SADE: A PROGRAMMING ENVIRONMENT FOR DESIGNING AND TESTING SYSTOLIC ALGORITHMS

Jean-Pierre Corriveau(+)
and
Nicola Santoro (++)

SCS-TR-40
February 1984

(+) Computer Science Department, University of Ottawa. Presently at Department of Computer Science, University of Toronto

(++) Distributed Computing Group, School of Computer Science, Carleton University

This work was supported in part by the Natural Sciences and Engineering Council of Canada under Grant A2415, and is based on Corriveau's Master Thesis.

A preliminary version of this paper has been presented at the IASTED International Conference on Computer Aided Design, Nice, June 19-21, 1984

1. SADE: A Systolic Algorithm Design Environment

1.1 Introduction

In this paper, we describe the major features of a design environment, for the specification and testing of systolic algorithms. This environment is based on the concept of a 'reconfigurable systolic architecture' (RSA) [3] which we study in section 1.3.

Based on this idea, we have focused on the problem of 'embedding' structures onto the reconfigurable systolic architecture. Such embedding process must be a systolic process itself.

Many of the results of our investigation on the RSA have been incorporated in the Systolic Algorithm Design Environment (SADE). In particular, SADE allows for both (standard) graphical as well as systolic specification of topologies. Other factors which have influenced the realization of SADE include the beliefs that a standardization of design techniques for systolic algorithms is not only desirable but also easily attainable, and that ad-hoc simulation tools would not be general enough to satisfy the design needs of different systolic structures.

SADE presents the user with a simple yet powerful tool, which allows the specification of arbitrary systolic structures, and can simplify the verification process of complex systolic algorithms. Furthermore, the tool is portable, extensible and provides for the graphical simulation of an algorithm's execution on a predefined structure.

A basic characteristic of SADE is that the two fundamental steps of the design task [1] are clearly separated:

(a) specification of the structure on which the algorithm will be executed (i.e. topology of the connections between the cells and cell type).

(b) design of the algorithm itself, achieved by specifying the algorithmic behaviour of each (type of) cell within the structure.

We shall devote a chapter to the study of each of these two steps, but first we will briefly examine the basic ideas of SADE.

1.2 VLSI Design Rules

SADE follows the VLSI design guidelines stated in [7,10]:

1. A systolic structure must have a simple and regular design for its control and data flows.

2. There must as few types of cells as possible.

3. All the cells are fully synchronized and a clock pulse is taken to be the time required for a cell to complete its corresponding algorithm. Currently, self-timed computations cannot be simulated using SADE's routines.

4. A cell can only be assigned a local (i.e. non-shared) memory; this is to say that a cell cannot examine the contents of its neighbours.

1.3 The Reconfigurable Systolic Architecture

Recall that several alternatives exist for the implementation of systolic cells [8]. Among those, the programmable building block approach [4] is the most desirable one from a flexibility viewpoint.

Recently, Kung has introduced the concept of a systolic system on a wafer [9], which offers a very economical, efficient alternative to both chip and board implementations. Informally, the idea is that an entire systolic structure (cells and wires) can be directly etched in a silicon wafer, thus drastically reducing area, communication and control costs. It is possible that the wafer may contain faulty processors; however, in this paper, we will assume an 'ideal' manufacturing process which produces flawless wafers!

Combining the programmable building block approach with the 'system on a wafer' concept, we propose a reconfigurable systolic architecture (RSA) which allows for the execution of different systolic algorithms on different systolic structures, all embedded on the same wafer. The motivations for our investigation of this architecture lie upon two observations:

1. The structures used by most of the existing systolic algorithms are from a very small set of topologies.
2. All these topologies (as well as many other unused ones) are easily "embeddable" in a square of hexagonally-connected processors, as shown in the figure below.

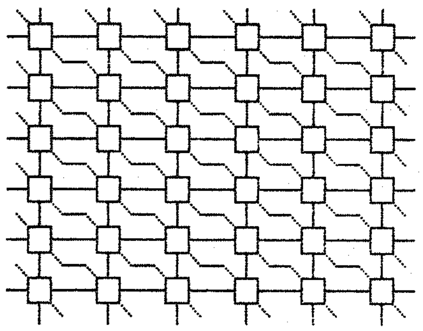

Figure 1. A square of hexagonally-connected processors.

A cell and its numbered links are displayed in figure 2.

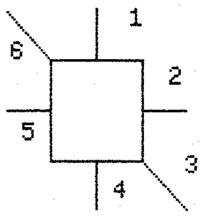

Figure 2. A cell of an RSA and its links.

The reconfigurable systolic architecture consists of a square wafer of hexagonally-connected <u>programmable synchronized</u> processors, where each processor (i.e. cell) of the RSA has a set of activated links and can execute several different algorithms. In particular, since every cell may run an algorithm to modify its set of activated links, it becomes possible to repeatedly change the systolic structure embedded in the RSA.

When a systolic structure is embedded, some processors function as links between the active nodes of the structure. Following Chazelle and Monier's model [2], we assume that it takes one clock pulse to traverse such links. In other words, the time of propagation across a wire is at least proportional to the length of this wire.

Several of the general VLSI design rules stated in [6,7,10] are implicitly enforced by this architecture. In particular, it becomes impossible to implement systolic structures with long or irregular wires. If two non-adjacent active nodes of a structure must communicate, then some processor(s) must act as link(s) between these two cells. Furthermore, the computation of the area is greatly simplified and standardized. When computing the area of a systolic structure, we must include <u>all</u> the processors in the smallest square containing the structure. We justify the latter claim by observing that a host computer will probably be connected to several wafers (of possibly different sizes), each consisting of an RSA. After having recognized a problem for which the computer can use a systolic algorithm, the host will decide which size of RSA offers the best per processor utilization's ratio and allocate an available RSA accordingly. An optimal allocation can only be made if the host uses the area computation method we suggested above.

1.4 The Placement Algorithm [5]

The placement of a systolic structure onto the RSA can be done either in a centralized manner, i.e. totally from the outside (by the host), or in a distributed fashion, i.e. internally performed by the systolic cells. The first approach relies on a "configuration string" externally generated, containing setup instructions that are distributed to all relevant cells. In the second approach, the process is initiated externally, but the exact configuration is determined internally by the processors. An important advantage of a distributed placement algorithm is that it is independent of the exact size of the available RSA. It dynamically places the structure and consequently, RSAs of different sizes may use the same placement algorithm. Furthermore, since the RSA consists of programmable systolic chips [4] which are powerful microprocessors, there is no increase in the complexity of a systolic cell. The distributed placement algorithm obviously must be a systolic process itself to be used with the RSA.

1.5 Transferable Data Types

In an actual implementation of an RSA, only bits can be transferred from one cell of the structure to another. Furthermore, the wires between these cells must have a fixed bandwidth. Specifying data transfers at the bit level is a cumbersome task that SADE avoids via a simple abstraction mechanism.

In SADE, the designer does not have to worry about the bandwidth of the wires of the RSA. The messages transferred from one cell to another consist of two entities:

1. a data type descriptor from the following enumeration:

    (a) INT denoting an integer

    (b) REL denoting a real

    (c) CHA denoting a character

(d) STR denoting a string of characters

2. a message value which must be compatible with the specified data type descriptor.

Chapter 3 details the use of these messages (which are implemented using a variant record).

1.6 SADE's Philosophy

In SADE, the underlying structure, an RSA, is a square of hexagonally-connected processors; all user-defined structures are obtained by 'cutting' the RSA.

The design process is conceptually divided in two distinct steps. The user must first define a structure on which one or more systolic algorithms are to perform. This is achieved by dynamically cutting the structure out of the RSA; i.e. by specifying, for each cell, its "activated" links and its cell type. This cutting process duplicates the activation mechanism which would be used in an actual reconfigurable systolic architecture. Hence <u>the cutting algorithm is in itself a systolic algorithm.</u>

The second step associates a procedure to each different type of cell in the defined (i.e. cut) systolic structure; within each procedure, the local memory and the data transfers corresponding to the cell type are fully described.

For example, if we consider the problem of multiplying two n x n matrices, we can use the diamond-shaped structure proposed by Leiserson and Kung [10,11] (see figure 3). In SADE, the first step of the design consists of a cutting algorithm that embeds this diamond in an RSA. In this example, all cells have a same cell type and an identical topology. The second design step must describe the systolic algorithm used to multiply the two matrices. In SADE, this is simply achieve by defining the algorithmic behaviour of each cell. For this example, all cells execute a same algorithm which consists of a multiply-accumulate operation (i.e. multiply the two current inputs and add this product to the sum stored in the cell).

SADE is implemented as a set of predefined MODULA-2 routines [13]. This approach eliminates the need for a separate compiler, allows the user to work in a somewhat familiar PASCAL-like environment, and thus reduces apprenticeship to a minimum so that the novice can focus on the design rather than on the syntax. Several features are included to avoid the repetition of work by the designer. The tool is not restricted to systolic algorithms and can be used for certain types of distributed algorithms.

1.7 SADE's Modus Operandi.

In SADE, the entire design process can be handled in one of two modes: procedural and graphical. In the procedural mode, both structure specification and algorithm definition are performed via high-level programs; in the graphical mode, the structure can be specified via an interactive graphical package. In this paper, we will focus on the procedural mode; some of the features of the graphical mode are briefly discussed in chapter 4.

