A system to help students analyze errors in their programs by supporting programming comprehension in assembly programming exercises

Yuichiro Tateiwa
Graduate School of Engineering
Nagoya Institute of Technology
Nagoya, Japan
tateiwa@nitech.ac.jp

Daisuke Yamamoto
daisuke@nitech.ac.jp

Naohisa Takahashi
naohisa@nitech.ac.jp

***Abstract*—There are several students who give up exercises because they cannot specify their program errors to fix. We considered the reason were the following. One is that the students do not have enough comprehension of their programs – questions ask their understanding of control structure, computer resource control, and behavior. Another is that procedures to specify program errors are complex because an assembly program has a lot of instructions. Furthermore, oversight, which is caused by misunderstanding questions and checking a lot of items, is also the causes. The purpose of this study is to develop a system which generates expression for specifying program errors by helping students understand their program comprehension. The features on realizing the system are making use of chunks, dynamic backward slices, and correct answer samples. We conclude that the expression is helpful to specify program errors according to an evaluation experience.**

Keywords-programming; assembly language; program slice; chunk;

I. Introduction

The "Systems Programming" course offered by the Department of Computer Science, Nagoya Institute of Technology, aims to help students understand hardware activities that occur in response to application software requests for computer resource control (e.g., controlling registers and main memory) and control structures (sequencing, selection, iteration, and function/procedure). Therefore, the class includes an assembly programming exercise in which students translate high-level-language (e.g., C) programs, whose activities are regarded as application software requests, into low-level-language programs (e.g., assembly language CASLL-II [1], whose activities are regarded as hardware activities.

In class, students solve exercises on structured programming using the above-mentioned control structure. Questions in the exercise include the requirements of program behavior, computer resource control, and control structure in the form of text and C programs. Students' answers (hereafter, "answer program") are considered correct if they do not contain all of the following error types.

・Behavior error: answer program does not behave according to the requirements of questions.

・Computer resource control error: computer resources are not used according to the requirements of questions.

・Control structure error: control structure is not designed according to the requirements of questions.

We developed a programming exercise system that can automatically detect these error types [2]. This enables students to immediately confirm whether their answers are correct. First, the system obtains two detailed "trace data" (i.e., a sequence of pairs of instructions that are executed stepwise and the computer resources that are updated by the execution) by the stepwise execution of the answer program and the correct answer program (hereafter, simply "correct program") with test cases. Each pair of trace data for a given step in the sequence is called "step data". Next, the system extracts characteristic points (e.g., order of label appearance, variable values, relation between locations of instructions) from both trace data. If differences are detected between corresponding characteristic points, the system judges that the answer includes errors. After the evaluation is completed, the results are displayed to the student; if errors are present, the test case is also displayed. A student whose answers are incorrect is expected to try to specify the causes of errors by using her/his program, question, and test case. S/he should analyze control structures, trace the dependency of computer resource control, simulate the test case behavior, and so on. S/he should consider whether these satisfy the requirements of the question. If not, s/he should specify error instructions in her/his program.

However, several students are unable to complete these tasks. We believe that some students lack an understanding of the control structure, computer resource control, and program behavior (we collectively term this as "program comprehension"). Others find the procedures for specifying causes of errors too complex because an assembly program contains many instructions. Finally, some problems are due to oversight stemming from, say, misunderstanding of program requirements and the need to check many items.

In this study, we develop a function that generates expressions that help students specify causes for their errors by supporting their program comprehension. We call these expressions "assistance expressions." The function detects errors in answer programs and classifies them as 1) control structure errors or 2) behavior errors and computer resource control errors, and it generates assistance expressions for each error.

In order to implement this function, we use "chunks," "dynamic backward slices," and correct answers. A "chunk" is a meaningful block (sequential elements). It is easier to understand and memorize programs when they are expressed as a sequence of chunks. A "dynamic backward slice," a type of program slice, is a sequence of instructions that influences variable v, which is defined at time r when an instruction is executed, when executing a program with input arguments x. Hereafter, we call time r the "execution point" and the triple of x, r, and v, the "slicing criterion (x, r, v)." The dynamic backward slice at an execution point where variables differ between the answer and the correct program includes the causes of errors (inclusion of error instructions and lack of necessary instructions). Therefore, its use can help students specify the causes of errors.

The function generates the following three types of assistance expressions.

(1) Chunk expression of programs: It is important to read and understand programs in order to specify the causes of errors. However, as mentioned above, assembly programs are difficult to read and understand. We solve this problem by explicitly expressing control structures in a program and meaningful instruction sequences in control structures by using chunks. A chunk containing instructions is called a "static chunk." Expressing programs using static chunks helps in understanding control constructions and specifying causes of errors, in addition to reading programs. For example, although instructions of "readout arguments" are connoted by the control structure "sequence," it is possible to show a composition of the control structure "sequence" to students by defining a static chunk "readout arguments." In addition, it is possible to show the control structures that contain errors to students by defining a static chunk of error implementations. This function generates an expression that is a static chunk sequence of both an answer and a correct program, and the expression includes instructions that constitute the chunks of the answer program. The aim of the expression is to help students notice the differences in control structures and specify the causative instructions.

(2) Chunk expression of trace data: Trace data is useful for specifying the causes of behavior error and computer resource control errors. However, it is difficult to read and understand because it is large in amount, and it is troublesome to match to a program because of the use of different expressions. We solve this problem by expressing trace data using chunks that are related to static chunks. Hereafter, a chunk containing trace data is called a "dynamic chunk." This function generates an expression that is a dynamic chunk sequence of both an answer and a correct program to help students notice the difference between the two.

