A survey of microprogram verification and validation methods

Tony P. Lucido*, Rahul Chattergy† and Udo W. Pooch*

Microprograms are increasingly being used to replace software controlled hardware in digital systems. Microprograms are at the heart of digital systems which use them. Consequently it has become necessary to guarantee that these microprograms are correct, i.e. (i) they are free of programming errors, and (ii) they satisfy the given specifications. Verification of microprogram correctness uses formal mathematical methods to provide rigorous proofs of their correctness. Validation of microprograms, on the other hand, aims to increase our confidence in their correctness by means of extensive simulation and testing. This paper provides a survey of verification and validation methods of microprograms along with a discussion of the differences in this respect between microprograms and software.

(Received July 1979)

Introduction

The technique of microprogramming has experienced a long delay from the time of its conception to the time of widespread use, due to the unavailability of technology to make such use efficient and economic. However, microprogramming is currently being used in diverse areas such as emulation, program enhancement, direct execution of high level languages, operating systems support, signal processing, computer graphics, fault diagnosis etc. (Agrawala and Rauscher, 1976). Such widespread application has naturally raised the question of correctness of these microprograms being used to replace software controlled hardware. It can be argued that the current concern with the correctness of microprograms is simply a reflection of a similar concern in the realm of software. However it should be noted that microprograms are at the heart of the digital systems that use them; correct software will not execute correctly when interpreted by erroneous microprograms.

A survey of various techniques for the verification of correctness of microprograms is presented in this paper. Two basically distinct approaches to this problem are identified; these are (i) formal methods of proving microprograms correct and (ii) empirical methods based on simulation and testing to ascertain the reliability of microprograms. The paper does not make an exhaustive survey of all the possible techniques that have been published but rather concentrates on a few representative ones in each of the two approaches to the problem.

The next section contains a discussion of two automated techniques of microprogram verification. This is followed by a section on simulation and testing and a discussion of the various aspects of microprogram verification. The reader may refer to Carter *et al.* (1978) as an additional source of information on the subject.

Formal verification techniques

As with software, the formal verification procedures for microprograms start with a specification of the algorithm to be microprogrammed or the architectures of the host machine and the target machine to be emulated. Various specification schemes can be found in the literature; a combination of the Vienna definition language and APL is used in Joyner *et al.* (1976b) to specify the macro and micro level machines; a Pascal-like language called STRUM (structured microprogramming language) is used in Patterson (1976) as a high level language for specifying microprograms.

A lucid discussion of the requirements of such specification languages for microprograms can be found in Bouricius (1974). The notation for such a specification language should be easy to learn, easy to use and easy to manipulate. An implication of this requirement is that mathematical symbols with their conventional meanings should be used as far as possible. The notation should be concise for ease of manipulation and a rich enough character set for operations should be available. A straightforward syntax for formulae and equations, simple, consistent and easy to use, is also desirable. The data structure should be flexible enough to specify the different data objects used in microprogramming. Bouricius (1974) advocates the use of array theory developed in More (1973) as the basis of such a specification language. The principal objects used in this theory are called arrays which are suitable for representing data structures used in microprogramming as well as in software. Items in arrays can be arrays and operations are mappings from the domain of all arrays to the codomain of all arrays. Operators are defined as transformations that change their mappings; most primitive operators are monadic and the system is closed under these operations. Bouricius (1974) notes that many useful identities, theorems and corollaries are readily available in array theory. He points out that many of the commonly used specification languages such as VDL, LISP, APL etc. are inadequate for the job because they do not have an axiomatic basis, they are many-sorted and not closed under operations defined in the language.

A partially automated system, called the microprogram certification system (MCS), for detecting errors in microcode has been reported in the literature (Joyner *et al.*, 1976b), and its various stages of development can be found in a series of papers (Leeman *et al.*, 1977; Carter *et al.*, 1977; Joyner *et al.*, 1976a; Birman and Joyner, 1974; Leeman *et al.*, 1974). In this system, the specifications for the correct implementation of an architecture are described by an abstract machine with a tree control structure. The elements of the control structure are created by a library of macroroutines. The state of the machine is given by a state vector whose components are the components of the machine such as registers, storage, switches, lines etc. The attributes of the microprogrammed computer, on which the specified architecture is to be implemented, are also specified by such an abstract machine. The problem of verification then reduces to a demonstration that the second abstract machine is an algebraic simulation (Milner, 1971) of the first abstract machine.

*Industrial Engineering Department, Texas A & M University, College Station, Texas 77843, USA.
†Department of Electrical Engineering, University of Hawaii, Honolulu, Hawaii, USA.

