23

A temporal logic language for system implementation

Z. Tang
Institute of Software, CAS
P. O. Box 8718, Beijing, 100080, P. R. China
Tel: (8610) 62556910, Fax: (8610) 62562533
E-mail: cst@ox1.ios.ac.cn

Abstract

The XYZ system consists of a temporal logic language XYZ/E and several supporting tools. This paper introduces many special features of XYZ/E with its tools which are desirable for a good implementation language in addition to portability and efficiency. These features include (1) its suitability for stepwise refinement, specification of different abstract levels, verification and rapid prototyping, (2) modules and visual tools to support programming in the large and and (3) a simple, formal method and a corresponding tool for transforming a program from another source language into XYZ/E, a feature which is useful for software reengineering and for embedding standard programs from other languages into an XYZ/E.

Keywords

Temporal logic language, system implementation language, stepwise-refinement, specification, verification, prototyping, module, visual tools

Systems Implementation 2000 R. Nigel Horspool (Ed.)

1 INTRODUCTION

The C language and its derivatives have been accepted for decades as satisfactory system implementation languages, perhaps due to their portability and efficiency. Recently, Java is being considered a good replacement for C in networking applications. However, from a software engineering methodology point of view, there are persuasive reasons to believe that both C and Java still lack features which are required when building a complicated large system, such as a reactive and/or a distributed system. Although portability and efficiency are important properties of a system implementation language, reliability, maintainability and reusability are also important. With respect to modularity and the concepts of programming in the large, object-oriented programming still leaves much to be desired. In this paper, we explain how these problems are dealt with when using our temporal logic language (TTL) XYZ/E as a system implementation language (Tang, 1996) (Pnueli, 1996).

With XYZ/E, there are four groups of tools which support the following features: (1) stepwise refinement, specification, verification and rapid prototyping, (2) modularity and the corresponding intelligent integrated visual environment to support analysis and design of the programs represented with four kinds of modules, (3) formal practical translation of programs in other source languages to the XYZ/E language, and (4) miscellaneous tasks.

XYZ/E is both a linear time temporal logic system and a practical programming language with a conventional programming style. It is characterized through its representation of the dynamic semantics (i.e. state-transition) and the static semantics (i.e. pre-post specification) in a uniform framework. Its data structures are Pascal-like (with some extensions). Its control structures can be represented in three forms: (a) the state transition command form, (b) the higher level statement form, and (c) the production rules form. In addition, it has modules for programming in small, i.e. procedures and processes of flowchart form structures. There are two kinds of modules for programming in large with many novel characteristics, which seem to be essential for system implementation.

To take advantage of C and Java, the executable sublanguage of XYZ/E (XYZ/EE) is implemented by translating to the C family or to Java. The correctness of one version of the translation has been proved formally with the operational semantics invented by Plotkin-Liwei (Shen and Tang, 1994). We believe this two level structure is a practical approach for new system implementation languages. The basis of the implementation is to reduce to the Kripki model, then to the Lucid model (Ashcroft and Wadge, 1977), and finally to reduce to the von Neumann model.

2 STEPWISE-REFINEMENT, SPECIFICATION, VERIFICATION AND PROTOTYPING

The XYZ System distinguishes two kinds of program evolution. One kind is stepwise refinement, where each step realizes a replacement of one well-formed piece of abstract specification with a meaningful section of program containing some state-transition details (e.g. assignments). The semantics of the former are preserved by (i.e. deducible from) the latter. The other kind covers all development processes different from stepwise refinement, where there are modifications of the program at each step, resulting in changes to the semantics of the program. Although both kinds of program evolution can occur in the development of a XYZ/E program, our attention is focussed on stepwise-refinement, because it is more concerned with the transition from static semantics to dynamic semantics. In the transition process, the consistency of semantics must be checked by verification or by other means, such as rapid prototyping. During this process, a program will often become a mixture of dynamic semantics with static semantics. XYZ/E is especially good at representing this process of evolution.

