Lock-Free Distributed Objects: A Shared Spreadsheet *

Christopher R. Palmer and Gordon V. Cormack

Department of Computer Science
University of Waterloo
Waterloo, Ontario
{crpalmer,gvcormack}@plg.uwaterloo.ca

Abstract

A *Lock-Free Distributed System* is a purely-replicated environment (replication with no central authority) in which updates are performed with no inter-object communication. All replicated copies of an object take on a common state when there are no messages in transit. Such a system is often used for groupware applications and operation transformations are a popular technique in this field. We broaden the scope of operation transformations by using a lock-free concurrency system to specify a distributed spreadsheet. The resulting spreadsheet is highly fault tolerant and the interactive performance is independent of the speed and the quality of the underlying network. To specify the concurrent semantics of the spreadsheet, an abstract data type is defined and a general model of a subset of the operations is proposed.

1 Introduction

In the most general sense, an object is a store that has well-defined access methods. That is, an object may assume any of a (potentially infinite) set of values X and only some predefined set of operations, O, may cause the state of the object to change.

A *lock-free distributed object* is a *purely-replicated* object that assumes a common state at every site whenever there are no messages in transit. We also require that any updates to the object be performed with no network communication. All updates must be performed locally and only notification of the update is distributed to the other sites. A *purely-replicated* system is a distributed system in which there is no central authority and each site maintains a complete copy of the distributed object. We have named these systems "lock-free" to emphasize the difference between them and those that use some form of network synchronization such as semaphores or "locks." There are numerous advantages to lock-free systems over their traditional counterparts. The speed of the network does not affect the interactive performance of the system, and since local updates to the object are independent of the network, the network may even fail without impeding the user's ability to operate with the object. Similarly, any number of the distributed nodes may fail because each copy of the object may be updated independently.

There are many other techniques that may be used to design a distributed spreadsheet. One possible solution is to use a *client/server* approach. There is a single copy of the object at a server site and clients use the network to access and update the object. Client/server solutions have a single point of failure and usually require network transmission for every update. It is possible to allow sequences of updates to be grouped together into a transaction. A transaction may only be processed if it does not conflict with a higher priority transaction. Many spreadsheet operations update the state of the entire object and consequently, transactions are infeasible. On the other hand, there are algorithms for providing consistency in a purely-replicated system. It is possible to use a distributed semaphore or a central locking authority. But, a central authority provides the same problems as the client/server model and there is no known algorithm for efficiently handling distributed semaphores [10]. It is also possible to use a causal ordering [7] or even a totally ordered communication system. That is, each update is augmented

*University of Waterloo Technical Report CS-98-04

with a clock and all sites perform the updates in an order that agrees with the causal ordering and/or the total order. This approach is difficult to implement, reduces the possible parallelism inherent in a replicated architecture and causes undo delay in message processing [1].

There has been recent research in operation transformations. If an object has a set of possible update functions, O, then an operation transform is a mapping $O \times O \rightarrow O$. Operation transformations were pioneered by Ellis and Gibbs who proposed their dOPT algorithm [4]. It has been pointed out that this algorithm is in fact flawed (for example, see [8] or [9]). A corrected version of dOPT that handles the special case of two communicating processes (point-to-point communication) has also been devised [3]. It does not seem possible to directly repair dOPT handle the more general case of n communicating processes. Instead, there have been algorithms proposed which are motivated by dOPT.

Cormack has proposed A Calculus for Concurrent Update (CCU) [2]. This algorithm appears to be the most general. We will make use of this algorithm to design a lock-free distributed shared spreadsheet. Because we are using a consistency model that allows for lost messages, the distributed system is highly fault-tolerant. As we claimed before, local operations are not affected by a network failure. When the network failure occurs, updates will not be delivered between affected nodes. When the fault is corrected and the network is reconnected, there will be a period of time in which sites see a large number of concurrent operations. They will have no other ill-effects due to the network outage. In the process of designing such an object, we have developed a preliminary classification of different operation transformations which should assist others attempting similar work.

The remainder of this paper is organized as follows. In section 3, a formal model of a spreadsheet is proposed. This model is augmented in section 4 to define a concurrent model for the spreadsheet. Finally, we present our transformation classes in section 5. However, before delving into the spreadsheet, we present a brief overview of the concurrency control algorithm that we employ.

