Dynamic model checking for concurrent programs in control system

Hao Liang[1]*, Yunfeng Ai[2], Huairong Shen[3], Yongchao Zhao[4]

[1]*Company of Postgraduate Management, the Academy of Equipment, Beijing 101416, China*

[2]*College of Engineering & Information Technology, University of Chinese Academy of Sciences, Beijing 100049, China*

[3]*Department of space equipment, the Academy of Equipment, Beijing 101416, China*

[4]*National Defense University, Department of battle command and training, Beijing 100091, China*

Received 1 October 2014, www.cmnt.lv

Abstract

In recent years, the complexity of programs in control systems continues to increase with the growing of automation. Concurrent programming methods have been widely used in designing. However, it is a lack of effective concurrent error checking tool for control system programs. Therefore we proposed a statefull DPOR method with sleep set, and designed a dynamic checking tool for control systems Multithread programs, in which we expand Labelled Transition Systems to record the priority of interrupt and the enabled flag as a system model. We gave formal description for deadlock, data race and atomicity violation three concurrency errors. Finally we realize the testing tool which can detect multi-threaded and multi-interrupt concurrent errors in the control system. The result of Experiment shows that our method has higher efficiency and accuracy.

Keywords: concurrent program, multithread, multiple interrupts, concurrency errors

1 Introduction

As increasing sensors are used in control systems nowadays, multi-threaded and multi-interrupt designing methods gain their popularity. However, due to the randomness of parallel execution, the process of designing and testing are more and more difficult. Generally speaking, a concurrent program, which has *n* threads (or interrupts) with k steps in each code block, may have possible interleaving.

So far there are some concurrent program testing tools, such as VeriSoft, Inspect and so on. MIDAC [1] use function summary technology to reduce state space needed to be traversed in static analysis process in MIDAC. The main principle of documentation [2,3] is to revise the interrupt functions to "semantic" equivalent to multi-threaded programs. Verisoft [4] is a tool for automatically searching coordination problems (deadlocks, atomicity violations, data race etc.) and assertion violations in a software system by generating, controlling, and observing possible executions and interactions of its all components. Inspect [5] is a runtime model checker for multithreaded C programs. It examines all relevant thread interleavings under dynamic partial order reduction, revealing concurrency bugs including deadlocks, data races and assertion violations. Documentation [6] proposes a checking method for multiple interrupts by comparing interrupts with each other through rising interrupt frequency.

By studying the Testing technology of concurrent programs, we design an error verification tool that can test multithreaded and multi-interrupt programs in a real-time system, revealing three kinds of errors: atomicity violation, deadlock, and data race. In this paper, Section 2 describes the model for concurrent programs in a real-time operating system; Section 3 explains the dynamic partial order reduction algorithm and its extension, which can handle the state space of multi-threaded and multi-interrupt programs; Section 4 illustrates three concurrency bug detection algorithms and formal definitions; Section 5 presents the implementation and experiment of our algorithm.

2 Concurrent Programs LTS

2.1 LABELED TRANSITION SYSTEMS MODEL

We use Labeled Transition Systems (LTS) [7] as the basic model for concurrent programs.

Definition 1 LTS is a four-tuple: $M = (S, init, T, R)$, where S is the finite set of concurrent program, $init(S_0)$ is the initial state of concurrent program, T is the finite set of transitions, and $T \subseteq S \times S$, R is the set of relations of transitions.

*** *Corresponding author's*** e-mail: haorenlianghao@126.com

2.2 MODELLING FOR MULTITHREAD PROGRAMS

Given a Multithread program, which contains *α* functions, we extend LTS model as a five-tuple:

$$M_{fid}^{p} = (S_{fid}, init_{fid}, T_{fid}^{p}, R\ , IFlage_{fid}),$$

where *fid* is the only ID for each function(threads and interrupts), *IFlage* represents the interrupt flag. Thus the parallel migration of LTS model for concurrent programs is:

$$M_{||} = (S_{||}, init_{||}, T_{||}, R_{||}, IFlage_{||}) =$$
$$< S_1 \times S_2 ... \times S_n, (init_1, init_2 .. init_n),$$
$$\bigcup_{i=1}^{n} T_i^P, R_{||}, IFlage_{fid} >$$

A global state $S_{||}$ of a concurrent program consists of local state of each function and the shared state of all global objects. Functions (threads and interrupts) communicate with each other via global objects. The operations which access global objects are called visible operations; likewise the operations on local objects are invisible operations. A transition transforms the system from one state to another by performing one visible operation on global objects.