XYZ/E contains two basic command forms as follows:

$LB=y\text{^}R=>\$O(v1,\ldots,vk)=(e1,\ldots,ek)\text{^}\$OLB=z$ (1)
$LB=y\text{^}R=><>(Q\text{^}LB=z)$ (2)

Here "*LB*" represents a system control variable denoting a label; "y" and "z" represent the definitional label and the forward label respectively; "=>" and";" are notations occurring in the syntactic level of the command that represent the logical connectives of implication and conjunction, respectively; "R" and "Q" are two First-Order logic (FOL) formulas that represent conditions and actions (or pre and post assertions), respectively; "$O" and "<>" are temporal operators meaning "next time" and "eventually", respectively. The formula "$O(*v1*,...,*vk*)=(*e1*,...,*ek*)" represents a parallel assignment of the values of the expressions *e1*,...,*ek* at the current instant of time to the corresponding variables *v1*,...,*vk* with identical types at the next time instant. These kinds of command can be connected together to form an algorithm by using the connective ";". Commands of form (1) can be used to represent a step of an executable algorithmic program, while a command of form (2) represents the pre and post assertions of one procedure call. Consequently, the former are used to represent the dynamic semantics, while the latter represent static semantics. These two kinds of commands can be combined to form a program at various levels of abstraction during the process of stepwise refinement. Let us see a small example:

Ex. 1. s=gcd(a,1)+gcd(a,2)+ ... +gcd(a,b) for a>0 and b>0

Step one: starting with specification of pure abstraction.

[] [*LB*=START_sg^a>0^b>0=><>(SUMGCD(a,b,*s*)^*LB*=STOP)]

WHERE (GCD(a,1,1)–>SUMGCD(a,1,1))^
(SUMGCD(a,*t*,*z*)^GCD(a,*t*+1,*f*)^*s*=*z*+*f*–>SUMGCD(a,*t*+1,*s*))^
(GCD(*m*,*n*,*f*)≡*f*\|*m*^*f*\|*n*^∀*w*=<MIN(*m*,*n*)(*w*\|*m*^*w*\|*n*–>*w*\|*f*))^
(*p*\|*q*≡rem(*p*,*q*,0))^
(rem(*e*,0,0))^
(rem(*e*,*d*,0) –>rem(*e*,*d*+1,1))^
(rem(*e*,*d*,*c*)^*c*>0^*c*=*e*-1–>rem(*e*,*d*+1,0))^
(rem(*e*,*d*,*c*)^*c*>0^*c*/=*e*-1–>rem(*e*,*d*+1,*c*+1))

Step two: to change the summation part of the specification into executable commands but to leave GCD(a, b, *f*) still in abstract form.

[] [*LB*=START_sg^a>0^b>0=>$O*f*=1^$O*i*=1$O*LB*=l1;
LB=l1^*i*/=b+1=>$O*LB*=l2;
LB=l1^*i*=b+1=>$O*s*=*r*^$O*LB*=STOP;
LB=l2=><>(GCD(a,b,*f*)^*LB*=l3);
LB=l3=>$O*r*=*r*+*f*^$O*i*=*i*+1^$O*LB*=l1]
WHERE(GCD(*m*,*n*,*f*)≡*f*\|*m*^*f*\|*n*^∀*w*=<MIN(*m*,*n*)(*w*\|*m*^*w*\|*n*–>*w*\|*f*))^
(*p*\|*q*≡rem(*p*,*q*,0))^
(rem(*e*,0,0))^
(rem(*e*,*d*,0) –>rem(*e*,*d*+1,1))^
(rem(*e*,*d*,*c*)^*c*>0^*c*=*e*-1–>rem(*e*,*d*+1,0))^
(rem(*e*,*d*,*c*)^*c*>0^*c*/=*e*-1–>rem(*e*,*d*+1,*c*+1))

Step three: to change the abstract predicate GCD into executable form.

[] [*LB*=START_sg^a>0^b>0=>$O*f*=1^$O*i*=1^$O*LB*=l1;
LB=l1^*i*/=b+1=>$O*LB*=l2;
LB=l1^*i*=b+1=>$O*s*=*r*^$O*LB*=STOP;
LB=l2=>$O*x*=a^$O*y*=1^$O*LB*=l21;
LB=l21^*x*=*y*=>$O*LB*=l23;
LB=l21^*x*/=*y*=>$O*LB*=l22;
LB=l22^*x*>*y*=>$O*x*=*x*-*y*^$O*LB*=l21;
LB=l22^*x*<*y*=>$O*y*=*y*-*x*^$O*LB*=l21;
LB=l23=>$O*f*=*x*^$O*LB*=l3;
LB=l3=>$O*r*=*r*+*f*^$O*i*=*i*+1^$O*LB*=l1]

