Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
The Journal of Systems and Software 78 (2005) 101–110
The Journal of Systems and Software
www.elsevier.com/locate/jss

Automatic detection and correction of programming faults for software applications

Prattana Deeprasertkul [a,*], Pattarasinee Bhattarakosol [a], Fergus O'Brien [b]

[a] *Department of Mathematics, Faculty of Science, Chulalongkorn University, Phayathai Road, Patumwan, Bangkok 10330, Thailand*
[b] *School of Information Technology, Faculty of Informatics and Communication, Rockhampton Campus, Central Queensland University, Australia*

Received 29 September 2004; received in revised form 9 February 2005; accepted 10 February 2005
Available online 8 April 2005

Abstract

Software reliability is an important feature of a good software implementation. However some faults which cause software unreliability are not detected during the development stages, and these faults create unexpected problems for users whenever they arise. At present most of the current techniques detect faults while a software is running. These techniques interrupt the software process when a fault occurs, and require some forms of restart.

In this paper *Precompiled Fault Detection* (PFD) technique is proposed to detect and correct faults before a source code is compiled. The objective of the PFD technique is to increase software reliability without increasing the programmers' responsibilities. The concepts of "pre-compilation" and "pattern matching" are applied to PFD in order to reduce the risk of significant damage during execution period. This technique can completely eliminate the significant faults in a software and thus, improves software reliability.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Programming error; Software fault; Software failure; Fault detection; Pattern matching; Software inspection

1. Introduction

The task of implementing a program without faults and errors is challenging. Currently, the various compilers for languages have been progressively improved. However, some faults and errors which are the results of human oversight are still left out and interrupt the system processing at operation time. The existence of the faults in applications can increase the number of software failures and can thus decrease the reliability of software. Of course, the software reliability is improved if the risks of software failure are avoided.

Achieving reliable software is an objective of developers and users. In order to prevent such faults and errors, programmers and software inspectors must verify software for all possible faults during the development stages, and also validate the software product before delivering it. Therefore, it challenges researchers to develop methods or techniques to detect or prevent the faults during development period in order to obtain a high level of reliability for software product.

Currently many software detection techniques have been proposed and implemented. One of these techniques is code inspection, first introduced by Fagan (1976). This technique can detect the software coding errors at early stage in lifecycle. Although code inspection's effect is that software quality can be improved, all the existing techniques for maintaining software reliability are reliant on the "checklist" approach to verify the software instructions and data sets. If the software size is small and not so complicated, the checklist process can be performed manually, otherwise it can

* Corresponding author. Tel.: +6623145054; fax: +6622249852.
E-mail address: prattana.d@student.netserv.chula.ac.th (P. Deeprasertkul).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.02.027

become too unwieldy. In this paper, we show how to automatically detect and correct the hidden faults in the software application prior to compilation time.

1.1. Problem description

Software reliability is partially depended on capabilities built into the languages' compiler. If the interpreters or compilers of languages are able to detect all common faults and errors, software reliability can be enhanced. Thus, Java and Erlang (Ganapathy et al., 2003; Armstrong et al., 1993) were developed with capabilities aimed at the objective of obtaining software reliability.

Java is a popular language which is widely used and classified as an object-oriented language. It is incorporated significant error checking such as the feature of detecting array indexes exceeding the array bounds during run-time, therefore containing the array indexes out-of-bounds handling.

Another functional programming language, Erlang (Armstrong et al., 1993) developed by Ericsson Sweden, is used to develop highly reliable the communication software products. A characteristic of this language is the pattern matching functionality which assists in tightly coupling faults and failures, so that, whenever a failure arises, the Erlang interpreter can immediately locate the cause of such failures. So, the software implemented in Erlang exhibit a very high level of software reliability.

There are, however, some faults and errors that cannot be detected by the compiler of software programming languages. Considering C programs, for example, the faults include cases such as array indexes out-of-bounds, passing the wrong types of function arguments, no-default-case in *switch* statements, or infinite loops. Furthermore, it is not uncommon that programmers or developers ignore warning message at compile time when, in fact, there warning messages may indicate the potential for a critical fault during software execution.

1.2. Approach

This paper proposes the design and implementation of a technique that can improve the software reliability of a system in a manner that cannot be achieved by any current methods. The major difference of PFD from the other existing techniques is the automatic detection and correction of faults performed prior to compile time. The software programs are preprocessed through PFD for detecting and correcting faults. The programmers are not allowed to ignore any warnings of the potential critical faults in the source code until proper actions have been performed. Consequently, faults and errors will be reduced, the system will then improve software reliability. Note that this technique applies many of the built-in reliability features of Erlang such as the feature of detecting array indexes exceeding the array bounds or the feature of detecting types of function arguments matching.

1.3. Contribution

The contribution of this paper is an introduction of a *Precompiled Fault Detection* (PFD) technique. This technique is a novel approach for automatically detecting and correcting the programming errors, which are the results of programmers inadvertence and cannot be detected by a compiler, in the source code prior to compilation time. The PFD technique can be applied to C applications and will be applied to other language applications in the future. Furthermore, we present experimental results that demonstrate an effectiveness of our technique.

