Using Certification Trails to Achieve
Software Fault Tolerance

Gregory F. Sullivan and Gerald M. Masson

Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
Report JHU-89/26
Using Certification Trails to Achieve Software Fault Tolerance

Gregory F. Sullivan
Gerald M. Masson

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

We introduce a conceptually novel and powerful technique to achieve fault tolerance in hardware and software systems. When used for software fault tolerance, this new technique uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a trail of data which we call a certification trail. In the second phase, another program is run which solves the original problem again. This program, however, has access to the certification trail left by the first program. Because of the availability of the certification trail, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certification trail it receives from the first program is incorrect. We formalize the certification trail approach to fault tolerance and illustrate realizations of it by considering algorithms for the following problems: minimum spanning tree, Huffman tree, and convex hull. We discuss cases in which the second phase can be run concurrently with the first and act as a monitor. We compare the certification trail approach to other approaches to fault tolerance.

Keywords: Software fault tolerance, error monitoring, design diversity, data structures.

1Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.
2Research partially supported by NASA Grant NSG 1442.
1 Introduction

In this paper we introduce a novel and powerful technique for achieving fault tolerance in systems. Although applicable to both hardware and software, we restrict our discussion of this technique in the following to software fault tolerance. To explain our new technique for software fault tolerance, we will first discuss a simpler fault tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so called time redundancy [23, 42]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming[12, 8] (in this case N=2), allows for the detection of errors caused by some faults in the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

The technique we will describe is designed to achieve similar types of error detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a certification trail. This data is chosen so that it can allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains a error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error
occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the data trail. Finally, it should be noted that in Figure 1 both executions can signal an error. These errors would include run-time errors such as divide-by-zero or non-terminating computation. In addition the second execution can signal error due to an incorrect certification trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem $P$ is formalized as a relation (that is, a set of ordered pairs). Let $D$ be the domain (that is, the set of inputs) of the relation $P$ and let $S$ be the range (that is, the set of solutions) for the problem. We say an algorithm $A$ solves a problem $P$ iff for all $d \in D$ when $d$ is input to $A$ then an $s \in S$ is output such that $(d, s) \in P$.

Definition 2.2 Let $P : D \rightarrow S$ be a problem. Let $T$ be the set of certification trails. A solution to this problem using a certification trail consists of two
functions \( F_1 \) and \( F_2 \) with the following domains and ranges \( F_1 : D \to S \times T \) and \( F_2 : D \times T \to S \cup \{ \text{error} \} \). The functions must satisfy the following two properties:

1. for all \( d \in D \) there exists \( s \in S \) and there exists \( t \in T \) such that \( F_1(d) = (s,t) \) and \( F_2(d,t) = s \) and \( (d,s) \in \mathcal{P} \)
2. for all \( d \in D \) and for all \( t \in T \) 
   - either \( F_2(d,t) = s \) and \( (d,s) \in \mathcal{P} \) or \( F_2(d,t) = \text{error} \).

The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct. It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

The certification trail approach also allows for the detection of faults in software. As in 2-version programming, separate teams can write the algorithms for the first and second executions. Note that the specification now must include precise information describing the generation and use of the certification trail. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. This will be illustrated in the convex hull example to be considered later. Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. This will be illustrated in the minimum spanning tree and Huffman tree examples to be considered later. When significantly different algorithms are used, the probability that both algorithms will contain or be effected by faults which generate matching errors should be reduced. When very similar algorithms are used it is sometimes possible to save programming effort by sharing program code. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier.

Throughout this section we have assumed that our method is implemented with software; however, it is clearly possible to implement the certification trail technique by using dedicated hardware. It is also possible to
generalize the basic two-level hierarchy of the certification trail approach as illustrated in Figure 1 to higher levels. Finally, we note that a wide variety of approaches to software and hardware fault tolerance have been proposed which bear resemblances to the certification trail approach; we contrast our method to the most closely related ideas in a later section.

3 Examples of the Certification Trail Technique

In this section we illustrate the use of certification trails by means of applications to three well-known and significant problems in computer science: the minimum spanning tree problem, the Huffman tree problem, and the convex hull problem. It should be stressed here that the certification trail approach is not limited to these problems. Rather, these algorithms have been selected only to give illustrations of this technique.

3.1 Minimum Spanning Tree Example

The minimum spanning tree problem has been examined extensively in the literature and an historical survey is given in [17]. Our certification trail approach is applied to a variant of the Prim/Dijkstra algorithm [38, 13] as explicated in [44]. We will begin our discussion of the application of the certification trail approach to the minimum spanning tree problem with some preliminary definitions.

Definition 3.1 A graph $G = (V, E)$ consists of a vertex set $V$ and an edge set $E$. An edge is an unordered pair of distinct vertices which we notate as, for example, $[v, w]$, and we say $v$ is adjacent to $w$. A path in a graph from $v_1$ to $v_k$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_i, v_{i+1}]$ is an edge for $i \in \{1, \ldots, k - 1\}$. A path is a cycle if $k > 1$ and $v_1 = v_k$. An acyclic graph is a graph which contains no cycles. A connected graph is a graph such that for all pairs of vertices $v, w$ there is a path from $v$ to $w$. A tree is an acyclic and connected graph.

