Journal of Microcomputer Applications (1982) 5, 209–223

SAS—an experimental tool for dynamic program structure acquisition and analysis

V. Callaghan and K. Barker

Department of Electronic and Electrical Engineering,
The University,
Mappin Street,
Sheffield S1 3JD, UK

This paper describes an experimental microprocessor-based tool, SAS (Software Analysis System), which has been developed to enable dynamic program structure acquisition and analysis to be made on digital computing machines.

The system uses a universal hardware extraction technique to obtain branch vectors which are used to analyse and display the structure of the software being monitored. A display, especially designed for small instrument screens, is used to present this structure. Emphasis has been directed towards development of methods with high degrees of machine independence and it is envisaged that such techniques could either be integrated into the new generation of logic analysers or form part of a universal tool for computer programmers. Initial research has been guided towards the application of these techniques to compiled, assembled, or machine coded systems and in this context a number of techniques are described.

The motivation for this research has been provided by the present escalating software costs, in particular those in post development which account for approximately 75% of the total software expenditure.

1. Introduction

Software is presently dominating the cost of computing systems. The price of computer hardware is falling at a rate of approximately 28% p.a. whilst programmer productivity is rising at only 4–7% p.a. This indicates an escalating dominance of software costs on computer systems (Allison, 1980) (see Figure 1). The software demand growth rate is

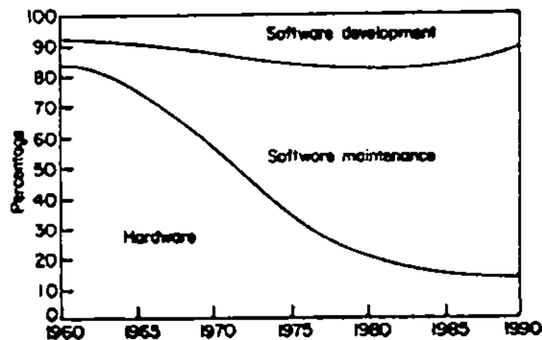

Figure 1. Life-cycle costs of computer systems.

0143–3792/82/030209+15 $03.00/0

© 1982 Academic Press Inc. (London) Limited

estimated to be in the order of 21–23% p.a. whilst the software labour force and its productivity per individual are producing a combined growth rate of only 11·5–17% p.a. (Boehm, 1976). Recent American government figures indicate that, if this trend continues, by 1990 there will be a shortage of between 1–2 million programmers (Schindler, 1981). One solution to this problem would be to substantially raise the level of programmers' productivity. It is this environment which is motivating the research for new tools and techniques to assist the programmer in his efforts throughout the software life cycle.

1.1 *Software life cycle*

The life cycle of a program may be envisaged as comprising two stages; namely, development and maintenance. Software development accounts for approximately 25% of the total cost, the remainder being attributed to maintenance (Mills, 1980). An estimate of the order of expenditure involved is provided by Boehm who reckoned that the annual cost of software in the United States during 1976 was some 20 billion dollars (Boehm, 1976).

1.2 *Software maintenance*

The term 'maintenance' (Munson, 1981) is misleading because when used in this context, it refers to the following post delivery activities defined by Swanson (Munson, 1981; Swanson, 1976) as:

(i) Corrective—fixing a pre-existing error (in either specification or code).
(ii) Adaptive—modifying the software to accommodate environment change.
(iii) Perfective—improving or augmenting the performing capabilities.

Boehm (1976) has defined maintenance as 'the process of modifying existing operational software whilst leaving its primary functions intact'. These post-delivery activities usually continue for considerably longer periods than their corresponding development time thus accounting for the high maintenance overheads. Reducing any of the activities defined by Swanson can thus potentially have a profound influence on software expenditure. Unfortunately, as Boehm (Boehm, 1976) has stated 'Despite its size, software maintenance is a highly neglected activity'. SAS has been constructed to address the problem of maintenance by providing a tool which can counter the programmers' intrinsic intellectual limitations (Gries, 1980) by, in the first instance, restricting software complexity and enforcing adherence to structural constructs during software development and quality assurance checks and, in the second instance, supporting maintenance by providing an aid for deciphering poorly documented or complex code.

