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

<table>
<thead>
<tr>
<th>7. AUTHOR(s)</th>
<th>8. CONTRACT OR GRANT NUMBER(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chih-sung Tang</td>
<td>F49620-80-C-0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. PERFORMING ORGANIZATION NAME AND ADDRESS</th>
<th>10. PROGRAM ELEMENT, PROJECT, TASK AREA &amp; WORK UNIT NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science Department</td>
<td>61102F; 2304/A2</td>
</tr>
<tr>
<td>University of Maryland</td>
<td></td>
</tr>
<tr>
<td>College Park MD 20742</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. CONTROLLING OFFICE NAME AND ADDRESS</th>
<th>12. REPORT DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directorate of Mathematical &amp; Information Sciences</td>
<td>NOV 81</td>
</tr>
<tr>
<td>Air Force Office of Scientific Research</td>
<td></td>
</tr>
<tr>
<td>Bolling AFB DC 20332</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. DISTRIBUTION STATEMENT (of this Report)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release; distribution unlimited.</td>
</tr>
</tbody>
</table>

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.
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
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 F49620-80-00001 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
"\( \l_1 \l_2 \) changed into "\( \l_1 \l_2 \)"

P.17 line 10 changed into
"h-(sem)-(v): \( \forall v-(sem)-(v)-(1) = T \Rightarrow 0 \forall v-(sem)-(v)-(1) = F \) \( \l_0 \) end;"
"h-(sem)-(v): \( \forall v-(sem)-(v)-(10) = T \Rightarrow 0 \forall v-(sem)-(v)-(10) = F \) \( \l_0 \) end;"

P.17 line 17 changed into
"h-(sem)-(p): \( \forall v-(sem)-(p)-(1) = T \Rightarrow 0 \forall v-(sem)-(p)-(1) = F \) \( \l_0 \) end;"
"h-(sem)-(p): \( \forall v-(sem)-(p)-(10) = T \Rightarrow 0 \forall v-(sem)-(p)-(10) = F \) \( \l_0 \) end;"

PP.9-12 The semantic transformations of "\( \rightarrow \)" ought to be changed according to the following pattern:
\( \beta(A \rightarrow B) \leftrightarrow \{ \beta(A); \beta(B) \} \)
\( \beta(A \rightarrow (B \rightarrow C)) \leftrightarrow (A \rightarrow B \rightarrow C) \leftrightarrow \{ \beta(A); \beta(B); \beta(C) \} \)
\( \beta(A \rightarrow (P \mid Q)) \leftrightarrow \{ \beta(A); \l_1 \}; 1: \beta(P); 1: \beta(Q) \)

So, the programs in Example 2, 3 must be changed accordingly.
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 a 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-(#abc)-age is a name schema, here #abc is a counter corresponding to abc.

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

\[
\begin{align*}
\&[\text{#lb=sqrt } \Rightarrow \text{o#lb=Im};
\&\&[\text{#lb=In } \Rightarrow \text{I#o#lb=In}];
\&\&[\text{#o#lb=In } \Rightarrow \text{I#o#lb=Ik}];
\&\&[\text{#lb=Ik } \Rightarrow \text{I#o#lb=Ip}];
\&\&[\text{#lb=Ip } \Rightarrow \text{I#o#lb=il}];
\&\&[\text{#lb=1l } \Rightarrow \text{o#Ik=0#o#Ip=1#o#lb=12}];
\&\&[\text{#1b=12#Ip)#In } \Rightarrow \text{#o#lb=14}];
\&\&[\text{#1b=12#Ip)#Im } \Rightarrow \text{#o#lb=13}];
\&\&[\text{#1b=13 } \Rightarrow \text{o#Ip=Ip+2#Ik+3#o#lk=Ik+1#o#lb=12}];
\&\&[\text{#1b=14 } \Rightarrow \text{#Im=Ik{o#lb=stop}]}]
\end{align*}
\]

This is the internal form of a program. It is a 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:

\[
\begin{align*}
\&[\text{n=sqrt;}
\&\&[\text{#m=1}];
\&\&[\text{#n=1}];
\&\&[\text{#k=1}; \text{p=1}];
\&\&[\text{o#k=0}; \text{o#p=1}];
\&\&[\text{#lb=1}];
\&\&[\text{#p=#m } \Rightarrow \text{#14}];
\&\&[\text{#p=#m } \Rightarrow \text{o#p=2#k+3#o#k=Ik+1#o#lb=12}];
\&\&[\text{#m=#k/#14}]
\end{align*}
\]

