A PARADIGM FOR SOFTWARE MODULE SPECIFICATION WITH EXAMPLES

by

D. L. Parnas

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

March, 1971

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44610-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

ABSTRACT

This paper presents a method for writing specifications of parts of software systems. The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal. The technique is illustrated by means of a variety of examples from a tutorial system.

A PARADIGM FOR SOFTWARE MODULE SPECIFICATION WITH EXAMPLES

D. L. Parnas
Computer Science Department
Carnegie-Mellon University

Because of the growing recognition that a major contributing factor in the so-called "software engineering" problem is our lack of techniques for precisely specifying program segments without revealing too much information [1,2], I would like to report on a paradigm for module specification which has proven moderately successful in a number of test situations.

Without taking the space to justify them [see [2]] I would like to list the goals of the specification scheme to be described:

1. The specification must provide to the intended user all the information that he will need to use the program correctly, and nothing more.
2. The specification must provide to the implementer, all the information that he needs to complete the program, and no additional information; in particular, no information about the structure of the calling program should be conveyed.
3. The specification must be sufficiently formal that it can conceivably be machine tested for consistency, completeness (in the sense of defining the outcome of all possible uses) and other desirable properties of a specification. Note that we do not insist that machine testing be done, only that it could conceivably be done. By this requirement we intend to rule out all natural language specifications.
4. The specification should discuss the program in the terms normally used by user and implementor alike rather than some other area of discourse.

> By this we intend to exclude the specification of programs in terms of the mappings they provide between large input domains and large output domains or their specification in terms of mappings onto small automata, etc.

The basis of the paradigm is a view of a program module as a device with a set of switch inputs and readout indicators. The notation allows for some of the pushbuttons to be combined with indicator lights or readouts (with the result that we must push a button in order to read), but we have not yet found occasion to use that facility. The paradigm specifies the possible positions of the input switches and the effect of moving the switches on the values of the readout indicators. We insist that the values of the readout indicators be completely determined by the previous values of those indicators and the positions of the input switches. A simple extension of the notation allows the specification of mechanisms in which the values of the readout indicators are not determined by the above factors, but can be predicted only by knowing the values of certain "hidden" readout indicators which cannot actually be read by the user of the device. We have considerable doubts about the advisability of building devices which must be specified using this feature, but the ability to specify such devices is inexpensively gained.

In software terms we consider each module as providing a number of subroutines or functions which can cause changes in state, and other functions or procedures which can give to a user program the values of the variables making up that state. We refer to these all as Functions. We distinguish two classes of readout functions: the most important class provides information which cannot be determined without calling that function unless the

user maintains duplicate information in his own program's data structures. A second class, termed mapping functions, provides redundant information, in that the value of these functions is completely predictable from the current values of other readout functions. The mapping functions are provided as a notational convenience to keep the specifications and the user programs smaller.

For each function we specify:

1. the set of possible values: (integers, reals, truth values, etc.)
2. initial values: (either "undefined" or a member of the set specified in item 1. "Undefined" is considered a special value, rather than an unpredictable value.)
3. parameters: each parameter is specified as belonging to one of the sets named in item 1.

With the exception of mapping functions, almost all the information in the specification is contained in section 4. Under "effect": we place two distinct types of items which require a more detailed discussion.

First, we state that if the "effect" section is empty, then there is absolutely no way to detect that the function has been called. One may call it arbitrarily often and observe no effect other than the passage of time.

The modules that we have specified have "traps" built in. There is a sequence of statements in the "effect" section which specifies the conditions under which certain error handling routines will be called. These conditions constitute incorrect usage of the module and imply an error in the calling program. For that reason it is assumed that the error handling routine's body will not be considered part of the module specified, but

will be written by the users of the module. If the error is made, there is to be no observable result of the call of the routine except the transfer of control. When there is a sequence of error statements, the first one in the list which applies is the only one which is invoked. In some cases, the calling program will correct its error and return to have the function try again; in others, it will not. If it does return, the function is to behave as if this were the first call. There is no memory of the erroneous call.

