Simulation of real-time program faults

P. Burnett[†], P. A. Kidd[*], and A. M. Lister[*]

Two techniques are presented for making simulation a powerful aid in the development of real-time programs. The techniques have been implemented in a Honeywell 516 simulator on a PDP-10 and have proved successful in the development of software for a real-time medical information system.

(Received August 1972)

1. Introduction

Simulation of one computer on another has long been a recognised way of developing software for the simulated machine. While the value of this technique is apparent from its widespread use, it has hitherto been limited by two factors. These are, firstly the difficulty of simulating real-time phenomena, which is essential if real-time programs are to be debugged, and secondly the generally inflexible way in which breakpoints can be introduced into a program. This paper presents techniques for overcoming these deficiencies and for making simulation a powerful interactive debugging tool.

The first technique is for simulating real-time events such as peripheral transfers and interrupts. Since a large proportion of program bugs are concerned with these events, and since they become apparent only in real-time situations, the authors believe that this feature can prove valuable. It is described in Section 2.

The second technique is for halting the simulated execution of a program should any specified combination of possible states arise. These states are such things as the values of program registers, the number of instructions executed, and the relationships between the contents of memory locations. The conditions for halting may be changed at any time by the user issuing commands from a terminal, and he thus has a powerful debugging aid at his disposal. This feature is described in Section 3.

The stimulus for this work arose from the development of the Guys/Essex Medical Information Project (Abrams *et al.*, 1968; Bowden *et al.*, 1971) on a Honeywell 516. Because of the large amount of software to be written and the limited 'hands-on' time available, a simulator with the above features was clearly desirable. Such a simulator has been written for a DEC PDP-10 (Burnett, 1972), and has proved extremely valuable for program development.

2. Peripheral simulation

Realistic simulation of peripheral activity is essential if a simulator is to be a powerful tool for detecting real-time program faults. Necessary requirements are that the simulator should:

(*a*) test all parts of any peripheral handling code
(*b*) reproduce the relative speeds of CPU and peripherals
(*c*) preserve crisis times
(*d*) simulate interrupts.

Requirement (*a*) rules out any method which makes all peripheral transfers instantaneous and which therefore does not test those pieces of code which are entered if a device is busy. An attempt to overcome this deficiency by making tests for 'device ready' fail once and succeed thereafter will fall foul of requirements (*b*) and (*c*).

What is needed is a method of delaying peripheral transfers and of altering the flags associated with them. This delay is to be until such time as the simulator has executed a comparable number of instructions to that which the actual machine would execute during the course of a real transfer. The degree of comparability need not be exact; it is sufficient that it be within the same order of magnitude, and hence it can be achieved by counting instructions executed by the simulator without taking account of their varying execution times.

Thus when a peripheral activity is initiated the appropriate status flags are set and a count associated with that activity is initialised. On each subsequent instruction cycle the count is decremented. When it expires the transfer of data or altering of flags is performed according to the manner in which the simulated peripheral is supposedly active.

To implement such a scheme directly would be restrictive since it would impair the speed of a large number of programs in those situations where real-time control is not necessary, interrupts are switched off, and peripheral activity is directed by means of wait loops. For example, consider as a worst case the output of characters to a teletype. Taking the output speed as 10 characters per second and the average instruction time as 3 microseconds gives a count for this device of around 30,000. The time taken to execute 30,000 instructions under simulation may be prohibitive, and will usually be largely wasted in a two instruction test loop (skip if ready; jump . − 1).

Some advantage may be gained by allowing the device counts to be varied, within limits, by the programmer. In effect this allows the speed of the peripherals to be controlled as may be felt necessary. The user is then in a strong position in most program development situations. On one hand, if his program has peripherals active but is not dependent on their relative speeds, then he can set device counts low; on the other hand, if the relative speeds are critical then the device counts can be set so as to gear the peripherals almost exactly to real time.

