178-71
P-6

Experimental Evaluation of Certification Trails using Abstract Data Type Validation

Dwight S. Wilson[1]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gregory F. Sullivan[2]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gerald M. Masson[3]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [11, 12]. Recent experimental work [13] reveals many cases in which a certification-trail approach allows for significantly faster program execution time than a basic time-redundancy approach. Algorithms for answer-validation of abstract data types are presented in [12] and allow a certification trail approach to be used for a wide variety of problems. In this paper, we report on an attempt to assess the performance of algorithms utilising certification trails on abstract data types. Specifically we have applied this method to the following problems: heapsort, Huffman tree, shortest path, and skyline. Previous results used certification trails specific to a particular problem and implementation. The approach in this paper allows certification trails to be localized to "data structure modules," making the use of this technique transparent to the user of such modules.

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

1 Introduction

To explain the essence of the certification trail technique for software fault tolerance, we first discuss 2-version programming [4, 2]. Using 2-version (or more generally, N-version) programming, two (or N) implementations of an algorithm are executed on a given input, and the results compared. If the outputs agree, they are accepted, otherwise an error is flagged. This technique will detect a variety of software faults as well as transient hardware faults. A variation of this technique is to execute a single program twice and compare results, this is called time redundancy. Although there are a few software faults that may be detected using time redundancy (e.g., uninitialised pointer errors), it is more effective in catching transient faults.

The certification trail technique is designed to achieve similar types of error detection capabilities but expend fewer resources. The central idea, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. The second algorithm may then make use of this data, which is chosen so that the algorithm executes more quickly and/or has 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 certification trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It 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.

Early work on the certification trail focused on creating trails for specific implementation of problems. For example the trail given in [11] for the convex hull problem is specific to the Graham scan algorithm. In general, the two algorithms used in this approach can be quite different. A more recent approach is to construct a certification trail for an abstract data type. That is, given the answers to operations allowed on that type, our algorithm checks the correctness of these answers. This method has the advantage that the certification trail techniques are localized to the

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

outines implementing data structure operations, and nay then be applied to a wide variety of problems without special coding. In many cases it may be possible to use existing code with only minor modifications. Code using these routines is run twice, the first ime generating the trail, the second time using it. Alternately, the trail checking may be done, in parallel, .e., we perform the checking as the trail is being genrated. A programmer using a library of these routines need not be familiar with certification trail techniques. Object oriented programming techniques may be paricularly useful for implementation of such "certified" data types.

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 realsations and uses.

Definition 2.1 A problem P is formalised 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 $d \in \mathbf{D}$ when d is input to A then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ 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 $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. T is the set of *certification trails*. The functions must satisfy the following two properties:

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

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. 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). In addition, 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 may 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.

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.

The answer validation problem is similar to the idea

of an acceptance test which is used in the recovery block approach [10] 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 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.

The most important aspect of the answer validation problem is the fact that is 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 in the decision tree model; however, it is possible to check the correctness of the answers in only $O(n)$ time [12]. This speed 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 twice.

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.

This schema can yield execution times which are significantly faster than the execution time obtained by running algorithm A twice. Yet the schemes yield comparable fault detection capabilities. 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 yield a major speed-up.

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 generalised certification trail which consists of a seperate certification trail for each of the abstract data types. The effect on the speed up of the second execution will be cumulative.

5 Generalized Priority Queue

We now describe a somewhat general abstract data type. We are able to 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).

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

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

max and deletemax these operations are similar to min and deletemin, using the largest element instead of the smallest one.

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. Specifically we are interested in allowing the insert, delete, min, and deletemin operations. It is possible to process a sequence of $O(n)$ operations in $O(n\log(n))$ with implementations using heaps or balanced search trees such as AVL trees [1], red-black trees [6] or b-trees [3]. Answer validation of these operations can be performed in $O(n)$ time [12, 13].

