

การปรับปรุงความน่าเชื่อถือของซอฟต์แวร์โดยการตรวจสอบความผิดพลาดของโปรแกรม
ที่สร้างโดยภาษาโปรแกรมเชิงโครงสร้าง

นางสาวปรารถนา ดีประเสริฐกุล

สถาบันวิทยบริการ

จุฬาลงกรณ์มหาวิทยาลัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรดุษฎีบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์ ภาควิชาคณิตศาสตร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

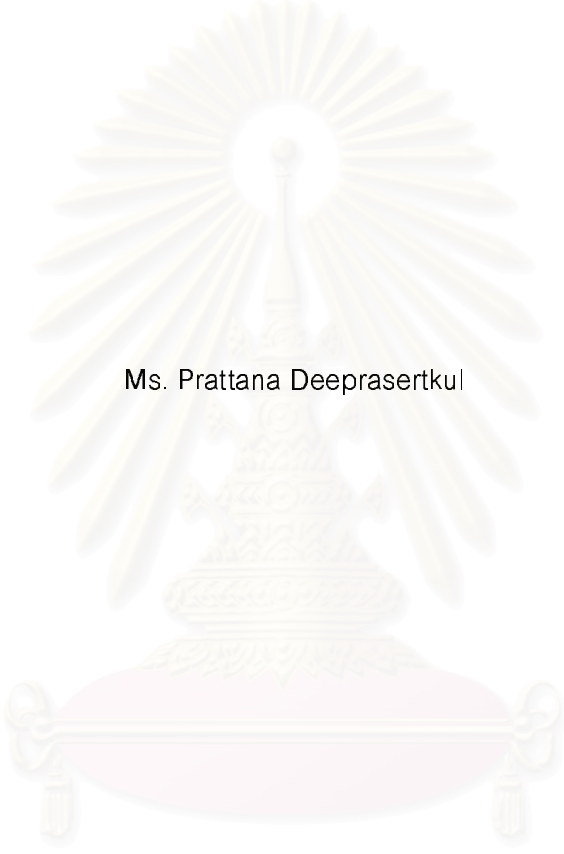
ปีการศึกษา 2547

ISBN 974-17-6584-3

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

AN IMPROVEMENT OF SOFTWARE RELIABILITY BY DETECTING FAULTY CODE
IN STRUCTURED PROGRAMMING

Ms. Prattana Deeprasertkul



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Computer Science

Department of Mathematics

Faculty of Science

Chulalongkorn University

Academic year 2004

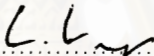
ISBN 974-17-6584-3

Thesis Title AN IMPROVEMENT OF SOFTWARE RELIABILITY BY DETECTING
FAULTY CODE IN STRUCTURED PROGRAMMING
By Ms. Prattana Deeprasertkul
Filed of study Computer Science
Thesis Advisor Assistant Professor Dr. Pattarasinee Bhattarakosol
Thesis Co-advisor Professor Dr. Fergus O'Brien

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the
Requirements for the Doctor's Degree


..... Dean of the Faculty of Science
(Professor Dr. Piamsak Menasveta)

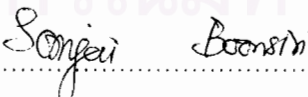
THESIS COMMITTEE


..... Chairman
(Professor Dr. Chidchanok Lursinsap)


..... Thesis Advisor
(Assistant Professor Dr. Pattarasinee Bhattarakosol)


..... Thesis Co-advisor
(Professor Dr. Fergus O'Brien)


..... Member
(Associate Professor Wanchai Rivepibool)


..... Member
(Assistant Professor Dr. Somjai Boonsiri)


..... Member
(Dr. Virach Sornlertlamvanich)

นางสาวปรารถนา ดีประเสริฐกุล : การปรับปรุงความน่าเชื่อถือของซอฟต์แวร์โดยการตรวจสอบความผิดพลาดของโปรแกรมที่สร้างโดยภาษาโปรแกรมเชิงโครงสร้าง (AN IMPROVEMENT OF SOFTWARE RELIABILITY BY DETECTING FAULTY CODE IN STRUCTURED PROGRAMMING) อ.ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร. ภัทรสินี ภัทรโกศล, อ.ที่ปรึกษาร่วม : PROFESSOR FERGUS O'BRIEN, หน้า. ISBN 974-17-6584-3.

วิทยานิพนธ์นี้ได้นำเสนอการปรับปรุงความน่าเชื่อถือของซอฟต์แวร์โดยการตรวจสอบความผิดพลาดของโปรแกรม (Software Fault) ที่สร้างโดยภาษาโปรแกรมเชิงโครงสร้าง เครื่องมือซอฟต์แวร์ที่ได้จากการศึกษาวิทยานิพนธ์นี้จะทำการตรวจสอบแต่ละคำสั่งในโปรแกรมเพื่อหาความผิดพลาดที่เกิดจากการเขียนโปรแกรมและแก้ไขความผิดพลาดเหล่านี้ ความผิดพลาดของโปรแกรมจะทำให้เกิดความผิดพลาดของระบบงาน (Software Failure) ดังนั้นเมื่อความผิดพลาดของโปรแกรมถูกแก้ไขให้มีจำนวนลดลงความน่าเชื่อถือของระบบงานก็จะเพิ่มมากขึ้นด้วย

ในการศึกษาปัจจุบันส่วนใหญ่จะมีการทำการตรวจสอบความผิดพลาดของโปรแกรมในช่วงของการพัฒนาระบบงาน เนื่องจากระบบซอฟต์แวร์ในปัจจุบันมีขนาดใหญ่และซับซ้อน การตรวจสอบในช่วงของการพัฒนาระบบงานจะทำให้ต้องเสียค่าใช้จ่ายมาก ดังนั้นในวิทยานิพนธ์นี้จะทำการแก้ไขปัญหาดังกล่าวโดยการตรวจสอบความผิดพลาดของโปรแกรมซึ่งเป็นสาเหตุของความผิดพลาดของระบบงานก่อนที่โปรแกรมจะถูกคอมไพล์

สถาบันวิทยบริการ จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา คณิตศาสตร์
สาขาวิชา วิทยาการคอมพิวเตอร์
ปีการศึกษา 2547

ลายมือชื่อนิพนธ์.....
ลายมือชื่ออาจารย์ที่ปรึกษา.....
ลายมือชื่ออาจารย์ที่ปรึกษาร่วม.....

4373824223 : MAJOR COMPUTER SCIENCE

KEY WORD: Software reliability / Software failure / Software fault / Fault detection / Pattern matching

PRATTANA DEEPRASERTKUL : AN IMPROVEMENT OF SOFTWARE RELIABILITY BY
 DETECTING FAULTY CODE IN STRUCTURED PROGRAMMING. THESIS ADVISOR :
 ASSIST. PROF. PATTARASINEE BHATTARAKOSOL, Ph.D., THESIS COADVISOR :
 PROFESSOR FERGUS O' BRIEN, Ph.D., pp. ISBN 974-17-6584-3.

Software reliability is an important feature of a good software implementation. However, some faults that cause the software failures are not detected during the development stages. These faults will create unexpected problems for users whenever they arise. At present most of the current techniques detect faults while the software is running.

Unlike other techniques, the **Precompiled Fault Detection and Correction (PFDaC)** technique introduced in this dissertation detects and corrects the faults before the source code is compiled. The objective of the PFDaC technique is to increase software reliability without increasing the programmers' responsibilities. The concept of "pre-compilation" is applied to PFDaC in order to reduce the risk of significant damages during the execution period. PFDaC technique can completely eliminate the significant faults in the program source code. Consequently, the risks of software failure based on these faults can be avoided and the reliability of the software is improved.

สถาบันวิทยบริการ
 จุฬาลงกรณ์มหาวิทยาลัย

Department Mathematics

Field of study Computer Science

Academic year 2004

Student's signature.....

Advisor's signature.....

Co-advisor's signature.....

Acknowledgements

I would like to express my deepest gratitude and thanks to my advisors Assist. Prof. Dr. Pattarasinee Bhattarakosol for her contributions to all aspects of my works on this thesis. She was a great advisor who provided a lot of advise for the thesis. I would also like to thank Professor Fergus O'Brien who is my co-advisor for his support during I stayed and did research at SERC, without his contributions and guidance work would not have been possible.

I am greatly thankful to DPST scholarship for the financial support since Master Degree until I graduate Ph.D. I would like to thank my best friends for their help, caring and many pleasant memories over years.

Finally, I would like to thank my parents for their love, support, and encouragement. I will forever be indebted to them for everything they have done for me. My parents, my brother and his family also stood by me.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Table of Contents

	Page
Thai Abstract	iv
Abstract	v
Acknowledgments	vi
List of Tables	x
List of Figures	xi
1 Introduction	2
1.1 Motivation	2
1.1.1 Static Array	5
1.1.2 Function	6
1.1.3 <i>switch</i> Statement	7
1.1.4 Dynamic Array	8
1.1.5 Infinite Loop	9
1.2 Objectives	11
1.3 Scope of Work	11
1.4 Contributions of the Dissertation	12
1.5 Expected Outcomes	13
1.6 Dissertation Organization	13
2 Related Work	14
2.1 Background	14
2.1.1 Erlang Programming Structure	14
2.1.2 Faults and Failures in Erlang	15

	Page
2.2 Related Work	17
2.2.1 Software Inspection	17
2.2.2 The Current Software Fault Detection Methods	18
2.2.3 Erlang Programming Language	19
3 The Precompiled Fault Detection and Correction	20
3.1 A Program Dependence Graph	21
3.2 Pattern Language	22
3.2.1 Using a Pattern	23
3.3 Detection Module	23
3.4 Correction Module	26
3.5 Complexity	28
3.6 Summary	29
4 The Implementation and Experimental Results of the Proposed Technique ...	30
4.1 Implementation of Precompiled Fault Detection and Correction Technique	30
4.2 Experimental Results	35
5 Theoretical Analysis	41
5.1 Fault Detection	41
5.2 Fault Correction	43
6 Discussion and Conclusion	45
6.1 Discussion	45
6.2 Conclusion	46

	Page
References	48
Appendices	52
Appendix A	53
Appendix B	57
Appendix C	68
Vita	74



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

List of Tables

Table	Page
3.1 Symbols used for syntactic entities in source code	22
3.2 Named symbols used for syntactic entities in source code	23
4.1 The number of programming faults in each C application	37



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

List of Figures

Figure	Page
1.1 An example of an application that contains a static array index out-of-bound	5
1.2 An example of an application that contains a string array index out-of-bound	6
1.3 An example of passing wrong type of function arguments	7
1.4 An example of no-default-case in <i>switch</i> statements	8
1.5 An example of a dynamic array index out-of-bound	9
1.6 An example of the first case of infinite loop	10
1.7 An example of the second case of infinite loop	10
2.1 A structure of an Erlang program	15
3.1 Precompiled Fault Detection and Correction in context	20
3.2 The Program Dependence Graph for the small program shown on (b)	21
3.3 The detecting function of Precompiled Fault Detection and Correction	24
3.4 An example of system graph (a) for a part of program in Figure 1.1 shown on (b)	25
3.5 An algorithm of a main functionality for detecting and correcting faults	26
3.6 An example of the <i>pattern</i> and <i>match</i> graphs for the program in Figure 3.4(b)	26
3.7 The correcting function of Precompiled Fault Detection and Correction technique	27
4.1 An example of a log file: array index out-of-bound detection	34
4.2 An example of a log file: string array index out-of-bound detection	35

Figure	Page
4.3 An example of a log file: passing wrong type of function argument detection	36
4.4 A flowchart of the steps involved in the evaluation of using the PFDaC technique	36
4.5 The resulting graph before and after correcting by the PFDaC technique in File1.c	38
4.6 The resulting graph before and after correcting by the PFDaC technique in File2.c	39
4.7 The resulting graph before and after correcting by the PFDaC technique in File3.c	40
A.1 The interface prototype of the proposed technique for file inputs	50
A.2 The interface prototype of array indices detection	51
A.3 The interface prototype of passing the wrong function argument type detection	51
A.4 The interface prototype of <i>switch</i> statement case detection	52
A.5 The interface prototype of dynamic array index detection	53
A.6 The interface prototype of infinite loop detection	53

CHAPTER I

Introduction

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

1.1 Motivation

Software reliability is partially depended on capabilities built into the compilers. If the interpreters or compilers are able to detect all common faults and errors, software reliability can be achieved. Although many languages are developed to serve various types of humans' needs, only some of them can guarantee the reliability of their applications. The languages such as Java and Erlang are examples of high quality programming languages [1,9] that reliability of their applications can be ensured.

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 indices exceeding the array bound during run-time. Since objects in Java programs are responsible for operations to be performed, the parameters to perform such tasks are checked and informed during the compile time. Thus, the faults can be detected and corrected before the applications are delivered. Therefore, Java can be called as a language that supports the fault-avoidance method.

Another functional programming language, Erlang, [1] is used to develop highly reliable communication-software products. A characteristic of this language is the pattern matching functionality which assists in tightly coupling faults and failures. Therefore, 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.

Even though programmers and testers have performed the verification for faults detection during the software development process, unfortunately, there are some faults and errors which cause the critical failures still remaining in the program. One reason for the remaining faults and errors is the inefficient task of compilers. Since some compilers cannot detect some faults or errors, therefore, there is no error or warning message presented to programmers while the programs are compiled. Examples of the languages that their compilers cannot fully detect faults are C, FORTRAN, Turbo Pascal, etc.

Considering C programs, for example, the faults include cases such as array indices out-of-bound, 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 messages at the compile time when, in fact, these warning messages may indicate the potential for a critical fault during software execution.

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. Thus, the programmers and testers must ensure that the developed software operates under the fault-free situation. 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 [10]. 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, the fault detection method is likely to fail unless the extreme care is taken during a program inspection process.

Since there are various types of applications such as game applications, network applications, and web applications. Thus, different applications generally are developed using different languages. For examples, the network management application may be developed by C whereas the e-commerce applications on web will not be implemented by C. Thus, there are some differences of errors existing in programs, depending on the error-prone feature of the programming languages. For instance, in C++ and Java, many mismatches between actual and formal parameters can be caught at the compile time, but there might be an exception in C, etc. The following is a list of some classical programming errors [10].