Definition 3.2 Let $G = (V, E)$ be a graph and let $w$ be a positive rational valued function defined on $E$. A subtree of $G$ is a tree, $T(V', E')$, with $V' \subseteq V$
and $E' \subseteq E$. We say $T$ spans $V'$ and $V'$ is spanned by $T$. If $V' = V$ then we say $T$ is a spanning tree of $G$. The weight of this tree is $\sum_{e \in E'} w(e)$. A minimum spanning tree is a spanning tree of minimum weight.

### 3.1.1 Data structures and supported operations

Before we discuss the minimum spanning tree algorithm, we must describe the properties of the principle data structure that are required. Since many different data structures can be used to implement the algorithm, we initially describe abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set; however, at all times, the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and $n$, inclusive. Our default convention is that $i$ is an item number, $k$ is a key value and $h$ is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i, k) < (i', k')$ iff $k < k'$ or $(k = k'$ and $i < i')$. Our data structure should support a subset of the following operations.

- **member**($i, h$) returns a boolean value of true if $h$ contains an ordered pair with item number $i$, otherwise returns false.

- **insert**($i, k, h$) adds the ordered pair $(i, k)$ to the set $h$.

- **delete**($i, h$) deletes the unique ordered pair with item number $i$ from $h$.

- **changekey**($i, k, h$) is executed only when there is an ordered pair with item number $i$ in $h$. This pair is replaced by $(i, k)$.

- **deletemin**($h$) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If $h$ is the empty set then the token "empty" is returned.

- **predecessor**($i, h$) returns the item number of the ordered pair which immediately precedes the pair with item number $i$ in the total order. If there is no predecessor then the token "smallest" is returned.
Many different types and combinations of data structures can be used to support these operations efficiently. In our case, we will actually use two different data structure methods to support these operations. One method will be used in the first execution of the algorithm and another, faster and simpler, method will be used in the second execution. The second method relies on a trail of data which is output by the first execution.

### 3.1.2 MINSPAN algorithm

Before discussing precise implementation details for these methods we present the overall algorithm used in both executions. Pidgin code for this algorithm appears below. In addition, Figure 2 illustrates the execution of the algorithm on a sample graph and the table below records the data structure operations the algorithm must perform when run on the sample graph. The first column of the table gives the operations except member and with the parameter \( h \) dropped to reduce clutter. The second column gives the evolving contents of \( h \). The third column records the ordered pair deleted by the deletemin operation. The fourth column records the certification trail corresponding to these operations and is further discussed below.

The algorithm uses a "greedy" method to "grow" a minimum spanning tree. The algorithm starts by choosing an arbitrary vertex from which to grow the tree. During each iteration of the algorithm a new edge is added to the tree being constructed. Thus, the set of vertices spanned by the tree increases by exactly one vertex for each iteration. The edge which is added to the tree is the one with the smallest weight. Figure 2 shows this process in action. Figure 2(a) shows the input graph, Figures 2(b) through 2(e) show several stages of the tree growth and Figure 2(f) shows the final output of the minimum spanning tree. The solid edges in Figures 2(b) through 2(e) represent the current tree and the dotted edges represent candidates for addition to the tree.

To efficiently find the edge to add to the current tree the algorithm uses the data structure operations described above. As soon as a vertex, say \( v \), is adjacent to some vertex which is currently spanned it is inserted in the set \( h \). The key value for \( v \) is the weight of the minimum weight edge between \( v \) and some vertex spanned by the current tree. The array element prefer(\( v \)) is used to keep track of this minimum weight edge. As the tree grows, information is updated by operations such as insert(\( i, k, h \)) and changekey(\( i, k, h \)). The
<table>
<thead>
<tr>
<th>Operation</th>
<th>Set of Ordered Pairs</th>
<th>Delete</th>
<th>Trail</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(2,200)</td>
<td>(2,200)</td>
<td></td>
<td>smallest</td>
</tr>
<tr>
<td>insert(6,500)</td>
<td>(2,200),(6,500)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>deletemin</td>
<td>(6,500)</td>
<td>(2,200)</td>
<td></td>
</tr>
<tr>
<td>insert(3,800)</td>
<td>(6,500),(3,800)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>changekey(6,450)</td>
<td>(6,450),(3,800)</td>
<td></td>
<td>smallest</td>
</tr>
<tr>
<td>insert(7,505)</td>
<td>(6,450),(7,505),(3,800)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>deletemin</td>
<td>(7,505),(3,800)</td>
<td>(6,450)</td>
<td></td>
</tr>
<tr>
<td>insert(5,250)</td>
<td>(5,250),(7,505),(3,800)</td>
<td></td>
<td>smallest</td>
</tr>
<tr>
<td>changekey(7,495)</td>
<td>(5,250),(7,495),(3,800)</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>deletemin</td>
<td>(7,495),(3,800)</td>
<td>(5,250)</td>
<td></td>
</tr>
<tr>
<td>changekey(3,350)</td>
<td>(3,350),(7,495)</td>
<td></td>
<td>smallest</td>
</tr>
<tr>
<td>insert(4,700)</td>
<td>(3,350),(7,495),(4,700)</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>deletemin</td>
<td>(7,495),(4,700)</td>
<td>(3,350)</td>
<td></td>
</tr>
<tr>
<td>changekey(4,650)</td>
<td>(7,495),(4,650)</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>deletemin</td>
<td>(4,650)</td>
<td>(7,495)</td>
<td></td>
</tr>
<tr>
<td>deletemin</td>
<td></td>
<td>(4,650)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Data structure operations and certification trail for MINSPAN

deletemin(h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly keep a set of edges representing the current tree. Implicitly, however, if (v, k) is returned by deletemin then prefer(v) is added to the current tree.

