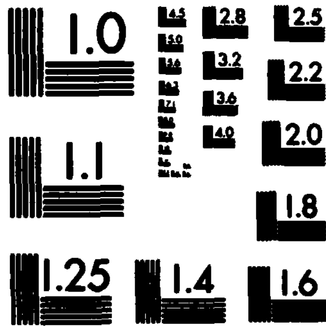
AD-A130 232 SIMULATION OF SYSTOLIC NETWORKS WITH A SYNTAX DIRECTED SOLVER FOR SYSTEMS..(U) PITTSBURGH UNIV PA INST FOR COMPUTATIONAL MATHEMATICS AND APP.. R G MELHEM JAN 83 1/1

UNCLASSIFIED ICMA-83-58 N00014-80-C-0455 F/G 9/2 NL

END
FILMED

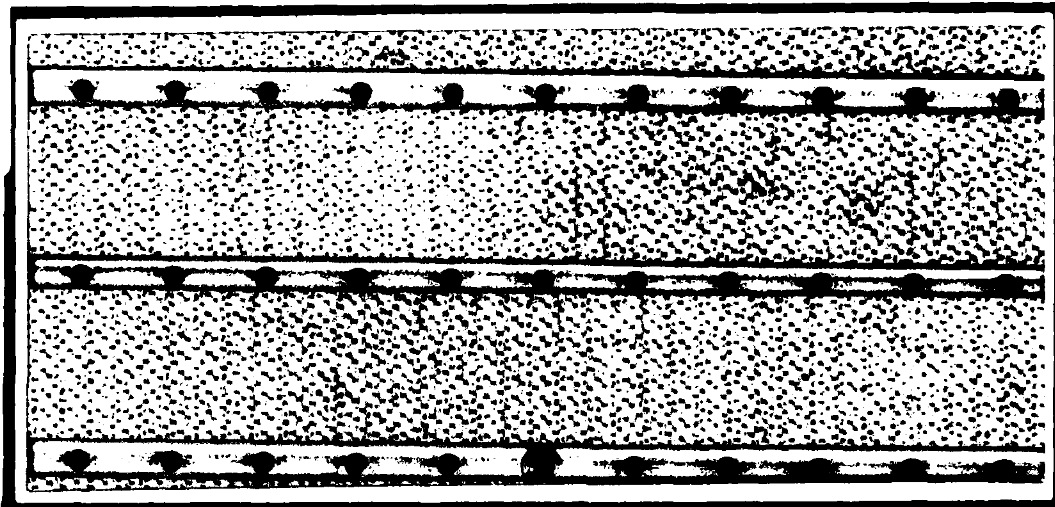


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(15)

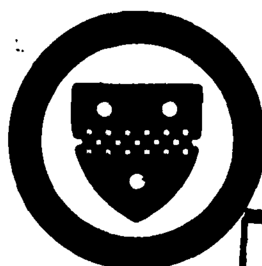
ADA130232

INSTITUTE FOR COMPUTATIONAL MATHEMATICS AND APPLICATIONS



Department of Mathematics and Statistics

University of Pittsburgh



DTIC
ELECTE
JUL 8 1983
S D
A

This document has been approved for public release and sale; its distribution is unlimited.

DTIC FILE COPY

83 06 15 123

Technical Report ICMA-83-58

SIMULATION OF SYSTOLIC NETWORKS WITH A SYNTAX DIRECTED SOLVER FOR SYSTEMS OF SEQUENCE EQUATIONS*)

by

Rami G. Melhem

JAN. 1983

Department of Computer Science
and
Department of Mathematics and Statistics
University of Pittsburgh
Pittsburgh, PA 15261

DTIC
SELECTE
JUL 8 1983
S D
A

This document has been approved for public release and sale; its distribution is unlimited.

*) This work in part was supported under ONR Contract N00014-80-C-0455.

-a-

Simulation of Systolic Networks with a Syntax Directed Solver for Systems of Sequence Equations

Rami G. Melhem

ABSTRACT

The main objective of this paper is to supplement a model that was previously suggested for the verification of systolic networks. A simple language is presented to express the system of sequence equations that models the operation of the network. Then a syntax directed interpreter is developed to solve this system for specific forms of the inputs and to produce the corresponding outputs. This technique for the verification of a certain computation on a systolic network is equivalent with the simulation of its execution.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A

DTIC COPY INSPECTED

1. Introduction

An abstract mathematical model for the specification and verification of systolic networks [1] was introduced in [2] and extended in [3]. The basic idea of this model is the representation of the data appearing on any one communication link of the systolic network by a data sequence. Here the tth element of the sequence associated with some link is the data item that appeared on that link at time t, where as usual, one time unit is the period of the clock that synchronizes the network.

The computation performed by each computational cell in the network is then modeled using causal operators on data sequences. More specifically, the sequences on the output links of a certain cell are described as a function of the sequences on the input links to that cell, with the condition that the tth element of any output sequence cannot depend on some $(t+i)^{th}$ element in any input sequence for $i>0$. In other words, the output cannot depend on future inputs. This is called the causality condition.

In order to model the computation of a systolic network, we establish for each node in the network the sequence equations describing its operation. The solution of the resulting system of equations is called the network I/O description and gives the output sequences of the network in terms of its input sequences. In the remainder of this report, we will use the term sequence equation in a restrictive manner to indicate an equation in which the left side is a sequence identifier and the right side a sequence expression. This is the only type of equations that is needed for modeling the operation of systolic networks.

It is clear that the I/O description of a systolic network specifies the global behavior of the network by providing a means of obtaining the network output for any given inputs. However, we do not need to find explicitly the I/O description of a certain network in order to verify its operation for specific inputs. In fact, it suffices to substitute the known input sequences into the system of equations modeling the network, and then, by sequence manipulation, determine the description of the output sequences.

Unfortunately, our tools for manipulating sequences are still limited and in many cases, we are not able to obtain analytically the description of the output sequences. In order to alleviate this problem, we describe in this report a computer system that was developed to solve iteratively any system of consistent causal sequence equations for specifically given input sequences. In this context, an input sequence is simply defined as a sequence that does not appear on the left side of any equation of the system.

The system of causal equations is provided to the solver in terms of a very simple language called SCE (Systems of Causal Equations) that we describe in Section 2. The equation solver then represents a syntax directed interpreter that executes any correct SCE program. Such an interpreter is outlined in Section 3: it reads the elements of the input sequences from an input file, and calculates the elements of the sequences on the left side of the equations specified by the program.

It should be noted that, by definition, data sequences have infinite length. However, for practical reasons, an upper bound MAXT is given by the user for the number of data items of all sequences. Elements beyond MAXT in any sequence are considered to be the don't care element defined in [2].

Although the solver defined in this report is quite general, its immediate application is to the verification of computations on systolic networks. More descriptively, in order to verify a computation on a systolic network, we first write an SCE program containing the equations for the operation of the network, and create an input file containing the elements of the input data sequences. Then a run of the SCE interpreter provides the elements of the output data sequences. The verification of a systolic network by this method is equivalent with the simulation of its execution. However this approach to simulation seems to be more structured and separates the internal details of the simulator from the concern of the user. It also has the advantage of allowing the user to solve partially the system of equations modeling the network and then to use the SCE interpreter on the portion of the system that could not be solved analytically. As an example, we show in Section 4 how the operation of a network designed for the LU decomposition of a symmetric banded matrix is simulated by means of the SCE interpreter.

It should be mentioned here that the model in [2] and another model developed independently by Chen and Mead [4, 5] are based on the same idea of separating the formal specification of the operation of the network from the data for specific computations. However, in [4] the operation of each cell is defined by a functional approach, while we follow an algebraic approach to define relatively few sequence operators that are used to model computational cells.

2. The SCE language for specifying Systems of Causal Equations

The SCE language is a simple expression language augmented with some input/output facilities and looping capabilities that provide for efficiency in the writting of programs. In its current form, SCE may be used to model the operation of a special class of systolic networks in which the units of information are real numbers. However, by the addition of new rules to the grammar, it is possible to model other types of systolic networks at higher or lower levels of architectures.

By the first rule in the grammar given in Appendix A, it is readily seen that an SCE program consists of the following four parts: 1) The declarations, 2) the input part, 3) the programs body, and 4) the output part. In the rest of this section, we will clarify the semantics of the language assuming familiarity with the data sequence operators defined in [2] and [3].

Terminal symbols in the SCE language (see Appendix A) can be classified into four categories, namely, special symbols (e.g. +, -, *, ...), reserved words (e.g. FOR, OUT, ...), identifiers and constants, where a constant is either a positive integer or a positive real number written in floating point format.

In order to ensure a clear distinction between identifiers and reserved words in SCE, we have chosen all the reserved words in the language to start with capital letters. On the other hand, any string of alphanumeric characters starting with a small letter can be used as an identifier, with the understanding that only the first six characters are significant. Identifiers in SCE should be declared in the declaration part of the program to be of one of the following types:

1) **Parameter** (rules 3-7): A parameter is assigned an integer value at the time of its declaration and this value is substituted textually whenever the identifier appears in the program.

2) **Index** (rules 8-11): An index in SCE is an integer variable used in loop control

and in selecting elements of sequence arrays.

