Recent developments in SAIL—An ALGOL-based language for artificial intelligence

by J. A. FELDMAN, J. R. LOW, D. C. SWINEHART and R. H. TAYLOR

Stanford University
Stanford, California

INTRODUCTION

Progress in Artificial Intelligence has traditionally been accompanied by advances in special purpose programming techniques and languages. Virtually all of this development has been concentrated in languages and systems oriented to list processing. As the efforts of Artificial Intelligence researchers began to turn from purely symbolic problems toward interaction with the real world, certain features of algebraic languages became desirable. There were several attempts (notably LISP2 and FORMULA ALGOL) to combine the best features of both kinds of language. At the same time, designers of algebraic languages began to include features for non-numerical computation. No new general purpose language without some sort of list processing facility has been suggested for several years. We have followed a tack somewhat different from either of these in the design of SAIL and in its subsequent modifications.

The starting point for the development of SAIL was the recognized need for a language incorporating symbolic and algebraic capabilities, primarily for Hand-Eye research. The problems are somewhat similar to those in Computer Graphics and one of us had just developed a language, LEAP,[4] for such applications. After an attempt to honestly evaluate alternative techniques, we decided that the associative processing features of LEAP were the way to go. There are important differences between LEAP and the first SAIL (primarily in input-output, string manipulation, and implementation), but these differences are not relevant here. It is essentially this system for the PDP-10 which is distributed by DECUS and is being used for Artificial Intelligence and other research in a number of laboratories.

This original SAIL met our needs for about two years before requiring serious change. Then we began to face the problem of putting together a hand-eye system which was much bigger than the available main memory and which did not lend itself to a static overlay structure. Our solution involves a number of language additions which facilitate the treatment of jobs under the time-sharing system as a set of cooperating sequential processes, and has been described in Reference 5. The three main additions were: a monitor for user control and debugging, a shared data facility, and the introduction of message procedures. The shared data facility makes use of the second relocation register of the PDP-10 to allow jobs to access a common global data area in a natural and efficient manner. The message procedures are the main mechanism for asynchronous communication and control between jobs. A message procedure is a procedure in one job which can be invoked from another job. Control information associated with the invocation can provide the effect of subroutines, coroutines, parallel processes, events, and a variety of other disciplines. These multitasking modifications to SAIL have enabled researchers to assemble and modify large collections of jobs with a minimum amount of attention to system problems.

A number of factors have combined recently to cause us to make a second set of major modifications to SAIL. The multi-tasking facilities of the second SAIL were seen to be at least as useful within a single job as they were across jobs. In addition, the ability to assemble large collections of routines brought us to the point of facing one of the core problems of Artificial Intelligence—what is the right sequence of actions for carrying out a given task in a particular environment. This strategy problem is currently very popular and is the driving force behind much of the recent development in languages for Artificial Intelligence. Our view of the problem is somewhat unorthodox and merits some discussion.

Problem solving for an entity which deals with the

real world is fraught with uncertainty. The state of the world cannot be assumed to be known—in fact, one of the main goals of a strategy must be to gain enough information to carry out the task. An additional problem arises in resource allocation; even if an exhaustive search of the environment will yield a solution, it may not do so at an acceptable cost. Considerations of this sort cause us to view the strategy problem as inherently involving numerical estimates of probabilities, costs, etc. A complete discussion of these issues is beyond the scope of this paper, but the recent SAIL modifications have been influenced by our model of the strategy problem.

Our recent language work has been intended to facilitate the design of programs for the construction and execution of strategies for interaction with the real world. The facilities are being applied to other problems, but we will concentrate on the original theme. However, the language design effort was concerned with expanding the power of SAIL as a general purpose language as opposed to developing a special purpose system. One critical design constraint was that the features not entail large hidden overheads or appreciably degrade the performance of programs not making use of them. We believe we have found a set of features which meet our design goals. The major additions are: backtracking, procedure variables, matching procedures, and a general multi-tasking facility.

STATE SAVING AND BACKUP

In order to try several different alternative strategies it is often necessary to save the current state of the computation. Thus, if the first attempt does not succeed, we may "back up" and try one of the other alternatives. We may also switch between alternatives, continuing with one only until it no longer seems the most promising, but retaining the option of resuming it later if the other alternatives do not prove to be satisfactory. Another technique used in programming non-deterministic algorithms, parallel processes, will be discussed later in this paper.

