量也可以作为形式化的规格说明插入到代码中作为注释,帮助提高程序可理解性.

通常,默认不变量有 196 个类型,这些类型基本上满足了我们的要求.当然,我们可以在此基础上定义更多的不变量类型.如果在现有的类型下所得到不变量是不完全的,这说明涉及这些不变量的语句没有被执行到.一种情况是这些语句是多余的,另一种情况是测试用例的选取不够充分.如果我们的覆盖率能够达到希望的要求,那么即便是有少量的不变量没有计算到,也不会对结果产生太大影响.

下面我们通过一个计算器例子来说明如何得到程序不变量(见表 1):

Table 1 An example of calculator

表 1 计算器例子

Code

JButton b[]=new *JButton*[16];
String s[]={“0”,“1”,“2”,“3”,“4”,“5”,“6”,“7”,“8”,
“9”,“+”,“–”,“*”,“/”,“*C*”,“=”};
…
public void *actionPerformed*(*ActionEvent e*)
{ …
if (*flag*==10) *m*2=*m*1+*m*2;
else if (*flag*==11) *m*2=*m*2–*m*1;
else if (*flag*==12) *m*2=*m*2**m*1;
else if (*flag*==13) *m*2=*m*2/*m*1;
else *m*2=*m*1;
…
}
…

(1) 生成测试用例集.在该例子中,我们用原先已有的测试用例生成技术[30]生成一个测试用例集,该技术结合白盒测试和黑盒测试方法将输入域分成若干个等价类,再从等价类中选取测试用例.为了更好地提取不变量,我们还采用 Gupta[31]提出的方法进一步提炼测试用例集.在给定的例子中,测试用例集包括 80 个测试用例;

(2) 定义不变量类型.Daikon 不变量类型有很多种,如 $x=a,x\neq 0,a\leqslant x\leqslant b,y=ax+b,x\leqslant y,x=fn(y),x\in y$ 等,我们还可以自定义程序不变量的类型;

(3) 运行程序和测试用例,得到轨迹文件;

(4) 得到程序不变量.在该例子中得到的部分不变量见表 2,其中,不变量 *size*(*this.b*[])==*size*(*this.s*[])是指数组 *b* 的长度等于数组 *s* 的长度.

Table 2 A part of the program invariants of the program

表 2 通过原程序得到的部分不变量

Code (*C*1)	Invariants (*TI*)

JButton b[]=new *JButton*[16];
String s[]={“0”,“1”,“2”,“3”,“4”,“5”,“6”,“7”,“8”,
“9”,“+”,“–”,“*”,“/”,“*C*”,“=”};
…
public void *actionPerformed*(*ActionEvent e*)
{ …
if (*flag*==10) *m*2=*m*1+*m*2;
else if (*flag*==11) *m*2=*m*2–*m*1;
else if (*flag*==12) *m*2=*m*2**m*1;
else if (*flag*==13) *m*2=*m*2/*m*1;
else *m*2=*m*1;
…
}
…
	1) *JB*:::*OBJECT*
	2) *size*(*this.b*[])==*size*(*this.s*[])
	3) *this.m*1 ⩾ 0
	4) *this.flag* ⩾ 0
	5) *this.b*!=*null*
	6) *this.b*[] contains no *nulls* and has only one value, of length 16
	7) *this.b*[] *elements*!=*null*
	8) *this.s*[] contains no *nulls* and has only one value, of length 16
	9) *size*(*this.b*[])==16
	10) *this.s.getClass*()==*java.lang.String*[].*class*
	11) *this.m*1⩽*this.m*2
	12) *this.flag*<*size*(*this.b*[])–1

接下来,我们描述如何得到不好的程序不变量,即失效数据.程序 *C*1 改为程序 *C*2,将 *JButton b*[]=new *JButton*[16]中的 16 改为 17,以及调换 *String s*[]={“0”,“1”,“2”,“3”,“4”,“5”,“6”,“7”,“8”,“9”,“+”,“–”,“*”,“/”,“*C*”,

“=”}中‘+’和‘−’的位置时,我们会得到新的不变量集 *T*2,见表 3.通过对比发现:不变量 1)、不变量 3)~不变量 5)、不变量 7)、不变量 10)~不变量 12)跟原先一样,而不变量 2)、不变量 6)、不变量 8)、不变量 9)发生了变化.如原先的不变量 2)是 *size*(*this*.*b*[])==*size*(*this*.*s*[]),而改变后的是 *size*(*this*.*b*[])−1==*size*(*this*.*s*[]).显然,这里描述数组 *b* 的长度等于数组 *s* 的长度的不变量发生了变化,因此认为新的不变量 *size*(*this*.*b*[])−1==*size*(*this*.*s*[])是一个失效数据.

