International Journal of Learning Technologies and Learning Environments
International Institute of Applied Informatics
2025, Vol. 8, No. 1, IJLTL874

A Method to Detect Structure Errors using a Model Program

Ryosuke Nakai [*], Tetsuo Kamina [†]

Abstract

It is difficult for novice programmers to solve compilation errors and perform debugging using only the error messages output by the compiler and execution results. Failure to solve errors may cause students to lose interest in programming. In this study, we propose a method to support the learning of C language by detecting the cause of errors that the compiler cannot indicate. This detection is performed by comparing the differences between the model program and that of the novice, assuming a situation where model programs are prepared in advance, that is, an exercise-style class. In the proposed method, syntactic elements in the source code processed by the parser are expressed in the XML format. The proposed method compares this XML representation with the document type definition (DTD) generated from the model program. First, the syntactic elements in the source code of the program written by the novice are translated into XML format. Next, a DTD is generated from the model program. This DTD defines the structure that the model program should satisfy. The DTD detects the structural difference between the model and program written by the novice, which are likely to be the cause of errors. The feedback provided is expected to enhance the effectiveness of C language learning for novice programmers.
Keywords: C language program, Syntax Tree, Similarity Degree, Program Similarity, XML

1 Introduction

The C language is still widely used today and is one of the representative languages that programming beginners often first learn. An essential tool for C language beginners when creating programs is the compiler. The compiler displays error messages for various issues within the program. However, these error messages are often challenging for beginners to understand. For instance, when a beginner omits the closing parenthesis of a block structure, existing compilers do not accurately indicate the line number causing the

[*] Graduate School of Engineering, Oita University, Japan (Current affiliation: Mitsubishi Electric Information Network Corporation (MIND))
[†] Division of Computer Science and Intelligent Systems, Oita University, Japan

error. Difficulty in understanding error messages can hinder the progress of writing programs as beginners may struggle to comprehend them, potentially dampening their learning enthusiasm. Furthermore, situations arise where the program structure does not accurately represent the algorithm, leading to incorrect results upon execution, even though there may not be a compilation error. In such cases, the compiler may not pinpoint these errors, making it challenging for beginners to identify and rectify them.

In this study, we envision scenarios where beginners tackle exercises with model answers in the context of programming, where compilation errors or structural errors may occur. We propose a method to assist novice learners in programming by appropriately highlighting the causes of compilation errors or structural mistakes in areas where differences exist between the model answer program and the program written by the beginner. Here, "beginner" refers to those getting started with programming, such as students taking their first programming courses.

To determine the differences between the model answer and the program written by the beginner, the proposed method involves representing the algorithm expressed by the source code in XML format, by abstracting the details of the program. Then, using a separately prepared document type definition (DTD) generated from the model answer, the XML document is validated to pinpoint error locations. This approach enables feedback by indicating areas in the XML document where the structures of the program written by the beginner and the model answer differ; these differences may be the sources of errors.

To evaluate the proposed method, we created a codebase with programs con-taining nested structures, such as bubble sort, based on multiple textbooks for C language programming. Subsequently, we conducted an experiment to investigate whether the proposed method can accurately detect the location of the missing closing parenthesis by removing one closing parenthesis from every block structure for every program within the codebase. Results confirmed that approximately 80% of their locations could be accurately detected. Additionally, for a subset of the codebase, intentional algorithmic errors were generated by changing the position of closing parentheses in block structures, and the tool accurately detected the loca-tions of all these intentional errors. We also investigated how the proposed method can accommodate individual differences in programs written by users, by considering programs generated by ChatGPT as ones written by the beginner. In some cases, the proposed method could not identify the sources of errors appropriately, opening the further research issues.

The structure of the remainder of this paper is as follows. In Section 2, the problem statement is presented. Section 3 provides a detailed explanation of the proposed method. In Section 4, the proposed tool is evaluated, and the results are discussed. Section 5 introduces related research, and finally, Section 6 concludes the paper and discusses future research.

