An Overtime-Detection Model-Checking Technique for Interrupt Processing Systems

Xiaoyu Zhou and Jianhua Zhao

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
Dept. of Computer Sci. and Tech. Nanjing University, Nanjing, China
sandzhou@seg.nju.edu.cn, zhaojh@nju.edu.cn

Abstract. This paper presents an overtime-detection model-checking technique for interrupt processing systems. It is very important to verify real-time properties of such systems because many of them are safety-critical. This paper gives a method to check that critical interrupts can be handled within their timeout periods. Interrupt processing systems are modeled as extended timed automata. Our technique checks whether the system under check can handle critical interrupts in time using symbolic model-checking techniques. Taking an aerospace control system as an example, we show that our technique can find time-scheduling problems in interrupt processing systems.

Keywords: interrupt processing system, overtime detection, model checking.

1 Introduction

Nowadays, real-time embedded systems are widely used in safety-critical systems, such as flight control, railway signaling, health care and so on. Interrupt processing systems play important roles in such safety-critical systems.

In interrupt processing systems [1, 2], it is vital to handle interrupts within given time limits. Fully testing these systems is unrealistic because of the randomly-arrived interrupt sequence, the varying interrupt processing period, and the complexity of both hardware and software systems. For such systems, testing is expensive and low-efficient. Test can only cover a small fraction of state space of the interrupt processing systems under test. Model checking [3] has been introduced as a promising tool to analysis and verify (the models) of interrupt processing systems. It is possible to check whether the specified deadline for the interrupt-handling could be met by exhaustive exploration of the state-spaces of system models [4].

In this paper, we present a new approach to model interrupt processing systems and to detect the overtime situation of interrupt-handling by exhaustively exploring the state spaces of interrupt processing systems.

2 Preliminary

Many existing formal models can be used to model real-time interrupt processing systems, for example, timed automata [5] or hybrid automata [6]. In this paper, we extend timed automata to model such systems.

Y. Yuan, X. Wu, and Y. Lu (Eds.): ISCTCS 2012, CCIS 320, pp. 482–489, 2013.

2.1 Timed Automata

Timed automata can be used to simulate time behaviors of real-time systems by adding a finite set of real-valued clocks and time guards to conventional finite-state automata.

Let $\mathcal{C}$ be a set of clock variables. A clock valuation u over $\mathcal{C}$ is a map from $\mathcal{C}$ to $\mathbb{R}$. For $\in \mathbb{R}$, $u + t$ is also a map. It maps each x in $\mathcal{C}$ to $u(x) + t$. We use $\mathcal{G}(\mathcal{C})$ to stand for time guards over $\mathcal{C}$, which is a conjunction of atomic formulas of the form $x \sim n$, where $x \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and n is an integer.

A timed automaton $\mathcal{A}$ is a tuple $< L, l_0, \mathcal{C}, \Sigma, E, F >$, where L is a finite set of locations; $l_0 \in L$ is the initial location; $\mathcal{C}$ is a finite set of clocks; $\Sigma \subseteq \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}}$ is a finite set of transitions, i.e. a transition e in Σ is a tuple $e = (g, r)$, where $g \subseteq \mathcal{G}(\mathcal{C})$ is the time guard of e and $r \subseteq \mathcal{C}$ is the set of clocks reset by e; $E \subseteq L \times \Sigma \times L$ is a finite set of edges; $F \subseteq L$ is the set of acceptance locations.

We write $l \xrightarrow{e} l'$ if *(l, e, l')*$\in E$. We also use $enable(l)$ to denote the transition set $\{e \mid l \xrightarrow{e} l'$ for some $l'\}$. We say e is enabled at l if $e \in enable(l)$.

A concrete state of this timed automaton is a tuple (l, u), where $l \in L$ and u is a clock valuation over $\mathcal{C}$. A timed automaton may evolve by either time-elapsing or concrete transitions.

- Time-elapsing: $(l, u) \xrightarrow{t} (l, u + t)$.
- Concrete transition: $(l, u) \xrightarrow{e} (l', u')$ where $e = (g, r)$ if following conditions hold:

  — *(l, e, l')*$\in E$;
  — For each time guard $x \sim n$ in g, $u(x) \sim n$;
  — For each clock $x \in r$, $u'(x) = 0$; and for each $x \in \mathcal{C} - r$, $u'(x) = u(x)$.

The basic reachability analysis calculates symbolic successors of every enabled transition at each reachable state till no more new state can be generated [7].

3 Modeling Interrupt Processing Systems

3.1 Modeling Interrupt Sources

In an interrupt processing system, there are usually two kinds of interrupt sources: regular interrupts which occur repeatedly with fixed time intervals; and contingency interrupts which occur randomly. Each interrupt source is assigned with a unique priority.

Regular interrupts are spontaneous activities which are issued every 0.5s, 1s, 2s and 4s respectively. Regular interrupts are usually used to maintain the system state. It includes routine tasks such as information backup, time-triggered communications, etc. Each regular interrupt is assigned a fixed priority in direct proportion to its frequency. So, the assigned priority of 0.5s-interrupts is higher than that of 1s-, 2s- and 4s-interrupts and so on.

Contingency interrupts are used to dealing with events that happen randomly, for example, start or shut-down of the engine, adjusting the direction of the aircraft. The occurrence frequency of the contingency interrupt is unpredictable. Generally, the priority of a contingency interrupt is higher than that of regular interrupts.

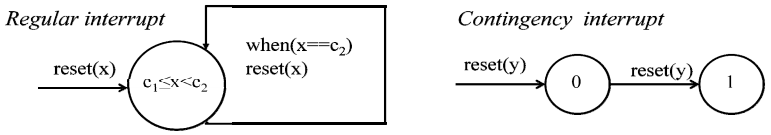

Fig. 1. The timed automata model of interrupt sources

In Figure 1, we use timed automata to model two kinds of interrupt sources. The automata modeling a regular interrupt consists of a local clock *x* and a self-loop state. Supposing the interval of the regular interrupt source is 2s, c_1 and c_2 are set to 0 and 2000 respectively.

3.2 Modeling the Interrupt Vector Table

The interrupt vector table in an interrupt processing system is a table of interrupt descriptors that associates an interrupt handler with an interrupt request in a machine-specific way. In our work, we simplify this concept and use a table, denoted as $InterruptVector$, to map interrupt sources to corresponding interrupt procedures. The elements in this table are sorted by interrupt priorities. Given an interrupt source with the priority p_i the mapping relation in the $InterruptVector$ points out the corresponding interrupt handler,

$$InterruptVector[p_i] = InterruptHandler_{id}.$$

3.3 Modeling Interrupt Handlers

In response to each interrupt source, we abstract the interrupt handler, denoted as $InterruptHandler$, as a list of to-do tasks with upper and lower execution time bounds. Each task in the to-do list is viewed as an atomic transaction which takes some time units to execute. Let the $InterruptHandler_i$ contains n tasks: $task_1$, $task_2$, … , $task_n$, where $task_k$ takes t_k time units to be accomplished. There exists a time constraint as follow.

$$lowerBound(InterruptHandler_i) \leq \sum_{j=1}^{n} t_j \leq upperBound(InterruptHandler_i)$$

4 Deadline Detection

4.1 Simulating the Interrupt Processing

In our algorithm, a stack is used to model the state of interrupt processing. The stack is a snapshot of the current system context. Figure 2 shows the overall organization of the interrupt processing model.

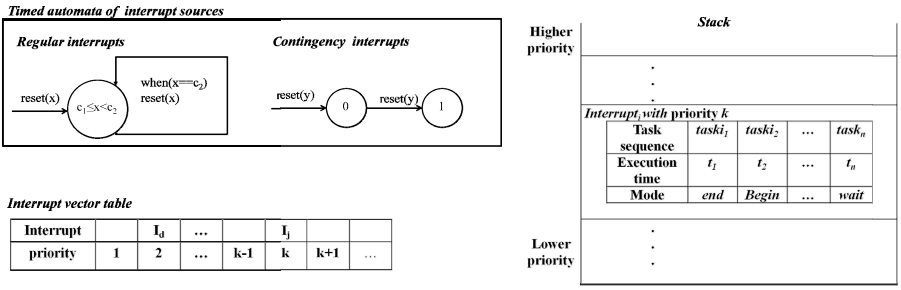

Fig. 2. The organization of our model system

A stack member keeps the information of an $InterruptHandler$ and records the details of execution situation of the task sequence. Each task of the $InterruptHandler$ in the stack has three modes: wait to be executed, begin to execute and finish the execution. The priority of stack member increases from bottom to top such that the $InterruptHandler$ at the top of the stack has the highest priority, while the $InterruptHandler$ at the bottom of the stack has the lowest priority.

4.2 Calculating Symbolic Successor of Interrupt Processing Systems

A global state of an interrupt processing system is consists of three components:

- the state (l, D) of timed automata modeling interrupt sources;
- an interrupt vector table $InterruptVector$ to record interrupt requests;
- a *stack* contains runtime context of involved interrupt handles.

Let $topPRIvector$ and $topPRIstack$ be the current highest priority respectively in the $InterruptVector$ and the $stack$. The set of enabled events of the global state $globalstate = <(l, D), InterruptVector, stack>$ is calculated by the following algorithm `enabled(globalState)`. The successor of $globalstate$ is calculated according to different kinds of enabled events.

```
for each edge leaving e from (l,D)
  add e to globalState.enabledEvents;
if ( topPRIvector > topPRIstack)
  add topPRIvector to globalState.enabledEvents;
else if(topPRIvector <= topPRIstack)
  add stack to globalState.enabledEvents;
```

For a transition e leaving from a state (l, D) of the interrupt source model, the successor is calculated by the following algorithm `successorOfTA(globalState)`.

```
Let e_i be an enabled transition at (l,D)
(l',D') = sp((l,D),e_i);
if (D' ≠ EMPTYSET)
p:= the priority of the corresponding TA;
InterruptVector[p]:= InterruptHandler_i;
return <(l',D'), InterruptVector, stack>;
```

The operator sp is used to calculate the symbolic successor of a symbolic state (l, D). $sp((l, D), e)$ represents the set:

$$\{(l', u') \mid \exists\, (l, u) \in (l, D) \cdot ((l, u) \xrightarrow{e} (l', u'))$$

Using the data structure DBM [8] to represent the symbolic state, the operator sp can be evaluated effectively.

For the enabled event $InterruptVector$, the successor of $globalstate$ is calculated by the algorithm `successorOfIV(globalState)`. The $Interrupt\text{-}Seq$ is a record of the interrupt sequence being added into or removed from the stack. We use the $InterruptSeq$ to record the execution order of tasks in the stack.

```
InterruptHandler_i := InterruptVector[p];
Create a stack member stMember of InterruptHandler_i;
Push stMember into the stack;
topPRIstack := the priority of InterruptHandler_i;
Execute the first task of InterruptHandler_i;
Record information of task_1 in InterruptHandler_i to
InterruptSeq;
return <(l,D), InterruptVector, stack>;
```

For an enabled event corresponding to a task of the interrupt handler in the stack, the successor w.r.t. the interrupt handler is calculated by the following algorithm `successorOfStack(globalState)`.

```
InterruptHandler_i:= the top member of the stack;
TASK_k := the current task in InterruptHandler_i;
if (the mode of TASK_k is wait)
{   change the mode of TASK_k to begin;
    record new TASK_k to InterruptSeq;
} else if (the mode of TASK_k is begin)
{   Change the mode of TASK_k to end;
    Record new TASK_k to   InterruptSeq;
    if (TASK_k is not the last task in
              InterruptHandler_i)
    { Set the current task be the one next to TASK_k;
```

```
        Set the mode of the current task to wait;
      } else
      { Remove InterruptHandler_i from the stack; }
  }
  return <(l,D), InterruptVector, stack>;
```

4.3 The Model Checking Algorithm

The model-checking algorithm exhaustively explores the state space of the interrupt processing system, using the depth first search method. Initially, the $InterruptVector$ and $stack$ are empty, the initial global state is $globalstate_0 = <(l_0, D_0), InterruptVector, stack>$. This method guarantees that all permutation of occurrence sequences of interrupt sources are considered.

```
Unexplored :={globalState0}
while (Unexplored ≠ NULL) do
{   select a  global state curState from Unexplored;
    remove curState from Unexplored;
    for each event e in enabled(curState)
    { if (e is an edge leaving from curState)
        add successorOfTA(curState) to Unexplored;
      else if (event is InterruptVector)
        add successorOfIV(curState) to Unexplored;
      else if (event is stack)
        add successorOfStack(curState) to Unexplored;
    }
}
```

We only record generated symbolic states into the reachability graph when the stack is idle. When removing $InterruptHandler$ from the $stack$ by the algorithm `successorOfStack(globalState)`, we retrieve the execution order of tasks occurred in the $stack$ from $InterruptSeq$, and check whether every interrupt has been processed within its timeout period using the linear programming technique. If so, we determine the containment relation and record the symbolic state into the reachability graph if necessary. Otherwise, a counter example of overtime schedule can be retrieved from $InterruptSeq$. The following algorithm clarifies this method.

```
Let GRAPH be the generated reachability graph;
Let the current global state be
 <(l,D), InterruptVector, stack> with InterruptSeq;
if(stack is empty)
{  Retrieve a task execution sequence from InterruptSeq;
   for each InterruptHandler_i occurs in InterruptSeq
   {   executionTime:=sumOf(taskExecutionTime);
       if(lowerBound(InterruptHandler_i)⩽executionTime
             ⩽higherBound(InterruptHandler_i))
```

```
        {   if (there exist a node (l,D') in GRAPH such
                    that (l,D)⊆(l,D'))
            { add InterruptSeq to (l,D') in GRAPH;}
            else
            { add (l,D) with InterruptSeq  to GRAPH;}
        }else
          report InterruptSeq as a counter example;
    }
}
```

However, the state-space may be infinite. We solve this problem based on the following observation. An interrupt processing system contains several regular interrupt sources with different frequencies. It implies that there exists a periodical interrupt sequence of these regular interrupts. As the priorities of regular interrupts are lower than that of contingency interrupts, to guarantee that these regular interrupts are handled in time, all the interrupts in the $stack$ must be finished within some time period. This period is proportional to the maximal interval of the regular interrupts. If the stack keeps busy for a time period longer than this period, we can already conclude that the system under-check is unable to handle all interrupts within time limits. When the stack becomes idle, the symbolic system state is just a state for timed automata. There are finite number of such states if an appropriate equivalence relation is applied. We add an auxiliary timer to check whether the stack is always busy. The termination of our algorithm is guaranteed.

5 Examples

Here, we present a simplified example taken from an aerospace control system. It has two regular interrupt sources: the 2s-interrupt for the satellite-ground communication and the 0.5s-interrupt for the system information backup. It also has two contingency interrupt sources: the navigation interrupt and the mode-switch interrupt with the highest priority. The corresponding timed automata model of these interrupt sources is shown in Figure 3.

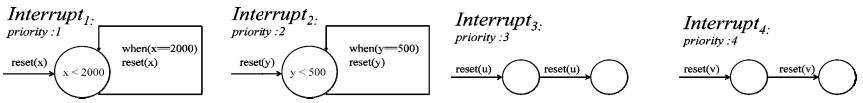

Fig. 3. The timed automata model

Each interrupt contains several to-do tasks. The maximum execution-time of these interrupt-handlers are $500ms$, $250ms$, $100ms$, and $100ms$ respectively. The system requires that each regular interrupt must be handled within 80 percent of its time period. Our algorithm checks this model by exploring the state-space exhaustively. The algorithm finds that when an execution of the regular interrupt handler is nested

by two contingency interrupts continuously, it cannot be finished before the time deadline.

We also checked that if the execution time upper-bound of the second interrupt handler is $200ms$, all interrupts could be accomplished within its time limits.

6 Conclusion

In this paper, we present a technique to model and check interrupt processing systems using extended timed automata. The state space of the model system can be explored exhaustively to check whether the schedule of interrupt processing can meet the time requirements. A simplified real example is used to demonstrate how our method works.

Acknowledgment. This work is supported by the National Natural Science Foundation of China (No.91118002).

References

1. Silberschatz, A., Galvin, P.B.: Operating System Concepts (1998)
2. Walker, W., Cragon, H.G.: Interrupt Processing in Concurrent Processors. IEEE Computer 28(6), 36–46 (1995)
3. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
4. Brylow, D., Palsberg, J.: Deadline Analysis of Interrupt-driven Software (2003)
5. Alur, R., Dill, D.: A Theroy of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
6. Henzinger, T.A.: The Theory of Hybrid Automata. LICS, 278–292 (1996)
7. Zhao, J., Li, X., Zheng, T., Zheng, G.: Removing Irrelevant Atomic Formulas for Checking Timed Automata Efficiently. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 34–45. Springer, Heidelberg (2004)
8. Dill, D.L.: Timing Assumptions and Verification of Finite-state Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)