2 A Calculus for Concurrent Update

Central to CCU is the ability to define transformations on the update operations that define "equivalent" updates and consequently have the same behaviour as some canonical model of the concurrent operations. We assume that there are n sites, each of which has a unique site identifier, s_i, and each site maintains a fully-replicated copy of the distributed object that takes on states X and has update functions O. We assume that there exists a communications medium that never corrupts messages but may result in lost messages or out of order delivery of messages.

To define the meaning of a set of concurrent updates, two fully-defined transformations (these transformations are functions $O \times O \rightarrow O$) must be provided. The two transformations are: f/g, read f *after* g, which defines a function that is equivalent to f after g has been applied concurrently; $f \backslash g$, read f *before* g, which is the dual of f/g, and is defined by $f/g(g(x)) = g \backslash f(f(x))$ for all states $x \in X$. It is with these two functions that we will build non-canonical update sequences.

We use a vector based clock to define Lamport's *happens before* relation [7] and a total order that is consistent with this causal relationship. If the i^{th} component of a time vector represents the number of operations, generated by site s_i, that have been processed, we can define the *happens before* relation as $T1 \subset T2$ if $T1 \subseteq T2$ and $T1 \neq T2$ where:

$$T1 = (a_1, \cdots, a_n) \subseteq (b_1, \cdots, b_n) = T2 \text{ if } a_i \leq b_i \text{ for all } 1 \leq i \leq n$$

A total order which is consistent with the causal order is implemented by:

$$(a_1, \cdots, a_n) < (b_1, \cdots, b_n) \text{ if } a_1 < b_1 \text{ or } a_1 = b_1 \text{ and } (a_2, \cdots, a_n) < (b_2, \cdots, b_n)$$

Given an update sequence $g_1, g_2, \cdots, g_m$ (where $i < j$ implies that g_i occurs before g_j in the total order) and an initial state x, we define the effect of the update sequence on the object to be the state $g_m(\cdots(g_2(g_1(x)))\cdots)$. If there are m concurrent updates, $f_1, f_2, \cdots, f_m$, then the canonical model of these updates is defined to be the sequence $f_1, f_2/f_1, \cdots, f_m/(f_{m-1}/(\cdots/f_1\cdots))$. It is important to remember this canonical sequence because we will define transformations f/g to give it meaning.

The task of the CCU algorithm is to process any of the $m!$ different update sequences that could be generated from m concurrent events. The order that the updates are received and processed should have no effect on the final state of the object. That is, a combination of / and \ transformations will

```
e      --> @function(e1, ..., en)
e      --> cell
e      --> range
e      --> literal
cell   --> [rowref, colref]
range  --> {cell, cell}
rowref --> row | :row
colref --> col | :col
```

Figure 1: Spreadsheet Expressions

be applied to the updates at each site so that the constructed update sequence is equivalent to the canonical sequence, above. This algorithm is presented in [2]. Because the final state is modeled by the canonical update sequence, there is a simple technique for making a sequence of updates atomic. If a site temporarily stops processing remote events, the set of updates that it generates will be consecutive in the total order.

CCU differs from the other known algorithms because it does not require that the elementary operations on an object have an inverse. While it is possible to define inverse functions by saving the state of the store, this is not always reasonable (for example, updating the *Oxford English Dictionary* makes the cost of inverse functions prohibitive). It is for this reason that we feel CCU is the most general known algorithm that addresses the lock-free distributed consistency problem.

3 A Spreadsheet Model

Spreadsheets are simple objects which most people have used at some point in their lives. A spreadsheet is a store that organizes information into rows and columns. The entries in this two dimensional object are *cells*. A cell holds text, numbers and even complex formulas. To ease formula writing, it is possible to define symbolic names to represent areas of the spreadsheet. These areas are referred to as *ranges* of cells and serve as basic units when writing formulas. A rich library of functions gives the user the ability to specify complex relationships among cells and form the heart of the spreadsheet. To provide aesthetically pleasing output, the appearance of entire columns may be defined while still allowing individual cells to specify an alternative display format for their contents. In addition to simply storing literal values and formulas, there are operations which make editing the spreadsheet easier. It is always possible to insert and delete both rows and columns while ensuring that all formulas in the spreadsheet continue to reference the same cells after the update operation. A copy operation is provided that provides two different cell referencing modes. A reference to a cell may either be *fixed* in that copying the formula will keep the same reference or *relative* in that copying the formula will update the reference to the same relative position vis-a-vis the destination cell.

All of these operations are provided in a public domain spreadsheet called `sc`. We define a fairly general spreadsheet but make use of the semantics of `sc` when there are multiple possible interpretations (such as references to cells that have been deleted). The typical convention for references to a cell is to use one or more letters (defined by the sequence `A`, `B`, $\cdots$, `Z`, `AA`, $\cdots$) to represent the column and a non-negative number to represent the row. While this is convenient for users, it is not amenable to mathematical notation. Instead, we use non-negative numbers for both the row and column references. It is now possible to define a spreadsheet as an abstract data type.

A spreadsheet may be modeled using unconstrained arrays. A spreadsheet is a triple $S =< C, F, M >$ where C is a two dimensional array of cells, F is a one dimensional array of per-column formatting information and M is a mapping between names and ranges of cells. The cell at position row and col is a triple $C[row, col] =< l, e, f >$ where l is a (potentially empty) string label, e is an expression defined by a simple grammar (figure 1) and f is a set of per-cell flags. The expression language used in this paper does not necessarily correspond to any existing spreadsheet language. Instead, the expression language tends to correspond to the parsed operator tree that would be derived from any actual spreadsheet formula language. Cell references of the form $[: row, : col]$ indicate that the row and/or column number is fixed. Cell references of the form $[row, col]$ indicate that the row and/or column number is "relative" to the current cell. To be completely general, we allow fixed references to be independently attached to the row

and the column numbers.

While the formatting information is particularly useful for users, differences in formatting do not impact on the definition of a spreadsheet. The only exception to this rule is the ability to lock cells. We assume that the per-cell formatting information includes the attribute $islocked(C[row, col])$ to record these locks.

We now define a the basic set of operations which we outlined, above. This is not necessarily an exhaustive list as there are differences among spreadsheets. This is a complete list of the functionality of `sc`. We define a spreadsheet to have the following operations:

$set(cell, value)$ - Replace the contents of the cell at location *cell* with the triple: $value =< l, e, f >$.

$format(column, format)$ - Replace the formatting information for the *column* with the new formatting information $format$.

$insert(row, nrows, col, ncols)$ - Insert rows and/or columns by adding $nrows$ before row and adding $ncols$ before col. Update the references so that they continue to point to the same cells.

$delete(row, nrows, col, ncols)$ - Delete $nrows$ beginning with row and $ncols$ beginning with col. Update the references so that they continue to point to the same cells.

$copy(src, dest, value)$ - Copy the cell at position src, which is $value = C[src]$, to $dest$ updating the relative references in v at the destination. We add the $value$ parameter to this operation as our only concession required to define reasonable concurrent semantics.

$define(name, range)$ - Create a mapping from a name to a specific range of the spreadsheet.

$undefine(name, range)$ - Remove a mapping from a name.

We formally define the behaviour of the following subset of the operations: $format$, set, $delete$ and $copy$. We ignore the name definition facilities because they are trivial in `sc` and even more complicated name definition systems are still easy to model. We discuss the $format$ operation because of the simplicity of the concurrent semantics. These semantics will provide a gentle introduction to the more complicated definitions. The two operations, set and $copy$ are fundamental and may not be ignored. Finally, we have chosen to define $delete$ and not $insert$ because there is an obvious parallel between them, and $delete$ has slightly more interesting concurrent semantics. The definitions appear in figure 2.

4 Concurrent Semantics

To produce a distributed lock-free spreadsheet, we need to augment the spreadsheet definitions with concurrent semantics. Since we are using the CCU algorithm, we must define for all pairs of functions, f and g, the two transformations f/g and $f\backslash g$. Technically, we should prove that, for each pair of transformations and any spreadsheet S, $f/g(g(S)) = g\backslash f(f(S))$. We will generally ignore this requirement; the reader can easily verify this condition for the majority of the transformations that we define. Instead, we concentrate our effort on defining reasonable semantics for f/g. That is, we want $f/g(g(S))$ to reasonably model the two concurrent updates f and g.

An identity function is used to assist the definition of the concurrent semantics, but this operation is not strictly necessary. It could be replaced with $delete(0, 0, 0, 0)$, which will leave the spreadsheet unchanged. We have adopted the $ident$ function because it tends to make the definitions more comprehensible.