The organization of this paper is as follows: In Section 2, the related work is discussed. Section 3 introduces the problems and motivations considered in this research. Section 4 presents an overview of pattern language used in PFD technique. Section 5 describes an architecture of PFD for detecting and correcting faults and Section 6 describes an implementation details of PFD technique. The testing method with results is covered in Section 6. The experimental results are shown in Section 7. Section 8 contains a discussion of our research. The final section is a conclusion of this paper.

2. Related work

Since software faults and errors interfere with normal processing, a number of techniques have been devised to minimize their effect. Many software inspection tools are used to inspect the running processes of software applications, such as ICICLE (Sembugamoorthy and Brothers, 1990), ASSIST (Macdonald, 1998), and Suite (Drake et al., 1991). Macdonald et al. (1995) compared the inspection processes of these software techniques. One tools for identifying faults during inspections is a "checklist". This checklist helps inspectors by listing all the fault types to look for (Rady de Almeida Jr. et al., 2003). The difficulty of manually verifying that the software under inspection conforms to the rules is partly to mistake.

One critical problem which is considered by many researchers as an example is the buffer overrun of array indexes. This problem can be solved by either dynamic or static techniques. Dynamic techniques such as Stackguard (Cowan et al., 1998), CCured (Necula et al., 2002) and High Coverage Detection of Input-Relate Security Faults (Larson and Austin, 2001) have been proposed to prevent the incorrect memory accesses without eliminating bugs in the source. These tools are applied at runtime, in a reactive fashion, attempting to catch invalid accesses. On the other hand, the static analysis tools

proposed to prevent and detect buffer overrun cases are mentioned in (Wagner, 2000; Ganapathy et al., 2003; Xie et al., 2003). These static tools focus on either the buffer overruns or memory access error detection looking for equivalent faults to the dynamic techniques. Once the problem of buffer overrun is detected, a warning message will be presented to the user.

Even though many software detection techniques and tools are proposed, the reliability of the software application is still largely reliant on the human designer's skills. Since the techniques noted above cannot avoid human errors, the potential improvement offered by inherently reliable programming languages such as Erlang (Armstrong et al., 1993) is needed. Erlang is a functional programming language that can guarantee the software reliability without permitting a wide range of human errors. The Erlang compiler uses a pattern matching technique that assists in tight coupling between faults and failures, therefore it can detect most of the hidden faults such as the incorrectness of array indices, the mismatch of function arguments types, and no-default-case in *switch* statements.

This paper proposes a technique that is to apply to program's source code before passing through the compilation process. The software source code will be analyzed to automatically detect and correct the coding errors before they will be released. This technique is called *Precompiled Fault Detection* (PFD). In this paper, we address the fault examples in C (Spuler, 1994; Harbison and Steele Jr, 1995) which are the case studies and, hence, they have a little difference of detection and correction procedure in each other. The reliability features of Erlang are applied to C programming language by the PFD technique.

3. Problem descriptions and motivations

Having a hidden fault in an application program can create the critical problems for an organization. Although software faults are rare ones in production cases, once a fault occurs, some critical system failures can occur. Since these faults cannot be detected by the compiler, it is the responsibility of programmers and testers to ensure that the developed software contains minimal faults. One way of performing fault detection is to take an advantage of software inspection. A source code is general examined by checking it for the presence of errors, rather than by simulating its execution (Ghezzi et al., 2003). Using this mechanism, it can detect and eliminate faults and errors in the software products developed during the software life cycle. Consequently, the reliability of applications are increased. However, fault detection is likely to fail unless extreme care is taken during a program inspection process.

Currently, the various compilers for languages have been progressively improved. However, programming languages have the different errors which still exist in the programs, depending on the error-prone features of the language. For instance, in C++ and Java, many mismatches between actual and formal parameters can be caught at compile time, but there might be an exception in C, etc. The following is a list of some classical programming errors (Ghezzi et al., 2003).

- array indexes out of bounds;
- mismatches between actual and formal parameters in procedure calls;
- nonterminating loops;
- use of uninitialized variables.

Fig. 1 shows a program about the seat allocation of flight. The program contains various faults including array indexes out-of-bound, passing incorrect types of function parameters and no-default-case in *switch* statements. The C compiler cannot detect these faults that have been identified as being responsible for many system essences.

```
1  #include <stdio.h>
2  main() {
3    int F_cls[5], B_cls[5], E_cls[10], i, j;
4    char cls;
5    for(i = 0; i <= 5; i++) {
6        printf("%d: ", i++);
7        scanf("%d", &F_cls[i]);
8    }
9    printf("\n");
10   for(i = 0; i < 5; i++) {
11       printf("%d: ", i++);
12       scanf("%d", &B_cls[i]);
13    }
14   ...
15    switch(cls){
16        case 'f' :
17              INSURANCE(cls);
18              for(j = 0; j <= 5; j++)
19                  printf("Seat %d\n", F_cls[j]);
20              break;
21       case 'b' :
22              INSURANCE(cls);
23              for(j = 0; j < 5; j++)
24                  printf("Seat %d\n", B_cls[j]);
25              break;
26       ...
27 }
28 INSURANCE(int class)
29 {
30    if(class == 1)
31        printf("ins of first class: 400,000\n");
32    else if(class == 2)
33        printf("ins of business class: 100,000\n");
34    else
35        printf("ins of crew: 100,000\n");
36 }
```

Fig. 1. An example of an application that contains faults.

