The Twentieth International Symposium on Fault-Tolerant Computing (1990)

Using Certification Trails to Achieve Software Fault Tolerance

Gregory F. Sullivan[1]
Gerald M. Masson[2]
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 it by applying it to the fundamental problem of finding a minimum spanning tree. 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. Because of space limitations we have ommited examples of our technique applied to the Huffman tree, and convex hull problems. These can be found in the full version of this paper.

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 [14, 22]; 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[8, 4] (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 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

[1] Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.

[2] Research parially supported by NASA Grant NSG 1442.

CH 2877-9/90/0000/0423/$01.00 – 1990 IEEE

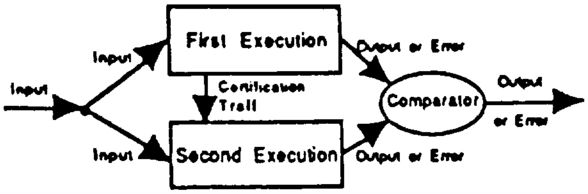

Figure 1: Certification trail method.

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 $\rightarrow$ S $\times$ T and $F_2 :$ D $\times$ T $\rightarrow$ S $\cup \{$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 $(d, s) \in$ P
(2) for all $d \in$ D and for all $t \in$ T
either ($F_2(d, t) = s$ and $(d, s) \in$ P) or
$F_2(d, t) =$ 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 is illustrated by the convex hull example in the full paper. Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. This is illustrated by the minimum spanning tree example 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. A more comprehensive comparison appears in the full paper.

3 Minimum Spanning Tree Example

In this section we illustrate the use of the certification trail method by applying it to the minimum spanning tree problem. Because of space limitations we have ommited other applications, e.g., to the Huffman tree and the convex hull problems. It should be stressed here that we believe the technique has wide applicability and these problems were chosen simply for illustration.

The minimum spanning tree problem has been examined extensively in the literature and an historical survey is given in [11]. Our certification trail approach is applied to a variant of the Prim/Dijkstra algorithm [19, 9] as explicated in [24]. 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.

3.0.1 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.

3.0.2 MINSPAN alg rithm

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 *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.

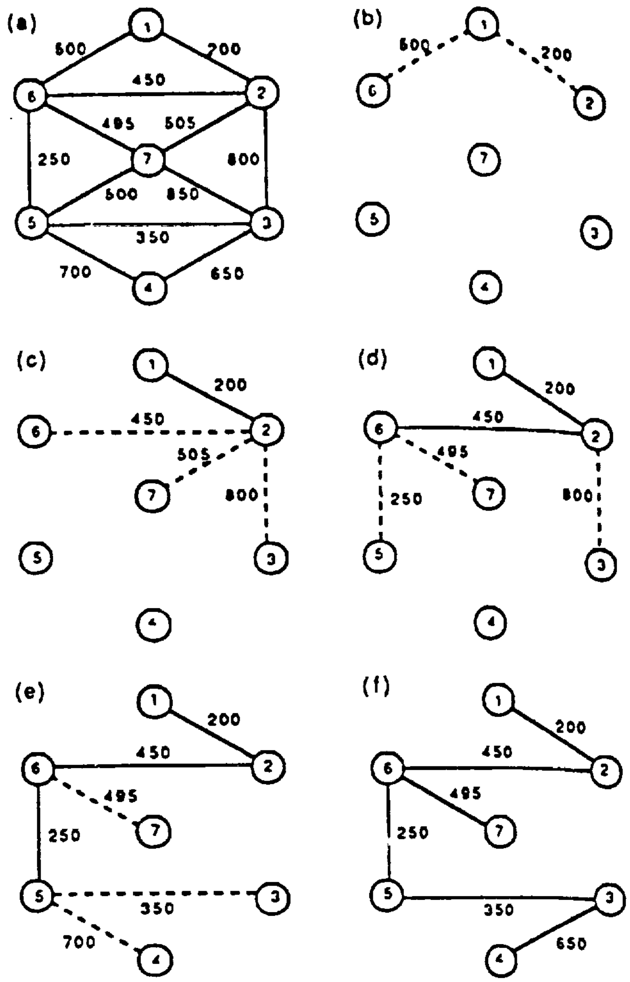

Figure 2: Example for minimum spanning tree algorithm.

