STRUCTURED PROCESS DESCRIPTION

by

Fred M. Tonge*
Robert S. Barton+
Richard M. Cowan+

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

March 1979

Technical Report #130

*University of California, Irvine
+Burroughs Corporation

This work was supported in part by the Burroughs Corporation and by National Science Foundation grant no. MCS77-02715. The authors acknowledge the helpful comments of R. Flint, K. Gostelow, and R. Thomas.

Abstract

Structured process description employs a small set of constructs which facilitate functional programming with concurrency. These constructs can be mapped directly onto a process interconnection architecture. Using these structured connectives, maximum pipelining throughput can be achieved. The constructs are presented and illustrated with example programs.

Index terms: structured processes, dataflow, multiprocessing, concurrency, parallel processors, distributed processing, functional programming.

1. Introduction

Structured process descriptions are a means of talking about functional programming emphasizing concurrency. In this paper we first present an informal discussion intended to motivate the approach, followed by formal definitions of relevant concepts and process structuring constructs, together with an illustrative example. Next are sections discussing measures of concurrent performance of structured process description, data structures, and the process allocation environment. Finally, several additional functional forms are introduced, and more example process descriptions are given.

2. Informal Discussion

Informally, a process is a device (machine, algorithm, mathematical function, ...) which when presented with a set of input values later produces a set of output values. The relationship between an input set and the corresponding output set expresses the function of the process. This relationship is described by a process description, which serves as a blueprint for the process. The process is realized from its process description by allocating resources, such as processing elements, to the component parts of the process.

Structured process description is motivated by a desire to exploit the potential for concurrency inherent in possible LSI-based system architectures involving large numbers of microprocessors. Our emphasis is on expressing process descriptions such that there is a natural mapping of process descriptions onto processors at many levels. As such, this work is related (and complementary) to work on functional programming languages [BA78,BU75,LA64,MC60], dataflow [AR77a,DE75]], and structured programming [DA72,DI76,KO74]].

Our underlying model of a computer system architecture includes:

> hierarchical decomposition (a processor-storage module is composed of some structural interconnection of processor-storage modules, down to some atomic level);
>
> functionality (at any level, the outputs of a process are expressions of the inputs; i.e., there are no "side effects" of a process);
>
> data-driven (a process is initiated when its inputs are available, and no other form of synchronization among processors is used).

In keeping with this model of a processing element (p.e.), process descriptions are themselves hierarchically decomposed into interconnected process descriptions, down to some atomic level. Functionality dictates the order of process activation since the outputs of one expression are the inputs of the next.

Structured process descriptions describe the interconnection of processes, not the interconnections of the processing elements on which they are realized. The basic principles are applicable across a broad range of specific p.e. capabilities and interconnection architectures. In this discussion we do not specify a particular p.e. architecture, system interconnection architecture, or procedure for mapping process descriptions onto particular p.e.'s (although we introduce some assumptions for the sake of examples).

This approach to structured process description can be viewed either as specifying connectives for process interconnection or as specifying rules for function

composition. Process descriptions can be written as network diagrams or as equivalent functional expressions.

A complete specification of a system would include a language for expressions in which atomic processes (functions) are described, a set of inter-process connectives (metafunctions), and a scheme for the allocation of processes to a network of p.e.'s. In this paper we concentrate on the principles of process composition, and develop only such details of a language for expressions and an allocation scheme as are needed for examples.

We envisage the language for expressions, rules for functional composition, and allocation scheme all as being implemented in each p.e., and so executed as "machine language". Under these conditions, process descriptions would be realized as programs almost directly, subject only to the level of translation done in simple assemblers.

3. Definitions

A process description consists of a process expression and a process body. The process expression specifies the function of the process by giving either a base level computation rule or an expansion of the process into components. The process body consists of specifications of the input and output sets and of minimum resource requirements (including shared resources). For the remainder of this paper, we concentrate on process expressions and their input and output sets.

A processing element consists of processors and associated storage, typically interconnected with other processing elements in a regular manner.

A process is a process description and a set of resources (minimally, processing elements) allocated to the process description.

An input case is a complete set of values for the input set of a process. An output case is a complete set of values for the output set of a process. For each input case there is (assuming process termination) a corresponding output case. Together, these are a data case.