In general the state of a SAIL computation includes the current control environment, the input and output which have been requested, the contents of the LEAP associative store and the contents of all variables. New SAIL has features which will help handle the last of these components: the contents of variables.

We normally do not want to have the values of all variables "backed-up" when we switch between alternatives. One reason is that it is often useful for one alternative to communicate certain pieces of information it has acquired to the other alternatives. This information is usually saved in certain variables. If we back-up those variables, we lose the information. Another reason for not backing-up all variables is that often only a small subset will have meaning for more than a single alternative, and it is very costly to back-up large amounts of data which may not be relevant for the other alternatives. Therefore we have implemented ways of saving the values of specific variables and then restoring them at a later time.

The state-saving mechanism is based on two new statements: REMEMBER, and RESTORE. Each of these operate on a new SAIL data-type called a "context." A context consists of a set of references to variables and their values.

We save the contents of variables by means of REMEMBER statements,

```
REMEMBER (i, j, a[3]) IN context1:
```

This statement would save the values of "i," "j," "a[3]" in the context named "context1." If any of these variables had been previously saved in "context1," the old values would be lost.

An alternate form of the REMEMBER statement is:

```
REMEMBER ALL IN context1;
```

The current value of each variable which has been remembered in "context1" would replace the value that was previously stored there.

The RESTORE statement also has two forms. The first has an argument-list,

```
RESTORE (j, a[3]) FROM context1;
```

This would search context1 for the arguments and give an error indication if any were not "remembered" within that context. The values saved for those arguments "remembered," would be restored to the appropriate variables.

The other form of the RESTORE statement is:

```
RESTORE ALL FROM context1;
```

This would restore the contents of all variables saved within the named context.

These new features seem to provide the most important features of state-saving without the large overhead imposed by automatic back-up of the entire state or incremental state-saving as implemented in some other programming systems.

LEAP

SAIL contains an associative data system called LEAP which is used for symbolic computations. LEAP

is a combination of syntax and runtime subroutines for handling items, sets of items and associations.

An item is similar to a LISP atom. Items may be declared or obtained during execution from a pool of items by using the function NEW. Items may be stored in variables (itemvars), be members of sets, be elements of lists, or be associated together to form triples (associations) within the associative store.

A set is an unordered collection of distinct items. Items may be inserted into set variables by "PUT" statements and removed from set variables by "REMOVE" statements. Set expressions may also be assigned to set variables. The simplest set expression is of the form:

{item1, item2, item3 . . .}

which represents the set consisting of the denoted items. More complicated set expressions involving set functions, set union, subtraction and intersection are also provided. Sets are stored in a canonical internal form which allows us to carry out such operations as intersection, union and comparison in a time proportional to the lengths of the sets involved.

Sets are deficient in some applications, though, because they are unordered. Thus we could not easily try different alternatives in order of their expected utility. To remedy this, as well as provide a mechanism for creation of parameter lists to interpretively called procedures (see PROCEDURE VARIABLES below), SAIL now contains a data-type called "list." A list is simply an ordered sequence of items. An item may appear more than once within a list. List operations include inserting and removing specific items from a list variable by indexed PUT and REMOVE statements. List variables may also be assigned list expressions, the simplest of which is of the form;

{{item1, item2, item3 . . .}}

which represents the explicit sequence of denoted items. Other list expressions include list functions, concatenation, and sublists.

Triples are ordered three-tuples of items, and may themselves be considered items and occur in subsequent associations. They are added to the associative store by executing MAKE statements. For example:

MAKE use ⊗ plan1 ≡ task1;

The three item components of an association are refered to as the "attribute," the "object," and the "value," respectively. Associations may be removed from the store by using ERASE statements such as:

ERASE use ⊗ plan1 ≡ ANY;

Each item other than those representing associations may have a DATUM which is a scalar or array of any SAIL data-type. The data-type of a DATUM may be checked during execution. DATUMs are used much as variables. For example:

DATUM(it) ← 5;

would cause the datum of the item "it" to be replaced with "5."

