View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library

BACKTRACKING IN GENERALIZED CONTROL SETTINGS

BY

Gary Lindstrom

UUCS - 77 - 105

This work has been supported in part by the National Science Foundation under grant DCR73-03441 A01 to the University of Pittsburgh

July 6, 1977

Abstract

Backtracking is a powerful conceptual and practical technique in programming. However, its application in general has been limited to global control over recursive programs. In this paper we explore through several examples the coherence and utility of applying backtracking in more general control settings, notably coroutine environments. The examples include: (i) a dual tree walk program using coroutine-managed backtracking subsystems, (ii) a context-free language intersection tester using bi-level hierarchical backtracking, and (iii) a minimizing computer job scheduler using backtracking in a simulation language setting. Full programs are given for each example, expressed in a PASCAL extension offering both coroutines and nondeterministic control.

Computing Reviews categories: 4.22; 4.42, 8.1.

Key words and phrases: backtracking, nondeterministic programming, coroutines, simulation.

1. Control Structures and Software Engineering

A central goal of software engineering is the construction of programs that are concise, comprehensible, reliable, and efficient. There is little dispute today that higher-level programming languages (HLLs) are indispensible aids in the pursuit of that goal. One feature contributing to this indispensibility is the richness and coherence of the data structures found in modern HLLs (e.g. PASCAL [JW74]). Another such feature, less widely recognized but of growing importance, is the power and variety of control structures found in newer HLLs (e.g. SIMULA-67) [IM72]).

The control structures available in a HLL delimit the range of algorithms that can conveniently be programmed therein. For example, naturally recursive algorithms (such as certain enumeration techniques) can be expressed in nonrecursive languages only after a machine mapping process, in which the inherent recursion of the algorithm is converted to sequential control manipulating explicit stacks, global storage pools, and control routing markers.

In comparison with a program written in a HLL with appropriate control structures, such an overtly "control engineered" program typically manifests the following undesirable properties:

i) it is <u>larger</u>, due to the greater level of explicit detail;

ii) it is <u>more complex</u> (and hence less comprehensible) for the same reason;

iii) it is <u>less efficient</u> since the programmer is forced to simulate the desired control regime within the available HLL, when a careful system implementation could be done with greater economy;

iv) it is less reliable, for the programmer must scrupulously follow his or her own ad hoc control conventions throughout the program (and hence the potential for error is increased), and

v) it is harder to document, test and verify, because the program's global control organization is expressed in more rudimentary form, thereby obscuring its proper framework for systematic documentation, testing, and verification.

On a higher level, one may view the development of new control structures as an appropriate software engineering concern in itself, for the range of control regimes in one's linguistic command governs the range of the algorithms which can be conceived. To paraphrase Wittgenstein, what cannot be written cannot be designed.

2. Backtracking and Nondeterministic Programming

A prime example of an HLL control structure natural for an important class of algorithms is backtracking control [GB65]. Problem areas such as topdown parsing, game tree searching, and finite optimization are among the many in which backtracking is a common and powerful programming technique. Backtracking control primitives have been incorporated into a number of HLLs, notably in the area of artificial intelligence systems [BR74].

Floyd has pointed out [F167] that a backtracking problem is often best first approached in the conceptual domain of nondeterministic (ND) programming. ND programming permits the programmer the luxury of an oracle that selects program branches with unfailing prescience toward a final goal state. Thus whenever the program is poised at a ND branch point, the question "which branch should be taken?" receives the wise answer "the correct one, of course, namely ... ". The programmer need only provide code to detect the achievement of final goal states and the recognition of blind alleys (i.e. states from which all final goal states are known to be inaccessible). The execution of a ND program results in the selection of one path through the program leading to a final goal state. If no such path exists, then the program never executes at all!

Of course, such a mystical mode of execution must be implemented through systematic searching of all execution paths and their retraction upon arrival at blind alleys. For this reason, backtracking is a natural implementation strategy for ND programs. Several options may be followed in the application of backtracking to ND programs, including:

i) the ND program can be executed as written if a backtracking implementation of the ND language is available;

ii) the ND program can be translated systematically (e.g. by hand, macros, or compiler) into an equivalent program doing explicit backtracking in a nonback-tracking language, and

iii) the program resulting in (ii) can then be optimized through the use of selective state saving, heuristic search ordering, or even reverse execution techniques [Gb72].

In light of their intimate connection, we will use the two terms "backtracking" and "ND programming" somewhat interchangeably throughout this paper. Nevertheless, our selection in each case will be prejudiced toward "ND programming" when dealing with programming concepts, and "back-tracking" when dealing with implementational or executional effects.

3. A Current Narrowness in Backtracking

Despite its attractiveness as a general search strategy, backtracking seems to have been largely applied only within rather limited control settings. That is, one finds the basic single path algorithm (i.e. the ND program with branch points viewed as being bound to single "headstrong" choices) expressed in sequential or at most recursive form. Also, the notion of multiple interacting backtracking subsystems seems to be unexplored.

