Program Structure Formalizing Technology for Static Analysis

Kishimoto Yorinori[†] Itou Takako[‡] Satou Tadamasa[‡]

[†]Electrical and Computer Engineering
Nagoya Institute of Technology
Gokisochyou Nagoya
Japan

[‡]Faculty of Science and Engineering
Shimane University
1060 Nishikawatsu Matsue-shi Shimane
Japan

Abstract: - In software testing, maintenance and design skill education static analysis approaches are effective and fundamental to improving software quality and productivity. We can assess program qualities of correctness and traceability through analyzing program codes. This paper proposes a method for formalizing program structure by representing it in the form of regular expressions, where these are derived from a program by representing its scheme non-deterministically. The major feature of the method is to provide conversion formulae for representing the structures with compulsive controls, when it is well known that well-structured controls can be easily represented as regular expressions. It also discusses a way of applying the abilities to detect problems by using two program examples that have appeared in published books.

Key-Words: - Regular Expression, Program Structure, Static Analysis, Traceability

1 Introduction

In software development and education static analysis technology is an effective method of showing the correctness of logic and the traceability of the specifications in programs by analyzing program code.

As an approach to this problem we focus on the idea that a program is concerned not with code meanings (semantics) but with code sequences. We propose an idea of program structure, that is non-deterministic and basically represents the scheme of the program by neglecting every proposition in the selections and the iterations, and by removing all constructs which compose the mechanisms which become deterministic by all the propositions.

The program structure means the maximum framework of the program which is specified by the mechanisms. Different programs could be derived by applying different mechanisms to one program structure. Since the description level is higher in the abstract form than in code, it becomes close to the specifications, and is effective for analyzing the traceability of programs to their specifications and checking logics for them. In the analysis, the original structure needs to be transformed in order to compare and check the similarities or differences among the original and the transformed one, and to show the transforming process and the results.

Since a structure represented in regular expressions can be definitely described in an algebraic fashion, it is effective to illustrate the relationships among the structures in equations.

A similar idea with program structure and formalization was proposed by M.A.Jackson and J.W.Hughes. M.A.Jackson proposed a program design method by which programs can basically be formed through input and output data structures[2]. J.W.Hughes formalized Jackson's method by including definitions of both input and output data structures as labeled trees and showed the correspondence between them on a generalized sequential machine.

However, Jackson's method and its formalization is concerned with showing the traceability of the specification from the input to the output data structures, not to the program. In static program analysis for checking specification traceability, an approach would be required for formalizing not only the data structures but also the program structures.

This paper proposes a method for formalizing the program structure in a program using regular expressions. The method includes a structure with compulsive controls such as RETURNs and BREAKs, along with ones without them (well-structured). It also presents examples of practical applications that include analyzing traceability to improve them according to the results, and in discussing problems including errors in a program.

2 Formalization of Program Structure

2.1 Basic Idea

A program structure is derived from a program by neglecting any kind of proposition in it[3]. If a program is well-structured, regular expressions can be applied to represent the program construct sequence with three kinds of constructs, in which the concatenations are represented by $\cdot$(AND), the selections by $+$(OR), and the iterations by Kleene closures ($^*, ^+$).

Some program constructs could have relations to others in a program context. In this case, a program does not always keep the context in it in handling equations regularly. To resolve the problem of context linkages, the concept of ' the **program structure** and **the program mechanism**' have been introduced[3]. The structure is non-deterministic in expressing the frame of a program, while the mechanism puts a constraint on the production of a specific program from a structure. In other words, a program consists of two parts; the structure and the mechanism. Formalization of the regular expressions can be applied to the program structure alone, due to its being non-deterministic. A program structure can be identified from a program by removing the mechanisms and constructing program constructs.

2.2 Conversion Formulae for Compulsive Constructs

Compulsive controls of the RETURN constructs in a function and the BREAK constructs in an iteration construct can be used for specific controls even in a well-structured program. It is useful to describe directly such structures as with compulsive constructs. We show the formulae for converting regular expressions with RETURNs and/or BREAKs to regular ones (without them).