3 Reduction for State Space

We make use of Dynamic Partial-Order Reduction (DPOR) [9-11] to reduce the state space. However DPOR is designed for multi-threaded programs, failing to deal with real-time system programs. As a result we redesign DPOR to reduce the state space for multi-threaded and multi-interrupt programs.

3.1 DEFINITIONS FOR PARTIAL ORDER REDUCTION

DPOR focuses on reducing state space. We shortly introduce some basic principles of DPOR algorithm.

Definition 2 $R \subseteq T \times T$ is an independent relation, if and only if for each $< t_1, t_2 > \in R$, it holds the following two properties:

1) If transition t_1 is enabled in state s and $s \xrightarrow{t_1} s'$, if and only if transition t_2 is enabled in state s', t_2 is also enabled on state s.

2) If t_1, t_2 are enabled in state s, and there is a unique state s', leading to $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.

Definition 3 A set T_p of enabled transitions in state s is a persistent set if and only if for each nonempty sequence of transitions $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 ... \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ from s in a concurrent programs and only includes transitions $t_i \notin T_p$, $1 \le i \le n$, t_n is independent with any transition in T_p.

Definition 4 The Happens-Before relation is a relation on a sequence of transitions $\pi = (t_1, t_2, .. t_n)$, such that:

1) if $i < j$ and t_i, t_j is dependent then $t_i \underset{\pi}{\rightarrow} t_j$;

2) $\underset{\pi}{\rightarrow}$ relation is a transitively close.

3.2 THE STATEFUL DPOR

Given a concurrent program, we divide the function space into two sets, a thread set Tid, and an interrupt set Iid. The set Fid is the total set of the function space and $Fid = Iid + Tid$ .we use $fid(t)$ to denote the identity of the function that executes transition t. $pre(s,t)$ denotes the state s' that $s' \xrightarrow{t} s$. $next(s,t)$ denotes the state s'' that $s \xrightarrow{t} s''$. $s.enabled$ denotes the set of transition enabled in state s. $s.backtrack$ refers to the set of functions with transitions $t \in s.enabled$ that will be executed in the next execution. $s.done$ is the set of transitions that are already executed. $pre(s,t)$ is the transition from which can reach state s.

Hash table H is to record the visited states, so the scheduler will not search the visited state again. We employ an efficient mechanism called Happens-Before Dependency Graph (short for G) to avoid unsoundness. Let $M = (S, s_0, T, R)$ be a model for a concurrent program. G includes the Happens-Before relationships between all visible operations in the visited state space. is directed graph for M, which contains all the Happens-Before Dependency of visible operations. Each note $v \in V$ is a visible operation, that is $\forall v \in V : \exists t \in T : t_g = v$. Given a sequence of transitions $s_1 \xrightarrow{t} s_2 \xrightarrow{t'} s_3$, the algorithm will add a directed edge (t_g, t'_g) into G.

Definition 5 mapping for visible operation relations, for each node $g \in G$, $v \in V$ is a visible operation, that is $\forall v \in V : \exists t \in T : t_g = v$. Likewise, $e \in E$ is a visible operation, and $\forall e \in E : \exists t' \in T : t'_g = e$. For each sequence of transition $s_1 \xrightarrow{t} s_2 \xrightarrow{t'} s_3$ in dynamic checking, the node t_g will be added into the mapping for visible operation relations $g(t') \overset{\Delta}{=} \{t \mid t \rightarrow t'\}$.

Take $T_s = \{\pi_1, \pi_2, ..., \pi_3\}$ be the set of all sequence of transitions from state s, the graph of mappings for visible operation decency $G_T(s) = \{g_{\pi_1}, g_{\pi_2}, ..., g_{\pi_n}\}$.