CCC-0010-4620/81/0024-0139 $02.00

```
[(MEM (16777216 32)
 (STK 32)
 (X   32)
 (IC  32)
 (SW  1)]
```

Fig. 1(a) Description of a machine state by means of the components of the machine in MCS (Joyner *et al.*, 1976b)

```
execpgm =
        SW = 1 → exec − pgm
                 execinstr (op, adl)
                 advctp
                 adl: instrprep (id, ix, op, ad)
                 id:  a[0]
                 ix:  a[1]
                 op:  a[2 + ι 6]
                 ad:  a[8 + ι 24]
                 a:   fetchword (IC)
        SW = 0 → Ω
```

Fig. 1(b) A tree control in MCS (Joyner *et al.*, 1976b)

Fig. 1(a) shows a description of the S-machine (Haralson and Polivak, 1972) state vector in the MCS specification language (Joyner *et al.*, 1976b). The S-machine has a one-bit switch SW, a main memory MEM of 2^{24}, 32 bit words, an index register X, an instruction counter IC and a top-of-stack pointer STK, all of 32 bits. The state vector components are treated as APL-like variables and their sizes are declared in the declaration statement as shown in Fig. 1(a). **Fig. 1(b)** shows a sample macro in APL-like notation which is used in forming the control tree (Joyner *et al.*, 1976b). The **execpgm** macro is used when the S-machine is ready to fetch and execute a machine instruction. For more details on the execution of this macro, the reader should consult Leeman *et al.* (1977).

The concept of algebraic simulation (Milner, 1971) is used in MCS to determine whether one abstract machine correctly simulates another. A theorem, stating the conditions for correct algebraic simulation, is given in Leeman *et al.* (1977). The problem is reduced to showing that the hypotheses of this theorem are true. In MCS, this last step is carried out interactively using automated expression simplifiers and theorem provers and the goal-directed problem solving techniques of artificial intelligence (Nilsson, 1971). An example of application of MCS to a real system can be found in Carter *et al.* (1978).

Another automated verification system called STRUM is described in Patterson (1976); an overview of STRUM is shown in **Fig. 2.** The STRUM system uses the inductive assertion method of proving programs correct, developed by Floyd (1967), Manna (1969) and others. In this method, non-executable logical assertions are added at suitable points in a program to describe the state of the variables in the program at those points. These assertions are combined with the program text to create logical statements. If these logical statements are verified they demonstrate a kind of consistency between the program text and the assertions, often called partial correctness. In STRUM these logical statements are called verification conditions (VCs).

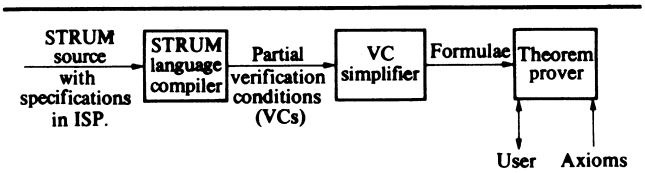

Fig. 2 STRUM verification system (Patterson, 1976)

The STRUM system uses an ISP (Bell and Newell, 1971) description of a computer to be the specification of the microprogram that emulates it. The algorithms are specified in the STRUM language which is a language with Pascal-like structure and syntax. The compiler accepts the ISP description of the machine and the STRUM algorithms and generates a prefix PASCAL and some tables. These are used by a VC generator to generate the necessary verification conditions. The VC simplifier simplifies complex VCs and proves the truth of the simple ones. The remaining VCs are verified by the user in an interactive manner by using a theorem prover (see Fig. 2).

MCS and STRUM are two of the major automated microprogram verification systems reported in the literature that use formal program proving techniques. Another such system has been reported earlier in Ramamoorthy and Shankar (1974) although it is not certain if it has been fully implemented. For an algebraic method of microprogram verification, the reader should consult Blikle and Budkowski (1976).

Simulation and test techniques

A formal proof of the correctness of a microprogram is the best guarantee that it meets the given specifications. The next best thing to a formal proof is extensive simulation and testing of the microprogram and its target machine. An automated tool that combines a hardware description language with a simulation system capable of simulating microprogrammed machines has been described in Adamowicz and Mirza (1977). This tool is based on a microcomputer design and simulation language (MDSL) and uses the basic model of microprogrammed machines given in Wilkes (1951).

