ISSN 0254-2757

QUÆSTIONES INFORMATICÆ

Volume 6 • Number 3 November 1988

The official journal of the Computer Society of South Africa and of the South African Institute of Computer Scientists

Die amptelike vaktydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUÆSTIONES INFORMATICÆ

The official journal of the Computer Society of South Africa and of the South African Institute of Computer Scientists

Die amptelike vaktydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Production

Mr Q H Gee
Department of Computer Science
University of the Witwatersrand
Johannesburg
Wits
2050

Subscriptions

The annual subscription is

	SA	US	UK
Individuals	R20	$7	£5
Institutions	R30	$14	£10

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Quæstiones Informaticæ is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Experiences in CSP Trace Generation

D G Kourie and R J van den Heever
Department of Computer Science, University of Pretoria, Hatfield, Pretoria 0083

Abstract

Experiences and insights gained from implementing and using a Prolog trace generator for CSP specifications are discussed. The use of Prolog as an implementation language is evaluated. The value of trace generation as both an educational tool and a practical software development tool are considered.
Keywords: *Trace, trace generation, Prolog, specification, verification, validation, software development tool*
Computing Review Category: *D.2.2*

Received August 1988, Accepted October 1988

1. Introduction

In [8] the design and implementation in Prolog of a trace generator for CSP specifications was described in some detail. Since then, the generator has been used in various contexts, and some efforts at enhancements have been made. This has provided several insights into the nature of traces, and their value and drawbacks in software development. The purpose of this paper is to outline these experiences and insights.

In the section 2. of this paper a brief review of CSP is given in order to define terminology. This is followed in section 3. by an overview of the Prolog trace generator. Section 4. then gives a critical evaluation of trace generation, both with respect to the particular Prolog implementation, and from a wider perspective.

2. Overview of CSP

The CSP language is intended to describe the behavior of processes which communicate (or interact) with one another. A process thus described is said to have been defined in CSP. Such a definition involves references to interactions (or events) between the process and its environment, and also to subprocesses of the process defined. When an event occurs, a process evolves (or enters) a subprocess, which may in turn be defined in CSP.

Events are considered to be atomic and synchronous. A trace of a process is any finite event sequence which may be exchanged between the process and its environment over an arbitrary time period. Every subtrace, including the empty subtrace, is also considered to be a trace of the process. The entire set of traces which a process may potentially accept is called its trace set, and the set of events in the trace set is called the process's alphabet. One way of defining a deterministic process is to explicitly specify its trace set, as well as its alphabet; defining non-deterministic processes requires additional information.

CSP is a notation which implicitly and compactly defines the trace set of a process. A subset of its operators are used to express non-determinism. The language consists of a set of symbols to represent events in the alphabet of a process, a set of symbols to represent (sub)processes and a set of operator symbols some of which allow for the expression of non-determinism. A definition of a process involves valid strings of these symbols.

3. The trace generator

The Prolog trace generator code and its input are placed in two separate files called the generator file and the specifications file respectively. Once these files have been consulted into the Prolog environment, and the generator has been activated, the user is prompted for the name of a process to be traced. Resulting output appears on screen, but can easily be directed to a file (e.g. using the Prolog tell predicate).

3.1 Specifications file

Appendix A gives the BNF for the variation of CSP which is required in the specifications file. Operator symbols differ slightly from the original CSP notation [5] to accommodate keyboard and Prolog implementation peculiarities. The following table summarises the semantics of CSP constructs implemented by the trace generator. The symbol m denotes a main process name, p and q denote process names, e and f denote

Experiences in CSP Trace Generation

D G Kourie and R J van den Heever
Department of Computer Science, University of Pretoria, Hatfield, Pretoria 0083

Abstract

Experiences and insights gained from implementing and using a Prolog trace generator for CSP specifications are discussed. The use of Prolog as an implementation language is evaluated. The value of trace generation as both an educational tool and a practical software development tool are considered.
Keywords: *Trace, trace generation, Prolog, specification, verification, validation, software development tool*
Computing Review Category: *D.2.2*

Received August 1988, Accepted October 1988

1. Introduction

In [8] the design and implementation in Prolog of a trace generator for CSP specifications was described in some detail. Since then, the generator has been used in various contexts, and some efforts at enhancements have been made. This has provided several insights into the nature of traces, and their value and drawbacks in software development. The purpose of this paper is to outline these experiences and insights.