1.3 *Present technology*

The main impetus for innovation and development of program execution monitoring tools has been provided by companies with a commercial interest (Marshall, 1978). Results of such research usually manifest themselves in marketed products. Reported research is sparse, a fact supported by a recent survey (Plattner & Nievergelt, 1981) which reports, 'program execution monitoring has been a neglected research topic' and

concludes by stating 'program execution monitoring has not received attention commensurate to its practical importance'.

1.3.1 *Commercial systems.* Commercially available tools which provide facilities for program execution monitoring are; (i) performance monitors, (ii) logic analysers and (iii) development-emulation systems.

Performance monitors (Nutt, 1975) are normally used on computer systems which manage such facilities as multi-user, virtual storage and multiprogramming. They gather and analyse information concerning the monitored system by either timing or counting the occurrence of specific events or conditions. Activities monitored by these systems include CPU activity, channel activity and I/O activity. Analysis of this data is then used to (i) investigate resource utilization, (ii) determine the characteristics of the job load, (iii) remove bottlenecks and (iv) tune software and gather data for system monitoring. Due to the large quantity of data produced by these systems, the information is normally gathered and presented statistically. These tools are usually used on large systems and cost in the region of £40,000 to £150,000.

Logic analysers (Marshall, 1978) are tools which log absolute time sequential data present on a number of parallel channels. Data acquisition may usually be started on the occurrence of a pre-specified combinational trigger and continues until the analyser memory is full. Data is normally displayed either as a timing diagram, state map or as a table. A current trend by manufacturers is the adaptation of logic analysers to directly support program development by including facilities such as disassemblers. Typically, a logic analyser may contain 24 channels, a memory depth of 256 words, an operating speed of 100 MHz and cost between £4000–£8000.

Development systems and hardware emulators (Krummel, 1977) include facilities such as dynamic tracing and breakpoint execution to aid program development and debugging. Although many systems implement these features in software, some systems, particularly emulators, provide hardware for this purpose. Professional development and emulation systems cost in the region of £5000 to £25,000.

1.3.2 *Research activities.* Research activities concerned with program execution monitoring are reported in an early paper by Stockham (Stockham, 1965) and a recent paper by Plattner & Nievergelt (1981). Fryer (1973) has described a dumb system, 'The Memory Bus Monitor' which utilizes the stream of addresses and data travelling the memory bus in conjunction with hardware comparitors, timers and counters. These provide such measures as branch ratio, routine timing and variable behaviour. An eight-word shift register provides a limited trace facility. Lemon (1979) describes an improved version of the monitor, 'SOVAC', which uses a PDP-11/34 to support a graphic terminal, simplify the user interface and provide an analysis capability. IBM's recent reports have described a 'Programable Map and Trace Instrument—PMATI' (Lloyd *et al.*, 1980) and a 'Program Counter Sampling Tool' (Armbruster *et al.*, 1978). PMATI maps and traces program execution by interfacing to the system address bus. The trace function records the sequential stream of address whilst the mapping facility is implemented by associating a bit with each possible address occurrence. The program counter sampling tool periodically samples the instruction counter and increments a counter associated with a window which the value of the program counter lies between. The window widths and address space coverage are variable whilst the number of counters and windows is

fixed at 4096. In applications where the loss of time sequential data is not of significance an advantage of increased sampling periods may be achieved by use of this technique. A debugging tool 'The Program Tracer' (Antoine *et al.*, 1979) interfaces to the system address, data and control buses. Upon triggering it selectively acquires data from the monitored buses according to a set of initialization conditions. The selective acquisition capability both differentiates it from, and provides a sizeable data reduction over the conventional logic analysis techniques. Results are presented as text on either a printer or VDU. Versions for tracing the ITT 3202, Intel 8085 and the RCA 1802 processors have been reported.

1.3.2 *Summary*. The majority of these tools and techniques use the monitored systems buses to extract *direct* program execution data in the form of real time traces. As such, they are primarily debugging tools. Performance monitors extract *indirect* data concerning the effects of the program execution from various system testpoints and perform analysis to produce certain measures on characteristics of the software. SAS differs from these tools by *directly* extracting a fundamental structural program property, performing analysis and presenting the programmer with data concerning the program's complexity and structure.

2. The SAS system

2.1 *Physical description*

SAS consists of a cabinet which houses two single sided, double density 8 in. disc drives, a power supply, cooling unit and a 12-slot rack containing:

(i) a CPU board;
(ii) a disc controller board;
(iii) two structure table RAM boards;
(iv) an EPROM system software board;
(v) a structure monitor board;
(vi) a control board.