6 Examples of the use of Data Structure Certification

In this section we evaluate the use of certification trails for data structures as applied to four well-known and significant problems in computer science: sorting, the shortest path tree problem, the Huffman tree problem, and the skyline problem. We have implemented basic algorithms for these problems and algorithms which generate and use certification trails. Timing data was collected using a SPARCstation ELC.

The timing information reported in the tables consists of the run time of the basic algorithm (i.e., no certification trail), the run time of the trail-generating algorithm, the run time of the trail-using algorithm, the percentage savings of using certification trails, and the speedup achieved by the second phase of the certification trail method. The percentage savings is computed by comparing the total run time of algorithms for generating and using trails against twice the run time of the basic algorithm. The speedup is computed by dividing the run time of the basic algorithm by the run time of the algorithm that uses the certification trail.

Apart from the data structures, the implementation of both phases of the certification trail version of each algorithm is nearly identical to the implementation of the basic version. The only difference in the code for the two phases is a parameter passed to the data structure code indicating whether a certification trail should be generated or used. All code implementing the certification trails is localised to the modules implementing the data structures, allowing the generation and use of the trail to be transparent to the user of these modules. Due to space constraints only an abbreviated discussion of the algorithms is given.

6.1 Heapsort

Sorting is a fundamental operation in computer systems, and there exist several sorting algorithms. Sorting may be implemented with a priority queue (or more specifically, a heap) by inserting all elements and performing deletemin operations until the queue is empty.

Input data was generated by creating sets of integers chosen uniformly from the interval $[0, 10000000]$. Timing results are based on fifty executions at each input size.

6.2 Huffman Tree

Given a sequence of frequencies (positive integers), we wish to construct a Huffman tree, i.e., a binary tree with frequencies assigned to the leaves, such that the sum of the weighted path lengths is minimised. This is a classic algorithmic problem and one of the original solutions was found by Huffman [7]. It has been used

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
10000	0.44	0.45	0.11	36.36	4.00
20000	0.98	1.00	0.23	37.24	4.26
50000	2.71	2.80	0.60	37.27	4.52
100000	5.87	6.05	1.23	37.99	4.77
200000	12.71	13.01	2.47	39.50	5.15
500000	19.67	20.25	3.73	39.04	5.27

Table 1: Heapsort

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
5000	0.38	0.41	0.14	27.63	2.71
10000	0.83	0.87	0.29	30.13	2.86
20000	1.79	1.90	0.61	29.89	2.93
50000	4.93	5.30	1.53	30.73	3.22
100000	10.75	11.47	3.13	32.14	3.45
150000	16.70	17.87	4.66	32.54	3.58

Table 2: Huffman Tree

extensively in data compression algorithms through the design and use of so called Huffman codes. The tree structure and code design are based on frequencies of individual characters in the data to be compressed. In this paper we are concerned only with the Huffman tree, the interested reader should consult [7] for information about the coding application.

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, ptr, is used to point to the subtrees as they are constructed. Initially, n single vertex subtrees are constructed, each one associated with a frequency number in the input. The algorithm repeatedly merges the two subtrees with the smallest associated frequency values, assigning the sum of these frequencies to the resulting tree. A priority queue data structure allows the algorithm to quickly find the subtrees to merge at each step.

Data for the timing experiments was generated by choosing integer frequencies uniformly from the range $[0, 100000]$. Timing results are based on fifty executions for each input size.

6.3 Shortest Path

Given a graph with non-negative edge weights and a source vertex, we wish to find the shortest paths from the source vertex to each of the other vertices. This is another classic problem and has been examined extensively in the literature. Our approach is applied to Dijkstra's algorithm.

Dijkstra's algorithm is a greedy algorithm. At each step, there exists a set of vertices S to which shortest paths are known, and a set T of vertices adjacent to members of this set. The best paths known to members of T are examined, and the vertex v, with the minimum path length is removed from T and added to S. A data structure that supports insert, delete, and deletemin can be used to implement this algorithm.