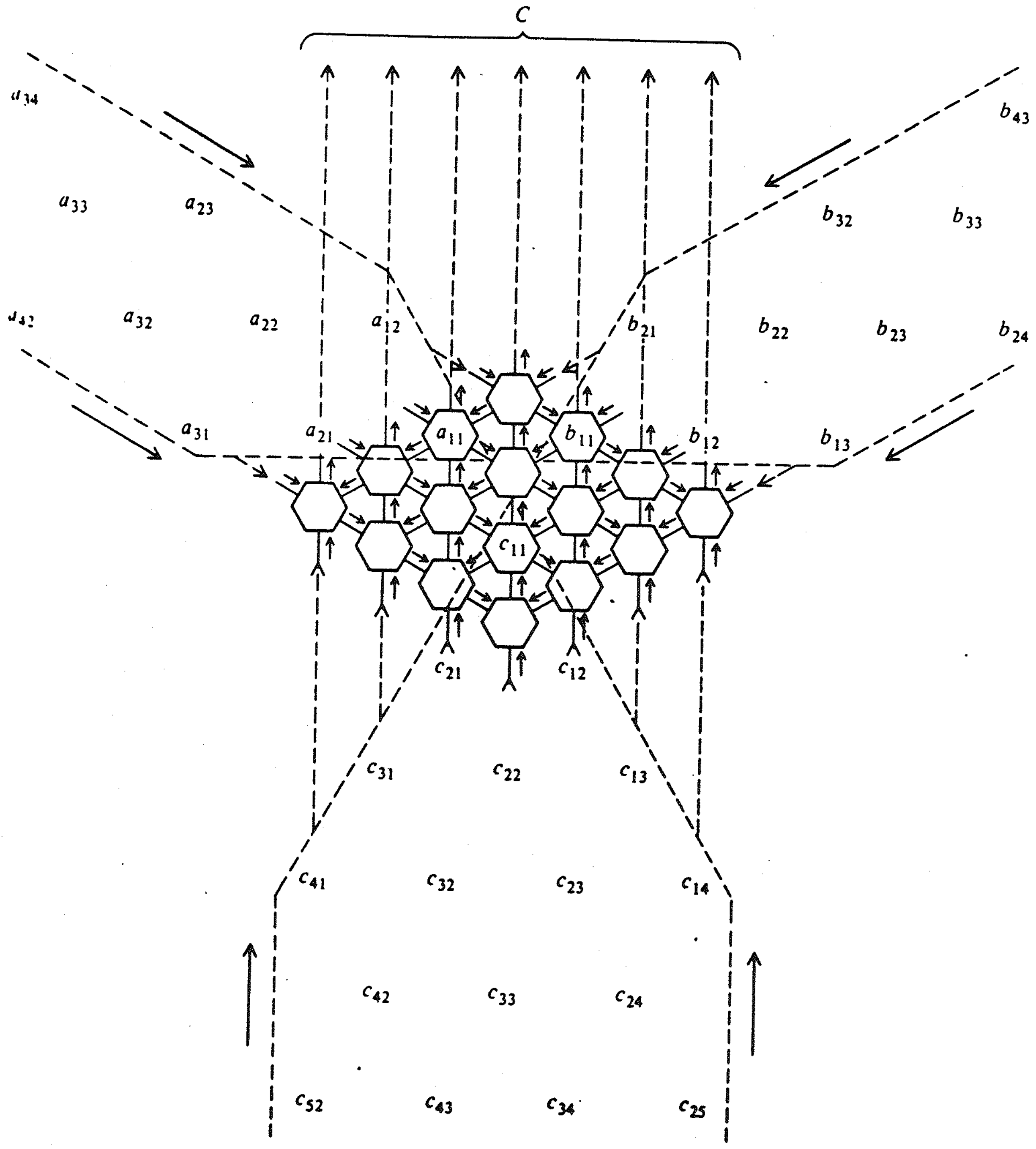

Figure 3. A Systolic Structure for Matrix Multiplication

2. Systolic Structure Specification

2.1 Introduction

In SADE, the first step of the design process is the definition of a **cutting algorithm** that places the targeted systolic structure onto the RSA. In the procedural mode, this cutting algorithm is coded as a MODULA-2 program that calls SADE's routines to actually embed the structure in an RSA which may be subsequently archived.

This chapter introduces these cutting routines and illustrates their use via several examples. The basic activation mechanism of SADE is discussed in section 2.2. The programming techniques used for inter-module communication and recursion elimination are studied in section 2.3. SADE's main cutting and archiving routines are then described in section 2.4. Subsequent sections refer to several simple examples (that were run on an APPLE-IIe microcomputer) to display SADE's versatility and typical program organization.

2.2 Basic Principles

The cells on the boundaries of the RSA are assumed to be directly connected to invisible I/O ports; therefore, certain links of these boundary cells will carry an output to (or an input from) the outside world when appropriate. For example, the cell in the upper left corner of an RSA will generate an output if data flows out of its links 1,5 or 6 (see figure 2).

To simplify the expression of the cutting algorithm as a MODULA-2 program, a system of planar coordinates is imposed on the RSA; according to this system, the corners and boundaries are identified as follows:

Corners:

UL:Upper Left, UR:Upper Right, LL:Lower Left, LR:Lower Right

Boundaries:

TB:Top, RB:Right, LB:Left, BB: Bottom

For simplicity, the cutting process is always initiated from one of the four corners.

Since a user-defined structure does not necessarily activate the entire RSA, a number of cells in the RSA might not be part of the structure and thus play only a 'passive' role in the algorithm's execution. These passive cells may however act as relayers of data; in fact, since i/o is performed only on the boundary cells of the RSA, the i/o data must be routed from the active cells to the I/O ports (and vice versa) through passive ones. We shall refer to those passive carriers as **relayers.** The size of the RSA, which directly controls the number of relayers, is interactively specified at the start of the execution of the cutting algorithm.

During the cutting process, some links will become **in-edges,** others **out-edges.** Roughly speaking, data flows in and out of a cell through an in-edge and an out-edge respectively; any link can be simultaneously both an in-edge and an out-edge. The out-edges are activated (i.e. specified as such) by the designer; the in-edges can be automatically activated by SADE, or may be explicitly listed by the designer.

Recall that the cutting algorithm is itself a systolic algorithm: the cells of the structure are activated in a distributed fashion. The host computer first creates a packet containing the information used by each cell to establish its own configuration. The host then activates the first cell of the structure by sending it this packet. Each cell that receives a packet follows the steps listed below:

1. Consult the information of the received packet to select your own activated links and cell type.
2. Select the adjacent cells (i.e. neighbours) that must be activated.
3. Send a packet with the appropriate updated information to each selected neighbour.

This description of the cutting strategy is direclty duplicated in the organization of the MODULA-2 program that represents it. In the main program, the designer creates the initial packet, chooses the starting corner cell and initiates the cutting process which is controlled by SADE. The designer then needs to code the procedure (henceforth called the configuration procedure) that describes the algorithm that each activated cell uses to establish its configuration. This procedure proceeds directly from the steps mentioned above and requires one parameter which gives access to the packet of information from which the configuration is derived.

2.3 Implementation Considerations

The technique used to code a cutting algorithm is quite simple but requires a specific feature of MODULA-2 to handle inter-module communication. An a priori discussion of some relevant implementation details should prevent the reader from feeling SADE does not lead to easier algorithm designing.

SADE obviously must not import any data objects from the designer's program if it is not to be recompiled for each cutting algorithm! Thus the parameter, used to give the configuration procedure, access to its corresponding packet, must be of a predefined type. The information packet is implemented as a user-defined record which is therefore invisible to SADE. A pointer to this record still is an invisible type. In fact, the predefined type ADDRESS (which is compatible with any pointer type) is the only efficient solution. The parameter of the configuration procedure therefore denotes the system address of the packet.

Unfortunately, using an ADDRESS pointer leads to pecularities (due to strong-type checking) of the MODULA-2 language: one needs an assignment statement to store the packet-record pointer into a variable of type ADDRESS and another for the inverse operation. Some experience with MODULA-2 should correct this initial impression of futile complexity.

As for the inter-module control flow, we remark that the cutting process cannot be implemented using recursion, for the simple reason that even a small RSA will inevitably cause a stack overflow (on a 64K machine)!! SADE implements the cutting process as a linked list of events (which allows for the simulation of the inherent parallelism of this process): at each clock pulse a certain number of events occur. Each event, which denotes the activation of a specific cell, consists in some internal information that uniquely identifies this cell, the name of the procedure that is used to set the cell's configuration and finally, an address pointer to the packet used for this cell.

Relayers are treated somewhat differently than other cells of the structure and are studied in subsection 2.4.4.

A clear understanding of the implementation details discussed in this section will greatly simplify the following study of SADE's cutting routines.

2.4 SADE's Modules and Routines

SADE is implemented using three different library modules:

1. SADECUT which contains all the routines used to cut a systolic structure out of an RSA (of size inferior to a system-dependent constant).
2. SADEIO which holds the archiving procedure used to store a 'cut' systolic structure into a file (on a diskette).
3. SADEALG which holds the routines used to retrieve a cut systolic structure (from the diskette), define the algorithmic behaviour of each of its cell types and execute the corresponding systolic algorithm(s).

When using a library module, the programmer only has access to the objects exported by the module. In this section, we shall study the data types and procedures exported by the first two modules of SADE. (Refer to the listing of SADECUT's definition module, found in Appendix 1, for an exact description of the parameter list of each procedure.)

2.4.1 BOUNDARIES and CORNERS

The two first objects exported by SADECUT are the types BOUNDARIES and CORNERS which implement the conventions adopted in section 2.2. Here are their definitions:

1. CORNERS=(UL,UR,LL,LR)
2. BOUNDARIES=(TB,LB,RB,BB)

2.4.2 Procedure CUTCELL

SADECUT uses an internal representation of the RSA to keep track of the cutting process. Within this model, each cell is denoted by a record which has a set of in-edges, a set of out-edges and a cell type. As we have mentioned earlier, SADE is able to extrapolate the in-edges of a cell from the out-edges of its neighbours. The procedure CUTCELL is called to cut the out-edges of a cell, to give it a cell-type (an integer) and to create an event that will cut one selected neighbour on the next clock pulse. The specification of the event requires:

(a) the link used to get to the selected neighbour

(b) the name of the configuration procedure to use for this neighbour

(c) the address of the corresponding updated packet.

For example the call **CUTCELL({2,5},4,2,KIND4,T)** requires SADECUT to perform the following list of actions:

1. Activate links 2 and 5 as out-edges of the current cell.

2. Store 4 as the cell-type of the current cell.

3. Use the internal position of the current cell to compute the position of the neighbour reached via its link 2.

4. Create an event, for the next clock pulse, that, when executed:

   (a) makes the neighbour of the previous step (accessed via link 2) the current cell.

   (b) calls the specified configuration procedure (KIND4), using the variable T (of type ADDRESS) as its parameter.

5. Insert this event in an internal list of events ordered by time.

So far, we have described a SADE cutting program where only one configuration procedure was used. However the designer may want to have different configuration procedures which could be, for example type-dependent. This explains why each call to CUTCELL requires the name of a configuration procedure.

2.4.3 Procedure STARTCUT

Recall that the cutting process starts in one of the four corners. The designer creates the initial packet and calls STARTCUT to commence the cutting. STARTCUT therefore requires the name of a configuration procedure, the address of its corresponding packet and the starting corner.

The first cell of the structure cannot have its boundary in-edges extrapolated by SADECUT since there is no surrounding cell from the out-edges of which, these in-edges could be derived. Thus, when calling STARTCUT, the user must explicitly list these boundary in-edges of the first cell!

The call **STARTCUT(LR,{3},ACTIVATION,T)** indicates that the starting cell is in the lower right corner of the RSA, that its link 3 is an in-edge, and that its configuration procedure is ACTIVATION with variable T as the parameter.

2.4.4 Procedure CUTREL

Recall that the role of a relayer is to carry some information towards a certain boundary. In SADE, all relayers that lie between an active cell and a boundary of the RSA can be specified by a unique call to CUTREL.

We first notice that only the in-edges of a relayer are needed to establish the complete configuration: to each in-edge corresponds an out-edge which is 180 $^{\circ}$ away. For example, if link 2 is an in-edge then link 5 is an out-edge. Secondly, we remark that we systematically use the same link to get from one relayer to the next one. Specifying the in-edges of the first relayer and the link it uses to get to the next one suffices to describe the complete chain of relayers.