### 3.1.3 First execution of MINSPAN

In the first execution of the algorithm, the MINSPAN code is used and the principle data structure is implemented with a balanced search tree such as an AVL tree [1], a red-black tree [18], or a b-tree [9]. In addition, an array of pointers indexed from 1 to n is used. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. The
Figure 2: Example for minimum spanning tree algorithm.
Algorithm MINSPAN($G$, weight)

Input: Connected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ with edge weights.
Output: Spanning tree of $G$ which has minimum weight

1. CHOOSE root $\in V$
2. FOR ALL $u \in V$, key($u$) := $\infty$ END FOR
3. $h := \emptyset$; $v :=$ root
4. WHILE $v \neq$ empty DO
5. key($v$) := $-\infty$
6. FOR EACH $[v, w] \in E$ DO
7. IF weight($[v, w]$) < key($w$) THEN
8. key($w$) := weight($[v, w]$); prefer($w$) := $[v, w]$
9. IF member($w, h$) THEN changekey($w, key(w), h$)
10. ELSE insert($w, key(w), h$) END IF
11. END IF
12. END FOR
13. $(v, k) :=$ deletemin($h$)
14. END WHILE
15. FOR ALL $u \in V - \{\text{root}\}$, OUTPUT(prefer($u$)) END FOR
END MINSPAN

Figure 3: Code for MINSPAN Algorithm
array of pointers is initially all nil. For each item $i$, the $i$th pointer of the
array is used to point to the location of the ordered pair with item number $i$
in the balanced search tree. If there is no such ordered pair in the tree then
the $i$th pointer is nil. This array allows rapid execution of operations such
as $\text{member}(i, h)$ and $\text{delete}(i, h)$.

The certification trail is generated during the first execution as follows:
When $\text{CHOOSE } \text{root} \in V$ is executed in the first step, the vertex which
is chosen is output. Also, each time $\text{insert}(i, k, h)$ or $\text{changekey}(i, k, h)$ are
executed, $\text{predecessor}(i, h)$ is executed afterwards, and the answer returned
is output. This is illustrated in column labeled “Trail” in the table above.

### 3.1.4 Second execution of MINSPAN

The second execution of the algorithm also uses the MINSPAN code; how-
ever, the $\text{CHOOSE}$ construct and the data structure operations are imple-
mented differently than in the first execution. The $\text{CHOOSE}$ is performed
by simply reading the first element of the certification trail. This guarantees
the same choice of a starting vertex is made in both executions. Figure 4
depicts the principle data structure used which we call an $\text{indexed linked list}$.
The array is indexed from 1 to $n$ and contains pointers to a singly linked list
which represents the current contents of $h$. Each element in the list stores
an ordered pair in $h$ except the head of the list which contains the special
ordered pair $(0, -\text{INF})$. The list is organized such that a traversal from the
head gives the sorted ordering of the current contents of $h$ from smallest to
largest. The $i$th element of the array points to the node containing the or-
dered pair with the item number $i$ if it is present in $h$; otherwise, the pointer
is nil. The $0$th element of the array points to the node containing $(0, -\text{INF})$.
Initially, the array contains nil pointers except the $0$th element. We now
show how to implement the data structure operations.

To perform $\text{insert}(i, k, h)$, it is necessary to read the next value in the
certification trail. This value, say $j$, is the item number of the ordered pair
which is the predecessor of $(i, k)$ in the current contents of $h$. A new linked
list node is allocated and the trail information is used to insert the node into
the data structure. Specifically, the $j$th array pointer is traversed to a node
in the linked list, say $Y$. (If $j = \text{"smallest"}$ then the $0$th array pointer is
traversed.) The new node is inserted in the list just after node $Y$ and before
the next node in the linked list (if there is one). The data field in the new
node is set to \((i, k)\) and the \(i\)th pointer of the array is set to point to the new node. Figure 4 shows the insertion of \((7, 505)\) into the data structure given that the certification trail value is 6. Figure 3(a) is before the insertion and Figure 3(b) is after the insertion.

When the \(\text{insert}\) operation is performed, some checks must be conducted. First, the \(i\)th array pointer must be nil before the operation is performed. Second, the sorted order of the pairs stored in the linked list must be preserved after the operation. That is, if \((i', k')\) is stored in the node before \((i, k)\) in the linked list and \((i'', k'')\) is stored after \((i, k)\), then \((i', k') < (i, k) < (i'', k'')\) must hold in the total order. If either of these checks fails then execution halts and “error” is output.

To perform \(\text{delete}(i, h)\) the \(i\)th array pointer is traversed and the node found is deleted from the linked list. Next, the \(i\)th array pointer is set to nil. Figure 4 shows the deletion of item number 7 if one considers Figure 3(a) as depicting the data structure before the operation and Figure 3(b) depicting it afterwards. When the \(\text{delete}\) operation is performed one check is made. If the \(i\)th array pointer is nil before the operation then the execution halts and “error” is output.

