Designing Checkers for Programs that Run in Parallel

Ronitt Rubinfeld [1]

TR-90-040

August, 1990

Abstract

We extend the theory of program result checking to parallel programs, and find general techniques for designing such result checkers. We find result checkers for many basic problems in parallel computation. We show that there are P-complete problems (evaluating straight-line programs, linear programming) that have very fast (even constant depth) parallel result checkers. Sorting, multiplication, parity, majority and the all pairs shortest path problem all have constant depth result checkers. In addition, the sequential versions of the parallel result checkers given for integer sorting and the all pairs shortest path problems are the first deterministic sequential result checkers for those problems.

[1] Computer Science Division, University of California, Berkeley, California 94720 *and* International Computer Science Institute, Berkeley, California 94704. Supported in part by an IBM Graduate Fellowship and NSF Grant No. CCR 88-13632.

1 Introduction

Verifying a program to see if it is correct is a problem that every programmer has encountered. Even the seemingly simplest of programs can be full of hidden bugs, and in the age of massive software projects, this problem seems to be increasingly important. A general theory of *result checking* algorithms was recently suggested in [5, Blum]. The philosophy behind result checking the algorithm is to rigorously emulate what we were taught to do in school - to check our work. This approach recognizes that proving programs correct is very difficult to do, and with this in mind, aims at the easier task of checking that a program is correct on any given input. This easier problem is not only feasible, but often yields result checkers that are much simpler than the original program and therefore less likely to contain bugs.

Many result checkers in the sequential model of computation have been found for various types of problems. However, a user is unlikely to be willing to use a sequential result checker to verify the correctness of a result produced by a fast parallel algorithm. In this paper, we extend the program result checking framework to the setting of checking parallel programs and find general techniques for designing such result checkers. For example, we find techniques for result checking programs which compute certain types of functions that have the property that they can be computed be computed "indirectly", by calling the program on another, related input. We also present a techniques based on quickly reconstructing the computation of a simple sequential algorithm, on duality and on constant depth reducibility among problems. We find result checkers for many basic problems in parallel computation. Many of these result checkers are rather straightforward, such as the result checkers for parallel prefix and straight-line programming. Others involve a more intricate design and more complex proofs, such as the result checkers for majority, unary to binary conversion, parity and convex hull. In addition, the sequential versions of the parallel result checkers given for integer sorting and the all pairs shortest path problems are the first deterministic sequential result checkers for those problems. All of the examples in this paper are written for the arbitrary and priority CRCW PRAM models.

The difference in the complexity of solving a problem as compared to the complexity of result checking a problem is often very dramatic. For example, we show that there are P-complete problems (evaluating straight-line programs, linear programming) that have very fast (even constant depth) parallel result checkers. Integer GCD is not known to be in RNC, yet a logarithmic depth parallel result checker exists for it [1, Adleman Huang Kompella]. Maximum Matching is not known to be in NC (though it is in RNC), and it has a deterministic NC result checker. Multiplication, parity and majority all have lower bounds of $\Omega(\log n/\log\log n)$ depth, yet all have (completely different) constant depth result checkers.

2 The Parallel Program Result Checking Model

In this section, we describe the extension of the program result checking model proposed in [5, Blum] to result checking for parallel programs.

DEFINITION 2.1 (probabilistic program result checker) *A probabilistic program result checker for f is a probabilistic oracle program R_f with oracle P, which is used to verify,* for any program P *that supposedly evaluates f, that P outputs the correct answer on a given input in the following sense. On a given input x and confidence parameter α, R_f^P has the following properties:*

1. *If $P(x) \neq f(x)$ then R_f^P outputs "FAIL" (with probability $\geq 1-\alpha$).* [1]

2. *If P is a correct program for every input then R_f^P outputs "PASS" (with probability $\geq 1-\alpha$).*

The parallel result checker is allowed to call the program as many times as desired at each parallel step.

Note that if $P(x) = f(x)$ but P is faulty on other inputs then R_f^P may output either "FAIL" or "PASS".

R_f is only allowed to access P as a black-box oracle.

We refer to the parallel running time of the checker, not including the running time of the calls to the program, as the *checking time.* We refer to the parallel running time of the checker including the running time of the calls to the program as the *total time.* We make the same conventions with respect to the number of processors. When describing the total running time of a result checker, we will use $D(n)$ to refer to the total time of the program running on an input of size n, and $N(n)$ to refer to the total number of processors used by the program when running on an input of size n. In both cases, we ignore the dependence on the confidence parameter α. In all of the examples, the result checker first calls the program on the input which is being checked.

The problem remains of determining the correctness of the result checker. In the sequential setting, Blum [5, Blum] suggests that instead, the result checker should be forced to be quantifiably "different" than any program for f by limiting the checking time to be less than that of the fastest correct program known for computing the function. We consider analagous notions of "different" for parallel result checkers. Suppose that any parallel program which computes the function f requires at least d depth. Suppose the number of processors required when computing f in depth d is p. We say that R_f is *quantifiably different* if the checking time is $o(d)$ depth *or* if the checking time is $O(d)$ and (simultaneously) the checking number of processors is $o(p)$ on the same model of parallel computation. The result checkers described in this work are all quantifiably different. So far, most of the result checkers found which are quantifiably different seem to be simpler than any program for the function as well.

In the most straightforward applications of checking, whenever the program is executed the result checker is also executed. Thus, it is critical that the overhead cost of running the result checker does not neutralize the benefit from knowing that the output is correct (or the knowledge that the program is faulty). All of the result checkers in this paper call the program at most once on any computation path, so the total depth is *big oh* of the depth of the program being checked. Many of the result checkers have the property that the total number of processors used is *big oh* of the number of processors used by the program (e.g. sorting, parity, convex hull)

3 Computability by Random Inputs

[9, Blum Luby Rubinfeld] shows that one can design result checkers for many functions that have the property of random self-reducibility - that the function can be computed by computing the function on one or more "random" instances. We show that often the property that a function can

[1] The probabilities are with respect to a source of truly random independent bits available to the result checker, and *not* with respect to any assumptions about the input distribution.

can be computed by computing the function on one or more "almost-random" instances can also be utilized in designing a result checker.

We concentrate on symmetric functions - functions on n bits whose output depends only on the *number* of 1's in the input. Thus, the value of the function can be computed indirectly by computing the function on a "shuffle" (random permutation) of the input bits. However, the techniques in this chapter are applicable to other functions as well. For example, the running time of the sequential checker for the matrix rank function given in [9, Blum Luby Rubinfeld] can be dramatically improved using the technique in Section 3.2.

The techniques in this section are based on testing the program on random inputs for which the answer is known, and then verifying that the program's answer on the particular input being checked is consistent with the program's answer on most other inputs.

3.1 Any Symmetric Function on n Bits

We give a result checker for any symmetric function:

Input: A list of input bits $\hat{a} = a_1, a_2, ..., a_n$, a table of values $t_0, ..., t_n$.

Output: $b = t_i$ where $i = \Sigma_{1 \leq j \leq n} a_j$.

The majority, exactly i and parity functions are all examples of symmetric functions. As mentioned before, [6, Beame Hastad] show that $\Omega(\log n / \log\log n)$ depth is required to compute these functions. For these and other examples, no table is needed as input because the table can be computed in constant depth by the result checker.

Let P be the program that supposedly computes the symmetric function. P is checked by partitioning the inputs of size n into $n+1$ equivalence classes, where all inputs in a particular equivalence class contain the same number of 1's. Intuitively, the result checker verifies that P is correct on more than 1/2 of the members of each equivalence class, and that the answer of P on the input in question is consistent with more than 1/2 of the members within its own equivalence class. Therefore, even if the result checker cannot determine which equivalence class the input is in, it can verify that the answer of P on the input is correct. In the result checking algorithm, several random permutations of the input bits are made; [4, Ajtai] gives a way of doing this in constant depth.

Result Checking Algorithm:

$k \leftarrow \log(1/\alpha)$
$b \leftarrow P(\hat{a})$
In parallel, compute k random permutations $\pi_1, \ldots, \pi_k$ of $\{1, \ldots, n\}$
Phase 1: (Consistency with our input)
    In parallel, for $i = 1, \ldots k$
        If $P(\pi_i(\hat{a})) \neq b$ then output "FAIL" and halt
Phase 2: (Testing Correctness of most inputs)
    In parallel, for $j = 0, \ldots, n$
        In parallel, for $i = 1, \ldots, k$
            If $P(\pi_i(1^j 0^{n-j})) \neq t_j$ then output "FAIL" and halt.
Output "PASS".

Proof of Correctness: Clearly if P is correct on all inputs, the result checker will output "PASS". Assume that P is incorrect on input $\hat{a}$, we show that the result checker outputs "FAIL" with probability $\geq 1-\alpha$. Let j be the number of ones in $\hat{a}$. Suppose that P is correct (and consequently differs from the output on $\hat{a}$) on $\geq 1/2$ the inputs with j ones. Then with probability $\geq 1-\alpha$, an input that is inconsistent with $\hat{a}$ is found in Phase 1. Suppose that the program errs on $\geq 1/2$ the inputs of size n with j ones. Then with probability $\geq 1-\alpha$, the jth group of processors in Phase 2 finds that the program is buggy. Notice that by this argument the same k permutations can be used in Phase 1, and by every group of processors in Phase 2.

Running Time: The checking time is $O(1)$ and checking number of processors is $O(C(n)+n^2)$ where $C(n)$ is the number of processors necessary to compute a random permutation in $O(1)$ parallel steps. The total time is $O(1+D(n))$ and the total number of processors is $O(C(n)+nN(n))$. ■

3.2 Special Symmetric Functions

A factor of n in the number of processors can be saved when the symmetric function f is of a special type: Let $t_0, \ldots, t_n$ be the input table for problems of size n and $t'_0, \ldots, t'_{2n}$ be the input table for problems of size $2n$. We say that f is of this special type if there is an easily computable function $g(b,j)$ such that if $t'_i = b$ then $t_{i-j} = g(b,j)$.