This is essentially the method adopted, and it should be noted that it is adequate for the simulation of crisis times. The crisis time for, say, reading consecutive characters from a paper tape in motion is simulated by the length of time taken for the tape reader count to expire. On expiry of the count the next character is transferrred to the input buffer, overwriting the previous one irrespective of whether the buffer has been read or not. Thus the obligation of the programmer to transfer out of the buffer sufficiently quickly in real life is reproduced exactly under simulation.

Setting device counts low will reduce crisis times and so if a program can meet crisis times under simulation it will certainly meet them in real life. However, the possibility arises that a valid program can go wrong through failure to meet crisis times when run on the simulator. Provided device counts are not set too low this should seldom happen, and experience has shown that quite low device count settings can be achieved before any differences between programs run on the real machine and under simulation is observed.

[†]ICL, Bracknell, [*]Computing Centre, University of Essex, Wivenhoe Park, Colchester, Essex CO4 3SQ

A further benefit of variable device counts is that it is possible to investigate the critical parts of a real-time system by selectively varying the device speeds under simulation and observing the system's behaviour.

It is worth noting that as far as data transfer is concerned the simulated peripherals may be represented on the host machine by any convenient peripheral. For example, data being input through a simulated paper tape reader may in fact come from a disc file or from magnetic tape; the correspondence between the simulated peripherals and the host's actual peripherals is made by the user.

The overhead of peripheral simulation is two instructions per simulated machine instruction for each active peripheral; if no peripherals are active the overhead is a single (test) instruction.

Interrupts

The scheme for peripheral simulation described above makes no mention of interrupt handling. In fact interrupts can be fitted into it with little extra effort. What is required is a flag for each peripheral which, when set, means 'this peripheral is in interrupt condition'. The flag will be set on expiry of a count in the same way as the ready flag is set—indeed for most peripherals it will be set at the same time. At the beginning of each instruction cycle the simulator checks whether any peripheral is interrupting, i.e. whether the following three conditions all hold:

(*a*) the peripheral is in interrupt condition

(*b*) the bit corresponding to the peripheral in the interrupt enable mask is set

(*c*) interrupts are enabled.

If so then a jump to the interrupt routine is effected in the same manner as the real machine.

3. Conditional halts

The conditional halt is the second feature of a simulator which makes it of great value in interactive debugging. It is used to halt simulation and return to the user whenever a specified set of conditions holds. The power of the feature lies in the fact that the conditions are evaluated during each instruction cycle *at run-time*, as opposed to more traditional break-point insertion which is effectively performed at load-time.

We shall first give a few examples of the kind of conditions which may be specified and then proceed to describe the implementation.

Conditions are concerned with the values of registers and memory locations and with the number of instructions executed. Examples of *simple conditions* are:

(i) $P < 20$
satisfied if the content of the program counter is less than 20_8.

(ii) $A = [1000]$
satisfied if the content of the accumulator is equal to that of location 1000.

(iii) $N = 1$
satisfied if one instruction has been executed since the simulator was last entered.

Simple conditions may be combined into *multiple conditions* by the connectives AND(.) and OR(+), and inverted by the use of NOT(%). Round brackets may be used to resolve ambiguities. Concatenation of two simple conditions implies an omitted '.'. Examples of multiple conditions are:

(i) $1000 = [1001] = [1002] = 0$
satisfied if the contents of locations 1000, 1001, 1002 are all zero

(ii) $P < 5000 + P > 6400$
satisfied if the program counter does not lie between 5000 and 6400. This type of condition can be used to trap any attempt to jump out of code into data.

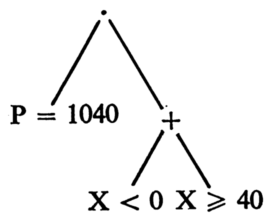
(iii) $\%((P >= 5000 . P <= 5777) + (P >= 1000 . P <= 1077))$
This will trap any attempt to jump out of the two areas 5000 − 5777 and 1000 − 1077.

(iv) $P = 1040 . (X < 0 + X >= 40)$
This will detect an out of bounds array subscript for an array of length 40_8 accessed via indexing by an instruction in location 1040.

