# A detection of ill-formed patterns about programming style

Rika SEKIMOTO and Kenji KAIJIRI

*Department of Information Engineering, Faculty of Engineering*
*Shinshu University, 500 Wakasato, Nagano 380-8553, JAPAN*
*E-mail:rika@cs.shinshu-u.ac.jp*

**Abstract:** Programming style plays an important role in program understanding and maintenance. We have implemented a plan recognition system for source code search, and have attempted detection of ill-formed patterns about programming style through it. In this paper, we present experimental result obtained from a prototype system. We ascertained that our system is able to diagnose various items about programming style, so it may be used as an effective tool of programming style diagnosis.

## *1. Introduction*

In programming, it is important to keep a coding standard and to implement an understandable program for everyone. The guideline to produce code that is clear and easily understood without sacrificing performance is called programming style. In particular, programming style plays an important role in program education for novice programmers and in program development by many programmers. By obeying the programming style, quality of the program and maintenance efficiency will increase.

We aim at developing a system which diagnoses programming style. It assists programmers to obey the programming style. Our system detects program fragments, which match the ill-formed patterns against programming style within target programs, and outputs the diagnosis message. The target language is ANSI C.

We have already proposed program plan representation for source code search, and have implemented a plan recognition system that detects program fragments which match plans [1, 2]. We attempted the recognition of ill-formed patterns about programming style through this plan recognition system.

Several programming style analyzers have been developed. For example, there are Code Analyzer for Pascal (CAP) [3] and STYLE [4]. CAP is an automated self-assessment tool to aid novice student programmers. STYLE outputs messages about the programming style based on six-quality metrics: documentation, layout, etc. These systems center on statistics quantity, indentation and comment style analysis and they don't deal with structural patterns and context dependent patterns. These systems realized each analysis as procedures, so there are problems about extensibility of systems. In our method, we assume that a check item of programming style is a search pattern, and define it as a plan. The system detects program fragments against each programming style by searching plan. When we'd like to add new check item, we need only to define it as plans. Furthermore our system can deal with structural patterns and context dependent patterns.

In this paper, we describe the description experiment of ill-formed patterns about programming style and recognition experiment for programs of the GNU text utility library.

## *2. Overview of plan recognition*

We have proposed the representation method for source code search using programming plan, and have implemented a plan recognition system which locates source code fragments that are matched with given plans.

Our proposed plan consists of plan 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. From the description form of the template, a program plan is classified into two types: fundamental plans and compound plans. A fundamental plan is assumed to be a sentential form of the C grammar. A compound plan consists of sets of instances of fundamental plans.

In the recognition process of plan, at first, system tries to compare the template in body part with target program. In template matching of fundamental plans, we use technique based on the finite state automaton proposed by [5]. The template is transformed into an extended non-deterministic finite state automaton called plan automaton (PA). The system simulates a PA on a source program which is transformed into an abstract syntax tree. We presented the recognition mechanism in [2]. In the template matching of compound plans, we use the fundamental plan recognition module. Next, program fragments that matched templates are checked about context condition in constraint part. For this separation of process, addition of the new condition becomes easy. Furthermore, the complex matching technique specialized to various patterns need not be done, so the template matching algorithm becomes simple and well-formed.

## *3. Diagnosis of programming style*

We apply the plan recognition system to detect of ill-formed patterns about programming style. We have defined various kinds of ill-formed pattern using plan format, and done recognition experiment for programs in the GNU text utility library.

### *3.1 Description experiment*

The aim of this description experiment is as follows: 1) To investigate whether we can define various kinds of item about programming style with our proposed notation; 2) To investigate what kind of item we can’t define. The programming style we used is in [6], and we also used the programming style that is used in programming course at our university.

In [6], some pitfalls, which can’t be detected by compilers and may cause serious errors, were collected. They were classified into seven categories: lexical pitfalls, syntactic pitfalls, semantic pitfalls, pitfalls about linkage, pitfalls about library functions, pitfalls about pre-processor, and portability pitfalls. In this experiment, we targeted syntactic and semantic pitfalls.

In the description of the syntactic pitfalls, we can define five kinds of concepts out of six. We defined 12 plans which are concerned with the following concepts: priority between operators, missing or extra semicolon, case label without break, and dangling else. We have not yet prepared pattern variable about declaration, so we can't define the concept about function definition.

In the description of the semantic pitfalls, we can define three kinds of concepts out of seven. We defined seven plans, which are concerned with the following concepts: use of pointer variable, boundary usage, and evaluation order. There are some concepts which we can't define, and these are concerned with declaration. In check of return variables, it is possible to check it procedural, but this method is not adequate for our plan representation style.

