ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.4, April 2007, pp.796−807
DOI: 10.1360/jos180796
© 2007 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于模式分析的特征交互检测*

左继红+, 王千祥, 梅 宏

(北京大学 信息科学技术学院 软件研究所,北京 100871)

Detecting Feature Interactions by Pattern Analysis

ZUO Ji-Hong+, WANG Qian-Xiang, MEI Hong

(Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

+ Corresponding author: Phn: +86-10-62757801, E-mail: zuojh@pku.edu.cn

Zuo JH, Wang QX, Mei H. Detecting feature interactions by pattern analysis. *Journal of Software*, 2007,18(4): 796−807. http://www.jos.org.cn/1000-9825/18/796.htm

Abstract: This paper aims at proposing an approach that can detect feature interactions through patterns analysis. The basic idea is to extract the common conflict patterns from the known feature interactions and reuse them to detect the unknown ones. With this approach, the conditions for conflicts are described using a set of predicate formulae and the system model is specified in Java language. With the aid of an external tool, all the execution scenarios can be systematically explored. During the execution of the system model, all the feature behaviors will be collected and analyzed. Once a conflict condition is satisfied, a conflict report is produced. The approach has been applied to an E-mail system. The experimental results show that the approach can effectively detect feature interactions, both the known and the unknown, after handling more than 1 million execution scenarios.
Key words: feature interaction; model checking; model; pattern matching; Java

摘 要: 旨在提出一种基于模式分析的特征交互检测方法,其基本思想在于从已知的交互中提取具有共性的冲突模式,并以之检测新的特征交互.该方法使用一组谓词公式描述交互发生的条件,使用 Java 语言对系统建模,借助于一个外部工具,系统模型可以遍历所有的运行场景.在模型运行期间,所有特征的行为将被收集和分析,一旦发现某个交互的条件得到满足,即产生冲突报告.该方法被用于一个 E-mail 系统的分析.实验结果显示,在处理了超过 100 万个运行场景后,该方法能够有效地检测出已知和未知的特征交互.
关键词: 特征交互;模型检测;模型;模式匹配;Java
中图法分类号: TP311 文献标识码: A

为了适应丰富多变的用户需求,许多软件系统采用模块化开发和演化机制:先设计出一个稳定的基础子系统(简称基系统),然后通过插入新的特征或服务来增强系统的功能.例如,E-mail 系统除了提供基础的邮件收发功能以外,还提供一组可选的特征,如加密、解密、签名、验证等等[1].尽管这种开发方法有助于提高软件开发的效率,但是,它也会导致特征交互问题(feature interaction problem),即当不同的特征或服务应用到基系统中时,

* Supported by the National Natural Science Foundation of China under Grant Nos.60233010, 90412011 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321805 (国家重点基础研究发展规划项目(973))

Received 2006-07-12; Accepted 2006-09-30

可能发生非预期的相互干扰.第 2.1 节列举了 E-mail 系统的一个特征交互实例.

特征交互问题并不仅局限于 E-mail 系统,实际上,这个问题最先出现于电信领域.现在的电信网络通常会部署数十个特征以提供增值服务,但这些特征会发生不符合预期的交互,导致混乱甚至出现安全漏洞[2,3].基于 SIP(session initiation protocol)协议的 VoIP 系统能够提供不同于传统电信的某些服务,而这些服务也会发生特征交互[4].此外,由于 SIP 允许用户使用 CPL(call processing language)脚本语言定制服务,特征交互也会在脚本的层次发生[5].近来在 AOP(aspect oriented programming)领域,aspect 之间也被发现存在着交互[6],随着 aspect 被用于实现具有贯穿特性的特征,特征交互可能表现为 aspect 交互[7].特征交互在多个领域的广泛存在表明这个问题是软件工程的基础问题[8].因此,如何预防、检测和解决特征交互就成为软件开发和演化的一个关键问题.对于特征交互的检测已提出许多不同的方法,它们大致可分为在线检测和离线检测.在线技术是在运行时刻对程序进行检测;而离线技术是在设计阶段对形式化模型进行分析[9].对相关工作的深入介绍请参见第 1 节.

本文提出一种基于模式分析的特征交互检测方法.该方法基于如下观察:在不少特征交互中常会发现具有共性的行为模式.例如,可以从第 2.1 节的例子提取出如下模式:给定一个消息 m,若特征 $F1$ 修改了域 $m.p1$,之后,特征 $F2$ 读取域 $m.p2$,且 $m.p2$ 的值依赖于 $m.p1$ 的值,则容易导致交互.因此,如果可以设法从已知的特征交互中抽取出反复出现的冲突模式,这些模式就可以被用于检测新的特征交互.

为了实现基于模式的分析,首先要求得到系统的可执行模型.我们使用 Java 语言对系统建模,相比于其他形式化语言,Java 语言更容易掌握和理解.每个特征被实现为一个 Java 类,所有的特征将按照系统的体系结构被组合起来.此外,还需要一个场景产生器用于驱动模型的运行.场景产生器负责模拟外部环境的动作和事件,以产生运行场景.场景产生器受一个外部工具 Java PathFinder(JPF)的控制.JPF 是专门用于对 Java 程序进行模型检测的工具[10].当 JPF 启动时,它会驱动场景产生器产生各种可能的运行场景.每当一个运行场景产生,系统模型就会被运行.在其运行期间,每个活动的特征在完成其特定任务的同时会产生行为序列.该方法的另一个要求是描述出冲突规则,每条规则由一组谓词公式的合取组成,它刻画了某一类交互可能发生的条件.其中一个公式被称作模式公式,用于描述特征行为中出现的冲突模式;还有其他一些辅助公式,它们用于描述数据关系,以提高交互识别的准确性.在系统模型运行之前,交互描述文件将被载入分析;在系统模型运行期间,特征的行为将被收集和分析,如果特征的行为满足了某条冲突规则,则将产生冲突报告.

本文实验选择对一个 E-mail 系统的模型进行验证.该系统模型的构造参考了文献[1],之所以选择该系统模型进行实验有以下原因:首先,先前已有若干针对此系统的研究,因此可以作为比较;其次,该系统的特征较为丰富,总共有 10 个特征;再次,E-mail 系统被广为使用,因此,这有助于系统建模和对实验结果的理解.不过,我们也对系统模型做了一些修改,以使得模型更简洁或者更贴近实际系统.实验共计产生和分析了超过 100 万的运行场景,实验结果是令人满意的,总共有 20 个特征交互被发现,其中有 3 个从未被报道.相比于以前的类似工作,本文方法有如下特点:

(1) 已有的在线检测只能覆盖有限的运行场景,而本文方法能够系统地覆盖所有的运行场景.

(2) 以前的离线检测一般只对特征进行两两检测,而本文方法可以同时分析的特征并无数量的限制,因此可扩展性有所提高.

(3) 能对数据关系进行分析.当特征交互的分析需要判断某种数据关系时,基于传统的模型检测的方法就难以处理.而我们的方法能够分析数据关系所导致的交互.

(4) 所用建模语言是 Java 而非形式化语言,这有助于减小建模的难度.

基于以上原因,我们认为本文方法与传统的检测方法可以互相补充.

本文第 1 节介绍相关工作.第 2 节介绍方法.第 3 节介绍系统实现.第 4 节是实验.第 5 节总结全文.

1 相关工作

特征交互的检测可以分为在线和离线两大类.Tsang 等人提出了一种基于特征交互管理器(feature interaction manager,简称 FIM)的在线检测方法[11].FIM 有两种工作模式.在学习模式下,特征只能单个运行,此时,

特征的行为将作为“正确”的行为被 FIM 收集存储起来.FIM 使用动作序列或者触发-响应规则来表示特征行为.在管理模式下,多个特征可以任意运行,FIM 监视每个特征的行为,并与“正确”的行为进行对比.一旦发现某种偏离就会报错.该方法与本文方法的相似之处在于,二者都是基于对特征行为的分析来检测交互.不过,本文方法是检测特征行为中是否存在冲突模式,而 Tsang 等人的方法是检测特征行为是否偏离“正确”的模式.我们认为,相比于“正确”的模式,冲突模式更具有共性,相似的冲突模式甚至可以在不同的系统和领域出现,因此,冲突模式可被复用的机会更大.而且,FIM 所提取的“正确”模式不包含数据关系,这使其难以检测那些与数据关系相关的交互.另外,本文检测作用于 Java 模型,而 FIM 的检测作用于实际程序.借助于 JPF 工具,本文方法可以覆盖所有的运行场景,而 Tsang 等人的方法无法实现这一点.

对 E-mail 系统特征交互的研究最先出现于文献[1],Hall 提出一种半自动的离线检测方法.他使用 P-EBF 语言来构造模型,模型在工具的支持下可以被模拟运行,产生输出,然后由人根据输出判断是否存在交互.每次模拟只包含两个特征的实例.Hall 一共检查了 100 对特征,他估算这些特征组合可产生大约 34 560 个运行场景.由人来检查所有的运行场景是不太可能的,因此,他只选择了 155 个场景进行分析,并检测出 26 个特征交互.Calder 等人使用模型检测技术对文献[1]中的特征子集进行了离线分析[12].她们使用 SPIN 工具的 Promela 语言进行建模,并使用线性时序逻辑(LTL)来描述特征的属性.她们开发了 Perl 脚本程序来选择待检测的特征对,对特征进行配置,并将它们结合成一个复合模型,这个复合模型将输入到 SPIN 进行检测.因此,每轮检测只包含两个特征.使用她们的方法,总共检测出 8 个特征交互,它们都是文献[1]中已知的.显然,这两种方法都是对复合模型进行检测,而且,复合模型只包含两个特征.这正是大多数离线检测方法的特点.本文方法也是对复合模型进行检测,但是,本文方法可以同时分析的特征的数量并没有受到限制.在实验中,复杂的运行场景会包含 10 多个特征实例.因此,本文的方法有能力检测 3 个甚至更多特征之间的行为交互.事实上,在本文实验中就检测出一个包含 3 个特征的交互(参见表 2,reverse_transform 规则).另外,虽然 Hall 的方法能够检测出多达 26 个特征交互,但是,检测的关键部分是由人完成的,而本文方法是全自动的.

Li 等人提出一种模块化的模型检测技术[13].每个特征被建模为一个状态机,特征可以通过接口状态被组合起来.特征的属性由计算树逻辑(CTL)表示.要验证特征的组合是否会有交互,Li 等人不是对复合模型进行分析,而是设法从每个特征的属性导出接口状态需满足的保持约束(preservation constraint),然后检测一个特征的保持约束是否能被其他特征满足,如果不满足就意味着存在交互.这样,验证就能简化为单个特征之上的分析.他们将此方法应用到文献[1]中的特征,总共检测出 10 个特征交互,全部都是文献[1]中已知的.作者承认,他们的方法尚不支持对数据关系的分析,这是因为对数据检测会明显增大状态空间.这或许解释了他们为什么没有考虑对数据做复杂操作的特征,如 VerifyMessage.与此方法相比,本文方法是在全局系统层面进行分析.全局验证和模块化验证互有长短,彼此可以补充.另外,本文方法允许用户在冲突规则中描述数据关系,因此支持对数据关系的分析.本文方法能够覆盖文献[1]中的所有特征.

从实验结果来看,除了 Hall 所检测到的 26 个特征交互之外,另两项研究未能发现任何新的特征交互.而本文实验则发现了 3 个新的特征交互,这展示了本文方法的效果.

2 方 法

本节首先提出一个 E-mail 系统特征交互的实例,然后以此为线索,依次介绍特征行为的表示、冲突规则的表示、特征的建模、运行场景的产生以及模式分析.

2.1 一个具体的特征交互实例

这里我们来看 E-mail 系统中的具体实例.假设地址 A 处的用户 u(简记为 $u@A$)使用证书对邮件 m 进行签名(签名由 SignMessage 特征实现),签名的结果记录在 *m.signature* 域.显然,*m.signature* 的值由整个邮件的内容决定.接着,邮件 m 被发送到服务器 S,服务器提供匿名转发服务(由 RemailMessage 特征实现),此服务的功能是提供伪地址和真实地址之间的转换,以达到匿名的效果.假设 *m.receiver* 是伪地址 $x@R$,服务器将查找对应的真实地址 $v@B$,并用它更新 *m.receiver*.然后,m 将被转发到地址 B 处的客户端,由于 *m.signature* 非空,B 处的特征

VerifyMessage 将被激活,它首先获取相应的证书,并以此对 *m.signature* 进行验证.可是,*m.receiver* 已发生改变,因此,*m.signature* 将不再与 *m* 的内容一致,验证失败[1].

2.2 特征行为的表示

每个特征的行为都可以用一组基本动作的序列予以描述.特征的行为表示只包括那些与特征交互有关的动作,而忽略掉那些无关的动作.比如,E-mail 特征是通过对消息的操作来实现交互的,因此,其行为表示只包含那些与消息操作相关的动作.例如,RemailMessage 的行为可以用如下序列表示:

$$\langle rem@S{:}rcv(m);rd(m.rec);md(m.rec);snd(m)\rangle.$$

其中,*rem@S* 表示 *S* 处的特征 RemailMessage.它先接收到消息 *m*,接着读取 *m.receiver* 域,然后修改 *m.receiver* 域,最后将消息 *m* 发送出去.而 VerifyMessage 的行为则可以表示如下

$$\langle ver@B{:}rcv(m);rd(m.sdr);rd(m.sig)\rangle.$$

其中,*ver@B* 表示 *B* 处的特征 VerifyMessage.它先接收到消息 *m*,接着读取 *m.sender* 域,然后读取 *m.signature* 域.

2.3 冲突规则的表示

上述特征交互的原因有两方面:第一,先后发生了 RemailMessage 修改 *m.receiver* 域和 VerifyMessage 读取 *m.signature* 域两个动作;第二,*m.signature* 的值依赖 *m.receiver* 的值.我们可以从这个具体的交互原因中抽象出一般的冲突规则:给定一个消息 *m*,若特征 $F1$ 修改了域 $m.p1$,之后,特征 $F2$ 读取域 $m.p2$,且 $p2$ 的值依赖于 $p1$ 的值,则容易导致交互.注意,$p2$ 对 $p1$ 的依赖是一种数据关系,而且,这种数据关系不是只对消息 *m* 成立,而是对所有的消息都成立.

这条冲突规则如图 1 所示,它被命名为 modify_read_1 conflict,它的条件由两个公式组成.其中,occur 公式是模式公式,impact 公式是辅助公式.occur 公式的作用是描述特征行为中可能发生的冲突模式.冲突模式使用扩展的正则表达式表示,表达式位于引号之内.该扩展正则表达式的最显著特点是使用了模式变量,模式变量的作用是记录模式匹配中成功匹配的子字符串,以供后面引用.该冲突规则包含了 5 个模式变量,其中,$f1$,$f2$ 表示两个特征,*m* 表示消息,$p1$,$p2$ 表示两个域.每个变量的定义都声明了该变量所对应的模式串.给定一个模式变量 *v*,%*v* 表示 *v* 所对应的模式串,$*v* 表示 *v* 的模式串所匹配的字符串.例如,%*f*1 表示模式串(\w*@\w*),$*f*1 则表示%*f*1 所匹配的某个字符串,比如说 *rem@S*.分析时,%*v* 和$*v* 都将被替换为对应的字符串.该冲突模式包含两部分:第一部分是说,特征 $f1$ 的行为包含一个修改消息 *m* 的域 $p1$ 的动作;第二部分是说,特征 $f2$ 的行为包含一个读取消息 *m* 的域 $p2$ 的动作.impact 公式描述了域 $p1$ 和 $p2$ 的依赖关系,即 $p1$ 的值会影响 $p2$ 的值.由于这种依赖关系对所有消息都成立,因此 $p1$ 和 $p2$ 的前缀是 *Msg*,表示消息类.

```
f1,f2:(\w*@\w*)
m:(m\d)
p1,p2:(\w+)

name=modify_read_1 conflict
condition=occur("*⟨%f1*md(%m.%p1)*⟩*⟨%f2*rd($m.%p2)*⟩*") && impact(Msg.$p1,Msg.$p2)
```

Fig.1 The description of a conflict rule
图 1 冲突规则的描述

从这个实例可以看出,任意一个冲突规则都由一组原子公式组成,其中包含一个描述冲突模式的公式以及一些描述数据关系的辅助公式.利用辅助公式,可以更准确地描述交互发生的条件.

2.4 特征建模

前面提到,每个特征都被实现为一个 Java 类,所有的特征都继承自抽象类 Feature.如图 2 所示,每个特征所处的主机节点用 *bearer* 记录.同一个节点上的所有特征通过 *nextDown*,*nextUp* 以管道的形式相互连接起来.这是

因为消息在特征之间流动有两个方向:若消息从用户流向网络,则称作下行(DOWN);反之则称作上行(UP).因此,每个特征用 *nextDown* 记录下行方向下一个特征,用 *nextUp* 记录上行方向下一个特征.Feature 类有两个关键的方法:passMessage 的作用是准备将一个消息传递给下一个特征;processMessage 的作用是处理消息.在 passMessage 中,当前特征根据消息的流向选择下一个特征,如果下一个特征存在,则将消息传给它,否则将消息返回给主机.processMessage 是一种抽象方法,因为具体的处理细节必须由具体的特征来实现.

```
public abstract class Feature {
    ServiceBearer bearer;
    Feature nextUp;
    Feature nextDown;

   protected void passMessage(Message m,boolean direc) {
         logBehavior(m);
         if (direc==DOWN) {
           if (nextDown==null)
              bearer.returnOutgoingMessage(m);
           else nextDown.passMessage(m,DOWN);
        } else {
           if (nextUp==null)
              bearer.returnIncomingMessage(m);
           else nextUp.passMessage(m,UP);
        }
   }

   protected abstract void processMessage(Message m);
```

Fig.2 The code of the Feature class

图 2 Feature 类的代码

图 3 显示出 EncryptMessage 特征是如何继承 Feature 类的.EncryptMessage 实现了 processMessage 方法.该方法首先获取密钥,然后通过密钥 k 和域 $m.mark$ 上的异或操作来模拟加密.加密的结果记录在 $m.mark$ 中.如果 $m.mark$ 为 0,则表示消息 m 未经加密;否则表示 m 经过加密.加密后的消息将通过 passMessage 方法准备传给下一个特征.此外,在完成加密任务的同时,EncryptMessage 将其对消息的操作记录在 *reports* 变量中,在准备传递消息时,*reports* 所记录的内容将通过 logBehavior 方法发送给监视器(如图 2 所示).

```
public class EncryptMessage extends Feature {
   protected void processMessage(Message m) {
      if (key==null) applyKey();
      int k=key.intValue();
      reports=new LinkedList();
      reports.add(TRANSFORM+"("+m.getIdString()+")");
      m.setMark(m.getMark()∧k);
      reports.add(SEND+"("+m.getIdString()+")");
      passMessage(m,DOWN);
   }
}
```

Fig.3 The code of the EncryptMessage feature

图 3 EncryptMessage 特征的代码

2.5 运行场景的产生

系统模型可能在不同的场景下运行.系统的运行场景由系统拓扑(topology)、特征配置和外部输入组成.在

类似 E-mail 的消息处理系统中,系统拓扑就是由系统的节点构成,特征配置则包括每个节点上特征的组合以及每个特征的参数配置,外部输入则是用户产生的具体消息.运行场景的变化会影响特征的激活与否、特征的具体行为以及特征的执行顺序,因此,它可能改变模型的执行路径和模型运行所产生的特征轨迹集合.即使对于一个不太复杂的系统而言,其所有的运行场景的数量也可能是非常巨大的.这是因为整个系统的运行场景的数量等于所有节点的特征配置数量的乘积与外部输入的数量的乘积.假设系统包含 1 台服务器,u 台客户机,每台客户机部署着 n 个特征$\{F_1,\ldots,F_n\}$,服务器部署着 m 个特征$\{F_{n+1},\ldots,F_{n+m}\}$,设特征 $F_i(1\le i\le n+m)$有 k_i 种参数配置,且外部可能产生 t 种输入,则整个系统的运行场景的数量为

$$NS=(k_1\times\ldots\times k_n)^u\times(k_{n+1}\times\ldots\times k_{n+m})\times t \tag{1}$$

其中,第 1 个括号内表示每台客户机上特征的配置数量,由于共有 u 个客户机,所以该数量需要相乘 u 次;第 2 个括号内表示服务器上特征的配置数量;最后是外部输入的数量.由式(1)可知,运行场景的数量与客户机的数量呈指数关系.因此,限制运行场景数量的关键是控制系统中客户机的数量.一旦我们根据具体问题确定了客户机的最小数量,就能据此确定系统的拓扑.在系统模型初始化时,可以通过创建所需的节点实例来建立系统拓扑.

确定系统拓扑之后,还需要确定每个节点上特征的组合以及外部输入.由于这两方面都源于用户动作,因此,可以创建一个 User 类来模拟用户对其客户端特征的配置和待发送邮件的配置.无论是对特征还是对邮件的配置都可以通过对相关变量取值范围的遍历来实现.为此,我们使用了外部工具 JPF.JPF 提供了 Verify.randomBool()和 Verify.random(n)两种方法,前者随机产生 true 或 false,后者随机产生 0~n 的一个整数值.配合状态空间搜索,JPF 可以遍历相关变量的所有取值[14].例如,假设场景产生器包含如下一条语句:x=Verify.random(n),在 JPF 的控制下,x 会被随机分配一个值 k;当 JPF 回溯以产生新的状态时,x 又会被随机分配另一个不同的值 k'.这样,直到 0~n 中所有可能的取值都被遍历为止.另一种重要的方法是 Verify.ignoreIf(*cond*),它的作用是剪除掉满足表达式 *cond* 的状态[14].

作为例子,图 4 显示了 User 类是如何配置客户端特征和邮件的.其中,特征配置位于 configureFeatures 方法,邮件配置位于 writeMail 方法.特征配置包括两方面——激活或关闭特征,为激活的特征设置必要的参数.

```
public class User {
  public void configureFeatures() {
     if (Verify.randomBool()) {
       filtMsg.enable();
       Address item=pickAddress();
       Verify.ignoreIf(item.equals(Remailer_Addr);
       filtMsg.addFilterItem(item);
     }
    else filtMsg.disable();
    …
  }

  public void writeMail() {
    Message msg=Message.create();
     if (Verify.randomBool()) {
       receiver=Address.createAliasLabel();
     } else {
       receiver=pickAddress();
     }
     msg.setReceiver(receiver);
     …
     Agent.messageGenerated(msg);
  }
```

Fig.4 The code of the User class to configure features and messages

图 4 User 类配置特征和邮件的代码

图 4 的代码显示了对 FilterMessage 特征的配置,当 Verify.randomBool()返回 true 时,则该特征被激活;否则,该特征被关闭.该特征包含一个地址参数,以指名来自哪个地址的邮件需要被过滤.当该特征被激活时,就会通过 pickAddress 方法随机选择一个地址作为参数,而且,所选择的地址不能是 Remailer 地址.如果当前选择的参数恰好是 Remailer 地址,则此选择无效,JPF 会回溯,以便选择下一个尚未访问过的地址.为了随机选择地址,所有已创建的地址都被保存在 Address 类的一个列表中,通过 Verify.random 就可以从这个列表中随机挑选出一个地址来.相关的代码位于 pickAddress 方法内.

对邮件的配置主要是选择邮件的接收者.由图 4 可以看出,writeMail 方法首先创建新的消息,然后设置该消息的 receiver 域.当 Verify.randomBool 返回 true 时,receiver 是一个别名(细节参见第 3.1 节),否则通过 pickAddress 方法随机产生一个地址作为 receiver.配置好的消息将传给该 User 实例所对应的客户端,这由 Agent.messageGenerated 方法实现.一旦新的消息发送给客户端,则模型开始运行.

2.6 模式分析

当描述冲突规则的文件载入时,该文件经过分析将产生规则的内部表示.每个规则对应一个 Rule 实例,而每个 Rule 实例都作为监听器(listener)注册到一个监视器(monitor)上.模型运行期间,每个活动的特征都会向监视器汇报其行为.监视器将所有特征的行为收集起来,进行必要的格式处理,然后再将多个特征的行为组合为该场景下的行为轨迹(behavior trace).当一个新的行为轨迹产生后,监视器会通知所有监听的 Rule 实例.当一个 Rule 实例接收到一个输入的行为轨迹后,它首先作模式匹配.如果匹配成功,模式变量所匹配的字符串将被保存起来.例如,设有模式变量 *v*,如果%*v* 的表达式匹配成功,则%*v* 所匹配的字符串将存储起来,在后续分析中,该字符串将用于替换$*v*.如果规则还有辅助公式,则需要在模式匹配成功后分析辅助公式是否成立.当辅助公式也满足时,则产生一个冲突报告.

辅助公式的判定较为简单.辅助公式的作用是表示数据关系,例如,impact(*Msg*.$*p*1,*Msg*.$*p*2)表示了消息的 *p*1 域和 *p*2 域的依赖关系.在系统初始化时,消息的不同数据域的依赖关系已作为必要的知识被提供给系统.如图 5 所示,Rule 类在初始化时,就已将 *Msg*.*sender* 和 *Msg*.*signature*,*Msg*.*receiver* 和 *Msg*.*signature*,*Msg*.*destination* 和 *Msg*.*signature* 之间的依赖关系存入一个 HashMap 结构.当要判断 *p*1 域和 *p*2 域的依赖关系时,可调用 Rule 类的 impact 方法,该方法通过查找该 HashMap 结构来判定二者是否存在依赖关系.

```
public class Rule implements MonitorListener {
  static HashMap impacts;
  public static void initializeKnowledge() {
      impacts=new HashMap();
      impacts.put("Msg."+Message.SENDER,"Msg."+Message.SIGNATURE);
      impacts.put("Msg."+Message.RECEIVER,"Msg."+Message.SIGNATURE);
      impacts.put("Msg."+Message.DESTINATION,"Msg."+Message.SIGNATURE);
  }

  public static boolean impact(String item1,String item2) {
      String val=(String) impacts.get(item1);
      if (val.equals(item2)) return true;
      else return false;
  }
```

Fig.5 The initialization and evaluation of data dependencies

图 5 数据依赖关系的初始化和判定

图 6 显示了对一个运行场景下的行为轨迹分析的结果.该场景包含 3 个特征的行为,分别是特征 SignMessage,RemailMessage 和 VerifyMessage.消息在客户端 *A* 处经过 SignMessage 处理后,通过 *ch_A* 信道传到服务器 *S* 上,接着由 RemailMessage 处理,然后,消息又通过 *ch_A* 信道返回到客户端 *A* 处,被 VerifyMessage 处

理.结果,在 *A* 处的验证失败.这些特征的行为已在第 2.2 节中介绍过,验证失败的原因在于后两个特征存在着交互,该交互已在第 2.3 节解释过.

{⟨*sgn*@*A*:*add*(*m*1.*sig*);*snd*(*m*1)⟩
|*ch_A*|
⟨*rem*@*S*:*rcv*(*m*1);*rd*(*m*1.*rec*);*md*(*m*1.*rec*);*snd*(*m*1)⟩
|*ch_A*|
⟨*ver*@*A*:*rcv*(*m*1);*rd*(*m*1.*sdr*);*rd*(*m*1.*sig*)⟩
*m*1(*u*@*A*,*x*@*R*):*verif* failed}

modify_read_1 conflict: *rem*@*S* *ver*@*A*

Fig.6 The analysis result of a behavior trace
图 6 行为轨迹的分析结果

3 实 现

我们将此方法应用到了一个 E-mail 系统的特征交互分析.该系统的模型基于文献[1].

3.1 E-mail系统模型

该 E-mail 系统由一个服务器和 3 个客户端组成,服务器和每个客户端通过信道相连(如图 7 所示).在本文的模型中,信道用于模拟服务器和客户端之间的远程消息通信.当客户端(服务器)发送一条消息时,该消息将通过方法调用立即传给对应的信道,该信道然后通过方法调用将消息传给服务器(客户端).这种以方法调用来简化异步通信的做法是为了提高验证效率,并且不会对验证结果造成影响,这是因为 E-mail 系统的交互源于特征对消息的操作,而与通信的机制无关.

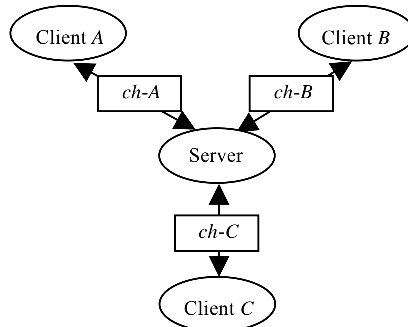

Fig.7 The structure of the E-mail model
图 7 E-mail 模型的结构

该系统包括 10 个特征,它们分布在客户端或服务器端.为了表示上的简洁,我们为每个特征赋予了一个短名.这些特征分别是:

- AddressBook(adr):该特征允许用户为一组接收者地址定义一个别名.如果消息的 receiver 域是一个别名,则该特征将产生多份消息的拷贝,并将每份拷贝的 receiver 域换为一个具体的接收者地址.
- AutoResponder(aut):当该特征被激活时,将自动为每个接收到的消息产生回复.为了避免冗余,它只为每个发信者产生一份回复.
- EncryptMessage(enc):当该特征被激活时,会对将要发送的消息加密.在本文模型中,密钥由一个 Authority 类来管理.加密被模拟为密钥和消息的 mark 域的异或操作.
- DecryptMessage(dec):如果接收到的消息经过加密,该特征则对消息解密.解密所需的密钥来自

Authority 类,假定总能获得正确的密钥.解密被模拟为密钥和消息的 mark 域的异或操作.

- SignMessage(sgn):当该特征被激活时,会对将要发送的消息签名.在本文模型中,签名的证书也由 Authority 类来管理.签名被模拟为证书和消息 signature 域的异或操作.
- VerifyMessage(ver):如果接收到的消息经过签名,该特征则对消息验证.验证所用的证书来自 Authority 类,假定总能获得正确的证书.验证被模拟为证书和消息 signature 域的异或操作.
- ForwardMessage(fwd):如果被激活且被设置目标地址,则该特征将把接收到的消息转发到目标地址.
- FilterMessage(flt):如果被激活且被提供了目标地址,则该特征将检查接收到的消息的 sender 域,如果 sender 域与目标地址相同,则将消息过滤掉.其他消息则被传送下去.
- RemailMessage(rem):该特征用于实现匿名转发.为此,它存储着用户真实地址到伪地址的映射.该特征有两种工作模式.在第 1 个模式下,输入消息的 receiver 域是 *r@R*(代表 remailer 的特殊地址),而真正接收者地址置于 destination 域,此时,该特征将查询 sender 所对应的伪地址,以此修改 sender 域,并用 destination 域替换 receiver 域,之后发送修改过的消息.在第 2 个模式下,输入消息的 receiver 域是某个用户的伪地址,此时,该特征将查询对应的真实地址,并用它修改 receiver 域,然后发送修改过的消息.
- MailHost(mho):该特征有一组用户列表,当它接收到一个消息时,如果消息的 receiver 在列表中,则输入消息将发送给对应的用户,否则就产生一条 postmaster 回信,将其发送给原消息发送者,告诉他目标地址有误.

上述特征可以分为两类,前 8 个属于客户端特征,后 2 个属于服务器端特征.客户端特征按照管道方式被组合成一个特征链.特征的组合顺序可以有多种,而现实中,特征的组合顺序是由开发者固定好的.因此,本文模型不必考虑所有的特征组合顺序,而只需考虑一个典型的组合顺序,如图 8 所示.该顺序类似于文献[1]中的模型,区别是我们在 DecryptMessage 前加了 FilterMessage.当用户消息传给客户端时,该消息将依序传递给 AddressBook,SignMessage 和 EncryptMessage 特征,最后,消息将被发送出去.当客户端从信道接收到一条消息时,该消息将依序传递给 FilterMessage,DecryptMessage,VerifyMessage,AutoResponder 和 ForwardMessage 特征.如果 AutoResponder 被激活,它将产生一条回复,并将该消息传给 SignMessage.若 ForwardMessage 被激活,接收到的消息将被传给 SignMessage,否则该消息将传给用户.服务器端特征也被连接起来,如图 8 所示.当服务器接收到一条消息,它将消息依次传给 RemailMessage 和 MailHost 特征.最后,选择一个合适的信道将该消息发送出去.

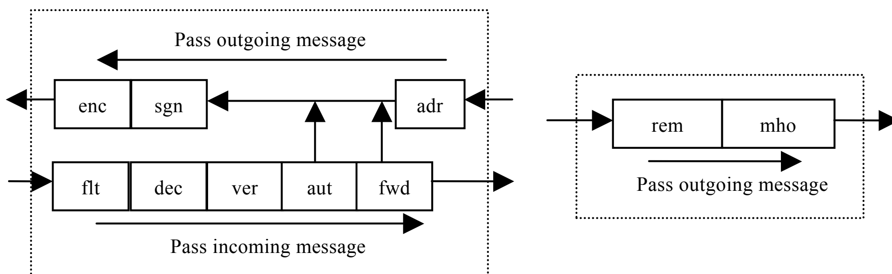

Fig.8 The topology of the client-side (left) and the server-side (right) features

图 8 客户端(左)和服务器端(右)特征的组合顺序

3.2 验证系统结构

整个验证系统的体系结构如图 9 所示.JPF 控制着场景产生器,以产生所有的运行场景.每当一个运行场景产生时,就会有一个新的消息传递给 E-mail 的 Java 模型,此时,该模型将开始运行.在运行时,每个特征在完成特定任务的同时,还会将其行为报告给监视器.监视器收集特征的行为,将其进行必要的格式处理,然后组合起来

形成一个场景下的行为轨迹.然后,监视器将通知所有注册的规则实例.规则实例是规则描述的内部表示,当规则描述文件被载入后,经过分析(parsing)就会产生一组规则实例,这些规则实例会作为监听者注册在监视器上.每个规则实例对行为轨迹进行模式分析,如果模式匹配且辅助公式满足,则产生冲突报告.

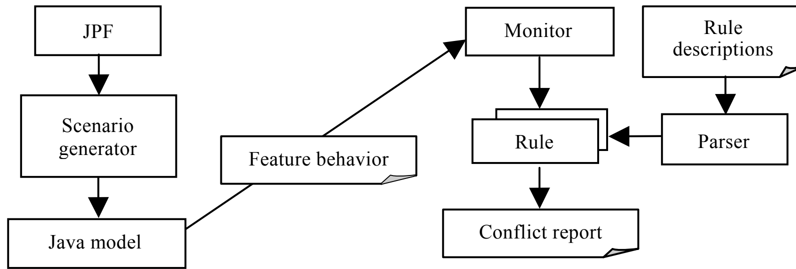

Fig.9 The architecture of the verification system

图 9 验证系统的体系结构

4 实验结果

本文实验在一台 2.8GHz,512M 内存的 PC 机上完成.软件平台是 Windows 2000 加 Cygwin.由表 1 可见,实验共花去 71 个小时,共产生 1 327 104 个运行场景.除掉冗余之后,系统产生了 30 个冲突报告,其中 18 个是真冲突,其余 12 个是假冲突.

Table 1 The statistics of the result

表 **1** 实验结果统计

Time cost	Total scenarios	Num. of reports	True confl.	False confl.
71h.	1 327 104	30	18	12

根据文献[1],我们共拟定了 11 条规则.这些规则及其所检测出的真冲突在表 2 中列出.注意,由于空间限制,这里使用特征的短名(参见第 3.1 节).

Table 2 The conflict rules and the detection results

表 **2** 冲突规则以及检测出的冲突

Rules	Conflicting features	Num. of confl.
copy_modify_1		0
copy_modify_2		0
modify_read_1	rem-ver	1
modify_read_2	rem-flt; fwd-flt; rem-ver; fwd-ver	4
reverse_transform	enc-dec-fwd	1
transform_read	enc-rem	1
add_modify	sgn-rem	1
loop	fwd-mho; fwd-rem; fwd-fwd; aut-fwd	4
alternative_path	rem-fwd; rem-aut	2
discard_1	rem-flt; fwd-flt	2
discard_2	mho-flt; aut-flt	2

这里对文献[1]的结果与本文结果做一个对比分析.文献[1]中共检测出 26 个交互,而本文检测出 18 个交互,原因在于本文的模型与文献[1]的模型有些不同.例如,本文模型假定 DecryptMessage 总能获得正确的密钥,VerifyMessage 也总能获得正确的证书.我们认为,由于密钥或证书不正确所导致的解密或验证失败不应看作特征交互.而这些在文献[1]中都被归为特征交互.因此,文献[1]中的 SignMessage 和 VerifyMessage, EncryptMessage 和 DecryptMessage,EncryptMessage 和 VerifyMessage,EncryptMessage 和 AutoResponder, DecryptMessage 和 AutoResponder 之间的交互在本文模型中都不存在.总共算下来,模型的差异导致文献[1]的 10 个交互在本文模型中不会出现.此外,本文实验还检测出文献[1]中没有报道的 3 个交互,这 3 个交互按照文献[1]的模型是应该存在的.交互如下:

- ForwardMessage vs. FilterMessage(modify_read_2 rule):假设地址 *u@A* 被 *C* 处的 FilterMessage 列为过滤目标,可是如果来自 *u@A* 的消息先到达 *v@B* 处,且被 *B* 处的 ForwardMessage 转发,则消息的 sender 域会变为 *v@B*,从而无法被过滤.
- RemailMessage vs. ForwardMessage(alternative_path rule):假设用户 *v@B* 使用了匿名转发服务,其伪地址为 *y@R*.用户 *u@A* 向 *y@R* 发送消息,该消息被 RemailMessage 转发到 *v@B* 处,如果 *B* 处的 ForwardMessage 恰好把该消息转发给 *u@A*,就会暴露 *y@R* 的真实地址.
- RemailMessage vs. FilterMessage(discard_1 rule):假设用户 *u@A* 使用了匿名转发服务,其伪地址为 *x@R*.当用户 *u@A* 发送的消息经过 RemailMessage,消息的 sender 域将变为 *x@R*.假设消息到达 *B* 处,如果 *B* 处的 FilterMessage 将 *x@R* 列为过滤目标,则所有经过匿名处理的消息都将悄然无息地被过滤掉.

实验结果显示出本文方法是有效的.该方法的效果来源于冲突模式与系统化搜索的结合.一方面,它说明从已知的特征交互中抽取的冲突模式确实可用于检测新的特征交互;另一方面,它说明系统化地搜索确实是必要的.相比之下,文献[1]中的检测方法是基于人的经验,虽然这种方法能够提供对交互的准确判断,但是它的缺陷在于人的分析只能覆盖很少的运行场景.因此,该检测方法有所疏漏也就不足为奇了.

不过,实验结果也表明本文方法存在以下几个问题:首先,尽管使用了辅助公式增强交互描述的准确性,冲突报告中仍然存在着较多的假冲突,这主要是由于非预期匹配.给定一个冲突规则,若一个特征轨迹可以多次匹配该规则的模式公式,而在实际分析时,所用的模式匹配工具总是给出第 1 次匹配.如果第 1 次匹配是不符合预期的,那么就可能产生假冲突;其次,分析时间长达 71 小时,这是为了分析大量运行场景所付出的代价.因此,需要进一步考虑简化措施,或选择效率更高的模型检测工具.

5 结 论

本文提出了一种基于模式分析检测特征交互的方法.其基本思想是,特征的行为可以表示为一个基本动作序列,而当特征交互发生时,往往可以从特征行为中提取出反复出现的模式,这些模式可以用于检测新的特征交互.本文使用一组谓词公式来描述交互发生的条件,使用 Java 语言描述系统的模型,并借助 JPF 工具遍历所有的运行场景.模型运行时,特征的行为将被收集起来进行分析.我们将此方法应用于一个 E-mail 系统的分析.实验结果令人满意,该系统能够处理超过 100 万个运行场景,并有效检测出特征交互,其中包括以前从未报道的一些交互.未来,我们希望从以下几个方面改进本文方法:首先,将特征属性和冲突模式结合起来,以提高分析的准确性;其次,寻找解决非预期匹配的技术方案;第三,应用某种简化技术以减少需要分析的运行场景,或选择效率更高的模型检测工具,以提高分析的速度;第四,本文的实现仅考虑了数据的直接依赖关系,这对于 E-mail 系统而言已经足够,但是在其他系统中,也许还存在着传递依赖关系.因此,未来有必要增加对传递依赖关系的支持.

References:

[1] Hall RJ. Feature interactions in electronic mail. In: Calder M, Magill E, eds. Feature Interactions in Telecommunications and Software Systems VI. Amsterdam: IOS Press, 2000. 67−82.

[2] Cameron EJ, Griffeth ND, Lin YJ, Nilson ME, Schnure WK, Velthuijsen H. A feature interaction benchmark for IN and beyond. In: Bouma LG, Velthuijsen H, eds. Feature Interactions in Telecommunications Systems II. Amsterdam: IOS Press, 1994. 1−23.

[3] PBX vulnerability analysis. 2000. http://csrc.nist.gov/publications/nistpubs

[4] Lennox J, Schulzrinne H. Feature interaction in internet telephony. In: Calder M, Magill E, eds. Feature Interactions in Telecommunications and Software Systems VI. Amsterdam: IOS Press, 2000. 38−50.

[5] Nakamura M, Leelaprute P, Mtsumoto K, Kikuno T. On detecting feature interactions in the programmable service environment of internet telephony. Computer Networks, 2004,45(5):605−624.

[6] Rinard M, Sălcianu A, Bugrara S. A classification system and analysis for aspect-oriented programs. In: Taylor RN, Dwyer MB, eds. Proc. of the 12th ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE). New York: ACM Press, 2004. 147−158.

[7] Zhang C, Jacobsen HA. Resolving feature convolution in middleware systems. In: Vlissides J, Schmidt D, eds. Proc. of the 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM Press, 2004. 188−205.

[8] Bruns G. Foundations for features. In: Reiff-Marganiec S, Ryan MD, eds. Feature Interactions in Telecommunications and Software Systems VIII. Amsterdam: IOS Press, 2005. 3−11.

[9] Calder M, Kolberg M, Magill EH, Reiff-Marganiec S. Feature interaction: A critical review and considered forecast. Computer Networks, 2003,41(1):115−141.

[10] Visser W, Havelund K, Brat G, Park S, Lerda F. Model checking programs. Automated Software Engineering Journal, 2003,10(2):203−232.

[11] Tsang S, Magill EH. Learning to detect and avoid run-time feature interactions in intelligent networks. IEEE Trans. on Software Engineering, 1998,24(10):818−830.

[12] Calder M, Miller A. Generalising feature interactions in email. In: Amyot D, Logrippo L, eds. Feature Interactions in Telecommunications and Software Systems VII. Amsterdam: IOS Press, 2003. 187−204.

[13] Li H, Krishnamurthi S, Fisler K. Verifying cross-cutting features as open systems. In: Soffa ML, Griswold W, eds. Proc. of the 10th ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE). New York: ACM Press, 2002. 89−98.

[14] Java pathfinder. 2005. http://javapathfinder.sourceforge.net

左继红(1974－),男,四川成都人,博士生,主要研究领域为特征交互,软件工程.

梅宏(1963－),男,博士,教授,博士生导师,CCF高级会员,主要研究领域为软件体系结构,软件工程.

王千祥(1970－),男,博士,副教授,CCF 高级会员,主要研究领域为软件工程,软件中间件,软件自适应技术.