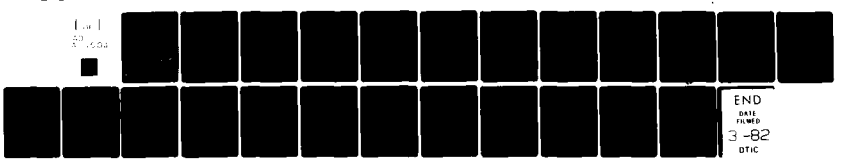
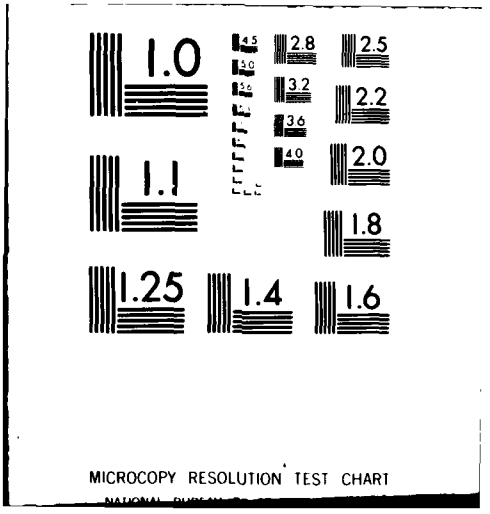
AD-A111 504 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
EXPRESSING THE FORMAL SEMANTICS OF CSP AND CP OR ADA TASKING WI--ETC(U)
NOV 81 C TANG F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-82-0076 NL



END
DATE FILMED
3-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 82-0076	2. GOVT ACCESSION NO. AD-A111 504
	3. RECIPIENT'S CATALOG NUMBER 3
4. TITLE (and Subtitle) EXPRESSING THE FORMAL SEMANTICS OF CSP AND CP OR ADA TASKING WITH THE TEMPORAL LOGIC LANGUAGE XYZ/E	5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Chih-sung Tang	8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Maryland College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Directorate of Mathematical & Information Sciences Air Force Office of Scientific Research Bolling AFB DC 20332	12. REPORT DATE NOV 81
	13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A temporal logic language XYZ/E is introduced. It is a temporal logic system as well as a programming language. It has two forms: the internal form is in lower level, but there are several abbreviation rules which can transform a program in internal form into a higher level external form and vice versa.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DISCLAIMER NOTICE

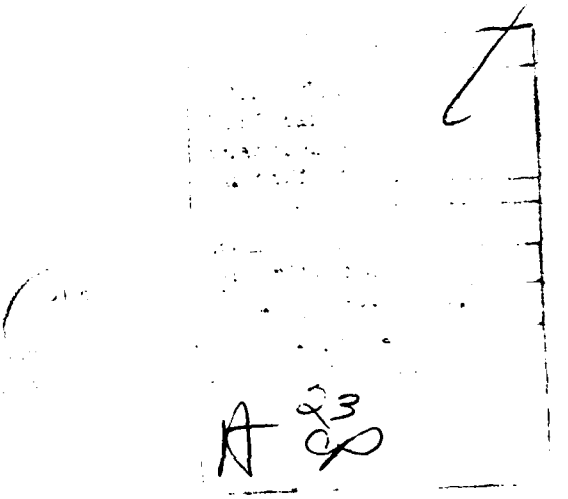
THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Technical

EXPRESSING THE FORMAL SEMANTICS OF CSP AND CP OR ADA TASKING WITH THE TEMPORAL LOGIC LANGUAGE XYZ/E

Chih-sung Tang

Computer Science Department
University of Maryland
College Park, MD 20742



*This work was supported in part by the Air Force Office of Scientific Research Contract [redacted]-F49620-80-C0001 to the University of Maryland. Computer support was provided in part by the Computer Science Center at the University of Maryland.

Erratum

P.7 bottom up line 5
"$\uparrow_1 1_{13} \hat{} \uparrow_2 p_2$" changed into "$\uparrow_1 1_{11} \hat{} \uparrow_2 1_{22}$"

P.17 line 10 changed into
"h-(sem)-(v): #v-(sem)-(v)-(1)= T => 0#v-(sem)-(v)-(1)= F ^ $\uparrow_0$end;
h-(sem)-(v): #v-(sem)-(v)-(10)= T => 0#v-(sem)-(v)-(10) = F ^ $\uparrow_0$end;"

P.17 line 17 changed into
"h-(sem)-(p): #v-(sem)-(p)-(1)= T => 0#v-(sem)-(p)-(1)= F ^ $\uparrow_0$end;
...
"h-(sem)-(p): #v-(sem)-(p)-(10)= T => 0#v-(sem)-(p)-(10)= F ^ $\uparrow_0$end;"

PP. 9-12 The sematic transformations of "->" ought to be changed according to the following pattern:

β(A-> B) $\leftrightarrow$ [β(A); β(B)]
β(A-> (B-> C)) $\leftrightarrow$ (A-> B-> C) $\leftrightarrow$ [β(A); β(B); β(C)]
β(A-> (P | Q)) $\leftrightarrow$ [β(A); $\uparrow$1]; 1: β(P); 1: β(Q)

So, the programs in Example 2, 3 must be changed accordingly.