2 Problem Statement

Many programming beginners learn C language as one of the initial languages. However, the learning environment for C language is challenging for beginners, espe-cially the error messages generated by compilers. For instance, in a program like the one shown in Figure 1, existing compilers cannot accurately pinpoint the

```
int main(void){
  int i, j;
  int num[5] = {7, 1, 3, 8, 5};
  int tmp;
  for (i = 0; i < 5; i++){
    for (j = i + 1; j < 5; j++){
      if (num[i] > num[j]){
        tmp = num[i];
        num[i] = num[j];
        num[j] = tmp;
      }
    //Closing parentheses is missing
  }
  i = 0;
  while (i < 5){
    printf("%d␣", num[i]);
    i++;
  }
  return 0;
}
```

Figure 1: Compilation error example

```
int main(void){
  int i, j;
  int num[5] = {7, 1, 3, 8, 5};
  int tmp;
  for (i = 0; i < 5; i++){
    for (j = i + 1; j < 5; j++){
      if (num[i] > num[j]){
        tmp = num[i];
      }
      num[i] = num[j];
      num[j] = tmp;
    }
  }
  i = 0;
  while (i < 5){
    printf("%d", num[i]);
    i++;
  }
  return 0;
}
```

Figure 2: Example of an algorithm error

location of the missing closing parenthesis within the main function. Consequently, beginners find it difficult to understand the source of errors indicated by the compiler output[1]. Additionally, compilers fail to identify some algorithm errors. In the case of programs like the one depicted in Figure 2, although there is no compilation error, the incorrect placement of the assignment operation yields undesired results. Since the compiler does not flag this as an error, beginners must engage in trial and error to identify the cause of the problem. These factors are considered to contribute to reduced learning motivation for beginners. Consequently, this study aims to support beginner learning by identifying compilation and algorithm errors that compilers cannot accurately indicate.

3 Proposed Method

3.1 Assumptions

We consider that the errors mentioned in the previous section can be accurately identified by comparing the expected correct program with the program written by the beginner. Therefore, under the assumption that a model answer exists, we propose a method to determine whether the structure of the model answer aligns with the program written by the beginner and use this information to identify the locations of errors. This assumption is applicable in situations such as classroom lectures, exercises, or scenarios where model answers are provided to programming material exercises.

3.2 Method overview

One challenge in this approach is how to abstract the details of the programs that do not contribute to the essence of the algorithms that the programs represent. This abstraction is necessary to accommodate individual differences in programs written by users. For this purpose, it is desirable to extract the *pattern* of the correct program from the model answer. One of the well-known methods to represent such patterns of tree structures is using DTD; thus, in this method, a DTD is used to represent the structure of the model answer. The program of the beginner is expressed as an XML document, which is validated using DTD to identify errors in the program structure.

First, the user and model answer programs are parsed and translated to abstract syntax trees (ASTs). Each AST node such as if, for, while, and return, expression statements and declaration statements, are traversed in the depth-first manner, and both the visit and leave actions, which correspond to a start-tag and end-tag of the corresponding XML element, respectively, are recorded. These records of the AST of the program written by the beginner are used to create an XML document. Subsequently, a DTD is generated from the model answer accordingly. Finally, using the DTD, validation is performed on the XML document, detecting error locations.

[1] Autocomplete mechanisms supported by IDEs may not help in several cases where, e.g., the beginner modifies the source code and accidentally removes the closing parentheses.