3) **Sequence** (rules 12-19): Sequences are represented on the machine by vectors. An identifier of type "sequence" may be associated with either a single sequence (rule 16) or with an array of sequences (rule 15). For arrays of sequences, the dimension and the lower and upper bounds are also specified in the declaration by enclosing these bounds in curly brackets. For example, the following SCE statement declares s as an n dimensional sequence array

$$SEQN\ s\{l_1:u_1\ ,\ \cdots\ ,l_n:u_n\}$$

where l_i and u_i, $i=1,\cdots,n$ are the lower and upper bounds for the ith dimension. Bounds may be negative but should of course satisfy the restriction that $u_i \geq l_i$. We also note that there is no limit on the dimensionality of an array.

After the declaration of an array of sequences, its elements may be identified (rules 38-41) by using the usual selection notation $s(p_1\ ,\ \cdots\ ,\ p_n)$, where each p_i falls into its corresponding range, that is $l_i \leq p_i \leq u_i$.

In the context of the abstract systolic model, a data sequence is associated with a communication link which is identified by its color and the label of the node at which it terminates. In order to simplify the SCE specification of a systolic network, we label the nodes in the network by n-tuples of integers $(v_1,\cdots,v_n)$ for some n. This enables us to group in an n-dimensional array all the sequences associated with the links that have the same color, and naturally use the color of the links to identify the array. With this, the sequence associated with any link $y_{v1,...,vn}$ is simply the element $(v_1,\cdots,v_n)$ of the array of sequences y0.

Although this usually leads to a very clear SCE specification of systolic networks, it is sometimes inefficient because some of the elements in the array may not be used. For example in the LU network described in section 4, a triangular

array of sequences would be more space efficient than the rectangular array allowed by SCE. In such cases, more efficient storage could be obtained by applying any one of the techniques used for storing triangular and sparse matrices.

In addition to arrays of sequences, the language allows the user to declare single sequences. Three standard single sequences are predefined by the language, namely the don't care sequence d, the zero sequence o and the unity sequence u. As their names imply, the elements in the d, o and u sequences are all δ-elements, zeroes or ones, respectively. The user however may re-declare the identifiers d, o or u if he wishes to change their definitions.

The input part of an SCE program takes the form of a single INPUT statement (rules 68-73). It serves two purposes: First it assigns an integer value to MAXT, which specifies the number of elements to be considered in any sequence, and second, it specifies the sequences to be read from the input file. Nested FOR loops can be used, to any level, in specifying the input sequences. For example, in the program presented in Section 4, the statement

INPUT(MAXT 18 , FOR I=0,3 r(I,1));

specifies that MAXT=18 and that the input sequences are r(0,1), r(1,1), r(2,1) and r(3,1).

Similarly, the output part of the program takes the form of a single statement (rules 74-78) that specifies the sequences to be printed on the output file. FOR loops are also allowed.

The body of the SCE program is the part that contains the specification of the system of sequence equations. It consists of a list of statements, where a statement may be either a sequence equation or a FOR loop that encloses a list

of statements. Each equation has the form

sequence specification = sequence expression

where the left side identifies a particular sequence and the right side is an expression composed from sequence identifiers and sequence operators. Square brackets may be used in sequence expressions to override the precedence rules defined by the grammar. Basically, in the evaluation of expressions, the grammar associates the highest priority with the operators defined directly on sequences. Next in priority is the scalar multiplication operator '.', followed by the operators '*' and '/'. Finally, the operators '+' and '−' are evaluated with the lowest priority. With these precedence rules the sequence expressions are evaluated from left to write.

Although many other sequence operators may be incorporated into the language, we only considered the operators that were formally defined in [2] and [3], namely:

The positive shift operator Ω^r, written in SCE as O(r),

The zero shift operator Ω_0^r, written in SCE as Z(r),

The spread operator Θ^r, written in SCE as T(r),

The expansion operator E_r^k, written in SCE as E(r,k),

The accumulator operator $A^{r,k,s}$, written in SCE as A(r,k,s) and

The multiplexing operator $M_r^{w1,...,wn}$ written in SCE as M(r;w1,· · ·,wn).

Frequently, the shift, zero shift and spread operators are used with r=1. For this, the short hand notation O, Z, and T may be used instead of O(1), Z(1) and T(1), respectively.

The two element-wise operators U1 and U2 of rules 47 and 48 have the same priority as the operators * and / but their semantics are not defined by the

language. As indicated in Section 3, U1 and U2 may be defined by the user.

Finally, we note that rule 55 restricts the operands of the accumulator operator to single sequences rather than sequence factors as is the case with the other operators. This restriction is not necessary and was only imposed because it leads to a more efficient implementation of the SCE interpreter. However, it should be noted that this does not affect the expressive power of the language because we can always define intermediate sequences to get around this restriction. For example, the sequence equation

$$x(i) = O(2)\ x(i-1) + A(1,3,1)\ [\ y(i+1)\ *\ T\ x(i-1)\ ]$$

which is not permitted in SCE can be split into the two SCE legal equations

$$v(i) = y(i+1)\ *\ T\ x(i-1)$$

$$x(i) = O(2)\ x(i-1) + A(1,3,1)\ v(i).$$

3. Overview of the SCE Interpreter

As is the case with most language interpreters, the SCE solver has two distinct phases, namely, the syntax analysis phase which, using a parse tree of the program, produces an intermediate language program, and the actual interpretation phase which executes the intermediate program.

For the syntax analysis phase, we used the automatic parser generator YACC [6] existing on the UNIX operating system to generate an LR(1) bottom-up parser that accepts any syntactically correct SCE program and generates an intermediate program in the form of a sequence of tuples. It is basically a finite state machine with a stack. It scans the input program from left to right and is capable of reading and remembering the next input token (terminal symbol) which is called the look-ahead token. Depending on the look-ahead token and the content of the stack, the parser takes one of the following actions:

1) **Shift:** The current look-ahead token is pushed into the stack and the next token is read in. Also a tuple describing the action is generated. If the token being shifted is a special symbol, an identifier, or a reserved word, the tuple generated has the form (Shift,n), where n is a number identifying the token. On the other hand, if the token is an integer or a real constant, then the tuple generated has the form (Shift-integer,c) or (Shift-real,r), respectively, where c or r is the value of the constant.

2) **Reduce:** This action is taken when the parser recognizes that the stack contains the right hand side of a grammar rule, say rule n, and that this rule should be applied at this point. It then pops from the stack the tokens forming the right side of rule n and pushes onto the stack the token on its left side. It also generates a tuple (Reduce,n).

3) **Accept:** This action is taken when the parsing process is successfully completed. The tuple generated in this case is (Accept,0).

4) **Error:** If the parser discovers that the program is syntactically incorrect, it simply gives a warning and halts. Of course, more elaborate error handling actions could have been taken if our goal was to produce a more sophisticated parser. For the details and internal forms of LR parsers, we refer to [7].

The second phase of the interpreter reads the intermediate program (sequence of tuples) and reproduces the actions taken by the parser on an action stack. Simultaneously, an adjoint value stack is used to store temporary values needed in the interpretation. In Appendix B, we give the complete listing of a C program for the second phase of our SCE solver, and in the rest of this section we will outline the main features of this solver/interpreter.

The program uses a location counter "location" to indicate the intermediate tuple being interpreted. Starting with location =1, the interpreter reads the tuple pointed to by "location", takes a certain action, increases location by one and then repeats the above cycle. The action taken in each cycle depends on the type of the tuple being interpreted, namely

1) If the tuple is of the type (Shift,n) or (Shift-integer,n), then n is pushed into the action stack.

2) If the tuple is of the type (Shift-real,r), then r is pushed into the value stack and a zero is pushed into the action stack.

3) If the tuple is of the type (Accept,0), then the interpretation is terminated.

4) If the tuple is of the type (Reduce,n), where grammar rule n has the form $b \rightarrow a_1\ a_2\ \cdots\ a_k$ then the interpreter pops the k top locations of the stack, which should contain the symbols $a_1, \cdots, a_k$ and pushes b. It also may execute a semantic routine if any is associated with this grammer rule. These routines manipulate the data on the value stack to reflect the semantics of the grammar rule.

At this point we note that we do not actually have to push or pop the grammar symbols in the action stack, and that a sufficient action will be to keep track of the top of the stack. Then at the time of a reduction, the grammar rule and the top of the stack determine uniquely the location that each symbol would occupy on the stack. With this, we can transmit information from one semantic routine to another by pushing this information into the stack in the place of the grammar symbol. For example, the semantic routine associated with rule 36 uses the location of the FOR symbol on the stack to store the starting address of the first statement in the FOR loop body. Then, when the execution of the FOR body is terminated, the routine for rule 35 retrieves this address from the stack to re-initiate the execution of the FOR body if the final value of the index of the loop is not yet reached.

In order to reduce the storage required for holding the intermediate tuples, the program in appendix B reads and executes the tuples in four stages: in the first and the second stage, the declaration and the input part of the program are processed, respectively. In the third stage, the system of equations is solved, and finally in the fourth stage the output is printed. We briefly comment on each stage.

