2 < p_1 - l_2)$

=

	$p_3 < p_2$	$p_2 \le p_3$, $p_3 < p_2 + l_2$	$p_2 + l_2 \le p_3$
$p_1 + l_1 \le p_2$	del(p_1, l_1) $(p_1 + l_1 \le p_3)$ del$(p_1, l_1 + 1)$ $(p_1 \le p_3 < p_1 + l_1)$ del$(p_1 + 1, l_1)$ $(p_3 < p_1)$	del(p_1, l_1)	del(p_1, l_1)
$p_1 \le p_2 \le$ $p_1 + l_1 \le p_2 + l_2$	del$(p_1, p_2 - p_1 + 1)$ $(p_1 \le p_3)$ del$(p_1 + 1, p_2 - p_1)$ $(p_3 < p_1)$	del$(p_1, p_2 - p_1)$	del$(p_1, p_2 - p_1)$
$p_1 \le p_2 \le$ $p_2 + l_2 \le p_1 + l_1$	del$(p_1, l_1 - l_2 + 1)$ $(p_1 \le p_3)$ del$(p_1 + 1, l_1 - l_2)$ $(p_3 < p_1)$	del$(p_1, l_1 - l_2)$	del$(p_1, l_1 - l_2)$ $(p_1 + l_1 \le p_3)$ del$(p_1, l_1 - l_2 + 1)$ $(p_3 < p_1 + l_1)$
$p_2 \le p_1 \le$ $p_1 + l_1 \le p_2 + l_2$	NOOP	NOOP	NOOP
$p_2 \le p_1 \le$ $p_1 + l_1 \le p_2 + l_2$	del$(p_2 + 1,$ $p_1 + l_1 - p_2 - l_2)$	del$(p_2, p_1 + l_1$ $-p_2 - l_2)$	del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_1 + l_1 \le p_3)$ del$(p_2, p_1 + l_1 - p_2 - l_2 + 1)$ $(p_3 < p_1 + l_1)$
$p_2 + l_2 \le p_1$	del$(p_1 - l_2 + 1, l_1)$	del$(p_1 - l_2, l_1)$	del$(p_1 - l_2, l_1)$ $(p_1 + l_1 \le p_3)$ del$(p_1 - l_2, l_1 + 1)$ $(p_1 \le p_3 < p_1 + l_1)$ del$(p_1 - l_2 + 1, l_1)$ $(p_3 < p_1)$

Also,

$$\begin{aligned}(a\backslash c)\backslash(b\backslash c) &= (\mathrm{del}(p_1,l_1)\backslash \mathrm{ins}(p_3,c_3))\backslash(\mathrm{del}(p_2,l_2)\backslash \mathrm{ins}(p_3,c_3))\\ &= \begin{pmatrix} \mathrm{del}(p_1,l_1) \quad (p_1+l_1 \le p_3)\\ \mathrm{del}(p_1,l_1+1) \quad (p_1 \le p_3 < p_1+l_1)\\ \mathrm{del}(p_1+1,l_1) \quad (p_3 < p_1)\end{pmatrix} \Bigg\backslash \begin{pmatrix} \mathrm{del}(p_2,l_2) \quad (p_2+l_2 \le p_3)\\ \mathrm{del}(p_2,l_2+1) \quad (p_2 \le p_3 < p_2+l_2)\\ \mathrm{del}(p_2+1,l_2) \quad (p_3 < p_2)\end{pmatrix},\end{aligned}$$

which evaluates as follows:

	$p_2 + l_2 \le p_3$	$p_2 \le p_3 < p_2 + l_2$	$p_3 < p_2$
$p_1 + l_1 \le p_3$	del$(p_1, l_1)\backslash$del(p_2, l_2)	del$(p_1, l_1)\backslash$del$(p_2, l_2 + 1)$	del$(p_1, l_1)\backslash$del$(p_2 + 1, l_2)$
$p_1 \le p_3$ $< p_1 + l_1$	del$(p_1, l_1 + 1)\backslash$del(p_2, l_2)	del$(p_1, l_1 + 1)\backslash$del$(p_2, l_2 + 1)$	del$(p_1, l_1 + 1)\backslash$del$(p_2 + 1, l_2)$
$p_3 < p_1$	del$(p_1 + 1, l_1)\backslash$del(p_2, l_2)	del$(p_1 + 1, l_1)\backslash$del$(p_2, l_2 + 1)$	del$(p_1 + 1, l_1)\backslash$del$(p_2 + 1, l_2)$

		$p_2 + l_2 \le p_3$	$p_2 \le p_3 < p_2 + l_2$	$p_3 < p_2$
	$p_1 + l_1 \le p_3$	$\mathrm{del}(p_1, l_1)$ $(p_1 + l_1 \le p_2)$ $\mathrm{del}(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ $\mathrm{del}(p_1 - l_2, l_1)$ $(p_2 + l_2 \le p_1)$	$\mathrm{del}(p_1, l_1)$ $(p_1 + l_1 \le p_2)$ $\mathrm{del}(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_1, l_1 - l_2 - 1)$ $(p_1 \le p_2 \le p_2 + l_2 + 1$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_2,$ $p_1 + l_1 - p_2 - l_2 - 1)$ $(p_2 \le p_1 \le p_2 + l_2 + 1$ $\le p_1 + l_1)$ $\mathrm{del}(p_1 - l_2 - 1, l_1)$ $(p_2 + l_2 + 1 \le p_1)$	$\mathrm{del}(p_1, l_1)$ $(p_1 + l_1 \le p_2 + 1)$ $\mathrm{del}(p_1, p_2 - p_1 + 1)$ $(p_1 \le p_2 + 1 \le p_1 + l_1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_1, l_1 - l_2)$ $(p_1 \le p_2 + 1 \le p_2 + l_2$ $\le p_1 + l_1 + 1)$ NOOP $(p_2 + 1 \le p_1 \le p_1 + l_1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_2 + 1,$ $p_1 + l_1 - p_2 - l_2 - 1)$ $(p_2 \le p_1 - 1 \le p_2 + l_2$ $\le p_1 + l_1 - 1)$ $\mathrm{del}(p_1 - l_2, l_1)$ $(p_2 + l_2 + 1 \le p_1)$
$=$	$p_1 \le p_3$ $< p_1 + l_1$	$\mathrm{del}(p_1, l_1 + 1)$ $(p_1 + l_1 + 1 \le p_2)$ $\mathrm{del}(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1 + 1$ $\le p_2 + l_2)$ $\mathrm{del}(p_1, l_1 - l_2 + 1)$ $(p_1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1 + 1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1 + 1$ $\le p_2 + l_2)$ $\mathrm{del}(p_2,$ $p_1 + l_1 - p_2 - l_2 + 1)$ $(p_2 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1 + 1)$ $\mathrm{del}(p_1 - l_2, l_1 + 1)$ $(p_2 + l_2 \le p_1)$	$\mathrm{del}(p_1, l_1 + 1)$ $(p_1 + l_1 + 1 \le p_2)$ $\mathrm{del}(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1 + 1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2 + 1$ $\le p_1 + l_1 + 1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1 + 1$ $\le p_2 + l_2 + 1)$ $\mathrm{del}(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2 + 1$ $\le p_1 + l_1 + 1)$ $\mathrm{del}(p_1 - l_2 - 1, l_1 + 1)$ $(p_2 + l_2 + 1 \le p_1)$	$\mathrm{del}(p_1, l_1 + 1)$ $(p_1 + l_1 \le p_2)$ $\mathrm{del}(p_1, p_2 + 1 - p_1)$ $(p_1 - 1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_1, l_1 - l_2 + 1)$ $(p_1 - 1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1 - 1 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_2 + 1,$ $p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 - 1 \le p_2 + l_2$ $\le p_1 + l_1)$ $\mathrm{del}(p_1 - l_2, l_1 + 1)$ $(p_2 + l_2 + 1 \le p_1)$
	$p_3 < p_1$	$\mathrm{del}(p_1 + 1, l_1)$ $(p_1 + l_1 + 1 \le p_2)$ $\mathrm{del}(p_1 + 1, p_2 - p_1 - 1)$ $(p_1 \le p_2 - 1 \le p_1 + l_1$ $\le p_2 + l_2 - 1)$ $\mathrm{del}(p_1 + 1, l_1 - l_2)$ $(p_1 + 1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1 + 1)$ NOOP $(p_2 \le p_1 + 1$ $\le p_1 + l_1 + 1 \le p_2 + l_2)$ $\mathrm{del}(p_2, p_1 + l_1 + 1 - p_2 - l_2)$ $(p_2 \le p_1 + 1 \le p_2 + l_2$ $\le p_1 + l_1 + 1)$ $\mathrm{del}(p_1 - l_2 + 1, l_1)$ $(p_2 + l_2 \le p_1 + 1)$	$\mathrm{del}(p_1 + 1, l_1)$ $(p_1 + l_1 + 1 \le p_2)$ $\mathrm{del}(p_1 + 1, p_2 - p_1 - 1)$ $(p_1 \le p_2 - 1 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_1 + 1, l_1 - l_2 - 1)$ $(p_1 \le p_2 - 1 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 - 1 \le p_1$ $\le p_1 + l_1 \le p_2 + l_2)$ $\mathrm{del}(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_2 - 1 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ $\mathrm{del}(p_1 - l_2, l_1)$ $(p_2 + l_2 + 1 \le p_1 + 1)$	$\mathrm{del}(p_1 + 1, l_1)$ $(p_1 + l_1 \le p_2)$ $\mathrm{del}(p_1 + 1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ $\mathrm{del}(p_1 + 1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1$ $\le p_1 + l_1 \le p_2 + l_2)$ $\mathrm{del}(p_2 + 1,$ $p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ $\mathrm{del}(p_1 - l_2 + 1, l_1)$ $(p_2 + l_2 \le p_1)$

