软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(7):1688−1701 [doi: 10.3724/SP.J.1001.2012.04126]
©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

基于约束的软件失效域识别与特征分析*

孙昌爱[1,2+]

[1](北京科技大学 计算机与通信工程学院,北京 100083)

[2](中国科学院 软件研究所 计算机科学国家重点实验室,北京 100190)

A Constraint-Based Approach to Identifying and Analyzing Failure-Causing Regions

SUN Chang-Ai[1,2+]

[1](School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China)

[2](State Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: casun@ustb.edu.cn

Sun CA. A constraint-based approach to identifying and analyzing failure-causing regions. *Journal of Software*, 2012,23(7):1688−1701 (in Chinese). http://www.jos.org.cn/1000-9825/4126.htm

Abstract: Random testing is a widely practiced black-box testing technique. Recently, adaptive random testing has been proposed to improve the random testing, and simulation results show that the improvements depend on the characteristic of failure-causing regions of program under test. This paper presents the concept of test constraints and employs them to specify the distribution of failure-causing regions within the input domain of program under test. Characteristic analysis of failure-causing regions can be conducted on the base of their test constraints, which are derived using the available program analysis techniques. To evaluate the proposed technique, a case study on a real-life application was conducted, and the results show that the proposed test constraint provides an insight into how a failure is triggered and propagated, and the constraint-based analysis helps to improve the quality of test case design and assess the applicability of the adaptive random testing.

Key words: software testing; test case; software failure; software defect; program analysis

摘 要: 随机测试是实践中广泛采用的一种黑盒测试方法.近年来提出的适应性随机测试方法改进了随机测试的不足,仿真实验结果表明,改进效果取决于软件失效域的特征.提出以测试约束刻画软件失效域在输入域上的分布,探讨了基于现有的程序分析技术构造测试约束的过程,讨论了基于测试约束的软件失效域的特征分析方法.以一个实例软件验证所提出的测试约束构造过程及其软件失效域特征分析方法.测试约束揭示了软件故障的触发与传播的内在机制,基于测试约束的软件失效域的特征分析方法有助于改进测试用例的设计质量以及评价适应性随机测试方法的适用性.

关键词: 软件测试;测试用例;软件失效;软件缺陷;程序分析

中图法分类号: TP311 **文献标识码**: A

* 基金项目: 国家自然科学基金(60903003); 北京市自然科学基金(4112037); 国家教育部博士点基金(20080004010 51); 中国科学院软件研究所计算机科学国家重点实验室开放课题(SYSKF1105); 中央高校基本科研业务费资助项目(FRF-SD-12-015A)

收稿时间: 2011-01-26; 修改时间: 2011-08-23; 定稿时间: 2011-09-30

© 中国科学院软件研究所 http://www.jos.org.cn

为了提高软件的可靠性,并尽可能地发现潜藏在程序中的缺陷,通常要对软件进行测试,即通过尝试输入一些测试数据来观察被测软件是否失效.由于软件固有的复杂性,穷尽测试是不可能的[1].随机测试是一种在实践中广泛采用的黑盒测试技术[2],由于没有利用被测程序的相关信息,例如导致软件失效的输入往往是群束的(clustered)[3−8],随机测试并不是高效的.为了改进随机测试方法的不足,Chen 等人[9]提出了适应性随机测试(adaptive random testing)方法,通过使测试用例在输入域的分布更加均匀来改进随机测试方法的测试用例的选择过程[10],并开发了多种适应性随机测试的高效实现算法[9−14].通过仿真实验,验证了在软件失效域表现为不同模式下的适应性随机测试的性能,结果表明,适应性随机测试明显优于随机测试.理论研究也表明[15],以检测到第 1 个失效所需要的测试用例数为度量,适应性随机测试改进随机测试的性能的上限是 50%,这意味着适应性随机测试可以节约大量的测试成本.仿真实验进一步揭示了影响适应性随机测试性能的几种因素[16,17],这些因素都直接与软件失效域的特征有关.当适应性随机测试用来测试真实的程序时,如何识别软件失效域、如何分析软件失效域的特征是一个尚未解决的问题.

本文提出了测试约束的概念,即为了检测某个故障输入变量应遵循的一组约束.运行满足约束的测试数据可以检测到该故障,测试约束仅仅与输入域有关,其构造过程依赖于现有的程序分析技术.基于测试约束,可以进一步分析故障对应的软件失效域的分布特征.测试约束的概念揭示了软件故障触发与传播的内在机制,基于测试约束的软件失效域特征分析,可以用来指导高质量的测试用例设计、评价适应性随机测试的适用性.

与已有研究不同,本文提出的软件失效域特征分析方法针对的是实际程序.需要指出的是,本文讨论的对象设定为数值型程序;研究动机是回答适应性随机测试在实际程序中的适用性问题,即如果程序中存在某类错误,那么如何分析失效域的特征,以便决定是否运用适应性随机测试方法.

1 测试约束

软件开发过程中人为的错误将会导致设计上的缺陷,这些缺陷在编码阶段演变为故障.如果含有故障的代码在运行时被执行,那么将发生软件失效,即软件系统在指定的范围内没有完成规定的功能[18].为了检测出程序中潜藏的故障,必须选择并运行属于软件失效域上的测试用例.

变异测试(mutation testing)[19]又称为变异分析,最早是由 DeMillo 等人提出用来评价测试用例集的充分性.通过对一个正确的程序 p 执行变异算子,得到一个或多个变异体 m(即潜藏故障的程序),如果测试用例集能够区分 p 和 m,则称该变异体 m 被杀掉;如果不存在测试用例能够杀死 m,则称该变异体为等价变异体.不同的变异算子导致了不同类型的故障.Andrews 等人[20]指出,自动生成的变异体与真实的故障非常相似,因此本文运用变异分析技术产生程序中的各种故障.本文研究的问题之一可以归结为,假设某条语句存在一个已知的变异(对应于一个故障),那么如何找到导致程序失效的输入子域(失效域)?为了便于讨论,我们假设与原始程序相比潜藏故障的程序仅包含一个故障(在后文的讨论中,考虑了多个故障并存的情形).

定义 1(软件失效域φ). 给定一个原始程序ρ,输入域为ψ,在ρ中引入一个故障α后得到ρ'(即ρ'为ρ的一个非等价变异体),则α的软件失效域φ定义为

$$\{\tau|\tau\in\psi\wedge\rho(\tau)\neq\rho'(\tau)\},$$

其中,$\rho(\tau)$和$\rho'(\tau)$表示输入数据为τ时程序ρ和ρ'的输出.

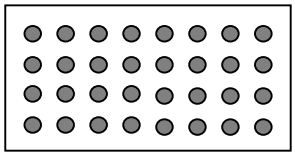
假设被测程序的输入域是二维的(具有两个输入变量).Chen 等人[7]将软件失效域粗略地划分为点模式(point)、带模式(strip)和块模式(block),如图 1 所示.图 1 的上部是几种常见的含有故障的源程序,图 1 的下部是相应的软件失效域示意图.其中,矩行框表示被测程序的输入域,而灰色填充部分表示软件失效域.图 1(a)示意了点模式的软件失效域,软件失效域并不集中于一个或几个区域中,而是分散在整个输入域中.在图 1(a)所示的源程序中,if 语句的真分支应该执行语句“z:=0”,但错误地写成“z:=1”;该程序的输入域是一个二维整数矩阵.若要检测出该程序中的故障,测试用例必须满足要求条件:x mod 2=0∧y mod 2=0.因此,软件失效域为分散的点.类似地,图 1(b)和图 1(c)示意了导致软件失效域为带模式和块模式的软件故障、相应的源程序例子.

