An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors

Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser
Institute for Computer Science
University of Stuttgart
Breitwiesenstr. 20–22,
D-70565 Stuttgart, Germany
{ressel,nitsche,rul}@informatik.uni-stuttgart.de

ABSTRACT
Concurrency control and group undo are important issues in the design of groupware, especially for interactive group editors. We present an improved version of an existing distributed algorithm for concurrency control that is based on operation transformations. Since the usability of the algorithm relies on its formal correctness, we present a set of necessary and sufficient conditions to be satisfied in order to ensure consistency in a replicated architecture. We identify desirable properties of operation transformations and show how our approach can be employed to implement group undo. The approach has been applied to build a prototypical group editor for text; some experiences gained are presented.

Keywords
Concurrency Control, Group Editors, Group Undo, Groupware, Interaction Model, Operation Transformation.

INTRODUCTION
Groupware systems, i.e., multi-user applications for cooperative work [2, 5], present remarkable challenges for application designers. Concurrency control and group undo are two of them.

Concurrency control covers the management of parallel threads of human-computer interaction including potential conflicts. These typically arise in synchronous collaborative work when different users concurrently manipulate a shared object. A replicated architecture, where users work on copies of the shared object and instructions are exchanged by message passing, has been identified as appropriate for groupware [4]. In this way, users may continue local work even if the network fails. Screen updates – a frequent operation in common graphical user interfaces – can be accelerated, independently of network delays. The responsiveness of the user interface can be improved by the immediate execution of each individual's operations at the local site. Methods for concurrency control supporting this are called optimistic [6]. This entails, however, that operations are not necessarily executed in the same sequence at different sites. Certain operations have then to be executed in an application state different from the one they were originally issued in. The usability of a method for concurrency control relies both on its correctness, i.e., being able to maintain consistency among the copies of a shared object, and on the production of meaningful results that meet users' expectations.

Undo in a groupware system has to enable a user to revert the effects of an operation even if other users have issued further operations afterwards [1]. Therefore group undo has to define the effects of undoing an operation that might not be the last one in the interaction history. Undo of such operations has to be interpreted and executed, however, in the current state of the application.

The transformation-oriented dOPT-algorithm by Ellis and Gibbs [4] forms a promising approach to optimistic concurrency control. Their basic idea is to take an operation executed in some past state and to transform it in a way that it can be applied to the current state. We included the dOPT-algorithm in a framework for building group editors. In practise, however, the dOPT-algorithm did not work in all cases. We found out, that inconsistent application states may result if one user issues and executes more than one operation concurrently to an operation of another user. An example will be given in the body of this paper. Fortunately, we came up with an improved version of the dOPT-algorithm that will be presented here. A novel multi-dimensional model of concurrent interaction forms the core of our approach, enabling us to prove its correctness. It turned out, that our algorithm can help to implement group undo as well. Thus we contribute a new, integrating, transformation-oriented approach to both concurrency control and group undo. Finally, we present some experiences we have gained in building a prototypical group text editor.

MODEL OF A GROUPWARE SYSTEM
Before presenting our approach of request transformations some understanding of a groupware system is necessary. The definitions have been taken in main parts from [4], but they have been modified if appropriate. A *groupware system* con-

sists of a set S of *application states*, a set U of *users* and a set O of *operations*. For convenience, users shall be identified by natural numbers, hence $U = \{1, 2, \ldots, m\}$. Since we are interested in groupware with a replicated architecture each user's site will hold a local application state s that is modified only by the local application process. $s_o \in S$ shall denote the *initial application state* which is identical for all sites. *apply* is a function from $S \times O \times U$ into S and computes the resulting state s' when user u applies operation o to application state s: *apply*$(s, o, u) = s'$. We assume the existence of an *identity operation* I, that applied to any state s results in the same state s, independently of the user u. A groupware application has to define the function *apply* for any allowable combination of states, operations and users. The domain of definition of *apply* is usually a proper subset of $S \times O \times U$ and can be used to specify restrictions that certain users may not issue certain operations or may not issue any operations at all. A groupware application has to define an inverse operation $\overline{o}$ for every operation o that should be reversible, where *apply*(*apply*$(s, o, u), \overline{o}, u) = s$. New here is that the user information acts as an argument of the *apply* function.

In a simple group editor, e.g., S is the set of character strings, O consists of operations to delete and insert characters, to move the cursor, etc. The insertion of a character c at buffer position p is denoted by $Ins[p, c]$. The deletion of character c from position p is denoted by $Del[p, c]$. Operation $Del[2, a]$, e.g., can only be applied if the second element of the current string is an "a". The inverse operation of $Del[2, a]$ is $Ins[2, a]$.

Input from a user generates a *request*, that is broadcast to the other participants for execution. We make the assumption that a user generates requests sequentially, hence they can be serially numbered starting with number 1. If this is seen as a restriction, each parallel input thread may be regarded as another virtual user. So a request can be identified uniquely by the user identification and its serial number.

Each site executes the requests in a sequential but possibly different order. Typically, a process will execute local requests as soon as possible in order to guarantee short response times. Since by assumption the requests from each user are executed in the same order as generated, i.e., as given by their serial number, a set of executed request can be described by the number of requests it contains from each user. This motivates the definition of a *state vector* $v = (x_1, x_2, \ldots, x_m)$, where x_i denotes the number of executed requests from user i. The i-th component of a vector v will be written as $v[i]$. A state vector v_1 is defined to be smaller (*earlier*) than v_2 if each component of v_1 is smaller than or equal to the corresponding component in v_2 and at least one component is smaller.

We do not consider application states when a request is in execution. Therefore the state vector corresponding to any application state that is of interest is uniquely defined.

We define a *request* r more formally as a quadruple (u, k, v, o), where $u \in U$ is the user issuing the request, $k \in \mathbf{N}$ is its serial number, $v \in \mathbf{N}^m$ is the state vector that describes what requests had already been executed in the state when the request was generated and should have been executed originally, and finally $o \in O$ is the operation to be applied. $u(r), k(r), v(r)$, and $o(r)$ will be used to denote the corresponding components of a request. $w(r)$ shall describe the state vector after the execution of request r. For a given request r, it follows: $w[u] = v[u] + 1$ and $w[i] = v[i], i \neq u$, where $u = u(r)$, $w = w(r)$, and $v = v(r)$. Compared to [4], no special priority value is needed, the user id fulfills this role sufficiently. The serial number is new, it is mainly used in combination with the user id to identify user requests.

