An attempt to increase software quality by detecting irregular styles

Masayoshi Sakakura

The Centre for Informatics
Waseda University
Tokyo 169, Japan

Yoshiaki Fukazawa

Dept. of Information & Computer Science
Waseda University
Tokyo 169, Japan

Abstract

A static program analyzer has been designed and developed. Even though it is natural that large-scale software is developed by a number of groups under standard guidelines, it is often convenient that each group have group-oriented guidelines in addition to the standard guidelines. This is because the standard guidelines do not reflect factors unique to each group. Our tool regards coding styles as a set of static properties of a program. With this tool, the user can define such static properties to be detected, called irregular styles. This paper describes the definition of an irregular style, a detection strategy, and evaluation of its results.

1 Introduction

A number of methodologies have been proposed to increase reliability of large-scale software. Among them, it is plausible to adopt standard guidelines in each step of the software development process[1,2]. Use of some tools has been attempted to support software development activities under standard guidelines. Nevertheless, according to our experience with large-scale software development, it is necessary to have group-oriented guidelines, that is, rules unique to each group in addition to the standard guidelines[3]. This is because standard guidelines do not reflect factors unique to each group, such as the aim of the system, development environments, knowledge and skill of staff and so on. This means that even though a program structure is considered to be normal in a project, the same structure may be considered to be anormal in other project. Following cases are these practical examples.

- For each programming language, some standards are proposed by standardization organization like ISO, ANSI and so on. But almost all commercial language processors have more facilities than these standards. For the purpose of implementation of highly portable software, these extended facilities, which do not contained in every standards, must not be used.

- In case that future extensions are expected, a redundant representation may be permitted. For example, following segment is usually redundant.

switch := 1; if switch = 1 then prog-a;

But if the variable switch is assured to have many kinds of values by a future extension, above segment is suitable.

- Programs are used as a communication media for various kinds of project members. Therefore their know-hows and skills must be considered. For example, in software development by an organization which contains many beginners, too complicated software structure should be avoided.

Therefore, it is desired that we have an automatic tool that is adaptable to the rules of each group. We have designed and developed a static program analyzer. In this tool, a user can easily define static properties of program to be detected. A set of defined static properties reflect the target of the attained software properties. The tool is intended to detect violations automatically in the coding stage of the software development process.

2 Concept of the tool

With this tool, we regard the coding style at a coding stage as a static attribute of a program, and call the static attribute to be detected ***an irregular style*** of

0-8186-4240-8/93 $03.00 © 1993 IEEE

the program. The user can define irregular styles by describing their static properties. The tool analyzes a target program statically, detects positions corresponding to the models and reports them.

To describe an irregular style, it is essential for the tool to have the following features:

- User can describe irregular styles as naturally as possible.

- Behavior of values of variables and mutual relationships among them can be described.

In order to realize these features, the tool is designed using the following strategies:

- The description of a model is divided into a structure description and a condition description.

- A symbolic execution facility is adopted to investigate relationships of variables[4].

An irregular style recognized by the tool is an attribute detectable at the syntactic level. Therefore, it does not recognize the following rules:

- A rule not detectable on a parsing tree, such as a suitable comment or a program indentation.

- A rule not recognizable by the source program, for example, whether or not the name of a variable represents its intrinsic meaning faithfully.

These are explained in the following section. We have selected Pascal as our prototype language because it facilitates description of the syntax of a model and can be used conveniently for the description of a model of an irregular style.

3 Definition of an irregular style

An irregular style is defined in two ways: a condition satisfied by a portion of a program and its relative positioning in a program control structure. They are called the condition description and structure description, respectively. Fig.1 shows the structure of the irregular style.

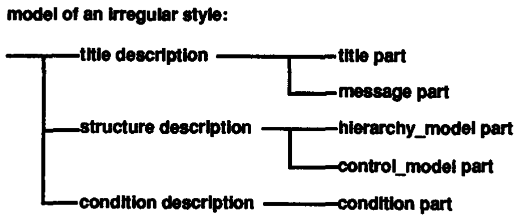

Fig.1 The structure of an irregular style

The title description consists of a title part and a message part. The name of an irregular style is specified in the title part. The message in the message part is issued when an irregular style is detected.

