INSIGHT: A DATAFLOW LANGUAGE FOR PROGRAMMING VISION ALGORITHMS IN A RECONFIGURABLE COMPUTATIONAL NETWORK

LINDA G. SHAPIRO
Department of Electrical Engineering, FT-10
University of Washington
Seattle, WA 98195, USA

ROBERT M. HARALICK
Department of Electrical Engineering, FT-10
University of Washington
Seattle, WA 98195, USA

and

MICHAEL J. GOULISH
Department of Computer Science
Michigan State University
East Lansing, MI 48824, USA

Received 15 October 1986
Revised 18 February 1987

Machine vision systems used in industrial applications must execute their algorithms in real time to perform such tasks as inspecting a wire bond or guiding a robot to install a part on a car body moving along a conveyer. The real time speed is achieved by employing simple-minded algorithms and by designing parallel architectures and parallel algorithms for some tasks. The majority of the work on parallel architectures has been limited to architectures that support image processing, but not mid- or high-level vision. In order for more complex vision algorithms to execute in real time, a more flexible architecture is needed.

Our conceptual approach to the problem is a reconfigurable computational network. Each configuration of the network implements an algorithm or class of algorithms. A high-level language expresses the algorithms in a relational form that can be easily translated to the specification for a configuration. The language must be able to encode low-, mid-, and high-level vision algorithms and to efficiently handle not only pixel data, but also higher level structures. In this paper we describe a dataflow language called INSIGHT, which we have designed to meet these needs, and give several examples of parallel machine vision algorithms expressed in the language.

Keywords: Vision hardware; Dataflow language; Reconfigurable computational network.

INTRODUCTION

The main difference between industrial machine vision and general computer vision research is that the industrial applications must run in real time. Real time execution is achieved either by employing very simple-minded algorithms or by designing algorithms that can make use of parallel architectures. The parallel machines that have been designed for vision are mainly pipeline machines and cellular array machines[1]. The pipeline machines consists of a sequence of stages. The pixels of an image pass sequentially through each of the stages where they may be delayed or may become one of the operands of the operation performed by that stage. If there are n stages, then n operations at a time can be performed in parallel.

A cellular array machine consists of an interconnected array of processors. A subimage enters the array machine, one pixel per processor and neighborhood operations are performed on all the pixels in parallel. Again the amount of parallelism depends on the number of processors. Both pipeline and array machines have mainly been used for image processing, not for mid-level or high-level vision.

In order to execute more complex vision algorithms in real time, a more flexible parallel architecture is needed. The architecture must be able to handle more than 8 bit integers representing pixels. It should be able to handle a variety of datatypes representing a number of entities such as region labels and attributes, arc segment properties, and relational types. It should be able to execute efficiently not just a few classes of processing operations, but a large variety of vision algorithms. It must be a multi-instruction, multi-datastream, reconfigurable architecture capable of operating systolically to maintain efficiency. Its reconfiguration capability must include connections of processor to processor or memory to processor. (For a description of a reconfigurable architecture, see Snyder.[2]

Traditionally, useful algorithms that must run at a high speed have been translated to a hardware implementation designed to maximize parallelism and minimize execution time. The translation is done for a small specified set of algorithms by a skilled hardware engineer. Eventually a machine is produced that executes only this set of algorithms.

This approach is not feasible for building a vision system that must perform a variety of different algorithms and should be able to incorporate new algorithms as they are developed. Here, there must be automatic translation of each algorithm from a high-level language into an efficient hardware configuration for a machine that reconfigures itself for each task it is commanded to perform. Such a flexible machine is called a *reconfigurable computational network* (RCN), and the type of language that translates directly into a configuration of the network is called a *dataflow language*.

There are a variety of languages for expressing parallel algorithms, beginning with Concurrent Pascal[3]. More recently, Hoare[4] has developed a theory for communicating sequential processes and a LISP-like language to go with the theory. Another example is OCCAM[5], a rather simple programming language that allows expression of parallel constructs. In the object-oriented programming domain are such languages as Orient[6] and Concurrent PROLOG[7]. While all these languages feature parallelism, none lends itself to the one feature we especially need—direct translation to a configuration of the RCN.