[redacted] (AFSC)

AIR FO[redacted]
NOTI[redacted] end is
This [redacted] 9-12.
[redacted]
[redacted]
[redacted] J. [redacted]
Chief, Technic[redacted] Division

APPENDIX

Some Basic Features of XYZ/E

(1) In XYZ/E, each name is divided into three parts:⟨type symbol⟩⟨root⟩⟨index⟩. The type symbol is always omitted; the root is an identifier and the index is a string of nodes connected by hyphens. A node can be an identifier or an integer or an integer grouped by braces. E.g. Iabc-(3)-age is an name in which I is the type, abc is the root and the rest is the index. Related to the concept of name, a name schema is an expression whose value is a name. The form of a name schema differs from a name in that in the former an integral variable can occur within the braces of a node, e.g. Iabc-(#Kabc)-age is a name schema, here #Kabc is a counter corresponding to abc.

(2) For a name v, #v is a general variable. It represents its value at present time t. o#v represents the value of this variable at next time, i.e. t+1. o^k#v represents the value at time t+k. To make use of this convention, we can represent assignment v←v+1 as o#v=#v+1; and also goto p can be expressed as o#lb=p, here #lb is a special general variable used to indicate the current control labels. In this way, both assignment and goto become equations. We call the equation with #lb in it a lb equation. o#lb=p can be abbreviated as ^p and #lb=p as p:.

(3) We use the lb equations or their abbreviated forms to express not only the control flow of statements i.e. compound statement, case statement and loop, but also various data structures, i.e. record, type union and array, for both are correspondent to sequencing, branching and iterating respectively.

(4) Just as assignment is the basic element in statements, so I (integer), C (character string) and B (boolean) are basic elements in data type. They can be used both as type symbols and as allocational formulas. The latter are introduced in order to abbreviate the former. Besides, there is another allocational formula or type which is used to represent pointer, i.e. PT here T is the type of the object to be referenced.

(5) A program is always written with a special form which is a series of so called conditional elements separated by ;. A conditional element is of the form A⇒B which means that if A then B else false. A program is always quoted by brackets and

a subsequence of conditional elements can also be grouped in that way. The following is an example of integral sqare root:

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

This is the internal form of a program. It is on lower level. In order to transform it into a higher level external form, some abbreviation rules are given,e.g. a goto statement leading to next label can always be omitted, etc. (T1). The external form of above example is:

```
□[ sqrt:
    %i[ m:I];    %o[ n:I];    %v[ k:I; p:I] ;
    %a[o#k=0∧o#p=1;
       l2: #p>#m => ↑l4;
           #p≤#m => o#p=#p+2*#k+3∧o#k=#k+1∧↑l2;
       l4: #n=#k∧↑stop ]]
```

(6) In order to use XYZ/E to express the formal semantics of other languages, a metalanguage is introduced in which 'β' is a semantic mapping, it maps each construct of the object language into a string of entities in XYZ/E; '↔' is a replacing symbol; 'if - then-elseif...else' is used to distinguish different cases; 'it begins with' is to indicate the beginning section of the string. The intuitive meaning of all these symbolisms in metalanguage are obvious.

Expressing The Formal Semantics of CSP and CP or ADA Tasking With The Temporal Logic Language XYZ/E

Chih-sung Tang

In (T1), a temporal logic language XYZ/E is introduced. It is a temporal logic system as well as a programming language. It has two forms: the internal form is in lower level, but there are several abbreviation rules which can transform a program in internal form into a higher level external form and vice versa. In the APPENDIX of this paper, some basic features of this language are illustrated.

One of the major applications of XYZ/E is to use it as a means to express the formal semantics of other programming systems including conventional higher level languages. For it is a logic system, its own semantics is as easily defined denotationally as any logic system; while it is also a lower level language, the semantics of other programming system can be conveniently mapped into it. It seems to me that this approach is more natural than denotational semantics and also more elegant than operational semantics. This paper may help to show that the more complicated is the problem, the more obvious is the advantage of this approach. Besides, this approach is very close to the real compiler.

XYZ/E in its original version (T1) contains a layer to describe Petri net. It can express those concurrent or nondeterminate algorithms such as producer-consumer problem or five philosophers problem quite neatly. But this author finds that Petri net may not be a suitable mean to express such concurrent constructs as Hoare's CSP, Mao-Yeh's CP or ADA Tasking. This paper will show how to express them with XYZ/E. To my surprise, the result seems very satisfactory.

On Time Philosophy

How to treate time order of a distributed system is known a difficult problem (L1). In this paper, a new approach of time philosophy is adopted which seems able to simplify the situation.

We assume: (1) in a distributed system, the processes not only run independently, but also have their own private time systems which are unknown to each other; so each process has its own control flow system; (2) some of the names of variables and labels must be knowable from the outside, otherwise no communication is possible.