The domain of a process is the set of resources (for example, processing elements) assigned to the process. A process accepts (begins processing) a data case when that input case enters its domain, and produces (finishes processing) a data case when that output case leaves its domain. A process is active when any data case is within its domain.

Within a process, an input case may subdivide into several

data cases, eventually recombining into a single output case. A process is capable of parallelism if within its domain several data cases derived from the same input case can be processed concurrently. A process is capable of pipelining if it can accept a second (or further) input case before producing the first output case. A process can parallel or pipeline (or both) only if both its process description and its resource allocation permit parallelism or pipelining.

A process (and so its process description) is order-preserving if data cases are produced in the same order as they are accepted. This can be achieved trivially by not accepting a second input case until the current output case is produced; thus, order preservation is of interest only in the presence of pipelining. If a pipelining process is order-preserving, then the correspondence between input cases and output cases is directly derivable from the orders of acceptance and production. If not, additional case identification information is necessary to maintain that correspondence, plus some form of sorting to maintain order. In the following, only order-preserving processes are considered.

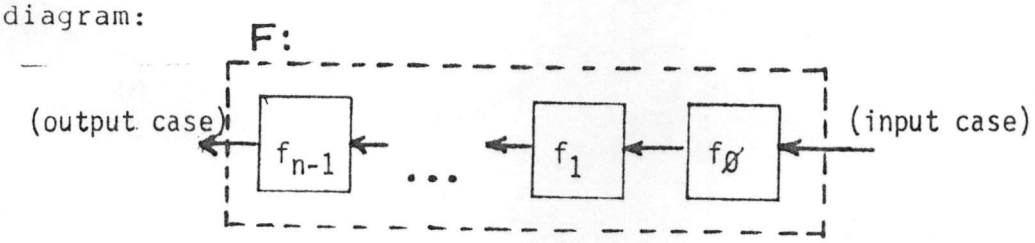
The allocation of processing elements to processes is done hierarchically on a data-driven basis. The immediate components of a process are allocated processing elements when the first input case is presented to the process.

4. Structural Connectives and Metafunctions

The decomposition of process expressions into their components (or, the synthesis of components into larger process descriptions) is specified by three structural connectives. Corresponding to each connective is an exactly equivalent metafunction. In the following we give for each connective a diagrammatic representation of the process expression and the equivalent metafunction.

In the serial connective, components occur in a specified order.

diagram:



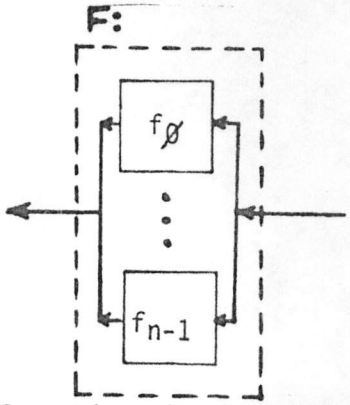
(Note that the flow in network diagrams is depicted right to left, corresponding to composition of functions.)

The equivalent metafunction is functional composition:

$$F = f_{n-1} \circ \ldots \circ f_1 \circ f_\emptyset$$

In the parallel connective, components occur (may be carried out for a particular data case) in any order, including simultaneously if sufficient resources are available.

diagram:

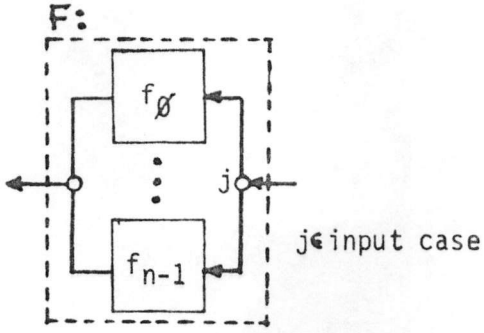


The equivalent metafunction is functional construction:

$$F = (f_\emptyset, \ldots, f_{n-1})$$

In the alternative connective, one of the component processes is performed as selected by an index.

diagram:



The equivalent metafunction is functional selection:

$$F = (f_\emptyset, \ldots, f_{n-1})_j$$

By convention, a predicate used as a selector evaluates to $\emptyset$ if false and 1 if true.