The language that is needed must allow the expression of vision algorithms in a form that can be easily translated to an architectural configuration for both pixel pushing and higher level vision data structure processing. Hardware programming languages and graph description languages are at too low a level. The interesting thing about the data flow in a systolic network is that a high level specification of the configuration of the network is a specification of the program the network is executing. This is different from Von Neumann architectures in which a specification of the architecture tells nothing about what program is executing. The low level specification of a network is a graph having labeled arcs and nodes and has nothing about it which is sequential or procedural. Likewise, a high level specification need not be sequential or procedural. A high level specification of a network is just a specification of the relations which hold in the network. So specification of the configuration of a systolic network amounts to specifying relations. Since the specification is the program which the network executes, the language used to program a systolic network is a language of relations. The language must be naturally non-procedural. From a high level perspective, the semantics of the language describe the essence of the architecture.

LUCID[8] is a dataflow language that we feel comes closest to having the features required in a language for the RCN. INSIGHT is a language in the LUCID family that we have developed to meet the needs described above. In Section 2 we describe the reconfigurable network. In Section 3 we describe the INSIGHT language, and in Section 4 we give some preliminary examples of algorithms encoded in INSIGHT.

2. RECONFIGURABLE COMPUTATIONAL NETWORK: IMPORTANT CONCEPTS

The design of a reconfigurable computational network architecture for computer vision is being done in the opposite order from the design of a traditional machine. Instead of designing the hardware first and then the languages and software, we are designing the language first. The exact design of the RCN will depend on simulation results obtained after encoding a representative set of algorithms. However, in order to specify the language, there must be some understanding of the concepts that will be embodied in the architecture.

The operation of the RCN involves the flow of sequences of values through a network of architectural primitives. More formally, a *configuration* consists of a set of *processors* P and a specification c of the interconnections between the processors. In this discussion a memory is considered a processor. Each processor $p \in P$ is a pair $p = (I_p, O_p)$ where I_p is a named set of input lines and O_p is a named set of output lines. Each connection $c \in C$ is a quadruple $c = (o, p_1, i, p_2)$ specifying that output line o of processor $p1$ connects to input line i of processor $p2$. Since different processors may take different amounts of time to process their inputs and produce their outputs, there must be some conventions that insure a processor will only execute when it has valid data. For this purpose, there is a state associated with each data line. A state is a pair of values $s = (r,q)$ where r is an indication of readiness and q is an indication of acceptance. Legal values for r are:

1) preactive: the processor that produces the data on this line has not yet produced any values,
2) active and ready: valid data, ready to be used by the processors that require it,
3) active and not ready: the data on the line is an old value that was already consumed, but the new value is not yet ready for consumption,
4) postactive: the processor that produces the data on this line has terminated production; no new values will appear in this execution of the algorithm.

Legal values for q are consumed and unconsumed.

A process can execute when all of its inputs are in the state (active and ready, unconsumed) and all of its outputs from its previous execution have been consumed by every process to which they are inputs. The execution of the process takes some finite amount of time. When the execution is just starting, the output lines are in the state (active and not ready, unconsumed). During execution, each one eventually becomes (active and ready, unconsumed). As soon as an output line reaches this state, it is available to the processes that wish to consume it. It reaches the state (active and ready, consumed) only when all of its potential consumers have consumed it.

If at least one input to a process is in state (preactive, *), then all of its outputs are in state (preactive, *), after its previous outputs have been consumed. If at least one input to a process is in state (postactive, *), then all of its outputs are in state (postactive, *) after its previous outputs have been consumed. In general, INSIGHT programmers do not have to worry about these states; the synchronization is taken care of by the hardware.

A *sequence* is an ordered stream of values that are either input to the RCN or are produced by one of the processors of the RCN. In either case, the values of the sequence are associated with a group of data lines. There are two kinds of conceptual orderings associated with the data lines. The first ordering is based on the counting of the real time clock. Each group of data lines necessarily has a value at each real time clock tick. The value of the data line group at the ith clock tick constitutes the ith element in the *real time sequence* associated with those lines. However, in the RCN, it is a processor that puts a value onto a group of data lines. As described above, a processor generates a new value only when each of its old output values has been consumed by all of its consumers and when the inputs it requires are all active and ready. A clock tick occurring when all outputs have been consumed and all inputs are ready generates a *permission tick*. The second kind of ordering is based on the permission ticks. The value of the data line group at the *i*th permission tick is the *i*th element in the *process time sequence* associated with those lines.

