ICOT Technical Report: TR-068

TR-068

FORMAL SPECIFICATION AND VERIFICATION FOR CONCURRENT SYSTEMS BY TELL

by
Hajime Enomoto, Naoki Yonezaki,
Motoshi Saeki, and Hiroshi Aramata
(Tokyo Institute of Technology)

June, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

FORMAL SPECIFICATION AND VERIFICATION FOR CONCURRENT SYSTEMS BY TELL

Hajime Enomoto, Naoki Yonezaki, Motoshi Saeki, and Hiroshi Aramata

Department of Computer Science, Tokyo Institute of Technology,
2-12-1 Ookayama, Meguro-ku, Tokyo 152, Japan

Formal specification and verification of concurrent systems with layered architecture by software development system 'TELL' is presented. Tell/NSL is a specification language which is a fragment of English. Using Tell/NSL, we specify the first layer communication protocol, which tranceives bit sequence synchronized with a clock. This protocol is considered as an implementation of service, which is used by the next upper layer protocol (in our case, alternating bit protocol). Interface specification between them is also described in Tell/NSL. Specifications are translated into temporal logical formulas using semantic rules associated with syntax rules of Tell/NSL. Furthermore, we show the example of verification showing that alternating bit protocol is implemented by our first layer protocol.

INTRODUCTION

To verify various properties of concurrent systems, which are executed in parallel, e.g. partial correctness, dead lock freedom, reachability, consistency, etc., we must specify not only their input-output relations but also their execution sequence formally. It is also required that specifications of their execution sequence are comprehensive for efficient proofs of their properties.

Many researchers have studied specification and verification technique for concurrent systems (5)~(10). Finite state machines such as state transition diagram are generally used to specify concurrent systems, but has a several shortcomings such as state explosion which makes the analysis of the system behavior somewhat difficult, or not supporting sufficient techniques for hierarchical decomposition of the systems based on abstract level which playes an important role on the construction of comprehensive specifications and verifications of huge systems (1)~(3).

Usually software systems are designed as series of layers in the form of hierarchical support for the reason of facility to change them and to use them for general purpose. Layered architecture systems need interface specifications between adjacent layers, but no specification languages ever developed has ability to support it.

Tell/NSL is a specification language whose semantical basis is temporal logic. Temporal logic is suitable for formal specification and reasoning about the execution sequence of a system (5)~(9), but temporal logic formulas themselves as specification language have no hierarchical decomposition nor abstraction mechanism. Furthermore, temporal logic is not comprehensive except for trained persons. In Tell/NSL, hierarchical decomposition is supported based on abstraction by lexical decomposition method. In this paper, we focus on the

specification and verification method of concurrent systems with layered architecture using Tell/NSL. First, we discuss the specification method of layered architecture system by Tell/NSL using the example of simple communication protocols. Next section presets formal semantics of Tell/NSL, i.e. temporal logic. Finally, we show the example of a verification of the system specified by Tell/NSL.

SPECIFICATION OF CONCURRENT SYSTEMS WITH LAYERED ARCHITECTURE BY TELL/NSL

Tell/NSL is a specification language based on unambiguous natural language (a fragment of English). The specification technique based on Tell/NSL is described in detail in (12),(13). In Tell/NSL, specifications are written hierarchically in the form of defining the meanings of words used in specification sentences. They are translated into logical formulas by the method based on Montague's (8). English sentences in specifications are simple declarative sentences, in which relative pronoun clause and negative are also available.

To implement concurrent systems from their specifications, we usually set up several intermediate layers between abstract specifications and concrete implementations, and then gradually refine the specifications to the implementations according to discipline of each layer. A system in each layer provides services to the next upper layer and is supported by the next lower layer. In this sense, layered structure is a hierarchical structure about implementations of the system. On account of independency among layers, layered architecture prevents propagation of changes when the change is made in specific part. This is one of reasons why many software systems are designed as a series of layers.

Communication protocols have layered architecture and their hierarchical structures have been standardized such as OSI Reference Model of ISO (11). AB-protocol in (10),(13) belongs to the second layer - data link layer in OSI model. To clarify the concept of layered architecture and its specification technique using protocol examples, we will introduce a conceptual model of communication systems shown in Fig.1. A protocol machine consists of two virtual terminals which are communication entities, and a virtual transmission line with which the terminals are connected. On the model, protocol specifications describe interactions between terminals through the transmission line by specifying interaction sequences consisting of r_1, s_1, r_2, s_2. Service specifications specify interactions between users and the protocol machine, S_1, R_1, S_2, R_2, i.e. input-output actions of the machine (10).

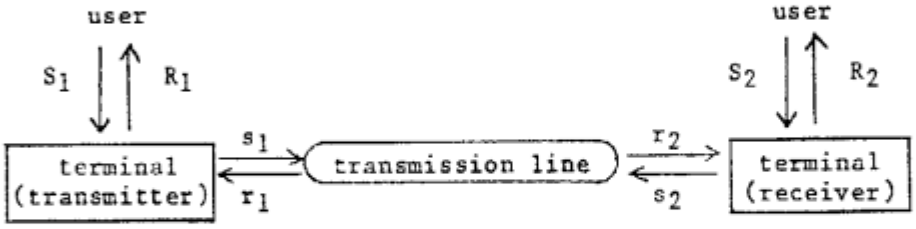

Fig.1 Conceptual model of communication systems

Thus we consider such interactions as processes which communicates concurrently with each other and a transmission line as shared resources used by the communication processes.

We show an example of simple protocol, which corresponds to the first layer protocol, i.e. the next lower layer to AB protocol, and specify it by Tell/NSL based on the model in Fig.1. We call it hardware level protocol. Hardware level

protocol and AB protocol is useful to clarify the concept of layered architecture. The specifications of AB protocol used in this paper is described in (13).

Hardware level protocol tranceives a bit sequence synchronized with a clock. Fig.2 shows the block diagram of it. The scenario of transmitting a bit sequence is as follows. When the line is empty, it is forced to high level 1. Whenever transmitter transmits a bit sequence, it adds a start bit 0 followed by user's data bits (rightmost bit first), and stop bit sequence (01111) to the data. This bit sequence is successively shifted to right and transmitted one by one bit synchronized with a transmitter clock (Sclock). Receiver can detect a start bit by observing whether the line changes high to low. For the purpose of preventing four consecutive 1's in user data from interfering with receiver's detecting the stop bit sequence, transmitter stuffs a 0 into it whenever it has continuously transmitted 1's for three times. When receiver detects a start bit, it start a receiver clock (Rclock) whose cycle is the same as Sclock's. Then it receives one by one bit from the line synchronized with Rclock until it detects stop bits. It ignores stuffed 0's during receiving.