Our basic philosophy is that even though above assumption (1) is made, these processes are still coexisting in the same world; thus there is only one real TIME. Different private time systems of the processes are only different kinds of representations of this TIME. Consequently, in spite of their representative differences, there are still some common temporal concepts which reflect the identical TIME those processes of . . . although the measure of present time of process Pi may be quite different from that of Pj, both must be coincident in THIS PRESENT TIME. The minimum set of these common temporal concepts just constitutes the set of basic operations in temporal logic, i.e. present time (#), next time (○), eventuality (◇), and necessity (□). By means of these basic common mechanism, it is possible to express such kind of communications as " the next time value of variable v in Pj (according to Pj's time system) is equal to the present time value of variable u in Pi (according to Pi's time system)" in Pk without presupposing any of Pi,Pj, Pk knowing other's time system. I think, this might be the weakest presupposition that makes the synchronization and communication among these processes possible.

The Formal Semantics of Hoare's CSP

(1) CSP and guarded command.

concurrent program: P :: [P1//...//Pn]

here: Pi :: Si, i=1,...,n, Pi process, Si statement.

statements includes:

(a) input command: Pj?x (in Si)

(b) output command: Pi!y (in Sj)

(c) guarded commands: [B1→S1 ▯ B2→S2 ▯ ... ▯ Bk→Sk]

(d) repetition: *[B1→S1 ▯ B2→S2 ▯ ... ▯ Bk→Sk]

(e) skip and assignment

(f) statements sequence: Si1;...;Sim

boolean expressions includes:

(a) input command: Pj?x (in Sj)

(b) output command: Pi!y (in Si)

(c) conventional boolean expression: ba

(d) boolean sequance: Bi1;Bi2;...;Bim

The exact explanation of these constructs, see (E1),(A1).

(2) Presupposition before transformation.

(i) Assuming that in front of each statement there has already been a label and to the right of each process Pi there is a label endi. Consequently, for each boolean expression or statement, a label to its nearest right can always be found. Let nexti be the nearest label right to Bi or Si, it can be determined in following way:

(a) For (i=1,n), in 'bi;...;bi;...;bk→ l:S', l is the nexti for bi.

(b) For (i=1,n), in 'l1:S1;...;li:Si;...;ln:Sn;' ln is the nexti for Si.

(c) If S is the last statement in any Si (1≤i≤n) in following statement:

θ[B1→S1 ▯ B2→S2 ▯ ... ▯ Bn→Sn] ; l:S' , here θ=* or empty,
then l is the next for this S.

(ii) To each process Pi in [P1//P2//...//PN] a private control variable #lbi or its abbreviated form ↑i is assigned and in addition, a common control variable #lb or ↑ is also available.

(iii) To each input command Pj?xij in Si (or Pi), a boolean #rij is assigned; To each output command Pi!yji in Sj (or Pj), a boolean #sji is assigned. Let Pu@vvu represent either Pj?xij or Pi!yji.

(3) Transformation from CSP to XYZ/E.

(i) Each statement sequence is transformed into a compound statement, i.e.

$\beta(S1;...;Sk) \Leftrightarrow [\beta(S1);...;\beta(Sk)]$

(ii) Each boolean sequence b1;...;bk is transformed into a conjunction, i.e.

$\beta(b1;...;bk) \Leftrightarrow \beta(b1)\wedge\beta(b2)\wedge...\wedge\beta(bk)$

$\Leftrightarrow \beta(b)$ if there is no input-output command in b1,...,bk;

$\Leftrightarrow \beta(Pu@vvu)\wedge\beta(b)$ if Pu@vvu occurs in b1,...,bk.

(iii) The concurrent program is transformed as follows:

$\beta(P:: P1::S1//...//Pn::Sn) \Leftrightarrow P:\uparrow 1P1\wedge...\wedge\uparrow nPn;$

$P1: \beta(S1);...; Pn:\beta(Sn);$

(iv) The guarded commands are transformed in following way (assuming in Pi):(*)

$\beta(n:[B1\rightarrow l1:S1[]...[]Bk\rightarrow lk:Sk];\ next:S') \Leftrightarrow n:\beta(B1)\Rightarrow l1:\beta(S1)\wedge\uparrow i\ next;$

.....

$n:\beta(Bk)\Rightarrow lk:\beta(Sk)\wedge\uparrow i\ next;$

$n:\neg\beta(B1)\wedge...\wedge\neg\beta(Bk)\Rightarrow \uparrow i\ next;$

$next: \beta(S');$

Note that in the original definition of conditional elements in (T1), two conditions with one common label must be mutually contradictory, i.e.

n: B' ⇒ S1'; n: ¬B' ⇒ S2';

These two conditional elements are logically equivalent to following formula: ' if #lb=n then (if B' then S1' else S2') else false ' .

But in present case the conditions can be true simultaneously, and in that situation we have to choose one of those Si' following them.

(*) (iv) and (v) are only for the case that no input-output commands are in B.

The logical meaning of the latter is quite different from the former.