The implementation of conditional halts is divided into two parts. The first part occurs at user level, where the specified conditions are formed into a *condition tree*, the leaves of which are simple conditions and the nodes of which are the connectives '.' and '+'. The second part occurs during simulation at the beginning of each instruction cycle, when the condition tree is evaluated and if the result 'true' is obtained an exit is made to the user.

Building the condition tree

Building the condition tree is a two-pass process. The first pass is a lexical scan which reduces the input string of characters specifying a multiple condition to a canonical form. Pass two acts on the resulting character string to produce a tree by an algorithm closely resembling that for the conversion of an arithmetic expression into reverse Polish form. The 'operators' are '.' and '+', and the 'operands' are the individual simple conditions. Thus the condition given in example (iv) above is compiled into the tree



NOT (%) -operators are eliminated from the condition tree by use of De Morgan's laws, yielding two advantages; firstly the size of the tree is reduced, thus saving both space and evaluation time, and secondly the tree becomes uniformly binary. The leaves of the tree, which are simple conditions, consist of pointers to the operands of the condition and to the appropriate machine code routine for evaluating it.

Evaluating the condition tree

Evaluation of the condition tree occurs at the beginning of each instruction cycle, and if the result is 'true' then an exit is made to the user. Evaluation is by means of a recursive routine which calls itself to evaluate sub-trees and calls the appropriate machine code routine to evaluate leaves. Redundant evaluation of sub-trees is avoided whenever the result is apparent from evaluation of the left-hand branch alone.

The time overhead depends on the multiplicity of the condition being evaluated, but as this in practice is rarely greater than four it has not proved to be a critical consideration. When no condition is being evaluated the overhead is a single (test) instruction.

4. Conclusion

The authors feel that simulation of one machine on another can be a powerful aid in developing real-time programs for the simulated machine. Experience shows that when a simulator is written to incorporate the features described in this paper then the environment provided by the simulator is almost indistinguishable, as far as real-time program running is concerned, from the actual machine itself. The additional benefit of being able to control the running and halting of the program at will makes the simulator a useful debugging tool, with negligible overhead in cases where this facility is not required. The experience gained in use of the conditional halt facility is currently being evaluated with a view to providing similar facilities implemented in hardware for a particular class of machine.

Acknowledgements
The authors wish to thank their colleagues for their help throughout this project, and the referee for his constructive comments on the preparation of this paper.

References

Abrams, M. E., Bowden, K. F., Chamberlain, J. O. P., and MacCallum, I. R. (1968). *A Computer-Based General Practice and Health Centre Information System*, Journal of the Royal College of General Practitioners, Vol. 16, p. 515.

Bowden, K. F., MacCallum, I. R., and Patience, S. P. (1971). *Data Structures for General Practice Records*. IFIP Congress Proceedings.

Burnett, P. (1972). *An Interactive System to Simulate a Small Computer*. M.Sc. Dissertation, Computing Centre, Essex University.

Book reviews

Numerical Methods for Unconstrained Optimimization, edited by W. Murray, 1972; 144 pages. (*Academic Press Inc.* (*London*), £3·00)

The main criticism I have to make about this book concerns the rather excessive delay in publication of the work. Dr. Murray explains in the preface that this was partly due to the delay in the decision to publish the proceedings of the IMA/NPL conference, held in January 1971. However, it is commendable that most of the contributors took advantage of the delay to revise their original papers to include material which has appeared since the conference.

The book provides an excellent introduction to the subject of unconstrained optimisation which is suitable not only for the non-specialist but also as an undergraduate course text. There are one or two printing errors, and, personally, I prefer the use of a heavy type **x** for the position vector of the variables, but the notation used is clearly explained in the Glossary of Symbols—an example other authors would do well to follow!

The chapters, or sections, of the book correspond to the papers given at the conference; after a brief introduction to 'Fundamentals' given by the editor, W. H. Swann gives a good practical survey of Direct Search methods, which could have included with advantage a numerical comparison of the methods described. This is followed by a chapter on 'Problems related to unconstrained optimization' by M. J. D. Powell, dealing with methods for functions which are sums of squared terms, and with the use of unconstrained techniques in constrained optimisation problems. The latter topic includes both the penalty and barrier function methods and the use of Lagrange parameter methods for problems with equality constraints.