The remaining statements are sequence independent. They can be "shuffled" without changing their meaning. These statements describe changes in the values of the other functions in the module. It is specified that no changes in functions (other than mapping functions) occur unless they are included in the effect section. The effect section can refer only to values of the function parameters and values of readout functions. The value changes of the mapping functions are not mentioned; those changes can be derived from the changes in the functions used in the definitions of the mapping functions. All of this will become much clearer as we discuss the following examples.

NOTATION

The notation is mainly Algol-like and requires little explanation. To distinguish references to the value of a function before calling the specified function from references to its value after the call, we enclose the old or previous value in single quotes (e.g. 'VAL'). Brackets ("[" and "]") are used to indicate the scope of quantifiers.

In some cases we may specify the effect of a sequence to be null. By this we imply that that sequence may be inserted in any other sequence without changing the effect of the other sequence.

Figure 1

```
PUSH(a)
value:    none
integer:    a
effect:   call ERR1 if a > p2 ∨ a < 0 ∨ 'DEPTH' = p1
      else [VAL = a;
            DEPTH = 'DEPTH'+1;]

POP
value:    none
no parameters
effect:   1.  call ERR2 if 'DEPTH' = 0
          2.  the sequence "PUSH(a); POP" has no net effect if no error
              calls occur.

VAL
value:    integer initial value undefined
no parameters
effect:   error call if 'DEPTH' = 0

DEPTH
value:    integer initial value 0
no parameters
no effect.
```

We propose that the definition of a stack shown in Figure 1 should replace the usual pictures of implementations (e.g., the array with pointer or the linked list implementations). All that you need to know about a stack in order to use it is specified above. There are countless implementations (including a large number of sensible ones). The implementation should be free to vary without changing the using programs. If the using programs assume no more about a stack than is stated above, that will be true.

Figure 2

Introduction

In the following module all function values and parameters are integers except where stated otherwise. In the interest of brevity we shall not state this repeatedly. For some values the values are not predicted by the definition. They are chosen arbitrarily by the system. This is done because the user should not make use of any regularity which might exist in the values assigned. The necessary relations between the values of those functions and the values of other functions are stated explicitly. Such incompletely defined functions are noted with an *. The user may store the values of those functions and use them to avoid repeated nested function calls.

Note: FA = father, LS = leftson, RS = rightson, SLS = set ls, SRS = set rs, SVA = set val, VAL = value, DEL = delete, ELS = exists ls, ERS = exists rs.

*FA(i)
initial value: FA(0) = 0; otherwise undefined
effect: if FA(i) undefined, then error call else none

*LS(i)
initial value: undefined
effect: error call if value is undefined

*RS(i)
initial value: undefined
effect: error call if value is undefined

SLS(i)
no value
effect: error call if FA(i) is undefined
error call if LS(i) is defined
else
LS(i) and FA(ls(i)) are given values such that
[FA(LS(i)) = i and 'FA(LS(i))' was undefined
ELS(i) = true;]

SRS(i)
no value
effect: error call if FA(i) is undefined
error call if RS(i) is defined
else
RS(i) and FA(RS(i)) are assigned values such that
[FA(RS(i)) = i and 'FA(RS(i))' was not defined
ERS(i) = true;]

SVA(i,v)
no value
effect: error call if FA(i) is undefined
else
VAL(i) = v

```
VAL(i)
  initial value:  undefined
  effect:         error call if VAL(i) is undefined

DEL(i)
  no value
  effect:         error call if FA(i) is undefined
                  error call if LS(i) or RS(i) are defined
            else
                  FA(i), VAL(i) are undefined
                       if i = 'LS('FA(i)')' then [LS('FA(i)') is undefined and
                          ELS(FA(i)) = false]
                       if i = 'RS('FA(i)')' then [RS ('FA(i)') is undefined
                          ERS(FA(i)) = false]

ELS(i)
  possible values:  true, false
  initial value:    false
  effect:           error call if FA(i) undefined

ERS(i)
  possible values:  true, false
  initial value:    false
  effect:           error call if FA(i) undefined
```

Figure 2 shows a "binary tree." This example is of interest because we have provided the user with sufficient information that he may search the tree, yet we have not defined the values of the main functions, only properties of those values. Thus, those values might well be links in a linked list implementation, array indices in a TREESORT [3] implementation or a number of other possibilities. The important fact is that if we implement the functions as defined by any method, any usage which assumes only what is specified will work.