1) RETURNs:

Let the symbol # represent a RETURN construct in a program structure. Other program constructs are represented by alphabetic characters. A program structure can be represented by a set of words (the sequence of variables including the # symbol).

The set R is introduced, whose element is a word including #.

$$\begin{aligned} \Sigma_0 &= \{alphabetic\ character\} \quad , \Sigma_1 = \Sigma_0 \cup \{\#\} \\ R &\equiv \{x \in \Sigma_1^* | W \overset{*}{\Rightarrow} x\} \end{aligned}$$

,where W is a symbol given in the following production rule.

$$W \rightarrow f | f\#W, \qquad f \in \Sigma_0^*$$

Now, $p \in R$ can be seen as a list as follows;

$$p = ((f)|(f,\#,f,\cdots,\#,f)) = ((f_E)|(f_R,\#,f_0))$$

,where $f_E \equiv (f)$,$f_R \equiv (f)$,$f_0 \equiv (f,\#,f,\cdots,\#,f))$

The meaning of RETURNs in a function can be interpreted to neglect the sublist that starts with the first # symbol in a list. The $car()$ function for a list can be applied to represent the RETURN context.

$$car(p) = f_E | f_R$$

This can be expressed in a regular expression and simplified in form. In the program context, the function $car()$ can be omitted. The RETURN construct conversion formula is as follows;

$$p = f_E + f_R \tag{1}$$

2) Pre-tested iteration with BREAK:

It is well known that iterations with BREAK constructs can be converted into ones without them. The BREAK construct conversion formula for a pre-tested iteration can be developed using an idea similar to the RETURN case, as BREAKs terminates an iteration. Let the symbol $\rfloor$ represent a BREAK construct. The formula derived is as follows;

$$f = (f_E + f_B \rfloor f_0)^* = (f_E)^* + (f_E)^* f_B \tag{2}$$

where, $f_E \subseteq \Sigma_0^*, f_B \subseteq \Sigma_0^*, f_0 \subseteq \Sigma_2^*, \Sigma_2 = \Sigma_0 \cup \{\rfloor\}$

3) Post-tested iteration with BREAK:

$$f = (f_E + f_B \rfloor f_0)^+ = (f_E)^+ + (f_E)^* f_B \tag{3}$$

4) Continuous iteration with BREAK:

A continuous iteration can be converted into a regular one, if they have some BREAK constructs. Let the symbol $^\infty$ represent a continuous iteration instead of * or $^+$. The conversion formula is as follows;

$$f = (f_E + f_B \rfloor f_0)^\infty = (f_E)^* f_B \tag{4}$$

2.3 Compulsive Control Conversion Example

It is possible for both RETURN(#) and BREAK($\rfloor$) to exist in a regular expression. In this case, all # in an iteration must be replaced with the symbol concatenation of #$\rfloor$ at any nested level.

Figure1 shows the program with a RETURN construct and a BREAK construct in an HCP[1] chart. This program source code appeared in a book on JavaScript[2] . Program structure of this sample can be lead to the following regular expression.

$$S = (a(b\# + \varepsilon)c(\varepsilon\rfloor + \varepsilon))^*$$

[1]ISO8631

[2]Danny Goodman, "Danny Goodman's JavaScript Handbook" John Wiley & Sons, 1996

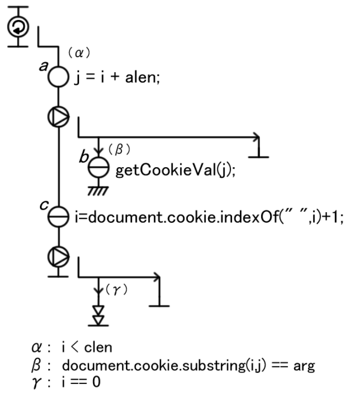

(α)
a
j = i + alen;
(β)
b
getCookieVal(j);
c
i=document.cookie.indexOf(" ",i)+1;
(γ)
α : i < clen
β : document.cookie.substring(i,j) == arg
γ : i == 0
:RETURN
:BREAK