The declarations: The main objective of this stage is to construct the symbol table and to allocate storage for the declared sequences. The symbol table "sym_tab[]" is an array of records with three fields. The first field contains a character that indicates the type given to the identifier, namely P, I or S for parameters, indices or sequences, respectively. The interpretation of the integers in the second and third fields, called entry1 and entry2, depends on the type of the identifier. For parameters, entry1 contains the value of the parameter and entry2 is immaterial. For index variables, entry1 holds the value of the index, initialized to zero, and entry2 is set during the execution of a FOR loop to the final value of the loop

index.

Finally, if the identifier is declared as a sequence variable, then it may denote a single sequence or an array of sequences. Single sequences are distinguished by setting entry2=-1, with entry1 pointing to the location where the sequence is stored. For arrays of sequences, entry2 holds a pointer to a bound table that indicates the dimension and the bounds of each array, and entry1 points to the location where the first sequence in the array is stored. The first three locations in the symbol table are reserved for the identifiers d, o and u, respectively that are preset to the don't care, the zero and the unity sequences, respectively. However, if any of these identifiers are declared in the program, then the corresponding entry in the symbol table is overwritten by the semantics routine corresponding to the new declaration.

The sequences are stored in a two dimensional array seq_store[][]. Each row in the array has a length at least equal to MAXT and is used to store the elements of a sequence. Arrays of sequences are stored in consecutive rows such that any index changes slower than the one to its right, if any. In order to keep track of don't care elements, an array d-table[][] of bits is used such that for each element in seq_store[][], there is a corresponding bit in d-table[][]. This bit is set to one, if the element in seq_store[][] is a don't care, and to zero, otherwise. Consequently, any part of the program that reads an element from seq_store[][] should also inspect the corresponding entry in d-table[][].

The implementation of the SCE solver listed in Appendix B allows for full causality in the sense that an output may depend on any previous input. Consequently, storage is provided for the retention of at least MAXT elements from any sequence. A more space-efficient implementation would be to retain only the last C elements from each sequence in a circular buffer, where C is given. This

allows only for C-order causality in the sense that the output of a certain cell at any given time t may only depend on the inputs to that cell during the time period from t-C to t-1.

The input part: The INPUT statement specifies the sequences to be read from the input file as well as the number MAXT of elements in each sequence. The interpreter reads, as a stream, the MAXT elements of the first specified sequence followed by those of the second sequence.... etc. provided that the elements are separated by at least one space. No special characters are required to separate the elements of the different sequences. Each element in the input file may be either a floating point number or the letter "d" representing a don't care element. The interpreter also recognizes the string "...." in the input file as an indication that the remaining elements in that sequence are don't cares.

The equation solver: The sequence of tuples in the body of the program are executed iteratively MAXT times. A global clock "TIME" is initialized to 1 and incremented at every step of the iteration. At every step, the expression on the right side of each equation is evaluated at time TIME and assigned to the corresponding element of the sequence on the left side of the equation.

The value stack is used during the evaluation of sequence expressions to store temporary results. For example, the semantic routine associated with the grammar rule 52 (seq_factor → seq_spec) reads the value of the element TIME in the specified sequence from seq_store[][], and pushes it onto the value stack. The result of any subsequent sequence operation is stored on the stack until rule 37 is executed and the final result is stored back into seq_store[][].

The sequence operators O, Z, T and E operate on sequence factors and have the effect of changing the global clock during the evaluation of the corresponding factor. The old clock value is stored in the action stack for

retrieval after the evaluation of the factor is complete (rule 53). If the result of any operation involving the above operators is the don't care element, then the flag "skip" is set which causes the execution of the semantic routines to be skipped until the corresponding tuple (Reduce.53) is encountered. Of course provisions are made to deal with arbitrary degrees of nesting.

In a similar way, the flag "Mskip" is used to chose the appropriate operand in the multiplexer operator. Finally, we note that by restricting the operands of the accumulator operator to sequences instead of sequence factors, we simplified greatly the action associated with that operator. For a detailed description of the different semantics routines, we refer to the complete listing of the program in Appendix B.

It is important to note that the SCE interpreter detects any inconsistency in the given equations or any attempt for solving equations which are not causal or weakly causal. It does so by associating with each sequence an entry in the array last_computed[] to keep track of the last element that has been computed in the sequence so far. Any attempt to overwrite an already calculated element or to read an element that has not yet been calculated is then easily detected and reported as a run time error. The interpreter also detects other types of run-time errors that are listed in the function *run_error* in the Appendix.

The output part: After completing the interpretation of the body of the program, the sequences specified by the user are printed on the standard output file.

Finally, we note that a possible optimization of the implementation of the interpreter could be achieved by replacing the single value stack by k stacks, for some optimal k. This would reduce the total number of iterations through the body of the program by considering at each step the elements TIME, TIME+1, ··· , TIME+k of the sequences instead of only one element at a time. However, if

the system contains any recursion, then only few of these k elements (and in many cases only one) can be considered at each step, and this requires more complicated book keeping to update the array last_computed and the global clock. We decided not to implement this optimization because we intended the solver to be used in cases where analytical solutions of the system of equations are not possible, and hence where recursivity is usually present.

4. Example: An LU factorization network

In this section the SCE interpreter is applied to the simulation of the computations of a network for the LU or the $U^T DU$ factorization of a symmetric banded matrix A and the solution of the linear system of equations Ax=y with a given vector y. It will be shown also that, with slight modifications, the same network can be used to compute the Cholesky decomposition LL^T of the matrix A.

The first systolic network for factoring a banded matrix into the product of a lower triangular matrix and an upper triangular matrix was suggested by Kung and Leiserson [8]. Later, Brent and Luk [9] modified the Kung and Leiserson network to compute the Cholesky decomposition of symmetric matrices. The network described in this section is also designed for symmetric matrices but is different in its operation principle from the one in [9]. Both networks use almost the same number of computational cells and achieve approximately the same speed-up over serial execution. They differ however in the type of computational cells and in their interconnections.

Consider the system of linear equations

$$A\ x = y \tag{4.1}$$

where A is an $n \times n$ matrix and x and y are n dimensional vectors. The solution x of (4.1) may be obtained by finding a lower triangular matrix L and an upper triangular matrix U such that $A = L\ U$, and then solving the two triangular systems $L\ z = y$ and $U\ x = z$. More specifically, assuming that A is symmetric and banded with band width 2k+1, and denoting the elements of the matrices A, L and U by $a_{i,j}$, $l_{i,j}$ and $u_{i,j}$, respectively, we can use the following algorithm to compute the LU decomposition of A. Note that only the elements $a_{i,j}$ with $j \geq i$ are used and that only the non zero elements of L and U are computed.

ALG1 : LU factorization.

FOR i=1, ⋯ ,n DO

1) FOR j=i, ⋯ , min(n, i+k) DO

1.1) $l_{j,i} = a_{i,j}$

1.2) $u_{i,j} = \frac{a_{i,j}}{a_{i,i}}$

2) FOR q=i+1, ⋯ ,min(n, i+k) DO

FOR j=q, ⋯ ,min(n, i+k) DO

$$a_{q,j} = a_{q,j} - l_{q,i}\, u_{i,j}$$

At this point we note that the matrix L obtained by the above algorithm satisfies $L = U^T D$, where D is the diagonal matrix defined by $d_{i,i} = l_{i,i}$. Also, by replacing steps 1.1 and 1.2 in ALG1 by

$$l_{j,i} = u_{i,j} = \frac{a_{i,j}}{\sqrt{a_{i,i}}}$$

we obtain an algorithm for the Cholesky decomposition of A into LL^T.

After having performed the LU decomposition of A, we may compute the vector $z = L^{-1} y$ by the following algorithm

ALG2 : Back substitution.

FOR i=1, ⋯ ,n DO

$$z_i = \frac{y_i}{l_{i,i}}$$

FOR q=i+1, ⋯ ,min(n, i+k) DO

$$y_q = y_q - l_{q,i}\, z_i$$

Finally, the solution of $Ux = z$ may be obtained by an algorithm similar to ALG2, or alternatively by using ALG2 itself for the solution of $\bar{U}\,\bar{x} = \bar{z}$, where $\bar{z}_i = z_{n-i+1}$, $\bar{x}_i = x_{n-i+1}$ and $\bar{U}$ is a banded lower triangular matrix given by

$$\bar{u}_{i,j} = u_{n-i+1,n-j+1}.$$

As an example, let n=5 , k=2 and

$$A = \begin{bmatrix} 2 & 4 & 6 & 0 & 0 \\ 4 & 11 & 15 & -3 & 0 \\ 6 & 15 & 20 & 2 & -2 \\ 0 & -3 & 2 & -19 & 1 \\ 0 & 0 & -2 & 1 & 14 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} 4 \\ 14 \\ 15 \\ 0 \\ -6 \end{bmatrix} \tag{4.2}$$

By ALG1 we obtain

$$L = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 4 & 3 & 0 & 0 & 0 \\ 6 & 3 & -1 & 0 & 0 \\ 0 & -3 & 5 & 3 & 0 \\ 0 & 0 & -2 & -9 & -9 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -5 & 2 \\ 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

and by ALG2

$$z = \begin{bmatrix} 2 \\ 2 \\ 3 \\ -3 \\ 3 \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} -41 \\ -19 \\ 27 \\ 6 \\ 3 \end{bmatrix} \tag{4.4}$$

The graph of the systolic network that executes ALG1 and ALG2 simultaneously is shown in figure 4.1. It is composed of $\frac{(k+1)(k+4)}{2}$ interior nodes. Each node is labeled by a pair (i,j), where i and j are the coordinates of the node with respect to the two axes shown in the figure. The color of each edge is determined by its direction, namely edges directed to the east, south and south west are given the colors s, b and c, respectively, and those directed north are assigned the colors r or p depending on their relative position.