=

	$p_2 + l_2 \le p_3$	$p_2 \le p_3 < p_2 + l_2$	$p_3 < p_2$
$p_1 + l_1 \le p_3$	del(p_1, l_1) $(p_1 + l_1 \le p_2)$ del$(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ del$(p_1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1$ $\le p_1 + l_1 \le p_2 + l_2)$ del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ del$(p_1 - l_2, l_1)$ $(p_2 + l_2 \le p_1)$	del(p_1, l_1) $(p_1 + l_1 \le p_2)$ del$(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1)$ NOOP $(p_2 \le p_1)$	del(p_1, l_1)
$p_1 \le p_3$ $< p_1 + l_1$	del$(p_1, l_1 - l_2 + 1)$ $(p_1 \le p_2)$ del$(p_2,$ $p_1 + l_1 - p_2 - l_2 + 1)$ $(p_2 \le p_1 \le p_2 + l_2)$ del$(p_1 - l_2, l_1 + 1)$ $(p_2 + l_2 \le p_1)$	del$(p_1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1 + 1$ $\le p_2 + l_2 + 1)$ del$(p_1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2 + 1$ $\le p_1 + l_1 + 1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1 + 1$ $\le p_2 + l_2 + 1)$ del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2 + 1$ $\le p_1 + l_1 + 1)$	del$(p_1, l_1 + 1)$ $(p_1 + l_1 \le p_2)$ del$(p_1, p_2 + 1 - p_1)$ $(p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ del$(p_1, l_1 - l_2 + 1)$ $(p_2 + l_2 \le p_1 + l_1)$
$p_3 < p_1$	del$(p_1 - l_2 + 1, l_1)$	NOOP $(p_1 + l_1 \le p_2 + l_2)$ del$(p_2, p_1 + l_1 - p_2 - l_2)$ $(p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ del$(p_1 - l_2, l_1)$ $(p_2 + l_2 \le p_1)$	del$(p_1 + 1, l_1)$ $(p_1 + l_1 \le p_2)$ del$(p_1 + 1, p_2 - p_1)$ $(p_1 \le p_2 \le p_1 + l_1$ $\le p_2 + l_2)$ del$(p_1 + 1, l_1 - l_2)$ $(p_1 \le p_2 \le p_2 + l_2$ $\le p_1 + l_1)$ NOOP $(p_2 \le p_1 \le p_1 + l_1$ $\le p_2 + l_2)$ del$(p_2 + 1,$ $p_1 + l_1 - p_2 - l_2)$ $(p_2 \le p_1 \le p_2 + l_2$ $\le p_1 + l_1)$ del$(p_1 - l_2 + 1, l_1)$ $(p_2 + l_2 \le p_1)$

.

Comparing this table with the previous one, we see that they are equivalent. Hence, $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$: By the total ordering of a, b, and c, we have $(b\hat{}a)\hat{}(c\hat{}a) = (b/a)\backslash(c/a)$ and $(b\hat{}c)\hat{}(a\hat{}c) = (b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)/\mathrm{del}(p_1,l_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{del}(p_1,l_1)) \\ &= (\mathrm{del}(p_2,l_2)\backslash\mathrm{del}(p_1,l_1))\backslash(\mathrm{ins}(p_3,c_3)/\mathrm{del}(p_1,l_1)),\end{aligned}$$

which is the expression for $(a\backslash b)\backslash(c/b)$, with subscripts 1 and 2 interchanged. Hence, the table for $(b/a)\backslash(c/a)$ is as follows:

	$p_3 < p_1$	$p_1 \le p_3$, $p_3 < p_1 + l_1$	$p_1 + l_1 \le p_3$
$p_2 + l_2 \le p_1$	$\mathrm{del}(p_2, l_2)$ $(p_2 + l_2 \le p_3)$ $\mathrm{del}(p_2, l_2 + 1)$ $(p_2 \le p_3 < p_2 + l_2)$ $\mathrm{del}(p_2 + 1, l_2)$ $(p_3 < p_2)$	$\mathrm{del}(p_2, l_2)$	$\mathrm{del}(p_2, l_2)$
$p_2 \le p_1 \le$ $p_2 + l_2 \le p_1 + l_1$	$\mathrm{del}(p_2, p_1 - p_2 + 1)$ $(p_2 \le p_3)$ $\mathrm{del}(p_2 + 1, p_1 - p_2)$ $(p_3 < p_2)$	$\mathrm{del}(p_2, p_1 - p_2)$	$\mathrm{del}(p_2, p_1 - p_2)$
$p_2 \le p_1 \le$ $p_1 + l_1 \le p_2 + l_2$	$\mathrm{del}(p_2, l_2 - l_1 + 1)$ $(p_2 \le p_3)$ $\mathrm{del}(p_2 + 1, l_2 - l_1)$ $(p_3 < p_2)$	$\mathrm{del}(p_2, l_2 - l_1)$	$\mathrm{del}(p_2, l_2 - l_1)$ $(p_2 + l_2 \le p_3)$ $\mathrm{del}(p_2, l_2 - l_1 + 1)$ $(p_3 < p_2 + l_2)$
$p_1 \le p_2 \le$ $p_2 + l_2 \le p_1 + l_1$	NOOP	NOOP	NOOP
$p_1 \le p_2 \le$ $p_2 + l_2 \le p_1 + l_1$	$\mathrm{del}(p_1 + 1,$ $p_2 + l_2 - p_1 - l_1)$	$\mathrm{del}(p_1, p_2 + l_2$ $- p_1 - l_1)$	$\mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1)$ $(p_2 + l_2 \le p_3)$ $\mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1 + 1)$ $(p_3 < p_2 + l_2)$
$p_1 + l_1 \le p_2$	$\mathrm{del}(p_2 - l_1 + 1, l_2)$	$\mathrm{del}(p_2 - l_1, l_2)$	$\mathrm{del}(p_2 - l_1, l_2)$ $(p_2 + l_2 \le p_3)$ $\mathrm{del}(p_2 - l_1, l_2 + 1)$ $(p_2 \le p_3 < p_2 + l_2)$ $\mathrm{del}(p_2 - l_1 + 1, l_2)$ $(p_3 < p_2)$

Also,

$$\begin{aligned}(b\backslash c)/(a\backslash c) &= (\mathrm{del}(p_2, l_2)\backslash\mathrm{ins}(p_3, c_3))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{ins}(p_3, c_3)) \\ &= (\mathrm{del}(p_2, l_2)\backslash\mathrm{ins}(p_3, c_3))\backslash(\mathrm{del}(p_1, l_1)\backslash\mathrm{ins}(p_3, c_3)),\end{aligned}$$

which is exactly the expression for $(a\backslash c)\backslash(b\backslash c)$, with subscripts 1 and 2 interchanged. Thus, the table for $(b\backslash c)/(a\backslash c)$ is as follows:

	$p_1 + l_1 \leq p_3$	$p_1 \leq p_3 < p_1 + l_1$	$p_3 < p_1$
$p_2 + l_2 \leq p_3$	del(p_2, l_2) $(p_2 + l_2 \leq p_1)$ del$(p_2, p_1 - p_2)$ $(p_2 \leq p_1 \leq p_2 + l_2 \leq p_1 + l_1)$ del$(p_2, l_2 - l_1)$ $(p_2 \leq p_1 \leq p_1 + l_1 \leq p_2 + l_2)$ NOOP $(p_1 \leq p_2$ $\leq p_2 + l_2 \leq p_1 + l_1)$ del$(p_1, p_2 + l_2 - p_1 - l_1)$ $(p_1 \leq p_2 \leq p_1 + l_1$ $\leq p_2 + l_2)$ del$(p_2 - l_1, l_2)$ $(p_1 + l_1 \leq p_2)$	del(p_2, l_2) $(p_2 + l_2 \leq p_1)$ del$(p_2, p_1 - p_2)$ $(p_2 \leq p_1 \leq p_2 + l_2)$ NOOP $(p_1 \leq p_2)$	del(p_2, l_2)
$p_2 \leq p_3$ $< p_2 + l_2$	del$(p_2, l_2 - l_1 + 1)$ $(p_2 \leq p_1)$ del$(p_1,$ $p_2 + l_2 - p_1 - l_1 + 1)$ $(p_1 \leq p_2 \leq p_1 + l_1)$ del$(p_2 - l_1, l_2 + 1)$ $(p_1 + l_1 \leq p_2)$	del$(p_2, p_1 - p_2)$ $(p_2 \leq p_1 \leq p_2 + l_2 + 1$ $\leq p_1 + l_1 + 1)$ del$(p_2, l_2 - l_1)$ $(p_2 \leq p_1 \leq p_1 + l_1 + 1$ $\leq p_2 + l_2 + 1)$ NOOP $(p_1 \leq p_2 \leq p_2 + l_2 + 1$ $\leq p_1 + l_1 + 1)$ del$(p_1, p_2 + l_2 - p_1 - l_1)$ $(p_1 \leq p_2 \leq p_1 + l_1 + 1$ $\leq p_2 + l_2 + 1)$	del$(p_2, l_2 + 1)$ $(p_2 + l_2 \leq p_1)$ del$(p_2, p_1 + 1 - p_2)$ $(p_1 \leq p_2 + l_2$ $\leq p_1 + l_1)$ del$(p_2, l_2 - l_1 + 1)$ $(p_1 + l_1 \leq p_2 + l_2)$
$p_3 < p_2$	del$(p_2 - l_1 + 1, l_2)$	NOOP $(p_2 + l_2 \leq p_1 + l_1)$ del$(p_1, p_2 + l_2 - p_1 - l_1)$ $(p_2 \leq p_1 + l_1$ $\leq p_2 + l_2)$ del$(p_2 - l_1, l_2)$ $(p_1 + l_1 \leq p_2)$	del$(p_2 + 1, l_2)$ $(p_2 + l_2 \leq p_1)$ del$(p_2 + 1, p_1 - p_2)$ $(p_2 \leq p_1 \leq p_2 + l_2$ $\leq p_1 + l_1)$ del$(p_2 + 1, l_2 - l_1)$ $(p_2 \leq p_1 \leq p_1 + l_1$ $\leq p_2 + l_2)$ NOOP $(p_1 \leq p_2 \leq p_2 + l_2$ $\leq p_1 + l_1)$ del$(p_1 + 1,$ $p_2 + l_2 - p_1 - l_1)$ $(p_1 \leq p_2 \leq p_1 + l_1$ $\leq p_2 + l_2)$ del$(p_2 - l_1 + 1, l_2)$ $(p_1 + l_1 \leq p_2)$

Comparing this table with the previous one, we see that they are equivalent. Hence, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$.