© 中国科学院软件研究所 http://www.jos.org.cn

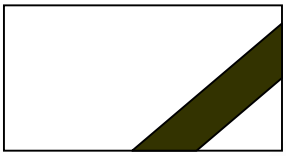
```
int cal1 (int x, int y) {
  if ((x mod 2)==0) && ((y mod 2)==0)
    z:=1;
   /*the correct one should be "z:=0"*/
  else
    z:= 1;
  return (z);
}
```

```
int cal2 (real x, real y) {
  if (x–y>4)
    /*the correct one should be
      "if (x–y>6)"*/
    z:=0;
  else
    z:= 1;
  return (z); }
```

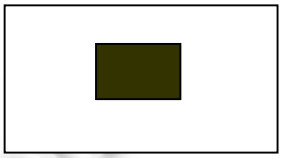
```
int cal3 (real x, real y) {
  if (x>4 && x<6) && (y>4 && y<6)
    z:=1;
  /*the correct one should be "z=0"*/
  else
    z:=1;
  return (z);
  }
```



(a) Point



(b) Strip



(c) Block

Fig.1 Typical patterns of failure-causing regions

图 1 3 种典型的软件失效域模式示意图

本文提出了测试约束的概念,以此刻画某个故障对应的软件失效域.

定义 2(测试约束Γ). 给定一个原始程序ρ,输入域为ψ,ρ'是ρ中引入故障α后得到的非等价变异体,则α对应的测试约束Γ定义为

$$\Gamma: \psi \rightarrow \varphi,$$

其中,φ为α的软件失效域,即$\varphi=\{\tau \mid \tau \in \psi \wedge \rho(\tau) \neq \rho'(\tau)\}$.

图 1(a)~图 1(c)示意的故障测试约束分别为 $x \bmod 2=0 \wedge y \bmod 2=0$,$4<x-y<6$ 和 $4<x<6 \wedge 4<y<6$.

2 测试约束的构造方法

测试约束是定义在输入变量上的一组约束条件,满足约束条件的所有输入数据构成了软件失效域.检测到某个故障导致的软件失效的必要充分条件包括:(1) 测试数据能够触发潜藏的故障代码;(2) 故障发生时原始程序和包含故障的变异体转移为不同的程序状态;(3) 错误的状态能够传播并产生不同的输出.

这些条件就构成了测试约束.我们用一个例子介绍故障触发与传播等相关概念,讨论测试约束的构造算法.

2.1 相关概念与程序分析技术

定义 3(原始程序ρ). 原始程序ρ是一个三元组$\langle I,O,S\rangle$,其中,I 是一组输入数据,表示为参数向量$\langle V_1,V_2,\ldots,V_n\rangle$ ($n>0$);O 是一组语句;S 是一组状态.ρ的一个状态 s_i 可以表示为输入向量的实例$\langle v_1,v_2,\ldots,v_n\rangle$,其中,$v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n$.$S_{init} \in S$ 和 $S_{final} \subseteq S$ 是ρ的初始试态和最终状态集合.ρ的一条语句 $o_i \in O$ 是一个状态转换 $s_i \xrightarrow{o_i} s_j$,其中,$s_i \in S, s_j \in S$.

定义 4(非等价变异体 f). 对原始程序ρ执行一个变异算子 m,可以得到一个变异体 f,即

$$\exists l,n(l \in O^{\rho} \wedge n \in O^{f} \wedge O^{f}=O^{\rho}(l/n)),$$

其中,O^{ρ}和 O^{f} 分别表示ρ和 f 的语句集合,$O^{\rho}(l/n)$表示将 O^{ρ}中的元素 l 替换为元素 n.当 f 满足如下条件时,则称 f 为非等价变异体(又称为ρ的错误版本):

$$\exists x(x \in I^{\rho} \rightarrow S^{\rho}_{final} \neq S^{f}_{final}),$$

其中,I^{ρ}表示ρ的输入集合,S^{ρ}_{final} 和 S^{f}_{final} 表示ρ和 f 的最终状态集合.

图 2 示意了原始程序 *alt_test* 的一个非等价变异体例子 f_1.程序的第 4 行出现了一个操作符故障,即操作符号"<"被错误地写成了"<=".*alt_test* 的输入集合包括 4 个参数:前 2 个是整型,后 2 个是布尔型.*alt_test* 的操作集合包括赋值、逻辑比较、关系比较.*alt_test* 的最终状态表示为输出一个整型数 *alt_sep*.显然,并不是所有的测试用例都能够检测出 f_1 中的故障.例如,当测试数据为$\langle 1,1,1,1\rangle$时,原始程序 *alt_test* 和变异体 f_1 都输出 0.该操作符

© 中国科学院软件研究所 http://www.jos.org.cn

故障没有被检测出来.

```
int alt_test (int own_alt, int other_alt, bool preflag, bool postflag) {
1   bool upward, downward;
2   int alt_sep=0;
3   if (!preflag) {
4   upward=own_alt<=other_alt ?1:0;
     /*The correct one should be "own_alt<other_alt ?1:0".*/
5    downward=other_alt<own_alt ?1:0;
6    if (postflag) {
7        if (upward && downward)     alt_sep=0;
8        else if (upward)            alt_sep=1;
9        else if (downward)           alt_sep=2;
10       else                        alt_sep=0;
11   }
12  }
13       return alt_sep;
}
```

Fig.2 An example of non-equivalent mutants

图 2 一个非等价变异体例子

定义 5(故障触发的前提条件 Pre-Condition). 原始程序ρ的一个非等价变异体 f,假设变异算子 m 施加于语句 o_i^ρ 上,即 $o_i^\rho \neq o_i^f$,其中, $o_i^\rho \in O^\rho \wedge o_i^f \in O^f$, O^ρ和 O^f 分别表示ρ和 f 的语句集合.那么,将如下约束θ称为故障触发的前提条件 Pre-Condition:

$$\theta: I^\rho \xrightarrow{c} \{x \mid x \in I^\rho \wedge x \xrightarrow{e} o_i^f\},$$

其中,I^ρ表示原始程序ρ的输入集合,$a \xrightarrow{c} b$ 的含义是将集合 a 映射为 b,$d \xrightarrow{e} g$ 的含义是输入测试数据 d 则导致语句 g 的执行.

图 2 示意的故障的触发前提条件为 *preflag*=0,即只要输入参数 *preflag* 限制为 0,则该故障就会被触发.其他输入参数没有限制,意味着可以取输入域中所有的值.

定义 6(故障传播条件 Propagation-Condition). 原始程序ρ的一个非等价变异体 f,假设突变操作 m 施加于语句 o_i^ρ 上,即 $o_i^\rho \neq o_i^f$,其中 $o_i^\rho \in O^\rho \wedge o_i^f \in O^f$, O^ρ和 O^f 分别表示ρ和 f 的语句集合.那么,将如下约束σ 称为故障传播条件 Propagation-Condition:

$$\sigma: I^\rho \xrightarrow{c} \{x \mid x \in I^\rho \wedge x \xrightarrow{e} o_i^f \ldots s_{final}^f \wedge x \xrightarrow{e} o_i^\rho \ldots s_{final}^\rho \wedge s_{final}^\rho \neq s_{final}^f\},$$

其中,I^ρ表示原始程序ρ的输入集合,$a \xrightarrow{c} b$ 的含义是将集合 a 映射为 b,$d \xrightarrow{e} g \ldots s$ 的含义是输入测试数据 d 则导致 g 以后的语句的执行、直至进入状态 s, s_{final}^ρ 和 s_{final}^f 分别表示ρ和 f 的最终状态集合.

图 2 中示意的故障的传播条件为 *postflag*=1∧*own_alt*=*other_alt*.如果 *postflag*≠1 或 *own_alt*≠*other_alt*,则该故障执行后并不导致不同的输出状态.

我们将测试约束的计算规则定义为:"对于原始程序ρ,输入域为ψ,f 是ρ中引入故障α后得到的一个非等价变异体,则故障α的测试约束$\Gamma(\alpha)$应为故障α的触发前提条件 *Pre-Condition*(α)与故障α的传播条件 *Propagation-Condition*(α)的交集,即$\Gamma(\alpha)$=*Pre-Condition*(α)∧*Propagation-Condition*(α)".

运用该规则,可以计算出图 2 示意的故障的测试约束:*preflag*=0∧*postflag*=1∧*own_alt*=*other_alt*.例如,〈50,50,1,0〉是满足该测试约束的一个测试用例,运行该测试用例,原始程序ρ和变异体 f 分别输出 0 和 1.