Examples of such functions are parity, where $g(b,j) = b \oplus (j \bmod 2)$, and the unary to binary conversion function, where $g(b,j) = b - j$.

Result Checking Algorithm:

$k \leftarrow \log(1/\alpha)$
$b \leftarrow P(\hat{a})$
In parallel, compute k random permutations $\pi_1, \ldots, \pi_k$ of $\{1, \ldots, 2n\}$
Phase 1: (Consistency with our input)
  In parallel, for $i = 1, \ldots, k$:
    Uniformly and randomly pick $j \in [0, \ldots, n]$
    Let s be the string $a_1, \ldots, a_n, 1^j 0^{n-j}$
    $s' \leftarrow \pi_i(s)$
    If $b \neq g(P(s'), j)$, output "FAIL" and halt.
Phase 2: (Testing Correctness of most inputs)
  In parallel, for $i = 1, \ldots, k/\log(4/3)$:
    Uniformly and randomly pick $j \in [0, \ldots, 2n]$
    Create the string $s = 1^j 0^{2n-j}$
    $s' \leftarrow \pi_i(s)$
    If $P(s') \neq t'_j$, output "FAIL" and halt.
Output "PASS".

Proof of Correctness (sketch): If P is correct on all inputs, then clearly the result checker outputs "PASS". Let l be the number of 1's in $\hat{a}$ and suppose $b = P(\hat{a}) \neq t_l$. Let $\mathcal{D}$ be the probability distribution defined by (j,r), where j is chosen uniformly at random in $[0, \ldots, 2n]$ and r is a random string of length $2n$ with j 1's. Let $\mathcal{D}'$ be the probability distribution defined by (j,r), where j is chosen uniformly at random in $[0, \ldots, n]$ and r is a random string of length $2n$ with $j+l$ 1's. Let p be the probability that $P(r) \neq t'_j$ when (j,r) is chosen according to $\mathcal{D}$. If $p \geq 1/4$, then each

execution of the loop in Phase 2 outputs "FAIL" and halts with probability at least $1/4$. Thus, the output is "FAIL" with probability at least $1-\alpha$. Now consider the case where $p \leq 1/4$. Let s' be a string of length $2n$ with $l+j$ 1's. By the properties of g, if $P(s') = t'_{l+j}$ then $g(P(s'), j) = t_l$. Furthermore, it can be shown that if $p \leq 1/4$ then $Pr[P(s') = t'_{l+j}] \geq 1/2$ when (j, s') is chosen according to $\mathcal{D}'$. These two facts imply that $Pr[g(P(s'), j) = t_l] \geq 1/2$ in each execution of the loop in Phase 1, and thus, since $b \neq t_l$, $Pr[g(P(s'), j) \neq b] \geq 1/2$ in each execution of the loop in Phase 1, in which case the output is "FAIL". Thus, the output is "FAIL" with probability at least $1-\alpha$.

Running Time: The checking time is $O(1)$ parallel steps and the checking number of processors is $O(C(n)+n)$. The total time is $O(1+D(n))$ and the total number of processors is $O(C(n)+N(n))$. ■

3.3 Randomly Self-Reducible, Linear and Smaller Self-Reducible Problems

If the program computes a function which is randomly self-reducible and either has the linearity property or is self-reducible to smaller inputs (see [9, Blum Luby Rubinfeld] for a definition), the general techniques described in [9, Blum Luby Rubinfeld] can be parallelized. This gives constant depth efficient result checkers for checking numerical problems such as integer multiplication, integer division, mod, modular multiplication, modular exponentiation, polynomial multiplication, squaring and matrix multiplication. The technique can also be used to give a result checker for parity that uses $O(1)$ checking time and $O(n)$ checking number of processors.

4 Consistency

Many problems have linear time sequential algorithms that are extremely simple and even possible to prove correct with formal verification methods. However, it is often the case that any parallel algorithm P for the same problem is necessarily radically different and more complex. Intuitively, a typical parallel result checker developed in this section calls P to reconstruct the computation steps of the extremely simple sequential algorithm, and then verifies the consistency between adjacent steps of the computation. This can be done very quickly, and independently of the algorithm actually used by P. This simple idea gives deterministic parallel result checkers for a surprising number problems. Some problems have result checkers that do not even need an additional call to P.

The *prefix sums* problem takes as input a list of elements $a_1, a_2, ..., a_n$, and outputs $(b_1, b_2, ..., b_n)$, where $b_i = a_1 \circ a_2 \circ a_3 \circ ... \circ a_i$ for an associative binary operator $\circ$. We assume that $\circ$ can be computed correctly by one processor in constant time. In order to verify that P computes the correct result, in parallel for $1 \leq i \leq n-1$, processor i checks that $b_i \circ a_{i+1} = b_{i+1}$. The checking time is $O(1)$ and the checking number of processors is n. The total depth is $O(D(n))$ with $O(N(n)+n)$ total processors. Note that the result checker makes no additional calls to P. A small variant of this result checker works for the *list ranking problem* as well in the same time and with the same number of processors.

The *sum* problem is similar to prefix sums, except that only b_n is output, and thus it is harder to check. The intermediate prefix answers $b_1, \ldots, b_{n-1}$ can be reconstructed as follows: In parallel

for $1 \leq i \leq n$, group i of processors calls the program to compute $b_i = P(a_1, a_2, ..., a_i)$. Then processor i verifies that $b_i \circ a_{i+1} = b_{i+1}$. The checking time is $O(1)$ and the checking number of processors is $O(n^2)$. The total depth is $O(1) + D(n)$ with $O(n \times N(n))$ total processors.

The ideas in this result checker can be used for various problems, including *parity, addition of n numbers*, and can be modified to work for *straight-line programming* (when the variables are each set only once) and the expression evaluation problem. When the variables can be set more than once, the checking time of straight-line programming is $O(\log n)$ using sorting.

A result checker for *integer multiplication* can also be constructed using this idea, where the input is 2 n-bit numbers a, b and the output is $a \times b$. The result checker algorithm is as follows: In parallel for $1 \leq i \leq n$, the ith group of n processors asks the program to multiply a by the last i bits of b to get r_i. If the ith least significant bit of b is a 0 then the result checker verifies that $r_i = r_{i-1}$, otherwise the result checker verifies that $r_i - r_{i-1} = a \times 2^i$. The checking time is $O(1)$, and the checking number of processors is $n \times A(n)$, where $A(n)$ is the number of processors required to do addition in constant depth. In [10, Chandra Fortune Lipton], it is shown that $A(n)$ is $O(ng^{-1}(n))$ for any strictly increasing primitive recursive function g. The total time is $O(D(n))$ with $O(n \times A(n) + n \times N(n))$ total processors.

Because of the following known results, all the checkers presented in this section are quantifiably different. The best known algorithm for prefix sums uses $O(n/\log n)$ processors and $O(\log n)$ depth ([15, Ladner Fischer],[11, Fich]). Multiplication can be done in $O(\log n/\log\log n)$ parallel depth, with $O(n^\epsilon)$ processors (for any ϵ). Any algorithm using only a polynomial number of processors for prefix sums, sum, parity and integer multiplication provably requires $\Omega(\log n/\log\log n)$ depth ([6, Beame Hastad]). Straight-line programming is P-complete.

4.1 Problems that can be solved using Dynamic Programming

Richard Karp has pointed out that the basic technique described in this section can be used to check any problem that can be solved sequentially using dynamic programming, regardless of the algorithm used by the program. By dynamic programming, we mean that there is some polynomial algorithm that computes the function on the whole set of inputs by evaluating the *same* function on smaller sets of inputs and somehow combining the results. This usually involves writing out the function on smaller sets of inputs in the form of a table. The idea behind the result checker is to call the program on *each* subproblem in parallel to fill in the table, and then verify that the entries of the table are consistent with each other. In most cases, this combination of results involves finding the minimum or maximum of a set of numbers. Since the minimum and maximum function can be computed in constant time, the checking time is constant.

The following is an example:

Longest Common Subsequence

Input: Two strings $x = x_1x_2x_3...x_n$ and $y = y_1y_2y_3...y_n$.

Output: The length of the longest common subsequence of x and y.

Let $lcs(l, k)$ denote the length of the longest common subsequence of $x_l x_{l+1}...x_n$ and $y_k y_{k+1}...y_n$. Then the sequential dynamic programming algorithm used to solve the longest common subsequence problem builds up the table as follows: if $x_l = y_k$ then $lcs(l, k) = 1 + lcs(l + 1, k + 1)$, otherwise $lcs(l, k) = max\{lcs(l, k + 1), lcs(l + 1, k)\}$.

The algorithm for the result checker is:

Do for all $1 \leq l \leq n$
  Do for all $1 \leq k \leq n$
    $s_{lk} \leftarrow P(x_l ... x_n, y_k ... y_n)$
    Verify consistency:
      If $x_l = y_k$ verify that $s_{lk} = 1 + s_{l+1,k+1}$
      else verify that $s_{lk} = max\{s_{l,k+1}, s_{l+1,k}\}$
If any of these verifications fail then output "FAIL" else output "PASS"

The checking time is $O(1)$ and the checking number of processors is $O(n^3)$. The total running time is $O(1 + D(n))$ with $O(n^3 + n^2 \times N(n))$ total processors.

4.2 All Pairs Shortest Path

Input: $n \times n$ adjacency matrix A, with a nonegative weight for each edge.