- array indices out of bound;
- mismatches between actual and formal parameters in procedure calls;
- nonterminating loops;

- use of uninitialized variables.

Considering the system software that is the heart of the computer's operations, most of these software are developed in C. Additionally, most compilers of the structured programming languages are not able to detect all faults. Therefore, in this dissertation, we will consider C as a representative of other structured programming languages and study faults that lead to software failures.

1.1.1 Static Array

The C compiler does not have the checking of array indices whether they are out of bound [24]. One example about an array index out of bound is shown in Figure 1.1.

```
1 main() {
2   int arr[5], temp[5], i, result, sum;
3   char mon;
4   i = result = sum = 0;
5   while(i <= 5) {
6     temp[i] = arr[i];
7     i = add(i, 1);
8     sum = add(sum, temp[i]);
9   }
10  ...
```

Figure 1.1: An example of an application that contains a static array index out-of-bound.

Example 1.1. Considering the case of an array index out of bound in Figure 1.1. The instruction at line 5 declares values for array *arr[0]* to *arr[5]* when the upper bound of the *arr* array should be 4. Consequently, the value of *temp[0]* is replaced by the value of *arr[5]* at line 6. Thus the value at the *temp[0]*'s location is automatically eliminated. This fault can affect the company's profit and loss, which uses this code, in business.

Finally, it will affect the company's reputation in the negative manner.

```
1 main() {  
2     char str1[] = "computer";  
3     char str2[] = "science";  
4     strcpy(str1, "mathematical");  
5     printf("%s %s", str1, str2);  
6 }
```

Figure 1.2: An example of an application that contains a string array index out-of-bound.

Example 1.2. Another example of array index out of bound is shown in Figure 1.2. The string “mathematical” which is 12 characters is greater than string “computer” which is 8 characters. The fault occurs when the string “mathematical” is copied into *str1* at line 4.

1.1.2 Function

When a function is generally called, some parameters may be passed to the 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 treats these functions as if it is a non-prototype for function arguments. Consequently, the parameter checking is ignored.

Example 1.3. Considering Figure 1.3 at line 14, the *add* function is declared, and the passing arguments are the integer named *x* and *y*. However at line 11, the *add* function is

```

1 main() {
2   int arr[5], temp[5], i, result, sum;
3   char mon;
4   i = result = sum = 0;
5   while(i <= 5) {
6     temp[i] = arr[i];
7     i = add(i, 1);
8     sum = add(sum, temp[i]);
9   }
10  ...
11  result = add(sum, mon)
12  print(result);
13 }
14 int add(int x, int y) {
15   return(x+y);
16 }

```

Figure 1.3: An example of passing wrong type of function arguments.

called and the passing arguments are *sum* and *mon*, which *mon* is declared as a character. Since the type of passing parameter, *mon*, is different from the declared parameter, *y*, of the *add* function, therefore there is no matched value.

1.1.3 *switch* Statement

In the *switch* statement there is the *default* case that is used when there is no match in the *switch* statement. However, programmers may ignore the use of the *default* case with various reasons. In some situation, it is dangerous if there is no matching case in *switch* statement and the *default* case does not exist. Thus, the execution continues and a serious accident occurs as shown in Figure 1.4.

Example 1.4. Considering the *switch* statement in Figure 1.4 at lines 6 to 11, it is a program about the aircraft landing control system. The *switch* statement in the program does not have *default* case. The failure may occur if the emergency case happens on that

```

1  main() {
2  ...
3  Domestic = 1;
4  International = 2;
5  ...
6  switch(type){
7      case 1 :
8          Runway_Free(type);
9      case 2 :
10         Runway_Free(type);
11     }
12     Landing();
13     ...

```

Figure 1.4: An example of no-default-case in *switch* statement.

aircraft and it cannot be specified the type. The unidentified aircraft may land on the runway that is not available. Therefore, If there is no matching case in *switch* statement, it should have the *default* case for resolving this problem.

1.1.4 Dynamic Array

Dynamic array is another type of array. It involves dynamic memory allocation. A piece of memory is allocated to define array index for a variable we have declared. Some failures may occur if some variables use the memory storage over its declaration as shown in Figure 1.5

Example 1.5. Considering an array index out of bound in Figure 1.5. The instruction at line 6 declares values for array *arr[0]* to *arr[256]* when the upper bound of the *arr* array should be 255. When the array *temp* was set the values from *arr[0]* to *arr[256]* at line 7, the value of *temp[0]* is replaced by the value of *arr[256]*. Thus the value at the *temp[0]*'s location is automatically eliminated. This fault also affects the company's

```

1 main() {
2   int *arr, *temp, i, sum;
3   arr = malloc(1); temp = malloc(1);
4   i = sum = 0;
5   ...
6   while(i <= 256) {
7     temp[i] = arr[i];
8     i = add(i, 1);
9     sum = add(sum, temp[i]);
10  }
11  ...

```

Figure 1.5: An example of a dynamic array index out-of-bound.

reputation in the negative manner.

1.1.5 Infinite Loop

Another failure that mostly meets in coding program is the infinite loop. In this dissertation focuses on two cases of the infinite loop. First case is shown in Figure 1.6; the value of *total* (or other values in *while* loop) will continue to increase infinitely. The *while* loop is never stopped as the variable *ok* never be FALSE. Another case of the infinite loop in this dissertation is shown in Figure 1.7. The value of *total* will continue to increase infinitely as well, if the variable *i* has never been less than zero (negative number) or greater than zero (positive number), respectively.

For more examples of problems, considering the examples of C programs in [4] the errors include cases such as array indices 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 data test sets, or program

```
1 main() {  
2   ...  
3   ok = TRUE;  
4   ...  
5   while(ok == TRUE)  
6   {  
7     ...  
8     total = total + n;  
9     ...  
10 }
```

Figure 1.6: An example of the first case of infinite loop.

```
1 while(i >= 0)  
2 {  
3   ...  
4   total = total + n;  
5   ...  
6 }  
7 or  
8 while(i <= 0)  
9 {  
10  ...  
11  total = total + n;  
12  ...  
13 }
```

Figure 1.7: An example of the second case of infinite loop.

inspection software, unfortunately, only some faults can be detected before software is delivered to users. Thus the reliability of the software cannot be fully guaranteed in the runtime process. The proposed technique called as **Precompiled Fault Detection and Correction (PFDaC)** helps programmers detect significant faults and errors that might be left in the programs. Additionally, it can also automatically correct some faults based on the programmers' desires.

1.2 Objectives

In this dissertation, our objectives are as follows:

1. To propose a method for detecting software faults in software programs to improve the software reliability by applying infrastructure of a functional programming language to the structured programming language environment.
2. To develop a static detection tool that can detect software faults in applications to improve the reliability of software modules which are investigated.

1.3 Scope of Work

Presently, there are two classes of the programming languages: structured programming languages, and object-oriented programming languages. As mentioned previously that some defects in the structure programming languages are hidden during the compile time, the software products cannot be called as high-reliable software.

When the executing software was interrupted, or the software failure occurs, it can be counted as a cost (or expense) of the organization. Resolving the failure software is

time-consuming, and it increases the risk of losing customers of the organization. Therefore, this research has an aim to propose a technique that can increase and guarantee the reliability of the software product before the software are delivered to clients. The proposed technique, PFDaC, is independent from the efficiency of compilers. Therefore, it can be applied to every programming languages. The significant functions of PFDaC are automatically detect and correct faults in the structured programming applications at the compile time.

In this dissertation, we focus on the case study of the investigated programs written in C language. The following C programming faults are considered.

- static and dynamic array indices out of bound,
- passing incorrect types of function's arguments,
- no default case in *switch* statements,
- some cases of infinite loops.
- some cases of dynamic arrays, this dissertation focuses on the dynamic array indices which can be computed their values at compiled time.

1.4 Contributions of the Dissertation

The contribution of this dissertation is an introduction of the PFDaC technique. PFDaC can automatically detect and correct the programming errors, which are the results of programmers inadvertence and cannot be detected by compilers, in the source code prior to the compile time. PFDaC can be applied to C applications and it will be applied to

other language applications in the future. Furthermore, the experimental results and theoretical approval have been presented to support the design of PFDaC.

1.5 Expected Outcomes

1. To decrease the number of failures in software applications and improve the software reliability.
2. To apply this technique to other structured programming languages.

1.6 Dissertation Organization

The rest of the dissertation is organized into five additional chapters. Chapter 2 discusses the background and related works while Chapter 3 proposes an architecture of PFDaC. Additionally the technique for pattern matching, detecting, correcting faults in software source code are described in this chapter. The implementation and experimental results of PFDaC technique are presented in Chapter 4. Furthermore, theoretical analysis of fault detection and correction is shown in Chapter 5. Finally, discussion and conclusion of this dissertation are elaborated in Chapter 6.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER II

Related Work

In this chapter, some characteristics of Erlang which is a functional language will be presented to show the pattern-matching features. The methods, tools, and techniques related to software fault detection are discussed to the related work section.

2.1 Background

2.1.1 Erlang Programming Structure

Erlang [1, 13] is a functional programming language that was developed by Ericsson and Ellemtel Computer Science Laboratories. According to the architecture of Erlang, the programs in Erlang are mostly free from side-effects. Additionally, Erlang generally has no reassignment statements. Furthermore, written programs in Erlang are about 5-10 times shorter than the equivalent programs in C [24].

An Erlang program consists of a set of functions which may be collected into modules [1] as shown in Figure 2.1. If the failure occurs in function B of Module I, the fault may be somewhere nearby it (in Function B or in some functions calling it).

In addition, neither global variables nor pointers are used in Erlang. Moreover, local variables are assigned inside functions and these variables are never changed. All of

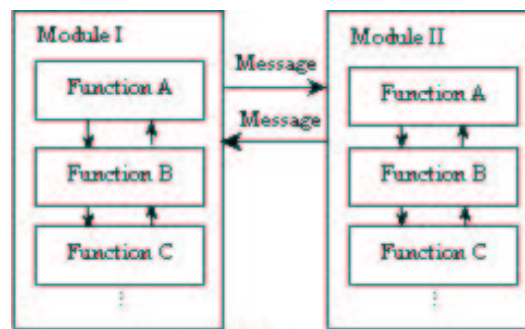


Figure 2.1: A structure of an Erlang program.

these advantages help the Erlang programmers tracking faults more easier and faster than the similar functions in C programs when failures occur. In this section, the pattern matching which is one significant characteristic of Erlang is considered. It makes the failures occur close to their causes.

2.1.2 Faults and Failures in Erlang

Although Erlang is a reliable programming language, there are some faults and failures that can be detected in Erlang programming which are described as follows.

Example 2.1 The Nth element of the tuple in Erlang

```
-module(divide1).
```

```
-export([divide/0]).
```

```
divide() ->
```

```
 $T = \{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\},$ 
```

```
 $D = 1 \text{ div element}(12, T),$ 
```

```
io:format("~w~n", [D]).
```

In Example 2.1, the process will terminate with an error at run-time because T has

just 10 elements. But D is the result of 1 dividing by 12th element of T that is not defined.

Example 2.2 *if* statement in Erlang

```
...
if
Score >= 80 -> io:format("High Distinction~n");
Score >= 70 -> io:format("Distinction~n");
Score >= 60 -> io:format("Credit~n")
end,
io:format("~w~n", [Score]).
```

Example 2.2 is an example of *if* statement in Erlang. The failure will occur if $Score < 60$, since there are no any matching cases in *if* statement. Consequently, a run-time error will be generated and the next instruction cannot be continuously executed.

Example 2.3 Function in Erlang

```
-module(com).
-export([compute/1]).
compute({add, A, B}) -> A + B;
compute({double, X}) -> X * 2;
compute({times, Y, N}) -> Y * N;
```

In Example 2.3, when the function is evaluated, arguments of the function are matched against the patterns occurring in the function definition. The arguments, which are variables, are also checked when the *compute* function is called. So, if the function

calls are as follows:

>com:compute({minus, 7, 0}).<1>

>com:compute({add, 'a', 10}).<2>

then a run-time error will occur. Because in <1> there is no clause defining {minus, 7, 0} in com:compute/1 or in <2> 'a' is an incorrect argument in computation.

As shown in the above examples of Erlang, programmers can detect faults' locations as soon as the run-time error occurs. So, the ability to detect these faults is the advantage of pattern matching in Erlang.

2.2 Related Work

2.2.1 Software Inspection

Since software faults and errors interfere with normal process, various 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 [22], ASSIST [16], and Suite [5]. [15] compared the inspection processes of these software techniques. One tool for identifying faults during inspections is a "checklist". This checklist helps inspectors by listing all the fault types to look for [19]. However, the software inspection is usually performed after the compile time. Furthermore, the checklist for software inspection is defined manually. Thus, there is some possibility of human' s error that some faults may be left out.

2.2.2 The Current Software Fault Detection Methods

One critical problem which is considered by many researchers as an example is the buffer overrun of array indices. This problem can be solved by either dynamic or static techniques. Dynamic techniques such as Stackguard [3], CCured [17] and High Coverage Detection of Input-Related Security Faults [14] have been proposed to prevent the incorrect memory accesses without eliminating bugs in the source code. These tools are applied at run-time, 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 [9, 25, 26]. 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 users.

The structured programming languages, such as C, are widely used for developing the software products. The C language programs are relatively large. When a failure occurs, it has to take a long time to find out the causes by tracing faults in the collected log file. Currently, there are tools such as Purify and Valgrind that can detect an array index out-of-bound. Purify is a commercial package tool that can find memory errors in programs, but it is very expensive [20]. Valgrind is a tool for finding memory management problems in x86 GNU/Linux executables. Valgrind is licensed under the GNU General Public License [23]. Software running under the current tools runs much more slowly, making testing more time-consuming and tedious. Moreover, the existing tools are applied at run-time, in a reactive fashion, attempting to catch invalid accesses

when they happen.

2.2.3 Erlang Programming Language

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 mentioned above cannot avoid human errors, the potential improvement offered by inherently reliable programming languages such as Erlang [1] 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 tightly 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.