When thinking logically about an algorithm in terms of sequences of values that the processors must generate, the INSIGHT programmer generally thinks about the process time sequence. Only when real time signals to or from an external device are input to or output from the RCN might the programmer wish to deal with real time sequences. From the point of view of the RCN hardware, however, it is the real time sequences that must be handled.

3. THE INSIGHT LANGUAGE

In this section we describe the major features of INSIGHT. We do not attempt to define the syntax and semantics of every INSIGHT construct; that belongs in a programmer's guide. Instead, we explain the major constructs and give some examples that illustrate the flavor of the language. The one feature of INSIGHT that is most important to its intended use as a language for the RCN is its use of *relationships* rather than statements or commands. An INSIGHT programmer merely specifies all the relationships that must hold among data elements. He does not have to explicitly state what can be done in parallel or what cannot. He does not have to explicitly pass messages or utilize synchronization primitives. He specifies relationally what the algorithm should achieve and the translator takes care of the rest.

3.1. Programs, Activities, and Functions

INSIGHT programs specify relationships among sequences that will translate to relationships among architectural primitives of the RCN. A *program* is a sequence of configurations of the RCN designed to achieve some goal. The arguments of a program are supplied by and its results must be received by entities outside the RCN such as frame buffers, other external memories, and CRTs. In the simplest case, a program maps to a single configuration of the RCN that can produce the desired results. If the available hardware is not sufficient, then the program maps to several configurations called *activities*, each of which write temporary results to memories and which, when performed sequentially, produce the desired results. INSIGHT programs are modular, they may invoke INSIGHT *functions* to perform subtasks. An INSIGHT *function* translates to a subgraph of the configured hardware that is useful in one or more parts of the total algorithm. It is, however, closer to the usual concept of a macro than a function in a procedural language, since a new copy of the subgraph must be included wherever the results of the function are needed. This allows all such copies of the function to operate in parallel. The following simple example illustrates the program and the function in INSIGHT.

```
program add-and-mult (a[0:99], b[0:99], c[0:99], d[0:99]: integer memory)
        result [0:99]: integer memory
where
  function add (x1, x2: integer sequence): integer sequence
    where
      add = x1 + x2;
    end;
  declare sa,sb,sc,sd,t1,t2, index: integer sequence;
  relations
    index = 0 fby index + 1 until index == 99
    sa = a[index];
    sb = b[index];
    sc = c[index];
    sd = d[index];
    t1 = add(sa,sb);
    t2 = add(sc,sd);
    result[index] = t1 * t2;
endwhere;
```

The inputs to the program are four separate integer memories named *a, b, c,* and *d,* which can be thought of as vector arrays, of 100 elements each. The output is a single integer memory called *result.* The purpose of the program is to produce the equivalent of the procedural code

```
for i = 0 to 99
  result[i] = (a[i] + b[i]) * (c[i] + d[i])
```

using an architectural configuration that performs the two adds in parallel. *Add*, which was written as a function just to illustrate what functions look like, takes in two sequences $x1$ and $x2$ of integers and outputs a single sequence of integers. Because *add* has only one output sequence, we don't have to give its output a separate name; the variable name *add* refers to the output sequence. The program add-and-mult generates a sequence called *index* of the integer values from 0 to 99 using the *fby* operation, which will be defined in Table 1. It uses this sequence of index values to address all five memories in parallel. The values that are indexed in memories *a, b, c,* and *d* become elements of the sequences *sa, sb, sc,* and *sd.* Pairwise elements of *sa* and sb are added together to produce temporary result sequence *t*1. In parallel, pairwise elements of *sc* and *sd* are added to produce temporary result sequence *t*2. Pairwise elements of sequences *t*1 and *t*2 are multiplied together to form an unnamed sequence which is routed back into the output memory, result. Figure 1 illustrates the architectural configuration generated for program add-and-mult. The program could be expressed more concisely as:

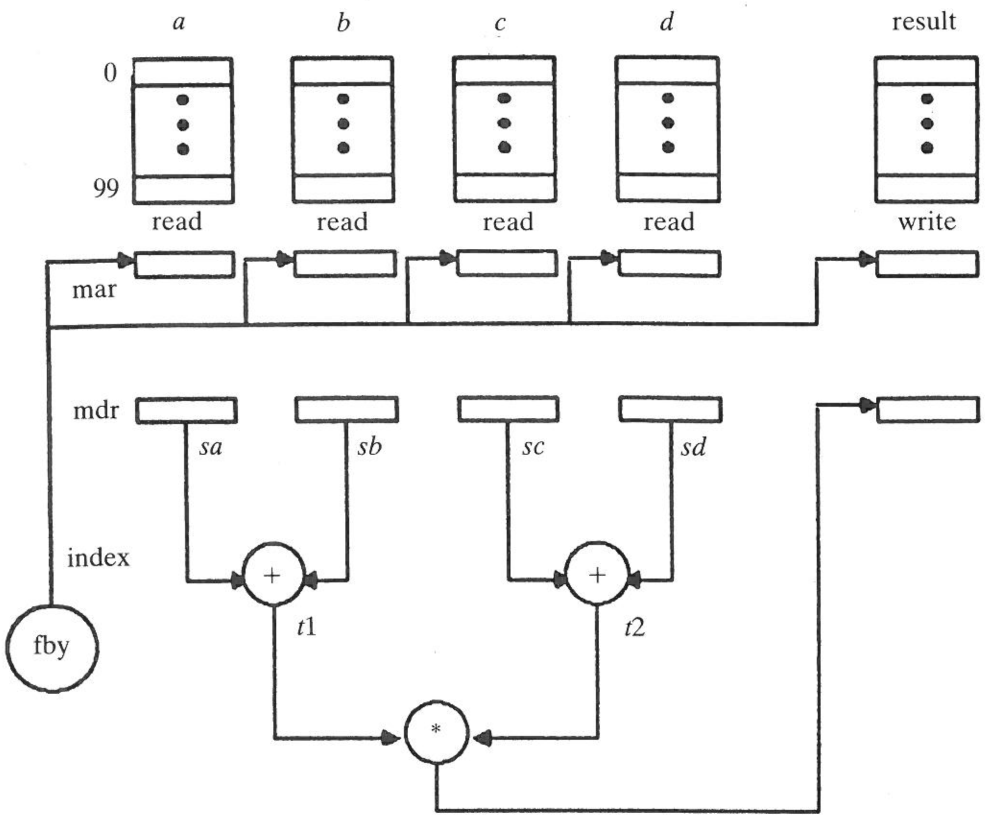


Fig. 1. This illustrates the configuration that would be produced for program add-and-mult.

```
 program add-and-mult (a[0:99], b[0:99], c[0:99], d[0:99]: interger memory)
         result[0:99]: integer memory
where
  declare index: integer sequence;
  relations
    index = 0 fby index + 1 until index == 99;
    result[index] = (a[index]+b[index])*(c[index]+d[index]);
endwhere;
```

3.2. Memories and Data Structures

A *memory* is a random access storage device that is external to, but accessible by the RCN. The purpose of a memory is to store results produced by one INSIGHT program or activity and used by another. Memories are assumed to have only one port, so that only one element of a single memory can be accessed at a time. The following example illustrates the use of memories in a histogram program.

```
program histogram (image[0:511, 0:511]: integer memory)
        hist[0:255]: integer memory
where
  declare row,rowsub,colsub,bin: integer sequence;
  activity initialize
    bin = 0 fby bin + 1 until bin == 255;
    hist[bin] = 0
  endactivity
  activity count
    row = 0 fby row + 1 until row == 511;
    rowsub = row repeat 512;
    colsub = row cycle 512;
    bin = image[rowsub,colsub];
    hist[bin] = hist[bin] + 1;
  endactivity
endwhere
```

In this example, the integer memory *image* can be a frame buffer, while *hist* is just an ordinary vector array. The first activity generates a sequence of indexes that address each bin in the histogram and set it to zero. The second activity generates a second sequence of bin indexes from the gray tone values of the image and increments each location so addressed by one. Pairs of indexes from sequences *rowsub* and *colsub* are used to address the two-dimensional frame buffer memory. The sequence *rowsub* consists of 512 0's followed by 512 1's followed by 512 2's, and so on, generated using the repeat operator. The sequence *colsub* consists of the integers 0 through 511, the entire sequence repeated 512 times. The INSIGHT translator will, of course, translate such two-dimensional references, to valid addresses in the frame buffer memory.

Memories can be used to store more complex data structures than arrays. In particular, linked lists can easily be implemented by using two parallel memories, one for the head of each cell and one for the tail. A list stored in this fashion is accessed through the generation of a sequence of memory addresses; the first is the address of the first cell in the list, and the *n*th comes from the tail field of the (*n*-1)st. Corresponding to the sequence of memory addresses is a sequence of data values that constitute the list. Thus, not only can a stream of pixels flow through the RCN, but also streams of list elements can flow through the network. We expect to provide primitives for list handling, so that mid and high-level vision will be as natural in INSIGHT as image processing.

3.3. Sequence Arrays

It will often be the case that the programmer wants to configure part of the RCN as a pipeline of several stages. We provide the notion of sequence arrays as a convenience tool, so that the programmer will not have to repeat any relations that are the same for each stage. A *sequence array* is an array of sequences declared as a translator time entity that expresses a configuration of several sequences that have some relation to each other. For example, the following INSIGHT code generates the configuration shown in Fig. 2. The output of each stage is its input plus one. Instead of having to repeat this relationship three times for three stages, the repetition is accomplished by the foreach construct.

```
declare
  stage: translator integer;
  output[0:3]: integer seqarray;
relations
  output[0] = 0;
  foreach stage = 1 to 3
    output[stage] = output[stage − 1] + 1;
  endfor;
```

In this example, the translator integer *stage* is used in the translation time *foreach* loop that defines the elements of the sequence array. The sequence array itself is merely a shorthand notation for defining four related sequences. In this case, the relationship is that the first of the four sequences is the input and the remaining three are outputs of three stages in a pipeline configuration.

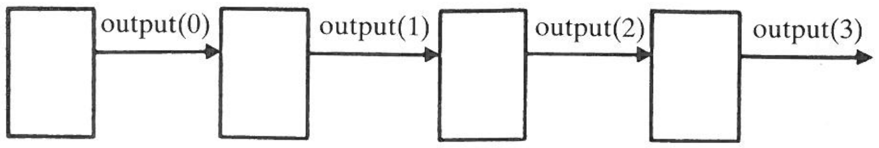


Fig. 2. This illustrates an example of a pipeline configuration expressed in INSIGHT with the sequence array notation.

Sequence arrays can also be used to express sequences whose relationship is parallel. For example, the following INSIGHT function inputs a sequence of pixels coming from an image and a vector of four thresholds. It applies the four thresholds in parallel producing four sequences of pixels representing binary images. The output is a sequence of integer values, each value corresponding to the sum of the corresponding pixels of the four binary images.

```
function threshold-and-sum (image:integer sequence;
        thresholds[1:4]: integer memory): integer sequence
where
  declare
    integer seqarray sum-images[0:4];
```

```
        binary seqarray thresholded-image[1:4];
        translator integer k;
    relations
        sum-images[0] = 0
        foreach k = 1 to 4
          thresholded-image[k] = if image > thresholds[k]
                then 1 else 0 endif;
          sum-images[k] = sum-images[k-1] +
                thresholded-image[k];
        endfor;
          threshold-and-sum = sum-images[4];
    endwhere
```

Table 1. The current process time INSIGHT sequence operators. Operators marked with an asterik (*) are from LUCID.