MDSL allows one to define the architecture of the target machine, the structure of the microprogram to be executed and the initial state before execution starts. The simulator outputs routine statistics and optional histograms indicating the performance of the microprogrammed machine under various test conditions. Other automated simulation and test techniques (Howden, 1977; King, 1976) developed for software testing can be adapted for the testing and verification of microprograms, and Carter *et al.* (1978) mention the use of symbolic simulation based on the idea of King (1976). The DISSECT system presented in Howden (1977) is reported to increase the rate of detection of errors in the programs tested in that paper. It appears that this system is more of a debugger than a tool for ascertaining the reliability of programs. In connection with the reliability of programs, the method given in Musa (1975) determines the mean time to failure (MTTF) of programs. It is conceivable that this information can be combined with the MTTFs of hardware components to compute the MTTF of a microprogrammed machine.

As a final technique one should not discount the various assemblers for microprogrammable processors. Although no specific processor will be discussed here, the use of an assembly language, with its symbolic names management, data area definition and other facilities, can go a long way in producing microprograms having a higher probability of success than those composed at the control word level. The direct composition of control word contents is, unfortunately, an approach used in all too many cases. This is especially true when a register arithmetic logic unit (RALU) is employed to create a processor. For example, the M2900 from Motorola, Inc. (undated) is an example of one such RALU. With the control function implemented directly at the control word level, errors are likely to arise. If one makes use of an assembler and emulator to provide an initial attack on the program development, at least some errors can be found before the program is committed to control memory and direct testing at the hardware level is performed.

For a microinstruction control word that is not encoded the

possibility exists that conflicting operations within the same microinstruction can be specified. For example, there may exist left and right shifts of the arithmetic logic unit output which can be specified as needed. If both are specified in the same microinstruction, anomolous behaviour will generally result. Any competent supporting software will at least provide an indication to the user that such a condition has been specified. In all, the support software should at least test for any conflict condition which can be reasonably located.

Discussion

In principle, the formal verification of microprograms is no different from that of software; however, as noted in Ramamoorthy and Shankar (1974), there is an important difference in practice. To quote from Ramamoorthy and Shankar, '. . . a program is said to be correct if its output is correct for all legal input values. This seems to be inadequate for microprogram correctness because what is significant here is the final state of the machine after the microprogram execution. State of the machine does not just mean the output of the microprogram, whereas in the case of programs it did because program correctness is defined independent of the machine. The end state of program variables in the case of programs are not of concern. In the case of microprograms this is significant because the final state of the machine must include the internal variable set condition'. This important distinction between software and microprograms should be kept in mind when verification techniques for software are adapted to microprograms.

As noted in Carter *et al.* (1978), an interpreter of a high level software is specified in an idealised manner but in connection with microprograms, a piece of hardware must be specified in detail as the interpreter. Special hardware features such as parallelism without any synchronising primitives to make executions orderly, absence of interlocks for shared components, possible race conditions etc. must be considered carefully at the time of specification.

It is clear from the published literature that most of the software [except perhaps London (1971)] that has been formally verified consists of very small programs artificially generated for the very purpose of verification. This state of affairs raises some doubts about the applicability of formal techniques for the verification of large pieces of software. However, most microprograms are reasonably small, use simple data structures and operations, and hence perhaps are ideally suited for formal verification techniques.

Although the automated verification systems are the most promising approaches to microprogram verification, they have some severe drawbacks. In some systems such as the MCS (Carter *et al.*, 1978; Joyner *et al.*, 1976b), many detailed specifications must be written in VDL/APL type languages. Although these languages are very precise, as many APL programmers realise, it is also very easy to make errors that are hard to locate. In systems such as STRUM (Patterson, 1976) which use the inductive assertion method, the most difficult part consists of generating these assertions (Carter *et al.*, 1978; Joyner *et al.*, 1976b). Of course, this problem can be somewhat simplified if the microprogrammer integrates design of the microprograms with consideration of techniques for proving their correctness (Dijkstra, 1976). A much more serious defect in the automated verification systems is discussed in the next paragraph.

From Joyner *et al.* (1976b) and Patterson (1976) it appears that some automated aids are necessary for the formal verification of microprograms since the number of logical relations to be verified is very large. However, the approaches used in Joyner *et al.* (1976b) and Patterson (1976), no matter how clever, rely heavily on *unverified* software and hence their results are open to question. This state of affairs is likely to continue because of the difficulty in verifying large pieces of software; the only example of a verified verifier known to the authors being Ragland (1973).

An alternate and sound approach to microprogram verification can be based on the concept of virtual machines used to implement operating systems (Hoare and Perrott, 1972). The lowest level of the microprogrammed machine should use very simple microprograms (nanoprograms?) which can be formally verified by hand without any unverified automatic aids. The next level of the machine should be based on microprograms that use the microprogrammed operations of the lower level machine that have already been verified. Complex microprograms should be specified in a higher level language, using the top-down approach, but implemented by a bottom-up approach of putting together small chunks of verified code. Once a completely verified machine of reasonably high enough level is generated, a verified compiler for microcode generation should be designed. Although this is a difficult task, it needs to be done only once. This is the only approach that can guarantee the generation of correct microprograms by being completely independent of software of doubtful validity.

References

ADAMOWICZ, M. and MIRZA, J. (1977). MDSL: A microcomputer design and simulation language, *ACM SIG Micro Newsletter*, Vol. 8 No. 2, pp. 21–39.

AGRAWALA, A. K. and RAUSCHER, T. G. (1976). *Foundations of Microprogramming*, Academic Press, New York.

BELL, C. G. and NEWELL, A. (1971). *Computer Structures: Readings and Examples*, McGraw-Hill, New York.

BIRMAN, A. and JOYNER Jr, W. H. (1974). MVS—A system for microprogram validation, Part 1: The skeleton, *IBM Research Report RC* 4923, Yorktown Heights, New York (July).

BLIKLE, A. and BUDKOWSKI, S. (1976). Certification of microprograms by an algebraic method, *Proc. Ninth Annual Workshop on Microprogramming*, New Orleans, Louisiana (September), *IEEE Catalog No.* 76*CH*1148-6*C*, pp. 9–14.

BOURICIUS, W. G. (1974). Procedure for testing microprograms, *IBM Research Report RC* 5017 (revised), Yorktown Heights, New York (September).

CARTER, W. C. *et al.* (1977). Techniques of microprogram validation, *IBM Research Report RC* 6361, Yorktown Heights, New York (January).

CARTER, W. C. *et al.* (1978). Microprogram verification considered necessary, paper presented at the 1978 NCC, Annaheim, California.

DIJKSTRA, E. W. (1976). *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs.

FLOYD, R. W. (1967). Assigning meanings to programs, *Proc. AMS Symp. Computation.*

HARALSON, K. and POLIVAK, R. (1972). Microprogram training—An APL application, *Proc. Fourth Int. APL User's Conf.*

HOARE, C. A. R. and PERROTT, R. H. (Eds) (1972). *Operating System Techniques*, Academic Press, New York.

HOWDEN, W. E. (1977). Symbolic testing and the DISSECT symbolic evaluation system, *IEEE Transactions on Software Engineering*, Vol. 3 No. 4, pp. 266–278.

JOYNER Jr, W. H. *et al.* (1976a). Automated verification of microprograms, *IBM Research Report RC* 5941, Yorktown Heights, New York (April).

JOYNER, W. H. *et al.* (1976b). Automated proofs of microprogram correctness, *Proc. Ninth Annual Workshop on Microprogramming*, New Orleans, Louisiana (September), *IEEE Catalog No.* 76*CH*1148-6*C*, pp. 51–55.

KING, J. C. (1976). Symbolic execution and program testing, *CACM*, Vol. 19 No. 7, pp. 385–394.
LEEMAN Jr, G. B. (1974). Some techniques for microprogram validation, *IBM Research Report RC* 4616 (revised), Yorktown Heights, New York (April).
LEEMAN Jr, G. B. *et al.* (1977). An automated proof of microprogram correctness, *IBM Research Report RC* 6587, Yorktown Heights, New York (June).
LONDON, R. L. (1971). Experience with inductive assertions for proving programs correct, *Symp. Sem. Algo. Lan.*, Lecture Notes in Maths No. 188, pp. 236–252, Springer.
MANNA, Z. (1969). The correctness of programs, *Journal of Computer and System Sciences*, Vol. 3 No. 5, pp. 119–127.
MILNER, R. (1971). An algebraic definition of simulation between programs, *Proc. Second Inter. Conf. on Artificial Intelligence*, London, pp. 481–489 (September).
MORE Jr, T. (1973). Axioms and theorems for a theory of arrays, *IBM Journal of Research and Development*, Vol. 17 No. 3, pp. 135–175.
MOTOROLA INC. (undated). Booklet on the M2900 TTL Processor Family.
MUSA, J. D. (1975). A theory of software reliability and its application, *IEEE Transactions on Software Engineering*, Vol. 1 No. 3, pp. 312–327.
NILSSON, N. J. (1971). *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York.
PATTERSON, D. A. (1976). STRUM: Structured microprogram development system for correct firmware, *IEEE Transactions on Computers*, Vol. 25 No. 10, pp. 974–985.
RAGLAND, L. C. (1973). A verified program verifier, PhD Dissertation, University of Texas, Austin (May).
RAMAMOORTHY, C. V. and SHANKAR, K. S. (1974). Automatic testing for the correctness and equivalence of loopfree microprograms, *IEEE Transactions on Computers*, Vol. 23 No. 8, pp. 768–782.
WILKES, M. V. (1951). The best way to design an automatic calculating machine, *Proc. Manchester University Inaugural Conference*, pp. 16–18 (July).