The part of the graph that is formed by nodes (i,j), $i=1,\cdots,k+1$, $j=1,\cdots,k-i+2$ represents a subnetwork that executes ALG1. It consists of three types of nodes whose operation is described by the following causal equations: (note that the sequence associated with a link $y_{i,j}$ is denoted by $\eta_{i,j}$, where η is the greek letter corresponding to the color of the link y).

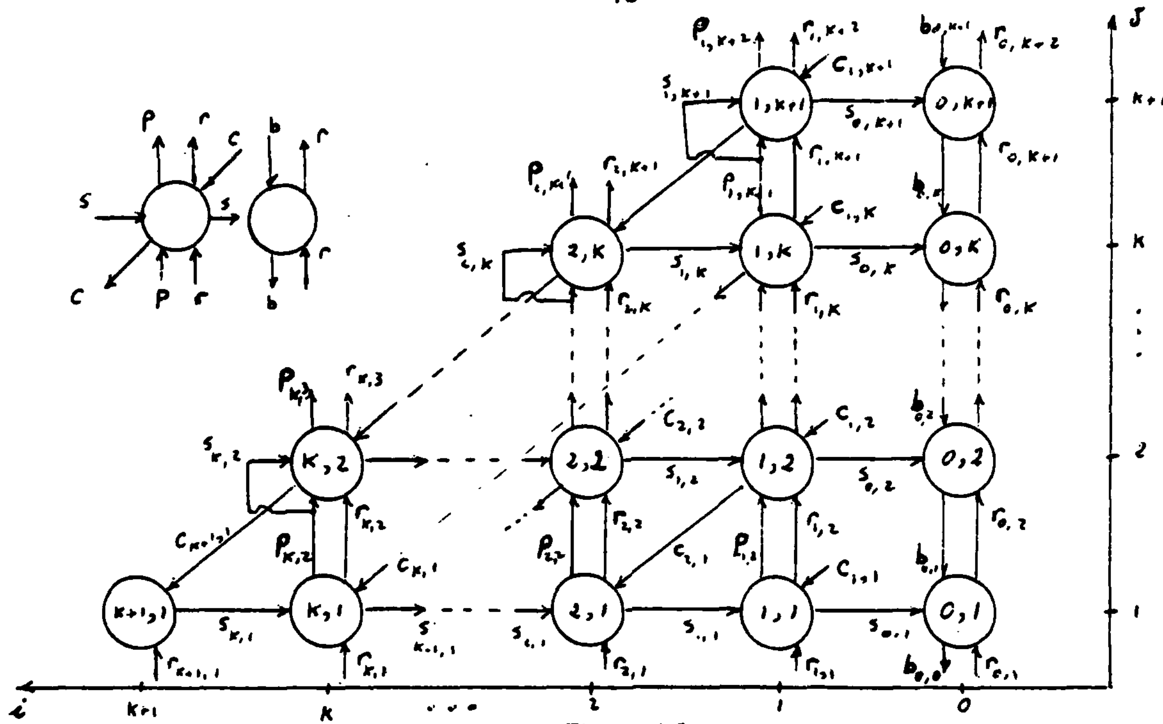


Figure 4.1

For node (k+1,1)

$$\sigma_{k,1} = \Omega_0 \, [\tau \; / \; [\rho_{k+1,1} - \gamma_{k+1,1}]] \tag{4.5}$$

where τ is the unity sequence defined by $\tau(t)$=1.0 for any t.

For nodes (i,1), i=1, $\cdots$,k

$$\rho_{i,2} = \Omega_0 \, [\rho_{i,1} - \gamma_{i,1}] \tag{4.6.a}$$

$$\pi_{i,2} = \Omega_0 \, [\sigma_{i,1} * [\rho_{i,1} - \gamma_{i,1}]] = \Omega_0 \, \sigma_{i,1} * \rho_{i,2} \tag{4.6.b}$$

$$\sigma_{i-1,1} = \Omega_0 \, \sigma_{i,1} \tag{4.6.c}$$

For nodes (i,j), j=2,$\cdots$,k+1, i=1,$\cdots$,k+2−j

$$\sigma_{i-1,j} = \Omega_0 \, \sigma_{i,j} \tag{4.7.a}$$

$$\pi_{i,j+1} = \Omega_0 \, \pi_{i,j} \tag{4.7.b}$$

$$\rho_{i,j+1} = \Omega_0 \, \rho_{i,j} \tag{4.7.c}$$

$$\gamma_{i+1,j-1} = \Omega_0 \, [\gamma_{i,j} + \rho_{i,j} * \sigma_{i,j}] \tag{4.7.d}$$

On the other hand, the part of the graph composed from the nodes (0,j), j=1,$\cdots$,k+1 corresponds to a subnetwork that executes ALG2. The operations of the cells in this subnetwork are described by

For node (0,1)

$$\rho_{0,2} = \Omega_0 \, [\rho_{0,1} - \beta_{0,1}] \tag{4.8.a}$$

$$\beta_{0,0} = \Omega_0 \, [\sigma_{0,1} * [\rho_{0,1} - \beta_{0,1}]] = \Omega_0 \, \sigma_{0,1} * \rho_{0,2} \tag{4.8.b}$$

For nodes (0,j), j=2, $\cdots$,k+1

$$\rho_{0,j+1} = \Omega_0 \, \rho_{0,j} \tag{4.9.a}$$

$$\beta_{0,j-1} = \Omega_0^2 \, [\beta_{0,j} + \sigma_{0,j} * \rho_{0,j}] \tag{4.9.b}$$

Note that the nodes $(i,1)$, $i=0,\cdots,k$ correspond to subtract/multiply cells, while the nodes (i,j), $j=2,\cdots,k+1$, $i=1,\cdots,k+2-j$, are multiply/add cells. Only the node (k+1,1) is a subtract/divide cell. In other words, the network is composed of three basic types of simple computational cells.

For proper operation of the network, the input sequences $\gamma_{1,j}$, $j=1,\cdots,k+1$ and $\beta_{0,k+1}$ are set to the zero sequence ι defined by $\iota(t)=0.0$ for all t, and the input links $s_{k+2-j,j}$, $j=1,\cdots,k+1$, are connected to the links $\rho_{k+2-j,j}$ (see figure), that is

$$\gamma_{1,j} = \iota \qquad j=1,\cdots,k+1 \tag{4.10.a}$$

$$\beta_{0,k+1} = \iota \tag{4.10.b}$$

$$\sigma_{k+2-j,j} = \pi_{k+2-j,j} \qquad j=2,\cdots,k+1 \tag{4.10.c}$$

The elements of the matrix A and the vector y are fed into the network through the links $r_{i,1}$, $i=1,\cdots,k+1$ and $r_{0,1}$, respectively. The precise input specification is given by

$$\rho_{i,1} = \Omega^{k+1-i} \, \Theta^2 \, \alpha_i \qquad i=1,\cdots,k+1 \tag{4.11.a}$$

$$\rho_{0,1} = \Omega_0^{k+1} \, \Theta^2 \, \eta \tag{4.11.b}$$

where $T(\alpha_i)=n-(k+1-i)$, $T(\eta)=n$ and

$\alpha_i(t) = a_{t,t+k+1-i}$

$\eta(t) = y_t$

that is, α_{k+1-q} contains the n−q elements of the qth off diagonal of A, and η the n elements of the right hand side vector y.

In order to understand the principle of operation of the network, we first note that at iteration step i of algorithm ALG1, the ith row of U and the ith colomn of L are computed from the ith row of A (steps 1.1 and 1.2). However, the elements of the matrix A are continuously modified. In particular, at the execution of step i, the elements in row i of A had been modified by subtracting from them different contributions (step 2) during the steps $i-k,\cdots,i-1$. In the systolic network of figure 4.1, the elements of the unmodified i^{th} row of A arrive at the cells $(q,1)$, $q=1,\cdots,k+1$, on the r colored links. At the same time, the sum of the contributions from the previous iterations $i-k,\cdots,i-1$ arrive at the same cells on the c colored links. The subtraction is then performed and the elements of the corresponding colomn and row of L and U are computed and sent out on the r and p colored links, respectively. These elements propagate upward in the network allowing the cells (q,i), $i=2,\cdots,k+1$, $q=1,\cdots,k+2-i$ to compute and sum the contributions for the modification of the subsequent rows of A. These contributions are sent downward on the c colored links. Finally, the subnetwork formed by the cells $(0,i)$, $i=1,\cdots,k+1$ operate in a similar way.