Figure 3

Definition of a "Line Holder" Mechanism

Introduction

This definition specifies a mechanism which may be used to hold up to p1 lines, each line consisting of up to p2 words, and each word may be up to p3 characters.

```
FUNCTION WORD
          possible values:  integers
          initial values:  undefined
          parameters:  l,w,c     all integer
          effect:
                   call ERLWNL(MN) if l < 1 or l > p1
                   call ERLWNL(MN) if l > LINES
                   call ERLWNW(MN) if w < 1 or w > p2
                   call ERLWNW(MN) if w > WORDS(l)
                   call ERLWNC(MN) if c < 1 or c > p3
                   call ERLWNC(MN) if c > CHARS(l,w)

Function SETWRD
          possible values:  none
          initial values:  not applicable
          parameters:  l,w,c,d     all integers
          effect:  CSUNDO
                   call ERLSLE(MN) if l < 1 or l > p1
                   call ERLSBL(MN) if l > 'LINES' +1
                   call ERLSBL(MN) if l < 'LINES'
                   if l = 'LINES' +1 then LINES = 'LINES' + 1
                   call ERLSWE(MN) if w < 1 or w > p2
                   call ERLSBW(MN) if w > 'WORDS'(l) + 1
                   call ERLSBW(MN) if w < 'WORDS'(l)
                   if w = 'WORDS'(l) +1  then WORDS(l) = w
                   call ERLSCE(MN) if c < 1 or c > p3
                   call ERLSBC(MN) if c .noteq. 'CHARS'(l,w)+1
                   CHARS(l,w) = c
                   WORD(l,w,c) = d

Function WORDS
          possible values:  integers
          initial values:   0
          parameters:       l     an integer
          effect:
                   call ERLWSL(MN) if l < 1 or l > p1
                   call ERLWSL(MN) if l > LINES
```

```
                call ERLWSL(MN) if 1 > LINES

Function LINES
          possible values:  integers
          initial value:  0
          parameters:  none
          effect:  none

Function DELWRD
          possible values:  none
          initial values:  not applicable
          parameters:  l,w   both integers
          effect:
                  call ERLDLE(MN) if 1 < 1 or 1 > LINES
                  call ERLDWE(MN) if w < 1 or w > 'WORDS'(1)
                  call ERLDLD(MN) if 'WORDS'(1) = 1
                  WORDS(1) = 'WORDS'(1) - 1
                  for all c WORD(1,v,c) = 'WORD'(1,v+1,c) if v = w or v >
                  for all v > w or v = w  CHARS(1,v) = 'CHARS'(1,v+1)

Function DELINE
          possible values:  none
          initial values:  not applicable
          parameters:  1    an integer
          effect:
                  call ERLDLL(MN) if 1 < 0 or 1 > 'LINES'
                  LINES = 'LINES' - 1
                  if r = 1 or r > 1 then for all w, for all c
                                    ( WORDS(r) = 'WORDS'(r+1)
                                      CHARS(r,w) = 'CHARS'(r+1,w)
                                      WORD(r,w,c) = 'WORD'(r+1,w,c)  )

Function CHARS
          possible values:  integer
          initial value:  0
          parameters  1, w     both integers
          effect:
                  call ERLCNL(MN) if 1 < 1 or 1 > LINES
                  call ERLCNW(MN) if w < 1 or w > WORDS(1)
```

Figure 3 shows a more specialized piece of software. It is a storage module intended for use in such applications as producing KWIC indices. It is designed to hold "lines" which are ordered sets of "words," which are ordered sets of characters, to be dealt with by an integer representation. For this example there are some restrictions on the way that material may be inserted (only at the end of the last line) which reflect the intended use. That might well be a design error, but for our purposes the important thing to note is that the restrictions are completely and precisely specified without revealing any of the internal reasons for making such restrictions.

