33

1981-15

STATE UNIVERSITY OF NEW YORK AT BUFFALO

DEPARTMENT OF COMPUTER SCIENCE

INFERENCE AND CONTROL IN MULTIPROCESSING ENVIRONMENTS

by

Harold Shubin

September 1981

Technical Report Number 186

This work was supported in part by the National Science Foundation, under Grant MCS80-06314.

INFERENCE AND CONTROL IN MULTIPROCESSING ENVIRONMENTS

by
Harold Shubin

A thesis submitted to the
Faculty of the Graduate School of State
University of New York at Buffalo in partial
fulfillment of the requirements for the degree of
Master of Science

Abstract

The ideas behind inference, as used in Artificial Intelligence (AI) systems, are similar to those of certain control structures used in other areas of computation.

This paper discusses those ideas and specifically studies forward, backward and bi-directional inference; and the data flow concept, lazy evaluation and bi-directional search. A model of computation caled the Supplier-Producer-Consumer (SPC) Model, is introduced as a vehicle for making contrasts between pairs of inference and control strategies. Contrasts along another dimension are made in the model by discussing static and eager evaluation schemes.

Multiprocessing is an idea which has been implemented differently in different areas of computer science. This paper discusses the benefits of software simulations of multiprocessing on uni-processing systems for these control and inference methods.

Finally, bi-directional methods (computation, inference and search) are discussed. The combination of forward and backward computation methods allows each to assist the other, and suggests new ways for a program to interact with a user.

1. INTRODUCTION

One idea may be used in a number of ways. In the area of computation, that is shown by the existence of forward inference and the data flow concept. Both of these techniques, one an inference technique and the other a control scheme, are based on the same idea: the immediate and full use of all input data. Neither one requires that computations proceed in a predetermined order -- any operation whose inputs are present may begin. Along with these data-entry actions, there is the data-request class consisting of lazy evaluation and backward inference. These act when a request is made for incompletely specified information or incomplete evaluations.

Bi-directional computation includes bi-directional inference and bi-directional search. These schemes provide for evaluations to proceed from opposite ends of the network which represents the problem being solved. We also discuss eager and static evaluation schemes, which allow another dimension of comparisons to be made. These two control techniques differ in the amount of parallelism involved and in the amount of unnecessary work that they allow.

A model of computation, called the Supplier-Consumer-Production (SPC) Model, is introduced as a vehicle for making the contrasts between the inference and control methods.

Many of these techniques benefit from multiprocessing, which allows more than one operation to be executed in parallel. The discussion of software-simulated multiprocessing shows how it improves the operation of the inference and control methods and especially the bi-directional computations.

1.1. Organization of the Paper

Section 2 gives some background information on the topics to be discussed -- inference, control, and multiprocessing. Demons are also introduced.

Section 3 explains the SPC model. All of the inference and control techniques are introduced and contrasted. The relationship between the model and computation is shown in the discussion of combinations of production and acquisition methods.

Section 4 discusses the result of combining forward and backward computation schemes. This combination gives bi-directional computation, inference, and search, which are discussed and contrasted.

Section 5 contains the summary.

1.2. Acknowledgements

I wish to thank my advisor, Stu Shapiro, for his guidance and advice in this work. I have learned a great deal during the time that I have worked with him.

The members of the SNePS Research Group at SUNY Buffalo, especially Don McKay and Joao Martins, contributed a great deal in the way of constructive criticism. The entire group was involved in the discussions which led to this thesis.

I am very grateful to Barb Buck for spending so much time editing the text, for being supportive, and for being a friend.

Last, but certainly not least, there's "The Team." It's been swell.

This work was supported in part by the National Science Foundation, under grant MCS80-06314.

2. INFERENCE, CONTROL AND MULTIPROCESSING

This section introduces the topics discussed in the remainder of the paper and gives general background information on each topic.

2.1. Inference

Simulating intelligence and intelligent behavior is a goal of AI. For instance, many current research projects involve natural language understanding. Inference is an important technique for this work, because much of what is done in understanding language involves making inferences by "applying previously gained knowledge to a text or utterance" [Charniak 76a]. Inference can be used to answer questions or to glean all possible information from a new input.

This paper makes some assumptions about inference:

* Rules are used to describe the action of a particular inference. The pattern, or pre-condition, of the rule is called an antecedent. A rule may have multiple antecedents, all of which need to be satisfied in order for the rule to fire. The result of an inference is the assertion of an instance of the consequent, which is stored in the data base as if it were an input.

Figure 2.1 shows two rules which will be used throughout this paper to demonstrate the actions of each of the inference and control schemes.

Rules to be used in all discussions of inference and control types:

```
1)          z = x * y + w          (algebraic version)
      or    (x & y) or w => z   (boolean version)
      or    IF (x & y) OR w   THEN z

2)    All students attend school
      ∀x  (Student(x) => AttendSchool(x))
```

In the first rule (all versions), w, x, y, and z may be any propositions of arbitrary complexity.

Figure 2.1
Two Simple Inference Rules

---

* The same rules are used for forward inference as for backward inference. The deduction component of SNePS has this property [Martins 81; Shapiro 79, 81b]. Other systems (e.g., PLANNER [Hewitt 69, 71]) do not, but instead specify that a particular rule (or "theorem") be used for either forward <u>or</u> backward deduction. Section 4 explains the importance of this to bi-directional inference.

2.2. Control Techniques

Control, in the sense used in this paper, relates to the method and extent to which a program evaluates statements. The techniques studied range from immediately evaluating all expressions to evaluating nothing until required to do so, and from using highly concurrent evaluation to purely sequential methods. Note that the difference between what is "control" and what is "inference" is not absolute.

Demons

A rule-based system can be implemented in a number of ways. One characterization is how the rules are checked and fired, i.e., whether they are passive or active.

In a production system, rules are looked at and checked by an interpreter until a match is found with some part of the data base [Davis 75]. The role of the rules in this type of system is a passive one.

Rules represented by demons are active participants. Using Davis and King's terminology, each demon has its own interpreter which tells it when to fire. A demon is

> a portion of a program which is not invoked explicitly, but which lays dormant waiting for some condition(s) to occur... For example, a knowledge manipulation program might implement inference rules as demons. Whenever a new piece of knowledge was added, various demons would activate and would create additional pieces of knowledge by applying their respective inference rules to the original piece. These new pieces could in turn activate more demons as the inferences filtered down through chains of logic. Meanwhile the main program could continue with whatever its primary task was. [Jargon 79].

Demons are useful because "a lot of common sense knowledge consists in knowing all sorts of tiny little facts that pop up when one is thinking about related objects" [Steels 79]. These processes can work without interrupting the main task if they are implemented as demons.

Charniak, in [Charniak 76b], states that all inferences should be made as soon as enough information to make them enters a system. Toward that end, he introduced demons to "look forward." His demons are implemented with <u>base routines</u>, <u>bookkeeping</u>, and <u>fact finders</u>. Base routines are used to activate demons, and may be responsible for more than one demon each. Bookeeping is used to keep the data base updated when new information enters the system. Factfinders are used to deal with facts which are not important enough to be asserted in the data base, but which must be dealt with.

Some degree of multiprocessing is required for demons to keep a watch on the proceedings. Each demon may be given a processor (real or simulated, see the discussion of multiprocessing below) when it begins executing. Charniak makes the point that one must set the demons up before running the program, and that they are not "learned" [Charniak 72]. Section 4, on bi-directional computation, describes situations where demons are created dynamically.