3.3 XML representation

The rules for translating a C language program into an XML document are defined for each syntax element of the C language:

- `if ( `*condition*` ) { `*statement*` }` →
  ```
  <if condition="condition">
      XML element obtained by converting statement
  </if>
  ```

- `for (`*initialization*`; `*condition*`; `*subscript_update*`) { `*statement*` }` →
  ```
  <Loop condition="condition" init="initialization" type="for">
      <control iteration="subscript_update"/>
      XML element obtained by converting statement
  </Loop>
  ```

- `while (`*condition*`) { `*statement*`; `*subscript_update*`; }` →
  ```
  <Loop condition="condition" type="while">
      <control iteration="subscript_update"/>
      XML element obtained by converting statement
  </Loop>
  ```

- *expression_statement* →
  ```
  <Statement expression="expression_statement"/>
  ```

- *declaration_statement* →
  ```
  <Statement declaretion="declaration_statement"/>
  ```

- `return`→
  ```
  <return/>
  ```

Regarding for and while statements, to represent the same repetitive structure for these statements, the XML elements for both are designated as Loop to ensure that regardless of how they are described, the resulting XML document remains the same. Additionally, statements updating the loop index are added as XML elements[2]. Other translation rules are straightforward. The root XML element is designated as main.

More precisely, to distinguish individual block structures within the XML, numbers are assigned to Loops and ifs in the order of appearance. For example, the XML document of the program in Figure 3 becomes Figure 4.

3.4 DTD expressions

DTD can be created using a procedure similar to that in Section 3.3. However, when outputting for each AST node, document type declarations (<!ELEMENT>) are generated rather than XML tags. Initially, the root of the element type declaration is declared as main. If an element has child elements, they are defined in order of appearance, such as <! ELEMENT element (child element 1, child element

[2]Concerning the reassignment expression of the while statement, it is considered an iteration update if it consists of either (1) an expression statement composed solely of increment or decrement operations or (2) an assignment statement where the same primitive variable is used on both sides.

```
int main(void){
  int i, j;
  int num[5] = {7, 1, 3, 8, 5};
  int tmp;
  for (i = 0; i < 5; i++){
    for (j = i + 1; j < 5; j++){
      if (num[i] > num[j]){
        tmp = num[i];
        num[i] = num[j];
        num[j] = tmp;
      }
    }
  }
  i = 0;
  while (i < 5){
    printf("%d", num[i]);
    i++;
  }
  return 0;
}
```

Figure 3: Example of a program

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<main>
  <Statement declaretion="int␣i,␣j;"/>
  <Statement declaretion="int␣num[5]␣=␣{7,␣1,␣3,␣8,␣5};"/>
  <Statement declaretion="int␣tmp;"/>
  <Loop1 condition="i␣&lt;␣5" init="i␣=␣0;" type="for">
    <control iteration="i++"/>
    <Loop2 condition="j␣&lt;␣5" init="j␣=␣i␣+␣1;" type="for">
      <control iteration="j++"/>
      <if1 condition="num[i]␣&gt;␣num[j]">
        <Statement expression="tmp␣=␣num[i]"/>
        <Statement expression="num[i]␣=␣num[j]"/>
        <Statement expression="num[j]␣=␣tmp"/>
      </if1>
    </Loop2>
  </Loop1>
  <Statement expression="i␣=␣0"/>
  <Loop3 condition="i␣&lt;␣5" type="while">
    <Statement expression="printf(&quot;%d␣&quot;,␣num[i])"/>
    <control iteration="i++"/>
  </Loop3>
  <return/>
</main>
```

Figure 4: Example of an XML document

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE main [
        <!ELEMENT main (Statement+,Loop1,Statement+,
                        Loop3,return)>
        <!ELEMENT Statement (#PCDATA)>
        <!ATTLIST Statement
                declaretion CDATA #IMPLIED
                expression CDATA #IMPLIED>
        <!ELEMENT Loop1 (control,Loop2)>
        <!ATTLIST Loop1
                condition CDATA #REQUIRED
                init CDATA #REQUIRED
                type CDATA #REQUIRED>
        <!ELEMENT control (#PCDATA)>
        <!ATTLIST control
                iteration CDATA #REQUIRED>
        <!ELEMENT Loop2 (control,if1)>
        <!ATTLIST Loop2
                condition CDATA #REQUIRED
                init CDATA #REQUIRED
                type CDATA #REQUIRED>
        <!ELEMENT if1 (Statement+)>
        <!ATTLIST if1
                condition CDATA #REQUIRED>
        <!ELEMENT Loop3 (Statement+,control)>
        <!ATTLIST Loop3
                condition CDATA #REQUIRED
                type CDATA #REQUIRED>
        <!ELEMENT return (#PCDATA)>
        ]>
```

