结合 Craig 插值分析的软件错误诊断方法

徐 勇[1,2],毋国庆[1],袁梦霆[1]
(1. 武汉大学计算机学院,湖北武汉 430072;2. 广东肇庆学院数学与统计学院,广东肇庆 526061)

摘 要: 基于模型诊断(MBD)的理论应用到软件错误定位中取得了一定的效果. 但是经典 MBD 理论基于元件间独立地发生故障这一假设,导致软件错误定位的结果中存在假阳性的诊断. 论文对现有基于 MBD 的软件错误定位方法进行了改进,提出了冲突中元件的冗余分析方法. 该方法既包括了基于 Craig 插值的元件冗余分析机制,同时利用条件语句取值的二元性(真或假)的特点,对冲突中的条件语句元件进行软件错误的无相关分析. 实验结果表明:冲突中的元件冗余分析方法可以有效地减少诊断的假阳性率,将诊断结果数减少了 48.4%,碰集树生成的结点数减少了 47.6%.

关键词: 基于模型诊断; 软件错误定位; 冗余分析; Craig 插值
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2016)10-2514-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2016.10.033

Software Fault Localization Based on Model-Based Diagnosis Combined Craig Interpolant Analysis

XU Yong[1,2], WU Guo-qing[1], YUAN Meng-ting[1]
(1. *Computer School, Wuhan University, Wuhan, Hubei* 430072, *China*;
2. *School of Mathematics and Statistics, Zhaoqing University, Zhaoqing, Guangdong* 526061, *China*)

Abstract: Model-based diagnosis, an intelligent diagnosis theory has been successfully applied in software fault localization with promising results. However, traditional MBD relies on the assumption that components in the system fail dependently which makes the diagnoses with high false positives in software fault localization. In this paper, a component redundancy analysis approach is presented. The approach not only uses Craig interpolant to filter redundant components, but also employs a fact that a branch predicate evaluates to either true or false to filter some branch condition components. Experimental results show that the proposed approach effectively reduces the false positive rates of diagnoses, i.e., reducing the number of diagnosis by 48.4%, and reducing the number of nodes of hitting set tree generated during diagnosis computation by 47.6%.

Key words: model-based diagnosis (MBD); fault localization; redundancy analysis; Craig interpolant

1 引言

软件错误定位是软件调试的第一步,其任务是找出软件中引发软件失效的程序部分. 软件错误定位费时、费力. 研究开发自动或半自动的软件错误定位方法一直是学术界关注的热点课题.

近年来的文献资料显示,人们对于自动软件错误定位方法展开了大量而深入的研究[1-5]. 在这些研究工作中,其采用的技术方法主要可分以下几类:(1) 基于频谱的错误定位方法[1-2];(2) 基于模型诊断(Model-Based Diagnosis)的软件错误定位方法[3,4];(3) 基于概率统计方法的软件错误定位[5]. 当然,也有将值替换[6],状态比较[7],程序不变式[8],测试用例分析[9]等应用到软件错误定位中. 本文关注的是基于 MBD 的软件错误定位.

针对电器元件中的故障诊断问题,Retier[10] 提出 MBD 理论和诊断方法. 其基本思想是,用合适的逻辑(例如一阶逻辑等)构造系统的行为和结构模型,通过

收稿日期:2015-11-03;修回日期:2016-02-16;责任编辑:孙瑶
基金项目:国家自然科学基金(No. 91118003, No. 61003071);深圳战略性新兴产业发展专项资金(No. JCYJ20120616135936123);中央高校基本科研业务费专项资金(No. 3101046, No. 201121102020006)

系统运行行为与系统预期行为之间的不一致的观察,结合逻辑的推理功能推导出哪个或哪些元件引发了系统工作异常;诊断的计算依赖于冲突集及碰集树(Hit Set Tree)的构造,每一个诊断都是冲突集的最小碰集.如果将程序中的语句视为 MBD 中的元件,可以将 MBD 应用于软件的错误定位[11-13].但是程序中的语句之间往往存在着数据及控制依赖关系,而 Reiter 在其 MBD 理论中假设了系统中的元件独立地发生功能故障.因此,直接应用 MBD 到软件的错误定位而不考虑元件之间存在的依赖关系将会影响错误诊断的准确性,即候选诊断中存在假阳性.

Birgit Hofer[13]等人提出应用程序切片获得语句间的依赖关系,并与 MBD 相结合提高诊断结果的质量.Jorg Weber[14]等人则提出增加元件依赖关系模型来表达元件之间可能发生故障的关系,从而支持对元件依赖性故障和独立性故障的诊断.MBD 中冲突的存在是系统中的元件集导致系统的行为模型与实际行为的不一致,也即是逻辑上的不可满足.Craig 插值提供了一种更简单的形式展示不可满足公式的原因[15].借助 Craig 插值,可以分析出冲突中的元件在系统行为模型逻辑上不可满足的作用,从而识别出冗余元件.与此同时,程序中条件语句的取值无非是真和假.而给定一个具体失败输入,其只可能执行程序的某一条路径.如果冲突中的条件语句元件无论是取真或假值,系统的合法行为模型仍不可满足,则表明冲突中的条件语句元件与当前失败输入所引发的软件失效是不相关的.事实上,文献中既有利用 Craig 插值对不可满足的逻辑公式进行化简分析的成功应用[16-17],也有利用分支条件语句取值二元性的特点进行程序错误定位[18-19],这些都启发我们在 MBD 框架下存在进行冲突中元件不相关分析的可能性.