Table 3 A part of program invariants of the modified program

表 3 修改程序后得到的部分不变量

Code (*C*2)	Invariants (*T*2)

JButton b[]=new JButton[17];
String s[]={“0”,“1”,“2”,“3”,“4”,“5”,“6”,“7”,“8”,
“9”,“−”,“+”,“*”,“/”,“*C*”,“=”};
…
public void *actionPerformed*(*ActionEvent e*)
{ …
if (*flag*==10) *m*2=*m*1+*m*2;
else if (*flag*==11) *m*2=*m*2−*m*1;
else if (*flag*==12) *m*2=*m*2**m*1;
else if (*flag*==13) *m*2=*m*2/*m*1;
else *m*2=*m*1;
…
}
…
	1) *JB*:::*OBJECT*
	2) ***size*(*this*.*b*[])−1==*size*(*this*.*s*[])**
	3) *this*.*m*1 ⩾ 0
	4) *this*.*flag*⩾0
	5) *this*.*b*!=*null*
	6) ***this*.*b*[] contains no *null*s and has only one value, of length 17**
	7) *this*.*b*[] *elements*!=*null*
	8) ***this*.*s*[] contains no *null*s and has only one value, of length 17**
	9) ***size*(*this*.*b*[])==17**
	10) *this*.*s*.*getClass*()==*java*.*lang*.*String*[].*class*
	11) *this*.*m*1⩽*this*.*m*2
	12) *this*.*flag*<*size*(*this*.*b*[])−1

在该程序中,执行上述所有测试用例,共生成不变量 3 207 个,其中,失效不变量有 282 个.

2 基于不变量的可靠性计算

2.1 可靠消息计算模型

本文采取 Nelson 可靠性模型来计算软件的可靠性.Nelson 软件可靠性模型对软件做出如下假设:

1) 程序被认为是集合 *E* 上的一个可计算函数 *F* 的一个规范,这里用 $E=(E_i:i=1,\ldots,m)$表示用于执行程序的所有输入数据的集合,一个输入数据对应一个程序执行回合;
2) 对于每个输入 E_i,程序执行产生输出 $F(E_i)$;
3) 由于程序包含缺陷,程序实际确定函数 *F*′,该函数不同于希望函数 *F*;
4) 对于某些 E_i,程序实际输出 $F'(E_i)$在希望输出 $F(E_i)$的容许范围之内,即$|F'(E_i)-F(E_i)|\leqslant\Delta_i$.但对另一些 E_j,程序实际输出 $F'(E_j)$超出容许范围,即$|F'(E_j)-F(E_j)|>\Delta_j$,这时认为程序发生一次失效;
5) 测试过程中不剔除程序缺陷.

Nelson 模型计算软件可靠性的函数为

$$R=1-\sum_{i=1}^{m}\frac{f_i}{n_i}P(E_i) \tag{1}$$

其中,E_i 域中数据被选取的概率是 $P(E_i)$;E_i 域中被选取的数据的个数为 n_i;E_i 域中 n_i 个被选取数据导致程序失效的个数为 f_i.

根据上述模型,接下来具体描述如何计算表 1 给出的计算器例子的可靠性.

Step 1. 生成测试用例集,在第 1 节中,我们共得到了 80 个测试用例.

Step 2. 将测试用例进行分类.具体的分类方法是:

a) 通过 gcov 编译,执行测试用例,生成测试用例对程序的覆盖信息;

b) 分析该测试用例对程序中每个分支或者子函数的覆盖率;

c) 测试用例 *A* 和测试用例 *B*,如果覆盖程序相同的部分超过一定的值(该值需要根据程序的结构以及测试用例集来确认),那么认为测试用例 *A* 和测试用例 *B* 是同一类测试用例.

接下来,我们选取计算器中如下的 4 个选择分支来阐述如何进行分类:

- 分支 1: if (*flag*==10) *m*2=*m*1+*m*2;
- 分支 2: else if (*flag*==11) *m*2=*m*2−*m*1;
- 分支 3: else if (*flag*==12) m2=*m*2**m*1;
- 分支 4: else if (*flag*==13) *m*2=*m*2/*m*1.

测试用例:

- *T*1: *x*1 2 3;
- *T*2: *x*1 9.1 8.5;
- *T*3: *x*2 89 63;
- *T*4: *x*2 77 34.9;
- *T*5: *x*4 9 3;
- *T*6: *x*3 76.1 63.

