A Process Algebra-Based Detection Model for Multithreaded Programs in Communication System

Tao Wang[1,2,3] , Limin Shen[1,3] and Chuan Ma[1,3]
[1] College of Information Science and Engineering, Yanshan University
Qinhuangdao, 066004 - China
[e-mail: tianyi_mc@126.com]
[2] Hebei Normal University of Science & Technology
Qinhuangdao, 066004 - China
[e-mail: yy_mma@126.com]
[3]The Key Laboratory for Computer Virtual Technology and System Integration, Hebei Province
Qinhuangdao, 066004, China
*Corresponding author: Tao Wang

Received October 31, 2013; revised February 7, 2014; accepted March 5, 2013; published March 31, 2014

Abstract

Concurrent behaviors of multithreaded programs cannot be described effectively by automata-based models. Thus, concurrent program intrusion attempts cannot be detected. To address this problem, we proposed the process algebra-based detection model for multithreaded programs (PADMP). We generate process expressions by static binary code analysis. We then add concurrency operators to process expressions and propose a model construction algorithm based on process algebra. We also present a definition of process equivalence and behavior detection rules. Experiments demonstrate that the proposed method can accurately detect errors in multithreaded programs and has linear space–time complexity. The proposed method provides effective support for concurrent behavior modeling and detection.

Keywords: intrusion detection, concurrent behavior, static analysis, process algebra, system call

This research was supported by the National Natural Science Foundation (61272125) of China, the specialized research fund for the doctoral program of Higher Education (20121333110014) and the Hebei Provincial Natural Science Foundation (F2011203234).

http://dx.doi.org/10.3837/tiis.2014.03.014

1. Introduction

Multithreading is an important mechanism for supporting program structuring and parallel computation. With the multithread technique, the application prospects of concurrent systems are becoming increasingly extensive. However, concurrent systems have specific characteristics, such as programming complexity, randomness of running results, and reproducibility. Therefore, security for concurrent systems is a concern, and constructing a model to describe and detect concurrent behaviors is an effective solution.

1.1 Related Work

Constructing a valid and precise program model is a challenging task. Because the original development of a model that takes advantage of the system call sequence for normal program behavior was originally presented by Forrest et al. [1], many scholars have researched software behavior using the system call. These studies are based on three basic techniques for model construction: system call short sequences [2-4], automata [5-8], and the Virtual Path [9]. Of these techniques, modeling based on system call short sequences is efficient and can be implemented easily. However, this method is imprecise, and these intrusion detection models are much more prone to false positives. Compared with short sequences, branch and loop structures of programs can be expressed. Modeling based on automata improves the precision of behavior modeling and reduces the false positive rate. Unfortunately, these models still have some limitations. For example, impossible paths, prohibitively high space–time complexity, and they are unsuitability for analyzing concurrent behaviors.

For concurrent behaviors, previous research has focused on two basic techniques: data race errors and timing analysis. Savage et al. discussed the potential data race problems based on the Lockset detection method when multiple threads access the same shared variables without locking [10]. Schonberg et al. analyzed data race problems that visited order uncertain, based on happen-before method [11]. Wang et al. presented a multilockset algorithm that considered the relation of happen-before and detected race condition at runtime [12]. Kong Deguang et al. presented a timing analysis method for multithreaded programs based on a hidden Markov model [13]. However, designing such models is complex because it is difficult to abstract a concurrent environment. Moreover, the space–time complexity is prohibitively high and unsuitable for practical use.

Modern software systems are prevalently concurrent; thus, they are difficult to get right. Unusual or unexpected behaviors in concurrent programs are difficult to discover using traditional detection techniques. Z. Rakamaric described a scalable, automatic, and precise approach to static unit checking of concurrent programs implemented in a tool called STORM [14]. To eliminate concurrency errors for a class of multithreaded programs, Berger et al. presented Grace, a software-only runtime system [15]. Tallent et al. described how to measure and attribute arbitrary performance metrics for high-level multithreaded programming models [16]. In addition, a technique to measure and analyze lock contention has been implemented [17]. To increase the reliability of multithreaded programs, a cooperative software–hardware mechanism to increase the performance of multithreaded applications was proposed, which was the first generalized mechanism to identify the most critical bottlenecks that cause thread waiting on multithreaded applications and accelerate those bottlenecks [18].

1.2 Main Contributions

We construct a process algebra-based detection model for multithreaded programs in a communication system. The basic idea is as follows: A system call is mapped to an action by static binary code analysis; a control flow graph (CFG) of the program is mapped to a process; process expressions are generated according to the process algebra algorithms; concurrency operators are added into process expressions; model construction algorithm and behavior detection rules are defined; and the *process algebra-based detection model for multithreaded programs* (PADMP) is used to detect concurrent behaviors. Our primary contributions can be summarized as follows:

1) The PADMP model enables efficient multithreaded program modeling. The PADMP model represents a substantial improvement in statically constructed multithreaded program models because it can describe concurrent behaviors of a multithreaded program effectively.

2) This method is also suitable for sequential behavior modeling and detection. To the best of our knowledge, we are the first to apply process algebra to behavior modeling, which is a profitable attempt in the field of multithreaded program behavior detection.

3) According to the properties of process algebra, some definitions and laws are given to provide a theoretical framework for concurrent behaviors. By reducing and merging process expressions, the PADMP model produces a smaller state space, moreover, it is complete.

The main advantage of the proposed PADMP model is that it can accurately detect errors in multithreaded programs, such as data race, deadlock, and abnormal time sequence errors. All test programs show an order of magnitude improvement in space–time complexity.

1.3 Organization of the Paper

The remainder of this paper is organized as follows. In Section 2, we introduce process algebra. In Section 3, the PADMP model construction algorithm is discussed. Section 4 presents the behavior detection rules. An experimental evaluation is discussed in Section 5, and we conclude the paper in Section 6.

2. Process Algebra

Process algebra is a mathematical tool used for depicting concurrent systems [19-20], and is used for researching concurrent, distributed, interactive systems [21]. At present, the Asynchronous Sequential Processes (ASP) [22-23] and Ambient Calculus [24-25] have more functions to describe the behavior of asynchronous concurrent system in theory research. This paper introduces process algebra for multithreaded programs modeling by extending its algorithms and describing the interaction of behaviors based on system calls. We extract a common subset of process algebra. Let Act be a finite set of given actions (A). The syntax specifications are defined as follows:

$$P ::= \underline{0} \mid \surd \mid a.P \mid P / L \mid P_1 + P_2 \mid P_1 \|_A P_2$$

Their corresponding meanings are as follows:

1) $\underline{0}$ stands for process down time, no action is performed. $\surd$ stands for process terminated successfully.

2) $a.P$ stands for executable action a, then transformed into process P, $a \in Act \cup \{\tau\}$; τ stands for unobservable action. Actions in this paper are the same as actions in CCS [26], divided into action (a) and co-action ($\bar{a}$), obviously $a = \bar{\bar{a}}$.