(3) Projection expression of error steps: Some instructions, called as an "error instruction sequence," contain important clues for specifying the causes of program behavior error and computer resource control errors, and their execution results influence the difference between the trace data of an answer and a correct program. A student specifies the causes of errors in the answer program by confirming the relationships between the error instruction sequence and the remaining instructions and by comparing the execution results of the answer and the correct program. However, (1) and (2) are not suitable for such procedures. The former does not show error instruction sequences and their execution results to a student, and therefore, it is difficult to compare an execution result between the answer and the correct program. The latter does not show all instructions of the answer program, and therefore, it is difficult to confirm the relationships between instructions in the answer program. Accordingly, we try to solve the problems by computing an error instruction sequence and expressing it and its execution result for the answer program. The function generates an expression that is based on (1) with an error instruction sequence of an answer program, the execution results of it and a correct program.

II. Related works

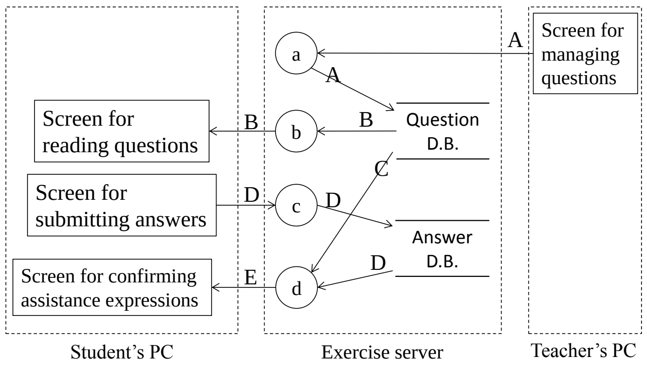
A "program slice" is a set of instructions that influences a certain instruction in a program [3][4][5]. Program slices are used for program debugging and program comprehension. A dynamic backward slice in this study is characterized by its slicing criterion which is a point that causes difference of program behavior and computer resource control between an answer and a correct program. Namely, its feature is to use not only an answer program but also a correct program. Using this criterion, we can compute a slice that includes the causes of errors.

Static chunks are calculated by pattern matching between a program and a pattern that defines a rule of a static chunk. As a related study that uses pattern matching in assembly programs, W.Kozaczynski et al. proposed the replacement of frequently used instruction sequences with simple expressions and comments for easy readability and understandability [6]. We, however, propose a method to generate information that simplifies the reading and understanding of programs and trace data and the correspondence between programs and trace data using static chunks.

III. Function for generating assistance expressions

A. *Placement in our programming exercise system*

Fig. 1 shows the structure of our system. The exercise system consists of an exercise server on a machine, and web browsers for each student and teacher on their PCs. And the machine and the PCs are connected to the Internet. A student receives a question from the exercise server ("b"), composes an answer to the question, and submits it to the server ("c"). The server detects errors in the answer, and generates assistance expressions that depends on the error type by using the answer program, a correct program, test cases, "static chunk conditions," and "evaluation item sets" ("d"). A "static chunk condition" is a condition for extracting


Screen for reading questions
Screen for submitting answers
Screen for confirming assistance expressions
a
b
c
d
A
B
C
D
E
Question D.B.
Answer D.B.
Screen for managing questions
Student's PC
Exercise server
Teacher's PC

a: Register questions
b: Get questions
c: Submit answers
d: Generate assistance expressions

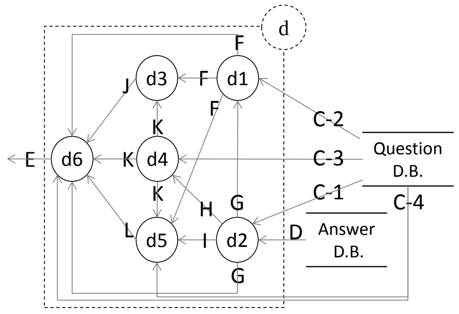
A: questions, correct programs, test cases, static chunk conditions, evaluation item sets
B: questions
C: 1) correct programs, 2) test cases, 3) static chunk conditions, 4) evaluation item sets
D: answers programs
E: assistance expressions

Figure 1. System structure

instructions from a program. An "evaluation item set" is a set of input/output variables in static chunks that are compared between the answer and the correct program. The student confirms the true/false judgment of her/his answer; if the answer is false, s/he corrects it on a Screen using assistance expressions. A teacher registers questions, correct programs, test cases, static chunk conditions, and evaluation item sets with the exercise server before the student starts the exercises ("a").

B. Function structure

Fig. 2 shows the structure of the function for generating assistance expressions in "d" of Fig. 1. "d2" extracts instructions of answer and correct programs and converts their formats to that shown in Fig. 3. The function then extracts bodies of "routines" from both converted programs (Section III.C.1). Henceforth, a "routine" is a main routine or a subroutine that is a sequence of instructions from "start" to "end." In addition, the body of a routine (called a routine body) is a sequence that consists of machine instructions and macro instructions. "d1" generates each step data by stepwise execution of an answer and a correct program with test cases (Section III.C.2). "d4" extracts static chunks of an answer and a correct program by comparing instructions of a routine body with static chunk conditions (Section III.C.3). "d3" extracts dynamic chunks by comparing stepwise executed instructions with instructions that consist of static chunks (Section III.C.4). "d5" compares input/output variables between an answer and a correct program in the order of executed instructions. In the comparison, the variables are selected in accordance with an evaluation item set, and their values are computed using step data. When "d5" detects a difference in the comparison, it computes a dynamic backward slice of the variable that causes the difference, and it regards the slice as an error instruction



C-1, C-2, C-3, C-4, D, E: refer to Fig. 1
F: step data of answer programs and correct programs
G: converted answer programs and converted correct programs
H: converted answer programs and correct programs, descriptors for routine bodies of answer programs and correct programs
I: converted answer program
J: descriptors for dynamic chunks of answer programs and correct programs
K: types and descriptors of static chunks in answer programs, types and descriptors of static chunks in correct programs
L: descriptors for error instruction sequences
d1: generate step data
d2: convert programs and extract routine bodies
d3: extract dynamic chunks
d4: extract static chunks
d5: extract error instruction sequences
d6: generate assistance expressions

Figure 2. Structure of function for generating assistance expressions

sequence (Section III.C.5). "d6" classifies an answer program as being correct or as containing a control structure error or behavior/computer resource control error, and it generates assistance expressions (Section III.C.6).