$(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$: By the total ordering of a, b, and c, we have $(c\hat{}a)\hat{}(b\hat{}a) = (c/a)/(b/a)$ and $(c\hat{}b)\hat{}(a\hat{}b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned}
(c/a)/(b/a) &= (\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_1, l_1))/(\mathrm{del}(p_2, l_2)/\mathrm{del}(p_1, l_1)) \\
&= \left(\begin{array}{c} \mathrm{ins}(p_3, c_3) \quad (p_3 < p_1) \\ \mathrm{NOOP} \quad (p_1 \le p_3 < p_1 + l_1) \\ \mathrm{ins}(p_3 - l_1, c_3) \quad (p_1 + l_1 \le p_3) \end{array}\right) \Bigg/ \left(\begin{array}{l} \mathrm{del}(p_2, l_2) \quad (p_2 + l_2 \le p_1) \\ \mathrm{del}(p_2, p_1 - p_2) \\ \qquad (p_2 \le p_1 \le p_2 + l_2 \le p_1 + l_1) \\ \mathrm{del}(p_2, l_2 - l_1) \\ \qquad (p_2 \le p_1 \le p_1 + l_1 \le p_2 + l_2) \\ \mathrm{NOOP} \\ \qquad (p_1 \le p_2 \le p_2 + l_2 \le p_1 + l_1) \\ \mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1) \\ \qquad (p_1 \le p_2 \le p_1 + l_1 \le p_2 + l_2) \\ \mathrm{del}(p_2 - l_1, l_2) \quad (p_1 + l_1 \le p_2) \end{array}\right),
\end{aligned}$$

which evaluates as follows:

	$p_3 < p_1$	$p_1 \le p_3 < p_1 + l_1$	$p_1 + l_1 \le p_3$
$p_2 + l_2 \le p_1$	$\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_2, l_2)$	$\mathrm{NOOP}/\mathrm{del}(p_2, l_2)$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{del}(p_2, l_2)$
$p_2 \le p_1 \le p_2 + l_2 \le p_1 + l_1$	$\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_2, p_1 - p_2)$	$\mathrm{NOOP}/\mathrm{del}(p_2, p_1 - p_2)$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{del}(p_2, p_1 - p_2)$
$p_2 \le p_1 \le p_1 + l_1 \le p_2 + l_2$	$\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_2, l_2 - l_1)$	$\mathrm{NOOP}/\mathrm{del}(p_2, l_2 - l_1)$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{del}(p_2, l_2 - l_1)$
$p_1 \le p_2 \le p_2 + l_2 \le p_1 + l_1$	$\mathrm{ins}(p_3, c_3)/\mathrm{NOOP}$	$\mathrm{NOOP}/\mathrm{NOOP}$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{NOOP}$
$p_1 \le p_2 \le p_1 + l_1 \le p_2 + l_2$	$\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1)$	$\mathrm{NOOP}/\mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1)$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{del}(p_1, p_2 + l_2 - p_1 - l_1)$
$p_1 + l_1 \le p_2$	$\mathrm{ins}(p_3, c_3)/\mathrm{del}(p_2 - l_1, l_2)$	$\mathrm{NOOP}/\mathrm{del}(p_2 - l_1, l_2)$	$\mathrm{ins}(p_3 - l_1, c_3)/\mathrm{del}(p_2 - l_1, l_2)$

=

	$p_3 < p_1$	$p_1 \le p_3 < p_1 + l_1$	$p_1 + l_1 \le p_3$
$p_2 + l_2 \le p_1$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3 < p_2 + l_2)$ ins$(p_3 - l_2, c_3)$ $(p_2 + l_2 \le p_3)$	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 - l_1 < p_2)$ NOOP $(p_2 \le p_3 - l_1 < p_2 + l_2)$ ins$(p_3 - l_1 - l_2, c_3)$ $(p_2 + l_2 \le p_3 - l_1)$
$p_2 \le p_1 \le p_2 + l_2 \le p_1 + l_1$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3 < p_1)$ ins$(p_3 - p_1 + p_2, c_3)$ $(p_1 \le p_3)$	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 - l_1 < p_2)$ NOOP $(p_2 \le p_3 - l_1 < p_1)$ ins$(p_3 - l_1 - p_1 + p_2, c_3)$ $(p_1 \le p_3 - l_1)$
$p_2 \le p_1 \le p_1 + l_1 \le p_2 + l_2$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3 < p_2 + l_2 - l_1)$ ins$(p_3 - l_2 + l_1, c_3)$ $(p_2 + l_2 - l_1 \le p_3)$	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 - l_1 < p_2)$ NOOP $(p_2 \le p_3 - l_1 < p_2 + l_2 - l_1)$ ins$(p_3 - l_2, c_3)$ $(p_2 + l_2 - l_1 \le p_3 - l_1)$
$p_1 \le p_2 \le p_2 + l_2 \le p_1 + l_1$	ins(p_3, c_3)	NOOP	ins$(p_3 - l_1, c_3)$
$p_1 \le p_2 \le p_1 + l_1 \le p_2 + l_2$	ins(p_3, c_3) $(p_3 < p_1)$ NOOP $(p_1 \le p_3 < p_2 + l_2 - l_1)$ ins$(p_3 - p_2 - l_2 + p_1 + l_1, c_3)$ $(p_2 + l_2 - l_1 \le p_3)$	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 - l_1 < p_1)$ NOOP $(p_1 \le p_3 - l_1 < p_2 + l_2 - l_1)$ ins$(p_3 - p_2 - l_2 + p_1, c_3)$ $(p_2 + l_2 - l_1 \le p_3 - l_1)$
$p_1 + l_1 \le p_2$	ins(p_3, c_3) $(p_3 < p_2 - l_1)$ NOOP $(p_2 - l_1 \le p_3 < p_2 - l_1 + l_2)$ ins$(p_3 - l_2, c_3)$ $(p_2 - l_1 + l_2 \le p_3)$	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 - l_1 < p_2 - l_1)$ NOOP $(p_2 - l_1 \le p_3 - l_1 < p_2 - l_1 + l_2)$ ins$(p_3 - l_1 - l_2, c_3)$ $(p_2 - l_1 + l_2 \le p_3 - l_1)$

=

	$p_3 < p_1$	$p_1 \le p_3 < p_1 + l_1$	$p_1 + l_1 \le p_3$
$p_2 + l_2 \le p_1$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3 < p_2 + l_2)$ ins$(p_3 - l_2, c_3)$ $(p_2 + l_2 \le p_3)$	NOOP	ins$(p_3 - l_1 - l_2, c_3)$
$p_2 \le p_1 \le p_2 + l_2 \le p_1 + l_1$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3)$	NOOP	ins$(p_3 - l_1 - p_1 + p_2, c_3)$
$p_2 \le p_1 \le p_1 + l_1 \le p_2 + l_2$	ins(p_3, c_3) $(p_3 < p_2)$ NOOP $(p_2 \le p_3)$	NOOP	NOOP $(p_3 < p_2 + l_2)$ ins$(p_3 - l_2, c_3)$ $(p_2 + l_2 \le p_3)$
$p_1 \le p_2 \le p_2 + l_2 \le p_1 + l_1$	ins(p_3, c_3)	NOOP	ins$(p_3 - l_1, c_3)$
$p_1 \le p_2 \le p_1 + l_1 \le p_2 + l_2$	ins(p_3, c_3)	NOOP	NOOP $(p_3 < p_2 + l_2)$ ins$(p_3 - p_2 - l_2 + p_1, c_3)$ $(p_2 + l_2 \le p_3)$
$p_1 + l_1 \le p_2$	ins(p_3, c_3)	NOOP	ins$(p_3 - l_1, c_3)$ $(p_3 < p_2)$ NOOP $(p_2 \le p_3 < p_2 + l_2)$ ins$(p_3 - l_1 - l_2, c_3)$ $(p_2 + l_2 \le p_3)$

.

Also,

$$\begin{aligned}(c/b)/(a\backslash b) &= (\text{ins}(p_3,c_3)/\text{del}(p_2,l_2))/(\text{del}(p_1,l_1)\backslash\text{del}(p_2,l_2)) \\ &= (\text{ins}(p_3,c_3)/\text{del}(p_2,l_2))/(\text{del}(p_1,l_1)/\text{del}(p_2,l_2)),\end{aligned}$$

which is exactly the expression for $(c/a)/(b/a)$, with subscripts 1 and 2 interchanged. Hence the table for $(c/b)/(a\backslash b)$ is as follows:

	$p_3 < p_2$	$p_2 \le p_3 < p_2 + l_2$	$p_2 + l_2 \le p_3$
$p_1 + l_1 \le p_2$	$\text{ins}(p_3, c_3)$ $(p_3 < p_1)$ NOOP $(p_1 \le p_3 < p_1 + l_1)$ $\text{ins}(p_3 - l_1, c_3)$ $(p_1 + l_1 \le p_3)$	NOOP	$\text{ins}(p_3 - l_1 - l_2, c_3)$
$p_1 \le p_2 \le p_1 + l_1 \le p_2 + l_2$	$\text{ins}(p_3, c_3)$ $(p_3 < p_1)$ NOOP $(p_1 \le p_3)$	NOOP	$\text{ins}(p_3 - l_2 - p_2 + p_1, c_3)$
$p_1 \le p_2 \le p_2 + l_2 \le p_1 + l_1$	$\text{ins}(p_3, c_3)$ $(p_3 < p_1)$ NOOP $(p_1 \le p_3)$	NOOP	NOOP $(p_3 < p_1 + l_1)$ $\text{ins}(p_3 - l_1, c_3)$ $(p_1 + l_1 \le p_3)$
$p_2 \le p_1 \le p_1 + l_1 \le p_2 + l_2$	$\text{ins}(p_3, c_3)$	NOOP	$\text{ins}(p_3 - l_2, c_3)$
$p_2 \le p_1 \le p_2 + l_2 \le p_1 + l_1$	$\text{ins}(p_3, c_3)$	NOOP	NOOP $(p_3 < p_1 + l_1)$ $\text{ins}(p_3 - p_1 - l_1 + p_2, c_3)$ $(p_1 + l_1 \le p_3)$
$p_2 + l_2 \le p_1$	$\text{ins}(p_3, c_3)$	NOOP	$\text{ins}(p_3 - l_2, c_3)$ $(p_3 < p_1)$ NOOP $(p_1 \le p_3 < p_1 + l_1)$ $\text{ins}(p_3 - l_2 - l_1, c_3)$ $(p_1 + l_1 \le p_3)$

Comparing this table with the previous one, we see that they are equivalent. Hence, $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$. This completes Case 5.