A careful study of the behavior of the network shows that the significant elements of the matrix U are sampled from the links $p_{q,1}$, $q=1,\cdots,k$, and the elements of the partial solution vector z from $b_{0,0}$. These results are sufficient for the computation of the solution $x=U^{-1}z$. However, the elements of the diagonal matrix D, where $L=U^TD$, are also available on the link $s_{k,1}$. More precisely, the output sequences are expected to have the following description:

$$\pi_{q,1} = \Omega^{k+2-q}\ \theta^2\ \mu_q \qquad q=1,\cdots,k \tag{4.12.a}$$

$$\beta_{0,0} = \Omega^{k+2}\ \theta^2\ \zeta \tag{4.12.b}$$

$$\sigma_{k,1} = \Omega\ \theta^2\ \lambda \tag{4.12.c}$$

where $T(\mu_q)=n-(k+1-q)$, $T(\zeta)=T(\lambda)=n$ and

$$\mu_q(t) = u_{t,t+k+1-q}$$

$$\lambda(t) = d_{t,t}$$

$$\zeta(y) = z_t$$

After the computation of U and z terminates, we may use the network for a second time to solve $\overline{Ux=z}$ and obtain the vector x. Of course only few cells will be doing useful work during this second run. At this point we note that we can add to the network any number of columns of cells identical to the column $(0,l)$, $l=1,\cdots,k+1$. This enables us to use the network to solve (4.1) for more than one right hand side vector y simultaneously.

Finally, we note that the network described in this section can be modified to perform the Cholesky decomposition LL^T instead of the U^TDU decomposition. For this, equation (4.5) that describes the operation of node (k+1,1) in the network have to be replaced by

$$\sigma_{k,1} = \Omega_0\ [\frac{1}{\sqrt{\rho_{k+1,1} - \gamma_{k+1,1}}}]$$

and the data on the links $\rho_{l,2}$, $l=1,\cdots,k$ have to be set equal to the data on $\pi_{l,2}$, $l=1,\cdots,k$. This has the effect of modifying (4.7.d) to

$$\gamma_{l+1,j-1} = \Omega_0\ [\gamma_{l,j} + \pi_{l,j} * \sigma_{l,j}]$$

It is clear that in this case, the links $r_{l,j}$, $j=2,\cdots,k$, $l=1,\cdots,k+2-j$ carry redundant information and hence can be removed from the network.

After this description of the network, we turn our attention to the task of simulating its operation. First, we write the SCE program that describes the network and contains the equations that model its nodes. In the following program, the parameter k which determines the size of the network is set to 2.

The SCE program for the network of figure 4.1

```
 ************************
PAR k=2 ;
INDEX i,j ;
SEQN s(0:k,1:k+1) ,
     r(0:k+1,1:k+2) ,
     p(1:k,1:k+2) ,
     c(1:k+1,1:k+1) ,
     b(0:0,0:k+1)    ;

INPUT( MAXT 18, For i=0,k+1 r(i,1) ) ;        /* input statement */

FOR j=1,k+1 DO c(1,j) = o END ;               /* equation (4.10.a) */
b(0,k+1) = o ;                                /* equation (4.10.b) */

s(k,1) = Z [ u / [r(k+1,1) - c(k+1,1)] ] ;    /* equation (4.5) */

FOR i=1,k DO
  r(i,2) = Z [ r(i,1) - c(i,1) ] ;            /* equation (4.6.a) */
  p(i,2) = Z s(i,1) * r(i,2)      ;           /* equation (4.6.b) */
  s(i-1,1)= Z s(i,1)                          /* equation (4.6.c) */
 END ;

FOR j=2,k+1 DO
  FOR i=1,k+2-j DO
    s(i-1,j) = Z s(i,j) ;                     /* equation (4.7.a) */
    p(i,j+1) = Z p(i,j) ;                     /* equation (4.7.b) */
    r(i,j+1) = Z r(i,j) ;                     /* equation (4.7.c) */
    c(i+1,j-1) = Z [ c(i,j) + r(i,j) * s(i,j) ] /* equation (4.7.d) */
  END ;
  s(k+2-j,j)   = p(k+2-j,j)                   /* equation (4.10.c) */
END ;

r(0,2) = Z [ r(0,1) - b(0,1) ] ;              /* equation (4.8.a) */
b(0,0)    = r(0,2) * Z s(0,1)    ;            /* equation (4.8.b) */

FOR j=2,k+1 DO
  r(0,j+1) = Z r(0,j) ;                       /* equation (4.9.a) */
  b(0,j-1) = Z(2) [ b(0,j) + s(0,j) * r(0,j) ] /* equation (4.9.b) */
END ;

OUT( b(0,0) , FOR i=1,k p(i,2) , s(k,1) ) ;   /* output statement */
```

Next, we will use the above program to simulate the computation of the matrices L, U and the vector z for the matrix A given in (4.2). In order to specify the input for this computation, we note that The INPUT statement in the above program limits the length of the sequences to 18 elements. It also determines the order in which the input sequences are read from the input file, namely $\rho_{0,1}$, $\rho_{1,1}$, $\rho_{2,1}$ and $\rho_{3,1}$, respectively. Accordingly, we follow the pattern specified by (4.11.a/b), and use the data from (4.1), to construct the following input file

Input file:

```
0.0   0.0   0.0   4.0    d    d   14.0   d    d   15.0   d   d  0.0   d   d  -6.0   ...
 d     d   6.0    d      d -3.0   d      d -2.0   d      d                   ...
 d   4.0   d      d    15.0 d     d    2.0   d     d    1.0                  ...
2.0    d   d   11.0      d   d    20.0    d   d  -19.0    d   d  14.0 ...
```

Finally, we use the SCE interpreter to run the above program with the given input, this produces the following output file

```
 **** OUTPUT SEQUENCES ****
 0.00 0.00 0.00 0.00 2.00    d     d   2.00    d     d   3.00    d     d  -3.00    d
 d   3.00    d
 **********************
 0.00    d    d 3.00    d    d   -1.00    d    d    2.00    d    d    d    d    d
 d    d    d
 **********************
 0.00    d 2.00    d    d   1.00    d    d  -5.00    d    d  -3.00    d    d    d
 d    d    d
 **********************
 0.00 0.50    d   d   0.33    d    d  -1.00    d    d   0.33    d    d  -0.11    d
 d    d    d
 **********************
```

where as specified by the OUT statement, the sequences are printed in the order $b_{0,0}$, $\pi_{1,2}$, $\pi_{2,2}$ and $\sigma_{k,1}$, respectivelly. It is easy to verify that this output agrees with the results in (4.3) and (4.4), and the formulas (4.12.a/b/c).

5. Conclusion

In this paper, we presented an equation solver to supplement the formal systolic model discussed in [2]. This solver is intended to be used in particular for the verification of systolic networks in the cases where the resulting system of equations modeling the network cannot be solved analytically.

The application of this solver to the verification of a computation of a systolic network is equivalent with the simulation of its operation for specific input data. More specifically, the user specifies the architecture of the network to the solver/simulator in the form of an SCE program that consists in essence of the system of equations modeling the network in a computer readable form. This program is then interpreted by the solver and the output data are computed for specific inputs.

It should be clear that the solver/simulator can only be used to verify specific instances of computations on networks; that is the input data have to be given specific values during the simulation. This limitation applies to any simulator or numeric solver, and the only way to allow for a more general simulation would be to consider symbolic manipulation and to obtain a symbolic description of the outputs in terms of the inputs.

The potential application of the SCE language presented in this paper is not limited to the solver/simulator. Namely, the SCE language may be used to specify precisely any systolic network that can be described in terms of the abstract model. In fact, for a given network, one may write an SCE program in which the causal equations and the sequence declarations describe completely the graph of the network as well as the operation of each of its cells. This SCE specification may be used, for example, as an input to an automatic lay-out program or to a translator that generates specifications in other languages used in computer aided

design of VLSI devices.

The syntax directed approach used in the implementation of the solver/simulator led to a very modular program. This simplifies the task of modifying the solver to incorporate new SCE grammar rules that need to be added when new types of sequence operators are introduced. Actually, the addition of a new sequence operator to the grammar requires only the implementation of a corresponding semantics routine that describes the effect of the operator.

Finally, we note that the solver was extensively tested in the solution of many systems of equations, including those resulting from the networks verified analytically in Melhem Rheinboldt and . Melhem Stiffness In this paper, we applied the solver only to the verification of a network which we designed for the LU factorization of symmetric banded matrices and the solution of the corresponding system of linear equations. This example shows the simplicity and effectiveness of our approach for the simulation of systolic networks.

Acknowledgment

I would like to take the opportunity to acknowledge the support and guidance of professor Werner C. Rheinboldt throughout the course of this research. I would also wish to thank Dr. Mary Lou Soffa for her comments on an earlier version of this paper.

REFERENCES

[1] Kung H. T., "Why Systolic Architecture," *Computer Magazine*, pp.37-46 (Jan. 1982).