3) P/L stands for action (a) in P appearing in L will be hided and be replaced by unobservable action τ at runtime.

4) P_1+P_2 stands for the choice of P_1 or P_2, according to the process subordinated by the following actions.

5) $P_1 \|_A P_2$ means that if action (a) in P_1 and co-action ($\bar{a}$) in P_2 are subordinated to set A, then P_1 and P_2 execute synchronously, while any other actions are executed asynchronously.

Definition 1 Guarded Expression. The process expression begins with the prefix action. e.g., $P=a.Q$, $P=a.b.R$.

Definition 2 Successful Termination Guarded Expression. The process expression that begins with the prefix action and ends with a successful termination process. e.g., $P=a.b.\surd$.

Definition 3 Recursive Guarded Expression. The process expression that begins with the prefix action and ends with itself. e.g., $P=a.P$.

Definition 4 Behavior Trace. Suppose the process P can be defined as a finite state transition of the form: $P \equiv P_0 \xrightarrow{a_1} P_1 \ldots \xrightarrow{a_{n-1}} P_{n-1} \xrightarrow{a_n} P_n$

$<a_1, a_2, \ldots a_n>$ is the behavior trace of process P. The set of all possible behavior traces is denoted by $traces(P)$.

3. Model Construction Infrastructure

We have developed the PADMP model. The development procedure can be divided into four steps. First, static binary code analysis for multithreaded programs was employed to generate CFGs for each function. Second, the process expressions were generated from the CFGs. Third, the process expressions were rewritten by adding concurrent operations. Finally, the process algebra-based PADMP model was constructed.

3.1 From Binary Code to CFGs

We use static binary code analysis to generate a CFG because it does not require human interaction, determination of representative data sets, or access to a program's source code. However, it should be noted that it is unsuitable for interpreted-language analysis. The techniques to generate a CFG from binary code are very mature [6–7]. We use the executable editing library method to generate CFGs [6]. If the transformation of a flow chart does not contain any function call, it is regarded as an empty operation ε. We eliminate all edges ε using a previously reported reduction algorithm [27]. A corresponding example is given in Section 5.2.

We replace library functions with system calls by comparing the library function and system call tables. The call instruction in assembly code calls library functions rather than system calls; thus, we must replace it with the corresponding system call. For example, sleep was replaced by nanosleep and printf was replaced by write. We then capture and analyze arguments. This is required to represent the behavior of a system call accurately. For example, we cannot know if semop execution is a P or V operation.

3.2 From CFGs to Process Expressions

We denote the CFGs as $G=\{V,E\}$, which were generated according to the process discussed in Section 3.1. Here, V denotes vertices, and E denotes the directed edges that were marked with system calls. For example, **Fig. 1** shows one possible CFG generated from some binary code. We use the adjacency list to store it, shown in **Fig. 2**.

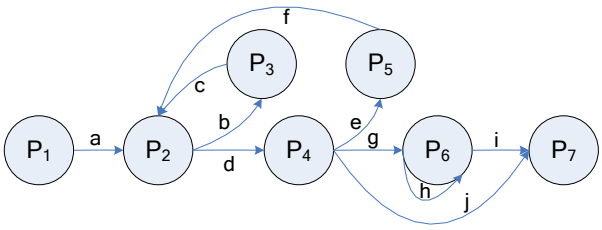

Fig. 1. A CFG generated from some binary code

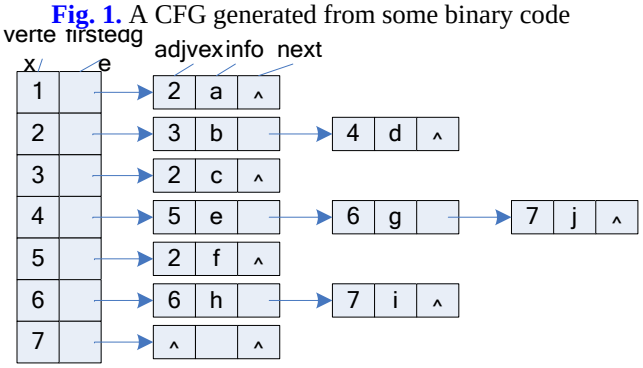

Fig. 2. The adjacency list of CFG in Fig.1

The algorithm that generates the process expressions from the CFGs is as follows.

(1) Find out the loop entry

According to sequential composition, if $\begin{cases}P_0=a.P_1\\P_1=b.P_2\end{cases}$, then expand $P_0=a.b.P_2$ and cut P_1. But if P_1 in a loop， e.g., $\begin{cases}P_0=a.P_1\\P_1=c.P_1\end{cases}$, then expand and obtain $\begin{cases}P_0=a.c.P_1\\P_1=c.P_1\end{cases}$. The equations do not be cut, thus they should not be expanded. In a loop, there is a node called the loop entry with the property that no other node in the loop has a predecessor outside the loop .That is, every path from the entry of the entire flow graph to any node in the loop goes through the loop entry. Therefore, we should find out the loop entry and don't expanded it.

By comparing the vertex and adjvex as shown in **Fig. 2**, if the number of vertexes is greater than or equal to adjvex, then the adjvex is the loop entry. The adjacency list of **Fig. 2** has three loops and has two loop entries (node 2($3\geqslant 2,5\geqslant 2$) and node 6($6\geqslant 6$)).

(2) Depth-first search to generate the process expressions

Selecting the loop entries and CFG entry (node 1, as shown in **Fig. 1**) as the root separately, we adopt depth-first search algorithm. We make the parent-child nodes sequential composition and make brother nodes alternative composition. To ensure that the loop entries do not be expanded, when we search a node that belongs to the loop entry, we backtrack to its parent node.

Fig. 1 has three nodes (1,2,6) as the root, so we get three process expressions:

P1=a.P2

P2=b.c.P2+d.(e.f.P2+g.P6+j.0)

P6=h.P6+i.0

3.3 Adding Concurrency Operators to Process Expressions

We obtain process expressions of functions according to the processes discussed in Sections 3.1 and 3.2. However, the expressions are not marked as an action (a) or a coaction ($\overline{a}$), which are used for concurrency. Thus, we must rewrite the process expressions.

We abstract mutual exclusion operations for a critical area as lock (l) and unlock (u). Concurrency operators are added to process expressions in the following situations:

(1) Situation 1. Multiple processes or threads: If the input edges are multiple processes or threads operation led by fork, vfork and clone, we should analyze jump sentences and change alternative composition of jump sentences to parallel composition('+'→'||A'), such as JLE and JNE.

(2) Situation 2. Mutual exclusion: If the input edges are mutual exclusion operation led by lock and unlock, we should add concurrency operators of exclusion operation to process expressions. Some coactions ($\overline{a}$) do not appear in expressions, such as critical sections and signal lamps. Thus, we create a process for each semaphore and make a parallel composition with the corresponding concurrent process.