Operation	Explanation
*R = first (X);	R will be a constant sequence whose values are equal to the first value of X.
R = last (X);	R will be a constant sequence whose value is the last value of X before X becomes postactive.
R = previous (X);	R_i is the same as X_{i-1}.
*R = next (X);	$R_i == X_{i+1}$. But R does not include the first value of X.
R = X as long as C;	C is a binary/boolean sequence. $R_i = X_i$ as long as $C == 1$. When C becomes 0, R becomes postactive. If the first value of C is 0, then R has no elements.
*R = X as soon as C;	*The first value of* R is X_i such that $C_i == 1$; each subsequent value of X becomes the subsequent value of R. If $C_0 == 1$, then R is identical with X.
R = X attime Y;	Y is a sequence of integers. Select the values of X such that their process-time subscripts are equal to the integers in Y. The values in Y must always ascend.
*R = X fby Y;	R's first element will equal the first element of X. The next element of R will be the first element of Y, and from then on: $R_i = Y_{i-1}$.
R = X dby K;	The first K elements of R will be the preactive token. From then on R_{K+i} will equal X_i for all $i>=1$. K must be an integer constant.
R = X concatby Y;	This is the concatenation operator. R will be X as long as X is not postactive. The remainder of R will be the entire sequence Y.
R = X until C;	$R = X$ up to and including X_i where C_i is the first true value of C.
*R = X upon C;	C is a boolean or binary sequence. The first value of R will be the first value of X. Thereafter, if C_i is true, then R_{i+1} takes the next value of X. If C_i is false, then $R_{i+1} == R_i$.

Table 1 cont'd

$A = B$ extended by C;	Until the first true C value, $A = B$. From the first time i such that $C_i ==$ true, $A_i = A_{i-1}$.
$A = B$ sustained by C;	If $C_i = 0$, then $A_i = B_i$. If $C_i = 1$, then $A_i = A_{i-1}$.
$*R = X$ whenever C;	R takes all values X_i such that $C_i = 1$.

3.4. Sequence Operators

INSIGHT provides a variety of operations for manipulating sequences. The *if-then-else* expression (illustrated in function threshold-and-sum above) allows a sequence to be generated, each of whose elements is chosen from one of two input sequences, depending on the value of a boolean expression sequence. The conditional expression allows a sequence to be generated, each of whose elements is chosen from one of many input sequences. This construct is similar to the *cond* function in LISP. All the standard numeric operators and functions such as +, −, *, /, **, abs, cos, sin, log, etc., are provided, as are all the standard logical operators and the comparative character operators. The interesting operators in INSIGHT are the *sequence operators*, a set of special sequence generation and manipulation operators. Some of these come from LUCID[8], and others were invented to express constructions that can occur in a configuration of the RCN. Some of the operators can be explained in terms of the process time sequences they produce, and a few can only be explained in terms of their effects in real time. Table 1 lists and briefly defines the current INSIGHT process time sequence operators, indicating which are from LUCID. The real time operators are under development.

4. EXAMPLES

We are in the process of encoding vision algorithms in INSIGHT, to be sure the language is sufficient for our needs. In this section we present a few of the simpler vision algorithms and utility algorithms as examples.

4.1. Count

Count is a utility function that generates useful index sequences. It has four arguments: start, step, increment, and reset. Start is a sequence of start values, and step is a sequence of corresponding step values. Increment is a boolean valued sequence that tells whether or not to increment the current value of the sequence being generated by the step value to obtain the next value or just to repeat the current value. Reset is a boolean valued sequence that tells when to reset the sequence being generated back to the next start value and begin using the next step value.

```
function count (start, step: integer sequence;
            increment, reset: boolean sequence)
        : integer sequence
where
  declare
    integer sequence hold-start, hold-step;
  relations
    hold-start = start upon (reset==true);
    hold-step = step upon (reset==true);
    count = start fby if (reset==true)
                  then next(hold-start)
                  else if (increment==true)
                       then count + hold-step
                       else count
                       endif
                  endif;
endwhere
```

SAMPLE TRACE:

```
           start= < 0 1 2 3 4...
            step= < 1 1 1 1 1...
       increment= < 1 1 1 1 1...
           reset= < 0 0 0 0 1 0 0 0 1 0 0 1...
      hold-start= < 0 0 0 0 0 1 1 1 1 2 2 2 3...
 next hold-start= < 0 0 0 0 1 1 1 1 2 2 2 3...
       hold-step= < 1 1 1 1 1 1 1 1 1 1 1 1 1...
    if-then-else= < 1 2 3 4 1 2 3 4 2 3 4 3...
           count= < 0 1 2 3 4 1 2 3 4 2 3 4 3...
```