2　基于 MBD 的软件错误定位

2.1　MBD 的基本定义[10]

定义 1　一个诊断系统 DS 是三元组(SD, CMP, OBS),其中 SD 表示系统描述;CMP 是有限整数集合,表示系统中的元件;OBS 表示系统行为的观察.其中 SD 和 OBS 一般用一阶句子集表示.

定义 2　对某个元件 $c(c\in \mathrm{CMP})$,谓词 $AB(c)$ 表示其异常工作,而 $\neg AB(c)$ 则表示其正常工作.

定义 3　一个组件集 C 是一个冲突当且仅当 $C\subseteq \mathrm{CMP}$ 且 $\{\mathrm{SD}\cup\mathrm{OBS}\}\cup\{\neg AB(c)\mid c\in C\}$ 是不可满足的.

定义 4　一个组件集 Δ 是一个诊断当且仅当 $\Delta\subseteq \mathrm{CMP}$ 且 $\{\mathrm{SD}\cup\mathrm{OBS}\}\cup\{\neg AB(c)\mid c\in \mathrm{CMP}\setminus\Delta\}$ 是一致的.

更进一步,Reiter 提出诊断计算方法,即有如下定理:

定理 1　一个组件集 A 是 DS 的一个诊断当且仅当 A 是 DS 所有冲突集的一个最小碰集.

2.2　基于 MBD 的软件错误定位

Robert Konighofer[3]等开发了一款基于 MBD 的程序错误定位和修复工具 Forensic.Forensic 以运行失效的程序及其规约作为输入,其中程序的规约既可以是程序断言,也可以是正确的参考程序,并且将程序中赋值语句的右值和条件语句视为元件.假设原失效程序为 P,Forensic 的错误定位由以下几个步骤来完成:

(1)程序预处理.用特殊函数 cmp 替换 P 中赋值语句的右值和条件语句,替换后的程序称 $\tilde{P}$.每个 cmp 函数表示一个元件,它的语义是当该元件正确时,则返回其原值表达式,否则返回一个修复符号 r,代表未知值.另外,Forensic 定义了两个映射函数,即 CmpOf 函数,将一个修复符号映射到产生该修复符号的元件编号.而 Org 函数则是将一个修复符号映射到其原来的值.

(2)程序的模型构造.通过符号执行收集程序 $\tilde{P}$ 的语义,即路径条件.这些路径条件分为两类,即不违反 $\tilde{P}$ 中的断言为正确执行路径,反之为错误执行路径.所有来自正确执行路径的路径条件集合(记为 PASS)就构成了 P 的合法行为模型,形式化地表示如下:

$$\pi[\boldsymbol{i}\parallel\boldsymbol{r}]=\bigvee_{p\in \mathrm{PASS}} PC^{p}[\boldsymbol{i}\parallel\boldsymbol{r}] \tag{1}$$

其中 $\boldsymbol{i}$ 表示输入符号向量,$\boldsymbol{r}$ 是由 cmp 函数返回的一组不确定值,也称为修复符号向量.同时,在程序分析过程中,无论正确或失效的执行路径,对路径条件用约束求解得到一组输入值.

(3)诊断的计算.冲突的计算依赖 Repair 函数,其定义如下.

$$\mathrm{Repair}(Q)\Leftrightarrow\forall \boldsymbol{i}.\ \exists \boldsymbol{r}.\ \pi[\boldsymbol{i}\parallel\boldsymbol{r}]\bigwedge_{r\in R} r=\mathrm{Org}(r)[\boldsymbol{i}\parallel\boldsymbol{r}] \tag{2}$$

其中,$R=\{r\mid \mathrm{CmpOf}(r)\in Q\}$,$Q$ 是元件的集合.

定义 5　元件集 Δ 是 P 的一个诊断当且仅当 $\mathrm{Repair}(\mathrm{CMP}\setminus\Delta)=\mathrm{true}$;元件集 $C(C\subseteq \mathrm{CMP})$ 是 P 的一个冲突当且仅当 $\mathrm{Repair}(C)=\mathrm{false}$.

由于 Repair 函数中存在的量词更替导致计算困难,因此,Forensic 使用一组具体的输入值代替输入符号向量进行 Repair 函数的计算.即在具体的实现中,使用 Repair′ 代替 Repair 函数(其中 J 是一组具体失败输入).

$$\mathrm{Repair}'(Q)\Leftrightarrow\bigwedge_{v_i\in J}\exists \boldsymbol{r}.\ \pi[\boldsymbol{v}_i\parallel\boldsymbol{r}]\bigwedge_{r\in R} r=\mathrm{Org}(r)[\boldsymbol{v}_i\parallel\boldsymbol{r}] \tag{3}$$