This control simplicity is unfortunate for two reasons:

i) there are a number of important single path algorithms involving control regimes beyond recursion (e.g. coroutines, scheduled processes, and (even!) backtracking itself) which merit the power of an upper level backtracking control regime, and

ii) one has the anomaly that backtracking itself in general requires a control implementation beyond recursion (cf. [PSW72] and [SE73]), and there is no reason in principle why these added powers should be reserved to the implementation and denied the source program itself.

Our purpose in this paper is to demonstrate through several examples the conceptual power, coherence, and convenience of using backtracking within control settings beyond simply recursion. We commence in section 4 by defining a particular extension to PASCAL involving both ND and coroutine control forms. Each of the next three sections then describes a particular algorithm beyond the scope of recursion-based backtracking, along with its expression in our generalized PASCAL. The first such example (section 5) deals with the control of parallel backtracking systems in a symmetric

coroutine fashion. Section 6 concerns two backtracking systems hierarchically arranged, with one controlling the other. The final example, given in section 7, applies ND programming within a simulation language setting (i.e. processes scheduled along a time base). Section 8 then concludes the paper with a brief consideration of some conceptual (i.e. semantic) and practical (i.e. implementational) issues associated with this control generalization.

4. A Generalized PASCAL

To provide a concrete setting for our examples of generalized control, we will now define a PASCAL extension offering both coroutines and nondeterministic programming.

4.1. Coroutine control.

The coroutine manipulation facilities of our pASCAL extension have been selected from those found in Coroutine PASCAL [Lm76]. We will need the following primitives:

i) the data type *ref*, which applies to values that are names of dynamically created coroutine instances;

ii) the function *CREATE(<procedure call>)*, which dynamically creates a new coroutine instance of the given procedure. Parameters are evaluated and bound, but execution of the instance does not yet commence. A value of type *ref* referring to the created instance is returned as the value of this call on *CREATE;*

iii) the function *CALL(<ref exp>)*, which passes control to the coroutine instance referred to by the given expression. If that instance is newly created, it begins execution at its first statement. If the instance currently is *DETACH*ed, it resumes following the statement that caused that detachment.

iv) the procedure *DETACH*, which suspends the most tightly surrounding coroutine instance (in the sense of *CALL/DETACH* nesting), and returns control to its most recent *CALL*er, with control resuming just following the statement doing that call; and

v) the procedure *TERMINATE* (equivalent to existing from the code body of the most tightly surrounding coroutine instance), is similar to *DETACH* except that the coroutine instance is no longer *CALL*able.

4.2. Nondeterministic control.

A number of linguistic formulations of backtracking and ND control have been offered in the literature (e.g. [Jh67], [BR75], [Ch75], and [Hn76]). We find the early work of Floyd [Fl67] to offer the best basis for our needs here. Our primitives are:

i) the function *NDCREATE(<procedure call>)*, which creates a coroutine instance of the given procedure operating as a ND system. This means:

   a) the instance may be manipulated (e.g. *CALL*ed and *DETACH*ed) as an ordinary coroutine instance, but in addition:

   b) one may use the special ND control primitives *CHOICE*, *SUCCESS* and *FAILURE* within its dynamic scope.

ii) the function *CHOICE(*<exp>), delivering an integer value from 1 to the value of *<exp>* nondeterministically selected;

iii) the procedure *SUCCESS*, which indicates attainment of a final goal state for the ND system most tightly surrounding. This has the net effect of "releasing" for external output any printing done by the ND system along this execution path, and

iv) the procedure *FAILURE*, signaling detection of a blind alley. This causes the following backtracking actions to occur in

the most tightly surrounding ND system:

a) the ND systems's control state is reset to that in effect at the time of the most recently executed *CHOICE* call in that system. If at least one value remains to be generated by that *CHOICE* operation, a new value is selected for generation. Otherwise, if that *CHOICE* operation is exhausted, then the system's control state is reset to that associated with the next most recent *CHOICE* operation, etc. If all previous *CHOICE* operations in this system have been exhausted, then a *TERMINATE* is done closing out this system's execution.

b) the local data state of this ND system (i.e. the set of all variables created within its dynamic scope) is reset to that associated with the selected *CHOICE* point. Note that variables outside the ND system are left unchanged by a *FAILURE* action.

c) any printing done by the ND system in the time period between the *FAILURE* point and the selected *CHOICE* point is "retracted", i.e. will not appear as output upon the occurrence of any future *SUCCESSes* in the system.

4.3. An example.

To illustrate our language on a traditional backtracking problem, we consider a variation of the Flagstone Problem:

Problem P0: We wish to tile a walkway with a linear sequence of m flagstones selected from ample supplies of k different colors. In particular, we are seeking a certain sense of "randomness" in our layout, such that no two adjacent innner sequences repeat a color pattern. Our task is to exhibit a satisfactory sequence, or report conclusively on its impossibility (e.g. for m=4 and k=2).