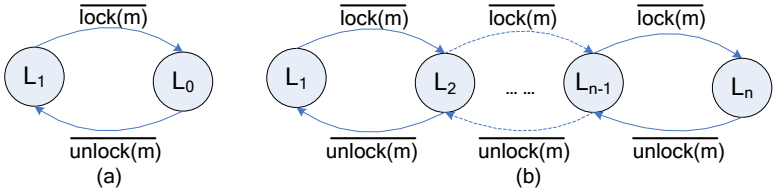

Fig. 3. State transition diagram for mutual exclusion

If the initial value of a binary semaphore is 1, its behaviors are described as $L_1 = \overline{lock}(m).\overline{unlock}(m).L_1$. Similarly, if the initial value of the binary semaphore is 0, its behaviors are described as $L_0 = \overline{unlock}(m).\overline{lock}(m).L_0$, as is shown in **Fig. 3**(a). Therefore, the initial value of a signal lamp is n (n>0) and can be described as $L_n = L_1 \|_A L_1 \|_A \ldots \|_A L_1$, as is shown in **Fig. 3**(b).

(3) Situation 3. Condition variable: If the input edges are condition variable operation led by lock, wait, signal and unlock, we should add concurrency operators of condition variable to process expressions. In concurrent programming, we construct a synchronization construct using a condition variable, which allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true, as shown in **Fig. 4.**

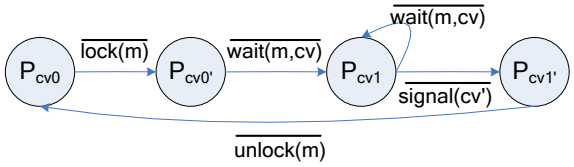

Fig. 4. State transition diagram for a condition variable

Its behaviors are described as follows.

$P_{cv0} = \overline{lock}(m).\overline{wait}(m,cv).P_{cv1}$

$P_{cv1} = \overline{wait}(m,cv).P_{cv1} + \overline{signal}(cv').\overline{unlock}(m).P_{cv0}$

(4) Situation 4. Read/Write lock: If the input edges are read/write lock operation led by rlock, wlock and unlock, we should add concurrency operators of read/write lock to process expressions. A read/write lock allows concurrent read access to an object; however, it requires exclusive access for write operations, as shown in **Fig. 5**.

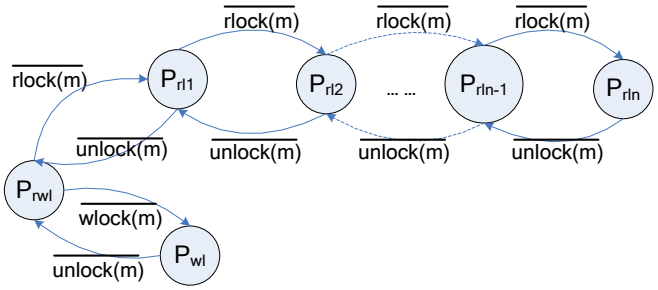

Fig. 5. State transition diagram for read/write lock

Its behaviors are described as follows.

$$P_{rwl} = P_{rl} \|_A P_{rl} + \overline{wlock(m)}.\overline{unlock(m)}.P_{rwl}$$

$$P_{rl} = \overline{rlock(m)}.\overline{unlock(m)}.P_{rl}$$

$P_{rl} \|_A P_{rl}$ denotes a read-lock. In this pattern, multiple readers can read the data in parallel. $\overline{wlock(m)}.\overline{unlock(m)}.P_{rwl}$ denotes a write-lock. When a writer is writing the data, readers will be blocked until the writer has finished writing.

3.4 Constructing the PADMP Model

Here, we present Definition 6 as an equivalence basis for the process expressions. Based on Definition 6, the laws of alternative composition and parallel composition are given. Then, we construct the PADMP model.

Definition 5 Action. A system call is an action.

In this paper, we map system calls to actions as the smallest unit to describe process behavior.

Definition 6 Process equivalence. If there are two different processes P Q ($traces(P) = traces(Q)$), i.e., $traces(P) \subseteq traces(Q)$ and $traces(Q) \subseteq traces(P)$, then P is equivalent to Q .

Our model is used to detect behaviors; thus, the process equivalence is based on the behavior trace. If two processes have the same behavior trace, they are considered equivalent. This also meets the requirements for behavior detection. However, this differs from equivalence based on mutual simulation of CCS [26]; therefore, the left distributive law of alternative composition states that ($a.P + a.Q = a.(P + Q)$). It also differs from equivalence based on refusal sets of CSP [28]. $\tau.P + \tau.\underline{0} = P + \underline{0} \neq P$ is different from $P + \underline{0} = P$ in CSP. As long as the stop sign $\underline{0}$ appears in the alternative composition, the process is considered to result in downtime and is therefore unsafe.

Here, we present some laws for alternative compositions and parallel compositions based on Definition 6.

Law 1 $P \|_A Q = Q \|_A P$.

Law 2 $(P \|_A Q) \|_A R = P \|_A (Q \|_A R)$.

Law 3
$$(P + Q) \|_A R = R \|_A (P + Q) = P \|_A R + Q \|_A R .$$

Law 4 $\underline{0}$ is a zero element and $\surd$ is an identity; i.e., $P \|_A \underline{0} = \underline{0}$; $P \|_A \surd = P$.

Law 5 If $a, \bar{a} \in A$, then $a.P \|_A \bar{a}.Q / a = \tau.(P \|_A Q) / a$.

Law 6 If $c,d \in A$ and $d \neq \bar{c}$, then $c.P \|_A d.Q = \underline{0}$.

According to Laws 5 and 6, we know the actions in A cannot execute independently. Thus, they must execute synchronously with corresponding coactions.

Law 7 If $a,\bar{a} \in A$, then $a.P \|_A a.Q \|_A \bar{a}.R / a = \tau.(P \|_A R) / a + \tau.(Q \|_A R) / a$.

Law 7 indicates that if processes $a.P$ and $a.Q$ compete for $\bar{a}.R$, they must be concurrent with $\bar{a}.R$.

Law 8 If $a \notin A$ and $c \in A$, then $a.P \|_A c.Q = a.(P \|_A c.Q)$.

Law 9 If $a,b \notin A$, then $a.P \|_A b.Q = a.(P \|_A b.Q) + b.(a.P \|_A Q)$.

Laws 8 and 9 indicate that actions without in the set A execute asynchronously.

Law 10 If $P = a.Q + a.R$, then $P = a.Q + a.R = a.(Q + R)$.

Law 10 transforms a nondeterministic alternative process into a deterministic process. Thus, we obtain Law 11.

Law 11 $P = P + P$.

Based on the above definitions and laws, we can construct actions, operators, and processes.

1) Action. We use a triplet to describe an action. $action ::= \{syscall, acttype, paramlist\}$. $syscall$ denotes system calls, $acttype$ denotes action type, including actions and coactions, and $paramlist$ denotes an argument list for $syscall$.