To perform \(\text{changekey}(i, k, h)\) it suffices to perform \(\text{delete}(i, h)\) followed by \(\text{insert}(i, k, h)\). Note, this means the next item in the certification trail is read. Also, the checks associated with both these two operations are performed and the execution halts with “error” output if any check fails.

To perform \(\text{deletemin}(h)\) the 0th array pointer is traversed. to the head of the list and the next node in the list is accessed. If there is no such node then “empty” is returned and the operation is complete. Otherwise, suppose the node is \(Y\) and suppose it contains the ordered pair \((i, k)\), then the node \(Y\) is deleted from the list, the \(i\)th array pointer is set to nil, and \((i, k)\) is returned.

Lastly, to perform \(\text{member}(i, h)\) the \(i\)th array pointer is examined. If it is nil then false is returned, otherwise, true is returned. The \(\text{predecessor}(i, h)\) operation is not used in the second execution.

This completes the description of the second execution. To show that what we have described is a correct implementation of the certification trail method requires a proof. The proof has several parts of varying difficulty. First, one must show that if the first execution is fault-free then it outputs a minimum spanning tree. Second, one must show that if the first and second executions are fault-free then they both output the same minimum spanning
tree. Both these parts of the proof are not difficult to show.

The third more subtle part of the proof deals with the situation in which only the second execution is fault-free. This means an incorrect certification trail may be generated in the first execution. In this case, we must show that the second execution outputs either the correct minimum spanning tree or "error". The checks that were described above have been carefully designed to assure precisely this property by detecting any errors that would prevent the execution from generating the correct output. Because of space restrictions we will not give the proof here.

3.1.5 Time complexity comparisons of the two executions

In the first execution each data structure operation can be performed in $O(\log(n))$ time where $|V| = n$. There are at most $O(m)$ such operations and $O(m)$ additional time overhead where $|E| = m$. Thus, the first execution can be performed in $O(m \log(n))$. We note that this algorithm does not achieve the fastest known asymptotic time complexity which appears in [15]. However, the algorithm we have presented has a significantly smaller constant of proportionality which makes it competitive for reasonably sized graphs. In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail. It should be mentioned that we have devel-
oped a more complex certification trail solution for an asymptotically faster minimum spanning tree algorithm which uses fibonacci heaps.

In the second execution each data structure operation can be performed in \( O(1) \). There are still at most \( O(m) \) such operations and \( O(m) \) additional time overhead. Hence, the second execution can be performed in \( O(m) \) time. In other words, because of the availability of the certification trail, the second execution is performed in linear time. There are no known \( O(m) \) time algorithms for the minimum spanning tree problem. Komlós was able to show that \( O(m) \) comparisons suffice to find the minimum spanning tree. However, there is no known \( O(m) \) time algorithm to actually find and perform these comparisons. Even the related "verification" problem has no known linear time solution. In the verification problem the input consists of an edge weighted graph and a subtree. The output is "yes" if the subtree is the minimum spanning tree and "no" otherwise. The best known algorithm for this problem was created by Tarjan [45] and has the nonlinear time complexity of \( O(m \alpha(m, n)) \), where \( \alpha(m, n) \) is a functional inverse of Ackerman's function. The fact that the data in a certification trail enables a minimum spanning tree to be found in linear time is, we believe, intriguing, significant, and indicative of the great promise of the certification trail technique.

3.2 Huffman Tree Example

Huffman trees represent another classic algorithmic problem, one of the original solutions being attributed to Huffman [21]. This solution has been used extensively to perform data compression through the design and use of so-called Huffman codes. These codes are prefix codes which are based on the Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to be compressed. We will be concerned exclusively with the Huffman tree. See [21] for information about the coding application.

**Definition 3.3** The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) \( f[1], f[2], \ldots, f[n] \), construct a tree with \( n \) leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: \( \sum_{i \in \text{LEAF}} \text{len}(i)f[i] \) where \( \text{LEAF} \) is the set of leaves, \( \text{len}(i) \)
is the length of the path from the root of the tree to the leaf \( l_i \), \( f[i] \) is the frequency assigned to the leaf \( l_i \).

An example of a Huffman tree is given in Figure 6. The input frequencies are: \( f(1) = 35 \), \( f(2) = 20 \), \( f(3) = 44 \), \( f(4) = 77 \), \( f(5) = 23 \), \( f(6) = 38 \), and \( f(7) = 88 \). The frequencies appear inside the leaf nodes as the second elements of the ordered pairs in the figure.

### 3.2.1 Huffman algorithm

The algorithm to construct the Huffman tree uses a data structure which is able to implement the `insert` and the `deletemin` operations which were defined above in the minimum spanning tree example. This type of data structure is often called a priority queue. The algorithm also uses the command `allocate` to construct the tree. This command allocates a new node and returns a pointer to it. Each node is able to store an item number and a key value in the field called `info`. The item numbers are in the set \( \{1, \ldots, 2n - 1\} \) and the key values are sums of frequency values. The nodes also contain fields for left and right pointers since the tree being constructed is binary.