3.0.3 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 [12], or a b-tree [5]. 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 array of pointers is initially all nil. For each item i, the ith pointer of the array is used to point to the lo-

```
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
CHOOSE $root \in V$
FOR ALL $u \in V$, $key(u) := \infty$ END FOR
$h := \emptyset$; $v := root$
WHILE $v \neq$ empty DO
  $key(v) := -\infty$
  FOR EACH $[v, w] \in E$ DO
    IF $weight([v, w]) < key(w)$ THEN
      $key(w) := weight([v, w])$; $prefer(w) := [v, w]$
      IF $member(w, h)$ THEN $changekey(w, key(w), h)$
        ELSE $insert(w, key(w), h)$ END IF
    END IF
  END FOR
  $(v, k) := deletemin(h)$
END WHILE
FOR ALL $u \in V - \{root\}$, OUTPUT($prefer(u)$)
END MINSPAN
```

Figure 3: Code for MINSPAN Algorithm

Operatio:	Set of Ordered Pairs	Trail
insert(2,200)	(2,200)	smallest
insert(6,500)	(2,200),(6,500)	2
deletemin	(6,500)	
insert(3,800)	(6,500),(3,300)	6
changekey(6,450)	(6,450),(3,800)	smallest
insert(7,505)	(6,450),(7,505),(3,800)	6
deletemin	(7,505),(3,800)	
insert(5,250)	(5,250),(7,505),(3,800)	smallest
changekey(7,495)	(5,250),(7,495),(3,800)	5
deletemin	(7,495),(3,800)	
changekey(3,350)	(3,350),(7,495)	smallest
insert(4,700)	(3,350),(7,495),(4,700)	7
deletemin	(7,495),(4,700)	
changekey(4,650)	(7,495),(4,650)	7
deletemin	(4,650)	
deletemin		
deletemin		

Table 1: Data structure operations and certification trail for MINSPAN

cation 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 ith pointer is nil. This array allows rapid execution of operations such as $member(i, h)$ and $delete(i, h)$.

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

3.0.4 Second execution of MINSPAN

The second execution of the algorithm also uses the MINSPAN code; however, the CHOOSE construct and the data structure operations are implemented differently than in the first execution. The 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 *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 ith element of the array points to the node containing the ordered pair with the item number i if it is present in h; otherwise, the pointer is nil. The 0th element of the array points to the node containing $(0, -\text{INF})$. Initially, the array contains nil pointers except the 0th element. We now show how to implement the data structure operations.

To perform $insert(i, k, h)$, 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 jth array pointer is traversed to a node in the linked list, say Y. (If j = "smallest" then the 0th 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 ith 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 *insert* operation is performed, some checks must be conducted. First, the ith 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 *delete*(i, h) the ith array pointer is traversed and the node found is deleted from the linked list. Next, the ith 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 *delete* operation is performed one check is made. If the ith array pointer is nil before the operation then the execution halts and "error" is output.

To perform *changekey*(i, k, h) it suffices to perform *delete*(i, h) followed by *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 *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 ith array pointer is set to nil, and (i, k) is returned.

Lastly, to perform *member*(i, h) the ith array pointer is examined. If it is nil then false is returned, otherwise, true is returned. The *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.

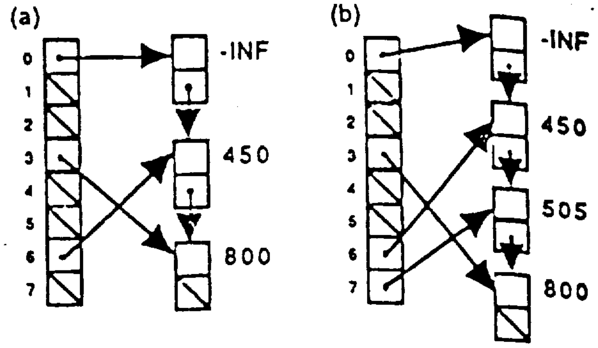

Figure 4: Example of the data structure used in the second execution of MINSPAN.

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.

3.0.5 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 [10]. 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 developed 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. Komlòs 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 ouput is "yes" if the subtree is the minimum spanning tree and "no" otherwise. The best known algorithm for this problem was created by Tarjan [25] 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.

3.1 Concurrency of Executions

In some cases, 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. This potential for concurrecy has been found in other problems we have examined, e.g., the Huffman tree problem.

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 there 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, 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, we compare the certification trail approach to N-version programming[8, 4]. 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 called the recovery block approach [20, 2, 17] 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 [2]. Indeed, formal methodologies for the definition and generation of acceptance tests have thus far not been fully 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.

Recently Blum and Kannan[7] 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[25] 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.