In our style guideline, we defined 19 plans about syntax and semantics.

Examples for plan definition are shown below.

```
[Example 1]
BEGIN
priority1_plan;
#v1
BODY COMPLEX
  l1:assignment_plan();
  l2:equality_exp_plan(#v1)
CONSTRAINT
  with_in(l2, l1)
END
```

```
[Example 2]
BEGIN
multi_assign_plan;
$v1, $v2
BODY COMPLEX
  l1:assign_plan($v1);
  l2:assign_plan($v2)
CONSTRAINT
  same_variable($v1,$v2),
  different_function($v1,$v2)
END
```

```
[Example 3]
  BEGIN
file_not_close_plan;;
$v1,$v2
BODY COMPLEX
    l1:file_open_plan($v1);
    l2:file_close_plan($v2)
CONSTRAINTS
      found(l1),
      not_found(l2)
END
```

The [priority1_plan] will match an assignment statement that includes an equality operator. The [multi_assign_plan] will match some program fragments that assign a value to the same variable in different function. The [file_not_close_plan] will detect the pattern that uses “fopen” function calls but does not use “fclose” function calls. Some component plans used are defined as follows;

```
BEGIN
assign_plan;
$v1
BODY
    $v1 = #;
END
```

```
BEGIN
file_open_plan;
$v1
BODY
    $v1 = fopen(#);
END
```

```
BEGIN
file_close_plan;
$v1
BODY
    fclose($v1);
END
```

### *3.2 Recognition experiment*

The aim of this recognition experiment is as follows: 1) To investigate whether this system can find the intended program fragments concerning programming style; 2) To investigate whether the plan recognition method is effective for the check of ill-formed patterns about programming style. We used plans which were described in the former section as the target plan data. The total number of plans is 38. The target of this experiment is a program set of the GNU text utility library.

We tried on 22 programs using 38 plan data. The number of matched elements per one plan is 26.5 on the average (the maximum is 195, the minimum is 0). The average number of line of source code without comment is 497 (the maximum is 1354, the minimum is 107).

### *3.3 Considerations*

#### *3.3.1 Recognizable pattern*

In this experiment, we ascertained that we could define the following patterns easily, and our system could recognize them: 1) context dependent pattern (e.g. Example 1); 2) dispersed pattern in two or more functions (e.g. Example 2); 3) pattern that specifies lack of some component plans (e.g. Example 3). It is difficult to recognize them by pattern matching based on the character string in the text.

### *3.3.2 Openness of our system*

In the first recognition experiment, there were some plans that matched verbose program fragments. We investigated the reason, and solved this problem by modification of context condition. We can add some new condition easily as local improvement, so we also ascertained that this system has openness.

For example, we consider the case that there are two same component plans in the compound plan (e.g. Example 2). In this case, there were two matched elements which have the same set of program fragments. We added a condition using before_text constraint which restricts program position of matched program fragments. As a result, we were able to omit verbose matched elements.

### *3.3.3 Problems about representation*

As the representation of body part, there are problems that the lack of representation about the concept of OR and abbreviation. For the lack of such representation methods, there are cases that we must describe a plan for similar pattern many times. If we can use these representations, we are able to define some plans as a single plan, and the number of plan which the system should try search decreases in diagnosis of similar pattern.

## *4. Conclusions*

We attempted the detection of ill-formed pattern about programming style through our proposed plan recognition method. In the description and recognition experiment, we ascertained the following matters. 1) The diagnosis of programming style is possible through plan recognition method. 2) The recognition for the dispersed or context dependent patterns is possible. 3) The expansion of system is easy by adding new conditions. In this system, check item was defined as a plan, so it is easy to add new query patterns. Therefore this system has extensibility and maintainability.

To make this system more powerful, we need to spread the variation of recognizable patterns. For example, in the compound plan recognition, we need new condition representation and recognition method for matched list as a whole. We also should evaluate the interface or execution time by the practical point of view.

### *References*


[1] Kenji Kaijiri, “Support of plan library construction,” JCKBSE'94 (1994).

[2] Rika Sekimoto and Kenji Kaijiri, “Plan Representation and Its Recognition Approach for Program Recognition,” JCKBSE'96 (1996).

[3] Tom Schorsch, “ CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors,” SIGCSE '95 (1995).

[4] Al Lake and Curtis Cook, “STYLE -An automated program style analyzer for Pascal-,” SIGCSE Bulletin, Vol. 22, No.3 (1990).

[5] S. Paul and A. Prakash, “A Framework for Source Code Search Using Programming Patterns,” IEEE Trans. Software Eng., Vol.20, No.6 (1994).
[6] A. Koenig, “C Traps and Pitfalls,” Addison-Wesley (1989).