Book reviews

Digital System Design with LSI Bit-Slice Logic, by G. J. Myers, 1980; 338 pages. (*John Wiley*, £16·25)

Bit slice logic forms a relatively unknown segment of the rapidly advancing LSI technology, but it is an important branch of this technology since it can and does provide solutions to design problems which that other, now well known, strain of the technology, the microprocessor, cannot provide due to its lack of speed or its inherent 'completeness'. Microprocessors are complete processing units whereas bit slices are fundamental building blocks, and consequently of immediate interest to a much smaller audience, mainly minicomputer and special purpose computer designers, system architects, and students.

The main thrust of this book is as a tutorial and reference work for the system architect and the digital design engineer, and being of a practical nature bears little resemblance to the more theoretical text which a student might require. The opening chapter provides an introduction to bit slice logic and develops the concept of the desirability of minimising the number of LSI chip types by making available a small set of universal chip types. The bit slice is examined in this context, and the most popular of all current bit slices, the AMD2901 is used as an example. A recurring theme throughout the book is that there is little relationship between bit slices and microprocessors other than that both are LSI components, and similarly that microprogramming and conventional programming are very dissimilar.

The following chapter introduces the concept of microprogrammed control, which is normally, but not necessarily used with bit slices. These first two chapters paint an easily readable picture of how bit slices came about and how they might be used, and would be useful reading to anyone who wished to gain a brief insight into the subject.

The next chapters survey the currently available bit slices, sequencing and support devices and are bread and butter to digital design engineers, but anyone else may find the going a little tough. The chapter on micro-instruction design is really the heart of the book, starting with pipelining and proceeding through various control storage techniques to a case study. A brief chapter on the programmable logic array (PLA) is strictly speaking outside the scope of the title of the book, but forms a useful interlude since the PLA is also a new and powerful fundamental digital building block.

The book is concluded by two short chapters, the first outlining the support tools which can be used by designers using microprogramming, such as assemblers and simulators, and the final chapter giving some thoughts on firmware engineering. The lasting impression that this book gives about designing systems around bit slices is that it is not yet a rigorous process but a subtle blend of art and science, based on intuition and experience, rather than established engineering principles. In places, the use of verbal explanations supported by an insufficient number of explanatory diagrams might make progress unnecessarily slow, even for experienced digital engineers. However, on balance this is a readable book, an excellent tutorial which can rapidly bring an engineer to the point where he can begin to use bit slices with confidence.

This book is thoroughly recommended to anyone who can, or wishes to, be styled a digital designer or system architect.

HARLEY QUILLIAM (Guildford)

Computer Logic, Testing and Verification, by Paul Roth, 1980; 176 pages. (*Computer Science Press Inc.*)

Dr Roth has a long standing reputation in the field of formal design and automatic logic testing. Users of his well known D-algorithm will therefore need no persuading as to the efficacy of this book. However, to the uninitiated, who may be looking for a general textbook, there are some surprises. The book is written in the style of a scientific paper and pays little regard to colloquialisms—for example you will be hard-pressed to find 'truth table' or 'nand gate' mentioned by name. Each chapter does indeed end with a short section entitled 'other work' and a bibliography, but these only tend to enhance the impression that the book is a very personal view of computer design.

Chapter 1 introduces cubical calculus for 2-level logic minimisation and develops the regular algorithmic notation, R-notation. Non-mathematicians will find this decidedly heavy going, but it is worth persevering because it is later generalised to a 'D-calculus' for the purposes of test generation. Chapter 2 deals with combinatorial logic design, with examples of the application of his P* algorithm for transforming multiple-level design to equivalent 2-level networks amenable to cubical calculus. Chapter 3 describes the testing of such networks via Roth's D-algorithm which, for a range of well defined faults in moderate networks, is quite helpful in automatic error detection. The next chapter, entitled logic automation, is really a description of how the R-notation was used as a basis for the PL/R design language. The tone of the book becomes somewhat parochial at this point. The difficult topic of testing sequential circuits is given sympathetic treatment in Chapter 5, via the concept of logic blocks bounded by registers. Finally, there follow three short chapters on logic verification, logic embedding (in the sense of VLSI cells), and repairable logic. At various points in the text reference is made to programs which use the formal techniques developed by the author; figures are given which show that, with ingenuity, practical design problems can be solved without prohibitively long run times. In summary, the book is a useful research monograph but the treatment is rather specialised.

S. H. LAVINGTON (Manchester)