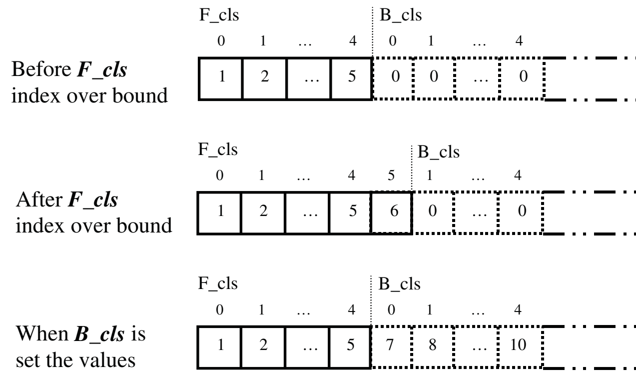

Fig. 2. Memory allocations for *F_cls*'s index out of bound; the replacement of *F_cls* with *B_cls*.

Example 1. Considering array indexes out-of-bound in Fig. 1, the instructions at line 5 to 8 declare values for array *F_cls* [0] to *F_cls* [5] when the upper bound of array *F_cls* should be 4. When array *B_cls* is declared the values from *B_cls* [0] to *B_cls* [4], the value of *B_cls* [0] replaces the value of *F_cls* [5]. Thus a person at *F_cls* [5] location is automatically eliminated. This error affects the company's reputation in negative manner. Fig. 2 shows the results of booking process and the memory declarations for *F_cls* and *B_cls*.

Example 2. When a function is generally called, parameters are passed to a called function. Since the old versions of C do not support function prototypes, therefore the passed type of function arguments are not checked. On the other hand, in the modern C, the programmers are able to declare the function before it is called. Thus its parameters' type are checked when the function is called. However some functions are not declared until the function has been used. Therefore the compiler treat these functions as if it is a non-prototype for function arguments. Once the function is recognized as the non-prototype for function arguments, the parameter checking is ignored.

Considering Fig. 1 at line 28, the INSURANCE function is declared and a passing argument is an integer named *class*. However at lines 17 and 22, INSURANCE function is called and the passing argument is *cls*, which is declared as a character. Since the value of passing parameter is "*f*", which is different from the declared parameter of INSURANCE function, there is no matched value in the *if*-statement and then the *else* command at lines 34, 35 are executed.

Example 3. Considering *switch* statement in Fig. 1 at lines 15 to 27, there is no *default* case. If the user types "*F*", instead of "*f*", to retrieve an insurance value of the first class, an user does not receive any values from the execution. Consequently, the user may misunderstand that the program is wrong, or malfunction occurs. If there is no matching case in *switch* statement, the *default* case should be defined in order to inform user that the program performs its task and cannot find any matching cases.

For more examples of the problems, considering the examples of C programs in (Deeprasertkul and Bhattarasinee, 2003) the errors include cases such as array indexes out of bound, passing the wrong types of function arguments, and no-default-case in *switch* statement.

Although programmers try to detect faults by running test data, or program inspection software, unfortunately some of faults may not be detected before software is delivered to users. Even though the faults do not cause an interruption in the software execution, the result from its execution cannot be trusted and, in a worse case, can produce a plausible but incorrect result. Thus the reliability of the software is not as high as expected. The PFD technique proposed in this paper helps programmers detect which faults and errors might be left in the programs. PFD can also automatically correct some faults if the programmers desires. The details of PFD technique are explained in Section 5 and 6.

4. Pattern language

The pattern language (Paul and Prakash, 1994; Hagemeister et al., 1996) is applied to check the programming language constructs such as variables declarations, type declarations, functions' argument types, etc. To illustrate our approach, we describe an overview of the pattern symbols in a sample pattern language for C. Table 1 lists the pattern symbols. We have developed the patterns using these symbols and collected them in *Pattern Library*. The brackets [...] and (...) in the array and function entries, respectively, stand for a list of arguments that can themselves be other identifiers or constants (Hagemeister et al., 1996).

All pattern symbols can be named where *name* can be any symbols made of alphanumeric characters. Named symbols can be used to express constraints within patterns, and to restrict the matching of pattern (Hagemeister et al., 1996). The list of them are given in Table 2.