Let the conditional elements transformed from guarded commands are:

n: B1'⇒ l1: S1'; n: B2'⇒ l2: S2'; n: ¬B1'∧¬B2'⇒↑next;

These three conditional elements are logically equivalent to following sense:

'if #lb=n then (if B1'∧B2' then S1'∨S2'

else (if B1' than S1'

else (if B2' then S2' else o#lb=next)))'

here S1'∨S2' means that (S1'∨S2')∧¬(S1'∧S2').

(v) The repetition is transformed as follows (assumong in Pi):

β(n:* [B1→ l1:S1 ▯ ... ▯ Bk→ lk:Skj) ⇔ n:β(B1)⇒ l1:β(S1)∧↑i n;

.....

n:β(Bk)⇒ lk:β(Sk)∧↑i n;

n:¬β(B1)∧...∧¬β(Bk)⇒↑i next;

For those guarded commands (similarly, repetition) with input-output in Bi the transformation is as in (4)(i).

(4) The transformation of input-output commands:

For input-output commands can occur as statements and as boolean expressions, we divide following discussion into two cases:

(i) As statements (assuming in Pi):

For any input command Pj?xij or output command Pk!yik in Pi, as pointed out in (2), two booleans #rij and #sik have been assigned corresponding to these two commands respectively. Then these commands are transformed as follows:

β(Pj?xij) ⇔ o#rij=T∧↑comm

β(Pk!yik) ⇔ o#sik=T∧↑comm

here 'comm' is a label in the part common to all processes. In addition to above transformation, corresponding to Pj?xij we must insert following conditional element under the label comm:

comm: #rij=T∧#sji=T ⇒ o#xij=#yji∧o#rij=F∧o#yji=F∧↑i nextrij∧↑j nextsji;

here nextrij, nextsji are the next label of Pj?ij in Pi and that of Pi!ji in Pj respectively. As for Pk!ik, we need not insert the corresponding conditional element in comm, for its input partner Pi?ki in Pk would do that.

(ii) As booleans (assuming in Pi):

Let us assume the guarded command $ where they occur is of following form:

n: [Pj1?xij1;b1→ l1:S1 ▯ ... ▯ Pjq?xijq;bq→ lq:Sq ▯

Pk1!yik1;b1'→l1':S1' ▯ ... ▯ Pkg!yikg;bg'→lg':Sg'] end:

$\beta(S) \leftarrow n;\ \neg b1 \wedge \cdots \wedge \neg bq \wedge \neg b1' \wedge \cdots \wedge \neg bg' \Rightarrow \uparrow end;$

$n:\ b1 \vee \cdots \vee bq \vee b1' \vee \cdots \vee bg' \Rightarrow \circ\#rij1=T \wedge \cdots \wedge \circ\#rijq=T \wedge \circ\#sik1=T \wedge \cdots \wedge \circ\#sikg=T \wedge \uparrow comm;$

As in the case (i), there must be a conditional element in comm corresponding to each input command. Without loss of generality, we need only consider Pj1?xij1; we can also be safe to assume its partner occurring in Pj1 with following context: Pi!yj1i;b"→l":S". Then the conditional element in comm for Pj1?xij1 here is as follows:

comm: #rij1=T∧#sj1i=T∧b1∧b" ⇒ o#xij1=#yj1i∧o#rij1=F∧o#sj1i=F∧

F(Pj1?xij1)∧F(Pi!yj1i)∧$\hat{i}$ l1'∧$\hat{j1}$ l";

l1' and l" are just the labels next to Pj1?xij1 and Pi!yj1i respectively.

If the partner Pi!yj1i is a statement, then F(Pi!yj1i) is empty.

here F(Pu&wvu) represents that all those boolean input-output commands other than Pu&wvu but occurring in the same guarded command with this Pu&wvu are assigned with the value false; As for the output boolean commands, they share the comm conditional elements with their input partner.

Finally, there are still following labels and conditional elements in Pi:

l1: β(S1)∧$\hat{i}$ next;

....

lq: β(Sq)∧$\hat{i}$ next;

l1': β(S1')∧$\hat{i}$ next;

....

lg': β(Sg)∧$\hat{i}$ next;

here next is the label before the statement to the right of $, i.e. end. If $ is a repetition next is the label in front of $ itself.

Let us explain the above transformation with a typical example quoted from (A1).

Example 1. P::[P1//P2//P3]

P1::[P2?x→ l11:S1 [] b1;P3!y→ l12:S2]; l13:*[b2;P2!u→l14:S3[]b3;P3?u→l15:S4]; end.

P2::*[P1?s→ l21:S5] P1!t→ l22:S6[] P3?s→ l23:S7]; end.

P3:: P1?z; l31: *[b1→ l32:S8[] b2→ l33:S9]; end.