In the section 2. of this paper a brief review of CSP is given in order to define terminology. This is followed in section 3. by an overview of the Prolog trace generator. Section 4. then gives a critical evaluation of trace generation, both with respect to the particular Prolog implementation, and from a wider perspective.

2. Overview of CSP

The CSP language is intended to describe the behavior of processes which communicate (or interact) with one another. A process thus described is said to have been defined in CSP. Such a definition involves references to interactions (or events) between the process and its environment, and also to subprocesses of the process defined. When an event occurs, a process evolves (or enters) a subprocess, which may in turn be defined in CSP.

Events are considered to be atomic and synchronous. A trace of a process is any finite event sequence which may be exchanged between the process and its environment over an arbitrary time period. Every subtrace, including the empty subtrace, is also considered to be a trace of the process. The entire set of traces which a process may potentially accept is called its trace set, and the set of events in the trace set is called the process's alphabet. One way of defining a deterministic process is to explicitly specify its trace set, as well as its alphabet; defining non-deterministic processes requires additional information.

CSP is a notation which implicitly and compactly defines the trace set of a process. A subset of its operators are used to express non-determinism. The language consists of a set of symbols to represent events in the alphabet of a process, a set of symbols to represent (sub)processes and a set of operator symbols some of which allow for the expression of non-determinism. A definition of a process involves valid strings of these symbols.

3. The trace generator

The Prolog trace generator code and its input are placed in two separate files called the generator file and the specifications file respectively. Once these files have been consulted into the Prolog environment, and the generator has been activated, the user is prompted for the name of a process to be traced. Resulting output appears on screen, but can easily be directed to a file (e.g. using the Prolog tell predicate).

3.1 Specifications file

Appendix A gives the BNF for the variation of CSP which is required in the specifications file. Operator symbols differ slightly from the original CSP notation [5] to accommodate keyboard and Prolog implementation peculiarities. The following table summarises the semantics of CSP constructs implemented by the trace generator. The symbol m denotes a main process name, p and q denote process names, e and f denote

events, *X* denotes a variable, *s* is as set name and *c* is a channel name.

Construct	Meaning
m := q.	*m is defined by q*
e --> p	*e then p*
e --> p / f --> q	*e then p choice f then q*
p <> q	*p choice q*
p /-/ q	*p or q (non-deterministic)*
p // q	*p concurrent with q*
p /// q	*p interleaved with q*
p ~>q	*p interrupted by q*
p ; q	*p followed by q*
p < b *> q*	*If b then p else q*
p \ s	*p without elements of s*
X:s --> p(X)	*X from s then p*
c % e	*output e on channel c*
s ? X	*input X on channel s*
stop	*deadlocked process*
skip	*process terminating successfully*

The interested reader is referred to [5] for the precise semantics of these operators, and for valid CSP expressions as a whole.

Certain constructs may only be used in conjunction with appropriate set declarations in the specifications file. Such declarations consist of a Prolog structure composed of a functor and 2 parameters. The first parameter is a set name and the second is a list of events. For the constructs, *p\s, X:s-->p(X)* and *s?X*, Appendix A shows the corresponding functors as *hide, sort* and *chan_alpha* respectively. In each case, the list denotes the set of events to be hidden, the set of events to be chosen from, or a channel alphabet respectively. The set name, *s*, is an appropriate name chosen by the user. In the case of the construct *p//q* the alphabets of both *p* and *q* must be specified, using *alpha* as the functor, *p* and *q* as the set names, and the alphabets of *p* and *q* in the respective lists.

3.2 Generator output

The output of the trace generator for a given process is the set of all *recursion traces* of that process, as defined below. It also indicates where subprocesses that have been explicitly named in the specification file are entered, as the process rolls forward in accepting a given trace.