(6) In order to use XYZ/E to express the formal semantics of other languages, a metalanguage is introduced in which 'p' is a semantic mapping, it maps each construct of the object language into a string of entities in XYZ/E; '->' is 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 symbolism 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 semantics is as easily defined denotationally as any logic system; while it is also a lower level language, the semantics of other programming systems 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 treat time order of a distributed system is known a difficult problem (11). In this paper, a new approach of time philosophy is adopted which seems able to simplify the situation.

To 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) none of the name of variables and labels must be knowable from the outside, otherwise no communication is possible.

Our basic philosophy is that even though above assumption (2) 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 or

although the measure of present time of process P_i may be quite different from that of P_j, both must be coincident in THIS PRESENT TIME. The minimum 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 P_j (according to P_j's time system) is equal to the present time value of variable u in P_i (according to P_i's time system)" or P_k without presupposing any of P_i, P_j, P_k 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: \[ P : [ P_1/\ldots/P_n] \]
here: \( P_i : S_1, i=1,\ldots,n \), \( P_i \) process, \( S_i \) statement.

statements includes:
(a) input command: \( P_j x \) (in \( S_i \))
(b) output command: \( P_i y \) (in \( S_j \))
(c) guarded commands: \( [B_1 \rightarrow S_1 \mid B_2 \rightarrow S_2 \mid \ldots \mid B_k \rightarrow S_k] \)
(d) repetition: \( = [B_1 \rightarrow S_1 \mid B_2 \rightarrow S_2 \mid \ldots \mid B_k \rightarrow S_k] \)
(e) skip and assignment
(f) statements sequence: \( S_1; \ldots; S_m \)

boolean expressions includes:
(a) input command: \( P_j x \) (in \( B_j \))
(b) output command: \( P_i y \) (in \( S_i \))
(c) conventional boolean expression: \( b_i \)
(d) boolean sequence: \( B_1; B_2; \ldots; B_m \)

The exact explanation of these constructs, see \((31),(31)\).

(2) Presupposition before transformation.

(1) Assuming that in front of each statement there has already been a label and to the right of each process \( P_i \) there is a label end. Consequently, for each boolean expression or statement, a label to its nearest right can always be found. Let next be the nearest label right to \( B_i \) or \( S_i \), it can be determined in following way:
(a) For \( \downarrow (i=1) \div \ldots \div i \mid B_i; \ldots; b_k \rightarrow \downarrow S_i \); \( l \) is the next for \( B_i \).
(b) \( \downarrow (i=n) \div \ldots \div i \mid S_i; \ldots; l \div S_i \); \( l \) is the next for \( S_i \).
(c) If \( S \) is the last statement in any \( S_i \) (\( i=1 \) to \( n \)) in following statement:
\[ B_1 \rightarrow S_1 \mid B_2 \rightarrow S_2 \mid \ldots \mid B_n \rightarrow S_n \] ; \( 1: S' \), here \( S' = \emptyset \),
then \( l \) is the next for this \( S \).
(ii) To each process \( P_i \) in \( [P_1//P_2//...//P_n] \) a private control variable \( t_i \) or its abbreviated form \( \uparrow t_i \) is assigned and in addition, a common control variable \( \# b \) or \( \uparrow t \) is also available.

(iii) To each input command \( P_j(x_i) \) in \( S_i \) (or \( P_i \)), a boolean \( t_j \) is assigned;
To each output command \( P_i(y_j) \) in \( S_j \) (or \( P_j \)), a boolean \( t_j \) is assigned.
Let \( Pu \# v \) represent either \( P_j(x_i) \) or \( P_i(y_j) \).

(3) Transformation from CSP to \( XZ/\xi \).

(i) Each statement sequence is transformed into a compound statement, i.e.
\[ \beta(S_1;...;S_k) \Rightarrow [\beta(S_1);...;\beta(S_k)] \]

(ii) Each boolean sequence \( b_1;...;b_k \) is transformed into a conjunction, i.e.
\[ \beta(b_1;...;b_k) \Rightarrow \beta(b_1) \wedge \beta(b_2) \wedge ... \wedge \beta(b_k) \]
\( \Rightarrow \beta(b) \) if there is no input-output command in \( b_1;...;b_k \);
\( \Rightarrow \beta(Pu \# v) \) if \( Pu \# v \) occurs in \( b_1;...;b_k \).