Space limitations preclude comparisons with the following other relevant techniques: watchdog processors [18, 6], algorithm based fault tolerance [13], executable assertions [3].

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 a minimum spanning tree algorithm. The full version of this paper includes applications to a Huffman tree algorithm, and a convex hull algorithm. It should be understood that the approach is in no way limited to these algorithms. We believe that our consideration of these 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 "certifying" 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 problem 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

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Andrews, D., "Using excutable assertions for testing and fault tolerance," *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 102-105, 1979, June 20-22.

[4] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[5] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[6] Blough, D., and Masson, G., "Performance analysis of a generalized concurrent error detection procedure," *IEEE Trans. on Computers* vol. 39, Jan., 1990.

[7] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[8] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[9] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[10] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica 6*, pp. 109-122, 2, 1986.

[11] Graham, R. L., and Hell, P., "On the history of the minimum spanning tree problem," *Ann. Hist. Comput.*, pp. 43-47, Jan., 1985.

[12] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[13] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

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

[15] Kane, J.R. and Yau, S.S., "Concurrent software fault detection," *IEEE Trans. Software Eng.* , vol. SE-1, pp. 87-99, March 1975.

[16] Komlòs, J., "Linear verification for spanning trees", *Proceedings of the 1984 Symposium on Foundations of Computing*, pp. 201-206, IEEE Computer Society Press, 1984.

[17] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[18] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors - a survey," *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[19] Prim, R. C., "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, pp. 1389-1401, Nov., 1957.

[20] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[21] Shen, J.P. and Schuette, M.A., "On-line self-monitoring using signatured instruction streams," *Proc. 1983 Int. Test Conf.*,, pp. 275-282, Oct., 1983.

[22] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[23] Sridhar, T. and Thatte, S.M., "Concurrent checking of program flow in VLSI processors," *Dig. 1982 Int. Test Conf.*, pp. 191-199, Nov., 1982.

[24] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[25] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[26] Tomas, S. P. and Shen, J. P., "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, pp.531-539, Oct., 1985.

[27] Yau, S.S, and Chen, F.-C., "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126-137, March 1980.

ORIGINAL PAGE IS
OF POOR QUALITY

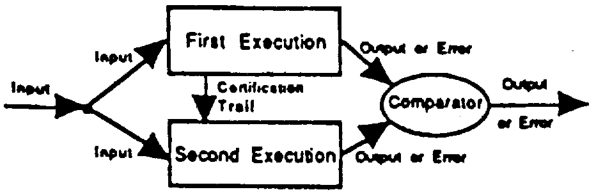

Figure 1: Certification trail method.

output such that $(d, s) \in \mathrm{P}$.

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

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

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error-detection capability of the certification-trail approach is similar 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.

Observant readers of our earlier paper [11] in which we introduced the notion of a certification trail might have noticed that our certification-trail solution for the min-spanning tree was generalizable. The generalized technique allows one to generate a certification trail for many algorithms which use a balanced binary tree data structure. However, the technique relies on the efficient execution of the predecessor operation and some data structures such as heaps cannot execute the predecessor operation efficiently. The techniques described in this paper are even more general and powerful, and they do apply to heaps.

The degree of diversity or independence achieved when using certification trails depends on how they are used. A fuller discussion of this and of the relationship between certification trails and other approaches to software fault tolerance is contained in the expanded version of [11]. This current paper presents asymptotic analysis which shows that the certification-trail approach is desirable even when the overhead of generating the certification-trail is included. We are currently working on an experimental analysis of the method and initial results are quite promising.

3 Answer-Validation Problem for Abstract Data Types

Our general approach to applying certification trails uses the concept of an abstract data type. Some examples of abstract data types are given later in this paper. Here we mention some important common properties and give a short illustration. Each abstract data type has a well defined data object or set of data objects, and each abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero), and some but not all operations return answers. An example of an abstract data type is a priority queue. The data object for a priority queue is an ordered pair of the form (i,k) where i is an item number and k is a key value. A priority queue has two operations: insert(i,k) and delmin. The insert operation has two arguments: item number i and key value k. The insert operation does not return an answer. The delmin operation has no arguments, but it does return an answer. The precise semantics of these operations are given later in this paper.

For each abstract data type we define an *answer-validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer-validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it. Examples of such inputs are given in the columns labelled "Operation" and "Answer" of table 1 and table 2.