Demons represent general rules. A demon would not say "If it is raining and John is outside, John will get wet." It would instead say "If it is raining and ACTORa is outside, ACTORa will get wet." While certain demons might only relate to certain situations, they can be used in all instances of those situations.

2.3. Multiprocessing

Multiprocessing is a technique used for speeding up execution in a computer system.

> A multiprocessor may be defined as an integrated computer system containing two or more central processing units. The qualification 'integrated' implies that the CPUs cooperate in the execution of programs. [Hayes 78, p 448]

By increasing the number of processes available to work on a program, the execution should require less real time to complete. There will not be an n-fold increase in performance with an n-fold increase in the number of processors, however, because of the need to share limited resources in the system.

Some further restrictions mentioned by Hayes are that the CPUs must be able to execute different processes of the same program, or that they must share memory and input/output systems and be controlled by the same operating system.

Hayes also makes a distinction between standalone, and indirectly and directly coupled multiprocessing systems. This distinction depends on the degree of communication between the CPUs involved. Note that in the discussion of software simulation of multiprocessing, these distinctions are not relevant.

Lessing and Corkill [Lessing 81] describe two types of multiprocessing systems: The first type is called CA/NA (Completely Accurate, Nearly Autonomous) systems, in which each node has all of the information that it needs, and can complete a computation on its own. FA/C (Functionally Accurate, Cooperative) systems, on the other hand need sharing of information between nodes. They may not be able to perform complete computations due to a lack of data and/or incomplete knowledge of an algorithm.

The nodes in an multiprocessing system can cooperate in two ways [Smith 81]: task-sharing, in which they split the work involved in the problem; and result-sharing, in which the experts report partial results to each other as they work. In the former case, the problem is such that it can be divided into sub-tasks to be solved by one or more experts working together. The other method is used in cases where the results of one sub-task are necessary for the completion of another sub-task and so communication between the experts is necesary.

In an article on AI programming languages, Bobrow and Raphael give a less strict definition of multiprocessing:

> The basic control innovation that is now being made available is the ability to save a module and its context in a state of suspended animation. ... The control structure induced by this model is a tree of modules, with control passing among any of the suspended modules in the structure. If only one process is active at a time, we call it a <u>coroutine</u>

regime. If processing can be thought of as going on simultaneously in several modules, we call it a multiprocessing regime. Multiprocessing is usually done by scheduling through time-quantum interrupts at the system level, or time allocation in the language interpreter [Queues are also used, see below.] [Bobrow 74, p 157].

By simulating multiprocessing, one gets a more dynamic flow of control than in a strictly sequential system as well as the ability to (seem to) do more than one thing at a time. The system is still a uni-processor, but over time it appears to be doing multiprocessing. This is helpful in using demons, for example, or for inference techniques which can then run while memory structures are being built for a story being read. Unless specified otherwise, "multiprocessing" in this paper should be taken to mean simulated multiprocessing.

A process relinqushes computing power to another process when it is deactivated, whether by the multiprocessing operating system, or by itself. Reasons for deactivation include successfully completing an assignment, no longer being needed because of the results of another process(es), or a process using up its complement of resources such as time or the number of inferences made. The latter reason is called resource-limited processing.

The next process to receive control can be the current process's parent, a process it has created, or one which was deactivated earlier and is now in a state of suspended animation. There is no requirement to return control to the previous process, as in a purely sequential system. As

Bobrow and Raphael note, control can pass among the processes according to the tree-structured hierarchy or not, as required by the execution.

One implementation for simulating this type of multiprocessing on a strictly sequential system is the use of queues. As processes are deemed executable, they are put on a queue (or one of many, depending on the system design). An evaluator takes processes from the queue one after another to re-activate them. Processes may be allowed to create and schedule new processes as they execute, so the flow of control may be very dynamic.

Stallman and Sussman [Stallman 77] use a number of queues, each with different priorities, to implement their EL program. In MULTI [McKay 80], there is a "process queue," which contains processes to be evaluated, and any of these processes may have its own queue of processes. Steels describes a similar organization for his Constraint Machine [Steels 80], where experts (constraints) wait on the "top-process-queue" for evaluation and each may have its own queue of activities to perform. Reiger and Small [Reiger 81] use a RUN-ME queue to hold the list of words to be parsed in their distributed word-expert natural language parser. As parsing of words is completed or suspended, they are taken off of the queue.

3. THE SPC MODEL

The Supplier-Producer-Consumer (SPC) Model is an analogy between a hypothetical manufacturing situation and the inference and control structures discussed in this paper. The system consists of three parts: a producer, suppliers, and consumers as shown in Figure 3.1. The producer manufactures only one type of product, using a number of supplies. Each supply is produced by one or more supplier. The consumers can each use the product.

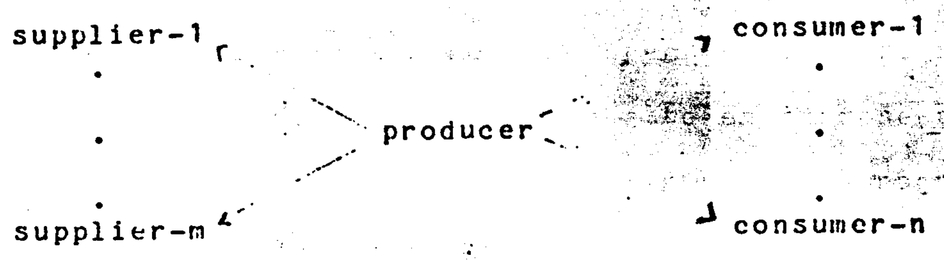


Figure 3.1
Diagram of the SPC analogy

Each member of the model plays each role at different times. In order to supply its producer, the supplier must consume the product of an earlier producer, and so on. The supply and consumption rates may therefore be as varied as the production rate.

The SPC system performs differently depending on the control flow schemes imposed on it. Data flow and lazy evaluation can be looked at as being means of production, while eager and static evaluation are methods of acquisition of supplies.

Each of the production techniques and its corresponding inference methods will be described. Its effect on the SPC model will also be discussed. The same treatment will be applied to the acquisition techniques. Four combinations of production and acquisition will be analyzed, considering production both as control and as inference.

3.1. Production Methods

This model includes two types of production, data-entry (forward) and data-request (backward). Each type will be discussed as an inference or control method in the SPC model.

3.1.1. Data-entry Production

The two data-entry production methods, data flow and forward inference, share the idea that full and immediate use of all input data should be made. As soon as new information enters such a system, all actors or rules which can use it may do so concurrently.

Forward Inference

When new knowledge is added to a system capable of forward inference, inference rules may be used to derive more information from it. According to Charniak, the purpose of forward inference is to find connections between new information and that which already exists in memory [Charniak 76b].

In order for a rule to fire in a forward inference mode, input assertions are matched against the antecedents of the rules in the system. As matches are found and rules have all of their antecedent conditions satisfied, forward inferences are made. In a multiprocessing environment, processes can be set up for inferences corresponding to each match found, and they can all be active simultaneously.

In <u>full forward inference</u>, the resulting assertions may also trigger new inferences. This infer-assert-infer loop will continue until all of the possible new inferences have been made. In <u>restricted forward inference</u>, only inferences directly resulting from the input are performed.

Charniak refers to forward inference as "read-time inference," because he deals with story understanding. A more general term is "knowledge-acquisition-time inference."

Wilks mentions a dispute over the amount of forward inference that should be made in a program in [Wilks 76]. Instead of allowing full forward inference, he suggests a <u>laziness hypothesis</u> to restrict the number of inferences made.

This hypothesis states that one should not introduce into a knowledge base "more information at any point than necessary ... unless the problem cannot be solved at a more superficial level" [Wilks 75]. The same paper describes his preference semantics system, which illustrates this point.

The first part of Figure 3.2 shows the algebraic equation expressed in terms of boolean operators. The rule only fires when the antecedent expression is satisfied, as when either (x & y) or w is asserted as true.

In the second part of the figure, a simple rule, "All students attend school", which might be used for reading a story is shown. This rule may seem a little redundant to a human. We "know" that students go to school, but a program needs this information explicitly stored, either as an assertion or in a rule. Instead of wasting space in the data base with information about <u>every</u> possible student, only those so identified in the input would have this asssertion made about them. A simple rule like this can act as a demon in a multiprocessing system which does full forward inference -- it can sit in waiting, looking for information that someone is a student and then add whatever it knows about students, such as the fact that he or she goes to school.

This rule would, of course, only be useful in a situation where being a student and attending school were important. In other situations, the information about someone's being a student could be added in a manner which would not trigger forward inference. A user could then query the system at a later time about whether a person attends school, and trigger backward inference. See the discussion of backward inference, in section 3.1.2.

```
Rule: (x & y) or w => z

Assertion: x
    Result: nothing, x is not enough to satisfy the
            antecedent conditions of the rule.
Assertion: w
    Result: z is asserted in the data base, because w
            alone is enough. If this were a full
            forward inference system, z could be used
            to trigger any other existing rules.


Rule:  ∀ x (Student(x) => AttendSchool(x))

Assertion: Student(Mary)
    Result: The rule fires, as its one antecedent
            is present. The fact that Mary attends
            school is recorded in the data base:
            AttendSchool(Mary).
            For the significance of this type of
            rule, see the text.

Final result: Student(Mary), and AttendSchool(Mary),
            x, w, and z are all asserted in the
            data base. The processes which were
            used in all of the deductions still
            remain in the system, so that if
            Student(Mary) were asserted again,
            a new deduction would not have to be
            made.
```

Figure 3.2
Forward Inference Examples

Data Flow

A data flow system is one which eschews the traditional concept of evaluating operations or instructions one at a time, in an order prescribed at the time of writing the code. An operation is performed when its data inputs are ready. Data flow systems use this data dependency as their only control strategy.

As described in [Weng 75], "the 'data flow' concept is based on the observation that an operation (or an instruction) should be executed as soon as the required input operands are made available by the completion of operations supplying the inputs." Because a data flow system waits only for required data inputs and does not process instructions sequentially, a number of operations may be ready for execution simultaneously and "thus, highly concurrent computation is a natural consequence of the data flow concept" [Dennis 80].

Data flow programs are generally represented as program flow graphs with actors, arcs and tokens. Figure 3.3 shows the graph for the expression "6=3*2." An actor is specified by a function name inside a circle (e.g., "*", ">", "merge"), and is used to specify an operation or an actual value used in a constant expression. All actors "generate some value as their result" [McGraw 79]. Tokens represent the values carried in and out of the actors on input and output arcs, respectively. In Figure 3.3, the actor is "*" for multiplication and the tokens representing the input and output values are written on the arcs.

A data flow system is data-driven, and computations are triggered by the input of data into the system. Existing data flow languages do not specify input/output procedures, because of the difficulty of incorporating them into the applicative nature of the data flow concept. (See [Arvind 76, Dennis 79b] for descriptions of two data flow languages). An actor can fire as soon as all of its input arcs have one token each, however they enter the system, and its output arcs are clear.

Firing rules show how an actor processes its inputs. An example of a rule for the multiplication operator ("*") is shown in Figure 3.4. The "before" pattern shows that the input arcs have tokens, but the output arcs are clear. The "after" pattern demonstrates how the operator processes the input -- the input arcs are clear and a token representing the result is on the outut arc. (The figures for program flow graphs are modeled after [Dennis 79a, Weng 75, Leung 75]).

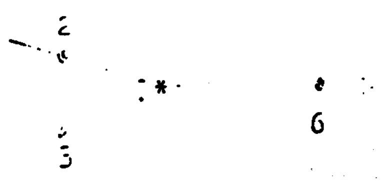


Figure 3.3
Example of a Data Flow Multiplication Actor

The multiplication actor is always shown with two input arcs and one output arc, even if the arcs have no values on them. Actors are connected in a network to represent a program as in Figure 3.5, which represents the algebraic expression of Figure 2.1. Notice that data flow systems do not use variables, and instead have "immutable" values [McGraw 79]. The values are held on the arcs as long as they are in use.

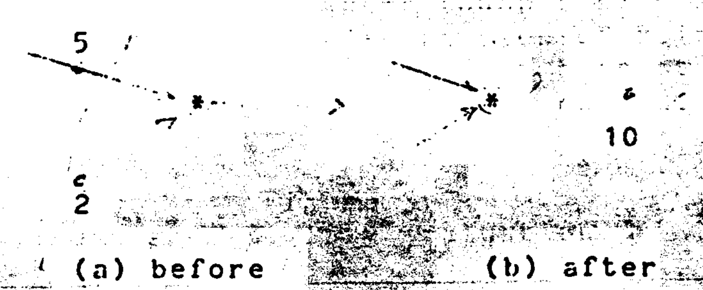


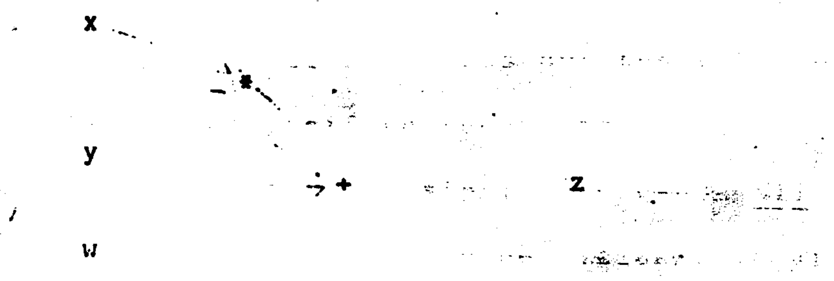
Figure 3.4
Firing Rule For Multiplication

Effect on the SPC Model

The producer has the highest throughput rate when it operates in the data flow mode. Work is begun on a product as soon as all of the necessary parts are available, which might be because of orders (see the discussion of acquisition in section 3.2), or because earlier producers completed products which this producer uses as input. Normally, only the whole product will be built at any one time, but if the product is decomposible into subparts, those parts may each be built whenever their input parts are present.

Equation: $z = (x * y) + w$

Program flow graph:



Input: x
    Result: no action, because x is not enough to trigger any actor.
Input: y
    Result: the multiplication actor fires, sending the value (x*y) to the addition actor.
Input: w
    Result: the addition actor fires, and the result is placed on its output arc for use by any other actor.

Final result: this part of the system is in the same state as before the example. All input arcs are clear and the actors are prepared to fire.

Figure 3.5
Data Flow Example

As soon as the whole product is complete, it is sent along for use by the consumers. It is expected that one of them will be able to use the product but if not, it is wasted. No orders are accepted or waited for by the producer; if parts are present, products are built. Excess product is not stockpiled.

Wasted output is hopefully made up for by the elimination of bookkeeping (e.g., recording orders from consumers) and by the efficient use of time in production. In addition, when the SPC system is organized, it is expected that the suppliers and consumers will be chosen in a balanced manner to minimize waste.

Contrast

There are some differences between data flow and forward inference as presented here. They are not necessarily differences in the techniques themselves, but in the implementations.

An inference system asserts its conclusion by building new structures or by asserting existing structures. In a data flow system, no permanent record of a calculation is left behind. The assertion is merely a token on an output arc which disappears if not used. This is a characteristic of data flow systems -- they are side-effect-free and nothing is changed but the values which pass along the arcs.

3.1.2. Data-request Production

Not doing any work until it is requested is the concept involved in the data-request production techniques, lazy evaluation and backward inference. Evaluations and inferences are held off until explicitly requested by the user or another part of the system.

Backward Inference

An inferencing system uses backward inference to answer questions. The questions may be explicit, as from the user of a computer program, or they may be goals which need to be proven, and were created in some other part of the system.

Backward inference is described in [Black 68] as an alternative to forward inference for question answering. Substituting forward inference for backward inference would require making all possible inferences at read-time, using a large number of rules. Doing this would slow the input part tremendously and hopelessly clutter up the knowledge base with information which might never be used.

Backward inference solves this problem with backward chaining. The goal, the question being asked, is matched against the consequents of rules which the system knows about. Those rules are used to try to derive the answer. For each rule found, the system can use the rule if it can show that the antecedent of the rule holds in the current environment. This requires matching the pattern in the rule's antecedent against the asserted knowledge in the data base. If a rule is found which matches the goal, but is itself in consequent position of another rule, then the superior rule must be shown to hold in the same environment.

Examples of backward inference are given in Figure 3.C, using the same rules as in the forward inference case. Note that backward inference is triggered by queries (questions about consequents), while forward inference is triggered by assertions of antecedents.

```
Rule: (x & y) or w => z

Query: x?
    Result: nothing, x is not in consequent position
            of a rule.
Query: z?
    Result: (assume that w has been asserted previously)
            The system notes that this rule has z in the
            consequent position, and tries to prove that
            its antecedents hold.  It finds the existence
            of w, and can thus conclude that z? is true.


Rule:  ∀x (Student(x) => AttendSchool(x))

Query: AttendSchool(John)
      Result: (assume that the system has already been
              told that John is a student).  A match is
              found between the query and the consequent
              of the rule.  Student(John) is found to
              exist in the system, and the user is told
              that John is indeed a student.

Final result: w and Student(John) are still asserted.
              The results of the deductions are also
              kept, and they may be used again without
              having to make the actual deductions.
```

Figure 3.6
Backward Inference Example

Lazy Evaluation

In [Friedman 76], the authors suggest that certain functions in applicative (side-effect-free) languages should be interpreted in a lazy fashion and should not evaluate their arguments, but instead should "promise to evaluate them at a later time:"

> It is our thesis that the fields of a newly allocated record can be filled with a structure representing the suspended evaluation of the respective argument, instead of the value of that argument, as is done on systems with strict implementation of <u>cons</u>. If all other elementary functions are able to detect these suspensions and to force evaluation only at the time that the value is genuinely critical to the course of the computation (necessary to the value of the main function), then the results are the same as those of a strict evaluation scheme whenever both converge.

By using a lazy evaluator, one need not spend time evaluating functions whose complete results are not necessary. The environments of the evaluations must be kept in the <u>suspension</u> of the evaluation, but the time saved is expected to make up for the extra storage.

Lazy evaluation is output-driven computation, and so actual evaluations are triggered by calls to PRINT functions. (Contrast this with data flow, which is data-driven). If the arguments to the PRINT are not actual values, such as constants or literal values, then the evaluator back-chains until it finds values for all of the variables involved. This process may continue until the evaluator has backed up as far as it can without resolving all of the variable bindings. In that case, one solution is to ask the user for further information, see section 4.3.

Without calls to PRINT functions, a purely lazy evaluator might ignore all of the calculations. Since no values need to be output, there is no need to perform the calculations.

```
                    (CONS A B)
                     //   \\
           Standard //     \\ Suspended
         Evaluation //      \\ Evaluation
                   \/        \\
   ( (Value A) . (Value B) )  \\
                               \/
                 ( (Susp A env) . (Susp B env) )

                   Where (Susp x e) means that
                   a suspension of x in the
                   environment e is stored
                   until it is needed.
```

Figure 3.7
standard vs. lazy evaluation

The standard, or strict, Lisp CONS function takes two arguments, and returns a two-part data structure, with each part pointing to the <u>value</u> of the respective argument. For example, Figure 3.7 shows how a Lisp CONS would be evaluated with the standard evaluator, and with one which returns suspensions instead of actual values. When a strict function accesses the suspension, it can coerce the evaluation.

Suspensions are useful when working with data structures which must be sequentially accessed, like lists in Lisp. The standard (strict) Lisp evaluator evaluates all of the arguments of a function before evaluating the function. In some situations, this is not desirable because of the unnecessary work which will be done.

The following Lisp form shows how using suspensions saves execution time:

```
        (PRINT (CADR (CONS (A)
                           (CONS 'B
                                 (CONS (C) 'D))) ))
```

The CADR function returns the second item in a list, in this case the atom B; the PRINT function prints out the value of its argument. A and C are unspecified functions.

A strict evaluation of the expression would cause all of these functions, including A and C, to be evaluated. If A and C perform complex calculations, then much time will be wasted because their values are not necessary to PRINTing the CADR of the list resulting from the CONS.

If this expression were evaluated by a lazy interpreter, then suspensions would be set up for evaluations of functions A and C. Some work has to be done to CONStruct a list of the suspensions, but this is less than evaluating all of the functions. Once this list of suspensions is created, the CADR of the list can be located, and its value returned, in this case just the atom B.

Effect on the SPC Model

A lazy SPC system eliminates the waste found in a data flow system. Nothing is built until a consumer makes a request for the product. At that time, requests are sent out to the suppliers for the needed parts. As a full complement of supplies comes in, the product is built and sent to the consumer who requested it.

Lazy producers are interested in using their workers's time wisely. A producer who has a near monopoly on a market, or one whose products are of high quality might best use this technique.

3.2. Acquisition of Supplies

The control structures discussed as acquisition methods in this model of computation differ in their use of parallelism and in the amount of unnecessary work that they allow to be done.

Eager Evaluation

According to Baker and Hewitt [Baker 77], an "eager beaver evaluator" is one "which starts evaluating every subexpression as soon as possible, and in parallel." Figure 3.8 shows a representation of how an eager evaluator would handle the sample rule.

Processes called futures are created and assigned to each of the propositions which are then all evaluated simultaneously. This is not strictly a parallel operation, however, because as soon as enough of the operands return the value TRUE (satisfying the antecedent conditions), all of the remaining evaluations are located and terminated. The authors liken futures to ALGOL-60 thunks, which are "simple parameterless subprograms [used] for simulating transmission by name" [Pratt 75].

Programming languages generally do not use eager evaluation. In a Fortran statement (or one in most other compiled languages), <u>all</u> subexpressions would commonly be evaluated, regardless of their results. A Lisp interpreter evaluates a Boolean expression in strict left-right order, but stops as soon as the value of the expression can be determined. Neither Lisp nor Fortran involve any concurrency or saving of results.

```
              Eager evaluation of (x & y) or w => z

     (Future x)          (Future y)        (Future w)
          \                  /                  :
           \                /                   :
            \              /                    :
             \            /                     :
               AND-------------------------OR      wait for FIRST
                        :                            one to finish
                        :
                        :
                        z
```

Figure 3.8
Example of Eager Evaluation

Results of completed evaluations are saved by the future, so the value of that particular process need not be recomputed. For parallels to this, see [McKay 80] for the description of the <u>data-collector</u> processes in MULTI, or [Kaplan 73], where <u>charts</u> are used to save the results of subgrammar evaluations in natural language processing.

Eager evaluation results in savings of execution time but it may also create many irrelevant processes. Irrelevant processes are those which have begun executing in parallel with others, but become extraneous when one of the others finishes. In the example in Figure 3.8, a large savings in overall execution time may be found in the case where x and y return FALSE after a series of long and complex calculations but w quickly evaluates to TRUE. All three of the processes begin evaluating at the same time, but as soon as the future for w finishes, the others become irrelevant. Baker and Hewitt note that without some means of determining irrelevancy, these processes can waste a large amount of processing time. They introduce a garbage-collection scheme to recycle irrelevant processes.

The nodes in the FA/C system of [Lessing 81] (page 10) are eager producers which can share information among themselves to fill in the gaps in their respective data bases, enabling them to compute more complete results. Lessing and Corkill state that nodes in an FA/C system may be "self-directed", in that they can each determine the direction of their work, and the extent of their message-passing based on the state of the rest of the network.

Effect on the SPC Model

Eager SPC producers are concerned with speed of production. Requests are sent out to all of the suppliers. The first part of each type received will be used because getting the product out to the consumer is more important than waiting for the best part. There will always be outstanding part orders unless some of each part is present. Partial productions are begun by completing a certain amount of the work involved, and waiting for an actual order before its completion.

The suppliers are not queried in any special order, nor are there any preferences as to which supplier provides which part. There may be a minimum-quality threshold for acceptance of parts supplied, but even in this case, all replies will not be waited for once a part arrives which is of high enough quality.

Static Evaluation

In static evaluation, evaluation of processes is in a prescribed (static) order, usually the order in which statements are written down in a program. For example, in Figure 3.9, x is evaluated before y, which is evaluated before w. Unlike some of the other methods described, a purely static evaluator will evaluate all of its arguments.

Static evaluation is standard in sequential systems. Its main advantage is that it requires no parallelism at all. Its drawback is that complex evaluations delay simpler ones which might resolve problems much faster.

Effect on the SPC Model

SPC systems which operate in a static evaluation mode are interested in quality of product rather than rate of production. When parts are needed, requests are sent to each supplier, but no action is taken until <u>all</u> of the results are in. Time may be wasted waiting for the slowest supplier, but the philosophy of the static producer is that quality is worth waiting for.

3.3. Combinations

The two supply methods and the two production methods can be combined to yield four combinations. Some of these will model real-world systems in computer science or business.

Static evaluation of (x & y) or w => z

- evaluate x to TRUE or FALSE
- evaluate y to TRUE or FALSE, even if x is FALSE.
- evaluate w to TRUE or FALSE, even if (x & y) is TRUE.
- assign z its value from the above evaluations.

Figure 3.9
Static Evaluation Example

If the production methods are looked at as the analogous inference types (forward inference for data flow and backward inference for lazy evaluation), then we have analogues to implementations for inference systems. Both types of combinations are discussed.

LAZY/EAGER: Nothing is done until a consumer orders a product. When a product is ordered, production proceeds as fast as possible. Necessary parts are requested from suppliers, and the first parts of each type that come in are used. When a new part is delivered, any outstanding orders for that item are stopped.

Baker and Hewitt suggest some degree of lazy evaluation in their scheme to take care of the problem of an eager evaluator getting stuck on an open-ended computation (for instance, computing a list of the squares of all of the integers). This would be used to halt a computation after some specified point, not to be continued until required.

LAZY/STATIC: A lazy/static system may be looked at as a made-to-order manufacturer for very demanding customers. Nothing is done until an order is received from a consumer. At that time, orders are sent to the suppliers, and only the best supplies are used. In this manner, only the best and the "freshest" products are made.

Both of the combinations involving lazy evaluation may be viewed as involving backward inference. The difference is similar to that between performing inference in a multiprocessing system and in a sequential one. Using an eager acquisition method allows more than one inference process to begin at the same time and/or the same inference may be run simultaneously on different sets of assertions. Using static evaluation would restrict the system to one inference and one assertion at a time.

DATA FLOW/STATIC: This is not the normal data flow operation because the data flow aspect of the operation is constrained by the need to wait for all of the suppliers of any part.

When all of the requested parts have arrived, the producer must spend time deciding which of each type is the best. As soon as all of the parts have been decided on, production begins. It terminates when the one product is finished. The check on the quality of the supplies holds up production.

Combining data flow and static evaluation seems to be a contradiction of sorts. The data flow part tries to produce as much and as fast as possible, while the static half puts a brake on the throughput by requiring an evaluation of all of the inputs. The data flow concept says that as soon as all data are present for any actor, it can fire. Static evaluation, however, requires that expressions be evaluated in a predetermined order. The result is the acquisition of supplies in a static order (one at a time), with production not begun until all are present.

DATA FLOW/EAGER: This is the standard data flow procedure. (See [Dennis 79b] for a description of the data flow language VAL). The SPC system is always prepared to produce. All available parts are used and orders for replacements are sent as soon as the supply is used up. Only the first of each type of product will ever be accepted.

There may be excess product which is never used. That does not concern the system, because the goal here is to produce.

The distinction between the data flow pairs when forward inference is substituted for data flow is similar to the distinction between the lazy evaluation pairs with backward inference substituted in. It is the difference between forward inference in multiprocessing and sequential environments.

4. BI-DIRECTIONAL COMPUTATION

All of the computation schemes discussed thus far have been uni-directional. In terms of the SPC model this means that production of a specific product might have been triggered by an order (representing demand-driven computation) _or_ by the presence of parts (data-driven). No combinations of forward and backward production were discussed.

A combination of forward and backward schemes can be called _bi-directional_. This class of computations includes combinations of inference, search and computation methods. Benefits of these combinations include savings of space and of the actual number of computations performed because each component can direct the other.

Dynamic demons will be introduced as a description of suspended processes. These are different than the static demons described in [Charniak 72] and above.

4.1. Bi-directional Search

Problems which are solved by incremental steps may be thought of as occupying a _space_ of _states_, as shown in Figure 4.1. A state represents one of the incremental steps, and one state may have a number of successor states. The state-space is the collection of all of those states. A _solution_ to the problem is represented by a path through the space, connecting the states between an initial and a goal state. Search is the process of finding that path. There may be many possible solutions to a problem, but only one path is required to solve it. There are many ways of attacking search. [Nilsson 80] is a good source of some of them.