System peripherals include a VDU, a printer, a colour monitor and a data acquisition probe set.

2.2 *Principle of operation*

Figure 2 shows a block diagram of the SAS system. The personality adaptor interfaces the program counter or memory address lines of the system under test to the structure monitor which extracts branch vectors. These vectors are stored in one of two memory blocks, structure tables 1 and 2, which in turn may be operated upon, displayed, or stored on the system discs.

2.2.1 *Structure acquisition principle*. The technique to be described is based upon the principle that branches in compiled and assembled code correspond directly to deviations from the normal sequential incrementation process of the program counter. Dynamically executed branches can therefore be logged during program execution by storing two

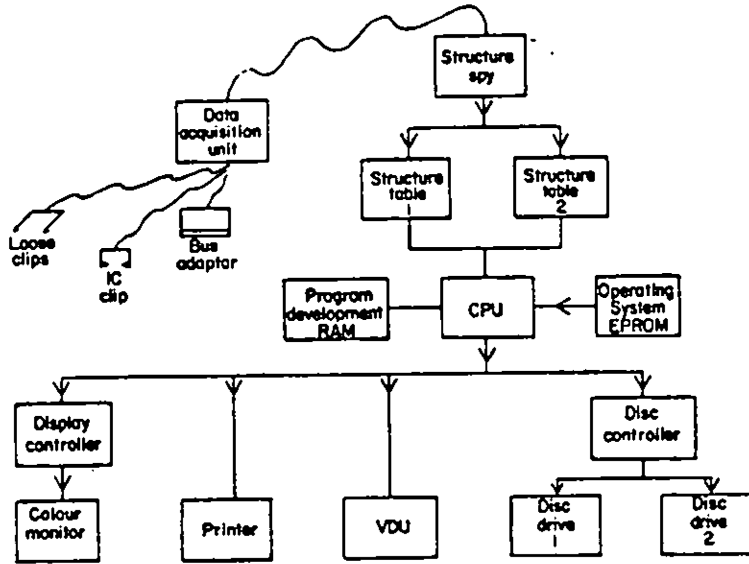

Figure 2. Schematic diagram of SAS.

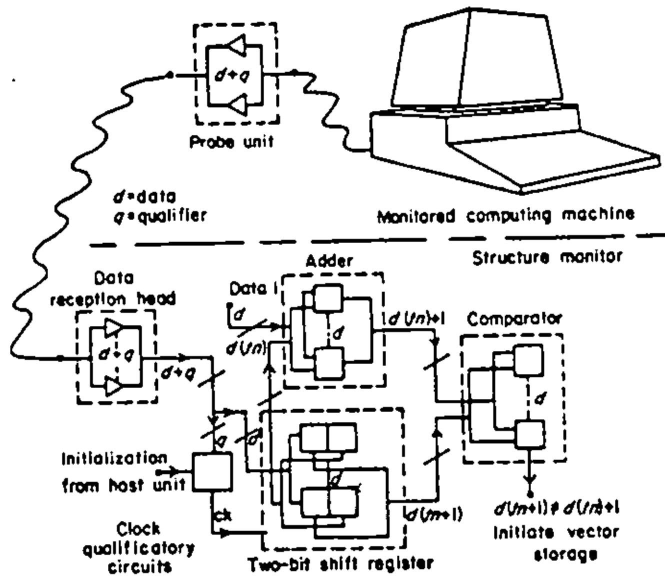

Figure 3. Structure acquisition scheme.

words which correspond to the value of the program counter immediately prior and following a non-incrementally sequential update. It is then possible to reconstruct the structural properties of the executing program in the form of a directed graph from the table.

Figure 3 shows a block diagram of the structure acquisition scheme. A probe unit is connected to either the program counter chip set, computer backplane or a micro-

processor. These probes fetch the program counter outputs including clock qualificatory signals through line driving and receiving circuits to the structure monitor. In the structure monitor the successive addresses are clocked into a two bit shift register which enable the time adjacent values of the instruction address register to be analysed for a branch by the succeeding circuitry. Analysis of branch conditions is performed by comparing the shift register word corresponding to the instruction address register's latest value to its former value plus one. An inequality in this comparison indicates that a branch has taken place and a sequence is initiated which causes the two non-sequential values of the program counter to be stored in a memory-based structure table.