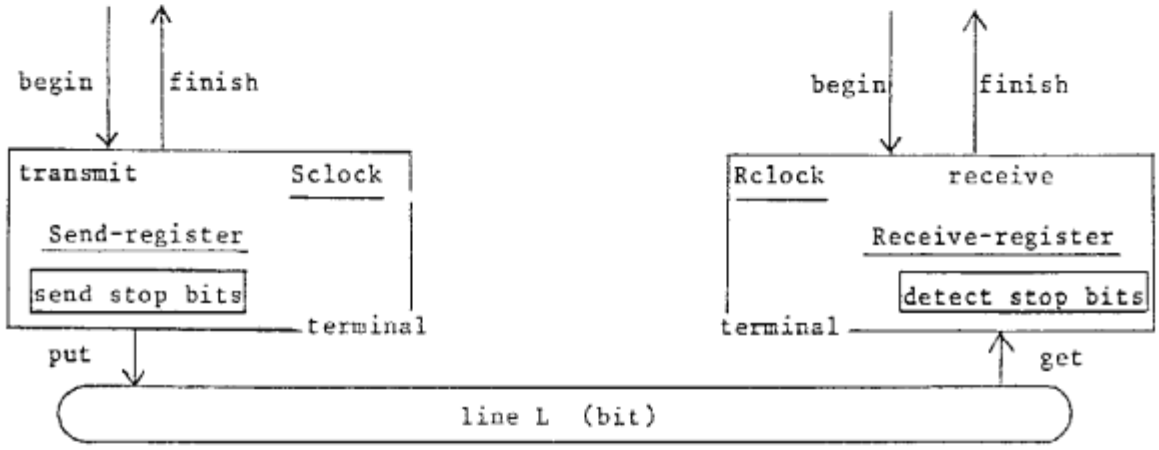

Fig.2 Block diagram of hardware level protocol

Fig.3 shows the specification of hardware level protocol using Tell/NSL. 'Send-register' and 'Receive-register' are shift registers of parallel-in-out/serial-in-out.

The conceptual model shown in Fig.1 is a model schema in each layer. In the figure, virtual users in a layer correspond to the terminals in the next upper layer. In the case of protocol, Fig.4 shows the relationship between adjacent layers, N-1th layer and Nth layer. In the figure, Nth line in Nth protocol machine is implemented by N-1th protocol machine which belongs to the next lower layer. In the layer model of communication protocols, the operations of a virtual transmission line are supported by the services of the next lower layer.

When we design layered architecture systems, we must specify not only specifications such as Fig.3 but also interfaces between adjacent layers. Interface specification describes how and which parts of the layer are supported by the next lower layer. In Tell/NSL, it is specified by letting sentences in the upper layer correspond to those in the lower layer which are semantically equivalent to it.

Fig.5 shows the interface specification between AB protocol as the second layer and hardware level protocol as the first layer. The first sentence in the figure describes that words associated with 'line' in AB protocol machine are decomposed into words in hardware level protocol machine, i.e. 'line' is implemented by

```
Hardware level protocol machine
                is the system such that
 Configuration
    1) There is line L.
    2) There is register Send-register.
    3) There is register Receive-register.
 Timing
  1) Initially L is 1.
  2) Initially it begins to receive.
  3) Initially it begins to transmit.
  4) Initially Send-register is cleared.
  5) Initially Receive-register is cleared.

  It transmit sequence of bit t means that
   configuration
    1) There is clock of baud-rate cycles Sclock.
   timing
    1) Initially it waits to transmit
         until Send-register is not empty.
    2) If it waits to transmit
       and Send-register is not empty,
         then it generate Sclock.
    3) If it finishes generating Sclock,
         then it is ready to transmit
              until Sclock is send-timing.
    4) If it is ready to transmit
       and Sclock is send-timing
         then it begins to put a start bit to L.
    5) If it finishes putting to L and
         it is not continuously transmitting 1
                    at three times and
      5-1) Send-register is not empty,
             then it begins to serial-output
                  from Send-register.
      5-2) Send-register is empty,
             then it begins to send stop bits.
    6) If it finishes serial-outputting bit b
         from Send-register,
       then it finishes serial-outputting bit b
            from Send-register
          until Sclock is send-timing.
    7) If it finishes serial outputting bit b
            from Send-register and
       it is not continuously transmitting 1
            at three times and
       Sclock is send-timing,
       then it begins to put bit b to L.
    8) If it finishes putting to L and
       it is continuously transmitting 1
          at three times,
       then it is ready to insert bit 0
            until Sclock is send-timing.
    9) If it is ready to insert bit 0
       and Sclock is send-timing,
         then it begins to put 0 to L.
   10) If it finishes sending stop bits,
         then it finishes transmitting.
   11) If it finishes transmitting,
         then (it waits to transmit
              until Send-register is not empty)
           in the next time.

  It send stop bits means that
    1) Initially it is ready to send stop bits
                until Sclock is send-timing.
    2) If it is ready to send stop bits and
       Sclock is send-timing,
         then it begins to put 0 to L.
    3) If it is not continuously transmitting 1
       at four times and Sclock is send-timing,
         then it begins to put bit 1 to L.
    4) If it is continuously transmitting 1
       at four times and Sclock is send-timing,
         then it finishes sending stop bits.
  end send stop bits;

  It is continuously transmitting 1
       at i times  means that
    1) It finishes putting 1 to L at i times
         since it finishes putting 0 to L.
  end continuously transmitting;

  Clock c is send-timing means that
    1) c is 1.
  end send-timing;
end transmit;

It receives sequence of bit t means that
 configuration
  1) There is
       clock of baud-rate cycles Rclock.
 state phrase
  1) ready to finish [adj].
 Timing
  1) Initially it waits to receive until
            a start bit is got from L.
  2) If it waits to receive and
     a start bit is got from L,
       then it begins to generate Rclock.
  3) If it finishes generating Rclock,
       then  it is ready to receive
         until Rclock is receive-timing.
  4) If it is ready to receive and
     bit b is synchronizingly got from L and
     it is not detecting stop bits
   4-1) it is continuously receiving 1
          at three times,
        then it stops receiving until
             Rclock is receive-timing again.
   4-2) it is not continuously receiving 1
          at three times,
        then it begins to serial-input b
                    to Receive-register.
  5) If it finishes serial-inputting
          to Receive-register
       and it is not detecting stop bits,
     then it is ready to receive
          until Rclock is receive-timing.
  6) If it stops receiving
       and Rclock is receive-timing,
     then it is ready to receive until
          Rclock is receive-timing again.
  7) If it is detecting stop bits,
       then it is ready to finish until
            it is continuously receiving 1
                  at four times.
  8) If it is ready to finish and
     it is continuously receiving 1
          at four times,
       then it finishes receiving.
  9) If it finishes receiving,
       then (it waits to receive
            until a start bit is got from L)
            in the next time.

  It is detecting stop bits means that
    1) 0 is synchronizingly got from L.
    2) ((Rclock is receive-timing
           then 1 is got from L)
         until Rclock is receive-timing
           at four times)
       in the next time.
   end detecting;

  Clock c is receive-timing means that
    1) c is 1.
  end receive-timing;

  It is continuously receiving 1 at i times
     means that
       1) 1 is synchronizingly got from L
            at i times since
            0 is synchronizingly got from L.
 end continuously receiving;
```