Output: Matrix $Dist$ specifying length of shortest path between every pair of nodes.

Result Checking Algorithm:

Do in parallel for each entry $D(u, v)$
  (1) check that $Dist(u, v) \leq A(u, v)$
  (2) check that for all w that are neighbors of v, $Dist(u, w) + A(w, v) \geq Dist(u, v)$
  (3) check that $\exists w$ neigbor of v such that $Dist(u, w) + A(w, v) = Dist(u, v)$
If any of these checks fail then output "FAIL" else output "PASS"

Proof of Correctness: It is clear that if the program is correct, the result checker will output "PASS". Suppose that the result checker outputs "PASS". Let $d(u, v)$ denote the correct shortest distance between u and v. We want to show that for all pairs (u, v), $Dist(u, v) = d(u, v)$.

Suppose for contradiction that there are nodes u, v such that $Dist(u, v) < d(u, v)$. Let u, v be nodes with $Dist(u, v) < d(u, v)$ such that v has the smallest possible index. Then because of step 3, there must be a w such that $Dist(u, w) < d(u, w)$ and w has smaller index that v. Therefore, for all u, v we have that $Dist(u, v) \geq d(u, v)$.

We will show by induction on the number of intermediate nodes along a shortest path between a pair of nodes that $Dist(u, v) = d(u, v)$.

Basis: The number of intermediate nodes visited when taking the shortest path from u to v is 0 (edge uv is the shortest path). Step 1 guarantees that $Dist(u, v) \leq A(u, v) = d(u, v)$.

Induction Step: Suppose that $Dist(u, v) = d(u, v)$ for all pairs (u, v) where there is a shortest path from u to v that visits i intermediate nodes. Consider pair (u, v) where there is a shortest path from u to v with $i+1$ intermediate nodes, and let w be the last node along this path. Then, step 2 verifies that $Dist(u, w) + A(w, v) \geq Dist(u, v)$. We know $d(u, w) + A(w, v) = d(u, v)$. By the induction hypothesis, since there is a shortest path between u and w of length i, $Dist(u, w) = d(u, w)$. Thus $Dist(u, v) \leq A(w, v) + d(u, w) = d(u, v)$ and so $Dist(u, v) = d(u, v)$.

Running Time: The checking time is $O(1)$ and the checking number of processors is $O(n^3)$. The total time is $D(n) + O(1)$ with $O(n^3) + N(n)$ total processors. Note that the result checker makes no extra calls. ■

5 Sorting and Computational Geometry

5.1 Sorting

Consider the problem of sorting integers with the following specifications:

Input: A set of integers $X = \{x_1, x_2, ..., x_n\}$ (not necessarily distinct).

Output: The elements of X in sorted order: i.e. a list $y_1 \leq y_2 \leq ... \leq y_n$ such that $Y = \{y_1, ..., y_n\}$ is equal to X.

The best known algorithms for sorting require $O(n \log n)$ time. The result checker must verify that the output is in sorted order, and that the set of elements in the input list is the same as the set of elements in the output list. The first task is quite easy, but the second task is nontrivial, and on the algebraic decision tree model, is as difficult a task as sorting. In [5, Blum], [7, Blum Kannan] there are randomized algorithms for verifying that $X = Y$ which use hashing and run in $O(n)$ time. We present a deterministic algorithm which checks sorting in $O(1)$ parallel time and $O(n)$ processors. This algorithm is the first deterministic sequential result checker for sorting that runs in $O(n)$ time.

Checker Algorithm: (For simplicity, assume that n is a power of 2.)

> $Y \leftarrow P(X)$
> Do in parallel for $1 \leq i \leq n$
>   append $\log n$ bits to the binary representation of the ith input
>   indicating its location in the input list, i.e. $x'_i \leftarrow (x_i) \times n + i$.
>   (Note that this does not affect the ordering of the elements.)
> Let $X' = \{x'_1, \ldots, x'_n\}$.
> $Y' \leftarrow P(X')$
> Let j be the last $\log n$ bits of y'_i: $j \leftarrow y'_i \bmod n$.
> Verify that $x'_j = y'_i$.
> Verify that x'_j has not been checked off yet and check off x'_j.
> Let $Y'' = \{y'_1 \mathrm{div} n, \ldots, y'_n \mathrm{div} n\}$.
> Verify that Y is in sorted order, $|Y| = n$ and that $Y = Y''$.
> If any verification fails, output "FAIL", else output "PASS".

5.2 Planar Convex Hull

Input: A list of points with their coordinates in R^2, labelled by their location in the input list: $(1, x_1, y_1), (2, x_2, y_2), ..., (n, x_n, y_n)$.

Output: A description of the boundary of the convex hull. This description will be a list of vertices of the convex hull in counter clockwise order around the hull.

Model of Computation: Real CRCW PRAM

The best algorithm for this problem in [Aggarwal, et. al.] runs in $O(\log n)$ depth and uses $O(n)$ processors.

The following algorithm result checks planar convex hull using constant checking time, but uses many processors:

Result Checking Algorithm: For each edge on the convex hull, n processors will be assigned to verify that all of the input points are on the same (correct) side of the edge. This can be done in constant parallel time with $O(n^2)$ processors.

The following algorithm result checks planar convex hull in a way that is is more efficient with processors.

Result Checking Algorithm: (Sketch) This result checker is a parallel implementation of the sequential result checker in [Gross, Irani, Rubinfeld, Seidel]. The result checker must verify that the polygon described in the output is simple and convex. This is done by verifying two things: that for each pair of consecutive edges, a left turn is made, and that a change in the x-direction of the walk is only made once when starting at the leftmost vertex. Next, the result checker must verify that all of the points not said to be on the hull are really inside the boundary. For each point not on the hull, it finds a "proof" that it is indeed inside the boundary. This proof will consist of four points in the input set whose convex combination contains the non-hull point. For each point not on the hull, the four points found that contain it will be the points on the upper and lower hulls that are immediately to the right and left of the non-hull point. To find these points, the result checker uses the program to sort the input points by x-coordinate by transforming the input points by $(i, x_i, y_i) \rightarrow (i, x_i, x_i^2)$. The hull points in the sorted list are marked. To implement this algorithm in parallel, a modification of the parallel prefix algorithm is used four times in order to find for each point p inside the hull, the closest point on the lower/upper hull to the left/right of p.

Running Time: The checking time is $O(\log n)$ and the checking number of processors is $O(n/\log n)$. The total time is $O(\log n + D(n))$ with $O(n/\log n + P(n)$ total processors. ■

5.3 Three Dimensional Convex Hull

Input: A list of points with their coordinates in R^3 (in general position).

Output: A description of the boundary of the convex hull. This will be a description of the vertices and edges of the hull, in the form of an adjacency list.

Model of Computation: Real RAM

The best known parallel algorithm from 3D convex hull mentioned in [Aggarwal, et. al.] requires $O((\log n)^2)$ time with $O(n)$ processors. The following result checking algorithm uses constant checking time, but uses many processors.

Result Checking Algorithm: For each face on the convex hull, n processors will be assigned to verify that all of the input points are on the same (correct) side of the face. Since there are only $O(n)$ faces on a convex hull, this can be done in constant parallel time with $O(n^2)$ processors.

The following result checking algorithm uses few processors:

Efficient, Fast Checker: (Sketch) The result checker must verify that the polyhedron described in the output is simple and convex. This is done by checking that the polyhedron is locally convex, which reduces to several two dimensional convex hull problems with total size $O(n)$. The partitioning of the two dimensional convex hull problems among the processors can be done in parallel using variants of a parallel prefix computation. Though not enough in two dimension, in three dimensions this is enough to show that the polyhedron is convex (see [Gross, Irani, Rubinfeld, Seidel]). Next the result checker must verify that all of the points not said to be on the hull are

really inside the boundary. For each point not on the hull, it finds a "proof" that it is indeed inside the boundary. This proof will consist of four points in the input set whose convex combination contains the non-hull point. It is not obvious how to parallelize the sequential method used in [Gross, Irani, Rubinfeld, Seidel], so a different technique is used. The idea is to reduce the search for a proof to a planar point location problem as follows: choose the point p in the input set with minimum x-coordinate. Imagine a wall perpendicular to the x-axis at the maximum x-coordinate. Suppose that someone standing at p aimed and shot a blue paint gun at every point on the convex hull and along every edge. This would paint a triangulated planar graph on the wall. If then the person standing at p shot a red paint gun at every other point in the input set, there would be several red dots on the wall. If one is told which face of the planar graph on the wall a particular red dot landed in, then one would have the proof that is being searched for. One must then just test that the point is really in the tetrahedron defined by p and the three points that determine the face that it landed in. Determining this face is simply a planar point location problem. [24, Tamassia Vitter] show how to create the planar point location data structure in $O(\log n)$ time with $n/\log n$ processors that supports point location queries in $O(\log n)$ time.

Running Time: The checking and total time is $O(\log n)$ and the checking and total number of processors is $O(n)$. ■

6 Duality

When result checking an optimization problem, it is necessary to check that the solution is as good as is claimed, and that it is the best solution. Duality can sometimes be used to show the latter.

For example, to result check a program that does linear programming, the result checker needs only check that the optimal solution is feasible, and to call the program again on the dual problem (again making sure that it is feasible) to check that the solution to the original problem is the same (and therefore optimal). If the program claims that there is no solution or that the solution is unbounded, this can be verified symbolically using the program in [8, Blum Kannan Rubinfeld]. This problem is P-complete, so no fast parallel algorithm is known for it. However, it can be result checked in logarithmic time with only two calls to the program.