Hereafter, this paper describes a new data type using a structure in the C language. However, "struct" is omitted in the member and variable declarations. For example, a data structure X with int type members a and b is described as "struct X {int a; int b;}." A variable z of data type X is described as "X z;." A member a of z can be referred to by "z.a."

C. Implementation methods

1) Convert programs and extract routine bodies

Answer and correct programs conform to the CASL-II grammar. We have extended this grammar by adding operation codes SSP and LSP, which save and load a stack pointer, respectively. These are necessary for the class to implement a general procedure of a function call, which is implemented by stack frame operations in most assembly languages such as GNU assembly language.

The data structure of a program in our algorithms is a character array. Instructions in a program are converted from their original formats into the one shown in Fig. 3, and then stored in the character array (e.g., Fig. 4). Henceforth, "<>" indicates that the elements therein can be omitted, and "{}" indicates that elements therein are necessary. We call

```
<label>{space}{operation code}<{space}{operand1}<,operand2><,operand3>>\n
```

Figure 3. Format of an instruction in our algorithms

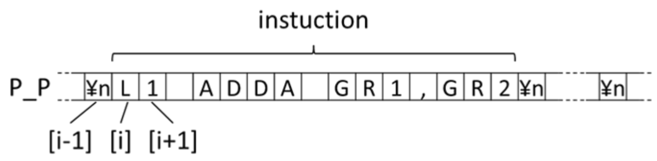


Figure 4. An example of a variable P_P that contains a program

character strings that are located at the label, operation code, operand1, operand2, and operand3 identifiers. Variables P_P and A_P store the answer program and correct program, respectively.

A routine is a main routine or a subroutine that is a sequence of instructions from start to end. In addition, the body of a routine (called a routine body) is a sequence that consists of machine instructions and macro instructions. Routine bodies are extracted from converted answer and correct programs. The data structure of a routine body in our algorithms is of the Block type. Block type is designed for pointing to a sub array "sub_array" in an array "array," and it is defined as "struct Block{int s; int e;}." Members s and e are the indexes of elements in "array" that are respectively the first and last elements of "sub_array." P_R and A_R, which are Block-type array variables, respectively store the routine bodies of the answer program and correct program in the order found in the programs. For example, a character string that is from P_P[P_R[0].s] to P_P[P_R[0].e] is the instruction sequence of the first routine body in P_P (Fig. 5).

2) Generate step data

A step datum is a result that is generated by stepwise executing a program with a test case as input. It consists of an executed instruction, names of computer resources that are updated by executing the instruction, and their values. The data structure of a computer resource in our algorithms is defined as "struct CmpRes{char[] n;char[] v;}"; members n and v are the first addresses of character arrays that stores a computer resource name and computer resource value, respectively . The data structure of a step datum is defined as "struct Step{int i;CmpRes[] cr;}"; member cr is the first address of a list that stores computer resources that are updated by stepwise execution on an instruction whose first character is the i+1st character in a program. Step type arrays P_S and A_S respectively store step data of the answer and the correct program in order of stepwise execution. For example, P_S[k] is a step datum of the k+1th stepwise execution in answer program P_P. P_P[P_S[k].i] is the first character that is an executed instruction; the character string whose first address is pointed to by P_S[k].cr[0].n is a

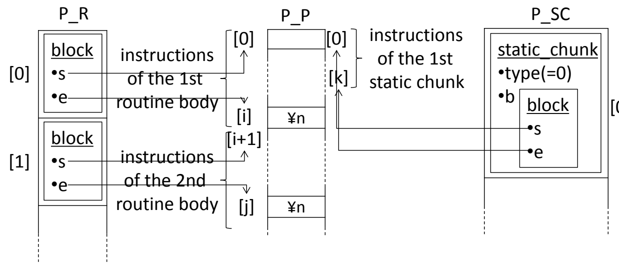

P_R
P_P
P_SC
[0]
block
•s
•e
instructions of the 1st routine body
[1]
block
•s
•e
instructions of the 2nd routine body
[0]
[i]
[i+1]
[j]
¥n
¥n
[0]
[k]
instructions of the 1st static chunk
static_chunk
•type(=0)
•b
block
•s
•e
[0]

Figure 5. An example of the arrays P_P, P_R, and P_SC

computer resource name that is the first updated by the executed instruction; and P_S[k].cr[0].v points to the character string of its value (Fig. 6).

3) Extract static chunks

As mentioned in Section I, static chunks help in reading/understanding programs, understanding control structures, and specifying causes of control structure errors. The control structure in the class includes the sequence, selection, repetition, and function/procedure. To extract other static chunks, it is necessary to define new static chunk conditions and register them in the system.

A static chunk is an instruction sequence that has a meaning in toto. The data structure of a static chunk in our algorithms is defined as "struct SChunk{int t; struct Block b;}"; member t is a type (Tabel 1), and member b is a character string that is from the b.s+1th character to the b.e+1th character, and this character string is an instruction sequence that constitutes this static chunk. SChunk type arrays P_SC and A_SC respectively store static chunks of an answer and a correct program in the order of extraction. For example, a character string that is from P_P[P_SC[i].b.s] to P_P[P_SC[i].b.e] is an instruction sequence that consists of the i+1th static chunk that is extracted from an answer program. P_SC[0] in Fig. 5 is extracted first; its type is a sequence according to P_SC[0].type=0, and it consists of a character string that is from P_P[P_SC[0].b.s] to P_P[P_SC[0].b.e].