The XYZ system provides two ways to check the semantic consistency between two consecutive steps. One is by means of Hoare Logic verification. The XYZ system contains an efficient Hoare Logic verifier called XYZ/VERI. Zhang (1995) has explained this system. The other way to check consistency of semantics is by so-called rapid prototyping. We consider the result of replacement at each step in the process of stepwise refinement, i.e. a program that is a mixture of static and dynamic semantics, as a prototype. By providing values for its input variables, we try to find the values of its output variables as the result of its evaluation. Because

it contains both executable commands and non-executable abstract commands, our approach is to treat each command of form (2) as a procedure call, and to transform its pre and post assertions (R and Q together with the definitions in the wherepart) into Horn clause form and then to evaluate them in a Prolog-like way by a subsystem called XYZ/PEO. For Ex 1, this process is straightforward.

Yan and Tang (1997a) provide a realistic example with much larger scale software. They implement the well known Abrial "Steam Boiler Control Specification" using XYZ/E. The example includes specification of safety and liveness properties of the problems, five steps of replacement in the process of stepwise refinement and the final implementation of the problem with XYZ/BE (Manna and Pnueli, 1996). Yan and Tang (1997b) show a visual representation of the dynamic process of executing the final program. The "Steam Boiler" example was presented at a Dagstuhl meeting as a standard problem to test whether a language system (together with its tools) is sufficiently realistic to handle the large scale software for a practical real-time control system (Abrial et al., 1996).

3 MODULES AND VISUAL TOOLS

In XYZ, modules for programming in large are divided into two kinds: one is oriented toward the problem domain, the other toward the process of computation in communication.

The Agent module

As a representative of the former, a new concept of OO module called an *Agent* (in a new sense of this term) is introduced to enhance the autonomy of objects in communication. To every class of repository data module called Package, a process is assigned to be paired as its "manager". The result is an agent.

Agent = (Process, Package) (3)

The relationship between the Package part and the Process part is, by default, to export all the variables and operations defined in the Package into the Process. As a result, they become the local variables and procedures in the Process part. All these IMPORT-EXPORT actions are realized at compilation time by changing the states of these entities in the storage shared by these two parts. When the Agent as the representative of the object communicates with outside, the message passing actions are really performed in the Process part on behalf of the Agent. Therefore, the Process part represents the dynamic semantics of the object; the Package part represents the static semantics and the characteristics of encapsulation and inheritance of the object. However, their connection is established at compilation time. Our experiences have shown the naturalness and the convenience of this new OOP mechanism in both its implementation and application. Objects become fully autonomous in communication on this basis.

Parallel statement process

We now introduce a new module oriented toward the process of computation in communicating concurrent programs. This kind of module is absolutely required by reactive and distributed systems. Its functionality cannot be properly captured by OO modules. Our new module concept is, in fact, the result of modularizing a parallel statement into a new kind of process called the *Parallel Statement Process* (PSP). PSP is a kind of process which contains in its body a unique parallel statement together with the instantiation equations of its items and possibly some communication commands to connect its items with outside. In fact, this kind of parallel statement has been frequently used as a major constituent of the higher level structure of many international standard languages oriented toward communication systems, such as SDL, Estelle, etc. What we have done here is only to refine it into an independent modular concept in contrast to OO modules, In order to distinguish this kind of processes from such processes in their ordinary sense, we call the latter *Flowchart Form Processes* (FFP).

PSP representing the semantics of a parallel statement is suitable to considered as a kind of module for programming in large because it is composable from its substructures. For example, a parallel statement of following form:

"||[P1; ||[||[P2;P3];P4;P5]]"

is meaningful. But this kind of "composability" is only syntactic. It presupposes a semantic condition of composability in the following sense. Let PRE and POST be pre and post assertions of the given parallel statement and PRE_i and POST*i*, *i*= 1,...,k be the pre and post assertions of its items respectively. The semantic conditions of the given parallel statement for composability from its items are represented by following equivalencies:

PRE ≡PRE1 ^ ... ^ PREk (4.1)

POST≡POST1 ^ ... ^ POSTk (4.2)

Clearly, these conditions reduce the semantics of concurrent programs to those of sequential programs. They can be considered as the basis for conventional concurrent programming. Without such a basis, the semantics of concurrent programs would become in practice as unmanageable as in CCS. The problem of guaranteeing the validity of (4.1) and (4.2) is a difficult task, left open for almost twenty years. I hope that adopting a simple approach helps shed some light on the solution to this problem in a practical way. The basic thought underlying our approach is to consider the input variable *x* of the input predicate "c?*x*" as a special kind of input parameter of the process. Just as ordinary input parameters require preconditions to constrain their values in order to guarantee the validity of the post assertions, input variables such as *x* in the input predicate c?*x* also need some constraints to protect the semantics of the process against destructive messages from outside. The problems are (1) how to find the constraint, (2) how to locate it in an appropriate place of the process, and (3) how to represent it in the

specification. Fortunately (1) can be solved by automatically generating the constraint using an ordinary Hoare logic verifier as our XYZ/VERI (Zhang, 1995). It starts with the post assertion at the end of the process and moves the assertion upward until the related input command is reached, Problem (2) can be solved by adding a new command with negation of the constraint res(x) as its condition to follow the original input command in following way:

LB=y^R=>$O(c?$x$^$LB$=NEXT); (5.1)
LB=y'^~res(x)=>OLB$=$y$|$OLB=forward. (5.2)

Here (5.2) means that if res(x) is not satisfied then the received message is discarded, and control returns to the original place waiting for a new message. Consequently, any destructive message from outside is rejected and all accepted messages are safe ones. Problem (3) is difficult because res(x) sometimes contains variables with dynamic values which cannot be used in static specifications. A practical approach is to replace these variables with their upper or lower bounds. This kind of situation does not occur very often. Let us see a simple example.

Ex. 2. To evaluate z=√(y)+ √(x) by ||[P1;P2].

```
P1: [LB1=l1=>$Os=√(y) ^ $OLB1=l2;
     LB1=l2=>$O(c?w^LB1=l3);
     LB1=l3=>$Ot=√(w)^ $OLB1=l4;
     LB1=l4=>$Oz=s+t ^ $OLB1=STOP]
P2: [LB2=m1=>$Ou=x ^ $OLB2=m2;
     LB2=m2=>$O(c!u^$OLB2=STOP)]
P: ||[P1;P2]
```

In order to guarantee the post assertion of P1, to start the verification of the post assertion z=√(y)+√(w) from the end of P1 with the pre assertion y>=0, a condition to constrain the w in c?w can be generated by XYZ/VERI. After overriding the command for $LB1$=l3, this constraint is "w>=0", and then the command for $LB1$=l2 is changed into following two commands:

```
LB1=l2=>$O(c?w^LB1=NEXT);
LB1=l21^~(w>=0)=>$OLB1=l2|$OLB1=l3
```

The specification of P1, P2, and P can be shown as follows:

```
{y>=0} P1 {z=√(y)+√(w)}
{$T}   P2 {u=x}
{y>=0} P  {(z=√(y)+√(w))^(u=x)}
 WHERE w=u^w>=0
```

There are four kinds of visual tool in the XYZ system corresponding to four kinds of module. These modules are procedures, processes of flowchart form, agent and PSP. The tools are XYZ/INF (XYZ/DFD + XYZ/DDA + XYZ/PAD), XYZ/CFC, XYZ/OOA, and XYZ/DPD. Each tool can only support programming with the respective kind of module. We are now implementing a unified intelligent environment to integrate these four visual tools so as to support programming with all four kinds of modules in an appropriate and balanced way.

4 TRANSFORMING OTHER SOURCE LANGUAGES TO XYZ/E

Embedding a program written in one language into a program in another language is a common systems engineering problem. Similar problems occur with software reengineering. For XYZ/E, we have a tool called XYZ/CCSS, which deals with the problem in a simple and formal way. Attached to every syntactic production rule for a nonterminal in the source language, there is a semantic replacement rule written in the meta language XYZ/ML. The semantic rule plays both the role of the semantic definition for the nonterminal and the role as a semantic subroutine for that syntactic production rule. Each time a reduction using that production rule occurs in parsing, there is a replacement of the semantic entity of the left side of the semantic rule with its right hand side. After all replacements of the related recursively defined semantic rules of the given program have been performed, the result is a XYZ/E program into which the given program has been transformed. Before defining the semantic replacement rule, some meta notations of XYZ/ML must be explained:

"LAB" denotes a label in front of the related syntactic entity in the given program, if there is such a label, otherwise the system generates a new label for that syntactic entity. "LAB*i*" denotes a different label for each subscript *i*.

"[|*X*|]" denotes the semantics of *X*.

