文章编号：0253-374X(2018)07-0972-10 DOI:10.11908/j.issn.0253-374x.2018.07.016

一种基于异常控制流的错误程序行为分析方法

江建慧，吴捷程，孙 亚
(同济大学 软件学院，上海 201804)

摘要：通过静态分析程序显式异常控制流收集到程序中可引起异常的差错信息，采用故障注入实验，分析了程序的“故障-差错-异常”传播过程.结合函数级异常控制流的描述，对异常相关的差错及其对程序行为的影响进行了分析，建立了基于异常控制流的错误程序行为模型，开发了相应的分析工具.以OpenStack核心组件为对象进行实验，结果表明从异常层次对错误程序行为进行分析是合理而有效的.该方法为具有异常处理机制的大规模程序的错误行为自动分析和差错数据的收集提供了新手段.

关键词：错误程序行为；软件差错；异常控制流；故障注入
中图分类号：TP311 **文献标志码**：A

An Approach to Analyzing Erroneous Program Behavior Based on Exception Control Flow

JIANG Jianhui, *WU Jiecheng*, *SUN Ya*
(School of Software Engineering, Tongji University, Shanghai 201804, China)

Abstract: In this paper, the propagation process of “fault-error-exception” chain in programs is analyzed by fault injection experiments. With representation of the exception control flow at function level, the error and its impact on program behavior are analyzed, a model of erroneous program behavior is established. An automatic analysis tool based on the proposed approach is developed and is used to analyze the erroneous behaviors of the significant components in OpenStack. The experimental results validate the validity and rationality of the proposed approach, which provides a new means to automatically analyze the erroneous behavior and collect the valid error set for large-scale programs with exception handling mechanism.

Key words: erroneous program behavior; software error; exception control flow; fault injection

错误程序行为(erroneous program behavior, EPB)分析主要是获取程序差错及其对程序行为的影响，包括差错类型、差错发生位置、差错发生时机，以及各个差错在程序中的表象.

软件故障注入技术通过人为地对软件或计算机等目标系统注入软件故障来加速其失效，从而实现目标系统的容错机制有效性验证、可靠性评估[1-2].然而，如何得到真实的软件故障或差错数据是该技术实现的前提[1-4].

具有代表性的软件差错数据获取需要分析软件的功能特性及其在某个故障/缺陷被激活后所表现出的错误行为[2-3,5]，工作量很大，且分析周期很长[1,3].常见的正交缺陷分类(orthogonal defect classification, ODC)所属故障被激活后所产生的程序差错形式随程序实现的不同而不同[1]，可能以返回码或共享变量的形式存在于应用程序与其使用的库函数之间[2,6]，也可能以异常的形式在程序中进行传播[5-6]，或以跳转指令的组合形式出现[7].

在具有异常处理机制的程序中，故障激活后所产生的差错若被异常处理机制侦测，则异常发生[8-9].若对该异常处理不当，则可引起程序失效[9].这一过程将形成“故障-差错-异常-失效”链.

通过对程序异常控制流(exception control flow, ECF)的分析可收集能够引起异常的差错信息，包括：由异常类型获取差错内容；由异常引发点获取差错被侦测时的物理位置；由异常传播路径获取差错的影响范围；由异常捕捉情况获取差错是否得到修正的信息.这些信息可指导软件故障注入方案生成所需的差错类型、注入位置、注入时机的选择[4-5,10-11].该过程通过源代码的静态分析，可在相对较短的时间内完成.

收稿日期：2017-07-17
基金项目：国家自然科学基金(61432017, 61772199)
第一作者：江建慧(1964—)，男，工学博士，教授，博士生导师，主要研究方向为可信系统与网络，软件可靠性工程，VLSI/SoC测试与容错.E-mail：jhjiang@tongji.edu.cn
通信作者：吴捷程(1993—)，男，博士生，主要研究方向为软件可靠性工程.E-mail：1510794@tongji.edu.cn

本文的主要工作是通过故障注入实验分析程序的“故障-差错-异常”的传播机理；结合函数级 ECF (function-level ECF，FECF)的描述，分析了同一差错与多个程序 ECF 之间的关系，利用 ECF 对程序中潜在可引起异常的差错进行分析，建立了 EPB 模型；实现了一个自动分析工具软件，并以 OpenStack 的核心组件为对象进行了实验验证.

1　相关工作

Christmansson 等[3]认为只有注入具有代表性的差错才能有效地模拟真实故障的发生，有效地验证软件的容错机制. 考虑应用程序和函数库之间的潜在错误的故障注入工具 LFI(库文件故障注入器)改善了应用程序的健壮性测试技术[2]. 基于穷尽异常模拟模式的测试方法结合异常发生的时序特点在异常引发点注入异常，简化了异常处理结构有效性验证的过程[5]. 上述工作强调了故障/差错数据的有效性，讨论了如何利用差错数据进行软件容错机制的验证，但其尚未涉及如何有效地收集合理的差错数据.

关于软件故障激活后的传播过程及证明故障激活后能否转化为差错的研究，有基于“故障-差错-失效”的传播关系选择具有代表性的差错集合的方法[5]. 通过代码内部的故障注入和接口差错注入的实验研究结果表明，两者对程序的影响存在一定的差异[4]. 代码故障激活后的差错不仅能以返回值、错误码、传入/传出参数的方式进行传播，还能以共享内存数据为载体对程序造成影响[11]. 应从多维度对故障注入后的 EPB 进行收集、分析、归类，从而使得各层次的 EPB 研究可得到相关数据的支撑，实验表明，故障激活后所产生的差错能以返回值或异常形式进行传播，且该差错被侦测的情况和是否引起程序失效取决于程序的实现[6]. 这些工作均未深入研究具有异常处理机制的程序中的“故障-差错-异常”过程.

大多数与程序 ECF 相关的研究工作主要聚焦在它对程序分析、开发、测试的影响上. 如基于异常传播分析的C++程序数据流分析方法[12]、依赖性分析方法[13]. Harrold 等[14-15]在对异常处理机制的测试标准进行研究时，考虑了异常处理机制和异常传播对程序控制流、数据流、控制依赖性等分析技术的影响. 用于程序开发、维护、测试的系统化方法通过分析显式 ECF，总结了与异常处理结构实现相关的编程错误模式，可指导开发者如何有效地避免异常处理机制的错误使用[16]. 面向切面机制在一些情景下会使异常处理结构的实现复杂化，从而降低系统的可靠性[17]. Jo 等[18]提出了一种基于集合分析的函数级别上的 Java 未捕捉异常分析方法. 结合运行时异常的静态测试方法通过交替执行缺陷检测及控制流扩展，提高了测试充分度，还分析了运行时异常对软件静态测试的影响[19]. 这些与程序 ECF 相关的研究工作并未考虑到异常本身就是程序错误的一种表象，所以没有利用异常相关的信息来分析潜在的 EPB.

2　程序“故障-差错-异常”机理分析

2.1　ODC 故障注入实验分析

异常处理机制(exception handling mechanism，EHM)是程序错误处理的常用手段之一. 当程序中 EHM 侦测到差错且引发异常之后，程序的“故障-差错-异常/失效”过程如图 1 所示.

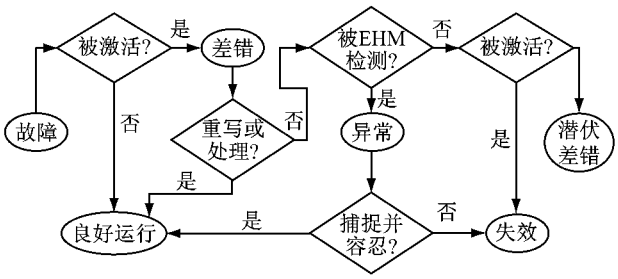


图 1　采用 EHM 的“故障-差错-异常/失效”传播过程

Fig.1　"fault-error-exception/failure" process with EHM

对 Android 上约 800 个应用组件所进行的 600 万次基于外界输入的健壮性测试实验数据表明，约 10%的应用组件发生了崩溃且均由未捕捉到的异常所造成[6].

本文以分布式网络基准程序 Ipbench 和 OpenStack 中的核心组件 nova 为对象，结合 ODC 故障类型及其分布数据，以代码变异的方式实施故障注入实验，实验结果见表 1(表中的 E 表示可引起异常的故障数量).

在 Ipbench 中共注入 348 个故障，其中 255 个故障导致了程序崩溃，且 249 次崩溃是由未捕捉异常引起.

在 nova 中共注入 1 018 个故障，未出现程序崩溃现象. 由表 1 可知，419 个故障激活后的差错得到程序的修正，可能的原因有：① 有差错的数据在被使用之前重新得到初始化；② 有差错程序的执行结

表1 故障注入实验结果

Tab.1 Results of fault-injection experiments

ODC类型	ipbench 正常输出 总数	E	错误输出 总数	E	程序挂起 总数	E	程序崩溃 总数	E
赋值	17	0	3	0	1	0	68	68
条件检查	6	0	3	0	2	0	65	65
算法	26	3	5	0	18	0	85	82
接口	1	0	0	0	2	0	11	11
功能	5	0	1	0	3	0	26	23
总数	55	3	12	0	26	0	255	249
引发率/%	5.5		0		0		97.6	

ODC类型	nova 正常输出 总数	E	错误输出 总数	E	程序挂起 总数	E	程序崩溃 总数	E
赋值	122	0	92	92	2	2	0	0
条件检查	125	0	126	126	0	0	0	0
算法	153	2	236	236	13	11	0	0
接口	19	0	60	60	7	7	0	0
功能	0	0	63	63	0	0	0	0
总数	419	2	577	577	22	20	0	0
引发率/%	0.5		100		91.0		0	

果与无差错程序的执行结果一致;③ 差错引起异常且被捕获,异常处理例程提供了正确的服务.

2.2 传播机理分析

以nova的_validate_cell函数为例,程序的"故障-差错-异常"传播过程分析如下:

(1) 故障激活所发生的差错直接引发异常. 如图2所示,故障激活后的差错直接在当前函数内使raise语句得到执行,从而引发异常. 同时,图2中场景a)与b)的故障激活后均引起了同一异常,说明程序中故障与异常存在多对一关系,即多个故障激活后的差错最终均以同一异常在程序中进行传播.

(2) 故障激活所发生的差错经过传播后引发异常. 图3所示的故障激活后的差错通过数据流影响控制流,最终使得raise语句得到执行,引发异常.

(3) 在程序无故障的情况下,外界输入或外界资源违反软件规约而导致异常发生.

因此,可从"异常"层次出发,分析程序中可引起异常的差错信息及其对程序行为的影响,达到分析与异常相关的EPB的目的.

图2 差错直接引起异常例子

Fig.2 Example of an error raises exception directly

3 基于ECF的EPB分析

为了获取可引起异常的差错的信息,本文方法并不需要构建带有程序ECF的完整过程间控制流[20],而只需获取独立的ECF信息,包括:ECF起始与中止位置信息;ECF对应的异常类型;异常是否被捕捉;ECF所影响到的函数;ECF所影响到的程序功能或服务. 这是一种基于源代码静态分析的方法,如何选择起始函数作为分析的起始点,则直接影响到了分析结果的有效性.

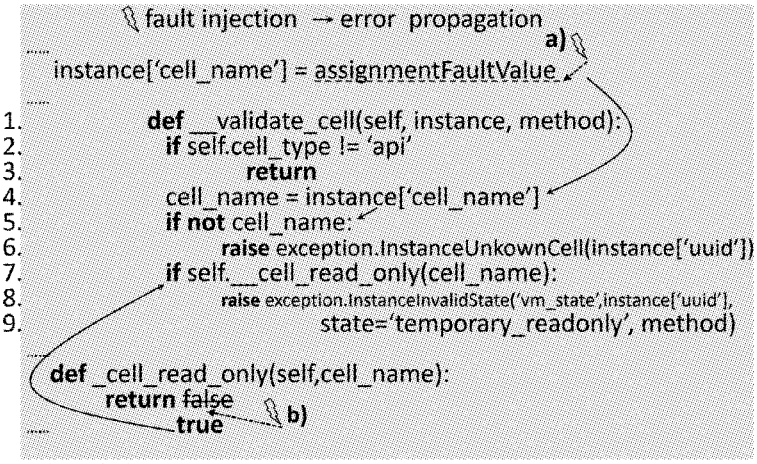
3.1 函数级ECF的描述

图4所示的是函数级异常传播过程,其中n_k($1\leqslant k\leqslant m, m\geqslant 1$)表示函数,其他符号的意思在图中已经加以说明. 以函数为粒度的异常传播过程可分为A,B和C三类.

(1) 过程A:异常在当前函数n_m被引发,而且被捕捉.

(2) 过程B:异常在当前函数n_m被引发,且沿着函数调用栈经过$m-k$逆向传播后在函数n_k中被捕捉,$1\leqslant k\leqslant m$.

(3) 过程C:异常在当前函数n_m被引发,且沿着函数调用栈经过$m-1$次逆向传播,变成未捕捉



a) 赋值故障: instance['cell_name']被赋予错误值，并且使第五行条件为“真”

b) 接口故障: __cell_read_only的返回值从“错”误写为“真”，并且使第七行条件为“真”

图 3　差错经过传播而引起异常的例子

Fig.3　Example of an error raises exception with certain propagation

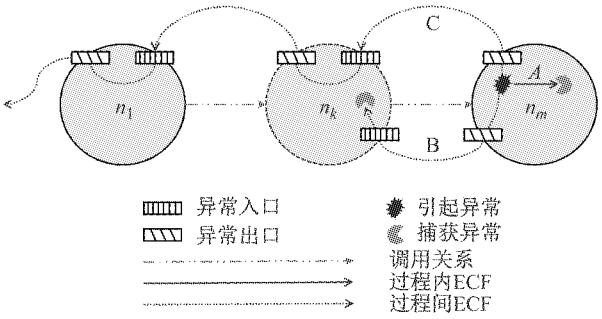


图 4　函数级的异常传播过程

Fig.4　Exception propagation process at function level

异常.

定义 1　程序 P 中由 raise/throw 语句显式引起的 FECF 可表示为 5 元组 $\Psi=<o,t,z,\pi,e>$，其中：

(1) $o_l=(n,\ l_o)$为 Ψ 的起始位置，表示 Ψ 由函数 n_o 中代码行号为 l_o 的 raise/throw 语句所引发.

(2) t 为 Ψ 所对应的异常类型.

(3) $z=(n_z,\ l_z)$为 Ψ 的终止位置. 若 Ψ 属于过程 A 或 B，则 n_z 为 Ψ 被捕捉时所在的函数，其对应的异常捕捉语句的行号为 l_z；若 Ψ 属于过程 C，则 Ψ 在程序 P 的函数 n_z 处终止且传播出去，l_z 为 Ψ 进入函数 n_z 时的异常入口所对应的语句的代码行号.

(4) π 以路径的形式给出了 Ψ 的传播过程. 若 Ψ 属于过程 A 或 B，则 π 为 $n_o\rightarrow n_m\rightarrow\cdots\rightarrow n_1\rightarrow n_z$，其中 $n_o,n_z,n_k(k=1,2,\cdots,m,m\geqslant 1)$为 Ψ 所影响到的函数；若 Ψ 属于过程 C，则 π 为 $n_o\rightarrow n_m\rightarrow\cdots\rightarrow n_1\rightarrow n_z\rightarrow n_{\text{out}}$，其中 n_{out} 为占位符，表示该路径对应的 ECF 未被捕捉.

(5) e 为 Ψ 被引发时程序 P 的栈底函数，表示被 Ψ 影响到的程序功能或服务.

与文献[4，20]一致，本文也将入口函数(entry function，EF)作为函数调用链的分析起点，计算程序中所有潜在的 FECF.

3.2　利用程序 ECF 表示 EPB

同一差错可引起不同的 FECF. 这些 FECF 组成的集合称为 FECF 簇，对于其中的任意 Ψ_i 与 Ψ_j，它们具有相同的 o 与 t 值.

根据“故障-差错-异常”传播机理，若故障激活后的差错引起异常，则该故障对程序带来的影响完全可由相应的 FECF 簇表示. 所以，可通过分析各 FECF 簇，反向推导程序中可引起异常的各个差错及其对程序行为的影响，从“异常”层次分析与异常相关的 EPB，收集可引起异常的差错集合.

3.3　基于程序 ECF 的 EPB 分析

定义 2　设 Ω 和 Ψ 分别为程序 P 的基于 ECF 的 EPB 和 FECF，集合 $D_{\text{P}}^{\text{EPB}}$ 由 P 的 $m(m\geqslant 1)$个 Ω 构成，集合 $D_{\text{P}}^{\text{FECF}}$ 由 P 的 $n(n\geqslant 1)$个 Ψ 构成. 属性 o 与 t 表示可引起 FECF 簇的差错，如下属性表示差错对程序行为造成的影响：

(1) $D_{\text{cluster}}^{\text{FECF}}=\{\Psi|\ \forall(\in D_{\text{P}}^{\text{FECF}},\ \Psi.o=\Omega.o\ \text{且}\ \Psi.t=\Omega.t\}$，表示由具有相同 o 与 t 值的 FECF 组成的 FECF 簇.

(2) $S_{\text{impact}}^{\text{EF}}=\{s|\ \forall\Psi\in D_{\text{cluster}}^{\text{FECF}},\ s=\Psi.e\}$为 $D_{\text{cluster}}^{\text{FECF}}$ 中各 FECF 发生时所对应的功能 EF 集合.

(3) w 为一权值，即：

$$w=\sum_{1}^{|D_{\text{cluster}}^{\text{FECF}}|}\varphi(\Psi_i),\ \Psi_i\in D_{\text{cluster}}^{\text{FECF}} \tag{1}$$

$\varphi(\Psi_i)$为Ψ_i的异常传播距离,即Ψ_i在传播过程中影响到的函数数量.若它属于过程A,则$\varphi(\Psi_i)=1$;若属于过程B,则$\varphi(\Psi_i)=m-k+1$;若属于过程C,则$\varphi(\Psi_i)=m+d_{\text{user}}$,$d_{\text{user}}$为指定的阈值,一般大于程序允许的函数调用栈最大深度.若$\varphi(\Psi_i)>d_{\text{user}}$,则$\Psi_i$为未捕捉的ECF.$w$为$D_{\text{cluster}}^{\text{FECF}}$中所有FECF所影响到的函数数量的总和,该值可用来衡量差错引起异常后对程序控制流、数据流、控制依赖所造成的影响大小.

观察1 w值越大,则EPB对程序可靠性造成危害的风险越大.

异常传播过程B与C的发生会使程序的过程间控制流中产生潜在的非返回调用[15],它的数量(即w值)越大,就意味着异常的传播范围越大,所以程序的控制流、数据流在错误状态下受到的影响越大.

基于ECF的EPB分析的核心是获取程序P中可引起异常的差错信息及各差错对程序行为所造成的影响,关键在于计算$D_{\text{P}}^{\text{EPB}}$和$D_{\text{P}}^{\text{FECF}}$.算法Gen_FECF可计算一个FECF实例.

算法 Gen_FECF:计算一个FECF实例

输入 s:函数m中的raise/throw语句

m:语句s所位于的函数

输出 Ψ:一个FECF实例

声明 callstack:遍历分析函数调用链时所维护的函数调用栈

$C_{\text{TS}}(m)$:函数m的调用者(caller)调用函数m时所使用的函数调用语句

开始 Gen_FECF(s, m)

1. 新建Ψ实例并设置其属性:$o\leftarrow$(m, s的行号),$t\leftarrow s$对应的异常类型,π设置为m,$e\leftarrow$callstack的栈底函数;
2. If SLEGS(s)存在c可捕捉由s引发的异常 then
3. $z\leftarrow$(m, c的代码行号),m加入π:$m\rightarrow m$;
4. return Ψ;
5. else then //当前函数m无法捕捉该异常
6. temp_stack为callstack的副本;
7. $s_{\text{end}}\leftarrow s$; //用于暂存$\Psi$终止时的语句
8. While temp_stack不为空:
9. $m_{\text{top}}\leftarrow$temp_stack.top();
10. temp_stack.pop();
11. if m! $=m_{\text{top}}$ then 将m_{top}加入π endif
12. if temp_stack is empty: //未捕捉异常
13. $z\leftarrow$(m_{top}, s_{end}的行号);
14. if m ! $=m_{\text{top}}$ then
15. 将n_{out}加入π;
16. else
17. 将$m\rightarrow n_{\text{out}}$加入$\pi$;
18. endif
19. return Ψ;
20. else if SLEGS($C_{\text{TS}}(m_{\text{top}})$)有$c$可捕捉 then
21. $z\leftarrow$(temp_stack.top(), c的行号);
22. 将temp_stack.top()加入π;
23. return Ψ;
24. endif
25. $s_{\text{end}}\leftarrow C_{\text{TS}}(m_{\text{top}})$;
26. endwhile
27. endif

结束 Gen_FECF

在计算所有FECF时,e可用于表示FECF所影响到的程序功能或服务.

若在函数递归过程中出现异常,则异常在该部分的传播路径呈现一定的模式.所以,可根据该模式制定合理的裁剪规则,对递归部分的调用链进行裁剪,使得分析过程能够有效地继续执行下去.

在算法Gen_FECF中,函数调用栈是根据函数调用链生成的.由于调用链经过了递归调用链裁减处理,并且调用链中所有函数都属于程序P,函数调用栈的深度最多为$2\cdot|F_{\text{DS}}(\text{P})|-2$,其中$F_{\text{DS}}(\text{P})$为定义于程序P中且在程序P中实现的函数集合.算法Gen_FECF在最好情况下的时间复杂度为$O(1)$,在最坏情况下为$O(|F_{\text{DS}}(\text{P})|)$.

根据EHM的性质[14],对于异常的传播及其处理过程,异常保护区z(try语句所保护的代码块)的异常保护序列EGS(z)指的是z对应的异常处理例程所组成的有序序列(c_1, c_2,…,c_n),$n\geqslant 1$,其顺序由z对应的异常处理例程的顺序决定.假如z_j嵌套于z_i之中,那么传播到z_j中的异常首先会被EGS(z_j)中的异常处理例程尝试捕捉处理,若该异常未被捕捉,则EGS(z_i)再尝试对其捕捉处理,则LEGS(z_j)=(c_{j1},c_{j2},…,c_{jn},c_{i1},c_{i2},…,c_{im}),若z_i不嵌套于任何异常保护区中,则LEGS(z_i)=EGS(z_i)=(c_{i1},c_{i2},…,c_{im}),$m\geqslant 1$,$n\geqslant 1$.若语句s位于z中,则语句s受到LEGS(z)的保护,记为SLEGS(z).若有异常在

s(raise/throw 语句)处被抛出或异常传播至 s(函数调用语句)处，若 SLEGS(s)能捕捉该异常，则该异常在 s 语句所在的函数中被捕捉，否则该异常从当前函数传播出去且沿着函数调用栈逆向传播.

在形如(m_1，m_2，…，m_{n-1}，m_n)所表示的调用栈中，m_1 表示栈底，m_n 表示栈顶，$n\geqslant 1$. $s_{uc}(m_i)$是 m_i 的后继 m_{i+1}，$n\geqslant i\geqslant 1$，$s_{uc}(m_n)=$null. $C_{TS}(s_{uc}(m_k))$为函数 m_k 中调用函数 $s_{uc}(m_k)$时的函数调用语句，且 C_{TS}(null) =null. 为不失一般性，调用链的递归裁剪遵循如下规则.

规则 1　设当前函数调用栈具有以下形式：(m_{bottom}，…，M_1，M_2，…，M_n，M_{11}，M_{12}，…，M_{1n}，…，M_{k1}，M_{k2}，…，M_{kn}，…，M_{q1}，M_{q2}，…，M_{qp}，m_{other}，…)，其中，M_{kn}表示函数 M_n 的第 k 次递归调用，$k\geqslant 1$. M_{qp}表示函数 M_p 第 q 次递归调用，$q>k$，$n\geqslant p\geqslant 0$. 若 $\forall i$，j，$k\geqslant i$，$j\geqslant 1$，式(2)得到满足，则调用栈可裁剪为(m_{bottom}，…，M_1，M_2，…，M_n，M_{q1}，M_{q2}，…，M_{qp}，m_{other}，…)，且 FECF 的 o、t、z、e 属性值保持不变，π 中受到 FECF 影响的函数及其顺序不变.

$$\begin{aligned}&SLEGS(C_{TS}(M_{i1}))=SLEGS(C_{TS}(M_{j1}))=\\&\quad SLEGS(C_{TS}(M_{q1}))\\&SLEGS(C_{TS}(M_{i2}))=SLEGS(C_{TS}(M_{j2}))=\\&\quad SLEGS(C_{TS}(M_2))\\&SLEGS(C_{TS}(M_{in}))=SLEGS(C_{TS}(M_{jn}))=\\&\quad SLEGS(C_{TS}(M_n))\end{aligned}\tag{2}$$

证明.

情形一：若在 M_{kn} 中无异常发生，$n\geqslant 1$，则无 FECF 的生成受到裁剪动作的影响.

情形二：假如异常 x 在函数 M_{kn} 中由 raise/throw 语句引发，$k\geqslant 1$，$n\geqslant 1$：① 若 x 在 M_{kn} 中捕捉，则会有与异常 x 相同的异常在函数 M_n 中由相同的 raise/throw 语句引发且被相同 catch/except 语句捕捉，因为 M_{kn} 与 M_n 为相同函数. ② 若 x 从 M_{kn} 传播出去，则会有与异常 x 相同的异常在函数 M_n 中由相同的 raise/throw 语句引发且从相同的异常出口从 M_n 传播出去，因为 M_{kn} 与 M_n 为相同函数.

情形三：假如异常 x 通过 $C_{TS}(s_{uc}(M_{kn}))$语句从 $s_{uc}(M_{kn})$函数传播至 M_{kn} 中，$k\geqslant 1$，$n\geqslant 1$：① 若 x 在 M_{kn} 中捕捉，则会有与异常 x 相同的异常 y 通过 $C_{TS}(s_{uc}(M_n))$语句从 $s_{uc}(M_n)$传播至 M_n，且在 M_n 中由相同的 catch/except 语句捕捉. 因为 $s_{uc}(M_n)$与 $s_{uc}(M_{kn})$为相同函数，$C_{TS}(s_{uc}(M_n))$与 $C_{TS}(s_{uc}(M_{kn}))$为同一函数调用语句，且 SLEGS($C_{TS}(s_{uc}(M_n))$) = SLEGS($C_{TS}(s_{uc}(M_{kn}))$). ② 若 x 从 M_{kn} 中传播出去，则会有异常 x 相同的异常 y 从 M_n 中的相同异常出口传播出去. 因为 $s_{uc}(M_n)$与 $s_{uc}(M_{kn})$为相同函数，$C_{TS}(s_{uc}(M_n))$与 $C_{TS}(s_{uc}(M_{kn}))$为同一函数调用语句，且 SLEGS($C_{TS}(s_{uc}(M_n))$) = SLEGS($C_{TS}(s_{uc}(M_{kn}))$).

根据情形二与三可知，对于在 M_{kn} 中被引发或是传播至 M_{kn} 的异常 x，无论该异常是在 M_{kn} 中被捕捉或是传播出去，在(M_1，M_2，…，M_n)中总有等同的异常与其对应并具有相同的异常传播过程. 所以，将“M_1，M_2，…，M_n，M_1^1，M_2^1，…，M_n^1，…，M_{k1}，M_{k2}，…，M_{kn}”部分裁剪成“M_1，M_2，…，M_n”后，FECF 实例的 o、t、z、e 及 π 中受到影响的函数及其顺序不变. 证毕.

计算 D_P^{FECF} 的算法如下：

算法　Gen_ D_P^{FECF}：计算所有 FECF 实例数据

输入 S_P^{EF}：程序 P 的入口函数集合

输出 D_P^{FECF}：程序 P 中所有潜在的 FECF

```
开始 Gen_ D_P^FECF(S_P^EF)
1. for each m in S_P^EF:
2.     将调用栈 callStack 初始化为空;
3.     Traverse_Invocation_Chain(m);
4. endfor
结束 Gen_ D_P^FECF
/*遍历以 m 开始的函数调用链，用 Gen_FECF 算法生成 FECF 实例*/
声明 Traverse_Invocation_Chain (m)
1. callstack.push(m); //维护调用栈，m 入栈
2. for each 语句 s in m then
3.     if s 是 raise/throw 语句 then
4.       Ψ←Gen_FECF(s, m);
5.       将 Ψ 加入 D_P^FECF;
6.     else if s 是调用函数 m_callee 的语句 then
7.       if m_callee 在 callstack 出现过两次 and
8.     SLEGS(C_TS(m_callee))=SLEGS(s) then
9.         continue; //满足要求，裁剪
10.       else then //继续分析调用链
11.      Traverse_Invocation_Chain (m_callee);
12.    endif
13.    endif
14. end for
15. Callstack.pop(m) //维护调用栈，m 出栈
```

算法 Gen_D_P^{FECF} 对 S_P^{EF} 中的每一个函数为入口的函数调用链进行了分析，该调用链的深度最多为

$2\cdot|\mathrm{FDS}(P)|-2$. 若一个函数中最坏情况下拥有 C 个函数调用点且都调用了 Gen_FECF 算法，则算法 Gen_D_P^{FECF} 在最坏情况下的时间复杂度为 $O(C\cdot|\mathrm{FDS}(P)|\cdot|\mathrm{FDS}(P)|\cdot|S_P^{EF}|)$.

计算 D_P^{EPB} 的算法如下：

算法 Gen_ D_P^{EPB}：计算所有 EPB 实例数据

输入 D_P^{FECF}：程序 P 中所有潜在 FECF

输出 D_P^{EPB}：程序 P 所有 EPB 错误行为实例

开始 Gen_ $D_P^{EPB}(D_P^{FECF})$

1. for each FECF Ψ in D_P^{FECF}：
2. existEPB← null
3. for each EPB Ω in D_P^{EPB}：//寻找相应 FECF 簇
4. If $\Psi.o=\Omega.o$ and $\Psi.t=\Omega.t$ then
5. existEPB ←Ω; break;
6. endif
7. endfor
8. if null = existEPB then //暂无相应 FECF 簇
9. 创建新的 EPB 实例 Ω_{new}，且初始化其属性：$\Omega_{new}.o\leftarrow\Psi.o$; $\Omega_{new}.t\leftarrow\Psi.t$; $\Omega_{new}.w\leftarrow\varphi(\Psi)$，其 $D_{cluster}^{FECF}$ 与 S_{impact}^{EF} 为空
10. 将 Ψ 加入 $\Omega_{new}.D_{cluster}^{FECF}$;
11. 将 $\Psi.e$ 加入 $\Omega_{new}.S_{impact}^{EF}$;
12. 将 Ω_{new} 加入 D_P^{EPB};
13. else then //将 Ψ 加入对应簇的 EPB 实例
14. existEPB.w ← existEPB.w +$\varphi(\Psi)$;
15. 将 Ψ 加入 existEPB.$D_{cluster}^{FECF}$;
16. 将 $\Psi.e$ 加入 existEPB.S_{impact}^{EF};
17. endif
18. endfor

结束 Gen_ D_P^{EPB}

算法 Gen_ D_P^{EPB} 在计算 EPB 过程中，需要判断是否已经存在相应的 EPB 条目用于处理当前的 FECF，最坏情况下的时间复杂度为 $O(|D_P^{EPB}|)$. 同时，算法 Gen_ D_P^{EPB} 需要处理程序 P 的 D_P^{FECF} 中所有元素，所以最坏情况下的时间复杂度为 $O(|D_P^{FECF}|\cdot|D_P^{EPB}|)$.

4 基于 Understand 的自动分析工具

4.1 EPB 自动分析工具的结构

采用本文提出的方法，用 Python 语言编写了一个基于 Understand 的 EPB 自动分析工具，其架构如图 5 所示. Understand 是一款代码审核工具，提供了可操作源代码相关数据的 API，如词法、语法、函数调用关系等信息.

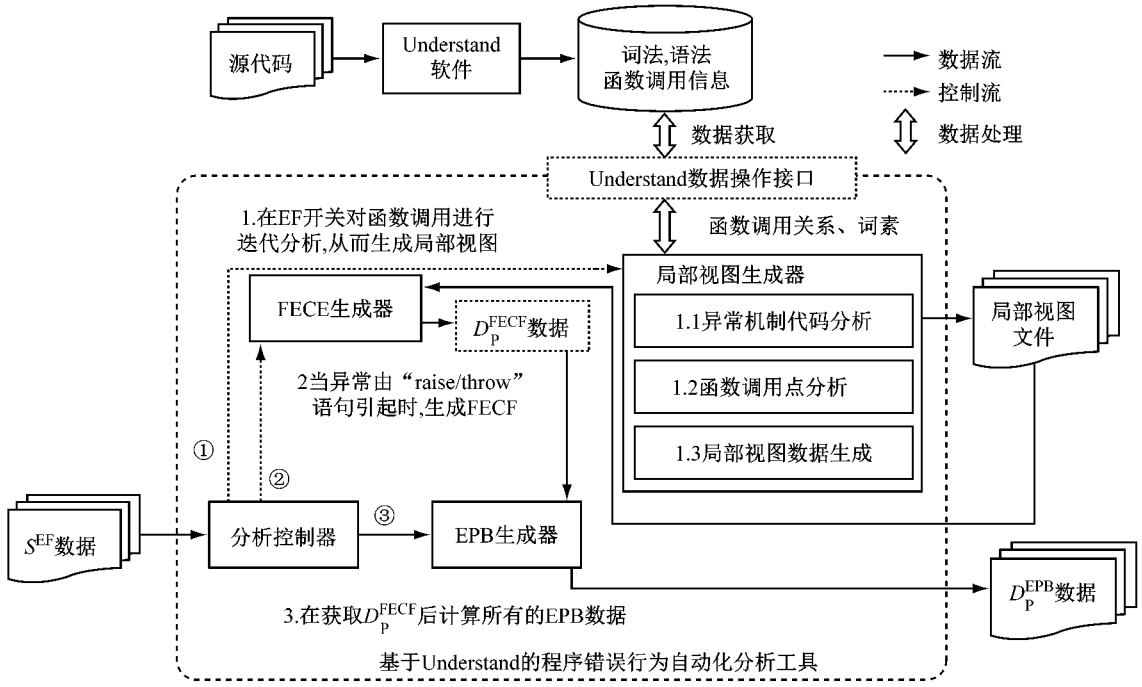


图 5 EPB 自动分析工具架构

Fig.5 Structure of automatic analysis tool of EPB

分析控制器：控制整个分析过程，包括：调用链的遍历分析、局部视图数据生成、FECF 数据生成、

EPB 数据生成、函数调用栈的维护等.

局部视图生成器：函数代码中与 FECF 生成相关的数据以文件形式存储，内容包括：① raise/throw 语句的位置、对应的异常类型；② 在当前函数被引发的异常捕捉情况，记录捕捉该异常的 catch/except 语句的位置及其对应的异常类型；③ 当前函数的函数调用关系信息；④ 当前函数中的函数调用语句是否处于异常保护区中，若是，则记录相应的 catch/except 语句位置及其对应的异常类型.

FECF 生成器：当异常以 raise/throw 形式被引发时，结合相应的局部视图数据，根据 Gen_FECF 算法生成相应 FECF 数据.

EPB 生成器：当 $D_{\mathrm{P}}^{\mathrm{FECF}}$ 计算完毕后，根据 Gen_$D_{\mathrm{P}}^{\mathrm{EPB}}$ 算法生成 $D_{\mathrm{P}}^{\mathrm{EPB}}$ 数据.

4.2　EPB 自动分析工具的使用步骤

自动分析工具进行 EPB 分析的主要步骤如下：

(1) 以源代码为输入，使用 Understand 软件生成程序的词法、语法、函数调用关系等数据.

(2) 以 s 为分析起点，$s\in S^{\mathrm{EF}}$，迭代分析其函数调用链，检查函数内是否有被 raise/throw 语句引发的异常. 若存在异常，则转(3)；否则，继续按照(2)分析函数调用链中未被分析的函数，直到所有调用链分析完毕，转(4).

(3) 结合当前函数调用栈及其相应函数的局部视图数据，计算该 FECF 实例，并将其加入 $D_{\mathrm{P}}^{\mathrm{FECF}}$，返回(2).

(4) 结合(3)中得到的 $D_{\mathrm{P}}^{\mathrm{FECF}}$，计算所有 EPB 数据，得到 $D_{\mathrm{P}}^{\mathrm{EPB}}$.

5　实验及其分析

为了验证本文方法及所开发的程序 EPB 自动分析工具的合理性和有效性，以 OpenStack 的核心组件作为对象进行了实验.

5.1　实验环境

实验在虚拟机上进行，操作系统为内核版本 2.6.32 的 32 位 Linux，硬件配置为 1GB 内存，Intel(R) Core(TM) i3-3217U @ 1.80GHz 单核 CPU. 实验负载为 OpenStack(G 版本)中的 5 个核心组件，该软件用 Python 语言编写，各组件的 S^{EF} 数据均根据相应的应用程序编程接口(application programming interface，API)文档人工收集而得.

5.2　实验结果与分析

对 27 876 行 OpenStack 组件的错误行为的实验结果见表 2，其中 $d_{\mathrm{user}}=10\ 000$ 表示用户指定的阈值.

表 2　OpenStack 组件错误行为分析结果

Tab.2　Analysis results of erroneous OpenStack component behavior

组件名称	代码总量	数据及其分析结果 ($d_{\mathrm{user}}=10\ 000$)								
		文件数量	代码行数	函数数量	调用链中函数调用点总数量	功能入口函数总数	$\lvert D_{\mathrm{P}}^{\mathrm{FECF}}\rvert$	$\lvert D_{\mathrm{P}}^{\mathrm{EPB}}\rvert$	$\lvert D_{\mathrm{P}}^{\mathrm{FECF}}\rvert/\lvert D_{\mathrm{P}}^{\mathrm{EPB}}\rvert$	耗时/s
nova	120 968	122	13 227	665	7 083	13	1 540	186	8.280	2 156
quantum	60 485	51	4 245	234	1 347	6	390	77	5.065	738
glance	21 261	44	2 320	137	251	5	86	58	1.483	107
keystone	20 071	21	1 512	77	195	6	104	57	1.825	119
cinder	49 797	60	6 572	313	1 532	6	376	134	2.806	404
总计	321 081	298	27 876	1 426	10 408	36	2 496	512	4.875	3 524

调用链中函数调用点总量与函数数量的比值为 10 408/1 426=7.3，说明每个函数平均被调用了 7.3 次，若差错在函数中引起异常，则该异常可潜在引起由 7 个不同 ECF 组成的 FECF 簇，表明从异常角度对差错及其对程序行为影响进行分析是可行的.

$\lvert D_{\mathrm{P}}^{\mathrm{FECF}}\rvert/\lvert D_{\mathrm{P}}^{\mathrm{EPB}}\rvert=4.875$，说明每个差错平均可引起近 5 个不同的 FECF，故障激活后的差错若被异常处理机制侦测，则该差错对程序行为的影响可通过对相应的 FECF 簇进行评估来分析，同时，以“故障-差错-异常”过程作为切入点，基于 ECF 对 EPB 进行分析是合理的和有效的.

图 6 显示了 nova 中以 w 排序的前十个 EPB 实例数据. 其中，纵轴左侧数值表示为 $D_{\mathrm{cluster}}^{\mathrm{FECF}}$ 的值，右侧数值表示为 $S_{\mathrm{impact}}^{\mathrm{EF}}$ 的值. 横坐标为 nova 具体 EPB 所对应的差错. 以 nova 中排序第一的 EPB 实例数据 $\Omega_{\mathrm{one}}^{\mathrm{nova}}$ 为例，$\Omega_{\mathrm{one}}^{\mathrm{nova}}$ 所代表的差错在 nova.utils.execute 函数中引起了类型为 nova.exception.NovaException 的异常，该差错是函数解析输入命令时出现了未定义关键字. 由于该函数用于解析输入命令，并根据命令创建新进程提供相应的服务，所以该函数被其他众多函数所调用. $\Omega_{\mathrm{one}}^{\mathrm{nova}}$ 的 $\lvert S_{\mathrm{impact}}^{\mathrm{EF}}\rvert=13$，说明 nova 的所有 EF 所对应的调用链均含有此函数，若该函数中存在故障且该故障激活后产生的差错引起异常，则会影响到 nova 所有功能的正常执

行.Ω_{one}^{nova}所代表的差错可潜在地引起近100个FECF,w≈640,小于d_{user},所以其$D_{cluster}^{FECF}$中不存在未捕捉异常,但仍然需要小心应对该差错.该数据也说明了观察1的合理性.

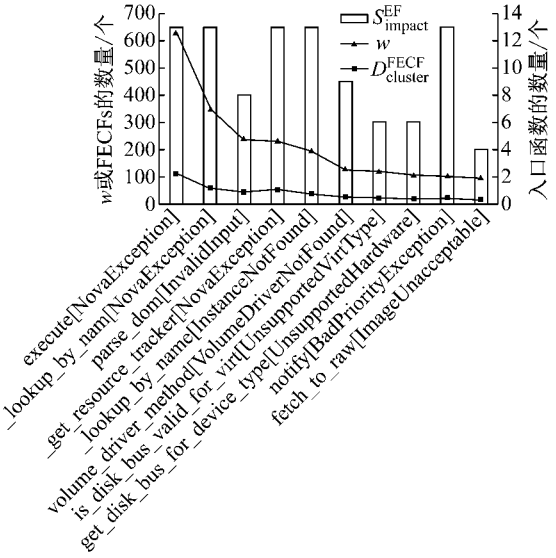


图6 nova组件中以w排序的前十个EPB实例数据

Fig.6 Top ten EPB Ω ordered by w in nova components

6 结束语

本文通过故障注入实验研究了具有EHM的程序的软件故障、差错、异常之间的传播机理.结合FECF的描述,说明了如何利用ECF收集与异常相关的差错信息,建立了基于ECF的EPB模型,开发了基于该模型的EPB自动分析工具,并用该工具对OpenStack核心组件进行了错误行为分析,结果表明了基于ECF的EPB分析的合理性和有效性.该方法及其工具为具有EHM的大规模程序的EPB自动分析,指导程序异常处理结构的重构,使程序可精确捕捉异常,提高程序可靠性,并为构造合理的外部激励精确激活差错,缩短故障注入实验周期提供了有效手段.

参考文献:

[1] DURAES J A, MADEIRA H S. Emulation of software faults: a field data study and a practical approach[J]. IEEE Transactions on Software Engineering, 2006, 32(11): 849.

[2] MARINESCU P D, CANDEA G. LFI: a practical and general library-level fault injector[C]//Proceedings of the 39 th IEEE/IFIP International Conference on Dependable Systems and Networks. Lisbon: IEEE, 2009: 379-388.

[3] CHRISTMANSSON J, CHILLAREGE R. Generation of an error set that emulates software faults based on field data[C]//Proceedings of the 26th IEEE Symposium on Fault Tolerant Computing. Sendai: IEEE, 1996: 304-313.

[4] MORAES R, BARBOSA R, DURAES J, *et al*. Injection of faults at component interfaces and inside the component code: are they equivalent[C]//Proceedings of the 6th European Dependable Computing Conference. Coimbra: [s. n.], 2006: 53-64.

[5] ZHANG P, ELBAUM S. Amplifying tests to validate exception handling code[C]//Proceedings of the 34th International Conference on Software Engineering. Zurich: IEEE, 2012: 595-605.

[6] ARLAT J, MORAES R. Collecting, analyzing and archiving results from fault injection experiments[C]//Proceedings of 5th Latin-American Symposium on Dependable Computing. [S. l.]: Sao Jose dos Campos, 2011:100-105.

[7] 张丹青, 江建慧, 陈林博.一种对程序故障行为和失效行为的聚类有效性验证方法[J]. 中国科学(信息科学), 2014, 44(10):1323.
ZHANG Danqing, JIANG Jianhui, CHEN Linbo. A method for validating the effectiveness of fault clustering and failure clustering of programs [J]. Science China (Information Science), 2014, 44(10): 1323.

[8] SHAH H B, GORG C, HARROLD M J. Understanding exception handling: Viewpoints of novices and experts [J]. IEEE Transactions on Software Engineering, 2010, 36(2): 150.

[9] CRISTIAN F. Exception handling and software fault tolerance [J]. IEEE Transactions on Computers, 1982, 100(6): 531.

[10] LEI B, LI X, LIU Z, *et al*. Robustness testing for software components[J]. Science of Computer Programming, 2010, 75(10): 879.

[11] SHAHROKNI A, FELDT R. A systematic review of software robustness[J]. Information and Software Technology, 2013, 55(1): 1.

[12] JOHANSSON A, SURI N, MURPHY B. On the impact of injection triggers for OS robustness evaluation [C]// Proceedings of 18th IEEE International Symposium on Software Reliability. Trollhattan: IEEE, 2007: 127-136.

[13] 姜淑娟, 徐宝文, 史亮. 一种基于异常传播分析的数据流分析方法[J]. 软件学报, 2007, 18(1): 74.
JIANG Shujuan, XU Baowen, SHI Liang. An approach of data-flow analysis based on exception propagation analysis [J]. Journal of Software, 2007, 18(1): 74.

[14] SINHA S, HARROLD M J. Analysis and testing of programs with exception handling constructs[J]. IEEE Transactions on Software Engineering, 2000, 26(9): 849.

[15] HARROLD M J, ROTHERMEL G, SINHA S. Computation of interprocedural control dependence [J]. ACM SIGSOFT Software Engineering Notes, 1998, 23(2): 11.

[16] SINHA S, ORSO A, HARROLD M J. Automated support for development, maintenance, and testing in the presence of implicit flow control [C]//Proceedings of the 26th IEEE International Conference on Software Engineering. Edinburgh: IEEE, 2004: 336-345.

[17] COELHO R, RASHID A, GARCIA A, *et al*. Assessing the

impact of aspects on exception flows: an exploratory study [C]//Proceedings of the 22nd European Conference on Object-Oriented Programming. Berlin Heidelberg: Springer, 2008: 207-234.

[18] JO J W, CHANG B M, YI K, *et al*. An uncaught exception analysis for Java[J]. Journal of Systems and Software, 2004, 72(1): 59.

[19] 金大海，宫云战，杨朝红，等. 运行时异常对软件静态测试的影响研究[J]. 计算机学报，2011，34(6)：1090.
JIN Dahai, GONG Yunzhan, YANG Zhaohong, *et al*. Research on the effect of runtime exception in software static testing [J]. Journal of Computers, 2011, 34(6):1090.

[20] 姜淑娟，徐宝文，史亮. 一种基于异常传播分析的依赖性分析方法[J]. 软件学报，2007，18(4)：832.
JIANG Shujuan, XU Baowen, SHI Liang. An approach to analyzing dependence based on exception propagation analysis [J]. Journal of Software, 2007, 18(4): 832.

(上接第 919 页)

[9] VEST A, STAMATIADIS N. Use of warning signs and markings to reduce speeds on curves[C]// 3rd International Symposium on Highway Geometric Design. Chicago Illinois: Transportation Research Board, 2005: 1-17.

[10] DING H, ZHAO X, RONG J, *et al*. Experimental research on the effectiveness of speed reduction markings based on driving simulation: a case study [J]. Accident Analysis & Prevention, 2013, 60(11): 211.

[11] GODLEY S T, TRIGGS T J, FILDES B N. Driving simulator validation for speed research [J]. Accident Analysis & Prevention, 2002, 34(5): 89.

[12] JAMSON S, LAI F, JAMSON H. Driving simulators for robust comparisons: a case study evaluating road safety engineering treatments [J]. Accident Analysis & Prevention, 2010, 42(3):961.

[13] 潘晓东，蒋宏，杨轸. 山区公路小半径曲线事故黑点案例分析[J]. 同济大学学报(自然科学版)，2007，35(12)：1642.
PAN Xiaodong, JIANG Hong, YANG Zhen. A case study of accident black spot in mountain highway with small radius [J]. Journal of Tongji University (Natural Science). 2007, 35(12): 1642.

[14] 张宝君，刘军，金灿. 公路长直线接小半径平曲线探究[J]. 交通科技，2014(3)：93.
ZHANG Baojun, LIU Jun, JIN Can. Study on long straight connecting sharp curve on highway [J]. Transportation Science & Technology, 2014(3): 93.

[15] 杨少伟，张碧琴，许金良，等. 道路勘测设计[M]. 第 3 版. 北京：人民交通出版社，2009.
YANG Shaowei, ZHANG Biqin, XU Jinliang, *et al*. Road surveying and design [M]. 3rd. Beijing: China Communications Press, 2009.

[16] 华杰工程咨询有限公司. 公路项目安全性评价规范：JTGB05—2015[S]. 北京：人民交通出版社股份有限公司，2015.
Chelbi Engineering Consultants, Inc.. Specifications for highway safety audit: JTGB05—2015 [S]. Beijing: China Communications Press, Co., Ltd., 2015.

[17] 彭其渊，徐进，罗庆，等. 公路平曲线参数对车辆轨迹和速度的影响规律[J]. 同济大学学报(自然科学版)，2012，40(1)：45.
PENG Qiyuan, XU Jin, LUO Qing, *et al*. Effect of horizontal curves design on track and speed of passenger car [J]. Journal of Tongji University (Natural Science), 2012, 40(1): 45.

[18] SHINAR D, COMPTON R. Aggressive driving: an observational study of driver, vehicle, and situational variables [J]. Accident Analysis & Prevention, 2004, 36(3): 429.

[19] SPACEK P. Track behavior in curve areas: attempt at typology [J]. Journal of Transportation Engineering, 2005, 131(9):669.