Note that in accepting a trace *t*, a process p_0 will evolve into a sequence of subprocesses, some of which may be explicitly named, while others are parenthesised process definitions. (Refer to the production for <*proc*> in Appendix A.) Denote the sequence of explicitly named processes (including the initial process p_0) by $<p_0,p_1..p_n>$. Now *t* will be called a recursion trace of p_0 if either of the conditions 1 to 3 below hold:

1. All processes in $<p_0,p_1..p_{n-1}>$ are different but $p_n = p_k$ for some $k = 0,..n-1$.
2. 1. does not hold, but p_n is either the *stop* or *skip* process, or a named but undefined process in the specifications file.
3. p_0 is defined as q_1 *op* q_2, where:
   - q_1 and q_2 are named processes
   - op is either ; or // or ///
   - t_1 and t_2 are recursion traces of q_1 and q_2 respectively
   - *t* is the result of combining t_1 and t_2 as prescribed by *op*.

It can easily be shown that the output of the generator (in terms of recursion traces interspersed with indications of entry points to subprocedures) provides sufficient information to deduce the entire trace set of the traced process. Of course, this is subject to the restriction that all subprocesses have to be explicitly defined in the specifications file.

3.3 Generator code

The overall structure of the generator code has been discussed in some detail in [8], and will not be repeated here. It is observed, however, that the generator regards the specifications file as Prolog facts. This is accomplished, inter alia, by defining as Prolog operators various symbols and operators used in the BNF in appendix A. Furthermore, lists, process names, channel names, etc. are all required to be valid Prolog constructs.

Another feature of the generator is its ability to perform arithmetic operations on the parameters of process names. This is important when specifying and generating traces for recursive-type definitions such as :
p(X) := e --> p(X+1).

4. Critical evaluation

The trace generator has been through a number of minor revisions, and has been used in a variety of contexts. The purpose of this section is to outline some of the lessons learned and experiences gained in this process. In 4.1 issues relating specifically to the fact that the generator has been implemented in Prolog are addressed. In 4.2 some observations about trace generation in general are made.

4.1 Prolog issues

In this section, three issues in relation to the use of Prolog as an implementation language for the trace generator are considered, namely its portability, flexibility and efficiency.

Portability

The trace generator was implemented on a mainframe using Waterloo Prolog, version 1.7. Except for the use of operators, the code adheres to Core Prolog described in [3]. However, in porting the code onto various Core Prolog PC implementations, the operators proved to be the main difficulty. In A.D.A. Prolog it was not possible, for example, to use symbols such as /, ; and \ as operators, and alternatives had to be chosen. Similar problems have arisen with Arity Prolog. As a consequence, the contents of the specifications file looks less like the original CSP notation than the initial mainframe implementation.

Flexibility

Prolog provides considerable flexibility in a number of dimensions for the trace generation problem space. To date it has been found relatively easy to enrich the operator set as the requirements arise, without disturbing the overall structure of the code. Thus, the ability to handle constructs such as *p <* b *> q* have been easily added to the original code. Other enhancements include the ability to handle the channel input and output notations (*c?X* and *c%e*) and enriching the allowable range of arithmetic operations performed on arguments of process names.

Another area of flexibility relates to the contents in the specifications file. Because this is actually Prolog code, a Prolog programmer can enrich the code in a number of ways. For example, process definitions need not necessarily be given as facts, but can be stated as rules. Hence it is perfectly legitimate to include process definitions of the form :

proc(X) := ..... :- condition1(X).
: :
proc(X) := ..... :- conditionN(X).

Because the conditions tested by condition1 to conditionN need not necessarily be mutually exclusive, the foregoing is more general than an *if..then..else* construction and can in fact form the basis for the implementation of LOTOS guards [6] - something not provided for in the definition of CSP [5]. In a similar way, an set declaration need not be limited to an instantiated list in its second parameter, but can be constructed by the Prolog code thus :

alpha(proc, List) :- make_list(List).

where *make_list* generates the required *List* in some or other way (e.g. recursively, or interactively with the user).

Finally, with minor adaptations to the driver procedures, the generator can be used to ask existence questions about recursion traces. The simplest such question would be to establish whether a given event sequence constitutes a trace for a given process. More sophisticated questions would be to ask for a recursion trace (or traces) of a given length, or containing a given subtrace, or containing a given number of occurrences of a given event, etc. Disregarding efficiency matters, the limitations here are determined by the ingenuity and effort available for writing Prolog list manipulation procedures.