The output for the answer-validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

Certification Trails for Data Structures

Gregory F. Sullivan[1]
Gerald M. Masson[2]
Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault detection and fault tolerance [11]. In this paper, we significantly generalize the applicability of the certification trail technique. Previously, certification trails had to be customized to each algorithm application, but here we develop trails appropriate to wide classes of algorithms. These certification trails are based on common data-structure operations such as those carried out using balanced binary trees and heaps. Any algorithm using these sets of operations can therefore employ the certification trail method to achieve software fault tolerance. To exemplify the scope of the generalization of the certification trail technique provided in this paper, constructions of trails for abstract data types such as priority queues and union-find structures will be given. These trails are applicable to any data-structure implementation of the abstract data type. It will also be shown that these ideas lead naturally to monitors for data-structure operations.

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

1 Introduction

In this paper we significantly generalize the novel and powerful certification-trail technique for achieving fault tolerance in systems that was introduced in [11]. Although applicable to both hardware and software, we restrict our discussion of the certification-trail technique in the following to software fault tolerance. To explain the essence of the certification-trail 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 [8, 10]; 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. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

The certification-trail technique is designed to obtain 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 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 an 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 generate a correct answer or signal that an error has been detected in the data 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, i.e., 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

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

434 — 439

The answer-validation problem is similar to the idea of an acceptance test which is used in the recovery-block approach [9, 2] to software fault tolerance. The main difference is that an answer-validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect answer will be detected at some point during the processing of the entire sequence. By allowing for this latency in detection, it is possible to create a much more efficient procedure for solving the answer-validation problem.

In this paper we shall solve the validation problem for two abstract data types. In the full version of this paper we solve the answer-validation problem for more general data types [12].

The most important aspect of the answer-validation problem is that it is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer-validation problem has a smaller time complexity than the original abstract-data-type problem. For example, to calculate the answers to a sequence of n priority-queue operations takes $\Omega(n \log(n))$ time, however it is possible to check the correctness of the answers in only $O(n)$ time. This speedup is very useful in fault-detection applications.

It is possible to run an answer-validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer-validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data-type operations a second time.

One possible application of the answer-validation problem occurs when it is used in conjunction with a repairable data structure which allows for repair but does not automatically attempt to detect faults [16]. Suppose an abstract data type is implemented with a repairable data structure. One can use an answer-validation procedure to detect errors in the answers generated by the abstract data type. When an error is detected, a repair of the data structure can be attempted. In some cases, recovery and continued execution will be possible.

In the next section, we will show how to create certification trails for programs which use abstract data types when those data types have efficient solutions for their answer-validation problems.

4 Schema for using Certification Trails

Suppose that we have developed an efficient solution to the answer-validation problem for some abstract data type. By efficient we mean the time complexity of the answer-validation problem is smaller than the time complexity of the original abstract-data-type problem. Further, suppose that we wish to run an algorithm, say A, which uses that abstract data type. To apply the certification trail method we can use the following schema to yield the two executions:

First Execution:

Execute algorithm A.
Each time an abstract-data-type operation is performed, append to the certification trail the identity of the operation, the arguments and the answer.

Second execution:

Phase One:
Validate the correctness of the operations and supposed answers given in the certification trail. If the validation returns "incorrect" or "ill-formed" then output "error" and stop. Otherwise, continue.

Phase Two:
Execute algorithm A.
Each time an abstract-data-type operation is performed, read the next entry in the certification trail. Make sure that the operation and the arguments in the certification trail agree with those requested in the algorithm. If not output "error" and stop. Otherwise, use the answer given in the certification trail and continue.

In the final step the outputs from the two executions are compared and the output is accepted or an error is signaled. This schema can yield execution times which are significantly faster than the execution time obtained by running algorithm A twice, yet these two methods give similar fault detection capabilities. 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. Note, the first execution can be slower than a simple execution of algorithm A since it must output a certification trail. However, the second execution can be significantly faster than a simple execution of the algorithm since the interactions with the abstract data type take less time overall. The net effect can be a major speedup.

Suppose an algorithm uses multiple abstract data types and suppose there are efficient answer-validation algorithms for each of these abstract data types. It is easy to see how our method generalizes. We can leave behind a generalized certification trail which consists of a separate certification trail for each of the abstract data types. The effect on the speedup of the second execution will be cumulative.

