PAPER *Special Issue on Knowledge-Based Software Engineering*

A Diagnosis System of Programming Styles Using Program Patterns

Rika SEKIMOTO[†] ***and*** **Kenji KAIJIRI**[†], ***Members***

SUMMARY Programming styles play an important role to promote maintainability of programs. The authors aim at developing a system for assisting a task that programmers rewrite programs in order to improve their readability, understandability and maintainability. This system detects program fragments which infringe programming styles in a C program and makes diagnosis on the programming style. This system has the following features: 1) It can detect various patterns, for example, context dependent patterns and dispersed patterns extending two or more functions. It is difficult to detect these patterns by character based pattern matching; and 2) Each style guideline is defined as program patterns. This system uses them as search data, so it becomes easy to add or change style guidelines which are to be checked. The authors validated that inspection of various style guidelines is possible through this system. Moreover, to evaluate the effectiveness of this system, they made experiments of inspecting a program for some style guidelines on 86 novice programmers. This result indicates that the system is effective in assisting a work that novice programmers check and/or correct programming styles.

key words: *programming style, program pattern, program recognition, programming course*

1. Introduction

Programs must be not only valid but also readable and understandable. The guideline to produce good style code is called programming style. There are various metrics in programming style, such as reliability, efficiency, testability, readability, understandability, modifiability, portability, and so on.

Compilers can't produce an executable code if a program includes some syntactic errors, and a program doesn't work correctly to programmer's intention if it includes some semantic errors. However a program infringing programming styles may be true to programmer's intention, so programmers are not in trouble at that time. As the result, maintenance problem will arise later. In programming courses, to learn programming styles as well as grammars and algorithms is very important. However, little time is spent on learning programming styles in introductory programming courses. One reason for this is the lack of supporting systems about programming style education.

We aim at developing a diagnosis system of programming style. It assists programmers to obey the programming style. Our system detects program fragments which infringe the programming style, and outputs the diagnosis message. The target language is the ANSI C. In this paper, we focus our discussion on the readability and understandability in programming styles.

We call the pattern which infringes programming styles as **"bad pattern."** We proposed a new representation method for various bad patterns, and implemented a diagnosis system that detects program fragments which match certain pattern [1], [2]. We validated that detection of various bad patterns is possible through this system. For example, it can detect context dependent patterns and dispersed patterns spanning multiple functions in C programs. In this system, each style guideline is defined as program patterns. The input is a target program and these patterns, so it is easy to add or change style guidelines which are to be checked. Moreover, to evaluate the effectiveness of this system, we made experiments of detecting bad patterns on 86 novice programmers. This result indicates that the system is effective in assisting novice programmer's task to check and/or correct programming styles.

This system detects bad patterns in C programs and makes diagnosis on programming styles. This detection is effective to improve the quality of programs. Moreover, it may be effective to learn good coding style.

In this paper, we describe the diagnosis system of programming styles using program patterns. Section 2 discusses programming styles. Section 3 discusses the representation method of program patterns for search. In Sect. 4, we outline the diagnosis system of programming styles. Section 5 describes the description experiment of various kinds of bad patterns and the recognition experiments for programs of the GNU text utility library. In Sect. 6, we compare our system with related works. In Sect. 7, we describe the experiment of detecting bad patterns using our system on novice programmers. Finally, in Sect. 8, we present our conclusions and plans for future works.

2. Programming Style

There are various guidelines in readability and understandability. For example, there are the following two kinds of guidelines: 1) Guidelines that do not reflect on the behavior of programs directly, such as, inden-

Manuscript received August 9, 1999.
Manuscript revised October 31, 1999.

[†]The authors are with the Faculty of Engineering, Shinshu University, Nagano-shi, 380–8553 Japan.

tation, naming, and commenting. 2) Guidelines about structure, such as, usage of statements and expressions. For the former class, indentation problem is solved using formatting tools and naming/commenting problems depend on semantics, so they are out of scope. In the conventional case, guidelines are described with natural languages, so novice programmers can not understand guidelines precisely. Consequently, it is difficult work to detect bad patterns for the guidelines about structure. Therefore in our research, we focus on the guidelines about structure and propose their recognition system.

Now two examples of the guidelines about structure are given.