The data structure of a static chunk condition in our algorithms is a pair of a static chunk type and a condition for extracting instruction sequences (called the "instruction sequence condition"). Because the identifier and number of instructions depend on the questions, it is difficult to define instruction sequence conditions using only instructions of CASL-II; it is necessary to define many conditions. Regular expressions and pattern matching are effective ways to solve this problem. However, it is necessary to consider the following points.

Requirement 1: It is necessary to extract instruction sequences that include arbitrary and same identifiers at multiple points of the sequence. For example, in an

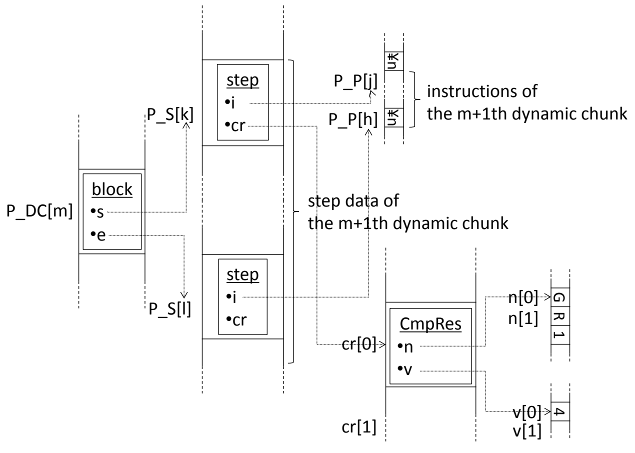


Figure 6. An example of the arrays P_S and P_DC

instruction sequence selection, the operand of the operation code branch and label of the branch destination are an arbitrary and same identifier.

Requirement 2: It is necessary to extract a character string that is matched to a particular part in a regular expression. This is used for specifying a character string, extraction target, by the character before and after it. For example, Fig. 7-a) shows that the instruction sequence from line 1–4 is the chunk selection. The tail of the chunk is characterized by an instruction immediately in front of the branch destination (line 5). In such cases, the first 5 lines are extracted, and then, line 5 is removed.

We adopted Perl, whose regular expressions enable a character string extracted by a regular expression to be referred to from the back of the expression itself (for requirement 1). It is possible to refer to a character string that is matched to a group by group numbers after completing the matching (for requirement 2). Therefore, an instruction sequence condition consists of a regular expression and a group number.

Fig. 7-b) shows an example of an instruction sequence condition for the chunk selection. It expresses 1 line character string by dividing it into multiple lines owing to space limitations. The first group is a regular expression from the left parenthesis in line 1 to the right parenthesis in line 4, and it is designed to extract an instruction sequence of a chunk selection. The second group is "(\w+)" in line 2, and it is referred to by "\2" in lines 3 and 5. When it applies this regular expression to the instruction sequence in Fig. 7-a), because line 2 in Fig. 7-a) matches the line 2 in Fig. 7-b), a character string that matches the second group is considered as "L1," and "\2" in lines 3 and 5 is specified as "L1." In addition, the character string from lines 1–4 in Fig. 7-a) matches the first group, and it is extracted as the instruction sequence of a chunk selection.

The data structure of a static chunk condition in our algorithms is defined as "struct ChunkCond{int t;char[] ptn; int g;}"; member t is a type of a static chunk (Table 1), member p is the first address of a character string of an instruction sequence, and member g is a group number for designating a character string in the extraction. A ChunkCond type array CC stores static chunk conditions.

Fig. 8 shows an algorithm for extracting static chunks. The program in the class conforms to the rule of structured programming, and it is not allowed to jump between routines by operation code jump. Therefore, we developed a simple algorithm that extracts static chunks from every routine because there is no chunk through multiple routines. A function "int N (T[] array1)" returns the number of elements in arbitrary type T array array1. A function "void merge(T[] array1, T[] array2)" merges the elements of arbitrary type T arrays array1 and array2 so that the elements of array2 are appended to the end of array1. A function "void add(T[] array1, T elem1)" appends arbitrary type T elem1 to the end of arbitrary type T array array1. When P_P and P_R, or A_P and A_R, are respectively stored in P and R and static chunk conditions are stored in CC, following which static_chunk(P, R, CC) is executed, an SChunk type variable that contains static chunks is obtained.

TABLE I. STATIC CHUNKS

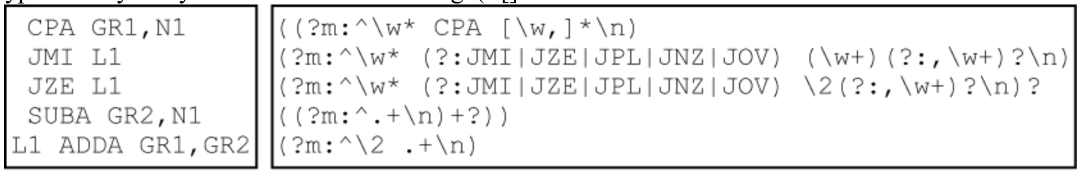
type	chunk name
0	sequence processing
1	start processing of function
2	start processing of function (with errors)
3	end processing of function
4	end processing of function (with errors)
5	readout processing of arguments
6	readout processing of arguments (with errors)
7	repetition processing
8	selection processing

4) *Extract dynamic chunks*

A dynamic chunk is a sequence of step data that is generated by single stepwise execution of instructions from the start to the end of a static chunk. The data structure of a dynamic chunk in our algorithms is Block type; members s and e in P_S and A_S indicate that step data from P_S[s] to P_S[e] or A_S[s] to A_S[e] is included in the extracted dynamic chunk. Dynamic chunks of an answer and a correct program are stored in Block-type arrays P_DC and A_DC, respectively, in order of extraction. For example, a Step-type array from P_S[P_DC[m].s] to P_S[P_DC[m].e] is a step data instruction of the m+1th dynamic chunks that are extracted. Fig. 9 shows an algorithm for extracting dynamic chunks. When P_SC and P_S, or A_SC and A_S, are respectively stored in SC and S, following which dynamic_chunk(SC, S) is executed, a Block-type array that contains dynamic chunks is obtained.

