Operation Transforms for a Distributed Shared Spreadsheet

Christopher R. Palmer and Gordon V. Cormack
Department of Computer Science
University of Waterloo
Waterloo, Ontario
{crpalmer,gvcormack}@plg.uwaterloo.ca

ABSTRACT

The Distributed Operation Transform (dOPT), proposed by Ellis and Gibbs, is used to define concurrently updatable shared objects. Ellis and Gibbs give the operation transforms that define a simple shared text editor supporting single character insertions and deletions on a linear buffer. We report here on the construction of operation transforms for a more sophisticated groupware application: a shared spreadsheet. We identify a set of abstract operations that characterize the operations on a spreadsheet. Using Cormack's Calculus for Concurrent Update, which extends and corrects dOPT, we give the transforms on these operations necessary to define a shared spreadsheet. We use the transforms to build a shared version of `sc`, the Unix spreadsheet due to Gosling.

KEYWORDS

Groupware, Operation Transforms, Distributed Spreadsheets

INTRODUCTION

We are concerned with asynchronous distributed groupware, in which any copy of a replicated object can be updated at any time, and each update is then propagated to all copies. Asynchronous distributed groupware offers responsiveness and freedom from locking or check-in/check-out procedures even with slow or part-time connectivity. The challenge is to deal with concurrent updates to different copies of the object.

As a simple example, consider a simple text object with the initial value "xyz". The update *insert "a" after the first character* is applied to one copy, with a resulting value of "xayz". Before this update is propagated the update *insert "b" after the second character* is applied to another copy with a resulting value of "xybz". If the updates were propagated naively, the update *insert "b" after the second character* would be applied to the first copy, while the update *insert "a" after the first character* would be applied to the second copy. The end result would be that the two copies would be inconsistent, containing "xabyz" and "xaybz" respectively.

Operation transforms, proposed by Ellis and Gibbs [4], modify concurrent updates as they are propagated so as to ensure that, in the end, all copies are consistent. For the example above, the update *insert "b" after the second character* would, when applied to the other copy, be transformed to *insert "b" after the third character* with the end result that both copies would have the value "xaybz".

To date, operation transforms have been defined only for very simple groupware objects like the example above. Defining operation transforms for a spreadsheet represents a more significant challenge. Our purpose in tackling this problem is twofold: first, a shared spreadsheet is a useful groupware object in its own right; second, its definition allowed us to explore the methodology of defining more complex groupware objects using operation transforms.

OPERATION TRANSFORMS

We use the notation of Cormack's Calculus for Concurrent Update (CCU) [3], which corrects and formalizes the dOPT algorithm of Ellis and Gibbs.

To define a replicated object we begin with an ordinary sequential object; that is, a store and a set of update operations. We then specify the meaning of the concurrent application of every pair of update operations. From these definitions the meaning of all possible combinations of concurrent and sequential updates is automatically derived.

Consider two update operations f and g. Their concurrent meaning is defined as follows. First, an arbitrary but *canonical* consistent order is placed on all operations. Second, a new operation f/g is defined that preserves the meaning of f although a concurrent appli-

e	$\rightarrow$	@function(e_1, ..., e_n)
e	$\rightarrow$	cell
e	$\rightarrow$	range
e	$\rightarrow$	literal
cell	$\rightarrow$	[rowref, colref]
range	$\rightarrow$	{cell, cell}
rowref	$\rightarrow$	row | :row
colref	$\rightarrow$	col | :col

Figure 1: Spreadsheet Expressions

cation of g has intervened. Third, a new operation $g\backslash f$ is defined that has the same effect as if a concurrent application of g had intervened before f. That is, g followed by f/g must have exactly the same effect as f followed by $g\backslash f$. Formally, for every state X of the object and any pair of updates f and g, $f/g(g(X)) = g\backslash f(f(X))$.

We use the text object from the previous section as an example. Suppose

$f =$ *insert "a" after the first character*
$g =$ *insert "b" after the second character*

we may define

$f/g = f\backslash g =$ f
$g/f = g\backslash f =$ *insert "b" after the third character*

In this case f/g and $f\backslash g$ are identical but this is not always the case.

If we choose

$f =$ *insert "c" after the first character*
$g =$ *insert "d" after the first character*

then / and \ must be different; one valid definition is:

$f/g =$ *insert 'c' after the second character*
$f\backslash g =$ *insert 'c' after the first character*
$g/f =$ *insert 'd' after the second character*
$g\backslash f =$ *insert 'd' after the first character*

The definitions can be shown to be valid because g followed by f/g always has the same effect as f followed by $g\backslash f$ (that is, *insert "dc" after the first character*). On the other hand, f followed by g/f has the same effect as g followed by $f\backslash g$ (that is, *insert "cd" after the first character*). Which sequence is used depends on the canonical ordering chosen for f and g.