Case 6 *$a = del(p_1, l_1)$, $b = ins(p_2, c_2)$, $c = del(p_3, l_3)$.*

We must show that $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$, $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$, and $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$.

$(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$: By the total ordering of a, b, and c, we have $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{\ }c)\hat{\ }(b\hat{\ }c) = (a\backslash c)\backslash(b\backslash c)$. We have

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\text{del}(p_1,l_1)\backslash\text{ins}(p_2,c_2))\backslash(\text{del}(p_3,l_3)/\text{ins}(p_2,c_2)) \\ &= (\text{del}(p_1,l_1)\backslash\text{ins}(p_2,c_2))\backslash(\text{del}(p_3,l_3)\backslash\text{ins}(p_2,c_2)),\end{aligned}$$

which by Case 5 (interchange subscripts 2 and 3, and compare with $(a\backslash c)\backslash(b\backslash c) = (a\backslash b)\backslash(c/b)$ in Case 5) is equal to $(\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))\backslash(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3))$. Hence,

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))\backslash(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3))\\ &= (\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))\backslash(\mathrm{ins}(p_2, c_2)\backslash\mathrm{del}(p_3, l_3))\\ &= (a\backslash c)\backslash(b/c).\end{aligned}$$

Thus, $(a\hat{\ }b)\hat{\ }(c\hat{\ }b) = (a\hat{\ }c)\hat{\ }(b\hat{\ }c)$.

$(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$: By the total ordering of a, b, and c, we have $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b/a)\backslash(c/a)$ and $(b\hat{\ }c)\hat{\ }(a\hat{\ }c) = (b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_1, l_1))\backslash(\mathrm{del}(p_3, l_3)/\mathrm{del}(p_1, l_1))\\ &= (\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_1, l_1))/(\mathrm{del}(p_3, l_3)/\mathrm{del}(p_1, l_1)),\end{aligned}$$

which by Case 5 (interchange subscripts 2 and 3, and compare with $(c/a)/(b/a) = (c/b)/(a\backslash b)$ in Case 5) is equal to $(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))$. Hence,

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_3, l_3))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))\\ &= (\mathrm{ins}(p_2, c_2)\backslash\mathrm{del}(p_3, l_3))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{del}(p_3, l_3))\\ &= (b\backslash c)/(a\backslash c).\end{aligned}$$

Hence, $(b\hat{\ }a)\hat{\ }(c\hat{\ }a) = (b\hat{\ }c)\hat{\ }(a\hat{\ }c)$.

$(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$: By the total ordering of a, b, and c, we have $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c/a)/(b/a)$ and $(c\hat{\ }b)\hat{\ }(a\hat{\ }b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned}(c/a)/(b/a) &= (\mathrm{del}(p_3, l_3)/\mathrm{del}(p_1, l_1))/(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_1, l_1))\\ &= (\mathrm{del}(p_3, l_3)/\mathrm{del}(p_1, l_1))\backslash(\mathrm{ins}(p_2, c_2)/\mathrm{del}(p_1, l_1)),\end{aligned}$$

which by Case 5 (interchange subscripts 2 and 3, and compare with $(b/a)\backslash(c/a) = (b\backslash c)/(a\backslash c)$ in Case 5) is equal to $(\mathrm{del}(p_3, l_3)\backslash\mathrm{ins}(p_2, c_2))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{ins}(p_2, c_2))$. Hence,

$$\begin{aligned}(c/a)/(b/a) &= (\mathrm{del}(p_3, l_3)\backslash\mathrm{ins}(p_2, c_2))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{ins}(p_2, c_2))\\ &= (\mathrm{del}(p_3, l_3)/\mathrm{ins}(p_2, c_2))/(\mathrm{del}(p_1, l_1)\backslash\mathrm{ins}(p_2, c_2))\\ &= (c/b)/(a\backslash b).\end{aligned}$$

Hence $(c\hat{\ }a)\hat{\ }(b\hat{\ }a) = (c\hat{\ }b)\hat{\ }(a\hat{\ }b)$. This completes Case 6.

Case 7 $a = \mathit{ins}(p_1, c_1)$, $b = \mathit{del}(p_2, l_2)$, $c = \mathit{del}(p_3, l_3)$.

We must show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$: By the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash c)\backslash(b\backslash c)$. We have

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2))\backslash(\mathrm{del}(p_3,l_3)/\mathrm{del}(p_2,l_2))\\ &= (\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_2,l_2))/(\mathrm{del}(p_3,l_3)\backslash\mathrm{del}(p_2,l_2)),\end{aligned}$$

which by Case 5 (interchange subscripts 1 and 3, and compare with $(c/b)/(a\backslash b) = (c/a)/(b/a)$ in Case 5) is equal to $(\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_3,l_3))/(\mathrm{del}(p_2,l_2)\backslash\mathrm{del}(p_3,l_3))$. Hence,

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_3,l_3))/(\mathrm{del}(p_2,l_2)\backslash\mathrm{del}(p_3,l_3))\\ &= (\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_3,l_3))\backslash(\mathrm{del}(p_2,l_2)\backslash\mathrm{del}(p_3,l_3))\\ &= (a\backslash c)\backslash(b\backslash c).\end{aligned}$$

Thus, $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$: By the total ordering of a, b, and c, we have $(b\hat{}a)\hat{}(c\hat{}a) = (b/a)\backslash(c/a)$ and $(b\hat{}c)\hat{}(a\hat{}c) = (b\backslash c)/(a\backslash c)$. We have

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1))\backslash(\mathrm{del}(p_3,l_3)/\mathrm{ins}(p_1,c_1))\\ &= (\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_1,c_1))/(\mathrm{del}(p_3,l_3)\backslash\mathrm{ins}(p_1,c_1)),\end{aligned}$$

which by Case 5 (interchange subscripts 1 and 3, and compare with $(b\backslash c)/(a\backslash c) = (b/a)\backslash(c/a)$ in Case 5) is equal to $(\mathrm{del}(p_2,l_2)/\mathrm{del}(p_3,l_3))\backslash(\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_3,l_3))$. Hence,

$$\begin{aligned}(b/a)\backslash(c/a) &= (\mathrm{del}(p_2,l_2)/\mathrm{del}(p_3,l_3))\backslash(\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_3,l_3))\\ &= (\mathrm{del}(p_2,l_2)\backslash\mathrm{del}(p_3,l_3))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_3,l_3))\\ &= (b\backslash c)/(a\backslash c).\end{aligned}$$

Thus, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$.

$(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$*:* By the total ordering of a, b, and c, we have $(c\hat{}a)\hat{}(b\hat{}a) = (c/a)/(b/a)$ and $(c\hat{}b)\hat{}(a\hat{}b) = (c/b)/(a\backslash b)$. We have

$$\begin{aligned}(c/a)/(b/a) &= (\mathrm{del}(p_3,l_3)/\mathrm{ins}(p_1,c_1))/(\mathrm{del}(p_2,l_2)/\mathrm{ins}(p_1,c_1)) \\ &= (\mathrm{del}(p_3,l_3)\backslash\mathrm{ins}(p_1,c_1))\backslash(\mathrm{del}(p_2,l_2)\backslash\mathrm{ins}(p_1,c_1)),\end{aligned}$$

which by Case 5 (interchange subscripts 1 and 3, and compare with $(a\backslash c)\backslash(b\backslash c) = (a\backslash b)\backslash(c/b)$ in Case 5) is equal to $(\mathrm{del}(p_3,l_3)\backslash\mathrm{del}(p_2,l_2))\backslash(\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_2,l_2))$. Hence,

$$\begin{aligned}(c/a)/(b/a) &= (\mathrm{del}(p_3,l_3)\backslash\mathrm{del}(p_2,l_2))\backslash(\mathrm{ins}(p_1,c_1)/\mathrm{del}(p_2,l_2)) \\ &= (\mathrm{del}(p_3,l_3)/\mathrm{del}(p_2,l_2))/(\mathrm{ins}(p_1,c_1)\backslash\mathrm{del}(p_2,l_2)) \\ &= (c/b)/(a\backslash b).\end{aligned}$$

Thus, $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$. This completes Case 7.

Case 8 *$a = del(p_1,l_1)$, $b = del(p_2,l_2)$, $c = del(p_3,l_3)$*

For the case where all three concurrent updates are deletions, we are effectively working in the set of operations

$$\{\mathrm{del}(p,l)|\ p \geq 1, l \geq 1\} \cup \{\mathrm{NOOP}\}$$

Any operation in this set can be represented as a composition of operations in the set

$$\{\mathrm{d}(p)|\ p \geq 1\} \cup \{\mathrm{NOOP}\},$$

where the operation $\mathrm{d}(p)$ deletes a single character at position p. The transformation rules on deletions then simplify as follows:

$$\begin{aligned}\mathrm{d}(p_1)/\mathrm{d}(p_2) &= \begin{cases}\mathrm{d}(p_1) & (p_1 < p_2) \\ \mathrm{NOOP} & (p_1 = p_2) \\ \mathrm{d}(p_1 - 1) & (p_1 > p_2)\end{cases} \\ \mathrm{d}(p_1)\backslash\mathrm{d}(p_2) &= \mathrm{d}(p_1)/\mathrm{d}(p_2)\end{aligned}$$

By Theorem 3.14, we can verify Case 8 by working in this simplified set of operations. We need to show that $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$, and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$.

$(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$: By the total ordering of a, b, and c, we have $(a\hat{}b)\hat{}(c\hat{}b) = (a\backslash b)\backslash(c/b)$ and $(a\hat{}c)\hat{}(b\hat{}c) = (a\backslash c)\backslash(b\backslash b)$. We have

$$\begin{aligned}(a\backslash b)\backslash(c/b) &= (\mathrm{d}(p_1)\backslash\mathrm{d}(p_2))\backslash(\mathrm{d}(p_3)/\mathrm{d}(p_2)) \\ &= \left(\begin{array}{cc} \mathrm{d}(p_1) & (p_1 < p_2) \\ \mathrm{NOOP} & (p_1 = p_2) \\ \mathrm{d}(p_1-1) & (p_1 > p_2)\end{array}\right)\Bigg\backslash\left(\begin{array}{cc} \mathrm{d}(p_3) & (p_3 < p_2) \\ \mathrm{NOOP} & (p_3 = p_2) \\ \mathrm{d}(p_3-1) & (p_3 > p_2)\end{array}\right),\end{aligned}$$

which evaluates as follows:

	$p_3 < p_2$	$p_3 = p_2$	$p_3 > p_2$
$p_1 < p_2$	$\mathrm{d}(p_1)\backslash\mathrm{d}(p_3)$	$\mathrm{d}(p_1)\backslash\mathrm{NOOP}$	$\mathrm{d}(p_1)\backslash\mathrm{d}(p_3-1)$
$p_1 = p_2$	$\mathrm{NOOP}\backslash\mathrm{d}(p_3)$	$\mathrm{NOOP}\backslash\mathrm{NOOP}$	$\mathrm{NOOP}\backslash\mathrm{d}(p_3-1)$
$p_1 > p_2$	$\mathrm{d}(p_1-1)\backslash\mathrm{d}(p_3)$	$\mathrm{d}(p_1-1)\backslash\mathrm{NOOP}$	$\mathrm{d}(p_1-1)\backslash\mathrm{d}(p_3-1)$

$=$

	$p_3 < p_2$	$p_3 = p_2$	$p_3 > p_2$
$p_1 < p_2$	$\begin{array}{cc} \mathrm{d}(p_1) & (p_1 < p_3) \\ \mathrm{NOOP} & (p_1 = p_3) \\ \mathrm{d}(p_1-1) & (p_1 > p_3)\end{array}$	$\mathrm{d}(p_1)$	$\begin{array}{cc} \mathrm{d}(p_1) & (p_1 < p_3-1) \\ \mathrm{NOOP} & (p_1 = p_3-1) \\ \mathrm{d}(p_1-1) & (p_1 > p_3-1)\end{array}$
$p_1 = p_2$	NOOP	NOOP	NOOP
$p_1 > p_2$	$\begin{array}{cc} \mathrm{d}(p_1-1) & (p_1-1 < p_3) \\ \mathrm{NOOP} & (p_1-1 = p_3) \\ \mathrm{d}(p_1-2) & (p_1-1 > p_3)\end{array}$	$\mathrm{d}(p_1-1)$	$\begin{array}{cc} \mathrm{d}(p_1-1) & (p_1-1 < p_3-1) \\ \mathrm{NOOP} & (p_1-1 = p_3-1) \\ \mathrm{d}(p_1-2) & (p_1-1 > p_3-1)\end{array}$

$=$

	$p_3 < p_2$	$p_3 = p_2$	$p_3 > p_2$
$p_1 < p_2$	$\begin{array}{cc} \mathrm{d}(p_1) & (p_1 < p_3) \\ \mathrm{NOOP} & (p_1 = p_3) \\ \mathrm{d}(p_1-1) & (p_1 > p_3)\end{array}$	$\mathrm{d}(p_1)$	$\mathrm{d}(p_1)$
$p_1 = p_2$	NOOP	NOOP	NOOP
$p_1 > p_2$	$\mathrm{d}(p_1-2)$	$\mathrm{d}(p_1-1)$	$\begin{array}{cc} \mathrm{d}(p_1-1) & (p_1 < p_3) \\ \mathrm{NOOP} & (p_1 = p_3) \\ \mathrm{d}(p_1-2) & (p_1 > p_3)\end{array}$

.

Also,

$$(a\backslash c)\backslash(b\backslash c) = (a\backslash c)\backslash(b/c),$$

which is exactly the expression for $(a\backslash b)\backslash(c/b)$, with subscripts 2 and 3 interchanged. Hence, the table for $(a\backslash c)\backslash(b\backslash c)$ is as follows:

	$p_2 < p_3$	$p_2 = p_3$	$p_2 > p_3$
$p_1 < p_3$	$\begin{array}{cc} \mathrm{d}(p_1) & (p_1 < p_2) \\ \mathrm{NOOP} & (p_1 = p_2) \\ \mathrm{d}(p_1-1) & (p_1 > p_2)\end{array}$	$\mathrm{d}(p_1)$	$\mathrm{d}(p_1)$
$p_1 = p_3$	NOOP	NOOP	NOOP
$p_1 > p_3$	$\mathrm{d}(p_1-2)$	$\mathrm{d}(p_1-1)$	$\begin{array}{cc} \mathrm{d}(p_1-1) & (p_1 < p_2) \\ \mathrm{NOOP} & (p_1 = p_2) \\ \mathrm{d}(p_1-2) & (p_1 > p_2)\end{array}$

.

Comparing this table with the previous one, we see that they are equivalent. Hence, $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$.

$(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$ and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$: Since all three of a, b, and c are deletes, and / and \ are identical over deletes, the task of verifying these two conditions is indistinguishable from the task of verifying $(a\hat{}b)\hat{}(c\hat{}b) = (a\hat{}c)\hat{}(b\hat{}c)$. Thus, $(b\hat{}a)\hat{}(c\hat{}a) = (b\hat{}c)\hat{}(a\hat{}c)$ and $(c\hat{}a)\hat{}(b\hat{}a) = (c\hat{}b)\hat{}(a\hat{}b)$. This completes Case 8.

Our verification of TP2 for text buffer operations is now complete.

Appendix B

Source Code

In this appendix, we give the source code for the CCU library we discussed in Chapter 4 and the text buffer we constructed in Chapter 5.

B.1 CCU Library

B.1.1 Timestamp Module

```
signature TIMESTAMP = sig
   exception Incompatible
   exception Range

   eqtype timestamp

   val mktimestamp : int -> timestamp
   val size : timestamp -> int
   val inc : (timestamp * int) -> timestamp
   val causalLT : (timestamp * timestamp) -> bool
   val totalLT : (timestamp * timestamp) -> bool
   val sup : (timestamp * timestamp) -> timestamp
   val inf : (timestamp * timestamp) -> timestamp
   val toString : timestamp -> string
end

structure Timestamp :> TIMESTAMP = struct
   exception Incompatible and Range
```

```
datatype timestamp = TS of int*int list

local
    fun mklist 0 = nil
    |   mklist n = 0::mklist(n-1)
in
    fun mktimestamp n = TS(n, mklist n)
end

fun size (TS(n, _)) = n

local
    fun inclist nil _ = raise Range
    |   inclist (x::xs) 1 = (x+1)::xs
    |   inclist (x::xs) n = x::(inclist xs (n-1))
in
    fun inc (TS(n, L), m) =
        if m < 1 orelse m > n then
            raise Range
        else
            TS(n, inclist L m)
end

fun equal(TS(a,b), TS(c,d)) = if a <> c then raise Incompatible
                              else b = d

fun causalLE(TS(a,nil), TS(c,nil)) = true
|   causalLE(TS(a,x::xs), TS(c,y::ys)) =
    if a <> c then raise Incompatible
    else
        if x <= y then causalLE(TS(a,xs),TS(c,ys))
        else false
|   causalLE(_,_) = raise Incompatible

fun causalLT(x,y) = causalLE(x,y) andalso (not (equal(x,y)))

fun totalLT(TS(a,nil), TS(c,nil)) = false
|   totalLT(TS(a, x::xs), TS(c, y::ys)) =
    if a <> c then raise Incompatible
    else
```

```
          if x < y then true
          else if x > y then false
               else totalLT(TS(a, xs),TS(c, ys))
   |   totalLT(_,_) = raise Incompatible

   fun max(a,b) = if a < b then b else a
   fun min(a,b) = if a > b then b else a

   fun sup(TS(a,nil), TS(c,nil)) = TS(a, nil)
   |   sup(TS(a, b::bs), TS(c, d::ds)) =
       if a <> c then raise Incompatible
       else
          let val TS(x, y) = sup(TS(a, bs), TS(c, ds))
          in TS(a, max(b, d)::y)
          end
   |   sup(_,_) = raise Incompatible

   fun inf(TS(a,nil), TS(c,nil)) = TS(a, nil)
   |   inf(TS(a, b::bs), TS(c, d::ds)) =
       if a <> c then raise Incompatible
       else
          let val TS(x, y) = inf(TS(a, bs), TS(c, ds))
          in TS(a, min(b, d)::y)
          end
   |   inf(_,_) = raise Incompatible

   fun toString (TS(_,x)) =
       let
          fun toString2 nil = ""
          |   toString2 [n] = Int.toString n
          |   toString2 (n::ns) = Int.toString n ^ "," ^ toString2 ns
       in
          "(" ^ toString2 x ^ ")"
       end
end
```

B.1.2 Network Abstraction

```
(* Our purpose here is to reimplement the MULTICAST signature to emulate
   communication over a network.  We do this by providing modified versions
   of the functions multicast and port that introduce random delays, thus
   destroying the FIFO ordering of events. *)

structure Netcast:MULTICAST = struct
    structure C = CML
    structure M = Multicast
    structure R = Random
    val r = R.rand(3,4)  (* Arbitrarily chosen integers to set the seed *)
    type 'a event = 'a M.event
    type 'a mchan = 'a M.mchan
    type 'a port = 'a Mailbox.mbox
    val mChannel = M.mChannel

    fun multicast (ch, a) = ignore (
      C.spawn (fn () => (
        C.sync(
          C.timeOutEvt(
            Time.fromMicroseconds(Int32.fromInt(R.randRange(1,100000) r))
          )
        );
        M.multicast(ch, a)
      ))
    )

    fun port mc =
        let
           val mb = Mailbox.mailbox()
           val p = M.port mc
           fun server () =
              let
                 val x = M.recv p
              in
                 C.spawn (fn () => (
                   C.sync (
                     C.timeOutEvt (
                       Time.fromMicroseconds(
                         Int32.fromInt(R.randRange(1,10000) r)
                       )
                     )
                   );
```

```
                  Mailbox.send(mb, x)
                ));
                server ()
             end
        in
           C.spawn server;
           mb
        end

        val recv = Mailbox.recv
        val recvEvt = Mailbox.recvEvt
        exception NotSupported
        fun copy _ = raise NotSupported
end
```

B.1.3 CCUOBJ and CCUAPI Signatures

```
signature CCUOBJ = sig
    type state
    val stateToString : state -> string

    eqtype operation
    val operationToString : operation -> string
    val apply : (operation * state) -> state
    val / : (operation * operation) -> operation
    val \ : (operation * operation) -> operation
end

signature CCUAPI = sig
    type state
    eqtype operation
    type siteid = int

    structure T : TIMESTAMP

    (* For communication with the driver. *)
    datatype message = MSG of operation * T.timestamp
                     | QUIT of T.timestamp

    (* For communication with peers. *)
    datatype netmessage = NMSG of operation * siteid * T.timestamp
                        | NQUIT of siteid * T.timestamp
    type ccuobject
    type commtoken

    (* Blocking *)
    val update : (commtoken * message) -> T.timestamp
    val query : commtoken -> (state * T.timestamp)
    exception Done

    (* Non-blocking -- use with caution. *)
    val send : (commtoken * message) -> unit
    val recv : commtoken -> message
    val recvEvt : commtoken -> message CML.event

    structure M : MULTICAST = Netcast
    val create : (state * netmessage M.mchan * netmessage M.port
                        * int * siteid) -> ccuobject
    val start : ccuobject -> commtoken
    val numSites : ccuobject -> int
end
```