SAIL contains a compile-time macro facility which allows such things as string substitution and conditional compilation. As is the custom of many SAIL programmers, we will use the macro "∂" to stand for the string "DATUM." Thus the above example would appear as:

∂(it) ← 5;

PROCEDURE VARIABLES

It is quite natural in an interpreter to allow for the execution of program generated sequences of actions. This is an important feature for artificial intelligence applications and is not easily made available for compiled programs. In new SAIL, the generation of such sequences is facilitated by a procedure variable mechanism which fits in quite nicely with the associative search features of the language. These procedure variables are created at runtime from items by statements of the form

ASSIGN(⟨item expression⟩, ⟨procedure specification⟩)

where

⟨procedure specification⟩ ::= ⟨procedure id⟩ |

DATUM (⟨procedure item expression⟩)

For instance,

ASSIGN(xxx, baz)

would cause the datum of item xxx to contain a description of baz, together with a pointer to baz's current environment. Similarly, the statement

ASSIGN(yyy, ∂(xxx))

would cause yyy to be made into a procedure item containing the same information as that in xxx.

In addition to dynamically specifying what procedure to execute, one would also like a convenient way to dynamically specify an argument list for a procedure call. This facility is provided by the APPLY mechanism:

APPLY(⟨procedure specification⟩, ⟨argument list⟩)

where ⟨argument list⟩ is any SAIL list and may be omitted if the procedure has no parameters. For example,

```
APPLY(foo)
APPLY(∂(xxx), list1)
APPLY(∂(APPLY(yyy)), {{x, y, z}})
```

APPLY uses the items in the argument list, together with the environment information from the procedure item (or from the current environment, if the procedure is named explicitly) to make the appropriate procedure call. If the called procedure produces a value, that value will be returned as the value of APPLY.

Procedure items permit a great deal of flexibility. For instance, the user can say things like

```
FOREACH x | xϵactions ∧ use⊗x≡fastening do
  BEGIN
  APPLY(∂(x), {{board1, board2}});
  IF together(board1, board2) THEN
  GO TO done_it;
  END;
done_it:
```

This would search the set "actions" for any procedures which have been asserted to be useful for fastening things together until either the list is exhausted or the task is successfully completed.

MULTIPLE PROCESSES

The control structure of SAIL was originally very much like that of Algol 60—that is to say block structured and procedure oriented. Although this structure is adequate for many problems, there are some cases in which it is uncomfortably restrictive. In hand-eye applications, for instance, there are frequently modules of code which are more or less mutually independent but that wish to call on each other for various services. Similarly, one may wish to investigate several possible strategies at once, with the results of one computation perhaps influencing the course of others. In such cases, it is much more natural to think of (and write) these modules as coroutines or independent processes rather than as nested procedure calls. To some extent, message procedures provided the desired facilities, with each job acting as a separate process. This solution has some rather severe drawbacks, since the overhead involved in switching control from process to process and in interprocess communication is so high that close interaction becomes prohibitively expensive. One of our goals in providing new control facilities was to make possible the close cooperation of many small-to medium-sized processes within a single job without imposing an excessive overhead either on old-style procedural programs or on users of the shiny new features. In doing this, we wanted to retain the block structure rules of Algol, since these rules are generally familiar to programmers and provide a useful means of determining which data is to be shared.

The implementation we have chosen somewhat resembles the mechanism described by Organick & Cleary[8] for the Burroughs B6700. In SAIL, a process is essentially a procedure activation which has been given its own run time stack and which thus does not have to return before the process that invoked it can continue. SAIL procedures normally make up-level references via a "static" (lexical nesting) chain maintained for that purpose in the stack. When a procedure is to be called as an independent process, a "process" routine first gets space for a new stack. It then sets up appropriate process control variables in the new stack area and in the "parent." Finally, the procedure is invoked using the new stack. When this procedure is entered, it will set up its static link by looking back along the static chain of the calling process until it finds an activation of its lexical parent. Thus, different processes will share data belonging to their common ancestors.

Many of the applications which we have considered do not permit us to predict just how many subprocesses a process might wish to spawn or require that several processes be instantiated using the same procedure on different data. Therefore, we have chosen to "name" processes by assigning them to LEAP items, rather than by using procedure names or some special data type called "process." This approach has the added advantage of allowing complex structures of processes to be built up using the mechanisms of LEAP. New processes are created by statements of the form:

```
SPROUT(⟨item expression⟩, ⟨procedure call⟩,
  ⟨options⟩)
```

where the item specified by ⟨item expression⟩ is to be used as the process name, the ⟨procedure call⟩ tells what this process is to do, and ⟨options⟩ is an integer which is used to specify how certain process attributes are to be set up. (If the ⟨options⟩ parameter is omitted or only partially specified, SAIL will provide default values). For instance, a procedure to nail two boards

together might contain a sequence like

```
:
ITEM p1, p2, p3;
:
:
SPROUT (p1, grab (hand1, hammer));
SPROUT (p2, grab (hand2, nail));
SPROUT (p3, lookat (tv1, boards));
:
:
JOIN ({p1, p2, p3});
pound (hammer, nail, boards);
:
:
```

In this case, grab(hand1, hammer) would be executed as process p1, grab(hand2, nail) would be executed as process p2, and lookat(tv1, boards) would be executed as process p3. The process creating them continues on its way down to the JOIN statement. In general,

JOIN(⟨set⟩)

causes the process executing it to be suspended until all the processes named by the ⟨set⟩ have terminated. Thus pound (hammer, nail, boards) will not be called until p1, p2, and p3 have all terminated. In our example, both SPROUTed processes and the original process would theoretically run in parallel. In fact, this is not possible with a single processor. Instead, the SAIL runtime system includes a scheduler that decides which process is to be executed at any given instant. Each process is given a priority and time quantum and may be in one of four states: "running," "ready" (i.e., runnable), "suspended," or "terminated." The scheduler, which is invoked either by a clock interrupt or by an explicit call by the user, uses a simple round robin algorithm to distribute service among the highest priority ready processes.

When a process is SPROUTed, the system assigns it a standard default priority and time quantum, unless the user specifies otherwise by appropriate options. The SPROUTed process usually becomes the running process, while the SPROUTing process reverts to ready status, unless some other option is specified. For instance, suppose we have some procedure "wander" which searches a data base or the real world at random for potentially useful objects. Then we might write something like:

```
SPROUT(wanderer←NEW, wander(world_model),
  PRIORITY(very_low)+QUANTUM(2)
  +RUN_ME)
```

The current process would continue to run, and wanderer would languish in ready status until everything of higher priority had been suspended.

Processes may be suspended or terminated via

SUSPEND(⟨process item expression⟩)

and

TERMINATE(⟨process item expression⟩)

which do just what one might expect. Similarly, SAIL provides system functions for changing a process' priority or quantum.

Co-routine style interactions are facilitated by the use of the RESUME construct:

```
x←RESUME(⟨process item expression⟩,
  ⟨return value⟩, ⟨options⟩)
```

where ⟨options⟩ is again optional. The usual effect of RESUME is to cause the currently running process to be suspended and the process specified by ⟨process item expression⟩ to become running. If the process being resumed had suspended itself by means of a resume statement, then it will receive ⟨return value⟩ as the value of the RESUME. For instance,

```
:
PROCEDURE tool_getter(ITEMVAR tool_type);
  BEGIN
  ITEMVAR tool;
  FOREACH tool | tool ∈ tool_box ∧ type⊗tool
    ≡tool_type DO RESUME (CALLER(THIS_
    PROCESS), tool);
  END;
:
SPROUT(tg←NEW, tool_getter (screwdriver),
  SUSPEND_HIM),
DO sd←RESUME(tg, NIC) UNTIL
  fits(sd, screw1);
TERMINATE(tg);
:
```

In this case, the tool getter process "tg" will be initialized and immediately suspended. Then, the RESUME (tg, NIC) will wake it up to find one screwdriver, which will be assigned to itemvar "sd" by the RESUME (CALLER (THIS_PROCESS), tool). (THIS_PROCESS and CALLER (⟨procid⟩) are system supplied routines that return the process items for the currently running process and for the process that last awakened process ⟨procid⟩, respectively). Later on, we will discuss a somewhat cleaner solution, using matching procedures, to the problem used for this illustration. We

will also show how the interprocess communication facilities of the language may be used to handle the problem of what to do if tool_getter runs out of tools.

FOREACH STATEMENTS

The standard way of searching the LEAP associative store is the FOREACH statement. A FOREACH statement consists of a "binding list" of itemvars, an "associative context" and a statement to be iterated. Consider the following example,

```
FOREACH gp, p, c | parent ⊗ c ≡ p ∧ parent
    ⊗ p ≡ gp DO MAKE grandparent ⊗ c ≡ gp;
```

In this example the binding-list consists of the itemvars "gp," "p," "c." The associative context consists of two "elements," "parent ⊗ c ≡ p," and "parent ⊗ p ≡ gp." The statement to be iterated is the MAKE statement.

Initially all three itemvars are "unbound." That is, they are considered to have no item value. Since "p" and "c" are unbound, the element "parent ⊗ c ≡ p" represents an associative search. The LEAP interpreter is instructed to look for triples containing "parent" as their attribute. On finding such a triple, the interpreter assigns the object and value components to "c" and "p" respectively. We continue to the next element "parent ⊗ p ≡ gp." In this element there is only one unbound itemvar, "gp," "p" is not unbound even though it is in the binding list because it was bound by a preceding element. A search is made for triples with "parent" as their attribute and the current binding for "p" as their object. If such a triple is found, its value component is bound to "gp" and the MAKE statement is executed. After execution of the MAKE statement, the LEAP interpreter will "back up" and attempt to find another binding for "gp" and then execute the MAKE statement again. When the interpreter fails to find another binding, it backs up to the preceding element and trys to find other bindings for "p" and "c." Finally when all triples matching the pattern of the first element have been tried, the execution of the FOREACH statement is complete.

In old SAIL, FOREACH elements consisted of either triple searches, set membership, or boolean expressions not dependent on unbound itemvars. Only triple searches and set membership were allowed to bind an unbound itemvar.

New SAIL contains a new way of binding itemvars called a MATCHING procedure. A matching procedure is essentially a boolean procedure which may have zero or more BINDING (written as "?") itemvars as formal parameters. These parameters are not necessarily bound at the time the procedure is called. If the procedure cannot find bindings for its unbound BINDING parameters, it FAILs, causing the LEAP interpreter to back up to the previous element within the associative context of the FOREACH. If it SUCCEEDs, bindings for the unbound parameters will be returned. The matching procedure is actually SPROUTed as a coroutine process. SUCCEED and FAIL are essentially forms of RESUME which return control to the caller with the values TRUE and FALSE, respectively. FAIL also causes the matching procedure process to be TERMINATEd. When the matching procedure is called by "back-up," it is merely RESUMEd. Thus, the entire environment in terms of the procedure's local variables, stack, etc., is the same as when the procedure executed the previous successful return. The matching procedure may continue from the point at which it left off, generating new bindings for its unbound parameters, In many respects matching procedures are similar to the IPL-V "generators" which have appeared in varied forms in other problem-solving languages.

To aid in the binding operations we have provided predicates to determine if a specific parameter is unbound for this call of the procedure. We also have introduced a new form of the FOREACH statement which conditionally adds itemvars to its binding list. Consider the following example of the new form:

```
MATCHING PROCEDURE tool_getter (?
  ITEMVAR tool, tool_type);
BEGIN FOREACH ?tool, ?tool_type | tool ∈
  tool_box ∧ type⊗tool ≡ tool_type DO
  SUCCEED;
FAIL;
END;
```

The binding list of the FOREACH would contain "tool" only if "tool" were unbound. Similarly it would contain "tool_type" if "tool_type" were unbound. The action of the matching procedure is to find a tool if the tool is unknown but the type is known; find the type if the tool is known but the type is not; verify that the tool is of the required type if both are known; or search through the toolbox and return tool, tool_type pairs if neither tool nor type is known. The actual semantics is determined by which, if either, of the parameters are bound.

Unfortunately in general, matching procedures with more than a single potentially unbound parameter are not so easy to code. The user may have to provide up to $2 \uparrow N$ different code sequences to handle the various combinations of N BINDING itemvars.

To illustrate one class of uses of matching procedures let us consider the following problem. We are given a set of cube shaped blocks of varying sizes and are requested

to pick a subset of the blocks such that when stacked they will form a tower of a given height. Assume that we will represent a cube by an item whose datum is the height of the cube. We may easily solve this problem by using a recursive procedure "find1."

```
RECURSIVE BOOLEAN PROCEDURE find1
  (SET bset, INTEGER diff; REFERENCE SET
  ans);
BEGIN INTEGER ITEMVAR newb;
  FOREACH newb | newb ∈ bset ∧ (∂(newb)
    ≤ diff) DO IF (∂(newb) = diff) ∨ find1 (bset-
    {newb}, diff-∂(newb), ans)
      THEN BEGIN PUT newb IN answer;
        RETURN (TRUE) END;
  RETURN (FALSE);
END;
```

However, now let us consider a slightly different problem. Suppose we wish to simultaneously build two towers from a single set of blocks. Calling "find1" twice, first with the entire set of blocks for for the first tower, then with the remaining blocks for the second, will not work. Though there may exist many possible subsets which will form the first tower, "find1" will always return the same one even though it is possible to construct the second tower only if a different subset of the blocks were chosen for the first tower. For example, if the set of blocks consisted of sizes 1, 4, and 5 and we were to construct towers of heights 5 and 4, "find1" would construct the first tower using blocks 1 and 4 and thus be unable to construct the second tower.

Now let us see how we would use matching procedures to overcome this problem. Let us write the matching procedure to solve a single tower problem [1],

```
MATCHING PROCEDURE find2 (SET bset;
      INTEGER height; ? SET ITEMVAR ans);
BEGIN
  RECURSIVE PROCEDURE aux
                    (SET s1; INTEGER diff);
  BEGIN INTEGER ITEMVAR newb;
    FOREACH newb | newb ∈ s1 ∧
      (∂(newb) ≤ diff) DO BEGIN PUT newb IN
      ∂(ans);
    IF (∂(newb) = diff) THEN SUCCEED
      ELSE aux (s1-{newb}, diff-∂(newb));
    REMOVE newb FROM ∂(ans);
      END;
  END;
    ans ← NEW({}); COMMENT new item. The
      empty set is datum;
      aux (bset, height);
    FAIL;
END;
```

To call the matching procedure we would simply have a FOREACH statement:

```
FOREACH ans | find2(blockset, height, ans) DO
        printset(∂(ans));
```

This is clearly equivalent to the solution given above for "find1." However, now consider the two tower case:

```
FOREACH ans1, ans2 | find2 (blockset, height1,
    ans1) ∧ find2 (blockset-∂(ans1), height2,
    ans2) DO
  printsets(∂(ans1), ∂(ans2));
```

This will find a solution if any exists, because if, after finding a solution to the first tower, it is impossible to find a solution to the second problem, we back-up and find a different solution to the first tower and then try the second again.

An interesting distinction between the programs for "find1" and "find2" may be found. Notice that "find1" only returns to its caller after "unwinding" the recursion, thus allowing the answer set to be constructed as the recursion is being "unwound" within a successful call. With "find2," however, the procedure may "return" or succeed while it is still deeply nested in recursion and thus the answer set must be constructed before the next recursive call of "aux" is made.

We envision that matching procedures will be used to simulate n-ary relations, serve as generators of moves or strategies, as well as simply aid in the coding of complex associative contexts.

INTERPROCESS COMMUNICATION

In complicated systems such as the Stanford Hand-Eye system, where there are many cooperating processes present, one would like to have a mechanism by which an occurrence in one process can influence the flow of control in other processes. Such occurrences frequently fall into several basic groups, with perhaps some distinguishing information associated with each occurrence of a given type. In designing interprocess communication facilities for SAIL we wanted to make it easy for the user to distinguish among happenings of the same general type and to define for himself just how each type is to be handled. We have chosen an "event" mechanism which is really a fairly general message processor. Any item may be used as an "event notice," or message, and each type of event in a program is represented by an item. With each such event type, SAIL associates:

1. A "notice queue" of items which have been "caused" for this event type.

2. A "wait queue" of processes which are waiting for an event of this type.
3. Procedures for manipulating the queues.

The two essential actions associated with any event type are

CAUSE(⟨event type⟩, ⟨notice item⟩, ⟨options⟩)

and

INTERROGATE(⟨event type⟩, ⟨options⟩)

where, as elsewhere, ⟨options⟩ may be left out if the default case is desired.

The statement

CAUSE(type1, ntc)

would cause SAIL to look at the wait queue for type1. If the queue is empty, then "ntc" would be put into type1's notice queue. Otherwise, a process would be removed from the wait queue and reactivated, with "ntc" as the awaited item.

If a process executes the statement

itmv←INTERROGATE(type1)

then the first item in the notice queue for type1 would be removed from the queue and assigned to itemvar itmv. If the queue is empty, then itmv would be set to the special item NIC. If a process wants to wait for an event of a given type, it may do so, as in

itmv←INTERROGATE(type1, WAIT)

In this case, if the notice queue is empty, then the process will be suspended and put onto the wait queue for type1.

Similarly,

itmv←INTERROGATE(type1, RETAIN)

causes the event notice to be retained in the notice queue for type1.

This event mechanism should prove useful in problem solving applications in which processes are sprouted to consider different actions. An "or" node in a goal tree, for example, might be represented by

```
:
:
SPROUT (p1, nail (sucevt, boards));
SPROUT (p2, glue (sucevt, boards));
SPROUT (p3, screw (sucevt, boards));
winner←INTERROGATE (sucevt, WAIT);
FOREACH p | p∈{p1, p2, p3} ∧ p≠winner
  DO TERMINATE(p);
:
```

When a branch discovers that it has succeeded, it can execute a statement like

CAUSE (sucevt, THIS_PROCESS);

which would announce success and cause its parent to terminate its less successful brothers.

Events give us a means by which some discovery made by one process can be made to "unstick" some other process which has gotten into trouble. Let's consider our tool getter again:

```
PROCEDURE tool_getter (ITEMVAR
                                    tool_type);
  BEGIN
  ITEMVAR tool;
  FOREACH tool | tool∈toolbox ∧ type⊗tool≡
                                    tool type DO
    RESUME (CALLER (THIS_PROCESS),
                                    tool);
  DO tool←INTERROGATE (tool_found,
        WAIT) UNTIL type⊗tool≡tool_type;
  RESUME (CALLER (THIS_PROCESS), tool);
  END;
```

If the FOREACH statement fails to find a tool of the correct type, then tool_getter will be suspended until some process causes an event of type tool_found, using the item representing the tool as the event notice. Suppose that our process "wanderer" has finally gotten a chance to run (everything of higher priority being stuck) and that it does, in fact, stumble across a screwdriver, which it knows to be a kind of tool. It might then do something like

```
:
MAKE type⊗thing≡screwdriver;
PUT thing IN tool_box;
CAUSE (tool_found, thing, TELL_EVERYONE
                              +DONTSAVE);
:
```

This would cause every process waiting on the event "tool_found" to be awakened. (If no process is waiting, the notice will not be saved on the notice queue.) This would wake up whoever called tool_getter, which would then see if it can use the "thing."

Frequently, one wishes to ask about one of several possible conditions. In some cases this could be done by a simple loop which INTERROGATEs each event type in a list. Unfortunately, if one wishes to wait for an occurrence within a given set of events, this doesn't work very well, since an attempt to wait for one event type will keep the other types from being seen. Therefore, SAIL allows a process to ask about a set or list

of event types directly, as in

itmv←INTERROGATE

(ev_type_lis, WAIT+RETAIN)

If WAITing is requested, then the process will only wait if all of the notice queues are empty, and it will be reactivated as soon as any of wait queue entries is serviced (All wait queue entries for this request will be deleted.) If it is necessary to know just which type was responsible for a given notice, the option SAY_WHICH may be used. Suppose the statement

itmv←INTERROGATE (ev_type_lis,

WAIT+SAY_WHICH)

returns item "notic," which was caused as an event of type catastrophe, as its value. Then the association EVENT_TYPE⊗notic≡catastrophe will be made by the system.

Thus, one way to program an "and" node within process "foo" might be something like

```
SPROUT(p1, fetch(hammer, hand1, sucevt,
                                    failevt));
SPROUT(p2, fetch(nail, hand2, sucevt, failevt));
:
SPROUT(pn, lookat(tv1, boards, sucevt, failevt));
FOR i ← 1 STEP 1 until n DO
  BEGIN
  p←INTERROGATE({{failevt, sucevt}},
                                    WAIT);
  IF EVENT_TYPE⊗p≡failevt THEN
     BEGIN
     MAKE failure_cause⊗foo≡p;
     FOREACH p | p ∈ {{p1, p2, . . . , pn}}
                        DO TERMINATE(p);
     CAUSE(foos_failure_event, foo);
     SUSPEND(foo);
     END;
  END;
CAUSE(foos_success_event, foo);
```

Here, it is assumed that each process is to take responsibility for making "life or death" decisions regarding any subprocesses. As soon as one of the p_i reports failure, foo will terminate all its "children" (whose appointed tasks have become pointless) report its own failure, and suspend itself. If all the p_i report success, then foo will do likewise.

Events may be used together with matching procedures to do deferred updating, as is shown by the following example. A matching procedure may want to make some change to the data base only if the rest of the associative context of the FOREACH succeeds. A simple way of implementing this is to have the matching procedure spawn a process which will do the updating. This process will go into event wait, and the event will only be caused if the entire associative context of the FOREACH succeeds. Consider the following guilt-by-association program. For each member of the suspect list, we first see if he is really undesirable by checking his bank account. If he doesn't have enough money to bribe us we will put another blackmark in the file of anyone who has any association with him, unless that person's only association with his is as an informer (in which case the fink will be given a "negative" black mark). When a person gets 5 black marks he then becomes a suspect.

```
SET badguys; LIST suspect;
MATCHING PROCEDURE linked
                    (BINDING ITEMVAR x);
BEGIN
  PROCEDURE UPDATE;
  BEGIN INTEGER ITEMVAR y, f;
    WHILE TRUE DO
    BEGIN f←INTERROGATE
                          (linkedok, WAIT);
          PUT x IN badguys;
          ∂(f)←∂(f)−2;
          FOREACH y | ANY ⊗ x ≡ y
          DO
          BEGIN ∂(y)←∂(y)+1;
            IF ∂(y) ≥ 5 THEN PUT y IN
                    suspect AFTER ∞;
          END;
    END;
  END;
  ITEMVAR z;
  z←NEW; SPROUT (z, update);
  FOREACH x | x ∈ suspect DO
    SUCCEED;
  TERMINATE (z);
  FAIL;
END;
⋮
⋮
⋮
COMMENT main procedure execution;
FOREACH person, fink | linked (person) ∧
                    (wealth (person) ⟨lots)
  ∧ Informer○person≡fink DO
BEGIN          CAUSE (linkedok, fink);
  ⋮
  ⋮
  ⋮
END;
```

This simple example does of course not really require either matching procedures or the event mechanism to cause the updating, but the technique it illustrates should be quite valuable in more complicated situations.

Although the provided event primitives are sufficient for most of the applications which we have considered, there are some cases for which they are not quite right. For instance, a process might want to wait for a given event only if no other process is already waiting for that event. Instead of trying to provide a special option to cover every possible contingency, we have instead provided a set of queue and process primatives with which the user can write his own CAUSE and INTERROGATE procedures. To substitute his own procedure for the one provided by SAIL, the user makes an association of the form

CAUSE_PROC⊗type1≡new_cause_proc

or

INTERROGATE_PROC⊗type1≡new_int_proc

where type1 is the event type and new_cause_proc and new_int_proc are procedure items bound to the substitute procedures. These procedures will be run as "atomic" operations, and will be allowed to finish without interruption. In particular, any CAUSEs or changes in process status requested by such a procedure will not actually take place until after the procedure exits. This "interrupt level" turns out to be quite useful and permits one to write interrupt handlers that look at a notice of some event, do what they can, and then either just return or else cause an event that will trigger some stronger condition.

CONCLUSION

Each of the features described in this paper was intended to solve particular programming problems. We have not yet had sufficient practical experience with the new system to say with certainty that they are the right ones. There is a great deal of work on these problems in several laboratories and new issues are being raised frequently. We do feel, however, that the basic solutions suggested here will prove useful and that they do significantly extend the capabilities of Algol-like languages.

ACKNOWLEDGMENT

While the work described in this paper was being done, there has also been a significant effort at the Stanford A. I. Lab to produce a new LISP system (LISP 70) which also includes provisions for multiple processes, backtracking, and other similar features. We would like to thank the authors of this effort, Horace Enea, Larry Tesler, and David Smith for several interesting conversations about their system. Although the approach they have taken is somewhat different from ours, these talks provided us with several useful insights.

REFERENCES

1 B ANDERSON
Programming languages for artificial intelligence: The role of non-determinism
School of Artificial Intelligence Univ of Edinburgh Experimental Programming Reports No 25

2 G BIRTWISTLE
Notes on the SIMULA language
Norwegian Computing Centre Publication S-7 April 1969

3 J A DERKSEN
The QA4 primer
SRI Project 8721 Draft Memo 15 June 1972

4 J A FELDMAN P D ROVNER
An Algol-based associative language
Comm ACM 12 8 August 1969 pp 439-449

5 J A FELDMAN R F SPROULL
System support for the Stanford hand-eye system
Proc Second IJCAI September 1971 pp 183-189

6 C HEWITT
Procedural embedding of knowledge in Planner
Proc Second IJCAI September 1971 pp 167-182

7 D V McDERMOTT G J SUSSMAN
The CONNIVER reference manual
MIT AI Memo 259 May 1972

8 E I ORGANICK J G CLEARY
A data structure model of the B 6700 computer system
SIGPLAN Notices 6 2 February 1971 pp 83-145

9 D C SWINEHART R F SPROULL
Sail manual
Stanford Artificial Intelligence Laboratory Operating Note No 52