2) Operators and states. The basic expressions, $a.P$, $a.P_1 + b.P_2$, and $a.P_1 \|_A b.P_2$, are made up of sequential, alternative, and parallel composition operators. We use a structure with at most two outputs to describe the states composed of different operators. The initial values of the structure are $out1 = NULL$ and $out2 = NULL$; i.e., $state ::= \{opetype, action, out1, out2\}$. $opetype$ denotes operator type, and $out1$ and $out2$ are links pointing to the next state. $opetype = 0$ indicates that, for a successful termination state ($\surd$), $action = \surd$, and $out1$ and $out2$ are not used, as shown in **Fig. 6**(a). $opetype = 1$ indicates that sequential composition operator (.) uses only $out1$, which describes the state shown in **Fig. 6**(b). $opetype = 2$ indicates alternative composition operators, as is shown in **Fig. 6**(c).

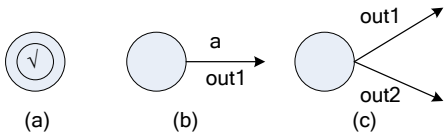

Fig. 6. Graphs of stuct state when opetype has different values

3) Process. A process is composed of actions and operators. We use two tuples to describe a process. i.e., $process ::= \{statelist, outlist\}$. $statelist$ denotes the set of states. If the $state$ in $statelist$ is greater than one, then suggest processes in the concurrent state. $outlist$ is a series of link lists pointing to the states.

4) Construction algorithm

According to the symmetric law, associative law, and distributive law of choice operators, any process that does not contain concurrency operators can be constructed by action a, successful termination $\surd$, $a.b$, and $a + b$. In this paper, we refer to a, $\surd$, $a.b$, and $a + b$ as meta process expressions.

Here, we describe the parallel composition operators (||A). Two concurrent processes $P_1 \|_A P_2$, $A=\{a,\bar{a},c,\bar{c}\}$ and $P_1 = a.b.\bar{c}.P_1^{'}$, $P_2 = \bar{a}.c.d.P_2^{'}$ are illustrated in **Fig. 7**.

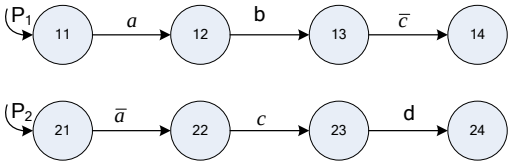

Fig. 7. Concurrent processes

According to the laws, we know the execution process of $P_1 \|_A P_2$. Initially, $P_1 \|_A P_2$ is in two states (11, 21) simultaneously; then, execute $a,\bar{a}$ simultaneously to obtain states (12, 22) simultaneously. Next, to obtain states (13, 22), c must wait for $\bar{c}$ to appear to execute simultaneously. When $\bar{c}$ appears, $P_1 \|_A P_2$ can obtain states (14, 23) simultaneously. Therefore, we describe parallel composition as a combination of multiple process states.

We can describe any process expression using meta process expressions a, $\surd$, $a.b$, and $a+b$. The PADMP model construction algorithm is constructed by meta process expressions.

PADMP Algorithm:

```
1: Input: PEs       /* process expressions list*/
2: Output: PADMP
3: Procedure:
4: PE *pe;  /*process expression */
5: process    p1, p2; /*process*/
6: state *s;    /*state*/
7: while(i<PEs→peNum) {/* Iterate through process expressions*/
8:    pe=PEs[i++]; /*get a process expression*/
9: for(; *pe; pe++){/* construct model for process expression */
10:    switch(*pe){
11:     default: /* construct action*/
12:              s = state(1，  *pe, NULL, NULL);
13:              push(process(s, outlist(s->out));
14:              break;
15:     case '.':/*process sequential operator (.)*/
16:              p2 = pop();p1 = pop();
17:              patch(p1.out, p2.start);/*make the output link of p1 point to p2*/
18:              push(process(p1.start, p2.out));/*push new process into stack*/
19:              break;
20:     case '+': /* process choice operator (+)*/
21:              p2 = pop();p1 = pop();
22:              s = state(2,NULL, p1.start, p2.start);
23:               push(process(s, append(p1.out, p2.out))); /*use append to connect two
pointers and return the result*/
24:            break;
25:     case '||A': /* process parallel operator (||A)*/
26:              p2 = pop();p1 = pop();
27:            push(process(join(p1.statelist,p2.statelist),append(p1.out, p2.out))); /* use join
to merge satatelist*/
28:            break;
```

```
29:     case ' √ ': /*process successful termination ( √ )*/
30:             s = state(0， √ , NULL, NULL);
31:             push(process(s, NULL);
32:             break;}}
33:     }
```

The action procedure is shown in **Fig. 8**(a), the sequential composition operator (.) procedure is shown in **Fig. 8**(b), and the alternative composition operator (+) procedure is shown in **Fig. 8**(c). According to ($\tau.P+\tau.Q=P+Q$), **Fig. 8**(c) is equivalent to **Fig. 8**(d). Therefore, the alternative composition operator (+) procedure is correct. The parallel composition operator ($\|_A$) procedure is shown in **Fig. 7**.

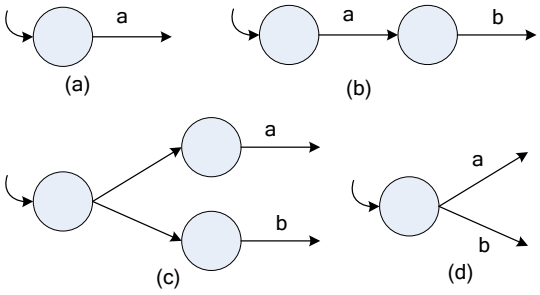

Fig. 8. Results of meta process expression compilation

4. Behaviors Detection

According to the operational semantics of process algebra (the above definitions and laws), we can obtain a process migration rule and a behavior decision rule, and obtain a behavior detection algorithm based on the PADMP model.

Process Migration Rule:

(1) $c.Q+d.R\xrightarrow{c}Q$， $c.Q+d.R\xrightarrow{d}R$；

According to (1) and Law 8, we know that, if $a\notin A$ and $c\in A$, then $a.P\|_A c.Q\xrightarrow{a}P\|_A c.Q$.

According to Law 9, we know that, if $a,b\notin A$, then $\begin{matrix}a.P\|_A b.Q\xrightarrow{a}P\|_A b.Q\\ a.P\|_A b.Q\xrightarrow{b}a.P\|_A Q\end{matrix}$.

(2) According to Law 10, we obtain $a.Q+a.R\xrightarrow{a}(Q+R)$.

(3) $a.P\xrightarrow{b}\underline{0}$， if $a\neq b$

Let s_0 denote the first action of behavior trace s; s' denotes the other actions; i.e., $s=<s_0,<s'>>$. P/s denotes a process, which is behavior after all trace s actions have been executed. Therefore, $P/s=(P/<s_0>)/<s'>$.

Behavior Decision Rule:

Suppose model P was obtained by static analysis, and model R was obtained by detection at runtime. If and only if $traces(R)\subseteq traces(P)$, then R is a normal behavior.

Behavior Detection Algorithm:

1) Monitoring the system calls of a program in real time forms a system call queue.

2) If the queue is null, then return TRUE to indicate that behaviors are normal. Otherwise, remove a system call from the head of the queue and place it into the PADMP model to

determine a match. If there is a match, loop (2); else proceed to (3).

3) Determine if actions in the current state set of the PADMP model belong to synchronous set (A). If they do not belong to A, return FALSE and issue an alert. If they belong to A, perform a breadth-first search of the current state set and execute a synchronous action. If the execution is successful, update the current state set and proceed to (4). Otherwise, return FALSE and issue an alert.

4) Match actions in the update state set. If the match is successful, proceed to (2); otherwise proceed to (3).

5. Simulation Results and Analysis

Here, we report experimental results for the PADMP model and behavior detection rules.

5.1 Behavior Detection

Fig. 9 is sample C code for a multithreaded program. According to the explanation presented in Section 3.1, we obtain the corresponding function CFG based on the Linux IA32 operating system, shown in **Fig. 10**. We only map a base block that contains a RET instruction into the successful termination process √. Next, we rewrite the assembly code.

```
void* Thread1(void* arg) {
pthread_mutex_lock(&lock);
a -= 50;
sleep(5);
b += 50;
pthread_mutex_unlock(&lock);
}
void* Thread2(void* arg) {
sleep(1);
pthread_mutex_lock(&lock);
printf("%d\n", a + b);
pthread_mutex_unlock(&lock);
}
```

```
int main() {
pthread_t tida, tidb;
pthread_mutex_init(&lock, NULL);
pthread_create(&tida, NULL, Thread1, NULL);
pthread_create(&tidb, NULL, Thread2, NULL);
pthread_join(tida, NULL);
pthread_join(tidb, NULL);
return 1;
}
```

Fig. 9. C code for a multithreaded program

We analyze the assembly code, obtain the corresponding system call, and extract the arguments. For simplicity, we rename the system calls that are in boldface in **Fig. 10**. For example, *pthread_mutex_lock* is renamed *futex* and *pthread_mutex_unlock* is renamed *lock(m)* or *unlock(m)* where m denotes mutual exclusion access to the critical area address. In addition, *clone*, which is called by *pthread_create*, is renamed *create*. Next, according to the algorithm presented in Section 3.2, we can obtain the process expression of the functions presented in **Fig. 10**.

$$P_{main} = init(m).P_1$$

$$P_1 = create(1).P_{thread1} \|_A create(2).P_{thread2}$$

$$P_{thread1} = lock(m).sleep(5).unlock(m).exit(1).join(1).\surd$$

$$P_{thread2} = sleep(1).lock(m).write.unlock(m).exit(2).join(2).\surd$$

According to the method presented in Section 3.3, process P_1 of a main function can be rewritten to add concurrency operators to process expressions for mutual exclusion.

$$L = \overline{lock(m)}.\overline{unlock(m)}.L$$

$$P_1 = create(1).P_{thread1} \|_A create(2).P_{thread2} \|_A L$$

$$A = \{lock(m), unlock(m), \overline{lock(m)}, \overline{unlock(m)}\}$$

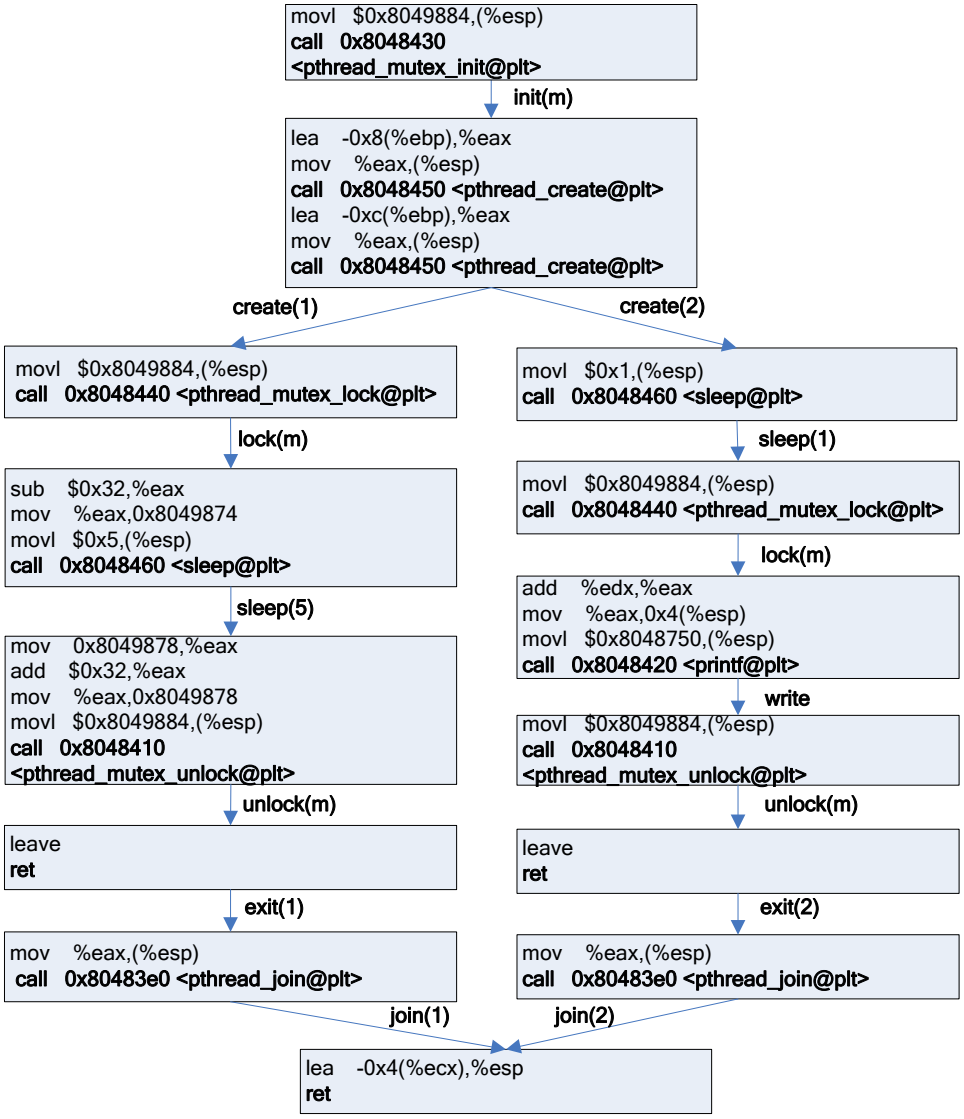

Fig. 10. Corresponding function CFG

According to Law 7 presented in Section 3.4, the process expression P_{main} satisfies the sequential relationship, as shown in **Fig. 11**. Detection processing is obtained by the behavior detection algorithm. Suppose that we capture the following call sequence at runtime:

(init(m), create(1), create(2), lock(m), sleep(1), sleep(5), unlock(m), exit(1), lock(m), write, unlock(m), exit(2), join(1), join(2))

According to Laws 5, 8, and 9, and the process migration rule, the final sequence can migrate successfully to the termination process. This indicates that it is a normal sequence. Suppose we capture the following call sequence:

(init(m), create(1), create(2), lock(m), sleep(1), sleep(5), lock(m), write, unlock(m), exit(2), unlock(m), exit(1), join(1), join(2))

If substituting the second lock(m) for the process expression P_{main} causes downtime $\underline{0}$,

then the sequence is an abnormal sequence and an alert will be issued. We can test and verify that the sequences acting against program timing cause downtime.