Figure 4

SYMBOL TABLE DEFINITION

p1=maximum number of symbols

p2=maximum number of characters per symbol

p3=maximum value of character

```
STRTSM
      possible values:  none
      initial values:   not applicable
      parameters:       none
      effects:          call ERFAST if 'MAYIN'=true
                        MAYIN=true

MAYIN
      possible values:  true, false
      initial values:   false
      parameters:       none
      effects:          none

CHARIN
      possible values:  none
      initial values:   not applicable
      parameters:       call ERCHIL if c < 0 or c > p3
                        call ERMNIN if 'MAYIN'=false
                        call ERBUFX if 'BUFFERCNT'=p2
                        BUFFER('BUFFERCNT'+1)=c
                        BUFFERCNT='BUFFERCNT'+1

BUFFER
      possible values:  integers 0 < BUFFER ≤ p3
      initial values:   not applicable
      parameters:       c, an integer
      effects:          call ERBUFE if c < 1 or c >'BUFFERCNT'

BUFCNT
      possible values:  integers 0 < BUFFERCNT ≤ p2
      initial values:   0
      parameters:       none
      effects:          none

SYMEND
      possible values:  integers 0 < SYMEND ≤ 'SMCNT')+1
      initial values:   not applicable
      parameters:       none
      effects:          call ERNOIN if 'MAYIN'=false
                        MAYIN=false
                        if there is an s (0 < s ≤ 'SMCNT') such that
                        'BUFFERCNT'='CHCNT(s)' and
                           [if for all c (0 < c < 'BUFFERCNT')
                            BUFFER(c)='CHAR(s,c)' ]then
                                SYMEND=s
                        else [ERSYL if 'SMCNT'=p1
                            for all c (0 < c < 'BUFFERCNT')
                                [CHAR('SMCNT'+1,c)='BUFFER'(c)
                                CHCNT('SMCNT'+1)='BUFFERCNT' ]
                                SMCNT='SMCNT'+1 ]
                         BUFFERCNT=0
```

CHAR

possible values:	integers 0 < CHAR ≤ p3
initial values:	not applicable
parameters:	s and c, both integers
effects:	call ERNOSY if a < 1 or s >'SMCNT'
	call ERNOCH if c < 1 or c >'CHCNT'(s)

SMCNT

possible values:	integers 0 ≤ SMCNT ≤ p1
initial values:	0
parameters:	none
effects:	none

CHCNT

possible values:	integers 0 < CHCNT < p2
initial values:	not applicable
parameters:	s, an integer
effects:	call ERNOSY if s < 1 or s >'SMCNT'

In making the line holder of Figure 3 it will probably prove advantageous to (1) separate out the problem of storing the individual characters that make up a word from the problem of storing the makeup of lines out of words, and (2) avoid duplicate storing of identical words. Both can be accomplished by use of the mechanism defined in Figure 4 as a submodule for that described in Figure 3. The implementor of the "line holder" would pass the individual characters of the "words" to the symbol table whose definition guarantees him that he will receive a unique encoding of every symbol. Note that the specification in Figure 4 does not rule out an implementation which stores duplicate copies of words, but it does require that all receive the same encoding.

It is important to note that the user of the "line holder" will never know or need to know of the existence of the symbol table inner mechanism.