测试用例中,*x*1,*x*2,*x*3,*x*4 表示标志位:当标志位是 *x*1 时,程序执行分支 1;当标志位是 *x*2 时,程序执行分支 2;当标志位是 *x*3 时,程序执行分支 3;当标志位是 *x*4 时,程序执行分支 4.当执行测试用例 *T*1 时,通过 gcov 可以知道:分支 1 被调用,测试用例覆盖分支 1,而没有覆盖其他 3 个分支.同理,我们可以得到 *T*2 仅覆盖了分支 1,*T*3 和 *T*4 覆盖了分支 2,*T*5 覆盖了函数分支 4,*T*6 覆盖了分支 3.因此,我们把 *T*1 和 *T*2 分为一组测试用例集,*T*3 和 *T*4 分为一组测试用例集,*T*5 分为一组测试用例集,*T*6 分为一组测试用例集.

根据上述方法,我们可以将所得的 80 个用例分成 5 类,具体分类结果见表 4:

Table 4 Classification of the test cases

表 4 测试用例的分类

Sort	1	2	3	4	5	Total
Number	17	11	23	26	3	80

Step 3. 分类后,运行每个测试用例集和程序,根据第 1 节所述,得到程序不变量和失效不变量.具体数据见表 5.

Table 5 Program invariants and failure data of the test cases

表 5 各组用例产生的不变量和失效数据(失效不变量)

Sort	1	2	3	4	5
Total invariants	1 286	999	1 542	2 201	47
Failure invariants	8	192	198	269	2

Step 4. 通过得到的失效数据进行可靠性计算.

根据 Nelson 模型,我们计算得到的可靠性为

$$R=1-\left(\frac{17}{80}\frac{8}{1286}+\frac{11}{80}\frac{192}{999}+\frac{23}{80}\frac{198}{1542}+\frac{26}{80}\frac{269}{2201}+\frac{3}{80}\frac{2}{47}\right)=0.894.$$

3 实验设置

为了与传统的基于不同测试方法的可靠性计算方法进行比较,以此来说明基于不变量计算可靠性的可行性和优点,我们采用了 3 个常用的测试用例生成方法:随机生成、基于分块覆盖生成和基于分支覆盖生成以及西门子程序包:the Software-artifact Infrastructure Repository[32].我们在实验中验证的内容有:(1) 失效数据(失效不变量)获取的可行性,即,不同测试方法对失效数据的比例影响不大;(2) 可靠性计算的可行性,即,基于不变量计算的可靠性与传统方法计算所得的可靠性接近.验证的优点是:基于不变量计算所得的可靠性更稳定,因此更能体现程序的可靠性.

3.1 3种测试方法

本文采用随机选取、基于分块覆盖选取和基于分支覆盖选取这 3 种不同的测试方法从测试用例集中选取测试用例.

随机测试即从程序的输入域中随机地选取一个测试用例,每一个测试用例都有一个被选中的概率.在我们的实验中,根据预先定义的发生概率,从程序的规格说明书中随机选取至少一个功能进行随机测试;然后,随机选取一个合适的测试用例来对这个说明书进行测试,直至选取的测试用例集覆盖程序的所有功能.分块覆盖测试是我们把只有一个入口和一个出口的顺序语句作为一个块.选取测试用例时,要覆盖所有的程序块及至少有一个测试用例覆盖一个程序块.按分支覆盖准则进行测试是指:设计若干测试用例,运行被测程序,使得程序中每个判断的取真分支和取假分支至少经历一次及判断的真假值均曾被满足.

3.2 程序来源

本文采用的实验程序来源于 the Software-artifact Infrastructure Repository[32].The Siemens set 包含 7 个 *C* 程序,见表 6.它们常用于错误定位和软件可靠性计算.

Table 6 Siemens set

表 6 Siemens set

Program	Faulty versions	Test cases	Description
print_tokens	7	4 130	Lexical analyzer
print_tokens2	10	4 115	Lexical analyzer
replace	32	5 542	Pattern recognition
schedule	9	2 650	Priority scheduler
schedule2	10	2 710	Priority scheduler
tcas	41	1 608	Altitude separation
tot_info	23	1 052	Information measure

表 6 详细描述了我们实验所需要的 7 个程序,每个程序包含一个正确的版本以及多个错误版本,每个错误版本包含一个错误(在我们的实验中,需要把所有的错误整合在一起,构成一个包含多个错误的版本).另外,每个程序有一个输入集(即测试用例集).

4 产生失效数据的可行性分析

我们通过 3 种不同的方法来选取测试用例集 *T*1(随机选取)、*T*2(基于分块覆盖选取)、*T*3(基于分支覆盖选取).分别运行正确版本程序 *C* 与 *T*1、*C* 与 *T*2、*C* 与 *T*3 得到程序不变量 *TI*1,*TI*2 和 *TI*3,运行错误版本程序 *F* 与 *T*1、*F* 与 *T*2、*F* 与 *T*3 得到程序不变量 *FI*1,*FI*2 和 *FI*3.比较 *TI*1 与 *FI*1、*TI*2 与 *FI*2 以及 *TI*3 与 *FI*3 的不同,分别得到各自的失效数据.