Currently, the existing tools are applied at run-time attempting to catch faults when failures appear in the system. However, it is costly and time-consuming to return the source code for tracing these faults if the software system is released to user. Therefore, the **Precompiled Fault Detection and Correction (PFDaC)** technique is proposed to help resolving this problem. PFDaC would be applied at the compile-time with the intent to reduce the time it takes to debug the code that caused faults and failures. The program source code is analyzed by PFDaC mechanism before passing through the compilation process. The fault examples in C [12, 24], which are the case study, are also address in this dissertation. There are some differences of detection and correction procedures in each case. Furthermore, the reliability features of Erlang are applied to C programming language by PFDaC technique.

CHAPTER III

The Precompiled Fault Detection and Correction

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 products.

In this chapter, the architecture and processes of PFDaC technique are described. The significant function of PFDaC technique is to perform the fault detection as a software guard. It preprocesses the programs before the compilation takes place as shown in Figure 3.1. The corrected software can be compiled only after the detected faults were corrected.

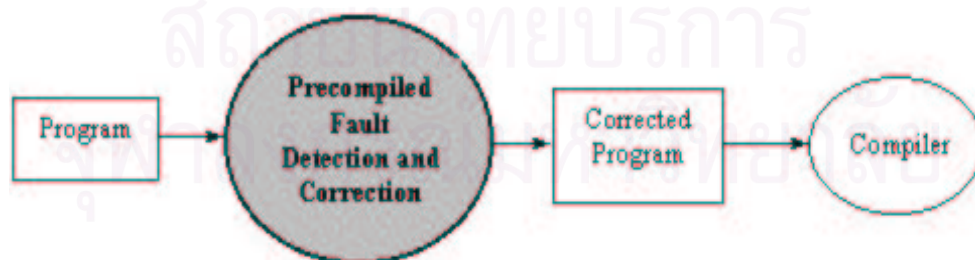


Figure 3.1: Precompiled Fault Detection and Correction in context.

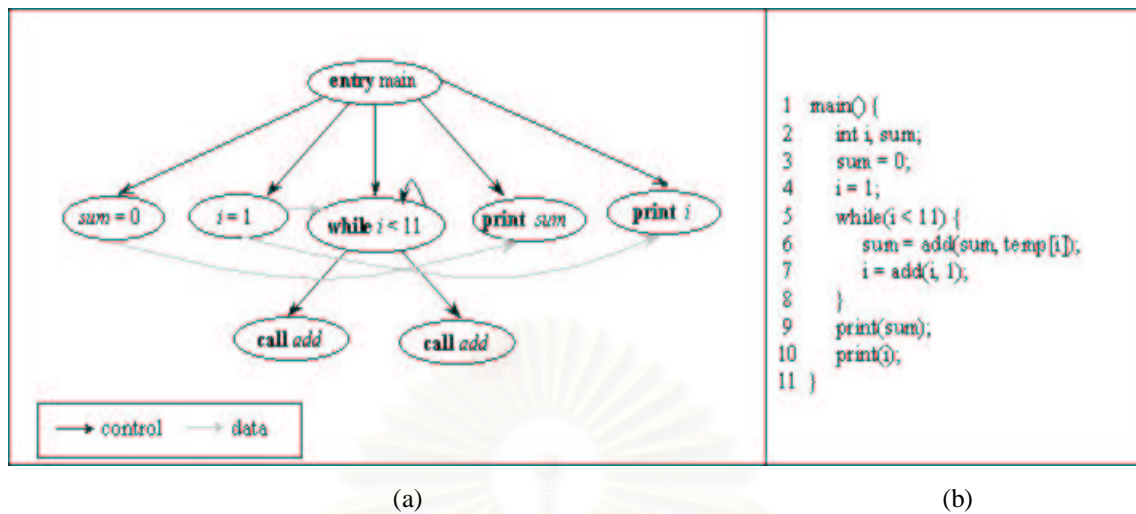


Figure 3.2: The PDG for the small program shown on (b).

According to the functionality defined for the PFDaC technique, it consists of two main modules: the detection module, and the correction module. Before describing the PFDaC technique in more details, a basic material on a program dependence graph and the pattern language are introduced.

3.1 A Program Dependence Graph

The *Program Dependence Graph* (PDG) [2, 7] is a directed graph for a single procedure of a program. The vertices of the graph represent constructs such as assignment statements, call sites, parameters, and conditional branches. An edge between the vertices indicates either a data dependence or a control dependence. The data-dependence edges indicate possible ways in which data values can be transmitted. For example, in Figure 3.2, there is a data-dependence edge between the vertex for $i = 1$ and the vertex for *while* ($i < 11$), which indicates that a value for i may flow between those two vertices.

A control-dependence edge between a source and a destination vertices indicates that the result of executing the source vertex controls whether the destination vertex is reached. For example, in Figure 3.2, there is a control-dependence edge between the vertices for *while* ($i < 11$) and the vertices for the two call sites on the function *add*.

3.2 Pattern Language

The pattern language [11, 18] is applied to check the programming language constructs such as variables declarations, type declarations, functions' argument types, etc. To illustrate the PFDaC approach, an overview of the pattern symbols in a sample pattern language for C is described. Table 3.1 lists the pattern symbols. The pattern have been developed using these symbols and collected in the *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 [11].

Table 3.1: 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	@

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 [11]. The list of them are given in Table 3.2.

Table 3.2: Named symbols used for syntactic entities in source code.

Syntactic Entity	Pattern Symbol
array variable	\$a[...]
function	\$f(...)

3.2.1 Using a Pattern

Using the symbols previously mentioned, the patterns can be written. For example, suppose that an array is needed to locate 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 the location of *add* function is needed to search in the source code, a named symbol \$f_add(...) is used to be the pattern.

3.3 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 the compiler, and informing the programmers about faults.

When a programmer needs to compile a program, the program is firstly analyzed by the PFDaC mechanism. Each statement is traced by the detection function of the PFDaC technique to look for the faults in the source code. The detection module in the PFDaC mechanism, then, generates a list of each fault and uses it as input to the correction module. This process corresponds to Step 1 and Step 2 in Figure 3.3.

Step 1: To detect the programming faults in program P , input P to PFDaC mechanism

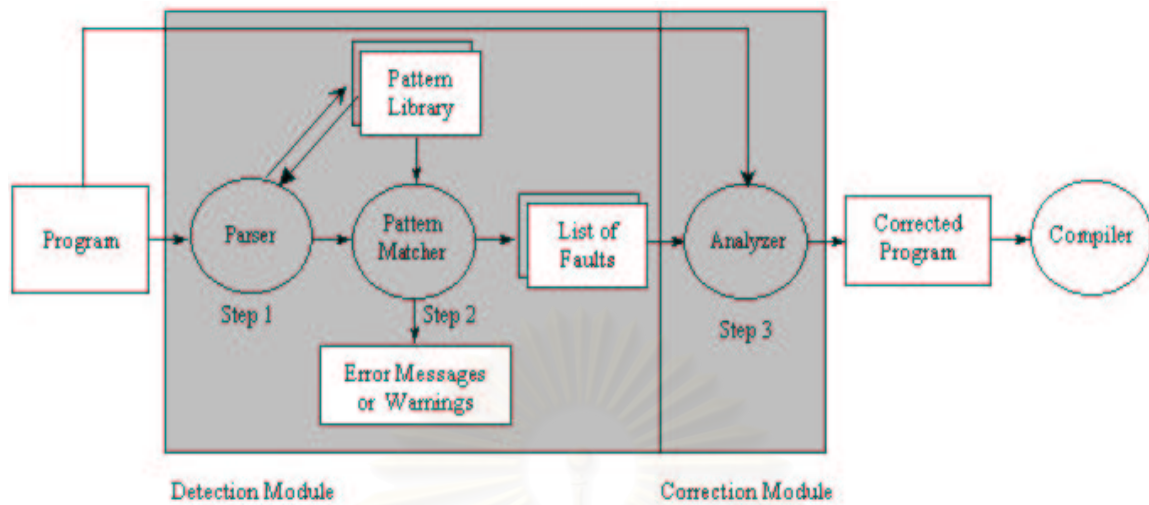


Figure 3.3: The detecting function of Precompiled Fault Detection and Correction.

for analyzing each statement in P . The *Parser* parses the source code to discover which statements contain the potential faults.

In PFDaC technique, the PDG is applied to easily describe the source code parsing. A graph in Figure 3.4 (a) [7] is a directed graph for the construction of a part of the program in Figure 3.4(b). The vertices represent statements in the program such as data types, variables, parameters, conditional branches, and assignment statements. The edges between the vertices indicate data, control dependence, or declaration. A data edge indicates a way in which a data value can be transmitted. For example, there is a data edge between the vertex for $i = 0$; and the vertex for $while\ i \leq 5$, which indicates that a value for i flows between these two vertices in Figure 3.4(a). A control edge indicates whether the destination vertex (e.g. $temp[i] = arr[i]$) is reached by the result of executing the source vertex (e.g. $while\ i \leq 5$). A declaration edge indicates the declaration of variables in programs (e.g. $arr[5]$). For example, a vertex $ind(arr) = 5$ means a size of arr index is 5.

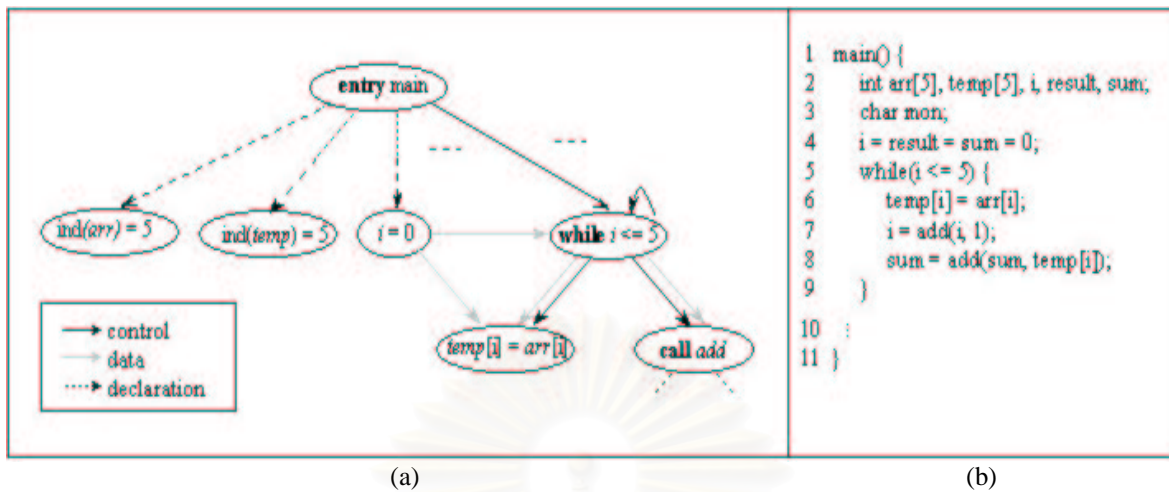


Figure 3.4: An example of system graph (a) for a part of program in Figure 1.1 shown on (b).

The build-in reliability features of Erlang are applied to the PFDaC mechanism. For example, a tuple [1], which is used to store a fixed number of elements, is data structures as an array in C. The number and type of elements in the used tuple are matched with the declared one. If it is not matched, an error message appears to inform the programmers.

In the PFDaC technique, a source code is parsed to look for the required variable declarations, functions, or statements, e.g. `int arr[5]`, `add(...)`. They match the pattern of PFDaC technique's faults in the *Pattern Library* described in Section 3.2. These required variable declarations or statements are, then, generated to be the new patterns in the *Pattern Library* by the *Parser*.

Step 2: The *Pattern Matcher* considers the used variables, function calls, 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 PFDaC technique as follow: assume that

```

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  }
```

Figure 3.5: An algorithm of a main functionality for detecting and correcting faults.

method D_i declares the detection of a fault type F_i in the program P_i . The *Pattern Matcher* creates the log file, $P_iF_i.log$. In the log file, there are n potential faults of F_i . An algorithm of main functionality of PFDaC technique is shown in Figure 3.5.

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 Figure 3.6. When the value of index i of *arr* in the *match* part does not match with its value in *pattern* ($ind(arr) = 5$), this fault is recorded in the list of faults. For example, when $i = 5$, it makes size of *arr* index is over its declaration (size of *arr* index is 6). An error message appears to warn the programmers and then, this fault is corrected in Step 3.

3.4 Correction Module

The aim of the correction module is to correct the faults detected from the detection module. Whenever any faults are detected, they must be fixed in the proper way. Otherwise, these faults may cause critical errors while the program is executed. Thus, the programmers cannot ignore these faults. After all detected faults are eliminated, the reliability of the programs is increased. To perform error correction, the PFDaC

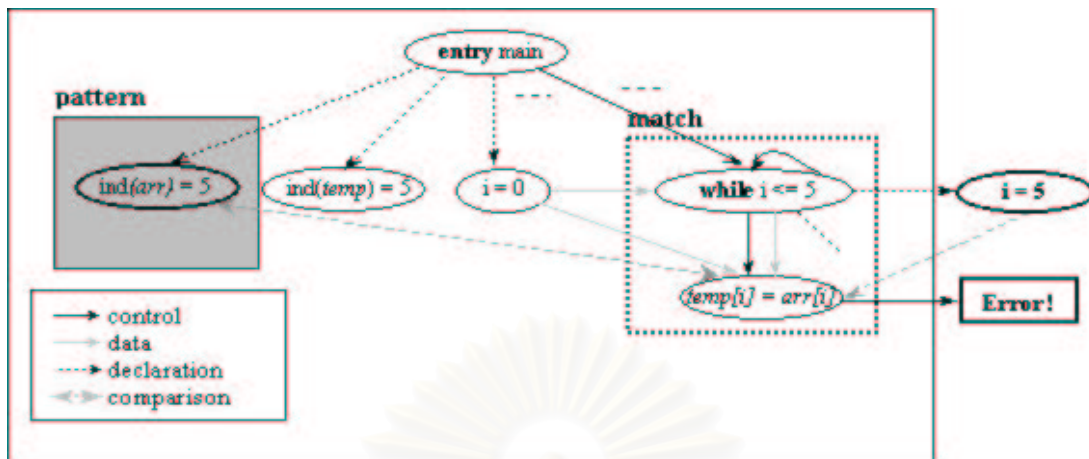


Figure 3.6: An example of the *pattern* and *match* graphs for the program in Figure 3.4(b).

correction module allows programmers to perform the correction, both manually or automatically. The architecture of the correction module is presented in Figure 3.7.