Figure 5

Alphabetizer for Line Holder

This module accomplishes the alphabetization of the contents of the modules referred to above by producing a pointer function, ITH, which gives the index of the ith line in the alphabetized sequence.

```
Function ITH:
        possible values:  integers
        initial values:  undefined
        parameters:  i  an integer
        effect:
                call ERAIND if value of function undefined for parameter given

Mapping Function ALPHC:
        possible values:  integers
         values:  ALPHC(l) = index of l in alphabet used
                        ALPHC (l)   infinite if character not in alphabet
        parameter:  l  an integer
        effect:
                call ERAABL if l not in alphabet being used

Mapping Function EQW:
        possible values:  true,false
        parameters:  l1,l2,w1,w2     all integers
         values:
                EQW(l1,w1,l2,w2)=for all c('CSWORD'(l1,w1,c)='CSWORD'(l2,w2,c))
        effect:
                call ERAEBL  if l1 < 1 or l1 > 'CSLINES'
                call ERAEBL  if l2 < 1 or l2 > 'CSLINES'
                call ERAEBW  if w1 < 1 or w1 > 'CSWORDS'(l1)
                call ERAEBW  if w2 < 1 or w2 > 'CSWORDS'(l2)
```

```
* Mapping Function ALPHW:
        possible values:  true,false
        parameters:  l1, 12,w1,w2    all integers
        values:
              ALPHW(l1,w1,12,w2) = if .not. 'EQW'(l1,w1,12,w2)  and
               k = min c such that ('CSWORD'(l1,w1,c).noteq.'CSWORD'(12,w2,c))
               then 'ALPHC'('CSWORD'(l1,w1,c))<'ALPHC'('CSWORD'(12,w2,c))
              else false
        effect:
              call ERAWBL if l1 < 1 or l1 > 'CSLINES'
              call ERAWBL if 12 < 1 or 12 > 'CSLINES'
              call ERAWBW if w1 < 1 or w1 > 'CSWORDS'(l1)
              call ERAWBW if w2 < 1 or w2 > 'CSWORDS'(12)

Mapping Function EQL:
        possible values, true, false
        parameters:  l1,12   both integers
        values:
             EQL (l1,12) = for all k ('EQW'(l1,k,12,k))
        effect:
              call ERALEL if l1 < 1 or 1  > 'CSLINES'
              call ERALEL if 12 < 1 or 12 > 'CSLINES'

Mapping Function ALPHL:
        possible values:  true,false
        parameters:  l1,12   both integers
         value:
              ALPHL(l1,12) = if .not.'EQL'(l1,12) then
              (let k = min c such that .not. 'EQW'(l1,k,12,k))
               'ALPHW'(l1,k,12,k)  else  true
        effect:
              call ERAALB if l1 < 1 or l1 > 'CSLINES'
              call ERAALB if 12 < 1 or 12 > 'GSLINES'

Function  ALPH:
        possible values:  none
        initial values:  not applicable
        effect:
              for all  i .not<. 1 and i.not>.'GSLINES' (
              ITH (i) is given values such that(
                  for all j .not<.1 and .not>. CSLINES
                    there exists a k such that ITH(k) = j
              for i >-1 and < 'CSLINES'    (that'ALPHL'(ITH(i), ITH(i+1)))
```

Figure 5 is intended to exhibit the situations in which mapping functions are useful in specifications. This module is an alphabetizer, intended to work with the "line holder" shown earlier. It determines values for ITH in such a way that (1) every integer between 1 and the number of lines is a value of ITH and if $i < j$ then the line numbered ITH(i) does not come before the line numbered ITH(j) in the alphabetic ordering.

Note that ITH as defined might be an array in which the values specified are stored by the routine ALPH, or it might be a routine which searches for the appropriate line each time called. An interesting alternative would be to make use of FIND [4] within ITH so that the computation is distributed over the calls of ITH and so that in some situations unneeded work may be avoided. We repeat that the important feature of this specification is that it provides sufficient information to use a module which is correctly implemented according to any of these methods, without the user having any knowledge of the method.

Using the Specifications

The specifications will be of maximum usefulness only if we adapt our methods to make use of them. Our aim has been to produce specifications which are in a real sense just as testable as programs. We will gain the most in our system building abilities if we have a paradigm for usage of the specifications which involves testing the specifications long before the programs specified are produced. The statements being made at this level are precise enough that we should not have to wait for a lower level representation in order to find the errors.

