CONCURRENT PROGRAM SYNTHESIS WITH REUSABLE COMPONENTS USING TEMPORAL LOGIC

Naoshi Uchihira
Toshiaki Kasuya
Kazunori Matsumoto
Shinichi Honiden

Reprinted from PROCEEDINGS OF THE ELEVENTH ANNUAL INTERNATIONAL COMPUTER SOFTWARE & APPLICATIONS CONFERENCE — COMPSAC, Tokyo, Japan, October 7-9, 1987

The Computer Society of the IEEE
1730 Massachusetts Avenue NW
Washington, DC 20036-1903

Washington • Los Alamitos • Brussels

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY PRESS

CONCURRENT PROGRAM SYNTHESIS WITH REUSABLE COMPONENTS USING TEMPORAL LOGIC

Naoshi Uchihira, Toshiaki Kasuya, Kazunori Matsumoto, and Shinichi Honiden

Systems & Software Engineering Lab.
TOSHIBA Corporation
Yanagicho 70-banchi, Saiwai-ku, Kawasaki, Kanagawa 210, JAPAN

ABSTRACT
A concurrent programming model is provided, which is oriented to data stream, software reuse, and prototyping. Based on this programming model, a program synthesis method is described. This synthesis method consists of two parts: (1) retrieving and interconnecting components from I/O data specifications and (2) synthesizing a synchronization supervisor from a propositional temporal logic specification. MENDEL/87, which is a Prolog-based concurrent object-oriented language, is used as the programming language in this model. This synthesis method has been implemented by PROLOG on a PROLOG machine.

I. INTRODUCTION

The two major purposes of program synthesis and automatic programming are to generate a program that is assured of being correct, and to increase software productivity. Our main goal is the latter, especially for concurrent programs.

Recently, software reuse is expected to greatly increase software productivity. In fact, many attempts have been made on research and practical levels, and in various ways. Because of these many efforts, sofware reuse is just getting under way. Accordingly, much research on software components interconnection and software-resuse-based program synthesis has been presented. However most of them are only for sequential programs, not for concurrent programs.

For concurrent programs, verification has been investigated for a long time. Some efforts involve verification using a linear time propositional temporal logic (PTL). In this case, PTL is a specification language, and a PTL decision procedure is able to verify whether the specification is consistent or not. As PTL is decidable and has various decision procedure algorithms [1,2], verification is accomplished automatically.

Manna and Wolper [3,4] use PTL for program synthesis. They show a theorem proving method which can synthesize synchronization parts of a concurrent program using PTL or extended temporal logic (ETL). In this method, a model graph, which is generated in the decision procedure, is considered as a state transition diagram for processes. From this state transition diagram, CSP program codes which execute synchronization are generated.

There are other works about synthesis using temporal logic in addition to Manna and Wolper's work. Clarke and Emerson [5] propose a synthesis method for the synchronization skeletons of a concurrent program using branching time temporal logic. Fujita, Tanaka, and Moto-oka [6] show a synthesis method of state transition diagrams using PTL for specifying hardware. Katai and Iwai [7] propose a method to generate scheduling rules of concurrent system from PTL specification and a Petri net.

We think the most practical approach to automatic programming in large scale applications is a program synthesis utilizing theorem prover with reusable components. Because theorem proving approach can synthesize only small scale programs and can not support large scale applications. In Manna and Wolper's synthesis method, a synchronization part is generated automatically. However, another part, say a functional part, must be created by the programmer. This paper proposes a new synthesis method which is a combination of software reuse and Manna and Wolper's method. This method consists of two major parts: (1) retrieval and interconnection of reusable components and (2) synthesis of a synchronization supervisor. This method generates a MENDEL/87 program. MENDEL/87 is a Prolog-based concurrent object-oriented language [8]. Now we have been implementing a concurrent program prototyping system based on this synthesis method.

II. PROGRAMMING MODEL

This section provides a programming model for a concurrent program, which is more or less restricted but very simple

0730-3157/87/0000/0455$01.00 © 1987 IEEE

and intuitive. This model is based on the following concepts:

(1) Data stream oriented programming

The program consists of a number of processes which can run concurrently. A process itself is a sequential program. Processes communicate with and synchronize each other only by the data stream through communication pipes; so have no shared variables. Communication pipes are statically defined before execution in the same way as Occam [9].

(2) Software reuse

Assume that a large enough number of processes have already been created and stored in a library. In principle, a goal program can be synthesized by interconnecting some of the processes in a library. In this case, a process is a reusable component, which cannot be modified when reusing it. Sometimes it is called a "black box" reusable component. Each process looks like an IC. Just as an electronic circuit is composed by connecting a number of LSIs, a program can be synthesized by interconnecting a number of processes.

(3) Synchronization supervisor separated from processes

When a programmer prepares reusable processes, he does not know how these processes will be interconnected. A reusable process should be independent from other processes, while synchronization is regarded as interaction between processes. It is difficult and undesirable that the synchronization codes be written in the internal part of each reusable process. The synchronization supervisor should be separated from reusable processes, such as a path expression [10,11]. Therefore, processes are retrieved and interconnected first, then the synchronization supervisor is provided on these processes (Fig. 1).

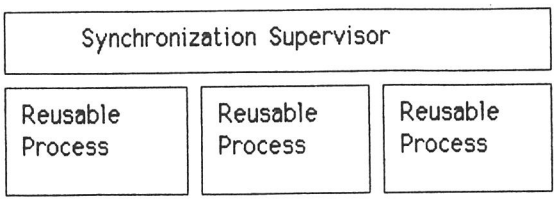


FIG.1 Synchronization supervisor separated from reusable processes

(4) Prototyping

Interconnected processes are executable with a synchronization supervisor. This is a prototype. This prototype can be examined to determine whether or not it runs in the way the user needs. If there is anything unsatisfactory, the user can easily transfer to any previous programming step and modify it. If this prototype is acceptable, the synchronization supervisor will be transformed and embedded inside of each process automatically; so the user can get a new, truly concurrent program without a synchronization supervisor.

III. MENDEL/87

a. Overview

We suggest a concurrent object-oriented programming language with something like Occam + OPS5 + PROLOG; call this language MENDEL/87. MENDEL/87 program consists of one main program and several objects; the object consists of several methods. The object can be regarded as a process. Each object has finite pipe caps and can transmit messages only through the pipe caps. The attribute is assigned to each pipe cap and is used to identify input/output messages. The message is transmitted between objects through the transmission pipe connected with pipe caps, as shown in Fig. 2. The pipe is a one-to-one asynchronous one-way path. Initial messages are applied through the input nodes, and goal messages are extracted from the output nodes. The gate and the signal gate are used for message stream control.

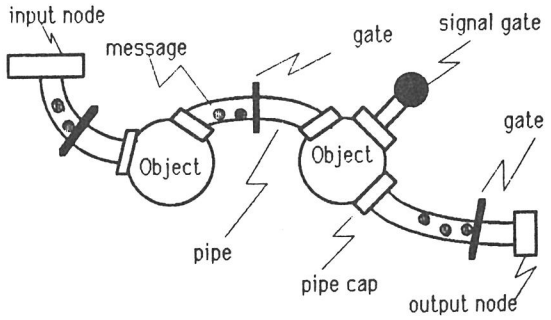


Fig.2 Objects, Pipes, and messages in MENDEL/87

b. Object

A MENDEL/87 object consists of a declaration part, a method part and a junk part.

```
<object name>
{
     dec :  { <declaration part> }
     meth :  { <method part>      }
     junk :  { <junk part>        }
}
```

The declaration part includes declarations of five items :

1) Input pipe cap attributes

inpipe(<attribute>, ...).

2) Output pipe cap attributes
outpipe(<attribute>, ...).
3) Input signal attributes
insignal(<attribute>, ...).
4) Output signal attributes
outsignal(<attribute>, ...).
5) Internal state variables
state(<attribute>!<initial value>,...).

The method part includes several methods. The method is described as follows :

```
method ( <attribute> ? <term>, ... ,
                <attribute> ! <term>, ... )
          <-  <guard> | <Prolog goals>.
```

where, <attribute> means pipe cap attribute, signal attribute, or internal variable attribute; "?" means input and and "!" means output, like CSP; "|" means commitment operator like GHC; and only Prolog predicates with no side effect can be written in <guard>.

```
ex.  method( hour ? H, minute ! M )
     <- H >= 0 | M is H*60, write(M), nl.
```

The method is selected when :
(1) Each of the terms after **<attribute>?** can be unified with a received message from the pipe, signal, or state variable indicated by the attribute.
(2) All predicates in <guard> succeed.

When the method is selected, all Prolog predicates in **<Prolog goals>** are evaluated. Each term after **<attribute>!**, which has been unified in **<Prolog goals>**, is sent as a message into the pipe, signal, or state variable indicated by the attribute. If **<Prolog goals>** fails, warning messages are given to the user and the method terminates without sending output messages.

This method selection mechanism is similar to a Dijkstra's guarded command, Occam's alternative construct, and GHC. Moreover, the method can be considered as a production rule in a production system, such as OPS5.

The junk part includes some Prolog clauses, which may be called by methods or other Prolog clauses. One example of MENDEL/87 object is shown in Fig. 3.

c. Main program

A MENDEL/87 program has one main program which interconnects several objects with pipes, and then activates them, sends initial messages to input nodes, and receives goal messages from output nodes. The main program seems like an extended module call and takes the following form:

```
communicate[
<object name>(
        <attribute>!<term or pipe>, ...
        <attribute>?<term or pipe>, ... ),
<object name>(
        <attribute>!<term or pipe>, ...
        <attribute>?<term or pipe>, ... ),
          ............  ............
].
```

All objects in the **communicate** construction run in AND-parallel and are interconnected explicitly with **<pipe>**. The **<term>** after **<attribute>!** is a list of initial messages and the **<term>** after <attribute>? is a variable waiting for a list of goal messages.

In the prototyping system described later, this main program is generated by system behind the scenes. The user need not know this syntax.

d. Gate and Gate controller

In MENDEL/87, a simple synchronization mechanism is given by a method selection mechanism. The object is suspended until it can receive all required messages. However, it is so simple that a complicated synchronization requires

```
KeyCheck
(
    dec :(
        inpipe(keyword,word).
        outpipe(ckeck_data).
        state(keywordlist![]).
    )
    meth :(
                % receive a keyword and store it in the keywordlist.
        method(keyword?KW,keywordlist?KWL,keywordlist![KW|KWL]).
                % receive a word and if it is a keyword, send it to check_data.
        method(word?W,keywordlist?KWL,check_data!W) <-
                member(W,KWL) | true.
                % receive a word and if it isn't a keyword, do nothing.
        method(word?W).
    )
    junk:(
        member(_,[]) :- !,fail.
        member(X,[X|_]).
        member(X,[_|Y]):- !,member(X,Y).
    )
)
```

Fig.3 MENDEL/87 Object Example (KeyCheck)

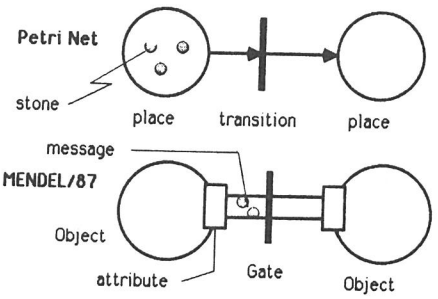


Fig.4. Petri Net and MENDEL/87

complicated pipe interconnection (many of them are only for control stream, not for data stream), so an additional synchronization mechanism using the gate and the gate controller is introduced. Every pipe has only one gate which controls the message stream. The gate opens, lets only one message pass through and then shuts. This is an atomic action of the gate. With no message in the gate, the gate cannot be opened. The gate is similar to the transition of a Petri Net (Fig. 4). The gate controller controls all gates, that is the synchronization supervisor.

e. Signal gate and end_of_gate

The signal gate itself is regarded as a kind of gate. When the input signal gate opens, it generates one signal message which has no value and is sent to an object only for the method selection, like a binary semaphor. When the output signal gate opens, it consumes one signal message which is sent from an object.

The end_of_gate is a system output signal gate. When detecting that there are no more messages passing through the gate **g** in the future, the system (interpreter) sends one signal message to output signal gate **eog(g)** (end of gate **g**). The end_of_gate fills the role of a terminal symbol, such as end_of_file and end_of_string in C language. As a terminal symbol itself is not a data, but a control signal, it should not be treated in the same way as other messages. In MENDEL/87, the signal message and the signal gate are distinguished from the ordinary message and gate.

IV. OBJECT RETRIEVAL AND INTERCONNECTION

MENDEL/87 objects can be retrieved and interconnected in two main ways; manually and automatically.

a. Manual

Write a MENDEL/87 main program. To be more precise, select an object from an object library and interconnect these objects with pipes.

b. Automatic

Give program specifications as a set of input/output attributes. That is a kind of I/O data type. Objects are then selected from an object library and interconnected automatically, according to the given I/O attributes. Automatic retrieval and interconnection are carried out, according to the following principles:

(1) A pair of pipe caps having the same attributes can be interconnected.

(2) All required output attributes must be reachable from given input attributes through connected objects and pipes.

For example, if the following attributes are gven, object B, C, and D are retrieved and interconnected as shown in Fig. 5.

Input attribute a, b ;
Output attribute e ;

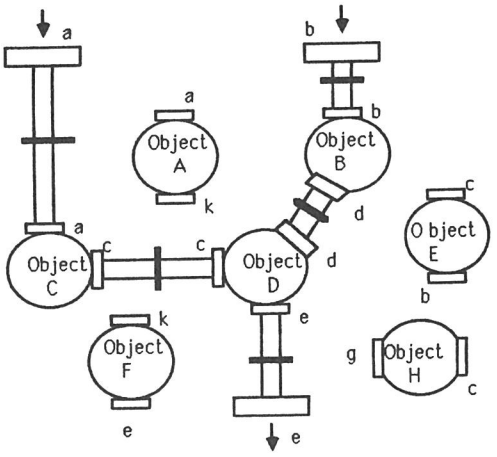


Fig.5. Automatic Obect Retrieval and Interconnection

c. More Flexible Automatic Binding

This automatic retrieval and interconnection, which we call "automatic binding", seems to be not enough powerfull. The binding mechanism depends on the simple pattern matching between output and input attribute names. In some cases, it might find no candidate to fit the given I/O attributes, or a lot of candidates in other cases. More information must be needed to select the most adequite candidate.

To overcome this problem, we adopt a kind of semantic network (called "Attribute Network" [12]) which represents the attribute structure and define a metric to order the candidates on the semantic network.

Attribute network

We define **simple attribute** as an attribute which has no structure (ex. heart, human, animal, and real). The simple attribute is also called **class**. We can introduce several interrelations among these simple attributes. In this paper, especially we consider 'IS_A' (a relation between superclass and subclass) and 'HAS_A' (a relation between class and its properties).

ex. human IS_A animal.
human HAS_A heart

The attribute network is defined as a semantic network where simple attribute is assigned to node and two relations ('IS_A' and 'HAS_A') are assigned to links respectively.

Example

An attribute 'heart' has the weight, the weight has a integer and the 'heart' is the lower concept of the 'pump'. Also, 'human's heart' is the constrained concept of the 'heart'. These relations about 'heart' can be expressed by the attribute network shown in Fig.6.

In the attribute network, there exists 'Object' which is the superclass of every class. Each class inherits properties of its superclass based on single inheritance rule. This network has the same class structure as Smalltalk-80.

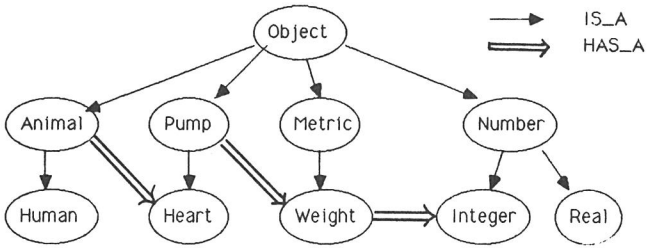


Fig.6. Attribute Network Example

Attribute definition

We define a new attribute syntax on the attribute network.

```
<attribute> ::=
  <simple attribute> | <complex attribute>.
<complex attribute> ::=
  <attribute> of <simple attribute>.
<simple attribute> ::= <class>.
   <class> ::= <Prolog atom>.
```

The 'of' expresses the 'HAS_A' relation, and in the example shown in Fig.6, 'integer of weight of heart of human' means the weight integer value of human's heart.

Binding Mechanism

The new binding mechanism will be defined. This mechanism makes it possible semantic matching of attributes by using a semantic net. For the given output attribute, the most similar input attribute is uniquely found automatically. Note that the binding direction is from the output to the input.

For given output attribute, we define the **bindable** input attribute which can be bound to the output attribute. An input attribute can be bindable to the output attribute iff the input attribute semantically includes the concept of the output one. For example, we consider following case.

1) integer of weight of heart of human
2) integer of weight of heart of animal

In this case, as 'human' is the a subclass of 'animal', attribute 1) is able to be bound to attribute 2), but attribute 2) is not able to be bound to attribute 1) because 1) is a more constrained concept than 2). We define it strictly.

DEFINITION 1. (Bindable Set)

Let A (= a_1 of a_2 of ... a_m) be an output attribute. U(A) is a **bindable input attribute set** from A iff U(A) = $\{$ B (= b_1 of b_2 of ... b_n) $|$ B is an input attribute, m>=n, and b_i is a superclass or same class of a_i for all i $\in \{1,2,\ldots,n\}$ $\}$.

We introduce an ordering of similarity of attributes. As this ordering construct total ordering, it can be used to get the most similar attribute.

DEFINITION 2. (Ordering of similarity)

Let A (= a_1 of a_2 of ... a_1) be an output attribute, and U(A) be a bindable input attribute set from A. $d_A(B) < d_A(C)$ means B is more similar to A than C, where B (= b_1 of b_2 of ... b_m) $\in$ U(A) and C (= c_1 of c_2 of ... c_n) $\in$ U(A).

More precisely, define

$d_A(B) < d_A(C)$ iff (1) or (2).

(1) there exists i $\in \{1,2,\ldots,\min(m,n)\}$ such that $b_1=c_1$, $b_2=c_2$, ... $b_{i-1}=c_{i-1}$, and c_i is a superclass of b_i.

(2) m>n, and $b_i=c_i$ for all i $\in \{1,2,\ldots,n\}$.

THEOREM

This ordering defined above is total ordering.

(proof) It is clear from the definitions.

For example, let
A = 'integer of weight of heart of human',
B = 'integer of weight of heart',
C= 'integer of weight of heart of animal', and
D = 'integer of weight of pump of human'.
In this case the ordering is $d_A(C) < d_A(B) < d_A(D)$.

By the way, the main topics of this paper are the how to synthsize the synchronization supervisor from PTL

specification and how to utilize the supervisor in the concurrent program prototyping sytem. Therefore, in the following sessions, we use only simple attributes for clear understanding of main topics.

V. SYNCHRONIZATION SUPERVISOR SYNTHESIS

a. Specification language for synchronization

(1) PTL

PTL is a linear time propositional temporal logic which has the following temporal operators in addition to usual logical symbols ('&' -- AND, '#' -- OR, '-' -- NOT, '=>' -- IMPLY) :

```
[]f (read always f) :
      f is true for all future states
<>f (read eventually f) :
      f is true for some future state
@f  (read next f) :
      f is true for the next state
f1 $ f2 (read f1 until f2) :
      f1 is true until f2 becomes true
```

(2) Model

An atomic proposition in PTL corresponds to an atomic action of the gate (includes signal gate) in MENDEL/87. That is, "**g** is true for the state" means "gate **g** opens, lets only one message pass through, and then shuts at the state". In the same way, "<>g is true" means "gate **g** will open at some future state", and "[]**g** is true" means "gate **g** is always open". Moreover, it is assumed that only one gate can open at the same state (this assumption called single-event condition). The single event condition means that only one atomic proposition is true for each state. For example, a specification that "gates **g1** and **g2** open by turns in Fig. 7" is expressed by the following PTL formulas:

```
[]( g1 => @g2 )
[](g2 => @g1 )
```

b. Scheduling rule synthesis for the gate controller

In MENDEL/87, the synchronization supervisor synthesis means synthesis of scheduling rules, by which the gate controller selects a gate to be opened. While the gate controller selects a gate according to the rules synthesized from a specification, the order of selected gates satisfies the specification. This synthesis method is based on Manna and Wolper's tableau-like PTL decision procedure [3]. A brief summary of the synthesis method is as follows:

(Step1) First initial PTL formulas are decomposed into current formulas, which include no temporal operator, and future formulas by the decomposition procedure. Future formulas are also decomposed into current and future formulas from the next state point of view. After every kind (a finite number) of future formulas has been repeatedly decomposed, a graph is derived. Each edge of that graph corresponds to current formulas for each decomposition. This graph is an incomplete model satisfying specifications other than eventuality formulas, such as <>F, -[]F and -(-F1 $ F2).

(Step2) Edges with unsatisfiable eventuality formula are deleted from the graph by the elimination procedure. The graph remaining after the elimination procedure is a complete model of the initial PTL specification.

(Step3) This model graph can be regarded as a state transition diagram. Scheduling rules are translated from this model graph. Each rule corresponds to a transition on the model graph. These scheduling rules are completed by adding the following fairness strategy:

Fairness Strategy: If there are several possible transitions/rules, one which has never been selected or for which the most time has elapsed from the last selection should be selected.

The state transition diagram and scheduling rules for the previous example are shown in Fig. 8.

VI. CONCURRENT PROGRAM PROTOTYPING SYSTEM. OVERVIEW

This system consists of four steps.

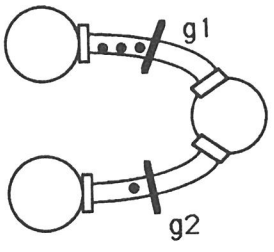
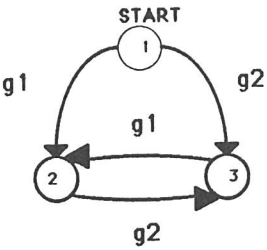


Fig.7 An example of gates



srule(n(1), n(2), g1).
srule(n(1), n(3), g2).
srule(n(2), n(3), g2).
srule(n(3), n(2), g1).

Fig. 8 State transition diagram and scheduling rules synthesized from PTL

(Step1) Make MENDEL/87 objects and store in a library. (Step2) Retrieve and interconnect objects. (Step3) Synthesize scheduling rules for the gate controller. (Step4) Set input data and Execute.

Since this system supports software prototyping, it is possible to go back to an arbitrary previous step (Fig.9). The system provides six windows as the user interface: (1) Command window, (2) Object window which displays MENDEL reusable objects in an object library, (3) I/O window which shows I/O specification of a goal program, (4) PTL window which shows a synchronization specification written by PTL, (5) Diagram window which illustrates objects and interconnections between their attributes, and (6) System message window, as shown in Fig. 10.

For each prototyping step, details will be described below.

(Step1) Make objects and store in a library.

Make MENDEL/87 objects in an integrated editor and store them in an object library. Synchronously, objects are compiled into intermediate codes. Objects in an object library are displayed in the object window.

(Step2) Retrieve and interconnect objects.

The system retrieves and interconnects objects manually and automatically as described in section IV. It is also possible to mix these two methods; objects are retrieved and interconnected automatically first and then modified manually. I/O specifications are entered using the I/O window, and the interconnection result is displayed on the diagram window.

(Step3) Synthesize sheduling rules.

While looking at interconnected objects and gates in the diagram window, the user provides a synchronization specification which define the order of gate to be opened, using the PTL window. The system then generates a state transition diagram from the PTL specification, and translates it into scheduling rules.

(Step4) Set data and execute.

From step1 to step3, the system can obtain the following.

* Compiled intermediate codes of objects
* Interconnection information
* Scheduling rules of the gate controller

The system generates executable codes by appending new codes to compiled intermediate codes to perform interconnection. The interpreter executes these codes. Execution is pseudo-concurrent on one CPU. The gate is implemented as a mail box. Therefore, the interpreter selects one of the objects waiting at the mail box and lets it receive a message according to the scheduling rules derived in step 3. This part of the interpreter that schedules a waiting object queue at each mail box is called the gate controller. A process of execution is displayed on the diagram window.

This system has been under construction using the object oriented Prolog (ESP [13]) on a PROLOG machine (PSI [14]).

VII. SYNTHESIS EXAMPLE: KEYWORD COUNT PROGRAM

We are going to synthesize the Key Word Count Program. This program reads a text (stream of characters) and a key word list, and then checks occurrence of key words in the text and reports its summary.

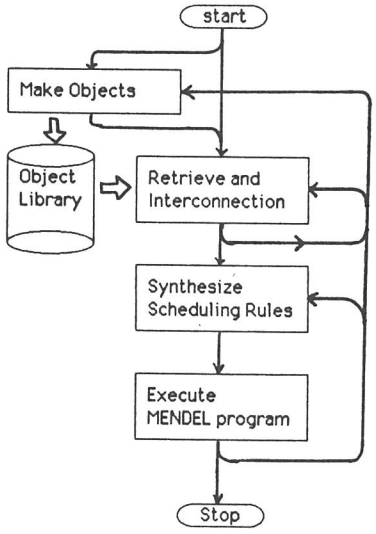


Fig.9. Prototyping Model in MENDEL/87

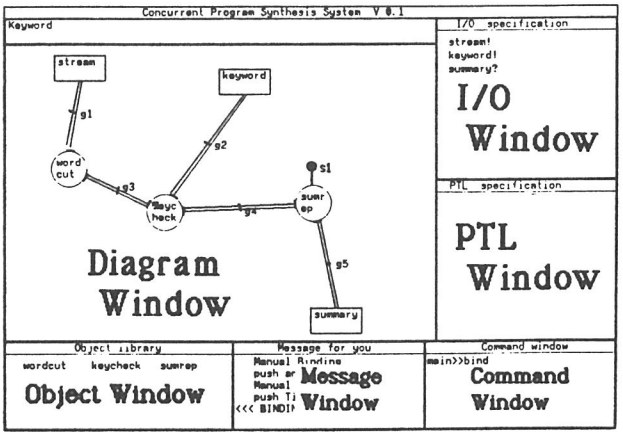


Fig. 10 Concurrent Program Prototyping System User Interface

(STEP1) It is assumed that there are many objects, especially text processing objects, in an object library.

(STEP2) In this example, the user selects the automatic mode and provides I/O attributes for a goal program using the I/O window,

```
goal program :
Input attribute        stream, keyword ;
Output attribute       summary ;
```

and chooses the menu command "automatic". The system then shows several objects in the diagram window, which have been retrieved and interconnected automatically (Fig.11). Here, three objects are selected:

```
WordCut :
Input attribute        stream ;
Output attribute       word ;

KeyCheck :
Input attribute        keyword, word ;
Output attribute       check_data ;

SumRep :
Input attribute        check_data ;
Output attribute       summary ;
```

WordCut reads a character stream and analyzes them to obtain words. KeyCheck reads keywords and sequence of words, checks for words that match one of keywords, and returns the result of checking. (This object was shown in Fig. 3.) SumRep sums up check data and makes a summary report.

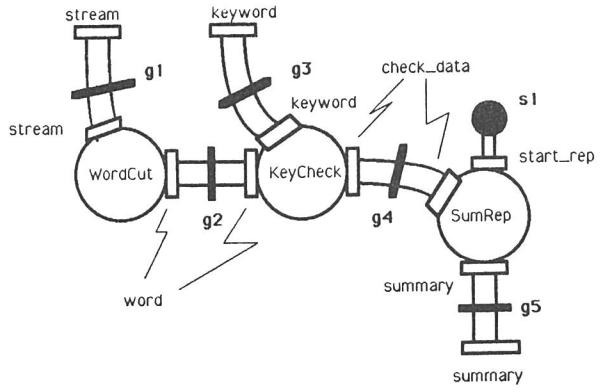


Fig. 11 Key Word Count Program

(STEP3) While looking at a diagram in the diagram window, the user inputs a PTL specification in the PTL window. The user may require the following :
(1) Both keyword and stream are finite. (Messages going through **g1**, **g3**, and **g4** are finite.)
(2) WordCut and KeyCheck can be processed concurrently. But KeyCheck must not receive words from WordCut and not start checking until all keywords have been received. (Does not open **g2** until there is no more message through **g3**.)
(3) SumRep analyzes all check data after KeyCheck has finished checking. (Does not open **s1** until there are no more messages through **g4**.)
(4) As an exception, **g5** is always open (i.e. out of synchronization).
(5) SumRep must make a summary report at last.

These requests are represented by the following PTL specification:
(1) FINITE(**g1**), FINITE(**g3**), FINITE(**g4**)
(2) - **g2** $ **eog(g3)**
(3) - **s1** $ **eog(g4)**
(4) **g5** doesn't appear in PTL.
(5) <> **s1**, [](**s1** => **@halt**), HALT
where FINITE and HALT are "macros" of this specification language. Each "macro" is expanded into the following PTL:

```
FINITE( g ) -->
<>eog(g) & [](eog(g)=>@([](-eog(g)&-g)))
```

The total number of messages passing through gate g is finite, and after all message have passed through the gate g, the system signal gate eog(g) must be opend only once.

```
HALT  -->  <>halt& [](halt=>@([]halt))
```

The program eventually terminates.

If an atomic proposition g dosen't appear in PTL, the gate g must be always open.

From these PTL formulas, the system generates a state transition diagram (Fig. 12) and translates it into scheduling rules.

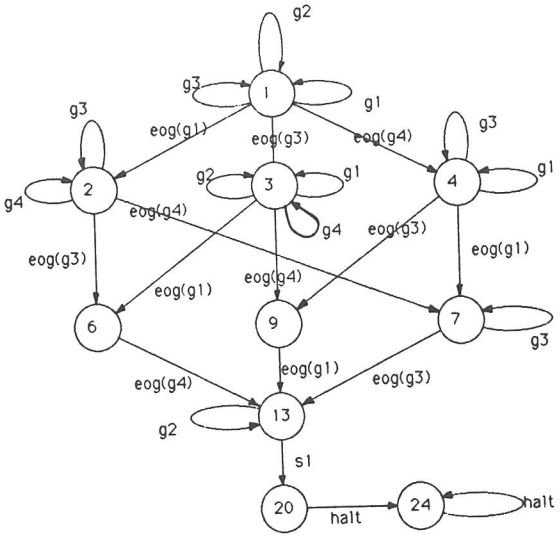


Fig.12 State transition diagram for Key Word Count Program

(STEP4) The user inputs two kind of data, a list of keywords and a text stream. The system then executes a synthesized MENDEL/87 program and outputs a summary report (Fig. 13).

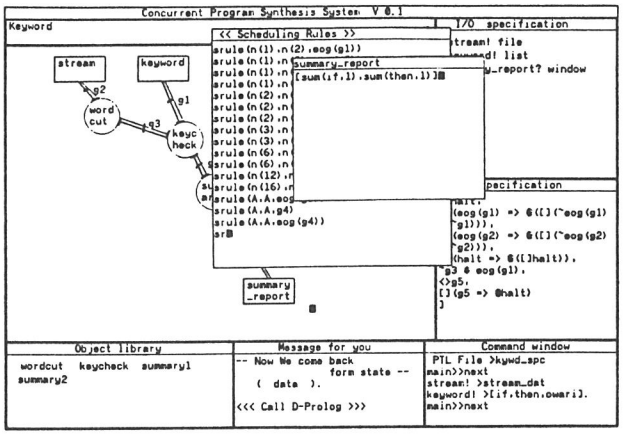


Fig. 13 Concurrent Program Prototyping System Execution of Keyword Count Program

VIII. CONCLUSION

We have proposed a concurrent program synthesis method and described the software prototyping system based on this method. This method consists of two major parts: (1) Retrieving and interconnecting components from I/O data specifications (2) Synthesizing the gate controller from propositional temporal logic specification. This method and its system are also illustrated with one example. Unique features of this method include: (1) A combination of software reuse and synchronization supervisor synthesis using temporal logic. (2) A new synchronization mechanism using gate, signal gate, and end_of_gate. We believe this method will become practical enough to help programming on a qualified domain, such as parallel text processing and concurrent business transactions.

At present, much remains to be explored:
(1) More expressive specification language than PTL should be investigated. It is another practical way to execute temporal logic specification directly as a program [15].
(2) We think that the object interconnection method based on the I/O specification has enough potential. But the current method is not fully satisfactory.
(3) The PTL specification has nothing to do with an internal program of each object. It is desirable to associate PTL specification with internal programs of objects:
(4) A more efficient implementation of the tableau-like PTL decision procedure should be developed.
(5) The degree of concurrency is low because of the limitation wherein the gate controller can only open one gate at a time. It seems possible to relax this limitation for some independent gates.
(6) A transformation technique that distributes synchronization supervisor function into each object is necessary for a distributed computing environment having no supervisor.

ACKNOWLEDGMENT

This research has been supported by the Japanese Fifth Generation Computer Project and its organizing institute ICOT, as a subproject of Intelligent Programming System. We would like to thank Toshio Yokoi and Hidenori Itoh of ICOT for their encouragement and support. We are also grateful to Seiichi Nishijima, Masahiko Arai, Takeshi Kohno, and Hideo Nakamura of Systems & Software Engineering Laboratory, TOSHIBA Corporation, for providing essential support, and Toshibumi Seki for implementing the attribute network.

REFERENCES

[1] Plaisted, D. A., A Decision Procedure for Combinations of Propositional Temporal Logic and Other Specialized Theories, Journal of Automated Reasoning 2, pages171-190, 1986.
[2] Fusaoka, A. and Takahashi, K., On QFTL and the refutation procedure on w-graphs, ICOT Technical Report TR-132, 1985.
[3] Manna, Z. and Wolper, P., Synthesis of communicating processes from temporal logic specification, ACM Trans. on Programming Languages and Systems, Vol.6, No.1, pages 68-93, 1984.
[4] Wolper, P., Synthesis of communicating processes from temporal logic specification, STAN-CS-82-925, Stanford University, 1982.
[5] Clarke, E. M. and Emerson, E. A., Design and synthesis of synchronization skeletons using branching time temporal logic, Logics of programs (Proceedings 1981), Lecture Notes in Computer Science 131, Springer-Verlag, pages 52-71, 1982.
[6] Fujita, M., Tanaka, H.,and Moto-oka, T., Specifying hardware in temporal logic & efficient synthesis of state-diagrams using Prolog, Proc. of FGCS'84, 1984.
[7] Katai, O. and Iwai, S., Construction of Scheduling Rules for Asynchronous, Concurrent Systems Based on Tense

Logic (in Japanese), Trans. of SICE (Japan) vol.18 no.12, 1982.
[8] Honiden, S., Uchihira, N., and Kasuya, T., MENDEL: PROLOG BASED CONCURRENT OBJECT ORIENTED LANGUAGE, Proc. of COMPCON'86, pages 230-234,1986.
[9] Occam Programming manual,INMOS Ltd., 1983.
[10] Habermann, A. N., Introduction to Operating System Design, SRA, 1976.
[11] Andler, S., Predicate Path Expression, Proc. of ACM 6th POPL, 1979.
[12] Chikayama, T., Unique features of ESP, Proc. of the international conference on FGCS1984, 1984.
[13] Taki, T. et al., Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI), Proc. of the international conference on FGCS1984, 1984.
[14] Uchihira, N., Seki, T., Kasuya, T., and Honiden, S., Program Synthesis in Prolog Based Concurrent Object Oriented Language MENDEL (in Japanese), WGSE Preprint SE-46-8, Information Processing Society of Japan, 1986.
[15] Moszkowski, B., Executing temporal logic programs, Cambridge Univ. Press, 1986.

Appendix

Keyword Count Program (WordCut, SumRep)

```
WordCut
{
    dec :{
        inpipe(stream).
        outpipe(word).
        state(inbuf![]).
    }
    meth :{
    method(stream?' ',inbuf?[]).
    method(stream?' ',inbuf?CList,inbuf![],word!Word) <-
        true|rev(CList,RCList2),name(Word,RCList).
    method(stream?C,inbuf?CList,inbuf![C|CList]).
    }
    junk:{
        rev(L1,L2):-revzap(L1,[],L2).
        revzap([X|L],L2,L3):-revzap(L,[X|L2],L3).
        revzap([],L,L).
    }
}

SumRep
{
    dec :{
        inpipe(check_data).
        outpipe(summary).
        insignal(start_rep).
        state(sumlist![]).
    }
    meth :{
        method(check_data?W,sumlist?X,sumlist!Y) <-
                true|countup_sum_list(W,X,Y).
        method(start_rep?_,sumlist?SL,summary!SL).
    }
    junk :{
        countup_sum_list(W,[],[sum(W,1)]).
        countup_sum_list(W,[sum(W,N)|SumList],[sum(W,M)|SumList]) :-
                M is N + 1.
        countup_sum_list(W,[Sum|SumList1],[Sum|SumList2]) :-
                !,countup_sum_list(W,SumList1,SumList2).
    }
}
```