3　启发式实例

图 1 中的程序 P 是运行失效的,其中第 4 行的语句是错误的,应该是“res = y”.Forensic 首先通过程序预处

理识别出所有的元件集 CMP = {0,1,2,4,5,6},元件 1,4 是条件语句,而其他元件属于赋值语句的右值. 然后,通过符号执行得到 $\pi[\boldsymbol{i} \| \boldsymbol{r}]$(表 1)和一失败输入 $\beta = \{x=0, y=1, z=0\}$. 表 2 列出了修复符号与其元件间的对应关系. 最后,Forensic 计算诊断即是构造碰集树的过程. Forensic 首先从给定的输入 β 利用 Repair′ 计算出一个冲突 $C_1 = \{1,2,3,4,6\}$(图 2 中的结点 n_0). 根据冲突 C_1 中的元件,以宽度优先,从左到右的顺序扩展分支和结点,最终得到一个指定深度的碰集树(图 2 所示). 碰集树中的结点分为三类:(1) 用√ 标记的结点,它表示计算出的一个诊断;(2) 用 × 标记的结点为剪枝结点;(3) 内部结点,对应一个冲突.

```
1.    int max (int x,int y){
2.      int res=x;   // c0
3.      if (y>x)     //c1
4.        res=x;   // res=y; c2
5.      return res;
6.    }
7.    int max3(int x, int y, int z) {
8.      int res;
9.      int two=max (x,y);    //c3
10.     if (two>z)          // c4
11.       res=two;          // c5
12.     else
13.     res=z;              // c6
14.   assert(res>=x && res>=y && res>=z);
15.   return res;
16.   }
```

图1 一个运行失效的程序

表 1 路径条件约束 $\pi[\boldsymbol{i} \| \boldsymbol{r}]$

$\pi[k]$	k
$s_{15} \geqslant s_{02} \wedge s_{15} \geqslant s_{01} \wedge s_{15} \geqslant s_{00} \wedge s_{14} = 0 \wedge s_{07} = 0$	0
$s_{16} \geqslant s_{02} \wedge s_{16} \geqslant s_{01} \wedge s_{16} \geqslant s_{00} \wedge s_{14} ! = 0 \wedge s_{07} = 0$	1
$s_{11} \geqslant s_{02} \wedge s_{11} \geqslant s_{01} \wedge s_{11} \geqslant s_{00} \wedge s_{10} = 0 \wedge s_{07} ! = 0$	2
$s_{12} \geqslant s_{02} \wedge s_{12} \geqslant s_{01} \wedge s_{12} \geqslant s_{00} \wedge s_{10} ! = 0 \wedge s_{07} ! = 0$	3

表 2 修复符号与元件之间的对应关系

修复符号	CmpOf(r)	Org(r)
s_{06}	0	x
s_{07}	1	$y > x$
s_{08}	2	x
s_{09}, s_{13}	3	$\max(x,y)$
s_{10}, s_{14}	4	two > z
s_{11}, s_{15}	5	two
s_{12}, s_{16}	6	z

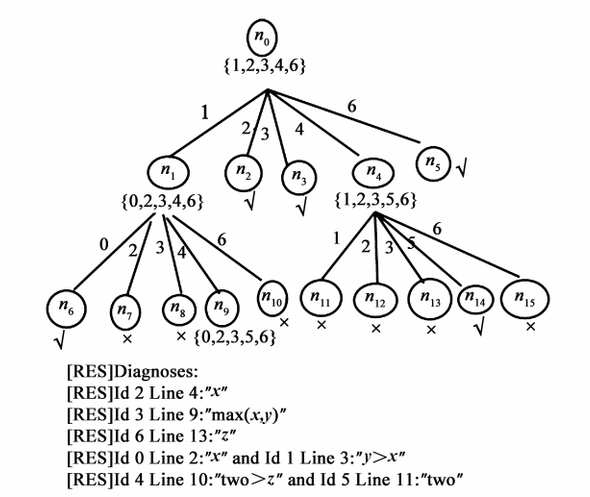


图2 碰集树构造过程和诊断结果

图 2 中给出最后诊断结果,即 5 个诊断,其中 3 个诊断的基是 1,2 个诊断的基是 2. 显然,除了第 1 个诊断以外,其他的诊断都属于假阳性. 如本文引言所述,程序中的语句间往往存在数据与控制流依赖的,而 MBD 中诊断的计算是依赖于冲突的概念. 但是冲突的概念并没有体现冲突中元件之间的依赖关系. 根据定义 5,冲突的存在即意味着公式不可满足. 而 Craig 插值提供了一种更简单的形式展示两个公式不可满足的原因. 因此存在着可能利用 Craig 插值表示形式去识别冲突中不相关的元件. 例如,对于表 1 中第 3 条路径条件($k=3$),对于冲突 C_1 的两次分割点{{1,2},{3,4,6}},{{1,2,3},{4,6}},其对应的 Craig 插值是 $s_{08} \leqslant 0, s_{09} \leqslant 0$,而这两个逻辑公式是语义等价的. 这表明,元件 3 的执行与否不影响该条路径条件的不可满足性,即元件 3 是冗余的. 图 1 中程序也反映到这一点,给定失败输入 β,程序中第 9 行的赋值语句的值实际是来自第 3 行语句的值.

另一方面,程序中条件语句的取值非真即假. 如果当某个条件语句无论取真或取假值都无法消除软件的失效,则可以认为该条件语句对于该软件运行失效是无相关. 例如,对于表 1 中的 $\pi[3]$,元件 4 属于条件语句,所对应的分支约束条件是 two > z. 那么当前的失败输入 β 无论元件 4 取真或假,$\pi[\boldsymbol{i} \| \boldsymbol{r}]$ 仍然不可满足,即程序运行失效仍旧存在. 因此,元件 4 是与当前失败输入软件失效无关的元件.