A request r is executed in the state $s \in S$ corresponding to the state vector $v(r)$ by evaluating *apply*$(s, o(r), u(r))$.

Definition of Correctness

Similarly to the definition by Ellis and Gibbs [4] we define a groupware system as *correct* when it satisfies the precedence property and the convergence property.

A request r_1 is said to precede r_2, when its execution might have an effect on r_2. E.g., request r_1 precedes request r_2, if r_2 is issued in an application state, where r_1 has already been executed. The precedence relation defines a partial order among requests. In particular, it is not possible that any requests depend upon each other cyclically.

The *precedence property* requires that – on every site – a request has to be executed after all preceding requests. As has been found by Ellis and Gibbs [4], the precedence property is satisfied, if a request r is made *executable* in a state with state vector v if and only if its state vector $v(r)$ is smaller than or equal to v: $v(r) \leq v$.

The *convergence property* requires that the application state depends only on the set of executed requests and not on the distinct execution sequence at each site.

A sequence of requests $p = r_1 r_2 \ldots r_l$ is said to be a *valid* sequence if $\forall i = 1 \ldots l - 1 : w(r_i) = v(r_{i+1})$. We define $v(p) := v(r_1)$ and $w(p) := w(r_l)$. The execution of a valid sequence of request is defined as the sequential execution of its requests. Two valid sequences p_1, p_2 of requests are said to be *equivalent*, $p_1 \equiv p_2$, if being executed in the same initial state they produce the same state (cf the use of this equivalence relation in Definition 1 below).

REQUEST TRANSFORMATIONS

Requests that cannot be executed in the original application state as specified by their state vector are transformed in order to take into account any requests that have been executed afterwards.

Example 1 Using a group text editor, user 1 deletes character "b" at position 2, $Del[2, b]$, and user 2 concurrently inserts character "a" at position 1, $Ins[1, a]$, (cf fig. 1). Both users apply their operations immediately to their own identical copies of the application state, here the string "xby". At site 1, the executed deletion request does not affect the insertion because the deletion may only have effects on the text to the right of position 2 (cf fig. 1a). At site 2, the executed insertion request, however, requires that the deletion operation is shifted one position to the right, yielding the operation $Del[3, b]$. In this way, both the resulting strings are "axy". □

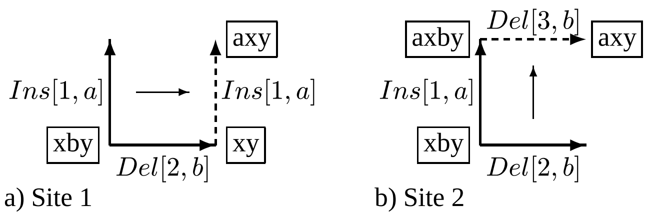


Figure 1: Necessary transformation in Example 1

The idea behind our transformation-based method is to apply the transformations repeatedly to obtain valid and thereby executable sequences of requests at every site. In contrast to the dOPT-algorithm, intermediate requests being the results of transformation steps are memorized. This forms the foundation for the correctness of our method.

Example 2 Let the initial string of a text editor be "abcd". User 1 deletes the character "c" at position 3, $Del[3,c]$. User 2 first deletes character "a" at position 1, $Del[1,a]$, and then inserts character "x" in front of the character at position 3, $Ins[3,x]$ (cf fig. 2). According to the dOPT-algorithm, a request r_j from site j has to be transformed with respect to all requests in the linear history, which were not accounted for by user j, i.e., requests which are already executed locally, but which were not executed on site j prior to the generation of r_j. Therefore, when site 1 receives the message from site 2 with operation $Del[1,a]$, this is transformed with respect to $Del[3,c]$, requiring no modification. When the second operation, $Ins[3,x]$, arrives this is also transformed with respect to $Del[3,c]$, again requiring no modification. These operations yield a final string of "bdx" at site 1 (s. fig. 2a). When site 2 receives a message from site 1 with operation $Del[3,c]$ this is transformed twice, first with respect to $Del[1,a]$ yielding $Del[2,c]$ and then with respect to $Ins[3,x]$, requiring no further modification. The final string at site 2 is "bxd". The final strings at site 1 and 2 are different, the application states became inconsistent. □

The reason, why the dOPT-algorithm fails in this case, is that transforming $Ins[3,x]$ with respect to the original operation $Del[3,c]$ and with respect to the transformed operation

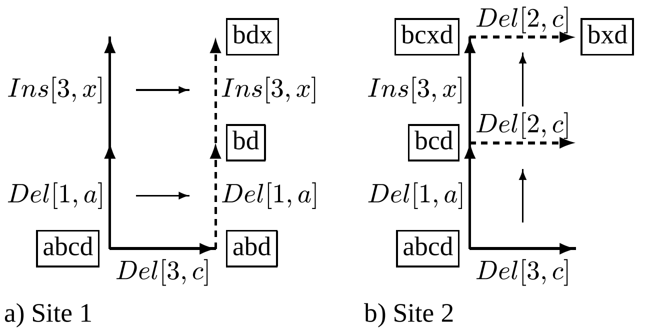


Figure 2: Inconsistent transformations at different sites (Counterexample for dOPT-algorithm)

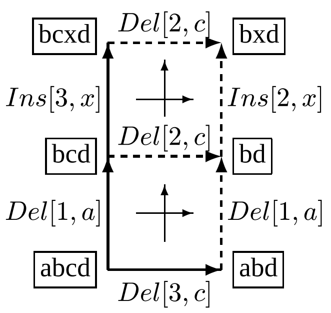


Figure 3: Consistent transformations

$Del[2,c]$ gives different results. To produce consistent application states, at site 1 (cf fig. 2a) the second operation from site 2, $Ins[3,x]$, should have been transformed with respect to $Del[2,c]$, the result of transforming $Del[3,c]$ with respect to $Del[1,a]$ (cf fig. 3). In fact, it took us a long time to detect this flaw. Our solution became evident, however, as we had the idea of visualizing transformations as grid-like diagrams.