Step 3: The faults are automatically corrected by the correction function of PFDaC. Some fault corrections cannot be automated. 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 programmers.

The program source codes are parsed to discover which statements are needed to correct faults. The *Parser & Corrector* performs parsing and correcting using the information from each log file provided by the detection module. Each fault-record in the log file is generated to be the pattern in the *Pattern Library*. The log file also exhibits the faults' locations to the PFDaC correction mechanism (C_1, C_2, \dots, C_n in Figure 3.5). Then, the *Parser & Corrector* considers statements in the program source code to match the pattern of faults in the *Pattern Library*. The detected faults are corrected by the *Corrector* mechanism of PFDaC or by the programmers. Therefore, the outputs of

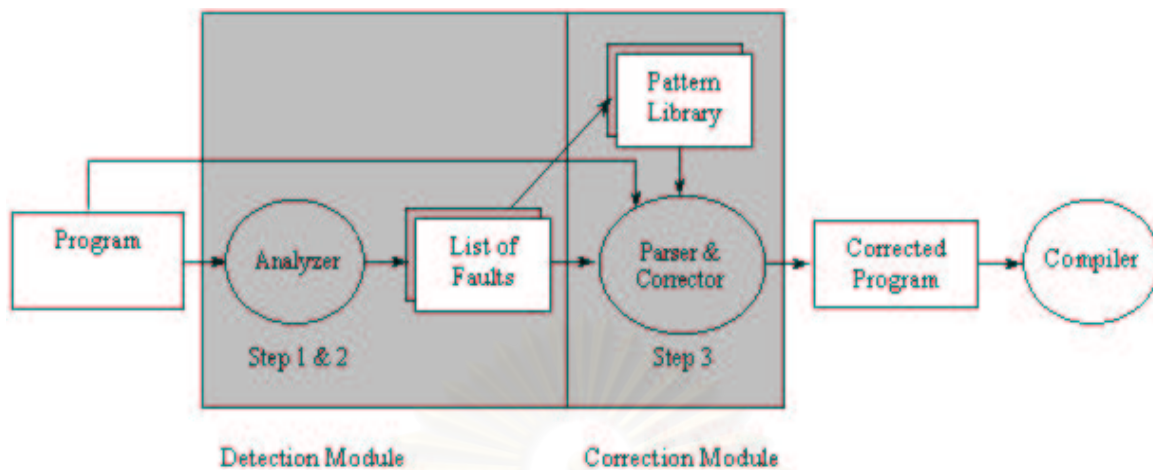


Figure 3.7: The correcting function of Precompiled Fault Detection and Correction technique.

PFDaC mechanism are the corrected programs. Then, these programs will be compile as a normal process without any hidden faults.

3.5 Complexity

Considering the algorithm in Figure 3.5, a program P with F fault types, and each fault type has N potential faults. Therefore, the number of detected fault are $F*N$ faults. However, the detection module in Section 3.3 can detect N faults of a fault type in one time detection. For example, in the testing program P_1 , there are three faults of the fault type F_1 . All of these faults, which are in the same fault type, are detected in one execution time for the input P_1 . Therefore, if the input program contains k fault types, PFDaC technique can detect all fault types by executing the program k times. The time complexity of detection module is $O(k)$.

3.6 Summary

In this chapter, the characteristics of the PFDaC technique are proposed. The PFDaC technique consists of two modules: the detection module and the correction module. The PDG and pattern matching are applied to consider faults in the source code.

The program source code are parsed by PFDaC for checking faults in the detection module. The PDG is used to describe the parsing source code. Each statement in programs is traced for debugging the faults. The pattern matching feature is applied to match used variables, function calls, or statements with the pattern in the *Pattern Library*. The outputs of this module are errors or warning messages, and the log files. The faults detected in the detection module can be solved in the correction module. The corrected programs are not only the outputs of the correction module but also the outputs of PFDaC.

CHAPTER IV

The implementation and Experimental Results of the Proposed Technique

4.1 Implementation of Precompiled Fault Detection and Correction Technique

Referring to the PFDaC technique architecture and algorithms in Chapter 3, this technique is implemented using C to perform the fault detection and correction.

The inputs of the PFDaC mechanism are assumed to be the applications written in C. The execution of mechanism starts by asking programmers to enter a program file. The interfaces of the PFDaC's prototype are illustrated in Appendix A.

The main program algorithm is shown in Algorithm 4.1. This algorithm is the full details of the algorithm presented in Figure 3.5. Algorithm 4.2 - Algorithm 4.5 show the examples of fault detection algorithms ($D_1()$, $D_2()$, ..., $D_n()$ in Figure 3.5). An algorithm for detecting array indices is shown in Algorithm 4.2. The array variables in the source file are inspected by comparing the declared indices with the used indices. If the fault of an array index out-of-bound is encountered, it is recorded into the log file named as the *array_log* file. This method is applicable for both static and dynamic array

Algorithm 4.1 An algorithm of a main program

```

1: function main(P)
2:   if check_sarray(char *name) == TRUE then
3:     correct_sarray(char *name);
4:   if check_function(char *name) == TRUE then
5:     correct_function(char *name);
6:   if check_switch(char *name) == TRUE then
7:     correct_switch(char *name);
8:   if check_darray(char *name) == TRUE then
9:     correct_darray(char *name);
10:  if check_loop(char *name) == TRUE then
11:    correct_loop(char *name);
12:  else
13:    compile P;

```

index detection.

The case of function argument types is shown in Algorithm 4.3. The line of the function declaration are recorded in the log file, *functional_log* file. The type of arguments in the function call are compared with argument types of the declared function. If an argument type of a parameter in the function call is not matched with the pattern in the *functional_log* file, this fault is recorded in the *functional_log* file.

Algorithm 4.4 illustrates the detection of no-default-case in a *switch* statement. If there is a *switch* statement without the *default* case, the warning messages appear to inform the programmer.

The other fault that is usually be detected during the execution time is the infinite loop; Algorithm 4.5 shows the algorithm of infinite loop detection. If there is no any statement which changes the value of a variable used in the conditional statement described in Chapter 1, the error message appears to inform the programmer.

In order to implement all detection mechanisms into PFDaC, each detection

Algorithm 4.2 An algorithm of static array index detection

```

1: function check_sarray(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:         endif
8:       endwhile
9:     else
10:      if item == array variable
11:        compare this array index with index in log file;
12:      endif
13:    endwhile

```

Algorithm 4.3 An algorithm of function argument type detection

```

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 type
5:       in a functional log file;
6:     else
7:       if item == name of function call
8:         compare function call and declared function in log file;
9:       endif
10:    endwhile

```

mechanism is implemented as the header file (.h) and embedded in PFDaC using *#include* statement. Therefore the source file is input to the detection mechanism before being compiled by its compiler.

The consequent of the *#include* statement is that the size of the PFDaC from using these header files is the same as direct implementation of all detection methods in the PFDaC at once. However, there is a benefit of implementing each detection into an individual file. The reason of creating each detection mechanism as a header file of C is that any application can embed this mechanism individually without PFDaC. Therefore,

Algorithm 4.4 An algorithm of no-default-case detection

```

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

```

without PFDaC, every program still can be verified using these headers.

Referring to Algorithm 4.1 - Algorithm 4.5, the data stored in each log file are errors and warning messages based on each fault case. Examples of log files are shown in Figure 4.1 - Figure 4.3.

According to the design of PFDaC, all correction mechanisms are implemented as the header files as same as the detection methods. The responsibility of each correction mechanism is to trace each record in the log file related to each correction technique. For example, the correction method for the *switch* statement without the *default* case reads the record in the *switch_log* file. Once a record is read from the log file, the correction mechanism starts and the programmer chooses the proper correction commands to be added or modified. After the fault case in the log file is corrected, the record is flagged. The compilation will automatically start when all records in every log file are flagged.

Algorithm 4.5 An algorithm of infinite loop detection

```

1: function check_loop(char *name)
2:   while read next character until end of file
3:     if item == "while"
4:       while read next character until item == ')'
5:         if item == variable in conditional statement
6:           put variable name in a loop log file;
7:         endif
8:       endwhile
9:     while read next character until item == '}'
10:      if item == variable
11:        compare this variable with pattern of statement in log file;
12:      endif
13:    endwhile
14: endwhile

```

Code segment	Log file	
	Name	Size
1 main() {		
2 int arr[5], temp[5], i, result, sum;	1. arr	5
3 char mon;	2. temp	5
4 i = result = sum = 0;		
5 while(i <= 5) {		
6 temp[i] = arr[i];	1. arr	6
7 i = add(i, 1);	2. temp	6
8 sum = add(sum, temp[i]);		
9 }		
10 :		
11 }		

→ Pattern (points to the first two rows of the log file)
 → Error! (points to the first row of the second log file entry)
 → Error! (points to the second row of the second log file entry)

Figure 4.1: An example of a log file: array index out-of-bound detection.

Code segment	Log file	
	Name	Size
1 main() {		
2 char str1[] = "computer";	1. str1	9
3 char str2[] = "science";	2. str2	8
4 strcpy(str1, "mathematical");	1. str1	13
5 printf("%s %s", str1, str2);		
6 }		

→ Pattern
 → Error!

Figure 4.2: An example of a log file: string array index out-of-bound detection.

4.2 Experimental Results

Since the cases of incorrectness of array indices, the mismatch of function arguments' types, and no-default-case in *switch* statements are mostly occur in C, a set of programs containing these cases is implemented to validate the efficiency of PFDaC. The comparisons among the normal execution process of these files and the process that pass through the PFDaC are performed.

There are 20 simulated program files in C. The first group of these program files, *File1.c*, *File2.c*, *File3.c*, *File4.c*, *File5.c*, *File6.c*, *File7.c*, and *File8.c*, contain two, one, one, two, one, one, one, and one, respectively, array indices out-of-bound. The second group of simulated program files, *File9.c*, *File10.c*, *File11.c*, *File12.c*, *File13.c*, *File14.c*, *File15.c*, and *File16.c*, have three, one, one, one, one, one, one, and one, respectively, faults about passing wrong type of function arguments. In the last group, *File17.c*, *File18.c*, *File19.c* and *File20.c*, each holds one of no-default-case in the *switch* statement.

Code segment	Log file	
	Function Name	Argument Type
1 main() {		
2 int arr[5], temp[5], i, result, sum;		
3 char mon;		
4 i = result = sum = 0;		
5 while(i <= 5) {		
6 temp[i] = arr[i];		
7 i = add(i, 1);	1. add	1. int
8 sum = add(sum, temp[i]);	2. add	2. int
9 }
10 ...		
11 result = add(sum, mon)	3. add	1. int
12 print(result);		2. char
13 }		
14 int add(int x, int y) {	1. add	1. int
15 return(x+y);		2. int
16 }		

→ Error!

→ Pattern

Figure 4.3: An example of a log file: passing wrong type of function argument detection.

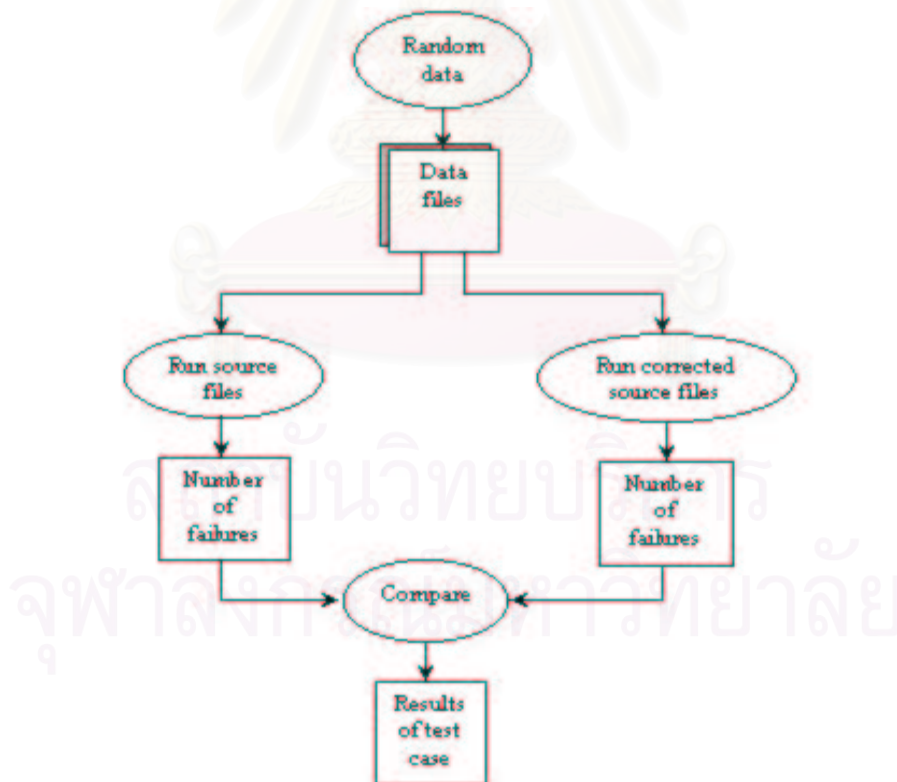


Figure 4.4: The flowchart of the steps involved in the evaluation of using the PFDaC technique.

Table 4.1: The number of programming faults in each C application.