4 MBD 的冲突中元件冗余分析

4.1 基于冲突的归纳 Craig 插值计算

定义 6 Craig 插值[15]. 给定公式 φ_1 和 φ_2,如果 $\varphi_1 \wedge \varphi_2$ 是不可满足,则 φ_1 和 φ_2 的插值 ψ 有以下性质:(1) $\varphi_1 \models \psi$,(2) $\psi \wedge \varphi_2$ 是不可满足的,(3) ψ 中的符号

仅与 φ_1 和 φ_2 中的符号有关.

为了叙述方便,引入一些符号. 用 Interp(φ_1,φ_2)表示逻辑公式 φ_1,φ_2 的 Craig 插值. 用 J 表示测试用例集作为输入来进行错误诊断,且$|J|=1$. 用 $\pi[k]$表示 $\pi[\boldsymbol{i}\|\boldsymbol{r}]$中某条路径条件,显然,若路径条件集合 PASS 且 $n=|\text{PASS}|$,则 $k=\{0,1,\cdots,n-1\}$. 表 1 列出了图 1 中的程序所有合法的路径条件即 $\pi[\boldsymbol{i}\|\boldsymbol{r}]$.

冲突是元件的集合,为了对冲突中的元件进行冗余分析,必须先将冲突映射成路径条件中的逻辑公式. 根据式(3),给定一冲突和失败输入集,最终一定可以获得一组不可满足的逻辑公式. 首先,我们定义函数 RepairSym,其将元件编号映射到修复符号. 而 Sym 函数是将元件集合 Q 映射到的修复符号集合,即 $\text{Sym}(Q)=\cup_{\text{RepairSym}(c)},(c\in Q)$. 显然,给定一个元件集和某路径条件,可以诱导出一个逻辑合取式,形式化描述如下.

定义 7　给定一元件集 Q 和 $\pi[k]$($\pi[k]=T_1\wedge T_2\wedge\cdots\wedge T_n$),其中 T_i 为路径条件中的约束条件. 则令 $\Gamma(Q,\pi[k])=\wedge T_i$,其中 T_i 来自 $\pi[k]$中的子句且 T_i 中使用了 Sym(Q)中变量符号.

设冲突 $C=\{c_1,c_2,\cdots,c_n\}$ 和 $\pi[k]$,假设冲突中的元件是有序的. 用 $F_{1,i}$,$F_{i+1,n}$表示冲突 C 可以分割成两个冲突子集,其中 i 的取值是 $1,2,\cdots,n-1$(i 也称为分割点). 对于每个分割点,就可以得到两个公式 $A=\Gamma(F_{1,i},\pi[k])$,$B=\Gamma(F_{i+1,n},\pi[k])$,且 $A\wedge B=\perp$. 令 $I_i=\text{Interp}(A,B)$,就可以构造一个 Craig 插值序列.

定义 8　冲突的 Craig 插值序列. 给定冲突 $C=\{c_1,c_2,\cdots,c_n\}$ 和 $\pi[k]$. Interps(C,$\pi[k]$)=$\{I_1,I_2,\cdots,I_i,I_{n-1}\}$是 C 相对于 $\pi[k]$的插值序列,其中 $i\in\{1,2,\cdots,n-1\}$,且 $I_i=\text{Interp}(\Gamma(F_{1,i},\pi[k]),\Gamma(F_{i+1,n},\pi[k]))$.

显然,计算出 $\pi[\boldsymbol{i}\|\boldsymbol{r}]$中的每条 $\pi[k]$路径条件的 Craig 插值序列,就可以得到一个 Craig 插值矩阵. 表 3 列出的是图 2 中冲突$\{1,2,3,4,6\}$的 Craig 插值矩阵,其中,对于 $\pi[k]$,若 $I_i=$false,则后继的 Craig 插值省略了,用“-”表示. 在冲突中多个连续的分割点处,若存在语义等价(用符号“≡”表示)的 Craig 插值,则称为归纳 Craig 插值. 归纳 Craig 插值的存在表明在该分割点处所对应的元件的执行与否不影响 $\pi[k]$不可满足,即可能存在冗余元件.

表 3　冲突的 Craig 插值序列

$\pi[k]$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3,4\}$
$\pi[0]$	false	-	-	-
$\pi[1]$	false	-	-	-
$\pi[2]$	true	true	true	true
$\pi[3]$	true	$s_{08}\leqslant 0$	$s_{09}\leqslant 0$	false

定义 9　给定冲突 C 和 $\pi[k]$,设有 Craig 插值序列 Interps(C,$\pi[k]$)=$\{I_1,I_2,\cdots,I_i,I_{n-1}\}$. 对于 C 的连续分割点 i,$j(j>i)$,若有 $I_i\equiv I_j$,则称 I_i 对于分割点 i,j 是归纳 Craig 插值.

需要注意的是:

(1) Interps(C,$\pi[k]$)中的归纳 Craig 插值可能不明显. 例如,在表 3 中,对于 $\pi[3]$中,虽然 I_2 和 I_3 不相同,但是语义等价,属于归纳 Craig 插值,因为根据表 2 中的元件 3 的原来值可知 $s_{09}=s_{08}$.

(2) 归纳 Craig 插值具有传递性,即 $I_i\equiv I_{i+1}$且 $I_{i+1}\equiv I_{i+2}$,则 $I_i\equiv I_{i+2}$. 因此,对于多个连续的归纳 Craig 插值,除第一个以外的插值称为后继归纳 Craig 插值,用特殊值 NULL 标记(如表 4). 并且称第一个归纳 Craig 插值在冲突分割点所对应的元件为归纳 Craig 插值的起始元件.