表 7 是 3 种方法得到的程序不变量总数和失效数据的个数.Random 表示随机选取测试用例得到的程序不变量,Block 表示基于分块覆盖选取测试用例得到的程序不变量,Branch 表示基于分支覆盖选取测试用例得到的程序不变量.False 指失效的程序不变量数,Total 表示程序不变量的总数.由表可以发现:对于同一个程序,采用不同的测试方法获得的不变量是不一样的;当然,获得的失效数据,即不好的不变量也是不一样的.

Table 7 Failure data and the total program invariants gotten by the three test methods

表 7 3 种方法得到的失效数据与程序不变量

Program	Random		Block		Branch	
	False	Total	False	Total	False	Total
print_tokens	1 038	16 925	1 015	16 851	998	16 832
print_tokens2	1 286	15 971	1 269	15 735	1 315	15 821
replace	2 410	18 065	2 359	17 906	2 187	17 318
schedule	1 472	15 126	1 354	14 973	1 308	14 865
schedule2	1 873	16 725	1 749	16 287	1 702	15 994
tcas	1 957	20 108	1 913	19 947	1 864	20 116
tot_info	1 965	20 294	1 870	19 083	1 897	19 525

接下来计算失效数据所占的百分比.表 8 给出了每种方法所获得的失效数据百分比,可以发现不同方法获得的失效数据百分比基本相近,误差不大.例如:对于程序 print_tokens,随机方法、基于分块覆盖方法和基于分支覆盖方法所得到的失效数据百分比分别为 6.13%,6.02%,5.92%,其最大的相对误差只有(6.13%−5.92%)/6.13%×100%=3.4%;其次,我们计算得到 3 种方法平均能够发现的失效数据百分比分别为 9.69%(random),9.49%(Block)和 9.32%(Branch),最大相对误差为 3.8%.

Table 8 Percentages of the failure data

表 8 失效数据所占程序不变量总数的百分比

Program	Random (%)	Block (%)	Branch (%)
print_tokens	6.13	6.02	5.92
print_tokens2	8.05	8.06	8.31
replace	13.34	13.17	12.62
schedule	9.73	9.04	8.79
schedule2	11.19	10.73	10.64
tcas	9.73	9.59	9.26
tot_info	9.68	9.79	9.71

综上所述,我们可得出结论:不同测试方法得到不同的不变量;但是就失效数据的百分比来说,它们之间相差较小,比较稳定.

5 可靠性计算和结果分析

根据第 2 节的方法,我们来计算 7 个程序的可靠性.我们以程序 schedule 为例,详细叙述失效数据的获得和可靠性的计算.通过上述 3 种方法,我们将随机选取的测试用例分为 15 个测试用例集,将基于分块覆盖选取的测试用例分为 14 个测试用例集,将基于分支覆盖选取的测试用例分为 14 个测试用例集.

Step 1. 用随机测试方法选取了 260 个测试用例,分块覆盖方法选取了 272 个测试用例,分支覆盖方法选取了 295 个测试用例.

Step 2. 将测试用例进行分类,分类结果见表 9.

Table 9 Result of the classification of the test cases from program schedule

表 9 程序 schedule 的测试用例集分类结果

Sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	总计
Random	1	25	14	2	7	82	4	6	7	40	4	34	2	3	29	260
Block	18	28	9	11	73	2	4	9	30	11	52	2	4	19	–	272
Branch	2	20	9	10	91	4	5	6	43	2	62	6	15	20	–	295

Step 3. 分类后,运行每个测试用例集和程序,根据前一节所述办法得到总的不变量和失效数据,见表 10.

Table 10 Failure data and total program invariants in each sort

表 10 分类后所得失效数据和总不变量数目

Sort	Random		Block		Branch	
	失效数据	总不变量	失效数据	总不变量	失效数据	总不变量
1	4	27	1 054	4 742	54	606
2	554	4 208	537	3 727	1 042	5 919
3	721	5 100	0	5 100	29	372
4	8	999	2 764	12 944	1 683	8 938
5	715	10 984	2 815	13 029	2 314	12 629
6	3 983	12 546	1 162	14 764	853	7 410
7	394	9 930	1 319	3 512	983	2 864
8	912	5 054	0	5 054	861	1 511
9	77	3 710	1 835	11 987	3 349	11 409
10	1 934	10 446	791	6 421	595	1 046
11	192	3 216	4 581	13 903	778	11 086
12	1 343	11 824	1 025	6 735	475	8 964
13	257	4 369	415	4 482	1 262	6 318
14	594	4 616	1 846	12 797	1 224	4 164
15	1 236	4 486				

Step 4. 将得到的数据代入 Nelson 模型进行可靠性计算.计算结果为:0.860 8(随机),0.862 5(基于分块覆盖),0.877 2(基于分支覆盖).