Table 1
Symbols used for syntactic entities in source code

Syntactic entity	Pattern symbol
Variable	$v
Array variable	$a[...]
Function	$f[...]
Type	$t
Declaration	$d
Expression	#
Statement	@

Table 2
Named symbols used for syntactic entities in source code

Entity	Pattern symbol
Array variable	$a_*name*[...]
Function	$f_*name*(...)

4.1. Writing a pattern

Using the symbols previously mentioned, the patterns can be written. For example, suppose we want to locate the arrays in a source code, a pattern is then $a[...]. Therefore, the entire arrays in source code are scanned from left to right to be the matches. Another example, if we want to locate INSURANCE function in source code, we use a named symbol $f_INSURANCE(...) to be the pattern.

5. The proposed technique

PFD technique performs the fault detection as a software guard. The PFD preprocesses the programs before the compilation takes place as shown in Fig. 3. Only after the detected faults were corrected can the corrected software be compiled.

According to the functionality defined for PFD, it consists of two main modules: detection module, and correction module. Before describing our system in more detail, we formally introduce the definitions of *a set of PFD faults*, *a fault detection function*, and *a fault correction function*.

Definition 1. Let F be a set of all faults and let F' be a set of faults detected by PFD. Let F^u be a set of undetected faults. A fault f is a fault in F' if the fault f is detected by PFD. A fault f is a fault in F^u if it is not a fault in F'.

$$F' = F - F^u \quad \text{or} \quad F' = \{f' | f' \in F, f' \notin F^u\}$$

Definition 2. Let S be a set of statements in source code. D_f is called a detection function of PFD if all faults of F' in S are detected by D_f.

$$D_f : S \rightarrow F' \quad \text{or} \quad f' = D_f(s) \quad \text{where } f' \in F',\ s \in S$$

Definition 3. Let S^r be a set of corrected statements in source code. C_f is called a correction function of PFD if all faults in F' are corrected by C_f.

$$C_f : F' \rightarrow S^r \quad \text{or} \quad s_r = C_f(f') \quad \text{where } s_r \in S^r,\ f' \in F'$$

When all faults in F' are corrected, all corrected statements S^r are executed without the faults in F'.

5.1. Detection module

The detection module is an important module that identifies and guarantees software reliability for the hidden faults. This module is responsible for detecting faults that cannot be detected by compiler, and informs the programmers about faults.

When the programmers need to compile the programs, the programs are first analyzed by PFD. Each statement is traced by D_f of PFD to look for the faults F' in source code. PFD then generates a list of each fault to be used as input to the correction module. This process corresponds to Step 1 and Step 2 in Fig. 4.

Step 1: To detect the programming faults in program P, we first input P to PFD for analyzing each statement in P. The *Parser* parses the source code to discover which statements contain the potential faults.

A graph in Fig. 5(a) (Ferrante et al., 1987) is a directed graph for the constructs of a part of program in Fig. 5(b). The vertices represent statements in the program such as data types, variables, parameters, conditional branches, and assignment statements. The edges

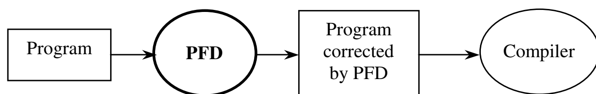

Fig. 3. Precompiled Fault Detection in context.

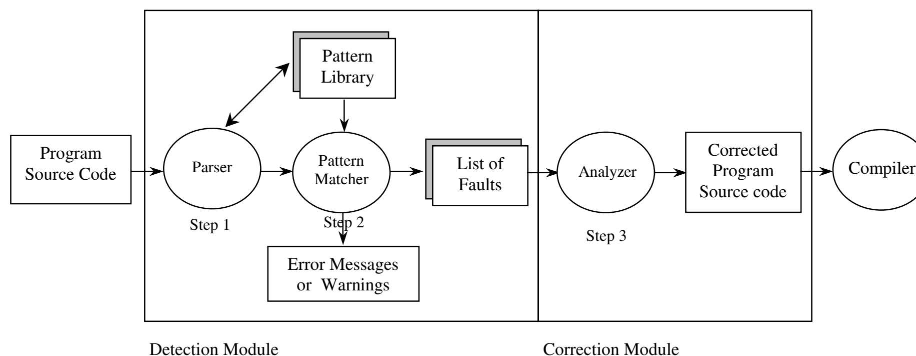

Fig. 4. The functionality of Precompiled Fault Detection.

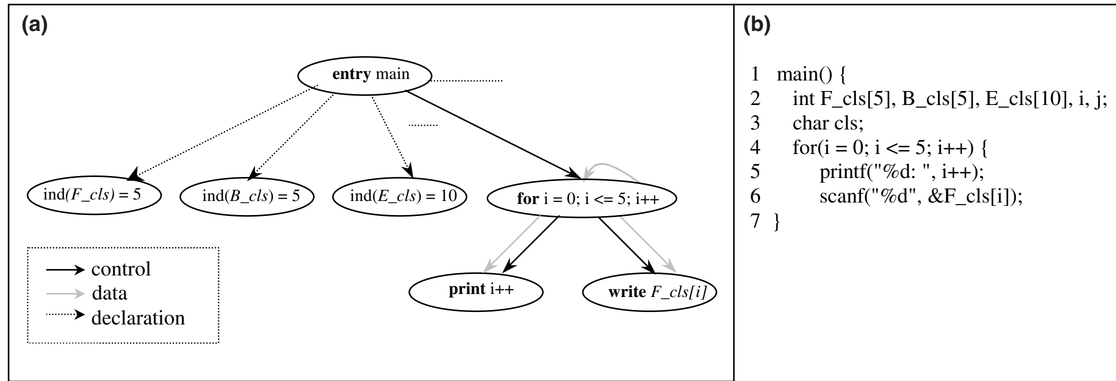

Fig. 5. An example of system graph (a) for a part of program in Fig. 1 shown on (b).

between the vertices indicate data, control dependence, or declaration. A data edge indicates a way in which the data value can be transmitted. For example, there is a data edge between the vertex for *for*($i = 0$; $i <= 5$; i++) and the vertex for *print i*++, which indicates that a value for i flows between these two vertices in Fig. 5(a). A control edge between a source vertex and a destination vertex indicates whether or not the destination vertex (e.g. *print i*++, *write F_cls*[*i*]) is reached by the result of executing the source vertex (e.g. *for*($i = 0$; $i <= 5$; i++)). A declaration edge indicates the declaration of variables in programs (e.g. *F_cls*[5]). For example, a vertex *ind*(*F_cls*) = 5 means that a size of *F_cls* index is 5.

The pattern matching in Erlang (Armstrong et al., 1993) provides the basic mechanism by which values become assigned to variables. Then, the value of these variables have been bound. The build-in reliability features of Erlang, such as the tuples are data structures which are used to store a fixed number of elements, are therefore applied in PFD.

In our approach, a source code is therefore parsed for looking for the required variable declarations or statements, e.g. *int F_cls[5], INSURANCE(...)*. They match the pattern of PFD's faults in *Pattern Library* described in Section 4. These required variable declarations or statements are then generated to be the new patterns in *Pattern Library* by the *Parser*.

Step 2: The *Pattern Matcher* considers the used variables, function call, etc. to match the pattern of declarations which are generated in Step 1. The *Pattern Matcher* also creates a log file for each fault defined in PFD as follows: Assume that method D_1 declares the detection of a fault type F_1 in program P_1. The *Pattern Matcher* creates the log file, P_1F_1*.log*. In a log file, there are n potential faults of F_1. An algorithm of main functionality of PFD is shown in Fig. 6.

An example of the *pattern* and the *match* graphs which are used to consider the programs in Step 1 and Step 2 is shown in Fig. 7. When the value of index i of

```
1  Function main_PFD_function(P) {
2      if (D1( ) == True) then C1( );
3      if (D2( ) == True) then C2( );
4          ⋮
5      if (Dn( ) == True) then Cn( );
6      else
7          compile P;
8  }
```

Fig. 6. An algorithm of a main functionality of PFD for detecting and correcting faults.

F_cls in *match* part does not match with its value in *pattern*(*ind(F_cls) = 5*), this fault is recorded in the list of faults. For example, when *i = 5*, it makes size of *F_cls* index is over its declaration (size of *F_cls* index is 6). An error message appears to caution the programmers and this fault is then corrected in Step 3.

5.2. Correction module

The aim of the correction module is to correct the detected faults during the detection module. Whenever any faults are detected, the programmer must correct them, otherwise the source code are not accepted by the compiler. Thus the faults cannot be bypassed by the programmer. A resulting program becomes more reliable since these detected faults which cause the critical system failures are corrected. Note that the correction module is optional, i.e., a programmer might prefer to fix a program manually instead of using automatic correction.

The correction module is Step 3 in Fig. 4.

Step 3: Most faults F' are automatically corrected by C_f of PFD. Some fault corrections cannot, however, be automatic. For example, the *default* case is automatically added to the no-default-case in *switch* statement, but the operations of *inserted default* case must be determined by the programmers.

The *Analyzer* in correction module performs this task by using the information from each log file provided by