A solution to this search problem expressed in our extended PASCAL is given in figure 1.

```
const   colormax = 10;  {maximum number of different colors}
        tilemax = 100;  {maximum number of tiles to be placed}

type    colors = 1 .. colormax; {color names}
        tiles = 1 .. tilemax;   {tile position names}

var     m: tiles;
        k: colors;
        goodpattern: Boolean;   {success signal variable}

procedure randomtiles (m: tiles; k: colors);

        var     tilenow, innerseqlength, q: tiles;
                tileslaid: array [tiles] of colors;

        begin   for tilenow := 1 to m do
                begin   tileslaid[tilenow] := choice(k);  {pick color of next tile}
                        writeln(tileslaid[tilenow]);  {print it fearlessly}
                        for innerseqlength := 1 to tilenow div 2 do
                        begin   {look for repeat pattern ending on new tile}
                                for q := 0 to innerseqlength-1 do
                                        if tileslaid[tilenow-q] ≠
                                                tileslaid[tilenow-innerseqlength-q]
                                        then goto ok;
                                failure;  {repeat pattern found}
                        ok:     {no repeat pattern on this inner sequence length}
                        end
                end;
                {have a good pattern now}
                success;  {release printing}
                goodpattern := true  {signal success to main program}
        end;    {randomtiles}

begin   {main program}
        read(m);  {number of tiles to be laid}
        read(k);  {number of colors available}
        goodpattern := false;  {assume the worst}
        call(ndcreate(randomtiles(m,k)));  {activate search algorithm}
        if not goodpattern then
                writeln("no acceptable pattern possible")
end.    {main program}
```

Figure 1. Random flagstone program.

5. Generalization 1: Coroutine-Related ND Systems

Our first example of generalized backtracking deals with the simple notion that backtracking need not be a global search strategy. Rather, it may be a local regime operating within a larger control framework. To illustrate, consider the following problem:

> Problem P1: Given two n-ary trees with non-negative integer node weights, determine which tree has the path with the smallest node weight sum (if both trees possess such a path, then either tree is an admissible answer).

Clearly one could solve this problem by a depth first traversal of each tree in turn, recording its minimum path weight. However, a more efficient approach is to pursue minimum path searches on both trees at once, with control alternating between them on a "can you top this?" basis. Thus we have two independent ND search algorithms controlled as coroutines by a supervisor. A program following such a control strategy is given in figure 2.

Note that such an approach is beyond the realm of a system offering only global backtracking, for the two ND subsystems operate asynchronously.

```
const   hugeweight = 10000;       {generous upper limit on path weight}
        wtmax = 100;              {max weight for each node}
        nodemax = 50;             {max number of nodes in both trees combined}
        degmax = 5;               {max degree (nr. of immed. desc.) of each node}

type    pathweight = 1 .. hugeweight;   {range of possible path weights}
        nodes = 1 .. nodemax;           {names in node pool}
        degrees = 1 .. degmax;          {degree range for nodes}
        weights = 1 .. wtmax;           {range of node weights}

var     desc: array [nodes,degrees] of nodes;   {immed. descendants of nodes}
        deg: array [nodes] of degrees;          {degree of each node}
        wt: array [nodes] of weights;           {weight of each node}
        root1, root2: nodes;                    {roots of two trees}
        tau1, tau2: ref;                        {ND subsystem reference variables}
        best, oldbest: pathweight;              {path weight variables}

procedure shortest (root: nodes);

        var     sum: pathweight;
                nodenow: nodes;

        begin   sum := wt[root];  nodenow := root;
                while sum<best do
                begin   if deg[nodenow]=0 {i.e., nodenow is terminal} then
                        begin   {have new globally shortest path}
                                best := sum;  detach; {let colleague try to beat}
                                {if control returns, he did}
                                failure  {keep on searching}
                        end;
                        {nonterminal node case; move down tree}
                        nodenow := desc[nodenow,choice(deg[nodenow])];
                        sum := sum+wt[nodenow]
                end;
                failure  {current path weight already too large}
        end;    {shortest}

begin   {main program; read in desc, deg, wt, root1 and root2}
        tau1 := ndcreate(shortest(root1));  {create first ND subsystem}
        tau2 := ndcreate(shortest(root2));  {create second ND subsystem}
        best := hugeweight;  {first path will surely beat this}
        repeat  oldbest := best;  {save current best path weight}
                call(tau1);  {let first ND subsystem try to beat it}
                if best=oldbest then
                begin   {it didn't}  writeln("tree 2 has minimal path");
                        goto exit
                end;
                oldbest := best;  {now similar code for second ND subsystem}
                if best=oldbest then
                begin   writeln("tree 1 has minimal path");
                        goto exit
                end
        until false;
exit:
end.    {main program}
```

Figure 2. Parallel tree walk program.

6. Generalization 2: Hierarchical ND programming.

The preceding example illustrates the utility of local ND systems operating as control equals in a coroutine relationship. We now examine a natural variation of that organization: hierarchically organized ND systems. In hierarchically organized ND programming there are two or more ND subprograms operating on an ordered scale of subservience. To illustrate, we take the following sample problem:

> Problem P2: Given two context-free grammars G_1 and G_2, find a minimal length string in $L(G_1) \cap L(G_2)$, i.e. the intersection of their languages. If no such string exists of length less then or equal to lim, report that fact and stop; otherwise, exhibit the string found and stop.

