软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(11):2452−2472 [doi: 10.13328/j.cnki.jos.004609]

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

使用事件自动机规约的 C 语言有界模型检测*

阚双龙, 黄志球, 陈 哲, 徐丙凤

(南京航空航天大学 计算机科学与技术学院,江苏 南京 210016)

通讯作者: 阚双龙, E-mail: kanshuanglong@nuaa.edu.cn

摘 要: 提出使用事件自动机对 C 程序的安全属性进行规约,并给出了基于有界模型检测的形式化验证方法.事件自动机可以规约程序中基于事件的安全属性,且可以描述无限状态的安全属性.事件自动机将属性规约与 C 程序本身隔离,不会改变程序的结构.在事件自动机的基础上,提出了自动机可达树的概念.结合自动机可达树和有界模型检测技术,给出将事件自动机和 C 程序转化为可满足性模理论 SMT(satisfiability modulo theory)模型的算法.最后,使用 SMT 求解器对生成的 SMT 模型求解,并根据求解结果给出反例路径分析算法.实例分析和实验结果表明,该方法可以有效验证软件系统中针对事件的属性规约.

关键词: 事件自动机;可满足性模理论;有界模型检测;自动机可达树;安全关键软件

中图法分类号: TP301

中文引用格式: 阚双龙,黄志球,陈哲,徐丙凤.使用事件自动机规约的 C 语言有界模型检测.软件学报,2014,25(11):2452−2472. http://www.jos.org.cn/1000-9825/4609.htm

英文引用格式: Kan SL, Huang ZQ, Chen Z, Xu BF. Bounded model checking of C programs using event automaton specifications. Ruan Jian Xue Bao/Journal of Software, 2014,25(11):2452−2472 (in Chinese). http://www.jos.org.cn/1000-9825/4609.htm

Bounded Model Checking of C Programs Using Event Automaton Specifications

KAN Shuang-Long, HUANG Zhi-Qiu, CHEN Zhe, XU Bing-Feng

(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Corresponding author: KAN Shuang-Long, E-mail: kanshuanglong@nuaa.edu.cn

Abstract: In this paper, a technology is presented to use event automata to specify the safety properties of C programs and apply bounded model checking to verify whether a C program satisfies an event automaton property. An event automaton can specify a safety property which is based on the events generated by a program. It can also specify a property with infinite states. Since an event automaton separates from C programs, it will not change the structures of programs. The paper introduces the definition of an automaton reachability tree based on an event automaton. It then uses automaton reachability trees and the bounded model checking to build the SMT (satisfiability modulo theory) models of event automata and C programs. Finally, it supplies the SMT models to an SMT solver. An algorithm for generating counterexamples is obtained according to the results of the solver. The case studies and experimental results demonstrate that the presented approach can verify the event properties of software systems.

Key words: event automata; satisfiability modulo theory; bounded model checking; automaton reachability tree; safety critical software

近年来,随着软件在航空航天、高铁和医疗等领域所占的比重越来越大,提高软件的安全性和可靠性成为工业界和学术界关注的热点.传统的测试仿真技术在保证软件可靠性方面发挥着巨大的作用,但是对于安全关键软件,传统技术没有穷尽程序输入,无法遍历所有的执行路径,因此难以发现隐藏极深的软件错误.而且对于并发、实时等不确定执行系统,这类技术无法起到很好的作用.因此,需要引入形式化验证方法来保证系统的安

* 基金项目: 国家自然科学基金(61272083, 61100034)
收稿时间: 2013-07-30; 修改时间: 2013-12-03; 定稿时间: 2014-03-28

全性.模型检测[1]作为一种形式化验证技术,已经在硬件设计[2]和网络协议[3]方面取得了重大的突破.将模型检测技术应用到软件验证领域,尤其是对软件代码的验证,是学术界近10年的研究难点和重点.

软件模型检测技术目前主要有以下几类:(1) 基于完全状态搜索(explicit model checking)的模型检测[1]技术,如贝尔实验室 Godefroid 等人开发的 Verisoft[4],该工具主要验证并发和实时软件系统;(2) 符号模型检测(symbolic model checking)技术[5],包括使用BDD的符号模型检测[5]和使用SAT(Boolean satisfiability problem)/SMT的符号模型检测[6],如McMillan提出的基于SAT的符号模型检测[6];(3) 基于抽象解释[7]的模型检测技术,包括谓词抽象[8](predicate abstraction)和惰性抽象[9](lazy abstraction),如微软的 Ball 等人基于谓词抽象开发的SLAM[10]和加州大学伯克利分校采用惰性抽象技术开发的 BLAST[11];(4) 通过限制程序执行路径长度的有界模型检测(bounded model checking)技术,如卡内基梅隆 Clarke 等人开发的 CBMC[12]以及在其上开发的ESBMC[13]和LLBMC[14].

软件模型检测主要存在以下两个问题:

(1) 对于无限状态的程序模型,由于状态的无限性,模型检测问题本身就是不可判定问题;

(2) 对于有限状态的软件系统,状态空间爆炸是一个非常重要且难以解决的问题.

针对以上问题,文献[12,15]提出了有界模型检测,对程序的递归调用层数和循环的次数都设定一个上界,这样程序必然终止,可以将不可判定问题转化为可判定问题.软件抽象解释技术[7]对于将软件无限状态转化为有限状态以及约简软件的状态空间具有非常重要的作用,例如,谓词抽象[8,16]是实现软件抽象自动化的非常成功的技术.SATABS[17]是采用谓词抽象技术开发的C语言模型检测工具.布尔可满足问题求解技术SAT[18]的发展,促使Clarke等人提出基于命题逻辑的C程序符号模型检测[12].将C语言程序与所验证的属性转化为命题逻辑公式,然后利用 SAT 求解器(SAT solver)对公式进行求解,如果发现公式可满足,则找到程序的一条错误执行路径.可满足性模理论SMT(satisfiability modulo theory)[19]是在一阶逻辑的基础上对数组、记录和算术运算等语义进行解释的判定过程.SAT/SMT求解器提高了模型检测的执行效率,文献[20−22]表明,基于SAT和SMT的软件形式化验证技术现在已经成为国内外学者的关注重点.

除了以上的验证技术以外,对于程序属性的规约也是模型检测的重要方面,例如,采用线性时序逻辑 LTL(linear temporal logic)[23]和计算树逻辑CTL(computation tree logic)[23]作为模型属性的规约语言.现在对C语言代码验证的属性规约可以分为两种类型:第1种是程序本身所固有的运行时属性,例如数组越界、算术除零和指针非法使用等错误;第2种错误是用户自定义错误,与应用语义相关,例如文件操作必须先打开后关闭、队列操作必须先进先出等.针对第2类属性的规约方法,现在的验证工具主要采用断言的形式.在所需验证的程序位置加入 *assert*(*expression*),判断该断言在所有执行路径下是否成立.

基于断言的属性规约描述了从程序的初始状态到某个状态集合的可达性.一个程序的状态可以表达为$(pc,var_1,var_2,\ldots,var_n)$,其中,$pc$ 为程序当前执行位置,$var_1,var_2,\ldots,var_n$ 为程序的所有变量.程序的初始状态就是 pc 指向程序起始位置,并且所有的变量值都为初始值.可达集合是 pc 指向程序的指定位置,并且变量满足一定的约束.这里的约束是一个谓词表达式,例如 $x>4\wedge y<4$,也就是所有的变量值满足 $x>4\wedge y<4$.在程序的位置 l 插入断言 *assert*($x>4\wedge y<4$),就表达了一个程序的状态集合.CBMC[12],BLAST[11]和 SATABS[17]等模型检测工具都采用断言作为用户自定义属性的规约形式.

但是,基于断言的属性规约具有一定的局限性:

(1) 无法表达无限状态的安全属性,例如队列的先进先出属性;

(2) 对于程序的一条执行路径,我们希望表达整条路径应该满足的属性,比如路径执行必须先满足命题 p 再满足命题 q,而可达性只表达最终到达状态的属性;

(3) 缺乏对程序中事件的表达.

针对以上的问题,结合近年来发展的运行时验证技术[24−26],本文在断言的基础上提出了带变量的事件自动机[27],对 C 语言属性进行规约.事件自动机带有变量,自动机发生转移的前提是事件的发生.事件自动机具有更强的表达力,可以描述事件发生的时序,可以描述无穷状态的安全属性.事件自动机是独立于程序本身的,因此

与程序分离不会影响程序自身的结构.在嵌入式和操作系统等多事件系统中,该方法将有利于对系统属性规约.

在事件自动机的基础之上,我们采用基于 SMT 的有界模型检测作为程序验证技术基础,首先使用文献[20]提出的方法将 C 程序转化为 SMT 模型,然后给出了自动机可达树的概念.通过自动机可达树,将事件自动机与 C 语言程序之间的约束关系转化为 SMT 模型.通过 SMT 求解器判断程序是否违反事件自动机所规约的属性,最后,根据求解器返回结果给出生成程序反例路径的算法.最后的实例分析表明:本文方法可以检测像变量溢出等底层错误,并且可以验证像队列先进先出等断言无法表达的属性.

本文第 1 节给出事件自动机的语法和语义规则及用事件自动机对程序属性规约的实例.第 2 节给出基于事件自动机规约的有界模型检测技术.第 3 节给出两个使用本文方法进行形式化验证的实例,并通过实验分析本文方法的执行效率.第 4 节给出结论.

1 使用事件自动机规约 C 程序安全属性

本节给出事件自动机的形式化定义、描述语言及其生成算法.

1.1 事件自动机的形式化定义

事件(event)在程序设计语言中可以定义为一条语句的执行或一个函数的调用.每一个事件都有一个标识作为事件的名称,本文使用Σ表示事件名的集合.

在程序中经常使用变量和常量,我们用 Var 和 Val 分别表示变量的集合和常量的集合.变量和常量统一称为符号量,用 Sym 来表示,$Sym=Var\cup Val$.我们首先给出本文对事件的定义:

定义 1(事件). 一个事件定义为 $(e,\overline{p})\in\Sigma\times Sym^*$, 或写为 $e(\overline{p})$,$e\in\Sigma$为事件名, $\overline{p}\in Sym^*$ 为事件参数序列.

根据事件的参数将事件分为两类:

1. 若 $\overline{p}\in Val^*$, 则该事件为实参事件,即,所有的参数都是常量;
2. 若 $\overline{p}\in Sym^*\times Var^+\times Sym^*$, 则该事件为形参事件.

$\overline{p}[i]$ 为 $\overline{p}$ 的第 i 个参数,对于参数个数相同的形参事件 $e(\overline{p})$ 和实参事件 $e(\overline{p}')$, 如果满足以下任意一个条件,则 $e(\overline{p})$ 和 $e(\overline{p}')$ 匹配:

(1) 若 $\overline{p}[i]\in Val,\overline{p}[i]=\overline{p}'[i]$;

(2) 若 $\overline{p}[i]\in Var,\overline{p}[i]$和$\overline{p}'[i]$类型一致.

对于程序的一次运行来说,所有变量的值都是确定的,所以产生的事件也都是实参事件.但因为程序的输入是不确定的,因此,使用变量可以让规约的表达能力更强.例如,对于文件操作 open 和 close,必须满足每一个 open 操作之后都要有一个 close 操作.使用正则表达式表达为$(\text{open}(f)\ \text{close}\ (f))^*$,这里的 f 为一个变量,它代表任意一个文件名,该正则表达式规约了任意文件的行为.

事件自动机是基于事件驱动转移的、监视源程序行为的一类自动机,事件自动机的变量机制可以增强自动机的表达能力.我们首先给出事件自动机的形式化定义:

定义 2(事件自动机). 事件自动机是一个六元组:$EA=(Q,Var,A,T,s_0,F)$,其中,

(1) Q 是自动机状态的有限集合.

(2) Var 为自动机变量的集合,自动机变量记录自动机转移中事件的信息,在自动机执行前部分变量要赋初始值.

(3) A 为事件的有限集合,自动机只有在接收到 A 中的事件才会发生转移,带有形参的事件中形参必须是自动机变量.

(4) $T\subseteq Q\times A\times Guard\times Assign\times Q$,其中:$Guard$ 是一个谓词表达式,表示转移能够发生的条件,它是由自动机变量和所监视程序中的变量组合的一阶谓词逻辑公式;$Assign$ 为一组赋值语句,赋值语句中所有的写操作都是针对自动机变量,所监视的程序中变量是只读的.我们用$\delta(s_i)$表示一个自动机状态的所有下一个状态的集合,即,对任意状态 $s_j\in\delta(s_i)$,都存在转移$(s_i,e,guard,assign,s_j)$.

(5) s_0 为初始状态,对于有多个初始状态的自动机,可以很容易地转化为只有 1 个初始状态的自动机.

(6) F 为接受状态集合,所有进入接受状态的程序执行路径都是反例路径.

在事件自动机中,我们定义了自动机变量集合.在自动机的状态转移过程中,对自动机变量进行写操作的方式有两种:第1种是通过转移函数中的 *Assign* 序列可以对自动机变量执行写操作;第2种方式是通过实参事件参数对变量进行绑定赋值.例如,事件自动机有一个转移(s_1,*start*(x),true,*empty*,s_2),当接收到一个事件 *start*(2)时,则将 x 赋值为2,记为[x↦2],这样就对自动机变量进行了赋值.

转移函数是事件自动机的核心概念.事件自动机转移发生的条件是:所监视的程序中出现了驱动事件,并且满足转移的 *Guard*.然后,执行赋值操作序列 *Assign*.如果在程序执行过程当中没有发生自动机所规约的事件,则自动机不发生任何行为.这与线性时序逻辑(LTL)对模型属性进行规约不同,模型每发生一次转移,由 LTL 转化为的 Büchi 自动机也要发生一次转移.

定义3(绑定). 一个绑定 *Bind* 是对一组变量的赋值,定义为一个映射,即 *Bind*:*Var*→*Expr*.*Expr* 为表达式集合,它由 *Sym* 中的符号组成.大多数情况下,绑定到一个变量的值为一个常量.这里,为了后面算法的表达方便,将其泛化为表达式.*Binds*(*Var*)表示一个绑定的集合,该集合中任意一个绑定都是对 *Var* 中的变量进行赋值.

定义 4(配置). 我们引入配置的概念来实例化事件自动机的一次运行,事件自动机的一个配置 *Config* 定义为一个状态和自动机变量一个绑定的组合,即 *Config*∈Q×*Binds*(*Var*).事件自动机的转移都是建立在配置上的,对于一个转移(s, $e(\overline{p})$,*Guard*,*Assign*,s'),并且在状态 s 的情况下绑定为φ,转移可发生的条件:

- 当程序中产生一个与 $e(\overline{p})$ 匹配的实参事件 $e(\overline{p}')$, 其中, $\overline{p}' \in Val^*$, 我们将对应的实参 $\overline{p}'$ 中的参数值按顺序分别绑定到形参 $\overline{p}$ 中的变量,得到新绑定φ';
- φ'必须满足转移中的 *Guard*,否则,转移无法发生,若转移不发生,则绑定φ'将被取消;
- 在转移可以发生的前提下执行 *Assign* 赋值序列,产生新的绑定φ''.

例如自动机接收事件 *start*(x),若程序产生事件 *start*(2),则生成绑定 x↦2.如果新绑定满足 *Guard*,*Assign* 也将对自动机变量进行操作并产生新的绑定;如果 *start*(2)不满足 *Guard*,则不被自动机接收.

定义5(事件自动机的事件序列). 对于事件自动机 *EA* 和一个事件序列 $e_0e_1\ldots e_{n-1}$,假设自动机变量初始绑定 φ_0.如果存在 *EA* 的配置序列(s_0,φ_0)(s_1,φ_1)…(s_n,φ_n),对于任意 0≤i<n,存在转移(s_i,e_i,*Guard*$_i$,*Assign*$_i$,s_{i+1})可发生,并且 s_n∈F,则事件自动机 *EA* 接收该事件序列.

定义 6(程序中事件的生成). 我们定义程序中一条语句的执行完成为一个事件的发生,所产生事件的名称可以为程序中某个函数的名称,也可以是我们自定义的名称.事件的参数可以是程序中的表达式,也可以是一个常量.将某个事件与一条执行语句绑定,表示当程序执行完该语句之后,程序将生成一个事件传递给事件自动机,事件自动机将会根据事件流对程序行为进行监控.

图1给出了对函数 *sum* 绑定事件 *sum*(*a*,*b*)的示例,在执行完最后一条语句 return s 之后将产生该事件,每当该函数被调用执行一次之后都会产生该事件.这里采用在注释中加入绑定事件的方法,不会影响程序的固有结构.

```
int sum(int a,int b){
    int s=a+b;
    return s;   /* $event{sum(a,b)} */
}
```

Fig.1 Binding of an event to a program

图1 事件到程序的绑定

本文事件自动机中的自动机变量和事件的参数类型限制在整型(int)、实型(real)和布尔型(bool).C 程序中也会存在字符串类型的变量,在绑定事件的时候可以做一个映射,即,将字符串型变量映射到整型.另外,事件自动机变量不可以为指针类型,因为指针类型如果指向程序中的变量地址,会导致自动机转移修改程序的变量,这在监视过程中是不允许的.

为了方便表达,本文引入两个专有事件:

- 第1个是结束事件:terminal,因为对程序中所有的循环和递归加了上界,所以程序必然是终止的,在编码

过程中,编码的最后产生一个结束事件.

- 第 2 个是完全事件:all,完全事件可以接收任何从程序产生的事件,也即当程序产生任意事件时,事件自动机的完全事件可以无条件匹配该事件.

对于事件的参数,我们引入"*"表达可以匹配任何参数.例如,*start*(*,*y*)表示接受任何事件名为 *start*,第 1 个参数为任意值,第 2 个参数赋值给 *y* 的事件.下面给出两个事件自动机规约的实例.

图 2 给出了一个描述文件操作的事件自动机实例,它有 3 个状态:s_0 是初始状态,s_1 是在打开一个文件之后的状态,$s_2 \in F$ 是接受状态.当自动机接收到一个 open 事件之后,就将 open 中的实参赋值给自动机变量 *f*.显然,自动机描述了所有打开一个文件之后,但是程序终止时却没有关闭文件的执行路径.在状态 s_0 使用了完全事件,它的好处在于:当程序同时打开多个文件时,可以只用这 1 个事件自动机进行表达.对于自动机中没有 *guard* 和赋值序列的转移,我们在图中忽略,没有标出.

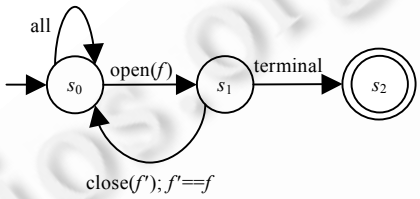


Fig.2 Event automaton of file operations

图 2 文件操作的事件自动机

图 3 给出了对一个队列先进先出(FIFO)属性进行规约的事件自动机,队列有两个操作:*insert*(*x*)将 *x* 插入到队列的末尾,*delete*(*y*)将队列头部数据删除并保存到变量 *y* 中.在初始状态,我们可以接收任意多个插入事件,但是一旦选择转移到状态 s_1 之后,则只接收删除事件,而忽略插入事件.通过计数变量 *cnt*,我们可以判断当 *cnt*==0 时,再发生删除操作并且删除的数据不等于 *x*,则必然不满足 FIFO 属性.因为规约的是错误属性,也就是说,当删除的不是 *x* 时,我们找到了一条错误路径.

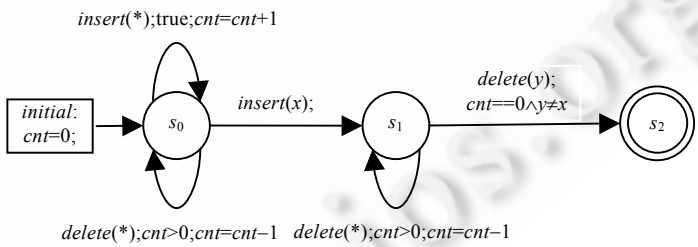


Fig.3 Event automaton specification of queues' FIFO property

图 3 队列的 FIFO 属性的事件自动机规约

1.2 事件自动机的描述语言及其生成算法

本文事件自动机描述语言的 BNF 范式定义如下:

EventAutomata::=Defines;

Defines::=Define;Defines|Define

Define::=Dstate|Dvariable|Dtransition

Dstate::=define state Id Stype

Stype::=0|1|2|3

Dvariable::=define Type Id=Expr

Type::=int|real|bool

Dtransition::=define transition Id (Id;Event;Expr;Assignments;Id)

Event::=Id(Syms)|Id(·)

Syms::=Sym,Syms|Sym

Sym::=Id|Constant

在 BNF 范式中,所有大写字母开头的都是非终结符,所有小写字母或数字开头的都是终结符.上述 BNF 范式所代表的含义如下:一个事件自动机(EA)是由一系列的定义(Define)组成,每一个定义(Define)可以是对状态的定义(Dstate)、对变量的定义(Dvariable)和对转移的定义(Dtransition):

- 一个状态的定义包括了表示该状态的标识符(Id)和该状态的类型(Stype),状态类型取 4 个值:0 表示该状态既不是初始状态也不是接受状态,1 表示该状态为初始状态,2 表示该状态为接受状态,3 表示该状态既是初始状态又是接受状态.
- 变量的定义包括变量的类型(Type)、变量的标识符(Id)和变量的初始值(Expr),其中,变量类型的定义包括 3 种数据类型:整型(int)、实型(real)和布尔型(bool).
- 转移的定义包括转移的标识符(Id)、转移的开始状态标识(Id)、转移的触发事件(Event)、转移应该满足的条件(Expr)、转移的赋值语句(Assignments)和转移的下一状态标识(Id).

事件的定义包括事件的标识符(Id)和事件的参数(Syms),事件可以没有参数或者事件的参数序列中每一个参数可以是一个变量(Id)或者一个常数 Constant.

上述 BNF 范式中的非终结符 Expr,Assignments,Id,Constant 没有给出定义,这部分可以参考 C 语言 BNF 范式进行定义,这里不详细展开.下面我们给出图 2 中事件自动机的描述:

```
define state s0 1;   define state s1 0;   define state s2 2;
define int f=nondet;   define int f′=nondet;
define transition t0 (s0;all;true;empty;s0);
define transition t1 (s0;open(f);true;empty;s1);
define transition t2 (s1;close(f′);f′==f;empty;s0);
define transition t3 (s1;terminal;true;empty;s2);
```

其中,关键字 all 表示完全事件.关键字 empty 表示空,例如上述的自动机描述中,empty 表示该处不存在赋值语句.关键字 nondet 表示赋值为一个随机值.关键字 true 表示永真.

下面我们给出从事件自动机描述语言生成事件自动机的算法.

算法 1. 事件自动机生成算法.

输入:$L=\{l_1,\ldots,l_n\}$为一组自动机描述语言的语句集合,它描述了一个事件自动机.

输出:事件自动机 $EA=(Q,Var,A,T,s_0,F)$.

算法描述:

```
CreateEA(L)
begin
    EA=EmptyEA;
    for each l_i∈L do
        if l_i is in form [define state s c] then
            add s to Q;
            if c==1 then s_0=s; endif
            if c==2 then add s to F; endif
            if c==3 then s_0=s; add s to F; endif
        endif
        if l_i is in form [define Type v=expr] then add v=expr to Var; endif
```

```
        endfor
        for each l_i∈L do
            if l_i is in form [define transition t (s_0;e;expr;assignments;s_1)] then
                add e to A;
                add (s_0;e;expr;assignments;s_1) to T;
            endif
        endfor
    end
```

上述算法分为两个循环:第 1 个循环生成自动机的状态集合和变量集合;因为转移集合需要用到状态集合和变量集合,在这两个集合已生成的基础上才能生成转移集合和事件集合.其中,*EmptyEA* 表示一个空事件自动机,空事件自动机的状态集合、变量集合、事件集合、转移的集合和接受状态集合都为空,并且初始状态未设置.

2 基于事件自动机规约的有界模型检测算法

基于 SAT 和 SMT 的有界模型检测在 C 语言程序验证方面占据重要地位,包括 CBMC[12],ESBMC[13],SATABS[17]和 F-Soft[28]等,都是基于 SAT/SMT 的模型检测工具.本文将采用基于 SMT 的有界模型检测,将事件自动机和 C 语言程序转化为 SMT 模型.

布尔可满足问题(SAT)[18]是针对命题逻辑的判定过程,其目的是确定一个由布尔变量和逻辑连接符析取、合取和取反组成的公式是否存在对其布尔变量的赋值使得公式的值为真.例如,$(p\vee r)\wedge(p\vee q)$在 p=true,q=false,r=false 的情况下为真.一些问题需要更丰富的逻辑描述,例如一阶谓词逻辑.一阶逻辑是由逻辑连接符、变量、全称量词、存在量词、函数和谓词符号组成,一阶谓词逻辑公式的可满足性是建立在一个模型的基础上.模型解释了公式中的变量、函数和谓词.SMT 对符号的解释是限制在一个理论背景中,例如算术运算(linear arithmetic)、位向量(bit vector)和自由函数(free function)等.

算术运算. 例如公式 $3x_1+2x_2\leqslant 5x_3\wedge 2x_1-2x_2=0$,SMT 求解器对其中的+,≤,−,=分别解释为算术加法、算术小于等于、算术减法和算术相等,并给出对 x_1,x_2,x_3 的赋值使其满足上述公式.

位向量. 定义了位向量的多种操作,包括与、或、非、异或、左移和右移等.例如公式 $a_{[8]}\wedge b_{[8]}=\langle 00000000\rangle_2$,SMT 求解器判断是否存在对两个 8 位的位向量赋值,使得它们的与为 0.

自由函数. 自由函数是不对函数做任何解释的函数,例如,f:$int\rightarrow int$ 表示一个从整数映射到整数的函数,但是没有给出具体的映射.现在给出公式 $f(2)=3\wedge f(3)=4$,可知该公式可满足.如果 $f(2)$已经绑定为 5,则该公式不可满足.

关于 SMT 的详细理论可以参考文献[19].本节主要分为 3 个部分:第 1 部分介绍将 C 程序转化为 SMT 模型的方法和原理,第 2 部分介绍事件自动机 SMT 模型转化方法,最后一部分给出反例路径的算法.

2.1 C程序到SMT模型的转化

为了解决无限堆栈程序模型状态的无限性和程序中对递归和循环次数难以确定的困难,本文对有界模型检测的中“有界”的定义为设定循环和递归执行次数的上界.如果程序的执行不可能超过上界则正确进行检测,否则提示增大循环和递归的上界.

本节介绍的转化过程主要分为 3 个步骤:步骤 1 将 C 程序中的循环和函数调用进行展开;步骤 2 将步骤 1 中的程序转化为只包含 if 语句、赋值语句和断言的程序;步骤 3 将步骤 2 的程序转化为单一赋值形式(static single assignment).本节的转化过程参考了 Clarke 等人[2,12]和 Armando 等人[20]的工作,但是在数组和结构体的转化方面存在一定的差异.

步骤 1. 循环与函数的展开.

C 语言中的循环结构包括 for,do...while,while 等,这里我们只以 while 循环为例来演示如何将其展开,其他

循环的展开方式类似.while 循环的一次展开过程如下:

$$\text{while(e) I;}\Rightarrow\text{if(e)\{I; while(e) I;\}}.$$

在已经设定上界为 n 的情况下,我们将上述循环展开 n 次.对于展开 n 次之后的循环,并不能代表循环在 n 次之后一定结束,因此,我们需要判断在展开之后循环是否终止.我们将在最后一次循环展开加入一个断言来表示,即

$$assert(!e).$$

这样,当程序循环展开没有达到循环终止的条件时,断言就会提醒用户重新设定循环上界,验证算法是不会验证之后的程序,而只是局限在已经展开的部分.从图 4(a)到图 4(b)的转化为循环的展开过程.

```
int x=2, sum=0;                x=2, sum=0;                    (1) x0=2; sum0=0;
while (x>0){                   if (x>0){                      (2) guard0 = x0>0;
    sum+=x;                        sum+=x; x--;               (3) sum1=guard0?(sum0+x0):sum0;
    x--;                           if (x>0){                  (4) x1=guard0?x0−1:x0;
}                                      sum+=x; x--;           (5) guard1=guard0∧x1>0;
                                       assert(!(x>0));        (6) sum2=guard1?(sum1+x1):sum1;
                                   }                          (7) x2=guard1?x1−1:x1;
                               }                              (8) assert(!(x2>0));
```

(a) (b) (c)

Fig.4 Translation from a C program to its static single assignment form

图 4 C 程序到单一赋值形式的转化

递归函数可以采用与循环相同的策略进行展开.对于带有返回值的函数,增加一个辅助变量存放该函数的返回值,这样,我们把计算结果直接赋值给辅助变量,即可代替程序中的 return 语句.

步骤 2. 转化为只包含 if 语句、断言和赋值语句程序.

经过步骤 1 的转化,程序中已经不存在循环语句和函数,因此,程序中只存在 if 语句、赋值语句、goto 语句、break 语句和 continue 语句等.continue 语句和 break 语句都可以很容易地转化为 goto 语句表达,goto 语句可以被转化为 if 语句表达的形式,switch 语句也可以转化为 if 语句表达.这几步的转化并不是很复杂,这里不再详细叙述,Armando 等人在文献[20]中有完整的描述.

步骤 3. 转化为单一赋值形式.

经过展开之后,程序将只包含 if 语句、赋值语句和断言(assertions).下面将其转化为单一赋值形式的 SMT 模型.

对每一个变量 x,它的 n 个复制设为 $\{x_0,x_1,\ldots,x_{n-1}\}$,其中,$n$ 是程序中对变量 x 赋值的次数.除了程序中的变量之外,单一赋值形式有一个 *guard* 变量,它表达的是程序执行到该位置所应满足的条件.对于每一个赋值语句,我们将赋值语句左边的变量下标加 1,赋值语句右边所有的变量保持该语句之前的标号.

条件赋值表达为 $x=guard?x_1:x_2$,即,*guard* 为 true,则 $x=x_1$;否则,$x=x_2$.

一维数组可以看作是一个函数 $f:\mathbb{N}\rightarrow data$,其中,$\mathbb{N}$表示数组的下标,是包含 0 的自然数;而 *data* 表示的是符合数组元素类型的数据的集合.这样,$f(i)$就表示取数组中第 i 个元素.因为 SMT 求解器对于函数的表达非常方便,所以本文将数组表达为函数.例如,赋值语句 $a[k]=e$,其中,a 为一个数组,e 为一个类型符合该数组的表达式,且在该语句之前数组的单一赋值下标为 i,则该语句的单一赋值形式为 $a_{i+1}:=a_i$ with $a_i(k)=e$,即

$$a_{i+1}(j)=\begin{cases}e, & j=k\\ a_i(j), & j\neq k\end{cases}.$$

对于多维数组可以转化为多个参数的函数,例如二维数组 $a[i_1][i_2]$,可以用 $a(i_1,i_2)$来表示.

对于 C 语言中的结构体,SMT 求解器中有针对 Record 的判定理论.而 Record 和结构体的特性非常相似,有利于我们直接对其进行转化.对于结构体赋值语句 $s.f=e$,其中,s 为结构体变量,f 为结构体变量的一个数据分量,它转化成单一赋值语句为 $s_{i+1}:=s_i$ with $s_i.f=e$,即

$$s_{i+1}.j=\begin{cases}e, & j=f\\ s_i.j, & j\neq f\end{cases}.$$

图 4(b)到图 4(c)给出了步骤 3 的转化过程.

在转化的处理过程中,指针是一个非常重要的特性,本文对指针的转化主要采用 Clarke E 在文献[2,12]中所给出的方法.对于一个变量 a,我们用$\&a$ 表示它的地址,地址是一个常数.当表达式中没有出现对指针表达式的间接引用时,对于一个指针变量 p,转化为单一赋值形式与非指针变量相同,例如,$p=p+1$ 转化为 $p_i=p_{i-1}+1$.当程序表达式中出现对指针表达式的间接引用时,情况比较复杂,下面将详细介绍.

通过使用一个递归函数$\phi(e,g,o)$消除程序中的指针表达式的间接引用,其中,e 表示一个将要被间接引用子表达式,g 表示该表达式被执行的 *guard*,o 表示偏移量.标准 C 语言提供两种间接引用操作符:“*”操作符和下标操作符:$*e\rightarrow\phi(e,g,0)$,$e[o]\rightarrow\phi(e,g,o)$.

对于任意一个指针都有它的类型$*T$,在语法分析的时候就可以确定.在介绍函数ϕ之前,首先定义$\rho(expr)$,其中,*expr* 为一个符合 C 语言语法的表达式,$\rho(expr)$表示该表达式在程序中的单一赋值形式.

ϕ函数的详细定义如下:

1. e 是一个指针变量,且在已经生成的单一赋值形式中存在$\rho(e)=e'$,那么$\phi(e,g,o)=\phi(e',g,o)$.
2. e 为一个数组,假设 a 为一个数组,$e=a$,即 $e=\&a[0]$,那么$\phi(e,g,o)=\phi(\&a[0],g,o)$.
3. e 为一个非数组变量的地址,假设 $e=\&s$ 其中,s 是一个变量,那么$\phi(e,g,o)$等于 s,并且 $o=0$,即,$\phi(e,g,o)=s$.并且 s 的类型必须为 T.
4. e 为某个数组元素的地址,假设$e=\&a[i]$,其中,a 为一个数组,那么$\phi(e,g,o)=a(i+o)$,数组的值在前面介绍中使用函数表达.
5. e 为一个条件赋值语句的条件表达式,假设 $e=c?e':e''$,其中,c 不包含间接引用操作符,那么:
$$\phi(c?e':e'',g,o)=c?\phi(e',g\wedge c,o):\phi(e'',g\wedge\neg c,o).$$
6. e 为一个指针算术表达式,指针算术表达式是一个指针和一个整数的和,假设 e'为指针部分,i 为整数部分,则 $e=e'+i$.那么$\phi(e'+i,g,o)=\phi(e',g,o+i)$.
7. e 为一个指针类型强制转换,即 $e=(Q*)e'$,Q 为任意类型,那么$\phi((Q*)e',g,o)=(Q)\phi(e',g,o)$.需要说明的是:在强制转化过程,需要用到 SMT 模型中的位向量进行建模.例如,对于一个整型变量 x,转化为 char 类型的操作为 $x\wedge(11111111)_2$.
8. 在其他情况下是没有意义的,例如,e 可能指向一个空(NULL)或者指针没有被初始化.为了保证这种情况不会发生,通过加入断言 *assert*$(!\rho(g))$来实现.

我们给出一个例子来说明上述的计算过程,请看下面的一段程序:

*int a, b, *p;*

if (x) $p=\&a$; else $p=\&b$;

$*p=1$;

第 1 条语句被转化为 $p_1=(x_0?\&a:p_0)$;$p_2=(x_0?p_1:\&b)$.间接引用$*p$ 可以通过如下的计算被消除:

$$\begin{aligned}*p&=\phi(p,\text{true},0)\\&=\phi((x_0?p_1:\&b),\text{true},0)(\text{因为 } p_2=(x_0?p_1:\&b))\\&=x_0?\phi(p_1,x_0,0):\phi(\&b,\neg x_0,0)\\&=x_0?\phi(p_1,x_0,0):b\\&=x_0?\phi(x_0?\&a:p_0,x_0,0):b(\text{因为 } p_1=(x_0?\&a:p_0))\\&=x_0?(x_0?a:\phi(p_0,\text{false},0)):b\\&=x_0?a:b.\end{aligned}$$

由此可知:在 x_0 的情况下,对 a 赋值为 1;在$\neg x_0$ 的情况下,对 b 赋值为 1.即,$a_1=x_0?1:a_0$,$b_1=\neg x_0?1:b_0$.

2.2 事件自动机SMT模型转化方法

本节通过引入自动机可达树的概念,给出事件自动机到 SMT 模型的转化.我们首先介绍一些符号表示,这有利于下面对转化算法的描述:

事件符号表示. 对于一个事件 e,它有多个参数,我们用 $e[i]$表示事件的第 i 个参数,$e.name$ 表示事件的名称,$e.size$ 表示参数的个数.

单一赋值操作符号表示. 对于一个表达式 $expr$,针对单一赋值形式,我们定义两种操作:

- $\rho(expr)$表示结合已有 SMT 模型将该表达式转化成单一赋值形式;
- 对于一个变量 v,我们使用$\alpha(v)$表示变量 v 被赋值的次数,每当变量 v 被赋值一次,$\alpha(v)$的值加 1.

事件自动机符号表示. 对于一个事件自动机 EA,假设事件自动机的状态为 $s_0,s_1,\ldots,s_m$,其自动机变量为 $x_0,x_1,\ldots,x_n$.我们给出以下重要的符号表示:

- $EA.Var$ 表示事件自动机中变量的集合,即,$EA.Var=\{x_0,x_1,\ldots,x_n\}$.
- 对于自动机中的每一个状态 s_i,与单一赋值形式类似对 s_i 进行标号,也就是说,当转移到该状态时,该状态的标号加 1.$s_{i,j}$ 为 s_i 的第 j 个复制,我们称 $s_{i,j}$ 为状态变量.
- $trans(s_i)$表示所有从 s_i 出发的转移的集合,对于任意 $t\in trans(s_i)$,$t.e$ 表示触发该转移的事件,$t.guard$ 表示转移发生的条件,$t.assigns$ 表示转移的赋值语句的序列,$t.start$ 表示转移的起始状态,$t.end$ 表示转移的下一个状态.
- $EA.s_0$ 为事件自动机的初始状态,$EA.F$ 为事件自动机的接受状态集合.

绑定操作符号表示. 一个绑定 $Bind$ 可以表示为$[x_1\mapsto expr_1;x_2\mapsto expr_2;\ldots;x_n\mapsto expr_n]$.我们给出如下操作:

- 变量在绑定中的取值:

$$Bind(x)=\begin{cases}s, & (x\mapsto s)\in Bind\\ x, & (x\mapsto s)\notin Bind\end{cases}.$$

- $Bind\langle x\mapsto v\rangle$:产生一个新的绑定,如果 x 在 $Bind$ 中已经赋值,则将 v 重新绑定到 x;否则,将 $x\mapsto v$ 加入到绑定中.即

$$(Bind\langle x\mapsto v\rangle)(y)=\begin{cases}v, & y=x\\ Bind(y), & y\neq x\end{cases}.$$

- 对于一个表达式 $expr$,变量 $x\in expr$ 表示表达式中存在变量 x.将一个绑定作用到该表达式上定义为 $expr(Bind)$:$\forall x\in expr$,在表达式 $expr$ 中,使用 $Bind(x)$代替 x.
- 对于一个变量 x,$x\in Bind$ 表示 x 在 $Bind$ 中被绑定,$x\notin Bind$ 表示 x 在 $Bind$ 中未被绑定.

针对自动机和 C 语言结合的转化,本文提出一种自动机可达树来描述自动机的所有可能的执行路径,自动机可达树是本文算法的核心概念.

定义 7(自动机可达树). 一棵自动机可达树定义在一个事件自动机上,它定义为一个五元组 $ART=(Nodes,r_0,Edges,leaf,acceptleaf)$,假设它的事件自动机为 EA,其中,

(1) $Nodes$ 为可达树中节点的集合,每个节点都由两个部分组成:一部分是状态变量,另一部分是一个绑定.即,$n=(s_{i,j},Bind)$,其中,$s_{i,j}$ 为状态变量,$Bind$ 为自动机变量的绑定.$n.state$ 表示节点的状态变量, $n.bind$ 表示节点绑定.对于一个状态变量 $s_{i,j}$,如果 $s_i\in EA.F$,则表示为 $s_{i,j}\in EA.F$.

(2) r_0 为根节点,它包含初始状态的第 0 个复制和所有自动机变量初始值的绑定,即

$$r_0=(s_{0,0},[x_0\mapsto initialval_0;\ldots;x_n\mapsto initialval_n]).$$

(3) $Edges$ 为可达树的边,对于边$(n_1,n_2)\in Edges$,其中,$n_1,n_2\in Nodes$,$n_1=(s_{i,j},Bind_1)$,$n_2=(s_{k,l},Bind_2)$,则 n_1,n_2 满足以下两种情况之一:

1) $s_k\in\delta(s_i)$,即,存在转移从 s_i 到 s_k;

2) $i=k,l>j,Bind_1=Bind_2$.

第 2 种情况是在事件不发生的条件下产生的节点,事件不发生会产生不发生约束,所以这里必须多生

成一个子节点,如果事件必然发生,则不生成该节点.

(4) *leaf*⊆*Nodes* 为自动机可达树的叶子节点集合,叶子节点定义为没有子节点的节点.

(5) *acceptleaf*⊆*leaf* 为自动机可达树的接受叶子节点集合,对于一个节点 n∈*acceptleaf*,*n.state*∈*EA.F*.

自动机可达树的构造是在对事件自动机和程序进行转化的过程中进行的,自动机可达树从根节点到接受叶子节点的路径,就是一条事件自动机的接受路径.自动机可达树中所有的节点的状态变量都不重复.

定义 8(自动机可达树路径). 自动机可达树的一条路径定义为 $n_0,\ldots,n_k$,其中,$n_0=r_0$ 且$\forall 0\leqslant i\leqslant k-1,(n_i,n_{i+1})\in Edges$.如果自动机可达树的一条路径为接受路径,则 $n_k.state\in EA.F$.自动机可达树的一条接受路径表达了程序中存在的一条错误路径.

图 5 给出了图 2 中对文件打开和关闭属性描述的一棵自动机可达树实例,左边的路径 $s_{0,0}s_{1,0}s_{0,2}$ 不是一条可接受路径,而 $s_{0,0}s_{1,0}s_{2,0}$ 是一条可接受路径,因为最后一个节点为接受叶子节点.节点$(s_{0,1},[\cdot])$为 open 事件不发生情况下的节点,在编码中会产生不发生事件约束.

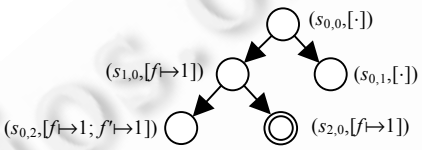


Fig.5 Automaton reachability tree of the event automaton in Fig.2

图 5 图 2 中事件自动机的自动机可达树

下面给出一些关于自动机可达树的符号表示.

自动机可达树符号表示. 对于一棵自动机可达树 *ART*,它的一个节点 n 的父节点表示为 *parent*(n),该节点的直接子节点集合表示为 *child*(n),该节点的状态变量分量表示为 *n.state*,绑定分量表示为 *n.bind*. *ART.root* 表示可达树的根节点,*ART.leaf* 表示当前所有叶子节点的集合,*ART.acceptleaf* 表示接受叶子节点集合.

在开始算法介绍之前,我们首先介绍一些全局存储变量:

- *ART*:自动机可达树.
- *Model*:存储事件自动机和 C 程序当前已经完成的 SMT 模型.*Model.add*(*statement*)表示增加一条 SMT 编码.
- *EA*:对属性进行规约的事件自动机,它的状态集合为$\{s_0,\ldots,s_m\}$.

下面给出本文结合 C 程序 SMT 模型生成事件自动机 SMT 模型的算法.

算法执行框架. 输入经过第 2.1 节步骤 2 处理过的程序 *Program*=$\{p_1,\ldots,p_n\}$,即,只包含赋值语句、if 语句和断言语句,p_i 为一条语句.输入一个对该程序属性规约的事件自动机 *EA*.

```
BuildSMT(Program,EA)
begin
    Initial(ART);
    for i from 1 to n do
        Model.add(translate(p_i));
        guard=the execution condition of p_i;
        if p_i generates event e then ReceiveEvent(e,guard); endif
    endfor
    MakeAssertion(·);
end
```

框架中的 *Initial*(*ART*),*ReceiveEvent*(*e*,*guard*),*MakeAssertion*(·)将在下面的算法 3、算法 6 和算法 7 中描述.上述执行框架首先使用 *Initial*(*ART*)初始化自动机可达树;然后将程序中的每一条执行语句转化为单一赋值形

式,本文使用 translate 对该语句进行转化,translate 为第 2.1 节中介绍的算法实现,文献[12,20]中已经给出,本节直接引用.如果该语句的执行会产生事件 e,则调用 *ReceiveEvent*(e,*guard*)生成事件自动机和程序之间约束关系的 SMT 模型;最后,调用 *MakeAssertion*(·)生成违反事件自动机属性的断言.该算法框架生成的 SMT 的模型将会使用 SMT 求解器求解,求解结果使用算法 8 生成反例路径.

算法 2. 事件匹配函数.

对于两个事件 e_1 和 e_2,判断两个事件是否匹配,当接收到匹配的事件,才会触发自动机的迁移.

输入:两个事件 e_1 和 e_2.

输出:如果两个事件匹配,则返回 true;否则,返回 false.

算法描述:

```
bool match(e1,e2)
begin
    if e1==Σ∨e2==Σ then return true; endif
    if e1.name≠e2.name then return false; endif
    if e1.size≠e2.size then return false; endif
    for i from 0 to e1.size−1 do
        if the types of e1[i] and e2[i] are not same then return false; endif
    endfor
    return true;
end
```

上述算法中,类型不一致相当于编译中的类型检测(type checking),因为实参事件的参数类型必须与自动机事件的参数类型一致.算法首先判断事件的名称是否一致,然后判断事件参数个数是否相等,最后判断每一个参数对应的类型是否一致.

算法 3. 自动机可达树初始化函数.

输入:自动机可达树 *ART*.

输出:初始化后的自动机可达树 *ART*.

算法描述:

```
void Initial(ART)
begin
    Bind=[·];
    for each v∈EA.Var do
        v0=initialvalue; Bind⟨v↦initialvalue⟩;
    endfor
    ART.root=(s0,0,Bind);
end
```

算法首先对所有自动机变量的第 0 个复制赋初始值,然后加入到绑定当中,最后将可达树根节点的自动机变量设置为初始状态,根节点的绑定设置为所有自动机变量的初始值.

算法 4. 判断一个表达式的值是否为常数.

输入:一个表达式 *expr*,一个变量的引用 t,如果在上下文中 *expr* 是常数,则将表达式的值赋值给变量 t.

输出:布尔值,若为真则表示该表达式为常数;否则,说明该表达式含有不确定值的变量.

算法描述:

```
bool Constant(expr,&t)
begin
```

Bind=[·];
for each *var*∈*expr* **do**
if the value of *var* is constant *val* **then** *Bind*⟨*var*↦*var*⟩ **else return** false **endif**
endfor
t=*expr*(*Bind*);
return true;
end

对于一个表达式中的所有变量,如果已有的值为常数,则将这个常数加入到绑定当中;如果不是一个常数,则返回 false.最后,将所有变量用它们的常数值代替,可以得到表达式的值,将该值赋值为变量 *t*.例如 *a*=3,表达式 *a*+4 的值计算为 7.

算法 5. 当产生事件 *e*,并且匹配自动机转移的事件 *t.e* 的编码.

输入:程序产生的事件 *e*,节点 *n* 为可达树的一个非接受叶子节点,该节点为准备扩展的节点,*t* 为可以发生的转移,*n* 的状态变量为 *t.start*.*φ*为该事件发生的上下文条件与 *n.state*==true 的合取.这里,前置条件为 *e*,与 *t.e* 是匹配的.

输出:新的 SMT 模型 Model.

算法描述:

void *codeTran*(*e*,*t*,*n*,*φ*)
begin
Bind=*n.bind*;
for *i* from 0 to *e.size*−1 **do**
if *t.e*[*i*]∈*Var* **then**
if *Constant*(*e*[*i*],*c*) **then** *Bind*⟨*t.e*[*i*]↦*c*⟩; continue; **endif**
Model.add($t.e[i]_{\alpha(t.e[i])+1}$=*ρ*(*e*[*i*]));
α(*t.e*[*i*])=*α*(*t.e*[*i*])+1;
Bind⟨$t.e[i]\mapsto t.e[i]_{\alpha(t.e[i])}$⟩;
endif
endfor
guard=*φ*∧*t.guard*(*Bind*);
Model.add($t.end_{\alpha(t.end)+1}$=*guard*?true:false);
α(*t.end*)=*α*(*t.end*)+1;
for each (*v*=*expr*)∈*t.assigns* **do**
if *Constant*(*expr*,*c*) **then** *Bind*⟨*v*↦*c*⟩; continue; **endif**
Model.add($v_{\alpha(v)+1}$=*ρ*(*expr*));
α(*v*)=*α*(*v*)+1;
Bind⟨$v\mapsto v_{\alpha(v)}$⟩;
endfor
n.addChild($t.end_{\alpha(t.end)}$,*Bind*);
if *guard* is not always true **then**
Model.add($t.start_{\alpha(t.start)+1}$=((*n.state*==true)∧$t.end_{\alpha(t.end)}$==false)?true:false);
α(*t.start*)=*α*(*t.start*)+1;
n.addChild($t.start_{\alpha(t.start)}$,*n.bind*);
endif

end

该算法描述了对一个转移的编码,当产生一个实参事件时,首先判断实参事件中参数是否为一个常数:如果为一个常数,则直接绑定,不生成 SMT 模型;否则,将转移 t 上的事件参数生成一条赋值约束,并加入到绑定 *Bind* 当中.然后处理转移条件 *t.guard*,算法中的变量 *guard* 是$\varphi\wedge t.guard(Bind)$,也就是说,只有当转移的条件也被满足之后才可以转移到下一个状态,然后对转移的下一个状态产生一个新的状态变量,新的状态的变量标号为$\alpha(t.end)+1$,也即该状态的第$\alpha(t.end)+1$ 个复制.然后处理转移中的赋值语句,将所有的赋值语句转化为单一赋值形式.最后,我们更新自动机可达树,在自动机可达树的节点 n 上增加一个子节点.

另外,最后要判断算法中的变量*guard*在上下文中是否永真:如果永真,则说明该转移必然会发生;否则,就需要考虑不发生的情况.在不发生的情况下,需要增加一个状态赋值约束,就是节点 n 的状态被复制到其子节点,并且这是在事件不发生的情况下才有的操作,所以它的条件是 *t.end* 为 false 且 *n.state* 为 true.子节点的绑定为节点 n 的绑定,最后把状态和绑定加入到 n 的子节点中.

算法 6. 接收事件.

输入:事件 e,φ为该事件可以发生的条件,也就是一个谓词表达式.

输出:新 SMT 模型 Model.

算法描述:

```
void ReceiveEvent(e,φ)
begin
    for each n∈ART.leaf–ART.acceptleaf do
        for each t∈trans(n.state) do
            if match(t.e,e) then
                guard=φ∧(n.state==true);
                codeTran(e,t,n,guard);
            endif
        endfor
    endfor
end
```

算法 6 给出了当产生一个事件之后的编码,当该事件和转移的事件能够相互匹配时,转移可以发生.首先生成事件发生和当前节点状态变量为真的条件,用变量 *guard* 表示,最后调用 *codeTran* 对该转移和事件进行编码.

算法 7. 生成 SMT 模型的断言.

输入:自动机可达树.

输出:增加断言的 SMT 模型 *Model*.

算法描述:

```
void MakeAssertion(·)
begin
    formula f=false;
    for each n∈ART.acceptleaf do
        f=f∨n.state
    endfor
    Model.add(assert(f));
end
```

算法 7 是在 C 程序和自动机两者的 SMT 模型都生成完毕之后,最后加入断言.如果模型满足断言,则说明我们找到了一条错误路径.

2.3 反例路径生成算法

我们使用 Yices[29,30]对 SMT 模型进行判定.Yices 将会对模型中的所有未初始化的变量或被赋值为非确定值的变量给定一个确定值,根据所有变量的初始值确定程序执行的路径和事件自动机的接受路径.反例路径算法将生成两个路径队列:(1) 一个队列给出事件自动机的一条接受路径;(2) 另外一个队列给出 C 程序中反例的执行路径.下面给出反例路径算法所使用的符号表示:

- K 表示所有变量的当前值的绑定,包括 C 程序的变量、自动机状态变量和自动机变量.
- 事件自动机和 C 程序的 SMT 模型定义为 $Z=\{z_1,z_2,\ldots,z_n\}$,其中,z_i 为一条 SMT 模型语句.
- C 程序定义为一系列语句的集合 $C=\{c_1,c_2,\ldots,c_m\}$,c_i 是 C 程序中的赋值语句和条件语句等.给出一个映射 $f:Z\to C\cup\{\varepsilon\}$,其中,$\varepsilon$表示对于一条 SMT 模型语句在 C 程序中不存在一条语句与之对应.在第 2.1 节介绍的 C 程序到 SMT 模型转化的过程中,每转化一条 C 语句,建立一个 f 的映射绑定.
- $Q=\{s_0,s_1,\ldots,s_k\}$为事件自动机状态的集合,给出一个映射 $g:Z\to Q\cup\{\varepsilon\}$,其中,$\varepsilon$表示对于一条 SMT 模型语句在一个事件自动机中不存在一个状态与之对应.在第 2.2 节事件自动机转化为 SMT 模型的过程中,每生成一个可达树节点,建立一个 g 的映射绑定.
- 给定两个队列 que_C 和 que_Q,que_C 存储 C 程序中错误路径,que_Q 存储事件自动机的一条接受路径.对于一个队列有两种操作:*queue.inqueue*(e)在队列末尾中插入一个元素,*queue.dequeue*(e)在队列的头部删除一个元素并将值赋值给 e.

算法 8. 反例路径生成算法.

输入:K 为所有变量的当前赋值绑定,i 为当前的语句标号,语句标号从 1 开始一直到 SMT 模型结束.

输出:打印反例路径.

算法描述:

```
void FindCounterExample(K,i)
begin
    (a):
    if z_i is in form [v_i=c?e_1:e_2] then
        if c(K)==false then FindCounterExample(K,i+1) return;
            else K⟨v↦e_1(K)⟩
        endif
    endif
    (b):
    if z_i is in form [v_i=e_1] then K⟨v↦e_1(K)⟩ endif
    (c):
    if f(z_i)≠ε then que_C.inqueue(f(z_i)); PTINT(f(z_i)); endif
    (d):
    if g(z_i)≠ε then que_Q.inqueue(g(z_i)); PTINT(g(z_i)); endif
    FindCounterExample(K,i+1);
end
```

算法中,$c(K)$==false 表示用绑定中变量的值代替表达式 c 中的变量,然后判断 c 为真或假.标号(a)表示对于一条 SMT 模型语句,如果它是一个条件赋值,且当前所有变量绑定的值不能够满足条件 c,则该条语句在 C 程序或事件自动机中没有被执行,直接进入下一条语句;如果满足,则更新绑定 K 中变量的值.标号(b)表示一条简单赋值语句,更新绑定 K 中变量的值.标号(c)表示当 SMT 模型语句对应的是一条 C 程序语句,则将对应的 C 程序语句加入到队列 que_C 中.标号(d)表示当 SMT 模型语句对应的是一个自动机状态时,则将该自动机状态加入到队列 que_Q 中.程序最后将新的变量绑定值传递到下一条语句来处理.

我们通过调用 *FindCounterExample*(*K*,1)就可以打印程序中存在的错误路径,z_1 为 SMT 模型的第 1 条语句.上述算法将会打印一条 C 程序的执行路径,并且当某条语句产生一个事件的时候,会打印出事件自动机发生转移的状态.两个队列 que_C 和 que_Q 则分别保存了 C 程序的反例路径和事件自动机的一条接受路径.

3 实例分析

3.1 实例1:lock和unlock属性验证

首先,我们学习 CBMC 主页的一个示例程序,参见文献[31].如图 6 所示的一段 C 语言程序,它定义了两个函数:*lock*(·)和 *unlock*(·).程序中的定义 *_Bool nondet_bool*(·)表示该函数不确定返回布尔类型的值,这是 CBMC 的函数库所定义的,而且不需要函数体.当 *nondet_bool*(·)返回 true 时加锁,设置 *LOCK*=1,否则不进行加锁操作.*unlock*(·)是开锁操作,直接设置 *LOCK*=0.main 是对两个函数的使用.我们现在要规约对程序的所有执行路径开锁操作的前提是已经上锁.

```
1  _Bool nondet_bool(·);
2  int LOCK=0;
3  _Bool lock(·){
4     if (nondet_bool(·)){
5        LOCK=1;   /* $event{lock(·)} */
6        return 1;
7     }
8     return 0;
9  }
10 void unlock(·){
11    LOCK=0;   /* $event{unlock(·)} */
12 }
13 int main(·){
14    unsigned get_lock=0;
15    int times;
16    while (times>0){
17       if (lock(·)){
18          get_lock++;
19          /* critical section */
20       }
21       if (get_lock!=0)
22          unlock(·);
23       get_lock−−;
24       times−−;
25    }
26 }
```

Fig.6 Program of Case 1

图 6 实例 1 的程序

图 7 给出事件自动机在所有未加锁的情况下执行了解锁操作路径的规约,该自动机有 3 个状态:s_0 为自动机初始状态,s_1 为已加锁状态,s_2 为接受状态.也就是说,当程序进入该状态时,说明已经找到一条违反该属性的执行路径.

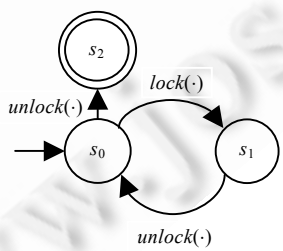


Fig.7 Property specification of Case 1

图 7 实例 1 的属性规约

对已有的 C 语言程序和事件自动机,执行本文的转化算法可以得到一个 SMT 模型,保存在文件 case1.ys.转化的 SMT 模型可以在网上下载(本文实例 1 和实例 2 的 SMT 模型可以通过 http://www.kuaipan.cn/file/id_218529353359693077.htm?source=1 下载),我们首先把 main 中的 while 循环展开 1 次,然后转化为 SMT 模型.使用 SMT 求解器 Yices 进行判定,执行命令 yices –e case1.ys 返回 unsat,即,没有发现错误.然后,我们将循环展开 2 次,重新生成 SMT 模型,使用 Yices 对 SMT 模型重新进行判定,得到分析的结果为 sat,如图 8 所示.判断结果是 SMT 模型可满足,即存在一条执行路径使事件自动机到达接受状态,程序中存在错误.根据 Yices 所提供的反例

可知,在 t_1=false,t_0=false,$times_0$=2 的情况下程序出现错误.程序在循环展开 2 次的情况下,t_0 代表第 1 次执行 *lock* 操作时不确定选择为 false,即程序没有选择加锁.t_1 代表第 2 次执行 *lock* 操作时不确定选择也为 false. $times_0$ 确定循环展开了两次.所以,反例就是在循环的两次执行中 *lock* 操作选择不加锁的情况下产生.

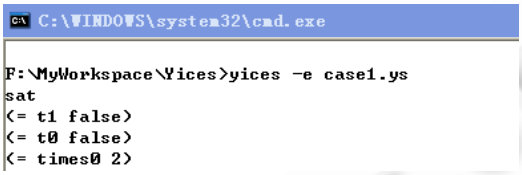


Fig.8 Verification result of Case 1

图 8 实例 1 的验证结果

出错的原因:程序第 1 次没有加锁的时候,*unlock* 也不会执行,但是 *get_lock* 会减 1;又因为 *get_lock* 的初始值为 0,减 1 会出现溢出问题,*get_lock* 不再为 0;在第 2 次循环执行的情况下,依然没有执行加锁操作,但是因为 *get_lock*≠0,所以进行解锁操作.也就是程序从状态 s_0 进入状态 s_2,从而找到一条反例路径.由此可以看出:SMT 对变量的取值范围的解释和 C 程序中的一致,也就是说,本文方法可以检测变量的溢出等比较底层的逻辑错误.

3.2 实例2:队列FIFO属性验证

图 9 给出了一段队列操作的代码,参考文献[20]中的一段程序.它定义了插入操作、删除操作,队列我们用结构体来定义.队列的定义中,nelem 表示当前队列保存的数据个数,head 和 tail 分别代表队列的头部和尾部,buffer 存放队列中的数据.队列的插入操作是在队列的队尾插入,删除操作是队列的头部删除,队列的实现是循环队列.在代码中加入生成事件的注释,看出我们只有在真正将数据插入到队列中才会产生事件,如果队列是满的,则不会产生插入事件.同理,删除操作中,只有当队列非空才会产生删除事件,删除的数据被保存在变量 *res* 中.队列插入的数据使用随机函数生成.本文使用图 3 中对队列先进先出属性的规约验证这段程序的所有执行路径是否符合先进先出的规则.

```
1  #define N 1000
2  int nondet_Int(·);
3  struct queue{
4     int nelem;
5     int head;
6     int tail;
7     int buffer[N];
8  }global_queue;
9  void insert(int i){
10    if (global_queue.nelem==N)
11       return;
12    else{
13    global_queue.buffer[global_queue.tail]=i;
         /* $event{insert(i)} */
14       if (global_queue.tail==N–1)
15          global_queue.tail=0;
16       else
17          global_queue.tail++;
18       global_queue.nelem++;
19    }
20 }
21 int res;
22 void delete(·){
23    if (global_queue.nelem==0)
24       return;
25    else{
26    res=
      global_queue.buffer[global_queue.head];   /* $event(delete(res)) */
27       if (global_queue.head==N–1)
28          global_queue.head=0;
29       else
30          global_queue.head++;
31       global_queue.nelem––;
32    }
33 }
34 int main(·){
35    int i;
36    global_queue.nelem=0;
37    global.queue.head=global_queue.tail=0;
38    for (i=0; i<2; i++)
39       insert(nondet_Int(·));
40    for (i=0; i<2; i++)
41       delete(·);}
```

Fig.9 Program of Case 2

图 9 实例 2 的程序

使用本文转化到 SMT 模型的算法,可以得到转化后的模型 case2.ys.这里,在程序中有两个循环,他们循环的次数都是 2,所以该程序不需要设定循环上界.使用工具 Yices 对该模型进行判定 yices –e case2.ys,得到判定结果

如图 10 所示.结果为 unsat,也就是 SMT 模型无法满足,程序的所有路径都不会进入图 3 所规约的事件自动机的接受状态,该段代码在图 3 的事件自动机下是正确的.图 11 是该段程序验证过程中生成的自动机可达树,表 1 存放的是自动机可达树每个节点的详细信息.自动机可达树中所有的接受叶子节点中状态变量都为 false,也就是在实际的执行路径中无法到达该节点.

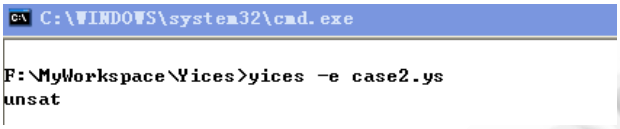


Fig.10 Verification result of Case 2

图 10 实例 2 的验证结果

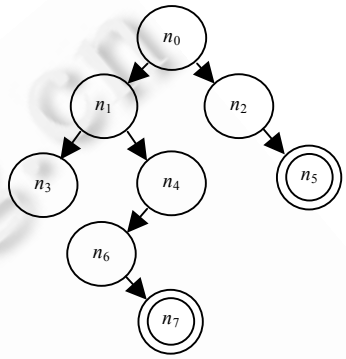


Fig.11 Automaton reachability tree of Case 2

图 11 实例 2 的自动机可达树

Table 1 Node information of Fig.11

表 1 图 11 的节点信息

节点	状态变量	绑定
n_0	$s_{0,0}$	$cnt\mapsto 0$; $x\mapsto x_0$; $y\mapsto y_0$
n_1	$s_{0,1}$	$cnt\mapsto 1$; $x\mapsto x_0$; $y\mapsto y_0$
n_2	$s_{1,0}$	$cnt\mapsto 0$; $x\mapsto x_1$; $y\mapsto y_0$
n_3	$s_{0,2}$	$cnt\mapsto 2$; $x\mapsto x_0$; $y\mapsto y_0$
n_4	$s_{1,1}$	$cnt\mapsto 1$; $x\mapsto x_2$; $y\mapsto y_0$
n_5	$s_{2,0}$	$cnt\mapsto 0$; $x\mapsto x_1$; $y\mapsto y_1$
n_6	$s_{1,2}$	$cnt\mapsto 0$; $x\mapsto x_2$; $y\mapsto y_0$
n_7	$s_{2,1}$	$cnt\mapsto 0$; $x\mapsto x_2$; $y\mapsto y_2$

3.3 性能分析

本节对事件自动机模型检测算法的性能进行评估.有界模型检测的效率与循环和递归的展开有密切的关系,因为:(1) 循环和递归展开会使代码量激增;(2) 代码的增加会导致变量复制的增加.本节的实验基于第 3.2 节中队列先进先出的实例,将图 9 中第 38 行和第 40 行的循环改为 for (i=0;i<M;i++).M 从 50 开始每次增加 50 次,记录每次运行的时间和内存.

为了形成对比效果,本文使用文献[20]中的方法,通过加入断言的形式来验证第 3.2 节中图 9 的代码.这里必须要说明的是:图 9 中的代码是无法通过加入断言来验证 FIFO 属性的,因为无法知道第 i 次插入队列中的数据是多少,这也是事件自动机表达能力上的体现.为了能够使用断言表达,修改图 9 中第 39 行的代码为 *insert*(i);并在第 41 行代码之后加入断言 *assert*(*res*==i).这样确定了第 i 次插入队列中的数据是 i,所以第 i 次删除的数据也是 i.实验硬件条件为 Intel® Pentium® CPU P6200 处理器和 2G 内存.

实验结果如图 12 和图 13 所示,其中,SMT-ASSERT 表示使用断言规约的 FIFO 属性验证,SMT-EA 表示使用事件自动机规约的 FIFO 属性验证.从图中可以得出如下结论:

(1) SMT-ASSERT 验证的最大循环次数为 750,当循环次数超过 750 时,Yices 求解器无法给出结果(Yices 求解器返回 3 类结果:sat,unsat,unknown),SMT-EA 验证的最大循环次数为 900.所以,图中实线的最大

循环次数达到 750,虚线的最大循环次数达到 900.

(2) SMT-EA 的运行时间要优于 SMT-ASSERT,SMT-ASSERT 在每次进行删除的时候都加入断言,断言不提供上下文信息,SMT-EA 的自动机可达树会记录路径的信息.

(3) SMT-ASSERT 使用的内存要远小于 SMT-EA,这是因为 SMT-ASSERT 在每次执行队列删除的操作时只需加入一个断言即可,而 SMT-EA 则需要记录事件自动机的所有可能的执行情况,也就是说,自动机可达树会不断增大.这是事件自动机在表达能力上增强的后果.

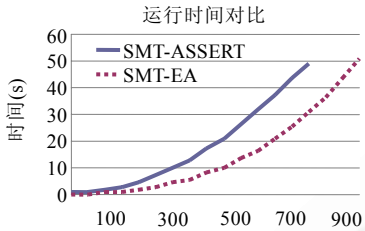


Fig.12 Run time comparison of EA and assertion

图 12 事件自动机和断言运行时间对比

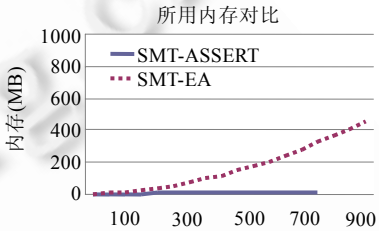


Fig.13 Memory comparison of EA and assertion

图 13 事件自动机和断言所用内存对比

综上所述,实验中,SMT-EA 在最大验证循环次数和验证的时间上要优于 SMT-ASSERT,SMT-ASSERT 所用的内存要少于 SMT-EA.这里需要说明的是:有界模型检测对验证规模的评估不能使用代码行数,而应该是程序执行的路径长度.例如图 9 中的程序,SMT-EA 验证 900 次的 *insert* 和 *delete* 操作,*insert* 和 *delete* 操作展开的步数为 6,那么验证的程序执行最长路径长度为 5 400 步.

4 总结与下一步工作

本文针对已有的 C 语言形式化验证工具中对属性描述表达力不够丰富、属性与程序不分离和缺乏对事件的表达等缺点,提出使用事件自动机对程序属性进行规约,给出从 C 程序生成事件的方法.在有界模型检测方法的基础上,结合本文提出的自动机可达树,给出了从事件自动机到 SMT 模型的转化方法和建立反例路径的算法.实例分析表明:事件自动机可以有效地表达现有的 C 语言形式化验证工具中无法表达的属性,且可以描述软件中与事件相关的安全属性.基于 SMT 求解器的模型检测算法提高了算法的执行效率,且与 C 程序中数据类型的语义相符.因此,本文所提出的方法适合基于事件自动机属性描述的 C 程序形式化验证.由实验分析可知,本文方法所消耗的内存较多.因此,研究自动机可达树的约简技术,将作为本文的下一步工作.另外,本文没有给出并发程序验证的算法,这部分工作将在未来展开.

致谢 感谢审稿专家提出的宝贵意见和建议,并向对本文工作给予支持和建议的同行表示感谢.

References:

[1] Clarke EM, Grumberg O, Peled DA. Model Checking. Cambridge: MIT Press, 1999.

[2] Clarke EM, Kroening D, Yorav K. Behavioral consistency of C and verilog programs using bounded model checking. In: Proc. of the DAC 2003. New York: ACM Press, 2003. 368−371. [doi: 10.1109/DAC.2003.1219026]

[3] Chen Z, Zhang D, Zhu R, Ma Y, Yin P, Xie F. A review of automated formal verification of ad hoc routing protocols for wireless sensor networks. Sensor Letters, 2013,11(5):752−764. [doi: 10.1166/sl.2013.2653]

[4] Godefroid P. Software model checking: The verisoft approach. Formal Methods in System Design, 2005,26(2):77−101. [doi: 10.1007/s10703-005-1489-x]

[5] McMillan KL. Symbolic Model Checking. Norwell: Kluwer Academic Publishers, 1993.

[6] McMillan KL. Interpolation and SAT-based model checking. In: Jr. Hunt WA, Somenzi F, eds. Proc. of the Computer Aided Verification (CAV). LNCS 2725, Berlin: Springer-Verlag, 2003. 1−13. [doi: 10.1007/978-3-540-45069-6_1]

[7] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham RM, Harrison MA, Sethi R, eds. Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 1977. 238−252. [doi: 10.1145/512950.512973]

[8] Clarke EM, Kroening D, Sharygina N, Yorav K. Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design, 2004,25(2-3):105−127. [doi: 10.1023/B:FORM.0000040025.89719.f3]

[9] Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In: Launchbury J, Mitchell JC, eds. Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 2002. 58−70. [doi: 10.1145/503272.503279]

[10] Ball T, Bounimova E, Kumar R, Levin V. SLAM2: Static driver verification with under 4% false alarms. In: Bloem R, Sharygina N, eds. Proc. of the Formal Methods in Computer Aided Design (FMCAD). Los Alamitos: IEEE, 2010. 35−42.

[11] Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker blast. Int'l Journal on Software Tools for Technology Transfer, 2007,9(5-6):505−525. [doi: 10.1007/s10009-007-0044-z]

[12] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Jensen K, Podelski A, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS 2988, Berlin: Springer-Verlag, 2004. 168−176. [doi: 10.1007/978-3-540-24730-2_15]

[13] Cordeiro L, Fischer B. Verifying multi-threaded software using smt-based context-bounded model checking. In: Taylor RN, Gall H, Medvidovic N, eds. Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE). New York: ACM Press, 2011. 331−340. [doi: 10.1145/1985793.1985839]

[14] Sinz C, Merz F, Falke S. LLBMC: A bounded model checker for LLVM's intermediate representation. In: Flanagan C, König B, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS 7214, Berlin: Springer-Verlag, 2012. 542−544. [doi: 10.1007/978-3-642-28756-5_44]

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

[16] Qu WX, Li T, Guo Y, Yang XD. Advances in predicate abstraction. Ruan Jian Xue Bao/Journal of Software, 2008,19(1):27−38 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/27.htm [doi: 10.3724/SP.J.1001.2008.00027]

[17] Clarke E, Kroening D, Sharygina N, Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs N, Zuck LD, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS 3440, Berlin: Springer-Verlag, 2005. 570−574. [doi: 10.1007/978-3-540-31980-1_40]

[18] Davis M, Putnam H. A computing procedure for quantification theory. Journal of the ACM (JACM), 1960,7(3):201−215. [doi: 10.1145/321033.321034]

[19] Biere A. Handbook of Satisfiability. Washington: IOS Press, 2009.

[20] Armando A, Mantovani J, Platania L. Bounded model checking of software using SMT solvers instead of SAT solvers. Int'l Journal on Software Tools for Technology Transfer, 2009,11(1):69−83. [doi: 10.1007/s10009-008-0091-0]

[21] Cordeiro L, Fischer B, Marques-Silva J. SMT-Based bounded model checking for embedded ANSI-C software. IEEE Trans. on Software Engineering, 2012,38(4):957−974. [doi: 10.1109/TSE.2011.59]

[22] He YX, Wu W, Chen Y, Xu C. Path sensitive program verification based on SMT solvers. Ruan Jian Xue Bao/Journal of Software, 2012,23(10):2655−2664 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4196.htm [doi: 10.3724/SP.J.1001.2012.04196]

[23] Huth M, Ryan M. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge: Cambridge University Press, 2004.

[24] Barringer H, Goldberg A, Havelund K, Sen K. Rule-Based runtime verification. In: Steffen B, Levi G, eds. Proc. of the Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS 2937, Berlin: Springer-Verlag, 2004. 44−57. [doi: 10.1007/978-3-540-24622-0_5]

[25] Chen F, Roşu G. Parametric trace slicing and monitoring. In: Kowalewski S, Philippou A, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS 5505, Berlin: Springer-Verlag, 2009. 246−261. [doi: 10.1007/978-3-642-00768-2_23]

[26] Meredith PON, Jin D, Griffith D, Chen F, Rosu G. An overview of the MOP runtime verification framework. Int'l Journal on Software Tools for Technology Transfer, 2012,14(3):249−289. [doi: 10.1007/s10009-011-0198-6]

[27] Barringer H, Falcone Y, Havelund K, Giles R, David E. Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou D, Méry D, eds. Proc. of the Formal Methods (FM). LNCS 7436, Berlin: Springer-Verlag, 2012. 68−84. [doi: 10.1007/978-3-642-32759-9_9]

[28] Ivančić F, Yang Z, Ganai MK, Gupta A, Shlyakhter I, Ashar P. F-Soft: Software verification platform. In: Etessami K, Rajamani SK, eds. Proc. of the Computer Aided Verification (CAV). LNCS 3576, Berlin: Springer-Verlag, 2005. 301−306. [doi: 10.1007/11513988_31]

[29] Dutertre B, De Moura L. A fast linear-arithmetic solver for DPLL (T). In: Ball T, Jones RB, eds. Proc. of the Computer Aided Verification (CAV). LNCS 4144, Berlin: Springer-Verlag, 2006. 81−94. [doi: 10.1007/11817963_11]

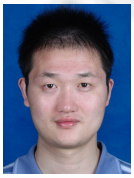
[30] Dutertre B, de Moura L. System description: Yices 1.0. In: Proc. of the 2nd SMT Competition (SMT-COMP). 2006. http://www.researchgate.net/publication/247695275_System_Description_Yices_1.0

[31] http://www.cprover.org/cprover-manual/cbmc.shtml

附中文参考文献:

[16] 屈婉霞,李暾,郭阳,杨晓东.谓词抽象技术研究.软件学报,2008,19(1):27−38. http://www.jos.org.cn/1000-9825/19/27.htm [doi: 10.3724/SP.J.1001.2008.00027]

[22] 何炎祥,吴伟,陈勇,徐超.基于 SMT 求解器的路径敏感程序验证.软件学报,2012,23(10):2655−2664. http://www.jos.org.cn/1000-9825/4196.htm [doi: 10.3724/SP.J.1001.2012.04196]



阚双龙(1988－),男,江苏连云港人,博士生,主要研究领域为模型检测,软件形式化验证.
E-mail: kanshuanglong@nuaa.edu.cn

陈哲(1981－),男,博士,副教授,CCF 会员,主要研究领域为模型检测,运行时验证.
E-mail: zhechen@nuaa.edu.cn

黄志球(1965－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为模型检测,嵌入式软件安全性,软件形式化验证.
E-mail: zqhuang@nuaa.edu.cn

徐丙凤(1986－),女,博士,CCF 学生会员,主要研究领域为模型检测,软件安全性.
E-mail: xubingfeng@nuaa.edu.cn