To apply the transformation given above to this CSP program, the XYZ/E program obtained is as follows:

P: ↑1 P1∧↑2 P2∧↑3 P3;

P1: o≠r12x=T∧o≠s13=T∧↑comm;

l13:¬b2∧¬b3 →↑1 end1;

l13: b2∨b3 → o≠s12u=T∧o≠r13u=T∧↑comm;

l11: β(S1)∧↑1 l13;

l12: β(S2)∧↑1 l13;

l14:β(S3)∧↑1 l13;

l15:β(S4)∧↑1 l13;

P2: o≠r21s=T∧o≠s21=T∧o≠r23s=F∧↑comm.

l21:β(S5)∧↑2 P2;

l22:β(S6)∧↑2 P2;

l23:β(S7)∧↑2 P2;

P3: o≠r31z=T∧↑comm;

l31:β(b1)→↑3 l32;

l31:β(b2)→↑3 l33;

l31:β(¬(b1)∧¬(b2))→↑3 end3;

comm: ≠r12x=T∧≠s21 b=T → o≠x=t∧o≠r12x=F∧o≠321 t=F∧ o≠s13=F∧o≠r21=F∧o≠r23=F∧

↑1 l13∧↑2 P2;

comm: ≠r21s=T∧≠s12u=T∧b2 → o≠s=≠r∧o≠r21s=F∧o≠s12u=F∧o≠S21 t=F∧o≠r23s=F∧
o≠r13=F∧↑l21∧↑l1;

comm: ≠r31z=T∧≠s13y=T∧b1 → o≠z=≠y∧o≠r31z=F∧o≠u13y=F∧o≠r12x=F∧↑3 l31∧↑1 l11;
comm otherwise → ↑comm.

The Formal Semantics of Zhou-Hoare's CSP with Named-Channels

Zhou-Hoare (ZH1) introduces a formal system to treate CSP with named channels and has proved the validity of the proof rules by means of denotational semantics. In spite of its mathematical sophistication, that approach seems too complicated. This author finds that the CSP with named channels can be easily and even more naturally described with the temporal logic language XYZ/E with the consequence that the proof rules in their system become theorems of XYZ/E which can be proved within the framework of temporal logic instead of using those complicated techniques of denotational semantics.

In order to describe the CSP with named channels, or equivalently, to transform the semantics of Zhou-Hoare's system into XYZ/E, we enumerate informally the correspondence of the basic notions of these two systems as follows:

·The process in their system is transformed into a procedure in XYZ/E;

·The channels in their system are transformed into I/O variables (i.e. input variable, output variable, or input-output variable, they are declared in %i part, %o part, or %io part respectively) of the procedure;

·The type of the values of the channels is treated as allocational formula ; as in (Z2), we allow sets as allocational formulas;

·In order to describe the relations between the channels (I/O variables) of different processes(procedures), an I/O variable used outside the procedure where it is defined, must have its name prefixed with the name of the procedure connected with a hyphen, for example,"%copier-wire=%reccpier-wire" means that the value of the channel wire in the process copier is equivalent to the value of the channel wire in the process recopier.

·The process expression of a process equation (p≙P) is transformed into the conditional expressions of the algorithmic part,i.e. %a part of the corresponding procedure with the convention that the process expressions are transformed in accordence with following rules:

(1) In order to transform the expression of the form "A->B" exactly, we need to give some comments on the operation "->" used in (ZH1). To my understanding, "A->B" in (ZH1) ought to have the meaning that "if A then subsequently (or eventually) B". But if we explain this operation in this way, the expression "A->(B->C)" ought to mean that "if A then subsequently that if B then subsequently C". In this case when A holds it is not necessary that B must holds subsequently. To judge from the examples in (ZH1), I believe it is not what its authors intend to mean. I think , it is misleading to allow replacing the B in "A->B" with "B-> C ". So in following discussion, no such kind of the form of the expression as "A->(B->C)" is allowed. We express what such expression intends to express as "A->B; B->C". For this kind of expression, the tranformations are $\beta(A\to B)\leftrightarrow\beta(A)\Rightarrow\Diamond\beta(B)$, $\beta(A\to B;\ B\to C)\leftrightarrow\beta(A)\Rightarrow\Diamond\beta(B)\ \beta(B)\Rightarrow\Diamond\beta(C)$;

(2) In "c!e->P", c is transformed into an output or input-output variable #c and e is transformed into a corresponding expression; so the whole expression is transformed as $\beta(c!e\to P)\leftrightarrow\#c=\beta(e)\Rightarrow\Diamond\beta(P)$. Note. this variable #c must be declared at the %o or %io part of the procedure.

(3) In "c?x:M->P", c is transformed into an input or input-output variable #c which is declared at the %i part or %io part of the procedure; x is a local variable of type M which is also c's type, declared at %v part of the procedure. Hence, the whole expression is transformed as $\beta(c?x:M\to P)\leftrightarrow\#c=\#x\Rightarrow\Diamond\beta(P)$.

(4) For the expression (P|Q), we assign a common label, say lpq before both the transformed expression of P and that of Q. Thus $\beta((P|Q))\leftrightarrow lpq:\beta(P);\ lpq:\beta(Q);$.

(5) As for (P x|y Q) and (chan L;P), it is natural to transform these forms into embedding subprocedures in an outlayer procedure and expressing the relations among their I/O variables as equations in the %a part of the outlayer procedure. We will explain the transformation by the examples below.

(6) Stop is transformed to stop.