[Guideline about side effect within conditional expressions]: The expression that has side effect, such as a conditional expression including assignment statement, is unfavorable.

[Guideline about file processing]: Files should be closed explicitly when the corresponding operation finished.

We will consider the detecting process of bad patterns. In the pattern infringing the first guideline, the system must detect the conditional expression which includes assignment statements. In order to detect such a pattern, the system should recognize a conditional expression within an if-statement or a while-statement, so it becomes necessary to parse programs. In the pattern infringing the later guideline, the system must detect the location where a file is opened and that file is not closed explicitly. This is the pattern that lacks the part of searching patterns. The location of "fclose" function calls may be apart from the location of "fopen" function calls. Sometimes those may appear in different functions/files in C programs. It is difficult to recognize these various bad patterns by traditional source code search tools based on string pattern matching, for example the `grep` family in UNIX.

We examined manually whether programs written by novice programmers have bad patterns. The objects are 30 programs which are the answer programs for the following problem: To merge two sorted data sequences. In this examination, we found 7 kinds of bad patterns about structure. One bad pattern was found in 90% of programs. The [*guideline about side effects*] was found in 80% of programs. The [*guideline about file processing*] was found in 30% of programs. This result indicates that there are many bad patterns in programs written by novice programmers. Therefore, the supporting tool for education of programming style is necessary.

3. Program Pattern

3.1 Program Pattern Representation

We examined some programming guidelines and deduced some specific characteristics about these guidelines:

1. Occurrence context of a pattern. A pattern is problematic in a certain context (context dependent pattern).

2. Nonexistence of one component for a pair of pattern. A pair of pattern is necessary, but one pattern is missing.

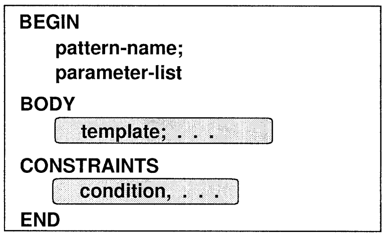
3. A set of pattern dispersed over several functions.

4. A kind of bad patterns, especially for educational use, are not closed, so the system must be extensible.

Considering these characteristics, we designed our program pattern representation.

We call a description for specific program fragments as a program pattern. In this system, a program pattern is used to represent a style guideline. Figure 1 shows our proposed description form of a program pattern. A program pattern consists of pattern name, parameter list, body part, and constraint part. In body part, the pattern to be matched is described, which we call the template. In constraint part, the context condition which is concerned with the template (e.g. condition about syntax and static analysis information) is described. By the separation of body part and constraint part, the system has the following features: 1) Flexibility of pattern representation: Using program patterns we are able to represent easily the context dependent patterns. Moreover, addition of a new condition becomes easy; and 2) The system becomes extensible to use some external informations, for example, static analysis informations.

For the flexibility and the extensibility, we divided the program pattern into two types: fundamental patterns and compound patterns. A fundamental pattern is a basic pattern for searching and uses sentential forms of the C grammar. The template consists of program fragments of the C language parameterized with regular expressions. A compound pattern consists of sets of instances of fundamental patterns. In description of compound patterns, a high level concept can be expressed using fundamental patterns. This is also useful for defining the following patterns: 1) Patterns spanning multiple functions; and 2) Patterns which could



```
BEGIN
    pattern-name;
    parameter-list
BODY
    template; . . .
CONSTRAINTS
    condition, . . .
END
```

Fig. 1 Description form of a program pattern.

appear in any order. These patterns are indispensable for programming style description.

3.2 Fundamental Pattern

In body part of fundamental patterns, the expanded C language is used. The extension includes a set of symbols that can be used as the substitution for syntactic entities of C. Some regular expressions can be used instead of the structure element of C. The template consists of the following elements;

- Program fragment based on the C grammar
- Pattern variable: Wild-cards for syntactic entities (@: statement, #: expression, $: variable)
- Special wild-cards: categorical representation (@if, @while, @do, @for, @switch, @assign), class representation (@alternate, @loop)

In constraints part of fundamental patterns, each condition indicates constraints about syntax structure, character string, data/control flow, and so on. Some examples of the constraints are shown below; pattern variables (p) and constants (c) may be used as parameters of constraint predicates.