"<—>" denotes replacement of the semantic entity occurring on the left side with the semantic formula occurring on the right side.

"*LB*=*y*{|*A*|}$O*LB*=*z*" indicates the replacement of the labels at the entry and exit of *A* with *y* and *z* respectively. Let us see an example.

Ex. 3. The syntactic production rule and its corresponding semantic replacement rule of the nonterminal <loop st> of a source language.

```
<loop st> ::= while <cond> do <st>
[|<loop st>|]<—>LB=LAB^[|<cond>|]=>$OLB=LAB1;
                LB=LAB1{|[|st|]|}$OLB=LAB;
                LB=LAB^[|~<cond>|]=>$OLB=NEXT
```

For every source language, one must write semantic replacement rules corresponding to its syntactic production rules, XYZ/CCSS is able to process this

formal semantics text in parallel with the parser of the source language. The generated result is the XYZ/E form for the source program.

However, before using XYZ/CCSS, it is required that all context sensitive constructs in the source language that occur in the source program have previously been eliminated by a preprocessor. It is, in fact, an easy job. In our view, context sensitive constructs in source languages are always the result of some pragmatic conventions for omission or simplification of redundant information. An example in a Pascal-like language is the simplification of "*v1*:INT;*v2*:INT;*v3*:INT" into "*v1*,*v2*,*v3*:INT". A second example is nested procedures that allow identical names to denote different variables at different scopes. A third example occurs in procedure calls, where the type information of actual parameters can be omitted because they it is given for the corresponding formal parameters. These are all typical examples. Such context sensitive aspects can easily be found by careful analysis of the source language, and eliminated by renaming or adding omitted informations by using a preprocessor that performs analysis of the static semantics (e.g. with an attribute grammar). With XYZ/CCSS, we have satisfactorily transformed many international standard language such as SDL, Estelle, VHDL and Pascal into XYZ/E. Shen and Tang (1998) explain XYZ/CCSS and its application to transforming SDL into XYZ/E in more detail.

5 RELATED WORK

The only existing system implementation language with a similar design goal appears to be Unity (Chandy and Misra). Unity has also solved the difficult problem of semantic composability of parallel statements, but this language seems to be too restrictive as a system implementation language. Its formal basis is new and seems to be too complicated for a programmer to manage. XYZ/E can be used as a system implementation language and does not possess these weaknesses.

6 CONCLUSIONS

The XYZ system was a long term project in our Institute for 15 years. It was finished in 1996. Since then we have used it in research projects on system programming. One example is Abrial's Steam Boiler control system, another is a cartoon animation system and a third one is a CSCW. The first has been successfully finished (Yan and Tang, 1977a) (Yan and Tang, 1977b). The other two projects are ongoing.

7 ACKNOWLEDGEMENTS

This research was supported by the National Natural Science Foundation, the National High Technology Project (863) and the Ministry of Education of China.

In the course of preparing this paper, Messrs Zhao Chen, Lu Yi, Shen Wei, Tang Xiaoping and Shen Xiaolong have provided much assistance. Their contributions are greatly appreciated.

8 REFERENCES

Abrial, J.R., Boerger, E., and Langmaack (1996) *Formal methods for industrial applications: specifying and programming the steam boiler.* Lecture Notes in Computer Science, volume 1165, Springer-Verlag, Berlin.

Ashcroft, E. and Wadge, W.W. (1977) Lucid: a non-procedural language with iteration, Communications of ACM, **20**, 7.

Chandy, K.M. and Misra, J. (1988) *Parallel Program Design*, Addison-Wesley.

Manna, Z. and Pnueli, A. (1996) Clocked transition system, *Logic and Software Engineering* (ed. A. Pnueli and H. Lin), World Scientific Singapore.

Pnueli, A. (1996) Preface to (Manna and Pnueli, 1996). Ibid.

Shen W. and Tang, Z. (1994) The correct translation for programming languages and its application in the XYZ system, *Chinese Journal Of Advanced Software Research* **4**.

Shen W. and Tang, Z, (1998) Domain orientation based on the XYZ system, To appear in *Chinese Journal of Advanced Software Research* **1**, Allerton Press, New York.

Tang, Z. (1996) An outline of the XYZ system. In (Manna and Pnueli, 1996). Ibid.