The Huffman tree is built from the bottom up and the overall structure of the algorithm is based on the greedy "merging" of subtrees. An array of pointers called `ptr` is used to point to the subtrees as they are constructed. Initially, \( n \) single vertex subtrees are constructed, with each one associated with one frequency number in the input. The algorithm repeatedly merges two subtrees with the smallest associated frequency values. To perform a merge a new subtree is created by first allocating a new root node and next setting the left and right pointers to the two subtrees being merged. The frequency associated with the new subtree is the sum of the frequencies of the two subtrees being merged. In Figure 6 the frequency associated with each subtree is shown as the second value in the root vertex of the subtree. Details of the algorithm are given below. Note that the priority queue data structure allows the algorithm to quickly determine which subtrees should be merged by enabling the two smallest frequency values to be found efficiently during each iteration.

The table below illustrates the data structure operations performed when the Huffman tree in Figure 6 is constructed. For conciseness the initial \( n \) insert operations have been omitted. The first column gives the set of
Algorithm HUFFMAN($FREQ$)
Input: Sequence of positive integers $FREQ = \{f[1], f[2], \ldots, f[n]\}$
Output: Pointer to a Huffman tree for the input frequencies
1 FOR $i := 1$ to $n$ DO
2 \hspace{1em} insert($i, f[i], h$)
3 \hspace{1em} ptr[$i$] := allocate()
4 \hspace{1em} info[ptr[$i$]] := $(i, f[i])$
5 END FOR.
6 FOR $j := n + 1$ to $2n - 1$ DO
7 \hspace{1em} (item1, key1) := deletemin($h$)
8 \hspace{1em} (item2, key2) := deletemin($h$)
9 \hspace{1em} ptr[$j$] := allocate()
10 \hspace{1em} info[ptr[$j$]] := $(j, key1 + key2)$
11 \hspace{1em} left[ptr[$j$]] := ptr[item1]
12 \hspace{1em} right[ptr[$j$]] := ptr[item2]
13 \hspace{1em} insert$(j, key1 + key2, h)$
14 END FOR.
15 OUTPUT(ptr[2n - 1])
END HUFFMAN

Figure 5: Code for HUFFMAN algorithm
ordered pairs in $h$. The second column gives the result of the two deleteMin operations during each iteration. Note that this column is labeled "Trail" because it is also output as the certification trail. The third column records the elements which are inserted by the command on line 13.

3.2.2 First execution of HUFFMAN

In this execution the code entitled HUFFMAN is used and the priority queue data structure is implemented with a heap [44] or a balanced search tree [18, 1, 9]. Actually, any correct implementation is acceptable; however, to achieve a reasonable time complexity for this execution the suggested implementations are desirable. The certification trail is generated as follows: whenever deleteMin($h$) is executed the item number and the key value which are returned are both output. In the table, the certification trail is listed in the second column.

3.2.3 Second execution of HUFFMAN

This execution consists of two parts which may be logically separated but which are performed together. In the first logical part, the code called HUFFMAN is executed again except that the data structure operations are treated differently. All insert operations are not performed and all deleteMin opera-
Set of Ordered Pairs
(2,20),(5,23),(1,35),(6,38),(3,44),(4,77),(7,88)
(1,35),(6,38),(8,43),(3,44),(4,77),(7,88)
(8,43),(3,44),(9,73),(4,77),(7,88)
(9,73),(4,77),(10,87),(7,88)
(10,87),(7,88),(11,150)
(11,150),(12,175)
(13,325)

Trail
(2,20),(5,23)
(1,35),(6,38)
(8,43),(3,44)
(9,73),(4,77)
(10,87),(7,88)
(9,73),(4,77)
(10,87),(7,88)
(11,150),(12,175)
(13,325)

Insert
(8,43)
(9,73)
(10,87)
(11,150)
(12,175)
(13,325)

Table 2: Data structure operations and certification trail for HUFFMAN

ditions are performed by simply reading the ordered pairs from the certification trail. In the second logical part, the data structure operations are “verified”. Note, by “verify” we do not mean a formal proof of correctness based on the text of an algorithm. The problem of verification can be formulated as follows: given a sequence of insert(i, k, h) and deleteMin(h) operations together with a sequence of answers for the deleteMin(h) operations check to see if the answers are correct. It should be noted that while in our example there is only one h, in general there can be multiple h’s to be handled.

The description of the algorithm for the second execution can be further simplified because only some restricted types of operation sequences are generated by the HUFFMAN code. First, it can be observed that all elements are ultimately deleted from h before the algorithm terminates; second, it can be further observed that when an element is inserted into h, its key value is larger than the key value of the last element deleted from h. These two important observations allow us to check a sequence using the simplified method which we describe next.

Our simplified method uses an array of integers indexed from 1 to 2n – 1. We use this array to track the contents of h. If the ordered pair (i, k) is in h, then array element i is set to a value of k; and if no ordered pair with item number i is in h, then array element i is set to a value of -1. Initially, all array elements are set to -1 and then operation sequence is processed. If insert(i, k) is executed then array element i is checked to see
if it contains \(-1\). (The value of \(-1\) is an arbitrary selection meant to serve only as an indicator.) If array element \(i\) does contain \(-1\), then it is set to \(k\). If \(deleteMin(h)\) is executed, then the answer indicated by the certification trail, say \((i, k)\), is examined. Array element \(i\) is checked to see if it contains \(k\). In addition, \(k\) is compared to the key value of previous element in the certification trail sequence to see if it is greater than or equal to that previous value. If both these checks succeed then array element \(i\) is set to \(-1\).