Further details of CCU and an algorithm to implement replicated objects appear elsewhere [2]. Other algorithms correcting the Ellis and Gibbs algorithm have been proposed [1, 7, 8].

A SPREADSHEET MODEL

Spreadsheets provide sophisticated features for manipulating numeric and textual data. Data is organized in a tabular format with rows and columns of data. Each entry in the spreadsheet table contains data which may include functions of the other entries in the table. To make it easier to duplicate sections of a spreadsheet, including their computation, a special *copy* operation copies entries and updates the formula. These features may be implemented in terms of a few primitives. The set of operations we define are sufficient to model Gosling's `sc`; other spreadsheets, such as `Excel` or `1-2-3`, are similar. Before defining the operations, we need to formalize the store of the spreadsheet.

The store of a spreadsheet is a two-dimensional array of *cells*, a one-dimensional array of default formats, and a set of name bindings.

Each cell contains a value, a formula, and a format. The value of a cell is the result of evaluating its formula, and the format controls the manner in which the value is displayed. Details of the values and formats are unimportant in this discussion. The syntax of a formula is shown in figure 1. Formulas consist of literal values, functions, cell references, and range references. A cell reference denotes the row and column of some other cell. Each row and column denotation may be *fixed* or *relative*. A *fixed* row or column denotation (preceded by a colon in the syntax) is not changed if the formula is copied to another cell. A *relative* row or column denotation is adjusted when the formula is copied so that the relative distance between the formula and the denoted cell is unchanged. For example, if a formula containing a *relative* column denotation were copied two positions to the right, the column denotation would be incremented by 2. A range reference comprises two cell references that specify the upper left and lower right corner of a rectangular area in the spreadsheet.

Seven primitive operations are sufficient to capture all updates on the store: set, format, insert, delete, copy, define, and undefine:

$set(cell, value)$ - Replace the contents of the cell at location *cell* with *value*. In `sc`, *value* is a triple $< l, e, f >$ where l is a potentially empty string label and e is either a numeric or string based expression. Expressions are defined in figure 1. Cell specific formatting information is stored in f.

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

$insert(row, nr, col, nc)$ - Insert rows and/or columns by adding nr rows before row and adding nc columns before col. All cell references will be updated to ensure that they continue to point to the same cell regardless of how the cell may have been moved by the insert.

$delete(row, nr, col, nc)$ - Delete nr rows beginning with

A spreadsheet is a pair $S = <C, F>$ where C is 2 dimensional array of cells, F is a 1 dimensional array of column formats and a cell is the triple $<l, e, f>$. Let C have size $maxr$ rows and $maxc$ columns.

set$([r, c],\ value)$ is
  $C[r, c] = value$

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

copy$([fr, fc],\ [tr, tc],\ <l, e, f>)$ is
  $C[tr, tc] = \ <l, cfix(e), f>$

delete$(row,\ nr,\ col,\ nc)$ is
  FOR all $1 \leq c \leq maxc$ and $1 \leq r \leq maxr$:
    $C[row, c] = dfix(C[row, c])$
  $C[row..maxr - nr, 1..maxc] =$
    $C[row + nr..maxr, 1..maxc]$
  $C[1..maxr, col..maxc - nc] =$
    $C[1..maxr, col + nc..maxc]$
  maxr = maxr - nr
  maxc = maxc - nc

insert$(row,\ nr,\ col,\ nc)$ is
  FOR all $1 \leq c \leq maxc$ and $1 \leq r \leq maxr$:
    $C[r, c] = ifix(C[r, c])$
  $C[row + nr..maxr + nr, 1..maxc] =$
    $C[row..maxr, 1..maxc]$
  $C[row..row + nr - 1] =$ empty
  $C[1..maxr, col + nc..maxc + nc] =$
    $C[1..maxr, col..maxc]$
  $C[1..maxr, col..col + nc - 1] =$ empty
  $F[col + nc..maxc + nc] = F[col..maxc]$
  $F[col..col + nc - 1] = default$
  maxr = maxr + nr
  maxc = maxc + nc

Figure 2: Spreadsheet Definitions

With auxiliary functions:
$cfix(<l, e, f> = \ <l, cfix(e), f>$ where
$cfix(e)$ is
  $cfix(literal) = literal$
  $cfix(@f(e_1, ..., e_n)) = @f(cfix(e_1), ..., cfix(e_n))$
  $cfix(\{cell_1, cell_2\}) = \{cfix(cell_1), cfix(cell_2)\}$
  $cfix([r, c]) = [r', c']$
  where

$r' = r$	(isfixed(r))
$r' = max(r + (tr - fr), 0)$	(isrel(r))
$c' = c$	(isfixed(c))
$c' = max(c + (tc - fc), 0$	(isrel(c))

$dfix(<l, e, f> = \ <l, dfix(e), f>$ where
$dfix(e)$ is
  $dfix(literal) = literal$
  $dfix(@f(e_1, ..., e_n)) = @f(dfix(e_1), ..., dfix(e_n))$
  $dfix(\{cell_1, cell_2\}) = \{dfix(cell_1), dfix(cell_2)\}$
  $dfix([r, c]) = [r', c']$
  where

$r' = r$	$(r < row)$
$r' = r - nr$	$(r \geq row + nr)$
$c' = c$	$(c < col)$
$c' = c - nc$	$(c \geq col + nc)$

$ifix(<l, e, f> = \ <l, ifix(e), f>$ where
$ifix(e)$ is
  $ifix(literal) = literal$
  $ifix(@f(e_1, ..., e_n)) = @f(ifix(e_1), ..., ifix(e_n))$
  $ifix(\{cell_1, cell_2\}) = \{ifix(cell_1), ifix(cell_2)\}$
  $ifix([r, c]) = [r', c']$
  where

$r' = r$	$(r < row)$
$r' = r + nr$	$(r \geq row)$
$c' = c$	$(c < col)$
$c' = c + nc$	$(c \geq col)$

Figure 3: Auxiliary Spreadsheet Definitions

row and nc columns beginning with col. As with the $insert$ operation, all cell references will be updated. Cell references into the deleted region will be left unchanged.

$copy(src, dest, value)$ - Copy the cell at position src, which contains $value$, to $dest$ updating the relative references in $value$ as it is copied. We require that the $value$ of the cell begin copied be attached to this operation to facilitate the 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.

Figure 2 contains a formal definition of the primitive operations, excluding $define$ and $undefine$. Figure 3 contains the auxiliary definitions used to define these operations. The definitions encode the actual implemented semantics of `sc` and they have not been modified in any way to facilitate the definition of their corresponding concurrent semantics.

The set and $format$ definitions merely update the data structures. It is worth noting that set and $format$ are actually updating different data structures. $copy$ defines a mapping, $cfix$, to update the expression be-

$$
\begin{aligned}
&\textbf{format}(c_1,\ f_1)\ /\ \textbf{format}(c_2,\ f_2) = \\
&\quad \mathrm{format}(c_1,\ f_1) \\
&\textbf{format}(c_1,\ f_1)\ \backslash\ \textbf{format}(c_2,\ f_2) = \\
&\quad \mathrm{format}(c_1,\ f_1) && (c_1 \neq c_2) \\
&\quad \mathrm{ident} && (c1 = c2) \\
&\textbf{format}(c,\ f)\ /\ \textbf{set}([r, c],\ v) = \\
&\quad \textbf{format}(c,\ f)\ \backslash\ \textbf{set}([r, c],\ v) = \\
&\qquad \mathrm{format}(c,\ f) \\
&\textbf{format}(c,\ f)\ /\ \textbf{copy}([r, c],\ [r', c'],\ v) = \\
&\quad \textbf{format}(c,\ f)\ \backslash\ \textbf{copy}([r, c],\ [r', c'],\ v) = \\
&\qquad \mathrm{format}(c,\ f) \\
&\textbf{format}(c,\ f)\ /\ \textbf{delete}(fr, nr, fc, nc) = \\
&\quad \mathrm{ident} && (fc \leq c < fc + nc) \\
&\quad \mathrm{format}(c,\ f) && (c < fc) \\
&\quad \mathrm{format}(c - nc,\ f) && (c \geq fc + nc) \\
&\textbf{format}(c,\ f)\ \backslash\ \textbf{delete}(fr, nr, fc, nc) = \\
&\quad \textbf{format}(c,\ f)\ /\ \textbf{delete}(fr, nr, fc, nc) \\
&\textbf{format}(c,\ f)\ /\ \textbf{insert}(fr, nr, fc, nc) = \\
&\quad \mathrm{format}(c,\ f) && (c \leq fc) \\
&\quad \mathrm{format}(c + nc,\ f) && (c > fc) \\
&\textbf{format}(c,\ f)\ \backslash\ \textbf{insert}(fr, nr, fc, nc) = \\
&\quad \textbf{format}(c,\ f)\ /\ \textbf{insert}(fr, nr, fc, nc)
\end{aligned}
$$

Figure 4: Semantics of **format**

$$
\begin{aligned}
&\textbf{set}([r, c],\ v)\ /\ \textbf{format}(c,\ f) = \\
&\quad \textbf{set}([r, c],\ v)\ \backslash\ \textbf{format}(c,\ f) = \\
&\qquad \mathrm{set}([r, c],\ v) \\
&\textbf{set}([r_1, c_1],\ v_1)\ /\ \textbf{set}([r_2, c_2],\ v_2) = \\
&\quad \mathrm{set}([r_1, c_1],\ v_1) \\
&\textbf{set}([r_1, c_1],\ v_1)\ \backslash\ \textbf{set}([r_2, c_2],\ v_2) = \\
&\quad \mathrm{ident} && ([r_1, c_1] = [r_2, c_2]) \\
&\quad \mathrm{set}([r_1, c_1],\ v_1) && ([r_1, c_1] \neq [r_2, c_2]) \\
&\textbf{set}([r, c],\ v_1)\ /\ \textbf{copy}([sr, sc],\ [dr, dc],\ v_2) = \\
&\quad \mathrm{set}([r, c],\ v_1) \\
&\textbf{set}([r, c],\ v_1) \backslash \textbf{copy}([sr, sc],\ [dr, dc],\ v_2) = \\
&\quad \mathrm{ident} && ([r, c] = [dr, dc]) \\
&\quad \mathrm{set}([r, c],\ v_1) && ([r, c] \neq [dr, dc]) \\
&\textbf{set}([r, c],\ v)\ /\ \textbf{delete}(fr,\ nr,\ fc,\ nc) = \\
&\quad \textbf{set}([r, c],\ v)\ \backslash\ \textbf{delete}(fr,\ nr,\ fc,\ nc) = \\
&\qquad \mathrm{ident} && (fr \leq r < fr + nr \text{ or} \\
&\qquad && fc \leq c < fc + nc) \\
&\qquad \mathrm{set}(dfix([r, c]),\ dfix(v)) && (\text{otherwise}) \\
&\textbf{set}([r, c],\ v)\ /\ \textbf{insert}(fr,\ nr,\ fc,\ nc) = \\
&\quad \textbf{set}([r, c],\ v)\ \backslash\ \textbf{insert}(fr,\ nr,\ fc,\ nc) = \\
&\qquad \mathrm{set}(ifix([r, c]),\ ifix(v))
\end{aligned}
$$

where $dfix$ and $ifix$ are defined in figure 2.

Figure 5: Semantics of **set**

ing copied in the manner alluded to above. *insert* and *delete* use the auxiliary functions $ifix$ and $dfix$ respectively to update all of the cells in the spreadsheet. Once the cells have been corrected, they are copied to perform the actual insertion or deletion.

$define$ and $undefine$ are easily defined, but are omitted for brevity. They are implemented by a simple dictionary data structure, and the cell references associated with defined names must be adjusted using $ifix$ or $dfix$ as the result of *insert* or *delete* operations.

CONCURRENT SEMANTICS

To implement the CCU algorithm for a distributed shared spreadsheet, we must define the two transformations, / and \, for all pairs of update operations. That is, we must define $5 \times 5 = 25$ of each transformation. The exact mathematical details of the transforms are shown in figures 4 through 9.

The transformations for *copy* concurrent with *insert* or *delete* are sufficiently complex that a proof that the CCU condition holds is provided. The remaining definitions are easily verified; we concentrate our attention on explaining and clarifying the formal definitions. It is important to see that the transformations "make sense". It is always possible to define transformations that are consistent (on any concurrent update, simply initializing the store defines a set of concurrent semantics that will be consistent). It's not possible to quantify the meaning of the transformations. Instead, as we present the transformations, we will try to make the rationale clear. It is interesting to note that the majority of the transformations are fairly simple yet they will be combined with the CCU algorithm to handle all possible sets of concurrent updates to the spreadsheet.

Semantics of $format$

Concurrently storing a value into a cell is independent of changing the format of a column. If we refer to the model of the spreadsheet, we see that these two operations neither access nor modify any common data structures. As such, we may also perform $format$ operations regardless of any *set* operations that occur concurrently. The same situation applies with *copy* operations.

On the other hand, two different $format$ operations are not necessarily independent. If they are applied to different columns then they actually are independent. If two format operations apply to the same column, we need to resolve them in such a manner as to ensure consistency. The only reasonable solution is to guarantee that in the final state, exactly one of the two format operations will "win" over the other. To do this, we make use of the total order defined by CCU. We say that one format after another format always wins. To achieve consistency, we say that one format before an-

For any of $format$, *set* or *copy* with any arguments, say α:

$$\textbf{delete}(fr,\, nr,\, fc,\, nc) \;/\; \alpha = \textbf{delete}(fr,\, nr,\, fc,\, nc) \setminus \alpha = \text{delete}(fr,\, nr,\, fc,\, nc)$$

$$\textbf{delete}(fr_1,\, nr_1,\, fc_1,\, nc_1) \;/\; \textbf{delete}(fr_2,\, nr_2,\, fc_2,\, nc_2) =$$
$$\text{delete}(frow(fr_1),\, frow(fr_1+nr_1) - frow(fr_1),\, fcol(fc_1),\, fcol(fc_1+nc_1) - fcol(fc_1))$$

where $frow(r) = r'$ and $fcol(c) = c'$ as:

$$\begin{array}{ll}
r' = r & (r < fr_2) \\
r' = fr_2 & (fr_2 \le r \le fr_2 + nr_2) \\
r' = r - nr_2 & (r > fr_2 + nr_2) \\
c' = c & (c < fc_2) \\
c' = fc_2 & (fc_2 \le c \le fc_2 + nc_2) \\
c' = c - nc_2 & (c > fc_2 + nc_2)
\end{array}$$

$$\textbf{delete}(fr_1,\, nr_1,\, fc_1,\, nc_1) \setminus \textbf{delete}(fr_2,\, nr_2,\, fc_2,\, nc_2) =$$
$$\text{delete}(fr_1,\, nr_1,\, fc_1,\, nc_1) \;/\; \text{delete}(fr_2,\, nr_2,\, fc_2,\, nc_2)$$

$$\textbf{delete}(r_1, nr_1, c_1, nc_1) \;/\; \textbf{insert}(r_2, nr_2, c_2, nc_2) =$$
$$\text{delete}(irow(r_1),\, irow(r_1+nr_1)\text{-}irow(r_1),\, icol(c_1),\, icol(c_1+nc_1) - icol(c_1))$$

$$\textbf{delete}(r_1, nr_1, c_1, nc_1) \setminus \textbf{insert}(r_2, nr_2, c_2, nc_2) =$$
$$\text{delete}(r_1, nr_1, c_1, nc_1) \;/\; \text{insert}(r_2, nr_2, c_2, nc_2)$$

$irow(row) = row'$ and $icol(col) = col'$ where:

$$\begin{array}{ll}
row' = row & (row < r_2) \\
row' = row + (nr_2) & (row \ge r_2) \\
col' = col & (col < c_2) \\
col' = col + (nc_2) & (col \ge c_2)
\end{array}$$

Figure 6: Semantics of **delete**

other format is nullified if they are operating on the same column.

Finally, if a format is concurrent with a deletion operation, we simply need to look at the three possible cases. First of all, we may have deleted only rows or the columns being deleted may be to the right of the format operation. In this case, the format operation may safely be applied independently of the deletion. Second, it may be the case that the column being formatted has been concurrently deleted. In this case, we nullify the format operation because it may not be reasonably applied to some other column. Finally, if columns are deleted to the left of the formatted column, the format operation must be transformed to the left by the number of columns that were actually deleted. The same analysis applies for insertions except that the format must be translated to the right.

Semantics of *set*

set and $format$ operations are similar in that they both update data structures and they are independent. Two *set* operations occurring concurrently is analogous to two concurrent $format$ operations since only one operation can "win". Thus, we use the combination of before and after definitions to declare one of the two operations the winner. We have defined the semantics of a *set* operation that occurs concurrently with a *copy* in a similar fashion.

Finally, if a *set* operation occurs concurrently with a *delete*, we have exactly the same three cases as we saw in the $format$ semantics. However, we must take care to apply the same transformation to the expression that is being stored. The *set* operation is transformed into an operation that can be applied after the deletion has occurred to give a common state. This state is the same as having applied the *set* operation followed by the deletion which models what would seem "normal" given that the two operations were applied concurrently to a common initial state. Similar semantics apply for a concurrent insertion of rows and columns.

Semantics of *delete*

The previous two sets of transformations have modeled the concurrent *delete* requests to logically occur after their concurrent counter-parts. This is reflected in the concurrent semantics defined in figure 6. For any of $format$, *set* or *copy*, we simply apply the *delete* untransformed.

All that remains is to define the meaning of two concurrent *delete* operations. We will discuss the deletion of rows (deletion of columns is similar). The first observation is that the order of two deletions does not mat-

$\mathbf{copy}([s_r, s_c], [d_r, d_c], v)$ / $\mathbf{format}(c, f) =$
$\mathbf{copy}([s_r, s_c], [d_r, d_c], v) \setminus \mathbf{format}(c, f) =$
copy$([s_r, s_c], [d_r, d_c], v)$
$\mathbf{copy}([s_r, s_c], [d_r, d_c], v_1)$ / $\mathbf{set}([r, c], v_2) =$
copy$([s_r, s_c], [d_r, d_c], v_1)$
$\mathbf{copy}([s_r, s_c], [d_r, d_c], v_1) \setminus \mathbf{set}([r, c], v_2) =$
ident $\quad ([r, c] = [d_r, d_c])$
copy$([s_r, s_c], [d_r, d_c], v_1) \quad ([r, c] \neq [d_r, d_c])$
$\mathbf{copy}([s_{r_1}, s_{c_1}], [d_{r_1}, d_{c_1}], v_1)$ /
$\mathbf{copy}([s_{r_2}, s_{c_2}], [d_{r_2}, d_{c_2}], v_2) =$
copy$([s_{r_1}, s_{c_1}], [d_{r_1}, s_{r_1}], v_1)$
$\mathbf{copy}([s_{r_1}, s_{c_1}], [d_{r_1}, d_{c_1}], v_1) \setminus$
$\mathbf{copy}([s_{r_2}, s_{c_2}], [d_{r_2}, d_{c_2}], v_2) =$
ident $\quad ([d_{r_1}, d_{c_1}] = [d_{r_2}, d_{c_2}])$
copy$([s_{r_1}, s_{c_1}], [d_{r_1}, d_{c_1}], v_1) \quad ([d_{r_1}, d_{c_1}] \neq [d_{r_2}, d_{c_2}])$
$\mathbf{copy}([s_r, s_c], [d_r, d_c], < l, e, f >)$ /
$\mathbf{delete}(fr, nr, fc, nc) =$
ident $\quad (fr \leq d_r < fr + nr,$
$fc \leq d_c < fc + nc)$
copy$([drow(s_r), dcol(s_c)], [drow(d_r), dcol(d_c)],$
$< l, dfix(e), f >)$ (otherwise)
$\mathbf{copy}([s_r, s_c], [d_r, d_c], v) \setminus$
$\mathbf{delete}(fr, nr, fc, nc) =$
copy$([s_r, s_c], [d_r, d_c], v)$ / delete(fr, nr, fc, nc)
$\mathbf{copy}([s_r, s_c], [d_r, d_c], < l, e, f >)$ /
$\mathbf{insert}(fr, nr, fc, nc) =$
copy$([irow(s_r), icol(s_c)], [irow(d_r), icol(d_c)],$
$< l, ifix(e), f >)$
$\mathbf{copy}([s_r, s_c], [d_r, d_c], v) \setminus$
$\mathbf{insert}(fr, nr, fc, nc) =$
copy$([s_r, s_c], [d_r, d_c], v)$ / insert(fr, nr, fc, nc)

Figure 7: Semantics of **copy**

$crow(r) = r \quad$ (isfixed(r))
$crow(r) = r + d_r - s_r \quad$ (isrel(r))
$ccol(c) = c \quad$ (isfixed(c))
$ccol(c) = c + d_c - s_c \quad$ (isrel(c))

$dfix(literal) = literal$
$dfix(@f(e_1, ..., e_n) = @f(dfix(e_1), \cdots, dfix(e_n))$
$dfix(\{cell_1, cell_2\}) = \{dfix(cell_1), dfix(cell_2)\}$
$dfix([r, c]) = [r', c']$
where
$r' = r + drow(crow(r)) - crow(r) + delta\_r(r)$
$c' = c + dcol(ccol(c)) - ccol(c) + delta\_c(c)$
where
$delta\_r(r) = (d_r - s_r) - (drow(d_r) - drow(fr))$
(isrel(r))
$delta\_c(c) = (d_c - s_c) - (dcol(d_c) - dcol(fc))$
(isrel(c))
$delta\_r(r) = 0 \quad$ (isfixed(r))
$delta\_c(c) = 0 \quad$ (isfixed(c))

$ifix(literal) = literal$
$ifix(@f(e_1, ..., e_n) = @f(ifix(e_1), \cdots, ifix(e_n))$
$ifix(\{cell_1, cell_2\}) = \{ifix(cell_1), ifix(cell_2)\}$
$ifix([r, c]) = [r', c']$
where
$r' = r + irow(crow(r)) - crow(r) + delta\_r(r)$
$c' = c + icol(ccol(c)) - ccol(c) + delta\_c(c)$
where
$delta\_r(r) = -(d_r - s_r) + (irow(d_r) - irow(fr))$
(isrel(r))
$delta\_c(c) = -(d_c - s_c) + (icol(d_c) - icol(fc))$
(isrel(c))
$delta\_r(r) = 0 \quad$ (isfixed(r))
$delta\_c(c) = 0 \quad$ (isfixed(c))
$irow(row)$ and $icol(col)$ are defined in figure 6.

Figure 8: Auxiliary Definitions for **copy**

ter. We need only define the semantics to transform a *delete* operation to reflect the change in context caused by a previously applied, concurrent, deletion. There are four cases. First, the previously applied *delete* may have only deleted rows below the new *delete* request at which point the new request may be applied untransformed. Second, the previous deletion may have been entirely above the current deletion. In this case, we need to translate the deleted area up by the number of rows previously deleted. The last two cases deal with overlapping deletions. If the previous deletion deleted all of the rows being deleted by the new deletion, it may be transformed into an identity function. Alternatively, if they truly overlap, we need to translate the first row to be deleted to be the first row in the deleted range that remains and we need to reduce the number of rows to be deleted by the number of overlapping rows.

This will result in deleting exactly those rows that at least one of the deletions had requested. The transformation has been slightly condensed by merging the last two cases knowing that, if the deletion has been subsumed, then we will have $frow(fr_1 + nr_1) = frow(fr_1) = fr_2$ and consequently no rows will be deleted.

The situation is somewhat simpler when handling a concurrently applied insertion of rows and/or columns. If we are applying a deletion of rows after a concurrent insertion has been applied, we may need to transform the position of the deletion or we might need to transform the number of rows being deleted. If the insertion occurs in the middle of the deleted area, we need to delete all of the original rows plus the number of rows that were inserted. On the other hand, if the insertion occurs above the deletion, we need to translate the deletion by the number of rows that were inserted. It is never possible for the insertion to overlap with a deletion.

Semantics of *copy*

The definitions of the concurrent semantics for the *copy* operation are presented in figures 7 and 8. The *copy* and *format* operations are independent and consequently no transformation is required. Before discussing the other transformations, it is worth commenting on the *value* argument on the *copy* operation. If some site receives a $set([r, c], v)$ operation and processes it before a concurrent $copy([r, c], [r', c'], v')$, the receiving site will no longer have access to the original value of the cell. Thus, we have preserved the value being copied.

A *copy* that occurs concurrently with a *set* or a *copy* is treated identically. One operation will be the "winner" and we model this to be consistent with the model applied for the *set* semantics. It should be noted that this is not the only possible definition. It might be appealing to rely on the total order. That is, we may choose to care whether or not the source cell of the *copy* operation was updated before or after the *copy* in the total order. If the source was modified then we would transform the *copy* operation to use this new value. Our alternative definition is possible and may actually make the most mathematical sense. However, we felt that users would generally choose to *copy* a cell because the contents were of interest. It is not necessarily true that the new contents would be of interest. To be fair, it should be noted that this is merely the preference of the authors and either definition is likely acceptable.

We have previously chosen to model a *copy* operation that is concurrent with a *delete* operation by conceptually applying the deletion after the copy has taken place. The *copy* transformations must be defined to be consistent with this model. This transformation is not obvious and was actually mathematically derived. The first part of the transformation transforms the source and destination cells of the *copy* operation exactly as was done with *set* and *format* operations that occur concurrently with a *delete*. The problem lies in transforming the expression so that when the *copy* algorithm is applied to the *transformed* source and destination cells in a spreadsheet in which the *delete* has already occurred, the resulting formula will be the same as the formula arrived at by performing the operations in the reverse order. A careful analysis of the transformations will reveal that insertions are handled identically except for a difference in sign (and that insertions may never nullify the *copy*).

We provide a proof that the CCU condition holds for the definition (for any spreadsheet). The manipulation at the end of the proof makes clear the relationship between *delta_r* and *crow* in the definition.

Theorem 1 (Correctness of ***copy/delete*****)**
If
 cpy = copy($[sr, sc]$, $[dr, dc]$, v) and
 del = delete(fr, nr, fc, nc)
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:

$$
\begin{aligned}
\text{fix(r')} &= \text{r' + dr'-sr'} \\
&= \text{r' + drow(dr) - drow(sr)} \\
&= \text{(r + drow(crow(r)) - crow(r) +} \\
&\qquad \text{dr-sr - (drow(dr)-drow(sr))) +} \\
&\qquad \text{drow(dr) - drow(sr)} \\
&= \text{r + drow(crow(r)) - crow(r) + dr-sr} \\
&= \text{drow(crow(r))}
\end{aligned}
$$

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. □

Semantics of *insert*

As with deletions, insertion is not affected by any operation other than a concurrent *insert* or a concurrent *delete* operation. We discuss only the row based operations while the column based analysis is similar.

If a concurrent *delete* has been applied, we will have to transform the *insert* request. If the area being deleted covers the insertion point then we will nullify

For any of *format*, *set* or *copy* with any arguments, say α:

$\mathbf{insert}(fr,\ nr,\ fc,\ nc)\ /\ \alpha =$
$\quad \mathbf{insert}(fr,\ nr,\ fc,\ nc) \setminus \alpha =$
$\quad \text{insert}(fr,\ nr,\ fc,\ nc)$

$\mathbf{insert}(r_1, nr_1, c_1, nc_1)\ /\ \mathbf{delete}(r_2, nr_2, c_2, nc_2) =$

ident	$(r_2 \leq r_1 < r_2 + nr_2,$ $c_2 \leq c_1 < c_2 + nc_2)$
$\text{insert}(0,\ 0,\ idcol(c_1),\ nc_1)$	$(r_2 \leq r_1 < r_2 + nr_2)$
$\text{insert}(idrow(r_1),\ nr_1,\ 0,\ 0)$	$(c_2 \leq c_1 < c_2 + nc_2)$
$\text{insert}(idrow(r_1),\ nr_1,\ idcol(c_1),\ nc_2)$	(otherwise)