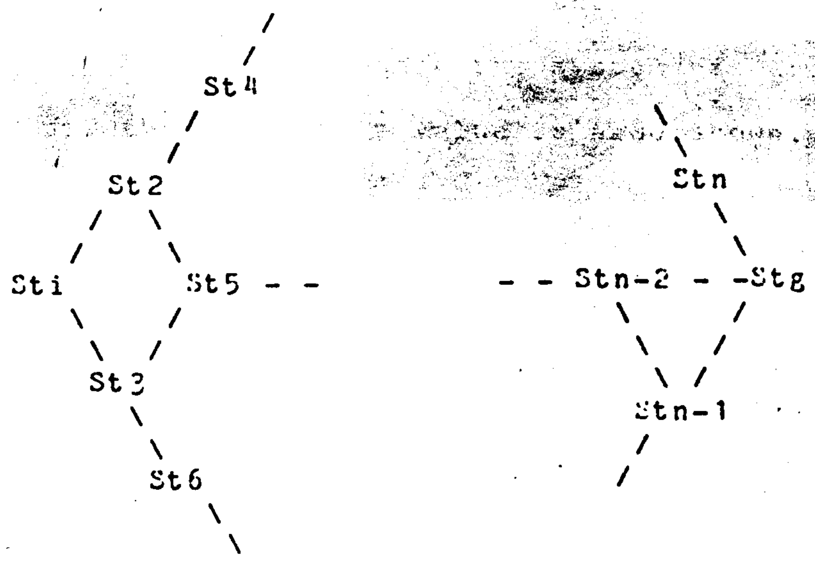


Figure 4.1
The State Space Representation of a Problem

Two standard search techniques are forward and backward search. In forward search, the procedure begins at some initial state, for example Sti (in Figure 4.1) and searches for a path to the goal state, Stg. A backward search would begin with the goal and work toward the initial state. Which direction is selected depends on the application. With both situations, the trick lies in having an algorithm (or heuristic) for selecting a next state to look at.

Bi-directional search is a combination of forward and backward search techniques. The search terminates when the two search trees intersect, indicating a complete path from the initial state to the goal state. As Pohl says,

> Rather than two independent searches, it can be advantageous to combine the two searches into a bi-directional search with each contributing part of the solution. The motivation is that search trees grow exponentially and two shorter search diameters generate fewer states than a single longer diameter tree. [Pohl 71, p 128]

Pohl gives a Bi-directional Shortest Path Algorithm (BSPA) and a Bi-directional Heuristic Path Algorithm (BHPA), while DeChampeaux and Sint [DeChampeaux 77] give an improved heuristic for the BHPA. The problem of selecting the next state is somewhat more complex in that there is a search tree growing from the initial state and one from the goal state. A number of solutions to this problem exist, including alternation of directions [Nicholson 66]. Pohl rejects that in favor of the cardinality comparison principle [Pohl 69], which compares the sizes of the sets of not-yet visited nodes reachable by one edge from the forward component and from the backward component. The direction providing the fewest candidates is chosen as the most promising. Once a direction is selected, the node which satisfies the particular test being used for closeness is selected.

Bi-directional search, like the other techniques presented here, benefits from a multiprocessing environment.

Parallel processes can be used to choose directions and to select nodes.

4.2. Bi-directional Inference

When forward and backward inference techniques are combined, Martins et al. call the result bi-directional inference [Martins 81]. By allowing inference to proceed in either direction, one can "focus the system's attention towards the interests of the user and can cut down the fan out of full forward or full backward chaining."

Specific modes for forward or backward inference are not necessary, because each input is used where it applies. Backward inference is initiated when a user asks a question. Forward inference may be initiated when the user adds new information. The forward inference component need not operate as full forward inference, but only enough to assist the backward part. This fits in with Wilks' "laziness hypothesis" [Wilks 75], and the idea of lazy evaluation. Forward inferences may be made only if they are directed at a known goal, not simply because a usable rule exists. If no goal exists, any rule may be used. This is not the same as restricted forward inference because the system is not necessary prevented from using a newly asserted proposition to trigger a new forward inference.

Forward inference demons may be created by backward inferences which cannot prove a sufficient number of antecedents. As soon as the missing propositions are asserted, the demons act. Alternatively, some of the expected queries can be made before any data has been entered. In the SNePS system, this is known as "pre-compiling", and is used to set up the inference processes at the start. It works because the processes are saved, unless explicitly cleared by the user, and because all rules can be used for either forward or backward inference. Another cause of forward demons is an inference procedure exhausting its allocation of resources. Until it is assigned more, and reactivated, the inference remains as a demon.

Some deductions may not be able to complete. Causes of this are the use of resource-limited processing, or that some of the required propositions are not available. In these cases, three options exist:

* Quit. If an inference cannot complete, the processes involved can be located and destroyed. When the needed propositions are found, the inference can be started again, with a new set of processes. Partial results and suspended processes are not saved or kept track of. The problem is that the same rule might be triggered at a later time, and partially execute to the same degree many times, never completing.

* Suspend. The process(es) involved in the inference can be located and suspended. The evaluation itself is then in a state of suspended animation and

has become a special type of suspension called a demon. As described in Section 2.2, the demon watches for its particular trigger pattern (the missing propositions) to be present in the data base and then acts. In this case, the demon has been dynamically created, as opposed to the demons discussed in [Charniak 72]. The demon can resume where it left off without having to re-do work which has already been completed. A data flow actor which is missing some of its required actors, but which has others present, is in a similar situation. When its missing inputs are present, it will fire, until then it is in a suspended state.

* Ask. The program can ask the user for the needed propositions. By doing so, the interaction, as well as the computation, becomes bi-directional. All missing items need not trigger extended question-answer sessions. Certain rules can be identified as having higher priorities than others and be able to stop the program's execution to query the user if they cannot complete. Alternatively, point-values can be assigned for each proposition involved in a deduction rule. When the cumulative total of missing items passes a specified threshold, a question can be asked. Processes which cannot continue, but do not trigger questions remain as demons.

When the backward aspect is stopped due to having exhausted its complement of resources, the processes involved are suspended as demons, ready to continue when more resources are allocated. Another way to hold backward computation back is for the user to refuse to answer a request made by the program. The involved processes are again suspended. An example of the latter situation is Shapiro's COCCI program, which is a use of his SNePS system

---

Rules: 1. Student(a) => AttendSchool(a)
2. (x & y) or w => z, where z represents Student(a), and x, y and w are some conditions which establish that one is a student.

Input: AttendSchool(John)? [backward aspect]

Result: Rule 1 is seen to have the goal in its consequent position and is selected for backward inference. To use this rule, Student(John) must be proved, but the system has no values for x, y or w. The inference is suspended as a demon.

Input: x [forward aspect]

Result: If y can be shown to hold, (possibly by performing a backward inference), then rule 2 can be used to assert Student(John), which can be used by the suspended rule 1 to conclude AttendSchool(John), and the system can report this to the user.
The system initiates a request for a value for y. If none is available, the inference is suspended.

Figure 4.2
Bi-directional Inference Example

---

as an expert in the domain of microbiology [Shapiro 1981]. A user of COCCI has the option of replying "Not applicable or unknown" to any request made by the program. As stated above, these demons work because of the assumptions made in section 2.1: Inference rules can be used in either direction; and processes and partial evaluations are retained.

A simple example of bi-directional inference is shown in Figure 4.2, using both rules from Figure 2.1. Substitutions are made in the first rule to allow it to be used with the second rule. A case where the inference is suspended is shown, and a case where the system has enough information to ask the user for a value is also shown.

4.3. Bi-directional Computation

Combining data flow and lazy evaluation yields a bi-directional computation scheme analogous to bi-directional inference. Again, there is no need for the system to have an algorithm to decide which component to use at any time, because the direction taken depends on what prompted the action -- a READ function, or a PRINT function.

As described above, lazy evaluation, the backward component, is output-driven. Bi-directional computations may therefore be initiated by the need to evaluate arguments of PRINT functions.

The forward component, data flow, is data-driven. If a program contains calls to READ functions, forward computation is initiated. As a value enters the system, it is used where it is appropriate, as long as a particular use is directed toward some previously set goal. If there are no existing goals, then the value can be used by any actor that needs it.

What happens if all tokens but one are present for a given actor operating in either a forward or backward mode? Why do computations exist as suspensions if the system has a data flow component? These questions are answered by looking at possible actions in bi-directional computation, which are similar to the options presented for bi-directional inference in Section 4.2. A process which finds a missing input has three choices as to how to continue:

* Quit. The processes involved in an inference which did not complete are destroyed. None of the partial results are saved. When the rule is triggered again, new processes are set up for the inference, but the same work may be done repeatedly.

* Suspend. The processes involved in the inference are suspended, and become demons. These dynamic demons are different from those of [Charniak 72] because they are created and destroyed as the program executes. When the missing propositions are found, the demon acts, and the deduction resumes where it left off.

* Ask. The program can ask the user for the missing information. In this case, the interaction, as

well as the inference, becomes bi-directional. As discussed in section 4.2., not all of the missing values need trigger questions of the user, only the more important ones. This interaction steps out of the normal bounds of data-driven programs, in that the program is actively acquiring information, not passively accepting it.

A node in an FA/C system [Lessing 81] which is missing information that it needs to complete a computation has choices similar to those presented in section 4.2: it can request the information (in this case from another node), it can suspend itself if no information is forthcoming, or it can begin a new computation.

An example of bi-directional computation is shown in Figure 4.3. Three possibilities were presented above for the situation where a calculation could not complete: Quit, Suspend, or Ask. In the figure, the latter two choices are shown. The computation is suspended when its result is requested and no inputs are present. A request of the user for more input values is made when two of the three inputs are present.

Each component is kept from taking complete control by including resource-limited processing (see section 2.3), which allows short computations to finish, but causes longer ones to be executed in bursts. When a computation exhausts the resources allocated to it (for instance, time or number of calculations), it it deactivated. When more resources

---

Equation: $z = (x * y) + w$

Input: request for value of z. Request may be from user or another part of the system. This is the backward component.

Result: Not enough data is present to ask the user for information to complete the computation. It is suspended, making it a demon.

Input: values for x, y. Again, this can be from the user or the program. It represents the forward component.

Result: There is almost enough data present to complete the calculation, so the program can request a value for w from the user, and complete the evaluation. If the user cannot supply a value, the computation remains suspended.

Figure 4.3
Bi-directional Computation Example

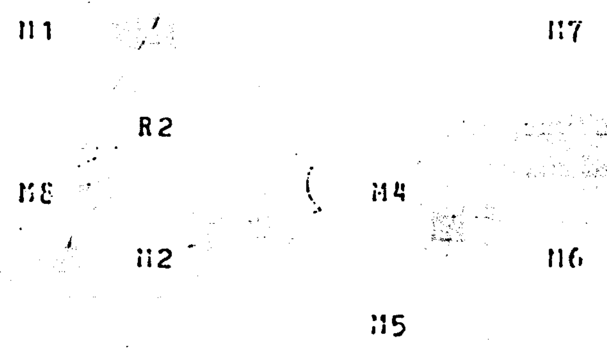
---

are provided for it, it may continue executing from the point it left off. Its environment will have been saved in the suspended processes.

4.4. Contrasts

Any of the bi-directional actions can be represented by a network of nodes, which is referred to as a state-space graph. (Figure 4.4, for example, shows an AND-OR graph representing a bi-directional inference example used in

[Martins 81]). Nodes represent propositions and rules if the graph is for bi-directional inference. The nodes in a network for bi-directional search or computation represent intermediate steps.



Nodes labeled "Mx" represent proposition-x, those labeled "Rx" represent rule-x.

Figure 4.4
An AND-OR Graph for Martins's Bi-directional Search

Because each of the bi-directional strategies can be thought of as seeking a path through a network, they all can be thought of as search.

Bi-directional search is the most restricted of the three techniques. The initial and goal states must be explicitly known when the search begins [DeChampeaux 77, Pohl 71]. The search is successful when the two search paths intersect.

Multiple start and goal states are possible in a bi-directional inference, representing nodes asserted or queried by forward or backward inference, respectively. They do not represent separate searches through the network. Rather, they are dynamic sets of initial and goal nodes used instead of the single states used in classical bi-directional search. As [Martins 81] reports, "The search is successful whenever the frontier growing from any start state meets the frontier growing from a goal state." New inferences can then be made reflecting the situation created by the previous one.

Multiple initial and goal states may also exist in bi-directional computation. Because backward computation is output-driven, any call to a PRINT function initiates backward search and becomes one of the goal nodes. Similarly, calls to functions which take input represent initial states.

The search through the state of nodes representing a bi-directional computation is not satisfied when the search tree of one goal meets the search tree of one initial node, as it is in the inference case. Because each goal state represents an output to be printed, the search path from each must be completed so that the value can be printed.

The search tree of an initial state meeting a goal indicates that the data input represented by that state has found a use. The data flow nature of forward computation implies that it is not necessary for all data or results to have further use. This suggests that forward search trees need not connect with backward search trees. However, the assumption that the forward component be restricted to focus on goals set by the backward component (if goals have been set) requires that a forward search not be undertaken unless it is directed at the tree of a backward search. If no goals exist, all possible forward searches are allowed.

Effect on the SPC Model

As discussed thus far, production in the SPC model is uni-drectional, that is, it can be triggered either by the presence of data or by a request from a consumer for a product. By allowing the producer to act in a multiprocessing mode, it can incorporate the ideas of bi-directional computation. (Remember that all of the components of the system act as producers at some point).

Look at the activities of a bi-directional producer. An inactive SPC system can be started up by an order from a consumer. This request represents backward computation. The order triggers requests to the suppliers for the appropriate parts. Those parts might be delivered immediately, or the suppliers might have to go through this same process. The requests for supplies will back-chain until a supplier which has the parts on hand is reached.

A producer who receives some supplies because those suppliers are acting in the forward direction might then explicitly request the rest of the parts it needs. Its other suppliers would then be demand driven.

The system's activities can also be initiated by input of supplies to an "original" producer, that is, one which has no suppliers inside the system. These supplies can be applied to any and all productions in the absence of requests for parts. If goals have been set, then the forward production is directed toward them.

Suspended production processes are created by limiting the resources allowed each producer, or by required parts which cannot be obtained. These suspensions, or demons, can be reactivated at a later time.

The original producer can initiate production processes for any of the sub-products whose parts have arrived, and for any number of requests for products. All of these processes can be active simultaneously, using a resource-limited processing scheme based on time, number of workers, amount of supplies, etc.

Bi-directional production, like bi-directional computation, allows a system to be more sensitive to a user's needs. By using multiprocessing, a producer can deal with more than one consumer and more than one supplier at a time, and requests for supplies can be focussed toward the product requests which have been made.

5. SUMMARY

Pairs of control and inference strategies, representing forward, backward and bi-directional techniques were discussed. The inference and control schemes in each pair are manifestations of the same idea.

Forward inference and data flow make up the forward computation, or data-entry group, which acts to use all new data immediately and to the greatest extent possible. The backward computation, or data-request group is represented by backward inference and lazy evaluation. These techniques wait for a request before performing evaluations. Bi-directional actions include bi-directional search, and bi-directional inference, all of which combine forward and backward operations to reduce the number of calculations required.