- type constraint
  type(p, c), type_equal(p, p)
- syntax structure constraint
  equality_exp(p), assign_exp(p), not_break(p)
- character string constraint
  literal_equal(p, c), constant(p), integer_large(p, c)
- flow constraint
  reaching_def(p, p), cflow(p, p)

For example, “reaching_def(p1, p2)” means that the definition in the program fragment matched with p1 reaches the program fragment matched with p2, and “literal_equal(p, 10)” means that the program fragment matched with p is a constant “10.”

3.3 Compound Pattern

In body part, a sequence of fundamental patterns with optional labels is used. We call this as a component pattern. A component pattern is described as follows;

```
label: pattern_name (parameter);
```

Labels are used to identify program fragments that match the corresponding component patterns.

The constraints which may be used in fundamental patterns and positional constrains may be used. The relation between the program fragments, which matched each component pattern, is indicated by positional constraints. Some examples of the constraint specific to compound patterns are as follows; In addition to pattern variables and constants, labels of component pattern (l) can be used as a parameter of constraint predicates.

- character string constraint
  node_literal_equal(p, p)
- positional constraint
  node_equal(p, p), condition_part(l), loop_body(l), with_in(l, l), before(l, l)
- others
  not_found(l), same_variable(p, p)

For example, “with_in(l1, l2)” means that the program fragment matched with the fundamental pattern labeled with l1 is included in the program fragment matched with the fundamental pattern labeled with l2.

3.4 Sample Program Patterns

We show two examples.

[Guideline about side effect within conditional expression]:

```
BEGIN                      BEGIN
side_effect1_pattern;      assignment_pattern;
BODY COMPLEX               BODY
  l1:assignment_pattern()    @assign
CONSTRAINTS                END
  condition_part(l1)
END
```

This pattern will match an assignment statement in a conditional expression. [side_effect1_pattern] said that an assignment statement binded with l1 is included within some conditional statement.

[Guideline about file processing]:

```
BEGIN
file_not_close_pattern;
$v1,$v2
BODY COMPLEX
  l1:file_open_pattern($v1);
  l2:file_close_pattern($v2)
CONSTRAINTS
   found(l1),
   not_found(l2)
END

BEGIN                   BEGIN
file_open_pattern;      file_close_pattern;
$v1                     $v1
BODY                    BODY
  $v1 = fopen(#);        fclose($v1);
END                     END
```

This pattern said that a file open statement exists and no file close statements exist. This is not a sufficient description for this pattern. This pattern does not warn for the case that the close statement is missing which corresponds to some open statement, but it only warns for the case that open statements exist but no close statement exists. More precise description is a open problem for our system.

4. Program Pattern Recognition

4.1 Outline of the System

In this system, users select the program and the style guideline, and the system detects program fragments which infringe the given style guideline. Each program pattern was enrolled in the pattern database in advance, and user selects a program pattern from it. The system identifies all program fragments which match with the specified program pattern. Finally, the system shows the number of matched elements and their locations. A sample message by the system is as follows;

```
Pattern[1]: Guideline about file processing

There is 1 match list.

There is a file which is opened, and is not closed.
In file processing, file is opened when it is used, and it
should be closed explicitly.

——(1) list of matching——
1st pattern - - - - - line 41
      input = fopen ( FileName, “r” ) ;
2nd pattern         == unmatched ==
```

In Fig. 2, we show the architecture of the system. This system consists of a parser and two submodules: base pattern search module and compound pattern search module.

4.2 Base Pattern Recognition

In base pattern recognition, the system produces a tree automaton based on templates of body part and identifies all program fragments which match with the pattern. In this template matching, it uses the technique based on the finite state automaton proposed by [3].

Figure 3 shows the flow of the *base pattern search module*. The *automaton generator* gets a template as an input and produces an automaton called a pattern automaton. After an AST (Attributed Syntax Tree) and a pattern automaton have been generated, the *base pattern template matching module* drives a pattern automaton on an AST and produces a template match list.

The template match list is checked by the *base pattern constraint check module* and the base pattern match list which fulfills all restrictions is selected.