Yan, A. and Tang, C.S. (1997a) A unified linear time temporal logic solution to the "Steam Boiler Control Specification Problem", Laboratory of Computer Science, Institute of Software, CAS. Technical report ISCAS-LCS-97-5.

Yan, A. and Tang, C.S. (1997b) Building hybrid real-time system in XYZ/E. Laboratory of Computer Science, Institute of Software, CAS. Technical report ISCAS-LCS-97-6.

Zhang, W. (1995) Verification of XYZ/SE programs, *Chinese Journal of Advanced Software Research*, **4**, Allerton Press, New York.

9 APPENDIX: A SKETCH OF TLL XYZ/E

(I) Data structures

The data structures are extensions of those in Pascal.

(II.1) Control structures: State-transition command form XYZ/BE

Executable Command

LB=*y*^R=>$O(*v1*,...,*vk*)=(*e1*,...,*ek*)^$O*LB*=*z* (1)

Here "*LB*" is a control variable referring to a label. "*y*", "*z*" are called the definitional label and the forward label, respectively; "$O(*v1*, ..., *vk*)=(*e1*, ..., *ek*)" is equivalent to "$O*v1*=*e1*^ ... ^$O*vk*=*ek*", "*v1*" ,..., "*vk*" are temporal variable, "*e1*" ,..., "*ek*" are expressions. The type of "*vi*" is identical to that of "*ei*", *i*=1,...k. "$O*vi*=*ei*" means assignment of the present time value of *ei* to the corresponding variable *vi* as its next time value. "$O" is a temporal operator for "next"; "=>" is a notation for "if ... then" (implication) on commands level, in contrast, "->" is the notation for "implication" on the logic expression level, the meaning of them are identical; R represents a FOL formula (without quantition) to serve as "condition" here. XYZ/BE commands are also called conditional element (or ce).

Specification command.

LB=*y*^R=><>(Q^*LB*=*z*) (2)

Here "<>" is a temporal operator for "eventually". (Here "<>" can be also replaced with "$O".) "Q" is a FOL formula to represent an "action" here, but Q and R together just form the pre-post specification of a procedure. A sequence of commands are connected with the connective ";" (it is a notation for "and" at the command level. In contrast, "^" is the notation for "conjunction" at the logic expression level. Their meanings are identical. That is, [c1;...;ck] following some regulations is called a unit, it forms a XYZ/BE command program. If all of these c*i*(*i*=1, ..., k) are of the form given in Eq.1, it constitutes a program effectively executable in the sense of a conventional program. (This sublanguage is also called XYZ/EE.) If all of them have the form given in Eq.2, it becomes the abstract specification of a program. (This sublanguage is also called XYZ/AE.) Otherwise, there is a mixture of both the Eq.1 and Eq.2 forms. This kind of unit represents a specification at different abstract levels.

Commands for interrupts, exception-handling and real time

LB=*y*^R =>(M)$U(N^$O*LB*=*z*) (3)

Here "$U" means "until" (it can be replaced with "$W" to mean "unless"; N and M represent a FOL and a temporal formula, respectively, "(M)$U(N^$O*LB*=*z*)" means "M stays true until N becomes true; when N becomes true, then in the next time instant *LB* refers to the label *z*". This form of command is convenient for representing interrupts, exception-handling and real time programs. The following commands are used to represent communicating concurrency.

Commands for concurrency

Parallel statement
LB=*y*^R=>||[Pr1(**par1**),...,Prk(**park**)] (4)

Select statement
LB=*y*^R=>!![C1|>D1 ,..., C*m*|>D*m*] (5)

Communication commands
LB=*y*^R=>$Oc?*x*^$O*LB*=*z* (6)
LB=*y*^R=>$Oc!*w*^$O*LB*=*z* (7)
LB=*y*^c?=>$O(Q^*LB*=*z*) (8)

Here Eq.4 is called a parallel statement, Pr*i*, *i*=1, ..., k in it are the calls of the instances of communicating processes. (The call of process or procedure would be explained below.) Eq. 5 is a select statement used to represent non-determinism, its right side formula is equivalent to the logic formula "C1^D1□...□Cm^Dm", where C*i* and D*i* are FOL formulas; In Eq. 6 - Eq. 8, "c!*y*" and "c?*x*" are predicates representing output and input commands respectively. Here "c" represents the name of the channel, *x* is the input parameter, *y* is an expression playing the role of an actual output parameter.

Besides the state transition command form sublanguage XYZ/BE, there are other two sublanguages; (a) the structured statement form sublanguage XYZ/SE and the production rule form sublanguage XYZ/PE. A special case is XYZ/PEO, which is a Prolog-like sublanguage. XYZ/SE is good for verification and XYZ/PEO is suitable for representing knowledge and prototyping.

(II.2) Control structures: Structured statement form: XYZ/SE

Conditional statement
LB=*y*^R=>OQ_1$^$O*LB*=NEXT|OQ_2$^$O*LB*=NEXT
(9)

Loop statement
*[*LB*=*y*^R=>($O*LB*=*w*|$O*LB*=EXIT); (10)
LB=*w*{|<statementseq>|}$O*LB*=*y*] (11)

Case statement
?[*LB*=*y*^R1=>$O*LB*=*z1*
|R2=>$O*LB*=*z2*
...
|R*m*=>$O*LB*=*zm*; (12)
LB=*z1*{|<statementseq>|}$O*LB*=EXIT;
LB=*z2*{|<statementseq>|}$O*LB*=EXIT;
...

LB=*zm*{|<statementseq>|}$O*LB*=EXIT]

(II.3) Control structures: Production rule form XYZ/PE

Here we only need to introduce a special case of XYZ/PE, i.e. the Prolog-like sublanguage XYZ/PE0. Its rule always has following form:

a1^...^a*n*=>a0 (13)

Here, the a*i* (*i*=0, ..., *k*) are atoms.

(III) Programs and modular structure

Program ::= %OOPROG ProgramName==
[][[SharedVarDeclParts;]
[OtherSharedItems;]
[ProcDeclPart;]
[ProcDeclPart;]
[AgentDeclPart;]
Probody]
[WherePart] (14)

The procedure and process are defined with similar structure except that procedure cannot contain declarations of processes and agents, process can not contain declaration of agents, but agent is defined with different structures. The Probody is a unit as explained above. The WherePart is always preceded by a reserved word "WHERE" and followed by a conjunction. It is used to represent a program constraint or the module or definition for some special operations or predicates used in its scope.

A procedure (or process) call is of following form:

LB=*y*^R=>P(%inp**ainp/finp**;%outp**aoutp/finp**) (15)

Eq. 15 is an abbreviation of following two ce's:

LB=*y*^R=>$O(**finp**)=(**ainp**)^$O(**foutp**)=(**aoutp**)^
$OPUSH(RST,*y*')^$O*LB*=START_P; (16)
LB=*y*'=>$O(**aoutp**)=(**foutp**)^$O*LB*=NEXT;

In the declaration of a procedure, the final forward label is "RETURN", which is a notation for POP(RST) where RST (*y*' here) is a stack to store the return labels.

An agent is defined with following structure:

AgentDeclPart ::= %AGT[Agent;...;Agent] (17)

```
Agent ::= [Process,Package]                                    (18)
Package ::= %PACK[PackageName==
                  [][[TypeDeclPart;]
                  [FatherNamePart;]
                  SharedVarDeclPart;
                  OperationDeclPart]
                  [WherePart;]]                                (19)
```

Here operations are procedures.

(IV) Real time programming

LB=y^R=>$O{l,$u$}(Q^$LB$=$z$) (20)

is defined as:

LB=y^R=>$O(DELAY(l)^$LB$=$y$')$\underline{W}$(~R^$\underline{O}LB$=$y$);
LB=y'=>(Q^LB=z)$\underline{W}$(OVER(u-l)^$\underline{O}LB$=exception$V~R^$\underline{O}LB$=$y$) (21)

Note: language level temporal operations: $O, <>, [], $U, $W;
model level temporal operations: $\underline{O}$, $\underline{<>}$, $\underline{[]}$, $\underline{U}$, $\underline{W}$;
|$O|=|($\underline{O}$)^m|=|$\underline{O}$|*m

10 BIOGRAPHY

Tang Zhisong was born in August 1925 in Hunan, China. He finished his graduate studies in Tsinghua University in 1952, specialised in logic. He has worked at the Chinese Academy of Sciences (CAS) since 1956. He is now a professor in the Institute of Software and an honorary professor at three universities. He has been a Member of CAS since 1991. His interests encompass programming languages, software engineering and formal semantics. He won the first award of the Chinese National Prize of Natural Science in 1989. He has been a member of IFIP WG 2.4 since 1980. He is the author of about 60 publications.