软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(9):2212−2230 [doi: 10.13328/j.cnki.jos.004713]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

中断驱动系统模型检验*

周筱羽[1,2], 顾 斌[3], 赵建华[1,4], 杨孟飞[5], 李宣东[1,4]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 软件学院,江苏 南京 210093)

[3](西北工业大学 计算机学院,陕西 西安 710072)

[4](南京大学 计算机科学与技术系,江苏 南京 210023)

[5](中国空间技术研究院,北京 100094)

通讯作者: 赵建华, E-mail: zhaojh@mail.nju.edu.cn

摘 要: 针对一类中断驱动系统提出了一种建模和模型检验的方法.该系统通常由中断处理程序和操作系统调度的任务组成,前者由中断源触发后处理中断事件,后者则负责处理系统的日常任务以及某些中断处理事件的后续处理.因为这类系统是实时控制系统,对中断事件的处理需要在规定时间内响应并完成,否则可能造成严重的系统失效.为了帮助系统设计人员在系统设计过程中应用模型检验技术来提高系统的正确性,首先确定了此类系统中与时序性质相关的系统要素(包括系统调度任务、中断源、中断处理程序)和相关参数,并要求设计人员在设计阶段明确指出这些要素的参数.然后,提出了将这些要素和参数自动转化为形式化模型的方法:使用时间自动机对中断事件进行建模,使用中断向量表和 CPU 处理栈对中断处理过程进行建模.对于得到的形式化模型,给出了针对中断处理超时错误的检测方法,并在此基础上给出了针对共享资源的完整性、子程序原子性的检验方法.

关键词: 中断驱动系统;模型检验;超时检测

中图法分类号: TP311

中文引用格式: 周筱羽,顾斌,赵建华,杨孟飞,李宣东.中断驱动系统模型检验.软件学报,2015,26(9):2212−2230. http://www.jos.org.cn/1000-9825/4713.htm

英文引用格式: Zhou XY, Gu B, Zhao JH, Yang MF, Li XD. Model checking technique for interrupt-driven system. Ruan Jian Xue Bao/Journal of Software, 2015,26(9):2212−2230 (in Chinese). http://www.jos.org.cn/1000-9825/4713.htm

Model Checking Technique for Interrupt-Driven System

ZHOU Xiao-Yu[1,2], GU Bin[3], ZHAO Jian-Hua[1,4], YANG Meng-Fei[5], LI Xuan-Dong[1,4]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Institute of Software Engineering, Nanjing University, Nanjing 210093, China)

[3](School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an 710072, China)

[4](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[5](China Academy of Space Technology, Beijing 100094, China)

Abstract: In this paper, an approach is proposed to model and verify a class of interrupt-driven systems. An interrupt-driven system usually consists of interrupt handlers and system-scheduling tasks. When an interrupt occurs, the corresponding interrupt-handler executes in response. The operating system schedules a set of tasks to deal with routine events and certain post-processing of some interrupts. In the real-time control system, it is important that interrupts are handled within their specific deadlines, otherwise, it may cause catastrophic system failures. In order to improve the reliability of interrupt-driven systems, model checking technique is introduced to the design and development process. Through analyzing numerous systems, the major system elements (including system scheduling tasks, interrupts and

* 基金项目: 国家自然科学基金(91118007); 国家高技术研究发展计划(863)(2011AA010103)
收稿时间: 2013-10-16; 修改时间: 2014-04-18; 定稿时间: 2014-08-08

their handlers) and their parameters relevant to time-related failures are identified. When these parameters are specified by system designer in the design process, formal models can be constructed by the modeling method in this paper: The interrupt source is modeled as timed automata. The execution processes of interrupt handlers are modeled by the interrupt vector and the CPU process stack. A model-checking algorithm is provided to check the above formal model whether interrupt handlers can be executed within their response deadlines. Moreover, a variation of this algorithm is developed to check properties of the integrity of shared resources and the atomicity of subprograms.

Key words: interrupt-driven system; model checking; deadline detection

实时嵌入系统被广泛应用于安全关键系统中,如航空航天控制系统、铁路交通控制系统、医疗辅助软件等.这类系统一旦发生错误,将会造成重大的生命及财产损失.因此,中断驱动实时处理系统的可靠性保障至关重要.

在中断驱动系统中[1,2],中断事件触发后需要在给定时间范围内被响应完成,而且高优先级中断可以打断低优先级中断的执行,形成中断嵌套.中断的多重嵌套,可能导致资源竞争、数据冲突、运行超时等多种问题.由于中断事件的发生顺序和发生时间是不确定的,这将导致不同的处理顺序,中断系统中包含的某些设计错误只有在特定的处理顺序之下才会显现出来.我们把这类错误称为中断处理系统的时序相关错误.时序相关错误是中断驱动控制系统中常见但难以处理的问题,根据对中断驱动控制系统的长期设计经验,我们发现,时序相关问题主要可以分为以下几类:

(1) 中断处理时间过长引起的问题:中断多重嵌套可能导致低优先级的中断事件无法在规定时间内完成响应处理;此外,中断超时还可能进一步导致其他时序相关错误;

(2) 中断引起的通信异常问题:例如在通信过程中,设备 a 每隔 t 时间向设备 b 发送一次信息包,如果在 b 接受信息的过程中被其他中断挂起,就可能导致 b 不能及时发送应答信息到设备 a,引发本次通信失败,造成数据丢失;

(3) 中断引起的竞争问题:两个不同优先级中断的处理程序共享同一个公共资源(如共享变量、寄存器、内存单元等)时,可能导致读-写或者写-写访问冲突.

在不同时间点触发的中断可能导致不同的系统行为.而中断驱动系统的运行环境一般非常复杂,不确定因素很多.测试者无法测试所有可能发生的中断事件序列.有时即使指定了中断事件的发生顺序,也无法测试中断事件在不同时刻发生而引发的不同系统行为,因此在中断驱动系统的设计过程中,测试开销极大且效率低下,对中断驱动系统进行完全的测试是无法实现的,通常仅能覆盖中断驱动系统的状态空间中的部分情况.

模型检验技术[3]的出现,使得全面分析和验证中断处理系统的系统行为成为可能.通过穷尽枚举中断驱动系统的状态空间,人们可以分析计算系统中某一中断事件是否一定能够在规定时间内完成响应.为了描述系统中的时间行为,模型检验技术通常使用时间自动机[4]或混成自动机[5]对系统进行建模.但是时间自动机模型中的时钟变量以相同速度向前演化,无法描述中断驱动系统中的中断处理过程被挂起后的时间信息;而混成自动机中各个变量的约束相当复杂,导致很多性质不可判定.即使对于可判定的性质,针对混成自动机的模型检验算法的时间复杂性也很高.因此,本文引入了中断时间自动机来对中断驱动系统进行建模,模拟中断触发及其响应过程,并给出了中断驱动系统的模型检验算法.

应用模型检验技术的另外一个问题是,该技术要求工程师首先使用形式化模型对系统进行建模.但是熟练掌握形式化建模技术(比如时间自动机建模)需要很长的学习过程,而且得到的模型比较复杂,难以判断形式化模型和实际系统之间的一致性.为了解决这个问题,我们分析了中断驱动控制系统中各个组成要素,确定了每个要素中和时序性质相关的参数,并给出了由这些参数自动建立形式化模型的方法.这个技术使得设计工程师可以方便准确地输入参数并自动得到正确的形式化模型,有效降低了模型检验技术的应用难度.

本文第 1 节简单介绍中断驱动系统的基本要素,并讨论如何自动建立形式化模型,包括使用中断时间自动机对中断源建模,使用中断向量表和 CPU 处理栈对中断处理过程建模,并给出中断驱动系统模型的总体结构.第 2 节介绍中断驱动系统模型验证的具体方法,分别给出如何计算中断时间自动机、中断向量表以及 CPU 处

理栈相关事件的后继状态的计算方法,并给出检测系统模型的中断处理超时问题的模型检验算法.随后,本文进一步给出该可达性算法的变体,能够分别检验资源竞争问题和程序原子性问题.第 3 节给出一个实例,并使用本文的算法进行形式化建模和验证.最后一节是相关工作的比较和对全文的总结.

1 中断驱动系统模型

1.1 系统调度任务加中断处理的中断驱动系统简介

我们研究的中断驱动系统使用操作系统调度任务加中断处理的软件体系结构,这个体系结构将需要定时执行的处理任务分为若干个功能相对独立的模块.操作系统周期性地按照固定次序调度执行各个任务,每个任务需要在规定的时间片内完成.中断发生后,当前任务被挂起并执行相应的中断处理程序;当中断执行结束后,继续执行当前处于挂起状态的任务.当然,中断处理程序的执行过程也可能被更高级的中断打断.

中断源可以分为周期中断和偶发中断.周期中断每隔固定时间单位自动触发.周期中断通常用于维护系统的状态、执行系统信息备份和定时通信等例行任务.每类周期中断被赋予一个优先级.通常,周期越长的中断其优先级越低.偶发中断描述随机触发的中断事件,如接受人工命令而修改内存数据、调整系统运行模式等.偶发中断的重复频率无法预期,通常由实时系统中的外部事件触发.偶发中断的优先级通常高于周期中断优先级.

由于中断嵌套的原因,在某一时刻可能存在多个处于执行过程中的任务或中断处理程序.这些处理程序按照优先级高低自顶向下记录在CPU栈中:栈顶的处理程序处于执行状态,而栈中其他处理程序则处于挂起状态.当中断源触发后,中断向量表被相应地置位.如果该中断的优先级高于正在执行的中断的优先级,该中断的处理程序将被压入 CPU 栈并立即开始执行,同时,中断向量表中的标识位被清零.在栈顶的中断处理程序执行完成后,CPU 栈中以及中断向量表中具有最高优先级的中断处理程序将获得执行权力.

1.2 和时序性质相关的系统要素信息

为了方便软件工程师在软件开发过程中使用模型检验技术,同时避免使用复杂的形式化模型来建模,我们为中断驱动系统中的各个要素设立了一个模板,要求工程师按照模板填写相应的信息,然后,通过工具自动化地生成模型.这样做一方面简化了建模过程并提高了模型的正确性,另一方面也促使工程师在软件开发过程中系统化地考虑与时序相关的信息.在采用任务加中断体系结构的系统中,与时序相关的设计要素包括任务的调度信息、中断事件的信息和中断处理程序的信息.