4.3 Compound Pattern Recognition

In the *compound pattern search module*, at first, the template of an inputted pattern is divided to some component patterns. Each component pattern is searched using the *base pattern search module*. After the base pattern match list of all component patterns is obtained, all combinations of the base pattern match list were checked in the *component pattern constraint check module*. The compound pattern matched list is a set of program fragments that fulfill all conditions.

For example, we supposed that a compound pattern consists of three fundamental pattern (a, b, c), and the system finds l program fragments for a, m for b, and n for c. We make all combinations of these patterns. In this case, $l \times m \times n$ combinations are to be generated, and for all of these combinations, we try to check the constraints.

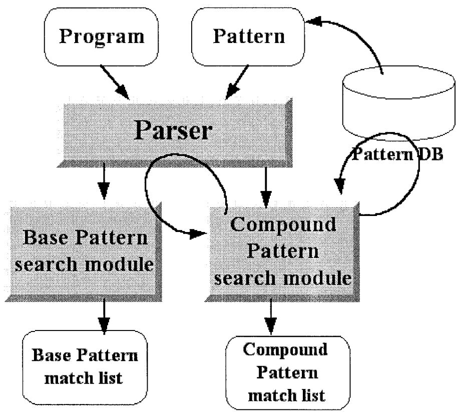

Program
Pattern
Parser
Pattern DB
Base Pattern search module
Compound Pattern search module
Base Pattern match list
Compound Pattern match list

Fig. 2 The architecture of the system.

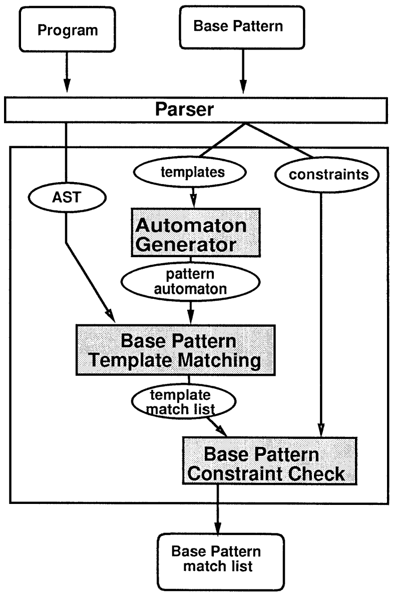


Fig. 3 Base pattern recognition flow.

5. Performance: Detection of Bad Pattern

In order to evaluate whether our system is useful as a diagnosis system of programming styles, we made the following two experiments;

1. Description experiment: The aim of this experiment is to evaluate description capability of program patterns. We investigate whether we can define various kinds of style guidelines with program patterns.

2. Recognition experiment: The aim of this experiment is to investigate whether this system can find the intended program fragments in practical programs.

5.1 Description Experiment

As a target of the style guideline, the following guidelines were used:

- Six syntactic guidelines and seven semantic guidelines in [4].
- Nine guidelines that are used in a programming course at our university.

In the description of the syntactic guidelines, we can define five guidelines out of six using 12 program patterns which are concerned with the following guidelines: priority between operators, missing or extra semicolon, `case` label without `break` statement, function call without parameter list, and dangling `else`. We have not yet prepared pattern recognition about declaration, so our system can't deal with the guidelines about function definition. However the expansion of the system is easy.

In the description of the semantic guidelines, we can define four kinds of guidelines out of seven using seven program patterns which are concerned with the following concepts: use of pointer variables, boundary usage, check of return variables, and evaluation order. There are three guidelines which we can't describe. These guidelines are also concerned with declaration.

In our style guideline, we described nine kinds of guidelines using 19 program patterns.

As the result, we can define 18 guidelines using 38 program patterns within the 22 guidelines,

5.2 Recognition Experiment

As a target of recognition, a program set of the GNU text utility library (for example, cat, head, wc, and so on) is selected and 38 program patterns which were described in the description experiment were used.

We tried on 22 programs using 38 program patterns. The average number of program pattern found for one program is 45.7 (the maximum is 162, the minimum is 7). The number of program fragments found per one program pattern is 26.5 on the average (the maximum is 195, the minimum is 0). Besides, the average number of line of source code without comment is 497 (the maximum is 1354, the minimum is 107).