2.2.2 *Structure display principle.* The technique utilized to display the program's dynamic structure is based upon a directed graph and has been particularly devised for use in conjunction with small instrument display screens. Essentially it is a circle, the circumference of which is calibrated to correspond to the portion of memory being monitored. Branches in the program's normal sequential flow are depicted as chords on the circle. A clockwise rotation corresponds to the normal positive sequential incrementation process of the program counter. On a colour display the chords are colour coded to indicate the direction of the branch, the execution frequency being impressed as the

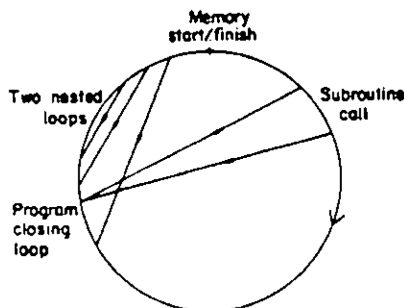

Figure 4. Structure map.

intensity of the chord. Figure 4 illustrates a measurement being made on a Texas Instruments TM990/101 which has a simple program containing three loops, two of which are nested and a subroutine call.

2.2.3 *Structure analysis principles.* SAS provides a set of software-implemented algorithms which augment the hardware-based acquisition and display system by providing a means of testing, modifying and presenting the acquired branch vectors in a manner which may be readily interpreted. A complete list of SAS commands and algorithms is provided in Figure 5. These commands may be divided into three classes, namely, test, control and operator commands. Test commands are concerned with verifying various functional elements in SAS itself such as the structure tables and display (e.g. TM, SB). Control commands supervise the acquisition, movement, storage and display of data within SAS (e.g. DP, DV). Operator commands are responsible for analysis of the data usually operating on data stored in the structure tables.

The structure tables are the nucleus of the analysis system (see Figure 6). All data which is communicated to the user is obtained directly from the structure tables. Data

1	AC	Analyse Complexity	35	FO	Frequency Operator
2	AO	And Operator	36	GP	Get Program
3	AZ	Acquisition on Zero	37	G1	Get 1st Structure Table
4	AO	Acquisition on One			
5	CB	Colour Bar Generator	38	G2	Get 2nd Structure Table
6	CD	Continually Display Vectors	39	LV	List Vectors
7	CM	Clear Memory	40	MA	Magnitude Acquisition
8	CP	Continually Print Texas 990 Vectors	41	MO	Magnitude Operator
			42	PC	Program Cru Bits
9	CS	Clear Screen	43	PE	Print Expansion Parameters
10	CT	Configure Texas 990 Personality Card	44	RE	Reset Expansion
11	DC	Draw Circle	45	RI	Retrieve Image
12	DD	Disc Directory	46	RM	Return to Monitor
13	DI	Display Image	47	SB	Star Burst
14	DM	Display Mark	48	SC	Set Colour Table
15	DO	Data Operator	49	SV	Sort Vectors
16	DP	Dump Program	50	TM	Test Memory
17	DV	Display Vectors	51	TO	Texas Operator
18	D1	Dump 1st Structure Table	52	TW	Transfer Word Block
19	D2	Dump 2nd Structure Table	53	WA	Window Acquisition
20	ED	Expand Display	54	WL	Wandering Line
21	EO	Exor Operator	55	WO	Window Operator
22-23	E 'x'	Execute Program at F 'x' 00	56	WZ	Wand Zero Acquisition
34	FD	Format Disc	57	WO	Wand One Acquisition

Figure 5. SAS command index.

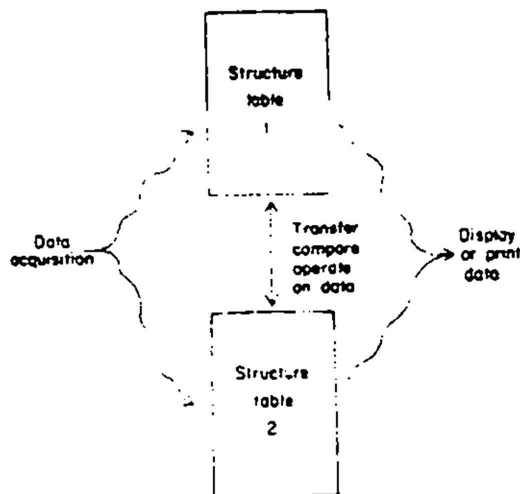

Figure 6. The structure tables are the nucleus of the software.

intended to be ignored by SAS or the user is nulled in these tables. This principle is utilized by the analytical routines which null the vectors in locations which have been either eliminated or made redundant.

Five main analysis techniques are employed in SAS which are described in the following paragraphs.

(i) *Vector magnitude, frequency and window filtering.* Filtering in the SAS context, refers to the elimination of branch vectors which do not conform to prescribed conditions. Two types of filtering process are employed in SAS; pre-storage and post-storage. Pre-storage filters examine and eliminate, if necessary, the branch vectors as they are acquired before storage. They can be implemented in either hardware or software. Post-storage filters process the branch vectors stored in the structure tables, eliminated vectors being set to zero. Data null vectors are ignored by the output processors of SAS.

Magnitude filtering. These algorithms such as MA (pre-storage), and MO (post-storage) determine the magnitude of each vector and compare this to a magnitude window supplied by the user. Vectors with a magnitude not between the limits set by the window are nulled.

Frequency filtering. Frequency filtering refers to the elimination of vectors from a specified table whose frequency of occurrence lies outside the boundaries of a window supplied by the user. Intrinsically, frequency filtering can be only of a post storage nature, as pre-storage implementation would imply prior knowledge of vectors yet ungenerated. An example of this filter is the FO algorithm.

Window filtering. The elimination of vectors whose source or destination lies outside a user specified window is referred to as window filtering. WA is a pre-storage implementation of this algorithm whilst WO is a post-storage version.

(ii) *Complexity and structure analysis.* Computer programs may be assembled using arbitrary control structures. SAS extracts the branch vectors dynamically from the program whilst it is running and uses them to fabricate a diagram which mirrors the program structure. The freedom allowed in being able to use arbitrary control constructs can lead to the production of highly complex programs which are difficult to understand, maintain, adapt and test. To combat this type of complexity, a methodology which allows the programmer to build programs from only a limited set of structures is often adopted. This type of methodology is already in frequent use amongst high level language programmers who commonly use the three constructs; linear sequence, selection and iteration (Jensen & Williams, 1981). Unlike high level languages whose algorithmic implementations are based on the virtual machine reflected by the language, low level assembly languages' algorithmic implementations depend on the actual machine. As such the use of GOTOs or absolute branches is unavoidable in all but trivial assembly or machine code programs. Further, it is felt to gain many of the intrinsic advantages of particular machines a more flexible structuring criterion is required. The approach on SAS is to allow the user the ultimate choice of which structure criteria is appropriate to apply by placing the structure analysis routines in RAM which is supported by the system discs and called by the EF command. A measure of the conformity of a program to specified constructs is presented as the number of instances in which these programming constructs are violated.

Predicate branching in the control flow of computer programs can potentially create control structures which are beyond the management intellect of program development, maintenance and adaption engineers (Gries, 1980, Mills, 1980).

Forward predicate branching causes the number of distinct control paths to increase in proportion to 2^n where n is the number of predicates, whilst backwards branching, can cause an infinite number of potential paths. Thus, even small programs may contain a number of control paths which is beyond the normal intellectual capacity of an individual (Gries, 1980). A measure for this type of complexity has been devised by Thomas McCabe (1976, 1978) and is known as cyclomatic complexity. This approach uses the cyclomatic number derived from graph theory as a measure. The cyclomatic number is the number of independent paths existing within a program module which, when taken in combination, generate all paths and is expressed as:

$$V(G)=e-n+2p$$

where

$V(G) \times$ cyclomatic number (complexity measure);
e = number of edges (branch vectors);
n = number of vertices (branch vector nodes);
p = number of connected components (modules).

McCabe suggests a limit of 10 as representing an optimum level of complexity. This algorithm is called on SAS by the command AC.

