June 1981

Report. No. STAN-CS-8 l-865

Toward a Unified Logical Basis for Programming Languages

by

Chih-sung Tang

Office of Naval Research
and
The Air Force Office of Scientific Rcscarch

Department of Computer Science

Stanford University
Stanford, CA 94305

Toward a Unified Logical Basis for Programming Languages

C hih- sung Tang
(Department of Computer Science, Stanford University)
(Institute of Computing Technology, Academia Sinica)

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.
>
> John McCarthy [11]

1. Foreword.

In recent years, more and more computer scientists have been paying attention to temporal logic, since there are many properties of programs that can be described only by bringing the time parameter into consideration. But existing temporal logic languages, such as Lucid, in spite of their mathematical elegance, are still far from practical. I believe that a practical temporal-logic language, once it came into being, would have a wide spectrum of applications.

XYZ/E is a temporal-logic language. Like other logic languages, it is a logic system as well as a programming language. But unlike them, it can express all conventional data structures and control structures, nondeterminate or concurrent programs, even programs with branching-time order. We find that the difficulties met in other logic languages often stem from the fact that they try to deal with these structures in a higher level. XYZ/E adopts another approach. We divide the language into two forms: the internal form and the external form. The former is lower level, while the latter is higher. Just as any logic system contains rules of abbreviation, so also in XYZ/E there are rules of abbreviation to transform the internal form into the external form, and vice versa. These two forms can be considered to be different representations of the same thing. We find that this approach can ameliorate many problems of formalization.

XYZ/E, like other temporal-logic languages, treats variables as temporal object, i.e. it looks upon them as potentially infinite vectors, each component of them corresponding to a time. Elephant[12] expresses a variable v as v(t) with the time parameter t explicitly indicated. But since this method of expression must make use of second-order logic to express assertions of a program; it seems too strong. Another approach is to represent the value of v as Apply(v, t). It is now within first-order theory. However we choose a third approach, the modal-logic approach, and we express Apply(v,t) as $\#v$ and Apply(v,t + 1) as $\circ\#v$. Here, "$\circ$" is an operator meaning *next time*. The symbol v is only a name and has no value; only $\#v$ or $\circ\#v$ has a value [13].

With this convention, the statement expressed conventionally as "$v \leftarrow$ v + 1" is expressed by "$\circ\#v = \#v + 1$." Control flow can be handled similarly; thus, the statement *goto* x is expressed in XYZ/E by $\circ\#\mathrm{lb} = x$, where lb is a special variable in the language used to indicate labels in control flow. The proposition that we are at label x in a program can be expressed by the equation #lb = x. These are the basic steps in expressing the statements of a conventional language as logical formula in XYZ/E.

This research was supported in part by the Office of Naval Research under contract N00014-76-C-0687 and in part by the Air Force Office of Scientific Research under contract AFOSR-81-0014.

In fact, the logical structures in both data and statements are identical; arrays correspond to loops, records to compound statements, pointers to goto statements, etc. Starting with a few basic lower-level items, we can construct complicated data and control structures with the same logical mechanism.

To make this possible, we create a new naming mechanism. A name has three parts: One is the *type symbol* which represents the type of the variable. Since XYZ/E is a many-sorted logical system, this part is necessary, but it can be omitted in abbreviation. The second part is a conventional identifier. The third is the name *index* which is used to express the component of a vector or a record. For example, *Iabc_(3)_age* represents a data item which is the age component of a record which is the third component of the array *abc*. This data item is of type integer, which is expressed by type symbol I. For the component of an array or loop, we can use a name *scheme* instead of a label, *e.g. abc_(#Kabc)*, where *#Kabc* is an integer variable which can have as its value the positive integers in the process of looping. Thus *abc_(#Kabc)* can produce different labels of the same pattern.

The two kinds of control structures have not been clearly distinguished in conventional languages: the local control structures, which are the control of information flow within one statement, and the total control, which expresses the information flow among modules and the whole program. In accordance with the differences in total structures, we divide the language into three layers: the flowchart structure, the module structure, and the nondeterminate structure (i. e. Petri net structure). Each structure is an extension of the previous one, but with more data types and control forms. The whole language is divided into XYZ/E0, XYZ/E1, and XYZ/E2.

With these conventions, XYZ/E can be used not only as a logic system to express the assertions and properties of the programs, but also as a practical programming language which can express any algorithm expressible by conventional languages.

Thus, XYZ/E is meaningful not only in program verification but in other areas, such as formal semantics. In my opinion, the essence of programming languages and systems is operational, so an operational approach to formal semantics looks more natural; however systems such as VDL involve details which become voluminous; while the denotational approach, such as the Scott-Strachey system, is more elegant, their treatment of the higher-level control structure, i.e. continuation, seems unnatural since the operational aspects of the system have been twisted. But in XYZ/E the operational characteristics of the language are concentrated into the temporal transition of the time parameter and all the rest are static. As a logic system, its formal semantics can be as easily defined denotationally as in any logic system; on the other hand, it is an assembly-like language, so that it is easy to construct a compiler-like transformation to map the semantics of higher-level language into this language. The formal semantics of any higher-level language can be expressed in this two-level way. Importantly, the formal semantics of a language becomes tightly connected with its compiler. Many people have shown a strong interest in using the compiler as the formal semantics of a higher-level language. They have failed because the semantics of the language to which the compiler is to map is not easily formalized. But XYZ/E, as a logic language does not have this defect; its own semantics can be easily formalized denotationally. We can use XYZ/E to describe the formal semantics of other programming systems. There is yet another fact often neglected: for almost all formal semantics methods, using their symbolisms to describe the semantics of a programming system is a hard job. Often, it is more difficult to do that than to write a medium-size compiler. But to use XYZ/E to describe the formal semantics of a language is identical to writing the semantic routines for its compiler.

With certain extensions, XYZ/E is also suitable to serve as an abstract specification language. It can also be used as a formal means of describing the process of hierarchical programming. This is another major application of XYZ/E, which we discuss in [16].

XYZ/E has yet other applications. It can be used as a portable intermediate language. Since it can be used to describe the semantics of compilable languages and is also portable to different machines, it has the characteristics of an 'UNCOL' in some restricted sense. Since both transformations from higher-level languages to XYZ/E and the inverse transformations exist, it may be used for source-to-source transformations [1].

Since it treats data structure and control structure in a similar way, this language can remember past history. Its application to data-base management systems is strongly expected.

Not only does XYZ/E have various practical applications, it also has a theoretical impact. As is well known, most higher-level programming languages can be roughly divided into two kinds: applicative and

algorithmic; LISP is representative of the first, and representatives of the latter include Algol 60, Fortran, etc. It has been widely agreed upon that λ calculus, or more generally, theory of computability, is the logic foundation of applicative languages; but what theory can be considered as the logic background of algorithmic languages? No satisfactory answer seems to exist yet. A natural conclusion after the overview presented above is that most probably, temporal logic can serve this purpose. The external form of XYZ/E can be considered as a model, and its internal form as a kind of intermediate representation.

2. XYZ/E0, Internal and External Form of The Language.

A *computer system* consists of different layers of an *abstract machine.* Each layer has its super *control structure,* which is the basic framework to control the flow of information within that layer. The most basic super control structure is flowchart structure. XYZ/E0 is a language which uses this structure.

The following are the features of XYZ/E0, and are also the common features of the whole XYZ/E family.

(1) Names:

All names used to represent the entities in the system consists of three parts concatenated consecutively: type symbol, name root and index part, which are defined as follows:

```
<name> ::= <type symbol><name root><name index>
<type symbol> ::= <data type> | <label>
<data type> ::= <basic type> | <structured type>
<basic type> ::= I | C | B | <extensible>
```

Hereafter, we use the convention <extensible> to mean that it is subject to possible extension.

```
<structured type> ::= V<data type> | R{<data type sequence>}
        | P<data type> | <extensible>
<data type sequence> ::= <data type> | <data type sequence>,<data type>
<name root> ::= <identifier>
```

Here *identifier* is used in the conventional sense, but the letters in it are limited to small Roman letters.

```
<name index> ::= <empty> | <name index>-<node>
<node> ::= <identifier> | <integer> | (<integer>)
<label> ::= L
```

We use the capital letters I, C, *B, V, R, P* to represent the types integer, character, boolean, vector, record, and pointer respectively. The index node *identifer* or *integer* is used to denote the component of **a** record or a compound statement; (<integer>) is used to denote the component of a vector or a loop. We use the Greek letters "∂, α, π" as meta symbols to represent <type symbol>, <name root>, and <name index> respectively; we also use λ to represent $\alpha\pi$. In order to be more readable, we also use x, y, *z,* u, *v,* w to represent any name if its type symbol is not L .