Murray's description of 'Second derivative methods' commences with a summary of the advantages and disadvantages of the 'classical' steepest descent and Newton methods (the former is not a second derivative method, of course) and includes a discussion of the problems which arise if the Hessian matrix is not positive definite. If this is not the case the importance of the use of Choleski's method is stressed and a numerically stable modified Newton algorithm using this factorisation is described. A useful inclusion in the chapter is a description of the application of the Marquardt-Levenberg method when second derivatives are available. Fletcher's survey of 'Conjugate direction methods' starts with a review of the objectives of optimisation and conditions required for efficiency of any method; he shows that the conjugate direction methods meet these criteria and considers in turn methods which do and do not require calculation of the derivatives of the objective function. The chapter is concluded by relating these methods to the general class of Quasi-Newton methods which are described in detail by Broyden in the following chapter. The underlying principles of this class of method form the basis of Broyden's survey, and the paper includes a comparison of the use of various forms of update formulae for the approximation to the Hessian and its inverse.

The final two chapters are essentially practical in nature. In 'Failure, the causes and cures', Murray gives useful hints for dealing with programming errors, rounding errors and problems arising in transformation of variables; he also compares the numerical stability of the various classes of method, and finally discusses the use of algorithms with regard to choice of input parameters and interpretation of computer results. Fletcher's 'Survey of Algorithms for unconstrained optimisation' is also extremely useful from the practical point of view, but is, unfortunately, a subject which 'dates' very quickly and has therefore suffered most from the delay in publication. No reference is made to the NAG library the aim of which is to include the best available routines, written in both ALGOL 60 and FORTRAN, for most of the important topics in numerical analysis, including optimisation. One of the most useful features in the original paper was the flow chart for choice of algorithm for any practical problem—this has been omitted from the book, although it was included in the Harwell report of the paper (TP456).

To summarise, the book is well written and a very worthwhile purchase for both the specialist and non-specialist. One hopes that the IMA/NPL conference on *Constrained Optimization*, to be held early in 1974 will live up to the high standard set by its predecessor.

Heather M. Liddell (London)

Numerical Methods for Nonlinear Optimization, edited by F. A. Lootsma, 1972; 439 pages. (*Academic Press Inc., London*, £9·00)

The problem of finding the maximum or minimum value of a function of several variables is one which appears trivial until one actually has to solve it. For some considerable time now the main strategies have been well known. These include variable-metric, conjugate-gradient and non-gradient methods. However as the number of variables increases the tactical problems multiply and the economics of obtaining solutions begin to obtrude. Fairly trivial changes in the tactics employed in developing algorithms for solving these problems can make quite radical changes in the economics of the methods. Thus the importance of efficient algorithms has emerged.

This book is the report of a conference held at Dundee University in June-July 1971. It includes papers on the theoretical aspects of methods for unconstrained optimisation using variable-metric and conjugate-gradient methods. Particular methods for non-linear least squares and curve-fittings—in which some defined distance-function has to be minimised—are studied. Some attention is paid to the design of methods for finding global optima of problems that may have local non-global optima. There are also papers discussing problems of constrained optimisation.

Of recent years non-gradient methods have been somewhat neglected, but since these promise to be more sparing of computer storage space they have a particular potential when the number of variables is very large. There is some discussion of simplex type algorithms demonstrating this.

It would be invidious to choose any paper from this collection for special mention—the general standard is so high. A wealth of computational evidence is given about the performance of the algorithms discussed, and the tests used appear convincingly realistic. All in all this book is a valuable 'tool-box' for anyone faced with a non-linear optimisation problem.

As a footnote to an entertaining query raised by F. H. Branin about who was Raphson, D. J. Wilde's off-the-cuff reply that he was Newton's programmer is wide of the mark. Raphson appears to be the true author of the so called Newton-Raphson method. He gave it in his book Analysis Aequationum Universalis published in London in 1690.

A. Young (Coleraine)