Figure 1: Example program with RETURN and BREAK in HCP

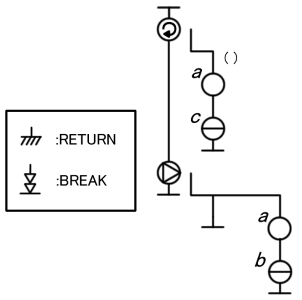


Figure 2: A rewritten regular structure in HCP

S can be developed by interpreting $\#$ as $\#\rfloor$ in an iteration.

$$\begin{aligned} S &= (a(b\# + \varepsilon)c(\varepsilon\rfloor + \varepsilon))^* \\ &= (a(b\#\rfloor + \varepsilon)c(\varepsilon\rfloor + \varepsilon))^* \end{aligned}$$

Conversion formula(2) has been developed to lead to the following equation.

$$\begin{aligned} S &= (a(b\#\rfloor + \varepsilon)c(\varepsilon\rfloor + \varepsilon))^* \\ &= (ac)^*(\varepsilon + ab + ac) \\ &= (ac)^*(\varepsilon + ab) \qquad (5) \end{aligned}$$

Figure2 could be rewritten from the structure of equation (5) by applying appropriate mechanisms.

3 Traceability Analysis Example

In testing, maintenance and design skill education it is important to know how programs have been implemented to reflect their specifications (traceability). The program structure formalizing method permits to analyze traceability, as it provides a program scheme in regular expressions that enables exact and easy handling. It can also be applied to programs in an object-oriented language. In this case, the formalization should be applicable to class methods.

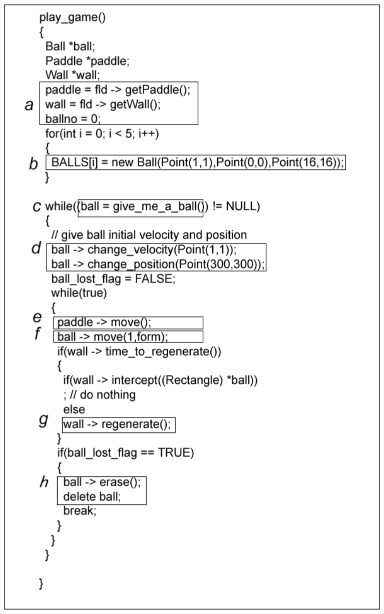
We introduce a class method as an example of analyzing traceability. This is a play_game class method for the breakout game which program in C++ appearing in a book[3].

1) Program structure formalizing

The following equation is the structure from which the code in Figure3 derived by excluding mechanisms.

$$S_P = ab^*(cd(ef((\varepsilon + g) + \varepsilon)(h\rfloor + \varepsilon))^\infty)^* \qquad (6)$$

[3]Richard C. Lee, William M. Tepfenhart, "UML and C++ A Practical Guide to Object-oriented Development," PRENTICE HALL, PP.419, 1997



```
  play_game()
  {
   Ball *ball;
   Paddle *paddle;
   Wall *wall;
   paddle = fld -> getPaddle();
a  wall = fld -> getWall();
   ballno = 0;
   for(int i = 0; i < 5; i++)
   {
b   BALLS[i] = new Ball(Point(1,1),Point(0,0),Point(16,16));
   }

c  while((ball = give_me_a_ball()) != NULL)
   {
    // give ball initial velocity and position
d   ball -> change_velocity(Point(1,1));
    ball -> change_position(Point(300,300));
    ball_lost_flag = FALSE;
    while(true)
    {
e    paddle -> move();
f    ball -> move(1,form);
     if(wall -> time_to_regenerate())
     {
      if(wall -> intercept((Rectangle) *ball))
      ; // do nothing
      else
g     wall -> regenerate();
     }
     if(ball_lost_flag == TRUE)
     {
h     ball -> erase();
      delete ball;
      break;
     }
    }
   }

  }
```

Figure 3: game_play() method

where, the variables a,b,c,d,e,f,g,h are specified in Figure3.