Fig.3 Specification of hardware level protocol

```
  lexicon
    1) Bit b is
         synchronizingly got from line L
       := Rclock is receive-timing
          and b is got from L.
end receive;

Line associated with put and got
     is implemented by bit.
  construction
    1) It puts bit b to line L
       := the result of putting b to L
            is a line.
  satisfy
    1) / put[L,b]=b /.
  Timing
    1) If it begins to put bit b,
       then it finishes putting bit b
            in the next time.

  Bit b is got from line L means that
    1) b is L.
  end got;

end line;

Bit b is a start bit means that
   1) b is 0.
end start bit;

Clock of i cycles associated with generate
  is implemented by integer.
  construction
    1)It generates clock
      := the result of generating
           is a clock.
    2)It advances clock
      := the result of advancing is a clock.
  satisfy
    1) / generate[]=0 /.
    2) / advance[x]=mod[x+1,i] /.
  Timing
    1) If it begins to generate clock,
       then (it finishes generating clock and
            it begins to advance clock)
              in the next time.
    2) If it begins to advance clock,
       then it finishes advancing clock
            in the next time.
    3) If it finishes advancing clock
       and it doesn't begin to generate clock,
       then (it begins to advance clock) until
            (it begins to generate clock
              in the next time).
end clock;
```

```
Register
   associated with clear, preset,
        parallel-load, serial-input,
        and serial-output
   is implemented by sequence of bit
  construction
    1) It clears register R
       := the result of clearing
            is a register.
    2) It presets sequence of bit x
         to register R
       := the result of presetting x to R
            is a register.
    3) It parallel-loads sequence of bit x
         from register R
       := the result of parallel-loading from R
            is a sequence of bit and a register.
    4) It serial-inputs bit b to register R
       := the result of serial-inputting b
         to R is a register.
    5) It serial-outputs bit b from register R
       := the result of serial-outputting
         from R is a bit and a register.

  satisfy
    1) The result of clearing
         is empty sequence.
    2) The result of presetting x to R is x.
    3) The result of parallel-loading from R
         is R and empty sequence.
    4) The result of serial-inputting b to R
         is the concatenation of b and R.
    5) The result of serial-outputting from R
         is the head of R and the tail of R.

  timing
    1) If it begins to clear,
       then it finishes clearing
            in the next time.
    2) If it begins to preset,
       then it finishes presetting
            in the next time.
    3) If it begins to parallel-load,
       then it finishes parallel-loading
            in the next time.
    4) If it begins to serial-input,
       then it finishes serial-inputting
            in the next time.
    5) If it begins to serial-output,
       then it finishes serial-outputting
            in the next time.
  end register
  lexicon
      1) Register r is empty := r is cleared.
end hardware level protocol machine
```

Fig. 3 Specification of hardware level protocol (continued)

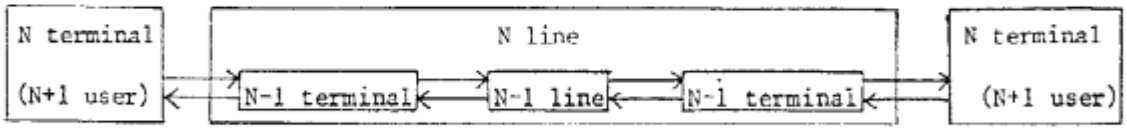

Fig.4 Relationship between layers

hardware level protocol machine. The next itemized sentences express the semantically equivalent relationship between words in AB protocol and those in hardware level protocol. For example, sentence 'l is empty' is semantically equivalent to 'register 1 is cleared', in which words only in the lower layer are used. Furthermore action definition 'pass down', which corresponds to 'send' in AB-protocol machine, is lexically decomposed into 'preset' and 'transmit' in hardware level protocol machine. We consider that the definition of 'pass down' corresponds to the implementation of the operation which is represented by 'send'. This implementation is composed of several modules in the lower layer.

```
Line in AB-protocol machine is implemented
  by Hardware level protocol machine
  means that
    1) l is empty
       := register 1 is cleared.
    2) It sends message m
       := it passes down message m.
    3) It reads message m
       := it passes up message m.
    4) l1 is the result of sending message m to l2
       := register l1 is the result of
            presetting m to register l2.
    5) m and l1 is the result of reading from l2
       := m and register l1
            is the result of parallel-loading
             from register l2.
    6) line := Receive-register.

  It passes down sequence of bit m
   means that
    1) Initially it begins to preset m
       to Send-register and it is passing down.
    2) If it finishes
            presetting to Send-register,
         then it is ready to finish passing down
              until it finishes transmitting.
    3) If it is ready to finish passing down
         and it finshes transmitting,
          then it finishes passing down
               in the next time.
  end pass down;

  It passes up sequence of bit m
   means that
    1) Initially it waits to pass up
       until Receive-register is not empty.
    2) If it waits to pass up and
            Receive-register is not empty,
         then it executes passing up
              until it waits to receive.
    3) If it executes passing up and
         it waits to receive
         then it is passing up.
    4) If it is passing up,
         then it begins to parallel-load
                from Receive-register.
    5) If it finishes parallel-loading m
                  from Receive-register,
         then it finishes passing up m.
  end pass up;

  lexicon
    1) Register r is empty := r is cleared.
```

Fig.5 Interface specification

TRANSLATION INTO TEMPORAL LOGIC

In this section we explain the mechanism translating specification sentences into formulas of modal logic. Translation rules are automatically generated from definitions of words used in the specification or associated with syntax rules of English. This translation method is based on that of Montague grammar(4). However, our method has some differences from Montague's as follows.

1) Types of logical expressions into which words are translated are not uniquely determind by syntactic categories of the words. Generally, a type of a logical expression is decided by a class definition or functionality of operations associated with a word. Although, syntactic categories of words representing classes are fixed into one, the logical types of the translated logical terms varies and the types are specified as type schematas using generic type α in the semantic rules.
2) Prepositions are introduced syncategorematically, in brief they have no translation, but control of matching with formal parameters and actual parameters.
3) Every sentence in natural language is always translated into only one logical formula i.e. there is no ambiguity.

The logic we use is a first order many sorted temporal logic with temporal operators $\Diamond$, $\circ$ and until. In our model of the temporal logic, we use a linearly ordered set of time. Intuitively speaking, $\Diamond A$ is true, if there is a time when A is true in the future. $\circ A$ is true, if A is true in the next time. A until B is true, if A is true until B becomes true. These operaters are translations of

auxiliary verb 'will', adverb phrase 'in the next time', and connective 'until' respectively. We will use symbol '□' as an abbreviation for '~◇~'.