(iii) The concurrent program is transformed as follows:
\[ \beta(P_i;P_1;S_1//...//P_n;S_n) \Rightarrow P_i;\uparrow t[P_1//...//P_n;S_n]; \]
\[ P_i;\beta(S_1);...;P_n;\beta(S_n); \]

(iv) The guarded commands are transformed in following way (assuming in \( P_i \):
\[ \beta(n;[B_1;...;B_k];l_k;S_k];\text{next} ;S'_1) \Rightarrow n;\beta(B_1) \Rightarrow l_k;\beta(S_1) \uparrow t; \text{next} ;S'_1; \]
\[ ............... \]
\[ n;\beta(B_k) \Rightarrow l_k;\beta(S_k) \uparrow t; \text{next} ; \]
\[ n;\beta(B_1) \wedge ... \wedge \beta(B_k) \uparrow t; \text{next} ; \]
\[ \text{next}; \beta(S'_1); \]

Note that in the original definition of conditional elements in \( (\xi) \),
two conditions with one common label must be mutually contradictory, i.e.
\[ n;E \Rightarrow S_1; n;\neg E \Rightarrow S_2; \]
These two conditional elements are logically equivalent to following
formula: ' if \( \# b = n \) then ( if \( E \) then \( S_1 \) else \( S_2 \) ) else false '.
But in present case the conditions can be true simultaneously, and
in that situation we have to choose one of those \( S_i \) following them.
The logical meaning of the latter is quite different from the former.

Let the conditional elements transformed from guarded commands are:

\[ n: B1 \rightarrow l1: S1'; n: B2 \rightarrow l2: S2'; n: \sim B1 \land B2 \rightarrow \text{next}; \]

These three conditional elements are logically equivalent to following case:

\[ \text{if } \#\text{lb}=n \text{ then } (\text{if } B1 \land B2 \text{ then } S1'; S2') \]
\[ \text{else if } B1 \text{ then } S1' \]
\[ \text{else if } B2 \text{ then } S2' \text{ else } \#\text{lb=next}); \]

here \( S1'S2' \) means that \((S1' \lor S2) \land \sim (S1 \land S2)\).

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

\[ \beta(n: \sim B1 \rightarrow l1: S1 \ldots Bk \rightarrow lk: S1) \leftrightarrow n; \]
\[ \text{.........} \]
\[ n; \beta(Bk) \rightarrow lk; \beta(Sk) \land \beta(S1) \land \beta(n); \]
\[ n; \gamma(B1) \land \ldots \beta(Sk) \land \beta(n); \]
\[ \text{for these guarded commands } (n: \beta(B1) \rightarrow l1: \beta(S1) \land \beta(R), \beta(n); \]

(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 (assuring in Pi):

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

\[ \beta(Pj?xij) \leftrightarrow \#xij=?\text{comm} \]
\[ \beta(Pk?yik) \leftrightarrow \#yik=?\text{comm} \]

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:

\[ \text{comm: } \#xij=?\text{comm} \rightarrow \#xij=?\text{comm} \rightarrow \#xij=?\text{nextxij} \land \#nextxij \land \#nextxij; \]

here \( \#\text{nextxij} \) are the next label of \( Pj?xij \) in Pi and that of \( Pk?yik \) in Pk respectively. As for \( Pk?yik \), we need not insert the corresponding conditional element in comm, for its input partner \( Pj?xij \) in Pi would do that.
(ii) As booleans (assuming in Pi):

Let us assume the guarded command \( \phi \) where they occur is of following form:

\[ \phi : \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаЫ} \rightarrow \text{condаХ}
Let us explain the above transformation with a typical example quoted from (A1).