Application	# Faults	# Failures before correction	# Failures after correction
<i>File1.c</i>	2	900	0
<i>File2.c</i>	1	511	0
<i>File3.c</i>	1	2800	0
<i>File4.c</i>	2	332	0
<i>File5.c</i>	1	1000	0
<i>File6.c</i>	1	1118	0
<i>File7.c</i>	1	397	0
<i>File8.c</i>	3	74	0
<i>File9.c</i>	1	1000	0
<i>File10.c</i>	1	440	0
<i>File11.c</i>	1	1000	0
<i>File12.c</i>	1	1000	0
<i>File13.c</i>	1	3000	0
<i>File14.c</i>	1	1000	0
<i>File15.c</i>	1	400	0
<i>File16.c</i>	1	2250	0
<i>File17.c</i>	1	477	0
<i>File18.c</i>	1	165	0
<i>File19.c</i>	1	0	0
<i>File20.c</i>	1	825	0

Table 4.1 lists a number of programming faults existing in the applications and the number of failures resulting from the detected faults.

Experiments were conducted follow the methodology described in Section 4.2; PFDaC was executed for analyzing each application. Figure 4.4 illustrates the flowchart of the PFDaC technique evaluation steps. After implementing PFDaC to detect and solve the faults in applications, a set of simulation data (1,000 data sets) has been applied in order to measure the reliability of the software. The resulting graphs of the running software using the data test sets in each case of faults before and after using the PFDaC technique are presented in Figure 4.5 - Figure 4.24. These graphs confirm that the

proposed PFDaC technique can completely remove the failures from the applications.

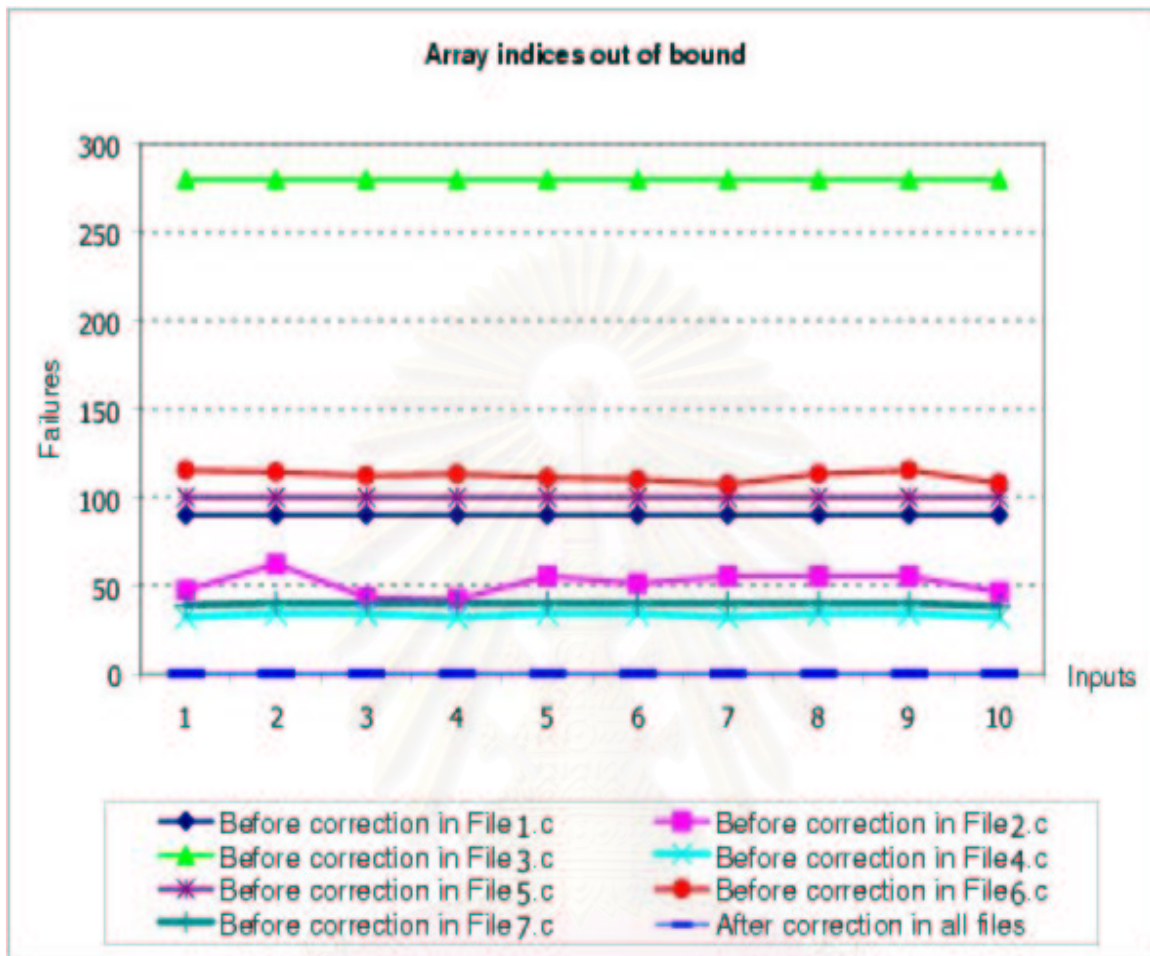


Figure 4.5: The resulting graph of array indices out of bound cases before and after correcting by the PFDaC technique in *File1.c*, *File2.c*, *File4.c*, *File5.c*, *File6.c*, and *File7.c*.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

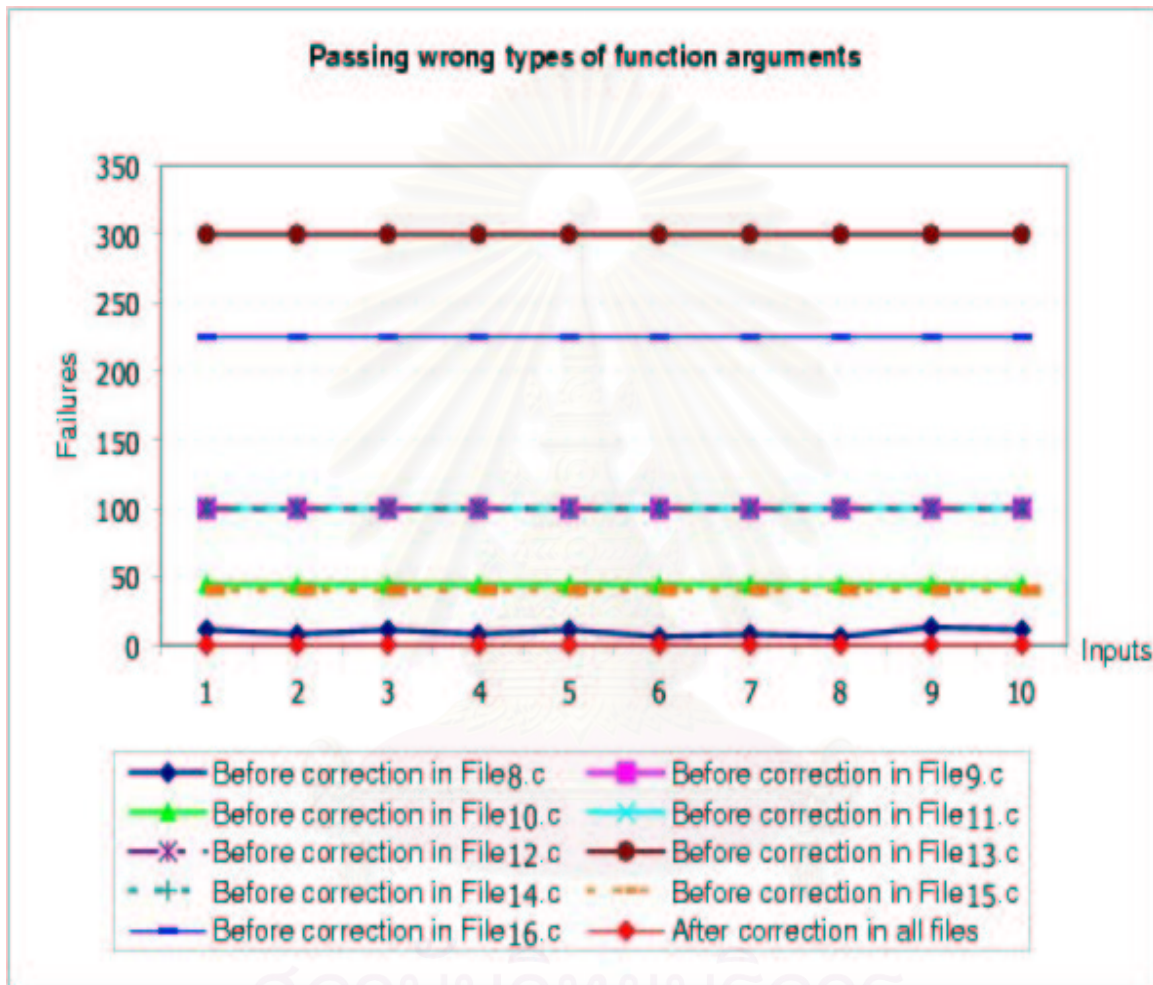


Figure 4.6: The resulting graph of passing wrong types of function arguments cases before and after correcting by the PFDaC technique in *File8.c*, *File9.c*, *File10.c*, *File11.c*, *File12.c*, *File13.c*, *File14.c*, *File15.c*, and *File16.c*.

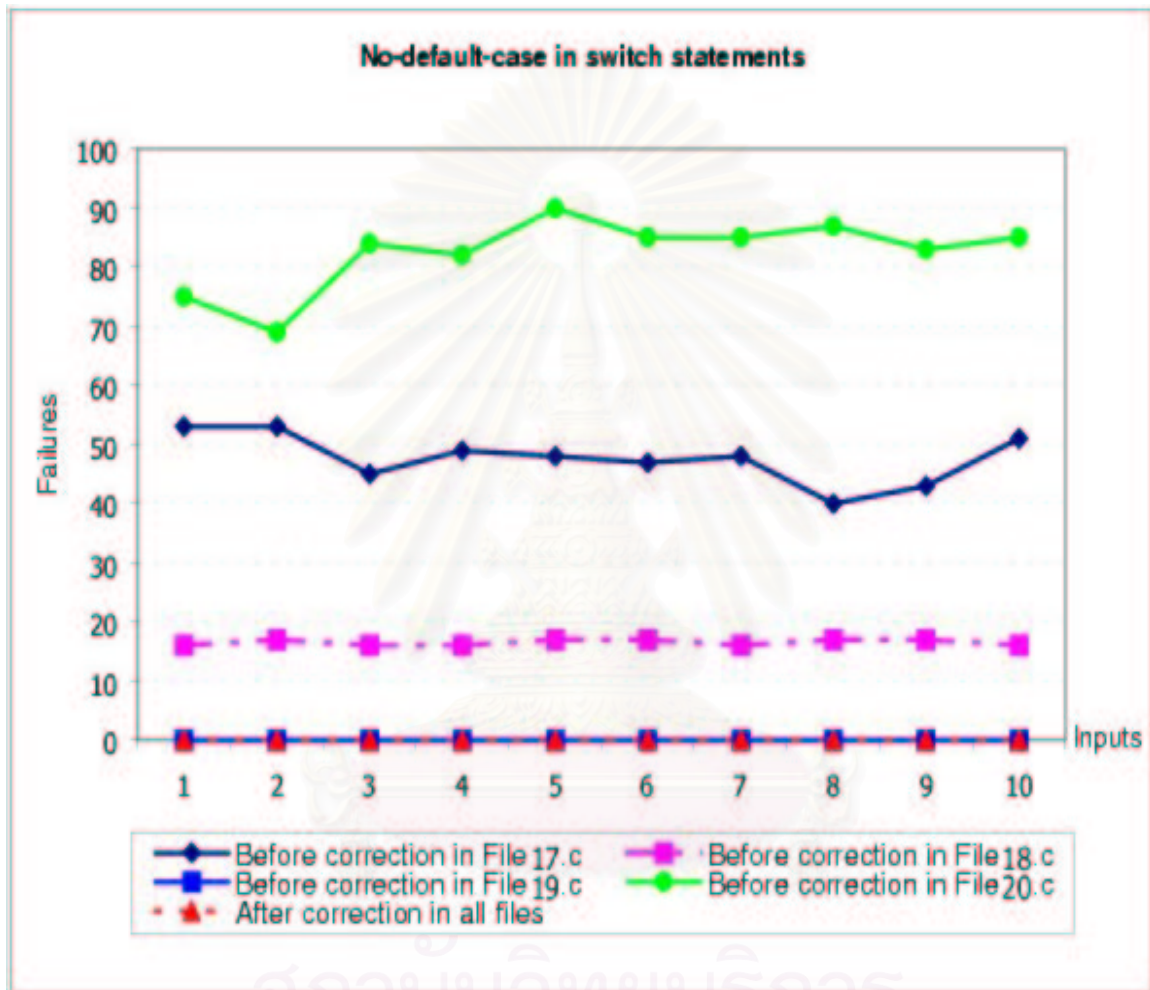


Figure 4.7: The resulting graph of no-default-case in the *switch* statements cases before and after correcting by the PFDaC technique in *File17.c*, *File18.c*, *File19.c*, and *File20.c*.

CHAPTER V

Theoretical Analysis

Referring to the implementation and testing of PFDaC in Chapter 4, the experimental results confirmed that the PFDaC technique is efficient and able to increase the reliability of the software products during the development process. However, the argue of general test cases may be arisen. Therefore, this chapter presents the theoretical analysis to ensure that the proposed method can be applied to any languages and the reliability of the software can be obtained.

Definition 1. $D = (C, F)$ where

C is a finite set of all commands in source codes.

F is a finite set of all faults.

D is called the software fault detection domain.

5.1 Fault Detection

Definition 2. Let F' be a set of faults detected by the PFDaC technique.

$$F' = \{f' \mid f' \in F'\}$$

Definition 3. Let F_u be a set of undetected faults or a set of faults which are not in F' .

$$F_u = F - F' \quad \text{or} \quad F_u = \{f_u \mid f_u \in F, f_u \notin F'\}$$

Definition 4. Let D be the software fault detection domain. F is a set of faults in D .

$$F = \{f \mid f \in F' \cup F_u\}$$

Definition 5. Let C be a set of commands in source codes. Let F' be a set of faults detected by the PFDaC technique. D_f is called the detection function of the PFDaC technique.

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

Lemma 1. Let D be the software fault detection domain and let F be a set of faults in D . Let F' is a set of fault detected by the PFDaC technique. Then $F' \subset F$.