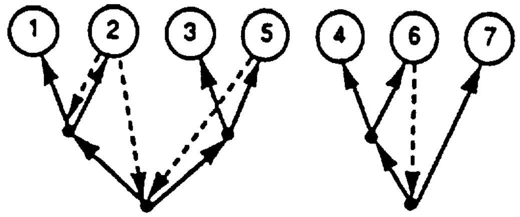

Figure 2: Union Tree and with Find Edges

5 Answer Validation for Disjoint-Set Union

As our first example we will discuss the disjoint-set union problem. This problem concerns a dynamic collection of sets in which pairs of sets can be combined to yield new sets. The underlying universe of set elements consists of the integers from 1 to n inclusive. Also, the universe of set names consists of the integers from 1 to n inclusive. There are three operations that can be performed:

create(A,x) creates a singleton set named A which contains element x. Since sets must be disjoint we require that x not already be in some set.

union(A,B) creates a new set which is the union of the sets named A and B. This new set is called A and the set named B becomes undefined. It is required that the sets named A and B are originally defined and are disjoint.

find(x) returns the name of the set which contains element x. It is required that x be a member of some unique set.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

In table 1 we give an example of a sequence of disjoint-set-union operations together with the answers for find operations. In addition, the collection of sets is depicted as it is changed by the operations. For simplicity, in this example each set name corresponds to the integer originally contained in the set when it is created. Sets are listed by first giving the name of the set followed by a colon and then the contents of the set.

Operation	Answer	Status of sets
create(1,1)		1:{1}
create(2,2)		1:{1},2:{2}
union(1,2)		1:{1,2}
find(2)	1	
create(3,3)		1:{1,2},3:{3}
create(4,4)		1:{1,2},3:{3},4:{4}
create(5,5)		1:{1,2},3:{3},4:{4},5:{5}
union(5,3)		1:{1,2},4:{4},5:{3,5}
union(5,1)		4:{4},5:{1,2,3,5}
find(2)	5	
find(5)	5	
create(6,6)		4:{4},5:{1,2,3,5},6:{6}
union(4,6)		4:{4,6},5:{1,2,3,5}
create(7,7)		4:{4,6},5:{1,2,3,5},7:{7}
union(4,7)		4:{4,6,7},5:{1,2,3,5}
find(6)	4	

Table 1: Sequence of operations for a Disjoint Set Union

The disjoint-set-union problem is a classic problem which has many applications [4] such as the off-line min problem, connected components, least-common ancestors, and equivalence of finite automata. Of particular interest is the time-complexity of performing a sequence of operations. Let us say the total number of operations is m, which is assumed to be greater than or equal to n. Recall, n is the number of set elements and set names.

Tarjan gave the tight upper bound of $O(m\alpha(m,n))$ [13, 14] for this problem. The α refers to the inverse of Ackermann's function which is a very slowly growing function. His solution and earlier solutions used a path-compression heuristic [15]. Fredman and Saks gave a lower bound of $\Omega(m\alpha(m,n))$ [5] in a general cell-probe model. Gabow and Tarjan show how to solve some important special cases of this problem in $O(m)$ time [6].

We now consider the answer-validation problem for the disjoint-set-union data type. We will show that this problem can be solved in $O(m)$ time where m is the number of operations. Note, this time complexity is superior to the complexity of actually performing the sequence of operations as discussed above. One method for solving this problem in $O(m)$ time uses the powerful techniques of Gabow and Tarjan [6]. However, we shall present a simpler method with a small constant of proportionality that is tailored to this problem.

To solve this problem we will build a forest based on the union operations in the sequence. In addition, we shall add edges to this forest based on the find operations. As a final step we will perform a traversal of the forest and perform appropriate checks. The solid edges in figure 2 indicate the forest we would build for the set of operations given in table 1. The dashed edges indicate the edges we would add to the forest based on the find operations.

Algorithm for Answer Validation for Disjoint-Set Union

Input: sequence of m operations together with arguments and supposed answers for the disjoint-set union data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Type *treenode* has fields left and right. Type *treeleaf* contains a list of pointers such that each pointer points to a treenode or a treeleaf. Array *activeset* is indexed by set name. Each array element is a pointer to a treenode or a treeleaf. Array *whereis* is indexed by an element number. Each array element is a pointer to a treeleaf. Initially, all pointers are nil and lists are null.

In the first phase of the algorithm we process each operation as it appears serially using the following rules:

create(A,x): If activeset[A] or whereis[x] are non-nil then output "ill-formed" and stop. Otherwise, allocate a treeleaf and set activeset[A] and whereis[x] to the allocated node.

union(A,B): If activeset[A] or activeset[B] are nil then output "ill-formed" and stop. Otherwise, allocate a treenode and set left to activeset[A] and right to activeset[B]. Next set activeset[A] to the treenode and set activeset[B] to nil.

find(x) A: (where A is the supposed answer to the find.) If whereis[x] is nil then output "ill-formed". Otherwise, whereis[x] points to some treeleaf. Call it tleaf. If activeset[A] is nil then output "ill-formed". Otherwise, activeset[A] points to some treeleaf or treenode. Call it t. Add a pointer to t to the list of pointers contained in treeleaf.