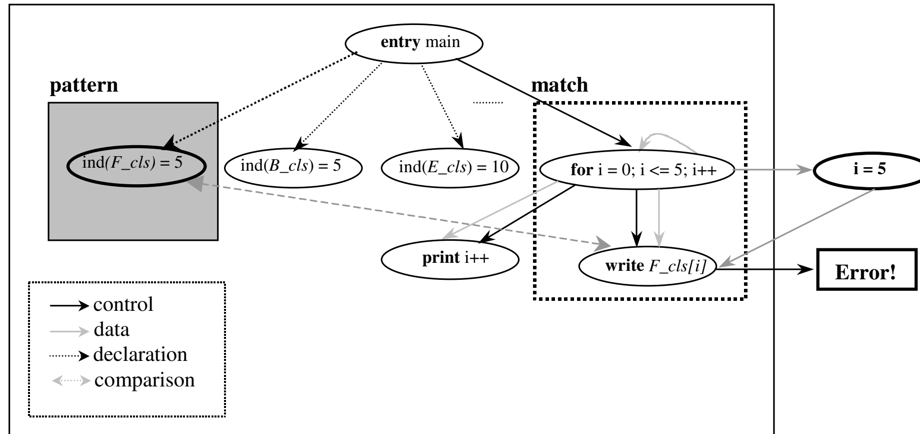

Fig. 7. An example of the *pattern* and *match* graphs for the program in Fig. 5(b).

the detection module. The log file exhibits the fault locations to PFD correction mechanism ($C_1, C_2, \ldots, C_n$ in Fig. 6).

5.3. Complexity

Considering the algorithm in Fig. 6, a program P with F fault types, a fault type has N potential faults. Therefore, the number of detected fault are F^*N faults. However, our approach mentioned in Section 5.1 can detect N faults of a fault type in one time detecting. For example, in a program P_1, there are three faults of the fault type F_1. All of three faults are detected in one execution time of the program input P_1. Thus, we implemented PFD that can detect all fault types by executing the program F times. The time complexity of detection module is $O(F)$.

6. PFD Implementation

According to the PFD architecture and algorithm in Section 5, PFD is implemented by using the C language to perform the fault detection and correction. The input of the PFD is an application written in C. The execution of PFD starts with asking the programmers to enter a program file.

The detection mechanism is the header files embedded in PFD implementation. Each source file is first passed to the detection mechanism. Fig. 8 shows the examples of fault detection algorithms in PFD ($D_1(), D_2(), \ldots, D_n()$ in Fig. 6). An algorithm for detecting array indexes is shown in Fig. 8(a). The array variables in source file are inspected to compare the declared indexes to the used ones. A fault is recorded in a log file, if the array index exceeds its bound. The case of function argument types is shown in Fig. 8(b). Fig. 8(c) illustrates the detection of no-default-case in *switch* statement.

```
1  function check_array(char *name)
2    while read next character until end of file
3        if  item == declared variable type
4           while read next character until new line
5               if  item ==  array variable
6                  put name and index in an array log file;
7           endwhile
8        else
9           if item  == array variable
10             compare the array index with index in the log file;
11       endif
12   endwhile
```

(a)

```
1  function check_function(char *name)
2    while read next character until end of file
3        if  item == name of declared function
4            put function name, line and argument types in the
5                functional log file;
6        else if item == name of function call
7            compare function call and declared function in log file;
8        endif
9    endwhile
```

(b)

```
1  function check_switch(char *name)
2    while read next character until end of file
3        if  item1 == "switch"
4           if  item2 == "default"
5              set TRUE;
6           endif
7        endif
8    endwhile
9    if not TRUE
10      display an error message;
11   endif
```

(c)

Fig. 8. Three examples of fault detection algorithm in PFD. (a) An algorithm of array index detection. (b) An algorithm of function argument types detection. (c) An algorithm of no-default-case in *switch* statement.

The detection mechanism is used to parse the source code of a given program for finding the potential faults. The given program input is parsed repeatedly to detect at all programming faults defined in PFD and the results of online checks are written out to log files by

Code segment	Log file	
	Name	Size
1 main( ) {		
2 int F_cls[5], B_cls[5], E_cls[10], i, j;	1. F_cls	5
3 char cls;	2. B_cls	5
4 for(i = 0; i <= 5; i++)	3. E_cls	10
5 {		
6 printf("%d", i++);		
7 scanf("%d", &F_cls[i]);	1. F_cls	6
8 }		
9 ⋮		
10 }		

Fig. 9. An example of a log file: array indices out-of-bound detection.

the detection mechanism. These log files are then processed to classify each fault. An example of a log file is shown in Fig. 9. Therefore, the outputs of this process are the log files and errors or warning messages.

The correction mechanism is also the header files in PFD implementation. This task traces each record in the log files provided by detection mechanism. The PFD requires access to the source code for correcting according to each record. If each fault in the log file is corrected, that record is flagged. After the given program is analyzed by PFD the compiler of language is called to compile the program.

7. Experimental results

To validate PFD technique, we first defined a set of programming faults which mostly occur in C programs such as the incorrectness of array indexes, the mismatch of function arguments types, and no-default-case in *switch* statements. These faults are encountered in the real applications. We used the applications containing them to make sure that our PFD correctly detects faults during the detection module and effectively corrects them during the correction module. Experiments were conducted following the methodology described in Section 5: We executed the PFD for analyzing each application. Table 3 lists a number of programming faults existing in the applications and a number of failures resulting from the detected faults. These testing applications are the prototypes of the seat allocation system and medical system. A source file *S_Darray.c* and *MedOrd.c* contains two array indexes out-of-bound each. *SeatRev.c*, which is the seat reservation program, has three array indexes out-of-bound. *Fmap.c*, *SeatCls.c*, and *PatType.c* have three, two, and one faults, respectively, about passing wrong type of function arguments. *Swcases.c* and *SeatPrice.c* hold one of no-default-case in *switch* statement each.

