Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp550-555)

A Static Model for Reverse Engineering of Software Threads and Their Interactions

[1] M M KODABAGI [2] B S ANAMI [3]G HEMANTHAKUMAR
[1&2] Department of Computer Science and Engineering
Basaveshwar Engineering College
Bagalkot, Karnataka
INDIA
[3] Department of Studies in Computer Science
University of Mysore, Mysore
INDIA

Abstract: - Reverse Engineering is a process of analyzing the subject system to identify its components, and relationships so as to represent the system at higher levels of abstractions to help developers understand the system for later maintenance and enhancement. Many software systems deployed in both military and industrial domains are very complex and comprise of multiple threads of control. Such systems are expensive and time consuming to build and must be evolved to meet new challenges. Hence, the challenge lies in discovering the information about the threads and their interactions for later maintenance and enhancement activities.
In this paper, we have proposed a static model that analyses C++ multithreaded LINUX source code, extracts information about threads, their interactions and record their understanding. The extracted information is used for pictorial presentation and program comprehension. The details obtained from multithreaded source code is helpful in clear understanding of architecture of threads and their interactions resulting into reduced effort in maintenance and enhancement of software.

Key-Words:- Reverse Engineering, Program Comprehension, Program Maintenance, Program Enhancement.

1 Introduction

Software engineering has undergone a paradigm shift as the sizes of the software systems deployed increased dramatically and businesses began to rely increasingly on computer systems and information systems. A substantial portion of the software development effort is spent on maintenance and enhancement of the existing systems rather than developing the newer systems [1]. It is estimated that 50% to 80% of the time and material involved in software development is devoted to maintenance of existing code [2]. Crucial to the maintenance of existing systems is the task of program comprehension, which is an emerging area of research in software engineering. Around 47% of the time is spent on enhancements to the existing programs and 62% of that is spent on program corrections, which involves program comprehension tasks like reading the documents, scanning the source codes, and understanding the changes required [3]. These tasks are achieved through reverse engineering.

Reverse Engineering is a methodology that greatly reduces the time, effort and complexity involved in solving the program comprehension problem [4][5]. The large multiprocess systems, often found in major aerospace systems are very costly to develop and must be designed for long useful lives. Such systems cannot be easily redeveloped; hence they must evolve to meet new challenges. It is the well-known fact that software evolution is extremely difficult even for complex single process systems when their developers knowledge is lost [6]. Hence, it is necessary to record the system understanding to help developers and beginners to understand new concurrent systems when they are asked to maintain and modify the existing systems. Some of the related works on static and dynamic reverse engineering are sited below.

The *Shimba* tool that automatically produces sequence diagrams of java programs is given in

[7]. With the *Shimba* tool trace information is acquired while such programs are executed. *Paradyn,* a parameter based performance prediction tool is given in [8]. *Paradyn* instruments the target system in order to find the portions of code, which use most of the resources, that help the programmer focus his attention on optimization.

The reverse engineering of software threads that uses traces of inter-pocess messages to recover functional software design threads from a large multiprocess system is given in [6]. A prototype approach to help extraction of architectural information in the re-engineering process is presented in [9]. A work on dynamically inferring program invariants is given in [10], which focuses on the value relationships among variables that are more relevant to dataflow. The work given in [11] presents two techniques one on static and other on dynamic, for inferring sequencing models of methods of a component, and built a dynamic model checker to check if the code conforms to the models discovered. The statistical techniques to discover patterns of concurrent behaviour for event traces is given in [12]. The techniques first extract a thread model out of the event traces, and then infer points of event traces, and then infer points of synchronization and mutual exclusion based on that model. A set of temporal property patterns developed based on a case study of hundreds of real time property specifications is given in [13]. The runtime analysis is used to pinpoint the problematic point in the program such that the state space for large program needs to be significantly pruned is given in [14].

The literature survey on program comprehension and design recovery is quite exhaustive. There seems to be almost no much work carried out that specifically addresses the problem of recording understanding of threads and their interactions to help developers to maintain and evolve the existing systems. The existing reverse engineering works on software threads use traces of inter-process messages to recover functional software design threads. Most of the works dealt with execution traces to discover patterns of interactions and temporal properties (sequence of event traces) during program dynamics for program evolution.

Our work focuses on development of a static model that helps programmer record the complete understanding of software threads and their interactions. The new as well as experienced developers can maintain and evolve the concurrent systems.

