And/Or Programs: A New Approach to Structured Programming

DAVID HAREL
IBM Thomas J. Watson Research Center

A simple tree-like programming/specification language is presented. The central idea is the dividing of conventional programming constructs into the two classes of *and* and *or* subgoaling, the subgoal tree itself constituting the program. Programs written in the language can, in general, be both nondeterministic and parallel. The syntax and semantics of the language are defined, a method for verifying programs written in it is described, and the practical significance of programming in the language assessed. Finally, some directions for further research are indicated.

Key Words and Phrases: alternation, and/or program, program verification, structured programming, textual complexity
CR Categories: 4.2, 5.24

> One of our aims is to make such well-structured programs that the intellectual effort . . . needed to understand them is proportional to program length . . . .
>
> —E.W. Dijkstra

> We do not know whether alternation will find its way into programming languages, or have a role to play in structured programming.
>
> —A.K. Chandra and L.J. Stockmeyer

1. INTRODUCTION

In this paper we present a programming/specification language based on the concept of alternating *and* and *or* subgoals. The general notion of and/or alternation is well known and occurs in mathematical logic (alternation of quantifiers, cf. [16]), game theory (and/or game trees, cf. [17]), and artificial

This research was supported in part by the National Science Foundation under Grants MCS77-19754 and MCS76-18461.
A version of this paper appeared in the *Proc. IEEE Specifications of Reliable Software Conf.*, Cambridge, Mass., April 1979.
Author's address: IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

intelligence (and/or problem solving trees, etc., cf. [18]). Recently, in [3] and [13], alternation was introduced into theoretical computer science as a powerful tool in classifying the computational complexity of classes of problems, and applications of it were illustrated and envisioned. It is from [3] that our second quotation above is taken. The language proposed here serves as a natural application of this concept to the discipline of structured programming (cf. [4]) from which the first quotation is taken.

It is commonly claimed that as much as the efficiency of a program is important (e.g., in terms of its consumption of resources such as time or space), its clarity, readability, and manageability are also to be a major concern of its author. To this end various disciplines have been advocated, including the use of structured programming and flowcharting. The former is an aid in *synthesizing* the program in a stepwise top-down fashion, and the latter is a tool for pictorially *describing* it.

One of the apparent problems with structured programming as it stands is in the fact that the *history* of the subgoaling process which produced the final program is not captured by the final text. Except for some consistent indenting of the program text (e.g., in Algol or PL/I) no "information-at-a-glance" pictorial description of the design structure of the program is present. Consequently, it is not always easy to carry out a modification without a considerable amount of insightful preparation. Thus, for example, if the first step of synthesizing a simple *compiler* was in refining the compilation process into the two subgoals of *parsing* and *coding*, this fact will not always be obvious from glancing at the final program text of the compiler; not until the text has been appropriately partitioned in a visual way, say by circling the two components. A complete logical structuring of the program text will consist typically of a nested set of such circles. In other words, the depth and structure of the stepwise composition of the program are not visible in the final product.

Similarly, the virtues of the visual representation supplied by flowcharts, while illustrating the flow of control, are of little help in capturing the structure of the logical design even when so-called "structured flowcharts" are employed (these corresponding essentially to indented textual programs), and a similar process of nested encircling has to be carried out. This situation is most problematic at later stages, i.e., when the program is to be constantly maintained and often modified. Cumbersome documentation then becomes a necessity.

The and/or programming language described in Section 3 is designed with this problem in mind; i.e., in the final tree-like progam the flow of control of a particular implementation of that program is secondary to its logical structure. It is precisely the structure of this natural stepwise synthesis of the algorithm that the tree captures. Each node of the tree represents the program consisting of the subtree rooted in that node, with its immediate descendants representing its decomposition into subgoals. Thus, in the above example, *parse* and *code* would be natural choices for the offspring of the node (in this case the root of the tree) denoted by *compile*. As will become clear, the "layers" in which the program is arranged, these being in the heart of the idea of structured programming (cf. [4, pp. 48–49]), correspond to the levels of the tree.

The resulting language possesses such pragmatic niceties as readability and ease of modification as well as considerable flexibility in choosing an implemen-

tation. This flexibility will be seen also to be the main drawback of our language; the flow of control can be obtained from the program text only by some moderate amount of analysis. Also, the language gives rise to rather natural versions of some of the standard methods for proving the correctness of programs. The programs, in general, admit both nondeterminism and a kind of parallelism, features which are lately being considered essential in many types of programming.

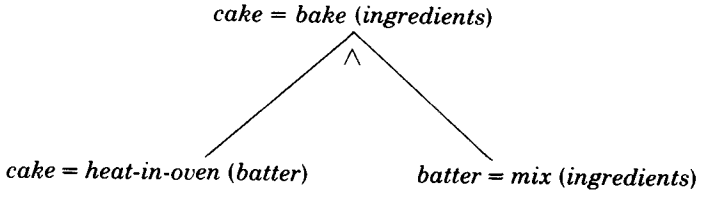
Section 2 contains an informal introduction to the language, and Section 3 presents the syntax and semantics. In Section 4 we illustrate how and/or programs are to be verified, while Section 5 is devoted to discussing the advantages the language offers the implementer and programmer. Some directions for future work are discussed in Section 6.