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
250,2500	0.15	0.14	0.06	33.33	2.50
500,5000	0.35	0.32	0.13	35.71	2.69
750,7500	0.56	0.52	0.19	36.61	2.95
1000,10000	0.79	0.73	0.25	37.97	3.16
2000,20000	1.74	1.65	0.52	37.64	3.35
2500,25000	2.22	2.08	0.65	38.51	3.42

Table 3: Shortest Path

Input graphs of $|V|$ vertices and $|E|$ edges were generated by choosing a set of $|E|$ distinct edges uniformly from all possible such sets, then rejecting graphs that were not connected. $|E|$ was chosen sufficiently large that each selection is connected with high probability, resulting in few rejections. The input sizes were chosen to keep the ration $|E|/|V|$ constant, for in practice the running time of the algorithm is affected by this ratio. Timing results are based on fifty executions at each input size. The size column of Table 3 contains an ordered pair indicating the number of vertices and edges.

6.4 Skyline

Given a set of rectangles with with collinear bottom edges, the *skyline* is the figure resulting from removing all hidden edges. The problem of computing the skyline of a set of rectangular buildings by eliminating hidden lines is discussed in [8]. The method used is divide and conquer and it constructs a skyline in $O(n \log(n))$ time. In this paper we use a plane sweep algorithm that can be easily implemented in terms of operations on priority queues. Plane sweep algorithms are widely used for computational geometry problems [9], and typically use a priority queue for event scheduling, and may be amenable to use of certification trail techniques.

Using a plane sweep algorithm, we compute the skyline as follows. Initialize a vertical sweep line to the left of all the rectangles (we may assume that all rectangle are to the right of the y-axis). As we sweep the line to the right we maintain a collection of the heights of the rectangles encountered. For each rectangle R, the height of R is added to the collection when we encounter R's left edge and removed when we encounter its right edge. The height of the skyline at any point x_0, is the maximum height in the collection when the sweepline is at $x = x_0$. Details are given below. A structure supporting insert and deletemin is

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
1000	0.35	0.27	0.11	24.00	2.27
2000	0.56	0.59	0.22	27.68	2.55
5000	1.71	1.79	0.58	30.70	2.95
10000	3.86	4.01	1.17	32.90	3.30
20000	8.30	8.76	2.36	33.73	3.56
30000	13.20	14.02	3.55	33.90	3.74

Table 4: Skyline

all that is needed to order the events, and a structure supporting insert, max, and delete is required to store the rectangle heights. A priority queue (supporting insert and can be used to order the sweepline events, and a generalised priority queue to store the rectangle heights.

Input data was generated by choosing integral rectangle heights uniformly over the range $[0, 100000]$. The x-coordinates of the left edges were chosen uniformly over the range $[0, 90000]$ and the width of each rectangle was chosen uniformly over the range $[1, 10000]$. Timing results are based on twenty executions for each input size.

7 Conclusions

The experimental data in this paper shows the utility of the certification trail approach using abstract data types. This paper supplements [13] which provides experimental data illustrating the advantages of implementation specific certification trails over classical time redundancy. We have shown that the more general approach of checking abstract data types also provides performance superior to classical time redundancy. This is significant because a wide range of algorithms may be represented as a sequence of operations on abstract data types. The certification trail approach may therefore be used on these programs, without requiring per problem "ad hoc" techniques. Creation of library routines or class libraries for these data types allows the certification trail technique to be used transparently, and may allow it's use with only minor modifications of existing code.

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

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

[4] Chen, L., and Avisienis 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.

[5] 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.

[6] 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.

[7] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[8] Manber U., *Introduction to Algorithms: A Creative Approach* Addison-Wesley, Reading, MA, 1989.

[9] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

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

[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," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[13] Sullivan, G.F., Wilson, D.S., Masson, G.M., Itoh, M., Smith, W.S., Kay, J.S., "Experimental evaluation of the certification trail method," Technical Report, Computer Science Department, The Johns Hopkins University