If the tested program contents no infinite loop, so the state space is limited, and the sequence of transitions from state s is finite. The set of finite sequence of transitions on state s if $T_s = \{\pi_1, \pi_2, ..., \pi_3\}$. The graph of mapping for visible operation decency is $G_T(s) = \{g_{\pi_1}, g_{\pi_2}, ..., g_{\pi_n}\}$. A sequence of transitions π arriving at state s, the all backtrack transition can be found out for $G_T(s)$ (Figure 1).

```
1.  UpdateBackTrackSetII (statestack $S$){
2.    let $\pi$ be the sequence of transitions associated with $S$;
3.    let $U = \{v | \exists v \in top(S).enabled,\ v$ is reachable in $G$ from the node $t_g\}$ ;
4.    for each $g_{\pi_n}(t) \in G(top(S))${
5.      for each $t \in g_{\pi_n}(t)$ {
6.        if $t_d = last(\pi),\ t_d$ is dependent and may be co_enabled with $t$ and $t_d \nrightarrow_{(\pi,t)} t$
                  and $\forall t' \in g(t)$: $t_d \nrightarrow_{(\pi,t)} t'$
7.        if ( $t_d$=null) continue;
8.        Let $s_d$ be the state in $S$ from which $t_d$ is executed from;
9.        let $E$ be $\{q \in s_d.enabled \,|\, tid(t_\tau) = tid(q)$, or $q$ in $\pi$, $t_d \underset{\pi}{\rightarrow} q\}$
10.       if ($E != \emptyset$) choose any q in E, add $tid(q)$ to $s_d.baktrack$;
11.       else $s_d.baktrack \leftarrow s_d.baktrack \cup \{tid(q) | q \in s_d.enabled\}$
12.     }
13.   }
14. }
```

FIGURE 1 BackTrackSet with Graph G

There are two important conditions in line 6:

There is not $t_d \rightarrow_{(\pi,t)} t$;

For each transition t', there is not $t' \rightarrow_{\pi_n} t$ and $t_d \rightarrow_{(\pi,t')} t'$.

From these two conditions, we can get that $t' \rightarrow_{(\pi,\pi_n)} t$ (according to theorem 1). Those conditions can guarantee that the transition t is the first transition that access the shared object $obj(t_d)$ in sequence of transitions π_n. So the transition t is the first backtrack point in $s_d.bakctrack$.

Theorem 1 Given the two sequence of transition π and π', for any transition $t_d \in \pi$ and $t \in dom(g_{\pi'})$, if there is no $t_d \rightarrow_{(\pi,t)} t$, such that for any transition $t' \in g_{\pi_n}(t)$, there is no $t_d \rightarrow_{(\pi,t')} t'$, and there must not exist $t_d \rightarrow_{(\pi,\pi')} t$.

Proof Let $t'_k \in \pi'$, t'_j meet the condition that $\{j \,|\, j < k \ \wedge \neg(t'_j \rightarrow_{\pi'} t'_k)\}$. That is t'_j happens before t'_k, and t'_j and may not be the last transition that happens before t'_k.

For all transition $t'_l (j < l < k)$, there not exists $t'_j \rightarrow_{\pi'} t'_l$, otherwise $t'_j \rightarrow_{\pi'} t'_k$ would be true. That is transition may not happen before the transition that is between the transitions t'_j and t'_k. The sequence of transitions received by taking the transition t'_j backwards some steps is equivalent with π'.

Let $\overline{\pi'}$ be the sequence of transitions that is equivalent with π'. Any transitions that happen before t'_k in $\overline{\pi'}$ is in mapping node $g_{\pi'}$.

Further in the sequence $(\pi, \overline{\pi'})$, all the transitions before t'_k happen before t_d. So $t_d \rightarrow_{(\pi,\overline{\pi'})} t'_k$ does not exist due to equivalence of the relationship between the $\overline{\pi'}$ and π'. If $t = t'_k$, $t_d \rightarrow_{(\pi,\pi')} t'_k$ does not exist, so Theorem 1 is true.

Theorem 2 If there is not infinite loop in tested program, for any transition t in π, any state s in the S_π,which is sequence of states associated with π.the set of backtrack point $s_d.backtrack$ computed with the method UpdateBackTrackSetII is same as the one classic method UpdateBackTrackSet in [9]

Proof To prove the Theorem, we need only prove that the $s_d.backtrack$ received form UpdateBackTrackSetII is as same as the one received form UpdateBackTrackSet.