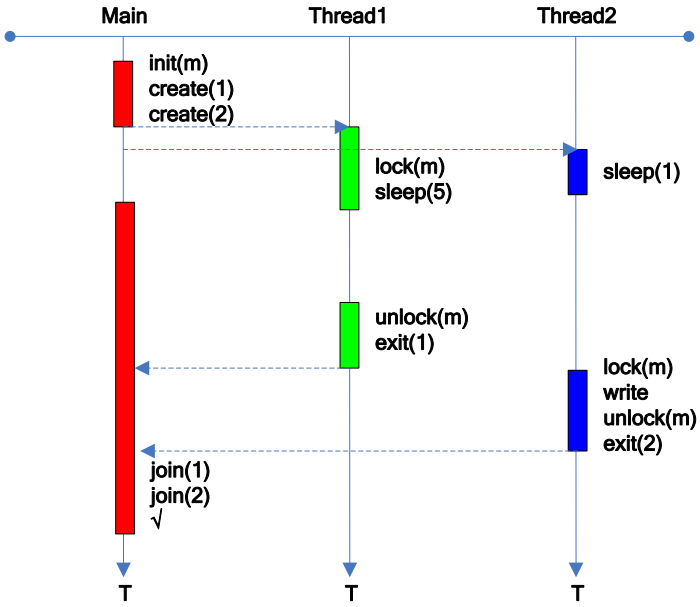

Fig. 11. Sequence diagram for C code

5.2 Deadlock Analysis

Fig. 12 illustrates a deadlock error caused by cyclic lock acquisition. This example spawns two threads, each of which attempt to acquire two locks A and B. However, the threads attempt to obtain the locks in different orders: thread 1 acquires lock A then lock B, while thread 2 acquires lock B then lock A. By following the behavior detection steps presented in Section 5.1, the following process expressions can be obtained.

$$P = P_{thread1} \|_A P_{thread2} \|_A L$$

$$P_{thread1} = lock(a).systemcall1.lock(b).systemcall2.unlock(b).unlock(a).P_{thread1}$$

$$P_{thread2} = lock(b).systemcall3.lock(a).systemcall4.unlock(a).unlock(b).P_{thread2}$$

```
thread1 () {                    thread2 () {
    lock (a);                       lock (b);
    systemcall1();                  systemcall3();
    lock (b);                       lock (a);
    systemcall2();                  systemcall4();
    unlock (b);                     unlock (a);
    unlock (a);                     unlock (b);
}                               }
```

Fig. 12. Deadlock example

We can also obtain process expressions for the mutual exclusion lock and synchronous action set.

$$L = L_a \|_A L_b = (\overline{lock(a)}.\overline{unlock(a)}.L_a) \|_A (\overline{lock(b)}.\overline{unlock(b)}.L_b)$$

$$A=\{lock(a),lock(b),unlock(a),unlock(b),\overline{lock(a)},\overline{lock(b)},\overline{unlock(a)},\overline{unlock(b)}\}$$

According to Laws 5 and 6, the following calculations can be performed.

$$\begin{aligned}
P &= \tau.systemcall1.lock(b).systemcall2.unlock(b).unlock(a).P_{thread1} \|_A \\
&\quad \tau.systemcall3.lock(a).systemcall4.unlock(a).unlock(b).P_{thread2} \|_A \\
&\quad (\overline{unlock(a)}.L_a) \|_A (\overline{unlock(b)}.L_b) / (lock(a),lock(b)) \\
&= systemcall1.systemcall3.(lock(b).systemcall2.unlock(b).unlock(a).P_{thread1} \|_A \\
&\quad lock(a).systemcall4.unlock(a).unlock(b).P_{thread2} \|_A (\overline{unlock(a)}.L_a) \|_A (\overline{unlock(b)}.L_b) \\
&\quad + systemcall3.systemcall1.(lock(b).systemcall2.unlock(b).unlock(a).P_{thread1} \|_A \\
&\quad lock(a).systemcall4.unlock(a).unlock(b).P_{thread2} \|_A (\overline{unlock(a)}.L_a) \|_A (\overline{unlock(b)}.L_b) \\
&= \underline{0}
\end{aligned}$$

Thus, it can be seen that program deadlock errors can be detected by the proposed PADMP model.

5.3 Efficiency Analysis

Here, we evaluate a parallel composition that contains two processes.

P=a.b. √ ||$_A$c.d. √
=a.(b. √ ||$_A$c.d. √)+c.(a.b. √ ||$_A$d. √)
=a.b.(√ ||$_A$c.d. √)+a.c.(b. √ ||$_A$d. √)+c.a.(b. √ ||$_A$d. √)+c.d.(a.b. √ ||$_A$ √)
=a.b.c.d. √ +a.c.b.(√ ||$_A$d. √)+a.c.d.(b. √ ||$_A$ √)+c.a.b.(√ ||$_A$d. √)+c.a.d.(b. √ ||$_A$ √)+c.d.a.b. √
=a.b.c.d. √ +a.c.b.d. √ +a.c.d.b. √ +c.a.b.d. √ +c.a.d.b. √ +c.d.a.b. √

From the above formula, if a parallel composition contains two processes (n=2), then six behavior traces are obtained. With increasing n, the number of behavior traces increases by order of magnitude. Adopting a training method will result in a state space explosion, which leads to low space–time efficiency. However, in this paper, space consumption is the cost necessary to maintain n state list; the scale is equal to the number of actions, i.e., linear growth with n. The PADMP model uses a state set to express concurrent and nondeterministic options without a backtrack algorithm. If the length of a process expression is m and an n status list must be maintained, then, total time cost is O (mn). Thus, running time is linear; increasing m will not increase the operational cost significantly.

Experimental environment: Linux rhel-server-5.4, Intel dual-core 1.73 GHz CPU, 2 GB RAM. Three test programs are listed in **Table 1.**

Table 1. Test Programs

Program	Action num	Concurrent-action num	User thread
Main	12	4	3
Philosopher	72	48	5
Reader-writer	266	34	n

We construct PADMP models for the test programs and record space–time consumption. By modifying the source code, we simulate data race, deadlock, and synchronous abnormality scenarios that were caused by design errors or intrusion. We use the PADMP model to detect the scenarios and record the space–time consumption. All abnormalities are detected. The time consumption for modeling and detection is shown in **Fig. 13**, and the space consumption is shown in **Fig. 14**.

