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: \(M = (S, \text{init}, T, R)\), where \(S\) is the finite set of concurrent program, \(\text{init}(S_0)\) 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 \(a\) functions, we extend LTS model as a five-tuple:

\[
M^P_{fid} = (S_{fid},init_{fid},T^P_{fid},R_{fid},IFlag_{fid})
\]

where \(fid\) is the only ID for each function THREADS and INTERRUPTS, \(IFlag\) represents the interrupt flag. Thus the parallel migration of LTS model for concurrent programs is:

\[
M|| = (S||,init||,T||,R||,IFlag||)
\]

\[
\bigcup_{i=0}^{n} T_i, R_i, IFlag_{fid_i} >
\]

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 \(s \rightarrow s'\), 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 \(s\), there is a unique state \(s'\), leading to \(s \rightarrow s'\) and \(s \rightarrow s'\).

**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 \(s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \rightarrow s_n \rightarrow s_{n+1}\) from \(s\) in a concurrent programs and only includes transitions \(t \notin T_p\), \(1 \leq i \leq n\), \(t_i\) is independent with any transition in \(T_p\).

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

1) if \(i < j\) and \(t_i, t_j\) is dependent then \(t_i \rightarrow t_j\);  
2) \(\pi\) 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 \(Id\). The set \(Fid\) is the total set of the function space and \(Fid = Fid + Tid\). We use \(fid(t)\) to denote the identity of the function that executes transition \(t\). \(pre(s, t)\) denotes the state \(s'\) that \(s' \rightarrow s\). \(next(s, t)\) denotes the state \(s''\) that \(s \rightarrow s''\). \(s\text{enabled}\) denotes the set of transition enabled in state \(s\). \(s\text{backtrack}\) refers to the set of functions with transitions \(t \in s\text{enabled}\) that will be executed in the next execution. \(s\text{done}\) 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 \(M = (S, G, R, T)\) 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 \(\forall v \in V: \exists t \in T: t_v = v\). Given a sequence of transitions \(s_1 \rightarrow s_2 \rightarrow s_3\), the algorithm will add a directed edge \((t_1, t_2')\) into \(G\).

**Definition 5** mapping for visible operation relations, for each node \(g \in G\), \(v \in V\) is a visible operation, that is \(\forall v \in V: \exists t \in T: t_v = v\). Likewise, \(e \in E\) is a visible operation, and \(\forall e \in E: \exists t \in T: t_e = e\). For each sequence of transition \(s_1 \rightarrow s_2 \rightarrow s_3\) in dynamic checking, the node \(t_2\) will be added into the mapping for visible operation relations \(g(r) = [t_1 \rightarrow r']\).

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

If the tested program contains 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 as
state \( s \) if \( T_s = \{ q_1, q_2, \ldots, q_n \} \). The graph of mapping for visible operation decency is \( G_T(s) = \{ (g_{q_1}, g_{q_2}, \ldots, g_{q_n}) \} \). A sequence of transitions \( \pi \) arriving at state \( s \), the all backtrack transition can be found out for \( G_T(s) \) (Figure 1).

Further in the sequence \( (\pi, \pi') \), all the transitions before \( t'_1 \) happen before \( t_d \). So \( t_d \rightarrow (\pi, \pi') t'_1 \) does not exist due to equivalence of the relationship between the \( \pi' \) and \( \pi \). 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 \( \pi \), any state \( s \) in the \( S_{\pi} \), which is sequence of states associated with \( \pi \), the set of backtrack point \( s_{d, \text{backtrack}} \) computed with the method \( \text{UpdateBackTrackSetII} \) is same as the one classic method \( \text{UpdateBackTrackSet} \) in [9]

**Proof** To prove the Theorem, we need only prove that the \( s_{d, \text{backtrack}} \) received form \( \text{UpdateBackTrackSetII} \) is as same as the one received form \( \text{UpdateBackTrackSet} \).

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

Further we can get that there must not exist \( s_d \rightarrow \_(\pi, \pi') t \) by the Theorem 1.

Let \( t = t'_k \), so that \( t = \text{next}(t'_k, \text{tid}(t_i)) \). So at the state \( s_{1, t'} \), there is not \( t \rightarrow (\pi, \pi') t_i \) tid(t) (\( \pi' \) | l) is the sequence of transition \( t'_1, t'_2, \ldots, t'_l \) that is the sequence of transitions before transition \( t'_1 \) in the sequence of transitions (\( \pi, \pi' \)).

When \( \text{UpdateBackTrackSetII} \) visit at the transition \( t'_1 \) during the dynamic checking, the set of backtrack point at state \( s_{1, t'} \) will be updated (Because \( t_i \rightarrow (\pi, \pi') t_i \) and it is the first transition, which access the shared object).

The following we will prove that the two sets of \( \text{tid} \) added into \( s_{d, \text{backtrack}} \) that are separately computed by \( \text{UpdateBackTrackSetII} \) and \( \text{UpdateBackTrackSet} \) 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:

1. **Conditions 1** \( \exists t_i \in g(l) \) such that \( q = \text{tid}(t_i) \) or \( \exists t_j \in g(p(j > i)) \) and \( q = \text{tid}(t_j) \) and \( t_j \rightarrow (\pi, \pi') t_i \) tid(t).

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

The conditions 1 and 2 are equivalent. Furthermore, if the set \( s_{1, t'} \) backtrack gotten by \( \text{UpdateBackTrackSetII} \) need to be update at \( s_{1, t'} \) in the sequence of transitions (\( \pi, \pi' \)). So that there not exists \( t_i \rightarrow (\pi, \pi') t'_j \) and \( t_i \) is the last transition that access the shared object \( \text{obj}(t'_i) \) in sequence of transitions (\( \pi, \pi' \)).
that access the shared object \( obj(t') \) in \( \pi' \) and \( t' \in \text{dom}(g_{\pi'}) \).

As \( t_i \rightarrow_{(\pi, x')} t'_i \) does not exist, there not exist \( t_i \rightarrow_{(\pi, x')} t'_i \) for each \( t' \in g_{\pi'} \). Therefore we can get that \( t_i \rightarrow_{(\pi, x')} t'_i \) from \( t'_i \rightarrow_{(\pi, x')} t'_i \).

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.

\( s.slee p \) 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 \( s.\text{enabled} \) 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 \( \text{PriorityJudgments}(s, t) \) 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).
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 \( \exists T_d : s \Rightarrow s_d \), \( s_d \) 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 \( s : s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s_d \). Let the persistent set \( T_p \) of state \( s \) is not empty, so that \( t_i, 0 \leq i \leq n \) 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 \( T_d = t_i, t_0, t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \). The different point is that \( t_i \) is the first transition between \( T_d \) and \( T_p \) \( \forall t_i, 0 \leq j \leq i \) is independent with \( t_i \) according to Definition 3.3, so there are \( s_0 \Rightarrow d \) and \( s_0 \Rightarrow d \) 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 \( \exists s_j \in S, i < j < n \) \( \phi \), then there is an atomicity violation. \( \phi \) 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 \( s \) \( OPW(s) = \{ op | op \in OP(s), \text{and}, op(s) = write \} \). If \( \exists t, t' \in s, enabled \) \( t \) 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.

---

**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.
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).

“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).

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


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


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.


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.