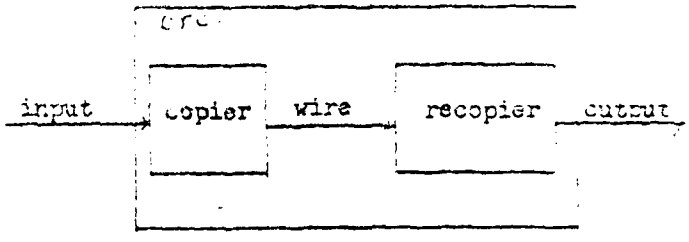
Example 2. Given following process equations in Zhou-Hoare's system:

copier $\triangleq$ (input?x: NAT $\rightarrow$ wire!x; wire!x $\rightarrow$ copier).

recopier $\triangleq$ (wire?y: NAT $\rightarrow$ output!y; output!y $\rightarrow$ recopier).

(chan wire; (copier || recopier))

which is represented pictorially by following figure:



The procedure into that the above processes transformed is as follows:

%p[crc:

%i[input:NAT];

%o[output:NAT];

%p[copier:

%i[input:NAT];

%o[wire:NAT];

%v[x:NAT];

%a[lc: #input=#x $\Rightarrow$ #wire=#x

#wire=#x $\Rightarrow$ ↑[lc]·

%p[recopier:

%i[wire:NAT];

%o[output:NAT];

%v[y:NAT];

%a[lr: #wire=#y $\Rightarrow$ #output=#y;

#output=#y $\Rightarrow$ ↑[lr];

%a[lcrc: ↑copier ∧ ↑recopier ∧ M1rco;

lcrc: #copier-input=#crc-input;

#recopier-wire=#copier-wire;

#crc-output=#recopier-output;↑[lcrc];

Example 3. The given processes equations are:

sender $\triangleq$ (input?y:M $\rightarrow$ q[y]).

q[x: M] $\triangleq$ (wire!x $\rightarrow$ (wire?y: {ACK} $\rightarrow$ sender | wire!y:{NACK} $\rightarrow$ q[x]))

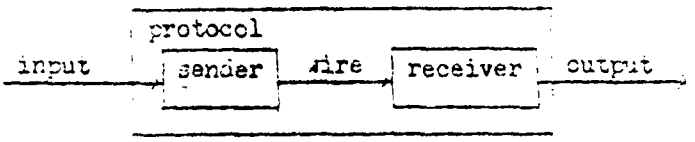
receiver $\triangleq$ (wire?z: M $\rightarrow$ (wire!ACK | wire!NACK);

wire!ACK $\rightarrow$ output!z; output!z $\rightarrow$ receiver;

wire!NACK $\rightarrow$ receiver).

protocol $\triangleq$ (chan wire; (sender || receiver)).

The corresponding pictorial representation is as follows:



The procedure obtained after transformation is:

%p[protocol:

%i[input: M];

%o[output: M];

%p[sender:

%i[input: M];

%o[wire: M];

%v[y: M];

%p[q:

%i[wire: {ACK,NACK}];

%o[wire: M];

%v[x: M];

%a[↑lq;

lq: #wire=#x =>◊#wire=ACK;

lq: #wire=#x =>◊#wire=NACK.

#wire=ACK =>↓sender;

#wire=NACK =>↓lq ;

%a[#input=#y =>◊#q-wire=#y∧↑[ ]];

%p[receiver:

%i[wire: M];

%o[wire: {ACK,NACK} ;

output: M];

%v[z: M];

%a[↑lr;

lr: #wire=#z =>◊#wire=ACK;

lr: #wire=#z =>◊#wire=NACK;

#wire=ACK => #output.s,↑,ir;

#wire=NACK => [ir]]

[a[prot:[sender,[receiver]

prot: #sender-input=#protocol-input;

#receiver-wire=#sender-wire;

#protocol-output=#receiver-output;]]]

It seems to me that above examples are sufficient to show the advantage of TTL/T in expressing the formal semantics of CSP with named channels. It looks natural and akin to conventional programming practice.

As for the proof rules of Zhou-Hoare system, they become metath orems of TTL/T system. To prove them within the framework of temporal logic seems more straightforward than to verify the validity through the denotational semantics model.

The Formal Semantics of Mao-Yeh's CP

Mao-Yeh's CP is a language concept for concurrent programming introduced in (MY1). It is substantially synonymous with ADA Tasking. For the concept of CP has been more neatly defined in (MY1), it is chosen as the model in our discussion in this paper. The method introduced here can be removed to deal with ADA Tasking in an obvious way.

In order to save time, we decline to introduce the concept of CP in detail here. Mao-Yeh's paper (MY1) is assumed. Roughly speaking, in a concurrent system, there are two kinds of process: master processes which receive messages; servant processes which send messages. They exchange informations through three kinds of statements: the connect statement which the servants use to send messages; the port statement which the masters use to receive messages from the servants through some ports which are buffers; and the disconnect statement which the master uses to release the servants after the messages have been received. The former two kinds of statements contain a list of parameters to indicate the names of the master, the servant and the port and also some input and input-output parameters to store the messages and informations to be exchanged.