(iii) *Instruction, data separation.* A need to separate program data from instruction addresses occurs in two main instances; program counter tracking which contains data words embedded in the program memory field and composite instruction-data tracking which gather data fields from both inside and outside the program memory area. The latter situation would arise if measurements were made on a microprocessor without instruction fetch cycle qualifying signals, whilst the former occurs on systems with such signals. Program counter tracking systems effectively branch around data blocks producing pseudo branch vectors which are not part of the program logic flow. Some processors treat the additional words in multiple word instructions as data words, thus inducing pseudo branch vectors. These false branch instructions are eliminated by using the 'not instruction fetch' qualifying signal to produce a data track and negating branch vectors, which correspond to these data domains. Composite instruction-data tracking results in the generation of mixed data fields and instruction branch vectors. Intrinsically, this system eliminates the multiple instruction word pseudo branch problem encountered in the former case. The most successful solution to separating the instruction and data activity is to window filter the program memory area, the disadvantage being that this requires some prior knowledge of the program being run. This data operation is called by the SAS command DO.

(iv) *Event correlation.* A requirement to correlate sections of code with certain events is evident when programs are being maintained or adapted, in particular when accompanying documentation is either not available or inadequate. In such circumstances, the correlation wand of SAS may be used. Essentially, the wand is an electrical probe which may be placed in contact with the conductor transmitting a signal, related to a section of software. The occurrence of this signal is then used to cause the structure spy to store the address it is currently monitoring.

Such values may then be either printed out or marked on the structure map. Event correlation on SAS is executed by using the WO and WZ commands.

(v) *Structure comparison*. SAS has two structure tables the contents of which may be compared, the results providing a list of branch vectors and events which are either equal or not equal.

AND operator. The AND operator, AO, compares the contents of a reference structure table to an operation structure table. Vectors which are in the operation table and not the reference table are set to zero.

EXOR operator. This algorithm, EO, compares the contents of two structure tables, a reference and operation table. Vectors which appear in both tables are nulled in the operation table.

3. SAS application

3.1 *Measurement considerations*

The application of SAS is affected by the type of hardware and software technology incorporated into the computer system it is intended to measure. The main application considerations quantitized from the SAS perspective are therefore discussed.

3.1.1 *Hardware*. The program counter on modern digital computing machines consists of either a set of discrete logic integrated circuits or is integrated into a VLSI device (Osbourne & Kane, 1978; ERA, 1979*a*, *b*; Healey, 1979). Discrete program counter chip sets are now mainly found in mini and mainframe computers where speed is a primary concern, whilst VLSI circuits dominate the microcomputer, embedded computer and instrumentation areas. Probes may be readily attached to discrete program counter integrated circuits, whilst VLSI devices present problems due to inaccessibility of their program counters. VLSI circuits may be considered as belonging to one of two groups. The first and largest group, microprocessors, are CPUs which usually do not contain any integral memory elements with the exception of registers. The second group—microcomputers and controllers—are CPUs with integral memory and sometimes I/O channels such as A/D conversion devices. As microprocessors require external memory elements, their memory address lines are always available for probing. In contrast, microprocessor circuits contain integral memory and rarely have their associated memory address lines externally available and are therefore unsuitable for monitoring by SAS. The majority of microprocessors provide qualificatory signals to indicate instruction cycle fetches (see Table 1) and where these are provided they are used to gate the memory bus data to provide an effective program counter. As described earlier, where no instruction cycle qualificatory signals are provided window operations may be used to isolate the relevant data.

3.1.2 *Software*. Programs may be written in a number of different languages the characteristics of which occupy a spectrum from those low level languages which reflect the computing machine's architecture to high level languages whose affinity is to the problem (McIntine, 1978; Calingaert, 1979). The program environment may vary from a simple single program situation common to many microprocessor and embedded systems

Table 1

Microprocessor	Manufacturer	No. of bits	No. of pins	Instruction cycle qualifying pins No.	Name
8080A	Intel	8	40	18, 19, 20, 21	STO-ST3
8085A	Intel	8	40	From data bus during T2	
Z80A	Zilog	8	40	29, 33	SO, S1
MC6800	Motorola	8	40		
MCS6502	MOS Tech.	8	40	7	SYNC
2650A	Signetics	8	40		
CDP18020	RCA	8	40	6, 5	SCO, SC1
SC/MP	Nat. Semi.	8	40	From data bus at beginning of input cycle	
TMS9980	Texas Inst.	8	40	3	AQ
IM6100	Intersil	12	40	36	IFETCH
INS8900	Nat. Semi.	16	40		
CP1600	Gen. Inst.	16	40		
TMS9900	Texas Inst.	16	64	7	IAQ
TMS9995	Texas Inst.	16	40	16, 20	IAQ, $\overline{\text{MEMEN}}$
8086	Intel	16	40	26, 27, 28	SO-S2
Z8002	Zilog	16	40	21, 20, 19, 18	STO-ST3
Z0001	Zilog	16	48	23, 22, 21, 20	STO-ST3
9440	Fairchild	16	40	6, 8	00, 01
F100-L	Ferranti	16	40	4	$\overline{\text{IR2}}$