L-Transformation functions

Formally, we define a transformation function *tf* as a function from $R \times R$ into $R \times R$ being defined for all $r_1, r_2 \in R$ with $v(r_1) = v(r_2)$ and $u(r_1) \neq u(r_2)$, such that $r_1 r_2'$ and $r_2 r_1'$ are valid sequences with $w(r_1 r_2') = w(r_2 r_1')$ if $tf(r_1, r_2) = (r_1', r_2')$. For convenience, we also define functions to retrieve the components of a transformation: $tf_1(r_1, r_2) := r_1'$ and $tf_2(r_1, r_2) := r_2'$, if $tf(r_1, r_2) = (r_1', r_2')$.

Definition 1 (Transformation Property 1)
If $tf(r_1, r_2) = (r_1', r_2')$ then $r_1 \; r_2' \equiv r_2 \; r_1'$

The transformation property ensures that the effect of executing request r_1 followed by the transformed request r_2' is the same as executing request r_2 followed by the transformed request r_1'. This property is illustrated in fig. 4.

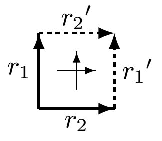


Figure 4: L-Transformation: $r_1 r_2' \equiv r_2 r_1'$

Definition 2 (Symmetry Property)
If $tf(r_1, r_2) = (r_1', r_2')$, then $tf(r_2, r_1) = (r_2', r_1')$

The Symmetry Property ensures that in a replicated architecture, given two requests, every user process will compute the same transformed requests, not depending on the order in which they are handed to the *tf* function.

A transformation function that satisfies the Transformation Property 1 and the Symmetry Property is referred to as an *L-transformation function*. The notion *L-transformation* is derived from the L-shape formed by the requests r_1 and r_2 that are to be transformed (cf fig. 4).

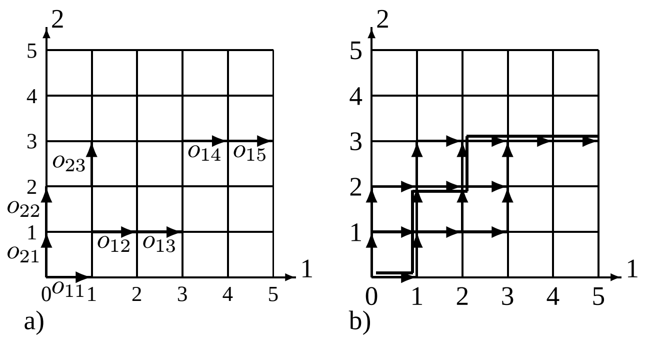


Figure 5: a) Set of user requests, b) Interaction model with a possible execution path

A transformation function *tf* that satisfies the Symmetry Property can be reduced to a function tf_1 by defining $tf(r_1, r_2) = (tf_1(r_1, r_2), tf_1(r_2, r_1))$. Since the user id, the serial number and the state vector of the resulting request can be deduced from the arguments of tf_1, it is actually sufficient to define the operation of the resulting request. So, for convenience, another function T_1 is introduced, where $T_1(r_1, r_2) = o'_1$ if $tf_1(r_1, r_2) = (u_1, k_1, v'_1, o'_1)$ (cf use of this function in the definitions of transformation rules below).

INTERACTION MODEL

As has been shown in preceding figures, requests and L-transformations can be represented as grid-based diagrams. The axes represent users, grid points represent states with a certain state vector, and arrows represent requests, being either original or the result of some transformation. Fig. 5 shows a set of eight requests, five from user 1 and three from user 2. The origin of each arrow describes the state vector of the request represented by this arrow. The operation of a request may be written next to the arrow. From our assumption, that each user generates and executes his or her requests in sequence, it follows, that each row and each column may contain at most one original user request.

Application of the L-transformation function will yield further requests that will build a dense network of arrows spreading across the coordinate grid, representing a model of interaction (fig. 5b). Any execution sequence of requests that satisfies the Precedence Property corresponds one-to-one to a path in the diagram.

More formally we define an *interaction model* for a set M of user requests and for an initial state s_0 as a labeled, directed graph G with vertices V and edges $E \subseteq V \times V$. Vertices are labeled with application states, edges with requests (although, it is often more convenient to label them just with the operation, since the other components of a requests can be determined by the position and direction of an edge). The set of vertices, edges and their labels are defined recursively as follows:

$$v_0 = (0, 0, \ldots, 0) \in V, \text{ labeled } s_0 \tag{1}$$

$$\begin{aligned} &\text{if } r \in M \text{ and } v(r) \in V, \text{then} \\ &\quad w(r) \in V \text{ and} \\ &\quad (v(r), w(r)) \in E, \text{ labeled } r \end{aligned} \tag{2}$$

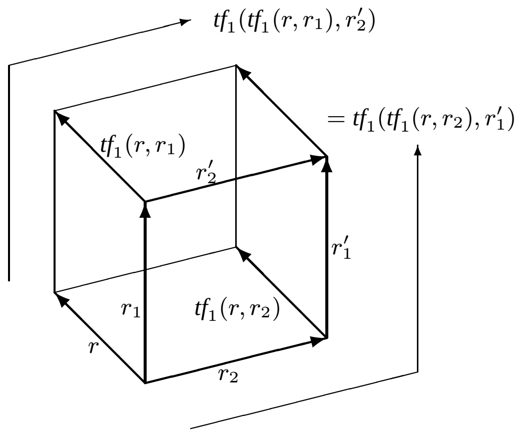


Figure 6: Transformation Property 2: Consistent transformations of request r

$$\begin{aligned} &\text{if } v \in V, \text{ labeled } s, \text{and} (v, w) \in E, \text{ labeled } r, \text{ then} \\ &\quad w \text{ is labeled with value of } apply(s, o(r), u(r)) \end{aligned} \tag{3}$$

$$\begin{aligned} &\text{if } (v, w_1), (v, w_2) \in E \text{ labeled } r_1, r_2, \text{ resp., then} \\ &\quad (w_1, w) \in E, \text{ labeled } r'_2 \text{ and} \\ &\quad (w_2, w) \in E, \text{ labeled } r'_1 \text{ where} \\ &\quad tf(r_1, r_2) = (r'_1, r'_2) \text{ and } w = sup(w_1, w_2) \end{aligned} \tag{4}$$

$sup(w_1, w_2)$ in step (4) denotes a state vector that has as coordinates the maximum of the corresponding coordinates of the vectors w_1 and w_2 (e.g., $sup((1, 3), (5, 2)) = (5, 3)$). Hence, if $tf(r_1, r_2) = (r'_1, r'_2)$, then $w(r'_1) = w(r'_2) = sup(w(r_1), w(r_2))$.