Figure 5: Example of DTD

2)>. The occurrence frequency of expression and declaration statements may not carry semantic in the structure of the program. Moreover, associating the document type with the occurrence frequency might fail to account for differences in the programs of beginners owing to individual variations (for instance, declaring multiple variables together in a single statement or separately in different statements, resulting in different XML documents for the same program). Therefore, if expression or declaration statements appear, they are abstracted in the DTD as one or more occurrences (as Statement+). Similar to XML document generation, numbers are assigned to Loops and ifs in the order of appearance.

Attributes attached to elements are defined in the attribute list declaration (<! ATTLIST>. In this case, the attribute type is set to character data (CDATA), and the default value is marked as required (#REQUIRED). As an example, considering the program in Figure 3 as the model answer, its corresponding DTD will be as depicted in Figure 5.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<main>
  <Statement declaretion="int␣i,␣j;"/>
  <Statement
   declaretion="int␣num[5]␣=␣{7,␣1,␣3,␣8,␣5};"/>
  <Statement declaretion="int␣tmp;"/>
  <Loop1 condition="i␣&lt;␣5" init="i␣=␣0;" type="for">
    <control iteration="i++"/>
    <Loop2 condition="j␣&lt;␣5" init="j␣=␣i␣+␣1;"
           type="for">
      <control iteration="j++"/>
      <if1 condition="num[i]␣&gt;␣num[j]">
        <Statement expression="tmp␣=␣num[i]"/>
        <Statement expression="num[i]␣=␣num[j]"/>
        <Statement expression="num[j]␣=␣tmp"/>
      </if1>
  </Loop2>
    <Statement expression="i␣=␣0"/>
    <Loop3 condition="i␣&lt;␣5" type="while">
      <Statement
        expression="printf(&quot;%d␣&quot;,␣num[i])"/>
      <control iteration="i++"/>
    </Loop3>
    <return/>
  </Loop1>
</main>
```

Figure 6: XML document converted from a program with errors

3.5 XML validation

Using the XML document and DTD created in Sections 3.3 and 3.4, respectively, an inspection is conducted to verify whether the structure of the program aligns with that of the model answer. This is verified as part of the validation for the generated XML document from the program. For instance, validating the XML document (Figure 6) created from a program in Figure 1, where closing parentheses are missing, using the DTD in Figure 5 reveals that the element Loop1 in the DTD is declared as <!ELEMENT Loop1(control,Loop2)>. In contrast, the XML document in Figure 6 exhibits a flaw in the block structure due to the missing closing parenthesis. The segment from <Statement expression="i = 0"/> is included in the child element of Loop1, indicating that there is an error in this part.

4 Evaluation

We conducted an evaluation to assess the accuracy of the proposed method in detect-ing error locations in programs with missing closing parentheses in block structures and programs with algorithm errors (generated by intentionally shifting the position of closing parentheses). The codebase used for this evaluation consists of nested programs (bubble sort, insertion sort, selection sort, Shell sort, and a program to find maximum and minimum values) created by one of the authors. This codebase excludes functions other than main.

4.1 Identification of missing closing parentheses

Firstly, we conducted an experiment to investigate the accuracy of our method to detect missing closing parentheses in block structures. This experiment involved initially removing one closing parenthesis from the source code and using this method to identify the error location. Subsequently, the identified position was checked to ensure its correctness. The evaluation criteria were whether the errors were correctly detected and if the program functioned correctly after inserting the missing closing parenthesis at the detected location. If the criteria were met, it was considered that the "error was correctly detected." This process was repeated for all closing parentheses in the source code.

Table 1: Detection results

Source code	Correct / Total numbers
Bubble sort	4 / 4
Insertion sort	3 / 4
Selection sort	4 / 5
Shell sort	5 / 6
Maximum and minimum	4 / 4

The evaluation results are presented in Table 1. For the bubble sort and program to find maximum and minimum values, the method correctly detected errors in all cases. In other source codes, for example, insertion sort achieved a correct detection rate of 75%, selection sort achieved 80%, and Shell sort achieved 83%.

We elaborate on the cases where errors were wrongly detected. In all instances where the method failed to detect errors, the closing parentheses, sandwiched between <Statement/> elements in the XML document, were missing. This occurred because in this method, the occurrence of expression or declaration statements in the DTD was defined as one or more occurrences. Consequently, it failed to precisely distinguish when the cause of the error was sandwiched between expression or declaration statements.

4.2 Identification of algorithm errors

Next, we conducted an experiment for detecting errors in positions of closing parentheses in block structures on the codebase, specifically targeting bubble sort and selection sort. In this experiment, we modified the positions of closing parentheses to create a program with algorithm errors that the compiler would not detect. The created programs are illustrated in Figure 2 and Figure 7. The evaluation criterion was to verify whether the method accurately identified the closing parentheses placed in the wrong positions. The evaluation results for both cases showed that the method accurately identified the locations of errors.

```
#include <stdio.h>
int main(void){
  int data[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
  int n, i, j, min, temp;
  printf("---selection␣sort---\nBefore␣:␣");
  for(n=0; data[n]!='\0'; n++){
    printf("%d␣",data[n]);
  }
  i = 0;
  while(i < n){
    min = i;
    for(j=i+1; j<n; j++){
      if(data[j]<data[min]){
        min = j;
      }
      temp = data[i];
    }
    data[i] = data[min];
    data[min] = temp;
    i += 1;
  }
  printf("\nAfter␣␣:␣");
  for(n=0; data[n]!='\0'; n++){
    printf("%d␣",data[n]);
  }
  return 0;
}
```

Figure 7: Selection sort with algorithm errors

4.3 Evaluation using ChatGPT

An evaluation regarding the extent to which the method can accommodate individual differences in programs written by users was conducted by treating programs generated by ChatGPT as if they were created by the user. Initially, ChatGPT was requested to write a C language program using bubble sort to sort the sequence “7,1,3,8,5” in ascending order. The program was supposed to be written within the main function, using only for, if, and while statements for control structures, and output the sorted result[3]. This request was made to make the generated program as similar as possible to the corresponding program in the codebase. Subsequently, similar to the evaluation experiment in Section 4.1, the closing parentheses of block structures were systematically removed, and an evaluation was performed to deter-mine whether the method could identify these removals. This process was applied to all programs in the codebase.

The evaluation results are presented in Table 2. Compared to the evaluation experiment in Section 4.1, the number of successful detections for bubble, insertion,

[3] This request to ChatGPT was made in Japanese.

Table 2: Detection results

Source code	Correct / Total numbers
Bubble sort	3 / 4
Insertion sort	2 / 4
Selection sort	3 / 5
Shell sort	0 / 6
Maximum and minimum	4 / 4

```
int main(void){
  int data[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0}; int n;
  printf("---shell␣sort---\nBefore␣:␣");
  for(n=0; data[n]!='\0'; n++){ printf("%d␣",data[n]); }
  int i, j, step, temp;
  step = 1;
  while(step < n/9){ step = step * 3 + 1; }
  while(step > 0){
    i = step;
    while(i < n){ temp = data[i];
      j = i - step;
      while(j >= 0 && temp < data[j]){
        data[j + step] = data[j];
        j -= step;
      }
      data[j + step] = temp;
      i++;
    }
    step /= 3;
  }
  printf("\nAfter␣␣:␣");
  for(n=0; data[n]!='\0'; n++){ printf("%d␣",data[n]); }
  return 0;
}
```

Figure 8: Shell sort program of a model answer

and selection sort was one less. Regarding Shell sort, it was not able to accurately detect any error locations.

In the cases of bubble, insertion, and selection sort, all the failed error detec-tions were instances where closing parentheses sandwiched between <Statement/> elements in the XML document were missing, similar to the failed examples in the evaluation experiment in Section 4.1. The decrease in the number of successful de-tections can be attributed to the presence of <Statement/> elements in programs generated by ChatGPT, which were not in the model answer. This led to an in-crease in the number of elements sandwiched between expression and declaration statements, which are the sources of errors.

We explain why the detection success count for Shell sort was zero. First, Figure 8

```
int main() {
  int arr[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Original␣Array:␣");
  for (int i = 0; i < n; i++) { printf("%d␣", arr[i]); }
  printf("\n");
  int gap = n / 2;
  while (gap > 0) {
    int i = gap;
    while (i < n) { int temp = arr[i];
      int j = i;
      while (j >= gap && arr[j - gap] > temp) {
        arr[j] = arr[j - gap];
        j -= gap;
      }
      arr[j] = temp;
      i++;
    }
    gap /= 2;
  }
  printf("Sorted␣Array:␣");
  for (int i = 0; i < n; i++) { printf("%d␣", arr[i]); }
  printf("\n");
  return 0;
}
```

Figure 9: Shell sort program created by ChatGPT

shows the model answer for Shell sort, and Figure 9 displays the program created by ChatGPT. As evident from the figures, the first while statement in the model answer is not in the ChatGPT program. The processing from the next while statement onward is the same. Owing to this misalignment in the while statement, the numbering assigned to the Loop in the XML document, used to distinguish individual block structures, became misaligned. As a result, subsequent comparisons in the Loop were not performed correctly, leading to the failure to detect missing closing parentheses. The difference between the two programs was caused by the fact that the model answer determines the width of grouping necessary for Shell sort in the first while statement, whereas ChatGPT performs it without using a while statement.

There are two possible interpretations for this experiment result. The first is to interpret it as correctly identifying algorithm errors that do not align with the intent of the model answer. This interpretation holds when the task intent includes using while or for statements to determine the width of Shell sort. In this case, ChatGPT should use either while or for statements, which would lead to an algorithm error by not aligning with the task s̊ intent. This was correctly pointed out by this method. The second interpretation is that the method failed to identify the errors (Table 2 stands for this interpretation).

This interpretation applies when the task does not specify how to determine the width of Shell sort and allows for freedom in writing. In this scenario, when determining the width of Shell sort, whether it is a while statement or not should be ignored. Since this method does not ignore this aspect, it fails to accurately point out the errors in this context.

The proposed method extracts task intent solely from the model answer; however, this Shell sort example demonstrates that this approach may not be sufficient in certain cases. Therefore, it is necessary to reconsider the method of extracting task intent in the future.

4.4 Discussions

From these experiments, it can be summarized that this method identified error lo-cations in approximately 80% of cases involving missing closing parentheses in block structures and cases where there were errors in the positions of closing parentheses in block structures. However, the method cannot identify missing closing parentheses sandwiched between expression or declaration statements.

From these findings, it is evident that this method can detect errors when there are differences in parent-child relationships among XML elements or other struc-tural issues in the program. Further detailed experiments, especially systematic evaluation experiments involving algorithm errors, need to be conducted to explore the effectiveness of this method. Additionally, evaluations should be carried out in real-lecture or exercise settings to determine if the method can accurately identify errors in programs written by students.

5 Related Work

5.1 Programming support specialized for the C language

For beginners starting with C language, there is a static analysis tool called C-Helper [1] designed to provide clear displays of compilation errors. Traditional compilers often present messages that are difficult for beginners to understand, lack solutions, and generally not suitable for programming novices. C-Helper aims to detect common mistakes made by beginners and provide user-friendly messages. According to reference [1], issues detectable by this tool include inconsistent indentation, assigning character arrays to variables of type char, parameter mismatches in printf, missing return statements, extra semicolons in function definitions, insufficient semicolons in structure declarations, and incorrect use of sizeof for dy-namically allocated arrays. However, C-Helper cannot detect common mistakes like missing closing parentheses often made when writing for, while, or if statements. Detecting missing closing parentheses is a challenge for existing compilers to pin-point accurately. In these aspects, this study differs from C-Helper and aims to provide support for beginners.