A structure description is specified when an irregular style has a relationship to the control structure or when relative positions, in which conditions are satisfied, must be specified. The structure of a program can be specified by a hierarchy module structure, and/or by three basic control structures: sequence, branch and iteration. These structures are specified in the hierarchy_model part and in the control_model part, respectively. The specification of the structure is based on graphical representation because it is easy to understand.

In the hierarchy_model part, a node represents a procedure or a function (the generalized term "module" is used hereafter) while an arc represents its relationship. Node numbers, distinction codes of a module, and arcs to lower modules are specified in the hierarchy_model.

In the control_model part, a node represents a control structure such as a branch, a specific position such as an entry or an exit of a program, or a portion of a program in which conditions are satisfied. If there exists a control flow from one portion of a program to another, the existence of a path between them is assumed. An arc represents such a path.

In the condition description part, a condition is specified which must be satisfied at each node in the structure description part or must be satisfied independently of the structure description. A predicate expression is used to represent the condition because it is easy to understand. The predicate used in this part is called *a model predicate*, and the variable used as its argument is *a model variable*. The behavior of a model variable is similar to that of a variable in Prolog. A condition is described by model predicates. Model predicates are categorized into two types: one expresses behavior such as "substitution" or "reference", and the other represents static attributes of arguments such as "type". The condition is satisfied if the model predicate matches the specific pattern of a parsing tree or if the value of a variable in the predicate satisfies a logical relationship in the program. Although general model predicates are prepared in the tool, the user can add new model predicates if necessary.

3.1 Examples of irregular styles

Examples of irregular styles are shown in Figs.2~4. Fig.2 shows the model of an irregular style in which there exists a variable reassigned after an assignment which is not referenced thereafter. In this model, it is not necessary to describe the hierarchy_model part since the model is independent of hierarchical relationships of modules. The specification in the control_model part indicates that there are two arbitrary nodes in a program and a path between them. These nodes are node 1 and node 2. The condition part indicates that the same variable is assigned at node 1 and node 2, and that the variable is not referenced between these nodes. The model predicate "assign" is satisfied in an assignment statement, and returns a variable as its first argument and an expression as its second argument. The predicate "varref" is satisfied for a variable reference, and a variable is returned as its argument. "#a" is a model variable, while "#_" is an anonymous variable as in Prolog.

```
title : reassign_without_reference
message : 'reassign some value to a variable before
referencing it'
hierarchy_model :
    (
    )
control_model :
    ( 1 : free
      2 : free
    )
condition :
    ( ( , 1 :      assign(#a,#_) ) and
      ( , 1~2 : noexist varref(#a) ) and
      ( , 2 :      assign(#a,#_) )
    )
```

Fig.2 A model description of the irregular style "reassign_without_reference"

Fig.3 shows an irregular style in which a value is assigned to a global variable in a module. The hierarchy_model part indicates that the type of module is arbitrary. Since a condition must be satisfied within the same module, there is no description for a lower hierarchy. "Mstart" and "mend" in the control_model part are the entry and exit of the module, respectively. The model predicate "var_no" is satisfied for each variable, and returns the variable number. The variable number is a serial number assigned to each variable in a program. "Var_def" and "fpara_def" are model predicates which are satisfied in a variable definition and in an argument definition, respectively, and return variable numbers as their second arguments. The condition description indicates that a value is assigned to either an undeclared variable or a variable not in an argument, in the module.

```
title : assign_global_variable
message : 'assign to global variable in submodule'
hierarchy_model :
    ( 1 : ( free )
    )
control_model :
    ( 1 : mstart
      2 : free
      3 : mend
    )
condition :
    ( ( 1 , 2 : assign(#a,#_) and
            var_no(#a,#no) ) and
      ( 1 , 1~3 : noexist ( var_def(#_,#no,#_) or
            fpara_def(#_,#no,#_,#_) ) )
    )
```

Fig.3 A model description of the irregular style "assign_global_variable"