定义 7(程序切片 program slicing[21]). 给定一个程序ρ,t 是ρ中的一条语句,语句 t 的程序切片 *slicing*(t)={s_1, s_2,…,s_n},其中,s_i(i=1,...,n)是ρ中的一条语句,且它的执行将影响程序ρ在语句 t 处的状态.t 称为切片标准.

程序切片是一种操作,抽取影响切片标准的一组语句集合,有助于程序理解与调试.例如,图 2 中第 8 条语句的程序切片为 *slicing*(8)={3,4,6},第 10 条语句的程序切片为 *slicing*(10)={3,4,5,6}.

定义 8(程序劈片 program chopping[21]). 给定一个程序ρ,s 和 t 是ρ中的源切片标准和目标切片标准,则 s 与 t 之间的程序劈片 *chopping*(s,t)={s_i|$s_i \in s \xrightarrow{*} t$ },其中, $s \xrightarrow{*} t$ 表示ρ中由 s 到 t 路径上语句的集合,且每条

© 中国科学院软件研究所 http://www.jos.org.cn

语句的执行将影响程序ρ在语句 t 处的状态.

程序劈片是一种过滤后的程序切片.例如,图 2 中第 3 条语句和第 8 条语句之间的劈片为 *chopping*(3,8)={3,4,6},第 3 条语句和第 10 条语句之间的程序劈片为 *chopping*(3,10)={3,4,5,6}.为了引入更高的抽象,用程序依赖图 PDG[22]表示程序劈片,源切片标准和目标切片标准被替换为相应的结点.

定义 9(路径条件 Path-Condition[21]). 给定一个程序ρ,s 和 t 是ρ中的源切片标准和目标切片标准,则 s 与 t 之间的路径条件 *path_contidtion*(s,t)={$ec|ec\mapsto s_i\wedge s_i\in chopping(s,t)$},其中,$ec\mapsto s_i$ 表示语句 s_i 执行的条件.

路径条件描述了程序劈片执行时应满足的传递依赖条件.例如,图 2 中语句 3~语句 10 的执行条件如下:

- *ec*(3):=true;
- *ec*(4):=true∧*preflag*=false;
- *ec*(5):=true∧*preflat*=false;
- *ec*(6):=true∧*preflat*=false;
- *ec*(7):=true∧*preflat*=false∧*postflat*=true∧*own_alt*≤*other_alt*∧*other_alt*<*own_alt*;
- *ec*(8):=true∧*preflat*=false∧*postflat*=true∧*own_alt*≤*other_alt*;
- *ec*(9):=true∧*preflat*=false∧*postflat*=true∧*other_alt*<*own_alt*;
- *ec*(10):=true∧*preflat*=false∧*postflat*=true∧*own_alt*>*other_alt*∧*own_alt*≤*other_alt*.

由于 *chopping*(3,8)={3,4,6},所以,

path_conditions(3,8):=*ec*(3)∧*ec*(4)∧*ec*(6)∧*ec*(8):=*preflat*=false∧*postflat*=true∧*own_alt*≤*other_alt*.

此外,不难计算出 *path_conditions*(3,10):=false,表明语句 10 是不可达的.

定义 10(故障敏感的偏移量约束条件 Offset). 给定一个程序ρ,语句 s 引入故障α后得到语句 s',且 s 中包含输入变量 V_i,那么α敏感的 V_i 偏移量约束条件 *Offset*(α)定义为

$$\{t|t\in V_i\}\rightarrow\{t|t\in V_i\wedge assume(s(V_i/t))\oplus assume(s'(V_i/t))\},$$

其中,$s(V_i/t)$表示语句 s 中所有变量 V_i 出现的地方用 t 替换,$s'(V_i/t)$表示语句 s'中所有变量 V_i 出现的地方用 t 替换.*assume*(s)的定义如下:

$$assume(s)=\begin{cases}V_i=exp, & \text{当 } s \text{ 是赋值语句,例如 } V_i=exp,\text{其中,}exp \text{ 表示常量或表达式,}\\ V_i \text{ op } exp, & \text{当 } s \text{ 是分支语句,例如 if } (V_i \text{ op } exp)\text{或者 when } (V_i \text{ op } exp),\\ & \text{其中,op}\in\{>,\geq,==,<,\leq,!=\},exp \text{ 表示常量或表达式.}\end{cases}$$

按照上述定义,图 2 中故障敏感的偏移量约束条件为 *own_alt*=*other_alt*.该故障α导致语句 *s*(即"*own_alt*<*other_alt* ?1:0")被错误地实现为 *s*′(即"*own_alt*≤*other_alt* ?1:0"),α涉及输入变量 *own_alt* 和 *other_alt*.以变量 *own_alt* 为例,*s* 和 *s*′是一个分支语句,*assume*(*s*)为"*own_alt*<*exp*",*assume*(*s*′)为"*own_alt*≤*exp*".为了使 *own_alt*<*exp*⊕ *own_alt*≤*exp* 成立,则有 *own_alt*=*exp*,其中,*exp* 为 *other_alt*.相似地,对于变量 *other_alt*,其故障敏感的偏移量约束条件为 *other_alt*=*exp*,其中,*exp* 为 *own_alt*.

2.2 测试约束的构造算法

基于上述程序分析技术,可以获取故障触发的前提条件和故障传播条件,进而构造某个故障相应的测试约束.不妨假设故障α发生在原始程序ρ中的语句 *s*,故障触发的前提条件为

$$Pre\text{-}Condition(\alpha)=Path\text{-}Condition(s_{init},slicing(s)),$$

其中,s_{init} 表示ρ的起始语句;故障传播条件为

$$Propagation\text{-}Condition(\alpha)=offset(\alpha)\wedge Path\text{-}Condition(s,chopping(s,s_{final})),$$

其中,*offset*(α)为故障敏感的偏移量的约束条件,s_{final} 表示ρ中产生可观测结果的输出语句集(或结束语句).

所以,故障α对应的测试约束为

$$\Gamma(\alpha)=Path\text{-}Condition(s_{init},slicing(s))\wedge offset(\alpha)\wedge Path\text{-}Condition(s,chopping(s,s_{final})).$$

算法 1 描述了测试约束的构造过程.该算法集成并利用了程序切片、程序劈片、路径条件和故障敏感的数

© 中国科学院软件研究所 http://www.jos.org.cn

据分析技术,假设被测程序是一个只包含一个函数的C程序、只有一个故障语句、而且只包含赋值(assignment)、分支(branch)、跳转(goto)和返回(return)这4种类型的语句.首先,算法构造被测程序 p 的程序依赖图PDG,将程序的起始语句 s_{init}、故障语句 s'和输出语句 s_{final} 映射到PDG的结点上.其中,$PDG=\langle N,E\rangle$,N 表示结点集合,E 表示边集合.对于每个语句 $s_i\in p$,则存在一个相应的结点 $n_k\in N$.语句 $s_i\in p$ 和 $s_j\in p$ 在PDG中相应的结点分别为 $n_l\in N$ 和 $n_m\in N$,如果 s_i 和 s_j 之间存在依赖关系,则PDG必然存在一个相应的边 $e_{l,m}\in E$;其次,利用程序切片分析技术获得从起始语句 s_{init} 到故障语句 s'的路径约束条件(第4步),算法构造故障语句 s'的程序切片 $slicing(s')$,构造故障触发的前提条件 $Path\text{-}Condition(s_{init},slicing(s))$;再次,利用程序劈片分析技术获得从故障语句 s'到输出语句 s_{final} 的路径约束条件(第6步),算法构造故障语句 s'至输出语句 s_{final} 之间的程序劈片 $chopping(s',s_{final})$,计算故障传播条件 $offset(\alpha)\wedge Path\text{-}Condition(s,chopping(s,s_{final}))$;最后,返回测试约束的结果.其中,过程 $Construct\_PDG(p)$,$Construct\_Slicing(PDG,s)$,$Construct\_Chopping(PDG,s,t)$,$Construct\_PathCondition(s)$,$Get\_Offset(s,s')$ 分别用来构造程序 p 的程序依赖图、构造语句 s 的程序切片、构造语句 s 和语句 t 之间的程序劈片、构造语句 s 的路径条件、计算故障$(s\rightarrow s')$敏感的偏移量约束条件.所谓路径约束条件,即指执行路径上所有条件为真的约束集合.上述程序分析技术已在文献[21,23,24]中有详细的讨论,本文不再赘述.