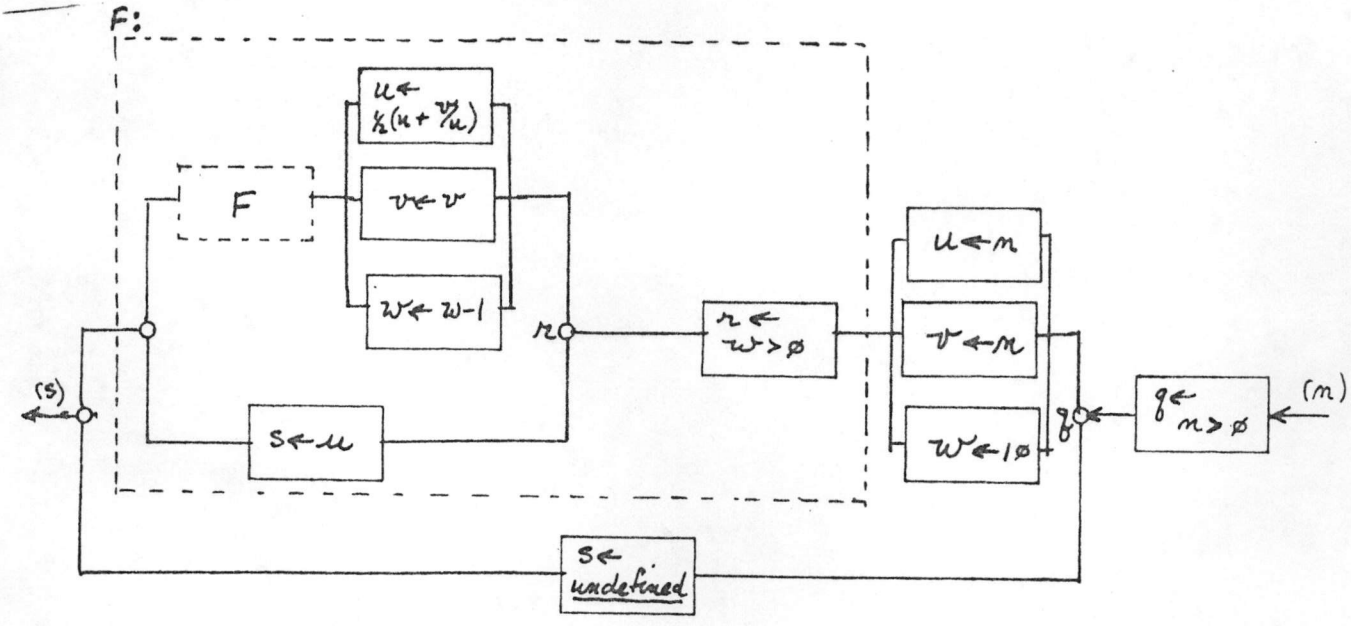
Definition of a function is specified as in the above examples. Reference to a function is indicated in diagrams as given below, and in expressions by use of the function name.

diagram:

As is shown elsewhere [TO78], structured process description networks derived using these connectives (and so functional expressions using the equivalent metafunctions) are equivalent to Dijkstra's D-charts with the addition of parallelism, and with loops achieved through the recursive use of function reference. Consequently, results concerning the power and applicability of that formalism hold also for structured process description.

5. An Example

The following procedure computes the square root of a number using the Newton-Raphson approximation, with ten iterations of the method and taking the number itself as the initial approximation. The procedure is represented as a structured process description network, with input n and output s. Each process is a simple expression evaluation. In this process description symbolic names are used as a convenient shorthand for describing elements of a data case; they in no way imply storage locations.



This process description implies no particular allocation of processes to p.e.'s. The entire process description could be allocated to a single p.e., where it would be executed as a sequential program; or it could be allocated to a network

of very small p.e.'s, each expression evaluation carried out by a distinct p.e.. The opportunities for parallelism and for pipelining in computing a sequence of square roots are directly indicated in the process description. The extent to which that concurrency can be achieved will depend upon the allocation scheme and available resources.