其余程序的可靠性计算结果见表 11,其中,R-reliability 表示随机选取测试用例计算的可靠性,Block-reliability 表示基于分块覆盖选取测试用例计算的可靠性,Branch-reliability 表示基于分支覆盖选取测试用例计算的可靠性.

Table 11 Reliability computed by program invariants

表 11 通过不变量计算的可靠性

Program	R-reliability	Block-reliability	Branch-reliability
print_tokens	0.854 1	0.860 5	0.867 9
print_tokens2	0.841 7	0.852 6	0.839 7
replace	0.804 4	0.817 0	0.820 3
schedule	0.795 0	0.800 3	0.810 7
schedule2	0.821 4	0.829 8	0.835 0
tcas	0.789 2	0.794 7	0.805 1
tot_info	0.835 3	0.827 8	0.821 3

由表 11 可知:不同测试方法算出的可靠性虽然不一样,但相差较小.例如:对于程序 print_tokens,随机方法算出的可靠性最低,为 0.854 1;基于分支覆盖方法算出的可靠性最大,为 0.867 9;而两者的差值为 0.013 8.

我们再来看一下不同方法产生的可靠性的波动大小,这里采用方差来刻画.根据概率统计的理论,方差越大,个体间的波动就越大.也就是说,不同测试方法对可靠性的影响就越大.表 12 计算了不同程序在 3 种测试方法下的可靠性方差. 3种方法对不同程序可靠性的影响动荡范围在0.000 003 2~0.000 004 7之间,因此,所得的可靠性是比较稳定的,测试方法对其影响不大.这是因为不变量反映的是程序整体的行为,所计算的可靠性是软件整体的可靠性,而不是在具体测试方法下的局部可靠性.

Table 12 Variance of the computed reliability

表 12 不变量计算的可靠性方差

程序	方差
print_tokens	0.000 003 2
Print_tokens2	0.000 003 2
Replace	0.000 004 7
Schedule	0.000 004 3
Schedule2	0.000 003 2
Tcas	0.000 004 4
tot_info	0.000 003 3

综上所述,我们发现:

1) 在不同测试方法下,由不变量计算出的可靠性不是完全相等,还有一定的误差,其原因是有些不好的不变量之间还有一定的依赖性;
2) 用不变量来计算可靠性时,测试方法对软件的可靠性影响不大.

6 可行性分析

本节利用现有的计算可靠性方法来计算所给 7 个程序的可靠性,即,用标准的 Nelson 模型计算.这里我们不再详细叙述计算过程,只给出计算的可靠性.然后,我们比较用不变量计算出的可靠性和用现有方法计算出的可靠性之间的差异.可靠性计算结果见表 13.

由表 13 得知,在同一测试方法下,用不变量和用现有方法计算出的可靠性有细微差异,有时不变量计算的可靠性大,有时现有方法计算的可靠性大;但是两者相差不大,这可由表 14 的相对误差的绝对值得到.在表 14 中,最大的相对误差绝对值仅有 3.787%,而大部分相对误差的绝对值在 1%左右.

Table 13 Reliability computed by program invariants and by traditional method

表 13 利用不变量和利用现有方法计算出的可靠性

Program	R-reliability		Block-reliability		Branch-reliability	
	不变量	传统方法	不变量	传统方法	不变量	传统方法
print_tokens	0.854 1	0.847 2	0.860 5	0.861 3	0.867 9	0.874 8
print_tokens2	0.841 7	0.851 8	0.852 6	0.849 1	0.839 7	0.864 7
replace	0.804 4	0.797 2	0.817 0	0.810 3	0.820 3	0.829 7
schedule	0.795 0	0.808 6	0.800 3	0.831 8	0.810 7	0.819 4
schedule2	0.821 4	0.819 9	0.829 8	0.824 7	0.835 0	0.831 5
tcas	0.789 2	0.801 7	0.794 7	0.794 8	0.805 1	0.815 9
tot_info	0.835 3	0.824 9	0.827 8	0.817 5	0.821 3	0.836 1

Table 14 Relative error between the two kinds of reliability

表 14 两类可靠性的相对误差

Program	R-reliability (%)	Block-reliability (%)	Branch-reliability (%)
print_tokens	0.814 4	–0.092 9	–0.7888
Print_tokens2	–1.185 7	0.412 2	–2.891 2
Replace	0.903 2	0.826 9	–1.132 9
Schedule	–1.681 9	–3.787 0	–1.061 8
Schedule2	0.182 9	0.618 4	0.420 9
Tcas	–1.559 2	–0.012 6	–1.323 7
tot_info	1.260 8	1.259 9	–1.770 1