算法 1. 测试约束的构造算法 *TestConstraint*.

INPUT:

p:$\{s_i|\text{type}(s_i)\in\{\text{assignment,branch,goto,return}\}\wedge 0<i\leqslant n\}$;

/*type(s_i)是语句 s_i 的类型*/

sc:$\langle s_m,s'_m\rangle$,其中,$s_m\in p$,s_m 施加一个变异算子得到 s'_m;

OUTPUT:

ts:$\{cs_j||cs_j|\in\{\text{true,flase}\}\wedge vars(cs_j)\subseteq paras(p)\}$.

/*$|cs_j|$表示约束条件 cs_j 的值,$vars(cs_j)$表示约束条件 cs_j 中出现的变量集合,$paras(p)$表示程序 p 中的输入参数集合.*/

PROCEDURE

1. 将 ts 设置为空,设置起始语句 s_{init}、结束语句 s_{final} 和输入参数集合 $paras(p)$.
2. 调用过程 $Construct_PDG(p)$得到 p 的程序依赖图PDG.
3. 在PDG上映射起始语句 s_{init}、故障语句 s'_m 和结束语句 s_{final},分别得到结点 N_{init},N_m 和 N_{final}.
4. 调用过程 $Construct_Slicing(PDG,N_m)$得到程序切片 $slice$.对于每个语句 $s_i\in slice$,执行
$$ts\leftarrow ts\wedge Construct\_PathCondition(s_i)$$
5. 调用过程 $Get_Offset(s_m,s'_m)$得到偏移量约束条件 $offset$,执行 $ts\leftarrow ts\wedge offset$.
6. 调用过程 $Construct_Chopping(PDG,N_m,N_{final})$得到程序劈片 $chop$,对于每个语句 $s_i\in chop$,执行
$$ts\leftarrow ts\wedge Construct\_PathCondition(s_i)$$
7. 返回 ts.

END

借助预处理,算法1可以应用到一般的C程序.预处理包括两个步骤(与文献[25]中提到的程序切片执行相似):

(1) 在每个函数调用的入口处,将函数体以内联的方式嵌入到程序中,得到只包含一个函数的等价C程序;

(2) 采用if语句和goto语句重写C程序中的所有循环.

预处理后的C程序只包含单一的函数,且只含有赋值、分支、跳转和返回这4种类型的语句.

算法1中,我们假设与原始程序相比,变异体程序中只含有一个变异算子(或一个故障语句).可以将这一算法拓展用来构造含有多个变异算子的测试约束.原始程序 ρ 经过一系列的变异算子得到一个非等价变异体 f,每次变异算子导致的故障分别记为 $\alpha_1,\alpha_2,\ldots,\alpha_n$,通过应用算法1得到相应的测试约束为 $\Gamma(\alpha_1),\Gamma(\alpha_2),\ldots,\Gamma(\alpha_n)$.那么,

© 中国科学院软件研究所 http://www.jos.org.cn

这些故障(即复合的变异算子)的测试约束$\Gamma(\alpha_1\cup\alpha_2\cup\ldots\cup\alpha_n)=\Gamma(\alpha_1)\vee\Gamma(\alpha_2)\vee\ldots\vee\Gamma(\alpha_n)$.

3 基于测试约束的软件失效域特征分析

软件失效域的特征是影响适应性随机测试性能的基本因素[15,16],这些特征包括软件失效率、失效域的形状、失效域的个数.本节讨论基于测试约束的软件失效域的特征分析方法.

故障α的测试约束$\Gamma(\alpha)$是一组定义在输入变量上的约束条件.连接这些约束条件的操作符包括析取($\vee$)、合取($\wedge$)、非(!)和括号.例如,图 2 中示意的故障的测试约束为 $preflag=0\wedge postflag=1\wedge own\_alt=other\_alt$.通过分配律和交换律,可以将一个任意形式的测试约束$\Gamma(\alpha)$变换为等价的析取范式$\Gamma'(\alpha)$.$\Gamma'(\alpha)$中的每个项就是一个可行的测试用例方案.也就是说,满足该项约束的测试数据可以检测出故障α.项中的每个文字通常定义为一个关系表达式.文字进一步区分为原子约束条件和复合约束条件.

定义 11(原子约束条件). 给定一个程序ρ,v是ρ的一个输入变量,故障α对应的测试约束为$\Gamma(\alpha)$且为析取范式,$\Gamma(\alpha)$的某个项中只有一个文字t中出现了v,则将t称为v的原子约束条件.

定义 12(复合约束条件). 给定一个程序ρ,v是ρ的一个输入变量,故障α对应的测试约束为$\Gamma(\alpha)$且为析取范式,$\Gamma(\alpha)$的某个项中文字$t_1,t_2,\ldots,t_n$中出现了v,则将$t_1\wedge t_2\wedge\ldots\wedge t_n$称为$v$的复合约束条件.

在图 1 所示的故障测试约束中,原子约束条件包括 $x \bmod 2=0$ 和 $y \bmod 2=0$(如图 1(a)所示),复合约束条件包括 $4<x-y\wedge x-y<6$(如图 1(b)所示),$4<x\wedge x<6$ 和 $4<y\wedge y<6$(如图 1(c)所示).

3.1 软件失效率分析

从软件输入域的角度来看,程序ρ的软件失效率γ可以定义为$\gamma=\frac{\|\varphi\|}{\|\psi\|}$,其中,$\|\psi\|$表示程序$\rho$的输入域$\psi$中所有测试数据的数目,$\|\varphi\|$表示程序$\rho$的软件失效域$\varphi$中的测试数据的数目.这里,我们假设每个输入值出现的概率是等同的.故障α的软件失效率γ越大,说明α对应的测试约束$\Gamma(\alpha)$中限制的测试数据的数目越多.因此,α的软件失效率表明了该故障被检测的难易程度.

当被测程序只有一个输入变量时,故障α测试约束$\Gamma(\alpha)$即为该变量的原子或复合约束条件.例如,程序ρ只有一个输入变量 X,X 为整数且其取值范围为[1…10000],某个故障α的测试约束$\Gamma(\alpha)$为 $4<x\wedge x<6$,则α导致的软件失效率γ应为 0.0001.

相似地,基于测试约束可以分析被测程序有 2 个或 3 个输入变量时的软件失效率.例如,图 1(c)中示意的故障α的测试约束$\Gamma(\alpha)$为 $4<x\wedge x<6$ 和 $4<y\wedge y<6$,不妨设变量 X 和 Y 的取值范围为[0…100],则α导致的软件失效率γ应为 0.0004(即$(6-4)\times(6-4)/(100\times100)$).

被测程序往往具有多个输入变量(即多维输入域),为此提出基于测试约束的软件失效率一般分析方法.假设被测程序ρ为数值计算类型;故障α的测试约束$\Gamma(\alpha)$为析取范式,即$\Gamma(\alpha)=T_1\vee T_2\vee\ldots\vee T_m$,其中,$T_1,T_2,\ldots,T_m$ 为$\Gamma(\alpha)$中的项,$m\geqslant1$;ρ的输入变量 $V_1,V_2,\ldots,V_n$($n>0$).按照如下步骤计算α导致的软件失效率γ:

(1) 计算$\|\psi\|=\prod_{i=1}^{n}|V_i|$.其中,$|V_i|$表示变量 V_i 的值域,其计算规则如下:

$$|V_i|=\begin{cases}v_i^u-v_i^d, & \text{其中,}v_i^u\text{和}v_i^d\text{为变量}V_i\text{的上界和下界;当}V_i\text{为实型数时,}\\ v_i^u-v_i^d+1, & \text{其中,}v_i^u\text{和}v_i^d\text{为变量}V_i\text{的上确界和下确界;当}V_i\text{为整型数时,}\\ \text{可取值的数目,} & \text{当}V_i\text{为布尔或枚举数时.}\end{cases}$$

(2) 依据$\Gamma(\alpha)$中的每个项 T_j($j=1,\ldots,m$)计算相应的软件失效域φ_j中的约束值域:

$$\|\varphi_j\|=\prod_{i\in a}|V_i|\times\prod_{i\in b}\lambda_i|V_i|\times\prod_{i,l\in c}\lambda_{i,l}|V_i|\times|V_l|,$$

其中,$|V_i|$的计算规则与步骤(1)相同,$a=\{i|T_j$ 中不存在变量 V_i 的原子约束条件或复合约束条件$\wedge i\in\{1,\ldots,n\}\}$,$c=\{i|T_j$ 中存在变量 V_i 的原子约束条件或复合约束条件 $tc\wedge tc$ 中出现变量 $V_k\wedge V_k\neq V_i\wedge i\in\{1,\ldots,n\}\}$,$b=\{1,\ldots,n\}-a-c$,$\lambda_i$ 表示变量 V_i 满足其原子/复合约束条件的值域与$|V_i|$的比值,$\lambda_{i,l}$ 表示变量 V_i

© 中国科学院软件研究所 http://www.jos.org.cn

和 V_l 满足其原子/复合约束条件的值域与$|V_i|\times|V_l|$的比值;

(3) 计算软件失效域φ中的约束值域$\|\varphi\|=\sum_{i=1}^{m}\|\varphi_i\|$;

(4) $\gamma=\frac{\|\varphi\|}{\|\psi\|}$.

3.2 软件失效域的群束性分析

给定程序ρ,故障α的测试约束$\Gamma(\alpha)$限定了软件失效域φ,φ中的测试数据分布于ρ的输入域中,并呈现出一些特征.例如,当被测程序具有两个输入变量时,软件失效域可能表现为特定的形状,如分散的点、带或者块.当输入的被测程序具有 3 个以上的输入变量(即多维输入域)时,则无法用具体的形状描述软件失效域.一般认为,点模式的软件失效域是非群束的,而带模式和块模式的软件失效域为群束的.对于高维输入域,我们将讨论是否是群束的[3,4,6,7].

定义 13(连续的约束值域). 给定一个程序ρ,v是ρ的一个输入变量,故障α对应的测试约束为$\Gamma(\alpha)$且为析取范式,即$\Gamma(\alpha)=T_1\vee T_2\vee\ldots\vee T_m$,其中,$T_1,T_2,\ldots,T_m$ 为$\Gamma(\alpha)$中的项,$m\geqslant 1$.给定$\Gamma(\alpha)$中的一个项 $T_i(i\in\{1,\ldots,m\})$,$\overline{\overline{v}}$ 表示满足 T_i 中 v的原子约束条件或复合约束条件的值域.当满足下列条件之一时,则称 $\overline{\overline{v}}$ 为连续的约束值域.

(1) 当v为实型数时,$v^u\neq v^d$,其中,v^u 和 v^d 为 $\overline{\overline{v}}$ 的上界与下界;

(2) 当v为整型数时,$\overline{\overline{v}}$ 中存在至少两个连续的整数值.

基于测试约束,可以采用如下方法分析软件失效域φ是否群束:给定故障α的测试约束为$\Gamma(\alpha)$,且$\Gamma(\alpha)$为析取范式,T_i 是$\Gamma(\alpha)$中的一个项,T_i 的软件失效域φ_i 是群束的,当且仅当存在一个 $V_j(j\in\{1,...,n\})$,且 T_i 中 V_j 具有连续的约束值域.

我们可以增强群束性的判定条件.例如,给定故障α的测试约束为$\Gamma(\alpha)$,且$\Gamma(\alpha)$为析取范式,T_i 是$\Gamma(\alpha)$中的一个项,T_i 的软件失效域φ_i 是群束的,当且仅当所有的 $V_j(j\in\{1,\ldots,n\})$,且 T_i 中 V_j 具有连续的约束值域.

投影技术可以扩展基于测试约束的软件失效域特征分析的应用范围和方式.当被测程序具有多维输入域时,仍可基于测试约束分析软件失效域在指定维上的群束性或者模式特征.图 2 示意的故障测试约束为 *preflag*=0∧*postflag*=1∧*own_alt*=*other_alt*,由于 *preflag* 和 *postflag* 是布尔变量类型,不存在群束特性,当运用投影技术时,只须忽略测试约束中 *preflag* 和 *postflag* 的原子约束条件或复合约束条件.不难分析,软件失效域在 *own_alt* 和 *other_alt* 维上具有群束性特征.

3.3 软件失效域的单域/多域分析

软件失效域的单域/多域分析是指判定故障α对应的软件失效域φ在输入域ψ中是一个还是多个区域.基于测试约束的软件失效域的单域/多域分析如下:

“给定一个程序ρ,故障α对应的测试约束为$\Gamma(\alpha)$且为析取范式,即$\Gamma(\alpha)=T_1\vee T_2\vee\ldots\vee T_m$,如果 $m>1$,则α的软件失效域φ为多域;否则,α的软件失效域φ为单域.”

当α的软件失效域φ为多域时,$\varphi_j(j=1,\ldots,m)$是 T_j 相应的软件失效域,$\varphi=\varphi_1\cup\varphi_2\cup\ldots\cup\varphi_m$.采用第 3.1 节和第 3.2 节的分析方法进一步分析φ中不同域φ_j 的失效率和群束性.软件失效域的多域特征分析有助于人们进一步了解成功的测试数据在输入域上的分布.例如,$\frac{\|\varphi_j\|}{\|\varphi\|=\sum_{i=1}^{m}\|\varphi_i\|}$ 反映了φ_j 在φ所占的比例.其中,$\|\varphi_j\|$表示失效域φ_j 的约束值域,该比例越大,则说明属于失效域φ_j 中能够检测出故障α的测试用例数越多.

基于测试约束还可以进行软件失效域的其他特征分析,如约束强度分析、包含关系分析等.其中,软件失效域的约束强度分析可以用来解释软件测试过程中为何一些难以捉摸的故障检测起来非常困难这一情况[26].

4 案例研究

我们用一个实际程序示例测试约束的概念及其构造方法,并基于测试约束分析该程序中各种故障的软件

© 中国科学院软件研究所 http://www.jos.org.cn

失效域的特征.TCAS 是欧洲宇航局飞机导航系统中的一个程序,完成避免飞机碰撞的主要功能.TCAS 为 C 程序,包含 9 个模块和 138 行可执行代码(http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/tcas.tar.gz).TCAS 有 12 个输入参数,其中 5 个为布尔型,7 个为整型.选择 TCAS 作为案例研究的对象,是因为该程序被广泛用于软件测试方面的经验研究且具有丰富的测试资源[27,28].在 TCAS 的原始版本中,人工植入各种类型故障得到 41 个错误版本(即变异体).每个错误版本是通过对原始版本中的一行或多行代码实施变异算子获得的.来自西门子有经验的软件工程师设计了这些故障,因此可以认为这些类型的故障是工程实践中常见的.

作为示例,我们随机选取第 6 个故障,简要讨论其测试约束的构造过程.与 TCAS 原始版本相比,包含该故障 α的错误版本 f 在函数 *Own_Below_Threat*(·)中(源程序的第 104 行处)存在一个操作符变异,即将关系符“<”错误地写成“<=”.故障α涉及到输入参数 *Other_Tracked_Alt* 和 *Own_Tracked_Alt*,进行故障敏感的偏移量约束条件分析得到 *Offset*(α)为

$$Other\_Tracked\_Alt=Own\_Tracked\_Alt \quad (\#1)$$

为了获取α的故障触发前提条件 *Pre-Condition*(α),必须分析故障α所在的函数 *Own_Below_Threat*(·)何时被调用.通过分析 TCAS 的程序依赖图、函数顺序调用图以及各个函数的控制流程图,不难发现如下函数调用链:*main*(·)→*alt_sep_test*(·)→*Own_Below_Threat*(·),*main*(·)→*alt_sep_test*(·)→*Non_Crossing-Biased_Climb*(·)→*Own_Below_Threat*(·)和 *main*(·)→*alt_sep_test*(·)→*Non_Crossing_Biased_Descend*(·)→*Own_Below_Threat*(·).

为了便于分析,采用预处理程序将所有的函数展开,从而得到只含有一个模块的被测程序.

利用后向程序切片分析技术,确定影响故障语句 s 的语句集合 *slicing*(s),确定 *slicing*(s)中各条语句执行的路径条件 *path-condition*(*slicing*(s)),最终计算得到 *Pre-Condition*(α)为

$$High\_Confidence=1 \quad (\#2)$$

$$Own\_Tracked\_Alt\_Rate<=600 \quad (\#3)$$

$$Cur\_Vertical\_Sep>600 \quad (\#4)$$

$$Other\_Capacity!=1\vee(Other\_Capacity=1\wedge Two\_of\_Three\_Reports\_Valid=1\wedge Other\_RAC=0) \quad (\#5)$$

相似地,利用程序劈片分析技术,确定故障语句 s 到输出语句或结束语句 t 的程序劈片 *chopping*(s,t),确定 *chopping*(s,t)中各条语句执行的路径条件 *path-condition*(*chopping*(s,t)),其结果如下:

$$(Down\_Separation<400\wedge Alt\_Layer\_Value=0)\vee(Down\_Separation<500\wedge Alt\_Layer\_Value=1)\vee$$
$$(Down\_Separation<640\wedge Alt\_Layer\_Value=2)\vee(Down\_Separation<740\wedge Alt\_Layer\_Value=3) \quad (\#6)$$

$$(Climb\_Inhibit=1\wedge Up\_Separation+100>Down\_Separation)\vee$$
$$(Climb\_Inhibit=0\wedge Up\_Separation>Down\_Separation) \quad (\#7)$$

约束(#1)~约束(#7)的合取得到了上述故障α的测试约束.为了分析软件失效域的特征,需要对测试约束进一步地变换,得到测试约束的析取范式.测试约束的析取范式$\Gamma(\alpha)$由 16 个布尔项组成,每个布尔项定义了一个软件失效域.其中,第 1 个布尔项 T_1 为

$$Other\_Tracked\_Alt=Own\_Tracked\_Alt\wedge Down\_Separation<400\wedge Up\_Separation+100>Down\_Separation\wedge$$
$$Alt\_Layer\_Value=0\wedge Climb\_Inhibit=1\wedge High\_Confidence=1\wedge Own\_Tracked\_Alt\_Rate<=600\wedge$$
$$Cur\_Vertical\_Sep>600\wedge Other\_Capacity!=1$$

我们以 T_1 为例,进行相应的软件失效域φ_1 的特征分析如下:

(1) φ_1 失效率分析

首先,依据输入变量的有效范围和第 3.1 节讨论的输入变量的值域计算规则,计算出各个输入变量的值域.例如,*Other_Tracked_Alt* 的值域|*Other_Tracked_Alt*|=8249,*High_Confidence* 的值域|*High_Confidence*|=2.

其次,计算不同输入变量的约束值域.根据第 3.1 节的步骤(2),将输入变量分为如下 3 类:

- 由于 T_1 不存在变量 *Two_of_Three_Reports_Valid* 和 *Other_RAC* 的原子约束条件或复合约束条件,这两个变量的约束值域等于输入变量的值域;
- T_1 中变量 *Alt_Layer_Value*,*Climb_Inhibit*,*High_Confidence*,*Own_Tracked_Alt_Rate*,*Cur_Vertical_Sep*,

© 中国科学院软件研究所 http://www.jos.org.cn

Other_Capacity 都只有原子约束条件,且不含其他变量,它们的约束值域分别为|*Alt_Layer_Value*|/4,|*Climb_Inhibit*|/2,|*High_Confidence*|/2,|*Own_Tracked_Alt_Rate*|×601/997,|*Cur_Vertical_Sep*|×1334/1935 和|*Other_Capacity*|/2;

- T_1 中变量 *Other_Tracked_Alt* 和 *Own_Tracked_Alt* 的原子约束条件出现了其他输入变量,它们对应于约束值域为 min(|*Other_Tracked_Alt*|,|*Own_Tracked_Alt*|).T_1 中变量 *Down_Separation* 的复合约束条件中出现了变量 *Up_Separation*,它们对应的约束值域为|*Up_Separation*|×400−301×300/2.

最后,可以计算出φ_1的软件失效率$\gamma_1=\frac{\|\varphi_1\|}{\|\Psi\|}\approx 4.47\times10^{-7}$.这里假设各种输入变量的取值都是均匀分布的.

(2) φ_1的群束性分析

T_1 中的变量 *Other_Tracked_Alt*,*Own_Tracked_Alt*,*Down_Separation*,*Up_Separation*,*Own_Tracked_Alt_Rate*,*Cur_Vertical_Sep* 均有连续的约束值域,因此,软件失效域φ_1是群束的.采用投影技术,可以基于 T_1 分析软件失效域φ_1在指定维上的一些特征.例如,结合 T_1 中变量 *Down_Separation* 的复合约束条件(即 *Down_Separation*<400∧*Up_Separation*+100>*Down_Separation*)和变量 *Up_Separation* 的原子约束条件(即 Up_*Separation*+100>*Down_Separation*),以及变量的有效值域,不难分析出φ_1在 *Down_Separation* 和 *Up_Separation* 这两维上的分布特征,如图 3 所示.

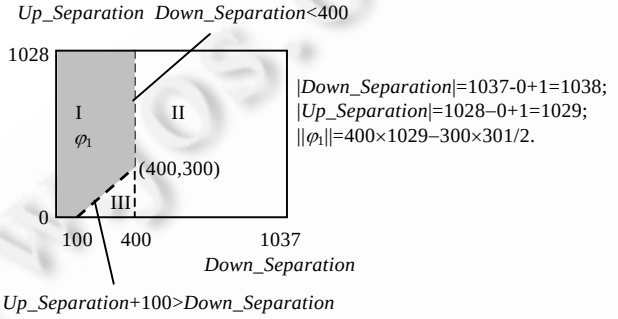


Fig.3 An illustration of failure-causing regions constrained by *Down_Separation* and *Up_Separation*

图 3 失效域在 *Down_Separation* 和 *Up_Separation* 上的分布特征示意图

(3) φ的单域/多域分析

由于α的测试约束的析取范式$\Gamma(\alpha)$有 16 个项,因此α对应的软件失效域φ是多域的.采用与φ_1类似的分析过程,基于$\Gamma(\alpha)$中的布尔项$T_j(j=2,\ldots,16)$对其相应的失效域$\varphi_j(j=2,\ldots,16)$进行软件失效率和群束性特征分析.表 1 总结了故障α的软件失效域的特征分析结果,其中,φ_j 的软件失效率"百分比%"反映了φ_j 在φ所占的比例,可计算为$\frac{\gamma_j}{\sum_{i=1}^{16}\gamma_i}$或$\frac{\|\varphi_j\|}{\sum_{i=1}^{16}\|\varphi_i\|}$.其中,$\|\varphi_j\|$表示失效域$\varphi_j$的约束值域,$\gamma_j$表示$\varphi_j$的软件失效率.