The first relayer may be activated in two different ways:

1. In the configuration procedure, the current cell recognizes that it is a relayer, in which case it calls CUTREL indicating that the chain of relayers starts at the current cell.
2. The current cell recognizes that one of its neighbours will be a relayer, in which case the current cell calls CUTREL indicating that this neighbour is the first relayer.

Procedure CUTREL has three parameters. The first one gives the link use to get from one relayer to the next; the second one specifies the set of in-edges of these relayers; and finally the third one is a boolean value which indicates whether or not the current cell is a relayer.

For example, **CUTREL(2,{2,5},FALSE)** indicates that the neighbour, via link 2, of the current cell is the first relayer of a chain of relayers that goes to the right boundary (since link 2 is used to get from one relayer to the next one). Each of these relayers has links 2 and 5 as in-edges, and therefore, the same links as out-edges since 5 is the reciprocal of 2!

2.4.5 Function ONBOUNDARY

This is a boolean function that returns TRUE if the current cell is on the boundary specified as parameter; otherwise FALSE is returned.

2.4.6 Procedure ACTIVATE.

Sometimes the designer must activate a certain set of links without calling CUTCELL which effectively transfers the control to SADECUT. Procedure ACTIVATE serves this purpose. Its first argument is a set of links to activate, and the second one is a boolean variable which indicates whether or not the specified links are in-edges.

2.4.7 Procedure MarkState

This is a convenience routine that has an integer as its unique parameter. The procedure sets the cell-type of the current cell to this parameter.

2.4.8 Procedure DUMPCUT

Recall that SADECUT simulates the cutting process by a link list of events. It is possible to follow the cutting by calling the parameterless procedure DUMPCUT. This procedure simply displays the current activated links of the RSA.

2.4.9 Procedure SAVECUT

This is the only procedure of module SADEIO. It has one parameter which is the name of the file in which SADE is to archive the cut systolic structure. Obviously, the call to SAVECUT should be the last statement of the main program.

As for implementation, the procedure requires almost every file-handling built-in of MODULA-2. These routines are found in a huge library module called File which needs to be imported, consequently consuming a good part of the (very limited) symbol table space. To maintain the extensibility of SADE, we placed the archiving procedure in an individual module.

2.5 Examples

2.5.1 A Simple Example

Consider the linear array shown in Fig. 4.a where the first and the last cell are each connected to an I/O port from which inputs are received and outputs are sent to. A SADE program to cut this structure out of an RSA is given below; the result of the execution of the program on an RSA of size 6 is shown in Fig 4.b where only the first row of the RSA appears. (Relayers are denoted by 'r'.)

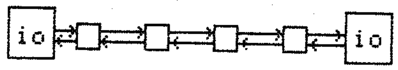

Figure 4.a A simple systolic array

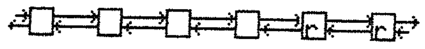

Figure 4.b Implementation of Fig 4.a

```
MODULE FOURCELL;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT ADDRESS;
FROM SADECUT IMPORT STARTCUT,CUTCELL,CUTREL,CORNERS;
FROM SADEIO  IMPORT SAVECUT;

TYPE
 PARMPTR = POINTER TO PARMREC;
 (* THIS IS THE RECORD USED TO HOLD THE INFORMATION USED BY ACTIVATION *)
 PARMREC = RECORD
             COUNT, NUMOFCELL: INTEGER;
            END;
VAR
 A: PARMPTR;
 INITIAL: CORNERS; (* CORNERS MUST BE IMPORTED FROM SADECUT *)

PROCEDURE ACTIVATION (T: ADDRESS);
(* This is the procedure that does most of the cutting job. It is recursively
   called via CUTCELL. Its parameter T is the system-defined ADDRESS type and
   gives the address of the record which contains the information used by the
   procedure. This is necessary because SADECUT does not 'see' the declaration
   of that record.  *)

VAR P: PARMPTR;
BEGIN
 P:= T;
 WITH P^ DO
  (* the algorithm is simple: if the current cell has a count greater than
     NUMOFCELL than we must cut a relayer in which case we use CUTREL.Otherwise
     we increment the cell count and call CUTCELL to cut the next cell which is
     reached via link 2 of the current cell.
     (Please check syntax and semantic of CUTCELL and CUTREL in PART II.)  *)
  IF COUNT> NUMOFCELL
    THEN CUTREL(2,{2,5},TRUE);
    ELSE COUNT:= COUNT+1;
         T:= P;
         CUTCELL({2,5},1,2,ACTIVATION,T);
  END; (* OF IF *)
 END (* OF WITH *)
END ACTIVATION;

BEGIN (* FOURCELL *)
 NEW(A);
 WITH A^ DO
   COUNT := 1; NUMOFCELL:= 4; (* we decide to cut four cells *)
 END;
 INITIAL:= UL;                 (* start in Upper Left corner  *)
 STARTCUT(INITIAL,{5},ACTIVATION,A);
 SAVECUT('FOUR.CUT');          (* save the structure in file FOUR.CUT *)
END FOURCELL.
```

Overview of the algorithm:

The cutting process starts in the Upper Left (UL) corner and the first cell has its link 5 directly activated as an in-edge since it cannot be extrapolated by subsequent activations. In the main program, once the cutting process is terminated, the designer saves the resulting structure in file FOUR.CUT using procedure SAVECUT.

The cutting process is initiated by STARTCUT which sets up the first call to the configuration procedure, called ACTIVATION. This procedure first translates the address of the packet (variable T) in a packet-pointer (variable P) and then checks the current cell count against the upper limit to see if the current cell is still part of the systolic structure. If the cell-count (field COUNT in the packet) is greater than the upper limit (field NUMOFCELL), the designer recognizes that the current cell is a relayer and calls CUTREL to activate the relayers, up to the right boundary of the RSA. The third parameter of CUTREL is set to TRUE to indicate that the current cell is a relayer.

If the current cell is not a relayer then the designer calls CUTCELL, activating links 2 and 5 as out-edges, giving a cell-type of 1 and selecting the cell to the right of the current one (link 2). At the next clock pulse, this selected neighbour will call the configuration procedure with an updated packet where the cell-count has been incremented by 1.

2.5.2 The General Linear Array

In the previous example, the entire structure was contained in one row of the RSA. In SADE, a general linear systolic array is cut according to the scheme shown in Fig. 5. A program which cuts the linear array of size 'NumOfCell' (shown in Fig. 6) is given below. Obviously, 'NumOfCell' should not exceed the maximal dimensions for an RSA.

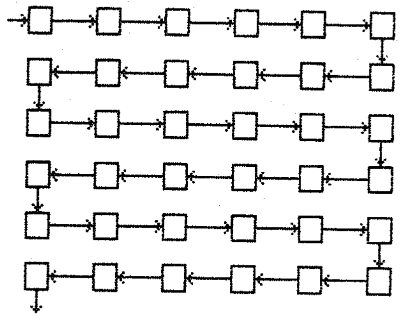

Figure 5. Linear Systolic Array Implementation Strategy

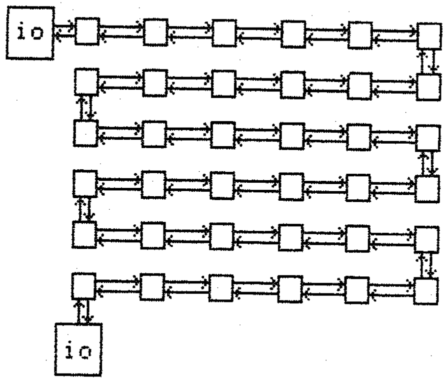

Figure 6. A Linear Systolic Array

```
MODULE LARRAY;
 ROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM InOut IMPORT WriteString, ReadInt;
FROM SYSTEM IMPORT ADDRESS;
FROM SADECUT IMPORT STARTCUT, CUTCELL, CUTREL, MarkState,
     ACTIVATE, ONBOUNDARY, DUMPCUT, BOUNDARIES, CORNERS;

TYPE
 PARMPTR = POINTER TO PARMREC;
 PARMREC = RECORD
            COUNT, NUMOFCELL: INTEGER;
            CURRENT : INTEGER;
            BOUNDARY: BOUNDARIES;
            DOWN,FIN : BOOLEAN;
           END;
VAR
 A: PARMPTR;
 INITIAL: CORNERS;

PROCEDURE ACTIVATION (T: ADDRESS);
VAR P: PARMPTR;
    NEIGHBOUR:[0..6]; (* gives the link to reach the next cell *)
    ALINKS:BITSET;    (* gives the set of links to activate for current cell*)
BEGIN
 P:= T;
 WITH P^ DO
  IF COUNT <= NUMOFCELL
    THEN  (* if we are dealing with one the cells of the array *)
     IF ONBOUNDARY(BOUNDARY)
       THEN (* if we are on a boundary then we must go down *)
            IF COUNT = NUMOFCELL
              (* if the last cell is on a boundary then we have the special
                 case where we do not need to cut relayers. We indicate this
                 by setting FIN to TRUE *)
              THEN FIN:= TRUE;
            END;
            NEIGHBOUR:= 4; (* neighbour is reached via link 4 i.e. go down *)
            DOWN      := TRUE;
            (* depending on which boundary the neighbour of the next cell will
               be either be on the left or the right. We must decide now and
               also reset the boundary variable which tells us when to go
               down. The variable CURRENT is used to tell the next cell where
               to find its neighbour when it is not on a boundary.  *)
            IF BOUNDARY=RB
              THEN ALINKS:= (4,5);
                   CURRENT:= 5;
                   BOUNDARY:= LB;
              ELSE ALINKS:= (2,4);
                   CURRENT:= 2;
                   BOUNDARY:= RB;
            END
    ELSE (* if current cell is not on boundary we use the variable CURRENT to
            tell us how to reach its neighbour. *)
```

```
              NEIGHBOUR:= CURRENT;
              (* if the current cell was reached by going down then one of its
                 neighbour is reached via link 1 (see figure) *)
              IF DOWN
                THEN IF BOUNDARY=LB (* IF LB WE WERE ON RB *)
                       THEN ALINKS:= {1,5};
                       ELSE ALINKS:= {1,2};
                     END
                ELSE ALINKS:= {2,5};(* normal set of links activated for a cell*)
               END;
               DOWN:= FALSE; (*reset the flag *)
       END;
       COUNT := COUNT+1;
       IF FIN
        THEN (* CASE WHERE WE DO NOT NEED RELAYERS: instead of using CUTCELL
                we call activate to define the in- and the out-edges of the
                current cell. MarkState is used to give a state.
                Please note that SADECUT extrapolates the in-edges of a cell
                only when using CUTCELL!
                                                                         *)
          ACTIVATE({2,5},TRUE); (* links 2 and 5 as in-edges *)
          ACTIVATE({2,5},FALSE);(* links 2 and 5 as out-edges *)
          MarkState(1);
        ELSE
          CUTCELL(ALINKS,1,NEIGHBOUR,ACTIVATION,P);
       END
      ELSE (* HANDLE RELAYERS *)
       (* NOTE: NO RELAYERS NEEDED IF WE WERE ON A BOUNDARY *)
       IF NOT FIN (*CHECK IF WE NEED RELAYERS *)
        THEN CUTREL(CURRENT,{2,5},TRUE);
       END
    END
  END (* OF WITH *)
 END ACTIVATION;

 BEGIN (* LARRAY *)
  NEW(A);
  WITH A^ DO
    COUNT := 1;
    WriteString(" How many cells in systolic array? ");
    ReadInt(NUMOFCELL);
    CURRENT:= 2;
    BOUNDARY:= RB; (*we will go down when we reach the right boundary *)
    DOWN:= FALSE;
    FIN := FALSE;
  END;
  INITIAL:= UL;
  STARTCUT(INITIAL,{5},ACTIVATION,A);
 DUMPCUT; (*to see the final cut...*)
 END LARRAY.
```

Overview of the algorithm:

After having read the desired number of cells in the array (variable NumOfCell), the designer specifies the current direction of the cutting and the boundary on which the array must go down one row. The cutting process is then initiated by STARTCUT. For each cell of the structure, ACTIVATION checks if we have reached a boundary, in which case we have to go down and change the direction of the cutting; the packet is updated and the appropriate set of links is activated. If the last cell of the structure is on a boundary, then obviously no relayer is needed. Otherwise, the relayers are activated by a call to CUTREL.