Fig.4 shows an irregular style in which two expressions have the same value for any input value. The predicate "relation" is a model predicate and it is determined as a result of symbolic execution. The predicate has an expression which contains model variables as an argument, and is satisfied if the expression satisfies a specified relationship. Moreover, the model predicate is qualified by the quantifier "for_any_input" which restricts two expressions to same value for any input value.

```
title : same_condition
message : 'inappropriate branch structure'
hierarchy_model :
    (
    )
control_model :
    ( 1 : branch()
      2 : branch()
    )
condition :
    ( ( , 1 : br_cond( #a ) ) and
      ( , 2 : br_cond( #b ) ) and
      ( , ~ : relation( #a = #b ) for_any_input )
    )
```

Fig.4 A model description of the irregular style "same_condition"

4 Detection of an irregular style

An outline of the detection mechanism of the tool is described. First, for each node in the control_model part, the portion of a program which satisfies a condition is searched and evaluated to see if it satisfies a positional relationship with other nodes. Next, a condition which is independently specified to a node in the control_model part or a condition as to whether or not it satisfies a structure in a hierarchy_model is evaluated. Fig.5 shows the general flow of the tool. The following is an explanation of execution of a model predicate, symbolic execution and the result of execution.

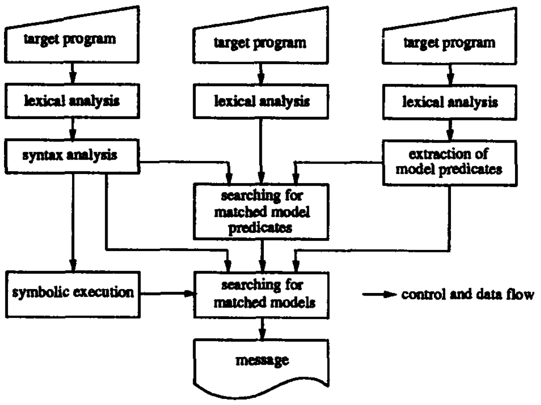

Fig.5 Outline of the tool

4.1 Execution of a model predicate

Fig.6 shows an example that indicates a correspondence between a model predicate and the specific pattern of a parsing tree in which the model predicate is satisfied. The result of the parsing is represented in tree format, in which each syntactic element is its node. A model predicate corresponds to the pattern of a parsing tree by describing what syntactic element has a child node. In Fig.6(a), the model predicate "br_cond" always holds in "if statement", "while statement" or "repeat statement", and returns an expression to the first argument of the model predicate. In Fig.6(b), part of the model predicate "var" holds if a unique variable is used in the expression. In Fig.6(c), the model predicate "varref" is satisfied for each variable contained in an expression, and returns the variable to the first argument.

The argument of the model predicate "relation" in Fig.4 is an expression in which the model variable is contained. The tool performs symbolic execution for a target program if necessary and determines its success or failure of the predicate "relation".

```
br_cond(1)
extent(target) :
    if_sta(
            1(1) : exp
            2 : free
            3 : free
    )
br_cond(1)
extent(target) :
    while_sta(
              1(1) : exp
              2 : free
    )
br_cond(1)
extent(target) :
    repeat_sta(
              1 : free
              2 : free
              3(1) : exp
    )
```

(a) Definition of the model predicate "br_cond"

```
var(2)
extent(target) :
    exp {
        1 : simp_expl {
            1 : term {
                1 : factorl {
                    1(2) : free
                }
                2 : =0
                3 : free
            }
            2 : =0
            3 : free
        }
        2 : =0
        3 : free
        4 : free
    }
```

(b) Definition of the model predicate "var"(part)

```
varref(1)
extent(target) :
    exp {
        include(1) : var
    }
```

(c) Definition of the model predicate "varref"

Fig.6 Examples of descriptions of model predicates

4.2 Symbolic execution

Symbolic execution is performed to investigate the behaviour of a variable and its mutual relationships. Capabilities of the symbolic executor are as follows :

- executes all paths in branching
- executes zero times or one time for an unpredictable number of iterations
- in an array variable, sets the value of the array to "undefined" if the subscript value is indefinite.