接下来,我们比较现有方法和不变量方法计算出的两类可靠性在不同测试方法下产生的波动大小.根据前面所述,我们可以用方差来刻画这一性质.表 15 列出了这两类可靠性计算方法在 3 种测试方法下的方差.由表 15 可以发现:两种方法计算出的可靠性方差都不是太大;但是相比较而言,现有方法的可靠性波动远大于不变量计算出的可靠性波动.这是因为现有方法计算出的可靠性一定程度上依赖于测试方法,反映软件的局部行为,计算出的可靠性是在该测试方法下的可靠性,所以在不同的测试方法下,可靠性会产生相对较大的波动.而不变量反映软件整体的行为,虽然不同的测试方法会产生不同的不变量,但这些不变量的合成效果却是一样的,都反映软件整体的行为,所以不变量计算出的可靠性是软件的整体可靠性,其波动产生的原因在于有些失效数据(不好的不变量)之间还有一定的相关性.

Table 15 Comparison on the variances of the two kinds of reliability

表 15 两类可靠性的方差比较

程序	方差	
	不变量(10^{-5})	传统方法(10^{-5})
print_tokens	0.32	13
Print_tokens2	0. 32	4.6
Replace	0. 47	1.8
Schedule	0. 43	9.0
Schedule2	0. 32	2.7
Tcas	0. 44	7.7
tot_info	0. 33	5.8

综述所述:相对于现有的计算可靠性方法,我们还可以利用不变量来计算软件的可靠性;且利用不变量计算的可靠性更接近软件的实际可靠性.

注 1:Simens 软件包通常被用来做软件测试、错误定位和不变量计算的开源程序包,所以我们用它来做基于不变量的软件可靠性的评价.但它只是针对序列化程序,并没有并发程序.今后的一个研究工作就是针对并发程序来验证我们的方法.并发结构可以给我们带来大规模的程序,我们将在此基础上研究我们方法的稳定性与适应性.

7 相关工作

7.1 软件可靠性模型

在过去的几十年里,有许多关于可靠性模型的研究.总的来说,这些模型大致可以分为软件可靠性增长模型(software reliability growth models,简称 SRGMs)和基于架构的模型.

软件可靠性增长模型 SRGMs[1−9]是时间域模型,它们可以用来估计软件可靠性和残留故障数量.这些模型首先假设测试是按照给定的操作剖面执行的[10],然后,通过在测试阶段的记录的故障历史来预测软件程序的行为.基于架构的可靠性模型[11−18]又可以分为两类,即,测试阶段的架构[11,12,16−18]和设计阶段的架构[13−15].前者只有程序代码可用,通过执行大量的测试来获取轨迹的数据,从而根据这些数据来决定软件系统的架构.利用在覆盖测试中获得的轨迹信息,可以构建程序的架构,如离散时间马尔科夫链[12]、连续时间马尔科夫链[16]以及半马尔科夫过程[11].构件的失效行为可以用泊松过程或基于覆盖测试和故障密度的时间独立失效率来建模,在这种情况下,覆盖测试和单元测试在决定所有信息过程中起重要作用.后者事先已经定义了构件以及构件之间的相关性.每个构件的可靠性依赖于操作剖面和其在系统中的位置,系统可靠性的计算等价于构件执行轨迹的可靠性,其中,构件执行轨迹可看作是一系列构件的执行,而构件执行轨迹的可靠性又等于该轨迹上所有构件可靠性的乘积.轨迹可靠性问题还可以转化为基于构件的问题[13].在这种情况下,可靠性的计算需要假设构件的失效是相互独立的以及接口的可靠性为 1.在无记忆程序中,构件运行的独立性是可接受的;但是对其他程序来说,该假设无法满足[15].特别地,当某个构件被循环执行时,该构件多次的执行是不独立的.

7.2 失效数据

失效数据的来源广、获取技术多[21],人们可以从历史版本或者类似软件中获得失效数据,此时我们无需拥有具体的需求的软件;在设计阶段,可以从需求文档、UML 图和项目计划书中获得相关信息和数据;通过测试获取相应的失效数据;用户反馈,例如故障报告、修改需求等,也可以作为获得数据的来源.通过测试来获取数据是其中使用最广泛的一种方法.测试阶段根据模型的不同主要收集两类不同的失效数据:

- 一是给定测试用例,收集失效的测试用例数[19];
- 二是给定测试环境,运行软件一段时间,收集在这段时间内的失效次数和失效间隔时间[33,34].

文献[23]归纳了用来获得失效数据的各种不同的测试方法.但是这些失效数据仅仅是软件测试时间内的输入/输出信息,而并没有考虑程序内部的信息.而软件的可靠性必然是由其内部信息,包括动态行为和内部结构来决定的,因此,根据这些失效数据算出来的可靠性并不十分准确.

程序不变量是通过收集软件的动态行为获取能够反映其内部信息的数据,因此它能够反映软件内部行为,比仅仅考虑程序的输入/输出的数据携带更多的信息,似乎可以作为更好的失效数据来源.然而对这方面的研究却还不多,仅有一些尝试.Pietrantuono 通过程序不变量来监测和预测在线软件可靠性[25];丁佐华等人通过程序不变量来计算服务组合的在线可靠性计算[26],但是这些工作并没有说明用不变量来计算可靠性的可行性.

8 结 论

本文通过实验来说明我们可以通过程序不变量来计算可靠性:其一,在同一测试方法下,不变量计算出的可靠性和现有方法计算出的可靠性相差不大;其二,在不同测试方法下,不变量计算出的可靠性波动较小,更接近软件的实际可靠性.从不变量的角度来刻画软件可靠性,使我们对软件的整体行为和软件可靠性有更深刻的认识,有利于软件可靠性的评估和预测.

接下来的工作是:

1) 针对不变量之间的相关性:如果不好的不变量之间存在相关性,会增加失效数据,影响计算结果.我们应尽量去掉这种相关性,使得剩下的不变量尽可能独立.一种办法是建立数据流的依赖图;
2) 针对假性测试用例:有些正确的测试用例会产生错误的不变量.在这种情况下,我们就丢失了失效数

据.如果假性测试用例较多,那样计算出来的结果就不准确.为了解决这种情况,我们需要对测试用例的结果建立频谱矩阵,基于频谱矩阵去除掉这些测试用例;

3) 运用更多的测试方法来计算软件可靠性,进一步验证测试方法对软件可靠性的影响;

4) 在实际应用中,我们并不知道正确的程序,这样我们就不能通过比较来获得不好的不变量,而必须通过一定的方法,根据已有的测试用例信息和已获得的不变量信息来确定不好的不变量,这将是我们接下来要解决的关键问题之一.

致谢 我们向对本文工作给予支持和建议的同行,尤其是北京大学高可信软件技术教育部重点实验室金芝教授表示感谢.

References:

[1] Yamada S. Software Reliability Modeling: Fundamentals and Applications. Tokyo: Springer-Verlag, 2014. 6–15.

[2] Wood A. Software reliability growth models. Technical Report 96.1, No.130056, Tandem Computers, 1996.

[3] Goel AL, Okumoto K. Time-Dependent error-detection rate model for software and other performance measures. IEEE Trans. on Reliability, 1979,28:206–211.

[4] Misra PN. Software reliability analysis. IBM Systems Journal, 1983,22:262–270.

[5] Ohba M, Chou XM. Does imperfect debugging affect software reliability growth? In: Druffel LE, ed. Proc. of the ICSE'89. Pittsburg, 1989. 237–244.

[6] Farr W. Software Reliability Modeling Survey. In: Lyu MR, ed. Handbook of Software Reliability Engineering. New York: IEEE Computer Society Press, 1996. 71–117.

[7] Chen M, Lyu MR, Wong WE. Effect of code coverage on software reliability measurement. IEEE Trans. on Reliability, 2001,50(2): 165–170.

[8] Zhao J, Zhang RB, Gu GC. Study on software reliability growth model considering failure dependency. Chinese Journal of Computers, 2007,30(10):1713–1720 (in Chinese with English abstract).

[9] Li HF, Wang SQ, Liu C, Zheng J, Li Z. Software reliability model considering both testing effort and testing coverage. Ruan Jian Xue Bao/Journal of Software, 2013,24(4):749–760 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4257.htm [doi: 10.3724/SP.J.1001.2013.04257]

[10] Horgan JR, Mathur AP. Software Testing and Reliability. In: Lyu MR, ed. Handbook of Software Reliability Engineering. New York: IEEE Computer Society Press, 1996. 531–566.

[11] Littlewood B. Software reliability modeled for modular program structure. IEEE Trans. on Reliability, 1979,28(3):241–246.

[12] Cheung RC. A user-oriented software reliability model. IEEE Trans. on Software Engineering, 1980,6(2):118–125.

[13] Dolbec J, Shepard T. A component based software reliability model. In: Bockus D, Bennet K, eds. Proc. of the Conf. of the Center for Advanced Studies on Collaborative Research. ACM Press, 1995. 19–29.

[14] Wang WL, Wu Y, Chen MH. An architecture-based software reliability model. In: Proc. of the 1999 Pacific Rim Int'l Symp. on Dependable Computing. IEEE, 1999. 143–150.

[15] Hamlet D. Are we testing for true reliability? IEEE Software, 1992,9(4):21–27.