5) *Extract error instruction sequences*

An "error instruction sequence" is a sequence of instructions that affects the difference between the trace data of an answer and a correct program. We call such instructions "candidates for error instructions." Error instruction sequences and their execution results are important clues for specifying the causes of program behavior errors and computer resource control errors. Trace data of an answer and a correct program are compared based on evaluation item sets. An evaluation item set is a designation of computer resources that are compared



```
 CPA GR1,N1
 JMI L1
 JZE L1
 SUBA GR2,N1
L1 ADDA GR1,GR2
```

a) Instructions

```
((?m:^\w* CPA [\w,]*\n)
(?m:^\w* (?:JMI|JZE|JPL|JNZ|JOV) (\w+)(?:,\w+)?\n)
(?m:^\w* (?:JMI|JZE|JPL|JNZ|JOV) \2(?:,\w+)?\n)?
((?m:^.+\n)+?))
(?m:^\2 .+\n)
```

b) Instruction sequence condition

Figure 7. An example of instructions and an instruction sequence condition for selection processing

```
SChunk[] static_chunk(char[] P,Block[] R, ChunkCond[] CC){
  SChunk SC[0];
  for(int i=0;i<N(R);i++)
    merge(SC, f(P,R[i],CC,0));
  return SC;}
SChunk[] f(char[] P,Block b,ChunkCond[] CC,int cc_i){
  SChunk SC[0];
  if(cc_i == N(CC)){
    SChunk sc = {0, b};
    add(SC, sc);
    return SC;
  }
  Block m = match(P,b,CC[cc_i]);
  if(m.s == -1 && m.e == -1)
    merge(SC,f(P,b,CC,cc_i+1));
  SChunk sc = {CC[cc_i].t, m};
  add(SC, sc);
    Block next = {m.e+1, b.e};
    Block prev = {b.s, m.s-1};
      if(b.s == m.s && m.e < b.e){
        merge(SC,f(P,next,CC,cc_i));
      }else if(s < m.s && m.e < e){
        merge(SC,f(P,prev,CC,cc_i+1));
        merge(SC,f(P,next,CC,cc_i));
      }else if(s < m.s && m.e == e){
        merge(SC,f(P,prev,CC,cc_i+1));
      }
      return SC;}
Block match(char[] P, Block b, ChunkCond cc){
      Block m = {-1,-1};
    Extract character string ss that matches the cc.g-th group in
cc.ptn from the character string that is from P[b.s] to P[b.e];
    if (ss exists)
    m.s=x and m.e=y on the condition that ss is the character
string that is from P[x] to P[y], and b.s<=x and y<=b.e
    return m;}
```

Figure 8. An algorithm for extracting static chunks

between an answer and a correct program in the input or output.

The data structure of an error instruction sequence in our algorithms is a Block-type array. An element e of the array indicates a candidate for error instruction, which is a character string from the e.s+1th to the e.e+1th character in a program. An error instruction sequence is stored in the order extracted from a dynamic backward slice. The data structure of an evaluation item set in our algorithms is a char-type four-dimensional array. CK is designed for holding evaluation item sets; CK[a][0][c] points to the first address of the c+1th variable name that is compared between an answer and a correct program in the input of the a+1th static chunk that is extracted, such as P_SC[a] and A_SC[a]. CK[a][1][c] points to the first address of the c+1th variable name that is compared between an answer and a correct program in the output of the a+1th static chunk that is extracted, such as P_SC[a] and A_SC[a].

Fig. 10 shows an algorithm for extracting an error instruction sequence. A function “error_ins” first judges, in the order of executing instructions, whether an answer

```
Block[] dynamic_chunk(SChunk[] SC,Step[] S){
  Block DC[0];
  for(int j=0;j<N(SC);j++){
    for(int i=0;i<N(S);i++){
      if(S[i].i==SC[j].b.s){
        Block dc = {i, -1};
        while((i+1<N(S)) &&
             (S[i].i<S[i+1].i) &&
             (S[i+1].i<=boi(SC[j].b.e))){
          i++;
        }
        dc.e = i;
        add(DC, make_array(dc));
      }}}
  return DC;
}
```

Figure 9. An algorithm for extracting dynamic chunks

```
Block[] error_ins(char[] P_P, char[] A_P, SChunk[]
P_SC,SChunk[] A_SC,Step[] P_S,Step[] A_S,char [][][][] CK){
for(int i=0,k=0;i<N(A_S)||k<N(P_S);){
  Using j and l, which are A_S[i].i == A_SC[j].b.s and P_S[k].i
  == P_SC[l].b.s, for(int m=0;m<N(CK[j][0][m];m++){
    if(value(P_S,k-1,CK[j][0][m]) differs from value(A_S,i-
    1,CK[j][0][m])){
      int s = def(P_S,k-1,CK[j][0][m]);
      if(s!=-1){
        b[w]={x,y} on the condition that the instruction that is
        from P_P[x] to P_P[y] is equal to the w+1th instruction
        in a dynamic backward slice on slicing criterion=(s,
        CK[j][0][m], the test case);
        return b;
      }else{
        b[w]={x,y} on the condition that the instruction that is
        from P_P[x] to P_P[y] is equal to the w+1th instruction
        from the first instruction in P_SC[l];
        return b;
  }}}
  Using j and l, which are A_S[i].i == boi(A_P,A_SC[j].b.e) and
  P_S[k].i == boi(P_P,P_SC[l].b.e), for(int m=0;
  m<N(CK[j][1][m]; m++){
    if(value(P_S,k-1,CK[j][1][m]) differs from value(A_S,i-
    1,CK[j][1][m])){
      int s = def(P_S,k-1,CK[j][1][m]);
      if(s!=-1){
        b[w]={x,y} on the condition that the instruction that is
        from P_P[x] to P_P[y] is equal to the w+1th instruction
        in a dynamic backward slice on slicing criterion=(s,
        CK[j][1][m], the test case);
        return b;
       }else{
        b[w]={x,y} on the condition that the instruction that is
        from P_P[x] to P_P[y] is equal to the w-th instruction
        from the last instruction in P_SC[l];
        return b;
  }}}
  if(i<N(P_DC)) i++;
  if(j<N(A_DC)) j++;
}}
int def(Step[] S,int i,char[] n){
  for(;0<=i;i--)
    for(int j=0;j<N(S[i].CR);j++)
      if(the character string that is pointed to by S[i].CR[j].n is
      equal to the character string that is pointed to by n)
      return i;
 return -1;}
 char[] value(Step[] S,int i,char[] n){
   int j=def(S,i,n);
   if(j!=-1)
    for(int k=0;k<N(S[j].CR);k++)
      if(the character string that is pointed to by S[j].CR[k].n is
      equal to the character string that is pointed to by n)
      return S[j].CR[k].v;
 return "";}
```