The rest of the paper is organized into three sections. Section 2 presents the proposed work. Results and discussions are provided in section 3. Finally, section 4 presents conclusions.

2 Proposed Model

The proposed model analyses multithreaded Linux based applications and recovers threads and their interaction details. The higher levels of abstractions like threads interaction diagrams are obtained. The proposed model consists of three phases namely, analysis, discovery and representation. The analysis phase scans multithreaded applications and prepares a token list, which is handled by the buffer manager. The discovery phase reads streams of tokens from the token list maintained by the buffer manager and recovers threads details like name of a thread, thread identification number, priority, attribute name, and policy and its interaction with other threads through mutex, semaphore and conditional variables. This recovered information is stored in tables, which acts as a data structure, which has the specific representation suitable for later diagrammatic presentation and program comprehension. The representation phase retrieves information from the tables and obtains higher levels of abstractions like threads inter-action diagrams and textual report of tables for documentation. The block schematic diagram of the proposed model is given in Figure 1.

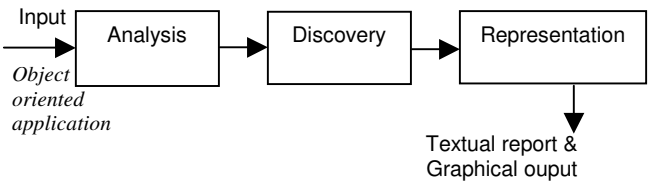

Fig. 1 Block diagram of the proposed model

2.1 Analysis Phase

The analysis phase is designed to perform two main tasks namely token recognition and buffer management. The token recognition uses a *buffer* divided into two portions, first and second halves of equal size. The two pointers *lexeme_beg* and *forward_ptr* are used to scan the buffer looking for the next token. Initially the first half of *buffer* is loaded with characters from the input file. This avoids referring to the input file for every character to be processed. Both the pointers point to the first character. The *forward_ptr* scans the buffer for the next token. Once the token is recognised, it is stored into the global array called *lexeme,* which is accessible to all other phases. The foreign tasks access the global array *lexeme* by using extern storage specifier or storage class. After recognising the token, both the pointers are set to a buffer position after the last token. If the *forward_ptr* moves beyond the first half of the buffer, then the second half of buffer is loaded. The process of scanning the buffer is repeated in Round Robin Fashion. The buffer manager encompasses a *list* to store all the tokens recognised and supplied by the token recogniser, and supports interfaces to manipulate the list. The design of *buffer* and its working procedure is given in Figure 2.

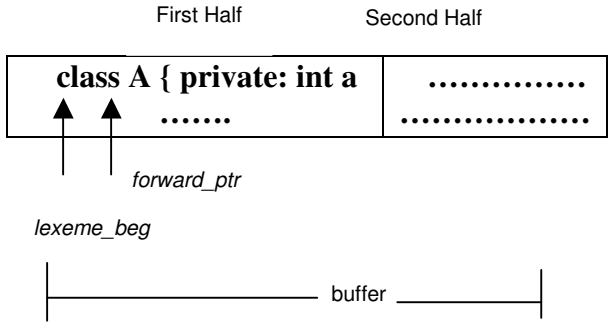

Fig. 2 Details of Buffer Management

2.2 Discovery Phase

The discovery phase comprises of algorithms to extract details of threads like name of thread, thread identification number, priority, attribute name, and policy and its interaction with other threads through mutex, semaphore and conditional variables. The discovery phase interacts with buffer manager to obtain stream of tokens for recovery details of threads. The recovered information is stored into tables namely, *threads information* and *threads relationship tables*. The advantage of this phase is its adaptation to other languages with little modification to these developed algorithms.

The methodology developed to extract threads details and their interactions from multithreaded applications are given in algorithm1 and algorithm 2.