In the second phase of the algorithm we shall traverse the structure we have built.

Scan thru the array activeset to find non-nil pointers. It is not hard to see that each non-nil pointer points to the root of a tree made up of nodes of type tnode and tleaf. The tree uses the edges in the left and right fields of tnode.

For each such tree perform a depth-first search. Whenever the search reaches a node of type tleaf traverse the list of pointers that it contains. Check that each pointer points to a node which is currently on the stack which is used to perform the depth-first search. This is equivalent to checking that each pointer in tleaf points to a node which is an ancestor of tleaf in the tree.

If some pointer does not point to an ancestor then output "incorrect" and stop. Otherwise, output "correct" and stop.

Theorem 5.1 *The algorithm for answer validation of the disjoint-set-union abstract data type is correct.*

Theorem 5.2 *The answer validation algorithm for disjoint set union has a time complexity of $O(m)$ for processing a sequence of m operations.*

We omit these two theorems which overall are not difficult to show. We comment on one aspect of implementation. In the second phase of the answer validation algorithm it is necessary to determine if certain nodes are on the stack during the tree traversal. This can be done efficiently as follows: First, each treenode and each treeleaf can be assigned a unique identifier in the range 1 to m as it is allocated. Next, a boolean vector of size m indexed by the unique identifiers described above can be allocated. This vector can be used to keep track of which nodes are on the stack during tree traversal by turning bits on and off. This modified tree traversal algorithm still takes $O(m)$ time.

6 Generalized Priority Queue

We now describe a somewhat general abstract data type. We will solve the answer validation problem for restricted versions of this data type. 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 $i < i'$). The abstract data types we will consider support a subset of the following operations.

member(i) returns a boolean value of true if the set contains an ordered pair with item number i, otherwise returns false.

insert(i,k) adds the ordered pair (i,k) to the set. We require that no other pair with item number i be in the set.

delete(i) deletes the unique ordered pair with item number i from the set. We require that a pair with item number i be in the set initially.

changekey(i,k) is executed only when there is an ordered pair with item number i in the set. This pair is replaced by (i,k).

T	Operation	Answer	Validation stack
1	insert(6,300)		
2	insert(2,404)		
3	insert(3,250)		
4	deletemin	(3,250)	(3,250,4)
5	insert(10,248)		
6	insert(12,245)		
7	insert(4,260)		
8	deletemin	(12,245)	(12,245,8),(3,250,4)
9	insert(13,140)		
10	insert(5,142)		
11	deletemin	(13,140)	(13,140,11),(12,245,8),(3,250,4)
12	deletemin	(5,142)	(5,142,12),(12,245,8),(3,250,4)
13	deletemin	(10,248)	(10,248,13),(3,250,4)
14	deletemin	(4,260)	(4,260,14)

Table 2: Sequence of Priority Queue operations illustrating answer validation algorithm

deletemin (or deletemax) returns the ordered pair which is smallest (or largest) according to the total order defined above and deletes this pair. If the set is empty then the token "empty" is returned.

min (or max) returns the ordered pair which is smallest (or largest) according to the total order defined above. If the set is empty then the token "empty" is returned.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

Many different types and combinations of data structures can be used to support different subsets of these operations efficiently.

7 Answer Validation for Priority Queue

We will first consider the priority-queue abstract data type which allows only two operations: insert and deletemin. An example of a sequence of such operations appears in table 2. Many different data structures can be used to implement priority queues including heaps [17], balanced search trees such as AVL trees [1], red-black trees [7], or b-trees [3]. It is possible to process a sequence of $O(n)$ operations in $O(n\log(n))$ time using the data structures above. Furthermore, there is a lower bound of $\Omega(n\log(n))$ because it is possible to sort using a priority queue. Remarkably, the answer-validation problem can be solved using only $O(n)$ time, as documented below.

Each operation is time-stamped, i.e., the operations are assigned integers sequentially starting with 1 which is easy to do with a counter. The answer-validation algorithm uses a stack called deletestack. The contents of this stack are illustrated in table 2. The top of the stack is on the left in table 2.