Figure 10. An algorithm for extracting an error instruction

program is equal to a correct program in terms of the input and output, which are designated in CK. Lines 3–5 search the first step data of a static chunk in the order of executing instructions, and lines 6–7 compare variables, which are designated in an evaluation item set, between an answer and a correct program at the input point that is immediately before the stepwise execution of the first instruction of a static chunk. In addition, lines 3 and 21–23 search the end step of the static chunk in the order of executing instructions, and lines 24–25 compare variables, which are designated in an evaluation item set, between an answer and a correct program at the output point that is immediately after the stepwise execution of the end instruction of a static chunk. Lines 10–13 and 28–31 compute a dynamic backward slice when the compared computer resource at the input or output point differs between the answer and the correct program, and the compared resource is defined at the comparison time. Otherwise, when the compared resource is not defined at the comparison time, lines 16–18 and 34–36 consider instructions that are from the instruction at the comparison time to the first instruction of a program as an error instruction sequence. When instructions from P[x] to P[y] include P[e], a function "int boi(char[] P, int e)" returns x. When error_ins(P_P,P_SC,A_SC, P_S, A_S, CK) is executed, a Block-type array that contains an error instruction sequence is obtained.

6) Generate assistance expressions

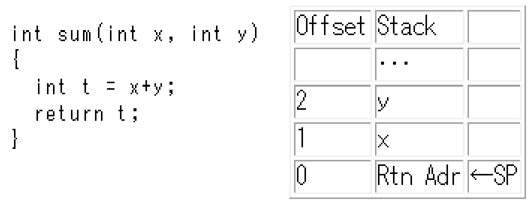
Our system first tries to detect a control structure error. If such errors are not detected, next, it tries to detect a behavior error and a computer resource control error. A control structure error is detected when a static chunk sequence of an answer program differs from that of a correct program. If such an error is detected, the system generates a chunk expression of the program by using static chunks of the answer and the correct program, and the answer program. A chunk expression of the program places static chunk sequences of the answer and the correct program side-by-side with the instructions of the answer program. A behavior error and a computer resource control error are detected when an error instruction sequence contains instructions. When such an error is detected, the system generates a chunk expression of trace data by using the answer program and dynamic chunks of the answer and the correct program. A chunk expression of trace data places dynamic chunk sequences of the answer and the correct program side-by-side. In addition, the system generates a projection expression of error steps by using the answer program, evaluation item sets, static chunks of the answer and the correct program, and step data of the answer and the correct program. A projection expression of error steps adds a chunk expression of the program to an error instruction sequence of the answer program and its execution results, and the execution result of a correct program.

IV. PROTOTYPE SYSTEM

This system holds the following static chunk conditions; sequence processing, start processing of function, start processing of function (with errors), end processing of function, end processing of function (with errors), readout processing of arguments, readout processing of arguments (with errors), repetition processing, selection processing. The question shown in Fig. 11 requires the implementation of a function sum that adds two arguments and stores the result in GR1. The behavior of the function in assembly is specified by a question sentence and a C program. For computer resource control, the use of all registers in the assembly program is specified by the corresponding C program. For example, the use of GR1 is specified by "assign GR1 to variable x" in the question sentence and variable x in the C program, which is assigned to the first argument.

Fig. 12 shows a correct answer (a) for the question shown in Fig. 11 and three incorrect answers (b, c, and d). In terms of behavior error, addition is correct instead of subtraction at line 6. In terms of control structure error, the instructions

Develop a function sum by CASL-II programming; the function is expressed by the following C program, and arguments of the function are held in the stack shown in the image below.

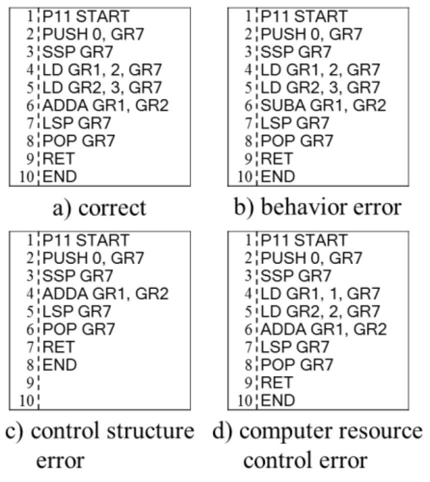


```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Offset	Stack	
	...	
2	y	
1	x	
0	Rtn Adr	←SP

However, your program should conform to the following conditions.
・Store the return value in GR1
・Consider GR7 as a stack frame pointer
・Consider GR1 as variable x
・Consider GR2 as variable y
・Implement readout processing of arguments and the end processing of function by using the stack frame pointer
・Implement save processing of a stack frame pointer in the readout processing of arguments by operation code PUSH
・Implement load processing of a stack frame pointer in the end processing of function by operation code POP
・Consider P11 as the label of the function sum

Figure 11. A question on function implementation



```
 1 P11 START
 2 PUSH 0, GR7
 3 SSP GR7
 4 LD GR1, 2, GR7
 5 LD GR2, 3, GR7
 6 ADDA GR1, GR2
 7 LSP GR7
 8 POP GR7
 9 RET
10 END
```

a) correct

```
 1 P11 START
 2 PUSH 0, GR7
 3 SSP GR7
 4 LD GR1, 2, GR7
 5 LD GR2, 3, GR7
 6 SUBA GR1, GR2
 7 LSP GR7
 8 POP GR7
 9 RET
10 END
```

b) behavior error

```
 1 P11 START
 2 PUSH 0, GR7
 3 SSP GR7
 4 ADDA GR1, GR2
 5 LSP GR7
 6 POP GR7
 7 RET
 8 END
 9
10
```

c) control structure error

```
 1 P11 START
 2 PUSH 0, GR7
 3 SSP GR7
 4 LD GR1, 1, GR7
 5 LD GR2, 2, GR7
 6 ADDA GR1, GR2
 7 LSP GR7
 8 POP GR7
 9 RET
10 END
```

d) computer resource control error

Figure 12. A correct answer and incorrect answers of the question in Fig. 11

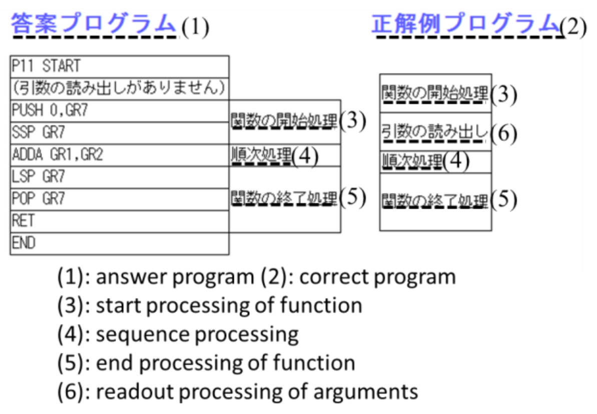
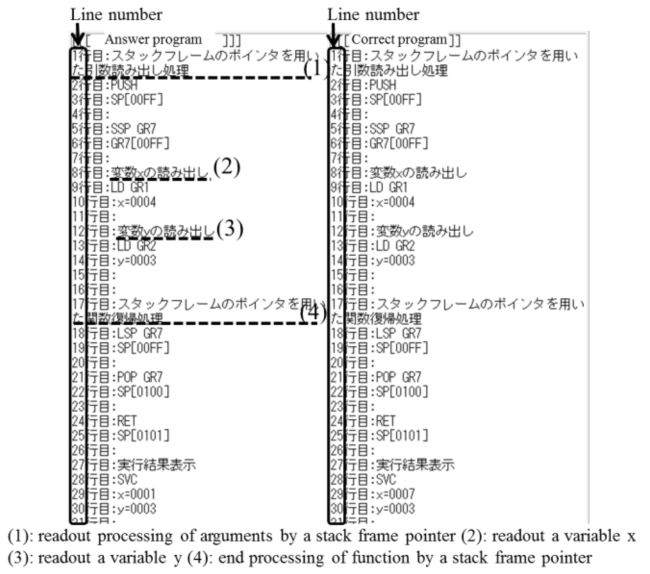


Figure 13. Chunk expression of program

described at lines 4 and 5 in the answer program are missing. The two instructions are used for readout of the arguments. In terms of computer resource error, the control structure is correct but the second operands of LD at lines 4 and 5, which are used for the readout of arguments, are incorrect.

Fig. 13 shows an assist expression for an answer program that contains a construction structure error (Fig. 12-c). A chunk "readout processing of arguments" is not shown because the answer program lacks instructions for the readout of arguments. A student notices this by comparing the chunk sequences of the answer and the correct paper.

Fig. 14 shows trace data of the correct and the answer program containing a behavior error (Fig. 12-b). A student specifies the causes of errors by confirming the difference between the trace data of his/her program and the correct program. Line 1 in Fig. 14 indicates the start of the readout of the argument. Lines 8 and 12 indicate the processing. Line 17 indicates the start of the end of the processing of the function. Furthermore, the student can notice the difference between the variables in his/her program and the correct program at line 29. The student can notice that lines 1–5 in his/her program in Fig. 12-b are correct because lines 1–14 in the chunk expression of the trace data of his/her program and the correct program are identical. Additionally, the student can notice that lines 7–9 in his/her program in Fig. 12-b are correct because lines 17–25 in the chunk expression of the trace data of his/her program in Fig. 14 and the correct program are identical. Therefore, the student can understand that the error in his/her program is caused at line 6 in Fig. 12-b.


Line number
Line number
(1)
(2)
(3)
(4)
(1): readout processing of arguments by a stack frame pointer (2): readout a variable x
(3): readout a variable y (4): end processing of function by a stack frame pointer

```
[[ Answer program ]]
1行目:スタックフレームのポインタを用いた引数読み出し処理
2行目:PUSH
3行目:SP[00FF]
4行目:
5行目:SSP GR7
6行目:GR7[00FF]
7行目:
8行目:変数xの読み出し
9行目:LD GR1
10行目:x=0004
11行目:
12行目:変数yの読み出し
13行目:LD GR2
14行目:y=0003
15行目:
16行目:
17行目:スタックフレームのポインタを用いた関数復帰処理
18行目:LSP GR7
19行目:SP[00FF]
20行目:
21行目:POP GR7
22行目:SP[0100]
23行目:
24行目:RET
25行目:SP[0101]
26行目:
27行目:実行結果表示
28行目:SVC
29行目:x=0001
30行目:y=0003
```