对于每个系统任务,我们要求给出如下的五元组信息:⟨*bcet*,*wcet*,*upbnd*,*period*,*offset*⟩,其中,

1. *bcet* 和 *wcet* 分别表示任务单独执行时的最短执行时间和最长执行时间;
2. *upbnd* 表示在实际运行中(即,在可能被其他中断打断的情况下)的最长允许执行时间;
3. *period* 表示任务的调度周期(所有系统任务的周期都相等);
4. *offset* 表示该任务的开始时刻相对于周期开始时刻的偏移量.

操作系统以 *period* 为周期调度执行这些任务,各个任务在每个周期中偏移量为 *offset* 的时间点开始执行.这个任务至少需要 *bcet*、至多需要 *wcet* 的 CPU 时间来完成执行.在实际运行时,任务的执行过程可能被打断,因此,从任务开始执行到执行结束的时间长度通常会超过 *wcet*.但是系统需求规定,这个时间长度最多不超过 *upbnd*.

对于每类中断事件及其中断处理程序,要求给出如下信息:⟨*period*,s_1,s_2,*priority*,*bcet*,*wcet*,*upbnd*⟩.其中,

1. *period* 表示这个中断事件发生的周期,如果这个中断不是周期性的,那么 *period* 表示该类型中断事件的两次中断事件之间的时间间隔;
2. s_1 和 s_2 分别表示在系统开始运行后最少 s_1 时间之后、最多 s_2 时间之前中断事件才会发生;
3. *priority* 表示该中断的优先级;
4. *bcet*,*wcet* 和 *upbnd* 的含义和任务信息中的含义相同,分别表示中断事件单独执行时的最短时间、最长时间和系统需求规定的实际执行时的最长允许执行时间.

根据这些信息,我们就可以生成系统模型,然后针对超时问题进行模型检验.本文的中断驱动系统模型中包含:由时间自动机建模的中断事件和任务开始事件、包含响应和执行时间信息的中断处理程序及其子处理程序模型,记录中断事件触发情况的中断向量表以及模拟中断处理过程的 CPU 处理栈模型.

例 1:以一个简单的中断驱动系统为例,需要系统的开发设计人员从系统中提取出表格化的时间相关要素如下:

这个中断驱动系统中包含 3 个任务、一个周期中断和一个偶发中断.表 1 描述了一组由 3 个任务事件依次执行的任务序列,这 3 个任务的调度周期都为 200 个时间单位,且分别在 0,100,160 个时间单位时开始执行.例如,其中第 3 个任务 T_3 在 160 个时间单位时开始执行,其执行中占用 CPU 处理器的执行时间在 24~32 个时间单位之间,且开始执行后必须在 40 个时间单位内执行完成.表 2 中抽象了两个中断事件.其中,周期中断 I_1 的优先级较低(为 1),它的中断周期为 20 个时间单位,在 0~8 个时间单位内开始执行,其单独执行所需的时长在 1~2 个时间单位之间,且必须在 8 个时间单位内响应完成.

Table 1 Task list

表 1 任务列表

任务	*bcet*	*wcet*	*upbnd*	*period*	*offset*
T_1	60	80	100	200	0
T_2	36	48	60	200	100
T_3	24	32	40	200	160

Table 2 Interrupt list

表 2 中断列表

中断	*period*	s_1	s_2	*priority*	*bcet*	*wcet*	*upbnd*
周期 I_1	20	0	8	1	1	2	8
偶发 I_2	0	0	200	2	1	2	4

1.3 中断时间自动机和中断源建模

为了对中断源以及任务开始事件建模,我们以时间自动机为基础进行扩展,定义了中断时间自动机.

1.3.1 中断时间自动机

时间自动机(timed automaton)模型是对实时并发系统进行建模的重要工具.时间自动机在传统的有穷状态自动机上添加了一组实数值时钟,用于模拟实时系统中的时间行为.这些附加的时钟可以描述自动机状态转换间的各种时间限制.根据我们对中断驱动控制系统的分析,虽然时间自动机不能描述中断处理程序占用 CPU 并执行的时间信息,它的描述能力足以描述中断事件发生的时间约束.因此,本文对时间自动机进行扩展,定义了中断时间自动机,用于对中断源和系统任务的开始事件进行建模.

首先给出中断时间自动机的定义.令 C 为一组时钟变量,在 C 上的一个时钟赋值 u 是从 C 到实数域的映射.对于实数 $t\in\mathbb{R}$,我们用 $u+t$ 表示将 C 中每一个时钟变量映射为 $u(x)+t$ 的时钟赋值.$G(C)$表示 C 上的时间卫式,它是一组形如 $x\sim n$ 的原子公式的合取,其中 $x\in C,\sim\in\{<,\leqslant,\geqslant,>\}$,且 n 为整数.$B(C)$表示 C 上的时钟限制,它是一组形如 $x-y\sim n$ 的公式合取,其中,$x,y\in C\cup\{0\},\sim\in\{<,\leqslant,\geqslant,>\}$,且 n 为整数.对于任意时钟集合 C,$G(C)\subseteq B(C)$.

中断时间自动机 ITA(interrupt timed automaton)是五元组$\langle L,l_0,C,E,\beta\rangle$,其中,

1. L 是一组状态的有限集合;
2. $l_0\in L$ 是初始状态;
3. C 是一组时钟变量集合;
4. $E\subseteq L\times G(C)\times 2^C\times L$ 是一组转换的有限集合,其中,对于转换 $e=(l,g,r,l')$,$g\subseteq G(C)$是 e 的时间卫式,$r\in C$ 是被 e 重置的时钟集合,转换 e 从 l 出发到达 l';
5. β是从 E 到中断事件的映射,$\beta(e)$表示转换 e 触发的中断事件.如果转换 e 不触发中断事件,$\beta(e)=\varnothing$.

设有 E 中的转换 $e=(l,g,r,l')$,那么从状态 l 出发,经过转换 e 可以到达状态 l',记为 $l\xrightarrow{e}l'$.从状态 l 出发(不

考虑时间约束)可能发生的转换集合记为 $enabled(l)=\{e \mid ITA$中存在状态l',使得$l \xrightarrow{e} l'\}$.如果 $e \in enabled(l)$,我们称转换 e 在状态 l 上可能发生.

中断时间自动机 *ITA* 的一个具体状态是二元组(l,u),其中,$l \in L$,u 是 C 上的时钟赋值.时间自动机的演化包括时间流逝和具体转换两种方式:

1. 时间流逝: $(l,u) \xrightarrow{t} (l,u+t)$,其中,$t>0$;
2. 具体转换: $(l,u) \xrightarrow{e} (l',u')$,其中,$e=(l,g,r,l') \in E$ 满足下列条件:
   a) 对 g 中的每一个时间卫式 $x \sim n, u(x) \sim n$;
   b) 对每一个时钟 $x \in r, u'(x)=0$;并且对 $C-r$ 中的每个时钟 $x, u'(x)=u(x)$.

如果存在 $u''(x)$,使得 $(l,u) \xrightarrow{e} (l',u'')$ 且 $(l',u'') \xrightarrow{t} (l',u')$,可以记为 $(l,u) \xrightarrow{e,t} (l',u'')$.中断时间自动机网络的一个执行记为如下序列:

$$\alpha=(l_0,u_0) \xrightarrow{t_0} (l_0,u_0') \xrightarrow{e_1,t_1} (l_1,u_1) \xrightarrow{e_2,t_2} \ldots \xrightarrow{e_n,t_n} (l_n,u_n).$$

这个执行的初始全局状态是(l_0,u_0),在 t_0 个时间单位后,状态到达 (l_0,u_0');然后执行转换 e_1,再过 t_1 个时间单位,自动机到达状态(l_1,u_1);如此继续,最终到达状态(l_n,u_n).转换序列 $p=e_1,e_2,\ldots,e_n$ 是α的执行路径,我们也称α是沿着路径 p 的执行.

由于时钟的取值在实数范围内,中断时间自动机的具体状态空间是无限的.根据在时间卫式中出现的最大常量来划分等价关系,我们可以将状态空间约减为包含有限数量的等价类集合.然而,这种方法仍然会生成大量的等价类.为了优化模型检验的过程,可达性分析通常使用时间区域(time zone)而不是细小的等价类来表示具体状态的集合.一个时钟区 $D \in B(C)$描述了时钟取值及它们之间的差值的上下限.中断时间自动机的符号状态(l,D)表示了如下一组具体状态的集合:

$$\{(l,u) \mid u \text{ 满足 } D \text{ 中所有时钟约束}\}.$$

操作符 *sp* 用于计算从符号状态(l,D)开始,经过转换 e 可以到达的符号后继,记为 $sp((l,D),e)$.$sp((l,D),e)$是从(l,D)中某个具体状态出发,执行转换 e 以及时间流逝后可以到达的具体状态集合,即,$sp((l,D),e)$表示集合:

$$\{(l',u') \mid \exists t \in \mathbb{R}, \exists (l,u) \in (l.D) \cdot ((l,u) \xrightarrow{e,t} (l',u'))\}.$$

使用差值范围矩阵(difference bound matrix,简称 DBM)[6]描述符号状态,可以高效地完成操作符 *sp* 的相关计算.

1.3.2 任务开始事件和中断事件的建模

我们研究的中断驱动系统采用系统调度任务加中断事件的体系结构,系统任务的开始事件可以使用中断时间自动机建模.如果系统包含了 n 个系统调度任务,那么我们可以用一个包含 $n+1$ 个状态的中断时间自动机来对任务开始事件建模.每个中断事件可以用一个包含两个状态的中断时间自动机模型进行建模.每个时间自动机都有一个本地时钟,初始时值为 0.在这些时间自动机中,初始状态表示中断事件未触发,或系统任务未开始运行.中断时间自动机中的转换代表一个中断事件被触发或是一个系统任务开始执行.本地时钟被用来控制中断发生的周期、间隔或者系统任务的周期以及偏移量.

如前所述,每个任务的信息是五元组⟨*bcet*,*wcet*,*upbnd*,*period*,*offset*⟩,且所有任务的周期 *period* 是相同的.在建模过程中,我们只考虑任务的开始事件,任务的执行过程将在 CPU 处理栈模型中处理.因此,只需要考虑其中的 *period* 和 *offset*.我们把全部任务的开始事件使用同一个包含多个状态的、循环执行的中断时间自动机来建模.这个自动机有一个时钟 z,用于控制任务调度周期和任务开始的偏移量.假设系统中有 n 个任务 $t_1,t_2,\ldots,t_n$,它们的周期为 *period*,而各个任务的偏移量分别是 $o_1,o_2,\ldots,o_n$,那么相应的时间自动机有 $n+1$ 个状态:$l_0,l_1,\ldots,l_n$,其中的 l_0 为初始状态.对于每个 $i(i<n)$,从 l_i 到 l_{i+1} 有一个转换,表示任务 t_{i+1} 开始执行的事件,该转换上的时钟约束是 $z==o_{i+1}$.从 l_n 到 l_1 有一个转换,该转换的时钟约束是 $z==period$,且需要重置 z 的时钟值,表示整个周期执行完成并开始新一轮循环.