采用与上述相似的过程,我们构造了 TCAS 其余 40 个错误版本的测试约束(限于篇幅,这里我们不再一一列出),得到 54 个不同的约束条件.后续错误版本的测试约束构造是一个加速过程.基于错误版本的测试约束,对各种故障的软件失效域进行特征分析.我们发现,绝大部分故障的软件失效率较小,相应的软件失效域为多域和群束的.软件失效域特征分析结果表明,当适应性随机测试方法应用于 TCAS 类似的程序时,可以获得较好的测试性能.尽管 TCAS 的规模较小,但具有多维的输入变量,展示了各种常见的故障类型,因此,该案例研究验证了本文提出的测试约束及其基于测试约束进行软件失效域特征分析方法的可行性和有效性.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 1 A summary of characteristic analysis of the fault in faulty Version 6

表 1 错误版本 6 中的故障的软件失效域特征分析

特征 / 失效域	软件失效率		群束特性	测试约束
	失效率	百分比(%)		
φ_1	4.47×10^{-7}	8.3	群束	项 1
φ_2	5.29×10^{-7}	9.8	群束	项 2
φ_3	6.25×10^{-7}	11.6	群束	项 3
φ_4	6.78×10^{-7}	12.6	群束	项 4
φ_5	4.04×10^{-7}	7.5	群束	项 5
φ_6	4.74×10^{-7}	8.8	群束	项 6
φ_7	5.53×10^{-7}	10.3	群束	项 7
φ_8	5.94×10^{-7}	11	群束	项 8
φ_9	1.12×10^{-7}	2.1	群束	项 9
φ_{10}	1.32×10^{-7}	2.5	群束	项 10
φ_{11}	1.56×10^{-7}	2.9	群束	项 11
φ_{12}	1.70×10^{-7}	3.2	群束	项 12
φ_{13}	1.01×10^{-7}	1.9	群束	项 13
φ_{14}	1.19×10^{-7}	2.2	群束	项 14
φ_{15}	1.38×10^{-7}	2.6	群束	项 15
φ_{16}	1.49×10^{-7}	2.8	群束	项 16
φ	5.38×10^{-6}	100	N/A	$I(\alpha)$

5 相关工作

程序切片回答了程序中潜在影响某个语句的语句集合[23],程序劈片解释了程序中一条传递依赖路径上源语句是如何影响目标语句[24],路径条件回答了程序中为什么源语句影响某个目标语句[21].程序切片和程序劈片都产生一个语句集合,路径条件则产生一组语句执行条件的集合,表达为布尔表达式.本文提出的测试约束揭示了程序中故障的触发与传播的内在机制,回答了起始语句为什么并且是如何影响错误语句、错误语句为什么并且是如何影响那些能产生不同观测结果的输出语句,最终产生一组定义在输入变量上的布尔表达式集合.测试约束的构造综合运用了程序切片、程序劈片和路径条件等程序分析技术.

近年来,人们提出了许多约束求解技术,并越来越多地应用于软件系统的测试或分析.在 DeMillo 和 Offutt 提出的变异分析技术中,基于路径表达式、谓词约束和必要性约束产生测试用例,必要性约束和谓词约束表示为析取范式[29].Morell 在研究基于错误的测试中,提出了错误发现条件包括创建条件和传播条件[30].在前期研究工作中,我们提出一种基于约束的回归测试用例选择策略[31].Voas 在研究软件可测试性时提出的 PIE 模型[32]包括故障感染分析、传播分析和执行分析,且是基于概率理论实现的.本文提出的测试约束概念与上述相关约束概念存在一些区别与联系.具体来说,测试约束的构造涉及到故障触发和传播的过程,触发条件对应于 PIE 模型的故障感染分析、变异分析中的必要性约束和基于错误测试中的创建条件.测试约束中的传播条件则包含了变异分析中谓词约束和基于错误测试中的传播条件.测试约束等价于在变异分析中变异体被杀死的必要性和充分性条件的交集.测试约束与上述相关约束在构造过程方面是相似的,但解决的问题有所不同.

通常从如下两个角度研究软件故障的特征:

(1) 基于程序代码,研究各种缺陷代码的结构特征.例如,变异分析中各种变异算子实际上是对常见缺陷代码结构的模仿[19],文献[33]归纳了 C 语言中常见的缺陷代码模式;

(2) 基于程序的输入域,研究软件失效域的分布特征.例如,Chen 等人在采用仿真实验的手段研究适应性随机测试的性能时,构造了 3 种失效模式的输入域[16,17].

本文提出的故障特征分析方法属于后者.与文献[7]相比,本文的研究对象并不局限于二维输入域的软件失效域;与文献[16,17]相比,本文不是采用仿真实验的方法,而是研究真实程序中的各种失效模式,并且提供了定量的分析方法.

White 和 Cohen[6]分析了数值型程序的各种类型的故障,发现源程序中的判定谓词容易出错,并导致执行不正确的计算路径.这样的故障导致连续的软件失效域.Ammann 和 Knight[3]也进行过类似的经验研究,他们选取

© 中国科学院软件研究所 http://www.jos.org.cn

了几个数值型程序,观测各种故障导致的失效分布情况,发现了这些故障也会导致局部连续的软件失效域.Bishop[4]对核反应堆控制函数中的故障进行了分析,得出了类似的结论,故障导致的软件失效域通常是群束的.Chen 等人[7]在对数值型程序中的各种故障的软件失效域进行分析的基础上,提出了失效模式的概念.现有的研究都证实了数值型程序中,故障导致的软件失效域是群束的[8].提出的测试约束提供了进一步刻画数值型程序中故障的失效域方法,基于测试约束可以分析失效域的各种特征(包括群束性特征).

6 结 论

本文从程序输入域的角度研究了软件失效域的特征,提出了一种软件失效域的识别与特征分析方法.该方法通过运用各种程序分析技术获得表达软件失效域的测试约束,测试约束是定义在输入变量上的一组布尔表达式,满足测试约束的所有测试用例的输入数据形成了软件失效域.基于测试约束可以进一步分析软件失效域的相关特征,包括软件失效率、失效域的群束性、失效域的多域/单域性等.将提出的软件失效域识别和分析方法应用于一个真实的程序 TCAS 及其多个错误版本,实例研究发现,绝大部分故障的软件失效率较小,失效域为多域和群束的.当适应性随机测试方法应用于这类程序时,可以获得较好的测试性能.

尽管测试约束提供了一种有效的软件失效域识别方法,有助于对软件失效域的各种特征进行形式化分析,但测试约束的构造涉及到大量的计算开销,不利于在实践中广泛运用.软件工程领域最近的一些进展为测试约束的构造提供了有效的技术和算法[34].我们将进一步采纳程序分析领域的最新研究成果(包括程序切片、符号执行等)和软件分析与验证领域中基于约束的新技术(包括使用约束求解器处理布尔型、整型、实数型、浮点型数据类型、枚举类型、控制结构、复杂的数据结构和方法调用等[35])以改进测试约束的构造过程.

致谢 作者感谢澳大利亚斯文本大学陈宗岳教授参与了本文部分研究工作的讨论.

References:

[1] Myers GJ. The Art of Software Testing. 2nd ed., New York: John Wiley and Sons, 2004. 10−20.

[2] Hamlet R. Random testing. In: Marciniak J, ed. Proc. of the Encyclopedia of Software Engineering. 2nd ed., New York: John Wiley and Sons, 2002.