Equation (6) applied to conversion formula (4) leads to the following.

$$S_P = ab^*(cd(e(fg + f))^+h)^* \qquad (7)$$

2) Specification formalizing

The specifications for the class method shown in Figure4 give the following regular expression.

The system for the example specifies the two objects of a ball and a brick wall.

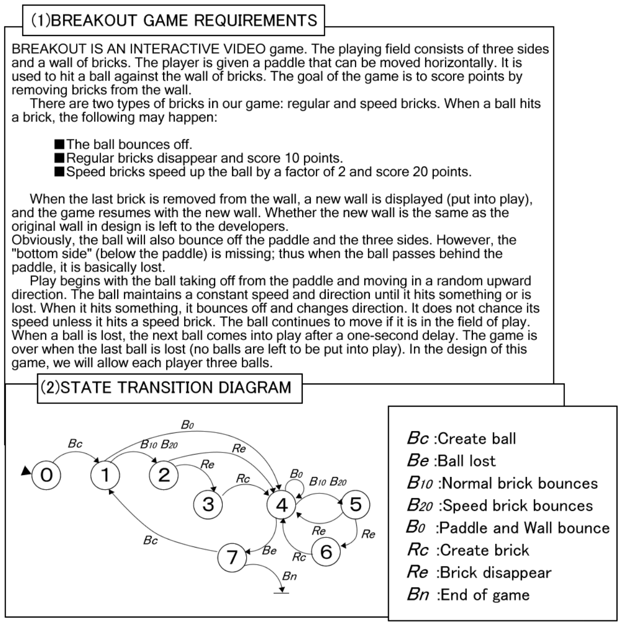
The operations for them specify the system status. The specifications for the system then can be considered as a sequential machine with the alphabet of the operations. The machine is shown in Figure4 in a state transition diagram. The diagram leads to the following regular expression.

$$\begin{aligned} &B_c((B_0 + (B_{10} + B_{20})(R_e + R_eR_c))((B_0)^* \\ &+((B_{10} + B_{20})(R_e + R_eR_c))^*)B_eB_c)^*(B_0 + (B_{10} \\ &+B_{20})(R_e + R_eR_c))((B_0)^* + ((B_{10} + B_{20})(R_e \\ &+R_eR_c))^*)B_eB_n \qquad (8) \end{aligned}$$

,where the variables B_c,B_e,B_{10},B_{20},B_0,R_c,R_e and B_n are specified in the Figure4 (2).

Equation (7) can be developed as follows;

$$\begin{aligned} S_P &= ab^*(cd(e(fg + f))^+h)^* \\ &= ab^*c(d(e(f(g + \varepsilon)))d(e(f(g + \varepsilon)))^*hc)^*d \\ &\quad (e(f(g + \varepsilon)))d(e(f(g + \varepsilon)))^*h \qquad (9) \end{aligned}$$


(1)BREAKOUT GAME REQUIREMENTS

BREAKOUT IS AN INTERACTIVE VIDEO game. The playing field consists of three sides and a wall of bricks. The player is given a paddle that can be moved horizontally. It is used to hit a ball against the wall of bricks. The goal of the game is to score points by removing bricks from the wall.

There are two types of bricks in our game: regular and speed bricks. When a ball hits a brick, the following may happen:

- ■The ball bounces off.
- ■Regular bricks disappear and score 10 points.
- ■Speed bricks speed up the ball by a factor of 2 and score 20 points.

When the last brick is removed from the wall, a new wall is displayed (put into play), and the game resumes with the new wall. Whether the new wall is the same as the original wall in design is left to the developers.
Obviously, the ball will also bounce off the paddle and the three sides. However, the "bottom side" (below the paddle) is missing; thus when the ball passes behind the paddle, it is basically lost.