```
[[Correct program]]
1行目:スタックフレームのポインタを用いた引数読み出し処理
2行目:PUSH
3行目:SP[00FF]
4行目:
5行目:SSP GR7
6行目:GR7[00FF]
7行目:
8行目:変数xの読み出し
9行目:LD GR1
10行目:x=0004
11行目:
12行目:変数yの読み出し
13行目:LD GR2
14行目:y=0003
15行目:
16行目:
17行目:スタックフレームのポインタを用いた関数復帰処理
18行目:LSP GR7
19行目:SP[00FF]
20行目:
21行目:POP GR7
22行目:SP[0100]
23行目:
24行目:RET
25行目:SP[0101]
26行目:
27行目:実行結果表示
28行目:SVC
29行目:x=0007
30行目:y=0003
```

Figure 14. Chunk expression of trace data

Fig. 15 shows the projection expression of error steps for the computer resource control error in Fig. 12-d. The instructions in the dashed rectangle include causes of errors, and they are a dynamic backward slice that is computed because the value of GR1 of the answer program differs from that of the correct program at the output point of the second chunk. GR7 of the answer program is equal to that of the correct program at the output point of the first chunk, but GR1 differs at the output point of the second chunk. Therefore, the causes are narrowed to the following; in the second chunk, the instructions for GR7 are missing, and the instructions for GR1 are missing or incorrect. In a similar manner, narrowing down instructions that need to be reviewed using a dynamic backward slice and showing instructions and their execution results can help students to specify the causes of errors.

V. EVALUATION EXPERIMENT

An experiment was carried out to evaluate the effectiveness of assistance expressions. Chunk expressions of programs (expression 1), chunk expressions of trace data (expression 2), and projection expression of error steps (expression 3) help students specify the causes of their errors. We conducted a questionnaire survey for our system in the "Systems Programming" class, which is for second-year students in our university. The class includes four assembly programming exercises, and the students attempted 13 questions using our system. After finishing the fourth exercise, we distributed the questionnaires, which were aimed at determining how effective our system was in helping students specify the causes of their errors. The subjects answered the questions using the following five-point scale: 5 - very much, 4 - a lot, 3 - somewhat, 2 - not much, 1 - not at all. The questionnaires also contained space for comments.

24 valid responses were obtained. The average is rounded off to three decimal places, and the p-value is the value computed by a Wilcoxon test. The average and p-value of expression 1 are 3.88 and 0.0007, those of expression 2 are

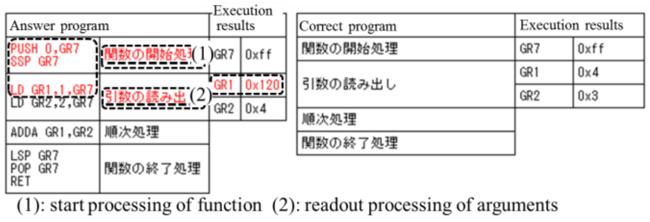


Figure 15. Projection expression of error steps

4.25 and 0.0002, and those of expression 3 are 4.29 and below 0.0001. The averages and the p-values confirm that all suggested expressions helped the subjects specify the causes of their errors. The comment for expression 2 is “I could notice errors in my program behavior using the expression, but I spent a lot of time specifying the corresponding instructions in my program.” Because expression 3 is designed to resolve such problems, we will link expressions 2 and 3 using a hyperlink in future work. The comment for expression 3 is “It is helpful to narrow down instructions that are related to errors.” On the other hand, “This expression was not very helpful to specify operand order errors.” Such a solution is beyond the scope of our study at this time. We will develop functions to solve such problems in future work.

VI. CONCLUSION AND SUMMARY

In this study, we proposed a system that generates expressions for helping students specify causes of errors by helping them comprehend their program; students can use the developed functions to automatically judge whether their programs are correct. We developed this system because some students are unable to perform the above mentioned tasks in assembly programming exercises. In this system answers, correct answers, test cases, evaluation item set, and static chunk conditions are provided as inputs, and true-false judgments, chunk expressions of programs, chunk expressions of trace data, and projection expressions of error steps are provided as outputs. To generate such expressions, we suggested extraction methods for static chunks, dynamic chunks, and error instruction sequences. Through a questionnaire survey, we evaluated the effectiveness of the suggested expressions in helping students specify the causes of their errors. The results suggested that the expressions were quite helpful.

REFERENCES

[1] IPA, JITEC, “Information Technology Engineers Examination,” http://www.jitec.ipa.go.jp/1_13download/shiken_yougo_ver2_1.pdf (in Japanese), pp. 3-8, 2011.

[2] K. Miyati, N. Takahashi, “Implementation and Evaluation of a Computer-Aided Assembly Programming Exercise System with a Function of Structural Anomaly Detection,” IEICE TRANSACTIONS on Information and Systems (in Japanese), Vol.J91-D, No.02, pp. 280-292, 2008.

[3] Mark Weiser, “Programmers Use Slices When Debugging,” Communications of the ACM, Vol. 25, no. 7, pp. 446-452, 1982.

[4] H. Agrawal, J. Horgan, “Dynamic Program Slicing,” SIGPLAN Notices, Vol.25, No.6, pp. 246-256, 1990.

[5] Tankut Akgul, Vincent J. Mooney III, Santosh Pande, “A Fast Assembly Level Reverse Execution Method via Dynamic Slicing,” Proceedings of ICSE, pp. 522-531, 2004.

[6] W.Kozaczynski, E.S.Liongosari, J.Q.Ning, “BAL/SRW : Assembler re-engineering workbench. Information and Software Technology,” Vol.33, no.9, pp. 675-684, 1991.