[3] Ammann PE, Knight KC. Data diversity: An approach to software fault tolerance. IEEE Trans. on Computers, 1988,37(4):418−425. [doi: 10.1109/12.2185]

[4] Bishop PG. The variation of software survival times for different operational input profiles. In: David P, ed. Proc. of the 23rd Int'l Symp. on Fault-Tolerant Computing (FTCS-23). Los Alamitos: IEEE Computer Society Press, 1993. 98−107. [doi: 10.1109/FTCS.1993.627312]

[5] Finelli GB. NASA software failure characterization experiments. Reliability Engineering and System Safety, 1991,32(1-2):155−169. [doi: 10.1016/0951-8320(91)90045-9]

[6] White LJ, Cohen EI. A domain strategy for computer program testing. IEEE Trans. on Software Engineering, 1980,6(3):247−257. [doi: 10.1109/TSE.1980.234486]

[7] Chan FT, Chen TY, Mak IK, Yu YT. Proportional sampling strategy: Guidelines for software testing practitioners. Information and Software Technology, 1996,38(12):775−782. [doi: 10.1016/0950-5849(96)01103-2]

[8] Chen TY, Kuo F C, Merkel R, Tse TH. Adaptive random testing: The ART of test case diversity. Journal of Systems and Software, 2010,83(1):60−66. [doi: 10.1016/j.jss.2009.02.022]

[9] Chen TY, Leung H, Mak IK. Adaptive random testing. In: Maher MJ, ed. Proc. of the 9th Asian Computing Science Conf. LNCS 3321, Heidelburg: Springer-Verlag, 2004. 320−329. [doi: 10.1007/978-3-540-30502-6_23]

[10] Chen TY, Tse TH, Yu YT. Proportional sampling strategy: A compendium and some insights. Journal of Systems and Software, 2001,58(1):65−81. [doi: 10.1016/S0164-1212(01)00028-0]

[11] Chen TY, Kuo FC, Merkel RG, Ng SP. Mirror adaptive random testing. Information and Software Technology, 2004,46(15):1001−1010. [doi: 10.1016/j.infsof.2004.07.004]

© 中国科学院软件研究所 http://www.jos.org.cn

[12] Chan KP, Chen TY, Towey D. Restricted random testing. In: Kontio J, Conradi R, eds. Proc. of the 7th European Conf. on Software Quality (ECSQ 2002). LNCS 2349, Heidelburg: Springer-Verlag, 2003. 321−330. [doi: 10.1007/3-540-47984-8_35]

[13] Mayer J. Lattice-Based adaptive random testing. In: Ireland A, ed. Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2005). New York: ACM Press, 2005 .333−336. [doi: 10.1145/1101908.1101963]

[14] Mayer J. Adaptive random testing by bisection with restriction. In: Lau KK, Banach R, eds. Proc. of the 7th Int'l Conf. on Formal Engineering Methods (ICFEM 2005). LNCS 3785, Heidelburg: Springer-Verlag, 2005. 251−263. [doi: 10.1007/11576280_18]

[15] Chen TY, Merkel R. An upper bound on software testing effectiveness. ACM Trans. on Software Engineering and Methodology, 2008,17(3):1−27. [doi: 10.1145/1363102.1363107]

[16] Chen TY, Kuo FC, Zhou ZQ. On the relationships between the distribution of failure-causing inputs and effectiveness of adaptive random testing. In: Smith G, ed. Proc. of the 17th Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE 2005). Skokie: Knowledge Systems Institute Graduate School, 2005. 306−311.

[17] Chen TY, Kuo FC, Sun CA. The impact of the compactness of failure regions on the performance of adaptive random testing. Journal of Software, 2006,17(12):2438−2449 (in English with Chinese abstract). http://www.jos.org.cn/1000-9825/17/2438.htm [doi: 10.1360/jos172438]

[18] IEEE. IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE-Std-982.1, 1988.

[19] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 1978,11(4): 34−41. [doi: 10.1109/C-M.1978.218136]

[20] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? In: Roman GC, Griswold WG, Nuseibeh B. eds. Proc. of the 27th Int'l Conf. on Software Engineering (ICSE2005). Los Alamitos: IEEE Computer Society Press, 2005. 402−411. [doi: 10.1109/ICSE.2005.1553583]

[21] Krinke J. Slicing, chopping and path conditions with barriers. Software Quality Journal, 2004,12(4): 339−360. [doi: 10.1023/B: SQJO.0000039792.93414.a5]

[22] Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. ACM Trans. on Programming Languages and Systems, 1987,9(3):319−349. [doi: 10.1145/24039.24041]

[23] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984,10(4):352−357. [doi: 10.1109/TSE.1984.5010248]

[24] Jackson D, Rollins EJ. A new model of program dependences for reverse engineering. In: Adrion WR, Wile D, eds. Proc. of the 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE'94). New York: ACM Press, 1994. 2−10. [doi: 10.1145/ 195274.195281]

[25] Yi X, Wang J, Yang X. Verification of C programs using slicing execution. In: Cai KY, Ohnishi A, Lau MF, eds. Proc. of the 5th Int'l Conf. on Quality Software (QSIC 2005). Washington: IEEE Computer Society, 2005. 109−116. [doi: 10.1109/QSIC.2005.72]

[26] Barzin R, Fukushima S, Howden W, Sharifi S. Superfit combinational elusive bug detection. In: Leppänen W, ed. Proc. of the 32nd Annual IEEE Int'l Computer Software and Applications Conf. (COMPSAC 2008). Washington: IEEE Computer Society, 2008. 144−151. [doi: 10.1109/COMPSAC.2008.5]

[27] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control flow- based test adequacy criteria. In: Fadini B, Osterweil L, Lamsweerde A, eds. Proc. of the 16th Int'l Conf. on Software Engineering (ICSE'94). Los Alamitos: IEEE Computer Society Press, 1994. 191−200. [doi: 10.1109/ICSE.1994.296778]

[28] Weyuker E, Goradia T, Singh A. Automatically generating test data from a Boolean specification. IEEE Trans. on Software Engineering, 1994,20(3):353−363. [doi: 10.1109/32.286420]

[29] DeMillo RA, Offutt AJ. Constraint-Based automatic test data generation. IEEE Trans. on Software Engineering, 1991,17(9): 900−910. [doi: 10.1109/32.92910]

[30] Morell LJ. A theory of fault-based testing. IEEE Trans. on Software Engineering, 1990,16(8):844−857. [doi: 10.1109/32.57623]

[31] Sun CA. A constraint-oriented test suite reduction method for conservative regression testing. Journal of Software (Academy Publisher), 2011,6(2):314−321.

[32] Voas JM. PIE: A dynamic failure-based technique. IEEE Trans. on Software Engineering, 1992,18(8):717−727. [doi: 10. 1109/32.153381]

© 中国科学院软件研究所 http://www.jos.org.cn

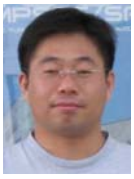
[33] Hu X, Liu B, Lu MY. Classification of software code errors and its application. Journal of Computer Engineering, 2009,35(2): 30−33 (in Chinese with English abstract).

[34] Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. In: Wong WE, ed. Proc. of the Mutation Testing for the New Century. Kluwer Academic Publishers, 2001. 34−44.

[35] Marinov D. Automatic testing of software with structurally complex inputs [Ph.D. Thesis]. Cambridge: Massachusetts Institute of Technology, 2005.

附中文参考文献:

[33] 胡璇,刘斌,陆民燕.软件代码缺陷分类及其应用.计算机工程,2009,35(2):30−33.



孙昌爱(1974－),男,江苏盐城人,博士,副教授,CCF 高级会员,主要研究领域为软件测试,程序分析,软件体系结构,服务计算.

2012 CCF 中国计算机大会

征 文 通 知

第 9 届 CCF 中国计算机大会(2012 CCF China National Computer Congress, CCF CNCC2012)将于 2012 年 10 月 18-20 日在大连世博广场举行,承办单位为大连大学.CCF CNCC 是由中国计算机学会 2003 年创建的系列性学术会议,已在不同的城市成功举办 8 届,现每年一次.

CCF CNCC 旨在探讨计算机及相关领域最新进展和宏观发展趋势,展示中国学术界、企业界最重要的学术、技术事件和成果,使不同领域的专业人士能够获得探讨的机会并获得所需信息.CCF CNCC2012 将有约 2000 人到会,有逾 100 项成果进行展览展示,是中国计算机界的又一次盛会.

CCF CNCC2012 现公开征集会议论文,征文范围涵盖计算机领域各专业.本次大会拟不出版论文集,不超过 50 篇的优秀论文将刊登在《计算机学报》上,其他所有大会入选论文也将发表在 CCF 会刊《小型微型计算机系统》和《计算机应用与软件》上.

投稿方式

此次将采取网上投稿方式,请作者将稿件通过以下地址提交:http://conf.ccf.org.cn/

如果是 CCF 会员,请在投稿时注明“CCF 会员”.

重要日期

录用通知发出日期: 2012 年 8 月 1 日

© 中国科学院软件研究所 http://www.jos.org.cn