2. AND/OR SUBGOALING

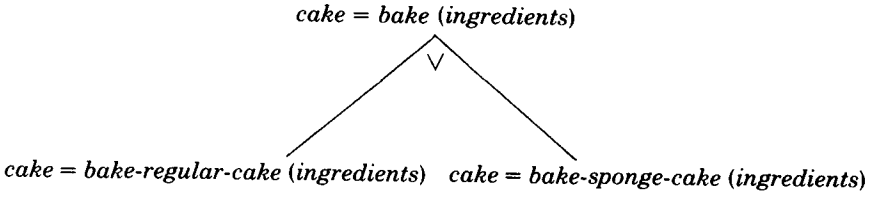
Consider the following example in which a cake is to be baked from some ingredients. Similarly to the line taken in [4, pp. 27], if baking cakes is a known primitive ("an instruction from a well-understood repertoire" [4]), then the one-node tree

$$cake = bake\ (ingredients) \tag{1}$$

completely solves the problem. If, on the other hand, we do no have *bake* in our repertoire, then the following are two suggestive ways of refining the main goal into subgoals:

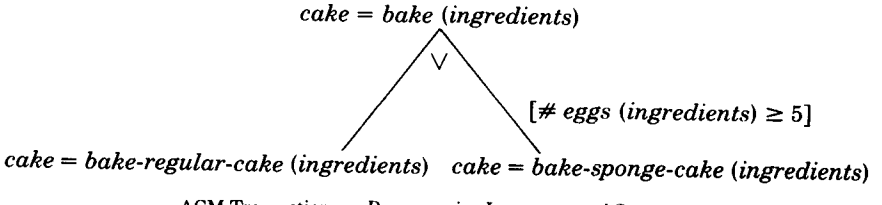


(2)



(3)

Tree (2) replaces *bake* by two subgoals, both of which are to be achieved for it to be completed, and (3) replaces it by two subgoals, one of which is to be achieved. We might want to restrict the choice of the *sponge-cake* alternative in (3) to the case in which the ingredients include at least five eggs, in which case we allow the attachment of this condition as a "guard" [5] on the appropriate edge, obtaining

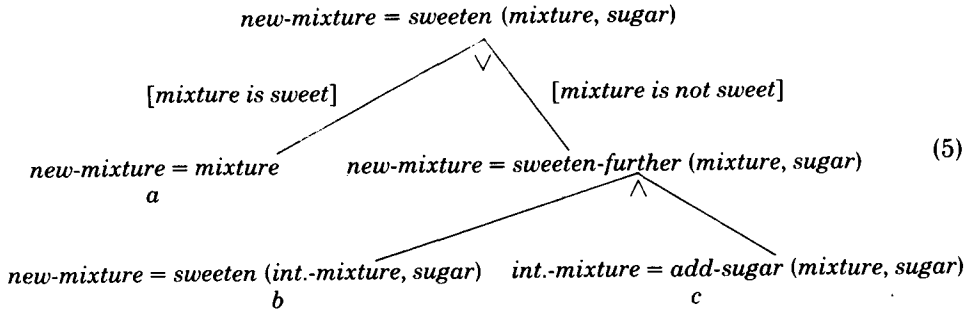


(4)

The decomposition then resumes, regarding each of the descendants as a root. The process gives rise to a tree with *and* and *or* nodes, and the decision as to where the decomposition stops depends upon the primitive operations at hand. In our example, the operation of adding a little sugar to the mixture might be primitive, and accordingly it is conceivable that

new-mixture = add-sugar (mixture, sugar)

will be one of the leaves of the tree. The power of iteration will be obtained by allowing leaves to be labeled not only with primitive operations but also with labels labeling (not necessarily immediate) ancestors, thus modeling procedure calls. For example, sweetening a mixture might be achieved by the following subtree:

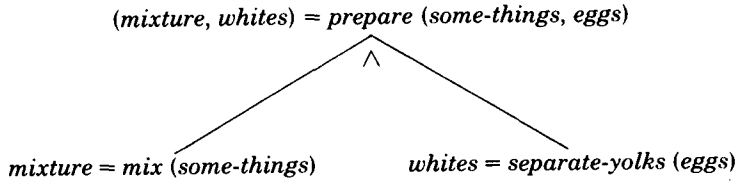


(5)

in which leaves *a* and *c* represent primitives and leaf *b* a recursive call to the root of the subtree. Clearly (5) is to act simply as a **while** statement; adding sugar is to be repeated as long as the mixture is not sweet. Note that on account of the outcome of an operation being different from its input (i.e., "=" denotes a definition, not an assignment) we do not write

mixture = add-sugar (mixture, sugar),

but rather introduce an intermediate variable, *int.-mixture*. Note also that just as the *or* node in (3) represents a nondeterministic choice but can be restricted (in the spirit of (5)) to be more like a deterministic **if then else,** so does an *and* node represent a clean parallel construct but can be restricted to be sequential as in (2). It will be perfectly legal to have a subtree of the form



(6)

in which clearly no ordering on the descendants is forced by the input–output pairs (as was the case in (2)). Still, (2) and (6) have in common the property that achievement of the goal in the root of the subtree is dependent upon achieving *both* subgoals in its descendants.

It will become evident that *and* nodes subsume sequential composition and parallel constructs (e.g., A;B, A ‖ B) and *or* nodes subsume nondeterministic choice and various versions of conditionals (e.g., A ∪ B, **if** P **then** A **else** B, **IF**

□ Q → B **FI** of [5], etc.). The point is that in the former cases both A and B are executed and in the latter only one is. The power obtained by leaves such as *b* in (5) will be the power of general, possibly mutual recursive calls and thus will be seen to subsume versions of the repetitive construct (e.g., **while** P **do** A, **repeat** A **until** P, **DO** P → A □ Q → B **OD** of [5], etc.).

We now turn to providing a more rigorous definition of the syntax and semantics of the language.

3. THE AND/OR LANGUAGE

We define a rather strict syntax for our language but later relax it somewhat by employing some helpful abbreviations. Our intent is to keep the basic constructs in the definition few and clear in order to simplify the semantics and later discussions.

Syntax

We have sets Σ, Δ, and Π of *variable*, *action*, and *test*, symbols, respectively. A *node-i.d.* s is a triple (f, a, b) where $f \in \Delta$ is the *label* of s, and a and b, called, respectively, the *output* and *input lists* of s, are finite tuples of elements of Σ. Similarly, an *edge-i.d.* t is a pair (p, a) where $p \in \Pi$ is the *test* of t, and a, the *argument list* of t, is also a finite tuple over Σ.

We now consider finite binary trees in which each node u is marked with a node-i.d., s_u, and each edge e with an edge-i.d., t_e. In addition, every internal (i.e., nonleaf) node carries an indication as to whether it is an *and* or an *or* node. The components of s_u and t_e are appropriately denoted by lab_u, out_u, in_u, $test_e$, and arg_e. In our examples we write a node-i.d. (f, a, b) as $a = f(b)$ and an edge-i.d. (p, a) as $p(a)$.

We freely use conventional terms for talking about trees, such as *root*, *leaf*, *path*, *ancestor*, and *descendant*. In addition, an immediate descendant (ancestor) of a node is called its *offspring* (*parent*), and the two offspring of an internal *and* (respectively *or*) node are *and-siblings* (respectively, *or-siblings*). When no confusion can arise we will apply set-theoretic operations to input and output lists, regarding them as multisets of the variables occurring in them. Thus, $in_u = \emptyset$ means that the input list of node u is the empty tuple.

A tree T is a *legal and/or program*, or *program* for short, if it satisfies the following requirements:

R1. One or more leaves and one internal node can have a common label, and in this case these leaves will be called *call leaves*. Every other leaf is a *primitive* one. Primitive leaves may also have common labels. No other pairs of nodes may have common labels. Any pair of nodes u and u' having a common label must satisfy $|in_u| = |in_{u'}|$ and $|out_u| = |out_{u'}|$.[1]

R2. For any variable $x \in \Sigma$ appearing in an input list in_u, exactly one of the following holds:
   (a) u is the root of T,
   (b) $x \in in_{u'}$, where u' is the parent of u,

[1] Our intention here is that any number of such sets (of leaves and internal node having a common label) is allowed. "Call leaf" is the general name we give a leaf in any one of these sets.

(c) $x \in out_{u'}$, where u' is the sibling of u, their (common) parent u'' is an *and* node, and $in_{u'} \subset in_{u''}$.

R3. For any variable $x \in \Sigma$ appearing in an output list out_u, exactly one of the following holds:
(a) u is a leaf of T,
(b) u is an *or* node and x appears in the output lists of both of the offspring of u,
(c) u is an *and* node and x appears in the output list of exactly one of the offspring of u.

R4. For any edge e leading from node u to (its offspring) u', $arg_e \subset in_u$. Furthermore, for any two edges e and e', if $test_e = test_{e'}$, then $|arg_e| = |arg_{e'}|$.

R5. For any node u, $in_u \cap out_u = \emptyset$.

Requirement R1 specifies that each node is to represent a unique action labeled with that node's label. The exceptions are primitive actions which can occur in more than one place and the "procedure calls" which are modeled by *call* leaves. Two nodes having the same label are called *similar*. The pairs of input and output lists of similar nodes must agree in length.

Requirements R2, R3, and R4 formalize the ways in which inputs and outputs flow through the tree. By R2, inputs are handed down from parent to offspring but can also be produced (as an output) and handed over by an *and*-sibling. In the latter case, however, the sibling must be executable independently of the node in question. This situation then induces an ordering on the two siblings, thus making the *and* node a *sequential* one. By R3, outputs are initially produced by leaves and are handed up from offspring to parent. By R4, inputs are also handed down from parents to tests guarding their offspring. The way in which the inputs in_r of the root of the tree flow down through the tree to produce the final outputs out_r should now be quite clear.

A word about R5. Our language is not to be thought of as a conventional programming language in which the values of variables representing memory locations are modified; we do not allow "changing" the value of x as in x = f(x). Indeed, the meaning of x = f(y) is that (the unique value of) x is to be *equal* to that of f applied to y. Thus, our variables are not really variables at all but rather represent concrete, fixed data, much like the wires in a network. Our method of imposing this interpretation is to require that input and output lists be disjoint.

The following lemma, which can be proved easily from the definitions, shows that an *and* node is well behaved in the sense that its inputs are sufficient for the requirements of its offspring and its offspring in return loyally produce its outputs. (In the following we constantly assume that all trees are legal programs.)

LEMMA 1. *For any and node u with offspring u' and u'', the following hold:*
(*i*) *either* $in_{u'} \subset in_u$ *or* $in_{u''} \subset in_u$;
(*ii*) $out_u \subset (out_{u'} \cup out_{u''})$.

Similarly, the inputs of both offspring of an *or* node are supplied by the parent and the parent's outputs are supplied in return by each of the offspring.

LEMMA 2. *For any or node u with offspring u' and u", the following hold:*
(*i*) $(in_{u'} \cup in_{u''}) \subset in_u$,
(*ii*) $out_u \subset (out_{u'} \cap out_{u''})$.

It is important to observe that a tree can be easily checked for legality; properties R2–R5 suggest a straightforward linear algorithm, whereas R1 can easily be tested in time proportional to the square of the number of nodes in the tree. Two examples of legal nodes are (7) and (8) below, and an example of a legal program appears in the appendix.

Semantics

(This is a semiformal definition only.) A program T obtains a well-defined meaning relative to an *interpretation* I. Each variable $x \in \Sigma$ is interpreted in I as ranging over a certain domain D_x. Some typical domains are the integers, the natural numbers, sets of numbers, character-strings, arrays, and lists of numbers or strings.

Let $\Delta_0 \subset \Delta$ be the set of action symbols labeling primitive leaves. By requirement R1, the number of inputs and the number of outputs in any node-i.d. of which a given $f \in \Delta_0$ is a label, are fixed. The interpretation I assigns to each $f \in \Delta_0$ a $(|in_u| + |out_u|)$-ary relation, where u is some primitive leaf with $lab_u = f$, in the obvious way; each component is an element of the domain corresponding to the variable in that position. Certainly then, for the interpretation to be "good" we require that two variables appearing in the same position in the input (or output) lists of two similar nodes have to range over the same domain. By convention, whenever the conditions imposed on an interpretation I are not met, we assign the empty relation as the meaning of T in I. For example, if we have two primitive leaves l and l' with i.d.'s $(f, (x, y), (a))$ and $(f, (v, w), (d))$ then (for an interpretation to be good) we must have $D_x = D_v$, $D_y = D_w$, and $D_a = D_d$. The action symbol f is then assigned a subset of $D_x \times D_y \times D_a$.

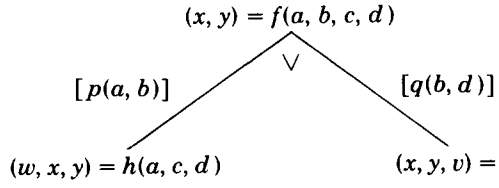
As a notational convenience, the meaning of a primitive leaf l, which we denote $m(l)$, is regarded as a binary relation over the output/input components, using a semicolon for separation. Thus, in the example above we write (x, y; a) $\in m(l)$, and $m(l)$ is thought of as a subset of $(D_x \times D_y) \times D_a$. The intuition is that (x, y; a) $\in m(l)$ if and only if l can produce (x, y) as an output list from the input list (a).

An interpretation I also assigns an $|arg_e|$-ary predicate to each test $p \in \Pi$ appearing in T, where e is an edge of which p is the test. Here too, when two edges have common tests, we adopt a condition for I to be "good," analogous to the one for node-i.d.'s with common labels.

We can now extend the above definition to give the meaning, under I, of any node u in T. In other words, I fixes domains and meanings for primitive actions and tests, and our semantics extends these to provide a meaning for T. First, consider the case in which T is free of call leaves. In this case it suffices to show how to obtain the meanings of *and* and *or* nodes given the meanings of their offspring. The meaning m(T) of a program T is then taken to be the meaning m(r) of the root of T. Thus, our semantics "extends I upward" in the tree.

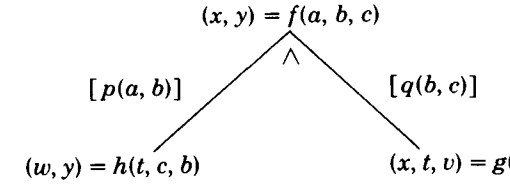
We omit a detailed general definition in favor of the definition for the following

representative examples of *and* and *or* nodes. (Note: We use m(f) and m(u) interchangeably to denote the meaning of a node u labeled by f.)



(7)

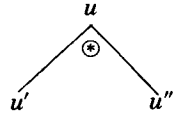
$$\mathrm{m}(f) = \{(\mathrm{x, y; a, b, c, d}) \mid (\exists \mathrm{v})(\exists \mathrm{w})(((\mathrm{b, d}) \in \mathrm{m}(q) \wedge (\mathrm{x, y, v; a, b}) \in \mathrm{m}(g)) \vee ((\mathrm{a, b}) \in \mathrm{m}(p) \wedge (\mathrm{w, x, y; a, c, d}) \in \mathrm{m}(h)))\}.$$



(8)

$$\mathrm{m}(f) = \{(\mathrm{x; a, b, c}) \mid (\exists \mathrm{t})(\exists \mathrm{v})(\exists \mathrm{w})(((\mathrm{b, c}) \in \mathrm{m}(q) \wedge (\mathrm{x, t, v; a, b}) \in \mathrm{m}(g)) \wedge ((\mathrm{a, b}) \in \mathrm{m}(p) \wedge (\mathrm{w, y; t, c, b}) \in \mathrm{m}(h)))\}.$$

These can be seen to be very natural definitions if v, w, and t are thought of as being *local* to the tree, since they do not appear in the node-i.d. of f. The structure of the definition for a node of the form



then becomes

$$\boxed{\begin{array}{c} lab_u \text{ can produce } out_u \text{ from } in_u \\ iff \\ \text{(there exist locals)((guard for } u' \text{ ok} \wedge lab_{u'} \text{ produces } out_{u'} \text{ from } in_{u'}) \\ \circledast \text{ (guard for } u'' \text{ ok} \wedge lab_{u''} \text{ produces } out_{u''} \text{ from } in_{u''})) \end{array}} \quad (9)$$

where ⊛ can be ∧ or ∨.

Considering now the general case in which call leaves are present, the standard least-fixpoint semantics [2, 14, 15] is adopted. In our framework this can be described as follows: First, let T be a program in which all call leaves happen to be labeled with the label, lab_r, of the root of T. Let T′ be T with all those leaves relabeled with some new symbol, say f. Note that T′ is free of call leaves. Now, given a relation R of appropriate arity and structure, we obtain I′ from an interpretation I by having I′ assign R to the new symbol f. R is said to be a *fixpoint* of T if m′(r) = R, where m′(r) is the meaning of the root of T′ under I′. The meaning of r under I is now defined to be the set-theoretically smallest (appropriately typed) relation R which is a fixpoint of T. By a well-known theorem due to Knaster and Tarski (cf. [14]) this *least-fixpoint* exists, and in fact

is unique, when certain conditions (entailing the property of *continuity*) are met by the primitive operations of I. In particular, conventional primitive operations on numbers, character-strings, etc., all fall within this category. Again, whenever the conditions are not met, we let $m(r) = \emptyset$. Thus, the relation defined by a node which calls itself recursively is the least defined relation which is produced when it itself is taken as the relation achieved by the recursive calls.

For the more general case where call leaves correspond to arbitrary internal nodes, the slightly more complex notion of simultaneous fixpoints is used. Here, briefly, the set of k subtrees whose roots correspond to the k different labels on the call leaves of the program are considered simultaneously. One then obtains fixpoints consisting of k relations $R_1, \ldots, R_k$ such that, for each i, interpreting the occurrences (including internal ones) of the k labels as $R_1, \ldots, R_k$ in the ith subtree results in the root of that subtree being assigned R_i as its meaning when the usual propagation process is carried out. Then the least such fixpoint, in the appropriate extension of the subset ordering to tuples of relations, is adopted as the meaning of the k kinds of call leaves. More details about least-fixpoint semantics can be found in the references. It is worth noticing that a call leaf can be viewed as an abbreviation of the infinite tree obtained by repeatedly copying the subtree of its similar ancestor using fresh variables whenever necessary. This remark in fact illustrates the correspondence between least-fixpoint semantics and the copy-rule of [1].

Note that nonrecursive subroutines are a special case of call leaves: A subroutine S can be modeled by a subtree which is then attached to an *or*-node (anywhere in the tree) with guard **false**, so that it does not get executed except as a result of the occurrence of an appropriate call leaf. Note also that, since the value of a variable is not modified at all, our "parameter-passing mechanism," which is the term one might use to describe the process of associating the input and output lists in a call leaf to those of its corresponding similar internal node, can be thought of as call by value-result.

A general correspondence between context-free grammers and and/or graphs, of which our language can be seen to be a refinement and an application, has been pointed out by Hall [7]. Also, a similarly motivated method of writing programs dominated, as is ours, by subgoaling using recursive calls, has been described recently by Hehner [11].

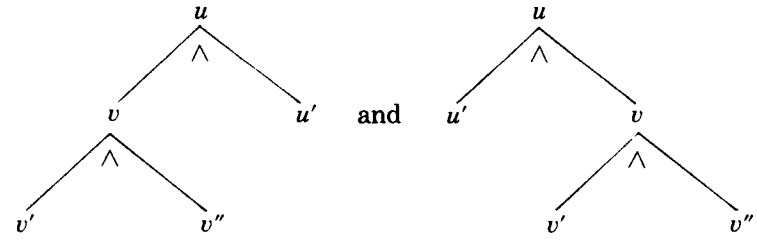
Abbreviations

We discuss some abbreviations useful in writing real and/or programs. We allow the omission of edge-i.d.'s; an edge without one is treated as being labeled with the test **true**, which has a fixed interpretation as the constantly true predicate. (In practice it turns out that tests are invariably attached to edges leading from *or*-nodes, in which case they act like conditionals or guards [5].)

We also allow tests to involve standard, say first-order, logical symbols with the understanding that interpretations are forced to interpret them in the standard way. For example, $\neg P$ is to be interpreted as the predicate which is the appropriate complement of P.

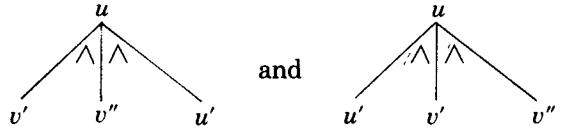
We allow using k-ary trees for $k \geq 2$, and the way in which nodes of out-degree

>2 are introduced is by abbreviating subtrees of the form



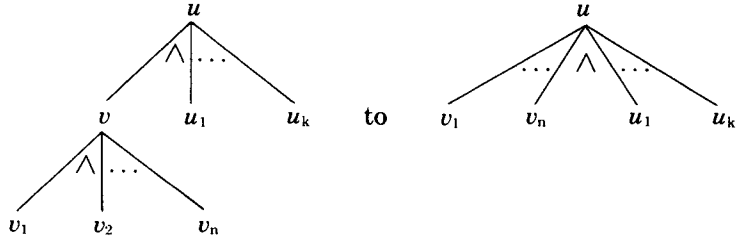
(10)

by the 3-ary *and*-nodes



(11)

and similarly for *or*-nodes. This can be done whenever the tree does not have a call leaf similar to v. Also, when the *and* nodes in (10) are of degree ≥ 2 they too can be collapsed (when no such call leaves are present) from, say



(12)

This way *and* and *or* nodes of arbitrary finite out-degree are obtained. One can easily prove the following result which confirms that these abbreviations are justified, in the sense that no semantic information is lost by adopting them.

LEMMA 3. *Let T be a tree which can be obtained from each of the legal and/or programs T' and T'' by abbreviating parts of them as k-ary nodes in the above manner. Then for any interpretaion I, $m(T') = m(T'')$.*

This is as far as we go in providing "official" shortcuts. However, for ease in writing and/or programs one might introduce additional ones such as abbreviating long lists of inputs by single symbols.

4. VERIFICATION OF AND/OR PROGRAMS

We describe the classical Floyd/Hoare invariant-assertion method [6, 12] as applied to and/or programs. The way in which partial correctness is established using this method is completely analogous to the calculation of the meaning of a program described in Section 3. As such, the method seems to render the and/or language a good tool for explaining the principles of program verification to the unknowing.

In the following, assume we are given a program T with root r labeled $x = f(y)$, and a fixed legal interpretation I. The *partial correctness* of T in I is defined with

respect to an assertion $P_r(x; y)$, to hold if and only if

$$(\forall x \forall y)((x; y) \in m(r) \supset P_r(x; y)). \tag{13}$$

This can be viewed as asserting that $m(r) \subset P_r$ for the *relations* $m(r)$ and P_r. (Note: The conventional definition of partial correctness with respect to a precondition R and a postcondition Q (cf. [12, 14]) is a special case, as observed by taking $P_r(x; y)$ to hold iff $R(y) \supset Q(x)$).

Thus one can think of proving the partial correctness of a program as showing that what indeed happens, i.e., $m(r)$, is at least as much as what we *think* happens, i.e., P_r. One can now conceive of a method in which every node u is annotated with an assertion P_u which is a relation of appropriate arity and type, and which captures our idea as to what u accomplishes. If we can then show $m(u) \subset P_u$ for every node u, the proof would be complete. Say P_u is a *good assertion at* u if $m(u) \subset P_u$.

Assume now that, having $P_{u'}$ and $P_{u''}$ attached to the offspring u' and u'' of a node u, we can calculate an assertion P_u which is good if $P_{u'}$ and $P_{u''}$ are. Clearly, if the tree is free of call leaves it would suffice to somehow annotate the leaves (all of which are primitive) and verify, appealing to $m(l)$ for this, that P_l is good for every leaf l. Then the method postulated above would be used to propagate good assertions up the tree until P_r is obtained. By our assumption this would establish that P_r is good, i.e., that T is partially correct with respect to P_r.

Before dealing with call leaves, we show that this goodness-preserving propagation is indeed possible, and is in fact completely analogous to the method for calculating the meaning of a node u given the meanings of its offspring. We describe it for the example nodes (7) and (8) of Section 3. Using P_f and P_u interchangeably to denote the assertion attached to a node u labeled with f, and assuming the offspring are labeled with P_h and P_g, we define, for cases (7) and (8), respectively,

$$\begin{aligned} P_f(x, y; a, b, c, d) \quad \text{iff} \quad & (\exists v)(\exists w)(((b, d) \in m(q) \wedge P_g(x, y, v; a, b)) \\ & \vee ((a, b) \in m(p) \wedge P_h(w, x, y; a, c, d))). \end{aligned} \tag{7'}$$

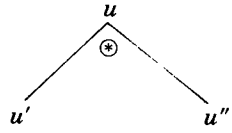
$$\begin{aligned} P_f(x, y; a, b, c) \quad \text{iff} \quad & (\exists t)(\exists v)(\exists w)(((b, c) \in m(q) \wedge P_g(x, t, v; a, b)) \\ & \wedge ((a, b) \in m(p) \wedge P_h(w, y; t, c, b))). \end{aligned} \tag{8'}$$

It is trivial to show that in both cases P_f is good whenever P_g and P_h are. Furthermore, this property is retained even when any assertion *weaker* than P_f of (7') or (8') is used. Thus, for example, if (7') produces the assertion $x = a! \wedge y = 0$, we might equally well use $x = a!$ alone for the annotation.

Turning to call leaves, since the relation we assign as a meaning to a node u with call leaves similar to itself is the *least* fixpoint, it is obvious that *every* fixpoint, when attached to u as an assertion, is a good assertion at u. The intuition is simple; a call leaf with the same label as u is asserted to behave on its inputs exactly as u would have behaved on the same inputs. It follows that the annotation method described above for trees free of call leaves extends to the general case: attach assertions to call leaves as well as to primitive ones, and propagate all assertions upward through the tree, making sure that when a node u which is similar to a call leaf l is reached, P_u and P_l are equal.

The method exhibited above can also be described as follows: A *full annotation* of a program T with respect to an interpretation I is one in which each node u is annotated with an assertion P_u of appropriate arity and type. A full annotation is said to be *good* if the following hold:

(a) For every primitive leaf l, $m(l) \subset P_l$.
(b) For every call leaf l with similar internal node u, $P_u = P_l$.
(c) For every node u of the form



the implication suggested by the following scheme holds:

(there exist locals) ((guard for u' ok $\wedge$ assertion for u' holds)

$\circledast$ (guard for u'' ok $\wedge$ assertion for u'' holds))

implies

assertion for u holds

where $\circledast$ can be $\wedge$ or $\vee$. One can then easily prove the following theorem, which amounts to the *soundness* of this proof method.

THEOREM 1. *A good annotation satisfies $m(u) \subset P_u$ for every node u. In particular, T is partially correct with respect to P_r.*

We remark that the assertions attached to call leaves are the inductive assertions of the Floyd/Hoare method, and their *invariance* is precisely the property required in (b) above. The reader familiar with the literature on logics of programs and program verification will notice that not treating the issue of the language in which the assertions are written permits us to work freely with relations and, for example, to use the term "equal" for assertions without having to define concepts such as logical equivalence and syntactic substitution in some formal logic. The question of *completeness* of the method (the formulation of which requires the notion of assertion language anyway) is avoided in this paper, as is the problem of specifying the restrictions on interpretations to be amenable to the least-fixpoint definition.

Notice that the frequently recommended discipline of developing program and proof simultaneously becomes very natural in the and/or language; a description of the intended behavior of each node can be given at the time of its construction, thus providing a full annotation which, if correct, should also be *good* in the sense of the above definition.

Similarly, *modular* or *hierarchical* verification, in which large components of a program are verified pending verification of smaller ones, is naturally carried out. Assumptions about the behavior of subtrees can be incorporated as assertions attached to the roots of those subtrees, and based upon this the larger tree can be

annotated and verified. Later the subtrees themselves can be reconsidered and their assertions verified independently.

These remarks, coupled with the simple analogy between (7), (8) and (7'), (8'), seem to indicate that the and/or language is a natural one for introducing the basic principles of program verification to a person with no prior knowledge of them. We will not go into known methods for proving total correctness, absence or presence of infinite loops, etc., but urge the reader familiar with such methods to observe the naturalness with which they too can be embedded within this framework.

The appendix contains an example of a program and a good annotation.

5. PRACTICALITY

In this section we comment on the virtues and limitations of our language from the viewpoints of the programming language designer and implementer, and the writer of large programs.

The two main drawbacks of the language seem to be its nontextual form and its rather unconventional control-flow-obscuring structure. The latter problem, discussed somewhat further in Section 6, is closely related to that of constructing a compiler or interpreter for the language. The former though, is merely technical and can be eliminated in a variety of popular ways, incorporating recent developments in the area of visual representation and text and data-structure editing.

A strange mixture of flexibility and restriction can be found in the language. On the one hand, we are given a rather large amount of freedom in choosing an implementation, as is evident, for example, from the fact that nonsequential *and*-siblings can be executed in parallel (and more generally, so can any two "independent" nodes). This though, as mentioned, places a larger burden on the implementer. On the other hand, the restrictions on the input and output lists eliminate many of the intricate and subtle problems with modern high-level programming languages, such as aliasing, scope and binding problems, parameter-passing mechanisms, and storage allocation. Again, this is at the expense of the implementer of the language and, if one wants, can be attributed to the inherent difference between a programming/specification language and a computer language (which is what the former becomes once implemented).

We now take the liberty of reporting some pleasant properties of the proposed language, encountered by the author (albeit subjectively) when writing a large and/or program. The program, a flowcharter capable of accepting inputs from many programming languages, is described in [9] and its details are unimportant here. The and/or tree itself appears in [10, app. II-5] and is of restricted form; *or* nodes are used only as **if** P **then else** (i.e., with P and $\neg$P labeling its outgoing edges), *and* nodes are not used for nontrivial parallelism, and call leaves are used only for simulating iterative loops (such as that in tree (5) of Section 2). In the terminology of programming theory, our program is *deterministic*, *sequential*, and *regular*.

Nevertheless, even with this restrictive language, we experienced considerable ease in designing the program. The requirement of having to label each node with a different action symbol quickly became an aid instead of a burden; the names we used proved valuable in remembering the functions that various

subtrees performed. Some examples are *add-one*, *advance*, *analysis*, *collate*, *count*, *finish*, *modify*, *next*, *parse*, *prepare*, *reduce*, *spread*, and *update*. Similarly, names for variables were chosen quite carefully, introducing multiply primed versions of them when needed.

As it turned out, the program, in each stage of its preparation, was extremely readable. The fact that at each stage the unfinished product looks just like a finished one, except that the primitive leaves are not yet primitive enough, helps considerably in the writing process. At various stages misjudgments were made as to how primitive to make the leaves, but the effort involved in, say, eliminating redundant subgoaling (for instance, when deciding that finding the first blank in a character string can be primitive, and that there is no need to subgoal down further) was negligible; all that was needed was to prune the subtree under the node *find-blank*.

Similarly, the effort involved in modifying a part of the program was not increased by having to worry about ruining other parts; subtrees can be removed and inserted, and it is only important to keep the input and output lists of the root of the subtree fixed. For example, suppose a change in the form of some output table is required. In most conventional programming languages, if the preparation of that table was not coded into an isolated subroutine, many problems can occur when attempting to modify some of the program text. In contrast, in an and/or program chances are that one will find a node labeled, say, *output* = *tabulate* (*data*). The subtree rooted in this node can then be modified without concern. In other words, in the and/or language "subroutines" are the rule and not the exception.

It was our pleasant (but again, subjective) experience to find that, in contrast to previous experience with conventional languages in which quite the opposite was true, months after the program was completed we were able to recomprehend any given part of it virtually at a glance. We are not claiming that these virtues are unique to our particular language. Certainly, employing any careful discipline of structured programming with stepwise refinement can produce the same pleasant results. Our point though is that the present language seems to *impose* this kind of discipline, and with a minimal amount of extra documentation, external devices, and programming aids.

We found, though, that at certain levels the and/or style became somewhat artificial and cumbersome when compared with conventional alternatives. For example, in our opinion, the and/or equivalent of a simple **do**-loop (say in PL/I) for finding the first blank character in a string is quite undesired. However, the and/or subgoaling process can be stopped whenever a node can be more succinctly and clearly defined in one's favorite programming language. That node can then be left as a primitive leaf in the and/or program and can be further specified elsewhere. Personally, in writing the flowcharter of [9, 10], and in further experiments, we acquired a reasonable sense of the level at which and/or programming was no longer appropriate for us. It seems that for any combination of programmer/large-program such a level exists and can be sensed by the programmer in the process of writing the large and/or program. We are confident that despite the inconvenience of having to write programs in treelike form (and having, therefore, to measure their size in square feet, or sometimes acres, instead of number of lines of code...), the task of producing the large program by the

programmer can be greatly eased by appealing to and/or programs until the level discussed above is reached.

We would also venture the opinion that and/or programs can be beneficially used to teach the basic principles of programming to students. The concepts of *and* and *or* subgoaling seem natural enough for even nonmathematically-inclined people to comprehend, thus capturing, as a bonus, the concepts of nondeterminism and a straightforward kind of parallelism. Also, the way in which recursion is embedded in the trees seems to give rise to a natural way of describing this, otherwise quite difficult, concept.

6. DIRECTIONS FOR FUTURE WORK

The most obvious project to be carried out is the implementation of a language such as this. As hinted earlier, such a task would obviously involve either a textual linear representation for the input tree or a two-dimensional method of entering it directly. Perhaps more substantially, a decent implementation will have to decide upon the sequence(s) of operations to be performed, i.e., the control flow of such a program. Note that while in conventional languages the control flow is usually given and is transparent to the compiler/interpreter but the data flow is obtained by some kind of analysis, here the situation is dual. The way in which the data flows through an and/or tree is more transparent than that of the control. Methods for optimizing and/or programs (in the sense, say, of using a small number of "real" variables to contain in succession the values of a large number of variables) should be developed. It might also be possible to develop a smooth interface with a mechanism enabling the programmer to define his own (abstract) data types.