Defining the formatting information for a column provides a gentle introduction to the concurrent semantics. It is obvious that changing the format of a column is essentially independent of the other operations. We must answer the question, "what does it mean, to some other concurrent operation, to change the formatting information for a column?" For set and $copy$, the answer is "nothing". It not does matter whether or not you change the format of the column while you are also copying or modifying a cell because these updates are truly independent. Thus, we can quickly and conveniently define these transformation:

```
format / f = format \ f = format     (f != delete and f != format)
f / format = f \ format = f          (f != delete and f != format)
```

However, $delete$ and $format$ operations are not entirely independent. If we assumed these operations to be independent and blindly applied the original format operation, we may end up affecting a different,

```
A spreadsheet is a pair S=<C, F> where C is 2D array of cells, F is
a 1D array of column formats and a cell is the triple <l,e,f>.

set([r, c], value) is
   C[r,c] = value           (not islocked(C[r,c])
   error                    (islocked(C[r,c])

format(c, f) is
   F[c] = f

copy([fr,fc], [tr,tc], <l,e,f>)
  error                     (islocked(C[tr,tc])
  C[tr,tc] = <l,fix(e),f>   (not islocked(C[tr,tc]))
where fix(e) is defined as
 fix(literal) = literal
 fix(@f(e1, ..., en)) = @f(fix(e1), ..., fix(en))
 fix({cell1, cell2}) = {fix(cell1), fix(cell2)}
 fix([r,c]) = [r',c']
 where
   r' = r                   (isfixed(r))
   r' = r+(tr-fr)           (not isfixed(r) and tr-fr+r >= 0)
   r' = 0                   (not isfixed(r) and tr-fr+r < 0)
   c' = c                   (isfixed(c))
   c' = c+(tc-fc)           (not isfixed(c) and tc-fc+c >= 0)
   c' = 0                   (not isfixed(c) and tc-fc+c < 0)

delete(row, nrow, col, ncol) is
   error                    (islocked(C[r,c]) for some r in row .. row+nrow-1 or
                                              for some c in col .. col+ncol-1)
Otherwise:
   C[row+i,c] = C[row+nrow+i,c] for all c, i >= 0
   C[r,col+i] = C[r,col+ncol+i] for all r, i >= 0
   F[col+i]   = F[col+ncol+i]   for all i >= 0
   C[r,c].e   = fix(C[r,c].e)   for all r, c
   Prune the last nrow rows and ncol columns of C and ncol columns of F.

where we define fix as
   fix(literal) = literal
   fix(@f(e1, ..., en)) = @f(fix(e1), ..., fix(en))
   fix({cell1, cell2}) = {fix(cell1), fix(cell2)}
   fix([r, c]) = [r', c']
   where
      r' = r                (r < row)
      r' = r-nrow           (r >= row+nrow)
      c' = c                (c < col)
      c' = c-ncol           (c >= col+ncol)
```

Figure 2: Spreadsheet Definitions

and wrong, column. We need to correct the *format* operation to ensure that it is applied to the same column after the *delete* operation has been applied. Therefore, we derive the following definition:

```
format(c, f) / delete(fr,nrows,fc,ncols) = format(c, f) \ delete(fr,nrows,fc,ncol) =
    ident             (fc <= c < fc + ncols)
    format(c, f)      (c < fc)
    format(c-ncol, f) (c >= fc + ncols)
```

If a column formatting operation occurs concurrently with a delete (which may or may not affect the columns of the spreadsheet), there are only a few cases that need to be considered. If the column being modified is concurrently deleted, the format operation can no longer be reasonably applied and it is mapped to the identity function. On the other hand, if the delete is "to the left" of the formatting request, the column number will need to be shifted to the left by the number of deleted columns. Finally, if the delete is "to the right" of the formatting request, it doesn't change the area of the spreadsheet on which the *format* operation was based.

We have defined all of the transformations on the *format* operation except for the transformations over a second, concurrent, *format* request. What does it mean for two *format* operations to happen concurrently? If they are changing different columns of the spreadsheet, there is no interaction between the two operations. On the other hand, if two users attempt to change the format of the same column, we need to devise some means of handling this conflict. We make use of the total order and the fact that CCU guarantees that all sites will have an update sequence that models this total order. So, we simply define the result of the concurrent format operations to be the one which occurs later in the total order:

```
format(c1, f1) / format(c2, f2) = format(c1, f1)
format(c1, f1) \ format(c2, f2) =
    format(c1, f1)          (c1 != c2)
    ident                   (c1 = c2)
```

Before we define more transformations, there are some general rules that we can extract from the process of definition these transformations. There are some pairs of operations which are truly independent. No transformation is required when these operations occur concurrently. There are other pairs of operations where it is the last operation at a location which "wins" and all others get masked. The last type of transformation was the deletion of rows and columns. The deletion request changes the relative position of the format operation. We can consider type of transformation as updating the context in which the operation is being applied. When the concurrent deletion has occurred, it has changed the context of the spreadsheet in a such a way that the format request has had its meaning changed. So, when events are not independent and they do not mask each other, we need to look at how the operations interact over the spreadsheet.