[2] Melhem R. G. and Rheinboldt W. C., "A Mathematical Model for the Verification of Systolic Networks," Technical Report ICMA-82-47, The University of Pittsburgh (Oct. 1982). (To appear in SIAM J. on Computing)

[3] Melhem R., "Formal Verification of a Systolic System for Finite Element Stiffness Matrices," Technical report ICMA-83-56 (May 1983). The University of Pittsburgh

[4] Chen M. C. and Mead C. A., "Formal Specification of Concurrent Systems," *USC Workshop on VLSI and Modern Signal Processing* (Nov. 1982).

[5] Chen M. C. and Mead C. A., "A Hierarchical Simulator based on Formal Semantics," *Proc. of the Third Caltech conf. on VLSI* (March 1983).

[6] Johnson S. C., "Yacc: Yet Another Compiler-Compiler," *Computer Science technical report No. 32, 1975*, Bell Laboratories, Murray Hill, NJ07974.

[7] Aho A.V. and Ullman J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. 1977.

[8] Kung H. T. and Leiserson C. E., "Systolic Arrays for VLSI," *in Introduction to VLSI Systems* (1980). ed by Mead C. and Conway L., Addison_Wesley, Reading Mass.

[9] Brent R. R. and Luk F. T., "Computing the Cholesky Factorization using a Systolic Architecture," Technical Report 82-521 (Sept. 1982). Dept. of Computer Science, Cornell University

Appendix A: The grammar for the SCE language.

Terminal symbols

```
PAR         INDEX  SEQN    FOR     DO     END     MAXTIME  INPUT  OUT
Comma       Semi   Colon   Equal   Plus   Minus   Mult     Div
Lbrak       Rbrak  Lcurl   Rcurl   Lpar   Rpar    Period
O           Z      T       A       E      M       U1       U2
Identifier     Positive_integer     Positive_real
```

Grammar rules

```
1)  <prog>          ::= <declare> <in_part> <body> <out_part>
2)  <declare>       ::= <par_decl> <index_decl> <seqn_decl>

        /* PARAMETER DECLARATIONS */

3)  <par_decl>      ::= PAR <par_list> Semi
4)                   |
5)  <par_list>      ::= <par_list> Comma <par_stmt>
6)                   |  <par_stmt>
7)  <par_stmt>      ::= Identifier Equal Positive_integer

        /* INDEX DECLARATIONS */

8)  <index_decl>    ::= INDEX <i_list> Semi
9)                   |
10) <i_list>        ::= <i_list> Comma Identifier
11)                  |  Identifier

        /* SEQUENCE DECLARATIONS */

12) <seqn_decl>     ::= SEQN <dim_list> Semi
13) <dim_list>      ::= <dim_list> Comma <seqn_dim>
14)                  |  <seqn_dim>
15) <seqn_dim>      ::= Identifier Lcurl <range_list> Rcurl
16)                  |  Identifier
17) <range_list>    ::= <range_list> Comma <range>
18)                  |  <range>
19) <range>         ::= <i_expr> Colon <i_expr>
```

```
/* INDEX EXPRESSIONS */

20) <i_expr>          ::= <i_expr> Plus <i_term>
21)                    |  <i_expr> Minus <i_term>
22)                    |  <i_term>
23) <i_term>          ::= <i_term> Mult <i_factor>
24)                    |  <i_factor>
25) <i_factor>        ::= <simple_factor>
26)                    |  Minus <simple_factor>
27)                    |  Lpar <i_expr> Rpar
28) <simple_factor>   ::= Identifier
29)                    |  Positive_integer


/* THE BODY OF THE PROGRAM */

30) <body>            ::= <stmt_list> Semi
31) <stmt_list>       ::= <stmt_list> Semi <stmt>
32)                    |  <stmt>
33) <stmt>            ::= <eqn>
34)                    |  <for_stmt>
35) <for_stmt>        ::= FOR <for_spec> <stmt_list> END
36) <for_spec>        ::= Identifier Equal <i_expr> Comma <i_expr> DO
37) <eqn>             ::= <seq_spec> Equal <seq_expr>


/* SEQUENCE SPECIFICATIONS */

38) <seq_spec>        ::= Identifier Lcurl <indicat_list> Rcurl
39)                    |  Identifier
40) <indicat_list>    ::= <indicat_list> Comma <i_expr>
41)                    |  <i_expr>


/* ELEMENT WISE OPERATORS ON SEQUENCES */

42) <seq_expr>        ::= <seq_expr> Plus <seq_term>
43)                    |  <seq_expr> Minus <seq_term>
44)                    |  <seq_term>
45) <seq_term>        ::= <seq_term> Mult <s_factor>
46)                    |  <seq_term> Div <s_factor>
47)                    |  <seq_term> U1 <s_factor>
48)                    |  <seq_term> U2 <s_factor>
49)                    |  <s_factor>
50) <s_factor>        ::= Positive_real Period <seq_factor>
51)                    |  <seq_factor>
```

/* OPERATORS DEFINED DIRECTLY ON SEQUENCES */

```
52) <seq_factor>    ::= <seq_spec>
53)                   |  <simple_op> <seq_factor>
54)                   |  <multiplex_op> Lpar <choice_list> Rpar
55)                   |  A Lcurl <i_expr> Comma <i_expr> Comma
                            <i_expr> Rcurl <seq_spec>
56)                   |  Lbrak <seq_expr> Rbrak


57) <multiplex_op>  ::= M Lcurl <i_expr> Semi <ratio_list> Rcurl
58) <simple_op>     ::= O <op_power>
59)                   |  Z <op_power>
60)                   |  T <op_power>
61)                   |  E Lcurl <i_expr> Comma <i_expr> Rcurl
62) <op_power>      ::= Lcurl <i_expr> Rcurl
63)                   |
64) <choice_list>   ::= <choice_list> Comma <seq_expr>
65)                   |  <seq_expr>
66) <ratio_list>    ::= <ratio_list> Comma <i_expr>
67)                   |  <i_expr>
```

/* INPUT SPECIFICATIONS */

```
68) <in_part>       ::= INPUT Lpar <inp_list> Rpar Semi
69) <inp_list>      ::= <inp_list> Comma <inp_spec>
70)                   |  MAXT Positive_integer
71) <inp_spec>      ::= <seq_spec>
72)                   |  FOR <io_for> <inp_spec>
73) <io_for>        ::= Identifier Equal <i_expr> Comma <i_expr>
```

/* OUTPUT SPECIFICATIONS */

```
74) <out_part>      ::= OUT Lpar <out_list> Rpar Semi
75) <out_list>      ::= <out_list> Comma <out_spec>
76)                   |  <out_spec>
77) <out_spec>      ::= <seq_spec>
78)                   |  FOR <io_for> <out_spec>
```