B.1.4 CCU Functor

```
(* This is the code for the actual CCU functor, which maps a specification
   for a shared state to a structure containing API functions for
   creating and manipulating CCU objects based on the shared state.
   object.  The structure ccuobj contains all the rules for transforming
   and applying updates to the state. The code for communicating with the
   driver and the other instances of the CCU object resides here. *)

functor CCU(structure ccuobj: CCUOBJ): CCUAPI = struct
    structure A = Array
    structure M = Netcast
    structure D = Debug
    structure T = Timestamp
    exception Error
    type state = ccuobj.state
    type operation = ccuobj.operation
    type siteid = int
```

```
datatype message = MSG of operation * Timestamp.timestamp
                 | QUIT of Timestamp.timestamp

datatype netmessage = NMSG of operation * siteid * Timestamp.timestamp
                 | NQUIT of siteid * Timestamp.timestamp

val operationToString = ccuobj.operationToString
val stateToString = ccuobj.stateToString

fun msgToString (NMSG (u, s, t)) =
    "NMSG(" ^ Int.toString s ^ "," ^ operationToString u ^ "," ^
       T.toString t ^ ")"
|   msgToString (NQUIT (s, t)) =
    "NQUIT(" ^ Int.toString s ^ "," ^ T.toString t ^ ")"

fun msgListToString x =
   let
      fun msgListToString2 nil = ""
      |   msgListToString2 [y] = msgToString y
      |   msgListToString2 (y::ys) = msgToString y ^ "," ^
                                     msgListToString2 ys
   in
      "[" ^ msgListToString2 x ^ "]"
   end

nonfix /
val / = ccuobj./
val \ = ccuobj.\
val apply = ccuobj.apply

(* The caret operator (^):  / or \ depending on how the timestamps are
   ordered. *)
fun op ^ (NMSG(u1, s1, t1), NMSG(u2, s2, t2)) =
   NMSG(if Timestamp.totalLT(t1, t2) then \(u1,u2) else /(u1,u2), s1, t1)
|   op ^ _ = raise Error

(* Our use of ^ obscures the ^ operator in the String structure, so we
   will assign String.^ to the operator &. *)
nonfix ^
val & = String.^
```

```
infix ^
infix &

(* ^^ is the ^ operator applied to lists (sequences) of updates. *)
infix ^^
fun op ^^ (x, nil) = x
|   op ^^ (nil, _) = nil
|   op ^^ ([x], [y]) = [x ^ y]
|   op ^^ (x, y::(ys as _::_)) = (x ^^ [y]) ^^ ys
|   op ^^ (x::(xs as _::_), y) = ([x] ^^ y) ^^ (xs ^^ (y ^^ [x]))

(* Here we define ||, the vertical bar (|) operator from the CCU paper.
   We need several auxiliary definitions as well. *)
local
   exception Bad
   val totalLT = Timestamp.totalLT
   val causalLT = Timestamp.causalLT
   val filter = List.filter

   fun max nil = raise Bad
   |   max [x] = x
   |   max ((x1 as NMSG(_,_,t1))::(x2 as NMSG(_,_,t2))::xs) =
         if totalLT(t1,t2) then max(x2::xs)
         else max(x1::xs)
   |   max _ = raise Error

   fun member (_, nil) = false
   |   member (x, y::ys) = if x = y then true else member(x, ys)

   infix U
   fun op U (x, nil) = x
   |   op U (nil, x) = x
   |   op U (x::xs, y) = if member(x,y) then xs U y
                         else xs U (x::y)
in
   infix ||
   fun op || (W1, W2) =
      if W1 = W2 then nil
      else
         let
```

```
                val u = max[(max W1 handle Bad => max W2),
                            (max W2 handle Bad => max W1)]
                val t = case u of NMSG(_,_,t') => t'
                                | _ => raise Error
                val fil = filter(fn (NMSG(_,_,t1)) => totalLT(t1,t)
                                    | _ => raise Error)
                val fil' = filter(fn (NMSG(_,_,t1)) => causalLT(t1,t)
                                    | _ => raise Error)
             in
                if member(u,W2) then
                   let
                     val W1' = fil W1
                     val W2' = fil W2
                     val W = W1 U W2
                     val W'' = fil' W
                   in
                     (W1' || W2') ^^ ([u] ^^ (W2' || W''))
                   end
                else
                   let
                      val W1' = fil W1
                      val W = W1 U W2
                      val W' = fil W
                      val W'' = fil' W
                   in
                      (W1' || W2) @ ([u] ^^ (W' || W''))
                   end
             end
    end

    structure CommToken = struct
        abstype commtoken = CT of message Mailbox.mbox * message Mailbox.mbox
                     * unit Mailbox.mbox * (state * T.timestamp) Mailbox.mbox
        with
           fun create () = CT(Mailbox.mailbox(), Mailbox.mailbox(),
                              Mailbox.mailbox(), Mailbox.mailbox())
           fun send (CT ct, m) = Mailbox.send(#1 ct, m)
           fun recv (CT ct) = Mailbox.recv(#2 ct)
           fun recvEvt (CT ct) = Mailbox.recvEvt(#2 ct)
           fun externalSend (CT ct, m) = Mailbox.send(#2 ct, m)
```

```
       fun externalRecv (CT ct) = Mailbox.recv(#1 ct)
       fun externalRecvEvt (CT ct) = Mailbox.recvEvt(#1 ct)
       fun query (CT ct) = (Mailbox.send(#3 ct, ()); Mailbox.recv(#4 ct))
       fun recvQryEvt (CT ct) = Mailbox.recvEvt(#3 ct)
       fun sendQryReply (CT ct, m) = Mailbox.send(#4 ct, m)
    end
end


(* The actual CCU object encapsulates the replicated state, a multicast
   channel for outgoing updates, a port on that channel to listen for
   incoming updates, a timestamp representing the site's view of the rest
   of the world, the site's ID, a queue of unapplied updates, and a
   history log. *)

abstype ccuobject = CCUOBJ of {id: {siteid: int,
                                    outgoing: netmessage M.mchan,
                                    incoming: netmessage M.port,
                                    archive: string->unit},
                               state: ccuobj.state,
                               timestamp: Timestamp.timestamp,
                               queue: netmessage Fifo.fifo,
                               log: netmessage list,
                               sitesDone: bool array}
with
    (* p is assumed to be a port on the multicast channel ch.  It is
       created prior to the call to create in order to make sure that
       no instance sends a message before all of the ports have been
       constructed (otherwise messages will be lost). *)
    fun create (x,ch,p,n,id) =
        let
           val entry = "Creating site " & (Int.toString id) &
                       " with initial state " & stateToString x & "\n"
           val archiver = D.updateLog id
        in
           D.createLog id;
           CCUOBJ {
              id = {
                  siteid = id,
                  outgoing = ch,
                  incoming = p,
                  archive = archiver
              },
              state = x,
              timestamp = Timestamp.mktimestamp n,
              queue = Fifo.empty,
              log = nil,
              sitesDone = A.array(n, false)
           } before
           (archiver entry)
       end

    fun print (CCUOBJ{state,...}) = TextIO.print(stateToString state & "\n")

    fun numSites(CCUOBJ{timestamp,...}) = T.size timestamp

    (* Handle an update that originated locally. *)
    (* We assume that updates coming from the local driver come in
       FIFO order. *)
    fun localUpdate (CCUOBJ {id = id as {siteid, outgoing, archive,...},
                         state, timestamp, log, queue, sitesDone}, ct)
                    (m as MSG(u, t)) =
        let
          (* Compute new timestamp *)
          val t' = T.inc(t, siteid)
          val timestamp' = T.sup(timestamp, t')
        in
          archive ("Received local update " & operationToString u &
                      " with timestamp " & T.toString t & ".\n");

          (* Transform the update *)
          let
             val temp = case (NMSG(u, siteid, t') :: log) || log
                        of [x] => x | _ => raise Error
             val (u',t') = case temp of NMSG(u1, _, t1) => (u1, t1)
                                      | _ => raise Error
             val state' = apply(u', state)
          in
             archive ("Transformed to " & operationToString u' & ".\n");
```

```
            (* Broadcast and apply the update *)
            M.multicast(outgoing, NMSG(u', siteid, timestamp));
            CommToken.send(ct, MSG(u, timestamp'));

            archive ("New state is " & stateToString state' & ".\n");
            CCUOBJ {
               id = id,
               state = state',
               timestamp = timestamp',
               queue = queue,
               log = NMSG(u, siteid, t') :: log,
               sitesDone = sitesDone
            }
         end
      end
|     localUpdate (ccu as CCUOBJ{id = {siteid, outgoing, archive, ...},
                         timestamp, sitesDone, ...}, _)
                (m as QUIT ts) = (
         archive ("Received local QUIT message with timestamp " &
                        T.toString ts & ".\n");

         A.update(sitesDone, siteid-1, true);
         M.multicast(outgoing, NQUIT(siteid, timestamp));
         ccu
      )

(* Handle an update that originated at another site. *)
fun remoteUpdate (ccu as CCUOBJ{id = id as {siteid,
                                            outgoing,
                                            archive, ...},
                           state, timestamp, queue, log, sitesDone})
                 (m as NMSG(u, s, t)) = (
    archive ("Site " & Int.toString siteid & " received message "
         & operationToString u & " from site " & Int.toString s &
         " with timestamp " & T.toString t & ".\n");

    (* Because all instances of the CCU object are sharing a single
       multicast channel, an instance will receive any message it
       sends.  We want to ignore all of these messages. *)
    (* Also ignore any messages coming from sites that have sent a
       quit message (there shouldn't be any of these). *)
    if s = siteid orelse A.sub(sitesDone,s-1) then (
       archive "Ignored.\n";
       ccu
    )
    else
       if T.causalLT(t,timestamp) orelse timestamp = t then
          (* Transform and apply the update *)
          (* Note that we are making use of Theorem 2 here *)
          let
             val t' = T.inc(t,s)
             val _ = archive(msgListToString(NMSG(u,s,t') :: log) & "|")
             val _ = archive(msgListToString(log) & "\n")
             val temp = case (NMSG(u, s, t') :: log) || log
                        of [x] => x | _ => raise Error
             val u' = case temp of NMSG(u1, _, _) => u1
                                 | _ => raise Error
             val state' = apply(u', state)
             val t'' = T.sup(t', timestamp)
          in
             archive ("Transformed to " & operationToString u' & ".\n");
             archive ("New timestamp is " & T.toString t'' & ".\n");
             archive ("New state is " & stateToString state' & ".\n");

             CCUOBJ {
                id = id,
                state = state',
                timestamp = t'',
                queue = queue,
                log = NMSG (u, s, t') :: log,
                sitesDone = sitesDone
             }
          end
       else (
          archive "Enqueued.\n";
          (* If the update cannot be applied, enqueue it *)
          CCUOBJ {
             id = id,
             state = state,
             timestamp = timestamp,
            queue = Fifo.enqueue(queue, m),
            log = log,
            sitesDone = sitesDone
         }
      )
)
|   remoteUpdate (ccu as CCUOBJ{id = id as {siteid, archive, ...},
                           state, timestamp, queue, log, sitesDone})
                (m as NQUIT(s, t)) = (
    archive ("Site " & Int.toString siteid &
        " received NQUIT message from site " & Int.toString s
        & " with timestamp " & T.toString t & ".\n");

    if s=siteid then ccu
    else
       if T.causalLT(t,timestamp) orelse timestamp = t then (
          A.update(sitesDone,s-1,true);
          ccu
       )
       else (
          (* If there are outstanding messages, enqueue the quit
             request *)
          archive "Enqueued.\n";
          CCUOBJ {
             id = id,
             state = state,
             timestamp = timestamp,
             queue = Fifo.enqueue(queue, m),
             log = log,
             sitesDone = sitesDone
          }
       )
)

(* Try again to apply the first update in the queue (which hasn't
   yet been applied because the prerequisites haven't been met). *)
(* Assumes:  m is the first element of !queue *)
fun checkQueue (CCUOBJ{id, state, timestamp, queue, log, sitesDone})
               m = (
    (#archive id) "Reexamining queue.\n";
```

```
            remoteUpdate (CCUOBJ {
               id = id,
               state = state,
               timestamp = timestamp,
               queue = #1(Fifo.dequeue(queue)),
               log = log,
               sitesDone = sitesDone
          }) m
        )

        (* This function is to be called by the driver.  It starts a
           server for the current instance of the CCU object, and returns
           a mailbox for sending it commands. *)
        fun start (y as CCUOBJ (x as {id, ...})) =
            let
               (* For communication with the driver.  Note that the order of
                  mbOut and mbIn is the reverse of what it is in driver.sml. *)
               val commToken = CommToken.create()

               (* This function shamelessly lifted out of Ullman's book,
                  p. 240.  :-)  *)
               fun checkAll(A,i) =
                   i < 0 orelse A.sub(A,i) andalso checkAll(A, i-1)

               fun loop (ccu as CCUOBJ (x as {id, ...})) =
                  let
                     (* Do not accept local messages after the driver
                        has issued a NQUIT request. *)
                     val e1 = if A.sub(#sitesDone x, #siteid id - 1) then
                                 CML.never
                              else
                                 CommToken.recvEvt commToken
                     val e2 = M.recvEvt (#incoming id)
                     val e3 = CommToken.recvQryEvt commToken
                  in
                     if checkAll(#sitesDone x, numSites ccu - 1) then (

                        (#archive id) ("Site " & Int.toString (#siteid id)
                            & " terminating.\nFinal state is " &
                            stateToString (#state x) & ".\n");
```

```
                    D.closeLog (#siteid id);
                    CML.exit()
                 )
                 else
                    loop (
                       CML.select [
                          CML.wrap(e1, localUpdate (ccu, commToken)),
                          CML.wrap(e2, remoteUpdate ccu),
                          CML.wrap(e3, (fn () => (CommToken.sendQryReply (
                              commToken, (#state x, #timestamp x)); ccu))),
                          CML.wrap(if Fifo.isEmpty(#queue x) then
                                      CML.never
                                   else
                                      CML.alwaysEvt(Fifo.head(#queue x)),
                                   checkQueue ccu)
                       ]
                    )
              end
        in
           D.print("Starting site " & (Int.toString (#siteid id)) & ".\n");
           CML.spawnc loop y;
           commToken
        end
end

(* For use by client modules. *)
type commtoken = CommToken.commtoken
val send = CommToken.externalSend
val recv = CommToken.externalRecv
val recvEvt = CommToken.externalRecvEvt

exception Done

fun update (ct, m) = (
    send (ct, m);
    let
       val temp = recv ct
       val ts = case temp of
                   MSG (_, ts') => ts'
```

```
                    |  _ => raise Done
        in
           ts
        end
    )

    val query = CommToken.query
end
```

B.1.5 Driver Signature

```
(* Signature for drivers.  Contains definitions the application designer
   must supply in order to use the initialization functor.  If the designer
   prefers to hand-code the initialization, then the drivers need not
   conform to this signature.  *)

signature DRIVER = sig
   type driver
   type commtoken
   type init
   val initialize : unit -> unit
   val initData : unit -> init
   val mkdriver : (unit -> commtoken) -> Timestamp.timestamp ->
                  init -> driver
   val main : driver -> unit
end
```

B.1.6 Initialization Functor

```
(* A functor to generate initialization code for the CCU objects *)
functor InitFn( structure Obj : CCUAPI;
                structure D : DRIVER;
                sharing type Obj.commtoken = D.commtoken
              ) : sig
    val init: (int * Obj.state) -> unit
end
= struct
    structure M = Netcast
    type state = Obj.state

    (* n is the number of instances to create *)
    (* initState is the shared initial state of the objects *)
    fun init (n,initState) =
       let
          (* Create a multicast channel and a list of n ports on it. *)
          val mc = M.mChannel()
          fun mkPorts 0 = nil
          |   mkPorts n = (M.port mc) :: mkPorts(n-1)

          (* Now create a CCU object for each port. *)
          fun mkObjs nil = nil
          |   mkObjs (p::ps) = Obj.create(initState, mc, p, n,
                                   n - length ps) :: mkObjs ps

          val objs = mkObjs (mkPorts n)

          (* Now make a driver for each instance of the CCU object. *)
          fun mkDrivers nil = nil
          |   mkDrivers (ob::obs) =
                 D.mkdriver (fn () => Obj.start ob)
                            (Timestamp.mktimestamp n)
                            (D.initData ()) :: mkDrivers obs

          val _ = D.initialize ()
          val drivers = mkDrivers objs
       in
          app (fn d => ignore (CML.spawn (fn () => D.main d))) drivers;
          CML.sync CML.never
       end
end
```