2.5.3 The Hexagonal Trellis

Another structure frequently used in the literature is the hexagonal trellis. The result of cutting an hexagonal trellis of size three is shown in Figure 7; the program follows. Notice that the diagonal activation of the relayers is arbitrary; other schemes are possible.

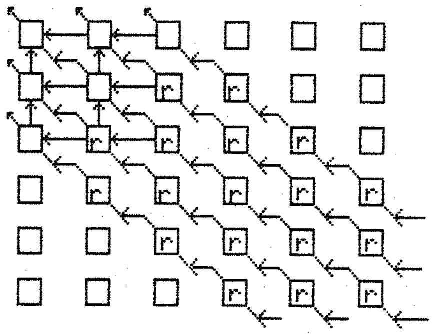

Figure 7. An hexagonal trellis of size 3

```
MODULE XTRELLIS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM IMPORT ADDRESS;
FROM SADECUT IMPORT STARTCUT, CUTCELL, CUTREL, MarkState,
     ACTIVATE, ONBOUNDARY, DUMPCUT, BOUNDARIES, CORNERS;

TYPE
 PARMPTR = POINTER TO PARMREC;
 PARMREC = RECORD
             I,J,IMAX,JMAX:INTEGER;
            END;
VAR
 A: PARMPTR;
 INITIAL: CORNERS;

PROCEDURE ACTIVATION (T: ADDRESS);
VAR P,PI,PJ,PK: PARMPTR; RELAYER:BOOLEAN;
BEGIN
 RELAYER:= FALSE;
 P:= T;
 PI:= NIL; PJ:=NIL; PK:= NIL;
 WITH P^ DO
  ACTIVATE({6},FALSE); (* activate link 6 for all cells *)
  (* please notice that the boundary cells have no links 1 and 5 *)
  IF NOT ONBOUNDARY(LB) THEN ACTIVATE({5},FALSE); END;
  IF NOT ONBOUNDARY(TB) THEN ACTIVATE({1},FALSE); END;
  (* if we are not at the limit of the current row we will activate the cell
     reached via link 2 *)
  IF I<IMAX
   THEN NEW(PI); PI^.I:= I+1; PI^.J:= J;
        PI^.IMAX:= IMAX;
        PI^.JMAX:= JMAX-1;
 END;
 (* if we are not at the limit of the current column we will activate the cell
    reached via link 4 *)
 IF J<JMAX
  THEN NEW(PJ); PJ^.I:= I; PJ^.J:= J+1;
       PJ^.IMAX:= IMAX-1;
       PJ^.JMAX:= JMAX;
 END;
 (* if the cell reached by link 3 still belong to the structure we will
    activate it using link 3 *)
 IF ((I+1)<IMAX) AND ((J+1)<JMAX)
  THEN NEW(PK); PK^.I:= I+1; PK^.J:= J+1;
       PK^.IMAX:= IMAX-1;
       PK^.JMAX:= JMAX-1;
 END;
```

```
    (* the next group of IF's could actually be done in parallel !
       Also note that CUTCELL has an empty st of links to activate since
       this algorithm uses ACTIVATE to select links  *)
    IF PI<>NIL THEN CUTCELL({},1,4,ACTIVATION,PI); END;
    IF PJ<>NIL THEN CUTCELL({},1,2,ACTIVATION,PJ); END;
    IF PK<>NIL THEN CUTCELL({},1,3,ACTIVATION,PK);
               ELSE (* the current cell has no neighbour within the structure
                       in which case we simply ask CUTREL to start activating
                       relayers using link 3. Please note that the last para-
                       meter of CUTREL is set to FALSE indicating that the
                       current cell is NOT to be taken as a relayer. The next
                       cell, reached by link 3 is the first relayer.     *)
                    RELAYER:= TRUE;
                    CUTREL(3,{3},FALSE);
   END;
   IF (I=IMAX) AND (J=JMAX) AND (NOT RELAYER)
    (* if we are on a diagonal and we haven't activated relayers so far we do
       it now. *)
    THEN CUTREL(3,{3},FALSE);
   END;
  END; (* OF WITH *)
 END ACTIVATION;

 BEGIN (* XTRELLIS *)
  NEW(A);
  WITH A^ DO
   I:=1; J:=1; IMAX:=3; JMAX:=3;
  END;
  INITIAL:= UL;
  STARTCUT(INITIAL,{},ACTIVATION,A);
 END XTRELLIS.
```

Overview of the algorithm:

In this example, instead of activating links using CUTCELL, the designer individually selects the link(s) of the current cell. A system of coordinates, kept in the packet, allows the configuration procedure to decide which neighbour(s) will still be part of the trellis. For each selected neighbour, the designer creates a new packet with the appropriate updated information and calls CUTCELL to cut, at the next clock pulse, this neighbour. It should be noted that the three consecutive calls to CUTCELL could actually be executed in parallel, which would truly duplicate the actual cutting process. Yet, by using a linked list of events, SADE is able to simulate this parallelism. Finally, the reader should notice that it is a current cell that recognizes if one or more of its neighbours are a relayer; thus the third parameter for the call to CUTREL is set to false. The relayers are activated using link 3 but could have as easily used links 2 and 4. A trace of the program's execution is included in Appendix 2 to demonstrate how SADE simulates the cutting process.

2.6 The Infinite Array Model

The preceding sections show how systolic structures can be cut in SADE. Other features are available to make the environment more versatile. This section presents one which is particularly useful: the infinite array model.

When using linear systolic arrays, it might be the case that the designer will want to define an algorithm for an infinite array before having to worry about the actual implementation onto an RSA. In other words, the designer may not want to immediately have to take into account the scheme presented in figure 4 to embed a linear array in an RSA. SADE allows the designer to change the underlying model from an RSA (a square of hexagonally-connected processors) to an 'infinite' linear array of processors. (The maximum size of this linear array is the square of the size of the largest possible RSA.) To request this linear model, the designer will call, in his main program, the parameterless procedure BOUNDLESS. All existing features of SADE perform identically on either model.

3. Systolic Algorithm Definition

3.1 Introduction

In SADE, the second step of the design process consists in the definition of one or more systolic algorithms that perform on a 'cut' systolic algorithms are expressed as a MODULA-2 program. In this paper, we will mainly restrict ourselves to the definition of a single systolic algorithm; pipelining will be briefly discussed in the last section of the current chapter.

Recall that the designer defines a systolic algorithm by specifying the algorithmic behaviour of each type of cell in the cut structure. (Cell-types are assigned in the systolic structure specification step.)

In SADE, the definition of a systolic algorithm follows a precise organization:

1. The designer must first describe the communication interface between the RSA and the outside world.

2. Each cell of the structure must partition its local memory: the programmable systolic cells (PSC) of the cut structure must allocate some part of their local memory to the variables used by the systolic algorithm.

3. Each cell-type of the cut structure must be mapped to a procedure that describes its algorithmic behaviour. This mapping is described in a procedure which we call the cell driver.

4. Finally, the designer must call SADE to start and control the execution process. (SADE will interactively request the name of the file that contains the cut systolic structure used for the execution.)

As with systolic structure specification, a brief discussion of some implementation details helps in understanding the programming techniques used to code systolic algorithms in SADE. In section 3.2 we study the internal cell model used to simulate the execution of systolic algorithms. The routines of module SADEALG (cf. section 2.4) are then described in section 3.3. A concrete example, a systolic algorithm for a priority queue, is presented in section 3.4. Lastly, in section 3.5 we discuss the pipelining capabilities of SADE.

3.2 The internal cell model

From the systolic structure specification step, each cell of the structure has been assigned:

(a) a set of in-edges.

(b) a set of out-edges.

(c) a cell-type (which is an integer).

To execute a systolic algorithm, every active cell in the systolic structure requires the following additional information:

(a) some pointer that gives access to the cell's local variables. These local (or state) variables are stored in a record and must preserve their values between clock pulses.

(b) the name of the cell driver which is used to select the procedure (henceforth called the <u>behavioural procedure</u>) that describes the cell's algorithmic behaviour: To each cell-type of the structure corresponds a unique behavioural procedure. The cell driver, which describes this mapping, merely consists in a CASE statement on the cell-type of the considered cell; each branch of the CASE calls the behavioural procedure corresponding to the selected cell-type. At each clock pulse, SADE goes through every cell of the structure and calls the cell driver of each cell to select and execute its behavioural procedure.

Both of the above items must be stored in the internal model of each active cell; this cell model consists of a record which holds all five items described in this section, as well as system-dependent information. To avoid the recompilation of SADEALG for each systolic algorithm, these five items must be of predefined types (cf. section 2.3). This imposes some constraints on the organization of the MODULA-2 program that implements an algorithm's definition:

As mentioned above, a step is required to allocate some (heap) memory for the local variables of each cell. Each active cell has, in its internal model, a field that gives the address of the record that contains its state variables. Thus the designer must code a function that allocates space for the different kinds of records of local variables, according to the cell-types of the structure. This function is organized as a CASE statement on the cell-type passed as parameter. Each branch of the CASE uses the built-in NEW to allocate space for the appropriate record. The function returns the address of the allocated record. At the start of the execution, SADE will refer to this allocation function to set up the local variables of each active cell of the RSA.

At each clock pulse, all active cells call their cell driver to select and execute 'simultaneously' the behavioural procedure associated with their cell-type. Each cell calls its cell driver, passing the address of its state variables as a parameter. By accessing the record pointed to by the parameter, the designer is able to refer to the identifiers he chose for the local variables. This is extremely helpful in that it ensures the internal model is completly invisible to the designer!