Among the many solutions to this problem lies the following relatively clean approach:

1) Set k=1.
2) Generate all length k strings of $L(G_1)$ via a ND procedure. As each new character is produced (in left-to-right order) in a potential length k member of $L(G_1)$, pass that character to a G_2 parser, operating under its own ND regime.
3) The G_2 parser attempts to accommodate the current string as extended by the new character. If the G_2 parser fails in this attempt, it signals failure to the G_1 generator, which then retracts that character. In either case, the G_1 generator continues its generation process as described in (2).
4) The G_1 generator continues in this manner until either
   i) an entire string of length k is produced and parsed as

a member of $L(G_2)$, in which case a success is reported (causing the string to be printed), or

ii) the G_1 generator has produced all length k members of $L(G_1)$.

5) In the latter case k is incremented and compared against lim. If $k > lim$, a message is printed indicating the absence of the desired string and the program halts. Otherwise the process is begun anew at step (2).

Clearly this approach cannot conveniently be programmed in a language offering only recursion and global backtracking, for, as in our solution to problem P1, there are two ND systems operating with unpredictable interleaving of failure actions. Moreover, our approach here has the added complexity of a nested ND control organization, causing quite different logical effects upon failure in each of the two levels (more will be said of this shortly).

To simplify our programming of this problem, we will assume our grammars to be in the following special form (without significant loss of generality):

1) Their nonterminal symbol sets are disjoint subsets of the upper case alphabet $\{A, \ldots, Z\}$;

2) Their terminal symbols are taken from the set of lower case alphabetics $\{a, \ldots, z\}$;

3) The grammars are non-left-recursive (this restriction can be eliminated by a variety of methods all complicating exposition);

4) There is only one rule for each nonterminal symbol α, and that rule obeys one of the following three forms:
(alternation) $\alpha \rightarrow \beta_1 | \beta_2$, with β_1 and β_2 nonterminals;
(concatenation) $\alpha \rightarrow \beta_1 \beta_2$, with β_1 and β_2 nonterminals, or
(terminal) $\alpha \rightarrow \tau$, with τ a terminal symbol;

5) In each grammar the rule for the root symbol ρ is of the form $\rho \rightarrow \alpha\beta$, where $\beta \rightarrow z$ is a rule in that grammar and ρ, β, and z appear in no other rules in that grammar. (This ensures that all strings in each language end with the special symbol z).

Figure 3 shows a solution to P2 given the approach outlined above and this assumed class of grammars. The generator and parser subsystems both use a ND technique due to Floyd, presented in [F164] and extensively studied in [Ln76]. Figure 4 provides a schematic representation of the hierarchical ND control used in the program.

An important control sublety should be pointed out in the notion of hierarchical ND programming. Notice in our example that the inner ND system (i.e. the parser) does not operate in a simple subroutine relationship with the outer ND system (i.e. the generator). If it did, it would be presented an independent new task on each activation to be conclusively pursued before returning. Rather, the inner system incrementally searches forward on each activation.

However, this relationship is not a symmetric one as was the case in our solution to P1. Instead, the inner ND system is local to the outer system, so there is a significant difference in the effect of failure in the two cases. When the inner ND system reaches a failure state, it alone

```
const   strmax = 25;     {maximum string length}

type    symb = 'a' .. 'Z';                {vocabulary of grammars}
        ntsymb = 'A' .. 'Z';              {nonterminal symbols}
        termsymb = 'a' .. 'z';            {terminal symbols}
        ptrval = 1 .. strmax;             {pointer into string}
        substrlength = 1 .. strmax;       {substring length in string}

var     rule: array [ntsymb,1..2] of symb;               {rules of grammars}
        ruletype: array [ntsymb] of (alt,conc,term);     {rule types}
        root1, root2: ntsymb;                            {grammar root symbols}
        ok: Boolean;                                     {signalling variable}
        k, lim: ptrval;                                  {string length variables}

procedure genboss (k: ptrval);

        var     str: array [ptrval] of termsymb;         {string buffer}
                parseref: ref;                           {ref to parser ND system}
                genptr: ptrval;                          {current string length}

        procedure generate (goal: ntsymb; k: substrlength);

                var     j: substrlength;        {length partitioning variable}

                begin   case ruletype[goal] of

                        alt:    generate(rule[goal,choice(2)],k);

                        conc:   begin   j := choice(k-1);  {form all 2-partitions}
                                        generate(rule[goal,1],j);
                                        generate(rule[goal,2],k-j)
                                end;

                        term:   begin   if k≠1 then failure;
                                        str[genptr] := rule[goal,1];
                                        writeln(str[genptr]);
                                        ok := false;
                                        call(parseref);  {see if parser obliges}
                                        if not ok then failure; {retract character}
                                        if rule[goal,1]='z' then {have full string}
                                        begin   success;  {release printing}
                                                terminate  {close out search}
                                        end;
                                        genptr := genptr+1
                                end

                        end     {case statement}
                end;    {generate}
```

Figure 3. Context-free language intersection program (cont'd next page).

```
procedure parseboss;

        var     parseptr: ptrval;

        procedure parse (goal: ntsymb);

                begin   case ruletype[goal] of

                        alt:    parse(rule[goal,choice(2)]);

                        conc:   begin   parse(rule[goal,1]);
                                        parse(rule[goal,2])
                                end;

                        term:   begin   if rule[goal,1]≠str[parseptr]
                                        then failure;  {try alt. parse}
                                        if parseptr=genptr then {caught up}
                                        begin   ok := true;
                                                detach  {wait for next char.}
                                        end;
                                        parseptr := parseptr+1  {got one}
                                end

                        end     {case statement}
                end;    {parse}

        begin   {body of parseboss}
                parseptr := 1;  {initialize parser's string pointer}
                parse(root2)    {call parser}
        end;    {parseboss}

begin   {body of genboss}
        parseref := ndcreate(parseboss);   {create parser subsystem}
        genptr := 1;  {initialize generator's string pointer}
        generate(root1,k)  {call generator}
end;    {genboss}

begin   {main program; read in rule, ruletype, root1, root2 and lim}
        ok := false;
        for k := 1 to lim do
        begin   call(ndcreate(genboss(k)));
                if ok then goto exit
        end;
        writeln("no common string of length less than or equal to",lim);
exit:
end.    {main program}
```

Figure 3. (conclusion)

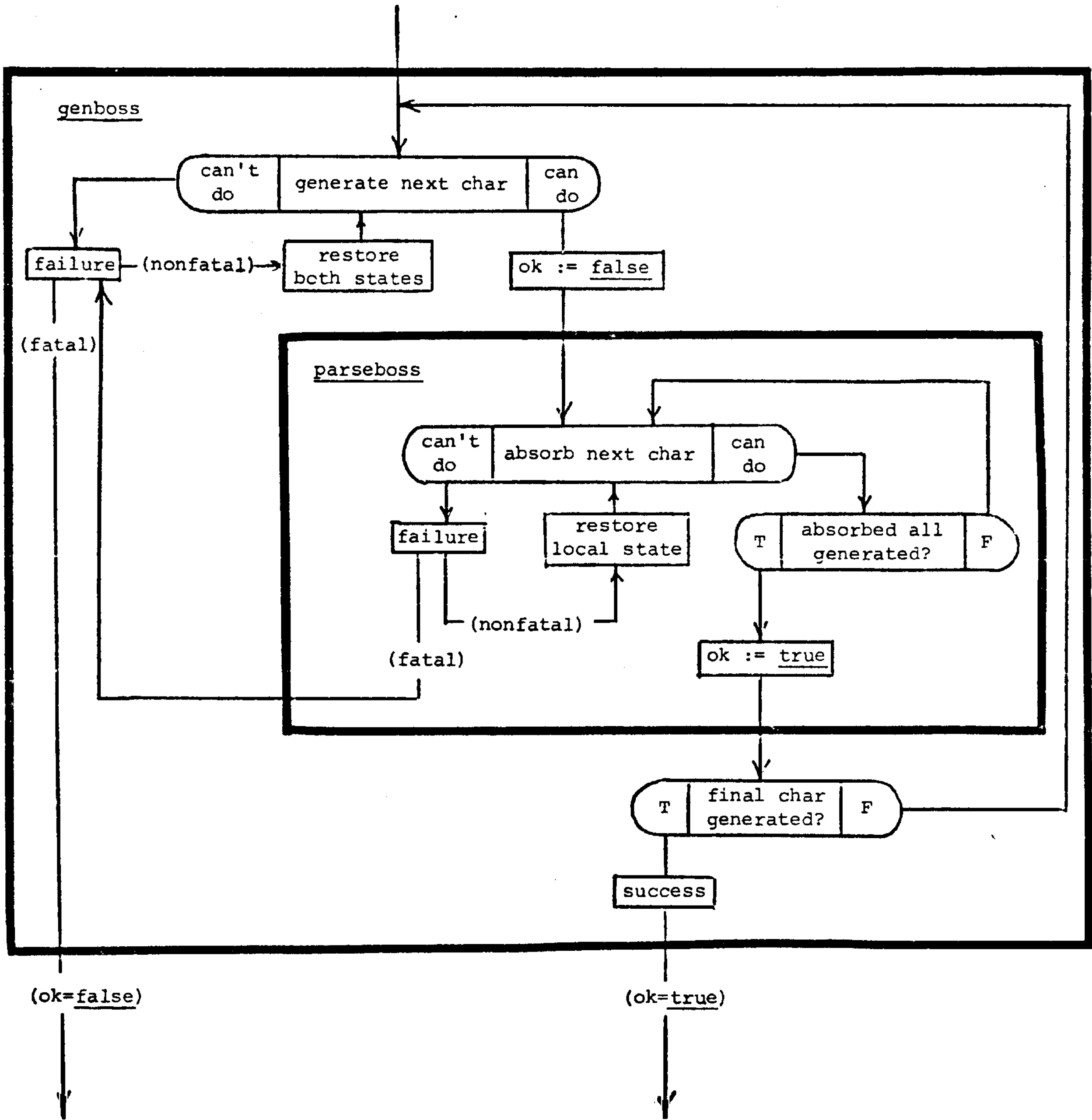


Figure 4. Hierarchical ND control in figure 3 program.

regresses to an earlier state. In contrast, when the outer ND system fails, both systems regress to earlier states.

In terms of our particular example, this produces the following salutary control effect. When the generator fails (either through a blind alley of its own or due to a parser rejection of the current string fragment), the following events occur:

i) the generator is restored to the state associated with its most recent unexhausted choice operation, thereby retracting some number of generated characters, and

ii) the parser concomitantly is restored to the state it possessed when the final character of the string (as just truncated) was first produced. Thus it is now "ready to go" on new successor characters to be concatenated onto that string fragment, even though it may have just previously been in a terminated state due to an unsuccessful parse attempt on the string prior to truncation.

7. Generalization 3: ND control over coroutine systems

In our previous two examples we have considered coroutines control over multiple ND systems operating with recursive internal control. As our final example we consider a converse situation: ND control applied globally to a system using coroutine control internally, _viz_. discrete simulation language programming.

We take the following as our motivating problem:

> _Problem P3_: A multiprogramming computer system is given a set of independent jobs to run. Each job consists of a sequence of timed phases delimited by the allocation or deallocation of a single, unique, nonsharable device. We consider the computing load of each phase to be negligible, so there is no _a priori_ limit on the number of jobs that can run concurrently, nor does that number have any impact on the real time speed of each phase in progress. Our question is: what is the earliest time at which that job mix can be finished?

The logical parallelism of this problem makes a simulation language approach ideal. However, two complications arise:

1) this is a minimization problem over the space of all possible allocation and deallocation sequences, so some type of enumeration strategy must be followed, and
2) the possibility of deadlock situations make only some of those sequences admissible.

Both of these added concerns make a minimizing ND search regime over a conventional simulation-style program a natural overall strategy for this problem.

In order to provide a convenient setting for programming a solution to this problem, we will extend our PASCAL variant further to include certain primitives abstracted from SIMULA-67. These include:

i) the function *SIMCREATE(<procedure call>)*, which creates a coroutine instance of the given procedure operating as a simulation system. This means:

   a) the instance may be manipulated (e.g. *CALL*ed and *DETACH*ed) as an ordinary coroutine instance, but in addition:

   b) one may use the special simulation control forms *SCHEDULE* and *PASSIVATE* within its dynamic scope;

   c) any coroutines instances created within its dynamic scope (but outside of any inner such system) are considered processes of this system;

   d) the body of the simulation system itself is a process with the special name *MAIN*;

   e) a special integer variable *TIME* is created local to the system and initialized to zero, and

   f) a special set variable called the event set is created local to the system and initialized to the empty set;

ii) the function *SCHEDULE(<process>, <time>)*, which creates an event notice (*<process>*, *<time>*) and adds it to the current system's event set, and

iii) the function *PASSIVATE*, which causes the following sequence of actions to take place:

   a) the current process is *DETACH*ed;

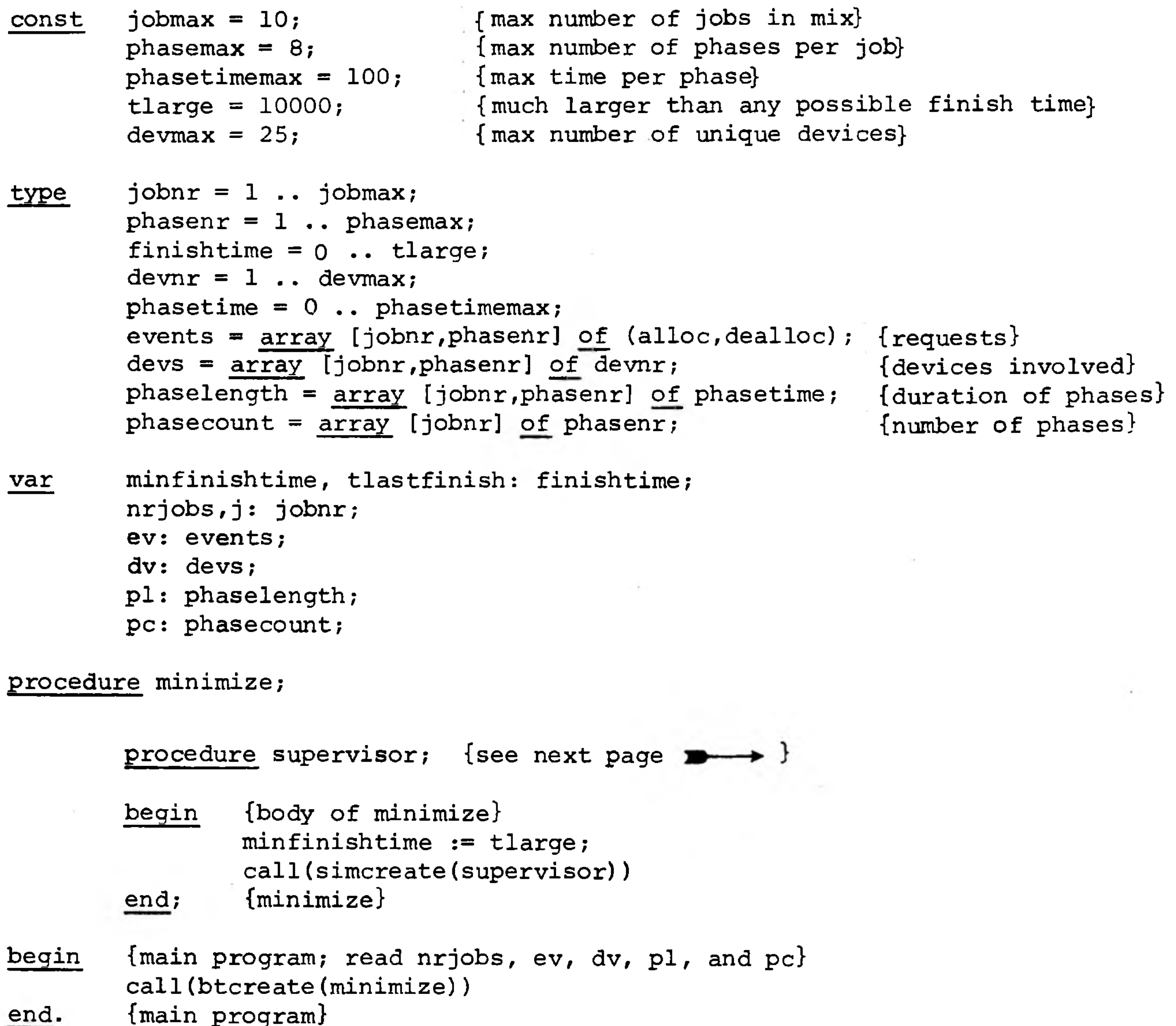
b) if the event set of the current system is empty, a *TERMINATE* is done exiting from the system;

c) otherwise, an event notice (p,t) with minimim t is removed from the current system's event set;

d) if t > *TIME*, *TIME* is set to t.

3) a *CALL(p)* is performed.

Figure 5 gives a solution to P3 phrased in these control terms. Note that the program prints a series of allocation and deallocation sequences, each finishing the job mix (without deadlock) at a time earlier than the previous. The final such sequence represents the minimum finish time.



```
const   jobmax = 10;              {max number of jobs in mix}
        phasemax = 8;             {max number of phases per job}
        phasetimemax = 100;       {max time per phase}
        tlarge = 10000;           {much larger than any possible finish time}
        devmax = 25;              {max number of unique devices}

type    jobnr = 1 .. jobmax;
        phasenr = 1 .. phasemax;
        finishtime = 0 .. tlarge;
        devnr = 1 .. devmax;
        phasetime = 0 .. phasetimemax;
        events = array [jobnr,phasenr] of (alloc,dealloc);  {requests}
        devs = array [jobnr,phasenr] of devnr;              {devices involved}
        phaselength = array [jobnr,phasenr] of phasetime;   {duration of phases}
        phasecount = array [jobnr] of phasenr;              {number of phases}

var     minfinishtime, tlastfinish: finishtime;
        nrjobs,j: jobnr;
        ev: events;
        dv: devs;
        pl: phaselength;
        pc: phasecount;

procedure minimize;

        procedure supervisor;  {see next page →}

        begin   {body of minimize}
                minfinishtime := tlarge;
                call(simcreate(supervisor))
        end;    {minimize}

begin   {main program; read nrjobs, ev, dv, pl, and pc}
        call(btcreate(minimize))
end.    {main program}
```

Figure 5. Optimizing job allocation program (cont'd next two pages).

```
procedure supervisor;

    var     devq: array [devnr] of set of jobnr;
            status: array [devnr] of (busy,free);
            jobref: array [jobnr] of ref;
            j: jobnr;
            d: devnr;

    procedure job (nr: jobnr);   {see next page ➡ }


    begin   {body of supervisor}
            for d := 1 to devmax do  {initialize devices}
            begin   devq[d] := nil;  status[d] := free
            end;
            for j := 1 to nrjobs do  {start up jobs}
            begin   jobref[j] := create(job(j));
                    schedule(jobref[j],0)
            end;
            schedule(main,tlarge);
            passivate;  {wait till finish or deadlock}
            for d := 1 to devmax do
                    if devq[d]≠nil then failure;  {have deadlock}
            minfinishtime := tlastfinish;  {new minimum}
            success;  {release printing of actions}
            failure   {keep trying to minimize}
    end;    {supervisor}
```

Figure 5. (continued; concludes next page)

```
procedure job (nr: jobnr);

        var     p: phasenr;
                j: jobnr;

        begin   {body of job}
                for p := 1 to pc[nr] do
                begin   case ev[nr,p] of

                        alloc:  begin   if status[dv[nr,p]]=busy or
                                                choice(2)=1 then
                                        begin   devq[dv[nr,p]] :=
                                                    devq[dv[nr,p]]+[nr];
                                                passivate; {wait till free}
                                                devq[dv[nr,p]] :=
                                                    devq[dv[nr,p]]-[nr]
                                        end;
                                        status[dv[nr,p]] := busy
                                end;    {allocation action case}

                       dealloc: begin   status[dv[nr,p]] := free;
                                        if choice(2)=1 then
                                        begin   {wake up a waiting job}
                                                j := choice(nrjobs);
                                                if not (j in devq[dv[nr,p]])
                                                    then failure;
                                                schedule(jobref[j],time)
                                        end
                                end     {deallocation action case}

                        end;    {case statement}
                        writeln("action",ev[nr,p],"on device",dv[nr,p],
                                "taken by job",nr,"at time",time);
                        schedule(jobref[nr],time+pl[nr,p]);
                        passivate;  {do phase}
                        if time>minfinishtime then failure {already too long}
                end;    {loop on phases of this job}
                tlastfinish := time
        end;    {job body}
```

Figure 5. (concluding page)

8. Future Work.

Our goal here has been to sketch via examples the synergetic effect of combining ND control with other advanced control forms (notably coroutines). Neither conceptual circumspection nor descriptive precision have been pursued as thoroughly as they deserve. Yet the worthiness of the basic notion, we hope, has been established.

We close by mentioning just four of the many areas for continuing research indicated by this brief study:

i) implementation: If one accepts the premise that ND control and coroutines can be mixed to advantage, how does one construct a language design and implementation scheme that is both easy to use and efficient;

ii) hierarchical simulation: An immediately appealing further application of this control combination lies in hierarchically organized simulation systems. The exact meaning of such a notion, and the set of problems (if any) for which it is a natural approach, remain to be investigated.

iii) simulation debugging: One traditional use of backtrack programming is in the management of tentative executions of undebugged programs. The control combination proposed here suggests that this technique might be applied to simulation language programs, which are notoriously difficult to bebug.

iv) general state management: There is no reason why the states associated with ND choice points must be restored in a stack-based order (as is done in true backtracking). Instead,

general mobility of ND search, including breadth-first, best-first, and general state recall (e.g. a retroactive SUCCESS to exhibit a path later found to be optimal) would be far preferable. One may conjecture that any implementation scheme supporting both ND control and coroutines must in fact be robust enough to support these more general forms of execution state management as well. Thus truly customized management of ND control may in fact be available at no added cost in such an implementation.

References.

[BR74] Bobrow, D. and B. Raphael, "New programming languages for artificial intelligence research", Computing Surveys, 6,3 (Sept. 1974) 155-174.

[BR75] Bitner, James R., and E. M. Reingold, "Backtrack programming techniques", CACM 18,11 (Nov. 1975) 651-656.

[Ch75] Cohen, Jacques, "Interpretation of non-deterministic algorithms in higher-level languages", Inf. Proc. Ltrs. 3,4 (March 1975) 104-109.

[F164] Floyd, R.W., "Syntax of programming languages: a survey", IEEE PGEC 4 (1964), p. 346. Also in Rosen, Programming Languages and Systems, McGraw-Hill.

[F167] Floyd, R.W., "Nondeterministic algorithms:, JACM 14,4 (Oct. 1967) 636-644.

[Gb72] Gibbons, Gregory Dean, "Beyond REF-ARF: toward an intelligent processor for a non-deterministic programming language", Ph.D. disser., Carnegie-Mellon Univ. (1972).

[GB65] Golomb, S.W. and L.D. Baumert, "Backtrack programming", JACM 12 (1965) 516-524.

[Hm76] Hanson, David R., "A procedure mechanism for backtrack programming", Proc. ACM Annual Conf. (Oct. 20-22, 1976), Houston, Texas, 401-405.

[IM72] Ichbiah, J.D. and S.P. Morse, "General concepts of the SIMULA 67 programming language", ARAP 7,1 (1972) 65-93.

[Jh67] Johansen, Peter, "Non-deterministic programming", BIT 7 (1967) 289-304.

[JW74] Jensen, K., and N. Wirth, PASCAL-User Manual and Report, Lecture Notes in Computer Science, Springer (1974) 170 pp.

[Lm76] Lemon, Michael, "Coroutine PASCAL: a case study in separable control", M.S. thesis, Tech. Report 76-13, Dept. of C.S., Univ. of Pgh. (Dec. 15, 1976) 68 pp.

[Lm76] Lindstrom, G., "Non-forgetful backtracking: an advanced coroutine application", Tech. Report 76-8, Dept. of C.S., Univ. of Pgh. (Dec. 6, 1976) 42 pp.

[PSW72] Prenner, Charles J., Jay M. Spitzen, and Ben Wegbreit, "An implementation of backtracking for programming languages", Proc. ACM Nat'l. Conf. (1972) 763-771.

[SE73] Smith, D.C. and H.J. Enea, "Backtracking in MLISP2", 3rd IJCAI, Stanford (1973).