4.3 Results of execution

Fig.8 shows the result of execution in which the irregular style shown in Fig.4 is detected from the target program shown in Fig.7. Fig.9 shows the correspondence between the target program and its control structure graph. The current version of the tool produces the node number of the control structure graph which corresponds to nodes in the structure description. A model variable is also produced as part of the parsing tree. It may be possible to produce a source program directly if a control structure graph includes source information.

```
program test(input);
var a,b,c : real;
    procedure sub( x,y : real );
        begin
            if ( x < 0 ) then y := x
                         else y := -x
        end;
begin
    a := 1;
    b := 2;
    if ( a > b ) then sub(a,c) else sub(b,c);
    if ( a < 0 ) then c := -c;
    c := c*2
end.
```

Fig.7 Example of target program

```
message  :  'inappropriate branch structure'

hno    cno   path    location in gr
free_  cn1   [9]     3-3  at  [3,6,3,1,3,7,2,3,1,2]
free_  cn2   [ ]     13-13  at  [3,7,2,3,4,2]
model variable
#a : [exp,[simp_exp1,[term,[factor3,[exp,[simp_exp1,
     [term,[factor1,[var,4,0,[ ] ] ],0,[ ] ],0,[ ] ],1,
     [rel_op,small],[simp_exp1,[term,[factor2,
     [number,0] ],0,[ ] ],0,[ ] ] ] ],0,[ ] ],0,[ ] ],0,[ ],[ ] ]
#b : [exp,[simp_exp1,[term,[factor3,[exp,[simp_exp1,
     [term,[factor1,[var,1,0,[ ] ] ],0,[ ] ],0,[ ] ],1,
     [rel_op,small],[simp_exp1,[term,[factor2,
     [number,0] ],0,[ ] ],0,[ ] ] ] ],0,[ ] ],0,[ ] ],0,[ ],[ ] ]
```

Fig.8 Result of execution

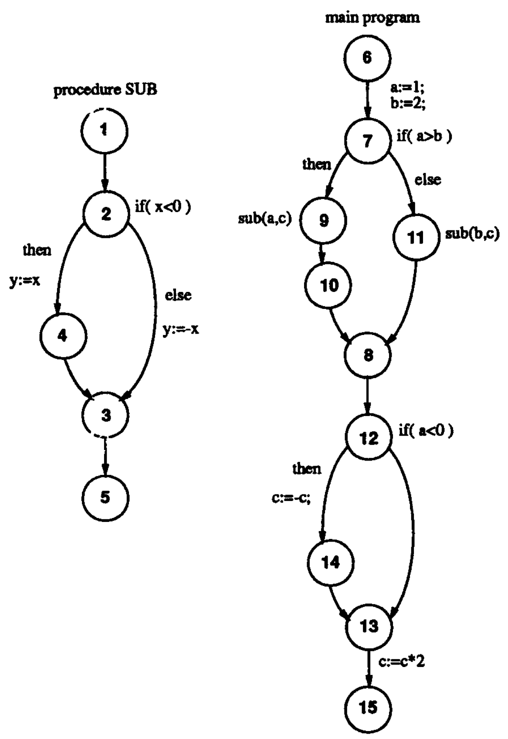

Fig.9 Correspondence between a program and its control structure graph

5 Evaluation

Our method has been evaluated from the viewpoints of feasibility, ease of creating and readability of model description.

5.1 Feasibility of model description

Table 1 shows the feasibility of the description of the rules under general coding styles[5]. Among these rules, some rules for a specific function, for example, error handling or initial setting of peripheral devices such as a printer, cannot be specified. It is difficult to describe these functions using our current method. As there is a research to describe these specific functions on a syntactic level[6], it may be possible to increase describability if such an attempt is successful. Regarding rules for comments, which are independent of content, it is possible to describe if rules are modified to reflect positions of comments in parsing trees.

In order to show that built-in model predicates are sufficient, model predicate utilization is listed in Table 2. Table 2 shows the types of model predicates used to describe the rules in Table 1 as models of irregular styles. Since a user-defined model predicate is not referenced more often than these three irregular styles, the current built-in model predicates are sufficient.

5.2 Ease of creating a model description

Next, a user-level tester attempted to define a specified irregular style so as to ensure ease of creating a description. As the result of the experiment, the following was made clear:

- It is possible to associate the semantics of an irregular style with a model of the irregular style.

- Although there is almost no problem with creating a structure description, it is necessary to have a basic knowledge of syntax analysis to create a condition description.

Regarding to the latter, we show an example in Fig.10. Fig.10(a) shows that after a variable is assigned a value, another variable is assigned the value of the first variable. In this case, it is necessary to describe these statuses as in Fig.10(b). In comparison, the tester attempted to define model predicates as in Fig.10(c). In a parsing tree, the right-hand side of an assignment statement is treated as an expression even if it consists of only a single variable. Therefore, although an expression is returned as the second argument of the model predicate "assign", it is often mistaken that a variable should be returned to it. The model predicate "var" takes an expression, a simple expression, a term or a factor as its first argument and, if it is a simple variable, it returns the variable as its second argument. Current efforts to exclude these mistakes involve illustrating these examples, emphasizing the difference between an expression and a variable, and training the tester.

Table 1 Possible model descriptions

object of rule	#1	#2	#2/#1	type of rule which is impossible to describe				
				com.	not.	fun.	mean.	others
reliability	8	7	.87	0	0	1	0	
simplicity	9	6	.67	1	0	0	0	redundancy
completeness	8	5	.63	1	0	2	0	
inconsistency	9	4	.44	1	1	0	3	
portability	10	4	.40	1	0	3	0	
validity of structure	12	4	.33	1	3	2	1	interface rule
ease of testing	15	4	.27	2	0	6	0	output format
understandability	21	3	.14	11	3	1	2	redundancy
servicesability	16	2	.13	5	4	2	2	module
usability	6	0	.00	0	0	3	2	output format
total	114	39	.34	23	11	20	10	

#1 : number of rules
#2 : number of rules which are impossible to describe
com. : a rule for existence and content of a comment
not. : a rule for notation such as indentation

fun. : a rule for a specific function such as error handling or initial setting
mean. : a rule for meaning of a variable

Table 2 Types of model predicates used

no. of model predicates of irregular styles used	0	1-2	more than 2	total
no. of model predicates of irregular styles predicted	47	7	14	68
no. of model predicates of irregular styles added	-	8	0	8
total	47	15	14	76

```
node1 : x := y+z;
node2 : u := x;
```

(a) Part of a program

```
node1 : assign(#a,#_)
node2 : assign(#_,#b) and var(#b,#a)
```

(b) Correct description of a model predicate

```
node1 : assign(#a,#_)
node2 : assign(#_,#a)
```

(c) Incorrect description of a model predicate

Fig.10 Example of mistakes in condition description

5.3 Readability of a model description

The degree of correct understanding of an irregular style is estimated to be about 90 percent from irregular style models. This is currently sufficient for a description on the syntactic level even when we do not attempt to understand the specification and the meaning of the target program. Therefore general users will find it easy to generate descriptions.

6 Conclusion

The tool automatically detects a violation of rules. Its other applications are considered to be as follows:

- A debugger representing debugging knowledge accumulated in a project.

- Usage checker for special syntactic elements for portability.

- Finder of missing functions or checker of expected errors when a problem is fixed, if used in program education[7].

Finally, the following extensions are expected:

- Obtain an extraction method of an abstract function of target program.

- Include learning facilities which reflect results of previous program analyses.

If these functions are incorporated, we will obtain a tool which is more powerful and easier to use.

References

[1] Y.Mizuno and M.Azuma, "Standardization of computer software", *Nihon Keizai Shinbunsha*, 1977, in Japanese.

[2] "SDEM : Software Development Engineer's Methodology", *FACOM Journal*, Vol. 3, No. 12, pp. 25-31, 1977.

[3] K.Shirai and M.Azuma, "Administration System Standard Manual", *Nikkan Kogyo Shinbunsha*, 1977, in Japanese.

[4] T.Tamai and K.Fukunaga, "Symbolic execution system", *Information Processing*, Vol. 23, No. 1, pp. 18-28, 1982, in Japanese.

[5] I.Miyamoto, "Software engineering", *TBS publishing*, 1982, in Japanese.

[6] R.L.Sedlmeyer, "Knowledge-based Fault Localization in Debugging", *Proc. of Software Engineering Symposium on High-Level Debugging*, pp. 25-31, 1983.

[7] W.L.Johnson, "PROUST:Knowledge_Based Program Understanding", *IEEE Trans. on Soft. Eng.*, Vol. SE-11, No.3, pp. 267-275, 1985.