B.1.7 Debugging Module

```
(* Print debugging messages; log actions to log files. *)

structure Debug: sig
   val print: string->unit
   val set: unit->unit
   val clear: unit->unit
   val createLog: int->unit
   val updateLog: int->string->unit
   val closeLog: int->unit
end
= struct
   val debug = ref true
   val logs = ref (nil: (int * TextIO.outstream) list)

   fun set () = debug := true
   fun clear () = debug := false

   fun print s = if !debug then TextIO.print s else ()

   fun hasLog n =
      let
         fun hasLog2 nil = false
         |   hasLog2 ((m,_)::xs) = if m = n then true else hasLog2 xs
      in
         hasLog2 (!logs)
      end

   exception notFound
   fun getStream n =
      let
         fun getStream2 nil = raise notFound
         |   getStream2 ((m,s)::xs) = if m = n then s else getStream2 xs
      in
         getStream2 (!logs)
      end

   fun removeLog n =
      let
         fun removeLog2 nil = nil
         |   removeLog2 ((x as (m, _))::xs) =
             if m = n then xs
             else x :: removeLog2 xs
      in
         logs := removeLog2 (!logs)
      end
   fun createLog n =
      if hasLog n then ()
      else
         let
            val outStream = TextIO.openOut("site" ^ Int.toString n ^ ".log")
         in
            logs := (n, outStream) :: !logs
         end

   fun updateLog n s =
       (TextIO.output(getStream n, s); print s) handle notFound => ()

   fun closeLog n = (
       TextIO.closeOut(getStream n);
       removeLog n
   ) handle notFound => ()
end
```

B.2 Shared Text Buffer

B.2.1 Shared Object Specification

```
structure CCUTextBuf = struct
    type state = char list
    val stateToString = implode

    datatype operation = Insert of int*string | Delete of int*int

    fun operationToString (Insert(n,s)) = "Insert(" ^ Int.toString n ^ ",\"" ^
                                          s ^ "\")"
    |   operationToString (Delete(m,n)) = "Delete(" ^ Int.toString m ^ "," ^
                                          Int.toString n ^ ")"

    exception BadUpdate

    fun apply(Insert(1,b), x) = (explode b) @ x
    |   apply(Insert(a,b), x::xs) = if a <= 0 then raise BadUpdate
                                    else x :: apply(Insert(a-1,b), xs)
```

```
|   apply(Delete(1,0), x) = x
|   apply(Delete(1,b), _::xs) = if b < 0 then raise BadUpdate
                                else apply(Delete(1, b-1), xs)
|   apply(Delete(a,b), x::xs) = if a <= 0 then raise BadUpdate
                                else x :: apply(Delete(a-1,b), xs)
|   apply _ = raise BadUpdate

nonfix /
fun / (Insert(a,b), Insert(c,d)) =
    if a < c then
        Insert(a, b)
    else
        Insert(a + size d, b)
|   / (Delete(a,b), Delete(c,d)) =
    if a + b <= c then
        Delete(a, b)
    else if a <= c andalso c <= a + b andalso a + b <= c + d then
        Delete(a, c - a)
    else if a <= c andalso c <= c + d andalso c + d <= a + b then
        Delete(a, d - b)
    else if c <= a andalso a <= a + b andalso a + b <= c + d then
        Delete(c, 0)
    else if c <= a andalso a <= c + d andalso c + d <= a + b then
        Delete(c, a + b - c - d)
    else
        Delete(a - d, b)
|   / (Delete(a,b), Insert(c,d)) =
    if a + b <= c then
        Delete(a, b)
    else if a <= c andalso c < a + b then
        Delete(a, b + size d)
    else
        Delete(a + size d, b)
|   / (Insert(a,b), Delete(c,d)) =
    if a < c then
        Insert(a, b)
    else if c <= a andalso a < c + d then
        Insert(c, "")
    else
        Insert(a - d, b)
    fun \ (i as Insert(a, b), Insert(c, d)) =
        if a <= c then i else Insert(a + size d, b)
    |   \ x = / x
end
```

B.2.2 Script-file Interface

```
structure GetCommands: sig
    type commandstream
    exception Command and EOF
    type operation = TextBuf.operation
    datatype command = Operation of operation | Delay of int | Quit
    val mkCommandStream: string -> commandstream
    val getCommand: commandstream -> command
    val closeCommandStream: commandstream -> unit
end
= struct
    structure TB = CCUTextBuf

    type commandstream = TextIO.instream
    exception Command and EOF
    type operation = TextBuf.operation
    datatype command = Operation of operation | Delay of int | Quit

    val mkCommandStream = TextIO.openIn

    (* I find that dealing with options in code is a bit awkward.
       I prefer this version of Int.fromString that raises an exception
       when the format of the string is bad.  *)

    fun removeOption e f x = case f x of NONE => raise e | SOME y => y
    exception BadInt
    val stringToInt = removeOption BadInt Int.fromString

    (* Convert a list of strings to a command. *)
    fun mkupdate [a,b,c] =
        (if a = "i" then Operation(TB.Insert(stringToInt b, c))
         else if a = "d" then Operation(TB.Delete(stringToInt b, stringToInt c))
              else raise Command
         handle _ => raise Command)
    |   mkupdate [a,b] =
        (if a = "w" then Delay(stringToInt b)
         else raise Command
         handle _ => raise Command)
    |   mkupdate [a] =
        if a = "q" then Quit else raise Command
    |   mkupdate _ = raise Command

    fun getCommand cs =
        let
           val s = let
                      val x = TextIO.inputLine cs
                   in
                      if x = "" then raise EOF else x
                   end
           val c = String.tokens(
               fn x => not (Char.isAlpha x orelse Char.isDigit x)
           ) s
        in
           mkupdate c
        end

    val closeCommandStream = TextIO.closeIn
end
```

B.2.3 Driver

```
(* The "driver" that contains the code that actually sends instructions to the
   CCU object.  Supports a constructor mkdriver and a function main that sets
   everything in motion.  This driver is specific to the text buffer. *)

structure Driver: DRIVER = struct
   structure TS = Timestamp
   structure C = GetCommands
   structure TB = TextBuf

   type operation = TB.operation
   datatype message = datatype TB.message
   type commtoken = TB.commtoken
```

```
exception Done of Timestamp.timestamp

type init = string

(* counter to properly assign input files *)
val x = ref 1

fun initialize () = x := 1

fun initData () = ("dr" ^ Int.toString(!x) ^ ".cmd") before x := !x + 1

abstype driver = Driver of (unit->commtoken) * TS.timestamp * string
with
    fun mkdriver f t s  = Driver (f, t, s)
    fun main (Driver (f, t, s)) =
       let
          val comStream = C.mkCommandStream s
          val commToken = f ()

          fun mainLoop ts =
             let
                val c = C.getCommand comStream
             in
                TextIO.print (s ^ "\n");
                case c of
                   C.Delay x => (
                     CML.sync (
                        CML.timeOutEvt (
                           Time.fromMicroseconds (Int32.fromInt x)
                        )
                     );
                     mainLoop ts
                   )
                |  C.Operation x => mainLoop (
                        TB.update(commToken, MSG(x, ts))
                        handle _ => raise Done ts
                   )
                |  C.Quit =>
                     raise Done ts
             end
```

```
                handle C.EOF => raise Done ts
          in
             mainLoop t
             handle Done ts  => (TB.send(commToken, QUIT ts);
                                 C.closeCommandStream comStream)
          end
   end
end
```

B.2.4 Shared Object

```
structure TextBuf = CCU(structure ccuobj = CCUTextBuf)
```

B.2.5 Initializer

```
(* Initialization code for text buffer. *)
structure Init = InitFn(structure Obj = TextBuf; structure D = Driver)
```

B.2.6 Mainline

```
(* Code that contains the call to RunCML.doit to set the scheduling
   quantum and set everything in motion. *)

structure Mainline: sig
   val main: (int * string) -> unit
end
= struct
    (* n is the number of sites to create *)
    (* initState is the shared initial state *)
    fun main (n,initState) = ignore (
        RunCML.doit(fn () => Init.init(n,explode initState), NONE)
    )
end

fun checkFinalStates () = ignore (
    RunCML.doit(fn () => ignore (OS.Process.system "grep Final *.log"), NONE)
)
```

Bibliography

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[2] Association for Computing Machinery. *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work*, Seattle, November 1998.

[3] Gordon V. Cormack. A calculus for concurrent update. *Research Report CS-95-06, Dept. of Computer Science, University of Waterloo*, 1995.

[4] Gordon V. Cormack. A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. *Research Report CS-95-08, Dept. of Computer Science, University of Waterloo*, 1995.

[5] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, Harlow, England, third edition, 2001.

[6] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[7] Clarence A. Ellis. A model and algorithm for concurrent access with groupware. *Technical Report CU-CS-593-92, Department of Computer Science, University of Colorado at Boulder*, 1992.

[8] Charles F. Goldfarb and Paul Prescod. *The XML Handbook.* Prentice Hall, Upper Saddle River, New Jersey, 1998.

[9] Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 207–217. Association for Computing Machinery, November 1994.

[10] Irene Greif, Robert Seliger, and William Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the 13th Annual Symposium on Principles*

of Programming Languages, pages 160–172. Association for Computing Machinery, January 1986.

[11] Christopher Hendrie. Objects which break "A calculus for concurrent update". *Research Report CS499, Dept. of Computer Science, University of Waterloo*, 1998.

[12] Michael J. Knister and Atul Prakash. Distedit: A distributed tookit for supporting multiple group editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 343–355. Association for Computing Machinery, October 1990.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge Massachusetts, 1997.

[15] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the ACM Symposium on User Interface Software and Technologies*, pages 111–120, November 1995.

[16] Christopher R. Palmer and Gordon V. Cormack. Operation transforms for a distributed shared spreadsheet. In *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work* [2], pages 69–78.

[17] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, New York, 1999.

[18] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 288–297, November 1996.

[19] C. Sun, X. Jia, Y. Zhang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-human Interaction*, 5(1):63–108, March 1998.

[20] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 582–591, January 1996.

[21] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work* [2], pages 59–68.

[22] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Upper Saddle River, New Jersey, ML97 edition, 1998.