(1) Assumptions.

(a) We assume there are n masters whose names are m1,...,mn; k servants whose names are s1,...,sk and l ports p1,...,pl.

(b) For simplicity, we assume the parameter lists of all connect statements and the port statements are similar: they all have a parameter for the index of the master i.e. mx, one for the port i.e. py, and one for the servant i.e. sz; besides, the formal parameters of informations in port statements are of following form:

the input formal parameters are: if1:T1,...,ifs:Ts,
the input-output formal parameters are: vf1:T1',...,vft:Tt'.
The corresponding actual parameters in connect statement are: ia1,...,ias; and va1,...,vat.

(c) Three kinds of statements to be discussed are of following forms:

1. the connect statement is of the form:

   CONNECT(mx; py; sz; ia1,...,ias; va1,...,vat)

2. the port statement has the form:

   PORT(mx; py; if1:T1,...,ifs:Ts; vf1:T1';...;vft:Tt') *)

3. the disconnect statement is: !!

(2) System variables.

We assume the system has been supplied with following variables:

(a) Three counters #Kmx, #Kpy, #Ksz, they are integral varibles, their ranges are: 0..n, 0..l, o..k respectively, they are used to indicate the index of the name of the master, the port, and the servant respectively.

(b) There are several arrays of auxiliary variables:

c-(#Kmx)-(#Kpy)-(#Ksz)-1 Of type T1

.....

c-(#Kmx)-(#Kpy)-(#Ksz)-s of type Ts

p-(#Kmx)-(#Kpy)-(#Ksz)-1 of type PT1' (i.e. pointer of T1')

.....

p-(#Kmx)-(#Kpy)-(#Ksz)-t of type PTt'

x-(#Kmx)-(#Kpy)-(#Ksz) of type B (i.e. boolean)

l-(#Kmx)-(#Kpy)-(#Ksz) of type B

n-(#Kmx)-(#Kpy) of type PB (i.e. pointer of boolean)

(3) Transformations.

(a) connect statement:

β(CONNECT(mx; py; sz; ia1,...,ias; va1,...,vat)) $\leftrightarrow$

o#c-(mx)-(py)-(sz)-1=#ia1$\wedge$...$\wedge$o#c-(mx)-(py)-(sz)-s=#ias$\wedge$

o#p-(mx)-(py)-(sz)-1=va1$\wedge$...$\wedge$o#p-(mx)-(py)-(sz)-t=vat$\wedge$

c#x-(mx)-(py)-(sz)=T$\wedge$ o#l-(mx)-(py)-(sz)=F;

ζ-(mx)-(py)-(sz): #l-(mx)-(py)-(sz)=F $\Rightarrow\uparrow$ g-(mx)-(py)-(sz);

g-(mx)-(py)-(sz): #l-(mx)-(py)-(sz)=T $\Rightarrow$

*) followed by condition and servant names

(b) port statement:

β(PORT(mx; py; if1:T1,...,ifs:Ts; vf1:T1',...,vft:Tt')

CONDITION cond;

SERVANT sz1,...,szq;

BEGIN &statement ENDPORT) ⟨-⟩

b-(mx)-(py):β(cond)⇒ d-(mx)-(py);

b-(mx)-(py):β(¬cond)⇒ b-(mx)-(py);

d-(mx)-(py): #x-(mx)-(py)-(sz1)=T ⇒ o#if1=#c-(mx)-(py)-(sz1)-1∧...∧

o#ifs=#c-(mx)-(py)-(sz1)-s∧

o#vf1=##p-(mx)-(py)-(sz1)-1∧...∧

o#v[illegible]-(py)-(sz1)-t∧

c[illegible]1-(mx)-(py)-(sz1)∧

[illegible];

.....

d-(mx)-(py): #x-(mx)-(py)-(szq)=T [illegible]c-(mx)-(py)-(szq)-1∧...∧

o#if[illegible]c-(mx)-(py)-(szq)-s∧

o#vf1=##p-(mx)-(py)-(szq)-1∧...∧

o#vft=##p-(mx)-(py)-(szq)-t∧

o#n-(mx)-(py)=1-(mx)-(py)-(szq)∧

↑e-(mx)-(py);

e-(mx)-(py): β(statement);↑h-(mx)-(py);

h-(mx)-(py): #x-(mx)-(py)-(sz1)=T ⇒ c##p-(mx)-(py)-(sz1)-1=#vf1∧...∧

c##p-(mx)-(py)-(sz1)-t=#vft∧

c#x-(mx)-(py)-(sz1)=F∧↑endport;

.....

h-(mx)-(py): #x-(mx)-(py)-(szq)=T = o##p-(mx)-(py)-(szq)-1=#vf1∧...∧

o##p-(mx)-(py)-(szq)-t=#vft∧

o#x-(mx)-(py)-(szq)=F∧↑endport;

endport;

(c) **disconnect statement:** β(!!) ⟨-⟩ o##n-(mx)-(py)=T

Following example is taken from (MY1).