Play begins with the ball taking off from the paddle and moving in a random upward direction. The ball maintains a constant speed and direction until it hits something or is lost. When it hits something, it bounces off and changes direction. It does not chance its speed unless it hits a speed brick. The ball continues to move if it is in the field of play. When a ball is lost, the next ball comes into play after a one-second delay. The game is over when the last ball is lost (no balls are left to be put into play). In the design of this game, we will allow each player three balls.

Figure 4: Specifications of breakout game

This shows the traceability analysis between equation (8) and equation (9). By observing $e(f(g+\varepsilon))$ in equation (9) it can be discovered that the f construct corresponds to B_10, B_20, and B_0 and it can be identified that the move() method in the f construct is implemented in polymorphism.

Correspondences of equation (8) to equation (9) are listed in Table 1 which identifies how the program structure is reflected in its specifications in the equations.

Table
1. Correspondences the structure to the specifications

Specifications(8)	Program structure(9)
B_c	c
$(B_0)^*, ((B_{10}+B_{20})R_e)^*$	f
R_c	g
B_e	h

4 Traceability Improvement Example

We can analyze traceability not only as shown in chapter 3, but also identify some difficulties, including errors, and improve traceability as well. To show these we introduce an example of a procedure that converts and arranges the command line from a full line appearing in the book[4].

Figure6 shows an HCP chart which has been rewritten from the pseudo code given in Figure5. Although the input command line syntax is not clearly specified even in the header comment part of Figure5, an interpretation of the code in constructing the Is_command flag-mechanism leads to the input syntax. It can be clarified by adding the following specifications interpreted from the code and the comments in Figure7.

```
  /*Read_Command_Line(Command_Length);
  This procedure reads a full line of command, eliminates all of the blank spaces, converts
  lower case letters into upper case letters, and stores the command line in array
  Command_Line. It also identifies the end of command line by placing a "$" as the last
  character in array Command_Line. The argument Command_Length will return a 1 if the
  command is in short form. If the command is in long form, this argument returns the
  length of the command. A value other than 1 means the command is in long form.
  Regardless of the length of the command entered (short or long), the first 8 positions
  in array Command_Line are set aside to store the command word.
  */

  Local Declarations
    Define I as integer
    Define Ch as character
    Define Is_Command as Boolean
  Begin Read_Command_Line
    Set I to 0 /*character count*/
    Set Is_Command to True
    Set Command_Line to a Black line
    While leading character is blank
g     Read(Ch)
    End-while
    While I < 80 and more characters on the line
g     Read(Ch);
      If Ch is lower case then
h       Change it to upper case
      End-if
      If Ch is blank and Is_Command is True then
i       Set Command_Length to I
        Set Is_Command to False
k       Set I to 8 /*the longest command*/
      End-if
      If Ch is not blank then
        Increment I
m       Set Command_Line[I] to Ch;
      End-if
    End-while
    If less than 8 characters on the line then
k     Set I to 8
    End-if
n   Set Command_Line[I+1] to '$'
    Finish the current line
  End Read_Com_Line
```

Figure 5: Pseudo code of the Read_Command_Line()

Let a command line be specified as a character string on the alphabet $\{B,C\}$ with the following syntax, where B is a blank character and C is any character except a blank.

$$L \equiv B^*C^+(B^+C^*)^* \tag{10}$$

The structure of the procedure in Figure5 can be shown in the following equation derived by excluding mechanisms.

$$P \equiv g^*(g(h+\varepsilon)(ik+\varepsilon)(m+\varepsilon))^*(k+\varepsilon)n \tag{11}$$

where the variables g,h,i,k,m,n are specified in Figure5.

1)Command head character problem

There are two problems. Firstly there is the possibility of not reading any first character in an input

[4]Ali Behforooz, Frederick J. Hudson, "Software Engineering Fundamentals," Oxford University Press, Cp.11, PP.345, 1996.

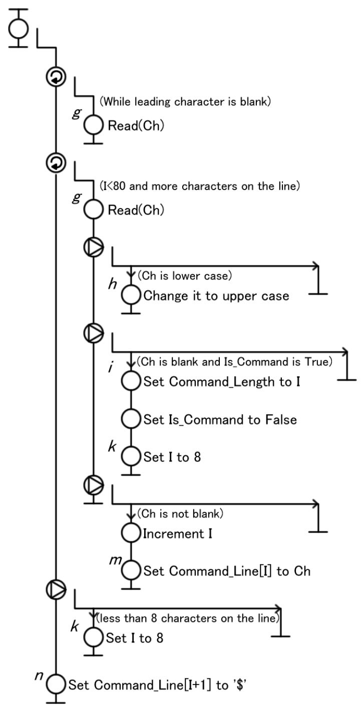

(While leading character is blank)
g
Read(Ch)
(I<80 and more characters on the line)
g
Read(Ch)
(Ch is lower case)
h
Change it to upper case
(Ch is blank and Is_Command is True)
i
Set Command_Length to I
Set Is_Command to False
k
Set I to 8
(Ch is not blank)
Increment I
m
Set Command_Line[I] to Ch
(less than 8 characters on the line)
k
Set I to 8
n
Set Command_Line[I+1] to '$'

Figure 6: Read_Command_Line() in HCP

1) An input has more than one character excluding an empty file and the lines have no character.
2) When the condition 'not more characters on the line' is true, a flag (such as an empty input flag) is set to 'on'.
3) The command-word length is in the range of 1 to 8.

Figure 7: Specifications for the Read_Command_Line()

except a blank, if the pseudo code 'while leading character is blank' is conditioned in the 'Ch == " "'. The term g^* in equation (11) could not be implemented because g is concerned with the context of the iteration. The proposition of the iteration depends on the processing of g. The implementation logic of this program is g^+ because checking the proposition occurs after the g construct (read a character).

We must assume 'Set Ch to blank' is an initialized value of Ch, or change the iteration type to a post-tested one. In any case g^* in equation (11) should be changed to g^+.

To clarify the discussion a variable $\alpha \equiv (h+\varepsilon)(ik+\varepsilon)(m+\varepsilon)$ is introduced.

$$P' \equiv g^+(g\alpha)^*(k+\varepsilon)n \qquad (12)$$

Secondly it is possible to discover the lack of a command head character by observing g in equation (12). The term $g^+(g(\alpha))^*$ means that there is the possibility of overwriting a head character with a second. The structure should be modified, such as the following, to correct the error.

$$P'' \quad = \quad g^+(\alpha g)^*\alpha(k+\varepsilon)n$$

2)Placement of ' \$ ' problem

We now observe the '\$' placement processing. The placement is determined by the program constructs concerned with the variable I in the P'' structure. It can identify the constructs m,k, and n. These constructs are included in the following structure by neglecting program constructs unrelated to them.

$$(km)^*(\varepsilon+k)n = ((km)^* + (km)^*k)n$$

k in $(km)^*$ is redundant, as the m^* provides the increment to I. $(km)^*$ in $(km)^*k$ is similarly redundant. k in $(ik+\varepsilon)$ can be omitted to give $(i+\varepsilon)$.

3)Traceability analysis

To discover what specifications lead to the program structure, how the input specifications are reflected on it (traceability) are analyzed.

The following equations can be introduced after correcting the two problems ((1) and (2) above).

$$P''' = g^+(\alpha'(\varepsilon\rfloor+\varepsilon)g)^\infty(k+\varepsilon)n$$

,where $\alpha' \equiv (h+\varepsilon)(i+\varepsilon)(m+\varepsilon)$

Applying the conversion formula (4) to P''' leads to the following.

$$P''' = g^*(g\alpha')^+(k+\varepsilon)n$$

Namely,

$$= g^*(g(h+\varepsilon)(i+\varepsilon)(m+\varepsilon))^+(k+\varepsilon)n$$

Replacement of the following variables is introduced to simplify the discussion.

$v = h+\varepsilon,\ b = \varepsilon+i+m,\ c = im,\ f = (k+\varepsilon)n$

$$P''' = g^*(gvb+gvc)(gvb+gvc)^*f \qquad (13)$$

Equation (10) can be converted as follows;

$$\begin{aligned} L &\equiv B^*C^+(B^+C^*)^* \\ &= B^*C(C^*)^*(B^+(C^*)^*)^* \\ &= B^*C(C^*+B^+)^* \\ &= B^*C(C+B)^* \qquad (14) \end{aligned}$$

Table 2 shows the correspondences by comparing equation (14) and equation (13). It is concluded that the construction basis of the program structure depends on the input string specifications extended from the original $C^*(B^+C^*)^*$ to $(C+B)^*$.

Table 2. Correspondences of the structure to the specifications

Specifications(14)	Program structure(13)	Description
B^*	g^*	
C	$gvb + gvc$	for first character of the command
C	gvc	for other characters of the command and other characters except blank
B	gvb	

4)Traceability improvement

Next, we try to arrange the program structure to improve traceability, and to check whether it is possible to implement a program from an arranged structure, since it is important to keep traceability for maintainance.

It is more natural to compose a structure to follow the word, not character sequence in the specifications. Program sub-structures can be considered to be composed of a command word, other words (such as parameters or switches), and blank strings. The word sequence is specified as follows;

$$S_B S_C (S_B S_P)^*$$

S_C:	command word processing
S_P:	word except command processing
S_B:	blank processing

A structure P'''' following the word sequence is introduced as follows;

$$P'''' = S_{B1} S_C (S_{B2} S_P)^* P_\$$$

This requires two types of blank string B_1 and B_2. B_1 is the leading input line, B_2 exists between S_C and S_P or among S_Ps. $P_\$$ is a construct for '$' placement.

The above words can be processed with program constructs in P''' as follows;

$$\begin{aligned} S_{B1} &= (b_1 g)^+ \quad , b_1 \ is \ the \ process \ for \ leading \\ & \qquad\qquad blank \ characters (= \varepsilon) \\ S_C &= ((h+\varepsilon)mg)^+ i \\ S_P &= ((h+\varepsilon)mg)^* \\ S_{B2} &= (b_1 g)^* \\ P_\$ &= (k+\varepsilon)n \end{aligned}$$

Eventually,

$$\begin{aligned} P'''' &= (b_1 g)^+((h+\varepsilon)mg)^+ i((b_1 g)^*((h \\ & \quad +\varepsilon)mg)^*)^*(k+\varepsilon)n \qquad (15) \end{aligned}$$

If a program can be implemented to P'''', this means there would be an improvement in traceability. It is not difficult to write a program with the above structure. The traceability of this one has been improved by comparing with the original.

5 Concluding Remarks

A method that permits the formalization of program structures in regular expressions which represent the non-deterministic scheme of a program as well-structured, along with compulsive controls such as RETURN constructs in a function and BREAK constructs in any type of iteration, has been proposed.

Also presented is an application of the method to sample programs appearing in books which has made it clear that it has effective application in practical fields. These are to analyze program traceability; that is to explain how the program structures in a program reflect the specifications, to improve traceability modifying structures to follow the specifications, and to discover and discuss difficulties in a program, including errors.

References:

[1] M.A.Jackson, "Principles of Program Design", Academic Press, 1975

[2] J.W.Hughes, "A Formalization and Explication of the Michael Jackson Method of Program Design", SOFTWARE-PRACTICE AND EXPRENCE Vol.9 PP.192-201,1979

[3] Satou Tadamasa, "A Formalization by S-Algebra for Program Algorithm Described in HCP-Charting,",IPSJ Vol.27 No.6 ,1986 (in Japanese)

[4] Satou Tadamasa, "Universal Form for Program Iteration Structure", ICCIT2000 Proceedings PP.240-243 ,2001

[5] Satou Tadamasa, Kishimoto Yorinori, "An Analysis Method of Program Traceability to the Specification through Program Construct Formalization", WSEAS2002 PP.19-24, 2002