If any of the checks just described above fails, then the execution halts and “error” is output. Otherwise the operation sequence is considered “verified”. It can be rigorously shown that the checks described are sufficient for determining whether the answers given in the certification trail are correct; this proof, however, has been omitted for the sake of brevity. Finally, it is worth noting that to combine the two logical parts of this execution, one can perform the data structure checking in tandem with the code execution of HUFFMAN. Each time an insert or deleteMin is encountered in the code, the appropriate set of checks are performed.

3.2.4 Time complexity comparison of the two executions:

Again, as in the minimum spanning tree example, the availability of the certification trail permits the second execution for the Huffman tree problem to be dramatically more efficient than the first.

In the first execution of HUFFMAN, each data structure operation can be performed in \(O(\log(n))\) time where \(n\) is the number of frequencies in the input. There are \(O(n)\) such operations and \(O(n)\) additional time overhead, hence, the execution can be performed in \(O(n \log(n))\). This is the same complexity as the best known algorithm for constructing Huffman trees.

In the second code execution of HUFFMAN, each data structure operation is performed in constant time. Further, verifying the data structure operations are correct takes only a constant time per operation. Thus, it follows that the overall complexity of the second execution is only \(O(n)\).

3.3 Convex Hull Example

The convex hull problem is fundamental in computational geometry. Our certification trail solution to the generation of a convex hull is based on a solution due to Graham [16] which is called “Graham’s Scan.” (For basic
definitions and concepts in computational geometry, see the text of Preparata and Shamos[37].) For simplicity in the discussion which follows, we will assume the points are in so-called "general position," (that is, no three points are colinear). It is not difficult to remove this restriction.

Definition 3.4 A convex region in $R^2$ is a set of points, say $Q$, in $R^2$ such that for every pair of points in $Q$ the line segment connecting the points lies entirely within $Q$. A polygon is a circularly ordered set of line segments such that each line segment shares one of its endpoints with the preceding line segment and shares the other endpoint with the succeeding line segment in the ordering. The shared endpoints are called the vertices of the polygon. A polygon may also be specified by an ordering of its vertices. A convex polygon is a polygon which is the boundary of some convex region. The convex hull of a set of points, $S$, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in $S$. It is specified by a counterclockwise sequence of its vertices.

Figure 8(c) shows a convex hull for the points indicated by black dots. Graham's scan algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects an "extreme" point and calls it $p_1$. The next two steps sort the remaining points in a way which is depicted in Figure 8(a). It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although, in general, this polygon is not convex.

3.3.1 Graham's scan algorithm

It is possible to think of Graham's scan algorithm as removing points from this simple polygon until it becomes convex. The main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at Step 6 reveals that it is not on the convex hull because it falls within the triangle defined by three other points. A "snapshot" of the algorithm given in Figure 8(b) shows that $q_6$ is removed from the hull. The
Algorithm CONVEXHULL(S)
Input: Set of points, S, in $R^2$
Output: Counterclockwise sequence of points in $R^2$ which define convex hull of S
1 Let $p_1$ be the point with the largest x coordinate (and smallest y to break ties)
2 For each point $p$ (except $p_1$) calculate the slope of the line through $p_1$ and $p$
3 Sort the points (except $p_1$) from the smallest slope to the largest. Call them $p_2, \ldots, p_n$
4 $q_1 := p_1; \ q_2 := p_2; \ q_3 := p_3; \ m = 3$
5 FOR $k = 4$ to $n$ DO
6 \hspace{1em} WHILE the angle formed by $q_{m-1}, q_m, p_k$ is $\geq 180$ degrees DO $m := m - 1$ END FOR
7 \hspace{1em} $m := m + 1$
8 \hspace{1em} $q_m := p_k$
9 END FOR
10 FOR $i = 1$ to $m$ DO, OUTPUT($q_i$) END FOR
END CONVEXHULL

Figure 7: Graham’s scan algorithm

angle formed by $q_4, q_5, p_6$ is less than $180$ degrees. This means, $q_5$ lies within the triangle formed by $q_4, p_1, p_6$. (Note, $q_1 = p_1$.) In general, when the angle test is performed, if the angle formed by $q_{m-1}, q_m, p_k$ is less than $180$ degrees, then $q_m$ lies within the triangle formed by $q_{m-1}, p_1, p_k$. Below it will be revealed that this is the primary information relied on in our certification trail. When the main FOR loop is complete, the convex hull has been constructed.

3.3.2 First execution of Graham’s scan

In this execution the code CONVEXHULL is used. The certification trial is generated by adding an output statement within the WHILE loop. Specifically, if an angle of less than $180$ degrees is found in the WHILE loop test then the four tuple consisting of $q_m, q_{m-1}, p_1, p_k$ is output to the certification trail. The table below shows the four tuples of points that would be output by the algorithm when run on the example in Figure 8. The points in the table are given the same names as in Figure 8(a). The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in $R^2$. Instead, it consists of
Point not on convex hull

\[ P_6, P_4, P_7 \]

Three surrounding points

\[ P_3, P_1, P_6 \]

\[ P_6, P_1, P_8 \]

Table 3: First part of certification trail for Graham's scan

indices to the original input data. This means if the original data consists of \( s_1, s_2, \ldots, s_n \) then rather than output the element in \( R^2 \) corresponding to \( s_i \) the number \( i \) is output. It is not hard to code the program so that this is done.

