On Proving Sequential Machine Designs
S. KAMAL ABDALI, MEMBER, IEEE

Abstract—Based on a program-proving technique, a method for proving sequential machine designs is presented. The method associates with each state of the machine an assertion about the sequences taking the machine into that state. The design is proved by: showing all assertions true irrespective of state changes; and deriving the design specifications from the assertions at final states. The method is illustrated on two finite automata.

Index Terms—Checking experiments, fault detection, finite automata, method of assertions, program proving, proof of designs, sequential machine design, switching theory.

INTRODUCTION

Recently, much effort has been devoted to proving the correctness of programs from the problem, program, and processor definitions [5]. It appears that program-proving techniques can also be adapted for proving the design of logical networks, machines, and systems in general. To be specific, we shall restrict our discussion here to sequential machines and describe one method for demonstrating that a certain sequential machine behaves in a certain specified manner.

If a given physical specimen of a sequential machine does not respond to some input in the manner predicted from the machine design specifications, then either the specimen is not a faithful realization of the logical design, or the design itself is faulty. The detection of the former type of errors by experiments has been treated widely, notably by Hennie [3]. Not only can such errors be detected, but a machine specimen can be conclusively shown to be free of such errors by demonstrating that it responds correctly on being submitted to some suitably constructed “checking experiment.” The case of the latter type of errors, the errors of design, is very different. In practice, experiments are also relied upon to detect design errors; however, for all but very simple machines, some uncertainty about the correctness of design seems to persist even after a reasonably thorough amount of experimental verification. A similar situation prevails in computer programming, where program errors may remain undiscovered, notwithstanding most careful debugging. Proving a machine design or program logic correct, on the other hand, positively establishes the absence of errors—a practically impossible thing to do, in most cases, by testing alone.

METHOD OF ASSERTIONS

A most successful program-proving method is that of proving assertions, originated by Naur [8] and Floyd [1], and extensively applied by, among others, London [6], [7], references in [5]. Automatic program-proving programs based on this technique have been written by King [4] and Good [2]. In this method the program is envisaged as a graph (flow chart), with control operations at each node. Assertions are associated with each branch, with desired results asserted on branches leading to program termination nodes. (In later versions of the method assertions need be associated only with certain key branches, there being at least one assertion in each loop.) The proof of the program correctness then consists in proving the assertions by showing that the truth of the assertions on the incoming branches to a node, together with the control operations at that node, imply the truth of the assertions at the outgoing branches.

The method of assertions is very naturally suited to proving the correctness of sequential machines. In fact, for the machines cited in this note, it amounts simply to induction on the length of input sequences. Allowing more general transition functions, such as those involving inputs of length zero or greater than one, it would be more appropriate to regard the following procedure for proving assertions as induction on state changes. Before illustrating its use, however, it should be pointed out that other methods are also available for proving automata correct. For example, one could obtain the regular expression form of the set of acceptable sequences of an automaton, and show that it is precisely the set specified to be accepted. However, except for very small machines, generating the regular expression from the state diagram of an automaton is rather cumbersome, and so is the proof of equivalence of differing regular expressions. Furthermore, some properties of sequences are not conveniently describable in regular expression notation, e.g., the property of having the number of 0’s equal to a multiple of n.

The assertion method for proving an automaton design correct consists of the following steps.

Step 1: With each state of the automaton, associate some assertion whose truth is claimed for all input sequences that take the automaton into that particular state.

Step 2: Prove the following propositions.

a) The assertion associated with the initial state is true for the empty sequence.

b) The truth of the assertions is preserved by all state
transitions. That is, for each transition of the automaton from a state \( A \) to a state \( B \) or some input symbol \( a \), prove that if the assertion at \( A \) is true for a sequence \( x \), then the assertion at \( B \) must be true for the sequence \( xa \).

Step 3: Prove that the assertions associated with the final states are true for precisely those sequences that the automaton is specified to accept, and that the assertions associated with the nonfinal states are not true for any such sequences.

By Steps 2a and b, it follows by induction that the assertion at each state holds true, no matter by which sequence of inputs the automaton is taken into that state. Hence, by Step 3, the design must be correct.

Step 1 is the main creative part of a design verification, the other steps being potentially mechanizable. This step of producing assertions resembles the process of design itself, dependent on the problem at hand and the skill and experience of the design prover. One can usually formulate assertions for small machines by inspecting the state diagram, possibly by looking at some examples of sequences for each state and discovering patterns in them. At present, there does not seem to be any systematic way of arriving at suitable assertions, that is, ones for which the proofs of the mutual implications of assertions (Step 2) and the implication of verbal specifications (Step 3) may be obtainable in a simple manner. (Regular expressions, for example, are hardly suitable in this sense.)

**Examples**

We present two simple automata and prove the correctness of their design from their specifications. The automata are represented by state diagrams, in which an unlabeled arrow points to the initial state, and the final states are doubly circled. \( R_A \) designates any sequence taking the automaton into state \( A \), and \( \lambda \) stands for the empty sequence. We restrict our consideration to an input alphabet with only two symbols, 0 and 1.

**Example 1:** The automaton \( M_1 \) of Fig. 1 accepts a sequence if and only if the sequence is the binary representation of some integer of the form \( 5k+1 \), where lower order digits appear first in the sequence. Thus, \( M_1 \) is to accept such sequences as 1, 1000, 011, 00001 but not 01, 11, 001, 101, 111.

In order to express our assertions, we first need some notation. For a sequence

\[
x = a_0 a_1 a_2 \cdots a_n
\]

where \( a_i \) are input symbols, let the length of \( x \),

\[
l(x) = n + 1
\]

and the value represented by \( x \),

\[
v(x) = \sum_{i=0}^{n} a_i 2^i.
\]

Then, for a sequence \( x \), we have

\[
l(x0) = l(x1) = l(x) + 1
\]

\[
v(x0) = v(x)
\]

\[
v(x1) = v(x) + 2l(x).
\]

We define

\[
l(\lambda) = v(\lambda) = 0
\]

which is consistent with the above equations and the property of \( \lambda \) that

\[
\lambda x = x \lambda = x.
\]

Our assertions for \( M_1 \) are compactly shown in Table I. For each state \( S \), a column of the table indicates what \( \sigma(R_S) \) should be for a given \( l(R_S) \). Thus, the column under \( A \) is an abbreviation for the following.

If \( l(R_A) = 0 \) (mod 4), then \( v(R_A) = 0 \) (mod 5)

else if \( l(R_A) = 1 \) (mod 4), then \( v(R_A) = 4 \) (mod 5)

else if \( l(R_A) = 2 \) (mod 4), then \( v(R_A) = 2 \) (mod 5)

else if \( l(R_A) = 3 \) (mod 4), then \( v(R_A) = 3 \) (mod 5).

These assertions may at first look strange, especially because of the apparently unwarranted "modulo 4" in the length of input sequences. While our concern is with the proof and not the origin of the assertions, let us note as motivation the cycles of length 4 formed by both the 0- and 1-arrows in the state diagram of \( M_1 \).

We proceed now to prove the assertions. For the initial state \( A \), taking \( \lambda \) for \( R_A \), the assertion is seen to hold by virtue of (1). Next, to show that the truth of the assertions is preserved by state changes, we consider the effect of all transitions on the values of \( l \) and \( v \), as listed in Table II. It has to be verified that if the \( l \) and \( v \)'s at a previous state (subscripted by 0 in the table, for distinction) satisfy the assertions, then so do the \( l \) and \( v \)'s at the present state. For example, for the second line of Table II we have the following.

1) Let \( l(R_{a0}) = 0 \) (mod 4). Since \( v(R_{a0}) = 3 \) (mod 5) by the assertion at \( B \), it follows that \( l(R_A) = 1 \) (mod 4) and \( \sigma(R_A) = 3 + 2^0 \) (mod 5) = 4 (mod 5), satisfying the assertion at \( A \).

[In the derivation for \( \sigma(R_A) \) we have made use of the fact that if \( m \equiv n \) (mod 4) then \( 2^m \equiv 2^n \) (mod 5).]

2) Let \( l(R_{a0}) = 1 \) (mod 4). Then \( v(R_{a0}) = 0 \) (mod 5). Hence, \( l(R_A) = 2 \) (mod 4), and \( \sigma(R_A) = 0 + 2^1 \) (mod 5) = 2 (mod 5).

3) Let \( l(R_{a0}) = 2 \) (mod 4). Then \( v(R_{a0}) = 4 \) (mod 5). Hence,
TABLE I
ASSERTIONS FOR $M_1$

<table>
<thead>
<tr>
<th>Present State</th>
<th>Previous State</th>
<th>Input Sequence</th>
<th>Transition effect on $t$</th>
<th>Transition effect on $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$D$</td>
<td>$R_a^0$</td>
<td>$t(R_a^0) = t(R_a^0) + 1$</td>
<td>$v(R_a^0) = v(R_a^0)$</td>
</tr>
<tr>
<td>$B$</td>
<td>$R_a^0$</td>
<td>$t(R_a^0) = t(R_a^0) + 1$</td>
<td>$v(R_a^0) = v(R_a^0) + 2f(R_a)$</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>$R_a^1$</td>
<td>$t(R_a^1) = t(R_a^1) + 1$</td>
<td>$v(R_a^1) = v(R_a^1) + 2f(R_a)$</td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>$R_a^0$</td>
<td>$t(R_a^0) = t(R_a^0) + 1$</td>
<td>$v(R_a^0) = v(R_a^0) + 2f(R_a)$</td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>$R_a^1$</td>
<td>$t(R_a^1) = t(R_a^1) + 1$</td>
<td>$v(R_a^1) = v(R_a^1) + 2f(R_a)$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$R_a^0$</td>
<td>$t(R_a^0) = t(R_a^0) + 1$</td>
<td>$v(R_a^0) = v(R_a^0) + 2f(R_a)$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$R_a^1$</td>
<td>$t(R_a^1) = t(R_a^1) + 1$</td>
<td>$v(R_a^1) = v(R_a^1) + 2f(R_a)$</td>
<td></td>
</tr>
</tbody>
</table>

$l(R_a) \equiv 3 \pmod{4}$, and $v(R_a) = 4 + 2^k \pmod{5}$ for all values of $l(R_a)$. Therefore, the design is correct.

Example 2: For given integers $q$ and $r (0 \leq r < q)$, an automaton is to accept precisely those sequences which are the binary representations, with higher order digits leading, of the integers of the form $qn + r$. A particular design, for $q = 5$ and $r = 1$, is shown as the automaton $M_2$ of Fig. 2. In the general case, the automaton has $q$ states, 0, 1, $\ldots$, $q-1$, with 0 and $r$ as its initial and final states, respectively. The transition function of the automaton is defined as follows. The present state being $i$ and the input being $k (\neq 0$ or 1). The next state is the remainder of division of $2i + k$ by $q$.

In this case, for a sequence $x = a_0a_1a_2\cdots a_n$ the value represented by $x$ is defined by

$$v(x) = \sum_{i=0}^{n} a_i 2^{n-i}.$$  

Consequently, we get

$$v(x) = \sum_{i=0}^{n} a_i 2^{n-i}.$$  

We choose the convention

$$v(\lambda) = 0.$$  

The assertions are rather simple this time. With the state $i$, we associate the assertion

$$v(R_i) = i \pmod{q}.$$  

To prove the assertions, first we note that for the initial state 0 the assertion is true when $R_0$ is taken to be $\lambda$. Next, for any transition from state $i$ to state $j$ for input symbol $k$, it is immediate from the transition function definition that

$$j = 2i + k \pmod{q}.$$  

Hence, by assuming

$$v(R_i) = i \pmod{q}.$$  

we obtain

$$v(R_i) = v(R_i, k) = 2v(R_i) + k \pmod{q}.$$  

Therefore, the correctness assertions $M_1$ would have been proven in an easier way. The contrast with the direct proof shows that the assertions

so that $R_i$ is the binary representation of some integer of the form $qn + r$.

Before concluding, we remark that the automata $M_1$ and $M_2$ are the reverse of each other, their corresponding states being $(A, 1), (B, 3), (C, 0), (D, 2), (E, 4)$. By first proving $M_2$ correct and then observing the reversal relationship, the correctness of $M_1$ would have been proven in an easier way.
and proofs can sometimes be considerably simplified by an expeditious choice of machine transformation.

References

The Organization and Use of Parallel Memories

Paul Budnik and David J. Kuck, Member, IEEE

Abstract—As computer CPUs get faster, primary memories tend to be organized in parallel banks. The fastest machines now being developed can fetch at the order of 100 words in parallel. Unless memory and compiler designers are careful, serious memory conflicts and resulting performance degradation may result. Some of the important questions of design and use of such memories are discussed.

Index Terms—Memory conflicts, parallel computer, parallel memory, pipeline computer, skewed array storage.

Introduction
As the effective speeds of computer processing units have increased, more parallel memory modules have been used in computer systems. The largest computers now under development (e.g., CDC STAR, Burroughs Illiac IV) have of the order of 100 memories in parallel, each memory being capable of producing one word per memory cycle. It seems likely that this trend will continue in high-speed machines.

A major difficulty in the use of such memories is that accessing conflicts may arise. The process may generate addresses for several words in one memory unit at one time, thus effectively slowing down the system. If the memory speed and processor speed are balanced and if processing cannot proceed until every element requested has been fetched, then the processor is slowed down by a factor equal to the maximum number of conflicts in any memory module. Thus two or three addresses generated for some memory module per vector fetch can result in a significant system slowdown. In this note we point out several relations between memory design and use which may help ease these access conflicts.

In some machine organizations (e.g., Illiac IV) it may be desirable to fetch words from memory in a particular order since each memory module is connected to one or a few processors. In other machines (e.g., STAR) the order in which they are fetched may be of less concern since all memory modules are connected to a small buffer which can be indexed in any order by one or a few processors. We shall discuss both kinds of fetches.

We shall restrict our attention to the storage of two-dimensional arrays, although the results can be generalized to other data structures. In terms of these arrays we shall discuss various kinds of partitions, e.g., rows, columns, diagonals, and blocks, which certain application programs are likely to request in one memory fetch. We consider both forward diagonals \( a_{11}, a_{22}, \cdots, a_{nn} \) and reverse diagonals \( a_{1n}, a_{2n-1}, \cdots, a_{nn} \).

The desirability of fetching rows, columns, and diagonals is clear from the consideration of various common matrix operations [1]. Row and column fetching is also discussed in [2]. While many computations on arrays may be formulated as row and column operations, square blocks are often desirable. Matrix multiplication by partitions is an obvious example. Square (or nonsquare) blocks become very important when one considers storing arrays which are much larger or smaller than the number of parallel memory modules. One would then like to have a memory with an appropriate "resolution" as possible. For example, if a memory system has \( M \) memory modules it can be used to access \( M \times M \) arrays, one row at a time or one \( \sqrt{M} \times \sqrt{M} \) block at a time. In the latter case, arrays as small as \( \sqrt{M} \times \sqrt{M} \) can be fetched in one memory cycle. Large arrays of dimension \( p \times q \times \sqrt{M} \) can also be handled with full memory speed if \( p \) and \( q \) are integers. If \( p \) and \( q \) are not integers, then some fetches yield less than \( M \) useful elements. Small blocks are also desirable if one is considering transmission between a small fast memory and slower levels of a hierarchy.

For simplicity of addressing hardware and economy of address space, it is desirable to have the number of parallel memories be a power of 2. We shall discuss the limitations imposed by this constraint and also consider other numbers which avoid these limitations.

Definitions
We consider parallel systems of \( M \) memory modules, each with its own index register. Each of the \( M \) memories contains \( K \) words, \( K > M \). An array stored in some order in such a memory is called a stored array. We shall deal with the storage of two-dimensional arrays of at most \( MK \) elements. The dimensions of the original array, say \( P \times Q \), may, of course, be other than \( M \) and \( K \). We refer to the standard arrangement of matrix elements as matrix space or \( P \times Q \) space (see Fig. 1). We call the order of any subset of elements in matrix space that arrangement derived by concatenating the \((i+1)\)th row at the right of the \(i\)th row for all \( i \) and then selecting the desired subset. In other words, for any pair of

Manuscript received December 28, 1970; revised April 5, 1971.
This work was supported in part by NSF Grant GJ-27446.
P. Budnik was with the Department of Computer Science, University of Illinois, Urbana, Ill. He is now with the Department of Computer Science, University of California, Los Angeles, Calif.
D. J. Kuck is with the Department of Computer Science, University of Illinois, Urbana, Ill. 61801.