4.2. Advance

Advance is a utility function that takes two arguments: a sequence of any type and an integer n. It produces a resultant sequence whose first value is the $(n + 1)$th value of the input sequence. It uses the function count without arguments which produces the default sequence <1 2 3 4 5 ...>.

```
function advance (seq: sequence; n: integer constant)
        : sequence
where
  relations
    advance = s wvr count( ) > n;
endwhere
```

4.3. Binary Clone

Binary clone is an image processing function that illustrates one implementation of the dilation operation in mathematical morphology[9]. It inputs a sequence of pixels from a binary image and a memory indicating the appropriate amount of delay needed by each stage in a pipeline of processors. It outputs a sequence of pixels for the resultant dilated image. For more details on the operations of mathematical morphology and a pipeline machine designed for executing them, see Sternberg[10].

```
function binary-clone (image: binary sequence;
                       delays: integer memory)
    : binary sequence;
where
  declare
    size, stages: translator integer;
    result[0:size]: binary seqarray;
  relations
    result[0] = image;
    foreach stage = 1 to size
      result[stage] = result [stage − 1]
                      or
                      result [stage − 1] dby delays[stage];
    endfor;
    binary-clone = result [size];
endwhere
```

4.4. Mapper

As a last example, we present a very simplified form of an order dependent structural shape matching algorithm[11]. Two shapes are to be compared and an error of the match computed. Each shape is represented by a list of its parts and their attributes. The parts of each shape are numbered from 1 to *max-parts*. Function *mapper* inputs two integer sequences *c-first* and *u-first*. Each pair of corresponding elements (c-first$_i$, *u*-first$_i$) represent the hypothesis that part number *c-first* of the first shape corresponds to part number *u-first* of the second shape. Thus the inputs can be thought of conceptually as a sequence of hypotheses. The three outputs are *best-error*, whose last value will be the error of the best match, and *(best-c-first, best-u-first)* whose last values indicate the best hypothesis. *Maperr* uses a utility function *compute-error* which is assumed to compute the error between a pair of parts, based on their attributes which are stored in the local memory of each stage, so that parallel access is possible.

```
function mapper (c-first, u-first : integer sequence)
              besterr: real sequence; best-c-first, best-u-first: integer sequence
where
  declare
    c-part[1:max-parts], u-part[1:max-parts]: integer seqarray;
    error[1:max-parts]: real seqarray;
    best-error: real sequence;
    best-c-first, best-u-first: integer sequence;
    max-parts, stage: translator integer;
  relations
    c-part[1] = c-first;
    u-part[1] = u-first;
    error[1] = compute-error (c-first, u-first);
    foreach stage = 2 to max-parts
        c-part[stage] = (c-part[stage-1] mod max-parts) + 1;
        u-part[stage] = (u-part[stage-1] mod max-parts) + 1;
        error[stage] = error[stage-1] +
                  compute-error(c-part[stage], u-part[stage]);
    endfor;
    best-error = MAXINT fby if error[max-parts] < best-error
                  then error[max-parts]
                  else best-error;
    best-c-first = 0 fby if error[max-parts] < best-error
                  then c-first
                  else best-c-first;
    best-u-first = 0 fby if error[max-parts] < best-error
                  then u-first
                  else best-u-first
endwhere
```

5. CONCLUSIONS

The INSIGHT language allows the expression of low, mid, and high-level vision algorithms in a relational form that can be easily translated to a configuration of a reconfigurable computational network. The language is still under development and is being tested for utility by encoding a set of representative vision algorithms and simulating their execution. Using the language is, in fact, teaching us how to write parallel algorithms for machine vision. We expect this research to lead to a fast, general purpose, machine vision system.

REFERENCES

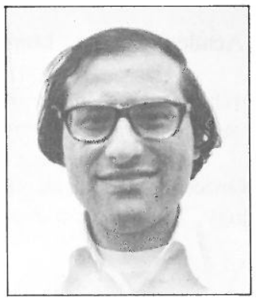
1. S. Yalamanchili, K.V. Palem, L.S. Davis, A.J. Welch, and J.K. Aggarwal, "Image processing architectures: A taxonomy and survey," in *Progress in Pattern Recognition 2,*