An intelligent instructional system for C language beginners is C-Tutor [2], which extracts the intent of the programmer as a program description through reverse engineering from sample programs. Subsequently, it uses the knowledge base to analyze the program, examines the unfulfilled goals in the program description while executing the input source code, queries the knowledge base for the set of plans to implement those goals, detects the differences between the plans and the source code, and provides feedback.

C-Tutor is similar to this study in providing the intent of the programmer through sample programs; however, it requires the preparation of test cases in advance, which takes time and effort. Furthermore, our scope is different from this study in that this study deals with compile-executable source code, while our proposal also targets syntax errors.

5.2 Analysis methods for program similarities

One method for analyzing source code is code clone detection [3]. Code clone detection involves identifying specific lexemes as clones and detecting whether the source code contains similar elements. Specific techniques for code cloning include using lexical analysis to identify lexemes as clones [4], employing hash functions to compress lines into strings of fixed length and detect clones [5], and considering entire functions, procedures, class definitions as clones and solving the clone detection problem by finding equivalent pairs of elements [6]. This study, as described in Section 3, shares similarities with various code cloning detection methods, particularly in comparing model answers and user programs after abstracting the source code. However, the comparison method employed in this study is specialized for detecting error locations and not intended for detecting code clones. It is challenging to apply the proposed approach in this study to code cloning detection. Similarly, using ex-isting code cloning detection methods for the purposes of this study may also prove challenging.

Program concept recognition [7] aims at supporting program comprehension by recognizing language-independent ideas of computation and problem-solving methods such as data structures and algorithms. Such concepts are represented using ASTs with additional constraints based on control-flows and data-flows. Quilici [8] extended this method by sacrificing the ability to recognize every concept located in the source code to improve efficiency. These approaches are similar to our proposal in that both compare the patterns of algorithms with the programs, even though our proposal only uses ASTs as patterns and such patterns are extracted from the model answers. Like code clone detection methods, program concept recognition methods are used to comprehend the large-sized source code, and not designed for detect flaws in the programs.

5.3 Programming support with model answers

A static analysis framework for beginners in Java [9] is one approach that uses static analysis for programming support. This method is mainly used to detect logical errors and check the code quality. It compares the model answer with the program of the beginner after normalization, considering the similarity in structure (such as loop structure, assignment, and method invocation order). It also measures software metrics. However, this method targets the fill-in-the-blank questions and it is uncertain whether it can be applied to programming exercises that allow writing source code freely from the beginning.

There is a study that considers the processing within a program as functionally meaningful units (referred to as components in related studies) and progressively comprehends the overall structure of the program [10]. This approach proposes a learning method for structural understanding in programming, where the code is recognized as meaningful chunks or components, and the learner progressively extends the entire program by assembling these components to meet the predefined structural requirements of the task. Initially, the learner writes code line by line, combining them to form the desired structure for a given task. When faced with difficulties owing to unfamiliarity or lack of knowledge, the learner can receive hints as feedback, contributing to the structural understanding of the learner. This method consists of a correctness judgment function that compares the answers of the learner with the predefined correct solutions for each task and a feedback function based on correctness judgments, thus sharing some similarities with this study. However, it differs from this study as it uses predefined components only and considers an answer as correct only when it perfectly matches a single correct example, lacking the flexibility to accommodate individual differences of user programs.

5.4 Program representation using XML

CX-Checker [11] is a tool to verify whether a developed program complies with coding conventions. It first parses the source code and represents it in XML. Users can then use XPath or DOM formats to specify coding conventions as rules and detect rule violations in the program. While this tool is not specifically designed to detect syntax errors or common errors for beginners, its generic rule-writing capabilities could potentially be used to describe the intentions of model answers as rules. However, it is unclear whether it can detect basic errors such as omitting closing parentheses. Additionally, it needs the preparation of the question intention as rules, making it different from this study, where the model answer can be used directly.

There is a method [12] that converts given programs into XML-formatted syntax trees and compares the similarity of C language programs. This approach focuses on the structure of the programs to evaluate their similarity by comparing the respective syntax trees. It aims to enable the comparison of block structures, including function call relationships, which is challenging with string-based comparisons. However, while this method proposes the conversion of programs into XML-formatted syntax trees, it does not provide specific details about the comparison method. This differs from this study, which proposes a specific comparison method.

6 Conclusion

In this paper, we addressed the limitations of existing compilers, which cannot accurately pinpoint the line numbers causing errors when a block structure bracket is omitted, or when the program structure fails to accurately represent the algorithm, resulting in incorrect results during execution without triggering compilation errors. To overcome these limitations, we proposed a method to compare the model answer with the program written by the beginner and identify error locations based on the differences. We represented the algorithm expressed by the source code processed by the parser in XML format and performed XML validation using a DTD generated from a separately prepared model answer to identify error locations. Additionally, we conducted an evaluation using a codebase created for this method, and the results confirmed that this method can identify many error locations. It should be noted that this method has not undergone an evaluation involving subjects. Therefore, in the future, it is necessary to investigate the extent to which individual differences between the model answer and the program of the beginner can be tolerated.

References

[1] K. Uchida and K. Gondow, "C-Helper: C latent-error static/heuristic checker for novice programmers," in *Proceedings of the 8th International Conference on Computer Supported Education (CSEDU 2016)*, pp. 321–329, 2016.

[2] J. S. Song, S. H. Hahn, K. Y. Tak, and J. H. Kim, "An intelligent tutoring system for introductory clanguage course," *Computers & Education*, vol. 28, no. 2, pp. 93–102, 1997.

[3] N. Saini, S. Singh, and Suman, "Code clones: Detection and management," *Procedia Computer Science*, vol. 132, pp. 718–727, 2018.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "A code clone detection technique for object-oriented programming languages and its emprical evaluation," in *Proc. of the 62nd National Convention of IPSJ*, pp. 23–28, 2001.

[5] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Inť Conf. on Software Maintenance (ICSM 99)*, pp. 109–118, 1999.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. A. Kontogiannis, "easuring clone based reengineering opportunities," in *Proc. of the 6th IEEE Int l Symposium on Software Metrics (METRICS'99)*, pp. 292–303, 1999.

[7] W. Kozaczynski, J. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, 1992.

[8] A. Quilici, "A memory-based approach to recognizing programming plans," *Commun. ACM*, vol. 37, no. 5, pp. 84–93, 1994.

[9] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students 'Java programs," in *Proceedings of the Sixth Australasian Conferenceon Computing Education (ACE 04)*, pp. 317–325, 2004.

[10] K. Koike, T. Tomoto, T. Horiguchi, and T. Hirashima, "Proposal of the expand-able modular statements method for structural understanding of programming, and development and evaluation of a learning support system," *Transactions of Japanese Society for Information and Systems in Education*, vol. 36, no. 3, pp. 190–202, 2019. (in Japanese).

[11] T. Osuka, T. Kobayashi, N. Atsumi, J. Mase, S. Yamamoto, N. Suzumura, and K. Agusa, "CX-Checker: A flexibly customizable coding checker for C," *IPSJ Journal*, vol. 53, no. 2, pp. 590–600, 2012. (in Japanese).

[12] H. Bao, M. Nakata, and Q.-W. Ge, "A proposal of syntax tree expression of C language programs," *CIEC Computer & Education*, vol. 36, pp. 56–61, 2014.(in Japanese).