If the set of backtrack points at state s_d needs to be updated through the UpdateBackTrackSet. We can conclude that there must exist a transition $t \in dom(g_{\pi'})$, which must not meet $s_d \rightarrow_{(\pi,t)} t$.

Further we can get that there must not exist $s_d \rightarrow_{(\pi,\pi')} t$ by the Theorem 1.

Let $t = t'_l$, so that $t = next(t'_{l-1}, tid(t))$. So at the state s_{l-1}, there is not $t_i \rightarrow_{(\pi,\pi'|l)} tid(t)$ ($\pi'|l$ is the sequence of transition $t'_1, t'_2, ..., t'_{l-1}$ that is the sequence of transitions before transition t'_l in the sequence of transitions (π, π')). When UpdateBackTrackSetII visit at the transition t'_{l-1} during the dynamic checking, the set of backtrack point at state s'_{l-1} will be updated (Because $t_i \rightarrow_{(\pi,\pi'|l)} tid(t)$ and t is the first transition, which access the shared object).

The following we will prove that the two sets of *tid* added into $s_d.backtrack$ that are separately computed by UpdateBackTrackSetII and UpdateBackTrackSet is same the each other.

To prove the above conclusion, we only need to prove the sets E separately computed by two method is same with each other.

According to definition 5 of mapping of visible operation decency, we can get the following conditions:

Conditions 1 $\exists t' \in g(t)$ such that $q = tid(t')$ or $\exists t_j \in \pi (j > i)$ and $q = tid(t_j)$ and $t_j \rightarrow_\pi tid(t)$

Conditions 2 $\exists t_j \in \{(\pi, \pi') | l\}$ such that $j > i$ and $q = tid(t_j)$, ($t_j \in \{(\pi, \pi' | l)\}$) and $t_j \rightarrow_{(\pi,\pi'|l)} tid(t)$.

The conditions 1 and 2 are equivalent. Furthermore, if the set $s_{i-1}.backtrack$ gotten by UpdateBackTrackSetII need to be update at s_{i-1} in the sequence of transitions (π, π'). So that there not exists $t_i \rightarrow_{(\pi,\pi')} t'_l$, and t_i is the last transition that access the shared object $obj(t'_i)$ in sequence of transitions ($\pi, \pi' | l$). So t'_l is the first transition that access the shared object $obj(t_i')$ in π' and $t_l' \in dom(g_{\pi'})$.

As $t_i \rightarrow_{(\pi,\pi')} t_l'$ does not exist, there not exist $t_i \rightarrow_{(\pi,t_l')} t_l'$. For each $t' \in g_{\pi'}$, there not exist $t_i \rightarrow_{(\pi,t_l')} t'$. Otherwise we can get that $t_i \rightarrow_{(\pi,\pi')} t_l'$ from $t' \rightarrow_{(\pi,\pi_l')} t_l'$.

So that the two sets E gotten by UpdateBackTrackSet and UpdateBackTrackSetII are the same. The Theorem 2 is true.

3.3 THE SLEEP SET

We introduce the sleep set method to stateful drop to reduce the state space further.

s.sleep is a set of transitions that is enabled but not be necessarily executed. DPOR with persistent set only cannot handle the program the infinite loops, since the state space will explore. But there are always infinite loops in the Real-time System Concurrent programs. We use sleep set to reduce redundant interleaving and avoid re-executing the transitions in *s.enabled* which have been executed already.

3.4 THE SDPORS ALGORITHM

Figure 2 presents our SDPORS algorithm. SDPORS first explores an arbitrary interleaving of the concurrent program, and thereafter uses depth-first search to explore the state space until all the interleaving are explored. In line 12 if $fid(t)$ is an interrupt function，only when the interrupt flag is true, the next state s' is reachable, otherwise we will pop a new state to continue searching. The function PriorityJudgments(s,t) is to analyze whether an interleaving does exit in an actual execution.

In Figure 3, intuitively a thread can be blocked regardless of its priority, but interrupts cannot. Hence in line 2, if $fid(t)$ and $fid(pre(s,t))$ are both thread functions, the algorithm allows DPOR to go ahead conservatively. An interrupt function $fid(pre(s,t))$ can interrupt the execution of a thread $fid(t)$ (line 3). If they are both interrupt functions, only a high-priority interrupt can interrupt the execution of a low-priority interrupt (line 4).

```
1.  Statestack $S$; state $s$; $H,G$ is empty;
2.  DPOR(){
3.    $S = H = \emptyset$;
4.    $S.push(s_0)$;
5.    if($\exists q \in s_0.enabled$), $s_0.backtrack = \{tid(q)\}$;
6.    while($S! = \emptyset$){
7.      $s = S.top()$;
8.      if($\exists q \in s.backtrack \backslash s.done$){
9.        $s.done \leftarrow s.done \cup \{q\}$;
10.       $s.sleep \leftarrow s.sleep \cup \{t \in s.enabled | fid(t) \in s.done\}$;
11.       $s.backtrack \leftarrow s.backtrack \backslash \{q\}$;
12.       if(($fid(t) \in Iid\ And\ fid(t).IFLAG = enabled$)or( $fid(t \in Tid)$)){
13.         let $s'$ be a state such that $s \overset{t}{\rightarrow} s'$;
14.         $s'.sleep \leftarrow \{t' \in s.sleep \wedge (t,t')$is indepantent$\}$;
15.         $s'.enabled \leftarrow s'.enabled \backslash s'.sleep\}$;
16.         $S.push(s')$;
17.         let $\pi'$ be the sequence of transitions associated with $S$;
18.         add $\pi'$ to $s.T_\pi$;
19.         if( (error=BUGDETECT($s'$)) ≠False) report error; exit;
20.       }
21.       else{
22.         $S.pop()$; continue;
23.       }
24.       if($s' \in H$) UpdateBackTracesSetII ($S$);
25.       else {
26.         UpdateBackTrackSet($s'$);
27.         if($\exists \tau \in Fid$: $\exists t \in \{s'.enabled: fid(t) = \tau$ &&PriorityJudgments$(s',t)\}$){
28.           $s'.backtrack \leftarrow \{\tau\}$;   $s'.done \leftarrow \emptyset$;
29.         }
30.         if$\left(\exists x \in S \text{ s.t. } x \overset{t_x}{\rightarrow} s \overset{t}{\rightarrow} s'\right)${
31.           let $\pi$ be the sequence of transitions such that $(\pi,t) = \pi'$;
32.           let $TEMPG = G(s')$,  Add $t_x$ to $g_{\pi'}(t)$;
33.           for each $g_{\pi_n}$ in $G(s)$ add $g_{(t,\pi_n)}$ into $G(s')$;
34.           if($TEMPG \neq G(s')$) for each $\pi$ such that $s_0 \overset{\pi}{\rightarrow} s$ UpdateBackTracesSetII (S);
35.           $H = H \cup \{s\}$;
36.         }
37.       }
38.     }
39.     else{
40.       $s = s.pop()$;
41.     }
42.   }
43. }
```

FIGURE 2 The SDPORS algorithm

```
1.  PriorityJudgments(state $s$,transition $t$){
2.    If($fid(t) \in Tid$ && $fid(pre(s,t)) \in Tid$) return true;
3.    else if($fid(t) \in Iid$  && $fid(pre(s,t)) \in Tid$) return true;
4.    else if($fid(t) \in Iid$  && $fid(pre(s,t)) \in Iid$ && $fid(pre(s,t)).PRI < fid(t).PRI$)
5.      return true;
6.    else return false;
7.  }
```

FIGURE 3 Judgments for Priority

DPOR uses UpdateBackTraceSet of FIGURE 4 to update the backtrack sets for each state with classic method. The UpdateBackTraceSet can compute the persistent set in s_d*.backtrack* . The proof have been presented by documentation [9].

```
UpdateBackTraceSet (state $s$){
  let $\pi$ be the sequence of transitions associated with $S$;
  for all Functions $f${
    let $t_f \in s.enabled$  $fid(t_f) = f$;
    let $t_d$ be the lasted transition in $\pi$ that is dependent and may be co_enabled with $t_f$;
    if (PriorityJudgments$(s,t_d)$ = False and $t_d$=null) continue;
    led $s_d$ be the state in $S$ from which $t_d$ is executed from;
    let $E$ be $\{q \in s_d.enabled \mid fid(t) = fid(q)$,or $q$ in $\pi$, $t_d \underset{\pi}{\rightarrow} q$ and there is $q \underset{\pi}{\rightarrow} fid(t_d)$ $\}$
    if ($E! = \emptyset$) choose any $q$ in E, add $fid(q)$ to $s_d.baktrack;$
    else  $s_d.baktrack \leftarrow s_d.baktrack \cup \{fid(q) | s_d.enabled\}$
  }
}
```

FIGURE 4 UpdateBackTraceSet

4 Concurrent error detecting

4.1 DEADLOCK

Definition 6 A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does[12]. Let s_d be a state in concurrent program model M . If $s_d.enabled = \phi$, s_d is a deadlock state, and if $\exists T_d : s \overset{T_d}{\Rightarrow} s_d$, s_d is reachable in DPOR.

Proof At first, if $Length(T_d) = 0$, intuitively the conclusion is established.

Secondly, if $Length(T_d) = n$, set $T_d' = t_0, t_1, \ldots, t_n$, and the sequence of transitions from s : $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_d$. Let the persistent set T_p of state s is not empty, so that $t_i, 0 \le i \le n$ is independent with the transitions in T_p according to Definition 3. As a result transition t in T_p can still be executed in s_d . That is contrary to the fact of the state deadlock. So there are invariably some transitions in T_d' and in T_p simultaneously.

Thirdly, set $t_i \in T_d'$ is the first transition in T_p , and $T_d'' = t_i, t_0, t_1, .. t_{i-1}, t_{i+1}, \ldots, t_n$. The different point is that t_i is the first transition between T_d'' and T_d' . $\forall t_j, 0 \le j \le i$ is independent with t_i according to Definition 3.3, so there are $s_0 \overset{T_d'}{\Rightarrow} d$ and $s_0 \overset{T_d''}{\Rightarrow} d$ simultaneously. Since $t_i \in T_p$, the conclusion for $Length(T_d) = n + 1$ is still established.

4.2 ATOMICITY VIOLATION

Definition 7 In concurrent programming, all operations with atomicity are executed to completion, or none are performed [12]. Given a sequence of transitions $< s_1, s_2, \ldots, s_n >$, $\exists i \in [1, n]$, state s_i is in the atomic block. $OPW(A)$ is the set of all write operations in atomic block A . $OPR(A)$ is the set of all read operations in atomic block A . $OP(A)$ represents the set of all visible operations. $op(s)$ denotes the operations on s . If $\exists s_j \subset S, i < j < n$, so that $\left(OPR(A) \cap write_j\right) \cup \left(read_j \cap OPW(A)\right) \cup \left(OPW(A) \cap write_j\right) \ne \phi$, there is an atomicity violation. $read_j$ is the operation that $op(s_j) = read$ and $write_j$ is the operation that $op(s_j) = write$.

4.3 DATA RACE

Definition 8 Race conditions occur when different processes access shared data without explicit synchronization [13,14]. For any state s in model M , set of all the write operations on s $OPW(s) = \{op \mid op \in OP(s), and, op(s) = write\}$. If $\exists t, t'$ $t, t' \in s.enabled$. t or t' in set $OPW(s)$ and the relationship of (t , t') is dependent, the race conditions may occur.

4.4 METHOD FOR CONCURRENT ERROR DETECTING

The function BUGDETECT is used to detect deadlock, atomicity violation and data race.

```
1.  BUGDETECT(s){
2.    if($s.disenabled \ne \emptyset$ and $s.sleep = \emptyset$)
3.      return a deadlock;
4.    if ($\exists$A: which is a atomic blook such that state $s$ is a state in a atomic block A){
5.      let $S_{Atom}$ be the sequence of state in atomic block A;
6.      let $T_{Atom}$ be the sequence of transition with $S_{Atom}$;
7.      let $OPW$ be the write operations in $T_{Atom}$;
8.      let $OPR$ be the read operations in $T_{Atom}$;
9.      for (each $t \in T_{Atom}$){
10.       if($\exists t'$:$fid(t').PRI > fid(t).PRI$ and $t'$ is co_enabled at the same state with $t$){
11.         if ($op(t') = write\ and\ op(t') \cup OPR \ne \emptyset$) return an atom write mistake;
12.         if ($op(t') = read\ and\ op(t') \cup OPW \ne \emptyset$) return an atom read mistake;
13.         if ($op(t') = write\ and\ op(t') \cup OPW \ne \emptyset$) return an atom Dual write mistake;
14.       }
15.     }
16.   }
17.   if($\exists t, t' : t, t' \in s.enabled$ and ($Fid(t').PRI \ne Fid(t).PRI\ and\ (t, t')$ is dependent){
18.     if($op(t') = write$ or $op(t) = write$) return maybe a Witten out of sync;
19.   }
}
```

FIGURE 5 Method for concurrent error detecting

In Figure 5, line 2-3 check deadlocks. According to section 4.2, we design the atomicity violation checking method from line 4 to 16. Line 17, 18 is to checking race conditions with the different priority.

5 Implementation and experiment

5.1 CONCURRENT TESTING TOOL DESIGNING

We implement the algorithms of Sections 3 and 4 based on our dynamic model checker.

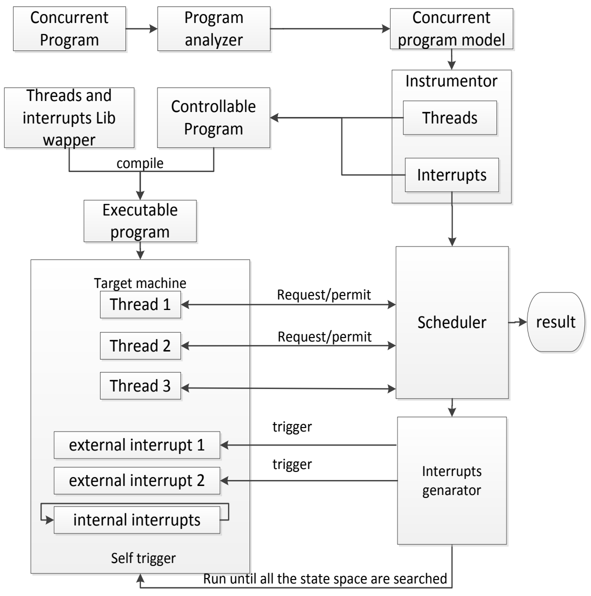

FIGURE 6 Concurrent testing tool Framework

Given a concurrent program, it first uses program analyzer to get the concurrent LTS model, then instruments the program with the code on threads, interrupts and shared objects to register the functions and objects information to scheduler. Instrumented code can communicate with scheduler during runtime. Thereafter it executes the program on the target machine, where scheduler controls threads to execute or block and interrupt generator to trigger specific interrupts at specific time.

5.2 EXPERIMENTAL RESULT

We select a coding and communication program on ARM platform, and a Four-rotor unmanned helicopter control system on PPC platform.

We use "/" to denote the runtime that is over 424 hours (86400s) in

Intuitively our approach is more effective and can avoid the state space overhead (Table 1).

TABLE 1 Result 1

Benchmarks	Threads/interrupts	Without DPOR		With DPOR	
		transitions	time	transitions	time
ARM1	2/2	41k	472s	3.6k	23s
ARM2	3/2	4875k	2293s	23k	294s
ARM3	4/3	/	/	1152k	678s
PPC1	2/2	612k	1145s	45k	93s
PPC2	4/4	6089k	6650s	482k	860s
PPC3	6/5	/	/	5157k	7985s

"T/I" denotes the number of threads and interrupts. "t" is the number of transitions, "T" presents the runtime, and "E" means concurrent errors. Verisoft and Inspect are applied to compare correctness and efficiency with our checker Verisoft is a very mature testing platform with the stateless DPOR, so correctness is guaranteed, but efficiency is low. In Inspect's checking procedure, interrupts are treated the same as threads, whereas they are fundamentally different, resulting in misinformation in the result. Our approach is similar to Inspect, but we make use of interrupt generator to trigger interrupts, and take the impact of priority into the checking method to acquire better efficiency and correctness (Table 2).

TABLE 2 Result 2

Benchmarks	T/I	Verisoft			Simulation of thread inspect			Our checker		
		t	T	E	t	T	E	t	T	E
ARM1	2/2	15k	267s	0	4k	25s	0	3.6k	23s	0
ARM2	3/2	91k	986s	1	27k	314s	2	23k	294s	1
ARM3	4/3	2970k	2898k	2	129k	690s	4	1152k	678s	2
PPC1	2/2	131k	310s	1	47k	101s	1	45k	93s	1
PPC2	4/4	1347k	2987s	1	544k	898s	2	482k	860s	1
PPC3	6/5	90465k	29981s	2	5832k	8764s	3	5157k	7985s	2

6 Conclusion

We present an efficient dynamic model checking method for testing real-time system concurrent programs. It incorporates and extends the functionality of DPOR to handle programs with both interrupts and threads. According to classic definition of threes common concurrency errors, we give formal description of LTS and the detecting algorithm, and realize the testing tool for real-time system concurrent programs. However, we do not incorporate the lockset methods, and therefore there is still space for further improvement in efficiency and correctness.

References

[1] Wu X, Wen Y, Wang J, Fu X, Qi Y, Gu B 2011 Data race and Atomicity Checking for C Porgrams with Multiple Interruptions *Journal of Frontiers of Computer Science and Technology* 1086-93

[2] Regehr J, Cooprider N 2007 *Interrupt verifition via thread verification Electronic Notes in Theoretical Computer Science* **174**(9) 139-50

[3] Hofer W, Lohmann D, Scheler F, Schroder-Preikschat W 2009 Sloth: Threads as interrupts *The 30th IEEE Real-Time Systems Symposium*

[4] Dingel J 2003 Computer-Assisted Assume/Guarantee Reasoning with VeriSoft *Proceedings of the 25th International Conference on Software Engineering* (ICSE'03) 138-48

[5] Yang Y, Chen X F, Gopalakrishnan G, Kirby R M 2009 Inspect: A Runtime Model Checker for Multithreaded C Programs *School of Computing* University of Utah Salt Lake City UT 84112 USA

[6] Fu X, Chen L 2012 Framework for testing multiple interrupts program *Computer engineer and design* **02**(2) 617-23

[7] Chaki S, Clarke E, Ouaknine J, Sharygina N 2004 Automated, Compositional and Iterative Deadlock Detection *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-design* 201-210

[8] Netzer R H B 1992 What are Race Conditions? Some Issues and Formalizations *ACM Letters on Programming Languages and Systems* **1**(1) 558-65

[9] Flanagan C, Godefroid P 2005 Dynamic partial-order reduction for model checking software *Proceedings of POPL 2005 Long Beach* California USA

[10] Godefroid P editor 1996 Partial-Order Methods for the Verification of Concurrent Systems-An Approach to the State-Explosion Problem Lecture Notes in Computer Science **1032**

[11] Yi X 2006 Slicing Execution for Verification of C Programs *National University of Defence Technology* 2006

[12] Silberschatz A, Galvin P B, Gagne G 2012 Operating System Concepts (9th Edition) *John Wiley & Sons Inc* **283**733-734

[13] Netzer R H B 1991 Race Condition Detection for Debugging Shared-Memory Parallel Programs *University of Wisconsin-Madison*

[14] Netzer R H B 1992 What are Race Conditions? Some Issues and Formalizations *ACM Letters on Programming Languages and Systems* 1992 **1**(1) 558-65

Authors

Hao Liang, March 1981, Shanxi, Taiyuan, China.

Current position, grades: PhD candidate at the Academy of Equipment.
Scientific interests: computer, automation, embedded systems design, real-time embedded systems, and models for complex systems.
Publications: 6.

Yunfeng Ai, September 1979, Shandong, Jinan, China.

University studies: PhD in Control Engineering at the Institute of Automation in Chinese Academy of Science.
Scientific interests: computer, automation, embedded systems design, real-time embedded systems, intelligent transportation systems, intelligent vehicles driver's modeling and behavior analysis.
Publications: 36.

Huairong Shen, July 1954, Anhui, Sucheng, China.

University studies: PhD in National University of defence technology.
Scientific interests: overall design of aerospace equipment, navigation and guidance, drone technology, damage mechanics, aerospace Science and technology.
Publications: 160.

Yongchao Zhao, 1982, Hebei, Shijiazhuang, China

University studies: PhD in PLA artillery Academy.
Scientific interests: military operational research, computer simulation.
Publications: 5.