A set of English sentences available in our specifications is very restricted. A part of their syntactic rules and semantic rules (translation rules into logical expressions) are shown in Appendix. If there are more than two associated classes, they are bracketted with '[' and ']'. For instance, <adjective>$[\alpha_1,\ldots,\alpha_n]$ represents that the number of the arguments of the defined word is n and that a class of each argument corresponds to a type α_i ($1 \le i \le n$) in this logic. Fig.6 shows the example translation of the third sentence in the specification of 'receive' shown in Fig.3 into a logical formula. It should be noted that conjugations in the sentene in Fig.6 have been restored to original forms. The numbers associated with nodes of the tree are rule numbers - listed in the appendix.

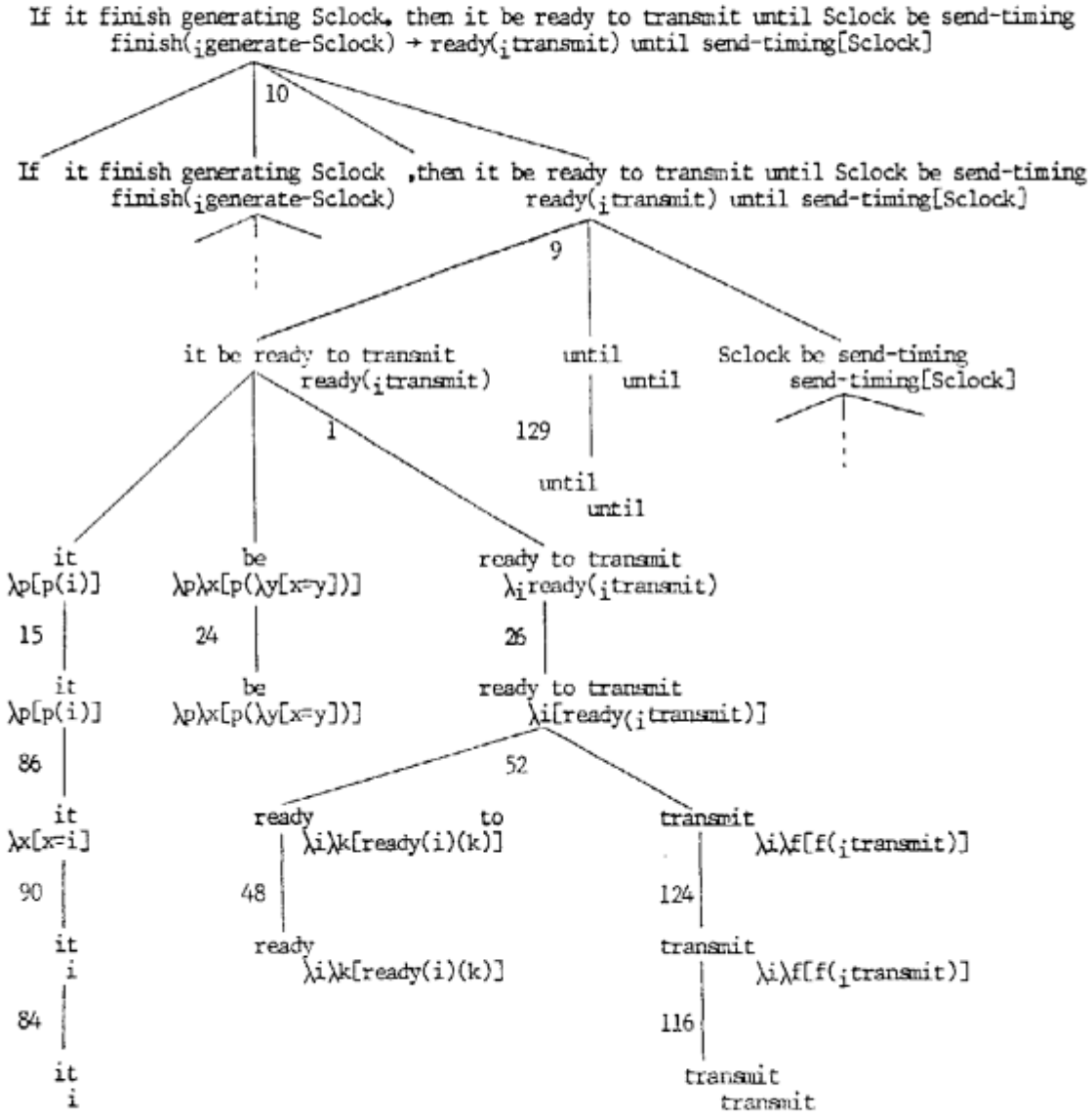

Fig.6 Derivation tree

As to dynamic class definition 'register' of hardware level protocol in Fig.3, we have the following set of translations. In Fig.7, it should be noted that index i is omitted. State predicates, which are the translation of state phrases, 'begin' and 'finish' represent the execution states of the operations whose name are their arguments. The 6)~10) sentences are generated automatically from a construction declaration, in such a way that a constructor name, its argument list, and dynamic class name are substituted in a schema of temporal logic. For instance, if a constructor's functionality is Dynamic class × In-type$_1$ × ... × In-type$_n$ → Dynamic class × Out-type$_1$ × ... × Out-type$_m$, then the schema is of the form :

$$_i\text{begin}(\underline{op}\text{-}) \wedge \underline{Dc} = x_0 \wedge {}_i\underline{op}\text{-arg1} = x_1 \wedge \ldots \wedge {}_i\underline{op}\text{-argn} = x_n$$
$$\rightarrow \Diamond\circ[\text{finish}({}_i\underline{op}) \wedge \underline{Dc} = \text{arg1}({}_i\underline{op}(x_0,\ldots,x_n))$$
$$\wedge {}_i\underline{op}\text{-argn+1} = \text{arg2}({}_i\underline{op}(x_0,\ldots,x_n))$$
$$\wedge\ldots\wedge {}_i\underline{op}\text{-argn+m} = \text{argm}({}_i\underline{op}(x_0,\ldots,x_n))]$$

An instance of the schema for a constructor is generated by replacing underlined parts of the schema, that is $\underline{op}$ is replaced by a constructor name and $\underline{Dc}$ is replaced by a instance name. State predicate 'finish' represents the termination state of the action and 'begin' stands for the state just before the termination. They are translations of verbs 'finish' and 'begin' respectively. If a sentence in present progressive form appears in the specification, 'begin' does not correspond to the state just before termination but the present participle of the verb does. It should be noted that the method translating a static input-output property description into formulas which express dynamic properties of the operator is properly new idea and it provides minimality of specification.