Step (1) represents the initialization of the application state, step (2) the generation of new user requests, step (3) the state transitions, and step (4) the request transformations. By definition, each edge and each vertex will have at least one label. We are interested only in interaction models whose vertices have unique labels. That implies that the convergence property is satisfied, provided that the initial states are identical at each site.

Proof of correctness

The Transformation Property 1 and the Symmetry Property are necessary conditions, since otherwise an interaction model that consists of just one transformation would lead to ambiguous labeling and thus to inconsistent application states at different sites. In fact, these two necessary conditions are even sufficient conditions for a groupware system with $m = 2$ users to be correct. It can easily be seen, that from a unique labeling of v_0 and unique labels at the edges representing original requests – due to the unambiguity of the transformation function – it follows that all additional labels of edges will be unique, too. With Transformation Property 1 it then follows, that the labels of the vertices will also be unique. Q.e.d.

The proof for more than two users is more complex: A request can be transformed along different, albeit equivalent paths, not necessarily yielding the same request. In the simplest case, a request can be transformed along the two paths of a simple transformation step, as shown in fig. 6. Request r may be transformed first with respect to r_1 and then to r'_2 yielding $tf_1(tf_1(r, r_1), r'_2)$ – or it may be transformed first with respect to r_2 and then to r'_1 yielding $tf_1(tf_1(r, r_2), r'_1)$. Note that different sites might choose different paths for r to be transformed. So we have to make sure that both paths lead to the same resulting request:

Definition 3 (Transformation Property 2) (fig. 6)
If $tf(r_1, r_2) = (r'_1, r'_2)$,
then $tf_1(tf_1(r, r_1), r'_2) = tf_1(tf_1(r, r_2), r'_1)$.

Transformation Property 2 is necessary to make the labeling of edges unique. That it is sufficient will be shown in the next theorem.

Theorem 1
The state vectors of an interaction model for $m > 2$ will have unique labels if the transformation function *tf* is an L-transformation function satisfying Transformation Property 2.

Proof:
Let the length $|v|$ of a state vector $v = (x_1, x_2, \ldots, x_m)$ be defined as follows: $|v| := \sum_{i=1}^{m} x_i$. We prove by mathematical induction that the following assertion is true for all $n \in N_0$: The labeling of edges (v_1, v_2), where $|v_1| = n$, and the labeling of vertices v, where $|v| = n$, will be unique, provided the transformation function used in step (4) satisfies Transformation Properties 1 and 2 and the Symmetry Property.

$n = 0$: Edges (v_1, v_2) with $|v_1| = 0$ start in v_0 and can be labeled only in step (2) of the definition of the interaction model. Thus they must have a unique label. The assertion is also true for vertices with $|v| = 0$, since $(0, 0, \ldots, 0)$ can be labeled only in step (1).

$n \to n + 1$: Suppose the assertion is true for $0 \leq i \leq n$. First let $e = (v_1, v_2)$ be an edge with $|v_1| = n + 1$. It either represents an original request, in which case the assertion follows immediately, or it is the result of some transformation. If there is only one such transformation, the uniqueness follows from the assumption and the unique definition of step (4). If there is more than one possible transformation, the assertion follows from Transformation Property 2 and the assumption. Second, let v be a vertex with $|v| = n+1$. If v is the endpoint of an edge for an original request, then the assertion follows immediately (no other edge may lead into this vertex since that would entail cyclic dependencies among requests, being impossible for a set of user requests). Otherwise there will be several edges leading into v. According to transformation step (4) edges leading into a vertex are always created in pairs. In this case the unique labeling of v follows from the assumption and the application of Transformation Property 1. Q.e.d.

THE ADOPTED-ALGORITHM

Next we present and discuss the adOPTed-algorithm, that implements the application-independent parts of our transformation-oriented approach to concurrency control (s. figs. 7 and 8). Large parts could be taken from the dOPT-algorithm [4]. The most important differences are: The adOPTed-algorithm introduces a multi-dimensional interaction model to store necessary information (2.5) and a double recursive function Translate Request (7.1). Both features combined guarantee the correctness of our adOPTed-algorithm.

Each user process manages the following local data structures: the application state s, a counter k for locally generated requests, a site's state vector v, its request queue Q, a request log L, and an n-dimensional interaction model G. The *state vector* v holds the number of executions for each user. The *request queue* Q is used to store generated (3.3) and incoming (4.4) requests that have to wait for execution. The request log L stores a copy of each original request (3.4, 4.5) so that a request can be easily accessed by its key consisting of user

```
(1.1) Main:
(1.2)    Initialize
(1.3)    while not aborted
(1.4)       if there is an input $o$
(1.5)          Generate Request ($o$)          ; locally
(1.6)          Execute Request
(1.7)       else
(1.8)          Receive Request                 ; from remote
(1.9)          Execute Request

(2.1) Initialize:
(2.2)    $s \leftarrow s_0$               ; the initial application state
(2.3)    $L \leftarrow$ empty set
(2.4)    $Q \leftarrow$ empty set
(2.5)    $G \leftarrow$ initial interaction model
(2.6)    $v \leftarrow (0, 0, \ldots, 0)$        ; initial state vector
(2.7)    $k \leftarrow 1$                        ; initial request id
(2.8)    $u \leftarrow$ identification of local participant

(3.1) Generate Request ($o$):
(3.2)    $r \leftarrow (u, k, v, o)$
(3.3)    $Q \leftarrow Q + r$
(3.4)    $L \leftarrow L + r$
(3.5)    $k \leftarrow k + 1$
(3.6)    broadcast $r$ to other participants

(4.1) Receive Request:
(4.2)    if there is a request from network
(4.3)       choose request $r$
(4.4)       $Q \leftarrow Q + r$
(4.5)       $L \leftarrow L + r$

(5.1) Execute Request:
(5.2)    if there is $r$ in $Q$ satisfying Executable? ($r$,$v$), then
(5.3)       choose executable request $r = (j, k_j, v_j, o_j)$
(5.4)          $Q \leftarrow Q - r$
(5.5)          $r'' \leftarrow$ Translate Request ($r$,$v$)
(5.6)          apply operation $o(r'')$ as user $j$ to state $s$
(5.7)          increment $j$-th component of $v$.

(6.1) Executable? ($r$,$v$):
(6.2)    logical value of $v(r) \leq v$
```

Figure 7: The adOPTed-algorithm for concurrency control as executed by each user process (part I)

```
(7.1)   Translate Request ($r$, $v$)
(7.2)     $(j, k_j, v_j, o_j) \leftarrow r$
(7.3)     if $v_j$=$v$ then
(7.4)       add request $r$ to model $G$
(7.5)       return $r$
(7.6)     else if $G$ contains $r'$ for $r$ translated to $v$
(7.7)       return $r'$
(7.8)     else
(7.9)       let $i$ be such that
(7.10)          Reachable? (Decr($v, i$)) and
(7.11)          $v_j[i]$ <= $v[i] - 1$
(7.12)        $v' \leftarrow$ Decr($v, i$)
(7.13)        $r_i \leftarrow$ Request ($i, v[i]$)
(7.14)        $r'_i \leftarrow$ Translate Request ($r_i$, $v'$)
(7.15)        $r' \leftarrow$ Translate Request ($r$, $v'$)
(7.16)        $(r'', r''_i) \leftarrow$ tf($r', r'_i$)
(7.17)        add requests $r'', r''_i$ to model $G$
(7.18)        return $r''$

(8.1)   Reachable? ($v$)
(8.2)     for every i in U:
(8.3)       $v[i]$=0
(8.4)       or $w($Request $(i, v[i])) \leq v$

(9.1)   Decr ($v, i$):
(9.2)     return copy of $v$ with $i$-th component decremented

(10.1)  Request ($i, j$)
(10.2)    return $j$-th request from user $i$ in $L$
```

Figure 8: The adOPTed-algorithm for concurrency control (part II)

id u and serial number k (10.1). The interaction model G is mainly used to store transformed requests that might be needed later. It would be possible to store application states like in the formal definition of the interaction model, but this is not necessary for the algorithm to work.

The main procedure of the adOPTed-algorithm initializes all data structures (2.2 - 2.8) and then runs into a loop (1.3 - 1.9) accepting operations from the local user, accepting requests from the network, and executing local and remote requests. Of special interest are the routines Execute Request and Translate Request. To execute a request, the request queue is scanned for executable requests: A request is executable if its state vector $v(r)$ is smaller than or equal to the current state vector v (6.1f.). If more than one request is executable, local requests are preferred in step (5.3). This guarantees that no local request is waiting for execution when the local user issues further operations. The chosen executable request is deleted from the queue (5.4) and then handed to the function Translate Request (7.1ff.), to translate it to the current state vector. Translate Request translates a request r from its state vector $v(r)$ to a state specified by the given state vector v. This is done recursively: If both states are equal (7.3) the request is returned. Otherwise (cf fig. 9) an immediate predecessor v' of v in the interaction model is determined (7.9 - 7.12), such that r may be translated into v'. Condition (7.10) makes sure that v' is contained in the interaction model. Condition (7.11) guarantees that v' is a successor of v_j, i.e., v' can be reached from v_j. If more than one such predecessor exist, it is safe to choose any one, since the proof of Theorem 1 shows that the path along which request r will be translated does not affect the result. If v' is a predecessor of v along the i-th coordinate axis, then let r_i be the $v[i]$-th request from user i (7.13). Request r_i is also translatable to v. The function Translate Request is called recursively to translate requests r and r_i into state vector v'. The requests obtained, r' and r'_i, are then fed into the transformation function *tf* (7.16). The transformed version r'' of request r' is finally returned by function Translate Request. The operation of the request r'' is then applied to the current state (5.6). The appropriate index of the current state vector is incremented (5.7).

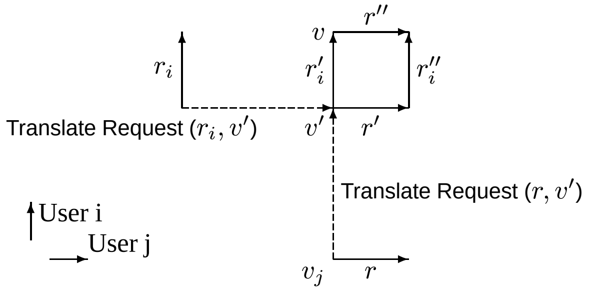


Figure 9: Recursive Definition of Translate Request (r, v)

The adOPTed-algorithm does not perform all possible transformations as suggested by step (4) of the formal definition of the interaction model. Rather only those requests needed in function Translate Request are actually computed. All original and computed requests are inserted into the interaction model (7.4, 7.17). Before translating a request it is first checked whether the result is already contained in the interaction model (7.6). In this way, every request has to be computed at most once.

When the request queue is growing too fast – in case received remote requests have to wait for execution – it may be appropriate to notify the local user and to lock the user interface so that for some time further input is impossible and the request queue can be shortened. If there is evidence that there will be no more requests with state vectors smaller than a certain value, then the corresponding information in the interaction model may be cleared. The information can be restored from the request log, since it is possible to rebuild the complete interaction model unambiguously from the original set M of user requests and from the initial application state s_0.

TRANSFORMATION RULES

While the algorithm is application-independent, the transformations have to be defined for every application. This happens by defining a set of partial functions that cover the whole domain. We call these partial functions transformation rules.

Transformation rules can be defined in several ways to satisfy the L-transformation properties. This may not be confused with the fact that the values of a transformation function *tf*

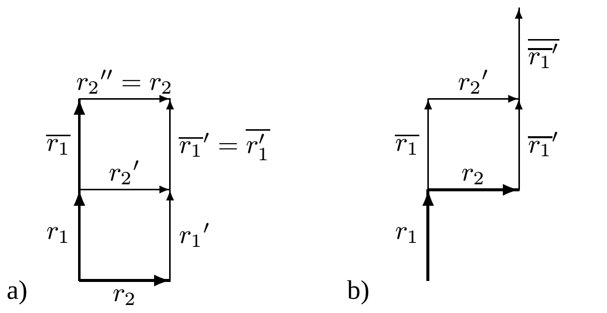


Figure 10: a) Undo Property, b) Duality of transformations

are unique, once the function is defined. Usually there is an infinite number of solutions. E.g., if the operations of the results of an L-transformation function are extended by applying some extra operation o_{extra}, the requests obtained will also satisfy the Transformation Properties. In practice, however, this will yield meaningless results. E.g., an extra operation of inserting an asterisk at the beginning of the text buffer will give consistent application states at all user sites, although not making any sense to the users.

Useful properties
The restrictions given for L-transformation functions are sufficient to guarantee consistent states at different sites, but from a user's point of view some transformations are more appropriate than others. We have identified especially two desirable properties. We will refer to a request whose operation is the identity operation I as a *noop request*.

Definition 4 (Noop Property)
Let r_I be a noop request. Then for all requests $r \in R$: $\mathit{tf}(r_I, r) = (r_I, r)$.

The Noop Property expresses the intuitively clear requirement, that users do not expect the effect of their operations being modified by some concurrently issued noop operation of another user. This property seems rather artificial, but it helps to identify inappropriate transformation rules, like the example given above that inserts an extra asterisk. Provided an undo command is supported, the following property is more important.

Definition 5 (Undo Property) (s. fig. 10a)
Let $\overline{r_1}$ be an undo request for r_1. For all requests $r_2 \in R$: If $\mathit{tf}(r_1, r_2) = (r_1', r_2')$ and $\mathit{tf}(\overline{r_1}, r_2') = (\overline{r_1}', r_2'')$, then $r_2 = r_2''$ and $\overline{r_1}' = \overline{r_1'}$

The motivation for the Undo Property is clear. When a user executes an operation and, changing his mind, immediately undoes the operation, other users' operations should not be affected. Another interesting property is implied by the Undo Property making it possible to compute transformations back and forth.

Theorem 2
If the Undo Property is satisfied, transformations are invertible:

$$\mathit{tf}_1(r_2, r_1) = r_2' \Longleftrightarrow \mathit{tf}_1(r_2', \overline{r_1}) = r_2 \qquad (5)$$

Proof:
Equivalence (5) follows easily from the Undo Property (cf fig. 10a).

Even L-transformation rules not satisfying the Undo or Noop Property may be valuable, e.g., in experimental scenarios of collaborative arts where the overall effect need not meet user expectations, but may in contrast be surprising.

Finding transformation rules
Probably there does not exist a constructive way of finding a set of transformation rules to define a transformation function for a given application which satisfies the necessary conditions for correct groupware, either for $m = 2$ or for $m > 2$ users. A heuristic method we used for finding rules, is the following: Mark the base edges on a transformation square with the concurrent operations to be transformed, then mark the successor states with the result from applying the operations to the original state. Decide what would be an appropriate total result after applying both operations and adjust the operations accordingly. This methods works fine, e.g., with simple text editing operations, the total result of concurrent operations usually being unambiguous. To find rules that also satisfy the Undo Property you have to make sure that the original request can be reconstructed from the information contained in any of the transformed requests.

It is an open question under what conditions it is possible to mix generic transformation rules with any special application-dependent rules without interfering with the validity of, e.g., the Transformation Property 2. It would be very helpful if such general conditions could be established since this would offer the opportunity to start with a set of generic transformation rules and to add specialized rules when appropriate.

Transformation rules for text editing
Next we show how to define a transformation rule for two insert operations of a text editor.

$$T_1((u_1, k_1, v_1, Ins[p_1, c_1]), (u_2, k_2, v_2, Ins[p_2, c_2])) \qquad (6)$$
$$:= \begin{cases} Ins[p_1, c_1] & \text{if } p_1 < p_2 \\ & \text{ or } (p_1 = p_2 \text{ and } u_1 < u_2) \\ Ins[p_1 + 1, c_1] & \text{otherwise} \end{cases}$$

If the insert operation to be transformed inserts its character at a lower position ($p_1 < p_2$) the operation will not be changed. If it inserts a character at a higher position ($p_1 > p_2$), then the position will be shifted to the right ($p_1 + 1$). If the positions are equal, the operation whose request has a larger user id is shifted to the right. This definition slightly differs from the one given by Ellis and Gibbs in [4]. They drop an insertion if position and character of both insertion are the same. For example, typing "`Hello Mark`" and typing "`Hello Matthew`" would result in something like "`Hello Marktthew`". We prefer our definition - keeping both inserted characters – for several reasons. First, the insertions are actually different, since they are performed by different users. Group text editors that distinguish between text inserted by different users make this obvious by presenting these text strings in different colors or in different fonts. Second, the Undo Property is not satisfied, since immediate undoing of typing "`Hello Mark`" in the example given above would yield "`tthew`" and not "`Hello Matthew`". Finally, from a user's point of view, it is better, by default, to leave information on the screen, instead of hiding it. The rare case when users concurrently insert the same character, e.g., to fix the same typographical error, is handled by the awareness provided: After noticing the redundant character on the screen it can be deleted. In fact, if both users would react concurrently, oscillation could result, but such a case has never been observed in practice.

Next we define the transformation of two delete operations.

$$T_1((u_1, k_1, v_1, Del[p_1, c_1]), (u_2, k_2, v_2, Del[p_2, c_2])) \\ := \begin{cases} Del[p_1, c_1] & \text{if } p_1 < p_2 \\ Del[p_1 - 1, c_1] & \text{if } p_2 < p_1 \end{cases} \tag{7}$$

This means that a delete operation remains unmodified if another deletion has occurred before to its right. The delete operation has to be shifted one position to the left if another deletion has happened before to its left.

For identical positions the following definition, given also by Ellis and Gibbs in [4], seems obvious:

$$T_1((u_1, k_1, v_1, Del[p_1, c_1]), (u_2, k_2, v_2, Del[p_2, c_2])) \\ := \quad I \quad \text{if } p_1 = p_2 \tag{8}$$

This rule works fine from the users' point of view as long as nobody undoes one of the delete operations. In this case either the Noop Property or the Undo Property will fail to hold. Therefore we defined the following alternative rule for the transformation of identical delete operations:

$$T_1(r_1, r_2) \quad := \quad \mathit{Shadow}(r_1, \{r_2\}) \\ \text{if } o(r_1) = o(r_2) = Del[p, c]. \tag{9}$$

Shadow is a meta operation. *Shadow*$(r_1, \{r_2\})$ means that a request, here r_1, cannot be executed as long as the requests in the set of conflicting requests, here $\{r_2\}$, have not been undone. This may be referred to as r_1 being shadowed by r_2. The transformation of a request containing a shadow operation with respect to a request undoing one of the conflicting requests, will delete the conflicting request from the set. If the set of conflicting request becomes empty, the shadowed request can be executed again. The transformation with respect to any other request r yields another shadow operation, where the set of conflicting operations is extended by r. In this way, it is possible to satisfy the Undo Property without offending the Noop Property.

Transformations for other combinations of characterwise insert and delete functions are straightforward and have already been proposed by Ellis and Gibbs [4]. More complex transformation rules for composed operations can be found in [10].

Duality of transformations for concurrency control and for undo
Prakash and Knister [9] provide a transformation-based mechanism for undo in groupware and selective undo in single-user applications. Transformations are applied to swap neighbored operations of a linear history in order to move an operation that is to be undone to the end of the history. Their transformations bear a strong resemblance to the transformations used by our method.

Let r_1 and r_2 represent a sequence of requests to be swapped (cf fig. 10b). The problem of transposing is to find a pair (r_2', r_1') of requests such that the two sequences $r_1 r_2$ and $r_2' r_1'$ are equivalent. $r_1 r_2 \equiv r_2' r_1'$.

Theorem 3
A solution to the transposing problem is formed by $r_2' = \mathit{tf}_1(r_2, \overline{r_1})$ and $r_1' = \overline{\overline{r_1}'}$, where $\overline{r_1}' := \mathit{tf}_1(\overline{r_1}, r_2)$.

Proof:

$$\begin{aligned} r_1 r_2 &\equiv r_1 r_2 \overline{r_1}' \overline{\overline{r_1}'} && (10) \\ &\equiv r_1 \overline{r_1} r_2' \overline{\overline{r_1}'} && (11) \\ &\equiv r_2' \overline{\overline{r_1}'} && (12) \end{aligned}$$

Equivalences (10) and (12) hold because of the neutrality of do-undo-pairs, equivalence (11) holds because of the transformation property 1 being applied to the transformation step (cf fig. 10b). Q.e.d.

In this way, the Transpose function given in [9] can be deduced from our transformation function. E.g., let r_1 and r_2 be requests to be swapped, where $o(r_1) = Ins[4, x]$ and $o(r_2) = Ins[1, y]$. Then $o(\overline{r_1})$ will be $Del[4, x]$. It follows, that $o(r_2') = Ins[1, y]$ and $o(\overline{r_1}') = Del[5, x]$. Finally, $o(r_1') = o(\overline{\overline{r_1}'}) = Ins[5, x]$. As expected, the pair $(Ins[1, y], Ins[5, x])$ is identical to the result computed by the Transpose function defined in [9]. In contrast to our transformation function, however, the Transpose function in [9] is only partially defined. In critical cases where insert or delete operations happen to overlap, the value of the function remains undefined and transposing will not be possible.

Not unexpectedly, several more similarities between the transformations used in both approaches can be found. E.g., our Undo property corresponds to the Inverse Property 2 in [9], our Identity Property to the Transpose Property 4, and our Transformation Property 2 to the Transpose Property 5.

GROUP UNDO

A problem related to concurrency control is group undo. Making some operation or several operations undone is an important user intention that any editor should support [1]. Two important types of group undo can usually be distinguished: global group undo and local group undo [1]. Global group undo applies the undo operation to the last operation executed, no matter who was the initiator. Local group undo only affects someone's own operations. Global group undo is appropriate for a sequential track of interaction as, e.g., enforced by some floor control mechanism. For group editors, where each user edits concurrently having his or her own insertion point, local group undo has to be provided. With global group undo, the operation to be executed only depends on the operation to be undone, e.g., being its inverse operation. In the case of local undo, it additionally depends on requests having been executed in the meantime.

Most known undo schemes rely on the fact, that the application has to be in the state immediately following the execution of

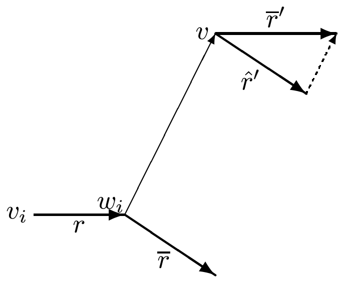


Figure 11: Definition of local group undo by transformation

the operation to be undone. Otherwise it will not be possible to apply, e.g., the inverse operation to the application state in order to make the effect undone.

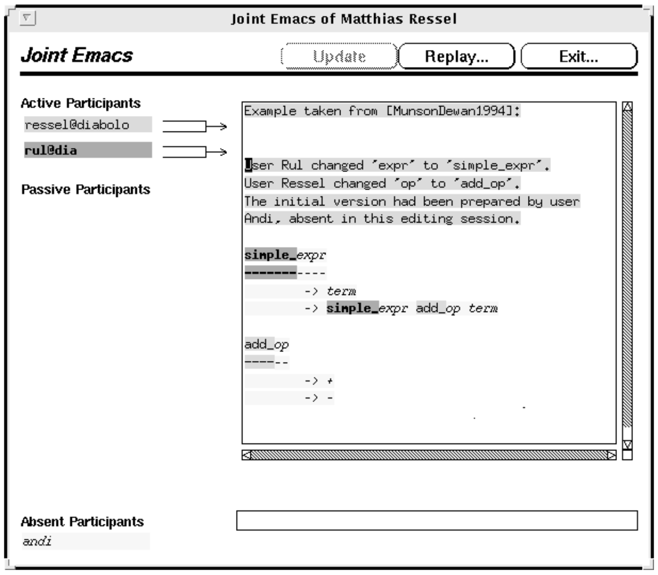
Example 3 An operation like *Ins*[2,*a*] can be immediately undone by the operation *Del*[2,*a*]. If, however, another operation like *Ins*[2,*H*] has already been applied, this would result in deleting the wrong character, namely the letter *H* and not *a* as desired. In order to delete the right character, *Del*[2,*a*] had to be transformed to *Del*[3,*a*], taking into account the later insert operation. □

As Example 3 suggests, the problem of single-step local group undo may be solved by applying transformations to the inverse operations. Assume, you want to undo a request $r = (u_i, k_i, v_i, o_i)$ from the history, v being the current state vector (s. fig. 11). Introduce an additional virtual user $\hat{u}$, that generates an undo request $\overline{r}$ that reverts the effect of r in the state described by state vector $w_i = w(r)$. The request $\overline{r}$ is then translated to the current state vector v of the application: Translate Request$(\overline{r}, v) = \hat{r}'$ (cf fig. 8). Now generate a request $\overline{r}'$ of your own in the current state v, whose operation equals the operation $o(\hat{r}')$ of the translated undo request. The extra user is needed, since the path from w_i to v may contain requests from the user that issued the undo command. All the transformations on this path are defined, however, only for requests from different users.