Fig. 10 illustrates a flowchart of PFD evaluation steps. After implementing PFD to detect and correct the faults in applications, a set of simulation data (10,000 data) has been applied in order to measure the resulting reliability of software. The resulting graphs of running software using the test data set before and after using PFD are presented in Fig. 11. Since there are a large number of testing data (10,000 data), all of them cannot be clearly represented in this paper. Thus, the graphs in Fig. 11 illustrate the only 100 testing data inputs. The number of failures, which are the effects of

Table 3
The number of programming faults in each C application

	Application	# Faults	# Failure	
			Before using PFD	After using PFD
1	*S_Darray.c*	2	9000	0
2	*SeatRev.c*	3	9002	2
3	*MedOrd.c*	2	9001	1
4	*Fmap.c*	3	987	0
5	*SeatCls.c*	2	424	0
6	*PatType.c*	1	296	6
7	*Swcases.c*	1	1443	0
8	*SeatPrice.c*	1	1755	4

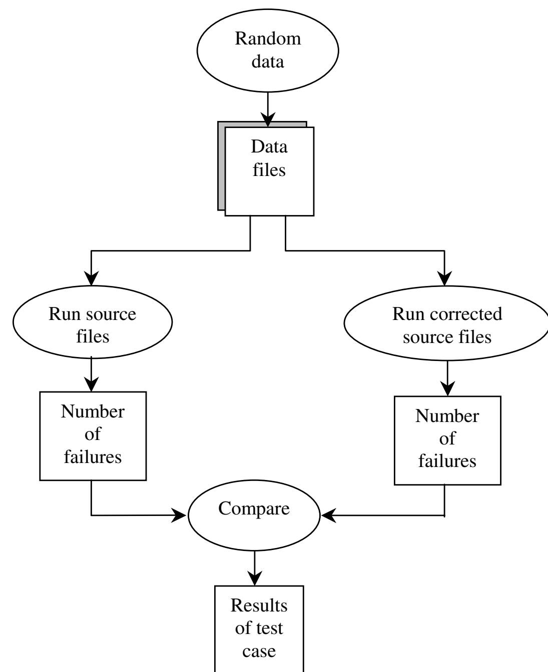

Fig. 10. A flowchart of the steps involved in the evaluation of using PFD.

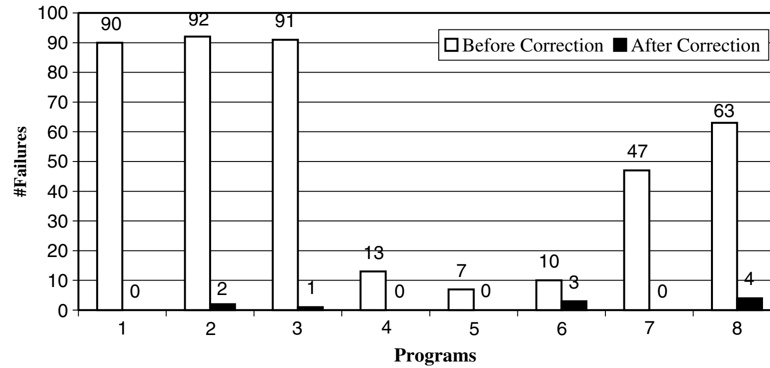

Fig. 11. A resulting graph before and after correcting by PFD.

the faults in Table 3, are completely removed from the applications. However, the failure occurrence after using PFD of *SeatRev.c*, *MedOrd.c*, *PatType.c*, and *SeatPrice.c* shown in Table 3 are not the effects of faults defined in PFD.

8. Discussion

Generally application code may contain faults both visible and invisible. These faults may cause the problems incorrect usage for the applications, thus effecting the reliability of usage. The reliability of software is a function of the number of faults in the program, therefore software developers must try to eliminate as many faults as possible. The consequence of fault elimination is that the risk of software failure is reduced and the reliability of the software can be significantly increased.

The objective of PFD is to detect the faults, and assist the software developer to correct these faults before passing the source code through to the compiler. These detected and corrected faults in the application software, after applying the PFD technique, will not occur again in the compiled applications.

Referring to the results presented in Section 7, these results confirm that the PFD technique has the capability of eliminating the critical faults that arise in C programming, such as the static array index out-of-bound, the passing of incorrect type of function arguments, or the no-default-case in *switch* statements. Software applications that utilize PFD during the software development process contain a significantly lower number of hidden faults than the software that compiles directly. Therefore the application software filtered by PFD will be efficient and reliable software as the users require.