register
1) empty-sequence[clear[R]]
2) preset[x,R]=x
3) parallel-load[R]=<R,empty-sequence>
4) serial-input[b,R]=concat[b,R]
5) serial-output[R]=<head[R],tail[R]>
6) begin(clear)∧register=r
→ ○◇(finish(clear)∧register=clear[r])
7) begin(preset)∧preset-arg=x∧register=r
→ ○◇(finish(preset)∧register=preset[x,r])
8) begin(parallel-load)∧register=r
→ ○◇(finish(parallel-load)
∧parallel-load-arg=arg1[parallel-load[r]]
∧register=arg2[parallel-load[r]])
9) begin(serial-input)∧serial-input-arg=b
∧register=r
→ ○◇(finish(serial-input)
∧register=serial-input[b,r])
10) begin(serial-output)∧register=r
→ ○◇(finish(serial-output)
∧serial-output-arg=arg1[serial-output[r]]
∧register=arg2[serial-output[r]])
timing
1) begin(clear) → ◇finish(clear)
2) begin(preset) → ◇finish(preset)
3) begin(parallel-load) → ◇finish(parallel-load)
4) begin(serial-input) → ◇finish(serial-input)
5) begin(serial-output) → ◇finish(serial-output)

Fig.7 Translation of register in hardware level protocol

We will introduce additional temporal advebs and conjunctives. They are very useful to write complex specifications in simple and comprehensive natural language sentences. Their meanings are provided by macro temporal operators or functions, which are defined in the recursive form as follows.

[until-$_i$]
A until-$_1$ B = A until B
A until-$_{i+1}$ B = A until ((B∧~○B)∧○(A until-$_i$ B)) where i≥1
[times]
(~A∧○A∧○B) → ○(times[A,B]=1)
(~A∧○A∧~○B) → ○(times[A,B]=0)
(~B∧○B) → (times[A,B]=x↔○(times[A,B]=x+1)
~○B → (times[A,B]=x↔○(times[A,B]=x))
(B∧○B) → (times[A,B]=x↔○(times[A,B]=x))

Temporal operators 'until-$_i$' and function 'times' provide formal semantics for phrase 'until...at i times' and 'at i times since' used in Fig.3 respectively. Intuitively speaking, A until-$_i$ B is true if A holds true until B becomes at i times, and times[A,B] examines how many times B becomes true from a time when A

became true most recently.

Fig.8, Fig.9 and Fig.10 shows the translations of 'line' in AB protocol, of 'receive' in hardware level protocol, and of interface specification respectively. The translations of dynamic class definitions, e.g. 'line', 'clock' and 'register' in Fig.7 and Fig.8 are described as schemas and their actual translations are generated from the schemas for each instance of the classes, e.g. for line RSL, formula 'sending-RSL∧send-RSL-arg=a∧RSL=q → ○◇(finish(send-RSL)∧RSL=send[a,q])' are generated from schema 2) in Fig.8.

```
Line
 (1) read[send[m,1]] = <m,empty-line> ∨
     send[m,1]=empty-line
 (2) sending ∧ send-arg=a ∧ line=q
       → ○◇(finish(send) ∧ line=send[a,q])
 (3) reading ∧ line=q
       → ○◇(finish(read) ∧ line=arg2[read[q]]
                         ∧ read-arg=arg1[read[q]])

[timing part]
 (4) begin(read)
       → waiting(read) until line≠empty-line
 (5) waiting(read) ∧ line≠empty-line → receiving
 (6) begin(send) → sending
 (7) □◇sending → ◇(finish(send) ∧ line≠empty-line)
```

Fig.8 Translation of line in AB protocol

```
receive
 1) begin(receive)
      → wait(receive)
        until ∃x[got[L,x] ∧ startbit[x]]
 2) wait(receive) ∧ got[L,b] ∧ start-bit[b]
      → begin(generate-Rclock)
 3) finish(generate-Rclock)
      → ready(receive) until receive-timing[Rclock]
 4) ready(receive) ∧ synchronizingly-got[L,b]
    ∧ ~detecting-stop-bits →
  4-1) continuously-receive[it,3]
         → stop(receive)
           until_2 receive-timing[Rclock]
  4-2) ~continuously-receive[it,3]
         → begin(serial-input-Receive-register)
           ∧ serial-input-Receive-arg=b
 5) finish(serial-input-Receive-register)
      ∧ ~detecting-stop-bits
        → ready(receive) until receive-timing[Rclock]
 6) stop(receive) ∧ receive-timing[Rclock]
       → ready(receive)
         until_2 receive-timing[Rclock]
 7) detecting-stop-bits
      → ready-finish(receive)
        until continuously-receive[it,4]
 8) ready-finish(receive)
    ∧ continuously-receive[it,4]
      → finish(receive)
 9) finish(receive)
      → ○(wait(receive)
          until ∃x[got[L,x] ∧startbit[x]])

detecting-stop-bits
 1) synchronizingly-got[L,0]
 2) ○(receive-timing[rclock]→got[L,1])
      until_4 receive-timing[Rclock]

receive-timing[c]
 1) c=1

continuously-receive[it,i]
 1) times[synchronizingly-got[L,1],
          synchronizingly-got[L,0]]=i

lexicon
 1) synchrinizingly-got[L,b]
    ↔ receive-timing[Rclock] ∧ got[L,b]
```

Fig.9 Translation of receive in hardware level protocol

VERIFICATION --- PROTOCOL EXAMPLE

As specifications written in Tell/NSL are translated into temporal logical formulas called axioms of specifications, we can reason about dynamic properties of system and verify the logical correctness on the basis of axiomatic system. As mentioned in (10), dynamic properties of interest can be classified into invariance (safety), eventuality (liveness), and precedence properties (until) and systematic proving methods are established (9) We can describe such properties using Tell/NSL in the same manner as the system specifications. As a result, comprehensive descriptions are provided. For example,

1) If transmitter begins to transmit message m,
   then transmitter is not sending the message different from m
   until receiver finishes receiving m.

is one of AB protocol's precedence properties written in Tell/NSL, and says that

```
Interface
 transformation schema
  1)   (l=empty-line) := (l=clear[])
  2-1) p(send) := p(pass-down)
  2-2) (send-arg=m) := (pass-down-arg=m)
  3-1) p(read) := p(pass-up)
  3-2) (read-arg=m) := (pass-up-arg=m)
  4)   (ll=send[m,12]) :=
           (ll=preset[m,12])
  5)  (<m,11>=read[12]) :=
           (<m,11>=parallel-load[12])
  6) line := Receive-register

 pass down
  1) begin(pass-down) ∧pass-down-arg=m
      → begin(preset-Send-register)
        ∧ preset-Send-register-arg=m
        ∧ passing-down
  2) finish(preset-Send-register)
      → ready-finish(pass-down)
        until finish(transmit)
  3) ready-finish(pass-down) ∧ finish(transmit)
      → ○finish(pass-down)

pass up
 1) begin(pass-up)
     → wait(pass-up)
       until ~empty[Receive-register]
 2) wait(pass-up) ∧ ~empty[Receive-register]
     → execute(pass-up) until wait(receive)
 3) execute(pass-up) ∧ wait(receive)
     → passing-up
 4) passing-up
      → begin(parallel-load-Receive-register)
 5) finish(parallel-load-Receive-register)
       ∧ parallel-load-Receive-register-arg=m
         → finish(pass-up) ∧ pass-up-arg=m
 lexicon
   1) empty[r] ↔ clear[]=r
```

Fig.10 Translation of interface specification

the protocol does not change order of transmission data. We can verify them directly using the proving method, but in this paper we do not touch upon the proof of these kinds of the properties on account of limited space.

In the case of layered system, we need verify not only dynamic properties of a system in a layer but also properties between adjacent layers. The verification of properties between layers is a kind of implementation verification, i.e. to verify whether operations in the layer are completely supported by the next lower layer. The step of implementation verification for layered system using Tell/NSL is as follows.

1) Translate a target specification sentence of the supported layer into a logical formula. Let the logical formula be A.

2) Transform formula A to formula A' which is a description of the next lower layer using the interface specification. Tranformation operator ' are defined by

  i) c'= A where c is a constant and c:=A appears in a set of translations of the interface specification.

  ii) x'= x where x is a variable. The type of left-hand side x may be different from that of right-hand side.

  iii) $(f(t_1,\ldots,t_n))' = F(t_1',\ldots,t_n')$ where f and F are function symbols or predicate ones and $f(x_1,\ldots,x_n):=F(x_1,\ldots,x_n)$ appears in a set of translations of the interface specification. We think that predicate symbols contain equality symbol =.

  iv) $(A \vee B)' = A' \vee B'$, $(\neg A)' = \sim A'$, $(\forall x A)' = \forall x' A'$

  v) $(\Diamond A)' = \Diamond A'$, $(\bigcirc A)' = \bigcirc A'$, $(A \text{ until } B)' = A' \text{ until } B'$

3) Show that formula Init → A' is derivable from the next lower layer specification, where Init is a initial condition in the lower layer.

We prove that a few formulas in the 'line' specification in AB protocol in Fig.8 is derived from the specification of hardware level protocol machine.

[example 1] : formula 3)

reading∧line=q → ○◇(finish(read)∧line=arg2[read[q]]∧read-arg=arg1[read[q]] ...(1)

It says the correctness of receive operations. Interface section in Fig.10 describes that formulas 'reading', 'finish(receive)', and 'line' in AB protocol correspond to formula 'passing-up', to 'finish(passing-up)' and to Receive-register in hardware level protocol respectively. Thus formula (1) is transformed to

passing-up∧Receive-register=q → ○◇(finish(pass-up)∧
Receive-register=arg2[parallel-load[q]∧
pass-up-arg=arg1[parallel-load[q]]) ...(2)

We show formula Init → □(2) is derivable from the specification of hardware protocol in Fig.7 and 9, and of the interface specification in Fig.10. Steps of proof are as follows.

⊢begin(parallel-load-Receive-register)∧Receive-register=r
→ ○◇(finish(parallel-load-Receive-register)∧
parallel-load-Receive-register-arg=arg1[parallel-load[r]]∧
Receive-register=arg2[parallel-load[r]]) ...(3)
: from the instatntiation of register 8) in Fig.7 by Receive-register

⊢passing-up∧Receive-register=r → ○◇(finish(parallel-load-Receive-register)∧
parallel-load-Receive-register-arg=arg1[parallel-load[r]]∧
Receive-register=arg2[parallel-load[r]]) ...(4)
: from 3], pass-up 4) in Fig.10, and ⊢A∧B→C and ⊢D→A implies ⊢D∧B→C

⊢ (2) : from (4), pass-up 5), and ⊢A→B and ⊢C→○◇A implies ⊢C→○◇B ...(5)

⊢ Init → □(2) : from (5), ⊢A implies ⊢□A, and ⊢B implies ⊢C→B

[example 2] : formula 1)

receive[send[m,1]]=<m,empty-line>∨send[m,1]=empty-line ...(1)

Formula (1) is equivalent to

(x=send[m,1]∧y=empty-line → receive[x]=<m,y>) ∨
w=send[m,1]∧u=empty-line → w=u) ...(2)

(2) is transformed into

(x=preset[m,1]∧y=clear[] → parallel-load[x]=<m,y>) ∨
w=preset[m,1]∧u=clear[] → w=u)

and then simplified to

(parallel-load[preset[m,1]]=<m,clear[]>∨preset[m,1]=clear[] ...(3)

⊢parallel-load[preset[m,1]]=<preset[m,1], clear[]> ...(4)
from register 1), 3), and ⊢A(x) implies ⊢A(t) for any term t

⊢parallel-load[preset[m,1]]=<m, clear[]> ...(5)
from register 2) and (5)

⊢(2)

⊢ Init → □(2)

[example 3] : formula 5) wait(read)∧line≠empty-line → ◇reading ...(1)

Formula (1) is transformed into

wait(pass-up)∧Receive-register≠clear[] → ◇passing-up ...(2)

⊢wait(pass-up)∧Receive-register≠clear[] → ◇(execute(pass-up)∧wait(receive))
from pass-up 3) and ⊢A until B → ◇(A∧B) ...(3)

⊢(2) : from pass-up 4), and ⊢A→◇B and ⊢B→◇C implies ⊢A→◇C ...(4)

⊢ Init → □(2)

CONCLUSION

We have introduced specification technique for concurrent systems having layered architecture by Tell/NSL and verified that AB protocol is supported by hardware level protocol. Tell/NSL provides a natural way of module design, readability and capability of semantical processing by machine. Our examples of verifications are by manual. We are developing a semi-mechanical, i.e. interactive verifier.

In Tell/NSL, specifications which describes detail actions in the system, e.g. protocol specification, looks like procedural programs. We are studying about synthesis of programs from the specifications of concurrent system described in Tell/NSL. This is also one major direction of future research.

[Acknowledgement]
We would like to thank Dr. A. Kurematsu, Dr. Y. Urano, Mr. K. Chiba, Mr. T. Takizuka of KDD Reasearch and Development Laboratories and Dr. T. Yokoi of Institute for New Generation Computer Technology for insightful comments and numerous discussions. Many colleagues at TELL project provided helpful comments on various versions of this paper, including Y. Shinoda.

[References]
(1) Parnas,D.L. : On the Criteria to Be Used in Decomposing Systems into Modules, Commun. ACM, 15, 12 (Dec.1972) pp.1053-1058.
(2) Liskov,B.H. and Zilles,S.N. : Specification Techeniques for Data Abstractions, IEEE SE-1, 1, (Mar. 1975) pp.7-19.
(3) Guttag,J.V., Horowitz,E., and Musser,D.R. : Abstract Data Types and Software Validation, Commun.ACM, Vol.21, No.12 (1978) pp.1048-1064.
(4) Montague,R. : The Proper Treatment of Qualification in Ordinary English, Approaches to Natural Language, Reidel Dordrecht (1973).
(5) Hailpern,B. : Verifying Concurrent Processes Using Temporal Logic. Technical Report 195. Computer Systems Laboratory, Stanford University. (Aug. 1980).
(6) Schwartz,R.L. and Melliar-Smith,P.M. : From State Machines to Temporal Logic : Specification Methods for Protocol Standards, IEEE Transaction on Communications, 30, 12 (1982) pp.2486~2496
(7) Lamport,.L : Proving the Correctness of Multiprocess Programs, IEEE Transaction on Software Engineering 3, 2 (1977) pp.125~143.
(8) Manna,Z. and Pnueli,A. : Verifications of concurrent programs : The Temporal framework, Correctness Problem in Computer Science, Academic Press (1981), pp.215~273
(9) Yonezaki,N. and Katayama,T. : Functional Specification of Synchronized Processes Based on Modal Logic, Proc. of 6th ICSE (Sept. 1982) pp.208-217.
(10) Sunshine,C.A. et al. : Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models, IEEE Transaction on Software Engineering 8, 5 (1982) pp.460~489
(11) Zimmermann,H. : OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Transaction on Communication 28 (1980) pp.425~432
(12) Enomoto,H. et al. : Paradigms of Knowledge Based Software System and Its Service Image, Economics and Technology of Software Engineering - 3rd Seminar for Technology of Software, 1983
(13) Enomoto,H. et al. : Natural Language Based Software Development System TELL, ECAI-84, 1984

APPENDIX : a part of syntax rules and semantic rules of sentences in Tell/NSL

Each rule is described in the following form.

$A ::= B_1(E_1)\ B_2(E_2)\ \ldots\ B_n(E_n)$ ---- syntax rule

E ---- semantic rule

A is a non-terminal symbol and B_i is a non-terminal symbol or a terminal symbol. E_i is a translation of B_i. E, which is a translation , is a meaningful expression in which B_i's appear. {B} represents that B is ommissible.

<sentence>::={<for head>$_\alpha$(Z)} {<reserved adverb> (Y)} <term>$_\alpha$(P) <be-verb>$_\alpha$(W) { **not** (~)}
<complement>$_\alpha$(Q) { in the next time (O)}| :1
{Z}({Y}({O}{~} P(W(Q))))
{<for head>$_\alpha$(Z)} {<reserved adverb> (Y)} **there** <be verb>$_\alpha$(W) { **not** (~)}
<term>$_\alpha$(P) { in the next time (O)} | :2
{Z}({Y}({O}{~} ∃x[W(P)(x)]))
{<for head>$_\alpha$(Z)} {<reserved adverb> (Y)} <term>$_\alpha$(P) **will** { **not** (~)}
<be-verb>$_\alpha$(W) <complement>$_\alpha$(Q) {in the next time (O)} | :4
{Z}({Y}({O}{~} ◇P(W(Q))))
{<for head>$_\alpha$(Z)} {<reserved adverb> (Y)} <term>$_\alpha$(P) {**do not** (~)}
<general verb phrase>$_\alpha$(F) {in the next time (O)} | :5
{Z}({Y}({O} {~}P(F)))
{<for head>$_\alpha$(Z)} {<reserved adverb> (Y)} <term>$_\alpha$(P) **will** { **not** (~)}
<general verb phrase>$_\alpha$(F) {in the next time (O)} | :6
{Z}({Y}({O}{~}◇P(F)))
<sentence>(M) <connective>(C) <sentecne>(L) :9
M C L
if <sentence>(M) , **then** <sentence>(L) :10

<for head>::= **for every** <proper noun phrase-1>$_\alpha(G_1)$,...,<proper noun phrase-n>$_\alpha(G_n)$, :12
{<relative noun clause>$_\alpha(G_{n+1})$} |
$\lambda m \forall name'_1,...,\forall name'_n[G_1(name'_1)\wedge...\wedge G_n(name'_n)\{\wedge G_{n+1}(name'_1)\wedge...\wedge G_{n+1}(name'_n)\} \rightarrow m]$
for some <proper noun phrase-1>$_\alpha(G_1)$,...,<proper noun phrase-n>$_\alpha(G_n)$, :13
{<relative noun clause>$_\alpha(G_{n+1})$}
$\lambda m \exists name'_1,...,\exists name'_n[G_1(name'_1)\wedge...\wedge G_n(name'_n)\{\wedge G_{n+1}(name'_1)\wedge...\wedge G_{n+1}(name'_n)\} \wedge m]$
where name$_i$ is in <proper noun phrase-i>$_\alpha$. See rule 84 and 90.

<term>$_\alpha$::=<proper noun clause>$_\alpha(F)$| :15
$\lambda g \exists x[F(x)\wedge g(x)]$

<determinar>$_\alpha(D)$ <common noun clause>$_\alpha(F)$| :16
$D(F)$

<common noun clause>$_\alpha(F)$ **-plural** | :17
$\lambda f \exists S \forall x[element(x,S) \leftrightarrow F(x)\wedge f(x)]$

<plural determinar>$_\alpha(D)$ <common noun clause>$_\alpha(F)$ **-plural** | :18
$D(F)$

<term>$_\alpha(P)$ <coordinate connective>(C) <term>$_\alpha(P')$:19
$\lambda f[P(f)\ C\ P'(f)]$

<term2>$_\alpha$::=<determinar2>$_\alpha(D)$ <common noun clause>$_\alpha(F)$:22
$D(F)$

<be verb>$_\alpha$::=**be** | :24
$\lambda p \lambda x[p(\lambda y[x=y])]$

<complement>$_\alpha$::=<adjective phrase>$_\alpha(F)$| :26
$\lambda g \exists x[F(x)\wedge g(x)]$

<term>$_\alpha(P)$:27
P

<adjective phrase>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}$::=adjective$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}$| :48
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n[adjective'(x_1)..(x_{i-1})(x_{i+1})..(x_n)]$

<adjective phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <preposition$_{i-1}$> <term>$_{\alpha_i}(P_i)$| :49
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n \exists x_i[U(x_1)..(x_i)..(x_n)\wedge P_i(\lambda y_i[y_i=x_i])]$

<adjective phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <preposition$_{i-1}$> <term2>$_{\alpha_i}(P_i)$| :50
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n \exists x_i[U(x_1)..(x_i)..(x_n)\wedge P_i(\lambda y_1 \lambda y_i[y_1=x_1 \wedge y_i=x_i])]$

<adjective phrase>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}(U_i)$ <coordinate connective>(C)
<adjective phrase>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}(U'_i)$| :51
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n[U_i(x_1)..(x_{i-1})(x_{i+1})..(x_n)\ C\ U'_i(x_1)..(x_{i-1})(x_{i+1})..(x_n)]$

<adjective phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <preposition$_{i-1}$> <verbal phrase>$_{\alpha_i}(P_i)$| :52
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n \exists x_i[U(x_1)..(x_i)..(x_n)\wedge P_i(\lambda y_i[y_i=x_i])]$ where i=2, n=2

<verbal phrase>$_\alpha(F)$ - ing :53
F

<common noun phrase>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}$::=<extended common noun>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}(U_i)$| :56
U_i

<adjective phrase>$_{\alpha_1}(F_1)$ <common noun phrase>$_{[\alpha_1,...,\alpha_{i-1},\alpha_{i+1},...,\alpha_n]}(U_i)$| :57
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n[U_i(x_1)..(x_{i-1})(x_{i+1})..(x_n)\wedge F_1(x_1)]$

<common noun phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <preposition$_{i-1}$> <term>$_{\alpha_i}(P_i)$:59
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n \exists x_i[U(x_1)..(x_i)..(x_n)\wedge P_i(\lambda y_i[y_i=x_i])]$

<common noun phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <preposition$_{i-1}$> <term2>$_{\alpha_i}(P_i)$| :60
$\lambda x_1..\lambda x_{i-1}\lambda x_{i+1}..\lambda x_n \exists x_i[U(x_1)..(x_i)..(x_n)\wedge P_i(\lambda y_1 \lambda y_i[y_1=x_1 \wedge y_i=x_i])]$

<common noun clause>$_{[\alpha_1,...,\alpha_n]}$::=<common noun phrase>$_{[\alpha_1,...,\alpha_n]}(U)$| :64
U

<common noun phrase>$_{[\alpha_1,...,\alpha_n]}(U)$ <relative pronoun clause>$_{\alpha_i}(F_i)$:65
$\lambda x_1...\lambda x_n[U(x_1)...(x_n)\wedge F_i(x_i)]$ where i=1, n=1

<relative pronoun clause>$_\alpha$ is ommitted. Subjective case and objective case are available, but modified words are restricted to the words which occur just on the left of relative pronouns.

<proper noun>$_\alpha$::=name$_\alpha$:84
name'$_\alpha$

<proper noun clause>$_\alpha$::=<proper noun phrase>$_\alpha(F)$ | :86
F

<proper noun phrase>$_\alpha(F)$ <relative pronoun clause>$_\alpha(G)$:87
$\lambda x[F(x)\wedge G(x)]$

<proper noun phrase>$_\alpha$::=<proper noun>$_\alpha(X)$ | :90
$\lambda x[x=X]$

<proper noun>$_\alpha$(X) <adjecive phrase>$_{[\alpha]}$(F) | :91
$\lambda x[x=X \wedge F(x)]$

<common noun phrase>$_\alpha$(F) <proper noun>$_\alpha$(X) :93
$\lambda x[x=X \wedge F(x)]$

<determinar>$_\alpha$::= a (**the**) | :96
$\lambda f\lambda g\exists x[f(x)\wedge g(x)]$

every | :97
$\lambda f\lambda g\forall x[f(x)\rightarrow g(x)]$

no :98
$\lambda f\lambda g\neg\exists x[f(x)\wedge g(x)]$

<determinar2>$_\alpha$::= **any other** :100
$\lambda f\lambda b\exists x\forall y[x\neq y\wedge f(y)\rightarrow b(x)(y)]$ (positive)
$\lambda f\lambda b\exists x\exists y[x\neq y\wedge f(y)\wedge b(x)(y)]$ (negative)

<plural determinar>$_\alpha$::= <number>$_{integer}$(N) | :108
$\lambda f\lambda g\exists S[\forall x[element(x,S)\leftrightarrow f(x)\wedge g(x)]\wedge number(S)=N]$

<number>$_{integer}$(N) **of** :109
$\lambda f\lambda g\exists S[\exists T[\forall x[element(x,T)\leftrightarrow f(x)]\wedge subset(S,T)]\wedge$
$\forall x[element(x,S) \leftrightarrow g(x)] \wedge number(S)=N\ ]$

some of :110
$\lambda f\lambda g\exists S[\exists T[\forall x[element(x,T)\leftrightarrow f(x)]\wedge subset(S,T)]\wedge$
$\forall x[element(x,S) \leftrightarrow g(x)] \wedge number(S)\geq 0\ ]$

<general verb phrase>$_\alpha$::=verb <verbal phrase>$_\alpha$ (A) :114
$\lambda x[A(x)(verb')]$

<verb phrase>$_\alpha$::=verb | :116
$\lambda x\lambda f[f(_x verb')]$

verb <term>$_{\alpha_1}$(P_1) | :117
$\lambda x\lambda p[p(_x verb')\wedge P_1(\lambda x_1[_x verb'\text{-}arg1=x_1])]$

<verb phrase>$_\alpha$(A) <preposition$_{i-1}$> <term>$_{\alpha_i}$(P_i) :119
$\lambda x\lambda f[A(f)(x)\wedge P_i(\lambda x_i{}_x verb'\text{-}argi=x_i])]$

<verbal phrase>$_\alpha$::=<verb phrase>$_\alpha$ (A)-**infinitive** | :124
A

<verb phrase>$_\alpha$ (A)-**ing** :125
A

<connective>::= <coordinate connective> (C) | :128
C

until | :129
until

<coordinate connective>::= **and** | :138
$\wedge$

or :139
$\vee$

<reserved adverb>::= **initially** | :142
$\lambda m[\ Init'\rightarrow m]$

Types of the variables and meta-variables in the above rules are as follows.

x,y: α , z: α_{n+1} , x_i,y_i: α_i , F,f,G,G_i,g: <α t> , p,P,P',Q: <<α t> t> , M,m,L,l: t , N: integer ,
Y,Z: <t t> , W: <<<α t> t> <α t>> , U_i,U_i': <α_1<...α_{i-1}<α_{i+1}<...α_n t>..> , U: <α_1<...<α_n t>..>,
A: <α <<α t> t>>, P_i: <<α_i t> t> , u: ω , h,H: <ω <α t>> , r,R,R': <<ω <α t>> <α t>> , k,K: <ω t> ,
O,O': <α_i <<ω t> t>> , V: <<α_i <<ω t> t>> <ω <α_i t>>> , s: <α <<ω t> t>> , S,T: **set of** α ,
C: <t <t t>> , P_ω: <<ω t> t> , B: <ω <α_i t>> , E: <<ω <α_i t>> <α_i t>> , D: <<α t> <<α t> t>> ,
F_i: <α_i t> , b: <α <t t>> , D_ω: <<ω t> <<ω t> t>>