Some comparative research is lacking in the present paper, comparing and/or programs with data flow procedures, applicative languages such as Lisp, parallel programming languages, networks, and decision trees. Some observations on the connection with the latter two will appear in a subsequent paper.

On a more theoretical level, there are questions which can be asked concerning the power of uninterpreted and/or schemes compared to various classes of program schemes considered in the literature. Also, various interesting measures of the *textual complexity* of and/or programs seem to justify investigation, such as the depth, width, and size of the tree. In this context, one can define the *and/or depth* of a tree as the maximal number of alternations of *and* and *or* nodes on paths from the root to a leaf, this measure having well-known counterparts in the other applications of alternation mentioned in Section 1. As a first result in this direction, we can show that with the addition of auxiliary variables, every and/or program can be equivalently written as one with and/or depth 1. Equivalently, that is, in the uninterpreted sense of having the same effect in all interpretations. This result too will appear in a subsequent paper.

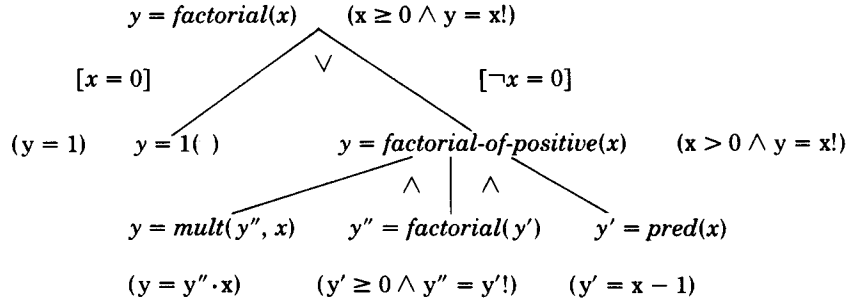
However, the most important question in our opinion is whether and/or programs will turn out in practice to be a real help to the everyday designer and writer of large programs.

APPENDIX

As claimed in the text, the virtues of and/or programs are *not* apparent in small toy programs, and only an example of a large real program could illustrate the

points we make. However, since a programming language cannot be introduced without at least one example of a program, and since we obviously cannot exhibit a large one here, we present the and/or tree of a simple recursive program for computing the factorial of a nonnegative integer, together with a good annotation. The reader is requested to view the example accordingly, i.e., as an illustration of some of the definitions, and not compare it with his own, simpler-looking version of a factorial program.

In the example, the interpretation consists of the domain of the natural numbers, and 1, *mult*, *pred*, and =0 are interpreted respectively as the number "one," the functions of multiplication and subtraction of 1, and the test "equal to zero." It is trivial to check the goodness of the annotation and hence the partial correctness of the program. The assertions attached to the nodes are parenthesized:



ACKNOWLEDGMENT

The research reported here was stimulated by, and to an extent is derived from, ideas implicit in a specification method which was described in Hamilton and Zeldin [8], and which is currently practiced on a commercial basis at Higher Order Software, Inc., Cambridge, Mass. The author is grateful for discussions with the staff members of H.O.S. The comments of the referees were most helpful.

REFERENCES

1. P. Naur, Ed. Revised report on the algorithmic language ALGOL 60, *Commun. ACM 6*, 1 (Jan. 1963), 1-17.
2. DeBakker, J.W., and Scott, D. A theory of programs. Unpublished notes, 1969.
3. Chandra, A.K., and Stockmeyer, L.J. Alternation. In *Proc. 17th IEEE Symp. on Foundations of Computer Science*, 1976.
4. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming*. Academic Press, New York, 1972.
5. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
6. Floyd, R.W. Assigning meaning to programs. In *Mathematical Aspects of Computer Science* (J.T. Schwartz, Ed.), *Proc. Symp. in Applied Mathematics, Vol. 19*, American Math. Soc., Providence, R.I., 1967, pp. 19-32.
7. Hall, P.A.V. Equivalence between AND/OR graphs and context-free grammers. *Commun. ACM 16*, 7 (July 1973), 444-445.
8. Hamilton, M., and Zeldin, S. Higher order software—A methodology for defining software. *IEEE Trans. Softw. Eng. SE-2*, 1 (March 1976), 9-32.

9. HAREL, D., NORVIG, P., ROOD, J., AND TO, T. A universal flowcharter. In *Proc. AIAA Computers in Aerospace Conf. II*, Los Angeles, Calif., Oct. 1979.
10. HAREL, D., AND PANKIEWICZ, R. A universal flowcharter. Tech. Rep. 11, Higher Order Software, Inc., Cambridge, Mass., Nov. 1977.
11. HEHNER, E.C.R. do considered od: A contribution to the programming calculus. *Acta Inform. 11* (1979), 287–304.
12. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.
13. KOZEN, D. On parallelism in Turing machines. In *Proc. 17th IEEE Symp. on Foundations of Computer Science*, 1976.
14. MANNA, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
15. PARK, D. Fixpoint induction and proofs of program properties. In *Machine Intelligence 5*, Edinburgh Press, Scotland, 1970.
16. SHOENFIELD, J.R. *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.
17. VON NEUMANN, J., AND MORGENSTERN, O. *Theory of Games and Economic Behavior*. Princeton Univ. Press, Princeton, N.J., 1953.
18. WINSTON, P.H. *Artificial Intelligence*. Addison-Wesley, Reading, Mass., 1977.
19. WIRTH, N. *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

Received January 1979; revised October 1979