Algorithm 1: Extraction of Thread Details

Input*: Token List maintained by buffer manager*
Output*: Thread Information Table*
begin
Step 1. Set a pointer to the *Token List*.
Step2. Read next token from *Token List.*
2.1 if (token == "pthread_create") **then begin**
- Extract the thread id, attribute name, and thread name.
- Store the information in Threads Information Table.
endif

2.2
if (token ==”pthread_attr_setschedpolicy’’) **then**
begin
- Extract the scheduling policy of thread.
- Store the information in Threads Information Table against the corresponding thread attribute.
endif

2.3 if (token == pthread_attr_setschedparam” or token == “setsched_priority”) **then begin**
- Extract the scheduling priority of thread.
- Store the information in Threads Information Table against the corresponding thread attribute.
endif

Step 3.repeat steps 1 to 3 until *token list is empty*
end // *Algorithm for Extraction of Thread Details*

Algorithm 2: Extraction of Thread Interaction Details

Input*: Token List maintained by buffer manager*
Output*: Thread Interaction Table*
begin
Step 1. Set a pointer to the *Token List*.
Step 2. Scan the *Token List* looking for the name of a function**.**
2.1 if (function name is the one of threads name listed in *Threads Information Table*) **then begin**
- Store name of thread into *Thread Interaction Table.*
//repeat
2.2 read next token.

2.3 if (token == "pthread_cond_signal") **then**
begin
- Extract the arguments.
- Store signaling conditional variable into *Thread Interaction Table* against the name of thread.
endif

2.4 if (token == "pthread_cond_wait") **then begin**
- Extract the arguments.
- Store waiting conditional variable into *Thread Interaction Table* against the name of thread.
endif

2.5 if (token == "sem_post") **then**
begin
- Extract the arguments.
- Store signaling semaphore variable into *Thread Interaction Table* against the name of thread.
endif

2.6 if (token == "sem_wait") **then**
begin
- Extract the arguments.
- Store waiting semaphore variable into *Thread Interaction Table* against the name of thread.
endif

2.7 if (token == "pthread_mutex_unlock") **then begin**
- Extract the arguments.
- Store signaling mutex variable into *Thread Interaction Table* against the name of thread.
endif
2.8 if (token == "pthread_mutex_lock") **then**
begin
- Extract the arguments.
- Store waiting mutex variable into *Thread Interaction Table* against the name of thread.
endif
2.9 repeat steps 2.2 to 2.8 until *end of function.*
Step 3. repeat steps 1 to 2 until *token list is empty.*
end // *Algorithm for Extraction of Thread Interaction Details*

2.3 Representation Phase

The representation phase is rendered language independent. This phase is responsible for retrieval of information from the tables, and higher levels of abstractions like threads interaction diagrams and textual reports for documenting the system details are obtained.

3 Results and Discussions

The proposed methodology is tested on large number of multithreaded applications of reasonable sizes (5000 to 10,000 lines of code). In addition to this the methodology is also subjected to rigorous unit and integration level tests with complex applications containing multiple threads of control and all possible synchronisation mechanisms like mutex, conditional, and semaphore variables under LINUX platform. The following example illustrates the working of the developed method with sample results.

3.1 Example

The sample code segment given in Figure 3 is considered illustrating the results of the proposed methodology. The sample code segment consists of two threads of control. The *producer1* and *consumer1* threads are synchronised with conditional variable *p1cond*.