Proof. Let C be a set of commands in source codes. Generally, c contains f or c does not contain f where $c \in C, f \in F$. By Definition 5, $f' = D_f(c)$ where $f' \in F', c \in C$, that is, c contains f' . By Definition 2 and Definition 4, $f' \in F$ for every $f' \in F'$, but $\exists f \notin F'$. Thus, $F' \subset F$.

Definition 6. Let C be a set of commands in D . Let C' be a set of commands containing the faults detected by the PFDaC technique. Let C^u be a set of commands containing the undetected faults or the faults which are not in F' . Let C^n be a set of faultless commands.

$$C = \{c \mid c \in C' \cup C^u \cup C^n\}$$

Definition 7. Let C and F be a set of commands in source codes and a set of faults detected, respectively. The product set $C \times F$ is defined as

$$C \times F = \{(c, f) \mid c \in C, f \in F\}$$

Definition 8. Let D be the software fault detection domain. $M(c, f)$ is a Boolean function of fault detection of $C \times F$ and

$$M(c, f): C \times F \rightarrow B \text{ where } c \in C, f \in F, B = \{TRUE, FALSE\}.$$

Definition 9. $M(c, f)$ is true if and only if c contains f .

Definition 10. Let C and F be a set of the commands in source codes and a set of the faults detected, respectively. By the **detection result** R , we mean that the R consists of the elements (c, f) in $C \times F$ for which $M(c, f)$ is true.

$$R = \{(c, f) \mid c \in C, f \in F \text{ and } M(c, f) = TRUE\}$$

Theorem 1. Let D be the software fault detection domain. Let $R = (C, F)$ be a set of detection results where C is a set of commands in D and F is a set of faults in D . Let F' be a set of faults detected by the PFDaC technique. If $R' = (C, F')$ then $R' \subset R$.

Proof. Assume that R' is a set of detection results where C is a set of commands, F' is a set of faults detected by the PFDaC technique, and $M(c, f')$ is true. That is, $(c, f') \in R'$ where $c \in C, f' \in F'$. By Definition 2 and Lemma 1, $f' \in F$ and $F' \subset F$. Then, $(c, f') \in R$, but $\exists(c, f) \notin R'$, by Definition 4 and Lemma 1. Thus, $R' \subset R$.

5.2 Fault Correction

Definition 11. Let F' be a set of faults detected by the PFDaC technique. Let C^r be a set of corrected commands in source codes. C_f is called the correction function of the PFDaC technique.

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

Definition 12. Let C^{new} be a set of new commands which have been corrected in source codes. Let C^r be a set of corrected commands by the PFDaC technique. Let C^u be a set of commands containing the undetected faults or the faults which are not in F' . Let C^n be a set of faultless commands.

$$C^{new} = \{c \mid c \in C^r \cup C^u \cup C^n\}$$

Definition 13. $N(c_r, f)$ is a Boolean function of fault detection of $C^r \times F$ and

$$N(c_r, f): C^r \times F \rightarrow B \text{ where } c_r \in C^r, f \in F, B = \{TRUE, FALSE\}.$$

Definition 14. $N(c_r, f)$ is true if and only if c_r does not contain f .

Definition 15. Let C^r and F be a set of the corrected commands in source codes and a set of the faults, respectively. By the **correction result** T , we mean that the T consists of the elements (c_r, f) in $C^r \times F$ for which $N(c_r, f)$ is true.

$$T = \{(c_r, f) \mid c_r \in C^r, f \in F \text{ and } N(c_r, f) = TRUE\}$$

Theorem 2. Let $T = (C^r, F)$ be a set of correction results where C^r is a set of corrected commands and F is a set of faults. If $T' = (C^r, F')$ then $T' \subset T$.

Proof. Assume that T' is a set of correction results where C^r is a set of corrected commands, F' is a set of faults detected by the PFDaC technique, and $N(c_r, f')$ is true. That is, $(c_r, f') \in T'$ where $c_r \in C^r, f' \in F'$. By Definition 2 and Lemma 1, $f' \in F$ and $F' \subset F$. Then, $(c_r, f') \in T$, but $\exists (c_r, f') \notin T'$, by Definition 4 and Lemma 1. Thus, $T' \subset T$.

CHAPTER VI

Discussion and Conclusion

6.1 Discussion

The software reliability is a significant feature of a good software implementation. However, the software that has high-level of reliability is hard to be obtained, since some faults cannot be detected during the software development process. Consequently, these faults cause unexpected problems or the serious accidents whenever they arise.

At present, there are many techniques for detecting faults, whereas most of these techniques detect faults while software are running. Thus, the software process is interrupted when a fault occurs. Therefore, the PFDaC technique is proposed to detect and correct faults before the program is compiled. After the PFDaC process, the software applications are utilized and the number of significant hidden faults is lower than the general software. As the results presented in Chapter 4 and the theoretical analysis in Chapter 5, these processes confirm the capability of the PFDaC technique in eliminating the critical faults that arise in the programs. Therefore, the application software filtered by the PFDaC technique will be efficient and software as the users expected. However, since the concept of the proposed PFDaC technique is the pre-compilation fault detection, the PFDaC technique excludes the detection of cases that the variables' values are

generated at run-time such as dynamic arrays and loops that are relied on the run-time values.

Since the process of PFDaC generates the log files for storing all warnings and faults cases, the size of each log file is depended on the number of cases that has been detected. However, each log file is a text file. Therefore, the total size of this file for the entire process will not be too large to be managed. Moreover, the overhead of PFDaC to process the source file is low as it treats the input as a sequential text file. Therefore, the entire process of PFDaC is small and does not affect to the total compile time. So, applying PFDaC to create a reliable software is an efficient method that requires small resources both for the CPU and the disk spaces.

Although the experiment was simulated in C environment, this technique is not limited only the C language. Therefore, if the programmers applied this technique to any languages, reliability of the final software products can also be ensured to be achieved.

6.2 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.

This research has proposed a new and significant technique called **Precompiled Fault Detection and Correction (PFDaC)** technique. The concept of pattern matching is applied to this technique for detecting and correcting hidden faults in the programs. The C programming language is used to be the case study. The PFDaC technique has

been tested by running a set of simulated programs with a test set of data, and the number of faults is counted before and after the program passes through the PFDaC mechanism. The results show that the number of faults that were detected by PFDaC is reduced or totally eliminated after the PFDaC process. Therefore, the program will not be affected by the detected faults while executing.

The applications that can run without termination or interruption from its internal faults is certainly classed as reliable software. The PFDaC 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 the PFDaC technique is shown to be much more reliable than software that is directly compiled. Therefore, our technique can guarantee the reliability of all the application software passed through.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

References

- [1] Armstrong JL, Viriding SR, and Williams MC. (1993). *Concurrent Programming in Erlang*, Englewood Cliffs, New Jersey: Prentice Hall.
- [2] Anderson P, Reps T, and Teitelbaum T. (2003). Design and Implementation of a Fine-Grained Software Inspection Tool. *IEEE Transactions on Software Engineering*. 29, 8: 721-733.
- [3] Cowan C, Beattie S, Day R-F Pu C, Wagle P, and Walthinsen E. (1998). Automatic Detection and Prevention of Buffer Overflow Attacks. *7th USENIX Security Symposium*.
- [4] Deeprasertkul P and Bhattarasinee P. (2003). Software Fault Detection in C Programs. *12th International Conference on Intelligent and Adaptive Systems and Software Engineering*. San Francisco, USA., 192-195.
- [5] Drake J, Mashayekhi V, Riedl J, and Tsai W. (1991). *A Distributed Collaborative Software Inspection Tool: Design, Prototype, and Early Trial*. Technical Report TR-91-30, University of Minnesota.
- [6] Fagan M. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*. 15, 3: 182-211.
- [7] Ferrante J, Ottenstein K, and Warren J. (1987). The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*. 3, 9: 319-349.

- [8] Fetzer C, Felber P, and Hogstedt K. (2004). Automatic Detection and Masking of Nonatomic Exception Handling. *IEEE Transactions on Software Engineering*. 30, 8: 547-560.
- [9] Ganapathy V, Jha S, Chandler D, Melski D, and Vitek D. (2003). Buffer Overrun Detection using Linear Programming and Static Analysis. *10th ACM conference on Computer and Communication Security*. 345-354.
- [10] Ghezzi C, Jazayeri M, and Mandrioli D. (2003). *Fundamentals of Software Engineering*. International edition. Upper Saddle River, New Jersey: Prentice-Hall.
- [11] Hagemester J R, Bhansali S, and 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*. IEEE computer society. Washington DC, USA., 209-214.
- [12] Harbison S P and Steele Jr G L. (1995). *C: A Reference Manual*. Fourth Edition. Upper Saddle River, New Jersey: Prentice-Hall.
- [13] Hausman B. (1994). *Turbo Erlang: Approaching the Speed of C, Implementations of Logic Programming Systems*. Vancouver: Kluwer Academic Publishers, 119-135.
- [14] Larson E and Austin T. (2001). High coverage detection of input related security faults, *12th USENIX Security Symposium*.
- [15] Macdonald F, Miller J, Brooks A, Roper M, and Wood M. (1995). A Review of Tool Support for Software Inspection. *Proceeding 7th International Workshop Computer-Aided Software Engineering (CASE-95)*.

- [16] Macdonald F. (1998). *Computer-Supported Software Inspection*. PhD thesis, Department of Computer Science, University of Strathclyde.
- [17] Necula G C, McPeak S, and Weimer W. (2002). CCured: Type-Safe Retrofitting of Legacy Code. *ACM Conference on the Principles of Programming Language (POPL)*.
- [18] Paul S and Prakash A. (1994). A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*. 20, 6: 463-474.
- [19] Rady de Almeida Jr J, Batista Camargo Jr J, Abrantes Basseto B, and Miranda Paz S. (2003). *Best Practices in Code Inspection for Safety-Critical Software*. IEEE Software.
- [20] Rational the Software Development Company. Rational PurifyPlus for Unix. http://www.rational.com/products/pgc/pplus_ux.jsp.
- [21] Royce T. (1996). *C Programming*. New Zealand: Macmillan Press Ltd.
- [22] Sembugamoorthy V and Brothers L. (1990). ICICLE: Intelligent Code Inspection in a C Language Environment. *Proceeding 14th Annual Computer Software and Applications Conference*, 146-154.
- [23] Seward J. The Design and Implementation of Valgrind: Detailed Technical Notes. <http://developer.kde.org/~sewardj/>.
- [24] Spuler D A. (1994). *C++ and C Debugging, Testing, and Reliability: The Prevention, Detection, and Correction of Program Errors*. Englewood Cliffs, New Jersey: Prentice-Hall.

- [25]Wagner D. (2000). *Static Analysis and Computer Security: New Techniques for Software Assurance*. Ph.D. Thesis, UC Berkeley.
- [26]Xie Y, Chou A, and 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)*.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



Appendices

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Appendix A

According to the proposed technique architecture and algorithms in Section 4.1, it is implemented by using C language to perform the detection and correction. The input of our mechanism is an application in C. The execution of the proposed technique starts with asking programmer to enter the program file in C, as shown in Figure A.1.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c):
```

Figure A.1: The interface prototype of the proposed technique for file inputs.

A.1 Static Array

Figure A.2 presents a case of index out of bound of array *arr*. From Figure A.2, if the programmer presses 'y', the declaration of *arr* will be changed as the informed screen, otherwise it is unchanged.

A.2 Function

Figure A.3 shows the error message when our mechanism detects the parameter's type mismatch (*mon*) from *add* function. When the programmer presses 'y', the mechanism swaps to the program editor so that the programmer can correct the error immediately.

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c): Addno.c

Error line 5: an index of 'arr' is out-of-bound.

Do you want to change a declaration of 'arr[5]' to 'arr[6]'?
Yes press [y] or No press[n]:

```

Figure A.2: The interface prototype of array indices detection.

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c): Addno.c

Error line 11: incorrect type of 'mon' an argument in 'add' function.

Do you want to change it?
Yes press [y] or No press[n]:

```

Figure A.3: The interface prototype of passing the wrong function argument type detection.

A.3 *switch* Statement

Another fault that can be detected is *default* case missing. Figure A.4 shows the warning message obtained from mechanism when *switch* statement does not contain the *default* case. If the programmer presses 'y', the mechanism will insert a *default* statement without any actions defined, and swap the checking mode to the program editor. Then the function of *default* case will be managed by programmer.

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c): Landing.c

Error line 11: No 'default' case in switch statement.

Do you want to add default case in switch statement?
Yes press [y] or No press[n]:

```

Figure A.4: The interface prototype of *switch* statement case detection.

A.4 Dynamic Array

Figure A.5 shows a case of dynamic array index out-of-bound detection.

A.5 Infinite Loop

Figure A.6 presents the interface of detecting infinite loop.

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c): Addnol.c

Error line 6: an index of 'arr' is out-of-bound.

Do you want to change it?
Yes press [y] or No press[n]:

```

Figure A.5: The interface prototype of dynamic array index detection.

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                          Program Files Checking
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Input file name (xxx.c): Loop.c

Error loop at line 5: No statement for changing a value of 'ok'.

Do you want to add it?
Yes press [y] or No press[n]:

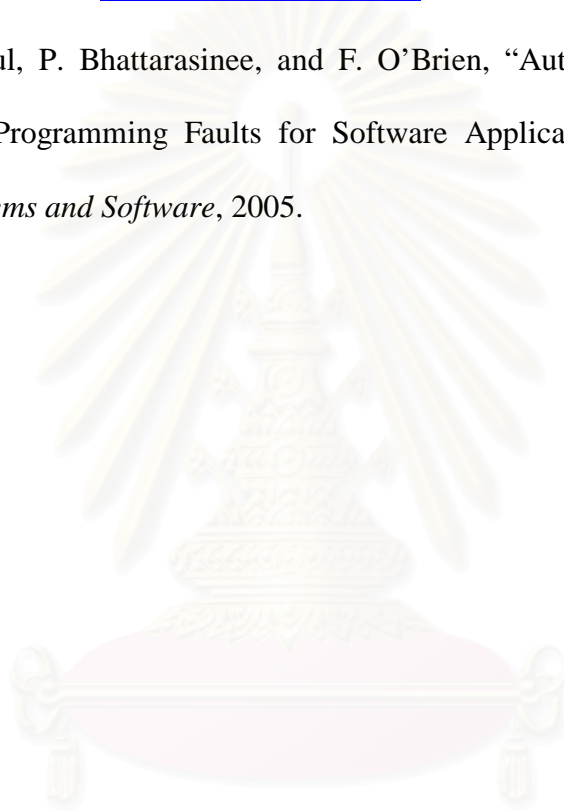
```

Figure A.6: The interface prototype of infinite loop detection.