Example 1. $F:[P1//P2//P3]$

$P1:[F3x−→111:510;P3;y−→112:52]; L13: [52;P2:u−→14;S]; P3?u−→15:S]; end$

$P2:[P1*−→121:55]; P11−→122:56; P3?e−→123:57]; end$

$P3: P17:131: [51−→132:55]; end$

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

$P: \uparrow 1 P1\uparrow 2 P2\uparrow 3 P3$

$P1: c*r12=d*15=b*15; c*15$

$P2: c*r21=b*15=d*15; c*15$

$P3: c*r31=b*4; c*4$

$P4: \uparrow 1 L1\uparrow 2 P2$

$comm: \#r12=x\#s21=y \#o=t \#o*r12=x\#o*r21=x\#o*r31=x\#o*e=r3 \#o*e=2 \#o*e=3 \#o$
The Formal Semantics of Zhou-Hoare's CSP with Named-Channels

Zhou-Hoare (ZHL) introduces a formal system to treat 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:

1. The process in their system is transformed into a procedure in XYZ/E;
2. 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 %1 part, %0 part, or %io part respectively) of the procedure;
3. The type of the values of the channels is treated as allocational formula; as in (T2), we allow sets as allocational formulas;
4. In order to describe the relations between the channels (I/O variables) of different processes, 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-rocopier-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.
5. 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 accordance 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 (2.91). To my understanding, "A→B" in (2.91) ought 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 (2.91), 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 transformations are;

\[(A→B)→β(A)≡β(β(B)), \ β(A→B; B→C)→β(A)≡β(B) \ P3(5) \ L3(5) \ L3(C)\]

(2) In "c!e→P", c is transformed into an output or input-output variable ec and e is transformed into a corresponding expression; so the whole expression is transformed as \((c!e→P)→c→β(e)→β\ P\). Note, this variable ec must be declared at the \%o or \%o part of the procedure.

(3) In "c?x:M→P", c is transformed into an input or input-output variable ec which is declared at the \%i part or \%i0 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 \(β(c?x:M→P)→c→x=β\ P\).

(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 \(β((P;Q))→lpqβ(P); lpqβ(Q)\).

(5) As for \((P;ijy Q)\) and \((\text{chain } L; P)\), 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 equation in Zebra-Share's system:

```plaintext
copier = (inputNat: NAT → wireNat; wireNat = copier).
recopier = (wireNat: NAT → outputNat; outputNat = recopier).

(ecn wire; (copier | recopier))
```

which is represented pictorially by following figure:

```
  src --------copier----wire----recopier----output.
```

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

```plaintext
*p src:
  %i[inputNat];
  %o[outputNat];
  *p copier:
    %i[inputNat];
    %o[wireNat];
    %w[wireNat];
    %a [le: inputNat → wireNat]
    %wireNat = [le]
  *p recopier:
    %w[wireNat];
    %o[outputNat];
    %y[outputNat];
    %a [lr: wireNat → outputNat]
    %outputNat = [lr]
  %a [lsrc: copier | recopier | Sink;
      lsrc | copier-input | wire-input;
      lsrc | recopier-wire | copier-wire;
      lsrc | output | recopier-output; lsrc]
```
Example 3. The given processes equations are:

\[
\begin{align*}
\text{sender} & \xrightarrow{\text{input}y} \{x \in \text{sender} \mid \text{wire}(x, y) = \text{ACK}\} \\
\text{receiver} & \xrightarrow{\text{wire}(y)} \{x \in \text{receiver} \mid \text{wire}(x, y) = \text{NACK}\}
\end{align*}
\]

The corresponding pictorial representation is as follows:

```
+----------------+----------------+----------------+----------------+----------------+
<table>
<thead>
<tr>
<th>protocol</th>
<th>input</th>
<th>wire</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+----------------+----------------+----------------+----------------+
```