Another example is the following:

Maximum Matching

Input: Graph $G = (V, E)$

Output: $k =$ the size of a maximum matching, and the edges in a maximum matching in G

No deterministic NC algorithm is known for this problem, but it is known to be in RNC ([20, Karp Upfal Wigderson], [22, Mulmuley Vazirani Vazirani]).

Result Checking Algorithm:(sketch)

The result checker first checks in parallel that no vertex is matched more than once and that the maximum matching is of size k. Then the algorithm in [18, Karloff] is used to find a proof that there is no matching of size $\geq k$. This proof will be an odd set cover of size k. Karloff's algorithm requires computing a maximal independent set. This computation can be checked using the result checker described earlier. Karloff's algorithm also calls a matching oracle on other problem instances. The result checker calls the matching program on these instances, and proceeds as if all of the answers are correct. If the output of his algorithm is an odd set cover of size $\neq k$, the result checker outputs

"FAIL". Otherwise, the odd set cover of size k is verification that the maximum matching is of size k, and the result checker outputs "PASS".

Running Time: The result checking time is $O(d^{MIS}(n))$ parallel steps and $O(p^{MIS}(n))$ processors, where $d^{MIS}(n)$ is the parallel depth and $p^{MIS}(n)$ is the number of processors required to find a maximal independent set in an n node graph. The total running time is $O(d^{MIS}(n) + D(n))$ with $O(n^3 \times N(n) + p^{MIS}(n))$ processors. ■

7 Constant Depth Reducible Functions

We can say something about the relationship among result checking problems that are AC^0 equivalent.

Proposition: Let π_1, π_2 be two AC^0 equivalent computational problems. Then from any fast program result checker C_{π_1} for π_1, is is possible to construct a fast program result checker C_{π_2} for π_2.

Proof: Similar to Beigel's trick described in [5, Blum]. We outline the proof for decision problems, but the general proof is similar. The idea is to construct a program result checker for π_2 by transforming it to an instance of π_1 and result checking that instance. Since the oracle program still only solves π_2, in order to get an oracle for π_1 on x, we use the reverse transformation on x into an instance of π_2, and call the oracle for π_2 on it. Since the transformation and the reverse transformation can be computed in AC^0, the depth of the result checker for π_2 will be at most a constant times the depth of the result checker for π_1. Since π_1 and π_2 are AC^0 equivalent, the fastest parallel program for each is related by a constant factor. Therefore, if π_1 is a fast program result checker, so is π_2. ■

More recently, in [17, Kannan] Section 2.3, Beigel's theorem was generalized to problems in the same *robust* complexity classes, and used to show that if two problems are equivalent under NC reductions, and if one has a result checker, then so does the other.

We have already shown two P-complete problems that are checkable in small depth: linear programming and straight line programming. In [17, Kannan] it is observed that since P-complete problems are all NC-reducible to each other, all P-complete problems are checkable in polylogarithmic depth. Moreover, a problem is presented for which programs can be written that run in small depth, but for which result checking the result is P-complete.

8 Acknowledgements

The author is especially grateful to Mike Luby for his many extremely helpful suggestions on this work and this manuscript, and to Dick Karp for pointing out that one can quickly check algorithms that can be solved by dynamic programming. The author is also very grateful to Manuel Blum, Sampath Kannan, Russell Impagliazzo, Yishai Mansour and Moni Naor for very helpful and interesting discussions.

References

[1] Adleman, L., Huang, M., Kompella, K., "Efficient Checkers for Number-Theoretic Computations", Submitted to *Information and Computation.*

[2] Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., Yap, C., "Parallel Computational Geometry", *FOCS 1985.*

[3] Alon, N., Babai, L., Itai, "A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem", *J. of Algorithms*, vol. 7, 1986, pp. 567-583.

[4] Ajtai, M., personal communication through M. Naor.

[5] Blum, M., "Designing Programs to Check Their Work".

[6] Beame, P., Hastad, J., "Optimal Bounds for Decision Problems on the CRCW PRAM", *STOC 1987.*

[7] Blum, M., Kannan, S., "Program Correctness Checking ... and the design of programs that check their work", *STOC 1989.*

[8] Blum, M., Kannan, S., Rubinfeld, R., "A Catalogue of Checkable Problems", in preparation.

[9] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," *STOC 1990.*

[10] Chandra, A., Fortune, S., Lipton, R., "Unbounded Fan-in Circuits and Associative Functions", *STOC 1983.*

[11] Fich, F., "New bounds for parallel prefix circuits", *STOC 1983*, pp.27-36.

[12] Furst, M., Saxe, J., Sipser, M., "Parity, Circuits and the Polynomial Time Hierarchy," *Math. Systems Theory*, vol. 17, 1984, pp.13-28.

[13] Goldberg, M., Spencer, T., "A New Parallel Algorithm for the Maximal Independent Set Problem," *FOCS 1987.*

[14] Gross, M., Irani, S., Rubinfeld, R., Seidel, R., "Checking Convex Hull Algorithms", in preparation.

[15] Ladner, R., Fischer, M.,"Parallel Prefix Computation", JACM, vol. 27, 1980, pp.831-838.

[16] Luby, M., "A Simple Parallel Algorithm for the Maximal Independent Set Problem", SIAM J. Comput., vol. 15, 1986, pp. 1036-1053.

[17] Kannan, S., "Program Result Checking with Applications", Ph.D. thesis, U.C. Berkeley, 1990.

[18] Karloff, H., "A Las Vegas RNC Algorithm for Maximum Matching," *Combinatorica*, vol. 6, 1986, pp.387-392.

[19] Karp, R., Ramachandran, V., "A Survey of Parallel Algorithms for Shared-Memory Machines", UC Berkeley Technical Report No. UCB/CSD 88/408.

[20] Karp, R., Upfal, E., Wigderson, A.,"Constructing a perfect matching is in random NC", *Combinatorica*, vol. 6, 1986, pp.35-48.

[21] Karp, R., Upfal, E., Wigderson, A.,"The Complexity of Parallel Search", *J. Comp. Syst. Sci.*, 1988.

[22] Mulmuley, K., Vazirani, U., Vazirani, V., "Matching is as Easy as Matrix Inversion", *STOC 1987*.

[23] Preparata, F., Shamos, M., *Computational Geometry.*

[24] Tamassia, R., Vitter, J. S., "Optimal Parallel Algorithms for Transitive Closure and Point Location in Planar Structures", *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures.*

ODA-Based Data Modeling in Multimedia Systems

Ralf Guido Herrtwich and Luca Delgrossi

TR-90-043

August 27, 1990

Abstract

A multimedia system can handle both discrete media (text, graphics) and continuous media (audio, video). The design of a multimedia system comprises processing and data modeling aspects. In this paper, we are concerned with data modeling only. We present a proposal to extend the ISO Office Document Architecture (ODA) to accommodate continuous media. To provide media flexibility, the needs for new ODA content architectures are identified. To take into account the timing requirements of continuous-media data, attributes for temporal synchronization are introduced for the logical and layout structure of an ODA document. To consider that multimedia information does not only appeal to the sense of vision, the layout structure is extended from two-dimensional visual space to arbitrary "presentation space". In addition, the inclusion of live information and hypertext features into ODA documents is proposed.

1. INTRODUCTION

A *multimedia system*, in our terminology, is a system capable of handling both *discrete media* (DM) such as text and graphics and *continuous media* (CM) such as audio and video. Today, the common way of building multimedia systems is to add facilities for handling CM data to an existing DM system. Traditionally, the functions of DM systems comprise *editing*, *formatting*, and *presentation* or *imaging*. Distributed DM systems also require *communicating* data. In a multimedia system, these functions not only have to be accomplished for DM data, but for CM data as well. As part of our ongoing research in the DASH Project at the International Computer Science Institute and the University of California at Berkeley, we have made proposals for this CM integration in previous papers [1-5], concentrating on aspects of *data processing*. In this paper, we focus on the other major issue in CM integration: *data modeling*, *i.e.*, how to structure and describe the information handled by multimedia systems.

Multimedia information comes in a variety of forms: DM data can be arranged by its authors to form books, reports, or letters; CM data is grouped into films, clips, and record albums; examples of combinations of DM and CM data include films with subtitles and voice-over text. We call any self-contained collection of coherent information, regardless of its complexity and inner structure, a *document*. To describe DM documents, ISO, CCITT, and ECMA have developed a set of international standards called the "*Office Document Architecture*" (ODA) [6]. We have chosen ODA as the basis of our multimedia data model for the following reasons:

- We believe that the structuring and handling of DM data, as defined by ODA, is well-understood today and poses no major research questions. Choosing a model that covers DM-related problems allows us to focus on CM data and the interaction between CM and DM data.
- Many elements of ODA are concerned with the structuring and handling of information, regardless of its nature; they are not only applicable to DM data, but to CM data as well. Hence, some of the problems in modeling CM data are already solved by ODA.
- The possibility of broad information exchange will be crucial to the success of future multimedia systems. As a widely supported international standard, ODA has the best chance of becoming a universally accepted framework for this communication. Furthermore, CM features for ODA have not yet been defined, so that some of our proposals may even find their way into the actual standardization.
- Finally, some DM systems incorporating ODA features are already available. Prominent examples include the Andrew system developed at Carnegie-Mellon University [7], the MULTOS system conceived by an ESPRIT project under the supervision of Olivetti [8], and the ISOTEXT system developed at the Technical University of Berlin [9]. We can use one of these systems as a starting point to implement our CM extensions, quickly developing an operational prototype that allows us to test our concepts in practice.

However, even for DM documents the current ODA version has one major problem: ODA is based on the traditional assumption that documents are organized sequentially so there is a single, predefined method of information consumption. In recent years, *hypermedia systems* [10] that allow users to browse through documents in arbitrary ways, following references between constituents of the document, have become common. This interactive access will be particularly useful for CM data; it offers new applications for audio and video [11]. Hence, when adding CM data to ODA we also have to add hypermedia features – otherwise we could not model the data of many interesting applications.

This paper is organized as follows: Section 2 summarizes the issues we consider important for the design of a multimedia data model. Section 3 gives a brief introduction to ODA; readers already familiar with the standard may skip this section. In Section 4 we propose ODA extensions to integrate CM elements. Our proposal does not have the same level of detail as the ODA standard, but it identifies necessary extensions and provides directions on how they can be accomplished. Section 5 concludes the paper with a brief comparison of our proposal and other suggestions for handling CM data within the ODA framework.

2. INTEGRATION ISSUES

When developing a data model for multimedia systems, we not only have to consider the new characteristics which distinguish CM from DM data, we also must take into account the applications that will make use of a multimedia data model. In the following subsections, we discuss three issues that we believe to be crucial for data modeling in multimedia systems.

2.1. Synchronization

Whereas DM data is time-invariant, CM data is time-critical, *i.e.*, not only the data values themselves, but also the times at which these values are presented (to a human user or a computer process) contribute to the data semantics. In order not to change these semantics unintentionally, any multimedia system has to ensure the correct (*i.e.*, author-defined) timing of CM data presentation. The temporal arrangement and ordering of operations is commonly termed *synchronization*. The following three synchronization problems have to be addressed by any multimedia system:

- *CM-internal synchronization*: Operations accessing consecutive values of the same CM data stream need to be synchronized. For example, the adjacent frames of a video need to be delivered to the monitor at regular intervals.
- *CM/CM synchronization*: If several streams of CM data are logically connected, operations accessing these streams need to be synchronized. For example, a movie and its soundtrack must be displayed in a way that synchronizes the spoken voice with the movement of the speaker's lips.
- *CM/DM synchronization*: If CM data and DM data are logically connected, operations on both kinds of data need to be synchronized. Examples include the presentation of voice-over text or adding subtitles to a movie as illustrated in Figure 1.

From the data modeling point of view, a multimedia system needs to provide the authors of a document with facilities to express their synchronization requirements. (In this paper, we are not concerned with how the system actually performs the desired synchronization. We refer to [5] and [12] for this purpose.)

2.2. Media Flexibility

Traditional DM data only comprises two-dimensional visual information that can be presented by a single device – a workstation screen or a printer. In a multimedia environment, these assumptions on how and where to present data are too restrictive. A multimedia data model has to be flexible in regard to the following aspects:

- *Sensual flexibility*: Multimedia systems will include media that appeal to different senses. Audio will soon play an important role in man-machine interaction. Future multimedia

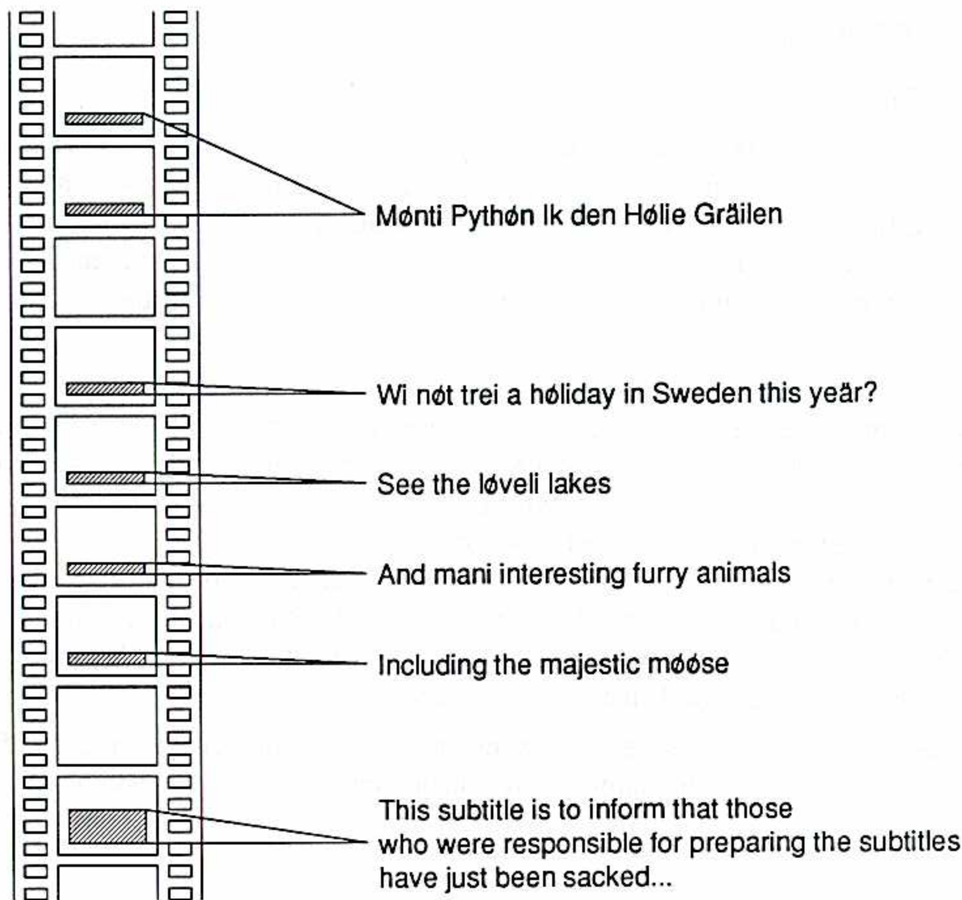

Figure 1: Subtitles in a movie have to be added to the right scenes to preserve the semantics intended by the authors.

systems aimed at creating a virtual reality (*e.g.*, for simulation purposes) may even appeal to our senses of touch or smell. Even the display of visual information may change: while text, graphics and video are all displayed in two visual dimensions, progress in computer-based generation of holographic images makes three-dimensional displays possible. Multimedia data can contain information for all senses, with arbitrary dimensions.

- *Device flexibility*: Media appealing to different senses need to be presented by different output devices. A multimedia system not only has to operate the workstation screen, but also a set of speakers and perhaps other devices. Multimedia applications in entertainment, *e.g.*, may require visual effects generated by laser beams; advanced multimedia presentations (such as Walt Disney's Star Tours) may even operate robots or use hydraulic devices to create gravitational forces.
- *Location flexibility*: Output devices of a multimedia document may be distributed. In many multimedia applications, such as video broadcast, different people at different locations must be provided with the same information at the same time. This is especially important in the

area of computer-supported cooperative work (CSCW) which will rely heavily on distributed multimedia information.

Since the methods of presenting information are potentially boundless, multimedia data models need to be extensible.

2.3. Live Information

DM data traditionally is exchanged in an asynchronous fashion: when exchanging electronic mail, *e.g.*, the sender and the receiver participate in the communication process at different times. Communication involving CM data is often synchronous: in a telephone conversation, *e.g.*, both parties are participating in the information exchange at the same time. Multimedia systems have to support both kinds of information exchange. This support results in the handling of two different kinds of data:

- *Recorded data*: If the entire information to be presented resides in storage, the multimedia system can process it as a whole, taking its overall structure and content into account. The system may work ahead, *i.e.*, it may prepare data for presentation before the actual presentation time arrives. The sequence in which the data is presented does not necessarily have to correspond to the sequence in which it was generated.
- *Live data*: If data evolves on the fly, as in a telephone conversation, its content is not known in advance. The multimedia system has less freedom in handling the data, both in respect to processing time and processing sequence. Also, the user cannot apply all presentation control functions (*e.g.*, fast-forward display) to live data.

The inclusion of live data goes beyond the traditional document model. A multimedia document becomes a means of describing information structure rather than a collection of static, reproducible data.

3. THE OFFICE DOCUMENT ARCHITECTURE (ODA)

The objective of the ODA standard is to make it possible to exchange, process, and present office documents in open distributed systems. ODA is based on a data model that distinguishes between the *content* and the *structure* of a document. The document content comprises all information that shall be presented to the human document user. It consists of *content portions*, each of which contains information of exactly one *content type*. Currently, ODA only considers three DM content types, namely character text, raster images, and geometric graphics. For each type, a *content architecture* is standardized, containing a specification of the type's elements, the control functions applicable to them, and the encodings to be used. It also determines the formatting and imaging process for these elements.

The structure of a document comprises meta-information about the document content which is useful for editing and formatting. Each document is divided into objects of a *logical structure* and objects of a *layout structure*. The logical structure of a document results from the editing process. It defines the document's components in terms meaningful to the human user, dividing the document into objects such as chapters, sections, paragraphs, or figures. ODA does not define the objects that can be part of a logical structure; this is up to the author. The layout structure of a document results from the formatting process. It groups the content portions of the document into presentation units, distinguishing between objects such as pages or columns. ODA defines a fixed set of objects that can be part of the layout structure and identifies their potential

relationships. Possible layout objects of a document are page sets, pages, frames, and blocks – nested in this order.

Both the logical and the layout structure are trees in which the objects constitute the nodes. The leaves of the trees are termed *basic objects* and can each be associated with content portions of the document. The tree structures of a document are determined by *attributes* associated with each node. Each of the following attributes defines a certain type of tree construction:

- *SEQUENCE*: The arguments (= subnodes) are sequentially ordered.
- *AGGREGATE*: There is no particular ordering among the arguments.
- *CHOICE*: One of the arguments is chosen.

In addition, each of these attributes can be prefixed by one of the following construction factors:

- *OPTIONAL*: The argument may or may not be included in the structure.
- *REPEAT*: The argument may be included in the structure one or more times.
- *OPTIONAL REPEAT*: The argument may be included in the structure zero or more times.

If no construction factor is specified, the argument is included in the structure exactly once.

Attributes can refer not only to the tree construction, but also to other properties; a typical attribute of a frame, *e.g.*, would be its spatial position within a page. Attributes can also be used to associate objects of different structures with each other. To determine attributes resulting from the formatting process, the user can specify formatting directives resulting in certain *styles*. *Layout styles* specify rules or restrictions for mapping logical objects onto layout objects. They describe, *e.g.*, that a figure and its caption should appear on the same page. *Presentation styles* are associated with basic logical or layout objects and define content-specific formatting attributes such as the font to be used for displaying text. By changing its style definitions, a document can be presented in a variety of ways although its content and logical structure remain unchanged.

Apart from the *specific structures* of a certain document, ODA also permits the definition of *generic structures*. These structures can be used to specify *document classes*. A document class comprises documents with identical structural characteristics. For example, all technical reports of a project can be considered as instances of the same document class. They all have a similar logical structure and shall be formatted in the same way. They may even share standard content portions such as the project logo or the list of sponsors. Similarities can also be seen within a document: each paragraph, *e.g.*, should be handled in the same manner. This is expressed by grouping objects into *object classes*. Generic mechanisms save the author specification work, particularly in defining document styles. They also allow more efficient document encoding and exchange.

4. EXTENDING ODA TO ACCOMMODATE CONTINUOUS MEDIA

The elements of ODA are as important for modeling CM data as they are for DM data. Consider the notion of generic structures as an example: all episodes of a certain TV program, *e.g.*, a game show, belong to the same document class and share common attributes. Also the concept of logical structure is essential for CM data: just as a story is divided into chapters and paragraphs, a movie is structured into scenes and shots, a symphony into sets and movements, *etc.* Many insufficiencies in handling CM data today result from the lack of information about its logical structure. For example, when watching a television program recorded with a VCR, we would

like to skip commercials automatically. If no information about the logical structure of the program is available, this function is hard to implement.

To accommodate CM data, we suggest the following three extensions to ODA:

(1) New kinds of document content need to be taken into account.

(2) Attributes for temporal synchronization between document objects should be included in the logical and layout structures.

(3) Hypermedia features need to be added to the ODA elements.

While the first two extensions are absolutely necessary to accommodate CM data, the third extension is desirable to arrive at a data model suitable for typical multimedia applications.

4.1. Document Content

Document content in the ODA standard is restricted to three DM content architectures for characters, geometric graphics, and raster graphics. The straightforward extension of the model is to introduce new content architectures for CM data. This extension, however, also requires broadening the overall ODA content model to take into account that data can be time-critical, that data can represent non-visual information (in fact, that it can represent any code to which a presentation device may respond), and that data does not need to be recorded in storage.

4.1.1. Audio and Video Content Architectures

To include CM data into ODA, at least two new content architectures are needed: one for audio and one for video. To encode video content portions, one needs to define the two dimensions of each image, the image sampling structure, and the encoding of color and luminance quantums. For audio, the number of channels and the coding of sample quantums needs to be given.

Just as existing DM content architectures refer to other standards to describe the encoding of values, CM content architectures can make use of encoding standards which already exist or are under development. For example, a set of CCITT G-Series recommendations defines PCM-based encodings for voice [13-15], and within the J-Series a recommendation for high-quality monophonic digital sound can be found [16]. Other *de facto* standards include compact disc audio, compact disc interactive (CD-I), and digital audio tape. In the area of video encoding, several HDTV proposals are currently under consideration. The CCITT H-Series comprises recommendations for audiovisual conferencing [17]. CCIR Recommendation 601 [18] defines a family of digital video encoding methods for studio production. Digital video interactive (DVI) is a video encoding scheme already in use with personal computers today [19].

Since video without audio is unrealistic, a video content architecture needs to provide the means to incorporate audio (in the same way as geometric graphics can include textual annotations). But even if audio and video are kept separate in different content architectures, the encoding of a video program and its soundtrack can be interleaved as in most of today's encoding schemes. If content portions which have to be presented at the same time are stored at adjacent physical locations (*e.g.*, on a disk) it is easier to accomplish tight timing requirements when accessing them.

4.1.2. Timed Content Portions

A common factor in both audio and video is that they constitute *timed content portions*. Each portion contains a sequence of time-critical values. Each value has a certain life-span resulting from the sampling rate at which CM data is recorded. The cumulative life-spans of all values in a

content portion define the *duration* of the entire content portion. We call this duration *closed* because its length is fixed and known in advance.

Apart from CM data, a multimedia document may contain other timed content portions that have *open* durations, *i.e.*, durations of undefined length. These timed content portions represent actions of presentation devices which take a certain amount of time that either varies or is unknown. An example is an output operation performed by a robot: the time the robot needs to reach a certain position depends on its previous position; this time may not be available beforehand to the author or the system.

4.1.3. Definition of New Content Architectures

The main reason for defining a fixed set of content architectures in the ODA standard is to ensure the exchangeability of documents in open systems. This objective conflicts with the wish to model arbitrary multimedia documents – even highly sophisticated, non-transferable multimedia presentations requiring special equipment such as lasers or holographic projectors – avoiding the need for specialized software to handle these documents.

To access non-standardized devices, users need the ability to define content architectures for these machines themselves. ODA must provide a specification language for this purpose. In the simplest case, this language would allow authors to specify the presentation operations of a device by denoting the code that causes the device to perform the operation. Once a new class of presentation devices becomes more widespread, a new standardized content architecture for the media that these machines provide can be defined.

4.1.4. Live Content Portions

Live information is a special kind of generic timed content portion. Its actual data is generated at the time the document is presented; it is not obtained from storage, but from some recording equipment that delivers the information directly to the imaging process. When the imaging process arrives at a point that makes the display of a live content portion necessary (or at some time earlier), it has to establish a connection with the device recording the data. In the case of video, this device will be a camera; in the case of audio, it will be a microphone. In typical applications such as video-conferencing, these devices can be attached to remote machines so that the connection with the imaging process must be established across a network.

Since the actual content of a live content portion is not available when the document structures are defined, a *fill-in value* is used instead. The attributes of this fill-in value are used in the formatting process. For example, depending on the size of a fill-in video frame, some area in a document page can be reserved to display a live video. In principle, these mechanisms – although they are intended for CM data – can be used for DM data as well, *e.g.*, to describe a textual conversation between users similar to the UNIX *talk* program.

4.2. Specification of Temporal Relations

Timing affects CM data semantics. In a subtitled movie, *e.g.*, a scene may take on an entirely different meaning if the wrong subtitles are added to it. Therefore, temporal synchronization is not merely a layout issue, but also a logical one. However, the logical structure of a document will only define those temporal relations that are important to ensure the correct document semantics. Other temporal relations for presentation purposes may be determined during the formatting process, perhaps resulting from layout directives.

4.2.1. Timed Objects

To express temporal relations within the logical and layout structure of a document we introduce the notion of *timed objects*. Like timed content portions, timed objects have a certain *duration*. There are two kinds of basic timed objects:

- Basic timed objects that are associated with a timed content portion have the duration of this content portion. Whether the duration is open or closed depends on whether the duration of the timed content portion is open or closed.
- Basic timed objects that are associated with explicit time intervals given in seconds have the duration of this time interval. Their duration is always closed. The main application of these explicit time intervals is to assign durations to DM data content portions.

Higher-level timed objects get their durations from *temporal construction attributes* that are used to combine timed objects to a tree structure. These attributes determine the temporal position of their arguments (called *sources*) within the duration of the parent node (called *destination*). This process is called ''mapping''. Let s_0 to s_{n-1} denote the sources and let d be the destination. To define the semantics of temporal construction attributes, let each interval i have a startpoint S, an endpoint E, and a duration D (denoted as $i.S$, $i.E$, and $i.D$). Using these definitions, we introduce the following attributes, illustrated in Figure 2:

- *MEETS*: The destination duration is the sum of the source durations. The sources are mapped so that the endpoint of the previous source coincides with the startpoint of the next.

  $$d.D = \sum_i s_i.D, \quad s_0.S = d.S, \quad s_i.S = s_{i-1}.E$$

  If one source duration is open, the destination duration is open.
- *BEFORE*: The destination duration is larger than the sum of the source durations. The sources are mapped so that the endpoint of the previous source lies some positive time ahead of the startpoint of the next.

  $$d.D = \sum_i s_i.D + \Delta \ (\Delta > 0), \quad s_0.S = d.S, \quad s_i.S < s_{i-1}.E, \quad s_{n-1}.E = d.E$$

  The mapping is nondeterministic; the absolute position of each source in the destination is not defined unequivocally because the lengths of gaps between the sources are not specified. If one source duration is open, the destination duration is open.
- *OVERLAPS*: The destination duration is smaller than the sum of the source durations. A source is mapped so that its startpoint is between the startpoint and endpoint of the previous source and that its endpoint is after the endpoint of the previous source.

  $$d.D = \sum_i s_i.D - \Delta \ (d.D > 0),$$

  $$s_0.S = d.S, \quad s_{i-1}.S < s_i.S < s_{i-1}.E, \quad s_i.E > s_{i-1}.E, \quad s_{n-1}.E = d.E$$

  The mapping is nondeterministic because the distance between the source startpoints is not defined. This attribute cannot be applied to open durations.
- *START*: The destination duration is the longest source duration. The sources are mapped so that their startpoints coincide.

  $$d.D = \max_i s_i.D, \quad s_i.S = d.S$$

  If one source duration is open, the destination duration is open.

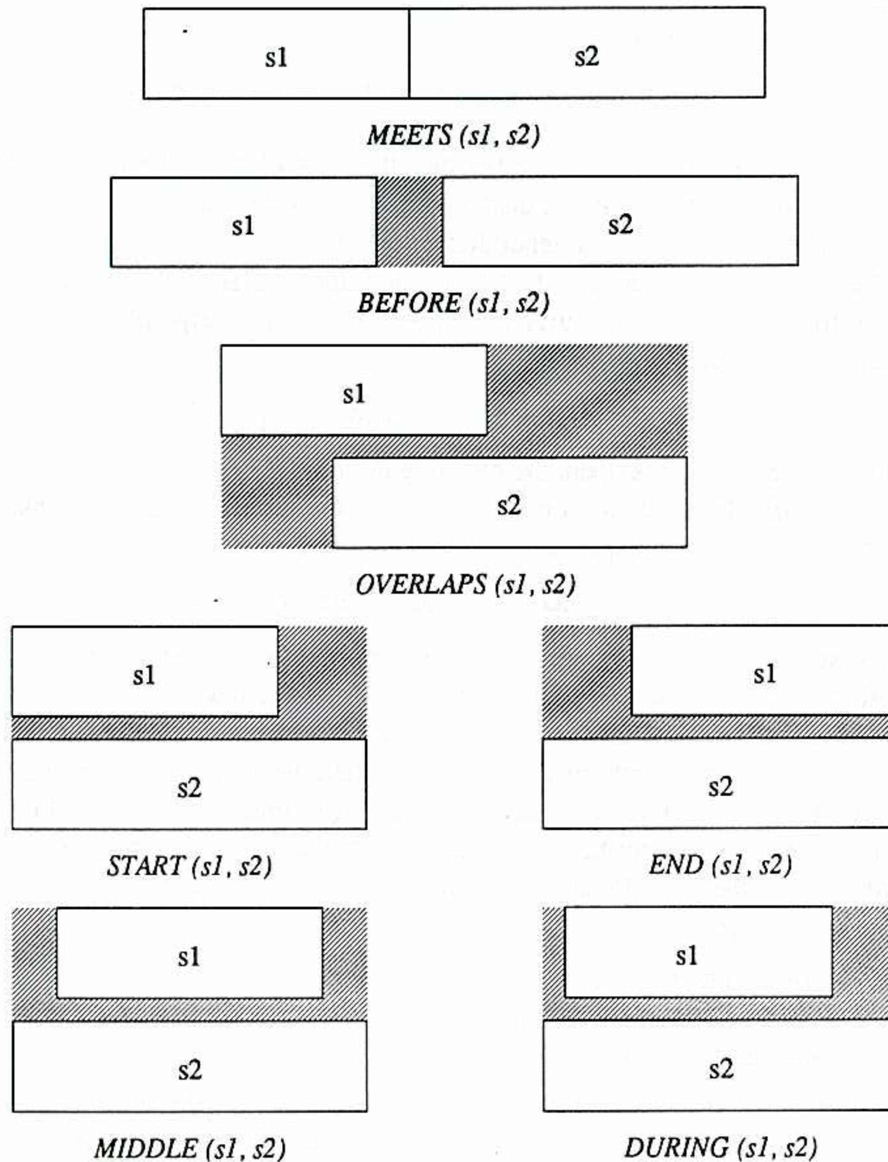

Figure 2: Temporal construction attributes.

- *END*: The destination duration is the longest source duration. The sources are mapped so that their endpoints coincide.

$$d.D = \max_i s_i.D\,, \quad s_i.E = d.E$$

  This attribute cannot be applied to open durations.
- *MIDDLE*: The destination duration is the longest source duration. The sources are mapped so that their middlepoints coincide.

$$d.D = \max_i s_i.D\,,\quad s_i.S = \frac{d.D - s_i.D}{2}$$

This attribute cannot be applied to open durations.

- *DURING*: The destination duration is the longest source duration. All sources are mapped so that they fit into the destination.

  $$d.D = \max_i s_i.D\,,\quad s_i.S \geq d.S\,,\quad s_i.E \leq d.E$$

  The mapping is nondeterministic because the startpoints of the intervals are not defined. If one source duration is open, the destination duration is open.
- *AT [t]*: The *AT* attribute is a generalization of the previous attributes for absolute timing. As for the previous attributes, the destination duration lasts from the start of the first source to the end of the last source. A source is mapped so that its startpoint is t moments later than the startpoint of the previous source.

  $$d.D = (n-1)t + s_{n-1}.D\,,\quad s_0.S = d.S\,,\quad s_i.S = s_{i-1}.S + t$$

  If one source duration is open, the resulting duration is open.
- *EQUAL [m]*: The destination duration is the same as all source durations. Startpoints of all sources coincide, and endpoints of all sources coincide.

  $$d.D = s_i.D\,, s_i.S = d.S\,, s_i.E = d.E$$

  If one source duration is open, the destination duration is open. If source durations are not equal, they need to be made equal. The parameter m defines in which of the following two ways this "duration adjustment" shall be accomplished: *CUT* or *FILL* (see Figure 3). In the first case, all source durations that are longer than the shortest one are cut. In the second case, all source durations that are shorter than the longest one are filled. If filling is desired, a filling method needs to be described. An argument for filling is an interval or a sequence of intervals combined by the *MEETS* attribute where at least one interval carries one of the following *filling attributes*:

  - $]A$: Use the first value of A for filling.
  - $A\,[$: Use the last value of A for filling.
  - $[A\,]$: Repeat A for filling.

  The arguments for filling can also be untimed objects. In combination with explicit time intervals, the *EQUAL* operator can be used to assign a duration to untimed objects and to transform them into timed objects. Assuming A is an untimed content portion, the following expression assigns a duration of 20 seconds to A:

  $$EQUAL\ [FILL]\ ([A\,], 20\,s)$$

  If the argument sequence offers more than one possibility for filling and all source durations are closed, filling is divided equally among the intervals. If a source duration is open, only filling at the end of each argument sequence can be used.

All relative attributes are derived from the operators used in temporal interval logic by Allen [20]. These operators are known to be complete so that all possible temporal relations between intervals can be described by logical disjunction of the operators. We have added the *MIDDLE* attribute for specification convenience. The *AT* attribute serves for absolute timing which is not included in the original interval logic.

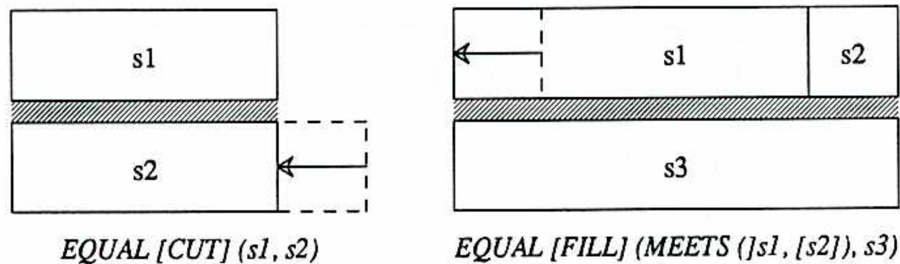

Figure 3: Cutting and filling timed objects of equal durations.

4.2.2. Timing in the Logical Structure

In the logical structure, timed objects and untimed objects are mixed depending on whether timing is essential for their logical correctness or not. Timed objects can be combined with untimed objects using the construction attributes introduced in Section 3. For example, a textual introduction to a movie and the movie itself would be combined using the *SEQUENCE* attribute, whereas a review or some press information about the movie is associated with it using the *AGGREGATE* construction. The result of the combination of timed and untimed objects is always an untimed object.

4.2.3. Timing in the Layout Structure

In the layout structure, every object is timed because presentation is never time-independent. Even the original ODA approach contains a notion of presentation time by introducing the separation of documents into pages. Pages are not only spatial divisions of a document, but also those document portions which a user sees at a certain time. Each page constitutes an open duration where the transition to the next duration is determined by the document user. Within each duration, a two-dimensional visual space is filled with values.

In the generalized layout structure that accommodates CM data all spaces, regardless of the senses to which they appeal (visual space, acoustic space, *etc.*), the dimensions they have (zero as monophonic audio, one as stereophonic audio, two as screen video, or more), and the places at which they are located, comprise the *presentation space*. The values of this presentation space are interpreted over time, *i.e.*, the layout structure constitutes a temporal and spatial division of the presentation space into timed objects as illustrated in Figure 4.

4.2.4. Timing by Layout and Presentation Directives

Layout directives describe certain ways to transform the logical structure into the layout structure. They are used by authors to direct the formatting process when several alternatives for formatting are possible. In our model for temporal synchronization, such alternatives occur in non-deterministic constructions of timed objects. For each of these constructions, a layout directive may specify the precise timing. For example, a layout directive may determine the time between two sources combined with the *BEFORE* attribute.