Such specifications are at least as demanding of precision as are programs; they may well be as complex as some programs. Thus they are as likely to be in error. Because specifications cannot be "run," we may be tempted to postpone their testing until we have programs and can run them. For many reasons such an approach is wrong.

We are able to test such specifications because they provide us with a set of axioms for a formal deductive scheme. As a result, we may be able to prove certain "theorems" about our specifications. Example "theorems" might be:

1. The specification never refers to F1(p) unless it is certain that p is less than 9.
2. Whenever F3(x) is true F4(x) is defined and conversely.
3. It is not possible for F5(x) to take on values greater than p3.
4. Error routine ERRX will never be called.
5. There exists a sequence of function calls which will set F2(x) = F5(x) = 0.
6. There will never exist distinct integers i and j such that F1(i) = F2(j).

By asking the proper set of such questions, the "correctness" of a set of specifications may be verified. The choice of the questions, hence the meaning of "correctness" is dependent on the nature of the object being specified.

Using the same approach of taking the specifications as axioms and attempting to prove theorems, one may ask questions about possible changes in system structure. For example, one may ask which modules will have to be changed, if certain restrictions heretofore assumed were removed.

It would be obviously useful if there were a support system which would input the specifications and provide question answering or theorem proving ability above the specifications. That, however, is not essential. What is essential is that system builders develop the habit of verifying the specifications whether by machine or by hand before building and debugging the programs.

Incidentally, the theorem proving approach might also be considered as a basis for a program which searches automatically for implementations of a specified module.

Hesitations

To date the paradigm has received only limited evaluation. It has been used with reasonable success in the construction of small systems with simple modules in an undergraduate class. Currently nearing completion is a description of a simplified man/machine interface for a graphics based editor system. However, any attempt to use this on a larger project (where the probability of failure without the technique is high) is in a very early stage. Clearly the idea needs further practical use before its usefulness can be evaluated. I hope that some of my readers will be in a position to do this.

There appears to be a weak limitation on the technique in that it makes it easy to describe objects which receive data in small units, and where the calling program must be aware of the period between receipt of such small units. As of yet we have not found a way to follow the paradigm for such objects as a compiler where the user sends one very large unit and does not want to know of internal steps in the processing of individual characters, phrases, etc. For such situations we have been

forced to make use of techniques similar to that of Wirth and Weber [5]. We did, however, combine the two techniques with some success.

In usage of these techniques it has become clear that there is a great initial resistence to their use. This approach to the description of programs as somewhat static objects, rather than sequential decision makers, is unfamiliar to men with lots of programming experience. The first few attempts always fail and require the patient guidance of an instructor. The idea is, however, simple and is eventually mastered by almost everyone.

References

[1] Buxton and Randell (eds.), Software Engineering Methods. Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 October 1969.

[2] Parnas, D. L. "Information Distribution Aspects of Design Methodology." Technical Report, Department of Computer Science, Carnegie-Mellon University, February, 1971. To be presented at the IFIP Congress, 1971, Ljubljana, Yugoslavia, and will be included in the proceedings.

[3] Floyd, R. W. "Treesort 3" Algorithm 245. Comm. ACM, December, 1964.

[4] Hoare, C. A. R. "Proof of a Program, FIND." Comm. ACM, January, 1971.

[5] Wirth, N. and H. Weber. "Euler: A Generalization of ALGOL and its Formal Definition." Comm. ACM, pp. 13-23, January, 1966.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Department of Computer Science Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

A Paradigm for Software Module Specification with Examples

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

D. L. Parnas

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
March, 1971	22	5

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-70-C-0107	
b. PROJECT NO.	
A0827-5	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
61101D	
d.	

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	Air Force Office of Scientific Research 1400 Wilson Boulevard (SRMA) Arlington, Virginia 22209

13. ABSTRACT

This paper presents a method for writing specifications of parts of software systems. The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal. The technique is illustrated by means of a variety of examples from a tutorial system.

DD FORM 1 NOV 68 1473

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT