软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(10):2485−2503 [doi: 10.13328/j.cnki.jos.004790]

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

中断驱动控制系统的有界模型检验技术*

周筱羽[1,2], 顾 斌[3], 赵建华[1,4], 杨孟飞[5]

[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)

通信作者: 赵建华, E-mail: zhaojh@mail.nju.edu.cn

摘 要: 针对一类中断驱动的航天控制系统,给出了有界模型检验的算法.这类系统由中断处理程序和操作系统调度的任务组成.当中断发生时,对应的中断处理程序响应中断事件,并可以修改控制变量值,以便在系统任务中完成后续工作.操作系统周期性地调度任务序列处理日常事务以及中断事件的后续工作.使用了带中断标记的时间自动机对中断事件和任务调度事件进行建模,并使用中断向量表和中断处理程序的伪代码模型共同描述中断的处理过程.控制变量将中断处理过程和系统任务相关联,中断处理程序可以设定某个控制变量,而系统任务则通过检查该控制变量来确定是否需要进行后续处理.对于这样的形式化模型,给出了检验关键时序性质的有界模型检验算法.该算法使用深度优先的方式遍历所有长度小于等于 K 的可行路径,并使用 SMT Z3 实现了对时间约束和规约的处理.

关键词: 中断驱动系统;有界模型检验;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.中断驱动控制系统的有界模型检验技术.软件学报,2015,26(10):2485−2503. http://www.jos.org.cn/1000-9825/4790.htm

英文引用格式: Zhou XY, Gu B, Zhao JH, Yang MF. Bounded model checking technique for interrupt-driven systems. Ruan Jian Xue Bao/Journal of Software, 2015,26(10):2485−2503 (in Chinese). http://www.jos.org.cn/1000-9825/4790.htm

Bounded Model Checking Technique for Interrupt-Driven Systems

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

[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: This paper proposes an approach to model and verify a certain class of interrupt-driven aerospace control systems. These interrupt-driven systems consist of interrupt handlers and system-scheduling tasks. When an interrupt event occurs, the corresponding interrupt-handler executes in response. An interrupt-handler may leave some post-processing works to the system tasks by setting some control variables to certain values. The operating system schedules a set of tasks periodically to deal with routine tasks and some post-processing of interrupt events. In this paper, timed automata labeled with interrupts are used to model interrupt events and task scheduling events. The execution processes of interrupts are modeled by pseudo-code of interrupt handlers and the interrupt vector. Control variables are used to model the interactions between interrupt processing and system tasks while the tasks perform

* 基金项目: 国家自然科学基金(91118007, 61321491); 国家高技术研究发展计划(863)(2012AA011205)
收稿时间: 2014-07-01; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

post-processing of interrupts according to the values of control variables set by interrupt handlers. A bounded model checking algorithm is presented in this paper to check these models w.r.t some important timing properties. The algorithm explores all feasible paths in K steps using the depth-first searching method. During the exploring process, time constraints and time requirements in the specification are calculated by the SMT solver Z3.

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

实时嵌入系统如今被广泛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.这类系统要求较高的安全性和可靠性,任何微小的错误都可能造成人员伤亡或后果难以估计的重大损失.因此,中断驱动实时处理系统的可靠性保障至关重要.

安全关键系统通常部署在特定的物理环境中,安全关键系统的可靠性由系统所属的外部环境、硬件设备和软件系统协同决定,一旦发生错误,很难定位和重现.对于一般的实时系统,可以通过重启系统或是重启任务来恢复系统的正常执行.可是对于安全关键实时系统,任务的执行有着严格的时间限制,如果无法在任务规定的截至期限内完成响应处理,就可能造成严重的系统失效.在安全关键实时系统的设计开发过程中,系统能够使用的资源受限于系统所属的物理环境,无法简单地使用冗余机制提高系统的可靠性.如何保障系统逻辑运算的正确性、如何保障系统中事件响应的实时性,都是安全关键系统可靠性保障中的关键研究问题.

为了实现复杂的系统行为,这类系统通常使用中断处理程序来实现同步通信、突发事件等的实时处理.中断驱动系统[1,2]很多都采用任务加中断或主程序加中断的结构.在中断驱动系统中,任务或主程序完成系统的主要控制逻辑,中断响应外部事件或处理其他突发请求.只要有中断事件发生,当前CPU中正在执行的任务事件或主程序立即被挂起.当系统中所有的中断事件都处理完成后,对于主程序+中断组织结构的系统,从挂起的程序点开始继续执行主程序;对于任务+中断组织结构的系统,需要判断当前任务是否已经超时,如果发现超时,需要报错、或是在系统允许的超时次数内,继续执行下一条任务.此外,任务+中断组织结构中,可以采用不同的调度策略执行任务.主程序+中断组织结构的系统中,主程序中事件的执行顺序主要依赖于主程序中的算法逻辑.

中断事件触发后,需要在给定时间范围内被响应完成,高优先级中断可以打断低级中断的执行,这样就形成了中断嵌套.由于中断事件的触发时间具有不确定性,中断事件的发生顺序也是不确定的,中断系统中包含的某些设计错误只有在特定的触发时间或是特定的处理顺序之下才会显现出来.常见的错误有超时问题、中断丢失、数据不一致、资源竞争等.我们把这类错误称为中断处理系统的时序相关错误.

由于中断驱动系统的运行环境一般非常复杂,系统的设计开发人员无法测试或模拟所有可能发生的情况.同时,由于中断事件发生的不确性,测试人员也无法测试所有可能发生的中断事件的序列组合.有时即使指定了中断事件的顺序,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.对中断驱动系统的测试工作开销极大且效率低下,难以有效保证系统的可靠性.

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.对于和时间相关的系统,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度也很高.因此,本文对时间自动机进行了扩展,用于对中断事件和任务的开始事件进行建模,并使用中断处理程序、中断向量表和CPU栈等来对中断响应过程进行建模.

有界模型检验(bounded model checking,简称BMC)[6]是符号化模型检验的一种优化方法,它的主要思想是:在系统的部分状态空间上检验属性的实效性,并将属性的实效性转换为命题公式的可满足性上.有界模型检验技术的核心思想是:从初始状态出发,仅仅检验路径长度小于 K 的所有系统路径,因此具有较低的时间复杂度.这使得我们不仅能够处理规模较大的系统,也能够在被验证的模型中描述更多的系统细节,以便发现更深入的系统错误.

文献[7]中指出:当实验验证的边界上界 K 小于60时,有界模型检验优于传统模型检验.根据我们对超过200

个已知错误实例的研究,大于 90%的时间相关错误可以在 20 步之内重现,即,只需要 20 个事件在特定时间点上发生就可以重现系统的错误.因此,只需将 K 设置为 20,就能够发现大部分的错误.对于中断驱动控制系统,我们可以采用有界模型检验技术来验证系统中的时间性质的正确性.

1 中断驱动系统建模

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,每个任务预先分配固定的时间配额.如果当前任务在分配的时间配额内未执行完成,那么当前任务被强行挂起,并记录一次超时,然后开始执行下一条任务.如果任务连续超时多次,并超出系统允许的超时次数上限,则会报告错误.如果当前任务提前执行完成,当前处理器处于空闲状态,则需要等待分配的时间配额到达后,再开始执行下一条任务.

中断由系统软件统一管理,包括中断向量初始化、上下文保护和上下文恢复等工作.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,对应的处理程序将被压入 CPU 栈并开始执行.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 中断时间自动机和中断源建模

我们以带有中断标号的时间自动机对中断源及任务开始事件建模.本文中,把这样的时间自动机简称为中断时间自动机.

1.2.1 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.因此,本文在时间自动机中的转换上添加了中断/任务开始事件的标号,用于对中断源和系统任务的开始事件进行建模.中断时间自动机主要用于描述中断事件和系统任务调度事件相关的时间信息,包括中断是否周期性的、周期性中断的发生周期、偶发中断两次中断事件之间的最小时间间隔、系统任务的调度周期以及各个任务在周期中的偏移量等.根据这些信息,模型检验工具可以枚举出所有可行的中断事件序列,从而验证系统能否在各种事件序列下都正确运行.

我们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,$t\in C$,$\sim\in\{<,\leqslant,\geqslant,>\}$,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,$G(C)\subseteq B(C)$.

添加中断标号的时间自动机 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,$r\subseteq C$ 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,$\beta(e)=\varnothing$.这个映射为自动机中的某些转换添加了相应的中断/任务事件的标号.

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 $l\xrightarrow{e}l'$.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 $x,u'(x)=u(x)$.

需要注意的是:这些自动机的转换上的标号不用于自动机之间的同步,而是用于表示中断/任务调度事件的发生.当这些转换发生时,中断向量表中相应的位置被置位.

1.2.2 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,操作系统周期性地调度执行这些任务.在一个调度周期内,各个任务以各自给定的偏移量为开始执行的时间点.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,那么这些任务的开始事件将被建模为一个包含 n+1 个状态的循环执行的中断时间自动机.其中,自动机中包含本地时钟 $x\in C$ 初始时值为 0,每经过一个调度周期后重置为 0.自动机中每个转换代表一个系统任务开始执行,如果任务 $task_i$ 的时间偏移量为 $offset$,其开始事件对应的转换 e_i 执行时需要满足时间约束 $x==offset$.

例 1:图 1 中给出了对一组任务序 T_1,T_2,T_3 进行建模的时间自动机,它们的调度周期为 200 个单位时间,其中,T_1,T_2,T_3 的偏移量分别为 0,100 和 160 个时间单位.中断自动机中,时钟 x 初始值为 0.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:

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

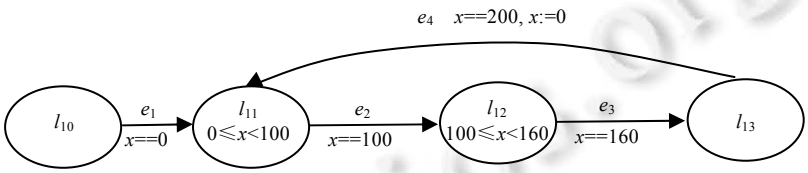

Fig.1 ITA model of a task sequence

图 1 任务序列的中断时间自动机模型

每个周期性中断被建模为一个包含两个状态的中断时间自动机.如图 2 所示,初始时 $x\in C$ 值为 0.e_1 和 e_2 都代表中断 I 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 $s_1\leqslant x\leqslant s_2$,其中,$s_1$ 和 s_2 分别表示在系统开始运行后最少 s_1 时间、最多 s_2 时间之后中断事件允许会发生.转换 e_2 的时间约束 $x=period$,并且将 x 重置为 0,其中,$period$ 表示这个中断事件发生的周期.

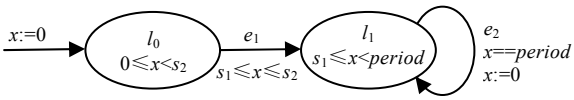

Fig.2 ITA model of the periodic interrupt

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.如图 3 所示,其中 *period* 值表示该中断两次触发之间的最小间距,其中,s_1 和 s_2 分别表示在系统开始运行后最少 s_1 时间、最多 s_2 时间之后中断事件允许发生.初始时,$x \in C$ 值为 0,偶发中断事件触发时,将 x 重置为 0,$\beta(e_1)=\beta(e_2)=I$.

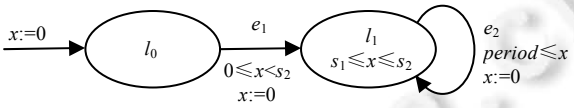

Fig.3 ITA model of the contingency interrupt

图 3 偶发中断时间自动机模型

1.3 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.

1.4 中断处理程序模型

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,原先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,也需要保证有些子处理过程及时完成.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.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.

在实际系统中,不同中断和系统任务之间通常通过控制变量相互关联.在常见的设计中,一些外部事件的处理会由中断和系统调度任务协调完成.外部事件对应的中断处理程序会设定某个控制变量,而系统任务则通过检查该控制变量来确定是否需要进行后续处理.如果忽略这样的关系,将会导致模型检验算法不能检测到由于错误使用控制变量而导致的系统设计错误.

此外,在中断处理程序和系统任务程序中,部分代码只有当控制变量取特定的值时才会被执行.也就是说,当控制变量取值不同时,系统任务执行的代码也有所不同.如果不考虑控制变量取值,我们只能给出这些处理程序的运行时间的粗略范围,这种粗略的时间范围会导致较多的误报.

为了更精确地表达系统模型,我们在模型系统中给出了控制变量的信息,而在中断/任务处理程序的模型中使用赋值语句和条件语句来改变和判断控制变量的值.

描述这些程序的语法如下:

Prog ::= *Statement*; *Prog*|*Statement*
Statement ::= *Subproc*()|*CloseInt*(*Integer*)|*OpenInt*(*Integer*)
|*VarName*:=*Integer*
|if (*Condition*) *Statement*; else *Statement*
|if (*Condition*) *Statement*
Condition ::= *VarName*==*Integer*

其中,*Subproc* 是子程序名字,*VarName* 是控制变量的名字.在模型中,建模人员需要给出各个子程序的基本信息,包括单独执行时的最短/最长时间、动态执行时的时间上界、对共享资源的访问信息等.这些信息和第 3 节中给出的中断处理程序子过程的信息相同.在程序中,人们可以用形如 *VarName*:=*Integer* 的控制遍历赋值语句给控制变量赋值,并在 if 语句中以 *VarName*==*Integer* 的方式来检测控制变量的值;然后,根据控制变量的当前值来判定应该执行哪个分支.*OpenInt*(*I*)和 *CloseInt*(*I*)分别表示开关中断 *I*.

例 2:假设在一个系统中中断事件 *event* 的处理和响应过程被分成两个阶段:当该事件对应的中断 *I* 到来时,相应的中断处理程序设置控制变量 V_e 为 1 并返回;当系统任务 $task_1$ 被调度运行时,如果 V_e 的值为 1,就执行对应于 *event* 的处理过程 *SubProc*.这个 $task_1$ 还可能通过执行其他代码来完成另外的任务.这样的处理方式在实际系统中很常见,因为中断处理程序(特别是高优先级中断的处理程序)必须在很短时间内执行完毕,否则可能造成中断丢失.那么,中断 *I* 的处理程序仅包含一个语句:V_e:=1;,而 $task_1$ 的代码被建模为

$SubProc_1$();
If (V_e==1)
{
SubProc();
V_e:=0;
}
$SubProc_2$();

假设 $SubProc_1$,*SubProc* 和 $SubProc_2$ 的执行时间区间分别是$[l_1,u_1]$,$[l_e,u_e]$,和$[l_2,u_2]$,使用带有 if 语句的建模方法可知:当 V_e==1 时,$task_1$ 的执行时间区间是$[l_1+l_e+l_2,u_1+u_e+u_2]$;当 V_e==0 时,$task_1$ 的执行时间是$[l_1+l_2, u_1+u_2]$.而如果不使用 if 语句,我们只能更加粗略地把 $task_1$ 的执行时间区间设定为$[l_1+l_2,u_1+u_e+u_2]$.这种粗略的设定虽然可以降低模型的复杂度,但却容易导致模型检验算法误报超时错误.

2 有界模型检验算法的基本框架

我们采用的有界模型算法的基本框架是:从系统的初始状态出发,以深度优先的方式枚举长度小于等于 *K* 的所有路径.在路径枚举的过程中,算法根据系统的形式化模型设定路径中的各个事件的发生时间必须满足的约束,并通过 SMT 求解器 Z3 来判断这些约束是否可解:如果可解,则表明该路径是可行的;如果无解,则表明该路径因为时间约束而导致不可行.当前路径可行时,算法就会判断当前路径是否违反了系统的规约.如果当前路径违反了系统规约,算法就会根据当前路径和它的时间约束生成相应的反例,并报告给用户.

下面我们首先描述系统不带时间的状态,然后给出算法的基本框架,而算法的细节将在后续的各节中详细描述.

2.1 不带时间信息的系统状态

有界模型检验算法在枚举路径时,首先根据当前状态计算出(不考虑时间约束时)所有可行的后继转换;然后再对时间约束进行求解,以确定该后继转换在时间上是否可行.因此在算法中,我们首先要确定中断驱动的控制系统的(不带时间信息的)状态包含哪些组成部分.

一个中断驱动的控制系统的模型主要包括中断时间/任务调度事件的模型、中断/任务处理程序的模型和共享变量的信息.同时,我们还需要考虑和系统运行相关的状态,比如中断开关情况、中断向量表的值、中断程

序的运行情况等等.因此,中断处理程序模型的(不带时间信息)的状态包括以下内容.

1. 各个中断时间自动机的当前状态.这些状态记录了与各个中断/任务调度事件相关的时间自动机的当前状态,在算法中使用数组 *Locs* 表示,其中,*Locs*[*i*]表示第 *i* 个自动机的状态;
2. CPU 栈 *Stack*.这个栈中保存了所有已经开始运行、但是(因为被高级中断打断而)尚未运行结束的中断/任务处理程序的运行状态.在栈中的程序按照优先级排列(系统调度任务的优先级最低),在栈顶的程序具有最高的优先级.栈顶的程序处于运行状态,而其余程序则处于挂起状态.栈中的每个元素是一个二元组(*Ip*,*pnt*),其中,*Ip* 表示某个中断处理程序,而 *pnt* 表示该中断处理程序中的某个程序点(即程序中某个语句之前或之后的位置).这个元素表示中断处理程序已经开始运行,且当前运行到 *pnt* 所对应的程序点.有关程序点的详细定义和描述见第 3 节;
3. 中断向量 *Vector*.这个向量模拟了实际系统中的中断向量表,表示了各个中断事件被触发的情况.当第 *i* 个中断被触发时,*Vector*[*i*]就被设置为 1;当第 *i* 个中断对应的处理程序开始运行时,*Vector*[*i*]被复位为 0.如果这个中断被触发时当前 *Vector*[*i*]的值为 1,就表明这个中断的前一次事件没有被及时处理,从而造成中断丢失;
4. 中断开关状态向量.状态中使用一个向量 *InterClosed* 来表示各个中断的开关情况.*InterClosed*[*i*]等于1,表明这个向量被关闭.这个向量中的元素会被中断处理程序中的 *OpenInt* 和 *CloseInt* 所改变:语句 *OpenInt*(*i*)将 *InterClosed*[*i*]的值设置为 0,*CloseInt*(*i*)将 *InterClosed*[*i*]的值设置为 1.当 *InterClosed*[*i*]=1 时,中断 *i* 的处理程序不会被执行;
5. 各个控制变量的取值.状态中使用一个向量 *Value* 保存了各个控制变量的当前值.对于第 *i* 个变量,*Value*[*i*]表示该变量的当前值.这些变量会被中断处理程序中的赋值语句改变,并且被中断处理程序中的 if 语句的条件表达式使用;
6. 共享资源的读写状态.状态中使用一个向量 *Resource* 来保存各个共享资源的状态.对于模型中的各个共享资源 *s*,*Resource*[*s*]的值是一个二元组(*R*,*W*),表示该进程被 *R* 个处理程序读、被 *W* 个处理程序写.显然,如果当前状态中某个共享资源 *s* 的 *R* 和 *W* 的值都大于 0,则表明当前状态发生了读-写冲突;*W* 的值大于 1,表明当前状态发生了写-写冲突.

因此,我们定义系统(不带时间)的状态为一个六元组〈*Locs*,*Stack*,*Vector*,*InterClosed*,*Values*,*Resource*〉,其中,

- *Locs* 是一个向量,对应于各个时间自动机的状态;
- *Stack* 对应于 CPU 运行栈,栈中的元素是相应处理程序的程序点;
- *Vector* 对应于中断向量表;
- *InterClosed* 对应于中断的开关状态向量;
- *Values* 对应于控制变量的取值,是从控制变量名字到整数值的映射;
- *Resource* 对应于各个资源的读写状态,是从共享资源名字到读写状态的映射.

2.2 有界模型检验算法的基本框架

我们的模型检验算法使用深度优先搜索来枚举所有的路径.对于每一条路径,算法生成这个路径对应的时间约束;然后,通过 SMT 求解器来判断这些约束是否有解以及这个路径是否可能违反系统的规约,包括是否超时、是否有共享资源冲突等.具体算法如图 4 所示.

这种算法有两个全局变量 *path* 和 *cons*,分别用于存放当前路径和路径相关的时间约束.其中,

- *path* 是形如(*e*,*s*)的二元组的序列:*e* 是一个转换事件,而 *s* 是和 *e* 对应的后继状态;
- 变量 *cons* 中存放了路径 *path* 对应的时间约束,这些时间约束规定了各个事件之间的时间约束.

算法 *BoundMC* 的参数 *length* 记录了需要搜索的路径长度.首次调用时,*length* 等于 *K*,即,需要检验的最长路径长度.当 *length*=0 时,路径长度已经到达 *K*,算法回溯.

算法首先枚举出 *path* 的最后状态 *s* 的所有可能的后继转换.对于每个后继转换 *e*,算法确定和 *e* 相关的所有时间约束,并加入到当前路径的约束中.通过 SMT 求解器,可判定这些约束是否有解:

- 约束无解,这表明该路径因为时间约束而不可行;
- 否则,算法开始判断该路径是否可能导致违反系统规约的情况:如果路径可能违反规约,算法通过 SMT 求解器求出各个事件发生的时间,构造反例并输出;如果路径不违反系统规约,算法递归调用自身来枚举更多的路径.递归调用结束后,算法恢复 *path* 和 *cons* 的值,并尝试状态 *s* 的下一个后继转换.

```
ListOfTuples(Transitions,States)        path;
SetOfConstraints                        cons;
BoundMC(int length)
{
    if (length==0)
        return;
    s=path 的最后状态;
    trans=s 的所有后继转换 e1,e2,…,em;
    for (s 中的每个转换 e)
    {
        根据 path,计算得到和 e 相关的约束集合 cons′;
        使用 SMT 求解器来判断 cons+cons′是否可解;
        if (cons+cons′无解)
            return;
        suc=GetSuccessiveState(s,e);
        将二元组(e,suc)添加到 path 的最后;
        判断是否可能出现共享资源冲突、中断丢失等错误,若可能,则给出反例并退出;
        生成和超时相关的规约约束集合 conSpec;
        使用 SMT 求解器判断 cons+cons′+NOT(conSpec)是否有解;
        if (cons+cons′+NOT(conSpec)有解)
            {根据 SMT 得到的解给出相应的反例; exit;}
        cons=cons+cons′;
        BoundMC(length−1);
        删除 path 的最后一个元素,即(e,suc);
        从 cons 中删除 cons′,即,所有和 e 相关的约束;
    }
}
```

Fig.4 Basic framework of the bounded model checking algorithm

图 4 有界模型检验算法的基本框架

本文将给出这个算法框架中的细节.第 3 节给出计算每个状态的所有后继转换及相应后继状态的方法.第 4 节给出如何计算得到当前转换对应的时间约束.第 5 节说明如何使用 SMT 求解器来处理约束和生成反例.

3 状态的后继转换以及相应的后继状态

在对中断驱动的控制系统进行模型检验时,我们关注 3 类事件,即:中断/任务调度事件、中断/系统任务处理程序开始或结束运行的事件和中断处理程序从一个程序点运行到下一个程序点的事件.这 3 类事件分别对应于时间自动机的转换、CPU 栈中的压栈/出栈操作以及栈顶元素中程序点的改变.在本节中,我们分别描述这些转换发生的条件以及这些转换发生之后新的状态.

3.1 中断程序开始执行的事件

当一个中断发生且相应的中断向量表被置位后,如果它的优先级高于正在运行的中断处理程序(或者 CPU 当前状态为空闲),且该中断的中断响应没有被关闭,那么当前正在运行的程序将会被打断,这个中断的中断处理程序会立刻开始运行.因此,如果当前状态的中断表中某个中断 *i* 被置位,且满足下列条件:

(1) *InterClosed*[*i*]=0,也就是说中断号 *i* 没有被屏蔽;

(2) 中断 *i* 的优先级高于 CPU 栈顶的处理程序的优先级;

(3) 中断 *i* 的优先级高于 *Vector* 中被置位的其他中断的优先级.

那么,当前状态的唯一后继转换是对应于中断 *i* 的处理程序开始运行的事件.根据中断执行的语义,相应的

后继状态按照如下方式计算得到:

(1) 中断向量表中对应于中断 i 的值被设置成 0;

(2) CPU 栈中压入一个新的元素(Ip,0),其中,Ip 是中断 i 的处理程序,而 0 是该处理程序的起始点;

(3) 根据程序点 0 对应的共享资源读写信息,修改相应共享资源的读取状态;

(4) 状态中的其余值不变.

3.2 中断事件/任务调度自动机的后继事件

在中断驱动控制系统的形式化模型中,中断源和任务调度事件采用中断事件自动机进行描述.当某个自动机 A 位于状态 l 上时,从 l 出发的所有转换都可能发生.如果发生的转换代表了中断发生的事件,那么相应的中断向量表被置位.

对应于第 i 个自动机的后继转换,就是该自动机中从状态 $Locs[i]$出发的自动机转换 e.根据自动机转换的语义,相应的后继状态按照如下方法计算得到:

(1) $Locs[i]$被改变为转换 e 的目标位置;

(2) 如果该转换 e 对应于某个中断事件的发生,则中断向量表中对应于该中断的标识位设置为 1;

(3) 状态中的其余值不改变.

3.3 与中断处理程序执行相关的后继转换和后继状态

如果有中断或系统任务尚未处理完毕,系统的CPU就会不断执行具有最高优先级的中断处理程序.相应地,形式化模型中 CPU 栈顶的中断处理程序将有状态变化(或者说,栈顶程序从当前程序点运行到新的程序点上).本节首先介绍中断处理程序的程序点的概念以及这些程序点之间的后继转换关系,然后介绍这些后继转换对应的后继状态的计算方法.

3.3.1 程序点以及程序点之间的后继关系

程序点是指各个语句(包括简单语句和 if 语句)之前和之后的点,我们的算法用这些点来表示中断/系统任务的处理程序的执行进度.程序位于某个程序点 p 上表明:

(1) 如果 p 位于某个原子语句(包括控制变量赋值语句、子过程调用语句和中断开关语句)之前,那么该程序正在执行 p 之后的原子语句;

(2) 如果 p 位于 if 语句之前,则该程序正在根据当前控制变量的值转向某个分支之前的程序点;

(3) 如果 p 位于 if 语句的某个分支之后,则程序正在转向 if 语句之后的程序点.

根据程序点的含义,我们可如下定义程序点之间的后继关系.

定义 1. 程序点的后继关系根据下面的规则来确定:

(1) if 语句之前的点有两个后继,分别是该 if 语句的 then 分支和 else 分支之前的点;

(2) if 语句的两个分支之后的程序点的后继都是 if 语句之后的程序点;

(3) 一个原子语句之前的程序点的后继是该语句之后的程序点;

(4) 一个顺序组合语句之前的程序点就是它的第 1 个子语句之前的程序点,而顺序组合语句之后的程序点就是它的最后一个子语句之后的程序点.对于顺序组合语句中两个相邻语句,前一个语句之后的程序点就是后一个语句之前的程序点.

例 3:图 5 的左边显示了一个处理程序的例子,右边的表格给出了这些程序点之间的后继关系.

给定一个处理程序,它的程序点集合以及程序点之间的后继关系可以通过语法分析程序扫描得到,这里不再赘述.

```
(0)
CV1 = 0;
(1)
if(CV2 == 3)
{
        (2)
        CV1 = 3;
        (3)
        subProc1();
        (4)
        subProc2();
        (5)
}
else
{
        (6)
        subProc3();
        (7)
}
(8)
subProc4();
(9)
```

程序点	后继
0	1
1	2,6
2	3
3	4
4	5
5	8
6	7
7	8
8	9
9	END

Fig.5 Program points of a handler program

图 5 处理程序的程序点及其后继关系

3.3.2 处理程序运行的后继状态

位于栈顶的中断处理程序占有 CPU,处于运行状态.因此,栈顶元素的程序点具有转向其后继程序点的转换.假设栈顶元素为(*Ip*,*i*),其中,*Ip* 是一个处理程序,且 *i* 是 *Ip* 中的一个程序点.下面我们根据程序点 *i* 的不同类型来计算其后继状态.

当程序点 *i* 位于一个控制变量赋值语句之前,假设该赋值语句为 *v*:=*c*,当程序继续运行时,控制变量 *v* 的值会被赋予新值 *c*.因此,其后继状态的计算很简单.

(1) 控制变量 *v* 的值设置成 *c*,其余变量的值不变;

(2) CPU 栈中的栈顶元素修改为(*Ip*,*j*),其中,*j* 是程序点 *i* 的后继,即,赋值语句之后的程序点;

(3) 如果程序点 *j* 之后是一个子过程调用语句 *proc*(),那么根据 *proc* 对共享资源的访问情况,设置相应共享资源的访问信息.例如:如果 *proc* 对资源 R_1 进行读访问,对资源 R_2 进行写访问,那么后继状态中,R_1 的读访问数和资源 R_2 的写访问数分别比原状态中的数值大 1;

(4) 新状态的其余各部分和原状态相同.

例 4:假设中断 I_1 的处理程序 Ip_1 如下:

```
(0)
v1=1;
(1)
Proc1();
(2)
```

这个程序有 3 个程序点.子过程 $Proc_1$()中对资源 r_1 进行读操作,对资源 r_2 进行写操作.如果当前栈顶元素为(Ip_1,0),且当前全局状态中的资源读写状态中,*Resource*[1]=(1,0),*Resource*[2]=(0,0),那么执行赋值语句 v_1=1 后,后继状态中栈顶元素为(Ip_1,1),控制变量 v_1 值为 1.Ip_1 位于程序点 1,表示程序已经开始执行子过程 $Proc_1$(),因此,资源读写状态中 *Resource*[1]=(2,0),*Resource*[2]=(0,1).

当程序点 *i* 位于一个子过程调用语句之前,此时程序继续运行,从程序点 *i* 到达其后继程序点 *j* 之后,该子过程调用执行结束,同时,该子过程对共享变量的访问也相应结束.因此,后继状态可按如下方法计算:

(1) CPU 栈中的当前栈顶元素修改为(*Ip*,*j*),其中,*j* 是 *i* 的后继,即,子过程调用语句之后的程序点;

(2) 根据子过程对共享资源的访问情况,修改各个共享资源的访问状态.例如:如果该子过程对资源 R_1 进行读访问,对资源 R_2 进行写访问,那么后继状态中,R_1 的读访问数和资源 R_2 的写访问数分别比原状态中的数值少 1;

(3) 如果新程序点 j 位于一个子过程调用语句之前,则需要相应地处理各个共享资源的访问状态.

例 5:给定前例中的处理程序 Ip_1,如果当前栈顶元素为(Ip_1,1),且在当前全局状态中的资源读写状态中,*Resource*[1]=(2,0),*Resource*[2]=(0,1),那么执行子过程调用语句 $Proc_1$()后,后继状态中栈顶元素为(Ip_1,2),且资源读写状态中 *Resource*[1]=(1,0),*Resource*[2]=(0,0).

当程序点 i 位于中断开关语句之前,那么在程序运行时,中断开关语句将打开或者关闭相应的中断响应.因此,后继状态的计算方法如下:

(1) CPU 栈中的当前栈顶元素修改为(Ip,j),其中,j 是 i 的后继,即开/关中断语句之后的程序点;

(2) 如果 i 之后的语句是 *CloseInt*(c),那么将状态向量 *InterClosed* 中对应于 c 的值设置为 1;如果 i 之后的语句是 *OpenInt*(c),那么将状态向量 *InterClosed* 中对应于 c 的值设置为 0;

(3) 如果 j 位于一个子过程调用语句之前,则需要相应地处理各个共享资源的访问状态.

例 6:假设给定中断 I_2 处理程序 Ip_2 的具体过程如下:

```
(0)
if (v1==1)
{     (1)
      OpenInt(I1);
      (2)
} else
{
      (3)
      Proc2();
      (4)
}
(5)
Proc3();
(6)
```

其中,子过程 $Proc_2$()中对资源 r_1 进行读操作,对资源 r_2 进行写操作,且中断 I_2 的优先级高于 I_1.如果当前栈顶元素为(Ip_2,1),且当前状态向量中 *InterClosed*[1]=1,那么执行中断开语句 *OpenInt*(I_1)后,后继状态中栈顶元素为(Ip_2,2),且当前状态中 *InterClosed*[1]=0,中断事件 I_1 对应的中断处理程序将可以被执行.

当程序点 i 位于 if 语句之前时,程序需要根据 if 语句的条件表达式来决定转向哪一个分支.假设 if 语句的控制表达式是 *conExp*,后继状态可按如下方式计算:

(1) 首先,根据当前状态中各个控制变量的值来计算 *conExp* 的值,并在 i 的两个后继程序点中进行选择.如果条件表达式 *conExp* 成立,那么下一个状态的程序点就是分支之前的程序点;否则就是 else 分支之前的程序点;

(2) 将 CPU 栈顶的元素修改为(Ip,j),其中,j 是上一步计算执行后得到的后继程序点;

(3) 如果新程序点 j 位于一个子过程调用语句之前,则需要相应地处理各个共享资源的访问状态.

例 7:给定前例中的处理程序 Ip_2,假设当前栈顶元素为(Ip_2,0),当前控制变量 v_1=0,且资源读写状态中 *Resource*[1]=(1,0),*Resource*[2]=(0,0).根据 if 语句的条件表达式,相应的后继程序点是 3.因此,状态中栈顶元素变为

(Ip_2,3),*Resource*[1]=(2,0),*Resource*[2]=(0,1).

当程序点 i 位于 if 语句的某个分支之后,程序继续运行时必然转向 if 语句之后的程序点.因此,后继状态计算方法如下:

(1) 将 CPU 栈顶的元素修改为(Ip,j),其中 j 是 if 语句之后的程序点;

(2) 如果新程序点 j 位于一个子过程调用语句之前,则需要相应地处理各个共享资源的访问状态.

例 8:给定前例中的处理程序 Ip_2,如果当前栈顶元素为(Ip_2,4),那么 if 分支执行完成,后继状态中栈顶元素为(Ip_2,5).

当程序点 i 是处理程序的出口程序点时,当前处理程序即将退出执行,而被当前中断处理程序打断的处理程序将恢复执行.其后继状态计算方法如下:

(1) CPU 栈顶元素(Ip,i)出栈;

(2) 新状态的其余各值和原来状态的值相同.

例 9:给定例 4 和例 6 中的处理程序 Ip_1 和 Ip_2,如果当前栈中有两个栈成员为(Ip_1,1)和(Ip_2,6),栈顶元素为(Ip_2,6),由于程序点 6 是 Ip_2 的出口程序点,因此 Ip_2 执行结束,栈顶元素(Ip_2,6)退栈,后继状态中的 *Stack* 仅包含栈成员(Ip_1,1).

4 时间约束和规约的处理

在枚举系统模型的所有路径时,算法不仅需要考虑第 3 节中描述的后继关系,同时还需要考虑路径中各个转换之间的时间约束.本节的有界模型检验算法在枚举路径的同时确定各个转换对应的时间约束,并通过 SMT 求解器来确定这些路径在时间上是否可行,并检查这些路径是否一定遵循时间规约.本节描述了不同种类的转换所对应的时间约束和时间规约的处理方式.

4.1 时间约束的表示

在本节的算法中,我们使用一系列的变量 $x_1,x_2,\ldots,x_k$ 来表示长度为 K 的路径中各个转换的发生时刻.这里,我们假设路径开始于 0 时刻.路径需要满足的各种时间约束可以使用有关这些变量的线性约束来表示.除了各种转换和状态需要满足的时间约束之外,这些变量还必须满足如下约束:

(1) 对于 i=1,2,…,K,约束 $x_i \geqslant 0$ 必须被满足.也就是说,路径中的转换都是在 0 时刻之后发生;

(2) 对于 i=1,2,…,K,约束 $x_i \leqslant x_{i+1}$ 必须满足.也就是说,路径中各个转换是顺序发生的.

除了这些基本的约束条件之外,我们还必须考虑与转换相关的时间约束和与状态对应的时间约束.

4.2 各类转换的时间约束

根据模型执行的定义以及对时间约束的处理方法,转换的时间约束可以分成 3 类.

(1) 时间自动机的转换的时间约束,也就是与中断发生/任务调度事件相关的时间约束;

(2) 中断/任务处理程序开始运行的时间约束;

(3) 中断/任务处理程序运行的时间约束.

下面我们详细描述这 3 类约束的生成方法.

4.2.1 时间自动机转换的时间约束

设当前路径长度是 i,当前转换属于某个中断自动机,且该转换的时钟测试条件是 $c \leqslant U$ 和 $c \geqslant L$.因为模型中的各个自动机都只有一个时钟,因此我们只需要找到当前路径中当前转换之前、重置该时钟的转换,设为路径的第 j 个转换(如果不存在这样的转换,表明该自动机一直没有被执行,此时可令 j=0).对于时间自动机的转换,需要添加时间约束:

$$x_i - x_j < U \;\&\&\; x_i - x_j > L.$$

4.2.2 中断开始运行的时间约束

设当前路径长度是 i,当前转换是某个中断/任务处理程序开始运行的时间.根据模型运行的定义,这个转换发生的条件必然是如下情况之一:

(1) 某个中断发生,且 CPU 栈为空或栈顶程序的优先级小于当前中断.此时,中断处理程序立刻开始运行;

(2) 中断发生时,CPU 栈顶程序的优先级高于此中断的优先级,那么该处理程序会紧接着较高优先级程序的运行结束之后开始运行;

(3) 中断发生时,相应的中断响应关闭,那么该处理程序会在中断响应打开之后开始执行.

在上述所有情况中,这个转换都是紧跟在前一个转换之后发生的(本文采用的模型不考虑中断上下文切换所需时间).因此,相应的时间约束如下:

$$x_i - x_{i-1} = 0.$$

4.2.3 中断处理程序运行的时间约束

在本节的模型中,中断/任务处理程序运行被建模为CPU栈顶的程序点的变化.当栈顶程序从一个程序点转换其某个后继程序点时,它需要占用一定的 CPU 执行时间.

由于子过程的执行过程可能会被更高级中断的处理程序打断,因此不能直接使用进入前驱/后继程序点的转换事件的时间差来表示该程序占用的 CPU 执行时间.我们使用这两个事件之间、该处理程序位于 CPU 栈顶的时间段之和来表示该程序占用的CPU执行时间.假设当前路径的第 j 个转换使栈顶程序进入程序点 p,而当前转换使这个中断处理程序从程序点 p 进入程序点 q,那么该中断处理程序在这两个转换之间占有的CPU处理时间就等于这个程序位于 CPU 栈顶的时间段的总和.我们要从第 j 个转换开始寻找该程序位于栈顶的所有状态,并用时间变量表示这些状态所占用的时间长度.图 6 给出了计算这个表达式的算法实例.该算法从首次进入程序点 p 的转换出发,不断寻找该中断处理程序位于栈顶的时间段的开始转换和结束转换,并使用这些转换的时间点的差值来表示其所占用的 CPU 时间.算法中调用的子函数 *GenerateEmptyExpression* 会生成一个空表达式,而子函数 *GenerateExpression* 根据第 1 个参数指定的运算符和后两个参数指定的运算分量生成相应的表达式.使用图 6 所示的算法得到了 CPU 执行时间的表达式之后,就可以生成有关处理执行的时间约束.

```
输入:当前路径;
      进入程序点 p 的转换的编号 i;
输出:表示该中断处理程序占用的 CPU 处理时间的表达式.

exp=GenerateEmptyExpression();
K=当前路径的长度;   //最后一个转换是第 K–1 个转换
prev=i;
cur=i+1;
while (cur<K)
{
      if (prev!=–1 and (第 cur 个转换之后的状态的栈顶程序不是该中断处理程序 or cur==K–1))
      {
            exp=GenerateExpression("+",exp,GenerateExpression("–",x_cur,x_prev));
            prev=–1;
      }
      if (第 cur 个转换之后的状态的栈顶程序是该中断处理程序 and prev=–1)
            prev=cur;
}
return exp;
```

Fig.6 Expression which represents the CPU time period occupied by the interrupt handler program

图 6 表示中断处理程序占用的 CPU 时间的时间变量

程序运行时,对变量的赋值和测试操作、开关中断的操作所需时间很小,与模型中的其他时间相比可以忽略不计.因此,如果程序点 p 到程序点 q 是下列情况之一,那么我们把 L 和 U 都设置为 0.

(1) p 是某个 if 语句之前的程序点,而 q 是 if 语句的某个分支的程序点;

(2) p 是某个 if 语句的某个分支之后的程序点,而 q 是 if 语句之后的程序点;

(3) p 和 q 分别是某个控制变量赋值语句之前和之后的程序点;

(4) p 和 q 分别是某个中断开关语句之前和之后的程序点.

对于上面的 4 种情况,相应的时间约束实际上是:*CPUTime_Exp*=0.其中,*CPUTime_Exp* 是使用图 6 所示算法

得到的、表示当前中断处理程序占用 CPU 执行时间的表达式.

中断处理程序的子过程需要执行一些复杂的操作,比如数据复制和传输、数值计算等.这些过程都需要占用一定的 CPU 时间,这些时间的上下界都已在建模时输入.假设 p 和 q 分别是子过程调用语句之前和之后的点,且该子过程执行时间的上下界分别是 L 和 U,那么相应的时间约束就是:$L \leqslant CPUTime\_Exp \leqslant U$,其中,$CPUTime\_Exp$ 是表示当前中断处理程序占用 CPU 执行时间的表达式.

4.3 状态对应的时间约束

当前转换发生时不仅需要遵循自身对应的约束,而且还要求当前状态下其他可能的后继转换可以在这个转换之后发生.因此,当前转换发生的时刻必须满足下面的两类约束:

(1) 如果当前转换是一个时间自动机转换,位于 CPU 栈顶的程序可以保持在当前的程序点,而且其他自动机也能够保持在当前状态;

(2) 如果当前转换是中断处理程序运行或者开始运行的转换,那么所有的时间自动机都可以保持在当前的状态.

下面讨论生成这两类约束的方法.

4.3.1 自动机状态的时间约束

对于系统中的每个自动机 A_i,假设当前状态中 A_i 位于 l_i 之上,且 A_i 中从 l_i 出发的转换的最大时间上界为 U.因为时间自动机 A_i 代表的中断必须不断发生,因此从 A_i 进入状态 l_i 的时刻到当前转换发生时刻的时间长度必须小于等于 U.所以,当前路径必须满足约束 $x_j - x_k \leqslant U$.其中,j 是当其路径的长度,x_j 代表路径的当前转换,而 k 是当前路径中使 A_i 最后一次进入位置 l_i 的转换的序号.

4.3.2 CPU 栈中的状态的时间约束

在当前状态中,位于栈顶的中断/任务处理程序处于运行状态.因此,从该程序进入当前程序点的时刻到当前转换发生时刻之间,这个中断/任务处理程序占用的 CPU 时间不得大于它进入下一个程序点所需要的最长时间.如果当前程序点位于 if 语句之前、if 语句的分支之后、控制变量赋值语句/中断开关语句之前,那么相应的最长时间是 0;如果当前程序点位于一个子过程调用语句之前,那么相应的最长时间就是该子过程单独执行时的时间上界.

我们可以使用图 6 所示的算法来获得栈顶程序到达当前程序点之后所获得的 CPU 时间的表达式 exp.而相应的时间约束就是 $exp \leqslant U$,其中,U 就是上面所描述的最长执行时间.

4.4 时间规约的处理

通过对当前路径的最后状态进行检查,就可以发现系统模型中包含的中断丢失、共享资源访问冲突等错误.除了这些错误,本节给出的有界模型检验算法还可以检验系统模型存在的超时问题.一个中断驱动的控制系统必须及时地响应中断事件.因此,路径中每个中断发生时刻到对应中断程序执行完毕时刻之间的时间差距必须小于这个中断的反应时间上界.这个时间上界就是建模时输入的各个中断的 $upbnd$ 参数.这些规约也可以被表示为路径中的时间变量的线性不等式.

假设当前路径的程度是 i,当前路径的第 j 个转换对应于某个中断事件的发生,且该中断对应的处理程序还没有执行完毕,我们就要求从第 j 个转换发生(即中断发生)到当前时刻之间的时间距离小于这个中断的处理时间上界 $upbnd$,相应的规约就是 $x_i - x_j \leqslant upbnd$.

对于枚举得到的每一条路径,算法对于路径中每个这样的转换都会生成相应的约束.系统模型的所有执行过程都必须满足这些时间规约.如果某个路径对应的某个执行不满足这些约束中的任意一个,那么,算法就找到了系统模型中存在的超时错误.

令 $Cons$ 表示当前路径的所有时间约束的集合,而 $Spec$ 表示该路径的时间规约,我们通过 SMT 求解器来检查 $Consx \wedge \text{NOT } Spec$ 是否有解.若这组条件有解,则表明该路径对应的某个执行可能不满足时间规约,算法会报告错误.

5 使用 SMT 求解器实现对时间约束和规约的处理

本节给出的有界模型检验算法使用深度优先的方式搜索所有的可行路径,并在遍历过程中对时间约束和规约进行处理.具体地:

(1) 算法在向前搜索时,把新的转换添加到当前路径,同时生成和这个新转换对应的时间约束,并判断整个路径的时间约束是否有解,以便确定新的路径在时间上是否可行;

(2) 对于每一条可行路径,算法把相关的时间规约的否定式加入到约束集合中并加以求解,以判断这条路径是否可能违反时间规约;求解完毕后,这些时间规约需要从约束集合中删除;

(3) 如果发现当前路径违反规约,算法将根据相应的约束计算出各个转换发生的具体时刻,并作为系统运行的反例反馈给用户;

(4) 算法在回溯时,把最后一条转换从当前路径中删除.此时,算法需要把那些与最后转换相关的时间约束从约束集合中删除.

SMT 求解器 Z3 能够很好地支持上述需求,因此,我们使用 Z3 作为算法的时间约束求解模块.

5.1 SMT求解器Z3相关命令的简介

Z3 是微软研究院研发的一个高效的 SMT 求解器,它可以检查一组包含多个领域理论(例如线性规划)的逻辑公式是否可以被满足.其输入命令格式遵循 SMT-LIB 2.0 标准,这些输入命令可以声明常量、输入约束、检查已经输入的约束是否可满足并给出满足约束的取值.同时,通过 *push* 和 *pop* 命令来修改栈中的命令集合.下面是在有界模型检验算法实现中用到的 SMT 求解器的相关命令的简介.

(1) 常量声明:我们可以使用命令(*declare-const X Float*)来声明一个浮点数类型的常量 *X*.需要注意的是,这里的常量和程序中的常量有不同的含义.这里对 *X* 的声明没有指定相关的值.SMT 求解器将检查是否可能给 *X* 赋予一个适当的值,以使得输入的约束集合都得到满足.因此,我们可以把模型检验算法中涉及的时间变量声明为 SMT 求解器中的常量;

(2) 约束:我们可以使用命令(*assert Prefix-Expression*)来输入一个约束.其中,*Prefix-Expression* 是这个约束的前缀表示形式.所有输入的命令都被 Z3 保存在一个内部栈中.例如,命令(*assert* (<(–*XY*) 0))把约束 $X–Y<10$ 加入到内部栈中;

(3) *push* 和 *pop* 命令:这两个命令可以帮助我们增减 Z3 的内部栈中的约束,其中,*push* 命令标记了栈中的一个范围,而 *pop* 命令则把栈顶到最靠近的 *push* 标记之间的所有约束(包括这个标记)全部删除掉;

(4) 可满足性检验命令:使用命令(*check-sat*)可以检验 Z3 的内部栈中的约束是否可满足;

(5) 获取模型取值的命令:如果(*check-sat*)命令表明 Z3 的内部栈中的约束是可满足的,使用命令(*get-model*)可以返回能够满足栈中所有约束的 SMT 常量取值.这些常量就对应于本章的模型检验算法中各个转换对应的时间变量.

使用这些命令,有界模型检验算法就可以在深度优先遍历系统路径的同时完成对时间约束和规约的处理.

5.2 使用Z3实现处理时间约束的方法

在图 4 的算法中,有两个地方需要使用 Z3 来处理时间约束.

(1) 在向当前路径添加一个新的转换时,需判断当前路径的约束加上和新转换相关的约束是否可满足;

(2) 在发现当前路径可行且没有发现资源共享等错误后,需要判断当前路径加上时间规约的否定形式是否可满足.

这两个处理都是用来检查一组约束集合的可满足性.算法只需要把相应的约束都输入到 Z3 的内部栈中,并使用 Z3 的 *check-sat* 命令就可以得到判断结果.在算法进行深度优先规约时,相应的约束集合的变化可以转换成栈操作.算法在 3 个地方需要改变被检验的约束的集合.

(1) 在把一个新转换添加到当前路径之后,算法把和这个新转换相关的约束添加到约束集合中;

(2) 在判断时间规约的时候,时间规约的否定形式被添加到这个约束集合中;在判断完毕之后,这个否定

式被从约束集合中删除;

(3) 在算法回溯时,最后一个转换的相关约束被从约束集合中删除.

这 3 个地方的执行顺序刚好是一个压栈/退栈的操作过程,因此,算法可以使用 Z3 中的 *push* 和 *pop* 命令方便地完成相应的操作.

(1) 在把新转换添加到当前路径之后,算法调用 Z3 的 *push* 命令标记新加入转换的范围,然后再把新转换相关的约束集合逐个加入到 Z3 的内部栈中;调用 Z3 的 *check-sat* 命令就可以判断当前路径是否可行;

(2) 在判断规约的时候,算法仍然先调用 Z3 的 *push* 命令,把规约的否定式加入 Z3 的内部栈,并调用 *check-sat* 判断可满足性;判断完成后,调用 *pop* 命令就可以把规约的否定式从内部栈中删除;

(3) 在算法回溯时,调用 *pop* 命令就可以把在步骤 1 中加入的、与路径最后转换相关的约束被从 Z3 的内部栈中删除.

上面的处理办法把约束集合存放在了 Z3 求解器的内部栈中,因此,图 4 中算法的变量 *cons* 可以省略,直接使用内部栈存放这些约束就可以完成算法的功能.

6 算法实现和反例显示

我们使用 Java 语言实现了本节描述的算法,得到了一个算法的原型工具.该工具要求设计人员输入中断及系统任务有关的时间信息以及中断处理和任务处理的伪代码,然后转换成为本文描述的形式化模型和规约.对这个形式化模型进行验证之后,工具可以输出相应的验证结果.在验证过程中,该工具调用 Z3 的 Java API 来实现对时间约束/规约的求解.如果该工具在检验模型时发现错误路径,则输出相应的反例,并可以使用轨迹图的方式直观地显示出来.轨迹图的 X 轴代表时间,而 Y 轴对应于不同中断的处理过程,并使用实线表示 CPU 运行各个处理过程的情况.这些反例可以有效地帮助设计人员发现并改正系统中的错误.

例 10:本例中的模型是由实际的航天控制系统经过抽象和简化而得到的.为了演示反例输出,我们在这个系统模型中手工植入了超时错误:系统第 i 个任务 $task_i$ 和两个中断 I_1 和 I_2.任务 $task_i$ 的执行周期为 800ms,即 U_{task}=800ms;中断 I_1 为周期中断,其周期为 400ms,即 U_{I1}=400ms;中断 I_2 为偶发中断,其 U_{I2}=150ms,I_2 具有较高优先级.这些任务和中断的处理程序(以及相应的程序点)见表 1.在 $task_i$,I_1 和 I_2 对应的处理程序 Ip_0,Ip_1 和 Ip_2 中,各个子处理函数的执行时间范围见表 2.

Table 1 Handler of task $task_i$, interrupt I_1 and I_2

表 1 任务 $task_i$ 及中断事件 I_1,I_2 的处理过程

$task_i$ (Ip_0)	Ip_1	Ip_2
(0)	(0)	(0)
$proc_1$();	v_2=1;	v_1=1;
(1)	(1)	(1)
If (v_1==1)	If (v_1==1)	$proc_7$();
{ (2)	{ (2)	(2)
$proc_2$();	$proc_5$();	
(3)	(3)	
v_1=0;	} else	
(4)	{ (4)	
}	$proc_6$();	
(5)	(5)	
If (v_2==1)	}	
{ (6)	(6)	
$proc_3$();		
(7)		
v_2=0;		
(8)		
}		
(9)		
$proc_4$();		
(10)		

Table 2 Execution time of subprogram of Ip_0, Ip_1 and Ip_2

表 2 Ip_0,Ip_1 和 Ip_2 中子处理函数的执行时间范围

	$proc_1$	$proc_2$	$proc_3$	$proc_4$	$proc_5$	$proc_6$	$proc_7$
b	64	96	48	112	160	120	40
w	80	120	60	140	200	150	50

使用本文给出的有界模型检验方法遍历这个简化的模型系统,将会报告如下反例:在 $task_i$ 的执行周期内依次触发中断事件 I_2,I_1,I_1,I_2,将可能导致 $task_i$ 对应的处理程序可能无法在规定时间 800ms 内执行完成.该反例具体的栈顶处理程序的执行轨迹如图 7 所示,初始时栈为空,随着时间的流逝,依次触发了自动机中代表任务/中断开始事件的转换事件:$task_i$,I_2,I_1,I_1,I_2,并执行了相应的处理程序.图 7 中的示意点依次标识了栈中处理程序被挂起时当前正在执行的程序点.

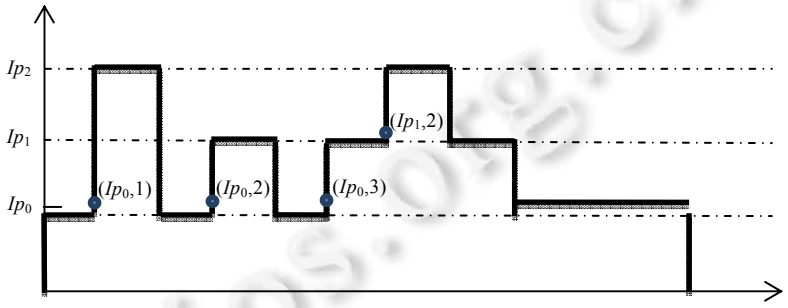

Fig.7 Execution trace of handlers in CPU stack

图 7 CPU 栈中中断处理程序执行轨迹

由于问题本身的复杂性,模型检验的时间/空间爆炸问题仍然存在,工具验证模型时所需要的时间和中断/系统任务的数量依旧呈指数关系.但是,我们的工具仍然具有较好的实用性.典型的航天控制嵌入式系统的规模大约是不多于 7 个中断和 4 个系统任务,我们使用这个工具对某航天器的着陆器控制系统的模型进行了验证,该控制系统包括 6 个中断和 4 个系统任务.中断的功能包括姿态控制、时间校准等,其中的 3 个中断通过控制变量和系统任务的处理过程相互作用.对于这个典型规模的控制系统,工具使用了大约 4 个小时的时间验证了系统模型中长度小于等于 20 的所有路径,表明了该系统的时序正确性.同时,对于在系统中人工植入的时间错误,该工具都能够检测到并给出相应的反例.

7 相关工作和总结

针对中断驱动系统,学者们提出了不同的建模和验证方法:在文献[8,9]中,作者给出了针对 Z86 中断驱动软件系统的超时分析工具,Z86 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.文献[8,9]中的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[10]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调度情况,且系统中缺少时间相关信息.文献[11]提出对实时嵌入系统的建模及检验的方法,给出结合静态分析、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.

同时,有界模型检验被广泛应用于实时系统的时序性质验证中,用于提高验证效率,降低算法所需的空间时间复杂度:在文献[12]中,使用时间自动机或带时间的 Kripke 结构对实时系统进行建模,使用 TECTL[13]描述系统中的时序逻辑,然后给出了基于 SMT 的有界模型检验方法.文中给出的方法可以高效地完成对实时系统的有界

模型检验,但无法描述中断事件这类复杂的系统行为.在文献[14]中,使用时间自动机对实时系统进行建模,扩展了线性时序逻辑 MITL[15]公式用于定量地规约事件能否在规定时间内完成,并给出了基于有界模型检验的正确性验证方法.在文献[16]中,首先使用 Petri 网对嵌入式系统中的中断嵌套行为进行建模;然后给出了从带优先级的时间 Petri 网到时间自动机的转换规则;随后,使用 SMT 求解器对模型系统地进行了有界模型验证.该文中给出的验证方法分别给出了单个中断事件执行的 Petri 网模型、中断嵌套执行的 Petri 网模型以及中断并发执行的 Petri 网模型,然后给出了针对某条特定执行轨迹的建模系统的可达性分析及时间约束相关验证,无法自动检验系统中所有的可行路径.此外,文中给出的算法将 Petri 网中的每一个转换建模为一个时间自动机,这使得模型系统的复杂性较高.下推(pushdown)自动机[17,18]可以用来表示中断处理过程中出现的嵌套情况.但是这些模型都没有对系统任务和中断处理过程中的交互情况进行建模.

本文针对一类中断驱动的航天控制系统给出了建模方法和针对超时问题的有界模型检验算法.本文首先给出了由任务和中断事件组成的中断驱动系统模型,其中,中断事件和任务可以通过控制变量相互关联;然后,确定了系统的无时间状态和相应的后继转换以及针对各类后继转换计算后继状态的方法;随后给出了这类转换和状态对应的时间约束和时间规约的生成方法;最后讨论了使用 SMT 求解器来处理时间约束和时间规约的方法.

本文给出的有界模型检验算法可以检查从初始状态出发的、长度不大于 K 的所有系统路径是否可能违反系统规约.根据我们对已知错误案例的分析,大部分的错误都可以在 20 步内被发现.本文给出的有界模型检验算法可以包括较为详细的中断/任务处理程序的描述,并且相对于需要遍历所有状态空间的传统可达性分析算法具有较高的效率.

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. In: Inan MK, Kurshan RP, eds. Verification of Digital and Hybrid Systems. 2000, 170:265−292. [doi: 10.1007/978- 3-642-59615-5_13]

[6] Biere A, Cimatti A, Clarke E, Zhu YS. Symbolic model checking without BDDs. In: Proc. of the Tools and Algorithms for the Construction and Analysis of Systems. LNCS 1579, Berlin, Heidelberg: Springer-Verlag, 1999. 193−207. [doi: 10.1007/3-540-49059-0_14]

[7] Amla N, Kurshan R, McMillan KL, Medel R. Experimental analysis of different techniques for bounded model checking. In: Proc. of the Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2619, Berlin, Heidelberg: Springer-Verlag, 2003. 34−48. [doi: 10.1007/3-540-36577-X_4]

[8] 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]

[9] 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]

[10] 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]

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

[12] Xu L. SMT-Based bounded model checking for real-time systems. In: Proc. of the 8th Int'l Conf. on Quality Software. Oxford, 2008. 120−125. [doi: 10.1109/QSIC.2008.10]

[13] Alur R, Courcoubetis C, Dill DL. Model-Checking in dense real-time. Journal of Information and Computation, 1993,104(1):2−34. [doi: 10.1006/inco.1993.1024]

[14] Kindermann R, Junttila TA, Niemela I. Bounded model checking of an MITL fragment for timed automata. In: Proc. of the 13th Int'l Conf. on Application of Concurrency to System Design (ACSD). 2013. 216−225. [doi: 10.1109/ACSD.2013.25]

[15] Alur R, Feder T, Henzinger TA. The benefits of relaxing punctuality. Journal of the ACM, 1996,43(1):116−146. [doi: 10.1145/227595.227602]

[16] Hou G, Zhou KJ, Chang JW, Li R, Li MC. Interrupt modeling and verification for embedded systems based on time Petri nets. In: Proc. of the 10th Int'l Symp. on Advanced Parallel Processing Technologies. LNCS 8299, Berlin, Heidelberg: Springer-Verlag, 2013. 62−76. [doi: 10.1007/978-3-642-45293-2_5]

[17] Bouajjani A, Esparza J, Maler O. Reachability analysis of pushdown automata: Application to model-checking. In: Proc. of the CONCUR'97: Concurrency Theoy, Vol.1243. Berlin, Heidelberg: Springer-Verlag, 1997. 135−150. [doi: 10.1007/3-540-63141-0_10]

[18] Jia XX, Liu C, Wu J, Liu YP, Jin MZ, Liu C. A novel Web test runner based on pushdown automation. Computer Science, 2006,33(4):269−273 (in Chinese with English abstract). [doi: 10.3969/j.issn.1002-137X.2006.04.077]

附中文参考文献:

[18] 贾晓霞,刘昶,吴际,柳永坡,金茂忠,刘超.一个基于下推自动机的 Web 测试自动执行器.计算机科学,2006,33(4):269−273. [doi: 10.3969/j.issn.1002-137X.2006.04.077]

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

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

顾斌(1968－),男,博士,研究员,博士生导师,CCF 高级会员,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.

杨孟飞(1962－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为航天器设计,空间自动控制.