The procedure obtained after transformation is:

```plaintext
*p[protocol;
  %i[input: M];
  %s[output: M];
  %p[sender;
    %i[input: M];
    %w[wire: M];
    %v[y: M];
    %p[q;
      %i[wire: \{ACK, NACK\}];
      %w[wire: M];
      %v[x: M];
      %a[1][lq:
        lq: \text{wire} = x \Rightarrow 0 \text{wire} = \text{ACK};
        lq: \text{wire} = x \Rightarrow 0 \text{wire} = \text{NACK};
        \text{wire} = \text{ACK} \Rightarrow \text{sender};
        \text{wire} = \text{NACK} \Rightarrow \text{sender};
      ];
    %a[2][lq:
      lq: \text{input} = y \Rightarrow \text{wire} = y \\
    ];
    %p[receiver;
      %i[wire: M];
      %o[wire: \{ACK, NACK\}];
      output: M];
    %v[z: M];
    %a[3][lr:
      lr: \text{wire} = x \Rightarrow \text{wire} = \text{ACK};
      lr: \text{wire} = x \Rightarrow \text{wire} = \text{NACK};
    ];
  ];
```

```
wireACK ⇒ wireACK

wireACK ⇒ wire

protocol sender receiver

protocol sender input = protocol input;
receiver wire = sender wire;
protocol output = receiver input;

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

As for the proof rules of Snoc-Hoare system, they become meta-rules of AX3/E
system. To prove them within the framework of temporal logic event more straight-
forward than to verify the validity through the computational semantics model.
The Formal Semantics of Mao-Yeh's CP

Mao-Yeh's CP is a language concept for concurrent programming introduced in M1. It is substantially synonymous with ADA Tasking. For the concept of CP has been more neatly defined in (M1), it is chosen as the model in our discussion in this paper. The method introduced here can be reviewed 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 (M1) 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 send 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 servent 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 i.e. \( i \) master i.e. \( m \), one for the port i.e. \( p \), and one for the servant i.e. \( s \); besides, the formal parameters of informations in port statements are of following form:

the input formal parameters are: \( i'_1 : i'_1, \ldots, i'_s : i'_s \),

the input-output formal parameters are: \( v'_1 : T'_1, \ldots, v'_t : T'_t \).

The corresponding actual parameters in connect statements are: \( i_1, \ldots, i_s \) and \( v_1, \ldots, v_t \).
Three kinds of statements to be discussed are of following form:

1. The connect statement is of the form:
   \[
   \text{CONNECT}( \text{mx}; \text{py}; \text{sz}; \text{ial}, \ldots, \text{iav}; \text{val}, \ldots, \text{vat} )
   \]

2. The port statement has the form:
   \[
   \text{PORT}( \text{mx}; \text{py}; \text{ifi}; \text{ti}, \ldots, \text{ifs}; \text{ti}'; \text{vifi}; \text{ti}', \ldots, \text{vifs}; \text{ti}' )
   \]

3. The disconnect statement is: !!

(2) System variables.

We assume the system has been supplied with following variables:

(a) Three counters \#Kx, \#Kpy, \#Ksz, they are integral variables, their ranges are: 0..n, 0..1, 0..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:

- \( c(\#Kx)(\#Kpy)(\#Ksz) \) of type T1
- \( p(\#Kx)(\#Kpy)(\#Ksz) \) of type T2
- \( p(\#Kx)(\#Kpy)(\#Ksz) \) of type T1' (i.e. pointer of T1')
- \( x(\#Kx)(\#Kpy)(\#Ksz) \) of type B (i.e. boolean)
- \( l(\#Kx)(\#Kpy)(\#Ksz) \) of type B
- \( n(\#Kx)(\#Kpy) \) of type PB (i.e. pointer of boolean)

(3) Transformations.

(a) connect statement:

\[
\beta(\text{CONNECT}( \text{mx}; \text{py}; \text{sz}; \text{ial}, \ldots, \text{iav}; \text{val}, \ldots, \text{vat} )) \leftrightarrow \\
\begin{align*}
&c\left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right) \left( \text{ial} \right) \ldots \left( c \left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right) \left( \text{iav} \right) \right) \land \\
&c \left( \text{py} \right) \left( \text{sz} \right) \left( \text{val} \right) \ldots \left( c \left( \text{py} \right) \left( \text{sz} \right) \left( \text{vat} \right) \right) \land \\