Appendix B

This section presents a following journal paper generated from this dissertation. This paper is available online at www.sciencedirect.com.

- P. Deeprasertkul, P. Bhattarasinee, and F. O'Brien, "Automatic Detection and Correction of Programming Faults for Software Applications.", *Elsevier: The Journal of Systems and Software*, 2005.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



ELSEVIER

Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

The Journal of Systems and Software xxx (2005) xxx–xxx

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

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).

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

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

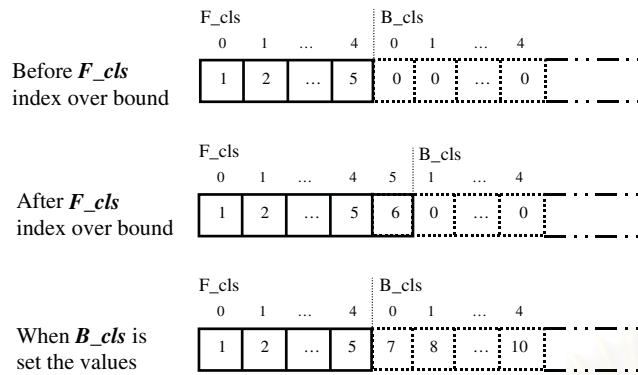


Fig. 2. Memory allocations for F_cls 's index out of bound; the replacement of F_cls with B_cls .

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 Bhattacharjee, 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; Hagemester 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 (Hagemester 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 (Hagemester 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	$\$a_name[\dots]$
Function	$\$f_name(\dots)$

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 $\$a[\dots]$. 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 $\$f_INSURANCE(\dots)$ 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'' 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'' if it is not a fault in F' .

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

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 .

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

Definition 3. Let S'' 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 .

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

When all faults in F' are corrected, all corrected statements S'' 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 be-

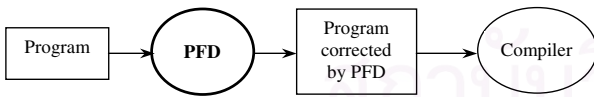


Fig. 3. Precompiled Fault Detection in context.

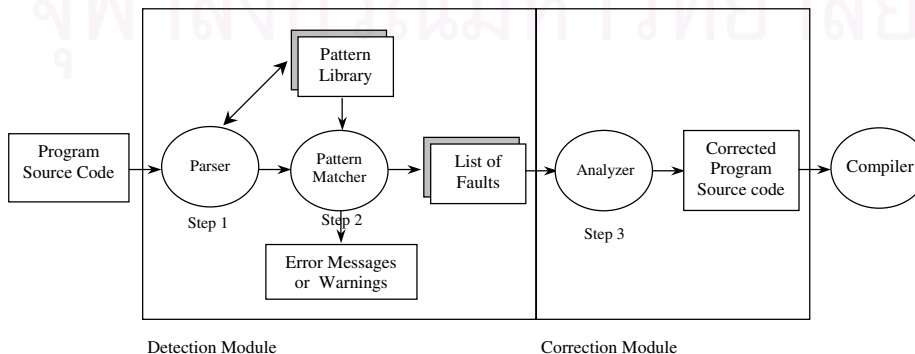


Fig. 4. The functionality of Precompiled Fault Detection.

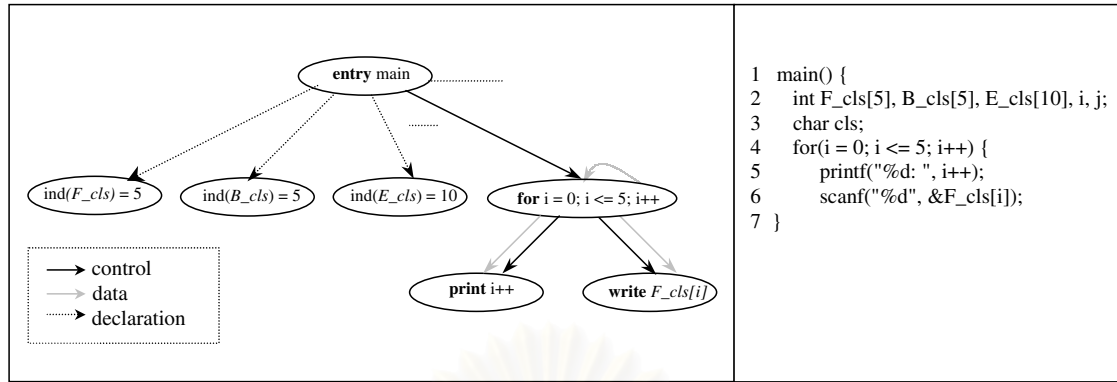


Fig. 5. An example of system graph (a) for a part of program in Fig. 1 shown on (b).

tween 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 $for(i = 0; i \leq 5; i++)$ 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 (e.g. $print\ i++$, $write\ F_cls[i]$) is reached by the result of executing the source vertex (e.g. $for(i = 0; i \leq 5; i++)$). 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 main() {
2   int F_cls[5], B_cls[5], E_cls[10], i, j;
3   char cls;
4   for(i = 0; i <= 5; i++) {
5     printf("%d: ", i++);
6     scanf("%d", &F_cls[i]);
7   }

```

```

1 Function main_PFD_function(P) {
2   if ( $D_1()$  == True) then  $C_1()$ ;
3   if ( $D_2()$  == True) then  $C_2()$ ;
4   :
5   if ( $D_n()$  == True) then  $C_n()$ ;
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

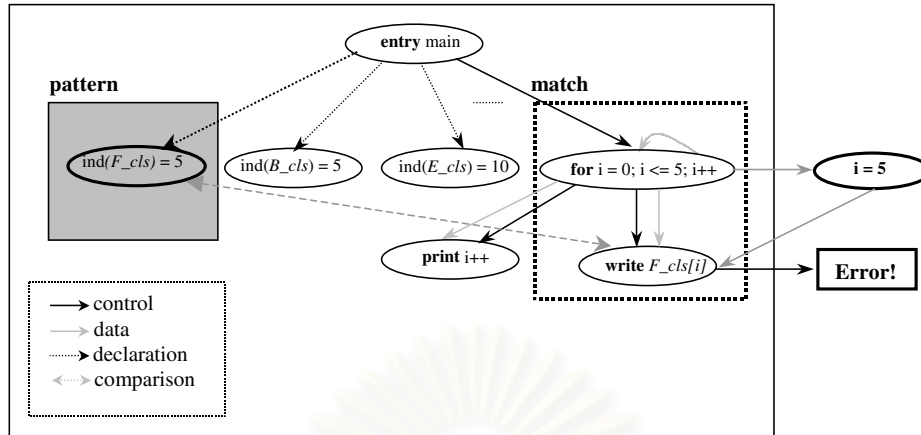


Fig. 7. An example of the *pattern* and *match* graphs for the program in Fig. 5(b).

the detection module. The log file exhibits the fault locations to PFD correction mechanism (C_1, C_2, \dots, 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 ($D_1(), D_2(), \dots, D_n()$ 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

Table 3
The number of programming faults in each C application

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

and medical system. A source file *S_Darray.c* and *Med-Ord.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

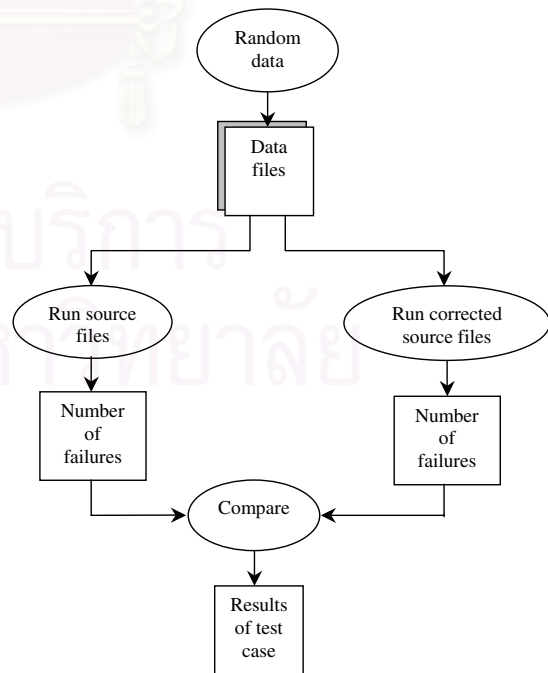


Fig. 10. A flowchart of the steps involved in the evaluation of using PFD.

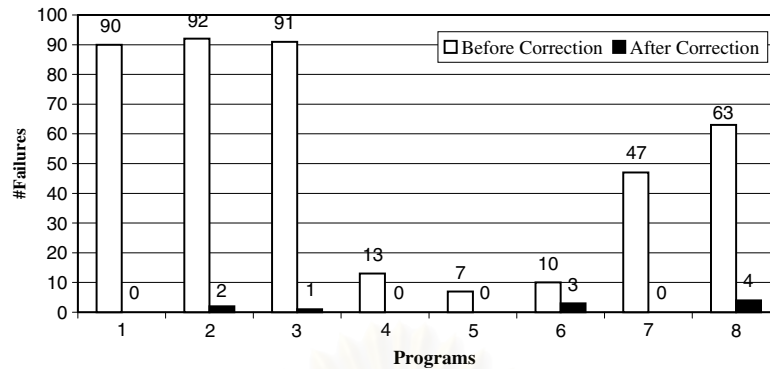


Fig. 11. A resulting graph before and after correcting by PFD.

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 *Seat-Price.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 pre-processed 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.

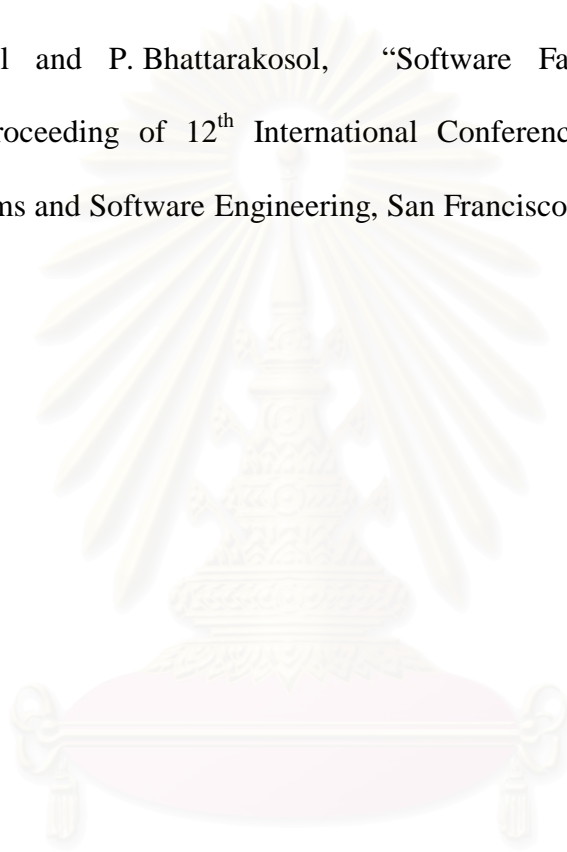
References

- Armstrong, J.L., Viriding, 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).
- Hagemester, 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).

Appendix C

This section presents a following conference paper which is a part of this dissertation.

- P. Deeprasertkul and P. Bhattarakosol, “Software Fault Detection in C Programs.”, Proceeding of 12th International Conference on Intelligent and Adaptive Systems and Software Engineering, San Francisco, USA., June 9-11, pp. 192-195, 2003.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

SOFTWARE FAULT DETECTION in C PROGRAMS

P. Deeprasertkul

Department of Mathematics, Faculty
of Science, Chulalongkorn University,
Phayathai Road, Patumwan, Bangkok,
Thailand, 10330.

Prattana.D@student.chula.ac.th

P. Bhattarakosol

Department of Mathematics, Faculty
of Science, Chulalongkorn University,
Phayathai Road, Patumwan,
Bangkok, Thailand, 10330.

Bpattara@sc.chula.ac.th

F. O'Brien

Royal Melbourne Institute of
Technology, Melbourne,
Australia.

Abstract

The analysis of software failures is significant for improving the software reliability. Therefore we need to understand and detect faults that are causes of the failures in order to improve software reliability.

We studied the Erlang programming language, a language that is used for high reliability software. Its faults and failures occur close together because of the pattern matching supported in Erlang. Therefore, faults which occur in Erlang can be easily and rapidly detected. The paper proposes how the lessons learnt from the Erlang infrastructure can be applied in C programming language environment. The proposed fault detection software has been created to perform as a software guard that can rapidly detect faulty code in C programming language. This detection software will operate at compiled time.

Keywords: Software reliability, Software failures, Software faults, Erlang programming language, Pattern matching.

1 INTRODUCTION

In software reliability, analysis of software failures is a very important subject. The evaluation of software reliability cannot be done without software failure data. Therefore, to improve the software reliability, we need to understand how failures occur, and how faults that cause the software failure are detected. Furthermore, improved software engineering techniques, better programming languages and better quality management are very important factors in improving software reliability[18].

There are two approaches that have been widely studied to improve the reliability of software. The first one is fault avoidance. It is the avoidance of faults that are detected before the software is delivered to the customers. Another approach is fault tolerance [5] where faults are detected during software execution. However, it takes a lot of time and money to develop this fault tolerant architecture. In addition, fault-free software is very difficult to develop in structured programming languages, such as C because the constructs of these programming

languages such as array index and function argument passing, often lead to software failures. Since, the C program structure is generally large, when some failures occur, it is time-consuming to detect faults which cause the failures.