The *set* operation includes similar transformations. We already know that changing the value of a cell and changing the per-column formatting information are independent. What does it mean for a *delete* and a *set* update to occur concurrently? As with the transformation of the *format* operation over the *delete* operation, the position of the *set* must be updated to correct for the change in relative contexts. This is not the only change that needs to be reflected. The formula being stored into a cell is also based on this incorrect context. We must recursively apply the context transformation to references in the formula. We have the definition:

```
set([r,c], v) / delete(fr, nrow, fc, ncol) = set([r,c], v) \ delete(fr, nrow, fc, ncol) =
   ident                    (fr <= r < fr+nrow or fc <= c < fc+ncol)
   set(fix([r,c]), f(v))    (otherwise)
where fix(<l, e, f>) = <l, fix(e), f>
      fix(literal) = literal
      fix(@f(e1, ... en) = @f(fix(e1), ..., fix(en))
      fix({cell1, cell2}) = { fix(cell1), fix(cell2) }
      fix([r,c]) = [drow(r), dcol(c)]
      where drow(r) = r      (r < fr+nrow)
                    = r-nrow (r >= fr+nrow)
            dcol(c) = c      (c < fc+ncol)
                    = c-ncol (c >= fc+ncol)
```

The transformation for two concurrent *set* operations is analogous to two formatting operation. The last operation updating a cell will be the "winner". That is:

```
set([r1,c1],v1) / set([r2,c2], v2) = set([r1,c1], v1)
set([r1,c1],v1) \ set([r2,c2], v2) =
   ident                    ([r1,c1] = [r2,c2])
   set([r1,c1], v1)         ([r1,c1] != [r2,c2])
```

This leaves only the concurrent application of a *set* and a *copy* operation. At the destination of the *copy* operation, the event that is later will mask the earlier event. Once again, we have a definition modeled on the transformations for two *format* operations. There is, however, a problem. What does it mean to change the source of the copy concurrent to the *copy* itself? One possible interpretation is to adhere strictly to the total order. If the *set* operation occurs before the *copy* operation, the *copy* operation should actually copy the new value and not the original value. This interpretation makes the most mathematical sense but doesn't seem to model the user's intention. The strict definition assumes that the user wanted to copy whatever was contained in the cell when in reality they probably meant to copy the specific contents of the cell. Thus, we define the semantics in a manner analogous to the pair of *set* operations:

```
set([r,c], v1) / copy([sr,sc], [dr,dc], v2) = set([r,c], v1)
set([r,c], v1) \ copy([sr, sc], [dr, dc], v2) =
   ident                    ([r,c] = [dr,dc])
   set([r,c], v1)           ([r,c] != [dr, dc])

copy([sr,sc], [dr,dc], v1) / set([r,c], v2) = copy([sr,sc], [dr,dc], v1)
copy([sr,sc], [dr,dc], v1) \ set([r,c], v2) =
   ident                    ([r,c] = [dr,dc])
   copy([sr,sc],[dr,dc],v1) ([r,c] != [dr,dc])
```

Hidden in this definition, there is the one concession we were forced to make when defining the sequential semantics of the spreadsheet operations. To preserve the contents of the copied cell, we augmented the arguments to the *copy* operation to include it. This is necessary to define *copy*/*set* operating on the same cell.

Instead of completing the *copy* transformations, we defer the discussion of *copy*/*delete* and *copy**delete* because they are the most complex. Instead, we turn our attention to the *delete* transformations. Concurrently deleting rows and/or columns will always "win" over the other three operations. Changing the format of a cell or storing a new value in a cell cannot possibly affect the context of the deletion. Thus:

```
delete / f = delete \ f = delete    (f != delete)
```

