Microprogram verification considered necessary

by W. C. CARTER, W. H. JOYNER, JR. and D. BRAND
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

INTRODUCTION

Microprograms have been around for some time now, and so have errors in them. Therefore techniques have been developed for finding the errors. We review these existing techniques and explain why we are working on a new one. The main advantage of our method is its ability not only to detect errors, but (if there are no errors) to give a formal proof that the microprogram does perform according to specifications. Using our approach we are working on actual machines and actual microcode. We discuss our experience with such proofs of correctness and explain what kinds of errors we are finding. The reader will also find a comparison of problems in microprogramming with problems of general software reliability.

WHY IS MICROPROGRAM VERIFICATION CONSIDERED NECESSARY?

"Microprogramming Trend Considered Dangerous" is the title of a letter in the ACM Forum. The letter continues, "Thus the development and maintenance process for microcode will increasingly resemble that of systems and applications programs. And therein lies mortal danger." This paper will briefly trace the history of microprogramming and microprogramming support systems. As microprogramming has developed, better support systems have evolved so that microprogramming has been a useful tool. Now, with the widespread development of microprocessors on a chip, and use of read/write control stores, new ideas for support systems must be developed. This paper then describes one such idea, symbolic simulation for microprogram verification, and indicates its necessity. The goal of this symbolic simulation of microprograms is to find all errors in the microcode (and correct them) while proving that the final code is correct. To do this, the Language for Symbolic Simulation (LSS) is used to prepare an APL-like executable description of the architecture (or program) specifications. The same language is used to prepare a description of the microprocessor. Then it is proved mathematically that the microprocessor when controlled by the actual microcode simulates the specifications, using symbols for data instead of bits. This language and procedures are described, together with an automated set of program aids, the Microprogram Certification System (MCS). Some results are given of applying these techniques and MCS to real programs and computers. These results indicate that microprogram verification by symbolic simulation is feasible, and micro code maintenance problems indicate that verification is necessary.

WHAT ARE PREVIOUS TECHNIQUES?

When Wilkes first proposed microprogramming in 1953 and built the EDSAC II, computers were built entirely of discrete components. The store holding the microprogram was also built of discrete components and the microprogram was debugged like the rest of the computer—by graduate students. When hardware components were available so that microprograms became commercially feasible (in the late 1950's), read-only storage was preferred because of its speed and low cost. Fixing errors in the microprogram then became slow and awkward, so special simulators were written to help debugging. Unfortunately every data flow or control flow change necessitated changes in both the simulation program and the microcode. A race developed—would the simulator or the computer be running first?

However, computers controlled by microprograms have many convenient features. An architecture for a family of computers is desirable, but needs a complex instruction set for the top of the line and for ease of programming. A rich instruction set costs very little more than a spartan set, if the computer is microprogrammed. Rewriting application programs for new machines is expensive. But emulation of the previous computer instruction set by a new computer using the facilities of microprogramming allows the old program to run. Maintenance of many distinct computers in a family is difficult, since the hardware for each computer is different. Again, using a microprogram allowed special maintenance techniques—microdiagnostics, scan-in, scan-out, log-out, instruction retry, etc. to be designed.

This microprogram complexity doomed the individually written simulator, and more complex design automation systems were written. One, the Controls Automation System, was written for the S/360 design effort. The computer hardware at the register transfer level was described by a Fortran-like programming language, including bit string handling. Design changes meant only a recompilation to produce an input for the simulator. The microprogram was written.
at an assembly level, but with mnemonics for the microinstructions chosen by the design engineers. The assembled input to the simulator was kept in a program file, in which any single subroutine could be changed. The event driven simulator could be instructed to set up desired starting and ending for the simulation, to select the desired microcode for simulation, to produce snapshots of the registers at variable breakpoints, to dump in emergencies, to trace with or without timing checking, etc. This and similar systems were a definite advance, but problems remained.

WHAT WAS WRONG WITH THEM?

Another litany on the difficulties of program testing in the middle 1960's is hardly needed. Anyone who has written a program to be used extensively by other people knows the problems. "Testing detects the presence of bugs, but never their absence" is the well-known refrain. "Enough" test cases can never be generated in our limited time on earth. How is a good selection to be made? The coverage depends upon the skill and dedication of the individual who wrote the microprogram. But even if all paths are traversed by test cases, faults which are data dependent can still remain. Moreover, simulation in any guise is slow and expensive.

Thus the systems did reach the field with faults in their control. Diagnosis of the computers themselves was difficult. Most faults occurred many levels deep—a particular combination of instructions and data turned them into observable errors. The best routines for exposing these faults were the higher level language compilers. (The diagnostics run, but the compiler doesn’t.) Therefore capturing the environment in which your micro will run. 12

WHAT HAPPENED?

Human ingenuity and perseverance triumphed again. Microprogramming was—and is—a great success. The computers worked well, and the new facilities made available more answers per dollar spent.

Hardware test support equipment came into use—expensive but necessary. Special computers, equipped with read/write control stores were built. Facilities for rapidly changing the control store contents were added, and fixes were performed in minutes. Changes could now be tried rapidly, and if they seemed to work all system and test programs could be run against the microprogram change before final commitment. Also, automatic data test generators were built, and millions of pseudo-random data patterns could be processed.

Now the bandwagon effect began and is still continuing. SIGMICRO (ACM) and the Technical Committee on Microprogramming (IEEE CS) were formed and began to hold their yearly workshops, presenting new ideas. If microprogramming is good for central processors it must be good for I/O equipment. Look at all the new functions which can be performed! Need reliability/availability? Add special progressive scan test routines. Help to do this is provided in abundance. Do you think that the use of higher level languages (Pascal, Algol, Fortran, etc.) will help your programming problems? Just provide the correct environment and the standard language can be used. Do you think that structured programming will help? Many advances in program testing are now available. Do you want a system which will analyze your program, and produce test inputs so that all paths will be traced at least once, and all instructions executed (if possible)? Such systems are available. Would you like to learn more about selecting test cases, devising adequate test procedures, applying support systems to your own problems? Help is available.

So what’s the problem? Once again in our enthusiasm we are building faster and faster—and reliability is falling behind. Letters to the editor such as Lehman’s "Microprogramming Trend Considered Harmful" point out the dangers in this. All of the techniques listed so far test specific cases—and the untested cases can still be erroneous. Microprograms are hidden from all but the designer. Should we explain to the checker at the local supermarket that the buttons on the cash register send data to a computer which then uses this information to get a "correct" answer? Most of the time everything works correctly—but when errors occur rectifying them is expensive—and fixing the system is even more expensive. (Have you bought a very cheap pocket calculator recently?)

WHAT ELSE IS AVAILABLE?

The problems that microcode poses are also posed by ordinary programming systems when such systems are written to be used extensively by people distinct from the originators. Ten years ago a formal, mathematically oriented approach, called program verification, was formulated. In this approach a series of assertions, written in first order
predicate calculus, are provided for the program input, output, and various intermediate points in the program. In particular every loop must satisfy an assertion called a “loop invariant” specified by the user. Devising these invariant assertions is not easy, even though many papers have been written, each proposing a procedure which is effective in various special cases. Various simple program modules (written in high level languages) have been proved correct, and more informal proofs have been given for larger examples. However, as with all proofs, the proofs are only as good as the hypotheses and the number of people who have actively checked them. A difficulty with the pure form of this standard approach is that by starting from an externally given program text it necessarily loses sight of the program’s genetic origins. This deficiency has been recognized and proposals made to cure it. In general the idea has been to recreate, or use, a formal expression of the generating steps by which programmers construct programs in the first place. These proposals are still in the research stage.

Another approach has been studied by King and experimental program aids have been constructed. They use as a major part of program validation the execution of the parts of the program between assertions using symbols, not the actual data values. The difficulty here (which is one of the difficulties in automatic program verification also) is defining and implementing the execution semantics for the symbols and the operators used in the programming language.

Symbolic execution can be seen to have several deficiencies. Symbol manipulation is costly and slow. Again the question of coverage arises. The data paths are represented symbolically, but which paths can be covered? Mostly main lines? What is the total cost? Will all errors be found? In Howden’s experiments neither ordinary good testing techniques nor symbolic execution found all known errors in a Fortran program. In addition there are the real world problems of the actual computer representation of data, with overflow, interrupts, etc. Another difficulty is that symbolic execution also ignores the knowledge that the programmer has about the specifications, and it does not check the specifications. Recent studies have shown that wrong or inadequate specifications account for 50 percent of program errors, and these errors are the most expensive to find.

WHY DO THESE TECHNIQUES HAVE PROMISE FOR VERIFYING MICROCODE?

The techniques of program verification have been applied for the most part to higher level languages and have met with limited success. Fully automatic methods are almost nonexistent—often a fairly detailed human analysis of the data structures and control structures used is necessary before automated aids can be applied. Also, such proof of necessity deals with some abstraction of what actually happens in a computer when the compiled code produced from these programs is executed. Errors can therefore result because the actual running program is not the program which was verified. Why should program verification techniques have any more promise when applied to microprograms, especially to actual microcoded implementations, than they have in other applications?

As is described in the next section, our approach requires a complete formal description of the computer on which the microcode is to be run, and the actual microcode—the array of bit patterns—is used with a symbolic interpreter. Though this makes necessary a more complex description, because overflow, invalid array indexing, etc. must all be explicitly described, errors which would go undetected if these factors were ignored can be detected.

The use of a single data structure—a two-dimensional array of bits—simplifies both the symbolic execution and the specification of the Floyd-type assertions or their counterparts. The registers, memory, I/O channels, switches, etc. of the computer are all represented as arrays. Also, the operations on these arrays are limited to storing and retrieving strings of bits addressed by indices, concatenation, re-shaping to other dimensions, and certain arithmetic and logical operations. Problems with program verification often arise when more complicated data structures are involved; if there are such structures in a microprogram, they are explicitly implemented in terms of the more primitive notions, and verification takes place over that more restricted domain.

The fact that microprogrammers are not constrained to write in a “structured” way (and often do not in order to increase time or space efficiency) probably increases the difficulty of formulating and placing correctness assertions in microcode. However, once that is done, the control structure of a machine description which interprets microcode is simple, and symbolic execution can proceed in a straightforward manner.

The way in which we use symbolic execution to prove microcode correct is another factor which improves our chances for success. As described in the next section, we do not place assertions at various points in the microcode which relate values of machine components to previous values, or describe a state (e.g., sorted, permuted, etc.) which the computer must satisfy. Rather, we equate the machine components with components of a higher level description, a description of the architectural specifications which the microcode is supposed to implement. In this way we can partially avoid the necessity for quantifiers, single-program inductive invariants, and references to initial or previous values of machine components.

The most compelling argument for the feasibility of microprogram certification is our experience. We have designed and implemented an automated system to aid in verifying microprograms, and have used it to detect and correct errors in “real” code.

WHAT ARE THE DIFFERENCES BETWEEN VERIFYING MICROCODE AND VERIFYING HIGHER LEVEL PROGRAMS?

A microprogram is just a special kind of program. The main difference between firmware and software is in the semantics of the programming language and the emphasis on the speed of execution for firmware. Microcode is inter-
interpreted by hardware, whereas higher level programs can easily have their semantics defined in terms of another program—an interpreter. This makes a difference because a verifier must be given the semantics of the programming language. Thus for verifying microcode it is necessary to describe a piece of hardware as an interpreter. An interpreter for a higher level programming language presents an idealized view of the environment (this is the whole idea of higher level languages). The lower we go towards the physical realities the more we have to face to them; here are some of the realities we are faced with.

Hardware has the inherent capability of operating in parallel. Parallelism in software is certainly very common, but software can use the cushions provided by the lower layers of software. For example, memory interlocks prevent two processes from accessing a shared variable at the same time. Nobody will provide such interlocks for hardware, where shared variables are physical wires; therefore we have to watch for the possibility of races. Cooperating processes in software can take advantage of synchronizing primitives, which shield the software designer from the idiosyncrasies of hardware. No such shield is available for a microprogram, which is the shield for others. The microcode itself must handle such things as I/O, interrupts because of external causes, program-caused overflows, and other exceptions, which are "hidden" from the high level language programmer and which most program verification techniques ignore.

At the specification level, there are parts of the description which are left intentionally ambiguous, so that different models of a simple architecture (such as the 360 series) may implement them in different ways. In verifying a particular implementation, we must handle this ambiguity in such a way that it does not "lock in" the specification to a particular implementation technique.

On the other hand there are aspects of microprogramming, which make verification easier. Microprograms operate on one fixed data structure—bit vectors. Further, these operations are not very sophisticated, in the sense that the bit vectors are not supposed to represent notions very different from themselves (at least as far as the architecture is concerned)—a bit vector typically represents a number. Thus the statement of correctness is usually close to what the microcode does, but the results are expressed in terms of numbers rather than bits.

All these characteristics of microprograms make them good subjects of mechanical verification. They are so concerned with small details that it is difficult for people to examine them carefully. At the same time they are not sufficiently intellectually challenging for a person. But these are exactly the characteristics which machines find easy to handle, and therefore computers have a better chance in competition with people when verifying microcode than when verifying higher level programs.

HOW SHOULD MICROCODE BE VERIFIED?

Testing techniques are based on the ability to recognize an erroneous procedure. Verification proves that a program written at a level above machine code is correct. What is needed is a system which will find all errors in a given program as it runs on a computer in a reasonable length of time. To do this three capabilities are necessary: formal specifications which are natural and easy to produce; coverage determination, i.e., the ability to recognize a correct procedure; and the ability to execute the code as existing in the microprocessor so that analysis of the results will determine the fault.

To formalize specifications we model all facets of system operation using the APL-like Language for Symbolic Simulation (LSS). This description is an abstract machine whose state consists of a facility vector describing the machine (or storage) components and a control structure which permits parallel asynchronous operation. The facility vector components, two-dimensional arrays of bits, are described by dimension statements, as in Figure 1. These facilities are an 8 bit wide Main Storage with $2^{24}$ locations, sixteen 32 bit General Purpose Registers and a 33 bit Accumulator. Operations upon the facility vector are specified by a library of macro routines written using APL-like operators, as in Figure 2.

This macro models a S/370 add register instruction. The 2's complement addition is modeled by decoding (+) the contents of the registers from bit strings to integers, adding the integers, then encoding (−) the result in 33 bits ((33p2)). This value is assigned (=: ) to a. SETCOND is called, which sets the S/370 condition code. The low order 32 bits of a (bits 1 to 32, 0 origin) are put in GPR[R1: ].

The macro SETCOND is shown in Figure 3. SETCOND shows the conditional structure of the LSS macros. OVF FLO is a predicate from the predicate library. If it is true, then bits 34 and 35 of the Program Status Word are both set to 1. Otherwise, if the result is zero, the condition code is (0 0), or positive (1 0), or finally negative (0 1).

The last example is OVERFLO, which shows some of the logic operations available. Here, z is the 32-bit result of adding the 32-bit quantities X and Y (Figure 4).

The LSS language is carefully defined in Reference 34. Only two kinds of objects are necessary for LSS, integers and two dimensional matrices of bits (truth values are one bit matrices).

Modeling computing systems by describing their state vectors and the state transformations is intuitively plausible. LSS specifications are executable, i.e. given a state vector...
value as input, a symbolic output state vector value may be obtained by symbolic execution. A symbolic value is an expression built from the operators of the LSS language and symbols representing the initial values of the variables, after appropriate simplification. For example, if A has the value (11100000) and B has a symbolic value $B$, then the assignment statement $A := A \& B$ becomes $A := (11100000) \& \$B; A is given the new value $B[4],(0000)$ (the "." here is concatenation). The adequacy of the system model may be tested by executing cases of special interest. The model acts as a special purpose machine, i.e., it carries out one set of specifications.

LSS is also used to describe the hardware attributes of the computer on which the specified architecture or program is to be implemented by code. This implementation description acts like a general purpose computer. When the assembled code to be verified (a matrix of bits) is inserted in memory, and a set of symbolic inputs is given, symbolic execution and simplification will determine the outputs.

To determine if the desired action is identical to the action of the code, we use symbolic simulation. Heuristically speaking, the execution of the machine description and actual code simulate the execution of the specifications if everything the specifications can do the machine and code can do, though possibly in a different way. Because the two facility vectors (specification and implementation) may have different components, to carry out the simulation it is first necessary to specify a set of relations between them. These simulation relation components specify, for selected pairs of specification and implementation control points, relations between the state vector quantities which must hold at these points. See Reference 35 for a complete and formal description.

To carry out the verification a symbolic execution tree, or proof tree, must be constructed.26 An automated, interactive system, MCS (Microprogram Certification System), has been designed and implemented to aid in this verification (see Figure 5). A node, or goal, in the tree represents a class of states of the system at one point in time. In general, a node will consist of:

(a) two state vectors—an assignment of symbolic values to facility variables;
(b) two control points—positions on the control trees;
(c) a predicate list—a list of conditions that the initial values must satisfy in order that the states represented by this node can be reached.

OVERFLOW(X,Y,Z) := \( \{X[0]=X[1]\} \& \{Y[0]=Z[0]\} \)

Figure 4

The root of the tree (which represents the entire problem of proving the simulation) has a son, called a simulation relation node, for each of the simulation relation components. The state vector, control, and predicate lists of these nodes correspond to points at which the specified relations must hold. The values of the elements of the facility vectors in these simulation relation nodes satisfy the relationships necessary so that the implemented machine, under the control of the program, will simulate the specification machine. A branch in the proof tree represents a computation path. The leaves of the proof tree represent all possible final states.

The following process is used to build the proof tree. First, the simulation relation nodes are generated. In these nodes the most general symbolic values such that the conditions of the corresponding simulation relation component hold are given to the variables of the state vector. Any conditions which cannot be asserted in this way are placed on the predicate list. The next step is to run each abstract machine, performing symbolic computation until another corresponding pair of control points in the simulation relation is reached.

One machine is arbitrarily chosen to be run first. At this initial point variables have as values either symbolic constants (symbols representing unknown but fixed values) or values determined from the simulation relation. Control proceeds as in normal execution. When symbolic constants are encountered in expressions being computed in assignment statements, the value assigned is a simplified combination involving operators and symbolic constants.

When symbolic constants occur in predicates evaluated to determine possible branches, a single flow of control may not be able to be determined. The predicate may evaluate to a Boolean expression involving symbolic constants, such as:

\[ \text{SETCOND}(X,Y,Z) = \neg PSW[34+Z] = (1 \ 1) \]

\[ \text{OVERFLOW}(X,Y,Z) \rightarrow PSW[34+Z] := (0 \ 0) \]

\[ \text{OVERFLOW}(X,Y,Z) \rightarrow PSW[34+Z] := (1 \ 0) \]

\[ \text{OVERFLOW}(X,Y,Z) \rightarrow PSW[34+Z] := (0 \ 1) \]

Figure 3

Theorem Prover

Interpretor Generator

Simplifier

User

Supervisor

Formal Descriptions

Figure 5—Parts of the microprogram certification system
as "\$X=0," which cannot be evaluated to "true" or "false." In such a case all possible logically independent results of evaluating the predicate must be considered; in the previous case \$X=0 and \$X\neq0. The program doing the symbolic evaluation will generate a node (subgoal) for each independent result, add a predicate expressing the truth of the result to the predicate list, and simplify the result. In the case being considered, two subgoals will be generated. For \$X=0, 0 will be substituted everywhere for \$X; for \$X\neq0, this predicate will be put on the predicate list. The user then chooses the path he wishes to follow (the program is interactive) and begins again. The remaining paths will be traversed later. If the predicate involving symbolic constants evaluates to true or false, no path branching occurs.

After generating a series of nodes and branches, the control corresponding to a simulation relation component is reached. Then the other abstract machine is run until a simulation control point for that level is reached.

Now the system verifies that the pair of control points reached defines the control component of a simulation relation node. The values of the two facility vectors are substituted into the simulation conditions of this component, and a goal is generated for each simulation condition. Using the predicate list, a proof is attempted for each simulation condition. The process is repeated for each node of the tree which is not a final node. In the process a complete goal tree is formed.

Newly created expressions are simplified at once to prevent the propagation of unsimplified forms. At present MCS has over 400 rules for the simplification of APL and logical expressions. If a theorem simplifies to "true," then the goal is achieved (and theorems are the only goals generated in proving simulation that can be achieved directly, without generation of subgoals). If the theorem cannot be proved by the theorem prover, its simplified form is the theorem of the pattern in a single generated subgoal so that analysis by the user is facilitated. It may occur that a theorem cannot be proved, or that a pair of stopping points not corresponding to a simulation relation component is reached, or that a stopping point is not reached. Then an error must be sought in the code or in one of the descriptions. In addition the occurrence of an unexplained branch will signal the presence of an error.37

In MCS, theorem proving, which is simplification of APL-like expressions, occurs at a number of places and forms an important part of the system.

This general process becomes more complicated when the asynchronous parallel operation of the system must be considered. Our method of modelling parallelism involves considering the processes as composed of \(K=2\) separate subprocesses running sequentially, but interleaving arbitrarily. No notion of time is built in. How the subprocesses interleave is immaterial as long as no shared variables are used, since these subprocesses are and remain independent. When shared variables are used, we have to consider which subprocesses can be executed between two references made by a single subprocess to a shared variable. We also must consider how the shared variables and the subprocesses interrelate in the process execution. To model this we provide LSS operators to express when interleaving may occur; when interleaving is not specified a subprocess runs without interruption.

A solution has been proposed by Brand and Joyner to consider the \(K\) separate subprocesses as each running with its own control tree and to add two statements to LSS, DELAY and WAIT, so that the shared variables can be correctly handled during the arbitrary interleaving of the subprocesses.38

EXPERIENCE USING SYMBOLIC SIMULATION

The MCS is now being applied to a version of a real computer, the NASA Standard Spaceborn Computer-2 (NSSC-2). This version is called the Hybrid Technology Computer (HTC). This version was produced at the IBM Federal Systems Division at Huntsville, Alabama.36 Its architectural specifications require it to support the 86 System/360 Standard Instructions (no decimal or floating point operations). It has the usual sixteen 32 bit general purpose registers and 16 to 64K bytes of storage (with an extended protection mechanism). A single I/O channel supports three types of I/O: device initiated I/O, CPU initiated I/O and Direct Memory Access I/O. Device initiated I/O has two classes. Buffered I/O allows a device to transfer data to/from a table in main memory without knowing the location or size of the table. External interrupt permits a device to interrupt the normal program sequence. CPU initiated I/O also provides two types of services: command and 16 bit data transfers to or from I/O devices or CPU to I/O 16 bit control operations. Direct Memory Access is transparent to the NSSC-2 microprogram controlled hardware unless a hardware error occurs, when the microprogram is called. Each use of the I/O interface, whether CPU or device initiated, follows a certain protocol, depending on the I/O command.

Describing the NSSC-2 CPU was straightforward. The state vector facilities are essentially those listed in the S/360 programming manual. The basic CPU macros are all of the type (and approximate length) of those described earlier in this paper. This control, however, involves recursive calls, so that instructions will be fetched and executed indefinitelly, or until some interrupt or other external signal intervenes. The basic idea is shown in Figure 6. The macro which handles the interaction between the CPU and the I/O is much more complicated because of the asynchronous parallel action of the CPU and I/O devices. This is carefully described in Reference 40. The description takes 25 double spaced pages of computer printout.

This architecture is implemented by a data flow that has a 16 bit wide data path, three working 16 bit registers, a 16

\[
\text{cpu} = \begin{cases} 
\text{(SOFT-STOP}=0\rightarrow\text{DO- INSTRUCTION}} & \text{SERVICE- INTERRUPT} \\
\text{cpu} & \text{CPU} \\
\text{(SOFT-STOP}=1\rightarrow\text{DO INTERRUPT}} & \text{CPU}
\end{cases}
\]

Figure 6
bit wide store whose storage data register does double duty, a 64×16 bit scratchpad memory, and a 16 bit I/O register. The data flow is controlled by a 1K read only memory of 64 bit wide horizontal micro-instructions, another 64×16 read only memory for computer instruction decoding, a 32 bit instruction register, and various sized registers and signals for speed of operation. The facility vector consists of the storage data flow registers and control signals. The macros describing the action of the 21 types of CPU oriented microcommands are again straightforward. The I/O protocols depend basically upon 2 other microcommands. These protocols require that certain acknowledgments be sent by the CPU, that the CPU wait until certain signals are received from devices, that interrupts not be ignored indefinitely, etc. Verification of these microprogram controlled protocols is an important part of microprogram certification (see Reference 40 for a complete description). This description takes 20 pages of computer printout.

One part of the HTC microcode, where the new treatment of I/O was heavily used, is system reset. This code executed upon start of the HTC, and its job is mainly to initialize various data structures. It also initializes the I/O channel status; here it is important to ensure that the protocols are strictly observed. During system reset the microcode uses a certain piece of code that is used also for other purposes. For these other purposes the request line (which can be raised by the I/O interface) is sensed. The operation of system reset, however, depends on this line being down. To ensure this the computer must prevent any device from being serviced by the I/O interface. Therefore before entering the shared piece of code the microcode resets the interface. (That is, service of any device will be terminated). After resetting the interface the proper operation of reset is checked using this shared code. To prevent a device from getting service during this checking, the command line (under CPU control) is enabled. If these precautions were not taken, system reset would not be performed correctly when a device happens to request service at exactly the wrong moment. Needless to say, we could hardly expect this kind of error to be detected by testing.

In the course of trying to prove correctness of system reset we found that, after checking that the reset was valid, the microcode jumped to a routine which implemented part of the protocol for sending the storage data register information to an I/O device. This is not only a violation of specifications, but the information transmitted is random—whatever happens to be in the storage data register when the power is turned on. Moreover, in this routine the microcode sent an acknowledgment to the I/O interface without checking that the correct I/O routine had begun. If an unknown device was requesting service, it would receive the service acknowledgment even though no device input has been read. The device—and its input—would be actually unknown. After consulting with the author of the microcode it was determined that he used this feature for debugging purposes (indicating to Test Support Equipment that reset was finished) and had left it in the final product by error.

After eliminating this jump to the routine which violated the architectural specifications, we were able to verify system reset, following 5 paths and proving 28 theorems.

The next part of the microcode to be tested concerned the CPU IFETCH, 15 microinstructions, 3 paths and 45 theorems. Figure 7 gives an outline of the proof tree developed for this HTC microcode. In this part of the code we found that if a program overflow had been signaled, but upon return to the beginning of IFETCH an I/O interrupt was waiting, then the overflow would be ignored. We also found that no half-word instruction may reside in the last two bytes of memory. Since the memory contents are not available immediately, the first thing the IFETCH code does is fetch a word (16 bits) from memory, then immediately read the next word so 32 bits are read to fill the instruction register. If the instruction is 16 bits there is no slowdown, while if the instruction is 32 bits there is an increase in instruction speed. Unfortunately if the first 16 bits read are a 16 bit instruction in the last two bytes of memory, IFETCH tries to read the next two bytes, a memory overcapacity interrupt is given and IFETCH is never completed. This is an example of a bending of specifications, not a microprogram fault. A note in the programming manual would be better than slowing the HTC down.

Another 97 microinstructions were executed during symbolic execution of 10 S/360 RR instructions. Thirty-two paths were followed, and 480 theorems proved. In the Branch and Link instruction (BALR) the link address was stored before the branch address was accessed. Thus BALR R,R became a NO-OP instead of a branch and link. This was not detected by being unable to prove a theorem; instead, it was detected by generation of an unexpected branch of the goal tree. (The error had been found by further testing.
before we found it.) One other specification exception was found: bit 31 in the key word in the Set Storage Keys (SSK) instruction must be zero. This is counter to the instruction description, which says that bit 31 is ignored.

CONCLUSIONS

Some architectures, data flows, programs and microprograms have been described using LSS. Microcode has been symbolically simulated and errors found. Microcode and machine level code may thus be the place where automated program verification techniques will reveal their applicability to actual software and firmware.

REFERENCES