The three cases of faults, the static array index out-of-bound, the passing of incorrect type of function arguments, and the no-default-case in *switch* statements, are representative of the scope of the PFD technique, a technique that has a wide applicability not restricted to the three chosen cases. In addition, we will apply this PFD technique to other programming languages.

9. Conclusion

The existence of faults in application code are both inevitable and can give rise to serious system outcomes. It is the responsibility of software developers to prevent and detect these hidden faults as far as possible. Currently there are a number of fault detection techniques such as buffer overrun or memory access error detection algorithms. But these techniques perform the fault detection at run-time, and may be unable to identify the fault's location easily, so that fault repair is difficult.

This paper has proposed a new and significant technique called *Precompiled Fault Detection* (*PFD*). The pattern matching in Erlang is applied to this technique for detecting and correcting hidden faults in a C implementation. The proposed technique has been tested by running a set of simulation programs with a test set of data, and the number of faults is counted before and after the program passes through the PFD. The result shows that, after passing the PFD, the number of faults from the application program is reduced or totally eliminated. Therefore the program execution will not be effected by the hidden faults.

The applications that can run without termination or interruption from its internal faults is certainly classed as reliable software. The PFD technique that supports automatic fault detection and correction of software, can be considered as a step towards increasing software reliability, in other words the software that has been preprocessed through PFD is shown to be much more reliable than software that is directly compiled. Therefore, PFD can guarantee the reliability of all the application software passed through.

Acknowledgment

We would like to thank Dr. Rob Rendell who was a staff at Software Engineering Research Centre, RMIT, Melbourne, Australia for his valuable comments on problems encountered in programming languages.

References

Armstrong, J.L., Virding, S.R., Williams, M.C., 1993. Concurrent Programming in Erlang. Prentice Hall.

Cowan, C., Beattie, S., Day, R-F, Pu, C., Wagle, P., Walthinsen, E., 1998. Automatic Detection and Prevention of Buffer Overflow Attacks, 7th USENIX Sec. Symposium.

Deeprasertkul, P., Bhattarasinee, P., 2003. Software Fault Detection in C Programs, 12th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering.

Drake, J., Mashayekhi, V., Riedl, J., Tsai, W., 1991. A Distributed Collaborative Software Inspection Tool: Design, Prototype, and Early Trial. Technical, Report TR-91-30, University of Minnesota.

Fagan, M., 1976. Design and code inspections to reduce errors in program development. IBM Systems Journal 15 (3), 182–211.

Ferrante, J., Ottenstein, K., Warren, J., 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 3 (9), 319–349.

Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D., 2003. Buffer Overrun Detection using Linear Programming and Static Analysis, 10th ACM Conference on Computer and Communication Security.

Ghezzi, C., Jazayeri, M., Mandrioli, D., 2003. Fundamentals of Software Engineering. Prentice-Hall (International edition).

Hagemeister, J.R., Bhansali, S., Raghavendra, C.S., 1996. Implementation of a Pattern-Matching Approach for Identifying Algorithmic Concepts in Scientific FORTRAN Programs, 3rd International Conference on High Performance Computing, pp. 209–214.

Harbison, S.P., Steele Jr., G.L., 1995. C: A Reference Manual, fourth ed. Prentice-Hall.

Larson, E., Austin, T., 2001. High Coverage Detection of Input Related Security Faults, 12th USENIX Sec. Symposium.

Macdonald, F., Miller, J., Brooks, A., Roper, M., Wood, M., 1995. A Review of Tool Support for Software Inspection, Proceeding 7th International Workshop Computer-Aided Software Engineering (CASE-95).

Macdonald, F., 1998. Computer-Supported Software Inspection, PhD thesis, Department of Computer Science, University of Strathclyde.

Necula, G.C., McPeak, S., Weimer, W., 2002. CCured: Type-Safe Retrofitting of Legacy Code, ACM Conference on the Principles of Programming Language (POPL).

Paul, S., Prakash, A., 1994. A framework for source code search using program patterns. IEEE Transactions on Software Engineering 20 (6), 463–474.

Rady de Almeida Jr., J., Batista Camargo Jr., J., Abrantes Basseto, B., Miranda Paz, S., 2003. Best practices in code inspection for safety-critical software. IEEE Software.

Sembugamoorthy, V., Brothers, L., 1990. ICICLE: Intelligent Code Inspection in a C Language Environment, Proceeding 14th Annual Computer Software and Applications Conference, pp. 146–154.

Spuler, D.A., 1994. C++ and C Debugging, Testing, and Reliability: the Prevention, Detection, and Correction of Program Errors. Prentice-Hall.

Wagner, D., 2000. Static Analysis and Computer Security: New Techniques for Software Assurance, PhD. Thesis, UC Berkeley.

Xie, Y., Chou, A., Engler, D., 2003. ARCHER: Using Symbolic Path-Sensitive Analysis to Detect Memory Access Errors, 9th European Software Engineering Conference and 11th ACM Symposium on Foundation of Software Engineering (ESEC/FSE).