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
int queue1[100],front1=0,rear1=0;
void *producer1(void *p);
void *consumer1(void *p);
pthread_mutex_t c1mutex;
pthread_cond_t p1cond;
void *producer1(void *p)
{       while(1) {queue1[rear1++]=10;
  pthread_cond_signal(&p1cond);}
}
void *consumer1(void *p)
{       while(1){
        pthread_cond_wait(&p1cond,&c1mutex)
        ;                   int item = queue1[front1];
                for(int I=0;I<rear1;++I)
                queue1[I] = queue1[I+1];
                --rear1;
        }}
main()
{
pthread_t tid1,tid2;
pthread_attr_t  attr1,attr2;
struct sched_param set1,set2;
set1.sched_priority=1;set2.sched_priority=2;
pthread_attr_init(&attr1);pthread_attr_init(&att
r2);
pthread_mutex_init(&lock,NULL);
pthread_attr_setschedpolicy(&attr1,SCHED_F
IFO);
pthread_attr_setschedpolicy(&attr2,SCHED_R
R);
pthread_attr_setschedparam(&attr1,&set1);
pthread_attr_setschedparam(&attr2,&set2);
pthread_create(&tid1,&attr1,producer1,NULL)
;
pthread_create(&tid2,&attr2,consumer1,NULL
);
return 0;
}//end main
```

Fig. 3 Sample multithreaded code segment

The proposed model generates two tables namely *Threads Information Table* and *Threads Interaction Table,* as a result of reverse engineering the sample multithreaded code segment given in Figure 3. The table1 and table2 contains details of all threads and their synchronisation information and helps to comprehend any multithreaded real time application. The developers use such information for maintenance and enhancement activities. Recovery and documentation of such information is an advantage of the proposed model, which helps to comprehend and reduce the effort required in maintenance and enhancement of complex applications. If such information is not available then developers needs to put effort to comprehend the system architecture by reading each line of source code. The model also generates threads interaction diagram as an alternative representation at higher levels of abstractions. The results of each phase of the proposed methodology are discussed in the following paragraphs.

The discovery phase extracts threads details and their interactions. The recovered information will be stored into *Threads Information Table* and *Threads Interaction Table.* The contents of *Threads Information Table* and *Threads Interaction Table* for the sample code segment are given in Tables 1 and 2 respectively.

Table 1: Text output of multithreaded code segment (Threads Information Table)

Thread Name	Identification	Attr Name	Policy	Priority
Producer1	Tid1	Attr1	SCHED_FIFO	1
Consumer1	Tid2	Attr2	SCHED_RR	2

Table 2: Threads Interaction Table

Thread Name	Signaling Variable	Waiting Variable
Producer1	P1cond	-
Consumer1	-	P1cond

The representation phase retrieves information from the tables and obtains higher levels of abstractions like threads interaction diagrams and textual report for documenting the system details. The textual report format is same as given in Table1 and Table2.

4 Conclusion

In this paper, we have proposed a static model for the analysis of multithreaded source code, information extraction about threads, their interactions and record their understanding. The recovery of complete system details and their documentation is an added advantage of this tool, which helps developers to comprehend, maintain and enhance complex multithreaded real time applications.

References

[1] Spencer Rugaber, "Program Comprehension for Reverse Engineering", http://www.cc.gatech.edu/reverse/papers.html, College of Computing Georgia Institute of Technology, March 1994.

[2] Barry W Boehm, *Software Engineering Economics,* Prentice Hall 1981.

[3] R. K. Fieldstad and W. T. Hamlen,"Application Program Maintenance Study: Report to our Respondents", *Proceedings Guide 48,* Philadelphia, PA, 1979. Tutorial on software maintenance, G Parokh and N Zvegintozov, editors, *IEEE Computer Society,* April 1983.

[4] Elliot J Chikofsky and James H Cross II, "Reverse Engineering and Design Recovery: A Texonomy", *IEEE Software,* vol 7, no 1, January 1990.

[5] Jinlin Yang, David Evans, "Automatically inferring temporal properties for program evolution", *15th IEEE International Smposium on Software Reliability Engineering (ISSRE 2004)"*, 2-5 Nov 2004, Saint, France.

[6] Joe Vandeville, Gary Trio, Dick Hotz, "Reverse Engineering of Software Threads: A Design Recovery Technique for Large Multi-Process Systems", SERC-TR-82-F, Software Engineering Research Center, Computer Science Department, Purdue University, West Lafayette, IN 47907, February, 1997.

[7] Tarja Systa, "Understanding the behaviour of Java Programs", *Proceedings of 7th Working Conference on Reverse Engineering,* pp.214, 2000.

[8] Miller, B. P. Callaghan, M. D. Cargille, J. M. Hollingsworth, J. K. Irvin, R. B. Karavanic, et al., "The paradyn Parallel Performance Measurement Tool", *IEEE Computer*, Novemeber 1995, pp.37-46.

[9] Sander Tichelaar, Stephane Ducasse, and Theo Dirk Meijler, "rchitectural Extraction in Reverse Engineering by Prototyping:an Experiment", www.iam.unibe.ch/~scg/Archive/papers/Tich97bArchExtraction.pdf.

[10] M. Ernst, J. Cockrell, W.Griswold, and D.Notkin, "Dynamically Discovering likely program invariants to support program evolution", *IEEE Transactions on Software Engineering,* February 2001.

[11] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic Extraction of object-oriented component interfaces", *International Symposium on Software Testing and Analysis",* July 2002.

[12] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf, "Discovering Models of Behaviour for Concurrent Workflows", *Computers in Industry,* pp.217-319, Vol.53, No.3, April 2004.

[13] M. Dwyer, G.Avrunin, and J. Corbett, "Patterns in property specifications for finite state verification", *21st International Conference on Software Engineering,* May 1999.

[14] K. Havelund, "Using runtime analysis to guide Model Checking of Java Programs", *7th International SPIN Workshop on Model Checking of Software",* August/September 2000.