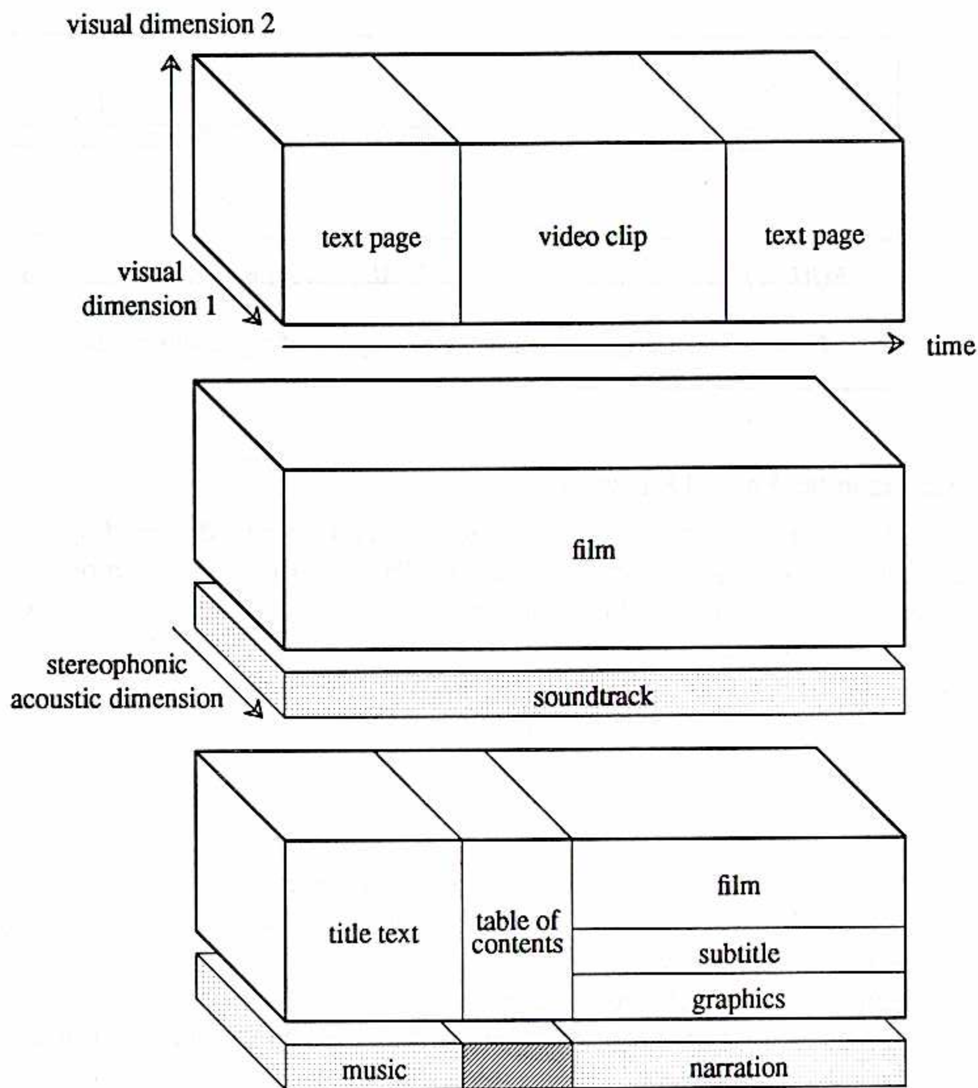

Figure 4: Separation of a presentation space.
Above: Over time. Middle: Over space. Below: Over time and space.

Presentation directives determine how content portions are presented to the user. Depending on the kind of data to be displayed, different kinds of presentation directives are possible. For video, *e.g.*, some digital filtering functions may be used. Common to all CM data content portions are presentation directives that refer to their temporal characteristics. The following attributes can be set by presentation directives:

- The *presentation speed* can be changed. This makes effects such as slow motion and time lapses possible.
- The *presentation direction* can be inversed. A video, *e.g.*, is then displayed backwards.

- The *initial presentation position* within a CM data portion can differ from the actual startpoint of the data portion. A piece of music, *e.g.*, may be started somewhere in the middle.

We have introduced these presentation attributes previously in the time capsule model [5]. In fact, time capsules, which are containers for time-critical data, can be used to store and access CM data content portions of ODA documents.

4.3. Hypermedia Features

Traditional documents, as those resulting from the ODA tree structures, are linear; there is a predefined way of consuming them which is determined by the author. For some applications in which each user follows a different path through the document, this linear organization is inadequate. Rather, here the information should be organized in *hypermedia* form, *i.e.*, as a network of nodes in which *links* show references between information portions. Examples of documents for which a hypermedia structure is adequate include on-line manuals of computer programs, self-guided tutorials, and encyclopedias.

The links in a hypermedia document enable users to browse through the information in a method of their own choosing. To traverse a link, the user has to select it explicitly. In today's hypertext systems, the traversal is initiated by the user clicking on a button with a mouse. Activation by text or voice input are other options. Transitions between the constituents of a hypertext document resemble turning from one page to another. However, the separation of a document into pages is merely made for presentation purposes. Hypermedia links, on the other hand, are part of a document's logical structure.

In the logical and layout structure, links can be represented as basic objects, *i.e.*, as leaves of a document tree that refer to other nodes within the same structure (*e.g.*, by name). For each link in the logical structure, there is a corresponding link in the layout structure. Each link is associated with one or more content portions that identify the means by which a user can initiate the traversal of the link. Obviously, links may be assigned durations just as DM data content portions, representing that the user input to traverse the link is subjected to a timeout.

5. CONCLUSION

We have identified necessary extensions to ODA for accommodating CM data. The core of our proposal – even if the proposal goes beyond this – are the mechanisms to specify temporal synchronization, a topic which has also attracted the interest of other research groups. An early proposal for temporal synchronization within the ODA framework was developed within the ANSA Project [21]; it is based on absolute time stamps in content portions. Our proposal is closer to the work by Little and Ghafoor [22] who suggest relative synchronization based on temporal interval logic just as we do. However, their system does not include mechanisms for duration adjustment such as our *CUT* and *FILL* attributes.

The duration adjustment features of our model were inspired by by the "restricted blocking" primitive introduced in [12] and a proposal currently being discussed in the ODA standardization committee [23]. This proposal uses a much simpler model for temporal synchronization with only two synchronization attributes:

- *SERIAL [n]*: The sources are presented sequentially. They are included in the destination *n* times.
- *PARALLEL [m]*: The sources are presented simultaneously, with their startpoint coinciding. Depending on the mode *m*, the destination duration ends with the shortest source duration (*S*) or with the longest duration (*A*).

Although these attributes seem to be general enough to express most synchronization requirements, they sometimes require the arbitrary separation of CM data portions for expressing parallelism (consider, *e.g.*, the display of a subtitle within a video sequence) and tend to support overspecification since no nondeterministic mapping is allowed.

ACKNOWLEDGEMENTS

The ideas contained in this paper evolved during the weekly Monday Multimedia Meetings at ICSI. We are indebted to the participants of these meetings, Ramesh Govindan, George Homsy, Yukkei Hui, and Kyoji Umemura, for their contributions and comments. Ute and Carsten Bormann deserve credit for drawing our attention to ODA. Seth Jacobson was instrumental in getting all the necessary documents to complete this study. As always, Renee Reynolds' suggestions to improve the style of this paper were invaluable.

REFERENCES

1. D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan and M. Andrews, "Support for Continuous Media in the DASH System", *Proc. of the 10th International Conference on Distributed Computing Systems*, Paris, May 1990.
2. D. P. Anderson and R. G. Herrtwich, "Resource Management for Digital Audio and Video", *IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, May 1990, 99-103.
3. D. P. Anderson, R. G. Herrtwich and C. Schaefer, "SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet", Technical Report 90-006, International Computer Science Institute, Feb. 1990.
4. D. P. Anderson, R. Govindan and G. Homsy, "Abstractions for Continuous Media in a Network Window System", *International Conference on Multimedia Information Systems*, Singapore, Jan 1991.
5. R. G. Herrtwich, "Time Capsules: An Abstraction for Access to Continuous-Media Data", *11th Real-Time Systems Symposium*, Orlando, Dec. 1990.
6. "Office Document Architecture (ODA) and Interchange Format", International Standard 8613, ISO.
7. J. Akkerhuis, A. Marks, J. Rosenberg and M. S. Sherman, "Processable Multimedia Document Interchange Using ODA", *EUUG Autumn Conference*, Vienna, Sep. 1989, 167-177.
8. C. Thanos, ed., *Multimedia Office Filing – The MULTOS Approach* , North-Holland, 1990.
9. U. Bormann and C. Bormann, "ISOTEXT - A WYSIWYG Editing and Formatting System for ODA and SGML Documents", *Offene Multifunktionale Arbeitsplaetze*, Berlin, 1988.

10. J. Smith and S. Weiss, ''An Overview of Hypertext'', *Comm. of the ACM 31*, 7 (July 1988), 816-819.

11. R. G. Herrtwich, ''Audio and Video in Distributed Computer Systems: Why and How?'', *IEEE Workshop on Communications and Artificial Intelligence*, Reisensburg, Sep. 1990. (Also: International Computer Science Institute, Technical Report 90-026).

12. R. Steinmetz, ''Synchronization Properties in Multimedia Systems'', *IEEE Journal on Selected Areas in Communications 8*, 3 (Apr. 1990), 401-412.

13. ''Pulse Code Modulation'', Recommendation G.711, CCITT.

14. ''Adaptive Differential Pulse Code Modulation'', Recommendation G.721, CCITT.

15. ''Sub-Band Adaptive Differential Pulse Code Modulation'', Recommendation G.722, CCITT.

16. ''High Quality Sound'', Recommendation J.41, CCITT.

17. *H-Series Recommendations (Audiovisual Conferencing)*, CCITT.

18. ''Encoding Parameters of Digital Television for Studios'', Recommendation 601, CCIR.

19. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.

20. J. F. Allen, ''Maintaining Knowledge about Temporal Intervals'', *Comm. of the ACM 26*, 11 (Nov. 1983), 823-843.

21. J. S. Sventek and J. Wratten, *Temporal Aspects of Multimedia Data Structures*, Advanced Networked Systems Architecture Project, Technical Report TI.31.00, Feb. 1988.

22. T. D. C. Little and A. Ghafoor, ''Synchronization and Storage Models for Multimedia Objects'', *IEEE Journal on Selected Areas in Communications 8*, 3 (Apr. 1990), 413-427.

23. ''Time Synchronization'', IEC JTC1/SC18/WG3, N1443 (Working Paper), ISO.