[16] Laprie JC, Kanoun K. Software Reliability and System Reliability. In: Lyu MR, ed. Handbook of Software Reliability Engineering. New York: IEEE Computer Society Press, 1996. 27–69.

[17] Mao XG, Deng YJ. A general model for component-based software reliability. Ruan Jian Xue Bao/Journal of Software, 2004,15(1): 27–32 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/27.htm

[18] Hou CY, Cui G, Liu HW. Rate-Based component software reliability process simulation. Ruan Jian Xue Bao/Journal of Software, 2011,22(11):2749–2759 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3930.htm [doi: 10.3724/SP.J.1001.2011.03930]

[19] Nelson E. Estimating software reliability from test data. Microelectronics Reliability, 1978,17(1):67–73.

[20] Nguyen EA, Rexach CF, Thorpe DP, Walther AE. The importance of data quality in software reliability modeling. In: Proc. of the 21st Int'l Symp. on Software Reliability Engineering (ISSRE). IEEE, 2010. 220−228.

[21] Buhnova B, Chren S, Fabriková L. Failure data collection for reliability prediction models: A survey. In: Seinturier L, Bures T, McGregor JD, eds. Proc. of the 10th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures. ACM Press, 2014. 83−92.

[22] Rapps S, Weyuker EJ. Selecting software test data using data flow information. IEEE Trans. on Software Engineering, 1985,(4): 367−375.

[23] Dimov A, Chandran SK, Punnekkat S. How do we collect data for software reliability estimation. In: Rachev B, Smrikarov A, eds. Proc. of the 11th Int'l Conf. on Computer Systems and Technologies and Workshop for Ph.D. Students in Computing on Int'l Conf. on Computer Systems and Technologies. ACM Press, 2010. 155−160.

[24] Chen MH, Mathur AP, Rego VJ. Effect of testing techniques on software reliability estimates obtained using a time-domain model. IEEE Trans. on Reliability, 1995,44(1):97−103.

[25] Pietrantuono R, Russo S, Trivedi KS. Online monitoring of software system reliability. In: Proc. of the 8th European Dependable Computing Conf. IEEE, 2010. 209−218.

[26] Ding ZH, Chen MH, Li XX. Online reliability computing of composite services based on program invariants. Information Sciences, 2014,264:340−348.

[27] Hangal S, Lam MS. Tracking down software bugs using automatic anomaly detection. In: Proc. of the 24th Int'l Conf. on Software Engineering. ACM Press, 2002. 291−301.

[28] Sahoo SK, Li M, Ramachandran P, Adve SV, Adve VS, Zhou YY. Using likely program invariants to detect hardware errors. In: Proc. of the 38th Int'l Conf. on Dependable Systems and Networks (DSN). IEEE, 2008. 70−79.

[29] Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. on Software Engineering, 2001,27(2):99−123.

[30] Ding ZH, Zhang K, Hu JL. A rigorous approach towards test case generation. Information Sciences, 2008,178:4057−4079.

[31] Gupta N. Generating test data for dynamically discovering likely program invariants. In: Proc. of the ICSE 2003 Workshop on Dynamic Analysis. 2003. 21−24.

[32] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005,10(4):405−435.

[33] Jeske DR, Zhang XM, Pham L. Adjust software failure rates that are estimated from test data. IEEE Trans. on Reliability, 2005, 54(1):107−114.

[34] Zhang XM, Teng XL, Pham H. Considering fault removal efficiency in software reliability assessment. IEEE Trans. on Systems, Man, and Cybernetics, Part A: Systems and Humans, 2003,33(1):114−120.

附中文参考文献:

[8] 赵靖,张汝波,顾国昌.考虑故障相关的软件可靠性增长模型研究.计算机学报,2007,30(10):1713−1720.

[9] 李海峰,王栓奇,刘畅,郑军,李霞.考虑测试工作量与覆盖率的软件可靠性模型.软件学报,2013,24(4):749−760. http://www.jos.org.cn/1000-9825/4257.htm [doi: 10.3724/SP.J.1001.2013.04257]

[17] 毛晓光,邓勇进.基于构件软件的可靠性通用模型.软件学报,2004,15(1):27−32. http://www.jos.org.cn/1000-9825/15/27.htm

[18] 侯春燕,崔刚,刘宏伟.基于率的构件软件可靠性过程仿真.软件学报,2011,22(11):2749−2759. http://www.jos.org.cn/1000-9825/3930.htm [doi: 10.3724/SP.J.1001.2011.03930]

周远(1989−),男,浙江台州人,硕士生,主要研究领域为软件建模,模型检查,软件测试和可靠性.

丁佐华(1964−),男,教授,博士生导师,CCF高级会员,主要研究领域为软件测试与可靠性,软件建模与分析,软件自适应控制系统,智能计算及应用.