3.3.3 Second execution for the convex hull problem

Let the certification trail consist of a set of four tuples, \( (x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r) \) followed by the supposed convex hull, \( q_1, q_2, \ldots, q_m \). The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

- First, the algorithm checks for \( i \in \{1, \ldots, r\} \) that \( x_i \) lies within the triangle defined by \( a_i, b_i, \) and \( c_i \).
• Second, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is less than or equal to 180 degrees.

• Third, it checks that there is a one to one correspondence between the input points and the points in \( \{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\} \).

• Fourth, it checks that for \( i \in \{1, \ldots, r\} \), \( a_i \), \( b_i \), and \( c_i \) are among the input points.

• Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local extreme point. We say a point \( q \) on the hull is a local extreme point if its predecessor in the counterclockwise ordering has a strictly smaller \( y \) coordinate and its successor in the ordering has a smaller or equal \( y \) coordinate.

If any of these checks fail then execution halts and “error” is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven. Because of space limitations we shall not give the proof here.

### 3.3.4 Time complexity of the two executions

In the first execution the sorting of the input points takes \( O(n \log(n)) \) time where \( n \) is the number of input points. One can show that this cost dominates and the overall complexity is \( O(n \log(n)) \).

It is possible to implement the second execution so that all five checks are done in \( O(n) \) time. Checking that a point lies within a triangle is a geometric calculation that can be done in constant time. Comparing the angle formed by three points to 180 degrees can be done in constant time. The third and fourth checks can be done in \( O(n) \) because the certification trail contains indices into the input data as described above. The uniqueness of the “local extreme” can also be checked in linear time.

It is important to note that, unlike the minimum spanning tree example and the Huffman tree example, the convex hull example utilizes an algorithm in the second execution that is not a close variant of that used in the first execution. However, like the previous two examples, the second execution
for the convex hull problem depends fundamentally on the information in the certification trail for efficiency and performance.

3.4 Concurrency of Executions

In the three examples discussed above, it is possible to start the second execution before the first execution has terminated. This is a highly desirable capability when additional hardware is available to run the second execution (for example, with multiprocessor machines, or machines with coprocessors or hardware monitors).

In the case of the minimum spanning tree problem, the two executions can be run concurrently. It is only necessary for the second execution to read the certification trail as it is generated - one item number at a time. Thus there is a slight time lag in the second execution. The case of the Huffman tree problem is similar. Both executions can be run concurrently if the second execution reads the certification trail as it is generated by the first execution.

The case of the convex hull problem is not quite as favorable, but it is still possible to partially overlap the two executions. For example, as each 4-tuple of points is generated by the first execution, it can be checked by the second execution. But the second execution must wait for the points on the convex hull to be output at the end of the first execution before they can be checked.

An additional opportunity for overlapping execution occurs when the system has a dedicated comparator. In this case it is sometimes possible for the two executions to send their output to the comparator as they generate it. For example, this can be done in the minimum spanning tree problem where the edges of the tree can be sent individually as they are discovered by both executions.

4 Comparison of Techniques

The certification trail approach to fault tolerance, whether implemented in hardware or software or some combination thereof, has resemblances with other fault tolerant techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions.
These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm or system uses the certification trail to indicate whether the execution of the primary system or algorithm was in error and/or to produce an output to be compared with that of the primary system.

To begin, the certification trail approach might be viewed as a form of N-version programming[12, 8] This approach specifies that N different implementations of an algorithm be independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary system to generate a trail of information while executing its algorithm that is critical to the secondary system's execution of its algorithm. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a null trail.

A software/hardware fault tolerance technique known as the recovery block approach [39, 2, 27] uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as a being structured into blocks of operations which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer, and many of the acceptance test that have been proposed for use tend to be somewhat straightforward [2]. Indeed, formal methodologies for the definition and generation of acceptance tests have thus far not been established. Regardless, the certification trail notion of a secondary system that receives the same input as the primary system and executes an algorithm that takes advantage of this trail to efficiently produce the correct output and/or to indicate that the execution of the first algorithm was correct does not fall into the category of an acceptance test.

A watchdog processor is a small and simple (relative to the primary system being monitored) hardware monitor that detects errors by examining information relative to the behavior of the primary system [32, 31, 34]. Error detection using a watchdog processor is a two-phase process: in the set-up phase, information about system behavior is provided apriori to the watchdog processor about the system to be monitored; in the monitoring phase,
the watchdog processor collects or is sent information about the operation of the system to be compared with that which was provided during the set-up phase. On the basis of this comparison, a decision is made by the watchdog processor as to whether or not an error has occurred. The information about system behavior by means of which a watchdog processor must monitor for errors includes memory access behavior [34], control and program flow [14, 22, 25, 28, 35, 36, 41, 43, 46, 47], or reasonableness of results [29, 30]. Using physical fault injection techniques, distributions of errors that could be detected using such types of information have been determined for some specific systems [40, 19], and the performance of models of error monitoring techniques that could be realized in the form of watchdog processors have been analyzed [10]. However, in contrast to the certification trail technique, a watchdog processor uses only apriori defined behavior checks, none of which is sufficient together with the input to the primary system to efficiently reproduce the output for direct comparison with that of the primary system.

Related to the watchdog processor approach is that of using executable assertions [3, 4, 29]. An assertion can be defined as an invariant relationship among variables of a process. In a program, for example, assertions can be written as logical statements and can be inserted into the code to signify that which has been predetermined to be invariably true at that point in the execution of the program. Assertions are based on apriori determined properties of the primary system or algorithm. This, however, again serves to distinguish executable assertion technique from the use of certification trails in that a certification trail is a key to the solution of a problem or the execution of an algorithm that can be utilized to efficiently and correctly produce the solution.

Algorithm-based fault tolerance [20, 33, 24] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique encodes data at a high level and algorithms are specifically designed or modified to operate on encoded data and produce encoded output data. Algorithm-based fault tolerance is distinguished from other fault tolerance techniques by three characteristics: the encoding of the data used by the algorithm; the modification of the algorithm to operate on the encoded data; and the distribution of the computation steps in the algorithm among computational units. It is assumed that at most one computational unit is faulty during a specified time period. The error detection capabilities of the algorithm-based fault tolerance approach are directly related to that of the
error correction encoding utilized. The certification trail approach does not require that the data to be executed be modified nor that the fundamental operations of the algorithm be changed to account for these modifications. Instead, only a trail indicative of aspects of the algorithm's operations must be generated by the algorithm. As seen from the above examples, the production of this trail does not burden the algorithm with a significant overhead. Moreover, any combination of computational errors can be handled.

Recently Blum and Kannan[11] have defined what they call a program checker. A program checker is an algorithm which checks the output of another algorithm for correctness and thus it is similar to an acceptance test in a recovery block. An example of a program checker is the algorithm developed by Tarjan[45] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree. The Blum and Kannan checker is actually more general than this because it is allowed to be probabilistic in a carefully specified way. There are two main differences between this approach and the certification trail approach. First, a program checker may call the algorithm it is checking a polynomial number of times. In our approach the algorithm being checked is run once. Second, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem. The certification trail approach is explicitly algorithm oriented. In other words, a specific algorithm for a problem is modified to output a certification trail. This trail sometimes allows the second execution to be faster than any known program checkers for the problem. This is the case for the minimum spanning tree problem.

Other hardware and software fault tolerance and error monitoring techniques have been proposed and studied that might be thought of as bearing some resemblance to the certification trail approach. Extensive summaries and descriptions of these techniques can be found in the literature [42, 5, 23, 31]. Examination of these techniques reveals, however, that in each case there are fundamental distinctions from the certification trail approach. In summary, the certification trail approach stands alone in its employment of secondary algorithms/systems for the computation of an output for comparison that because of the availability of the trail not only proceeds in a more efficient manner than that of the primary but also can indicate whether the execution of the primary algorithm was correct.
5 Concluding Discussion

We have presented a new, powerful fault tolerant computing technique called the certification trail approach. Our description of this technique has been only in terms of applications to software fault tolerance, but the certification trail approach can also be implemented with hardware. We have illustrated the certification trail technique by applying it to three fundamental, well-known algorithms: a minimum spanning tree algorithm, the Huffman tree algorithm, and a convex hull algorithm. It should be understood that the approach is in no way limited to these algorithms. Only because of space restrictions was this small sample of algorithms considered. Nevertheless, we believe that our consideration of these three algorithms gives insight into the significance and desirability of the approach. We have found several other algorithms to which our techniques apply including an algorithm for the shortest path problem and we believe the technique will be widely applicable. We have also examined the general problem of “verifying” data structure operations as discussed above and have proven results for additional data structures. These results are important because they allow the certification trail approach to be applied to any algorithm which uses one of these data structures.

In the three problems discussed an asymptotic speed up was achieved between the first execution and the second execution which was greater than any constant factor. We note, however, even if the speed up were only by a constant factor, it would still make sense to use the technique because execution time would be saved. We also note that the certification trail technique can be used in conjunction with other software fault tolerance techniques. For example, multiple algorithms can be developed which generate and read multiple (but different) certification trails. Further, these algorithms could be written by separate teams of individuals. A general architecture for the interaction of these algorithms is an important research topic. For example, a “cascade” of algorithms numbered from 1 to N could be designed such that algorithm i sends a certification trail to i + 1 which allows i + 1 to run faster than i. When errors are detected, other versions of algorithms can be invoked which may use an earlier certification trail or ignore it. The ideas developed in recovery blocks and N-version programming among others could be used as guidance in exploring such issues.
References


[23] Johnson, B., Design and analysis of fault tolerant digital systems
Addison-Wesley, Reading, MA, 1989.

[24] "Fault tolerant FFT networks," Dig. of the 1985 Fault Tolerant Com-
puting Symposium, June, 1985.


[26] Komlós, J., "Linear verification for spanning trees", Proceedings of the


[28] Lu, D., "Watchdog processor and structural integrity checking," IEEE

[29] Mahmood, A., Lu, D.J. and McCluskey, E.J., "Concurrent fault detec-
tion using a watchdog processor and assertions," Proc. 1983 Int. Test

error detection using a watchdog processor," Proc. 1985 Int. Test Conf.,

watchdog processors - a survey," IEEE Trans. on Computers, vol. 37,

watchdog processors", IEEE Trans. on Computers, vol. 37, pp. 160-174,

[33] Nair, V., and Abraham, J., "General linear codes for fault-tolerant ma-
trix operations on processor arrays," Dig. of the 1988 Fault Tolerant