Let us consider the kinds of tests that an answer-validation algorithm for a priority queue might perform. Suppose (i,k) is the answer to some deletemin operation. Further, suppose (i′,k′) was deleted in a previous deletemin operation. If the priority queue is correct then either (i,k)>(i′,k′) or (i′,k′) was deleted before (i,k) was inserted. This suggests that the time of insertion and deletion for elements should be recorded and the algorithm below does this. Unfortunately, if an algorithm compares an ordered pair which has been deleted against all previously deleted ordered pairs then the algorithm complexity is at least $O(m^2)$. To avoid this the deletestack is used. The deletestack was designed to allow many comparisons to be done implicitly and to reduce the complexity.

Algorithm for Answer Validation for Priority Queue

Input: sequence of $O(n)$ operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Array called *inserttime* indexed by item number. Array elements contain either "absent" or a time-stamp. Array called *keyvalue* indexed by item number. Array elements contain either "absent" or a key value. Initially, each element in these two arrays contains "absent". Stack of ordered triples called *deletestack*. Each ordered triple has the following form: first element is an item number, second element is a key value, and third element is a time-stamp. deletestack is initially empty.

In the first phase of the algorithm we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): If inserttime[i]≠"absent" then output "ill-formed" and stop. Otherwise, let inserttime[i] = currenttime and let keyvalue[i]=k.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) If inserttime[i]="absent" or keyvalue[i]≠k then output "ill-formed" and stop.

Otherwise, let (i′,k′) be the item number and key number of the triple on the top of deletestack (if there is one). Repeatedly pop the stack until (i,k)<(i′,k′) or until deletestack is empty.

If deletestack is empty then push the triple (i,k,currenttime) onto deletestack. Further, let inserttime[i]="absent" and let keyvalue[i]="absent" and process the next priority queue operation.

If deletestack is non-empty then let the top element be (i',k',deletetime'). If inserttime[i]<deletetime' then output "incorrect" and stop. Otherwise, push the triple (i,k,currenttime) onto deletestack. Next, let inserttime[i]="absent" and let keyvalue[i]="absent" and process the next priority queue operation.

In the second phase of the algorithm we operate on the items which have been inserted but have never been deleted.

Scan the array inserttime and for each item number for which inserttime[i]≠"absent" construct an ordered triple (i,keyvalue[i],inserttime[i]). Call this set of ordered triples remainders.

Use a bucket sort to sort the triples in remainders by their time-stamps, i.e., the third element of the ordered triple.

Merge the triples in remainders together with the triples in deletestack so that they are all ordered by their time-stamps, i.e., the third element of the ordered triple.

Scan the combined triples to determine if there exist two triples which satisfy the following: inserttime[i]< deletetime' and (i,keyvalue[i])<(i',k'); where one triple is from remainders and has the form (i,keyvalue[i], inserttime[i]) and where the other triple is from deletestack and has the form (i',k',deletetime');

If these two triples exist then output "incorrect" and stop. Otherwise output "correct" and stop.

Theorem 7.1 *The algorithm for answer validation of the priority queue abstract data type is correct.*

Proof: Clearly the algorithm for answer validation always terminates. We must show that the algorithm outputs "correct" iff the operations together with arguments and supposed answers are correct. Because of space limitations we will only give a proof for the more difficult half of this iff statement. We shall use a proof by contradiction. Assume that the sequence of operations, arguments and supposed answers is considered correct by the algorithm but actually is incorrect. The use of the array inserttime and the symbol "absent" assures that no item is deleted when it is absent or inserted when it is already present. The use of the array keyvalue assures that items do not change keyvalue when they are present in the data type set. There is only one remaining way in which a sequence can be incorrect. This occurs when an ordered pair is deleted by a deletemin operation, however, it does not really have the smallest key value.

This means, there exist ordered pairs (i_1,k_1) and (i_2,k_2) such that $(i_1,k_1)>(i_2,k_2)$ and (i_1,k_1) is deleted while (i_2,k_2) is present in the data type set. In addition, we may specify that (i_1,k_1) is the largest ordered pair deleted while (i_2,k_2) is present. Let ins_1 be the time that i_1 was inserted and let del_1 be the time that i_1 was deleted. Let ins_2 be the time that i_2 was inserted and let del_2 be the time that i_2 was deleted (if it was deleted). There are two cases.

Case 1: Suppose that (i_2,k_2) is ultimately deleted. We know that $(i_1,k_1)>(i_2,k_2)$ by assumption. $del_2>del_1$ since item i_2 is deleted afte. item i_1. $ins_2<del_1$ since item i_2 was present when item i_1 was deleted.