Thus $VR\{I,C,I\}abc$ means that *abc* is a vector whose element; are of the type record $R\{I,C,I\}$ with *I, C,* I as the types of its three subfields. *Iabc_(3)_age* is the data item of the record of the vector *abc* which -is labeled age.

We still need the concept of name *scheme.*

```
<name scheme> ::= <type symbol><name root><name sch index>
<name sch index> ::= <empty> | <name sch index>-<identifier>
        | <name sch index>-<integer> | <name sch index>-(<counter>)
<counter> ::= #K<identifier> | o<counter>
```

Every counter is always of type *I.* If x-y is a name scheme, and y does not contain symbol "_", we say that y is the *rightmost* or *last* node of x_y. A name scheme is an expression. As an example, *abc_(#Kabc)* **is a** name scheme, *#Kabc* is the counter corresponding to *abc,* and *(#Kabc)* is the rightmost node of this name scheme.

(2) Formation rules:

i) Variables:

General variables are those variables whose values are time dependent. We distinguish its name from its value. If $\partial\lambda$ is a name whose type symbol is not L then $\#\partial\lambda$ is a general variable Among general variables, there is one kind of variable whose value does not change with time. We call them *basic variables.* If v is a name, $.v$ is a basic variable.

ii) Constants:

Constants are expressed as in conventional formal languages. Thus, 0, 1, 2, . . . are used to represent integers, character strings are quoted with double quotation marks, T and F are used to represent true and false, etc. A name is a special case of a character string but with the quotation marks omitted.

iii) Expressions:

(1) every constant of type I or C is an expression.
(2) every basic variable of type I or C is an expression.
(3) every name scheme is an expression whose value is a name.
(4) if e is an expression of type ∂, where ∂ is I or C, and @ is an unary operator, then $@e$ is also an expression of type ∂.
(5) if e_1, e_2 are expressions of type ∂, where ∂ is I or C, and @ is a binary operation, then $(e_1 @ e_2)$ is also an expression of type ∂.
(6) if e is an expression then $\circ e$ is an expression.

iv) Logical formulas:

(1) any general variable or basic variable of type B is a logical formula.
(2) if e_1 and e_2 are two expressions both of type I or C, then $e_1 = e_2$ is a logical formula called an equation.
(3) there are some special formulas denoted by special names:
  (i) T and F are logical formulas which are called logical *constants*.
  (ii) A type symbol without a root and an index following it is called an *allocational formula*. Every allocational formula is a logical formula. Semantically, allocational formulas are equivalent to T, but they have a different pragmatic meaning (which is beyond what we are going to formalize in our system). They are introduced into the system in order to omit the type symbols in the names of the variables.
(4) if P, Q, and R are logical formulas, then so are $\neg P, P \vee$ Q, PA Q, $P \supset$ Q, $P \equiv$ Q, and IF P THEN **Q ELSE** R.
(5) if P is a logical formula, and x is the name of a variable of type I or C, then $\forall x P$ and $\exists x P$ are logical formulas.
(6) if P is a logical formula, then ❑ (*P), O(P),* and *o(P)* are logical formulas. (These three operators are read as necessary, *possible*, and *nexttime* respectively).
(7) if P and Q are logical formulas, then $(P)\mathcal{U}(Q)$ is also a logical formula. (It is read as *until.)* [8].

Among the equations, there are some special forms:

(1) An equation of the form $\circ\#x = e$ (or $e = \circ\#x$) is called an assignment equation if there is no o occurring in the expression e and $\#x$ is a general variable Here $\circ\#x$ is called the *next-time* side and e the *present-time* side of the equation.
(2) There is a special general variable #lb whose value is always a name. It always occurs in the equation of the form: #lb = e or o#lb = e, where e is a name or name schema. It is used to express the control flow of the whole formula. We give the special name *lb equation* to them. (#lb = e is called a *present-time lb equation* and o#lb = *e a next-time lb equation).* There is a special form $\circ\#$lb = stop which is called the *stop equation*. Hereafter we abbreviate **IF** *P* **THEN Q ELSE** F as **IF** *P* **THEN** Q.

Logical formulas are the well-formed formulas in our system. From a programming point of view, we have a special interest in well-formed formulas of the form:

where each $A_i, i = 1, \ldots, n$ has one of following two forms:

(i) **(IF** #lb $= e_i$ **THEN** $\ll Q_i \wedge \gg \circ$ #lb $= e_j$)

(ii) **(IF** #lb $= e_i$ **THEN (IF** P_i THEN $\ll Q_{i1} \wedge \gg \circ$ #lb $= e_{j1}$ **ELSE** $\ll Q_{i2} \wedge \gg \circ$ #lb $= e_{j2}$))

where <<X>> inplies that X may be empty. e_i, e_j are names or name schema. No lb equations are allowed to occur in P_i, Q_i and no allocational formulas are allowed to occur in P_i. In addition, no more IF. . . THEN... **ELSE.. .** form conditional statements are allowed to occur in P_i and Q_i. A formula of the above form satisfying the following condition is called a program (see 'frame problem' in [13]):

For any general variable, say $\#\partial\lambda$, an equation of the form $\circ\#\partial\lambda$ = e is called an *updating equation,* It has a special case, $\circ\#\partial\lambda = \#\partial\lambda$, which is called a *redundant equation.* Let $\#\partial_1\lambda_1, \ldots, \#\partial_m\lambda_m$ be all the general variables occurring in the program. Then for each $i = 1,\ldots, n$, the Q_i, or Q_{i1} and Q_{i2} must have an updating equation for each general variable as a conjunctive component. Redundant equations can be omitted.

We always express the program in another, more readable, form:

(1) we express (i) as "#lb = e $\Rightarrow\ll Q_i\wedge\gg$ o #lb = e_j."

(2) we express (ii) as "#lb = e A $P_i\Rightarrow\ll Q_{i1}\wedge\gg\circ$ #lb $= e_{j1}$; #lb = e $\wedge\neg P_i\Rightarrow\ll Q_{i2}\wedge\gg$ o $\#lb = e_{j2}$."

(3) We change the outermost parentheses of the program into square brackets.

A program is changed into the form: •I $[R_1\Rightarrow S1;\ldots;R_n\Rightarrow$ S,]. Each $R_i\Rightarrow S_i$ is called a conditional element. A program is regular if for any e_j the next-time lb equation $\circ\#lb = e_j$ occurs, then there must be a l $(1 \leq l \leq n)$ such that the present-time lb equation #lb = e_j occurs in R_l. We consider only regular programs.

Also for readability, we divide a program into several blocks by using square brackets to group conditional elements. A *block symbol* appears before the brackets. These block symbols are used to indicate the access rights for those names declared (i.e. occurring in a present-time lb equation) in the block. There are five kinds of blocks:

(1) $\%i[\ldots]$ is used to group the declaration of input basic variables. The variables in it must have initial values and these values cannot be changed within the program.

(2) $\%o[\ldots]$ is used to group the declaration of output basic variables. The variables declared in it must have a value which can be accessed outside the program.

(3) $\%io[\ldots]$ it is used to group the declaration of input-output general variables. The variables declared in it must have initial values, but their values can also be changed within the program and accessed outside the program.

(4) $\%v[\ldots]$ is used to group the declaration of local general variables.

(5) %a[. . .] is used to group the algorithm.

The syntax of an EO program can be described as

```
<EO prog> ::= ❑ [ <%i part> <%o part> <%io part> <%v part> <%a part> ]
<%X part>::=<%X block> | <empty>
```

X represents i, 0, io, v, or a.

```
<%X block> ::= %<X>[<conditional element seq>];
<X> ::=i | o | io | v | a
<conditional element seq> ::=<conditional element> |
        <conditional element seq>;<conditional element>
<conditional element> ::=
        <present lb eq><cond part> ⇒ <act part><next lb eq>:
<present lb eq> ::= #lb = <name scheme>
<next lb eq> ::= ∘<present lb eq>
<cond part> ::= ∧<logical formula> | <empty>
<act part> ::= <logical formula>∧ | <empty>
```

Example 1: The integral square root of a given non-negative integer.

The flowchart for this problem is shown in Figure 1.

By following the steps shown in the flowchart, we obtain a XYZ/E0 program. We change the assignments into assignment equations and use lb equations to express the flow of control:

❑ [#lb = *sqrt* ⇒ ○#lb = *Im*;
%*i*[#lb = *Im* ⇒ *I* ∧ ○#lb = *In*];
%*o*[#lb = *In* ⇒ *I* ∧ ○#lb = *Ik*];
%*v*[#lb = *Ik* ⇒ *I* ∧ ○#lb = *Ip*;
#lb = *Ip* ⇒ *I* ∧ ○#lb = *l1*];
%a[#lb = *l1* ⇒ ○#*Ik* = 0 ∧ ○#*Ip* = 1 ∧ ○#lb = *l2*;
#lb = *l2* ∧ #*Ip* > .*Im* ⇒ ○#lb = *l4*;
#lb = *l2* ∧ #*Ip* ≤ .*Im* ⇒ ○#lb = *l3*;
#lb = *l3* ⇒ ○#*Ip* = #*Ip* + 2*#*Ik* + 3 ∧ ○#*Ik* = #*Ik* + 1 ∧ ○#lb = *l2*;
#lb = *l4* ⇒ .*In* = #*Ik* ∧ ○#lb = **stop**]]

We have included the type symbols in this example. We will omit them in future examples.

Example 2: Binary search in an array of integers. We search for the integer stored in .obj. The flowchart is Figure 2, and the XYZ/E0 program follows.

❑ [#1 b = *binarysearch* ⇒ ○#lb = *Kar*;
%*i*[#lb = *K ar* ⇒ *I* ∧ ○#lb = *dpt*;
#lb = *dpt* ⇒ *I* ∧ ○#lb = *l1*;
#lb = *l1* ⇒ ○#*Kar* = 1 ∧ ○#lb = *ar*;
#lb = *ar A #Kar* ≤ .*dpt* ⇒ ○#lb = *ar_*(#*Kar*);
#lb = *ar_*(#*Kar*) ⇒ *I* ∧ ○#lb = *l2*;
#lb = *ar* ∧ #*Kar* > .*dpt* ⇒ ○#lb = *obj*;
#lb = *l2* ⇒ ○#*Kar* = #*Kar* + 1 ∧ ○#lb = *ar*;
#lb = *obj* ⇒ *IA* ○#lb = *place*];
%*o*[# 1 b = **place** ⇒ *I A* ○# 1 b = *low*];
%*v*[#lb = *low* ⇒ *I* ∧ ○#lb = *high*;
#lb = *high* ⇒ *I* ∧ ○#lb = *mid*;
#lb = *mid* ⇒ *I* ∧ ○#lb = *l3*];
%*a*[#lb = *l3* ⇒ ○#*low* = 1 ∧ ○#*high* = .*dpt* + 1 ∧ ○#lb = *l4*;
#lb = *l4* ∧ #*low* < #*high* ⇒ *o#mid* = [(#*low* + #*high*)/2] *A* ○#lb = *l5*;
#lb = *l4* ∧ #*low* ≥ #*high* ⇒ ○#lb = *l6*;
#lb = *l5 A .obj* > .*ar_*(#*Kar*) ⇒ ○#*low* = *#mid* + 1 ∧ ○#lb = *l4*;
#lb = 15 A .obj ≤ .*ar_*(#*Kar*) ⇒ ○#*high* = *#mid A* ○#lb = *l4*;
#1 b = *16 A .obj* = *.ar-(#high)* ⇒ .*place* = *#high* A ○# 1 b = *l7*;
#lb = *l6* ∧ *.obj* ≠ *.ar_*(# *high)* ⇒ .*place* = *0* ᴀ ○#lb = *l7*;
#lb = *l7* ⇒ ○#lb = stop]]

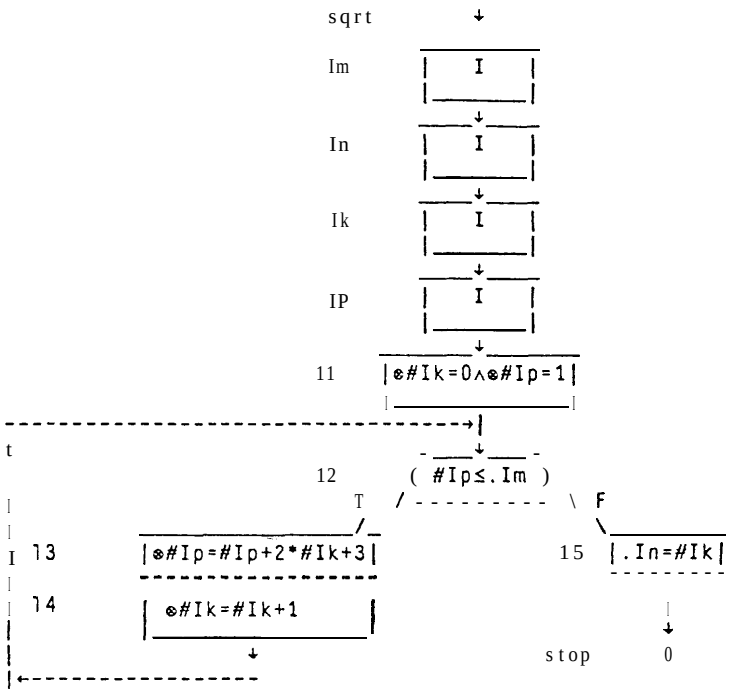
sqrt
Im
In
Ik
IP
11
⊛#Ik=0∧⊛#Ip=1
t
12
(#Ip≤.Im)
T
F
13
⊛#Ip=#Ip+2*#Ik+3
14
⊛#Ik=#Ik+1
15
.In=#Ik
stop
0

Figure 1

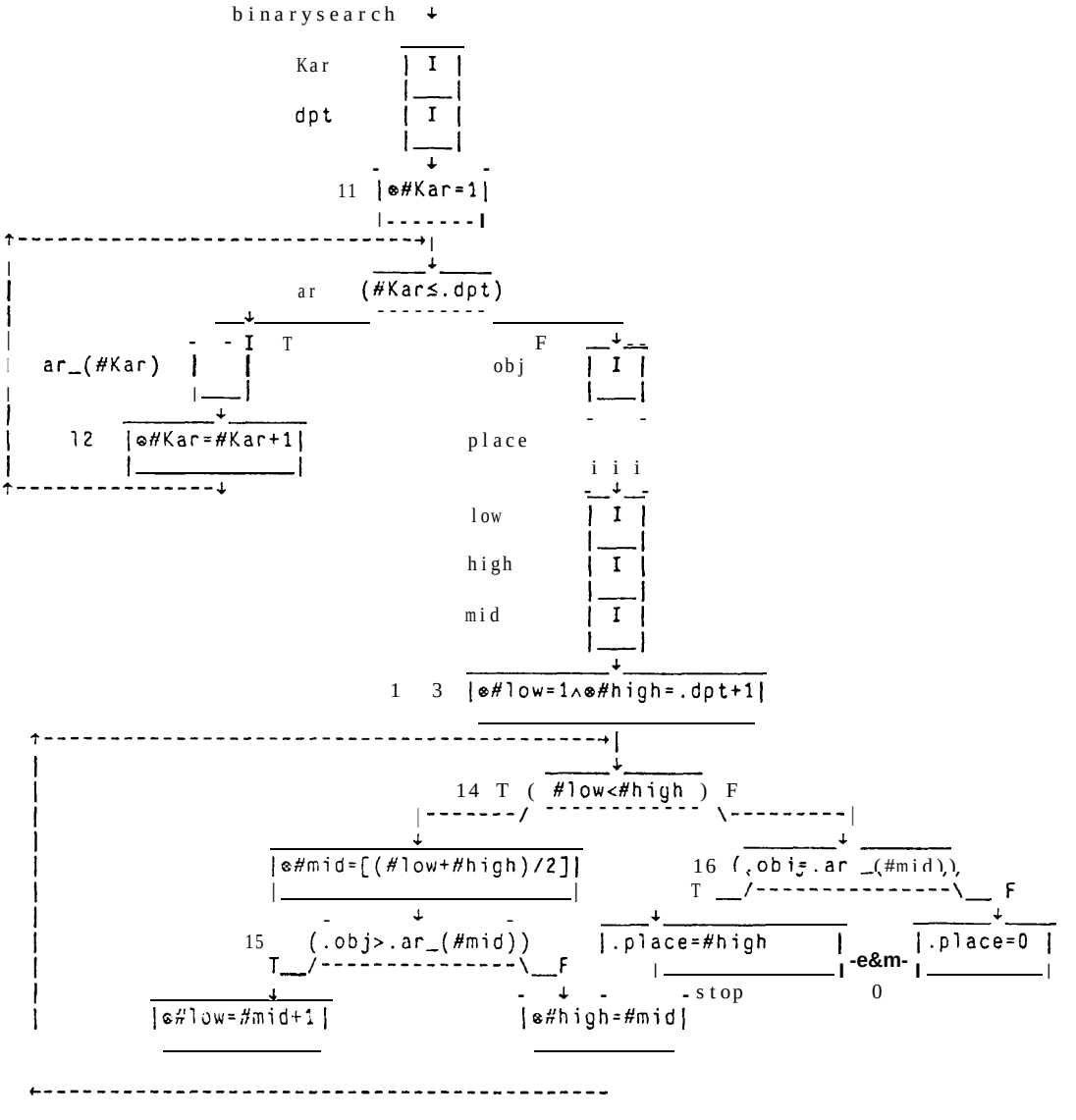

Figure 2

XYZ/E uses the same control structure for variable declarations and algorithms. In a compiler system, they are realized at different times; for an interpretive system, both are elaborated at the same time.

There are too many occurrences of lb equations in an XYZ/E program. It is easy to see that there are ways to simplify the representation. The result of this simplification is the external form of the language.

(1) Within each conditional element, the rightmost (next) lb equation "○#lb = ∂λ" is abbreviated as "↑ ∂λ".

(2) Within each conditional element, the leftmost (present) lb equation "#lb = ∂λ" is abbreviated as "∂λ :". If there is no formula between this leftmost lb equation and the "⇒", then the "⇒" sign can be omitted.

(3) For any rightmost lb equation, if the name in it is the name occurring in the leftmost lb equation of next conditional element, then this rightmost lb equation can be omitted; if after the omission, the conditional element becomes a formula of the form "$\partial\lambda\Rightarrow;$" or "$\partial\lambda$:;", then the semicolon can be omitted.

(4) For any sequence of formulas after the above abbreviation, say "$A_1;\ldots;A,$", if they do not contain the symbol "$\Rightarrow$", and all except the last do not contain formula of the form "$\uparrow x$", then they can be grouped by a pair of square brackets.

(5) For any leftmost lb equation "$\partial\lambda$:", if no rightmost lb equation except its immediate predecessor has its name occurring in its right side, then the "$\partial\lambda$:" or "#lb = $\partial\lambda$" together with a "A" sign occurring in its right side can be omitted, unless this lb equation occurs more than once.

(6) If two conditional elements have the same leftmost lb equation, but have contradictory conditions on the left of the sign "$\Rightarrow$", i.e. they have the form: "$\partial\lambda : P\Rightarrow Q_1;\partial\lambda:\neg P\Rightarrow Q_2;$" the second leftmost label can always be omitted.

(7) Any formula of the form "X : $P\Rightarrow$ Q $\Rightarrow R$" can be abbreviated as "$\lambda : P$ A Q $\Rightarrow R$"; Similarly, "$\lambda : P\Rightarrow Q\Rightarrow R,;\ \neg Q\Rightarrow R_2;$" can be abbreviated as "$\lambda : P$ A Q $\Rightarrow R_1; P$ A $\neg Q\Rightarrow R2;$".

(8) For each vector x, we always assume that there is a counter (i.e. a general variable of type I) defined together with it; its name always starts with a capital K and is followed by the name of this vector. This kind of counter is always initialized with the value 1.

(9) The type symbol of a name is omitted.

All these abbreviations can be easily restored mechanically. With these abbreviations, the Examples 1 and 2 above become:

Example 1 (cont.)

```
❑ [ sqrt :
  %i[m : I];
  %o[n : I];
  %v[k : I;
     :I];
  %a[o#k = 0 ∧ ∘#p = 1;
       12 :#p > .m ⇒↑14;
           #p ≤ .m ⇒ ∘#p = #p + 2*#k + 3 ∧ ∘#k = #k + 1∧ ↑ 12;
       14 : .n = #k ↑ stop]]
```

Example 2 (cont.)

```
❑ [ binarysearch:
  %i[dpt : I;
     ar : #Kar ≤ .dpt ⇒ [ar_(#Kar): I; ∘#Kar = #Kar + 1∧↑ ar] ;
          #Kar > .dpt ⇒ obj : I];
  %o[place : I];
  %v[low : I;
     high : I;
     mid : I];
  %a[∘#low = 1 ∧ o#high = .dpt + 1;
       14 : #low < #high ⇒∘#mid = [(#low + #high)/2]∧↑ 15;
            #low ≥ #high ⇒↑16;
       15 : .obj > .ar_(#Kar) ⇒∘#low = #mid + 1∧↑ 14;
            .obj ≤.ar_(#Kar) ⇒ o#high = #mid∧↑ 14;
       16 : .obj = .ar_(#high) ⇒.place = #high∧↑17;
            .obj ≠ .ar-(#high) ⇒.place = 0∧↑17;
       17:↑ stop]]
```

3. XYZ/E1.

XYZ/E0 introduced above is confined in the framework of flowchart structure, and thus cannot express subroutine calls, especially recursive ones. This can only be done in a higher layer super-control structure. This constitutes the basic novelty of XYZ/E1. XYZ/E1 is extented from XYZ/E0 with following new features:

A general variable of type stack is denoted by a name whose type symbol is S concatenated with the type symbol of its elements. The allocational formula for stack is "S <elementtype> ", where <elementtype> is the allocational formula of its elements.

There are several operations that can be performed on a stack, e.g. "push(<stack> ,<element>)", "pop(< stack>)", "top(< stack>)", and "depth(< stack>)". These operations are defined by well-known axioms.

We also introduce the concept of procedure and function modules. For XYZ/E, we add the following extentions:

```
<E1 prog>::=□[ <main> <module part> ]
<main>::= <%i part> <%o part> <%io part> <%v part> <%a part>
```

Now "<main>" is just as in "< EO prog >"and no more illustration is needed.

```
<module part> ::= ;<module seq> | <empty>
<module seq> ::=<module> | <module seq>; <module>
<module> ::= <module symbol>[ <module main> <module part> ]
<module symbol> ::= @p | @f | . . .
<module main> ::= <%i part> <%o part> <%io part> <%v part> <%a part>
```

We introduce two kinds of modules: One, with "%p" as its module symbol, we call procedure; the other, with "%f" as its module symbol, is called *function*. For the external form of these two kinds of modules, we adopt the convention that the "%i part", "%o part", and "%io part" of these modules are moved to the position following the name of the module and grouped by parenthesis as a parameter part, as in conventional languages. In order to distinguish among these three different parts, the block symbols "%i", "%o" and "%io" are put before their corresponding group of variables. For functions, the output variable will always be unique and have a name beginning with F and followed by its module name.

Example 1 becomes:

```
%f[sqrt(%im : I; %oFsqrt : I) :
%v[k : I;
   p : I];
   %a[. . .]]
```

The internal form of the procedure and function calls are exactly what one would do in a compiler written in assembly languages: There is a stack $\#SCreturn$ together with its counter $\#Kreturn$. To call a procedure $\partial\lambda$, we first assign the values of the input and input-output actual parameters to their corresponding formal parameters, and then push the name of the next conditional element onto the stack. We *goto* the label $\partial\lambda$ (*i.e.* $\circ\#\mathrm{lb} = \partial\lambda$). Upon completion, we pop the top label into a pointer and then *goto* the position corresponding to the content of that pointer. It is then necessary to assign all output **and** input-output formal parameters values corresponding their actual correspondents. Of course, this internal form is unreadable, so an external form is necessary.

In the external form, to call procedure $\partial\lambda(\ldots)$ we say "$\uparrow\partial\lambda(-,-,-)$", where "$(-,-,-)$" are'the actual parameters. To call a function $\partial\lambda(\%i\ldots;\%oF\partial\lambda : \partial)$ *we say* "$\#F\partial\lambda(-,-,-)$", where "$(-,-,-)$" are the actual parameters which correspond to the $\%i$ part of the formal parameters. Thus to apply the function sqrt to the integer '16' we would *say* "$\#Fsqrt(16)$.

4. XYZ/E2.

XYZ/E2 is the layer that has the super structure which can be used to express concurrency and nondeterminancy [14].

The new allocational formula for Petri *Net* is "N". A general variable of type N is also accompanied by variables of type I, which are called places.

There are also two new special operators from the domain of transition variables to {T, F}:

(1) isfired($\#N\lambda$)

(2) isenabled($\#N\lambda$)

Each Petri net must obey the following axioms:

$$\forall N\lambda(isfired(\circ\#N\lambda) \supset isenabled(\#N\lambda)) \quad A$$

$$\exists N\lambda(isenabled(\#N\lambda)) \supset \exists N\lambda'(isfired(\circ\#N\lambda')) \quad A$$

$$\forall N\lambda\forall N\lambda'(\#N\lambda \neq \#N\lambda' \supset \neg(isfired(\#N\lambda) \wedge isfired(\#N\lambda')))$$

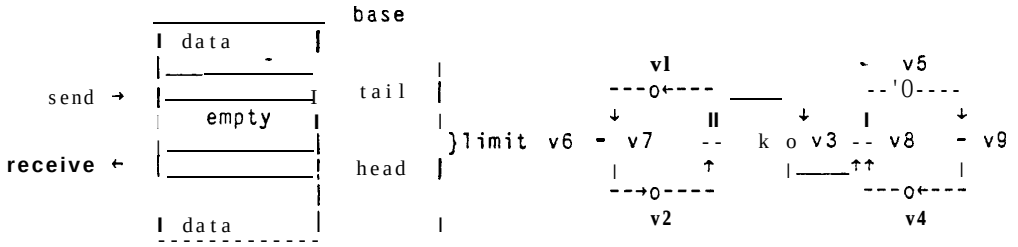

fig.3

fig.4

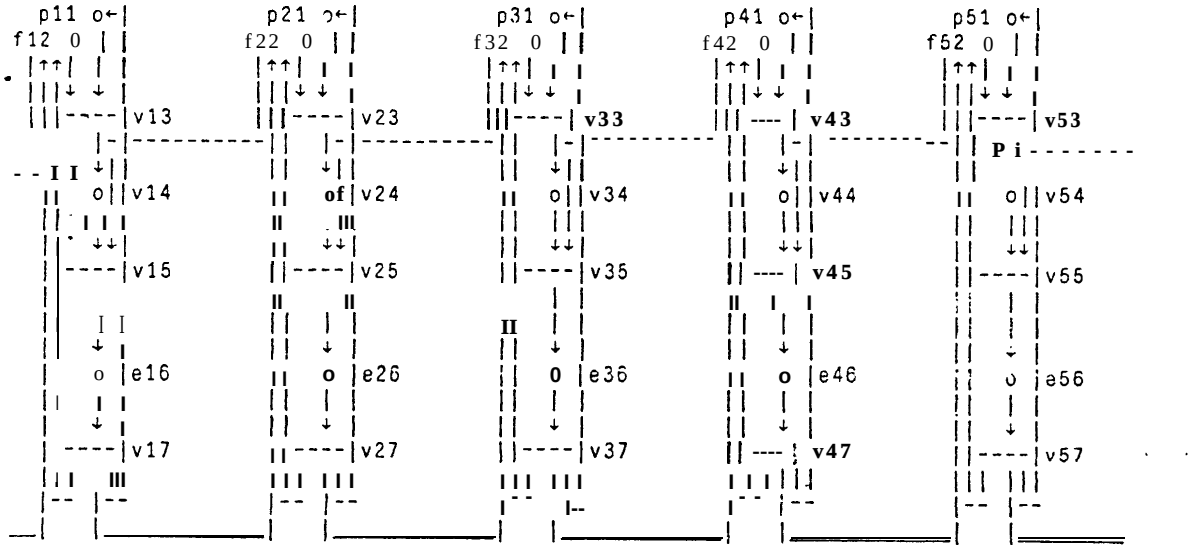

fig.5

Example 3: The well-known send-receive problem is shown in Figure 3. We assume that the buffer is circular and that only one integer is to be sent or received each time. Modelling this problem with a Petri net, we get a net of the form in Figure 4 where v_1 and v_2 model the sender, v_4 and v_5 the receiver, and v_3 is the buffer which is k-bounded. When the sum (called tokens) is more than k, the transition object is suspended.

The program corresponding to Figure 4 is as follows:

□ *[sendreceive :*
%v[v_1 : *I;*
v_2 : *I;*
v_3 : *I;*
v_4 : *I;*
v_5 : *I;*
v_6 : *N;*
v_7 : *N;*
v_8 : *N ;*
v_9 : *N];*
%a[$n : \#v_1 > 0 \Rightarrow$ isenabled$(\circ\#v_6) \wedge \uparrow n$;
n : isfired$(\#v_6) \Rightarrow \circ\#v_1 = \#v_1 - 1 \wedge \circ\#v_2 = \#v_2 + 1 \wedge \uparrow n$;
$n : \#v_2 > 0 \wedge \#v_3 < k \Rightarrow$ isenabled$(\circ\#v_7) \wedge \uparrow n$;
$n : \#v_2 > 0 \wedge \#v_3 \geq k \Rightarrow \uparrow n$;
n : isfired$(\#v_7) \Rightarrow$
$\circ\#v_2 = \#v_2 - 1$ A $\circ\#v_1 = \#v_1 + 1$ A $\circ\#v_3 = \#v_3 + 1 \wedge \uparrow n$;
$n : \#v_3 > 0 \wedge \#v_4 > 0 \Rightarrow$ isenabled$(\circ\#v_8) \wedge \uparrow n$;
n : isfired$(\#v_8) \Rightarrow$
$\circ\#v_4 = \#v_4 - 1 \wedge \circ\#v_3 = \#v_3 - 1 \wedge \circ\#v_5 = \#v_5 + 1 \wedge \uparrow n$;
$n : \#v_5 > 0 \Rightarrow$ isenabled$(\circ\#v_9) \wedge \uparrow n$;
n : isfired$(\#v_9) \Rightarrow \circ\#v_5 = \#v_5 - 1$ A $\circ\#v_4 = \#v_4 + 1 \wedge \uparrow n$]]

Example 4: The dining philolophers problem: For j = 1,. . . , 5, v_{j1} = 1 means the j^{th} philosopher holds at least one of his forks. v_{j2} = 1 means the j^{th} fork is held by the philosopher of its some side; v_{j6} = 1 represents the j^{th} philosopher is eating. The Petri net describing this problem is shown in Figure 5, and the XYZ/E2 program follows.

□[*phil osophers :*
%v[p_{11} : *I;*
f_{12} : *I;*
v_{13} : *N;*
v_{14} : *I;*
2'15 : *N;*
e_{16} : *I;*
v_{17} : *N;*

... similarly for philosophers 2,...,5

$$
\begin{aligned}
&\%a[n : \#p_{11} > 0 \wedge \#f_{12} > 0 \Rightarrow \mathtt{isenabled}(\circ\#v_{13}) \wedge \uparrow n; \\
&\quad n: \mathtt{isfired}(\#v_{13}) \Rightarrow \\
&\qquad \circ\#v_{14} = \#v_{14} + 1 \ \mathrm{A} \ \circ\#p_{11} = \#p_{11} - 1 \ \mathrm{A} \ \circ\#f_{12} = \#f_{12} - 1 \wedge \uparrow n; \\
&\quad n : \#v_{14} > 0 \wedge \#f_{22} > 0 \Rightarrow \mathtt{isenabled}(\circ\#v_{15}) \wedge \uparrow n; \\
&\quad n: \mathtt{isfired}(\#v_{15}) \Rightarrow \\
&\qquad \circ\#e_{16} = \#e_{16} + 1 \ \mathrm{A} \ \circ\#v_{14} = \#v_{14} - 1 \ \mathrm{A} \ \circ\#f_{22} = \#f_{22} - 1 \wedge \uparrow n; \\
&\quad n : \#e_{16} > 0 \Rightarrow \mathtt{isenabled}(\circ\#v_{17}) \wedge \uparrow n; \\
&\quad n : \mathtt{isfired}(\#v_{17}) \Rightarrow \\
&\qquad \circ\#p_{11} = \#p_{11} + 1 \ \mathrm{A} \ \circ\#f_{12} = \#f_{12} + 1 \\
&\qquad \wedge \circ\#f_{22} = \#f_{22} + 1 \ \mathrm{A} \ \circ\#e_{16} = \#e_{16} - 1 \wedge \uparrow n;
\end{aligned}
$$

... similarly for philosophers 2,...,5

One often runs into problems when using a Petri net to represent a concurrent program exactly since some properties may not be explicitly expressed. In Example 4, we must require that no two philosophers can hold a fork at the same time, and no philosopher can hold a fork not at his side. There is no problem in expressing these conditions in XYZ/E, since we can always supplement the program with a formula to express these conditions. For example, we need to supplement the dining philosophers program with the following formula:

$$
\begin{aligned}
&\forall j(1 \leq j' \leq 5)(\#f_{j2} = 1 \supset \exists k(1 \leq k \leq 5) HOLD(\#p_{k1}, \#f_{j2})) \ \mathrm{A} \\
&\forall j(1 \leq j' \leq 5)(\#p_{j1} = 1 \supset \exists k(1 \leq k \leq 5) HOLD(\#p_{j1}, \#f_{k2})) \ \mathrm{A} \\
&\forall j, k(1 \leq j, k \leq 5)(HOLD(\#p_{j1}, \#f_{k2}) \supset (j = k \ \mathrm{V} \ IFj \neq 5THENk = j + 1ELSEk = 1)) \wedge \\
&\forall j, k(1 \leq j, k \leq 5)(HOLD(\#p_{j1}, \#f_{k2}) \supset (\#p_{j1} = 1 \ \mathrm{A} \ \#f_{k2} = 1) \ \mathrm{A} \\
&\forall j, \ k(1 \leq j, \ k \leq 5)(HOLD(\#p_{j1}, \#f_{k2}) \supset \neg \exists m(1 \leq m \leq 5)(m \neq j \wedge HOLD(\#p_{m1}, \#f_{k2})))
\end{aligned}
$$

5. Axiomatization and Verification.

The axiomatic system of XYZ/E contains the following parts:

(1) the axioms of predicate calculus;

(2) the axioms for the modal operators Cl, 0, and $\mathcal{U}$;

(3) the axioms for integers and characters.

These are easy to find in the literatures.

(4) We need a new set of axioms for the next-time operator o, since we lay emphasis on the commutativity and distributivity of o with other operators in the system. These new axioms are **as** follows:

(i) For any logical formula A and any unary predicate operation $\partial \in \{\neg, \forall x, \exists x, \Box, 0\}$, $\circ\partial A \equiv \partial\ o\ A$.

(ii) For any logical formulas A and B and binary predicate operation $\partial \in \{A, \vee, \supset, \equiv, U\}$, $\circ[A \partial B] \equiv [\circ A \partial\ o\ B]$.

(iii) For any expression e and unary expresional operation $\in \{+, -\}$, $\circ\partial e = \partial \circ$ e.

(iv) For any name scheme α_β where β is its rightmost node, $\circ\alpha\_\beta = \alpha\_ \circ \beta$.

(v) For any expressions e_1 and e_2 and relational connective $\partial \in \{=, <, >, \ldots\}$, $\circ(e_1 \partial e_2) \equiv (\circ e_1 \partial \circ e_2)$.

(vi) For any expression e_1 and e_2 and binary operation $\partial \in \{+, -, *, /\}$, $\circ(e_1 \partial e_2) = (\circ e_1 \partial \circ e_2)$.

(vii) For any basic variables $.y$, $\circ.y = .y$.

(viii) For any constant c, $\circ c$ = c.

(ix) For any logical formula A, $\Box A \supset \circ A$.

By means of these axioms we can always move o's inward until all are located in front of general variables. XYZ/E1 and XYZ/E2 have additional axioms for stacks and transition types. Those for transition have been enumerated in section 4, while the axioms for stacks are well known.

This modal logic system can be easily reduced to the first order logic system. It has been shown [10] that for any $w, \Box w$ corresponds to $\forall t w$, and 0 w to $\exists t w$. Also, every general variable of the form $\#\partial\pi\lambda$ can be expressed in first order logic by $\mathrm{Apply}(\partial\pi\lambda, t)$ where Apply is a predicate defined by each given program, and $\circ^k \#\partial\pi\lambda$ is expressed by $\mathrm{Apply}(\partial\pi\lambda, t + k)$. $w_1 \mathcal{U} w_2$ corresponds to following formula:

$$\forall t_1 \exists t_2 (t_1 < t_2 \ \mathbf{A}\ \mathrm{Apply}(w_2, t_2) \wedge \forall t_3 (t_1 \leq t_3 < t_2 \supset \mathrm{Apply}(w_1, t_3))).$$

Feng [4] has established the Hoare-type proof rules for XYZ/E. Like Lucid, XYZ/E can also be used to do mathematical proofs directly as a logic system.

Since every program in XYZ/E is a well-formed logic formula, it can be used directly as the local axiom for proving any property from it. Thus, in Example 1, to prove its correctness, we need prove that if we start with a positive integer $\#m$, that lb = **stop** $\wedge \#n^2 \leq \#m < (\#n + 1)^2$ is provable. We only need to prove:

$$\#\mathrm{lb} = sqrt \wedge \mathrm{EX1} \wedge \#m > 0 \supset 0 \ [\#\mathrm{lb} = \mathbf{stop} \wedge \#n^2 \leq \mathbf{\#m}\ \mathbf{A}\ \#m < (\#n + 1)^2]$$

where EX1 represents the program in Example 1. Similarly, to prove the invariant of the loop starting at 12, we need only prove:

$$\#\mathrm{lb} = sqrt\ \mathrm{A}\ \mathrm{EX1}\ \mathrm{A}\ \#\mathrm{m} > 0 \supset 0\ [\#\mathrm{lb} = l2 \supset \#p = \#k^2].$$

One of the interesting aspects of XYZ/E in its application to verification is that invariant properties which can only be expressed by introducing so-called *ghost variables* and *passive statements* [8] in other logic systems can be expressed directly in XYZ/E. Take Example 2 in section 2. As pointed out in [8], the invariant corresponding to the loop from 14 to 16 is expressed as:

> *high* — low is strictly decreasing. If the object *.obj* is in the ordered array *.ar*, then one of its positions is between low and *high*.

Now to formalize it as [8]:

"A is ordered between 1 and N" is expressed as

$$ORDERED(A, 1, N) \equiv \forall x(1 \leq x < N \supset A\_(x) \leq A\_(x+1))$$

"B is in the interval $(L,$ R) of A/ is expressed as

$$ISINTERVAL(B, A, L, R) \equiv \exists x(L \leq x \leq R \ A \ A\_(x) = B)$$

So the last part of the invariant can be expressed as

$$ORDERED(.ar, 1, .dpt) \wedge$$
$$(ISINTERVAL(.obj, .ar, 1, .dpt) \supset ISINTERVAL(.obj, .ar, \#low, \#high))$$

How do we express the first sentence of the invariant? In a logic system which has no explicit method of expressing state transitions, we would have to add an extra ghost variable and an extra passive statements into the algorithm and the assertion as [8] suggests. In [8], the ghost variable is C, and the passive statements added to the positions corresponding to 13 and 14 respectively are "C := N + 1" and "C := (high — low)"; the first sentence of the invariant expressed in [8] is "low < $high \supset$ (high — low) < C". No extra variables or sentences are needed to express this sentence in XYZ/E. We can express this sentence as

$$\Box \ I(\#ZOW < \#high \supset (o\#high - o\#low) < (\#high - \#low)).$$

As is well known [5], the modal logic system with operators $\Box, \Diamond, o$ and $\mathcal{U}$ can express not only the properties of sequential programs, but also many of the significant properties of concurrent and nondeterministic programs. In Example 4, if each philosopher holds his left-hand fork, then they would all suffer from starvation (*i.e.* deadlock). In order to express this property, let Q represent the formula concerning HOLD given at the end of the last section, and let the program in Example 4 be represented by P. The deadlock property of Example 4 can be represented as follows:

$$P \ A \ Q \ A \ \forall j, k(1 \leq j, k \leq 5)(\#p_{j1} = 1 \ A \ \#f_{k2} = 1) \supset Cl \ \forall m(1 \leq m \leq 5)(\#e_{m6} = 0))$$

However, if some philosopher does not hold any fork but all the forks are held, then at least one philosopher can eat sometime. This property can be expressed as follows:

$$P \wedge Q(1 \leq m \leq 5)(\#p_{m1} = 0 \ A \ \forall j(1 \leq j \leq 5)(\#f_{j2} = 1 \ A \ (j \neq m \supset \#p_{j1} = 1))$$
$$\supset \exists k(1 \leq k \leq 5 \wedge k \neq m \wedge 0 \ \#e : k6 = 1)).$$

Since in the nondeterminate case, the time is in branching order, the operators $\Box$, **0** differ in meaning from when used in linear time order. There are many different ways to deal with the modal operators in the non-deterministic case:

(1) To explain the operator $\Box$ as "for all pathes and for all time nodes", and 0 as "there exists a path and there exists a time node". In this explanation, the system built above is applicable, but there are some properties unexpressible in this system. We lack operators to express "for each path there exists a time node" and "there exists a path for all time nodes". Consequently, is this system, not all properties of non-determinate programs are expressible.

(2) Another choice is to use the system given in [3]. The six modal operators are $\forall G, \exists G, \forall F, \exists F,$ VX, 3X. They correspond to "for all paths and all time nodes on it", "there exists a path for all time node on it", "for all paths there exists a time node on it", "there exists a path and a time node on it", "for each path, the next node on it", and "there exists a path,the next node on it" respectively.

This system of symbolism looks too novel and differs too much from the customary modal logic system given above. It would be more desireable to be able to express all the properties expressible by these six operators, but remain within the framework of the modal logic and first-order theory. Thus we instead do the following:

(3) Since we discuss nondeterminate properties on the basis of Petri nets which are expressed by XYZ/E2 programs, we define the concept of *path* in terms of XYZ/E2 constructs. To express this fact more explicitly, we use first-order symbolisms instead of modal theoretic ones, i.e. we use $\mathrm{Apply}(\partial\lambda, t)$, $\mathrm{Apply}(\partial\lambda, t+1)$, $\forall t, \exists t$ instead of $\#\partial\lambda, \circ\#\partial\lambda$, Cl, 0 respectively. Let P be the XYZ/E2 program given, $v_{n1}, \ldots, v_{nm}$ are transition variables occurring in P and nl, . . . , nm are different integers to represent the indices of these variables. Let q be an integer different from nl, . . . , nm. Then the definition of path variable x can be given as below:

$$P \supset \forall t(\forall \mathrm{j}(1 \leq \mathrm{j} \leq m)(\mathrm{isfired}(\mathrm{Apply}(v_{nj}, t)) \supset \mathrm{Apply}(x, t) = nj)$$
$$\mathrm{A}\ \neg\exists\ \mathrm{j}(1 \leq \mathrm{j} \leq m)\mathrm{isfired}(\mathrm{Apply}(v_{nj}, \mathrm{t})) \supset \mathrm{Apply}(x, \mathrm{t}) = q)$$

Call this formula $D(P, x)$. To make use of this formula and quantification over x and t, the six cases shown above can be easily expressed.

The deadlock property of Example 4 can be expressed as:

$$P\ \mathrm{A}\ \mathbf{Q}\ \mathrm{A}\ \forall j, k(1 \leq j, k \leq 5)(\mathrm{Apply}(p_{j1}, t) = 1 \wedge \mathrm{Apply}(f_{k2}, t) = 1)$$
$$\supset \forall x \forall t \forall g(g \in \{15, 25, 35, 45, 55\})(D(P, x) \supset \mathrm{Apply}(x, t) \neq g)))$$

We can similarly express the eating property as:

$$\ldots \supset \exists x \exists t \exists g(g \in \{15, 25, 35, 45\})(D(P, x)\ \mathrm{A}\ \mathrm{Apply}(x, \mathrm{t}) = \mathrm{g})))$$

Obviously, we are still within the framework of first order theory, a fact indeed beyond our imagination. All the expression above can also be expressed with modal theoretic symbolism. So the modal-logic system given in this paper can be used not only to express nondeterminate programs but also to express all sorts properties required in [3].

6. Two **Level Formal Semantics and Semantics-directed Compilation.**

Since XYZ/E is an assembly-like language, it can serve as the target language for the compiler of any higher-level language. Yet XYZ/E is also a logic system and its denotational formal semantics is well defined, so the compiler of the given higher-level language can be taken as the formal semantics of the given language. This is a natural way to treate formal semantics, it makes formal semantics closely related with compiler, and seems to give a satisfactory solution to the problem.

As an example, we first give a simple language **Sample** as follows:

```
<program> ::= <name>(IN <decl seq>; OUT <decl seq>;
          INOUT <decl seq>)<prog body>
<prog body> ::= BEGIN <decl seq> END; <state>.
<decl seq> ::= <decl> | <decl seq>;<decl>
<state seq> ::= <state> | <state seq>;<state>
<decl> ::= <name>:<type>
<type> ::= <elem type> | <pointer> | <array> | <record>
<elem type> ::= INT | CHARS | BOOL
<pointer> ::= ↑<type>
<array> ::= ARRAY (expression) OF <type> END
<record> ::= RECORD <decl seq> END
<state> ::= <assign> | <compound> | <conditional> |
          <while state> | <goto> | <name>:<state>
<assign> ::=<name>←<expression>
<compound> ::=BEGIN <state seq> END
<conditional> ::= IF <bool expression> THEN <state> ELSE <state>
<while state> ::= WHILE <bool expression> DO <state> END
<goto> ::= GOTO <name>
```

The transformation from Sample to XYZ/E consists of two major steps:

(1) *Name normalization:* The naming system in XYZ/E has been elaborated in Section 1. This system does not differ significantly from those used in high-lever languages except for the following points:

(i) Each name has a type part. Since It can always be restored by syntactical analysis, we omit it in following discussions.

(ii) The labels of the subfields of record are always concatenated with the label of the record.

(iii) The component of an array (or the body of a loop) is labelled with a name scheme which is formed by the name of that array (or loop) tagged with a node formed by the corresponding counter braced with parenthesis, i.e.:

```
λ: ARRAY(<expreession) OF λ_(#Kλ):<type> END
λ: RECORD λ_n1:<type>1;...; λ_nk:<type>k END
```

(2) The following are the equations which transform **Sample** into XYZ/E. In these equations, β represents the semantic mapping. The English construct "if – then – elseif – then – else –" is used as a metalanguage to express case situations. There are other phrases, such as "it begins with" which is used to express the leftmost symbol or symbol string, and "$<\ldots>$ is – – –" which is used to identify the kind of $<\ldots>$. "↔" means the left side of it can be replaced by its right side. In order to be more readable, we choose the external form of XYZ/E to represent the semantics. Of course it can be changed into the internal form easily. We call these kind of semantics equations β equations.

β(<program>) ↔ β(<name>(IN x11:<type>11;...;x1k:<type>1k;
OUT x21:<type>21;...;x2l:<type>2l;
INOUT x31:<type>31;...;x3m:<type>3m)
BEGIN x41:<type>41;...;x4r:<type>4r END;
1: <state>.)
↔ □ [<name>:
%i[x11: β(<type>11);...; xlk: β(<type>1k)];
%o[x21: β(<type>21);...; x21: β(<type>2l)];
%io[x31: β(<type>31);...; x3m: β(<type>3m)];
%v[x41: β(<type>41);...; x4r: β(<type>4r)];
%a[1: β(<state>)]]

β(<type>) ↔ if it begins with ''INT'' then β(INT)
elseif it begins with "CHARS" then β(CHARS)
elseif it begins with ''BOOL'' then β(BOOL)
elseif it begins with ''↑'' then β(<pointer>)
elseif it begins with "ARRAY" then β(<array>)
else β(<record>)

β(INT) ↔ I

β(CHARS) ↔ C

β(BOOL) ↔ B

β(<pointer>) ↔ β(↑<type>)
↔ Pβ'(<type>)

β'(<type>) ↔ if it begins with "ARRAY" then β'(<array>)
elseif it begins with "RECORD" then β'(<record>);
else β(<type>)

β'(<array>) ↔ β'(ARRAY(<expression>) OF $\partial\lambda\_\#K\partial\lambda$):<type> END)
↔ Aβ'(<type>)

@(<record>) ↔ #(RECORD $\partial\lambda$_n1:<type>1;...;$\partial\lambda$_nk:<type>k END)
↔ R{β'(<type>1),...,β'(<type>k)}

$\partial\lambda$:Q ⇒ β(<array>) ↔ $\partial\lambda$:Q ⇒ β(ARRAY (<expression>) OF $\partial\lambda\_(\#K\partial\lambda)$:<type> END)
↔ $\partial\lambda$:Q∧#K$\partial\lambda$ < β(<expression>)⇒ ·
[$\partial\lambda\_(\#K\partial\lambda)$:$\beta$(<type>); ∘#K$\partial\lambda$ = #K$\partial\lambda$ + 1; ↑$\partial\lambda$];
Q∧#K$\partial\lambda$ ≥β(<expression>)⇒

Similarly, we have the β equation for the case that $\partial\lambda : \beta(<$array>). Hereafter, we will no longer mention this corresponding case for other constructs.

```
∂λ:Q ⇒ β(<record>) ↔∂λ:Q ⇒
                    β(RECORD ∂λ_n1:<type>1;. . .; ∂λ_nk:<type>k END)
        ↔ ∂λ:Q ⇒ ∂λ_n1: β(<type>1); . . .  ; ∂λ_nk: β(<type>k)


β(<state>) ↔ if it begins with "IF" then β(<conditional>)
             elseif it begins with "BEGIN" then β(<compound>)
             elseif it begins with "WHILE" then β(<while state>)
             else β(<assign>)


β(<assign>) ↔ β(∂λ ←<expression>) ↔ ∘#∂λ =β(<expression>)


λ:Q ⇒ β(<compound>)
        ↔ λ:Q ⇒ β(BEGIN <state>1; . . .  ;<state>k END)
        ↔ λ:Q ⇒ [ β(<state>1); . . .  ; β(<state>k) ]


λ:Q ⇒ β(<conditional>)
        ↔ λ:Q ⇒β(IF <bool expression> THEN <state>1 ELSE <state>2)
        ↔ λ:Q∧β(<bool expression>) ⇒ β(<state>1)∧ ↑next;
        ↔ Q∧β(¬<bool expression>) ⇒ β(<state>2);
        next:


λ:Q ⇒ β(<while state>)
        ↔λ:Q⇒β(WHILE <bool expression> DO <state> END)
        ↔ λ:Q∧β(<bool expression>) ⇒ β(<state>)↑λ;
        Q∧β(¬<bool expression>) ⇒


β(<goto>) ↔ β(GOTO λ)
        ↔ ↑λ
```

These equations constitute the semantics of Sample in terms of XYZ/E, whose own denotational semantics is direct. They also describe a compiler which translates Sample programs into XYZ/E. This is what we call two-level formal semantics and semantic-directed compilation.

it is easy to see that an inverse transformation which translates an XYZ/E program back into Sample exists. ·

Acknowledgement. This work was done at the Artificial Intelligence Laboratory of the Stanford University Computer Science Department during 1979-1981. The author expresses his thanks to Professors John McCarthy, Zohar Manna, David Luckham and Carolyn Talcott, and to Mr. Frank Yellin and Mr. Ma Xiwen for their help. This work is based on an investigation on McCarthy's Elephant. Some basic thoughts are strongly influenced by it. Both Messrs. Yellin and Ma have given me many good suggestions and pointed out many errors. The author is also glad to express his gratitude to Professor A. N. Habermann of Carnegie-Mellon who gave much helpful advice to this work.

References

[1] Albrecht, P. F. et. al. "Source to Source Translation: Ada to Pascal and Pascal to Ada." Proc. Sigplan Conf. on Ada, Boston, 1980.

[2] Ashcroft, E .A. anti W. Wedge. "Lucid, A Nonprocedural Language with Iteration." CACM, vol. . 28, no. 7, 1977.

[3] Ben-Ari, M. et. al. "The Temporal Logic of Branching Time." Draft, Aug. 1980.

[4] Feng Yu-lin. "Program Logic and Program Correctness Proof." Dissertation, 1980.

[5] Gabbay, D., et. al. "On The Temporal Analysis of Fairness." Proc. Symp. POPL., 1980.

[6] Hoare, C. A. R. "Data Reliability." Intl. Conf. on Reliab. Software Proc., 1975.

[7] Igarashi, S., et. al. "Automatic Program Verification I: Logic Basis and its Implementation." Acta Inf., vol. 4, 1979.

[8] Luckham, D. "On the Specification and Verification of Ada programs." Draft, 1980.

[9] Manna, Z. "Logic of Programs." IFIP Proc., 1980.

[10] _____ and A. Pnueli. "The Modal Logic of Programs." Tech. Rept. Of Stanford Univ. Dept. of Comp. Sci., 1979.

[11] McCarthy, J. "Toward a Mathematical Theory of Computation." IFIP Proc., 1962.

[12] _____. "The Programming Language Elephant." Draft, 1979.

[13] _____ and P. J. Hayes. "Some Philosophical Problems of Ar tifical Intelligience." *Machine Intelligence*, vol. 4 (ed. D. Michie). Americal Elsevier, New York, 1969.

[14] Peterson, J. L. "Petri Nets." Comp Surv., vol. 9, no. 3, 1977.

[15] Pnueli, A. "The Temporal Semantics of Concurrent programs." Lect. Notes in Comp. Sci. 70, Springer Verlag, 1979.

[16] Tang, C. S. "Abstract Specification and Hierarchical Programming with Temporal Logic Language." Draft, Stanford. 1981.

[17] _____. "XYZ: A Family of Programming Languages." Abst. at IFIP WG 2.4 meeting at Twent, Netherlands' 1978.

[18] _____ T. M. He and F. C. Xu (writer). "The Common Base Language Of XYZ." [Chinese.] Computer Science, No. 2, 1980.

[19] Teller, J. "Intermediate Language." Draft, Siemens, 1980.