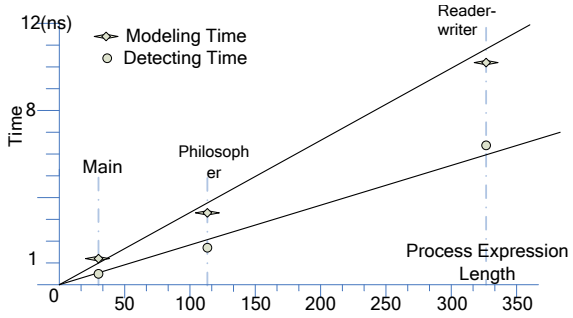

Fig. 13. Time Consumption

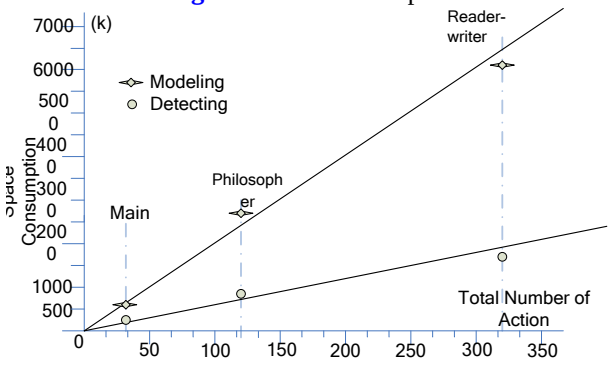

Fig. 14. Space Consumption

5.4 Experimental Design

These test programs and our runtime monitor run on Linux OS (rhel-server-5.4) on Intel 1.73Ghz cpu with 2GB of RAM. Behavior detection experiment program is made up of static analysis unit, modeling unit and detection unit. Static analysis unit is a tool program running in user-state, it reads the executable file and generates the corresponding process expressions files *.pe. According to laws presented in Section 3.4, modeling unit converts files *.pe to files *padmp using meta process expressions a, $\surd$, $a.b$, $a+b$. File *.padmp stores chain table structures of the corresponding process expressions. Detection unit uses hook technology to intercept information, which is the entry address of system calls service function in the sys_call_table with real-time software running, monitors and obtains system call sequences. By putting system call sequences into the file *.padmp, detection unit detects and reports the result, as is shown in **Fig. 15**.

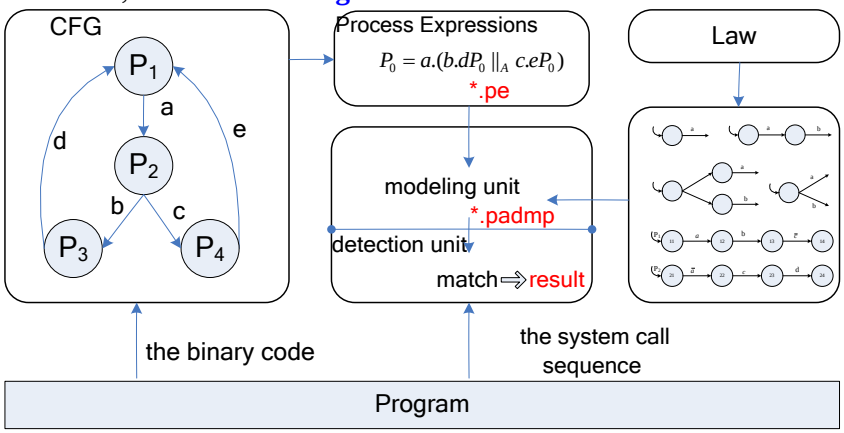

Fig. 15. Architecture of the model

We construct the models for some applications, such as apache, wu-ftpd, mysql and sqlite. **Fig. 16** and **Fig. 17** show the result of space and time overhead respectively.

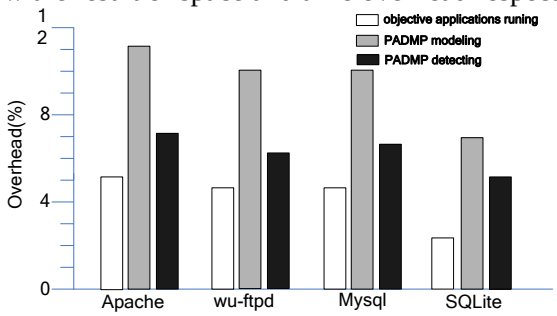

Fig. 16. Result of space overhead

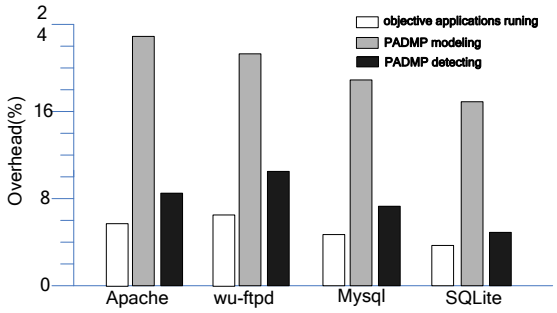

Fig. 17. Result of time overhead

5.4.1 Effectiveness of the PADMP Model

As can be seen from experimental results, the time and space overheads of PADMP model are acceptable. During modeling the overheads are high, but the PADMP model can perform on real applications well during detection.

5.4.2 Precision of the PADMP Model

Precision is behavior detection capability of model. We choose FTP server wu-ftpd-2.6.0, which is widely used on the Linux platform as the test objects. Wu-ftpd-2.6.0 exists multiple vulnerabilities, we select some representative vulnerabilities, and use existing programs to attack, detection results are shown in **Table. 2**.

Table 2. Typical vulnerabilities of wu-ftpd-2.6.0 and test results

Vulnerability CVE	Vulnerability type	Test result
CVE-2000-0573	Format string overflow	√
CVE-2001-0550	Heap overflow	√
CVE-2004-0185	Stack overflow	√
CVE-2004-0148	Logic error	×

The experimental results show that the prototype system can detect attacks based on stack overflow, heap overflow and the format string overflow. The attack doesn't change the call sequence for CVE - 2004-2004 vulnerabilities, it is similar to the mimicry attack, not being detected. We can't test all types of attacks, but the results in **Table. 2** partly prove the ability of intrusion detection system.

6. Conclusions

In this paper, the problem of intrusion attempts on multithreaded programs is investigated. Based on process algebra, the PADMP model is presented to solve the concurrent behavior description and detection problem. Experimental results show that this method can accurately detect errors in multithreaded programs, such as data race, deadlock, and abnormal time sequence errors. Moreover, all test programs show an order of magnitude improvement in space–time complexity. However, the PADMP model doesn't consider the parameters of the system calls, so it can't effectively detect attacks based on data flow. We will solve this problem next step.