Efficiency

Prolog is generally acknowledged to be less efficient in terms of time and space than conventional programming languages. From a practical point of view, the inefficient utilisation of space appears to be the more serious problem, in that stack space overflow errors occur whenever the number of recursion traces implied by the specification becomes too large. The problem can be slightly alleviated by implementing a number of well-known space optimising strategies. For example, difference lists [1] can be used for concatenation when dealing with the followed by operator, and careful use of the cut predicate can avoid unnecessary backtracking.

However, such measures merely postpone space/time problems, allowing specifications which are only slightly larger to be traced. The complexity inherent in the problem of generating all recursion traces is such that efficiency problems seem inevitable at some point, irrespective of implementation language and hardware. These matters will be addressed below.

4.2 General issues

Theoretical considerations

The space / time complexity of any algorithm which seeks to generate all recursion traces for a process is directly related to the number of recursion traces which have to be generated. This number tends to explode combinatorially when certain operators are used. The worst offender in this regard is the interleaving operator.

Consider, for example, a process defined as *p := q /// r.* and let *n(.)* and *l(.)* be functions mapping processes to the associated number of recursion traces and average recursion trace length (rounded to the nearest integer) respectively. If a recursion trace of *q* of length *l(q)* is interleaved with a recursion trace of *r* of length *l(r)*, then recursion traces of *p* will be obtained of length *l(q)+l(r)*. Such an interleaving can take place in ${}^{l(q)+l(r)}C_{l(q)}$ different ways. Hence, an order of magnitude estimate of *n(p)* and *l(p)* are respectively given by :

$n(p) = n(q)*n(r)*^{l(q)+l(r)}C_{k(q)}$, and
$l(p) = l(q) + l(r)$. Thus, not only does the number of interleavings grow combinatorially with respect to the lengths of recursion traces of contributing processes, but the average length of the new recursion traces also increases. Consequently, if more than two processes are joined by means of the interleaving operator the number of recursion traces can quickly become unmanageable.

The foregoing expressions also form approximate worst case upper bounds for $n(p)$ and $l(p)$ when $p := q \,//\, r$. The number of recursion traces implied by the concurrency operator are constrained in proportion to the number of synchronising events which have to match in the underlying interleavings of the traces of q and r. Whenever a set of interleavings have a common prefix succeeded by non-matching synchronising events they 'degenerate' into a single concurrency recursion trace - namely a trace consisting of the prefix followed by deadlock.

Clearly, then, the liberal use of interleaving and concurrency operators can easily result in specifications for which the problem of finding all recursion traces is not practical.

Other CSP operators which also lead to large numbers of traces, but are somewhat less offensive than the interleaving and concurrency operators are the interrupt operator, and the use of the set choice and input notations. In the latter two cases, the number of recursion traces is dependent on the size of the associated event sets. In the former case, if $p := q \sim> r$, then $n(p)$ is approximately $l(q) * n(q) * n(r)$.

From a theoretical point of view, then, the following claims regarding computability and tractability may be made. (Cf. [4] for an explanation of these concepts.) These claims assume that alphabets of processes are finite, and that traces are not represented in some closed form (such as using t^* to denote a trace consisting of 0 or more concatenations of trace t).

- Since traces may potentially be infinite in number, the general problem of trace set generation is clearly noncomputable.
- By limiting the problem to the generation of recursion traces of processes the problem appears to be computable but intractable.
- Hence the problem of inferring the trace set of a process from the recursion trace set is itself noncomputable.

The claim, therefore, that the trace set of a process can be inferred from the set of recursion traces should be seen in this light. In principle, a generalised algorithm can be written to generate all traces up to a given length from the recursion traces of a process. However, it is not possible to write a generalised algorithm to generate all traces.

The problem of whether a generalised algorithm can be written to generate the trace set of any process in a closed form from the process's recursion traces is a matter for further study.

Practical considerations

In the light of the foregoing it might seem that a trace generator can at best serve as a teaching tool, dealing with toy problems only. However, in many applications recourse to reduction and segmentation strategies render it a practical tool for software development. These strategies will now be discussed.