****

Apendix B: The listing of the SCE interpreter program.

```
                /** GLOBAL DECLARATIONS **/

#include <stdio.h>

#define Prog_length  650
#define N_rules       78       /* number of rules in the grammar */
#define N_dums        24       /* rules not requiring any action */
#define N_symbol      20       /* size of the symbol table       */
#define N_bound       20       /* size of the bound table        */
#define N_seqn       129       /* max. number of sequences used  */
#define Maxtime       30       /* upper limit on simulation time */
#define stack_length 40        /* length of working stack        */
#define N_words      ((Maxtime + 1) / 16 ) + 1
#define N_ratios      5        /* max. # of arguments in M operators */
#define N_real        5        /* max. # of real constants used      */

int overflow[8] = { Prog_length, stack_length, N_symbol, N_bound,
                    N_seqn,       Maxtime + 1 , N_ratios, N_real } ;

FILE *fopen() , *fp ;

 /* an array to store the program triples */
 struct { char action ;
          int  value  ;
          int  top     ; } prog[Prog_length] ;
 int  location     ;            /* pointer to prog array */

 /* adjust[i] contains the length of the R.H.S. of grammar rule
    i minus one, which is the adjustment in the top of the stack */
 int  adjust[N_rules] = {-3, -2, -2,  1, -2,  0, -2, -2,  1, -2,
                          0, -2, -2,  0, -3,  0, -2,  0, -2, -2,
                         -2,  0, -2,  0,  0, -1, -2,  0,  0, -1,
                         -2,  0,  0,  0, -3, -5, -2, -3,  0, -2,
                          0, -2, -2,  0, -2, -2, -2, -2,  0, -2,
                          0,  0, -1, -3, -8, -2, -5, -1, -1, -1,
                         -5, -2,  1, -2,  0, -2,  0, -4, -2, -1,
                          0, -2, -4, -4, -2,  0,  0, -2
                        };

 /* dums[] contains the number of the grammar rules not requiring
    any action */
 int  dums[N_dums]    = { 3,  4,  5,  6,  8,  9, 12, 13, 14, 22,
                         24, 25, 29, 31, 32, 33, 34, 44, 49, 51,
                         69, 74, 75, 76
                        };

 int stack[stack_length]     ;                 /* dynamic working stack */
 float fstack[stack_length] ;                  /* a matching value_stack */

 /* SYMBOL TABLE ; type= S, P or I for seqn., parameter or index, respectively. */
 struct { char type  ;
          int  entry1;
          int  entry2; } sym_tab[N_symbol] ;

 int lbound[N_bound] ;                         /*  lower bound table */
 int ubound[N_bound] ;                         /*  upper bound table */
 int ran_ptr = -1    ;                         /* pointer to bound table */

 /* SEQUENCE STORAGE */
 float seq_store[N_seqn][Maxtime+1]       ;  /* sequences storage */
 unsigned d_table[N_seqn][N_words]        ;  /* keeps tracks of don't cares*/
 int seq_ptr = 0     ;                       /* pointer to sequence store */
 int seq_size        ;                       /* actual size of the table  */
 int last_computed[N_seqn]                ;  /* for consistancy checks    */

 float r_store[N_real]                    ;  /* storage for real constants */
 int   r_ptr = -1                         ;  /* the corresponding pointer  */

 int ratio_temp[N_ratios] ;                  /* to store multiplexer ratios */
 int ratio_ptr            ;                  /* a pointer to ratio_temp */

 int not_done     ;                          /* a loop control variable */
```

```
/*************************************************************/
/*                        THE MAIN PROGRAM                   */
main()
{
 char action     ;
 int  value      ;
 int  top = -1   ;                        /* current top of stack */
 int j, looking  ;
 int stage       ;
 int end_stage[5] ;

 end_stage[1]=2 ; end_stage[2]=68 ;
 end_stage[3]=30 ; end_stage[4]=1 ;

 /* open the file that contains the program tuples */
 fp = fopen("out.parse", "r") ;
 for(seq_ptr=0 ; seq_ptr < N_seqn ; seq_ptr++)
    for(j=0 ; j < N_words ; j++)
       d_table[seq_ptr][j] = 0 ;
 seq_ptr = 0 ;

 /* Build standard entries in symbol table, they can be
    overwritten by proper declaration */
 for(j=0 ; j < 3 ; j++)
  { sym_tab[j].type = 'S' ;
    sym_tab[j].entry1 = seq_ptr ;
    last_computed[seq_ptr++] = Maxtime ;
    sym_tab[j].entry2 = -1 ;              /* indicating a single sequence */
  }
 for(j=3 ; j < N_symbols ; j++)
    sym_tab[j].type = ' ' ;
 for(j=1 ; j < Maxtime ; j++)
  { seq_store[1][j] = 0.0 ;
    seq_store[2][j] = 1.0 ; }
 for(j=0 ; j < N_words ; j++)
    d_table[0][j] = 0177777  ;

 for(j=3 ; j < N_seqn ; j++)
    last_computed[j] = 0      ;

/*    END OF INITIALIZATION AND BEGINNIG OF MAIN LOOP   */

 /* Outer for loop: stage=1 -> declarations,
                    stage=2 -> input part,
                    stage=3 -> program body,
                    stage=4 -> output part. */
 for(stage=1 ; stage <= 4 ; ++stage)
 {
  location = 0  ;
  not_done = 1  ;
  /* inner loop 1: read the tuples for the corresponding stage
     from file, keep track of the top of the stack and save in
     prog[] only those triples that require a certain action */
  do
  {
   fscanf( fp , "%c" , &action ) ;
   if(action != 'L') fscanf( fp , "%d\n" , &value ) ;
    else { check(7, ++r_ptr) ;
           fscanf( fp , "%f\n" , &r_store[r_ptr] ) ; }
   switch (action)
   {
    case 'R' : { looking = 1 ; j = 0 ;                /* REDUCE */
                 while(looking && j < N_dums)
                   if(value == dums[j++]) looking = 0 ;
                 if(looking) push_triple(action, value, top) ;
                 check(1, (top += adjust[value-1]) );
                 if(value == end_stage[stage]) not_done=0 ;
                 break                          ;
               }
    case 'C' : { check(1, ++top)              ;   /* SHIFT  IDENTIFIER */
                 push_triple(action, value, top );   /* OR INTEGER CONSTANT */
                 break                        ;
               }
```

```
    case 'L' : { check(1,++top) ;                           /* SHIFT REAL CONSTANT */
                 push_triple(action, r_ptr, top) ;
                 break ;
               }
    case 'S' : { check(1,++top) ;                           /* SHIFT */
               break           ;
               }
    case 'A' : { check(1, ++top)                ;           /* ACCEPT */
                 push_triple(action, value, top );
                 break          ;
               }
   }
  }
  while(not_done) ;

  location = -1 ;
  not_done = 1  ;
  /* inner loop 2: execute the action routines for that stage */
  while(not_done)
  {
   ++location ;
   value = pros[location].value ;
   top   = pros[location].top   ;
   switch( pros[location].action )
   {
     case 'C' : { stack[top] = value ; break ; }
     case 'L' : { fstack[top] = r_store[value] ; break ; }
     case 'R' : { semantic(value, top) ; break ; }
     case 'A' : not_done = 0 ;
   }
  }
 }
 fclose(fp) ;
}
/*           END OF THE MAIN PROGRAM      */


/*****************************************************************/
/*
A routine to store a triple in the array pros[]
*/
push_triple(a , v, t) char a ; int v,t ;
{
 pros[location].action = a ;
 pros[location].value  = v ;
 pros[location].top    = t ;
 check(0, ++location)      ;
 return(0)                 ;
}
/*****************************************************************/
```

```
/**********************************************************************/
/*                       THE SEMANTICS ROUTINES                       */
/**********************************************************************/

int declaring = 1 ;            /* =1 only during declaration       */
int skip = 0      ;            /* to skip calculation in case of don't cares */
int Mskip= 0      ;            /* to chose the argument in M operator        */
float getfloat()  ;
float tfloat      ;            /* temporary variable                         */
int TIME = 1      ;            /* global system time                         */
int d_flag        ;

semantic(rule, top) int rule,top ;
{
 int t0, t2, reading , j ;

 if(Mskip)
  { switch(rule)
    { case 54 : break ;
      case 64 : --stack[top-4] ;
      case 65 : --Mskip        ;
      default : return(0)      ;
    }
  }

 if(skip)
 { switch(rule)
    { case 53 :
      case 54 : { --skip ; return(0) ; }
      case 57 :
      case 58 :
      case 59 :
      case 60 :
      case 61 : { skip++ ; return(0) ; }
      default : return(0) ;
    }
 }

 switch(rule)
 {
  case 1 : { not_done = 0 ; return(0) ; }       /* signal end of stage 4 */
  case 2 : { declaring= 0 ;                     /* signal end of declarations */
             not_done = 0 ;                     /* that is stage 1            */
             seq_size = seq_ptr - 1 ; return(0) ; }

/*         PARAMETER DECLARATION        */

  case 7 : { t2 = stack[top-2] ; check(2,t2) ;
             if((t2 > 2) && (sym_tab[t2].type != ' ')) run_error(15) ;
             sym_tab[t2].entry1 = stack[top] ;
             sym_tab[t2].type   = 'P'        ;
             return(0) ; }

/*         INDEX DECLARATIONS           */

  case 10:
  case 11: { t0 = stack[top]    ; check(2,t0) ;
             if((t0 > 2) && (sym_tab[t0].type != ' ')) run_error(15) ;
             sym_tab[t0].entry1 = 0          ;
             sym_tab[t0].entry2 = 0          ;
             sym_tab[t0].type   = 'I'        ;
             return(0) ; }

/*         SEQUENCE DECLARATIONS          */

  case 15: { check(3, ++ran_ptr) ;
             lbound[ran_ptr] = 0 ;
             ubound[ran_ptr] = 0 ;
             seq_ptr = seq_ptr + stack[top-1] ;
             check(4, seq_ptr)    ;
             return(0) ; }
  case 16: { t0 = stack[top] ;
             if((t0 > 2) && (sym_tab[t0].type != ' ')) run_error(15) ;
             sym_tab[t0].type   = 'S'      ;
             sym_tab[t0].entry1 = seq_ptr  ;
             sym_tab[t0].entry2 = -1       ;
             check(4,++seq_ptr)            ;
             return(0) ; }
```

```
        case 57: { t2 = stack[top-3] ; t0 = stack[top-1] ;
                   if(TIME < t2) {skip=1 ;
                                  stack[top-5]  = 1 ;
                                  fstack[top-5] = 0 ; return(0) ; }
                   t2 = (TIME - t2) % t0 ;
                   for(j=0 ; t2 >= ratio_temp[j] ; j++)   ;
                   stack[top-5] = ratio_ptr ;              /* cardinality of list*/
                   Mskip = j                ;              /* chosen element     */
                   return(0) ; }

        case 58: { t0 = stack[top] ;
                   if(TIME <= t0) { skip=1 ;
                                    stack[top-1]  = 1 ;
                                    fstack[top-1] = 0 ; return(0) ;}
                   stack[top-1] = TIME     ;
                   TIME = TIME - t0        ;
                   return(0) ; }
        case 59: { t0 = stack[top] ;
                   if(TIME <= t0) { skip = 1 ;
                                    stack[top-1]  = 0 ;
                                    fstack[top-1] = 0 ; return(0) ; }
                   stack[top-1] = TIME ;
                   TIME = TIME -t0     ;
                   return(0) ; }
        case 60: { t0 = stack[top] ;
                   t2 = (TIME + t0) % (1 + t0) ;
                   if(t2) { skip =1 ;               /* don't care */
                            stack[top-1]  = 1  ;
                            fstack[top-1] = 0  ; return(0) ;}
                   t2 = (TIME + t0) / (1 + t0) ;
                   stack[top-1] = TIME         ;
                   TIME  = t2                  ;
                   return(0) ; }
        case 61: { t0 = stack[top-1] ; t2 = stack[top-3] ;
                   if(TIME < t2) { skip = 1 ;
                                   stack[top-5]  = 1 ;
                                   fstack[top-5] = 0 ; return(0) ; }
                   stack[top-5] = TIME        ;
                   TIME = TIME - ((TIME - t2) % t0)      ;
                   return(0) ; }
        case 62: { stack[top-2] = stack[top-1] ; return(0) ; }
        case 63: { stack[top+1] = 1 ; return(0) ; }
        case 64: { --stack[top-4] ;
                   stack[top-2] = stack[top] ;
                   fstack[top-2] = fstack[top] ;
                 }
        case 65: { --Mskip ;
                 return(0) ; }
        case 66: { check(6, ++ratio_ptr) ;
                   stack[top-2] += stack[top] ;
                   ratio_temp[ratio_ptr] = stack[top-2] ;
                   return(0) ; }
        case 67: { ratio_ptr = 0 ;
                   ratio_temp[0] = stack[top] ;
                   return(0) ; }

      /*     INPUT SPECIFICATIONS */

        case 68: { not_done = 0      ; /* end of stage 2 */
                   return(0) ; }
        case 70: { stack[1] = stack[top] ;
                   check(5, stack[top]) ;
                   return(0) ; }
        case 71: { t0 = stack[top] ; reading = 1 ;
                   for(j=1 ; j <= stack[1] ; j++)
                    if(reading)
                    { seq_store[t0][j] = getfloat(d_flag) ;
                      if(d_flag == 1)  write_d(t0,j)      ;
                      if(d_flag == -1) { reading = 0 ;
                                         write_d(t0,j) ; }
                    }
                    else write_d(t0,j) ;
                   last_computed[t0] = stack[1] ;
                   return(0) ; }
        case 72: { t0 = stack[top-1] ;
                   if(sym_tab[t0].entry1 >= sym_tab[t0].entry2) return(0) ;
                   ++sym_tab[t0].entry1   ;
                   location = stack[top-2] ;
                   return(0) ; }
```

```
  case 73: { t2 = stack[top-4] ; check(2,t2) ;
             if(sym_tab[t2].type != 'I') run_error(6) ;
             sym_tab[t2].entry1 = stack[top-2] ;
             sym_tab[t2].entry2 = stack[top]   ;
             stack[top-5] = location ;
             return(0) ; }

/*    OUTPUT SPECIFICATIONS */

  case 77: { t0 = stack[top] ;
             for(j=1 ; j <= stack[1] ; j++)
              if(read_d(t0,j)) printf("   d  ") ;
              else printf("%5.2f ", seq_store[t0][j])  ;
             printf("\n *********************\n") ;
             return(0) ; }
  case 78: { t0 = stack[top-1] ;
             if(sym_tab[t0].entry1 >= sym_tab[t0].entry2) return(0) ;
             ++sym_tab[t0].entry1 ;
             location = stack[top-2] ;
             return(0) ; }
 }
}
/*           END OF THE SEMANTICS ROUTINES          */



/**************************************************/
/*         User Defined Operators                 */
/**************************************************/
/*
These routines are provided by the user to define the
binary operators U1 and U2. The operands are passed in
o1 and o2 and the result is returned in r.
If any of the operands is the don't care symbol, t1 or t2
is set to 1, correspondingly, otherwise they are set to 0.
The return value of the functions should be 0 if the
result of the operation is not a don't care and 1 if it is.
*/
u_op1(o1,t1,o2,t2,r) int t1,t2 ;
                     float o1,o2, *r ;
{
/* Formulas for u_op1 and r */
}

u_op2(o1,t1,o2,t2,r) int t1,t2 ;
                     float o1,o2, *r ;
{
/* Formulas for u_op2 and r */
}

/**************************************************/
```

```
/*************************************************************************/
/*
The following routine reads the next item from the
input data file. It assumes that one of the following
exists on the file:
1) a floating point number,
2) a 'd' ; indicating a don't care item,
3) or '...' ; indicating that the remaining items in
the current sequence are don't cares.
The different cases are signaled by the global flag d-flag
*/
float getfloat()
{
 int c, int_part ;
 int minus = 1    ;
 float fraction , p_of_10 ;

 while((c=getchar()) == ' ' || c == '\n' ) ;
 if(c == EOF ) run_error(9) ;
 if(c == 'd' ) { d_flag = 1 ;          /* a don't care symbol */
                 return(0) ;}
 if(c == '.' && (getchar() == '.') && (getchar() == '.') )
                { d_flag = -1 ;        /* sequence terminated */
                  return(0) ;}

 if(c == '-') { c=getchar() ; minus = -1 ;}
 int_part = 0 ;
 while(isdigit(c)) { int_part = (10 * int_part) + (c - '0') ;
                     c = getchar() ; }
 if( c != '.') run_errror(8) ;
 fraction = 0.0 ;
 p_of_10 = 10.0 ;
 while(isdigit(c=getchar())) {fraction = fraction + (c - '0') / p_of_10;
                              p_of_10 = p_of_10 * 10.0 ; }

 if((c != ' ') &&  (c != '\n') ) run_error(8) ;
 fraction = minus * ( int_part + fraction ) ;
 d_flag = 0 ;
 return(fraction) ;
}

isdigit(d) int d ;
{
 if(d <= '9' && d >= '0') return(1);
 return(0) ;
}
/*************************************************************************/
/*
The following routines keep track of the position of the
don't care symbols in the data sequences; Each entry in
a sequence has a corresponding bit in the array d_table,
write_d(s,t) sets the bit corresponding to the element t
of the sequence s to 1 indicating a don't care , while
read_d(s,t) returns the value of the bit corresponding
to the element t in sequence s.
*/
write_d(s,t) int s,t ;
{
 int word, bit ;
 unsigned pattern = 1 ;
 word = t / 16 ;
 bit = (t % 16) ;
 pattern = pattern << (15-bit) ;
 d_table[s][word] = (d_table[s][word]) | pattern ;
 return(0) ;
}

read_d(s,t) int s,t ;
{
 int word, bit ;
 unsigned pattern = 1 ;
 word = t / 16 ;
 bit = t % 16 ;
 pattern = pattern << (15-bit) ;
 pattern = pattern & d_table[s][word] ;
 if(pattern) return(1) ;
 return(0) ;
}
```

```
        /**** ERROR ROUTINES ******/

/*
A routine to check the bounds of working arrays, the array
to be checked is determined by the  argument i.
*/
check(i, ptr) int i,ptr ;
{
 if(ptr < overflow[i]) return(1) ;
 switch(i)
 {
  case 0 : { printf("*** program array overflow ***\n") ;
             exit(0) ; }
  case 1 : { printf("*** working stack overflow ***\n") ;
             exit(0) ; }
  case 2 : { printf("*** symbol table overflow  ***\n") ;
             exit(0) ; }
  case 3 : { printf("*** bound table overflow   ***\n") ;
             exit(0) ; }
  case 4 : { printf("*** sequence store overflow **\n") ;
             exit(0) ; }
  case 5 : { printf("*** MAXT should be less than %d\n", Maxtime);
             exit(0) ; }
  case 6 : { printf("*** temporary ratio list overflow **\n");
             exit(0) ; }
  case 7 : { printf("*** real constants storage overflow **\n");
             exit(0) ; }
 }
 exit(0) ;
}

/*****************************************************/
/*
A procedure to print run time error messages and stop
execution. The message to be printed is determined
by the argument i.
*/
run_error(i) int i ;
{
 switch(i)
 {
  case 1 : { printf("** sequence array out of bound \n");
             exit(0) ; }
  case 2 : { printf("** too many array arguments \n") ;
             exit(0) ; }
  case 3 : { printf("** too few  array arguments \n") ;
             exit(0) ; }
  case 4 : { printf("** only parameters may be used in sequ. declaratrion \n");
             exit(0) ; }
  case 5 : { printf("** expecting a par. or an index in seq. specification \n");
             exit(0) ; }
  case 6 : { printf("** FOR variables must be declared as INDEX ") ;
             exit(0) ; }
  case 7 : { printf("** wrong number of arguments in Multiplexer list \n");
             exit(0) ; }
  case 8 : { printf("** Format error in input file **\n") ;
             exit(0) ; }
  case 9 : { printf("** insufficient data in input file **\n");
             exit(0) ; }
  case 10: { printf("** incorrect model or non causal equations **\n");
             exit(0) ; }
  case 11: { printf("** Inconsistent system of equations \n") ;
             printf("** Attempt to overwrite a sequence  \n") ;
             exit(0) ; }
  case 12: { printf("** array of sequences not declared  \n") ;
             exit(0) ; }
  case 13: { printf("** sequence not declared \n") ;
             exit(0) ; }
  case 14: { printf("** missing argument list \n") ;
             exit(0) ; }
  case 15: { printf("** variable already declared \n ") ;
             exit(0) ; }
 }
}

      /****** END OF PROGRAM LISTING  ********/
```

generates specifications in other languages used in computer aided

END

FILMED

8-83

DTIC