References

[1] S. Forrest, S.A. Hofmeyr, A. Somayaji and T.A. Longstaff, "A sense of self for UNIX processes," in *Proc.of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press*, pp. 120-128, May 6-8, 1996. Article (CrossRef Link)

[2] S.A. Hofmeyr, S. Forrest and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151-180, January, 1998. Article (CrossRef Link)

[3] P. Helman and J.Bhangoo, "A statistically based system for prioritizing information exploration under uncertainty," *IEEE Trans.on Systems,Man and Cybernetics, Part A:Systems and Humans*, vol. 27, no. 4, pp. 449-466, July, 1997. Article (CrossRef Link)

[4] Wenke Lee and Salvatore J. Stolfo, "Data mining approaches for intrusion detection," in *Proc. of the 7th USENIX Security Symp. San Antonio*, pp. 26-29, January, 1998. Article (CrossRef Link)

[5] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. of the IEEE Symp.on Security and Privacy.Oakland:IEEE Press*, pp. 156-168, May 14-16, 2001. Article (CrossRef Link)

[6] J. Giffin, S. Jha and B. Miller, "Efficient context- sensitive intrusion detection," in *Proc. of the 11th Network and Distributed System Security Symp. San Diego*, 2004. Article (CrossRef Link)

[7] R. Gopalakrishna, E.H. Spafford and J. Vitek, "Efficient intrusion detection using automaton Inlining," in *Proc. of the IEEE Symp.on Security and Privacy. Oakland, CA, IEEE Press*, pp. 18-31, May 8-11, 2005. Article (CrossRef Link)

[8] FU Jianming, TAO Fen and WANG Dan, "Software behavior model based on system objects," *Journal of Software*, vol. 22, no. 11, pp. 2716-2728, November, 2011. Article (CrossRef Link)

[9] H.H. Feng, J.T. Giffin, Y. Huang and S. Jha, "Formalizing sensitivity in static analysis for intrusion detection," in *Proc. of the IEEE Symp.on Security and Privacy. Oakland, CA, IEEE Press*, pp. 194-208. May 9-12, 2004. Article (CrossRef Link)

[10] S. Savage, M. Burrows, G. Nelson and P. sobalvarro, "Eraser: A dynamic data race detector for multi-threaded programs," *ACM Trans. on Computer Systems*, vol. 15, no. 4, pp. 391-411, November, 1997. Article (CrossRef Link)

[11] D. Schonberg, "On-the-Fly detection of access anomalies," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM Press*, vol. 24, no. 7, pp. 285-297, July, 1989. Article (CrossRef Link)

[12] L.Q. Wang and S.D. Stoller, "Runtime analysis of atomicity for multi-threaded programs," *IEEE Trans. on Software Engineering*, vol. 32, no. 2, pp. 93-110, February, 2006. Article (CrossRef Link)

[13] K. Deguang, Tan XB and Xi HS, "Hidden Markov Model for Multi-Thread Programs Time Sequence Analysis," *Journal of Software* , vol. 21, no. 3, pp. 461-472, March, 2010. Article (CrossRef Link)

[14] Z. Rakamaric, "STORM: static unit checking of concurrent programs," *In Proc. of the 32nd*

ACM/IEEE International Conference on Software Engineerin, Cape Town, South Africa , vol. 2, pp. 519-520, May 2-8, 2010.Article (CrossRef Link)

[15] E. D. Berger, Ting Yang, Tongping Liu and Gene Novark, "Grace: safe multithreaded programming for C/C++," in *Proc. of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, vol. 44, no. 10, pp. 81-96, October, 2009. Article (CrossRef Link)

[16] N. R. Tallent and J. M. Mellor-Crummey, "Effective Performance Measurement and Analysis of Multithreaded Applications," in *Proc. of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, vol. 44, no. 4, pp. 229-240, April, 2009 Article (CrossRef Link)

[17] N. R. Tallent, J. M. Mellor-Crummey and A. Porterfield, "Analyzing Lock Contention in Multithreaded Applications," in *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 45, no. 5, pp. 269-280, May, 2010. Article (CrossRef Link)

[18] J. A. Joao, M. A. Suleman, O. Mutlu and Y. N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *Proc. of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, vol. 40, no. 1, pp. 223-234, March, 2012. Article (CrossRef Link)

[19] J.H. Morris, "Lambda-calculus Models of Programming Languages," *MIT, Cambridge, MAC*, U SA, 1968. Article (CrossRef Link)

[20] H. Bekic, "Towards a mathematical theory of processes," *IBM Laboratory*, Vienna: Technical Report TR, 1971. Article (CrossRef Link)

[21] G.J. Milne and R. Milner, "Concurrent processes and their syntax," *Journal of the ACM*, vol. 26, no. 2, pp. 302-321, April, 1979. Article (CrossRef Link)

[22] D. Caromel and L.A. Henrio, "Theory of Distributed Objects," *Berlin:Springer-Verlag*, 2005. Article (CrossRef Link)

[23] D. Caromel, L. Henrio and B.P. Serpette, "Asynchronous sequential processes," *Information and Computation*, vol. 207, no. 4, pp. 459-495, April, 2009. Article (CrossRef Link)

[24] L. Cardelli and Gordon A.D, "Mobile Ambients," *Theoretical Computer Science*, vol. 240, no. 1, pp. 177-213, June, 2000. Article (CrossRef Link)

[25] L. Cardelli, G. Ghelli and A.D. Gordon, "Types for the ambient calculus," *Types for the Ambient Calculus*, vol. 177, no. 2, pp. 160-194, September, 2002. Article (CrossRef Link)

[26] R. Milner, "A calculus of communicating systems," *Lecture Notes in Computer Science*, Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1980. Article (CrossRef Link)

[27] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton," *Theory of Machines and Computations*, New York: Academic Press, January, 1971. Article (CrossRef Link)

[28] C. Hoare, "Communicating sequential processes," *Communications of the ACM* , vol. 21, no. 8, pp. 666-677, August, 1978. Article (CrossRef Link)

Tao Wang is currently having her Ph.D study in Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, China. She received her M.S. degree in the School of Information Science and Engineering, Yanshan University, Qinhuangdao, China in 2009. She is now working at Hebei Normal University of Science & Technology as a lecturer. Her current research interests are in the areas of intrusion detection and collaboration computing.

Limin Shen is currently a professor in the School of Information Science and Engineering, Yanshan University. He received his M.S. degree in computer applicationform, Hefei University of Technology, China, in 1987. He received his Ph.D degree in electronic circuit and system, Yanshan University, China, in 2005. He worked in Department of Computer Science, Illinois Institute of Technology, USA from 2005 to 2007 as a visiting scholar. His main research interests are focusing on flexible software technology and information security, which has been funded partially by the National Natural Science Foundation of China and Chinese Government.

Chuan Ma is currently having his Ph.D study in Information Science and Engineering, Yanshan University. He received his B.S. and M.S. degrees in the School of Information Science and Engineering, Yanshan University, Qinhuangdao, China in 2003 and 2009, respectively. He is an engineer with the School of Information Science and Engineering Yanshan University. His current research interests include information security and software formal methods.