&c \left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right) = \text{T} \land c \left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right) = \text{F}; \\
&c \left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right) \left( \text{ial} \right) = \Rightarrow g(\left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right)); \\
&g(\left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right)) \left( \text{ial} \right) = \Rightarrow c(\left( \text{mx} \right) \left( \text{py} \right) \left( \text{sz} \right)) = \text{T} \Rightarrow
\end{align*}
\]

* followed by condition and servant, names.
(b) port statement:

\[ \beta(\text{PORT}(mx; py; ifl:Tl, ..., ifs:Tf; vfl:Tf', ..., vft:Tf')) \]

CONDITION cond;

SERVANT s1, ..., szq;

BEGIN statement ENDPORT \arrow
\begin{align*}
&b.(mx)-(py); \beta(\text{cond}) \Rightarrow d.(mx)-(py);
&b.(mx)-(py); \beta(\text{cond}) \Rightarrow b.(mx)-(py);
&d.(mx)-(py): \#x.(mx)-(py)-(sz1)=T \Rightarrow o\#if1 \#c.(mx)-(py)-(sz1)-1\ldots 1
\end{align*}

\begin{align*}
&\text{condition:} \quad \text{cond};
&\text{servant:} \quad s1, ..., szq;
&\text{begin:} \quad \text{statement ENDPORT \arrow}
\end{align*}

\begin{align*}
&b.(mx)-(py): \beta(\text{cond}) \Rightarrow d.(mx)-(py);
&b.(mx)-(py): \beta(\text{cond}) \Rightarrow b.(mx)-(py);
&d.(mx)-(py): \#x.(mx)-(py)-(sz1)=T \Rightarrow o\#if1 \#c.(mx)-(py)-(sz1)-1\ldots 1
\end{align*}

\begin{align*}
&\text{condition:} \quad \text{cond};
&\text{servant:} \quad s1, ..., szq;
&\text{begin:} \quad \text{statement ENDPORT \arrow}
\end{align*}

\begin{align*}
&d.(mx)-(py): \#x.(mx)-(py)-(szq)=1 \Rightarrow o\#if1 \#c.(mx)-(py)-(szq)-1\ldots 1
\end{align*}

\begin{align*}
&\text{condition:} \quad \text{cond};
&\text{servant:} \quad s1, ..., szq;
&\text{begin:} \quad \text{statement ENDPORT \arrow}
\end{align*}

\begin{align*}
&e.(mx)-(py): \beta(\text{statement}); e.(mx)-(py);
&h.(mx)-(py): \#x.(mx)-(py)-(sz1)=T \Rightarrow o\#p.(mx)-(py)-(sz1)-1 \#vfl\ldots 1
\end{align*}

\begin{align*}
&\text{condition:} \quad \text{cond};
&\text{servant:} \quad s1, ..., szq;
&\text{begin:} \quad \text{statement ENDPORT \arrow}
\end{align*}

\begin{align*}
&h.(mx)-(py): \#x.(mx)-(py)-(szq)=T \Rightarrow o\#p.(mx)-(py)-(szq)-1 \#vfl\ldots 1
\end{align*}

\begin{align*}
&\text{condition:} \quad \text{cond};
&\text{servant:} \quad s1, ..., szq;
&\text{begin:} \quad \text{statement ENDPORT \arrow}
\end{align*}

endport;

(c) disconnect statement: \[ \beta(11) \iff o\#n.(mx)-(py)=T \]
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

  // PORT ( semaphore; p )
  CONDITION s>0;
  SERVANT q1, ..., q10;
  BEGIN

  s:=s+1; ENDPORT

  s:=s-1; ENDPORT
ENDCYCLE;
END;
PROCESS q1;
BEGIN
...
  CONNECT ( semaphore; p ; q1);
...
  CONNECT ( semaphore; v ; q1);
...
END

// PORT ( semaphore; p )
CONDITION s>0;
SERVANT q1, ..., q10;
BEGIN
  ++
  s:=s+1; ENDPORT

// PORT ( semaphore; v )
CONNECT ( semaphore; v ; q1);

...(processing without resource)
The transformation rules given in (3) applied to this CP program have the following result:

\[ \text{sem} : [\text{io} \{ \text{a: I} \} ] \]

\[ \text{\%a [ off=1, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

\[ \text{\%a [ of=0, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

\[ \text{\%a [ of=0, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

\[ \text{\%a [ of=0, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

\[ \text{\%a [ off=1, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

\[ \text{\%a [ off=1, cycle; cycle: } \hat{Y} = \text{sem}(v) \hat{Y} ; \text{b(sem)}(v) ; d(sem)(v); \hat{x} = \text{sem}(v) \hat{x} (1) \Rightarrow \text{off}(\text{sem})(v) = \text{sem}(v)(1) \hat{x} \text{in}(\text{sem})(v) \]

Of course, in this example there are many informations e.g. sem can be omitted.
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