In this paper, we studied the functional programming language, Erlang [1, 12, 19]. This programming structure is small and it has no reassignment statements. Its local variables are assigned inside functions and never changed. These advantages of Erlang help the distance between faults and failures to be shorter than similar programs in C. Thus, we will propose a corresponding approach that can rapidly detect faulty code in C at compiled-time.

The remainder of this paper is organized as follows. The Literature Reviews is introduced in Section 2. Section 3 is Faults and Failures in C. Section 4 describes Solution for Fault Detection. The final Section is the Conclusion and Future Work of this paper.

2 LITERATURE REVIEWS

Currently, there are many approaches, models and tools for estimating and predicting software reliability. Software Reliability Engineering or SRE[8] is one well-known approach for estimation software reliability. The objective of this approach is the reliable behavior of software systems. The dynamic reliability estimation is one classification of the software reliability assessment. It determines the current software reliability by using statistical theory techniques to failure data obtained during software test or during software operation. These failure data occur when the software is executing. They are the resulting behavior when the software does not deliver the service expected by the user, or the program's behavior departs from the specification. They may mean the inability to perform an intended function specified by a requirement or the halting of the software program due to the incorrect code or data.

Nowadays, there is increasingly interest in the integrating previously existing software components for building the software system products. This approach is called Component-Based Software Engineering or CBSE.

Therefore, the reliability assessment of components that are integrated into system is very significant. Component Based Reliability Estimation (CBRE) [6] and Software component reliability analysis approach [3] are two approaches for the estimation of the software system reliability using reliabilities of its components. In addition, there are a lot of current methodologies to be used for developing the reliable software. The objective of these methodologies is to develop the fault-free software. One of these methodologies is Cleanroom software development [7] based on avoiding software faults. It avoids the costly fault-removal processes by writing code increments and verifying their correctness before the software is tested.

The structured programming languages, such as C, are widely used for developing the software products. The C language programs are relatively large. When a failure occurs, we have to take a long time to find out the causes by tracing faults in the collected log file. Currently, there are tools such as Purify and Valgrind that can detect an array index out of bound. Purify is a commercial package tool that can find memory errors in programs, but it is very expensive.[13] Valgrind is a tool for finding memory management problems in x86 GNU/Linux executables. Valgrind is licensed under the GNU General Public License.[16] Software running under the current tools runs much more slowly, making testing more time-consuming and tedious. Moreover, the existing tools are applied at run-time, in a reactive fashion, attempting to catch invalid accesses when they happen.

The functional programming languages are used for high reliability software. One of these programming languages, which have been studied and described in this paper, is Erlang. It generally has no reassignment statements and has small programs that are about 5-10 times shorter than equivalent programs in C.[20] An Erlang program consists of a set of functions which may be collected into modules [1] as shown in Figure 1. If the failure occurs in function B of Module I, the fault may be somewhere not far from it (in Function B or in some functions calling it). In addition, neither global variables nor pointers are used in Erlang. Moreover, local variables are assigned inside functions. These variables are never changed and so functions have no side-effects. All of these advantages help the distance between faults and failures to be shorter than similar programs in C.

3 FAULTS AND FAILURES in C

Software reliability is defined as the probability of failure-free operation of a computer program for a specified environment in a given time. A good software process should have the objective of developing fault-free software. The minimizing software faults have a significant impact on the number of system failures. Many

program failures and faults are often a consequence of human errors.

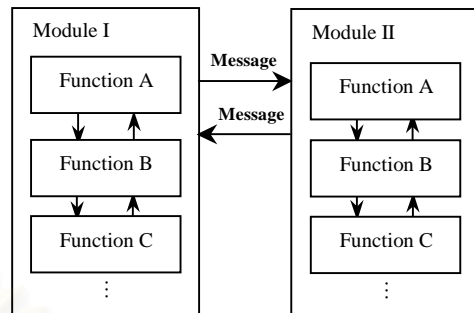


Figure 1 Structure of An Erlang Program.

Currently, the structured programming languages, such as C, are widely used for developing the software products. The fault-free software is very difficult to develop in these programming languages, especially C, because the constructs of these programming languages such as array indices and function argument passing by reference, often lead to software failures [19, 11, 15]. The paper focuses on three faults, which mostly occur in C programs.

3.1 Static Arrays

The C compiler does not have the checking of arrays indices whether they are out of bound [19]. One example about an array index out of bound is shown in Example 3.1

Example 3.1 Array index out of bound

```

...
{
    int arr_a[10], arr_b[3];
    ...
    i = 12;
    /* This is a fault since 'i' will be used as an index of an
    array, arr_a[i], and 12 is out of bounds) */
    arr_b[2] = 2;
    arr_a[i] = 0;
    ...
    x = 100/arr_b[2];
    ...
    /* The failure will occur since 100 is divided by arr_b[2] =
    0; */
}

```

3.2 Functions

In the old versions of C that do not support function prototypes, there is no checking of the types of arguments passed to functions [11]. However, in modern C, if a function is defined using a prototype, but is called in a separate file without a previous declaration in the current file, this causes the compiler to believe it is a non-

prototype for the function's arguments. Hence it performs no type checks on arguments passed to the function. This may cause the problem of arguments not matching the types of arguments or the wrong number of arguments passed as Example 3.2.

Example 3.2 Passing the wrong argument

```

...
main()
{
    ...
    char *a;
    int i;
    ...
    compute(a, i);
    /* This is a fault because the type of an argument 'a' does
    not match the declared argument in compute function
    (num1)*/
    ...
}
...
int compute(int num1, int num2)
{
    ...
    x = num1 + num2; /* The failure occurs */
    ...
}

```

3.3 Switch Statement

It is dangerous if there is no **default** label in switch case, or no **else** clause since execution then continues with the statement following the **switch** or **if** conditional statement. It is shown in Example 3.3 that is about the aircraft landing control system. The failure may occur if the emergency case happens on that aircraft and it cannot be specified the type. The unidentified aircraft may land on the runway that is not available. Therefore, it may have the **default** statement for resolving this problem.

Example 3.3 No Default Label in Switch case

```

...
Domestic = 1;
International = 2;
...
switch(type){
    case 1 :
        Runway_Free(type);
    case 2 :
        Runway_Free(type);
}
Landing();
...
/*This is dangerous if there is no default label in switch
case. The unidentified aircraft may land on the incorrect
runway.*/

```

4 SOLUTION FOR FAULT DETECTION

Generally, C language programs are relatively large. When a failure occurs, we have to take a long time to find out the causes by tracing faults in the collected log file. Furthermore, some faults cannot be checked such as array indices out of bound. Therefore, it is hard to detect these faults.

This proposed software will help to resolve the following problems.

1. The current tools are expensive and very hard to write.
2. Software running under the current tools runs much more slowly, making testing more time-consuming and tedious.
3. The existing tools are applied at run-time, in a reactive fashion, attempting to catch invalid accesses when they happen. The proposed tool would be applied at compile-time, in a pro-active fashion, with the intent to reduce the time it takes to debug the code by causing faults and failures to be more tightly coupled.

All advantages of the Erlang programming language which are described in Section 2 help the distance between faults and failures to be shorter than similar programs in C. Furthermore, it has the pattern matching construct [1, 12] which is one advantage because it assists in tightly coupling between faults and failures. If there is not one variable, clause of function, or message matching with their patterns, the fault message is immediately displayed to programmers. To make C programming language behave like the tightly coupling between faults and failures in Erlang, we have created software that work as software guards to detect the faults, which were mentioned in previous sections. This software will check C programs before they are compiled by C compiler as shown in Figure 2.

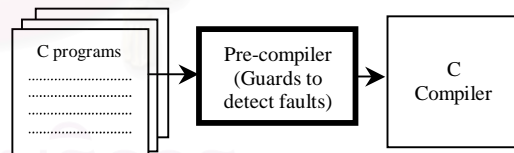


Figure 2 Pre-compiled program

In traditional C, there is no checking of the indices of array whether or not they are out of bound and no checking of the types of function's arguments. Therefore, the proposed software has two following main functions.

4.1 Fault Detection. In this part, it has three functions that automatically detect the coding errors in the programs. The first function is the array indices checking whether or not they are out of bound. The second is the **default** label checking in switch case. And the last function is the type of function's arguments checking.

4.2 Fault Recovery. If some faults occur, the error message will be displayed to warn the programmers.

Then, the proposed software automatically recovers some faults.

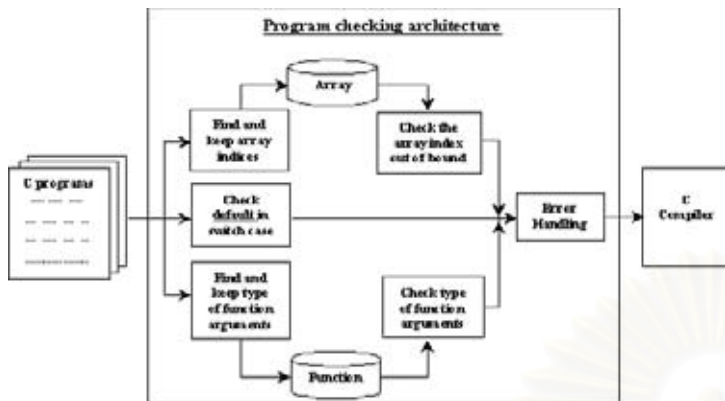


Figure 3 The pre-compiler architecture

5 CONCLUSIONS AND FUTURE WORK

This paper investigates the problem of fault detection in C programs. Since C language programs are relatively large, when a failure occurs we have to take a long time to find out the causes by tracing faults in the collected log file. And some faults cannot be checked such as array index out of bound. It is therefore hard to detect these faults. In this paper, like the tightly coupling between faults and failures in Erlang, we have created the software that work as a software guard for detecting faults, the faults that are the array indices out of bound and the incorrect type of function's argument, at pre-compiled time. The proposed software helps the faults to be detected easily and rapidly when the failures occur in C programs. In addition, the software which failures are reduced will be more reliable. The maximum time complexity of detection software's algorithms is $O(n \log n)$ where n is the number of characters in each C program.

The implementation of the proposed detection software have been created and operated at pre-compiled time. The examples illustrate a programming style for C based systems, that can be enforced through this pre-compiled programs, and raise the level of reliability towards that is achieved in Erlang implementations.

Another issue not addressed by this paper still has to be analyzed. In particular, the faults are detected by the fault detection software, how to manage these faults. Therefore, the error handling mechanism will be created in the near future.

ACKNOWLEDGEMENT

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

problems of programming languages

REFERENCES

- [1] Armstrong, J.L., Viriding, S.R. and Williams, M.C., Concurrent Programming in Erlang, Prentice Hall, 1993.
- [2] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, 2001.
- [3] Everett, W.W., "Software Component Reliability Analysis", ASSET'99, Proc. IEEE, pp. 204-211, 1999.
- [4] Fraser, C., Hanson, D., A Retargetable C Compiler: Design and Implementation, Addison-Wesley, 1995.
- [5] Jalote, P., Fault Tolerance in Distributed Systems, Prentice Hall, 1994.
- [6] Jiantao, P., Software Reliability, Dependable Embedded Systems, Carnegie Mellon University, Pittsburgh, USA, 1999.
- [7] Krishnamurthy, S., Mathur A., "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", Software Reliability Engineering Proc. IEEE, The Eighth International Symposium on, pp. 146-155, 1997.
- [8] Linger, R.C., "Cleanroom Process Model", IEEE Software, 11(2):50-58. 1994.
- [9] Lyu, M.R., Handbook of Software Reliability Engineering, McGraw-Hill, 1996.
- [10] Musa, J., Software Reliability Engineering, McGraw-Hill, 1999.
- [11] Harbison, S.P. and Steele Jr, G.L., C: A Reference Manual, Fourth Edition, Prentice-Hall, 1995.
- [12] Hausman, B., Turbo Erlang: Approaching the Speed of C, Implementations of Logic Programming Systems, 119-135, Kluwer Academic Publishers, 1994.
- [13] Pham, H., Software Reliability and Testing, IEEE Computer Society Press, 1995.
- [14] Rational the software development company, "Rational PurifyPlus for Unix", http://www.rational.com/media/products/ppc/D610C_PurifyPlus_UNIX.pdf, 2002.
- [15] Royce, T., C Programming, Macmillan Press, 1996.
- [16] Samani, M.M., Sloman, M., Monitoring Distributed Systems (A Survey), Imperial College Research Report, 1992.
- [17] Seward, J., "The design and implementation of Valgrind: detailed technical notes", <http://developer.kde.org/~sewardj/>, 2002.
- [18] Sommerville, I., Software Engineering, Sixth Edition, Addison-Wesley, 2001.
- [19] Spuler, D.A., C++ and C Debugging, Testing, and Reliability: The prevention, detection, and correction of program errors, Prentice-Hall, 1994.
- [20] Whatis.com, "Erlang programming language-a whatis definition", http://www.techtarget.com/definition/0,,sid9_gci212072,00.html

Vita

Name: Ms. Prattana Deeprasertkul.

Date of Birth: 8th December 1975.

Education:

- Ph.D. Program in Computer Science, Department of Mathematics, Faculty of Science, Chulalongkorn University, Thailand, (June 2000 – May 2005).
- Visiting Ph.D. researcher in SERC at Royal Melbourne Institute of Technology, Melbourne, Australia, (October 2001 – September 2002).
- M.Sc. in Computer Science, Department of Computer Engineering, Faculty of Engineer, Chulalongkorn University, Thailand, (June 1997 – May 2000).
- B.Sc. in Science, Department of Mathematics, Faculty of Science, Mahidol University, Thailand, (June 1993 – March 1997).

Publications:

- P. Deeprasertkul, P. Bhattarakosol, and F. O'Brien, "Automatic Detection and Correction of Programming Faults for Software Applications", *Elsevier: The Journal of Systems & Software*. 2005.
- P. Deeprasertkul and P. Bhattarakosol, "Software Fault Detection in C Programs", 12th International Conference on Intelligent and Adaptive Systems and Software Engineering, San Francisco, USA., June 9-11, pp. 192-195, 2003.

Scholarship and Awards:

Development and Promotion of Science and Technology Talents (DPST) Scholarship
of Thailand for B.Sc., M.Sc., and Ph.D.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย