软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(6):1224−1235 [doi: 10.3724/SP.J.1001.2011.04016]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于 AOP 的运行时验证中的冲突检测*

张 献, 董 威+, 齐治昌

(国防科学技术大学 计算机学院,湖南 长沙 410073)

Conflicts Detection in Runtime Verification Based on AOP

ZHANG Xian, DONG Wei+, QI Zhi-Chang

(School of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: wdong@nudt.edu.cn

Zhang X, Dong W, Qi ZC. Conflicts detection in runtime verification based on AOP. *Journal of Software*, 2011,22(6):1224−1235. http://www.jos.org.cn/1000-9825/4016.htm

Abstract: Nowadays, instead of only verifying systems at the model level, current state-of-the-art verification techniques tend to focus on real code and real system's execution. Runtime verification checks the system's execution and tries to bridge the gap between formal verification techniques and real systems. However, this brings about some problems which usually do not appear in model-level verification. This paper analyses the problem in runtime verification. It defines two kinds of conflicts and lists their corresponding detection algorithms. These algorithms are implemented based on an open source runtime verification tool and some real cases are examined. The results demonstrate the effectiveness of the proposed method.
Key words: runtime verification; aspect-oriented programming; pointcut; conflict detection

摘 要: 现有的形式化验证方法除了在模型层面对系统进行验证以外,越来越倾向于直接针对系统的实际代码和具体运行.运行时验证技术验证的对象是具体程序,它试图把形式化验证技术部署到程序的实际运行过程中.然而在把形式化技术部署到实际运行过程中会出现一系列在模型层面验证通常不会出现的问题,对这些问题中的冲突现象进行了研究,定义了运行时验证技术中存在的两种冲突,并给出了相应的检测算法.最后,对这些算法进行了实现和实例研究,结果表明了该方法的有用性.
关键词: 运行时验证;面向方面编程;切入点;冲突检测
中图法分类号: TP311 **文献标识码**: A

随着信息技术的发展,计算机软件已渗透到日常生活的各个方面,在某些关键领域,计算机软件起着维护国计民生系统正常运转以及保证人们生命财产安全的重要作用.这样,提高计算机软件的可靠性以保证其按照人们期望运行的要求,变得十分迫切.现有提高软件可靠性的方法主要考虑静态、动态两方面,典型的包括程序的静态分析、模型验证、软件测试等技术,但这些技术都无法保证最后部署在实际系统上的程序是无错误的.这样就需要在程序运行时对程序进行监控,以避免危害的发生.

运行时验证(runtime verification)是一种新兴的轻量级验证技术,它把形式化验证技术和系统的实际运行结

* 基金项目: 国家自然科学基金(60970035, 91018013, 90818024, 60803042)
收稿时间: 2010-07-10; 定稿时间: 2011-03-29

© 中国科学院软件研究所 http://www.jos.org.cn

合起来,监控系统的运行,以保证系统的运行符合用户的性质期望.在运行时验证中,监控性质通常从系统需求中产生,通过监控器监控程序的运行来检查程序的运行是否满足监控性质.它不但可以有效地检测系统运行中的异常行为,而且也可在检测到性质违背时执行修复代码,对系统进行维护.

运行时验证技术在很多领域都有应用,比如代码在线修复、故障隔离和对移动代码的安全防范等.现行的运行时验证技术研究主要集中在运行时能够强制哪些性质,如何减少运行时验证技术给被监控程序带来的运行负担等方面.

在运行时验证领域,为使用户能够方便地表达要监控的性质,可以对一系列常用的性质建立模板库[1,2].这样,用户在使用时只需选用相应的模板,然后建立模板中性质符号到程序事件的映射即可.

这样,在运行时验证中,需要用户建立性质符号与程序运行事件间的映射关系.在现行的运行时验证工具中,这种映射关系一般可以通过面向方面编程(aspect-oriented programming,简称 AOP)[3,4]中的切入点(pointcut)或者自己定义的事件选择语言[5]来建立.由于 AOP 在模块化、易维护性和易使用性上都为运行时验证技术提供了良好支持,越来越多的运行时验证工具如 Tracematches[6],JavaMOP[7],LARVA[8]都采用基于 AOP 的技术.它们使用 AOP 中的织入(weaver)机制来插入监控代码、切入点(pointcut)语言来建立性质符号到事件的映射以及通知(advice)来包装修复代码.本文主要研究运行时验证中使用 AOP 中的切入点来建立性质符号到事件映射的问题.一般来说,AOP 中的切入点语言主要从程序的静态结构、动态值和时态关系等 3 个方面来选择事件[4].由于 AOP 切入点语言的复杂性,往往会给运行时验证带来一些在模型层面验证不常遇到的问题.

比如,由于切入点语言的复杂性,用户表达的监控性质中不同符号可能映射到程序运行中的同一事件.这种情况的产生往往是由于监控器实现者没有对程序的各个方面进行全面考虑,使得他原本认为完全不相交的事件集在某些特殊的运行轨迹处相交了[9],这样就有可能使监控器无法正确地监控其性质.此外,在运行时验证领域中,当多个监控器添加到程序时,各个监控器间也可能相互冲突.这是因为监控器监控的对象是程序的运行,它往往不会考虑到还会有别的监控器存在.这样,当程序运行时的某个事件触发多个监控器执行修复代码时,各个监控器都会从自己的角度对程序进行调整,这样有可能造成整体的调整效果违背了用户的期望.在监控性质日益复杂、监控器日益增多以及监控性质提出与程序开发往往分开进行的情况下,这些问题会变得愈发严重.

比如在银行转账系统中,一般要求每个事务操作都被记录,以保证系统的可跟踪性.由于转账系统可能经历了多次升级或者可能其不同的组件由不同的团队开发,这使得进行事务处理操作和进行记录的操作分布在各个模块中,命名方法也各异.当采用运行时验证技术对该系统的运行进行监控时,用户可能用 tracAction=*Transaction()||transaction*Begin() 来捕捉系统中所有的事务操作,而用 logAction=logTransaction()||logInDatabase()来捕捉所有的记录操作(*是通配符,表示匹配任何字符串;||表示这两个方法描述间是或的关系).这样,要监控的性质是不允许系统运行中出现两个事务操作而中间没有出现记录操作.当该性质被违背时,监控器要执行相应的修复代码:对第 2 个事务操作执行前插入记录操作,以保证系统的可追踪性.但是由于各操作的分散性和捕捉各操作语言的复杂性,用户可能不会注意到记录操作 logTransaction 同时被 tracAction 和 logAction 符号捕捉.这样会使得当系统出现未违背监控性质的⟨logTransaction,doTransaction⟩序列时,也会触发修复代码的运行.

现有主流的运行时验证工具对这些情况都没有进行考虑.在运行时验证工具 JavaMOP[7]中,其根据监控性质符号对程序的运行轨迹进行切片来选取被监控的事件轨迹.这样,当一个事件属于多个性质符号时,会使得程序运行中的一个事件对应于被监控事件流中的多个事件,并且还无法确定这多个事件间的时序关系,这样就有可能阻碍监控器的正确运行.比如,在 JavaMOP 下,由于 logTransaction 事件既会被 logAction 捕获,又会被 tracAction 捕获,系统运行序列⟨logTransaction,doTransaction⟩会被切片为符号序列⟨logAction,tracAction,tracAction⟩或⟨tracAction,logAction,tracAction⟩,这样的切片符号序列会阻碍监控器的正确运行.而在运行时验证工具 Tracematches[6]中,该序列可能会触发修复代码的执行,也有可能不会触发修复代码的执行,修复代码的执行与否取决于监控器的具体实现,这显然也不是我们所期望的.

针对上述问题,本文定义了运行时验证中的两种冲突:单个监控器内部的冲突和多个监控器之间的冲突.在

© 中国科学院软件研究所 http://www.jos.org.cn

此基础上给出了相应的检测算法,并在开源运行时验证工具 Tracematches 中进行了实现和案例研究.

1 基于 AOP 的面向运行时监控的事件选择表达式

运行时验证中,监控器监控的对象是程序的运行轨迹.由于程序的运行过程中发生的事件非常多,监控器感兴趣的轨迹往往只是程序运行轨迹在该监控器监控事件集合上的一个投影.现有的运行时验证工具一般通过 AOP 的切入点来表示监控的事件集.在 AOP 中,切入点(pointcut)是连接点(join point)的集合,而连接点是一组预定义的程序执行点,如方法调用、属性读写以及对象初始化等.在基于 AOP 的运行时验证领域,一般把事件定义为进入(enter)和退出(exit)该连接点的动作,运行时验证的对象为一条由这些事件组成的有穷轨迹.对此,有如下定义.

定义 1. 运行时验证的对象是一条有穷轨迹 $T=\langle e_0,e_1,\dots,e_n\rangle,e_i=\{enter,exit\}\times jp,0\leqslant i\leqslant n$,其中:$jp$ 为连接点,表示某个预定义的程序执行点,$enter\times jp$ 表示进入该连接点的事件;$exit\times jp$ 表示退出该连接点的事件.

在表达运行时验证中要监控的性质时,为了抽象、简练以及重用以前定义好的性质模板,一般用符号来表达要监控的事件集合,用一定的操作符来表达要监控事件的序列(比如线性时序逻辑中的 X,U 操作符等).这样,当把一个性质监控器部署到实际程序中去时,需要建立一个由性质符号到实际事件集合的映射.在建立这个映射的过程中,一般通过事件选择表达式来表示程序运行中的事件集合.

由于本文研究的是运行时验证中的冲突,而只有同一类型的事件(都为 enter 类型或者都为 exit 类型)才会产生冲突.这样,在后文中提及事件时,只用连接点表示即可.对此,关于事件选择表达式,有如下定义.

定义 2. 面向运行时监控的事件选择表达式 *p* 可以通过下列语法规则构造:

p:=*call*(*m*)|*execution*(*m*)|*new*(*n*)|*within*(*t*)|*withcode*(*m*)
    |*target*(*v*)|*args*(*v*,…)
    |*p*&&*p*|*p*||*p*|!*p*
MethodPattern=*TypePattern*
        [*TypePattern*.]*IdPattern*(*TypePattern*|“..”,...)
ConstructorPattern=[*TypePattern*.]*new*(*TypePattern*|“..”,...)
TypePattern=*IdPattern*[+][[]...]|!*TypePattern*
        |*TypePattern*&&*TypePattern*
        |*TypePattern*||*TypePattern*
        |(*TypePattern*)
IdPattern=包含通配符*和...的字符串

其中,$m\in$\{*MethodPattern*\},$n\in$\{*ConstructorPattern*\},$v\in$\{*IdPattern*\} or \{*ConstructorPattern*\}.

在该定义中,*call* 表示函数调用,*execution* 表示函数执行,*new* 表示对象的初始化,*within*(*t*)表示该事件只能出现在 *t* 表示的类中,*withcode*(*m*)表示该事件只能出现在函数 *m* 中,*target*(*v*)表示该事件发生的对象是 *v*,*args*(*v*)表示该事件发生时的参数是 *v*,…表示前一元素可以在该表达式中多次出现.这些解释都跟 AspectJ 中切入点(pointcut)解释相同,关于更详细的信息可见文献[4].

在该定义中,通过 *MethodPattern* 表示方法集合,*ConstructorPattern* 表示构造方法集合,*IdPattern* 表示标识符集合.特别地,为增强其表达能力,可以用 *TypePattern*+表示 *TypePattern* 类及其子类,可以在 *IdPattern* 中引入通配符*表示任何字母等.比如,*call*(*boolean java.util.AbstractCollection*+.*add**(*))就表示开始调用(enter)或完成调用(exit)*java.util.AbstractCollection* 类及其子类中所有以 *add* 为起始的、接受任何参数的方法的事件.

2 单个监控器内的冲突检测

在运行时验证中,一般采用线性时序逻辑、正规语言以及自动机来表达要监控的性质.同时,为方便用户使用,也有论文提出用 UML 图表达监控的性质[10].由于运行时验证监控的对象是程序的有穷运行轨迹,现行的运

© 中国科学院软件研究所 http://www.jos.org.cn

行时验证算法一般均把上述语言表达的监控性质转换为有限状态自动机,通过该自动机的状态是否迁移到接收态来判断程序的运行是否违反监控的性质.为一般性起见,在本文中均以有限状态自动机来表达监控性质.而根据运行时验证使用的场合不同(比如内存受限等),一般又分为确定和非确定有限状态自动机两种.为方便起见,本文统一自动机为确定有限状态自动机,非确定有限状态方法可以通过经典的算法转变为一个确定的有限状态自动机.因此,本文给出的冲突检测算法是适用于上述两种自动机的.

对监控器中的确定有限状态自动机,我们将其定义为:

定义 3. 确定有限状态自动机 A 是一个五元组(Σ,Q,q_0,δ,F),其中,Σ为字母表,Q 是有穷的非空状态集合,$q_0\in Q$ 为初始状态,$\delta:Q\times\Sigma\to Q$ 迁移函数,$F\subseteq Q$ 是接收状态集合.

由于在运行时验证领域,该自动机是由程序运行中发生的事件驱动的,这样,需要把该自动机的字母表映射到程序运行过程中出现的事件.由于自动机模型的抽象性以及程序运行过程中出现事件的纷繁复杂,实际上,一般均把每个字母映射成一个事件集合,该事件集合由定义 2 中的事件选择表达式来确定.在本文中,称该映射为ε.

定义 4. ε为字母表到事件表达式集合的一对一映射:$\Sigma\to P$.对应的由 $P\to\Sigma$的映射记为ε^{-1}.

传统的监控器一般只检查程序的运行是否满足监控的性质,而并不对程序的运行行为进行调整.但是在现行的主流运行时工具[6−8]中,均允许用户提供一段代码,以在性质违背时执行.因此,本文定义的监控器在其监控的性质被违背时,会执行一定的修复代码来调整程序的运行.该修复代码与具体程序的具体性质密切相关,一般由用户提出.这样,运行时验证中的监控器可以定义如下:

定义 5. 运行时验证中的监控器 M 可以定义为一个三元组(A,ε,C),其中,A 为定义 3 中的一个确定有限状态自动机,ε为确定有限状态自动机中字母表到事件选择表达式集合的一个映射,C 为当确定有限状态自动机到达接收状态时要执行的一段修复代码.

监控器根据自己当前所处的状态以及程序的运行过程中发生的被监控事件进行状态迁移,当监控器迁移到接收状态时,称为监控器对程序运行的一次接收.

定义 6. 运行时验证中的监控器 $M=(A,\varepsilon,C)$接收程序的运行序列 $T=\langle e_0,e_1,\ldots,e_n\rangle$是指存在字母表中的串 $w=\langle a_0,a_1,\ldots,a_n\rangle$,使得对$\forall 0\leqslant i\leqslant n$,有$\delta(q_i,a_i)=q_{i+1}\wedge e_i\in\varepsilon(a_i)\wedge q_{n+1}\in F$ 成立.

由于本文定义监控器中的自动机为确定有限状态自动机,这样在性质表达层面,对于自动机中的某一状态 q,如果该状态存在两条迁移边,设其分别标记为 a,b,有$\delta(q,a)\neq\delta(q,b)$.由于该自动机为确定自动机,故有 $a\neq b$ 成立.但在现行主流的运行时验证工具中,一般均采用定义 2 中的事件选择表达式来表示事件集合,由于该表达式的复杂性,用户在建立字母表到事件表达式间的映射时,有可能使得在$\delta(q,a)\neq\delta(q,b)$的情况下,存在事件 e 使得 $e\in\varepsilon(a)$和 $e\in\varepsilon(b)$同时成立.这样在实际监控中,如果监控器处于状态 q,此时接收到事件 e,则监控器既有可能迁移到状态$\delta(q,a)$,也有可能迁移到状态$\delta(q,b)$.这样,在性质层面上的一个确定自动机在实现层面上变得不确定了,从而使得监控器的正确运行与否变得与实现相关,而不是完全取决于它要监控的性质和程序运行的轨迹.因此,需要对该种情形进行检测.我们给出监控器内部冲突的定义:

定义 7. 监控器 M 中存在冲突当且仅当存在状态 $q\in Q$,有字母表中的两个字母 $a,b(a\neq b)$,使得下列条件同时成立:(1) $\delta(q,a)\neq\delta(q,b)$;(2) 存在事件 e,有 $e\in\varepsilon(a)\wedge\varepsilon(b)$.

要检测监控器内部是否存在冲突,就需要检查是否存在一个事件同时满足两个不同的事件表达式.由于面向方面编程语言中切入点的强表达能力,一般认为这个问题的复杂度是 NP 完全的[11,12].对此,本文采用了一种更严格的方式,主要从语法层面来判断两个表达式是否会选择共同的事件,从而使得该问题在绝大部分情况下变得多项式可解.

单个监控器内的冲突检测算法见算法 1.算法 1 中的 list 的元素为一个包含了自动机状态 q、该状态迁出边标记 a 和迁出边标记 b 的一个记录.该算法遍历自动机中的每个状态,如果发现该状态有两个以上的出边,则判断这两个出边是否迁移到不同的状态,然后用 canUnify 函数判断这两个出边对应的事件选择表达式是否会选取共同事件.canUnify 函数接收字母 a,b 对应的事件选择表达式为参数,判断这两个事件表达式是否会选择共同

© 中国科学院软件研究所 http://www.jos.org.cn

的事件.canUnify 函数的算法实现见算法 2.

算法 1. 单个监控器内的冲突检测算法.

```
Input:监控器 M=(A,ε,C);
Output: A list of record(q,a,b).
for each q∈Q do begin
    if (the number of outgoing edge of q<2)
        continue;
    for every pair of outgoing edge label a, b do begin
        if (δ(q,a)≠δ(q,b)∧canUnify(ε(a),ε(b)))
            list.add(new record(q,a,b))
    end for
end for
```

算法 2. 判断两个事件选择表达式是否选择共同事件的 canUnify 算法.

```
Input:事件选择表达式 p1,p2;
Output: Boolean.
p1List=convertToDNF(p1);
p2List=convertToDNF(p2);
for every conjunctive term p1term in p1List do begin
    for every conjunctive term p2term in p2List do begin
        if (intersect(p1term,p2term))
            return true;
    end for
end for
return false;
```

在 canUnify 算法中,convertToDNF 根据德摩根律($\neg(p_1\&\&p_2)=\neg p_1||p_2;\neg(p_1||p_2)=\neg p_1\&\&p_2$)和分配律($p_1\&\&(p_2||p_3)=(p_1\&\&p_2)||(p_1\&\&p_3);p_1||(p_2\&\&p_3)=(p_1||p_2)\&\&(p_1||p_3)$),把一个事件选择表达式转化为一个析取范式,该析取范式用 *list* 来存储,每个 *list* 中的元素为一个合取项.对于析取范式中的每一个合取项(即 *list* 中的每个元素),用 intersect 函数来判断它们之间是否会选取共同事件.其中的 intersect 算法见算法 3.

算法 3. 判断两个关于事件选择表达式合取项是否选择共同事件的 intersect 算法.

```
Input:合取项 p1term,
      合取项 p2term;
Output: Boolean.
for every literal1 in p1term do begin
    for every literal2 in p2term do begin
        if (literal1.type==literal2.type) do begin
            switch(literal1.type) do begin
                case "call(m)", "execution(m)", "withcode(m)":
                    if (!MethodPatternMatch.match(literal1,literal2))
                        return false;
                    break;
                case "new(n)":
                    if (!ConstructorPatternMatch.match(literal1,literal2))
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
                    return false;
                break;
            case "cflow(p)", "!p":
                p1=literal1.getInternPointcut();    p2=literal2.getInternPointcut();
                if (!canUnify(p1,p2))
                    return false;
                break;
            case "within(t)":
                if (!TypePatternMatch.match(literal1,literal2))
                    return false;
                break
            case "argv(v)" "target(v)":
                if (!TypeOrIdPatternMatch.match(literal1,literal2))
                    return false;
                break
        end switch
    end if
  end for
end for
```

在 *intersect* 函数中,对合取项中的每个文字**(即简单事件选择表达式,其不包含&&和||连接符)进行判断,只有对合取项中的每个文字另外合取项中都存在文字和它会选择共同的事件时,才返回为真.对每个文字判断它们是否会选择共同事件时,算法根据文字的类型进行不同的判断.在判断的过程中,除用到每个文字的语法信息外,还需用到文字所引用类的层次结构信息.比如,在判断文字 *call*(*void A+.b*(·))和文字 *call*(*void B.b*(·))是否会选择共同事件时,需要知道类 *A* 的继承层次信息,以判断 *B* 是否为 *A* 的子类.

单个监控器内冲突检测算法复杂度分析:在监控器内部的冲突检测中,假定监控器中的自动机有$|Q|$个状态,该算法需要对每个状态的所有出边进行两两检测,则两两检测需要的次数为 $O(|\Sigma|^2)$,该检测只需遍历字母表一次,在别的状态中用到时,只需直接取其结果即可,故总的检测次数为 $O(|Q|+|\Sigma|^2)$.假定字母表字母对应事件选择表达式的平均长度为$|p|$,则检测两个事件选择表达式是否会选择共同事件的平均复杂度是 $O(|p|^2)$,故算法总的平均复杂度为 $O(|Q|+|\Sigma|^2\times|p|^2)$.但在最坏情况下,由于把一个公式转化为吸取范式可能在长度上带来指数级增长,从而使得总的算法复杂度为指数级的.

3 多个监控器间的冲突检测

在运行时验证领域,当程序有多个监控器部署时,监控器之间也可能产生冲突.随着监控器的日益复杂以及数目的增多,监控器间的冲突情形更加容易发生.

在定义多个监控器间的冲突时,首先需要定义针对每个监控器的程序运行轨迹的概念.不同的监控器有不同的监控事件集,针对某一监控器而言,程序的运行轨迹就是程序运行过程中出现的为该监控器所监控的事件组成的轨迹.而整个程序的运行轨迹可以看成针对部署在该程序中的所有监控器而言,是程序运行过程中出现的为任一监控器监控的事件组成的轨迹.也就是说,可以把程序针对单个监控器的轨迹,看成是该程序针对所有监控器的运行轨迹在该监控器监控事件集上的一个投影.在处理多个监控器间的冲突时,我们采用了保守性假

** 在命题逻辑中,形如$(p\wedge q)\vee(\neg p\wedge r\wedge s)$的公式称为吸取范式,$(p\wedge q)$和$(\neg p\wedge r\wedge s)$称为合取项,$p,q,\neg p$ 等称为文字.在事件选择表达式中,$\wedge$对应于&&,$\vee$对应于||,$\neg$对应于!,p,q 一般对应于 *call*(*m*)或 *target*(*v*)这样的简单事件选择表达式.

© 中国科学院软件研究所 http://www.jos.org.cn

设,假定所有监控器的修复代码是相互冲突的,只要触发它们同时执行,冲突就会发生.这样,也使得多个监控器间的冲突检测总可以转化为对两个监控器间的冲突检测.因而在本文中,只处理两个监控器间的情形.

定义 8. 假定程序中只存在两个监控器 $M_1=(A_1,\varepsilon_1,C_1)$和 $M_2=(A_2,\varepsilon_2,C_2)$,该程序的一条运行轨迹为 $T=\langle e_0, e_1,\ldots,e_n\rangle$,则对$\forall 0\leqslant i\leqslant n$,有 $e_i\in\varepsilon(a_1)\vee e_i\in\varepsilon(a_2)$成立,其中,$a_1\in\Sigma_1,a_2\in\Sigma_2$;该程序针对监控器 M_1 的运行轨迹为 $T|_{M_1}=\langle e_0', e_1',\ldots,e_m'\rangle$,对$\forall 0\leqslant j\leqslant m$,有 $(\exists a_j.\, a_j\in\Sigma_1\wedge e_j'\in\varepsilon(a_j))\wedge(\exists i.\, 0\leqslant i\leqslant n, e_j'=e_i)$.

在定义针对每个监控器运行轨迹的基础上,可以把两个监控器之间的冲突定义为:

定义 9. 监控器M_1和M_2间存在冲突当且仅当存在一条程序的运行轨迹$T=\langle e_0,e_1,\ldots,e_n\rangle$以及两个字母$a_1\in\Sigma_1$, $a_2\in\Sigma_2$,使得如下两个条件同时成立:

(1) $e_n\in\varepsilon(a_1)\wedge e_n\in\varepsilon(a_2)$;

(2) 监控器 M_1 接收轨迹 $T|_{M_1}$ 且监控器 M_2 接收轨迹 $T|_{M_2}$.

也就是说,两个监控器间存在冲突,是指程序存在一条运行轨迹,使得该轨迹被这两个监控器都接收,并且该运行轨迹的最后一个事件会触发两个监控器的修复代码同时执行.定义 9 并没有给出检测两个监控器间是否存在冲突的算法.在定义 9 中没有提及被监控程序,这是因为我们认为两个监控器间的冲突是监控器之间固有的性质,与具体的程序无关.即使给出了某一程序,也无法枚举出该程序的所有运行路径(特别简单的程序除外).

根据定义 9,检测两个监控器间是否存在冲突,可以转化为判断两个自动机接收的语言是否有交集.但由于即使针对同一程序的同一运行,不同的监控器根据其监控的事件集不同,也会对应不同的轨迹.在检测两个监控器间是否存在冲突时,我们感兴趣的是被两个监控器共同监控的事件,因为只有这些事件才可能导致冲突的发生.对此,监控器间冲突的检测算法分为两个阶段:第 1 个阶段判断定义 9 中条件(1)是否成立,第 2 个阶段判断定义 9 中的条件(2)是否成立.只有通过了第 1 个阶段的判断,才需进行第 2 个阶段的判断.

第 1 阶段的判断很简单,只需判断两个监控器中自动机接收态的迁入边字母对应的事件选择表达式是否会选择共同事件.判断两个事件表达式是否会选择共同事件,可以通过本文第 2 节的算法来判断.如果没有共同事件,则说明这两个监控器间不存在冲突;否则,则转入第 2 阶段的判断.

第 2 阶段的判断可以这样进行:首先,通过第 2 节的算法来判断两个监控器间事件选择表达式是否会选择共同事件,取得这些会选取共同事件的表达式集合;然后,根据ε^{-1} 通过事件选择表达式找到在各自自动机中对应的字母.如果来自不同自动机的字母选取了共同事件,则在这些字母间建立等价关系.这样就在各自自动机字母表中选取了一个子集,对该子集中的每个字母另外子集中都存在与其具有等价关系的字母.实际上是为这两个自动机的字母表找到了一个同名子集;最后,把自动机分别投影到该子集上,判断投影后的自动机接收的语言是否有交集,如果有交集,则表示这两个监控器间存在冲突.

定义 10. 确定有限状态自动机A是一个五元组(Σ,Q,q_0,δ,F),字母表$\Sigma_1\subseteq\Sigma$,自动机A在Σ_1上的投影记为$A|\Sigma_1=(\Sigma_1,Q',q_0',\delta',F')$,其中有:

(1) $Q'\subseteq Q, q_0'=q_0, F'\subseteq F$;

(2) 对 $A|\Sigma_1$ 中每个迁移关系 $\delta'(q_1',a)=q_2'$,在 A 中有 $\delta(q_1',w_1aw_2)=q_2'$ 成立,其中,$w_1,w_2\in(\Sigma-\Sigma_1)^*$.

算法 4. 把自动机投影到其字母表子集的算法.

Input:自动机 $A=(\Sigma,Q,q_0,\delta,F)$,$A$ 中字母表子集Σ_1;

Output:自动机 A 在其子集Σ_1上的投影 $A|\Sigma_1$.

```
for each q∈Q do begin
    for every outgoing edge label a do begin
        q′=δ(q,a);
        if (all the edge labels from q to q′⊄Σ1)
            q=collapse(q,q′);
    end for
```

© 中国科学院软件研究所 http://www.jos.org.cn

end **for**

算法中的 collapse 操作是接收自动机中的两个状态,把这两个状态合二为一,原来各个状态的迁入迁出边一并转移到合并后的状态.如果合并前的两个状态间存在迁移边,则舍弃这些迁移边.如果这两个状态的某一个状态是最终状态,则合并后的状态仍为最终状态;如果这两个状态的某一状态是起始状态,则合并后的状态也为起始状态.

通过上述算法,就可以把不同监控器中的自动机投影到共同的字母表(因为来自不同字母表中的字母根据它们的事件选择表达式是否会选择共同事件建立了等价关系)上,这样,检测两个监控器之间是否存在冲突就转化为两个投影后的自动机接收的语言是否有交集.由于这两个自动机的字母表是相同的,这个问题可以通过经典的算法检验出来,这里就不列举了.

算法复杂度分析:在检测监控器间存在的冲突时,第 1 阶段的复杂度为 $O(|\Sigma_1|\times|\Sigma_2|\times|p_1|\times|p_2|)$,其中,$|p_1|,|p_2|$分别为两个自动机中字母表对应事件选择表达式的平均长度.在第 2 阶段中,由于需要检测自动机间每个字母对是否选取了共同事件,总的检测次数是 $O(|\Sigma_1|\times|\Sigma_2|)$.由第 2 节的分析可知,检测两个事件选择表达式是否会选择共同事件的平均复杂度是 $O(|p_1|\times|p_2|)$,所以从各自字母表中建立同名子集的平均复杂度为 $O(|\Sigma_1|\times|\Sigma_2|\times|p_1|\times|p_2|)$.把两个自动机投影到字母集的复杂度分别为 $O(|Q_1|\times|\Sigma_1|)$和 $O(|Q_2|\times|\Sigma_2|)$,检测两个自动机是否有交集的复杂度为 $O(|Q_1|\times|Q_2|\times|\Sigma|)$,其中,$|\Sigma|$为两个自动机中字母表同名子集中的成员个数,有$|\Sigma|\leqslant|\Sigma_1|$和$|\Sigma|\leqslant|\Sigma_2|$成立.故检测两个监控器间总的平均复杂度为 $O(|\Sigma_1|\times|\Sigma_2|\times|p_1|\times|p_2|+|Q_1|\times|Q_2|\times|\Sigma|)$.

总之,在运行时验证领域中,当检测两个监控器间的冲突时,一方面不要像 AOP 中的冲突检测那样,需要对字母表中的每个事件(通过定义 3 中的映射)进行冲突检测,而只需对可能触发修复代码的事件进行冲突检测,这样就极大地减少了检测的复杂度;另一方面,运行时验证领域中的冲突不仅取决于当前事件,也取决于历史事件,这使得它具有与 AOP 中冲突检测不同的特点.

同时,文中算法还采用了最保守的策略,假定被同一个事件触发的不同监控器的修复代码是相互冲突的.实际上,对一些较简单的修复代码,可以通过对它们进行静态分析来确定它们之间是否互相冲突.但在理论上,由于修复代码可以是任意代码,因而不存在一个确定的算法能够判断任意两个修复代码之间是否会产生冲突,这也是我们的定义采用了这种保守性假设的原因.

4 实现与案例研究

Abc[13]是由英国 Oxford 大学和加拿大的 McGill 大学实现的能够处理 AspectJ 语法成分的一个可扩展编译器.Tracematches[6]是建立在其上的一个运行时验证工具,它使用正规语言描述要监控的性质,使用 AspectJ 的切入点来选择要监控的事件,但其没有考虑运行时验证中监控器可能存在的内部或之间的冲突.我们在 Tracematches 的基础上对其进行扩充,增加了冲突检测功能.

Tracematches 中,把与监控器相关的信息都存放在 abc.tm.weaving.aspectinfo.TraceMatch 类中,故在进行监控器内部的冲突检测时,只需在监控器被编译后(在织入前)在该类的实例中进行检查.由于 Tracematches 在内部已自动把正规语言转换为确定有限状态自动机了,从而本文中的算法可以直接使用.同时,Tracematches 把所有的监控器信息都存放在 abc.tm.weaving.aspectinf.TMGlobalAspectInfo 类中的 traceMatchList 属性中,故在检测监控器间的冲突时,只需对该属性包含的所有 TraceMatch 实例进行检测.对其扩充实现的框架如图 1 所示.主要修改包括:在 TMGlobalAspectInfo 类中增加了检测监控器间冲突的方法 detectConflictsBetweenMonitors,在 TraceMatch 类中增加了检测监控器内部冲突的方法 detectConflictInMonitor,在 PointCut 类中增加了判断两个 PointCut 能否选择相同事件的方法(canUnify)以及增加了若干个 Matcher 类判断 pattern 是否有相交.

© 中国科学院软件研究所 http://www.jos.org.cn

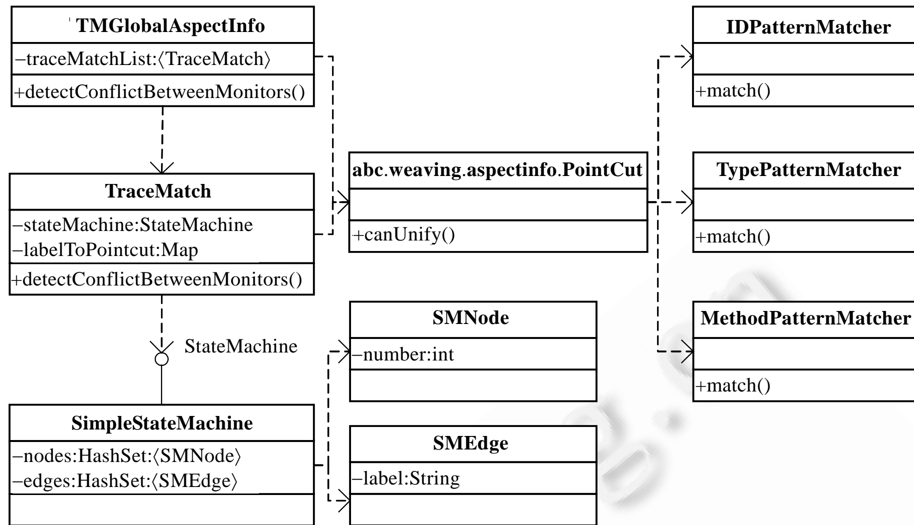

Fig.1 Implementation of conflicts detection about monitors based on Tracematches
图 1 基于 Tracematches 的监控器冲突检测算法的实现

在完成上述实现的基础上,我们对文献[9]中的实例进行了检测.结果显示能够检测出其中的冲突.比如对文献[9]中提及的信用卡处理系统实例,系统中包含一个 CreditCardProcess 类,该类包含了许多关于事务处理的方法,这些方法都以 Transaction 结尾.为了保证对信用卡操作的可追踪性,要求每个事务操作都被记录.在对该性质进行监控时,该性质的监控器如图 2 所示.图中的 logTransaction 操作用来对该类的事务操作进行记录.在对该监控器进行冲突检测时发现,该监控器的实现者没有注意到对事务操作进行记录的操作也是以 Transaction 结尾,这样便给监控器带来了冲突.要解决这个冲突,只需把 logTransaction 操作进行改名,使其不以 Transaction 结尾,或者把事件选择表达式 CreditCardProcessor.*Transaction() 加强为 CreditCardProcessor.*Transaction()&&!CreditCardProcessor.logTransaction().该例子虽然比较简单,但从中可以看出,随着监控器内部状态、迁移边的增多以及被监控程序的日益复杂,进行监控器内部的冲突检测是非常必要的.

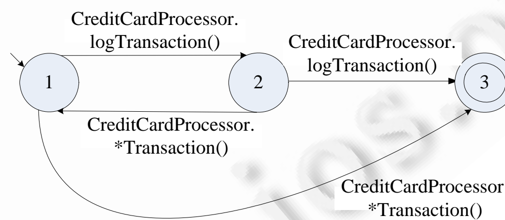

Fig.2 A Logger monitor deployed on a credit card processing system
图 2 部署在信用卡处理系统中的一个 Logger 监控器

此外,我们还对一系列部署在企业内部邮件系统的监控器进行了冲突检测[14],结果发现,如下的两条规则存在冲突:其中,规则 1 为当用户登录邮件系统并读取了敏感数据后,再发送邮件时必须以加密的形式发送;而另外一条规则为用户登录邮件系统后再发送邮件时,如果发送的邮件中附带了以 exe 或 vbs 为后缀名的的文件时,需要用户的确认.这两条规则如图 3 所示.

这样,在这两个监控器部署到系统上时,当程序中出现〈login readSensitiveData sendFile〉序列时,会使得两个

© 中国科学院软件研究所 http://www.jos.org.cn

监控器的修复代码同时被触发.如果当 encrypt 先执行时,会使得待用户确认的文件名以加密的形式出现,从而使用户无从选择.

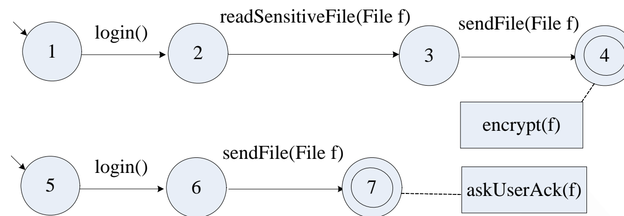

Fig.3 Two monitors deployed on a mail system

图 3 部署在邮件系统的两个监控器

5 相关工作

据我们所知,现行冲突检测的研究主要集中在面向方面编程领域.运行时验证领域中的冲突检测只在文献[15]中有所提及,该文给出了运行时验证中冲突的一个定义及相应的一些检查规则.但是该定义相当粗略,对程序违反性质后要执行的动作也作了过强的假设,并且没有给出规则的实现算法.同时,它也没有考虑到本文定义的监控器内部冲突的情形.

文献[14]提出了一套监控器组合语言,该组合语言一方面可以把简单的监控器组合成复杂的监控器;另一方面又可以对监控器进行组合控制,以避免它们之间的冲突.但是,该文并未提及冲突检测的算法,监控器要靠人工来组合,冲突也需要人工进行检测.

面向方面编程中的冲突检测是面向方面编程研究中的一个热点,主要集中在多个方面在同一连接点(join point)存在多个通知(advice)时如何确定各个通知的织入顺序或者修改通知,以避免冲突的发生.比如,AspectJ 中提供的 declare precedence 关键字来指定同一连接点各个通知的相对执行的顺序.文献[16−19]等采用了各种方法对这个问题进行研究.但我们研究的冲突是在运行时的验证背景下,我们的冲突不但取决于当前事件,而且还取决于历史以往的事件;这样就使得传统面向方面编程中的冲突在运行时验证领域不一定是冲突.比如在图 1 中的第 2 个监控器中,如果添加一个由状态 6 到状态 5 的标记为 readSensitiveFile 的迁移,会使得该冲突不会发生;但在传统的面向方面编程领域,则依然会认为这是个冲突.此外在运行时监控领域,大部分事件触发的通知都只是驱动监控器运行,只有当监控器到接收状态时才会触发修复代码执行,这样对运行时验证中的冲突检测,只需检测修复代码间的冲突,而不像面向方面编程中需要对每个通知(包括驱动监控器运行的通知和触发修复代码的通知)检测其中的冲突.

6 结论与工作展望

本文对运行时验证中通过 AOP 中的切入点来建立监控性质与程序运行事件间的映射进行了分析,并指出了切入点的复杂性可能会给监控器带来冲突.本文定义了两类冲突:监控器内部的冲突以及监控器间的冲突,并给出各自的检测算法.最后,在开源运行时验证工具 Tracematches 上对这些算法实现,并对一些实例进行了分析、检验,结果显示了我们算法的有用性.

据我们所知,这是在运行时验证领域最先提出的冲突定义和检测算法.我们给出的检测算法虽然是针对 AOP 中切入点语言的,但事实上,由于在性质表达层面的抽象性,在任何一种建立由性质符号到软件实际运行中具体事件的映射的语言中,如果要求表达能力足够强(因为要涵盖纷繁复杂的具体事件,一般不能采用逐个枚举的方法),本文定义的两种冲突都有可能发生.因此,本文中的定义和检测算法对别的事件选择语言也有一定的借鉴意义.

本文把监控器的冲突定义为尽可能地独立于具体应用场景,只在事件选择表达式中用到了一些程序的类层次结构信息.这种冲突定义反映了监控器本身固有的性质,能够反映冲突产生的根源.如果运行时验证中直接

© 中国科学院软件研究所 http://www.jos.org.cn

采用 AOP 中的冲突定义,会使得冲突与应用场景相关.在一个应用场景中,监控器不发生冲突并不能保证该监控器部署到别的场景也不会发生冲突.但这种定义也使得检测事件选择表达式之间是否存在共同事件的复杂度是 NP 完全的.为使算法变得现实可行,本文的检测算法主要基于表达式的语法信息,采用更加严格的标准判断两个事件选择表达式能否选择共同的事件.这样就使得文中的检测算法会出现漏报现象,比如对于两个事件选择表达式 *cflow*(*call*(*A*.*a*()))和 *cflow*(*call*(*B*.*b*())),文中的算法会判断出它们不会选择共同的事件;但在程序的具体实现中,如果 *A*.*a* 方法的实现中调用了 *B*.*b* 方法,或者 *B*.*b* 方法的实现中调用了 *A*.*a* 方法,则有可能使得这两个表达式选择共同的事件.由于我们判断两个表达式是否选择了共同事件的标准更为严格,这虽然会导致漏报现象的发生,但不会出现误报现象.

由于AOP的广泛应用,使用AOP中切入点语言来选择事件已经成了运行时验证领域的一个主流选择.本文定义的冲突及其检测方法能够有效地检测出监控器内部和监控器之间的冲突,是现有运行时验证技术的有效补充.

AOP 中切入点的强表达力会给监控器内部或监控器之间带来冲突,同时也使得不存在一个多项式复杂度的检测算法.在以后的工作中,我们准备根据运行时验证领域的特点对 AOP 中的切入点语言进行限定,以更好地达到表达力强度和检测算法复杂度间的一个平衡.

References:

[1] Bauer A, Leucker M, Schallhart C. Runtime verification for LTL and TLTL. ACM Trans. on Software and Methodology (TOSEM), upcoming. 2010.

[2] Dwyer MB, Avrunin GS, Corbett JC. Property specification patterns for finite-state verification. In: Proc. of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98). New York: ACM Press, 1998. 7−15. [doi: 10.1145/298595.298598]

[3] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-Oriented programming. In: Proc. of the European Conf. on Object-Oriented Programming. LNCS 1241, Springer-Verlag, 1997. 220−242.

[4] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Proc. of the ECOOP 2001. London: Springer-Verlag, 2001. 327−353.

[5] Kim MZ, Viswanathan M, Kannan S, Lee I, Sokolsky O. Java-MaC: A run-time assurance approach for Java programs. Formal Methods in System Design, 2004,24(2):129−155. [doi: 10.1023/B:FORM.0000017719.43755.7c]

[6] Allan C, Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J. Adding trace matching with free variables to AspectJ. In: Proc. of the OOPSLA 2005. New York: ACM Press, 2005. 345−364.

[7] Chen F, Jin DY, Meredith P, Rosu G. Monitoring oriented programming—A project overview. In: Proc of the ICICIS 2009. New York: ACM Press, 2009. 72−77.

[8] Colombo C, Pace GJ, Schneider G. LARVA—Safer monitoring of real-time Java programs. In: Proc. of the 7th IEEE Int'l Conf. on Software Engineering and Formal Methods (SEFM). Hanoi: IEEE Computer Society, 2009. 33−37. [doi: 10.1109/SEFM.2009.13]

[9] Jones M, Hamlen KW. Disambiguating aspect-oriented security policies. In: Proc of the AOSD 2010. New York: ACM Press, 2010. 193−204. [doi: 10.1145/1739230.1739253]

[10] Li XD, Qiu XK, Wang LZ, Lei B, Wang WE. UML state machine diagram driven runtime verification of Java programs for message interaction consistency. In: Proc. of the 2008 ACM Symp. on Applied Computing (SAC 2008). New York: ACM Press, 2008. 384−389. [doi: 10.1145/1363686.1363781]

[11] Palm J, Wu PC, Lieberherr K. Understanding aspects through call graph enumeration and pointcut satisfiability. Technical Report, NU-CCIS-04-01, Boston: Northeastern University, 2004.

[12] Lieberherr KJ, Palm J, Sundaram R. Expressiveness and complexity of crosscut languages. In: Proc. of the Workshop on the Foundations of Aspect Oriented Programming Languages (AOSD 2005). Chicago, 2005.

[13] Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhoták J, de Moor O, Sereni D, Sittampalam G, Tibble J. abc: An extensible AspectJ compiler. In: Proc. of the Trans. on Aspect-Oriented Software Development I. LNCS 3880, Heidelberg: Springer-Verlag, 2006. 293−334. [doi: 10.1007/11687061_9]

© 中国科学院软件研究所 http://www.jos.org.cn

[14] Bauer L, Ligatti J, Walker D. Composing expressive runtime security policies. ACM Trans. on Software Engineering Methodology, 2009,18(3):1−43. [doi: 10.1145/1525880.1525882]

[15] Malakuti S, Bockisch C, Aksit M. A rule set to detect interference of runtime enforcement mechanisms. In: Proc. of the 20th Annual Int'l Symp. on Software Reliability Engineering (ISSRE 2009). Mysore: IEEE Computer Society Press. 2009. 16−19.

[16] Katz E, Katz S. Incremental analysis of interference among aspects. In: Proc. of the 7th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008). New York: ACM Press, 2008. 29−38. [doi: 10.1145/1394496.1394500]

[17] Aksit M, Rensink A, Staijen T. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proc. of the 8th ACM Int'l Conf. on Aspect-Oriented Software Development (AOSD 2009). New York: ACM Press, 2009. 39−50. [doi: 10.1145/1509239.1509247]

[18] Kniesel G. Detection and resolution of weaving interactions. In: Proc. of the Trans. on Aspect-Oriented Software Development V. Heidelberg: Springer-Verlag, 2009. 135−186. [doi: 10.1007/978-3-642-02059-9_5]

[19] Bodden E. Specifying and exploiting advice-execution ordering using dependency state machines. In: Proc. of the Int'l Workshop on the Foundations of Aspect-Oriented Languages (FOAL). 2010.

张献(1982－),男,湖南浏阳人,博士生,主要研究领域为运行时验证.

齐治昌(1942－),男,教授,博士生导师,主要研究领域为软件工程.

董威(1976－),男,博士,教授,CCF 会员,主要研究领域为高可信软件工程,高可信软件技术.

© 中国科学院软件研究所 http://www.jos.org.cn