Often the dimensions of a problem can be reduced without losing any of its essential characteristics. For example, the dining philosopher problem defined in [5] required the use of 9 concurrency operators (8 of which act as interleaving operators) if there are 5 philosophers. If the problem is reduced to 2 philosophers, the number of concurrency operators reduces to 2 (1 of which acts as an interleaving operator). While such a reduction in the dimensions does not necessarily replace the need for formal arguments to guarentee liveness and/or other properties in the larger system, the resulting recursion traces may well provided insights as to how these arguments can be made. Indeed, the very process of thinking about how to reduce the dimensions serves to fix the intellect on important characteristics of the problem. In a similar vein, it has frequently been found that when using the choice set notation, the number events in the associated event set may be drastically reduced without losing useful information.

Another way in which recursion traces may be limited is by tracing only parts of a system at a time, i.e. by segmenting the problem. Consider, for example, the specification $p := q \sim> r$. Knowing a priori that r interrupts q, it may often be more useful to know what the individual traces of q and r look like separately, rather than to know exactly where r's traces interrupt those of q. To take a concrete example, if $n(q) = 10$, $l(q) = 5$, and $n(r) = 10$, then separate traces of q and r result in $n(q) + n(r) = 20$ traces versus a total of approximately $n(q) * n(r) * l(q) = 500$ for all those of p.

Clearly, even more dramatic reductions occur when segmentation is applied to processes using the interleaving operator. Note, however, that segmentation tends to loose its value when the process to be segmented is itself a subprocess of some larger process. This tends to be the case

with the interleaving operator, as in the dining philosophers problem. In isolation, the interleaving operator merely indicates that the argument processes function totally independently, and the resulting traces are usually only of limited value.

Nevertheless, an analyses of the segmented recursion traces frequently enhances insight into a specification. Consequently, it may be worthwhile to design a generator which functions in two modes, one which generates all recursion traces, and one which automatically applies segmentation, based on selected operators, to components of a specified process.

Software development

Frequently cited properties of high quality software include such characteristics as maintainability, reliability, readability, extendability, etc. (See, for example, [7].) Methodological approaches which promote these qualities include stepwise refinement, designing the solution space as close a possible to the problem space, and unambiguous specification of the problem. CSP is well-suited to support these methodological approaches, particularly in the domain of real-time interactive systems. The trace generator in turn, tends to encourage and promote the use of CSP in a number of ways.

The trace generator may be used (reducing and segmenting the problem where possible) to test the validity of CSP definitions. This leads into an interactive process, where visual representation of the recursion traces implied by a proposed specification invariably results in revisions, until a satisfactory specification is obtained. Since the meaning of traces tends to be easier to understand than the CSP specification as such, it has been found that end-users may frequently be involved in this interactive validation process at the level of functional specifications. At more detailed levels of specification, traces tend to facilitate unambiguous communication between various parties in the software development team.

Note that a CSP specification at the functional level typically describes, at a high level of abstraction, how an envisaged system should interact with the environment. Small subcomponents of the system and their interaction are not at issue. Hence, the events in the traces tend to be input and output information - in effect a statement of what output should result from a given set of input events. A stepwise refinement methodology involves successively refining the functional specification into increasingly detailed CSP specifications at the design level. The CSP concealment operator provides an approach to check the consistency of a refined specification with one at a higher level of abstraction. (Cf. [9] where a similar approach is followed for ESTELLE specifications.) In principle, an extension to the trace generator could provide for the automation of such consistency checking.

Once tracing has taken place at a given level of abstraction, not only has the specifier been forced to continually and critically re-evaluate his initial perception of the problem; there is the bonus of having a set of recursion traces for various segments of the specification which serve as the basis for test scenarios in the final phases of software development. Consequently testing turns out to be much more structured than previously.

5. Conclusion

The value one attaches to a trace generator critically depends on the value attached to formal specification in general. The case for formal specification is well-documented in the literature and will not be repeated here. Neither will the relative merits of the various formal specification techniques be evaluated, apart from the claim that CSP provides fairly easy entree to the general domain. (See, for example, [2] or [10] for evaluation criteria.)

Experience with the CSP trace generator has shown it to be an excellent aid for introducing CSP to the novice specifier. It has successfully been used to teach CSP not only to Computer Science students, but also to experienced software engineers working on projects within the R&D section of Pretoria University's Computer Science department. This has resulted in the evolution of a software development methodology which incorporates a CSP definition of critical parts of the problem as part of the functional specifications of each project.

In summary, then, recursive trace generation, though intractable in the general case, has proved to be a practical aid both for teaching and for software development.

References

[1] I. Bratko, [1986], *Prolog Programming for Artificial Intelligence*, Addison-Wesley.

[2] N.D. Birrel and M.A. Ould, [1985], *A Practical Handbook for Software Development*, Cambridge University Press.

[3] W.F. Clocksin and C.S. Mellish, [1984] *Programming in PROLOG*, Springer-Verlag.

[4] D. Harel, [1987], *Algorithmics : the Spirit of Computing*, Addison-Wesley.

[5] C.A.R. Hoare, [1985], *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science.
[6] ISO/TC 97/SC 21/WG 1/N1573 /DP 8807, [1986], *Information processing systems - open systems interconnection - LOTOS - a formal description technique based on observational behaviour'*.
[7] B. Kitchenham, [1987], *Software Quality Modelling, Measurement and Prediction*, Software Engineering Journal, **2**(4), 105-113.
[8] D.G. Kourie, [1987], The Design and Use of a Prolog Trace Generator for CSP, *Software - Practice and Experience*, **17**(7), 423-438.
[9] H. Ural and R.L. Probert, [1986] Step-wise Validation of Communication Protocols and Services', *Computer Networks and ISDN Systems*, **11**, 183-202.
[10] P. Zave, [1988], *Assessment*, Software Engineering Notes, **13**(1), 40-43.

Appendix A

KEY:
1. @ is used for choice
2. *{...}* is used for 0 or more occurences
3. Symbols enclosed in angle brackets are non-terminal. Exceptions are <> and <* and *> which are terminal symbols
4. Prolog terms etc. are also terminal symbols

<definition> = <proc_name> := <proc_def> .
{<subproc_def>}
{<set_declare>}

<proc_def> = <proc> <proc_op> <proc> @
<proc_name> // <proc_name> @
<guarded_exp> { | <guarded_exp>} @
<proc> \ <set_name> @
<set_choice_expression>

<subproc_def> = <definition>

<set_declare> = alpha(<set_name>, <list>) .@
sort (<set_name>, <list>) . @
chan_alpha (<set_name>, <list>) . @
hide (<set_name>, <list>) .

<proc> = <proc_name> @
(<proc_def>)

<proc_op> = <> @
/-/ @
/// @
~>@
; @
< <condition> *>*

<guarded_exp> =
<event> --> {<event> --> } <proc>

<set_choice_expression> =
<variable> : <set_name> --> <proc>

<event> = any valid prolog term @
<chan_name> % any valid prolog term @
<chan_name> ? <variable>

<list> = any valid prolog list of terms
<proc_name> = any valid prolog structure @
any valid prolog atom
<chan_name> = any valid prolog structure @
any valid prolog atom
<set_name> = any valid prolog structure @
any valid prolog atom
<condition> = any valid prolog goal
<variable> = any valid prolog variable

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles and exploratory articles of general interest to readers of the journal. The preferred languages of the journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:

Professor ~~J M Bishop~~ D.C. Kaire
Department of Computer Science
University of the Witwatersrand
Johannesburg
Wits
2050

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review Categories.

Manuscripts may be provided on disc ~~using any Apple Macintosh package or~~ in ASCII format, once that a submitted paper has been accepted.

For authors wishing to provide camera-ready copy, a page specification is freely available on request from the Editor.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Photographs as illustrations should be avoided if possible. If this cannot be avoided, glossy bromide prints are required.

Charges

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

[1] E. Ashcroft and Z. Manna, [1972], The Translation of 'GOTO' Programs to 'WHILE' programs, *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.

[2] C. Bohm and G. Jacopini, [1966], Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.

[3] S. Ginsburg, [1966], *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimise the risk of the author's contribution having to be held over to a later issue.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion ~~of recent problems~~.