Finally, recall that SADE has some pipelining capabilities. Without going into the details of implementation we observe that, in our model, the pipelining of algorithms simply corresponds to the ability to change the cell driver of an active cell!! If algorithms are pipelined, then the first cells of the structure may use a new cell driver while other cells will still refer to the old driver. This justifies why each cell stores the name of its cell driver in its internal model.

It is possible to pass any procedure type as a parameter in MODULA-2, provided that the parameter list of the procedure is static [13]. Since the designer does not have access to SADE's internal model, he must call one of SADE's routines to store the name of a cell driver. This implies that all the cell drivers have the same parameter list which consists of one parameter of type ADDRESS (giving the address of the record of the state variables).

The next section presents the routines used to perform the different tasks studied above.

3.3 SADEALG's Routines

(Refer to the listing of the definition module for SADEALG, found in Appendix 3. for an exact description of the parameter list of each procedure)

3.3.1 Procedure INPUTON

The first task of the designer is to describe the communication interface between the RSA and the outside world. By default, the outputs generated by a systolic algorithm are routed to the the screen of the microcomputer. SADE displays the value of the output, the time at which it is generated and possibly the absolute coordinates of the cell that produces it. (The cell in the upper left corner has coordinates 1,1).

The inputs of a systolic algorithm may come from a file on a diskette or from the keyboard of the microcomputer. In both cases, the designer calls procedure INPUTON which has two parameters, the second of which gives the name of the input file. The first parameter of INPUTON is a link number. When SADE realizes that the input coming from a certain link lies outside of the RSA, it checks a table (of filenames according to links) set up by INPUTON, to decide where to get this input from. If the input is to be received via a link to which no filename has been associated, SADE prints an error message and stops the execution of the algorithm.

For example, the call **INPUTON(5,'CONSOLE:')** indicates that any exterior input which is to be received via a link 5, is to be obtained from the keyboard.

It is quite convenient to gather all the inputs of a systolic algorithm in a disk file. Recall that at each clock pulse, all active cells 'beat' (i.e. execute their behavioural procedure). In particular, this means that all boundary cells that read an input, will in fact read one at each clock pulse! Furthermore, systolic algorithms frequently use a skewed input format [7,10,11]. Therefore a convention for a null input is required. In SADE, a period (".") as input is taken as a null input. The period is translated to a predefined constant MAXINT so that the designer may test for a null input in his behavioural procedures.

Finally, another symbol, the vertical bar ("|"), is used as a stop input which, when received, indicates the end of a systolic algorithm. This convention provides a clean mechanism to halt the execution of a systolic algorithm.

3.3.2 Procedure SetLocals

Recall that the designer must code a function that allocates memory space according to the cell-type passed as parameter. Each active cell of the structure will call this function to setup its state variables. However, it is SADE and not the designer, that performs this allocation step (since the designer does not have access to the internal model of the RSA). Thus, the designer simply calls SetLocals to transfer the control to SADE. SetLocals has one parameter which denotes the name of the aforementioned allocation function.

3.3.3 Procedure SetAlg

This procedure is quite similar to SetLocals. It transfers the control to SADE, passing the name of a cell driver as its parameter. SADE then stores this name in the internal model of each active cell.

3.3.4 Function CELLTYPE

Recall that the cell driver is organized as a CASE statement on the different cell-types of the structure. At each clock pulse, every cell of the structure calls its driver. Instead of passing the cell-type as parameter, SADE provides the designer with the function CELLTYPE which returns the cell-type of the current cell. In this way, the CASE statement simply uses CELLTYPE as the case selector (cf. example of section 3.4).

3.3.5 Procedure Execute

Procedure EXECUTE is called to start the execution of the systolic algorithm. It has one parameter which is a boolean value that indicates whether or not SADE's pipelining option is used. If only one systolic algorithm is to be executed, then this variable is set to FALSE.

3.3.6 Procedure SEND

In SADE, two primitives are used to describe the communications between the activate cells of the structure: SEND and RECEIVE.

Procedure SEND has the following format:

```
SEND(link:INTEGER,desc:MSGTYPE,value:MSGVALUE);
```

The first parameter gives the link used to reach the addressee of the message. As we have seen in section 1.5, the messages transferred from one cell to another consist of a data type descriptor (INT,REL,CHA or STR) and a compatible message value. The two last parameters of SEND are used to pass these items to SADE: the second parameter is the data type descriptor; the third is the corresponding message value. (Notice that the designer must import SADEALG's MSGTYPE type for the second parameter of SEND)

For example, **SEND(2,INT,reg1)** sends a message denoting the integer stored in variable reg1, to the cell to the right neighbour of the current one.

3.3.7 Function RECEIVE

When receiving a message, a cell must store the message's value in one of its local variables. Furthermore, it is possible that a cell receives a message from a specific neighbour at time t, but does not, at time t+1. The format of RECEIVE accounts for these possibilities:

RECEIVE(link:integer;a:ADDRESS): BOOLEAN;

The first parameter gives the link from which the message is to be received. The second gives the address of the local variable in which the message should be stored. The designer simply uses the built-in function ADR to get this address. Finally, RECEIVE returns TRUE if a message was indeed received, FALSE otherwise. (The internal message model allows SADE to efficiently decide if there is a message on the specified link. Also, SADE will abort the execution of the systolic algorithm if a message sent is not received at the next clock pulse.)

For example, the call **RECEIVE(5,ADR(rega))** means that the current cell may have a message coming on its link 5. If this message is present, it is to be stored in the local variable called rega. RECEIVE returns TRUE if the message is properly received, FALSE otherwise. Notice it is the designer's responsibility to ensure that the message value and the local variable are of a compatible type.

3.3.8 Functions CLOCK and IDLE

Two SADE primitives take care of the synchronization aspects of the systolic algorithm:

(a) The CLOCK function denotes a virtual global clock which starts at time t=0 and is incremented at each pulse. CLOCK returns the current value of t.

(b) IDLE is a function that "turns a cell off" for the current clock pulse. In other words, it indicates to the system that the cell currently considered is not active for the current clock pulse.

IDLE strictly plays an esthetical role: the designer calls it to explicitly show, in a behavioural procedure, that in certain cases, the cell does not perform any operation!

Use of this two primitives is quite straightforward and is best illustrated by an example.

3.3.9 Other features

One fundamental characteristic of SADE is that it is easily extensible. Additional features, currently under development, include:

1. Provision for data transfers at different speeds and self-time computations. This implies a redefinition of SADE's timing mechanism.
2. Layouts by strata. Essentially this means overlapping several sets of links on the same cell. Each stratum has its own sets of in- and out-edges.

3. Combinations of cut structures [10,11]. SADE would require a mechanism to define the interface between the two structures (each stored on an individual RSA) and some geometrical tool that would specify the topology of the whole architecture.

3.4 An example: a priority queue

In this section, the above concepts are illustrated by means of a concrete example: a priority queue definition [11].

3.4.1 A simple priority queue

(For a complete discussion of the priority queue refer to C.E. Leiserson's thesis [11].)

Many programming applications require the ability to insert records into a set, and at any time to retrieve from the set the record having the smallest key according to some linear ordering. Any data structure that provides such services is called a **priority queue.**

Priority queues are usally implemented in software, but a priority queue can be built in hardware as a systolic array. One method uses identical processors, each of which is capable of sorting three elements. The three-sorter has three inputs X,Y, and Z and produces three outputs X',Y', and Z' which are the minimum, median and maximum of the inputs. Figure 8 shows how three-sorters are interconnected to make a systolic priority queue. In the figure, the outputs from the top, middle, and bottom of a processor are respectively the minimum, median, and maximum of the inputs. The minimum from each processor is fed leftward, and the median and maximum are fed rightward. This tends to keep smaller elements on the left and larger ones on the right. An infinite key which is larger than all other keys is provided as constant input on the right. The two inputs on the left are connected to the host computer. Initially, all elements in the queue have an infinite key. As elements are inserted on the left, they move rightward displacing infinite keys which are output on the right. If ever the outputs on the right are not infinite keys, the queue overflows.

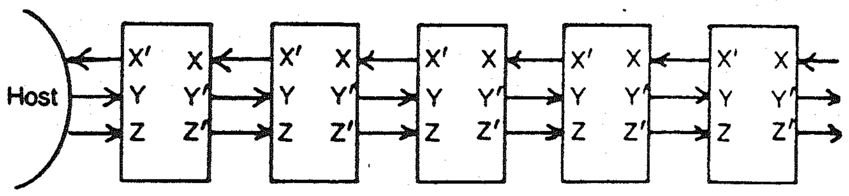

Figure 8. A Systolic Priority Queue.

Several steps of the operation of the systolic priority queue are illustrated in figure 9. The figure shows data on the wires between processors. The first column shows the communication between the host and the first processor on successive clock pulses, the second column shows the communication between the first and second processors, and so forth.

Notice that on even clock pulses, only odd-numbered processors do useful work, and that on even clock pulses, these cells are idle.

A SADE program to implement this priority queue and its operation is given in the following pages.

		−5	·	∞	·	∞
INSERT 6	6	·	−2	·	∞	·
	$-\infty$	·	4	·	∞	·
	$-\infty$	·	−2	·	∞	·
	·	−5	·	4	·	∞
	·	6	·	∞	·	∞
	·	−5	·	4	·	∞
EXTRACTMIN	∞	·	−2	·	∞	·
	∞	·	6	·	∞	·
	−5	·	−2	·	∞	·
(−5)	·	∞	·	4	·	∞
	·	∞	·	6	·	∞
	·	−2	·	4	·	∞
INSERT 3	3	·	∞	·	6	·
	$-\infty$	·	∞	·	∞	·
	$-\infty$	·	4	·	6	·
	·	−2	·	∞	·	∞
	·	3	·	∞	·	∞
	·	−2	·	6	·	∞
EXTRACTMIN	∞	·	3	·	∞	·
	∞	·	4	·	∞	·
	−2	·	3	·	∞	·
(−2)	·	∞	·	4	·	∞
	·	∞	·	6	·	∞