The situation is slightly more complicated when dealing with two concurrent deletion requests. We will only discuss the deletion of rows because rows and columns are symmetric. As with the other operations, we may have to translate the row positions to reflect the changes in context. We need to extend this translation slightly because we may have overlapping deletions. If we have already deleted part of the requested area, we simply crop the new area to this previously deleted boundary. Finally, we have defined *delete* in terms of a first row to delete and the number of rows to delete. To transform the number of rows, we project the first row beyond the deleted area into our new context and look at the number of rows between undeleted row and the start of the deleted range in this new context:

```
   delete(fr1, nrow1, fc1, ncol1) / delete(fr2, nrow2, fc2, ncol2) =
      delete(frow(fr1), frow(fr1+nrow1)-frow(fr1), fcol(fc1), fcol(fc1+ncol1)-fcol(fc1))
      where frow(r) = r' and fcol(c) = c' with
         r' = r                 (r < fr2)
         r' = fr2               (fr2 <= r <= fr2+nrow2)
         r' = r - nrow2         (r > fr2+nrow2)
         c' = c                 (c < fc2)
         c' = fc2               (fc2 <= c <= fc2+ncol2)
         c' = c - ncol2         (c > fc2+ncol2)
delete(fr1, nrow1, fc1, ncol1) \ delete(fr2, nrow2, fc2, ncol2) =
   delete(fr1, nrow1, fc1, ncol1) / delete(fr2, ncol2, fc2, ncol2)
```

For completeness, we must define the transformation of two concurrent *copy* operations which should be familiar:

```
copy([sr1,sc1],[dr1,dc1],v1) / copy([sr2,sc2],[dr2,dc2],v2) = copy([sr1,sc1],[dr1,sr1],v1)
copy([sr1,sc1],[dr1,dc1],v1) \ copy([sr2,sc2],[dr2,dc2],v2) =
   ident                        ([dr1,dc1] = [dr2,dc2])
   copy([r1,c1],[dr1,dc1],v1)   ([dr1,dc1] != [dr2,dc2])
```

The only transformations left to define are *copy/delete* and *copy\delete*. We will again have to change the context of the source and destination cells in the copy operation. We will borrow the *drow* and *dcol* functions defined in *set/delete*. However, unlike the set transformations, the relationship between the formula and the context is more complex. When copying cells, the *copy* operation itself defines a transformation on the formula. We derive the transformation for *copy/delete* by taking the definition of the transformations applied by *copy* and *delete* and then solve the required equivalence $f/g(g(S)) = g\backslash f(f(S))$. The definition that we derive is:

```
copy([sr, sc], [dr, dc], <l,e,f>) / delete(fr, nrow, fc, ncol) =
   ident                        (fr <= dr < fr+nrow and fc <= dc < fc+ncol)
   copy([drow(sr), dcol(sc)], [drow(dr), dcol(dc)], <l, fix(e), f>)
                                (otherwise)
   where
      crow(r) = r               (isfixed(r))
      crow(r) = r + dr - sr     (not isfixed(r))
      ccol(c) = c               (isfixed(c))
      ccol(c) = c + dc - sc     (not isfixed(c))

      fix(literal) = literal
      fix(@f(e1, ..., en) = @f(fix(e1), ..., fix(en))
      fix({cell1, cell2}) = { fix(cell1), fix(cell2) }
      fix([r,c]) = [r', c']
      where
         r' = r + drow(crow(r)) - crow(r) + delta_r(r)
         c' = c + dcol(ccol(c)) - ccol(c) + delta_c(c)
         where
            delta_r(r) = (dr-sr) - (drow(dr)-drow(fr)) (not isfixed(r))
            delta_r(r) = 0                             (isfixed(r))
            delta_c(c) = (dc-sc) - (dcol(dc)-dcol(fc)) (not isfixed(c))
            delta_c(c) = 0                             (isfixed(c))
copy(src, dst, v) \ delete(fr, nrow, fc, ncol) =
             copy(src, dst, v) / delete(fr, nrow, fc, ncol)
```

Here, *crow* and *ccol* perform the translation that would have been applied by the original *copy* and, once again, *drow* and *dcol* perform the transformation in positions resulting from the delete operation. The definition of r' and c' serve two purposes: $r'_0 = r + drow(crow(r)) - crow(r)$ corrects a fixed reference by storing the required change in cell position in the transformed formula. The final correction, $delta\_r(r)$ is defined as the change in the transformation that will be applied when the final *copy* operation is actually invoked.

While it is reasonable to trust that the other definitions satisfy the requirements of the CCU algorithm, the definition of *copy/delete* may seem unbelievable. We present the following proof that the definition is correct (satisfies the requirements of the CCU algorithm). The last part of the proof makes the relationship between *delta_r* and *crow* clear.

Theorem 1 (Correctness of *copy/delete*)
If $cpy = copy([sr, sc], [dr, dc], v)$ and $del = delete(fr,\ nrow,\ fc,\ ncol)$ are concurrent operations then $cpy/del(del(S)) = del\backslash cpy(cpy(S))$ for any spreadsheet S.

Proof: By definition, $del\backslash cpy(cpy(S)) = del(cpy(S))$. Let $cpy/del = copy([sr', sc'], [dr', dc'], v')$. There are two conditions that we must verify. First, that $[dr', dc']$ is the same destination cell as in $del(cpy(S))$. We have previously outlined this argument. Applying the delete operation moves the cell $[dr, dc]$ to $[dr', dc']$. By definition of *drow* and *dcol*, that is $[dr', dc'] = [drow(dr), dcol(dc)]$.

The second condition is more difficult. We must verify that if $[r, c]$ is a cell reference in v which becomes $[r'', c'']$ by the sequence $del/cpy(cpy(S))$ then $cpy/del(del(S))$ will also result in a reference of

$[r'', c'']$. Let $[r', c']$ be the transformation of $[r, c]$ as specified in *cpy/del*. Let fix be the transformation function used in the definition of *copy*. We will show that $fix(r') = r'' = drow(crow(r))$. For simplicity, we assume that $dr' - sr' + r \geq 0$. If this is not the case, intermediate results will contain negative cell references but when the final *copy* operation is applied this will be caught and handled correctly.

If r is a fixed reference then *copy/delete* defines $delta\_r(r) = 0$ and *copy* defines $fix(r) = crow(r) = r$. Thus, $fix(r') = r' = r + drow(crow(r)) - crow(r) = r + drow(r) - r = drow(r) = drow(crow(r)) = r''$. That is, fixed references are transformed correctly.

If r is a relative reference then, when applying *cpy/del*, by the definition of fix, we have:

```
fix(r')  =  r' + dr'-sr'
         =  r' + drow(dr) - drow(sr)                                  [def  copy/delete ]
         =  (r + drow(crow(r)) - crow(r) + dr-sr - (drow(dr)-drow(sr))) +
            drow(dr) - drow(sr)                                       [def  r' ]
         =  r + drow(crow(r)) - crow(r) + dr-sr
         =  drow(crow(r))                                             [def  crow(r) ]
```

Finally, a similar argument holds to show that column references are preserved under the two transformations. Thus, $cpy/del(del(S)) = del\backslash cpy(cpy(S))$ for any spreadsheet S. □

Looking beyond the transformations and returning to the spreadsheet abstract data type, we may now outline the additional transformations that would be required. It should be obvious that the semantics involving the *insert* operation are very similar to those with *delete*. The concurrent semantics of a *delete* and an *insert* operation are slightly more complex because the deletion may mask the insertion in the rows and/or the columns modified. The only other operations that we have not defined are name definition services. The simplest technique for handling the name definition services is to make *undefine* operations mask *define* operations and the semantics are then similar to those presented above.

5 Classifying Transformations

It would be desirable to be able to classify all of the possible transformations in order to partition them into related groups. This eases the process of deriving the transformations and simplifies the task of verifying their validity. It is, of course, not possible to define such a taxonomy. Instead, we extract from our concurrent semantics four basic classes of transformations which seem to apply to specific types of functions. The basic classes are pairs of functions that commutate, bully functions, masking functions and functions that affect the relative position of updates.

Commutative Functions When two functions commute, it is trivial to handle the concurrent semantics. This is exploited in [6] to reduce the number of operations that must be undone. For an operational transformation system, it involves an identity transformation. That is, if f and g commute, we define $f/g = f\backslash g = f$ and $g/f = g\backslash f = g$.

Bully Functions Closely related to the commutative functions, there are functions which always "win" against other functions. These functions are very much like the commutative functions except that the relationship holds in only one direction. It may not matter whether or not we have concurrently applied some function, g, when we apply the bully function f. In the spreadsheet, the *delete* function was a bully function. The transformations are simply $f/g = f\backslash g = f$. There is no natural analogy to a text buffer for these types of functions. However, if you consider a text buffer augmented with a locking operation, the locking operation is a bully function [5].

Masking Functions Masking functions are those functions where the earlier (conflicting) applications do not contribute to the final state of the object. For example, if two users set the value of a cell concurrently, it is only the last update that affects the final state of the spreadsheet. Making use of the total order, these transformations are of the form:

```
f(x1,y1) / g(x2,y2) = f(x1,y1)
f(x1,y1) \ g(x2,y2) =
   ident     (x1 = x2)
   f(x1,y1)  (x1 != x2)
```

Masking functions will generally be idempotent functions (functions such that $f(f(x)) = f(x)$). As a special case of the concurrency, two users may generate the same request and this will behave like an idempotent function.

Relative Functions Relative functions are functions which change the relative context in a simple, linear fashion. In terms of the spreadsheet, deletions offset row and column references. In terms of either a text buffer or a spreadsheet, update operations are relative to insertions and deletions. Typically there are up to three possible cases to handle. First, the operation may be nullified (eg: inside the deleted region). If the operation is not nullified then it will either be offset by the relative change or it will be preserved.

6 Future Research

We have presented the concurrent semantics for a spreadsheet but have avoided any discussion of the required infrastructure. This infrastructure must include functionality to create new distributed objects, to acquire a copy of an existing distributed object and to manage a general network topology. Further research is needed into these areas. These topics actually form two specific research tasks. First, there are the network issues. Treating the network as a distributed directed graph (using CCU) may be an effective solution. More thought is needed into the exact ramifications of using CCU to define the network layer used by CCU. Second, there are a set of CCU "meta-operations." It is possible to solve the requirements of the meta-operations in a specific setting, but it is not clear what defines a canonical set of meta-operations. In addition, while it is easy to solve the problems in a specific setting, the simple solutions require some form of global synchronization (among at least half the nodes). This is contrary to the design of the distributed object and thus there is a need to develop distributed, concurrent semantics for the meta-operations. That is, the correct solution to this problem appears to be in defining a concurrent object that includes in its store the original distributed object and defines transformations on top of the definitions provided by the base object.

There is also a strong need for an integrated environment for creating concurrent objects. Given that there are some simple classes of functions that adhere to basic types of transformations, it may be possible to automate a large portion of the definition process. Also, it is inconvenient to define the concurrent semantics using a general purpose language such as C. A simple specification language, possibly similar to the one used in this paper, would greatly assist anyone designing new objects. Due to the volume and potential complexity of the concurrent semantics, there is a definite need to make this process simpler to manage. By making the definition process simpler, it then becomes easier to experiment with new transformations. This ability to experiment is critical in allowing the definition of the concurrent semantics for other interesting objects.

7 Conclusions

An operation transformation algorithm, CCU, was used to develop a shared, distributed, lock-free spreadsheet. This spreadsheet has semantics which are considerably more complex than known operation transformation based systems. The concurrent semantics that were developed maintain a reasonable interpretation of the user's original intention. None of the semantics were sacrificed to ease they definition.

Due to the nature of a lock-free distributed system, local changes to the spreadsheet may always be applied. As these changes are assumed to be distributed over a potentially unreliable network, a highly fault tolerant system has been developed. Every node is able to operate in isolation and thus the system is immune to network outages and any number of nodes may fail without affecting those that remain.

The definition of the transformations that implement the concurrent semantics are mostly contained within four general classes of transformations. We isolated these transformation classes with the intention of making it easier and simpler to think about the transformations and to assist the development of transformations for new objects.

All transformations were based on a simple model of the public domain spreadsheet `sc`. A spreadsheet abstract data type was proposed and formal semantics were defined for a subset of the spreadsheet operations. This subset of operations was rigorously defined as being representative of the complete definitions. Consequently, we have defined a formal model of a spreadsheet with some rigour.

References

[1] David Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 44–57, Asheville, NC, December 1993.

[2] Gordon V. Cormack. A calculus for concurrent update. Technical Report CS-95-06, University of Waterloo, 1995.

[3] Gordon V. Cormack. A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. Technical Report CS-95-08, University of Waterloo, 1995.

[4] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 399–407, Portland, OR, June 1989. ACM SIGMOD Record, **18**:2.

[5] Lin Hu. Handling lock operations in distributed real-time cooperative editing systems. Technical Report TR-96-12, RMIT, Melbourne, Australia, 1995.

[6] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. 13th IEEE Int. Conf. on Distributed Computing Systems*, pages 195–202, Pittsburg, PA, May 1993. Los Alamitos, CA: IEEE Comp. Soc. Press.

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distrubuted system. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *ACM Computer-Supported Cooperative Work*. ACM, November 1996.

[9] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality-preservation and intention- preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interactions*, New York, NY, 1998. To be published.

[10] Andrew S. Tannenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 1992.