In order to evaluate description capability of program pattern, we investigated whether the system recognizes all program fragments which experts can recognize manually. The result of this investigation shows that system's match list agrees with the program fragments which experts recognized. From this, we ascertained that the system can recognize all guidelines correctly. This result indicated that our description method can describe a guideline of programming styles correctly. This result also shows that the system can recognize various patterns, for example, context dependent patterns and dispersed patterns spanning multiple functions. It is difficult to recognize these patterns by conventional character based pattern matching methods.

In the first try of recognition experiment, there were guidelines that don't match all the intended elements. We solved this problem by revising conditions or adding new constraints. We can do this easily as local improvement. This shows our system's extensibility.

6. Related Work

The source code search tool SCRUPLE [3] uses an original pattern languages. In SCRUPLE, the pattern language supports a rich set of features including named and unnamed wild-cards, matching of high-level data types such as sets and sequences, etc. It uses a pattern recognizer based on a finite state machine. We use this technique in template matching of base patterns, because it can detect pattern in data driven manner. SCRUPLE introduces concept of constraints, but it does not have generality because it only uses string constraints. In our system, the framework of constraints was expanded to use structured constraints and flow constraints.

On the other hand, there are some programming style analyzers, for example, Code Analyzer for Pascal (CAP) [5] and STYLE [6]. CAP is an automated self-assessment tool to aid novice student programmers. STYLE outputs messages about programming styles based on six quality metrics: economy, modularity, simplicity, structure, documentation, and layout. These systems center on statistics quantity, indentation and comment style, and they don't deal with structural patterns and context dependent patterns. These systems realized each recognition techniques as procedures, so expansion of systems is not so easy. In our method, the system becomes to detect the new guideline by adding only a program pattern.

There is a software tool Fall-in C which detects pitfalls in C programs [7]. We compare the description in Fall-in C and the description in our system. The example of "`case` label without `break`" in our system

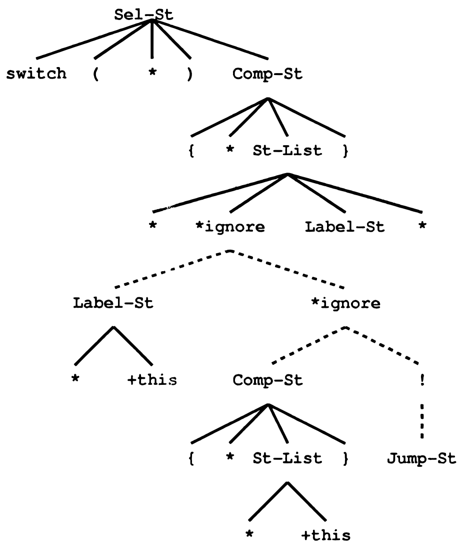


Fig. 4 Sample pattern used in fall-in C.

is as shown below.

```
BEGIN
case_pattern;
@v1
BODY
  @v1
  case # : @
  @ \ast
CONSTRAINTS
  not_jump_st(@v1)
END
```

The description of the same example in Fall-in C is shown in Fig. 4. In Fall-in C, a pitfall pattern for pattern matching is tree structure whose nodes are either C program elements or special symbols. In the case of tree structure, a teacher who describes a pattern needs to understand a grammatical structure neatly, so it is difficult to describe a pattern. In our system, a pattern is described using expanded C statements. For this, it is easy to describe a pattern. Moreover, because of introduction of constraints, description of patterns becomes simple and it is easy to write/understand a guideline by a teacher who describes a pattern. This is important to use our system generally. In Fall-in C, a pattern of code spanning multiple functions and a pattern that specifies lack of some program fragments cannot be described. In our system, as shown in the description of [guideline about file processing], we can describe the patterns of code spanning multiple functions using compound patterns. Also, we can describe the patterns that specify lack of some program fragments using the *not_found* constraint.

7. Experiment: Use by Novice Programmers

In programming style analyzers [5]–[7], some researchers have evaluated the performance but have not examined in real situation. In order to evaluate whether novice programmers were assisted by our system, we made experiment of bad patterns detection.