$\mathbf{insert}(r_1, nr_1, c_1, nc_1) \setminus \mathbf{delete}(r_2, nr_2, c_2, nc_2) =$
$\quad \text{insert}(r_1, nr_1, c_1, nc_1)\ /\ \text{delete}(r_2, nr_2, c_2, nc_2) =$

$\mathbf{insert}(r_1, nr_1, c_1, nc_1)\ /\ \mathbf{insert}(r_2, nr_2, c_2, nc_2) =$
$\quad \text{insert}(irow(r_1),\ nr_1,\ icol(c_1),\ nc_1)$

$\mathbf{insert}(r_1, nr_1, c_1, nc_1) \setminus \mathbf{insert}(r_2, nr_2, c_2, nc_2) =$
$\quad \text{insert}(r_1, nr_1, c_1, nc_1)\ /\ \text{insert}(r_2, nr_2, c_2, nc_2) =$

$idrow(row) = row'$ and $idcol(col) = col'$ where:

$row' = row$	$(row < r_2 + nr_2)$
$row' = col - nr_2$	$(row \geq r_2 + nr_2)$
$col' = col$	$(col < c_2 + nc_2)$
$col' = col - nc_2$	$(col \geq c_2 + nc_2)$

$irow(row)$ and $icol(col)$ are defined in figure 6.

Figure 9: Semantics of **insert**

the insert. Note, however, if the insertion occurs after the last row being deleted, it will still occur. That is, if we insert after row 2 and concurrently someone deletes row 2, the insertion will not be nullified. Otherwise, if the deletion applied to rows above the insertion point, we will translate the insertion point by the number of rows deleted.

If we have concurrently applied another *insert* operation, we need only transform the insertion point. That is, if the previously applied insertion was above or at the same insertion point of the operation that we want to apply, we must translate this new operation by the number of rows inserted. Otherwise, the *insert* operation is untransformed.

IMPLEMENTATION ISSUES

A prototype distributed shared spreadsheet was developed based on the public domain spreadsheet `sc`. The most difficult part of the implementation was the development of the concurrent semantics. By examining the behaviour of `sc`, we convinced ourselves that the set of operations we have proposed is sufficient to model the actual implementation. Once the model and concurrent definitions were developed, it was necessary to integrate them into `sc`. The integration was time consuming but simple. Any code that updated the spreadsheet had to be examined. The code was modified to make the appropriate core function calls. These core functions generate network packets that are distributed to all other sites as notifications.

In addition to changing the low level details of `sc`, care had to be taken to ensure that network traffic was handled correctly. Whenever the system blocks waiting for user input, network packets are read. If the user is in the process of entering a command (for example, writing an equation to store in a cell), remote updates are disabled and the network packets are simply queued. Whenever the user is idle, network packets are received and processed along with any packets that were queued. This automatically provides the required atomicity for aggregate operations and provides a rational interface. If remote updates were constantly being processed, it would be nearly impossible to write an expression because the underlying spreadsheet would be a moving target.

Communication is done using the TCP protocol. It is possible to implement lost packet detection directly using the CCU algorithm (see [2] for details). Switching to UDP based network protocols is an avenue for improvement in the current implementation.

The CCU algorithm is used to process the updates. No changes are required to the CCU algorithm and the transforms are implemented exactly as specified (using a suitable encoding of expressions and operations).

The initial prototype was used and an interesting flaw was noticed and corrected. When storing a value in the spreadsheet, a special case arises when the destination cell is outside of the current spreadsheet boundaries. There may be a high transmission delay (for example, when some site is not connected to a network). When transmission delays are low, this problem is easily corrected manually by the users involved. If two users decide to add new information to the spreadsheet, they will invariably start changing the cells at the bottom or the right side of the spreadsheet. CCU and our transformations will consistently declare the updates made by one of the two users the "winner" over the other set of updates. That is, one user will lose their data. This is a case in which the concurrent semantics do not match the intention of the users. The problem is corrected by inserting a row whenever a value is stored outside of the spreadsheet boundaries. Making this change to the system guarantees that the two sets of updates will be reflected in the spreadsheet.

This example reinforces the fact that an implementation and user trial is essential. The model and concurrent semantics were provably correct but an oversight in the meaning of an update seriously reduced the usefulness of the spreadsheet.

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 for which 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 (e.g. 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.

CONCLUSIONS

An operation transformation algorithm, CCU, was used to develop a shared, distributed, 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 modified to ease the definition of their corresponding concurrent semantics.

Due to the nature of an operation transform 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 defined transformations fall largely into four general classes. The nature of these transformation classes has been identified to provide a general framework that will assist others attempting to design large scale transformation based systems.

All transformations were based on a model of the public domain spreadsheet `sc`. A spreadsheet abstract data type was proposed and formal semantics were defined for the core components of this model.

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 calculus for concurrent update (abstract). In *Proc. ACM Symposium on Principles of Distributed Computing*, 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. 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.

8. 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.