例 2:图 1 是对表 1 中给出了任务事件集合进行建模的时间自动机 ITA_1.中断自动机时钟 x 初始值为 0,根据表 1 中给出的任务事件的 *period* 和 *offset*,时间自动机以 200 个单位时间为周期,在 0 时刻,100 个单位时间和 160

个单位时间时分别执行 T_1,T_2,和 T_3,200 个单位时间时重置时钟 x 并开始执行 T_1.如此循环,即:

$$\beta(e_2)=T_2,\beta(e_3)=T_3,\beta(e_1)=\beta(e_4)=T_1.$$

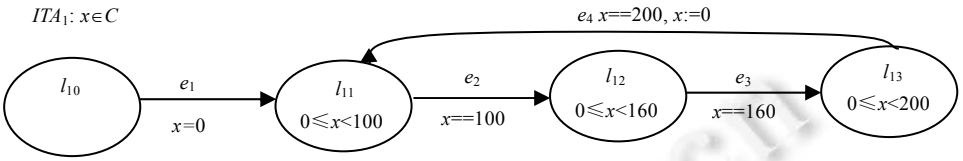

Fig.1 ITA model of the task sequence

图 1 任务序列的中断时间自动机模型

如前所述,对于每个周期性中断(及相关中断处理程序),工程师给出如下的信息:

$$\langle period,s_1,s_2,priority,bcet,wcet,upbnd\rangle.$$

因为中断时间自动机仅仅对中断事件建模,对中断处理程序及其运行情况在 CPU 栈模型中进行处理,因此只需要考虑 $period$,s_1 和 s_2.我们使用包含一个时钟和两个状态的中断时间自动机对中断事件进行建模,通过时钟来控制中断事件的初次发生时间和中断间隔.图 2 描述周期中断 I 的 ITA 模型,其中,$x\in C$ 是自动机的中断时钟,初始时值为 0.e_1 和 e_2 都代表这个中断的发生,也就是$\beta(e_1)=\beta(e_2)=I$.e_1 代表中断事件的第 1 次发生,有时间约束 $s_1\leqslant x\leqslant s_2$.转换 e_2 的时间约束 $x==period$,并且将 x 重置为 0.

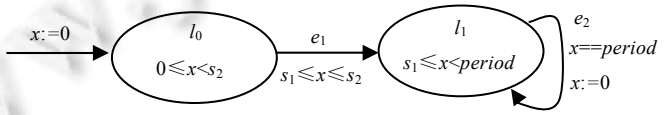

Fig.2 ITA model of the periodic interrupt

图 2 周期中断的中断时间自动机模型

系统中的非周期中断称为偶发中断,通常具有较高的优先级,且发生的次数较少.在建模时,我们会要求工程师给出七元组$\langle period,s_1,s_2,priority,bcet,wcet,upbnd\rangle$,其中的 $period$ 值表示该中断两次触发之间的最小间距.我们还要求工程师设定这个中断最多发生次数 f.根据对以往设计错误的分析表明,f=3 时就能够发现绝大部分的设计错误.我们可以用图 3 中的中断时间自动机进行建模,其中,$x\in C$ 是自动机的中断时钟,初始时值为 0.变量 v 表示中断已经发生的次数,初始值为 0,当 $v<f$ 时,允许转换 e_1 发生,即,偶发中断事件被触发,$\beta(e_1)=I$.此时有时间约束 $s_1\leqslant x\leqslant s_2$,变量 v 值加 1,并且将 x 重置为 0.当 $v=f$ 时,执行转换 e_2 到达状态 l_1,$\beta(e_2)=\varnothing$,当前中断处理系统不再触发该偶发中断事件.

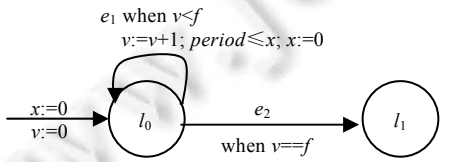

Fig.3 ITA model of the contingency interrupt

图 3 偶发中断的中断时间自动机模型

例 3:根据表 2 中给出的中断事件的时间信息,周期为 20 个时间单位的周期中断 I_1 和最多运行触发 3 次的偶发中断 I_2 分别被建模为图 4 中的中断时间自动机 ITA_2 和 ITA_3,其中分别包含时钟 y 和 z,初始时为 0,ITA_3 中包含变量 v 用于记录中断事件 I_2 已经发生的次数.

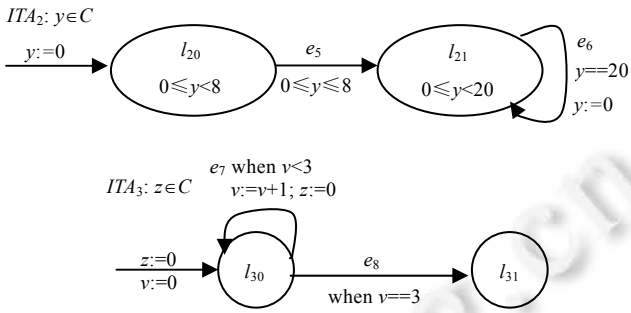

Fig.4 Example of the ITA model of interrupts

图 4 中断事件的中断时间自动机模型案例

1.4 中断向量表

我们在系统模型中使用一个向量 *vector* 来模拟实际系统中的中断向量表,它记录已被触发但尚未开始处理的中断事件.在初始状态下,对于所有中断 *I*,*vector*(*I*)=false.在模型检验过程中,当中断时间自动机中的转换 *e* 发生时,如果$\beta(e)$对应于一个中断 *I*,且 *vector*(*I*)已经为 true(也就是上一个中断还没有被响应),那么表示系统可能会发生中断丢失错误,模型检验工具将报告相应的错误;如果 *vector*(*I*)尚未置位,那么模型检验算法将 *vector*(*I*)标记为 true.当 *I* 对应的中断处理程序开始运行时,*vector*(*I*)被置为 false.

1.5 中断处理程序模型和CPU处理栈模型

中断驱动系统处理器根据中断向量表 *vector* 中的置位信息来响应中断事件请求,优先执行具有较高优先级的中断处理程序.在响应较高优先级的中断事件时,可以打断当前正在运行的较低优先级中断程序.当高优先级的中断处理程序运行结束后,原来被打断的中断处理程序可以恢复执行.

在模型检验时,我们不仅关心是否能够确保各个中断事件得到及时处理,也需要保证子处理过程能否及时完成.比如,系统在和串口通信时不能被其他中断程序长时间打断,否则会造成数据丢失.因此,我们把中断处理程序看作多个子程序组成的序列.这样做的另外一个好处是:我们可以更加精确地描述中断处理程序在哪个时间段对特定共享资源进行访问.中断处理程序的一个子程序被描述成一个五元组$\langle b,w,upbnd,Rr,Rw\rangle$:

1. $b,w\in\mathbb{N}$分别是该子程序在单独执行、不被其他中断打断的情况下所需的最短执行时间和最长短执行时间,即,*BCET* 和 *WCET*;
2. $upbnd\in\mathbb{N}$是该子程序允许的最长响应时间,也就是从该子程序开始执行到它执行结束的时间距离.其中,$0\leqslant b\leqslant w\leqslant upbnd$;
3. *Rr*,*Rw* 分别记录了该子程序读/写的共享资源的集合.

系统调度的任务的处理过程也类似地描述为一组子过程的序列.

例 4:例 1 中的中断驱动系统模型中给定任务 T_1,T_2,T_3,和中断 I_1,I_2 对应的中断处理程序模型分别为表 3 中 H_1,H_2,H_3,H_4,H_5.在后文中,为简化案例的分析过程,这里每个中断处理程序仅包含一个具有相同时间约束条件的子处理程序.

Table 3 Interrupt handlers

表 3 中断处理程序

中断处理程序	*b*	*w*	*upbnd*	*Rr*	*Rw*
H_1	60	80	100	–	–
H_2	36	48	60	–	–
H_3	24	32	40	–	–
H_4	1	2	8	–	–
H_5	1	2	4	–	–

在本文中,我们采用 CPU 处理栈 *stack* 来描述中断处理过程的运行情况,将正在执行的中断处理程序及其子处理程序序列加入栈中,并跟踪记录中断处理程序的执行状态.该栈中的每个成员是一个三元组$\langle H_k,mode,h_i\rangle$,其中:$H_k$ 是一个中断处理程序;而 h_i 是 H_k 的一个子过程;*mode* 的值集合是{*started*,*suspended*,*finished*},用于标识当前正在执行的中断子处理程序 h_i 的处理状态,其中,

1. *started*:表示 h_i 开始执行,且尚未处理完成;
2. *suspended*:表示系统中有高优先级中断事件触发,h_i 被挂起,栈中当前执行的是具有较高优先级的中断处理程序 H_j;
3. *finished*:h_i 执行结束.

中断向量和 CPU 栈联合起来就模拟了中断驱动系统的运行方式.假设当前中断向量为 *vector*,栈顶元素是$\langle H_k,mode,h_i\rangle$,系统的运行规则如下:

1. 如果存在中断 I 满足 $vector(I)$=true 且 I 的优先级高于 H_k,下一个事件必然是挂起当前执行的中断处理程序 H_k,并执行 I 对应的处理程序.因此,当前栈顶元素的 *mode* 被设置为 *suspended*,同时加入新的栈元素$\langle H_I,started,h_0\rangle$,其中,$H_I$ 是 I 的中断处理程序,h_0 是 H_I 的第一个子过程,新栈顶元素的 *mode* 值被设置为 *started*;
2. 如果不存在这样的中断,且当前栈顶元素的 *mode* 值为 *started*,可能发生的和 CPU 有关的事件是当前子过程运行结束.相应地,栈顶元素的 *mode* 被改变为 *finished*;
3. 如果没有更高级中断事件发生,且当前栈顶元素的 *mode* 值为 *finished*,那么:
   a) 如果 h_i 不是 H_k 的最后一个子过程,则相应的事件是下一个子过程 h_{i+1} 开始执行.相应的处理是当前栈顶元素出栈,$\langle H_k,started,h_{i+1}\rangle$入栈,新栈顶元素的 *mode* 值被设置为 *started*;
   b) 如果 h_i 是 H_k 的最后一个子过程,则相应的事件是本中断处理程序结束,需要恢复执行之前被挂起的处理程序.相应的处理是栈顶元素出栈,且把新的栈顶元素的 *mode*(必然是 *suspended*)改变为 *started*.如果 H_k 是最低级的处理程序,那么 CPU 为空闲.

当系统按照上述规则运行时,中断事件和系统任务开始事件仍然会不断发生.在模型系统演化过程中,当前状态所包含的 CPU 处理栈中,栈顶为当前正在执行的最高优先级中断处理程序,栈中其余成员依次包含了按中断优先级递减排列的、被挂起的低优先级中断处理程序和它们的当前处理状态.同时,上面的运行过程需要遵循相应的时间约束如下:

1. 当一个子过程结束时,从该子过程开始到结束期间占有的 CPU 时间必须不小于该子过程的 *BCET*、不大于该子过程的 *WCET*;
2. 当一个子过程被打断时,从该子过程开始到当前时刻之间它占有的 CPU 时间必须不大于该子过程的 *WCET*.

这些时间约束涉及到了多个已发生事件的开始/结束时刻,用于刻画中断处理程序实际的执行时间.由于这些时间约束的计算方式比较复杂,我们将在模型检验算法部分详述.另外,模型检验程序还需要不断地检验各个中断事件(或者子程序)的开始时刻到当前时刻的时间长度不超过相应的时间上界.

1.6 中断驱动系统的整体模型及其全局状态

图 5 给出了中断驱动系统的整体模型,包括中断时间自动机网络、中断向量表和 CPU 处理栈.对于这样的全局模型,模型的行为由以下信息决定:对中断源建模的中断时间自动机的状态(包括所在位置和时钟取值)、中断向量表的置位信息、CPU 处理栈中的处理程序以及各个处理程序已经使用的 CPU 时间.因此,中断驱动系统模型的一个具体状态是四元组$\langle(l,u),Vector,Stack,T\rangle$,其中,$(l,u)$是 *ITA* 的一个具体状态,*vector* 记录了已经触发的中断事件,*stack* 包含了正在执行(位于栈顶)和被挂起的处理程序,T 记录了 *stack* 中的各个处理程序已经运行的 CPU 时间.初始时,$l=l_0,u=\overline{0}$,*vector* 中无置位,$stack=\varnothing,T=\overline{0}$.

因为时间的取值是实数,具体的全局状态的数量是无穷的,不能直接使用模型检验技术.因此,在模型检验中必须使用符号化表示的状态.但是,CPU 处理栈中各处理程序已经执行的 CPU 时间的计算涉及到处理程序运

行过程中发生的很多事件(包括被更高级中断打断和恢复执行事件)的时刻,相关约束难以用可高效处理的符号化状态表示.因此,我们没有在符号状态中记录这些时间约束,而是使用一个变量 *eventPath* 来记录所有相关的事件.因此,中断驱动系统模型的符号化全局状态记为 *globalState*=⟨(*l*,*D*),*vector*,*stack*⟩,并且在计算后继的过程中,我们使用变量 *eventPath* 来记录相关事件顺序,并依靠 *eventPath* 来确定各个事件的发生是否满足时间约束.当 *stack* 为空且 *vector* 中无置位时,*eventPath* 中的信息无用,此时可以舍弃 *eventPath*,使得全局状态退化为中断时间自动机的状态.我们在模型检验的空间遍历过程中只保存了这些时间自动机的状态,便于高效地完成全局状态比较.根据文献[7]中的等价关系,可以把一个时间自动机的状态空间划分为有穷多个等价类,并可以使用符号状态来表示这些等价类.因此,我们的可达性分析算法仅仅会生成有穷多个符号状态.这样就保证了模型检验算法的状态空间遍历过程是收敛的.

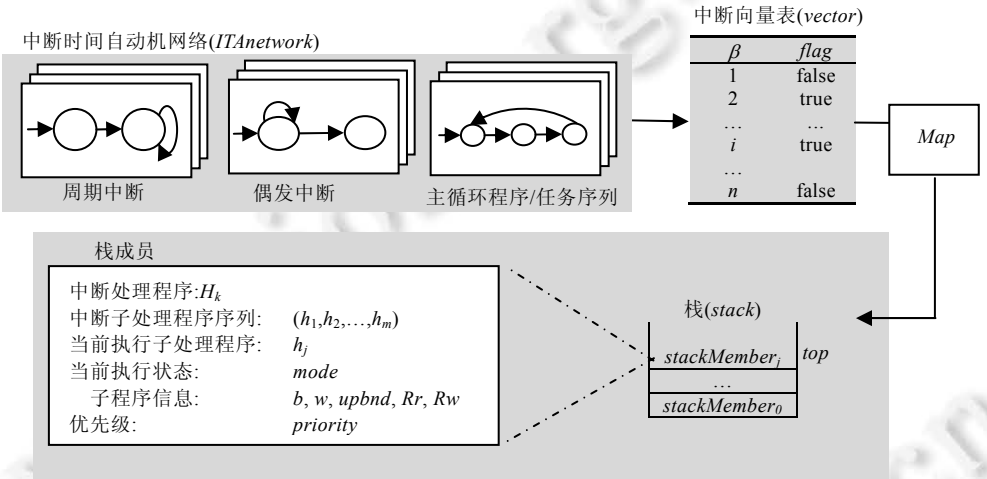

Fig.5 Model of the interrupt-driven system

图 5 中断驱动系统模型

2 中断驱动系统模型验证

本文提出的中断驱动系统的模型检验算法从初始状态出发,不断计算当前状态对应于所有可能发生事件的后继状态,遍历所有可达状态,并分析时间约束、检验当前中断事件序列是否可能出现超时情况或者中断丢失情况.如果有这类错误情况,算法就可以报错,并给出相应的反例供设计开发人员参考.如第 1.6 节中所述,我们的全局状态中没有包含那些与处理程序已执行时间相关的约束,解决办法是:使用一个变量 *eventPath* 来记录相关的事件,并且只有当 *stack* 和 *vector* 都为空时,才把这个全局状态记录到可达状态中.这个方法使得我们可以高效地判断符号状态之间的包含关系.

2.1 事件路径

在枚举遍历状态空间的过程中,我们引入变量 *eventPath*,记录从可达节点 *globalState* 开始经过的事件序列,*eventPath* 是一个 *list*,其成员是六元组⟨*globalState*,*eventList*,*event*,*type*,*constraints*,*bounds*⟩,其中:

1. *eventList* 是当前状态 *globalState* 上可能发生的事件的集合,具体计算方法如图 6 所示;
2. *event* 及 *type* 是 *globalState* 上当前执行的事件及其类型;
3. *constraints* 是该路径成员附加的一组时钟约束,形如 $b \prec \sum(c_j - c_i) \prec w$,其中,$c_i$ 是 *eventPath* 事件路径中 ep_i,ep_j 的发生时刻,$i<j$,$\prec=\{<,\leqslant\}$;b 和 w 取值非负,分别表示 *eventPath* 中一组路径段的执行时间总和的最小时间约束和最大时间约束.这样的约束可以表示栈中中断处理程序及其子程序已经执行的 CPU 时间的约束;
4. *bounds* 包含响应时间相关的界限条件,用于判断中断响应是否超时.如果 ep_i 对应于中断 I 的触发事

件,而 ep_j 是 I 的处理程序 H_i 处理结束的事件,那么在当前 ep_j 的 *bounds* 中包含形如 $c_j-c_i\prec upbnd$ 的时间约束,其中,*upbnd* 表示该中断事件的最大响应时间.中断处理程序的子过程响应时间界限也添加在 *bounds*,添加方法类似.

由于 *eventPath* 中事件是有序发生的,因此对于任意 $ep_i,i>0$,*eventPath*[*i*]的 *constraints* 中包含约束 $0\leqslant c_i-c_{i-1}$.

```
CalculateEnabledEvent(globalState gs)
{
    若 gs=((l,D),vector,stack)且 stack 栈顶下标为 top;
    priorityS=priorityV=−1;
    if (!IsEmpty(stack))
        priorityS=stack[top].priority;
    if (!IsEmptyVector(vector))
        priorityV=FindTheHightestPriority(vector);
    if (priorityV>priorityS)
        AddToEventList(event=H_vector,type=V);
    else
    {   for all leaving edges e from (l,D) in ITA
            AddToEventList(event=e,type=I);
        if (priorityS≥0)
            AddToEventList(event=H_stack,type=S);
    }
}
```

Fig.6 Calculate enabled events of a *globalState*

图 6 计算 *globalState* 上可能发生事件集合的函数

2.2 后继计算

本节讨论如何根据当前状态来计算相应的后继状态.在计算的过程中,我们会把和栈中处理程序相关的约束记录在 *eventPath* 中,并通过这些约束来计算各个事件是否可行,并判断是否可能出现中断处理超时的情况.

2.2.1 计算全局状态上可能发生事件的集合

给定中断驱动系统模型的全局状态 $globalState=\langle(l,D),Vector,Stack\rangle$.令 H_{vector},H_{stack} 分别是 *vector* 中被置位的最高优先级中断对应的处理程序和 *stack* 中栈顶的中断处理程序,*priorityV* 和 *priorityS* 分别是 H_{vector} 和 H_{stack} 的优先级.

globalState 上所有可能发生事件的集合 *EnabledEvent*(*globalState*)计算方法如下:

1. 当 $priorityV>priorityS$,*EnabledEvent*(*globalState*)只包含 H_{vector},即:当前发生的中断具有更高的优先级,必须立刻处理;
2. 当 $priorityV\leqslant priorityS$ 时,*EnabledEvent*(*globalState*)包含下列两类事件:
   a) 如果存在时间自动机的转换 $e\in enabled(l)$,那么 e 在 *EnabledEvent*(*globalState*)中,即,中断源可以并发执行;
   b) H_{stack} 在 *EnabledEvent*(*globalState*)中,即:当前栈顶中断事件和其他已经发生的中断事件相比具有较高优先级,可以继续执行当前栈顶的中断处理程序.

图 6 中算法 *CalculateEnabledEvent*(*globalState gs*)用于计算全局状态 *gs* 上(不考虑时间约束时)所有可能发生的事件集合,并通过函数 *AddToEventList* 加入到集合变量 *eventList* 中.其中,*eventList* 的成员是二元组⟨*event*,*type*⟩,事件 *event* 可以是中断时间自动机中转换 e 或者中断处理程序 H_i;类型 *type* 用于标识事件来源于中断时间自动机(*I*),或是中断向量表(*V*),或是 CPU 处理栈(*S*).在后继状态的计算过程中,将根据不同的类型分别采用不同方法计算.

2.2.2 计算中断时间自动机的转换后继

本节介绍针对中断时间自动机中转换的后继状态计算方法,即,*eventPath* 中当前路径成员 $ep=\langle globalState,eventList,event,type,constraints,bounds\rangle$的 *event* 为 ITA 中转换(即 *type=I* 时)的情况.

图 7 中给出了全局状态 *globalState*=⟨(*l*,*D*),*Vector*,*Stack*⟩上计算时间自动机转换 *e* 的后继算法.

```
CalculateSuccessorOfITA(globalState gs, event e)
{   设 gs=((l,D),vector,stack);
    (l',D')=sp((l,D),e);
    if (D'为空)
        return NULL;
    令βi是转换 e 触发的中断事件β(e);
    if (βi≠∅)
    {   if (vector[βi]==false)
        {   vector[βi]=true;
            AddConstraintOfITA(gs,e)
        }
    }
    return ((l',D'),vector,stack);
}
```

Fig.7 Calculate the successor of the transition e from ITA

图 7 计算 ITA 转换后继的算法

该算法首先判断当前转换 *e* 是否触发中断事件.如果触发中断事件,即β_i!=∅,需要检查中断向量表 *vector* 的置位情况,如果中断向量表中该中断事件未置位,则设置 *vector*[β(*e*)]为 true.

函数 *AddConstraintOfITA*(*globalState gs*,*event e*)用于计算后继状态在 *eventPath* 中的时间约束,方法如下(其中,*num* 是 *eventPath* 中事件的数目,*eventPath*[*num*]包含当前事件(*event*=*e*,*type*=*I*)):

对于转换 *e*=(*l*,*g*,*r*,*l*')中重置的每个时钟约束 $x\sim n\in g$,从后往前查找 *eventPath* 中是否存在 *eventPath*[*i*]满足下列条件:*eventPath*[*i*].*type*=*I*, *eventPath*[*i*].*event* = (l_1, g_1, r_1, l_1') 且 $x\in r_1$.

1. 如果存在满足条件的 *i*,那么向路径成员 *eventPath*[*num*]中添加约束 $0\leqslant c_{num}-c_i\sim n$;
2. 如果不存在满足条件的 *i*,即时钟 *x* 没有被 *eventPath* 中记录的转换重置过,那么 *x* 的当前值是 *eventPath* 开始时 *x* 的值加上 c_{num} 的和,因此,向路径成员 *eventPath*[*num*]中添加约束条件 $x_0+c_{num}\sim n$.

操作符 *sp* 用于计算中断时间自动机状态的符号后继,具体计算方法见文献[8].使用差值约束矩阵 *DBM* 可以高效的计算出这样的符号后继.

2.2.3 计算中断向量表的事件后继

当中断向量表中已置位的最高优先级中断高于 CPU 栈中当前正在运行的中断处理程序时,系统将挂起当前的中断处理程序,执行该中断事件对应的处理程序.图 8 中的算法计算了全局状态相对于此类事件的后继状态.*eventPath*[*num*]包含当前路径成员 *ep*=⟨*globalState*,*eventList*,*event*,*type*,*constraints*,*bounds*⟩,其中,*num* 是事件路径 *eventPath* 中包含的成员数目;当前 *event* 为中断处理事件 H_{vector},*type* 为 *V*.首先,将当前事件 *event* 对应的中断处理程序 H_{vector} 加入 *stack*,并开始执行第 1 个子处理程序.如果栈中包含被挂起的中断处理程序,则需要调用图 9 中函数 *AddConstraintOfHandler* 分别计算被当前 *event* 挂起的中断处理程序及正在执行的子程序的执行时间约束条件,并将约束条件添加到 *eventPath*[*num*]的 *constraints* 中,函数 *AddBoundOfSubHandler* 计算了被当前 *event* 挂起的中断处理子程序的响应时间的界限条件,并将界限条件添加到 *eventPath*[*num*]的 *bounds* 中.

图 9 中的函数 *AddConstraintOfHandler*(*handler H*,*subhandler* h_k)计算中断处理程序 *H* 及其子程序 h_k 在 *eventPath* 中的执行时间约束,并加入到 *eventPath*[*num*]的 *constraints* 中.函数首先寻找将中断处理程序 *H* 加入栈的事件 *eventPath*[*i*],然后从 *eventPath*[*i*]开始向后遍历 *eventPath* 中的各个成员.如果 *eventPath*[*z*].*event* 是 *H* 且 *type* 为 *S*,表示当前正在执行 *H*,此时,使用中间参数 $terms_1=\sum(c_{z+1}-c_z)$ 辅助计算栈中实际执行时间段.中断处理程序的子程序 h_k 的时间约束 $terms_2$ 的计算方法类似.最后,将 $terms_1$ 和 $terms_2$ 加入当前 *eventPath*[*num*]的 *constraints* 中.

```
CalculateSuccessorOfVector(globalState gs, event e)
{   设 gs=((l,D),vector,stack)且 stack 栈顶标记为 top;
    设 e 对应于事件 H_vector=⟨b,w,upbnd,Rr,Rw⟩;
    stack[++top]=⟨H_vector,start,h0⟩;
    向 eventPath[num]中添加时间约束:那么 b≤c_num−c0≤w;
    if (top>0)
    {   stack[top−1].mode=suspended;
        AddConstraintOfHandler(stack[top−1].H,hk);
        AddBoundOfSubHandler(stack [top−1].H,hk);
    }
  return ⟨(l,D),vector,stack⟩;
}
```

Fig.8 Calculate the successor of the handler from the vector

图 8 计算中断向量表中最高优先级中断处理程序后继的算法

```
AddConstraintOfHandler(handler H,subhandler hk)
{   若 H=⟨b,w,upbnd,Rr,Rw⟩且 hk=⟨bk,wk,upbndk⟩;
    从前向后查找 eventPath 中 epi 满足:event=H,且 type=V;
    terms1=terms2=∅;
    for (int z=i; z<num; z++)
    {   if (epz 中 event 为 H)
        {   terms1+=(cz+1−cz);
            if (epz 中 globalState 的栈顶成员的当前子处理程序 h 为 hk)
                terms2+=(cz+1−cz);
        }
    }
    向 eventPath[num]中添加中断处理程序的时间约束:b≤terms1≤w;
    向 eventPath[num]中添加子处理程序的时间约束:bk≤terms2≤wk;
}
```

Fig.9 Calculate constraints of the interrupt handler H and its subhandler h

图 9 计算中断处理程序 H 及其子程序 h 执行时间相关约束的算法

对于中断子处理程序 h_k,使用函数 *AddBoundOfSubHandler*(*handler H*,*subhandler* h_k)计算响应时间界限条件,并添加到当前 *eventPath*[*num*]的 *bounds* 中.具体过程如下:

1. 从当前 *eventPath*[*num*]开始从后向前寻找满足如下条件的第 1 个成员 ep_i:*event* 为 *H*,且 *type* 为 *V*;
2. 从 ep_i 开始向后找满足如下条件第 1 个成员 ep_m:*event* 为 *H*,*type* 为 *S*,且其 *globalState* 中栈顶执行的中子处理程序为 h_k;
3. 向 *eventPath*[*num*]的 *bounds* 中添加界限条件:$c_{num}-c_m \leqslant upbnd_k$,其中,$upbnd_k$ 为 h_k 的最长响应时间.

2.2.4 计算 CPU 处理栈内中断处理程序的事件后继

位于CPU处理栈的栈顶的中断处理程序可以占有CPU进入执行状态;而在执行一定时间后,正在执行的中断处理程序或者子程序可能会执行结束.图 10 给出了对应于这两种情况的后继状态计算方法.

设当前 *eventPath*[*num*]的 *event* 是中断处理程序 *H*,*type* 为 *S* 时:

1. 如果当前栈顶中断处理程序 *H* 的执行状态是 *started*,那么开始执行当前子处理程序 $h_k \in H$,执行状态变为 *finished*;
2. 如果当前执行状态是 *finished*,那么当前子处理程序 h_k 执行结束.此时,需要调用图 9 中的函数 *AddConstraintOfHandler* 来计算 h_k 的执行时间约束,并加入到 *eventPath*[*num*]的 *constraints* 中;同时,调用函数 *AddBoundOfSubHandler* 来计算 h_k 的响应时间界限,并添加到当前 *eventPath*[*num*]的 *bounds* 中.若 *H* 中还有未执行的子处理程序,则开始执行 h_{k+1},当前执行状态变为 *started*.若 *H* 执行结束,则移除中断向量表中置位信息,并移除栈顶的中断处理程序 *H*,调用函数 *AddBoundOfHandler*(*handler H*)来计算 *H* 响应时间相关界限条件,并添加到 *bounds* 中.如果栈中还有其他未完成的中断处理程序 *H*′,继续执行 *H*′中当前子处理程序 *h*′,当前执行状态变为 *started*.

其中,函数 *AddBoundOfHandler*(*handler H*)从当前 *eventPath*[*num*]开始从后向前寻找满足:*event*==*e*,*type*==*I*,且 *Map*[β(*event*)]==*H* 的第 1 个成员 ep_i,并向 *eventPath*[*num*]的 *bounds* 中添加时间界限:$c_{num}-c_i \leqslant upbnd$,其中,*upbnd* 为 *H* 的最长响应时间.

```
CalculateSuccessorOfStack(globalState gs,event e)
{   若 gs=((l,D),vector,stack)且 stack 栈顶标记为 top;
    stack[top]=⟨H,mode,hk⟩,H=⟨b,w,upbnd,Rr,Rw⟩且 hk=⟨bk,wk,upbndk⟩;
      if (mode==started)
            stack[top].mode=finished;
      else if (mode==finished)
      {     AddConstraintOfHandler(stack[top].H,hk);
            AddBoundOfSubHandler(stack[top].H,hk);
            if (hk+1≠∅)
                  stack[top]=⟨H,mode=started,hk+1⟩;
            else
            {     AddBoundOfHandler(stack[top].H);
                  在 eventPath 中查找 epi 满足:type==I,且 Map[ββ(event)]==H}
                  vector[β(event)]==false;
                  if (top>0)
                        stack[top−1]=⟨H′,mode=started,h′⟩;
                  top−−;
            }
      }
    return ⟨(l,D),vector,stack⟩;
}
```

Fig.10 Calculate the successor of the interrupthandler from stack

图 10 计算处理栈中中断处理程序的后继的算法

2.2.5 基于事件路径的超时检测

在后继状态计算方法中,我们把各类时间约束和系统模型必须满足的时间规约(即,各个中断处理程序/子程序的时间上限)都记录在 *eventPath* 的各元素中.在算法中,我们需要判断 *eventPath* 中的事件序列是否可能发生,以及当前事件路径中已触发的中断事件是否一定在其时间界限内执行完成.具体检验方法如下:

1. 令 *CONS* 是事件路径中各个成员的 *constraints* 的合集,即:

$$CONS=constraints_0 \cup constraints_1 \cup \ldots \cup constraints_n,$$

其中,$constraints_i$ 是 *eventPath*[*i*]中的时间约束集合.通过调用线性规划程序包对 *CONS* 进行求解:如果 *CONS* 有解,则表示事件序列可能发生;无解,则表示事件序列因为时间约束而不可能发生;

2. *eventPath*[*m*].*bounds* 中的界限条件描述了系统运行时必须满足的时间界限规约,对其中的每一个界限条件 *bound*,算法使用线性规划程序包检验~*bound*∪*CONS* 是否有解:有解就表明可能发生不满足该 *bound* 的事件序列,算法还可以根据线性规划包给出的具体解(即,各个事件的发生时间)得到模型的一个不满足该事件界限条件的运行路径;无解则表明满足时间约束的事件序列一定满足 *bounds* 所规定的时间界限规约,即与 *bounds* 相关的中断事件处理程序或子过程一定可以在其规定的响应时间界限内执行完成,也就是说不会产生超时错误.需要注意的是:为了避免遍历中产生过多的不合理的超时事件路径序列,在判断时间界限条件时,通过静态分析可以得到系统中最大界限条件 *K*,即,系统中最长任务周期值.对于包含 *n* 个成员的 *eventPath*,判断事件路径是否有解时,需要添加时间约束 $bound_{\max}: c_n-c_0 \leqslant K$.

2.3 可达性分析算法

本文的中断驱动系统的模型检验算法使用深度优先策略来遍历模型的状态空间,它穷尽枚举各个符号状态上可能发生的所有事件,计算后继并检验是否可能发生中断处理超时错误.在遍历时,算法需要把一些已经遍历过的状态加入到可达集合中去,并不断查看当前状态是否被该集合中的状态包含.如果是,则不需要继续向前遍历.由于与 CPU 栈相关的时间约束存放在 *eventPath* 中,如果把 *eventPath* 作为状态的一部分存放到该集合中,

将会引起空间效率的下降.因此,算法只在该集合中记录那些 CPU 栈和中断向量表为空的状态.当 CPU 栈为空且中断向量表为空时,*eventPath* 中的时间约束和后续的系统行为无关,因此只需要记录中断时间自动机的符号状态即可.这样的处理方法,使得我们可以使用 DBM 数据结构来高效地计算符号状态之间的包含关系.

图 11 中给出了针对中断驱动系统的中断处理超时问题的模型检验算法.在该算法中,变量 *Graph* 记录了在穷尽遍历中生成的状态和它们之间的关系,变量 *Unexplored* 存储了需要继续遍历的全局状态的集合.

```
Unexplored={globalState0},其中,globalState0=⟨(l0,D0),vector,stack=∅⟩,vector 中无置位;
Graph=globalState0;
while (Unexplored≠∅)
{    选择 Unexplored 中的一个全局状态 globalState=⟨(l,D),vector,stack⟩;
     从 Unexplored 中移除 globalState;
     InitalEventPath();
     eventList=CalculateEnabledEvent(globalState);
     AddToEventPath(globalState,eventList);
     While (num>0)
     {    GlobalState sucessor;
          GlobalState gs=eventPath[num];
          if (eventPath[num]的 eventList 中有未枚举事件)
               选择 eventPath[num]的 eventList 中的一个未枚举事件 event,其类型为 type;
          else
               RemoveLastStateFromEventPath(); continue;
          if (type==I && vector[β(event)]==true)
               报告反例 eventSequence=RetrievedFromEventPath(); return false;
          if ((type==I)
               sucessor=CalculateSuccessorOfITA(gs,event);
          if ((type==V)
               sucessor=CalculateSuccessorOfVector(gs,event);
          if ((type==S)
               sucessor=CalculateSuccessorOfStack(gs,event);
          设 sucessor=⟨(l',D'),vector',stack'⟩;
          if (D'==∅)
               continue;
          if (!IsSolvable(eventPath))
               continue;
          if (IsOverTime(eventPath))
               报告反例 eventSequence=RetrievedFromEventPath(); return false;
          eventList2=CalculateEnabledEvent(successor);
          AddToEventPath(successor,eventList2);
          if (stack==∅ && vector 中无置位)
          {    eventSequence=RetrievedFromEventPath();
               if (Graph 中存在状态(l',D'')且 D'⊆D'')
                    向图 Graph 中添加从(l,D)到(l',D'')标记为 eventSequence 的边;
               else
               {    向图 Graph 中添加从(l,D)到(l',D')标记为 eventSequence 的边;
                    AddToUnexplored((l',D'));
               } RemoveLastStateFromEventPath();
          }
}
```

Fig.11 Explore the state space of the model of interrupt-driven systems

图 11 中断驱动系统模型的状态空间遍历的算法

算法主要由两层循环构成:外循环中不断从 *Unexplored* 中获取状态,并通过内层循环来计算后继.当 *Unexplored* 为空时,所有可达状态都已经遍历完毕,因此模型检验过程结束;对于外层循环选取的每一个状态 *gs*,经过初始化后,*gs* 上的每个可能发生的事件都记录在 *eventPath*[*num*]的 *eventList* 中,内层循环根据这些事件的类型选择不同的方法计算全局状态的事件后继,并以深度优先的方式进行搜索.当 *eventPath*[*num*]的 *eventList* 中所有事件都已经被处理完毕,算法就会进行回溯.在内层循环中,算法对每个事件按照如下规则处理:

第 1 步:如果当前事件为 *ITA* 中对应于某个中断事件的转换,且中断向量表中该中断已被置位,那么之前发生的同一个中断事件尚未被处理.这表明系统可能出现丢中断的情况,因此算法终止遍历,报错并提交反例;

第 2 步:根据事件类型选择不同的方法计算后继 *successor*.如果事件是中断时间自动机的转换,且中断时间自动机后继是空集,那么该转换因为时间自动机的时间约束而无法发生,因此无后继状态,内层循环会尝试下一个事件;

第 3 步:使用函数 *IsSolvable*(*eventPath*)调用线性规划程序包判断当前 *eventPath* 中各个成员的时间约束集合是否可解:如果不可解,就表示当前事件因为时间约束而不可能发生,算法将尝试下一个事件;

第 4 步:使用函数 *IsOverTime*(*eventPath*)调用线性规划程序包来判断当前事件路径 *eventPath* 中是否存在超时情况:

1. 对于系统中最大时间界限 $bound_{\max}$,计算~$bound_{\max}\cup CONS$ 是否有解:如果有解,那么当前事件路径中必然存在中断事件无法在规定响应时间内完成;
2. 对 *eventPath*[*num*]的 *bounds* 中每个界限条件 *bound*,计算~*bound*∪*CONS* 是否有解:如果有解,那么当前的符号路径中必然存在某一条具体路径,使得 *bound* 对应的中断事件或中断处理子过程无法在规定时间内执行完成.

若任意一种情况有解,此时算法报错,并调用函数 *RetrievedFromEventPath*()从 *eventPath* 中提取当前中断事件执行序列信息,记为 *eventSequence*,并报告反例.

第 5 步:如果当前 *successor* 的栈为空且 *vector* 中没有中断置位,当前后继的时间自动机符号状态(l',D')以及从事件路径初始状态出发到达(l',D')的中断事件序列 *eventSequence* 被加入到图 *Graph* 中,同时执行回溯操作.在加入时,如果 *Graph* 中已经存在一个包含(l',D')的状态(l',D''),那么在 *Graph* 中添加从 *gs* 的自动机状态(l,D)到(l',D'')的、标记为 *eventSequence* 的边;否则,向 *Graph* 中添加新的可达状态(l',D')以及从(l,D)到(l',D')标记为 *eventSequence* 的边,并且将 *successor* 加入 *Unexplored* 中.

如果上面的 5 种情况都没有出现,那么内层循环将 *successor* 以及相应的时间约束加入到 *eventPath* 中,并继续向前计算新状态的后继.

当这个深度优先搜索过程结束后,内层循环结束.外层循环将从 *Unexplored* 中选择一个新的状态开始新的遍历.当 *Unexplored* 为空集时,整个空间遍历过程结束,表明系统模型不会发生中断处理超时的情况.

例 5:给定例 1 中表 1 和表 2 信息描述的中断驱动系统,经过建模后得到由图 1 和图 4 并发组成的中断时间自动机模型系统,其中,初始位置为(l_{10},l_{20},l_{30}),初始时,时钟 $x=y=z=0$,变量 $v=0$.分析由表 1 和表 2 中信息转换而得的表 3 中的中断处理程序响应时间上界信息,可知系统中最长响应时间上界为 K 为 100.那么,如果可达性分析算法从一个可达的全局状态出发,在 100 个时间单位内未能产生新的全局状态,那么系统必定存在超时问题.

表 4 中给出了使用图 11 中的可达性分析算法检验发现一个反例的计算过程.*eventPath*[0]从当前可达状态 $globalState_1=\langle(l_1,D_1),vector,stack\rangle$出发,其中,$l_1=(l_{12},l_{21},l_{30})$,且 $D_1=\{0\leqslant x<160;1\leqslant y<10;0\leqslant z\}$,*vector* 中无置位,*stack* 为空,依次执行转换序列 $seq=e_3,e_7,e_6,e_7,e_8,e_6$,可能产生的某条 *eventPath* 及其约束见表 4.表中各行分别表示事件路径中 *eventPath* 第 *i* 个成员的相关信息,*ITA*,*vector* 和 *stack* 各列分别代表 *eventPath*[*i*]所在的全局状态中当前 *ITA* 中的状态、中断向量表中的置位信息和 CPU 处理栈中包含的中断处理程序,其中,位置 $l_2=(l_{13},l_{21},l_{30})$,$l_3=(l_{13},l_{21},l_{31})$.*event*,*cons* 和 *bound* 各列分别表示当前执行的事件以及计算当前 *event* 后继时需要满足的约束条件和边界条件(为了简化案例,这里我们不考虑中断子处理程序的需要满足的约束条件和边界条件).

此外,对表格中每个 *eventPath*[*i*],还需要添加默认的约束条件 *cons*:$0\leqslant c_i-c_{i-1}$,其中,$i>0$.这里,引入代表 *ITA* 中零时刻的时钟 t,使得 D_1 中时钟约束满足$\{0\leqslant x-t<160;1\leqslant y-t<10;0\leqslant z-t\}$.需要注意的是:由于中断触发过程不占用 CPU 栈的处理时间,这里我们假设事件路径中中断时间自动机的转换事件的执行过程所需时间为 0.路径遍历中,对每个 *eventPath*[*i*]分别检验计算 *cons*∪*K* 和 *cons*∪~$bound_i$ 是否有解:如果有解,那么当前 *eventPath* 中存在无法在规定响应时间内处理完成的中断事件.

例如,*eventPath*[5]的当前全局状态中,当前位置为(l_{13},l_{21},l_{30}),中断向量表中有 *vector*(T_3)=true,*vector*(I_2)=true,且当前栈中包含两个待处理的中断处理程序 H_3,H_5,栈顶正在执行的中断处理程序为 H_5.当前事件为 CPU 栈顶中断处理程序 H_5 计算完成并退栈,此时需要向事件路径中添加与执行时间相关的约束条件 $1\leqslant c_4-c_3+c_5-c_4\leqslant 2$,

以及与响应时间相关的边界条件 $c_5-c_4\leqslant 4$,并调用线性规划程序包进行求解计算.

计算可知,表 4 中 *eventPath*[25]的 $cons\cup\sim bound_{25}$ 有解,即:从当前 $globalState_1$ 出发执行 *seq*,其中,*ITA* 中转换 e_3 对应的任务事件 T_3 中断处理事件 H_3 被中断事件 I_1,I_2 多次挂起后,将无法在规定响应时间内完成.

Table 4 An example of calculating *eventPath* from $globalState_1$ following by the transition sequence *seq*

表 4 从 $globalState_1$ 执行 *seq* 的某条枚举路径 *eventPath* 的检验案例

i	*ITA*	*vector*	*stack*	*event*	*cons*	*bound*
0	(l_1,D_1)	–	–	计算 *ITA* 中转换 e_3 的后继	–	–
1	(l_2,D_2)	T_3	–	计算 *vector* 的事件(H_3)后继	–	–
2	(l_2,D_2)	T_3	H_3	计算 *ITA* 中转换 e_7 的后继	–	–
3	(l_2,D_3)	T_3, I_2	H_3	计算 *vector* 的事件(H_5)后继	$c_2-c_1\leqslant 32$	–
4	(l_2,D_3)	T_3, I_2	H_3, H_5	计算 CPU 栈内 H_5 的后继	$c_4-c_3\leqslant 2$	–
5	(l_2,D_3)	T_3, I_2	H_3, H_5	*stack* 内 H_5 计算完成,退栈	$1\leqslant c_4-c_3+c_5-c_4\leqslant 2$	$c_5-c_2\leqslant 4$
6	(l_2,D_3)	T_3	H_3	计算 *ITA* 中转换 e_6 的后继;	–	–
7	(l_2,D_4)	T_3, I_1	H_3, H_4	计算 *vector* 的事件(H_4)后继	$c_2-c_1+c_6-c_5\leqslant 32$	–
8	(l_2,D_4)	T_3, I_1	H_3, H_4	计算 *stack* 内 H_4 的后继	$c_8-c_7\leqslant 2$	–
9	(l_2,D_5)	T_3, I_1	H_3, H_4	计算 *ITA* 中转换 e_7 的后继	–	–
10	(l_2,D_5)	T_3, I_1, I_2	H_3, H_4	计算 *vector* 的事件(H_5)后继	$c_8-c_7+c_9-c_8\leqslant 2$	
11	(l_2,D_5)	T_3, I_1, I_2	H_3, H_4, H_5	计算 *stack* 内 H_5 的后继	$c_{11}-c_{10}\leqslant 2$	–
12	(l_2,D_5)	T_3, I_1, I_2	H_3, H_4, H_5	*stack* 内 H_5 计算完成,退栈	$1\leqslant c_{11}-c_{10}+c_{12}-c_{11}\leqslant 2$	$c_{12}-c_9\leqslant 4$
13	(l_2,D_5)	T_3, I_1	H_3, H_4	计算 *stack* 内 H_4 的后继	$c_8-c_7+c_9-c_8+c_{13}-c_{12}\leqslant 2$	–
14	(l_2,D_5)	T_3, I_1	H_3, H_4	计算 *ITA* 中转换 e_8 的后继	–	–
15	(l_3,D_6)	T_3, I_1, I_2	H_3, H_4	计算 *vector* 的事件(H_5)后继	$c_8-c_7+c_9-c_8+c_{13}-c_{12}+c_{14}-c_{13}\leqslant 2$	–
16	(l_3,D_6)	T_3, I_1, I_2	H_3, H_4, H_5	计算 *stack* 内 H_5 的后继	$c_{16}-c_{15}\leqslant 2$	–
17	(l_3,D_6)	T_3, I_1, I_2	H_3, H_4, H_5	*stack* 内 H_5 计算完成,退栈	$1\leqslant c_{16}-c_{15}+c_{17}-c_{16}\leqslant 2$	$c_{17}-c_{14}\leqslant 4$
18	(l_3,D_6)	T_3, I_1	H_3, H_4	计算 *stack* 内 H_4 的后继	$c_8-c_7+c_9-c_8+c_{13}-c_{12}+c_{14}-c_{13}+c_{18}-c_{17}\leqslant 2$	–
19	(l_3,D_6)	T_3, I_1	H_3, H_4	*stack* 内 H_4 计算完成,退栈	$1\leqslant c_8-c_7+c_9-c_8+c_{13}-c_{12}+c_{14}-c_{13}+c_{18}-c_{17}+c_{19}-c_{18}\leqslant 2$	$c_{19}-c_6\leqslant 8$
20	(l_3,D_6)	T_3	H_3	计算 *ITA* 中转换 e_6 的后继	$10\leqslant c_{20}-c_6\leqslant 10$	–
21	(l_3,D_7)	T_3, I_1	H_3	计算 *vector* 的事件(H_4)后继	$c_2-c_1+c_6-c_5+c_{20}-c_{19}\leqslant 32$	–
22	(l_3,D_7)	T_3, I_1	H_3, H_4	计算 *stack* 内 H_4 的后继	$c_{22}-c_{21}\leqslant 2$	–
23	(l_3,D_7)	T_3, I_1	H_3, H_4	*stack* 内 H_4 计算完成,退栈	$1\leqslant c_{22}-c_{21}+c_{23}-c_{22}\leqslant 2$	$c_{23}-c_{20}\leqslant 8$
24	(l_3,D_7)	T_3	H_3	计算 *stack* 内 H_3 的后继	$c_2-c_1+c_6-c_5+c_{20}-c_{19}+c_{24}-c_{23}\leqslant 32$	–
25	(l_3,D_7)	T_3	H_3	*stack* 内 H_3 计算完成,退栈	$24\leqslant c_2-c_1+c_6-c_5+c_{20}-c_{19}+c_{21}-c_{20}+c_{24}-c_{23}+c_{25}-c_{24}\leqslant 32$	$c_{25}-c_0\leqslant 40$

2.4 其他性质的验证

前面给出的模型检验算法遍历了系统模型的所有状态,因此,我们对这个算法加以扩展,检验中断处理超时之外的一些其他性质.

2.4.1 资源竞争验证

当两个不同优先级的中断事件触发的中断处理程序以读写或写写的方式共享同一个公共资源时,系统在运行时就可能发生访问冲突.例如在航天器控制系统中,对于运行角度、方向速度等设计中包含很多浮点计算,这些计算的精度对于系统的正确性有很大的影响.我们可以把浮点寄存器看作一个共享资源,如果低优先级中断程序 H_i 对浮点寄存器进行读操作时被高优先级中断程序 H_j 打断,并且 H_j 执行过程中修改了该寄存器中数值,那么当 H_j 执行结束后恢复 H_i 就可能读取到错误的内容.

为了防止这类问题的发生,我们要求设计人员首先列出共享资源列表 *R*,并且指明每个中断处理程序的子过程 *h* 访问了哪些资源.我们用 $Rw(h)$ 表示中断处理程序子过程 *h* 写访问的资源的集合,$Rr(h)$ 表示中断处理程序子过程 *h* 读访问的资源的集合.对于分属两个不同中断处理程序的子过程 h_i 和 h_j,如果 $Rr(h_i)\cap Rw(h_j)\neq\varnothing$ 或 $Rw(h_i)\cap Rr(h_j)\neq\varnothing$ 成立,那么中断子处理程序 h_i 和 h_j 间存在资源竞争关系.给定各个中断处理子程序的 $Rw(h)$ 和 $Rr(h)$ 集合,我们就可以在模型检验算法执行之前,通过静态分析计算得到所有可能发生冲突的子过程对.

对于任意中断子处理程序 h_i,h_j,其中,$h_i\in H_i$,$h_j\in H_j$,$i\neq j$,且 $priority(H_i)<priority(H_j)$,如果当前 CPU 处理栈 *stack* 中 *stack*[*a*]的当前子处理程序为 h_i,*stack*[*b*]的当前子处理程序为 h_j,其中,$a<b\leqslant top$,那么称 h_i,h_j 同时活跃于 CPU 处理栈中,记为 $h_i,h_j\in active(stack)$.如果 h_i 和 h_j 间存在资源竞争关系,那么检验算法就可以报告错误.

2.4.2 子程序原子性的验证

中断处理程序中的某些特殊的子过程在执行过程中不可被打断,否则无法保证系统正确性.这些子过程通常必须在很短时间内执行完毕,一旦被中断,就可能导致错误结果,比如串口通信、外部计数寄存器的读取等.基于本文图10中的状态空间遍历算法,我们可以很方便地检验子程序的原子性:每次当算法在CPU栈中压入一个中断处理程序 H 之前,如果当前栈顶元素的中断处理子程序是一个原子性的子过程,那么模型检验算法就可以停止遍历,报错并给出相应的反例.

3 实例研究

本文给出的模型检验算法适用于任务加中断组织结构的中断驱动系统,其中的任务调度采用时间片轮询调度机制.我们使用工具对一类实际系统的模型进行了检验,在表5中给出了计算包含 n 个任务 m 个中断的模型系统所需的遍历时间(时间单位:s).随着模型系统中中断数目的增长,遍历过程所需时间的基本成指数级增长.任务的数量不会显著影响模型检验所需的状态空间.

Table 5 Exploration time w.r.t n tasks and m interrupts (s)

表5 包含 n 个任务 m 个中断的系统的遍历时间 (s)

n 任务	m 中断						
	1	2	3	4	5	6	7
1	<1	<1	1	11	102	1 249	15 481
2	<1	<1	2	18	155	1 753	18 689
3	<1	<1	3	24	253	2 086	21 149
4	<1	<1	4	29	312	3 133	29 674

针对航天控制系统这类安全关键的嵌入式中断驱动系统,本文进行了分析和统计,由于系统存储空间的限制和对系统可靠性的要求,这类系统的设计中大部分都不会设置超过3个中断优先级,中断数量不超过7个,系统调度任务数量不超过4个.这里,我们使用本文中的中断驱动系统建模及检验方法对某个给定的航天器的控制软件系统进行了建模和检验.该软件使用任务加中断的软件体系结构.软件运行时,操作系统对4个任务按照固定次序进行调度,并且由外部中断源触发中断处理程序对外界信号进行处理.表6中给出了该航天控制系统经抽象后的相关信息:包含4个系统调度任务和7个中断的输入信息(系统共有2个不同的优先级).其中,任务 T_1,T_2,T_3,T_4 以128个时间单位为周期顺序执行,中断事件 I_1,I_4 为周期中断,其余中断事件都是偶发中断.其中,中断 I_1 和任务 T_1 将对系统中的4个共享资源分别进行读操作或写操作,这4个共享资源分别是:*Sint*,*MSInt*,*MS*,*deltaMS*.

基于表6中的信息,我们使用第1.3节中的方法对该系统进行建模.在系统模型中,对所有任务开始事件和中断事件建模的时间自动机模型是由8个时间自动机(任务开始事件由一个时间自动机建模,其余每个中断源一个时间自动机)并发组合得到的.系统任务的优先级低于中断的优先级,模型系统中,任务 T_i 具有相同的优先级,中断 I_i 的优先级相同.根据表2中的时间相关信息和资源读写信息,可转换得到各个任务及中断对应的中断处理程序模型.例如,T_1 对应的形如$\langle b,w,upbnd,Rr,Rw\rangle$中断处理程序 H_1=⟨44,66,88,{*SInt*,*MSInt*},{*MS*,*deltaMS*}⟩.

通过分析可知:任务序列的执行周期为128个时间单位,且模型系统中任务具有较低优先级.系统中响应时间上界 K 可记为128个时间单位,如果可达性分析算法从一个可达的全局状态出发,在128个时间单位内未能产生新的、CPU栈和中断向量表全为空的全局状态,那么系统中必定有某个任务事件无法在规定时间内完成,即.存在超时问题.

我们使用图11中的可达性分析算法对该中断驱动系统模型进行状态空间遍历,证明了上述模型不会产生中断处理超时的问题.当我们改变其中某些中断的 *wcet* 或 *upbnd* 值时,该算法可以报告超时错误,并报告相应的反例.

Table 6 Task list and interrupt list

表 6 任务和中断列表

任务	*bcet*	*wcet*	*upbnd*	*period*	*offset*	*Rr*	*Rw*
T_1	44	66	88	128	0	*SInt*, *MSInt*	*MS*, *deltaMS*
T_2	8	12	16	128	88	–	–
T_3	8	12	16	128	96	–	–
T_4	4	6	8	128	112	–	–
I_1	1	1.4	2	30 000	–	*MS*, *deltaMS*	*SInt*, *MSInt*
I_2	1	1.4	2	–	–	–	–
I_3	0.3	0.5	1	–	–	–	–
I_4	0.3	0.5	1	1 000	–	–	–
I_5	0.3	0.5	1	–	–	–	–
I_6	0.3	0.5	1	–	–	–	–
I_7	0.3	0.5	1	–	–	–	–

4 相关工作和总结

对于中断驱动系统的可靠性问题,学者们提出了很多不同的建模和验证方法.在文献[9,10]中,作者给出了针对 z-86 中断驱动软件系统的超时分析工具,z-86 系统包含 6 个可以在任意时刻触发的中断源.文中使用控制流图静态分析自动标识,并隔离需要进一步检验的代码段,这些代码段描述了从一个程序点到另一个程序点的最坏情况执行时间,然后仅对这些代码段进行超时分析测试.该文献中仅对 z-86 系统中时间行为进行抽象,且超时分析依赖于对中断到达最小时间间隔的静态分析.文献[11]中基于状态机建模并检验中断驱动调度系统,该系统基于轮询调度策略循环执行任务序列,模型系统未考虑多优先级的中断调度情况,且系统中缺少时间相关信息.文献[12]中提出了对实时嵌入系统的建模及检验方法,给出了结合静态分析、抽象解释和模型检验等形式化方法的工具[MC]SQUARE,并在此基础上给出使用偏序约减计算进一步约减生成的状态空间.文中特别对中断处理程序间及程序间依赖关系进行静态分析,从而约减生成的状态空间,但模型系统中缺少中断相关时间约束信息,无法验证超时问题.文献[14]中引入了基于混成自动机的中断时间自动机(ITA),用于描述多任务单处理器软件系统,并且结合控制实时自动机(CRTA)和中断时间自动机得到可达性可判定的子集 ITA^+.文中的中断时间自动机是混成自动机的一种变体,计算可达性所需的时间复杂性较高;同时,虽然 ITA^+可以对包含任务和中断的中断驱动系统的时间行为进行建模并验证模型系统可达性,但是中断自动机无法描述中断源被触发后系统响应和处理中断事件的完整过程,并且无法验证中断响应时间相关的时间约束.

本文给出了一类中断驱动系统的建模方法,并设计实现了解决时序相关问题的模型检验算法.我们首先确定了此类系统的设计中和时序相关的要素:系统任务、中断源、和中断处理程序的各类时间信息.根据这些要素,我们给出了一个能够将这些要素转换为形式化模型的方法.这个方法建立了形式化方法和实际开发实践之间的桥梁,它使得工程师不必熟悉形式化建模技术,仅通过填写表格就可以应用模型检验算法,降低了模型检验技术的应用难度.最后,提出了一种算法来检验系统模型的中断处理超时问题.在此算法的基础上,我们还设计了检验资源竞争问题和子程序完整性的算法.基于我们的方法,系统设计开发者可以在系统设计的早期发现可能存在的超时等问题.此外,我们的工作可以在系统的开发实现过程中辅助系统开发人员检验实际系统是否满足系统规约.最后,本文对一个实际的中断驱动系统进行了抽象、建模和验证,实验表明了我们算法的有效性.

References:

[1] Silberschatz A, Galvin PB, Gagne G. Operating System Concepts. 8th ed., Boston: Addison-Wesley Longman Publishing Co., Inc., 2008.

[2] Walker W, Cragon HG. Interrupt processing in concurrent processors. IEEE Computer, 1995,28(6):36–46. [doi: 10.1109/2.386984]

[3] Clarke EM, Grumberg O, Peled DA. Model Checking. Cambridge: MIT Press, 2000.

[4] Alur R, Dill DL. A theory of timed automata. Theoretical Computer Science, 1994,126(2):183–235. [doi: 10.1016/0304-3975(94)90010-8]

[5] Henzinger TA. The theory of hybrid automata. Verification of Digital and Hybrid Systems, 2000,170:265–292. [doi: 10.1007/978-3-642-59615-5_13]

[6] Dill DL. Timing assumptions and verification of finite-state concurrent systems. In: Proc. of the Int'l Workshop on Automatic Verification Methods for Finite State Systems. LNCS 497, Berlin, Heidelberg: Springer-Verlag, 1990. 197–212. [doi: 10.1007/3-540-52148-8_17]

[7] Alur R. Timed automata. In: Proc. of the 11th Int'l Conf. on Computer-Aided Verification. LNCS 1633, Berlin, Heidelberg: Springer-Verlag, 1999. 8–22. [doi: 10.1007/3-540-48683-6_3]

[8] Zhao JH, Wang LZ, Li XD. A partial order reduction technique for parallel timed automaton model checking. In: Proc. of the Leveraging Applications of Formal Methods, Verification and Validation Communications in Computer and Information Science, Vol.17. Berlin, Heidelberg: Springer-Verlag, 2008. 262–276. [doi: 10.1007/978-3-540-88479-8_19]

[9] Brylow D, Damgaard N, Palsberg J. Static checking of interrupt-driven software. In: Proc. of the 23rd Int'l Conf. on Software Engineering. IEEE, 2001. 47–56. [doi: 10.1109/ICSE.2001.919080]

[10] Brylow D, Palsberg J. Deadline analysis of interrupt-driven software. IEEE Trans. on Software Engineering, 2004,30(10): 634–655. [doi: 10.1109/TSE.2004.64]

[11] Stoddart B, Cansell D, Zeyda F. Modeling and proof analysis of interrupt driven scheduling. In: Proc. of the Formal Specification and Development in B 2007. LNCS 4355, Berlin, Heidelberg: Springer-Verlag, 2006. 155–170. [doi: 10.1007/11955757_14]

[12] Schlich B. Model checking of software for microcontrollers. Journal of ACM Trans. on Embedded Computing Systems, 2010,9(4): 1–27. [doi: 10.1145/1721695.1721702]

[13] Schlich B, Noll T, Brauer J, Brutschy L. Reduction of interrupt handler executions for model checking embedded system. In: Proc. of the Hardware and Software: Verification and Testing. LNCS, Berlin, Heidelberg: Springer-Verlag, 2011. 5–20. [doi: 10.1007/978-3-642-19237-1_5]

[14] Berard B, Haddad S, Sassolas M. Interrupt timed automata: Verification and expressive. Journal of Formal Methods in System Design, 2012,40(1):41–87. [doi: 10.1007/s10703-011-0140-2]

周筱羽(1985－),女,江苏南京人,博士,主要研究领域为软件工程,形式化验证.

顾斌(1968－),男,研究员,博士生导师,主要研究领域为航天器控制与推进系统设计,计算机控制,嵌入式软件.

赵建华(1971－),男,博士,教授,博士生导师,主要研究领域为软件工程,形式化方法.

杨孟飞(1962－),男,博士,研究员,博士生导师,主要研究领域为航天器设计,控制系统和控制计算机.

李宣东(1963－),男,博士,教授,博士生导师,主要研究领域为建模与分析,软件测试与验证.