Undoing the most recent request will yield the expected result: the application state prior to the execution of this request. Note that our definition of group undo depends on the same transformation function *tf* used for the purpose of concurrency control.

USAGE EXPERIENCE

For testing the adOPTed-algorithm, we have built the prototypical group editor JOINT EMACS, an EMACS-style text editor. JOINT EMACS has been implemented completely in CLOS (Common Lisp Object System). Its user interface has been built with XIT [7]. The functionality of JOINT EMACS includes, among other things, stringwise insert and delete operations and local group undo. Each author's text is presented with a selectable background color or – optionally - in a certain font. JOINT EMACS provides a "Replay"-function allowing to review the interaction history. The current state of interaction can be saved and later reloaded. The grid structure can be visualized (limited to three dimensions) making it possible to analyze the interaction pattern.



```
Example taken from [MunsonDewan1994]:


User Rul changed 'expr' to 'simple_expr'.
User Ressel changed 'op' to 'add_op'.
The initial version had been prepared by user
Andi, absent in this editing session.

simple_expr
-----------
        -> term
        -> simple_expr add_op term

add_op
------
        -> +
        -> -
```

Figure 12: User interface of the group editor JOINT EMACS

JOINT EMACS supports a high degree of fine-grained mutual awareness, i.e., users are able to see and understand the activities of other users in order to gain the right context for their own work [3]. Conflicts that cannot be solved easily correlate with situations where the users are manipulating closely related parts. With a high probability, those conflicts can be avoided by seeing each other's operations.

JOINT EMACS is experimental. Its main intention has been to show that our approach for concurrency control and group undo is feasible. Nevertheless, the editor is in daily use in our group as an innovative communication tool. Compared to other text-based communication tools like *mail* or *talk*, JOINT EMACS offers the advantage that the communicated information can be modified by any participant. E.g., the first user may type some text, a second user may immediately fix any typos, and a third user may change the line breaking, all activities being performed concurrently. JOINT EMACS is also used to write group e-mails, i.e., e-mails written by several senders cooperatively.

JOINT EMACS is presently in use in various sessions with up to 6 users in real-world situations. A maximum of 9 users (the upper bound implemented) issuing operations concurrently have been tested successfully. Typical sessions, however, include 2 to 3 participants.

Unforeseen problems

In earlier versions of our editor we encountered a strange behavior. With cursors at the same position, when each user typed some text string – the individual characterwise insertions being not necessarily concurrent – the strings got intermixed. Actually, we found out the reason for that was not a mistake in the adOPTed-algorithm – the individual text copies showed consistent, although unexpected results – but a flaw in the definition of the *apply* function. Originally, for every insert operation any cursor located at the insert position was shifted to the right. We could fix the problem by shifting a cursor to the right only if its owner has a lower id than the user performing the insert operation.

When reverting the effect of several delete operations of neighbored characters, we observed another strange effect, described also in [9]. E.g., let the initial string be “ab”. User 1 deletes the character “a” and user 2 deletes the character “b”. Then both users undo their delete operations. The resulting string may be “ba”, if a certain sequence of delete and undo operations is chosen. This result is totally unexpected from a user’s point of view. The reason for this anomaly is, that the relative positions of the delete operations are not explicitly remembered. In fact, we found that the ordering between such delete operations may be determined – when needed – from the information implicitly contained in the interaction model.

Performance

As long as the interaction is sequential, our algorithm produces little overhead. The computation of transformations is only needed when concurrency increases. Unfortunately, this coincides with periods of high user activity. In this case, the right balance between awareness and responsiveness has to be found: The longer you wait for handling remote requests, the more concurrent requests will accumulate and the longer it will take to compute the transformations needed. We therefore included several adjustable parameters, that, e.g., allow to control the priority given to local user requests in relation to the handling of incoming requests from other users.

CONCLUSION

In our research we developed an integrating approach to concurrency control and group undo that is based on the dOPT-algorithm by Ellis and Gibbs. We proved the correctness of our adOPTed-algorithm by finding necessary and sufficient preconditions to be satisfied for producing identical application states in a replicated groupware architecture. The Noop and Undo Properties were identified as conditions to be satisfied if the algorithm is to produce acceptable results from a user’s point of view. We showed how the problem of local group undo can be reduced to a concurrency problem by regarding undo operations as concurrently generated operations. We further showed how the transformation-based approach by Knister and Prakash used for group undo relates to our approach. Many of the results mentioned in the theoretical part of this work would not have been attained without the experiences gained by implementing the group editor JOINT EMACS and putting it to use.

Our approach for concurrency control and undo in group editors is to be seen as an alternative and a complement to other known techniques rather than as a replacement. In fact, the detected strong relationship between transformations used for concurrency control and transformations used for group undo seems promising to put further research into the question, how results from different approaches can be combined. We hope, that our theoretically-based work will inspire other researchers to look for innovative ways of computer-supported interaction.

ACKNOWLEDGEMENT

We would like to thank the reviewers, whose valuable critique and comments helped to improve this paper. Without the support of our former colleagues in the research group DRUID, Jürgen Herczeg and Hubertus Hohl, this work could not have been realized.

REFERENCES

1. Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

2. Ronald M. Baecker, editor. *Readings in Groupware and Computer-supported Cooperative Work: Assisting Human-Human Collaboration.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

3. Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In Jon Turner and Robert Kraut, editors, *Proceedings of the CSCW ’92*, pages 107–114. ACM Press, 1992.

4. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD ’89 Conference on the Management of Data*, pages 399–407, Seattle, Washington, 1989. ACM, New York.

5. C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

6. Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In Richard Furuta and Christine Neuwirth, editors, *Proceedings of the CSCW ’94*, pages 207–217. ACM Press, 1994.

7. Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel. Progress in building user interface toolkits: The world according to XIT. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 181–190, November 1992.

8. Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In Richard Furuta and Christine Neuwirth, editors, *Proceedings of the CSCW ’94*, pages 231–242. ACM Press, 1994.

9. Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, December 1994.

10. Matthias Ressel. Kooperative Interaktionsunterstützung in Groupware. In Heinz-Dieter Böcker, editor, *Software-Ergonomie ’95*. German Chapter of the ACM, Teubner, Stuttgart, 1995.