Figure 9. Execution of the priority queue

```
MODULE QUEUE;
FROM SADEALG IMPORT INPUTON,IDLE,CLOCK,SEND,RECEIVE,
                    CELLTYPE,MSGTYPE,SetLocals,SetAlg,EXECUTE;
FROM SYSTEM  IMPORT ADR,ADDRESS;
FROM Storage IMPORT ALLOCATE;

CONST
 MAXKEY= 800; (* A KEY GREATER THAN ANY ALLOWED KEY IN THE QUEUE *)
TYPE
 (* THE FOLLOWING DECLARATIONS ARE USED TO DESCRIBE THE LOCAL MEMORY
    ORGANIZATION OF THE CELLS OF THE STRUCTURE. THIS ALLOWS FOR USER-
    DEFINED NAMES IN THE ALGORITHM BELOW.                              *)
 LOCPTR1= POINTER TO LOCALS;
 LOCALS = RECORD
           MED,MAXI,MINI,TEMP: INTEGER;
          END;

PROCEDURE ALGORITHM(T:ADDRESS);
VAR P1:LOCPTR1;

PROCEDURE SORT(Q:LOCPTR1);
(* This procedure uses SEND and RECEIVE to implement the priority queue
   algorithm described in C.E. Leiserson's thesis [11]. We first receive the
   possible new minimum from the right neighbour. RECEIVE will return
   FALSE is there is no new minimum. Then we receive from our left neighbour
   the two other keys and we sort them. Finally we use SEND to transmit the
   results to our neighbours (see figure 13). Please check the thesis for
   the syntax and semantic of the procedures.                        *)
BEGIN
WITH Q^ DO
 IF NOT RECEIVE(2,ADR(MINI))
   THEN MINI:= MAXKEY;
 END;
 IF RECEIVE(5,ADR(MED)) AND RECEIVE(5,ADR(MAXI))
  THEN
   IF MINI > MED
    THEN TEMP:= MINI; MINI:= MED; MED:= TEMP;
   END;
   IF MINI > MAXI
     THEN TEMP:= MINI; MINI:= MAXI; MAXI:= TEMP;
   END;
   IF MED > MAXI
     THEN TEMP:= MED; MED:= MAXI; MAXI:= TEMP;
   END;
```

```
    SEND(2,INT,MED);SEND(2,INT,MAXI);SEND(5,INT,MINI);
   END
  END
 END SORT;

 PROCEDURE CELLA; (*BEATS ON EVEN PULSES *)
 BEGIN
  IF (CLOCK() MOD 2) = 0
    THEN SORT(P1)
  END
 END CELLA;

 PROCEDURE CELLB; (*BEATS ON ODD PULSES *)
 BEGIN
  IF (CLOCK() MOD 2)<>0
    THEN SORT(P1)
  END
 END CELLB;

 BEGIN (*OF ALGORITHM*)
  (* this is the typical organization of a systolic algorithm using SADE.
     The user simply makes a CASE statement on the different cell types
     contained in the systolic structure. In this example both cell types
     use the same memory layout but we repeated the assigment of T to a
     pointer to illustrate how T would just have to be assigned to the
     different memory pointers.                                *)
  CASE CELLTYPE() OF
   15: P1:= T; CELLA; |
   14: P1:= T; CELLB
  END;
 END ALGORITHM;

PROCEDURE MEMORY(STATE:INTEGER): ADDRESS;
(* this procedure allocates memory to a cell of the structure according
   to its cell type which is passed as a parameter          *)
VAR P1: LOCPTR1;
BEGIN
 CASE STATE OF
  14: NEW(P1); RETURN P1; |
  15: NEW(P1); RETURN P1
 END;
END MEMORY;

BEGIN (* OF QUEUE *)
 (* the input on the link 5 of the leftmost cell will be read from the
    keyboard of our Apple... File i/o is possible when the user supplies
    a valid filename as parameter to INPUTON              *)
 INPUTON(5,'CONSOLE:');
 SetLocals(MEMORY); (* allocate local memory to all cells in the structure *)
 (* SADE allows for pinelining of algorithms, that is a cell may change its
    algorithm from one set of inputs to the other! The call to SetAlg tells
    the system which procedure holds the first algorithm. Then EXECUTE is
    called to start the execution. Its parameter indicates the pipelining
    option is used or not. Please refer to the thesis for a full discussion *)
 SetAlg(ALGORITHM);
 EXECUTE(FALSE);
END QUEUE.
```

3.4.2 Overview of the algorithm

The comments in the program provide a detailed description of the algorithm. Therefore, this subsection will only consists of a few brief remarks:

1. In this program, procedure ALGORITHM is the **cell driver** and CELLA and CELLB act as the behavioural procedures. MEMORY is the memory-allocation function; in this example we use the same kind of record for the local variables of both even- and odd-numbered cells (which respectively have cell-types 14 and 15) therefore only one pointer (P1) is required.

2. The designer could had explicitly shown that cells of type 15 are idle on odd clock pulses by adding an 'ELSE IDLE' to the IF statement of procedure CELLA.

3. The constant MAXKEY represents the infinite key in the algorithm.

3.5 Pipelining

One of the fundamental ideas of the programmable systolic architecture is that it should be possible to:

1. execute a systolic algorithm with several different sets of input data.

2. execute several different systolic algorithms on a same systolic structure.

3. execute different systolic algorithms on different structures cut out of a same RSA.

This section explains how SADE can achieve this.

3.5.1 Pipelining of Inputs

In SADE, two different approaches can be used to execute several different sets of inputs on a same systolic algorithm. The first method consists in simply placing these sets of inputs one after the other in the input file! It is possible that the sets would have to be separated by some null inputs to respect the period of the algorithm. In the second approach, at the end of each set of inputs, the designer simply redefines the input file(s) by calling INPUTON. Both methods are straightforward and have been tested with the priority queue example.

3.5.2 Pipelining Systolic Algorithms for A Same Structure

As mentioned earlier, it is possible to execute several systolic algorithms, one after the other, on a same cut structure. Recall that it suffices to update the internal cell driver field of each cell to do this. To indicate the pipelining of algorithms, the designer must first call EXECUTE passing TRUE as parameter. This tells SADE that as soon as a cell receives a stop signal, it should update its cell driver to the next one found in a list of systolic algorithms; this list is interactively created when SADE processes the call to EXECUTE(TRUE).

3.5.3 Different Algorithms on Different Structures

Executing different algorithms on different systolic structures is the most complex pipelining technique possible with an RSA. It takes full advantage of the fact that the processors of the RSA are programmable, and thus that the RSA is reconfigurable. In SADE, this technique requires the ability to alternate the execution of systolic algorithms with that of 'cutting' algorithms: the designer cuts a first structure and executes a few algorithms on it; then he executes a second 'cutting' algorithm to modify the configuration of the RSA and directly uses this new structure to execute some new algorithms, and so forth. Alternating between the cutting and the simulation processes is possible since cutting algorithms are systolic algorithms.

In SADE, to achieve this kind of pipelining, the designer must:

1. code all the cutting algorithms in separate modules.
2. code all systolic algorithms that use a same structure in a unique module.
3. write a **command file** which consists of pairs of names: The first name of a pair refers to the module containing a cutting algorithm; the second indicates the module that contains the systolic algorithms to be executed on the corresponding structure.

The command file is a simple representation of the desired pipelining process; SADE includes a program that can interpret this command file and simulate it.

4. Conclusions

4.1 Implementaton of SADE

The current prototype of SADE is implemented as a set of MODULA-2 library modules. This 'new' high-level language was chosen for several reasons:

1. a complete implementation of the language exists on several microcomputers.
2. the language allows for separate compilation, strong-type checking and the passing of any type of procedure as a parameter.
3. modular design and coding makes SADE a tool which is very easily extensible.
4. MODULA-2, under the USCD PASCAL system offers important advantages:
   (a) SADE can use the TURTLEGRAPHICS package to implement the graphical cutting of systolic structures, as well as the graphical simulation of systolic algorithms.
   (b) This entire MODULA-2 system is highly portable: the whole tool can be ported to another system provided that a P-code interpreter is available.

SADE's procedural mode can also be quite easily implemented in LISP and ADA! However the resulting programs do not offer MODULA-2 elegant organization and can't refer to the TURTLEGRAPHICS package which implements SADE's graphical mode.

4.2 Graphical Definition Mode

As mentioned earlier, SADE allows the systolic structure to be specified via an interactive graphical package. In this mode, the specification process proceeds as follows:

1. an RSA of user-defined size is displayed
2. Via a light-pen, the user cuts the desired structure by associating types with the active cells
3. for each cell-type, a basic cell (see Fig. 2) is displayed and the user, via the light-pen, activates the links as desired.

The system will automatically determine the entire topology as well as the relayers. The algorithm definition step is still carried out in a procedural mode. The execution of a systolic algorithm can be displayed on the screen if so desired.

4.3 Testing Facilities

SADE's internal model contains enough information in its data structures so that, when simulating a systolic algorithm, the designer can gather valuable testing results. Moreover, SADE will automatically measure complexity metrics such as: period, area and time.

4.4 Conclusions

In this paper, we have presented the specifications for a tool that allows for the design and testing of systolic algorithms on a programmable systolic architecture (RSA). In particular, we have illustrated how the designer can specify a systolic structure within an RSA and define several pipelined systolic algorithms to perform on this structure. Furthermore, we have briefly studied how an RSA could be reconfigured so that different algorithms for different structures could be executed. The expression of systolic algorithms as high-level programs suggests that a standardization of design techniques for systolic algorithms is easily attainable. Finally, our tool seems to offer an interesting alternative to existing ad hoc simulation tools.

As a design environment, SADE is to be viewed as a component of a much larger VLSI CAD system as shown in Fig. 10. Its place within such a system would be as a front-end entity, i.e. between the user and a semantic analyzer that would extract the layout requirements from SADE's internal model of the RSA.

User -- SADE -- Semantic Analyzer -- Layout Package -- Fabrication

Figure 10. A VLSI CAD System

We are currently studying the interface between SADE and ALI [12]. As a matter of fact, combining ALI's powerful abstraction mechanisms with SADE's explicit description of the semantics of the algorithm could fill in the gap between theoritical design and realistic VLSI implementations.

References

[1] R.P. Brent, H.T. Kung and F.T. Luk:Some Linear-Time Algorithms for Systolic Arrays, *TR 83-541 Dept. of Computer Science Cornell University* ,1983

[2] B. Chazelle and L. Monier: A Model of Computation for VLSI with Related Complexity Results, *Proc. of the Symposium on Theory of Computing*, 1981, pp. 318-325.

[3]J.-P. Corriveau and Nicola Santoro: Area-Efficient Embedding of Binary Trees,*Technical Report SCS-TR-*59, School of Computer Science, Carleton University, 1983

[4] A. Fisher, H.T. Kung, L. Monier and Y. Dohi: Architecture of the PSC: A Programmable Systolic Chip, *Proc. of the 10th Int. Symp. on Computer Architecture*,1983, pp. 48-53.