The equivalent functional notation for this process description is given below.

sqrt ((n)-->(s))

= (undefined,f•(n,n,1Ø))$_{n>Ø}$

where

f ((u,v,w)-->(s))

= (u,f•((u+v/u)/2,v,w-1))$_{w>Ø}$

The symbolic names used above stand for primitive selector functions, named by underlined integers, where i selects the i-th member of its input set. Thus, the above square root function could be expressed (much less legibly) as:

sqrt = (undefined,f•(1,1,1Ø)$_{1>Ø}$

where

f = (1,f•((1+2/1)/2,2,3-1))$_{3>Ø}$

While the network diagram form of process description is useful for reasoning about the properties of processes, and is often suggestive of potentials for concurrency, it is cumbersome for large process descriptions. In the remaining examples we use functional notation.

6. Data Structures

From the standpoint of inter-process data transfer, data structures are represented as bracketed sequences which are transmitted and processed serially.

For example, the list (or vector) of integers 1,2,3,5 could

be represented as (1 2 3 5). The two-by-two identity matrix could be ((1 Ø) (Ø 1)). The tree shown below could be represented as (a (b (e) c d (f g))).

The language of expressions must include at least a minimal set of operations for manipulating bracketed sequences. One such set would include (1) test if a data element is a sequence, (2) return first element of sequence, (3) return all but first element of sequence, (4) insert element at beginning of sequence, and (5) append element at end of sequence.

In later examples, we also use a syntactic shorthand to refer to specific sequence elements (x[i]) or subsequences (x[i:j]).

7. Measures of Performance

As stated above, a major consideration in structured process description is the exploitation of concurrency attainable through parallelism and pipelining. The structure of many (most?) large problems--e.g., matrix manipulations, business data processing--involve processing sequences of like data items. In such situations pipelining, as measured by the rate of production of the data items, may be a greater source of concurrency than parallelism.

In considering the amount of concurrency attainable by a process description, we are particularly interested in two measures of performance--the elapsed time to process a single data case, and the average throughput or rate of production of data cases.

It is easy to demonstrate an arbitrarily interconnected process description network with a smaller elapsed time than any equivalent (in terms of the computation performed) structured network. However, elapsed time is typically a measure of single case performance, and minimizing elapsed time need not maximize throughput. Elsewhere [TO78] we prove that for any arbitrary network and for any regular input sequence there exists an equivalent structured network with an equal or larger average throughput. That is, there need be no reduction in throughput (pipelining performance) from constructing process description networks using only a small set of structured process

interconnections, rather than allowing arbitrary interconnections.

8. Implementation of Order Preservation

Our approach to order preservation is to insure that each of the interprocess connectives preserves the order of data cases processed assuming that its component processes are order-preserving. Order preservation is accomplished by switches introduced where data paths branch out or merge together. At the lowest level, atomic processes are assumed to handle only one data case at a time, and so preserve order.

The serial connective (functional composition) is inherently order-preserving. The parallel connective (functional construction) preserves order in that its output case is constructed out of an output case from each component, effectively synchronizing the component processes. The alternative connective (functional selection) does not inherently preserve order. If successive input cases select different component processes, there is no constraint on the order in which output cases are produced. Order-preserving pipelining is achieved by preceding each change in selection with a switching signal which follows the previous data case through the process network. This signal is generated by the initial switch and informs the final switch that the next output in order will be produced by a different designated component.

9. Process Allocation Considerations

The effectiveness of various schemes for the allocation of processes to processing elements and for the release of processing elements for other tasks depends on such variables as numbers of p.e.'s, topology of p.e. interconnection, and the storage and computational power of individual p.e.'s.

We assume that the allocation of processing elements to processes is data-driven; that is, no component process is allocated to a p.e. until its containing process has received an input case. (This bounds recursive function reference to the depth required by the data at hand.) From this standpoint, the inter-process connectives are operators which effect component process allocation for the first data case and thereafter are identity processes.

Deallocation of processing elements can occur in at least two cases-- when their allocated process is completed, and when the allocated process must be suspended and the p.e.'s assigned to another process. The latter case arises when there are fewer p.e.'s than are implied by the potential

concurrency of the process description. For process completion, if all data cases are considered elements of some sequence, then the closing bracket of the top level sequence can cause process termination and p.e. deallocation as it moves through the network of p.e.'s.

Structured process description specifies a logical interconnection of processing elements, and can be realized on many physical interconnection topologies. Possibilities include processing elements arranged in a ring, array, hyper-array, tree, or linear pipe. The choice of process allocation strategy depends both on the particular physical organization of p.e.'s and on typical patterns of information flow among processes as described in this language. The latter is not yet well understood.

The choice of process allocation strategy, and indeed of what structure to use in expressing processes, also depends on the power and, primarily, the storage capacity of individual processing elements. This can range from p.e.'s with minimal storage to p.e.'s with substantial storage and corresponding computational power. For example, the interpretation of function reference would vary with different levels of p.e., ranging from repeating the same sequence of processes in a single p.e. to allocating new p.e.'s for each reference to attain pipelining. In the next section we present several additional functional forms whose realization in a particular instance would depend on the level of processing elements being used.

Process allocation must also deal with the use of shared and restricted resources such as input and output devices or data files. Some notion of dedicated processes (monitors) is required to handle such resources [AR77b]. One approach involves processes for requesting, using, and releasing each restricted resource, together with a requesting discipline that prevents deadlock. Some interesting questions arise as to how to achieve maximum pipelining among several concurrent users of a shared process while guaranteeing that the pipeline will not become deadlocked.

10. Additional Functional Forms

Additional functional forms may be introduced. Although these forms could be expressed using the metafunctions previously presented, there are advantages in stating them directly. First, they make explicit some common action (such as repetition). Second, they result in simpler and less error-prone process descriptions. And third, they provide information useful in some cases for achieving increased concurrency which may be difficult to extract from an extended expression.

As long as these forms can be defined in terms of

composition, construction, and selection, and are so implemented, the earlier statements concerning performance will hold. In some architectures it might be desirable to implement certain forms directly, as additional primitives depending, for example, on the amount of storage available to each processing element. In this section we discuss several additional forms.

Functional exponentiation indicates repetition of a function, and is indicated by an exponent (which must evaluate to an integer). It may be defined as:

$$f^e = \underbrace{f \circ \ldots \circ f}_{e \text{ times}}$$

For a predicate as exponent, this may be expressed as:

$$f^p = (\underline{\text{identity-function}}, f)_p$$

Another form of exponentiation, called functional iteration and indicated by an asterisk, involves repetition of the function as long as a predicate is true.

$$f^{p*} = (\underline{\text{identity-function}}, f^{p*} \circ f)_p$$

While all of these exponential forms can be realized using the metafunctions as indicated above, they could also be implemented directly by iteration within the storage of a single processing element, or by a cyclic connective if one were available. More generally, functional iteration could be implemented as though the component function were recomputed each iteration "in place" by a processing element with sufficiently large storage to contain the data structures involved. For example, the square root function could be expressed iteratively as:

sqrt ((n)-->(s))

$$= (\underline{\text{undefined}}, f \circ (n, n, 1\emptyset))_{n>\emptyset}$$

where

f ((u,v,w)-->(s))

$$= \underline{1} \circ ((u+v/u)/2, v, w-1)^{w>\emptyset *}$$

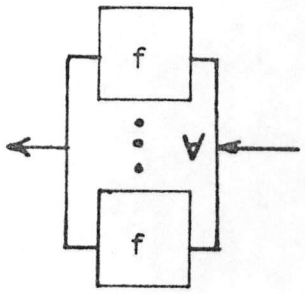
The storage implied by this square root function is small, only three scalars, but other uses of iteration could

involve storing large vectors or arrays for indexed access to components.

Finally, an additional construct is useful for achieving a variable number of parallel executions of a function. Such parallel execution can be achieved with the constructs already presented, using recursive expansion of a fixed number of copies (see [TO76]), but at a cost in elapsed time proportional to $\log_k n$, where k is the (fixed) size of each expansion and n is the total number of copies to be achieved. The for-all construct allocates a parallel copy of its component function for each element of the sequence which is its input case. The input case of each copy of the function is an element of the sequence which is the input of the for-all. This construct is expressed functionally as:

$$\forall f$$

and corresponds to the diagram:



11. Further Examples

Following are two example process definitions using the constructs introduced above. Underlined identifiers (such as size) denote undefined or primitive functions whose action is as suggested by the identifier.

Matrix Multiplication. Matrices are stored as sequences of sequences in row order. The similarity of this process definition to that presented by Backus [BA78] is intentional.

MM ((amat,bmat)-->(cmat))

= $\forall\forall$IP$\circ\forall$DISTL$\circ$DISTR$\circ$(1,TRANS$\circ$(2))

where

IP ((vector1,vector2)-->(scalar))

$$= \underline{1} \circ XIP^{\underline{size} \circ \underline{2}} \circ (\emptyset, vector1, vector2, 1)$$

XIP (sum,vec1,vec2,i)-->(sum,vec1,vec2,i))

= (sum+vec1[i]*vec2[i],vec1,vec2,i+1)

forms the inner product of two vectors.

TRANS (transpose an array), DISTR (distribute to the right), and DISTL (distribute to the left) are here taken as previously defined.

Sort. This function sorts a sequence of values in descending order (that is, produces an output sequence whose elements are those of the input sequence, but ordered) by first ranking the sequence and then constructing an ordering based on that ranking. If enough processing elements are available to exploit fully the potential concurrency (and ignoring communication costs), the function is of time complexity O(n).

SORT ((unsorted)-->(sorted))

= CONSTRUCT∘(unsorted,RANK)

where

RANK ((in)-->(ranking))

$$= \forall \, COUNTLARGER \circ DISTL \circ (in, in)$$

builds for the input sequence a ranking sequence showing how many elements of the input are larger;

COUNTLARGER ((seq,elem)-->(count))

$$= \underline{3} \circ COMPARE^{\underline{size} \circ seq} \circ (seq, elem, 1)$$

counts the number of elements in the input sequence larger than the element in position index;

COMPARE ((seq,elem,count)-->(seq,elem,count))

$$= (\underline{tail} \circ seq \; , elem, (count, count+1)_{\underline{head} \circ seq \, = elem})$$

adds one to count if sequence element larger than elem;

and where

```
CONSTRUCT    ((in,ranking)-->(out))

        = 3∘ENTER^size∘ in ∘(in,ranking,

              makeseq∘(size∘ in ,blank),1)
```

builds an output sequence by entering each element of in in the position given by the corresponding element of ranking;

```
ENTER    ((in,ranking,out,i)-->(in,ranking,out,i))

      = (in,ranking,

        PUTINPLACE∘(in[i],1∘SKIPEQUALS∘(ranking[i],out),

                  out),i+1)
```

puts element in[i] in place in sequence out after skipping over any equal elements;

```
PUTINPLACE    ((elem,index,out)-->(out))

          = concat∘(append∘(out[1:index-1],elem),

                   out[index+1:size (out)])
```

concatenates elem at position index in sequence out;

```
SKIPEQUALS    ((index,out)-->(index,out))

          = (index+1,out)^out[index]≠blank*
```

advances index past all non-blank elements.

13. References

[AR77a] Arvind and K.P. Gostelow, "A computer capable of exchanging processors for time", Proceedings IFIP Congress '77, Toronto, Canada, (1977).

[AR77b] Arvind, K.P. Gostelow and W. Plouffe, "Indeterminacy, monitors and dataflow", Proceedings of the Sixth ACM Symposium on Operating System Principles, Purdue University, Operating Systems Review II, 5, (Nov. 1977).

[BA78] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", CommACM 21, 8, (August 1978).

[BU75] Burge, W.H., Recursive Programming Techniques, Addison-Wesley, Reading, Mass., (1975).

[DA72] Dahl, O-J, E.W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, New York, (1972).

[DE75] Dennis, J.B. and D.P. Misunas, "A preliminary architecture for a basic data-flow processor", Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, (1975).

[DI76] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey, (1976).

[KO74] Kosaraju, S. Rao, "Analysis of structured programs", Journal of Computing and System Sciences 9, (1974).

[LA64] Landin, P.J., "The mechanical evaluation of expressions", Computer Journal 6,4, (1964).

[MC60] McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine, part 1", CommACM 3,4, (April 1960).

[TO76] Tonge, F.M., "Expressions for time and space in a recursive realization of parallelism", TR79, Department of Information and Computer Science, University of California Irvine, (May 1976).

[TO78] Tonge, F.M., "Pipelining performance of structured networks", TR117, Department of Information and Computer Science, University of California Irvine, (May 1978).