(3) 对于 Craig 插值矩阵中的每个 Interps(C,$\pi[k]$)进行归纳插值计算后,就得到归纳 Craig 插值矩阵(表 4).

表 4　对于表 3,计算了归纳 Craig 插值后的插值序列

$\pi[k]$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3,4\}$
$\pi[0]$	false	NULL	NULL	NULL
$\pi[1]$	false	NULL	NULL	NULL
$\pi[2]$	**true**	NULL	NULL	NULL
$\pi[3]$	true	$\mathbf{s_{08}\leqslant 0}$	NULL	false

4.2　冲突中元件的过滤方法

4.2.1　基于归纳 Craig 插值元件的过滤方法

给定冲突 C、归纳 Craig 插值矩阵 $\boldsymbol{M}$,若 Interps(C,$\pi[k]$)$\in\boldsymbol{M}$ 中存在后继归纳 Craig 插值 I_i 时(即 $I_i=$NULL),则冲突分割点 i 所对应的元件(用 $C[i]$表示)可能为冗余元件. 换言之,若 $I_i=$NULL,删除冲突中的元件 $C[i]$不会影响 $\pi[k]$的不可满足性. 但是删除元件 $C[i]$可能会影响到其他路径条件 $\pi[s]$($s\neq k$)的不可满足性. 因此必须给出条件判定后继归纳 Craig 插值所对应的元件是否为无相关冗余元件.

定义 10　给定冲突 $C=\{c_1,c_2,\cdots,c_n\}$ 和归纳 Craig 插值序列矩阵 $\boldsymbol{M}$,令 Interps(C,$\pi[k]$)=$\{I_1,I_2,\cdots,I_i,I_{n-1}\}\in\boldsymbol{M}$,若 $I_i=$NULL 且 $C[i]$属于冗余元件,当且仅当对于 Interps(C,$\pi[s]$)($s\neq k$),$C[i]$不属于其中归纳插值的起始元件.

例如,在表 4 中,对于 Interps(C,$\pi[2]$),$I_2=$NULL,但是 $C[2]$=元件 2 不能确定为无相关元件,因为对于 Interps(C,$\pi[3]$),元件 2 是归纳 Craig 插值 I_2 的起始元件.

4.2.2　无相关条件语句元件的过滤方法

考虑到给定具体的失败输入,其只能执行 $\pi[\boldsymbol{i}\|\boldsymbol{r}]$中的某一条路径条件,且该路径条件的不可满足是违反

程序合法行为模型造成的,而其他的路径条件的不可满足是因为其中的条件语句元件的分支约束不满足造成的. 称路径条件中的条件语句元件的分支约束不满足的路径条件为非执行路径条件. 下面的定义则给出了根据冲突的 Craig 插值序列识别非执行路径条件的方法.

定义 11 设有冲突 $C=\{c_1,c_2,\cdots,c_n\}$,$\pi[\boldsymbol{i}\parallel\boldsymbol{r}]$,失败输入和 Craig 插值序列矩阵. 对于某 $\pi[k]$,如果存在 Craig 插值 $I_i=\text{false}$ 且 $C[i]$为条件语句元件,则称$\pi[k]$对应于元件 $C[i]$的非执行路径条件.

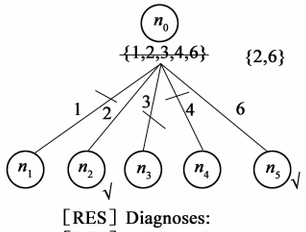
给定某失败输入,根据定义 11 可识别出所有的非执行路径条件集,记为 NE(K,c),其中 K 为路径条件的编号集,c 为对应的元件. 对于 NE(K,c),强制对其中的每条路径条件 $\pi[k](k\in K)$中对应的条件语句元件 c 所映射的子式赋真值,得到新的路径条件 $\pi'[k]$,这一过程称条件变更. 将条件变更后的 NE(K,c)中的每条路径条件 $\pi'[k](k\in K)$取合取式,即 $\bigwedge\limits_{k\in K}\pi'[k]$. 无相关条件语句元件的核心是验证当前失败输入条件下 $\bigwedge\limits_{k\in K}\pi'[k]$是否不可满足. 验证函数 Sat 的定义如下,(其中 Q 为冲突,失败输入 $v_i\in J$)

$$\text{Sat}\left(\bigwedge_{k\in K}\pi'[k],Q,v_i\right)\Leftrightarrow\exists\boldsymbol{r}.\bigwedge_{k\in K}\pi'[k]\bigwedge_{r\in R}r=\text{Org}(r)[v_i\parallel\boldsymbol{r}]\tag{4}$$

例如,在表 3 中,计算的非执行路径条件集 NE($K=\{0,1\}$,1). 对 $\pi[k](k\in\{0,1\})$中元件 1 所对应的分支约束($s_{07}=0$)更改为($s_{07}=1$)得到 $\pi'[k]$. 对于失败输入 β,应用 Sat 函数于 $\bigwedge\limits_{k\in\{0,1\}}\pi'[k]$得出其仍不可满足. 这说明元件 1 的分支条件语句取值与当前失败输入引发的程序失效不相关. 事实上,在失败输入 β 的条件下,无论元件 1 的取值为真或假,都使得 $\pi[\boldsymbol{i}\parallel\boldsymbol{r}]$不可满足.

值得注意的是,这里只是说相对于失败输入 β,元件 1 与程序失效无关. 在利用多个失败输入进行错误定位的情况下,先可以对于每一个失败输入进行条件语句元件的不相关分析,得到相对于每一个失败输入的不相关条件语句元件集,然后,将所有不相关条件语句元件集进行求交集即得到相对于所有失败输入的不相关条件语句元件集.

图 3 是将本文提出的冲突冗余元件分析方法应用到图 1 的过程所生成的碰集树. 与 Forensic 的诊断结果相比较,诊断数从 5 个减少到 2 个,碰集树中的结点数从 9 个减少到 3 个.



[RES] Diagnoses:
[RES] Id 2 Line 7:"x"
[RES] Id 6 Line 17:"z"

图3 改进的碰集树构造过程和诊断结果

5 相关算法

将上一节的思想表述成冲突元件过滤算法,该算法可以分三个步骤:

(1) 计算 Craig 插值序列矩阵数组

给定一个失败输入、冲突和 $\pi[\boldsymbol{i}\parallel\boldsymbol{r}]$,利用 ComputeInterMatrix 函数计算 Craig 插值序列矩阵. 计算完所有失败输入所对应的插值序列矩阵就得到 Craig 插值序列矩阵数组. 如算法 1 所示.

算法 1 计算 Craig 插值序矩阵数组

输入:$C=\{c_0,c_1,\cdots,c_{n-1}\}$:一个冲突,其中元件个数|$C$|
$J=\{\text{in}_0,\text{in}_1,\cdots,\text{in}_{n-1}\}$:一个失败输入集合,总失败输入数|$J$|
$\pi[\boldsymbol{i}\parallel\boldsymbol{r}]$:程序合法行为模型,总的路径条件数|PASS|
输出:Craig 插值序列的矩阵数组 $\boldsymbol{G}$

```
1.  for  s = 0; s < |J|; s ++ do
2.    G[s] ← ComputeInterMatrix(π[i ∥ r], C, in_i)
3.  end for
```

(2) 无相关条件语句元件的过滤方法

对于算法 1 产生的每一个 Craig 插值序列矩阵,调用 ComputeNEPath 函数找出非执行路径条件集 NE(K,c). 对 NE 中的路径条件进行条件变更后利用 Sat 函数进行验证元件 c 对于当前失败输入是否属于无相关条件语句元件. 所有的失败输入所对应的无相关条件语句元件的交集就是可以过滤的条件语句元件. 具体内容见算法 2.

算法 2 无相关条件语句元件的过滤方法

输出:Removable :相对于所有失败输入的不相关条件语句元件集.

```
1.  Removable ← ϕ
2.  for  i = 0; i < |J|; i ++ do
3.    Removed ← ϕ
4.    NE(K, c_i) ← ComputeNEPath(G[i], C) // c_i ∈ C
5.    将 NE 中的 π[k](k∈K)实现条件变更得到 π'[k]
6.    if  Sat(⋀_{k∈K} π'[k], C, in_i) = false  then
7.    Removed ← Removed ∪ c_i
8.    end if
9.    if Removable = ϕ then
10.     Removable ← Removed
11.     else
12.     Removable ← Removable ∩ Removed
13.     end if
14.  end for
```

(3)基于归纳 Craig 插值元件的过滤方法

首先调用 ComputeInductiveInterp 函数计算归纳插值矩阵. 考察归纳插值矩阵中的每个 Craig 插值序列,对每个后继归纳插值根据定义 10 判断是否分割点处所对应的元件属于冗余元件,其中 IsStartC 函数检验某元件是否属于起始元件(如算法 3 所示).

显然,算法中的基本操作是可满足性计算或者计算 Craig 插值,则算法 3 的时间复杂度最大. 而算法 3 在最坏情况下复杂度约为 $O(|\mathrm{PASS}\| C \| J|)$,这也是整个算法时间复杂度.

算法 3　基于归纳 Craig 插值元件的过滤方法

输出:完成过滤操作后的冲突.

```
for i = 0; i < |J|; i++ do
  M[i] ← ComputeInductiveInterp(G[i])
 end for
   for i = 0; i < |J|; i++ do
   for each I = Interps(C, π[k]) ∈ M[i]
   for z = 0; z < |C| − 1; z++ do
   if I_z = NULL and IsStartC(C[z]) = false then
   Removable ← Removable ∪ C[z]
   end if
  end for
 end for
 end for
 return C-Removable
```

6　实验结果

为了验证本文方法(记 PA)的有效性,在 Forensic 中实现了 PA 算法. 采用 Tcas 的 38 个错误版本(Tcas 有 41 个错误版本,但其他几个版本 Forensic 无法计算出诊断)作为实验对象将 PA 与 Forensic 进行比较研究. 实验参数设置列在表 5. 表 6 列出了将 PA 与原 Forensic 方法应用到 Tcas 进行错误定位产生的诊断结果(对版本号 v8, v16, v19, syb_ce_it 分别设置为 200, 400, 200). 从诊断总数来看,Forensic 总体计算出 254 个诊断,而 PA 计算出 131 个诊断,诊断数减少了 48.4%,其中,单元件诊断数减少率为 22.9%,双元件诊断数减少率为 66.4%.

表 5　实验参数设置

参数	取值	意义
b	syb	基于符号执行和模型诊断
syb_th	lin	符号执行中基于整数线性算术理论
syb_sv	z3[20]	使用 Z3 作为可满足性解析器
syb_cond	true	把条件语句视为元件
syb_ce_it	100	符号执行的最大执行次数
syb_dia_ni	1	用于诊断的失败输入数量

表 6　Forensic 与 PA 诊断结果的比较

版本号	元件数	|PASS|	Forensic 方法			本文方法		
			总诊断数	单元件诊断	双元件诊断	总诊断数	单元件诊断	双元件诊断
V1	75	91	3	3	0	2	2	0
V2	75	92	11	3	8	5	3	2
V3	75	92	12	3	9	5	2	3
V4	75	92	4	4	0	3	3	0
V5	74	79	14	3	11	8	3	5
V6	75	91	6	2	4	3	2	1
V7	75	84	7	5	2	3	1	2
V 8	75	188	6	2	4	2	1	1
V9	75	92	3	3	0	2	2	0
V10	75	84	1	1	0	1	1	0
V11	73	76	6	2	4	3	2	1
V12	75	76	14	3	11	6	3	3
V13	75	86	13	4	9	7	4	3
V14	75	89	1	1	0	1	1	0
V15	74	89	10	5	5	5	3	2
V 16	75	312	5	2	3	2	1	1
V17	75	84	5	1	4	2	1	1
V18	75	96	5	1	4	1	1	0
V 19	75	188	5	1	4	2	2	0
V20	75	92	7	7	0	3	3	0
V21	74	96	5	5	0	4	4	0
V22	74	96	1	1	0	1	1	0
V23	74	96	1	1	0	1	1	0
V24	74	96	5	5	0	4	2	2
V25	75	91	3	3	0	2	2	0
V26	74	85	12	3	9	6	3	3
V27	74	79	14	3	11	5	3	2
V28	75	84	11	3	8	5	4	1
V29	72	92	10	2	8	6	2	4
V30	72	92	10	4	6	7	2	5
V31	77	90	1	1	0	1	1	0
V32	77	89	1	1	0	1	1	0
V33	75	84	5	1	4	2	1	1
V34	75	76	14	5	9	7	4	3
V35	75	84	11	3	8	6	3	3
V37	75	84	5	1	4	2	1	1
V39	75	91	3	3	0	2	2	0
V41	73	92	4	4	0	3	3	0
总计			254	105	149	131	81	50

采用 Forensic 方法,在诊断的计算过程中碰集树总体生成 647 个结点,而 PA 总体生成 339 个结点,结点的减少率为 47.6%(图 4).从平均数来看,原 Forensic 方法诊断计算平均生成 17 个结点,而本文方法平均生成 9 个结点.

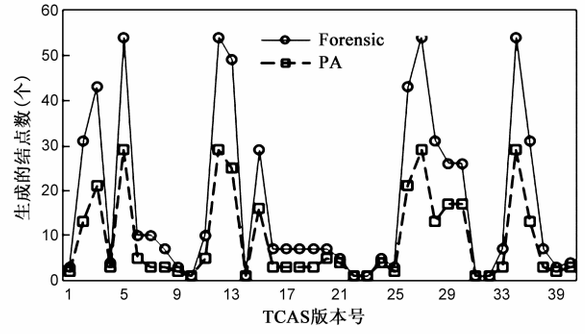


图4 Forensic与PA在碰集树产生结点数上比较

从诊断计算时间开销来看(图 5),Forensic 诊断计算总的时间花费是 64.8s,而 PA 诊断计算花费是 418.1s.显然,本文所提方法花费在诊断计算的时间是原 Forensic 方法的大约 6 倍.造成诊断计算时间增多的原因多次调用 Z3 进行 Craig 插值计算与可满足性判定.

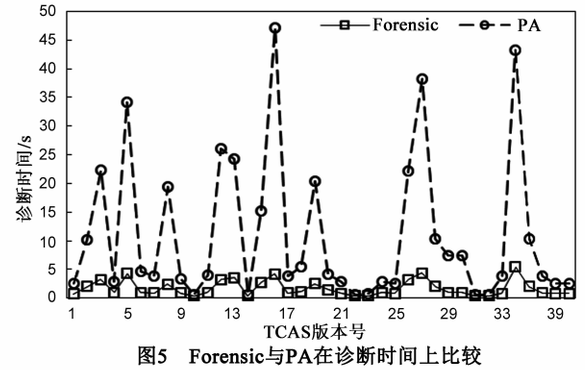


图5 Forensic与PA在诊断时间上比较

将 syb_dia_ni 参数取 1 和 2 时,对诊断数的差值进行比较(图 6),显然,与 Forensic 相比,PA 增加该参数的取值对诊断的诊断准确度的影响较小,也即是敏感度低于 Forensic.

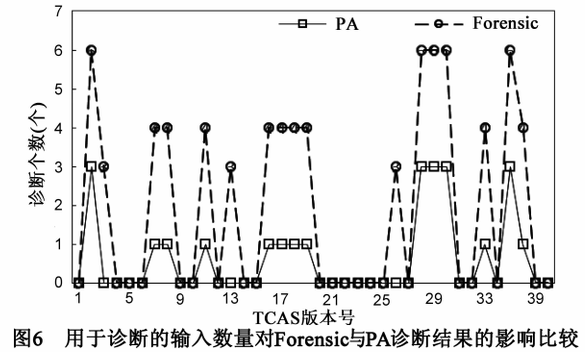


图6 用于诊断的输入数量对Forensic与PA诊断结果的影响比较

7 结语

本文分析了 MBD 软件错误定位中诊断假阳性的一些原因,并提出了一种冲突中冗余元件的分析方法.该方法既利用归纳的 Craig 插值去识别冲突中的冗余元件,同时能够清除无相关条件语句元件.实验结果表明该方法能够通过消除冲突的无相关元件,从而减少诊断的假阳性.下一步工作包括:更广泛地对本文所提方法进行实证研究.进一步结合其方法(比如基于统计错误定位方法,多层次的 MBD 诊断方法)对基于 MBD 软件错误定位方法进行优化改进.

参考文献

[1] Lee Naish, Hua Jie Lee, Kotagiri Ramamohanarao. A model for spectra-based software diagnosis [J]. ACM Transactions on software engineering and methodology (TOSEM), 2011, 20(3): 11.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, Arjan J C Van Gemund. A practical evaluation of spectrum-based fault localization [J]. Journal of Systems and Software, 2009, 82(11): 1780 – 1792.

[3] Konighofer R, Roderick Bloem. Automated error localization and correction for imperative programs [A]. Formal Methods in Computer-Aided Design (FMCAD) [C]. USA: IEEE, 2011, 91 – 100.

[4] Rui Abreu, Arjan J C Van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation [J]. Artificial Intelligence, 2010, 174(18): 1481 – 1497.

[5] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach [J]. IEEE Transactions on Software Engineering, 2006, 32(10): 831 – 848.

[6] Dennis Jeffrey, R Gupta. Effective and efficient localization of multiple faults using value replacement [A]. IEEE International Conference on Software Maintenance [C]. USA: IEEE, 2009. 221 – 230.

[7] Andreas Zeller, Ralf Hildebrandt. Simplifying and isolating failure-inducing input [J]. IEEE Transactions on Software Engineering, 2002, 28(2): 183 – 200.

[8] Swarup Kumar Sahoo, John Criswell, Chase Geigle, Vikram Adve. Using likely invariants for automated software fault localization [J]. ACM SIGARCH Computer Architecture News, 2013, 41(1): 139 – 152.

[9] 王建峰,魏长安,盛云龙,等.基于错误交互集的组合测试软件故障定位方法[J].电子学报,2014,42(6):1173 – 1178.
WANG Jian-feng, WEI Chang-an, et al. Locating errors in combinatorial testing using set of possible faulty interactions

[J]. Acta Electronica Sinica, 2014, 42(6): 1173 – 1178. (in Chinese)

[10] Raymond Reiter. A theory of diagnosis from first principles [J]. Artificial intelligence, 1987, 32(1): 57 – 95.

[11] Rui Abreu, Peter Zoeteweij, Arjan J C van Gemund. An observation-based model for fault localization [A]. Proceedings of the International Workshop on Dynamic Analysis [C]. USA: ACM, 2008. 64 – 70.

[12] Franz Wotawa, Mihai Nica, Iulia Moraru. Automated debugging based on a constraint model of the program and a test case [J]. The Journal of Logic and Algebraic Programming, 2012, 81(4): 390 – 407.

[13] Birgit Hofer, Franz Wotawa. Combining slicing and constraint solving for better debugging: The conbas approach [A]. Advances in Software Engineering [C]. Hindawi Publishing Corporation, 2012. Article ID 628571.

[14] Jorg Weber, Franz Wotawa. Diagnosing dependent failures in the hardware and software of mobile autonomous robots [A]. New Trends in Applied Artificial Intelligence [C]. Berlin: Springer, 2007. 633 – 643.

[15] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem [J]. The Journal of Symbolic Logic, 1957, 22(3): 250 – 268.

[16] Evren Ermis, Martin Schaf, Thomas Wies. Error invariants [A]. Lecture Notes in Computer Science: Formal Methods (Volume 7436) [C]. Berlin: Springer, 2012. 187 – 201.

[17] 陈祖希,徐中伟,霍伟伟,等. 基于 Craig 插值的线性混成系统符号化模型检测[J]. 电子学报, 2014, 42(7): 1338 – 1346.
CHEN Zu-xi, XU Zhong-wei, et al. Symbolic model checking for linear hybrid systems base on craig interpolation [J]. Acta Electronica Sinica, 2014, 42(7): 1338 – 1346. (in Chinese)

[18] Xiangyu Zhang, Neelam Gupta, Rajiv Gupta. Locating faults through automated predicate switching [A]. Proceedings of the 28th International Conference on Software Engineering [C]. USA: ACM, 2006. 272 – 281.

[19] Jurgen Christ, Evren Ermis, Martin Schaf, Thomas Wies. Flow-sensitive fault localization [A]. Verification, Model Checking, and Abstract Interpretation [C]. Brelin: Springer, 2013. 189 – 208.

[20] Leonardo De Moura, Nikolaj Bj? rner. Z3: An efficient smt solver [A]. Tools and Algorithms for the Construction and Analysis of Systems [C]. Berlin: Springer, 2008. 337 – 340.

作者简介

徐　勇　男,1977 年生,博士生,主要研究方向为软件工程、软件调试.
E-mail: xyus@ whu. edu. cn

毋国庆　男,1954 年生,教授、博士生导师,主要研究领域为软件需求工程、形式化方法.
E-mail: wgq@ whu. edu. cn