[5] D. Gordon, I. Koren and G. Silberman: Embedding Tree Structures in VLSI hexagonal Arrays, *IEEE Trans. on Computers*, Vol. C-31, September 1982, pp. 892-897

[6] M. Kramer and J. van Leeuwen: Systolic Computation and VLSI, *Part 1: Foundations of Computer Science IV*, Mathematical Centre Tracts 158, Amsterdam, 1983, pp. 75-103

[7] H.T. Kung: Why VLSI Architectures, *IEEE Computer 15*, 1982, pp. 37-45

[8] H.T. Kung: On the Implementation and Use of Systolic Array

Processors, *Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers*, 1983, pp. 370-373

[9] **H.T. Kung and S.Q. Yu: Integrating High-Performance Special-Purpose Devices into a System,** *VLSI Architecture*, **Randel and Treleaven eds, Prentice/Hall International, 1983, pp. 205-211**

[10] **C. Mead and L. Conway:** *Introduction to VLSI Systems*, **Addison-Wesley, Reading, MA, 1980**

[11] **C. Leiserson:** *Area-efficient VLSI Computation*, **MIT Press, Cambridge, MA, 1983**

[12] **R. Lipton, R. Sedgewick and J. Valdes: Programming Aspects of VLSI,** *CACM, January 1982*, **pp. 57-65.**

[13] **N. Wirth:** *Programming In MODULA-2*, **Springer-Verlag, Berlin Heidelberg New York, 1983**

Appendix 1

```
DEFINITION MODULE SADECUT;
(* $SEG := 43; *)
FROM SYSTEM IMPORT ADDRESS;
EXPORT QUALIFIED STARTCUT,CUTCELL,DUMPCUT,CORNERS,CUTREL,ACTIVATE,ONBOUNDARY,
        IMAX,JMAX,UC,CELLULE,STOPCUT,RECPR,MarkState,BOUNDARIES;


TYPE
 CORNERS= (UL,UR,LL,LR);
 LINKS   = [1..6];
 BOUNDARIES= (TB,LB,RB,BB);
 EDGES   = ARRAY [1..6] OF BOOLEAN;

 PROCTYPE = PROCEDURE (ADDRESS);
 FILENAME = ARRAY [0..7] OF CHAR;
 CALLPTR = POINTER TO CALLREC;
 CALLREC= RECORD
            CURRI,CURRJ: INTEGER;
            CURPROC    : PROCTYPE;
            CURPARM    : ADDRESS;
            NEXT       : CALLPTR;
           END;

 CELLULE = RECORD
             INE,OUTE: EDGES;
             STATE   : INTEGER;
            END;
  K2 = RECORD
        A,B:INTEGER; I:EDGES;
       END;
  K1 = RECORD
        A:BITSET; S:INTEGER; L:LINKS;
        P:PROCTYPE; T:ADDRESS;
       END;
 EVENTS= POINTER TO ACTION;
 ACTION = RECORD
            RELAYER:BOOLEAN;
            NEXT    :EVENTS;
            TEMPS   :INTEGER;
            OLDI,OLDJ:INTEGER;
            CASE INTEGER OF
             0: PARM1: K1 |
             1: PARM2: K2
            END;
           END;
 CHIP     = ARRAY [1..8], [1..8] OF CELLULE;

VAR
 SYSI,SYSJ,SYSTIME: INTEGER; SYSEND:BOOLEAN;
 STARTCALL: CALLPTR;
 IMAX,JMAX:INTEGER;UC : CHIP;

PROCEDURE ONBOUNDARY(B:BOUNDARIES): BOOLEAN;
```

```
     Size of PSA?
4

     Time 0

     Cell number 1 1
     state: 14   in-edges           out-edges 6

     Time 1
     Cell number 1 1
     state: 14   in-edges 2 3 4     out-edges 6
     Cell number 1 2
     state: 14   in-edges 4         out-edges 5 6
     Cell number 2 1
     state: 14   in-edges 2         out-edges 1 6
     Cell number 2 2
     state: 14   in-edges           out-edges  1 5 6

     Time 2
     Cell number 1 1
     state: 14   in-edges 2 3 4     out-edges 6
     Cell number 1 2
     state: 14   in-edges 2 3 4     out-edges 5 6
     Cell number 1 3
     state: 14   in-edges           out-edges 5 6
     Cell number 2 1
     state: 14   in-edges 2 3 4     out-edges 1 6
     Cell number 2 2
     state: 14   in-edges 3         out-edges 1 5 6
     Cell number 2 3
     state: 0    in-edges 3         out-edges 6
     Cell number 3 1
     state: 14   in-edges           out-edges 1 6
     Cell number 3 2
     state: 0    in-edges 3         out-edges 6
     Cell number 3 3
     state: 0    in-edges 3         out-edges 6

     Time 3
```

Appendix 3

```
(* $RECYCLE:= TRUE; *)
DEFINITION MODULE SADEALG;
(* $SEG := 44; *)
FROM SADEIO IMPORT GETCUT;
FROM SYSTEM IMPORT ADDRESS;
FROM Files IMPORT FILE;
EXPORT QUALIFIED SetAlg,SetLocals,RECEIVE,SEND,BROADCAST,
                 MSGTYPE,MSGVALUE,CELLTYPE,IDLE,CLOCK,PIPELINE,INPUTON,EXECUTE;
TYPE
 LINKS  = [1..6];
 BOUNDARIES= (TB,LB,RB,BB);
 EDGES  = ARRAY [1..6] OF BOOLEAN;

 PROCTYPE = PROCEDURE (ADDRESS);
 PROCTYPE1 = PROCEDURE(INTEGER): ADDRESS;
 FILENAME = ARRAY [0..7] OF CHAR;

 MSGPTR = POINTER TO MESSAGE;
 MSGVALUE= INTEGER;
 MSGTYPE = (INT,REL,ARR,REC,CHA);
 OUTREC = RECORD
           I,J,T:INTEGER; V:MSGVALUE;
          END;
 MESSAGE= RECORD
           KIND: MSGTYPE; LINK:LINKS;
           VALUE: MSGVALUE; NEXT:MSGPTR;
          END;
 ALGPTR = POINTER TO ALGREC;
 ALGREC = RECORD
           PROC:PROCTYPE; NEXT:ALGPTR;
          END;
 CELLULE = RECORD
            INE,OUTE: EDGES;
            STATE   : INTEGER;
            LOCAL   : ADDRESS;
            CELLALG : ALGPTR;    CURMSG,NEWMSG:MSGPTR;
           END;
 CHIP    = ARRAY [1..8], [1..8] OF CELLULE;

VAR
 STOPVALUE,SYSI,SYSJ,SYSTIME,SYSSTOP: INTEGER;
 IMAX,JMAX:INTEGER;
 UC : POINTER TO CHIP;
 IOTABLE: ARRAY [1..6] OF FILE; INFILE,OUTFILE:FILE;
 NOINPUT,FULLOUT : BOOLEAN;
 INPUTFLAG: ARRAY [1..6] OF BOOLEAN;

PROCEDURE SetAlg(P:PROCTYPE);

PROCEDURE CELLTYPE():INTEGER;

PROCEDURE PIPELINE(P:ALGPTR);

PROCEDURE BROADCAST(M:MESSAGE);

PROCEDURE RECEIVE(L:LINKS; T:ADDRESS):BOOLEAN;

PROCEDURE CLOCK ():INTEGER;

PROCEDURE IDLE;

PROCEDURE SEND(L:LINKS; K:MSGTYPE; V:MSGVALUE);

PROCEDURE SetLocals(PR1:PROCTYPE1);

PROCEDURE INPUTON(L:LINKS; NAME:FILENAME);

PROCEDURE EXECUTE(P: BOOLEAN);
END SADEALG.
```

```
Cell number 1 1
state: 14   in-edges 2 3 4   out-edges 6
Cell number 1 2
state: 14   in-edges 2 3 4   out-edges 5 6
Cell number 1 3
state: 14   in-edges 5       out-edges 5 6
Cell number 2 1
state: 14   in-edges 2 3 4   out-edges 1 6
Cell number 2 2
state: 14   in-edges 3       out-edges 1 5 6
Cell number 2 3
state: 0    in-edges 3       out-edges 6
Cell number 3 1
state: 14   in-edges 3       out-edges 1 6
Cell number 3 2
state: 0    in-edges 3       out-edges 6
Cell number 3 3
state: 0    in-edges 3       out-edges 6
Cell number 4 2
state: 0    in-edges 3       out-edges 6
Cell number 4 3
state: 0    in-edges 3       out-edges 6
Cell number 4 4
state: 0    in-edges 3       out-edges 6
```

```
PROCEDURE MarkState (I:INTEGER);

PROCEDURE RECPR(I:INTEGER): INTEGER;

PROCEDURE ERROR(I:INTEGER);

PROCEDURE PRINTLINKS(E: EDGES);

PROCEDURE DUMPCUT;

PROCEDURE ACTIVATE(A:BITSET; INEDGE:BOOLEAN);

PROCEDURE STARTCUT(C:CORNERS; A:BITSET; P:PROCTYPE; T:ADDRESS);

PROCEDURE CUTREL(L:LINKS; INR:BITSET;IMMEDIATE:BOOLEAN);

PROCEDURE CUTCELL(A:BITSET; S:INTEGER; L:LINKS; P:PROCTYPE; T:ADDRESS);

PROCEDURE STOPCUT;

END SADECUT.
```

CARLETON UNIVERSITY

School of Computer Science

BIBLIOGRAPHY OF SCS TECHNICAL REPORTS

SCS-TR-1	**THE DESIGN OF CP-6 PASCAL** Jim des Rivieres and Wilf R. LaLonde, June 1982.
SCS-TR-2	**SINGLE PRODUCTION ELIMINATION IN LR(1) PARSERS: A SYNTHESIS** Wilf R. LaLonde, June 1982.
SCS-TR-3	**A FLEXIBLE COMPILER STRUCTURE THAT ALLOWS DYNAMIC PHASE ORDERING** Wilf R. LaLonde and Jim des Rivieres, June 1982.
SCS-TR-4	**A PRACTICAL LONGEST COMMON SUBSEQUENCE ALGORITHM FOR TEXT COLLATION** Jim des Rivieres, June 1982.
SCS-TR-5	**A SCHOOL BUS ROUTING AND SCHEDULING PROBLEM** Wolfgang Lindenberg, Frantisek Fiala, July 1982.
SCS-TR-6	**ROUTING WITHOUT ROUTING TABLES** Nicola Santoro, Ramez Khatib, July 1982.
SCS-TR-7	**CONCURRENCY CONTROL IN LARGE COMPUTER NETWORKS** Nicola Santoro, Hasan Ural, July 1982.
SCS-TR-8 Out of Print	**ORDER STATISTICS ON DISTRIBUTED SETS** Nicola Santoro, Jeffrey B. Sidney, July 1982.
SCS-TR-9	**OLIGARCHICAL CONTROL OF DISTRIBUTED PROCESSING SYSTEMS** Moshe Krieger, Nicola Santoro, August 1982.
SCS-TR-10	**COMMUNICATION BOUNDS FOR SELECTION IN DISTRIBUTED SETS** Nicola Santoro, Jeffrey B. Sidney, September 1982.
SCS-TR-11	**SIMPLE TECHNIQUE FOR CONVERTING FROM A PASCAL SHOP TO A C SHOP** Wilf R. LaLonde, John R. Pugh, November 1982.
SCS-TR-12	**EFFICIENT ABSTRACT IMPLEMENTATIONS FOR RELATIONAL DATA STRUCTURES** Nicola Santoro, December 1982.
SCS-TR-13	**ON THE MESSAGE COMPLEXITY OF DISTRIBUTED PROBLEMS** Nicola Santoro, December 1982.