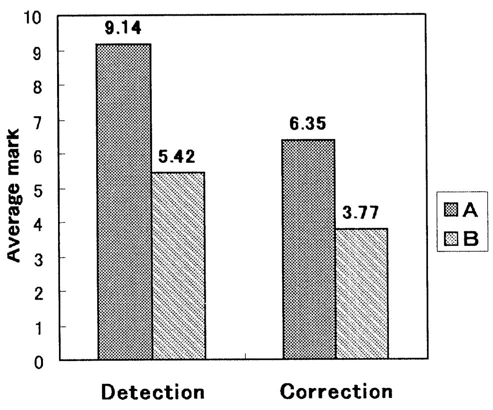


Fig. 5 Results of the experiment about style inspection.

7.1 Overview

Testees were 86 sophomore students in Shinshu University and they have one year programming experience. First, to divide testees between two groups properly (that is, A and B group), we give testees a test in programming. Next, we conducted the following experiment. Testees are requested to detect/correct the following five bad patterns in a target program.

1. Guideline about constants
2. Guideline about side effects within conditional expressions
3. Guideline about increment/decrement operators
4. Guideline about file processing
5. Guideline about conditional expressions

The objective program has about 100 lines and it's function is to merge two sorted data sequences. In addition, this program has the above mentioned five kinds of bad patterns. Group A does this experiment using this system and group B without using it.

7.2 Results and Discussions

We graded the results out of 10 according to the following two points. 1) Detection: Do students detect bad program fragments correctly? 2) Correction: Do students modify bad program fragments correctly? In Fig. 5, we show the average mark. From this result, in both detection and correction, the average mark of group A that uses the system is higher than the average mark of group B that doesn't use it. By the two-tailed test at a 0.01 level of significance, we confirmed that the above hypothesis is highly significant. In addition, the average time to do this experiment is 15.3 minute (in A group) and 17.5 minute (in B group). The result shows that the time to detect/correct bad patterns became short.

In conventional programming courses, teachers need to check the student's programs and to teach the programming style interactively. By this experiment, the following assumption was confirmed: Novice programmers can check programming styles efficiently by themselves using our diagnosis system.

8. Conclusion

We have proposed a diagnosis system that detects program fragments which infringe the programming style in C programs. In this system, style guidelines were described using the proposed program pattern. The input is a target program and this pattern.

The following was confirmed experimentally: 1) This system can recognize various guidelines correctly; 2) This system is effective for assisting novice programmers to detect/correct of bad patterns. By using this system, performance (average mark and average time) is improved.

We have treated the programming styles of some literatures. In order to improve the performance of our system, it is important to grasp stereotypical kinds of bad patterns. In the future, we will analyze the bad pattern in C programs written by novice programmers and will improve the pattern database. Moreover, we need to strengthen the description capability of our system.

References

[1] R. Sekimoto and K. Kaijiri, "Plan representation and its recognition approach for program recognition," Proc. JCKBSE'96, pp.198–201, Sept. 1996.

[2] R. Sekimoto and K. Kaijiri, "A detection of ill-formed patterns about programming style," Proc. JCKBSE'98, IOS Press, pp.165–168, Sept. 1998.

[3] S. Paul and A. Prakash, "A framework for source code search using programming patterns," IEEE Trans. Software Eng., vol.20, no.6, pp.463–475, June 1994.

[4] A. Koenig, C traps and pitfalls, Addison-Wesley, 1989.

[5] T. Schorsch, "CAP: An automated self-assessment tool to check pascal programs for syntax, logic and style errors," Proc. SIGCSE'95, pp.168–172, March 1995.

[6] A. Lake and C. Cook, "STYLE — An automated program style analyzer for Pascal," SIGCSE Bulletin, vol.22, no.3, Sept. 1990.

[7] M. Oda and T. Kakeshita, "Pitfall detection of C programs using pattern matching," Trans. IPS Japan, vol.35, no.11, Nov. 1994.

Rika Sekimoto received the Master of Engineering at Tokyo Denki University in 1992. She research now at Shinshu University as a research associate. Her current interests include program recognition and programming education support environment. She is member of IPS of Japan.

Kenji Kaijiri received the Ph.D. degrees in Communication Engineering from Osaka University in 1977. In 1977, he was a Research Associate of Information Engineering, Shinshu University, and in 1978 he was an Associate Professor, and in 1995 he was a Professor. His research area is software engineering, programming languages, and distance learning. He is a member of the IPS and JSISE of Japan, and IEEE, ACM.