eds. L.N. Kanal and A. Rosenfeld, Elsevier, Amsterdam, 1985, pp. 1–37.
2. L. Snyder, "Introduction to the configurable highly parallel computer," *IEEE Computer* **15** (1982) 47–56.
3. P. Brinch Hansen, "The SOLO operating system", *Software — Practice and Experience* Vol. 6, 1976.
4. C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall, N.J., 1985.
5. *OCCAM Programming Manual,* Prentice-Hall, N.J., 1984.
6. Y. Ishikawa and M. Tokoru, "A concurrent object-oriented knowledge representation language Orient 84/K: Its features and implementation," *OOPSLA '86 Proceedings,* ACM, Sept. 1986, pp. 232–241.
7. E. Shapiro and A. Takeuchi, "Object oriented programming in concurrent PROLOG," *New Generation Computing* **1** (1983) 25–48.
8. W.W. Wadge and E.A. Ashcroft, *LUCID: The Dataflow Programming Language,* Academic Press, London, 1984.
9. J. Serra, *Image Analysis and Mathematical Morphology,* Academic Press, London, 1982.
10. S.R. Sternberg, "An overview of image algebra and related architectures," in *Integrated Technology for Parallel Image Processing,* ed. S. Levialdi, Academic Press, London, 1985, pp. 79–100.
11. L.G. Shapiro, R.S. MacDonald and S.R. Sternberg, "Ordered structured shape matching with primitive extraction by mathematical morphology," *Patt. Recognition.* **20** (1987) 75-90.

Linda G. Shapiro was born in Chicago, Illinois, in 1949. She received the B.S. degree in mathematics from the University of Illinois, Urbana, in 1970, and the M.S. and Ph.D. degrees in computer science from the University of Iowa, Iowa City, in 1972 and 1974, respectively.

She was an Assistant Professor of Computer Science at Kansas State University, Manhattan, from 1974 to 1978 and an Assistant Professor of Computer Science from 1981 to 1984 at Virginia Polytechnic Institute and State University, Blacksburg. She was Director of Intelligent Systems at Machine Vision International in Ann Arbor from 1984 to 1986. She is currently an Associate Professor of Electrical Engineering at the University of Washington. Her research interests include computer vision, artificial intelligence, pattern recognition, robotics, and spatial database systems. She has co-authored a textbook on data structures with R. Baron.

Dr. Shapiro is a senior member of the IEEE Computer Society and a member of the Association for Computing Machinery, the Pattern Recognition Society, and the American Association for Artificial Intelligence. She is Editor of *Computer Vision, Graphics, and Image Processing* and an editorial board member of *Pattern Recognition*. She has been General Chairman of the IEEE Conference on Computer Vision and Pattern Recognition in 1986, General Chairman of the IEEE Computer Vision Workshop in 1985, and Co-Program Chairman of the IEEE Computer Vision Workshop in 1982; she has served on the program committees of a number of vision and AI workshops and conferences.



Robert M. Haralick was born in Brooklyn, New York, on September 30, 1943. He received a B.A. degree in mathematics from the University of Kansas in 1964, a B.S. degree in electrical engineering in 1966 and a M.S. degree in electrical engineering in 1967. He has worked with Autonetics and IBM. In 1965 he worked for the Center for Research, University of Kansas, as a research engineer and in 1969, when he completed his Ph.D. at the University of Kansas, he joined the faculty of the Electrical Engineering Department there where he last served as Professor from 1975 to 1978. In 1979 Dr. Haralick joined the Electrical Engineering Department at Virginia Polytechnic Institute and State University where he was a Professor and Director of the Spatial Data Analysis Laboratory. From 1984 to 1986 Dr. Haralick served as Vice President of Research at Machine Vision International, Ann Arbor, MI. Dr. Haralick now occupies the Boeing Clairmont Egtvedt Professorship in the Department of Electrical Engineering at the University of Washington.

Dr. Haralick has done research in pattern recognition, multi-image processing, remote sensing, texture analysis, data compression, clustering, artificial intelligence, and general systems theory, and has published over 200 papers. He is responsible for the development of GIPSY (General Image Processing System), a multi-image processing package which runs in a workstation environment.

He is a Fellow of the Institute of Electrical and Electronic Engineers, and a member of the Association for Computing Machinery, and the Pattern Recognition Society.

Biographical sketch and photo of M.J. Goulish was not received.