Consider the situation when item i_2 is deleted with a deletemin operation. The ordered triple for item i_1 must appear in deletestack just before the processing of the i_2 deletion operation. This follows because the triple for item i_1 can only be removed from deletestack by a larger element and yet (i_1,k_1) refers to the largest ordered pair deleted while (i_2,k_2) was present. Now, since $(i_1,k_1)>(i_2,k_2)$ the ordered triple for item i_1 will remain in deletestack even after deletestack is popped during the processing of the deletemin operation for item i_2. Suppose the top of deletestack is (i_3,k_3,del_3) after the popping.

It is easy to show that the time-stamps on deletestack are monotonically ordered with the largest time-stamp at the top. For this reason we know that $del_3 \geq del_1$. We noted earlier that $del_1>ins_2$. But if $ins_2<del_3$ then the algorithm outputs "incorrect" when it processes the deletemin operation. This contradicts our assumption that the sequence of operations, arguments and supposed answers was considered correct by the algorithm.

Case 2: Suppose the ordered pair (i_2,k_2) is never deleted. In the second phase of the algorithm the ordered triple (i_2,k_2,ins_2) is constructed and is compared against the ordered triples in deletestack.

The same argument that was used in case 1 above can be used to show that the test performed in the second phase of the algorithm would detect a problem and cause "incorrect" to be output. This contradicts our assumption that the sequence of operations, arguments and supposed answers was considered correct by the algo:ithm. Since both cases lead to a contradiction our proof is complete. ∎

Theorem 7.2 *The answer validation algorithm for priority queue has a time complexity of $O(n)$ for processing a sequence of $O(n)$ operations.*

Proof: We first analyze phase one of the algorithm. Note, there is a constant amount of work done for processing each single operation if we exclude the cost of popping the deletestack. Interestingly, popping the deletestack can take $O(n)$ time for the processing of a single operation. Luckily, the total amortized complexity for popping the deletestack while processing a sequence of $O(n)$ operations is still only $O(n)$. This is true because each item which is inserted and later deleted is placed on deletestack and is popped at most once.

We now consider phase two. The cost of array scanning and constructing the triples is $O(n)$. The cost of the bucket sort is $O(n)$ and the cost of the merge is also $O(n)$. The final test can be implemented with a simple scan with a complexity of $O(n)$. Hence the overall complexity is $O(n)$ ∎

We have solved the answer-validation problem for abstract data structures that support the following set of operations: member, insert, delete, deletemin, min, deletemax, and max. The algorithm used to solve this problem is intricate but efficient. It requires only $O(n)$ time to process $O(n)$ operations. A detailed description of our solution, however, is beyond the scope of this version of the paper.

8 Conclusions

The results reported in this paper significantly generalize the applicability of the certification-trail technique. In our previously reported work on certification trails [11], we had to customize each algorithm application, but we have now developed trails appropriate to wide classes of algorithms. These certification trails are based on common data-structure operations such as those carried out using balanced binary trees and heaps. Any algorithm using these sets of operations can therefore employ the certification trail method to achieve software fault tolerance. To express the full generality of these ideas, we have provided constructions of trails for abstract data types such as priority queues and union-find structures. These trails are applicable to any data-structure implementation of the abstract data type. These ideas lead naturally to monitors for data-structure operations. We are currently working on an experimental evaluation of the approach and initial results are promising.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[4] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[5] Fredman, M. L., and Saks, M. E., "The cell probe complexity of dynamic data structures," *Proc. 21st ACM Symp. on Theo. Comp. 1989*, pp. 109-122, 2, 1986.

[6] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[7] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

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

[9] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[10] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[11] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[12] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Department of Computer Science Technical Report JHU 90/17*, Johns Hopkins University, Baltimore, Maryland, 1990.

[13] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm," *J. ACM*, 22(2), pp. 215-225, 1975.

[14] Tarjan, R. E., "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. of Comp. and Sys. Sci.*, 18(2), pp. 110-127, 1979.

[15] Tarjan, R. E., and Leeuwen, J. van, "Worst-case analysis of set union algorithms," *J. ACM*, 31(2), pp. 245-281, 1984.

[16] Taylor, D., "Error Models for robust data structures," *Dig. 20th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 416-422, 1990 June 26-28.

[17] Williams, J. W. J, "Algorithm 232 (heapsort)," *Commun. of ACM*, vol.7, pp. 347-348, 1964.