to complex multiprogrammed, timeshared and paged systems found in large data processing systems (Anderson, 1981). The present configuration of SAS is designed to monitor the execution of single program systems machine coded from either a compiler, assembler or by hand, common to instrumentation, embedded and engineering applications.

3.1.3 *Speed.* A feature of the structure extraction technique is that the sequence of nodal branch data acquisition is irrelevant as structural data is independent of execution sequence. Thus, if the monitored program forms a closed loop, as is the case with most embedded or real time control systems, the instruction address register can be statistically sampled rather than traced in real time without incurring loss of structural data. This means that the monitoring system can be of slower speed than the monitored system.

3.2 *An example application: Location of the hexadecimal word output routine XOP 10 associated with the Texas Instruments TM990/101-1 and TM990/401-3 microcomputer and monitor*

Using SAS there are two principal methods which may be used to determine the memory position of XOP 10. For clarity any interaction with XOP 12 is ignored.

(i) The first method entails writing two trivial programs, one which includes XOP 10, the second which is identical except that it does not contain XOP 10. These programs are shown in Figure 8(a, b). Note that NOP, no operation, is used to replace XOP 10.

The procedure then is:

FE00	0300	LIMI	>0000	Interrupt
FE02	0000			
FE04	02EO	LWPI	>FE80	Workspace
FE06	FE80			
FE08	020C	LI	R12,>0080	CRU, main serial port
FEOA	0080			
FEOC	0201	LI	R1,>0000	Output data
FEOE	0000			
FE10	1000	NOP		Dummy instruction
FE12	1F15	TB	21	Has key been pressed?
FE14	16FD	JNE	>FE10	No, continue
FE16	0460	B	@>0080	Yes, GOTO monitor
FE18	0080			

Figure 7(b). Test program without XOP.

Source	Destination
FE14	FE10
036A	FE12
FE14	FE10
0358	035E
FE10	0348
0368	034E
0358	035E
0368	034E
036A	FE12
0368	034E

Figure 7(c). Structure table 1 after acquisition of vectors from program (a).

Source	Destination
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10
FE14	FE10

Figure 7(d). Structure table 2 after acquisition of vectors from program (b).

Source	Destination
0000	0000
036A	FE12
0000	0000
035B	035E
FE10	0348
0368	034E
0358	035E
0368	034E
036A	FE12
0368	034E

Figure 7(e). Vectors present in both structure tables are eliminated from structures table 1 leaving only vectors associated with the XOP.

Source	Destination	Frequency
0358	035E	0002
0368	034E	0003
036A	FE12	0002
FE10	0348	0001

Figure 7(f). Sorting the vectors makes it easier to identify the lowest and highest XOP routine values 0348, 036A which represent the entry and exit points.

Source	Destination
0368	034E
0368	034E
FE10	0348
0358	035E
FE14	FE10
036A	FE12
0368	034E
0358	035E
FE14	FE10
036A	FE12

Figure 7(g). Structure table 1 after acquisition of vectors from program (a).

Source	Destination
0000	0000
0000	0000
FE10	0348
0000	0000
0000	0000
036A	FE12
0000	0000
0000	0000
0000	0000
036A	FE12

Figure 7(h). Structure table 1 after application of MO = 30.

Source	Destination	Frequency
035A	FE12	0002
FE10	0348	0001

Figure 7(i). The contents of structure table 1 sorted and listed using commands SV and LV.

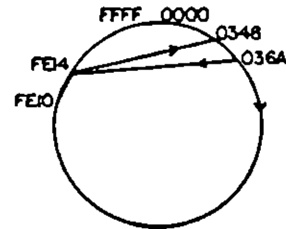

Figure 7(j) The dynamic structure map of program (a) as presented on the system display.

4. Conclusion

Program maintenance dominates the cost associated with the software life cycle. Research in this area and that of program execution monitoring is sparce. Escalating software costs make the research for new tools to increase software productivity increasingly urgent. The majority of existing hardware tools place an emphasis on program debugging and often either are very specialized or require the programmer to possess detailed knowledge of the machine to apply or interpret the results. In contrast, SAS is concerned with monitoring, analysing and presenting fundamental program properties which address program design and maintenance rather than debugging. It achieves this criterion by using a universal hardware technique to extract the dynamic structure of the software. A method based on directed graphs is used to provide a display particularly suitable for small instrument screens. It is proposed that such techniques could either be integrated into a new generation of logic analysers or as part of a universal test tool for computer programmers.

References

Allison, A. 1980. Follow three simple rules to improve software productivity. *EDN*, March, 167–171.

Anderson, D. A. 1981. Operating systems. *IEEE Computer*, June, 69–82.

Antoine, J. M., Decaesteke, P. & Wallstein, R. 1979. Effective software debugging using a program tracer. *Electrical Communication*, 54(2), 111–114.

Armbruster, C. E., Duke, A. H. & Dunbar, R. G. 1978. Hardware Sampler for system measurement. *IBM Technical Disclosure Bulletin*, 21(4), September, 1427–1429.

Boehm, B. W. 1976. Software engineering. *IEEE Transactions on Computers*, December, 1227–1241.

Calingaert, P. 1979. *Assemblers, Compilers and Program Translation*. London: Pitman.

Electrical Research Association 1979*a*. *Microprocessors: Their Development and Application*. ERA Technology.

Electrical Research Association (b) 1979*b*. *The Engineering of Microprocessor Systems*. Oxford: Pergamon Press.

Fryer, R. E. 1973. The memory bus monitor. *AFIPS Conference Proceedings, National Computer Conference*, 42, 75–79

Gries, D. 1980. Current ideas in programming methodology. In *Research Directions in Software Technology*, (P. Wegner, ed.), pp. 255–275. Amsterdam: North Holland.

Healey, M. 1979. *Minicomputers and Microcomputers*. London: Hodder and Stoughton.

Jensen, R. W. 1981. Structured programming. *IEEE Computer*, March, 31–48.

Krummel, L. 1977. Advances in microcomputer development systems. *IEEE Computer*, February, 13–19.

Lemon, L. M. 1979. Hardware system for developing and validating software. *Proceedings of 13th Asilomer Conference on Circuits, Systems and Computers*, Pacific Grove California USA, 5–7 November, pp. 455–459.

Lloyd, R., Ovies, H., Rosado, J. L. & Wilson, D. J. 1980. Programmable map and trace instrument. *IBM Technical Disclosure Bulletin*, 23(5), 2075–2078.

Marshall, J. S. 1978. Logic analysers provide an essential real-time view of digital system activity. *Proceedings of MIDCON Technical Conference*, Dallas, USA, 12–14 December, pp. 31–34.

McCabe, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320.

McCabe, T. J. 1978. Software complexity measurement. *Proceedings of 2nd Software Life Cycle Management Workshop*, Atlanta, USA, 21–22 August, pp. 186–190.

McIntine, T. C. 1978. *Software Interpreters for Microcomputers*. New York: John Wiley.

Mills, H. D. 1980. Software development. In *Research Directions in Software Technology* (ed. P. Wegner), pp. 87–105. Amsterdam: North Holland.

Munson, J. B. 1981. Software maintenability, practical concern for life cycle costs. *IEEE Computer*, November, 103–109.

Nutt, G. J. 1975. Tutorial, computer system monitors. *IEEE Computer*, November, 51–61.

Osbourne, A. & Kane, J. 1978. *An Introduction to Microcomputers Vol. 2, Some Real Microprocessors*. California: Osbourne and Associates Inc.

Plattner, B. & Nievergelt, J. 1981. Monitoring program execution—a survey. *IEEE Computer*, November, 76–93.

Schindler, M. 1981. *Software, technology forecast. Electronic Design*, January, 190–199.

Stockham, T. G. 1965. *Some methods of graphical debugging. Proceedings of IBM Scientific Computing Symposium on Man Machine Communication*, pp. 57–71.

Thornton, C. 1980. How to get the best performance from your system. *Data Processing*, January, 29–32.

Williams, G. 1981. Structured programming and structured flowcharts. *BYTE*, March, 20–34.