Two other control strategies, static and eager evaluation, were introduced to add another dimension to the comparisons. These differ in two ways: eager evaluation involves parallelism and avoids unnecessary computations, while static evaluation involves no parallelism at all and will perform some unnecessary work.

In order to study these techniques, the Supplier-Producer-Consumer (SPC) model of computation was introduced. The model uses production to represent data-entry and data-request computations, and acquisition to represent eager and static evaluation.

Each technique and its affect on the model was discussed. Combinations of production and acquisition were described to show how the overall operation of a producer representing a computation is affected. The result of combining the respective control and inference techniques was discussed using the combinations in the SPC model.

Bi-directional computation was introduced to show how forward and backward computation schemes could be combined to enable them to work together and cut down the size of the search space of a computation. Among the devices suggested for use in bi-directional computation were multiprocessing and resource-limited processing, to allow a breadth-first search through the search space; demons, to handle evaluations which could not complete; and having the program query the user for missing information.

6. REFERENCES

1. Arvind, K.P. Gostelow, W. Plouffe (1976), "Programming in a Viable Data Flow Language", University of California at Irvine, TR 89.

2. Baker, H.G. and C. Hewitt (1977) "The Incremental Garbage Collection of Processes," in combined SIGPLAN 12, 8; SIGART 64.

3. Black, F. (1968), "A Deductive Question-Answering System," in Semantic Information Processing, M. Minsky (ed), MIT Press, Cambridge, MA.

4. Bobrow, D. G., Raphael, B. (1974) "New Programming Languages for Artificial Intelligence Research," Computing Surveys, 6, 3.

5. Charniak, E. (1976a), "Inference and Knowledge I," in Computational Semantics, E. Charniak and Y. Wilks (eds), North-Holland Publishing Co., pp 1-22.

6. Charniak, E. (1976b), "Inference and Knowledge II" in Computational Semantics, E. Charniak and Y. Wilks (eds), North-Holland Publishing Co., pp 129-154.

7. Charniak, E. (1972), "Toward a Model of Children's Story Comprehension," PhD dissertation, MIT AI Laboratory TR-266.

8. Davis, R. and J. King (1975), "An Overview of Production Systems," Stanford University AI Memo 271.

9. Dennis, J.B. (1979a), "The Varieties of Data Flow Computers", "Proceedings of the First International Conference on Distrubutive Computer Systems", IEEE.

10. Dennis, J.B. (1979b), "VAL -- A Value-oriented Algorithmic Language, Preliminary Reference Manual", MIT LCS TR 218.

11. DeChampeaux, D. and L. Sint (1977), "An Improved Bidirectional Heuristic Search Algorithm," Journal of the ACM, 24, 2.

12. Friedman, D.P. and D.S. Wise (1976), "Cons Should Not Evaluate its Arguments," in S. Michaelson and R. Milner (eds), Automata, Languages and Programming, Edinburgh University Press, Edinburgh, pp 257-284.

13. Hayes, J.P. (1978), Computer Architecture and Organization, McGraw-Hill, Inc.

14. Hewitt, C. (1971) "Procedural Embedding of Knowledge in PLANNER", Proceedings of the Second International Joint Conference on AI, p 167-182.

15. Hewitt, C. (1969), "PLANNER: A Language for Proving Theorems in Robots," Proceedings of the First International Joint Conference on Artificial Intelligence. pp 295 - 301.

16. Jargon (1979), file GLS;JARGON, on-line INFOrmation file, on MACSYMA at MIT.

17. Kaplan, R.M. (1973), "A Multi-processing Approach to Natural Language," Proceedings of the National Computer Conference, 1973, pp 435-440.

18. Lessing, V.R. and Corkill, D.D. (1981), "Functionally accurate, cooperative distributed systems",IEEE Transactions on Systems, Man And Cybernetics, SMC-11, 1, pp 81-96.

19. Leung, C.K.C. (1975), "Formal Properties of Well-formed Data Flow Schemas", MAC TM66, MIT.

20. Martins, J., D.P. McKay, S.C. Shapiro (1981), "Bi-directional Inference," State University of New York at Buffalo, Technical Report 174.

21. McGraw, J.R. (1979), "Data Flow Computing: Software Development," Proceedings of the First International Conference on Distributive Computer Systems, IEEE.

22. McKay, D. and S.C. Shapiro (1980), "MULTI, a LISP Based Multiprocessing System," in Conference Record of the 1980 LISP Conference, pp 29-37.

23. Nicholson, T. (1966), "Finding the Shortest Route Between Two Points in a Network," Computer Journal, 9, pp 275-80.

24. Nilsson, N. (1980), Principles of Artificial Intelligence, Tioga Publishing Co.

25. Pohl, I. (1971), "Bi-directional Search," in Machine Intelligence, B. Meltzer, D. Michie (eds), American Elsevier Publishing Co. Inc., pp 127-140.

26. Pohl, I. (1969), "Bi-directional and Heuristic Search in Path Problems," SLAC report No. 104, Stanford CA.

27. Pratt, T. (1975), Programming Languages: Design and Implementation, Prentice-Hall, Inc., 1975.

28. Reiger, C. and Small, S. (1981), "Toward a theory of distributed word expert natural language parsing", IEEE Transactions on Systems, Man And Cybernetics, SMC-11, 1, pp 43 -- 51.

29. Shapiro, S.C. (1981a), "COCCI: A Deductive Semantic Network Program for Solving Microbiology Unknowns", SUNY Buffalo TR 173.

30. Shapiro, S.C. and The SNePS Implementation Group (1981b), "SNePS User's Manual", SUNY at Buffalo.

31. Shapiro, S. C., (1979), "The SNePS Semantic Network Processing System," in Associative Networks, N. V. Findler (ed), Academic Press, Inc., pp 179-203.

32. Sharp, J. (1980), "Data Oriented Program Design," SIGPLAN, 15, 9, pp 44-57.

33. Smith,R.G. and Davis, R. (1981), "Frameworks for cooperation in distributed problem solving", IEEE Transactions on Systems, Man And Cybernetics, SMC-11, 1, pp 61 -- 70.

34. Stallman, R.M, and G.J. Sussman (1977), "Problem Solving About Electrical Systems," Artificial Intelligence, 9, 2, pp 135-196.

35. Steels, L. (1980), "The Constraint Machine," AI Memo No. 1, Schlumberger-Doll Research, Ridgefield, CT.

36. Steels, L. (1979), "Reasoning Modeled as a Society of Communicating Experts," MIT AI Laboratory TR-542.

37. Sussman, G.J., T. Winograd and E. Charniak (1971), "Micro-PLANNER Reference Manual," MIT AI Report 203A.

38. Waterman, D.A., and F. Hayes-Roth (1978), Pattern-Directed Inference Systems, Academic Press.

39. Weng, K.-S., (1975),"Stream-oriented Computation in Recursive Data Flow Schemas," MAC Technical Memo No. 68, MIT.

40. Wilks, Y. (1976) "Parsing English II," in Computational Semantics, E. Charniak and Y. Wilks (eds), North-Holland Publishing Co., pp 155-184.

41. Wilks, Y. (1975),"A Preferential, Pattern-seeking Sematics for Natural Language Inference," Artificial Intelligence, 6, pp 53-74.