SCS-TR-14	**A COMMON BASIS FOR SIMILARITY MEASURES INVOLVING TWO STRINGS** R. L. Kashyap and B. J. Oommen, January 1983.
SCS-TR-15	**SIMILARITY MEASURES FOR SETS OF STRINGS** R. L. Kashyap and B. J. Oommen, January 1983.
SCS-TR-16	**THE NOISY SUBSTRING MATCHING PROBLEM** R. L. Kashyap and B. J. Oommen, January 1983.
SCS-TR-17	**DISTRIBUTED ELECTION IN A CIRCLE WITHOUT A GLOBAL SENSE OF ORIENTATION** E. Korach, D. Rotem, N. Santoro, January 1983.
SCS-TR-18	**A GEOMETRICAL APPROACH TO POLYGONAL DISSIMILARITY AND THE CLASSIFICATION OF CLOSED BOUNDARIES** R. L. Kashyap and B. J. Oommen, January 1983.
SCS-TR-19	**SCALE PRESERVING SMOOTHING OF POLYGONS** R. L. Kashyap and B. J. Oommen, January 1983.
SCS-TR-20	**NOT-QUITE-LINEAR RANDOM ACCESS MEMORIES** Jim des Rivieres, Wilf LaLonde and Mike Dixon, August 1982, Revised March 1, 1983.
SCS-TR-21	**SHOUT ECHO SELECTION IN DISTRIBUTED FILES** D. Rotem, N. Santoro, J. B. Sidney, March 1983.
SCS-TR-22	**DISTRIBUTED RANKING** E. Korach, D. Rotem, N. Santoro, March 1983.
SCS-TR-23	**A REDUCTION TECHNIQUE FOR SELECTION IN DISTRIBUTED FILES : I** N. Santoro, J. B. Sidney, April 1983.
Replaced by SCS-TR-69	
SCS-TR-24	**LEARNING AUTOMATA POSSESSING ERGODICITY OF THE MEAN : THE TWO ACTION CASE** M. A. L. Thathachar and B. J. Oommen, May 1983.
SCS-TR-25	**ACTORS - THE STAGE IS SET** John R. Pugh, June 1983.
SCS-TR-26	**ON THE ESSENTIAL EQUIVALENCE OF TWO FAMILIES OF LEARNING AUTOMATA** M. A. L. Thathachar and B. J. Oommen, May 1983.
SCS-TR-27	**GENERALIZED KRYLOV AUTOMATA AND THEIR APPLICABILITY TO LEARNING IN NONSTATIONARY ENVIROMENTS** B. J. Oommen, June 1983.
SCS-TR-28	**ACTOR SYSTEMS: SELECTED FEATURES** Wilf R. LaLonde, July 1983.

SCS-TR-29	**ANOTHER ADDENDUM TO KRONECKER'S THEORY OF PENCILS** M. D. Atkinson, August 1983.
SCS-TR-30	**SOME TECHNIQUES FOR GROUP CHARACTER REDUCTION** M. D. Atkinson and R. A. Hassan, August 1983.
SCS-TR-31	**AN OPTIMAL ALGORITHM FOR GEOMETRICAL CONGRUENCE** M. D. Atkinson, August 1983.
SCS-TR-32	**MULTI-ACTION LEARNING AUTOMATA POSSESSING ERGODICITY OF THE MEAN** B. J. Oommen and M. A. L. Thathachar, August 1983.
SCS-TR-33	**FIBONACCI GRAPHS, CYCLIC PERMUTATIONS AND EXTREMAL POINTS** N. Santoro and J. Urrutia, December 1983
SCS-TR-34	**DISTRIBUTED SORTING** D. Rotem, N. Santoro, and J. B. Sidney, December 1983
SCS-TR-35	**A REDUCTION TECHNIQUE FOR SELECTION IN DISTRIBUTED FILES: II** N. Santoro, M. Scheutzow, and J. B. Sidney, December 1983
SCS-TR-36	**THE ASYMPTOTIC OPTIMALITY OF DISCRETIZED LINEAR REWARD-INACTION LEARNING AUTOMATA** B. J. Oommen and Eldon Hansen, January 1984
SCS-TR-37	**GEOMETRIC CONTAINMENT IS NOT REDUCIBLE TO PARETO DOMINANCE** N. Santoro, J. B. Sidney, S. J. Sidney, and J. Urrutia, January 1984
SCS-TR-38	**AN IMPROVED AGLORITHM FOR BOOLEAN MATRIX MULTIPLICATION** N. Santoro and J. Urrutia, January 1984
SCS-TR-39	**CONTAINMENT OF ELEMENTARY GEOMETRIC OBJECTS** J. Sack, N. Santoro and J. Urrutia, February 1984
SCS-TR-40	**SADE: A PROGRAMMING ENVIRONMENT FOR DESIGNING AND TESTING SYSTOLIC ALGORITHMS** J. P. Corriveau and N. Santoro, February 1984
SCS-TR-41	**INTERSECTION GRAPHS, {B }-ORIENTABLE GRAPHS AND PROPER CIRCULAR ARC GRAPHS** Jorge Urrutia, February 1984
SCS-TR-42	**MINIMUM DECOMPOSITIONS OF POLYGONAL OBJECTS** J. Mark Keil and Jorg-R. Sack, March 1984
SCS-TR-43	**AN ALGORITHM FOR MERGING HEAPS** Jorg-R. Sack and Thomas Strothotte, March 1984

SCS-TR-44 **A DIGITAL HASHING SCHEME FOR DYNAMIC MULTIATTRIBUTE FILES**
E. J. Otoo, March 1984

SCS-TR-45 **SYMMETRIC INDEX MAINTENANCE USING MULTIDIMENSIONAL LINEAR HASHING**
E. J. Otoo, March 1984

SCS-TR-46 **A MAPPING FUNCTION FOR THE DIRECTORY OF A MULTIDIMENSIONAL EXTENDIBLE HASHING**
E. J. Otoo, March 1984

SCS-TR-47 **TRANSLATING POLYGONS IN THE PLANE**
Jorg-R. Sack, March 1984

SCS-TR-48 **CONSTRAINED STRING EDITING**
J. Oommen, May 1984

SCS-TR-49 **O(N) ELECTION ALGORITHMS IN COMPLETE NETWORKS WITH GLOBAL SENSE OF ORIENTATION**
Jorg Sack, Nicola Santoro, Jorge Urrutia, May 1984

SCS-TR-50 **THE DESIGN OF A PROGRAM EDITOR BASED ON CONSTRAINTS**
Christopher A. Carter and Wilf R. LaLonde, May 1984

SCS-TR-51 **DISCRETIZED LINEAR INACTION-PENALTY LEARNING AUTOMATA**
B. J. Oommen and Eldon Hansen, May 1984

SCS-TR-52 **SENSE OF DIRECTION, TOPOLOGICAL AWARENESS AND COMMUNICATION COMPLEXITY**
Nicola Santoro, May 1984

SCS-TR-53 **OPTIMAL LIST ORGANIZING STRATEGY WHICH USES STOCHASTIC MOVE-TO-FRONT OPERATIONS**
B. J. Oommen, June 1984

SCS-TR-54 **RECTINLINEAR COMPUTATIONAL GEOMETRY**
J. Sack, June 1984

SCS-TR-55 **AN EFFICIENT, IMPLICIT DOUBLE-ENDED PRIORITY QUEUE**
M. D. Atkinson, Jorg Sack, Nicola Santoro, T. Strothotte, July 1984

SCS-TR-56 **DYNAMIC MULTIPAGING: A MULTIDIMENSIONAL STRUCTURE FOR FAST ASSOCIATIVE SEARCHING**
E. Otoo, T. H. Merrett

SCS-TR-57 **SPECIALIZATION, GENERALIZATION AND INHERITANCE**
Wilf R. LaLonde, John R. Pugh, August 1984

SCS-TR-58 **COMPUTER ACCESS METHODS FOR EXTENDIBLE ARRAYS OF VARYING DIMENSIONS**
E. Otoo, August 1984.

SCS-TR-59 **AREA-EFFICIENT EMBEDDINGS OF TREES**
J. P. Corriveau, Nicola Santoro, August 1984.

SCS-TR-60 **UNIQUELY COLOURABLE m-DICHROMATIC ORIENTED GRAPHS**
V. Neumann-Lara, N. Santoro, J. Urrutia, August 1984.

SCS-TR-61 **ANALYSIS OF A DISTRIBUTED ALGORITHMS FOR EXTREMA FINDING IN A RING**
D. Rotem, E. Korach and N. Santoro, August 1984.

SCS-TR-62 **ON ZIGZAG PERMUTATIONS AND COMPARISONS OF ADJACENT ELEMENTS**
M. D. Atkinson, October 1984

SCS-TR-63 **SETS OF INTEGERS WITH DISTINCT DIFFERENCES**
M. D. Atkinson, A. Hassenklover, October 1984.

SCS-TR-64 **TEACHING FIFTH GENERATION COMPUTING: THE IMPORTANCE OF SMALL TALK**
Wilf R. LaLonde, Dave A. Thomas, John R. Pugh, October 1984.

SCS-TR-65 **AN EXTREMELY FAST MINIMUM SPANNING CIRCLE ALGORITHM**
B. J. Oommen, October, 1984.

SCS-TR-66 **ON THE FUTILITY OF ARBITRARILY INCREASING MEMORY CAPABILITIES OF STOCHASTIC LEARNING AUTOMATA**
B. J. Oommen, October, 1984. Revised May 1985.

SCS-TR-67 **HEAPS IN HEAPS**
T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 **PARTIAL ORDERS AND COMPARISON PROBLEMS**
M. D. Atkinson, November 1984.

SCS-TR-69 **ON THE EXPECTED COMMUNICATION COMPLEXITY OF DISTRIBUTED SELECTION**
N. Santoro, J. B. Sidney, S. J. Sidney, February 1985.

SCS-TR-70 **FEATURES OF FIFTH GENERATION LANGUAGES: A PANORAMIC VIEW**
Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 **ACTRA: THE DESIGN OF AN INDUSTRIAL FIFTH GENERATION SMALLTALK SYSTEM**
David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 **MINMAXHEAPS, ORDERSTATISTICSTREES AND THEIR APPLICATION TO THE COURSEMARKS PROBLEM**
M. D. Atkinson, J.-R. Sack, N. Santoro, Th. Strothotte, March 1985.

SCS-TR-73 **DESIGNING COMMUNITIES OF DATA TYPES**
Wilf R. LaLonde, May 1985.

SCS-TR-74 **ABSORBING AND ERGODIC DISCRETIZED TWO ACTION LEARNING AUTOMATA**
B. John Oommen, May 1985.

SCS-TR-75	**OPTIMAL PARALLEL MERGING WITHOUT MEMORY CONFLICTS** Selim Akl and Nicola Santoro, May 1985.
SCS-TR-76	**LIST ORGANIZING STRATEGIES USING STOCHASTIC MOVE-TO-FRONT AND STOCHASTIC MOVE-TO-REAR OPERATIONS** B. John Oommen, May 1985.