Example 2. Semaphore

```
PROCESS semaphore
    VAR s: INTEGER;
    BEGIN
      s:=1;
      CYCLE
      // PORT ( semaphore; v )
          SERVANT q1,...,q10;
          BEGIN
           !!
           s:=s+1; ENDPORT
      // PORT ( semaphore; p )
          CONDITION s>0;
          SERVANT q1,...,q10;
          BEGIN
           !!
           s:=s-1; ENDPORT
      ENDCYCLE;
      END;
PROCESS qi;
    BEGIN
       ...(processing without resource)
          CONNECT (semaphore; p ; qi);
           ...(using resource)
          CONNECT (semaphore; v ; qi);
    END...(processing without resource)
```

The transformation rules given in (3) applied to this CP program has following result:

```
sem:[%io[s: I];
     %a [ o#s=1∧↑o cycle;
          cycle: ↑v  b-(sem)-(v)∧↑2 b-(sem)-(P);
          b-(sem)-(v):↑v d-(sem)-(v);
          d-(sem)-(v):#x-(sem)-(v)-(1)=T⇒ o#n-(sem)-(v)=1-(sem)-(v)-(1)∧↑v e-(sem)-(v)
              .............
          d-(sem)-(v):#x-(sem)-(v)-(10)=T⇒ o#n-(sem)-(v)=1-(sem)-(v)-(10)∧↑v e-(sem)-(v)
          e-(sem)-(v):[0#=n-(sem)-(v)=T; o#s=#s+1; ↑v h-(sem)-(v)];
          h-(sem)-(v): o#x-(sem)-(v)-(1)=F∧↑end;
          b-(sem)-(p): #s>0 ⇒↑2 d-(sem)-(p);
          b-(sem)-(p): #s≤0 ⇒↑2 b-(sem)-(p);
          d-(sem)-(p):#x-(sem)-(p)-(1)=T⇒ o#n-(sem)-(p)=1-(sem)-(p)-(1)∧↑2e-(sem)-(p);
              .............
          d-(sem)-(p):#x-(sem)-(p)-(10)=T⇒ o#n-(sem)-(p)=1-(sem)-(p)-(10)-↑2e-(sem)-(p);
          e-(sem)-(p):[o#=n-(sem)-(p)=T; o#s=#s-1; ↑2 h-(sem)-(p)];
          h-(sem)-(p): o#x-(sem)-(p)-(10)=F∧↑end;
          end: ↑o cycle ] ]
qi: [ ...
     %a[ o#x-(sem)-(p)-(i)=T∧ o#1-(sem)-(p)-(i)=F;
          g-(sem)-(p)-(i): #1-(sem)-(p)-(i)=F ⇒ ↑i g-(sem)-(p)-(i);
          g-(sem)-(p)-(i): #1-(sem)-(p)-(i)=T ⇒
              ..............
          o#x-(sem)-(v)-(i)=T∧ o#1-(sem)-(v)-(i)=F;
          g-(sem)-(v)-(i): #1-(sem)-(v)-(i)=F ⇒ ↑i g-(sem)-(v)-(i);
          g-(sem)-(v)-(i): #1-(sem)-(v)-(i)=T ⇒
        ... ] ]
```

Of course, in this example there are many informations e.g. 'sem' can be omitted. In order to enforce readability, we write the names of the master and the ports instead of their indices.

Acknowledgement: This work was done at 1981 summer during my visit to The Department of Computer Science, University of Maryland, College Park. Prof. Raymond, T. Yeh had done a lot of help to me. The author is very grateful to him and also to Prof. H. Mills for their friendly encouragement. I also enjoyed much discussing the concurrency problems with Mr. B. Chen during my visit there. His warm help to me is appreciated.

REFERENCES

(AFR1) Apt, K. et al. A Proof System for Communicating Sequential Process. TOPLAS vol.2, no.3, 1981.

(CC1) Cousot, P. and R. Cousot, Semantic Analysis of Communicating Sequential Process. CRIN-80-P033 1980.

(F1) Francez, N. et al. Semantics of Nondeterminism, Concurrency and Communication. J. Comp. Syst. Sci. vol 19, 1979.

(H1) Hoare, C. A. R. Communicating Sequential Process. CACM vol, no. 8. 1978.

(I1) Ichbiah, J. B. et al. Preliminary ADA Reference Manual. SIGPLAN Notice, June 1979.

(L1) Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. CACM vol. 21, no. 7, 1978.

(MY1) Mao, T. W. and Y. T. Yeh, Communication Port: A Language Concept for Concurrent Programming, IEEE Tran. on Softw. Eng. vol. SE-6, no.2, 1980.

(T1) Tang, C. S. Toward a Unified Logic Basis of Programming Languages. To appear as Techn. Rept. of Dept. of Comp. Sci. Stanford Univ. 1981.

(T2) __ __ __, A Hierarchical Specification-directed Program Design Methodology. Draf 1981.

(ZH1) Zhou C. C. and C. A. R. Hoare, Partial Correctness of Communicating Sequential Processes. [illegible]


DAT
ILM