基于变异技术的程序故障自动化修复方法*

马春燕[1]，刘 杰[1]，赖文豫[2]
(1. 西北工业大学 软件与微电子学院，西安 710072；2. 中国人民大学 信息学院，北京 100872)

摘 要： 结合故障定位技术，提出了一种基于变异的程序故障自动化修复方法，并采用该方法阐释了 C 程序表达式故障的变异修复机制，研制了 C 程序故障自动化修复辅助工具。通过对实际故障系统案例的分析，证明了方法的有效性。

关键词： 变异技术；故障定位；故障自动化修复
中图分类号： TP311 **文献标志码：** A **文章编号：** 1001-3695(2014)01-0177-05
doi:10.3969/j.issn.1001-3695.2014.01.041

Method for repairing program automatically with mutation

MA Chun-yan[1], LIU Jie[1], LAI Wen-yu[2]
(1. *School of Software & Microelectronics, Northwestern Polytechnical University, Xi' an* 710072, *China*; 2. *School of Information, Renmin University of China, Beijing* 100872, *China*)

Abstract: Based on the bug localization technology, this paper proposed an automatic repair approach via mutation and applied this method to repair bugs of expressions of C language programs. It developed the automatic repairing tool based on the proposed method. Through case studies, it is found that the proposed method is effective.

Key words: mutation technique; bug localization; bug automatic repairing

随着计算机硬件水平快速发展以及应用程序规模和复杂度急剧增加，程序的可靠性和可维护性问题越来越突出，程序故障已经严重影响了软件产业的快速、健康发展。近年来，通过软件测试技术发现程序故障后，如何修复决定了软件的开发周期、质量和成本。2002 年，美国国家标准与技术研究所 NIST 的调查表明，在美国，故障修复的成本每年累计高达 595 亿美元(占 GDP 的 0.6%)[1]；2008 年，一份调查发现 139 个北美公司每年修复缺陷的花费额高达 2 200 万美元[2]。程序故障修复是软件工程实践中最重要的活动之一。目前，软件开发人员修复程序故障是困难、耗时的手工过程；故障定位和修复能力依赖于软件开发人员的专业知识和行业经验。这种现状导致了软件开发周期长、成本高，阻碍了软件产业的发展，因而程序自动化修复技术的理论研究与应用正在受到学术界和工业界的关注。目前，国内外专家针对故障自动修复理论、方法已展开探索。在修复数据结构、内存以及特定类型故障方面取得了一定的成绩。Dallmeier 等人[3]分析了软件成功和失败运行行为(采用扩展的状态机建模)之间的差异，产生可能的修复，并运行回归测试用例，减少程序修复后产生新问题的风险。Perkins 等人[4]通过不变式监控程序运行，自动检测和修补部署软件的汇编级错误。这两种方法修复的错误仅限于与选择的条件表达式或监控器相关的错误，产生自动修复仍有很大的提升空间。Arcuri 等人[5]首次提出利用遗传编程自动修复程序的方法，随着程序规模的扩大，遗传编程搜索空间的规模呈指数级增长，因此该方法适宜单个函数的修复。为了缩小进化算子的搜索空间，使得该方法适宜大规模、复杂程序的自动修复，美国弗吉尼亚大学 Weimer 等人采用抽象语法树对程序进行抽象，先后提出基于遗传编程对遗留 C 语言程序[6~8]和汇编语言程序[9]进行自动修复的方法。国内，南京理工大学何加浪等人[10]在 Weimer 等人的研究基础上，通过不变量约束对搜索空间进行划分和约简，并针对测试用例很难覆盖所有功能的需求，提出使用成功测试用例学习的不变量约束来解决该问题。但是文献[6~8]的遗传修复方法有一个修复的前提：成功测试用例和失败测试的执行路径不同，修复程序故障的可行解必须在程序的其他若干位置存在，这限制了修复的类型，而且也存在解空间的完备性问题；该方法在修复过程中通常大量引入无关的、冗余的代码改变，导致代码急剧膨胀；遗传算法本身并非一个稳定的算法，它依赖于程序本身的特性是否适应，因此该方法不能达到一个稳定的程序修复效果，且修复代价巨大。本文提出了一种简单的、易于实践的基于变异技术的故障修复方法。

1 基于变异技术的故障修复方法

1.1 基于变异技术的故障修复原理

传统基于变异的测试技术是事先已知一个期望程序，再根据所使用的变异算子对期望程序进行变异得到一批包含故障的变异体，最后利用这些生成的变异体来验证某个测试用例集 T 的充分性。而基于变异技术的故障修复思路与此相反，它是已知一个故障程序，故障修复方法根据所使用的变异算子对故障程序进行变异得到一批变异体，然后通过已知的测试用例集 T 来验证这些变异体是否是故障程序的修复版本。当利用已知测试用例集 T 来验证一个变异体是否修复故障程序时，需要定义杀死一个变异体的标准，即一个变异体被某个用例杀死指的是变异体执行完该用例后的输出与期望输出不一致。任

收稿日期：2013-04-26；修回日期：2013-06-05 基金项目：国家自然科学基金资助项目(61103003)；西北工业大学基础研究基金资助项目(JC201126)

作者简介：马春燕(1978-)，女，博士，主要研究方向为软件测试、软件故障定位和修复(machunyan@nwpu.edu.cn)；刘杰(1991-)，男，主要研究方向为程序自动化调试；赖文豫(1989-)，男，硕士研究生，主要研究方向为数据库.

何被杀死的变异体都不可能是故障程序的一个正确修复版本，这是因为它仍然导致某些测试用例执行失败。另一方面，如果一个变异体能够顺利通过所有可用的测试用例，即它是活着的，那么这个变异体可能是该故障程序的一个正确修复，称它为故障程序的一个潜在修复版本。通过对这些潜在修复版本程序的分析可以在一定程度上帮助程序开发人员最终得到期望的程序。本文提出的变异修复工作流程如图 1 所示。首先根据输入的故障程序判定是否存在适用变异算子的位置，然后依据合适的变异算子产生故障程序的变异体，最后验证变异体是否为潜在的修复方案。

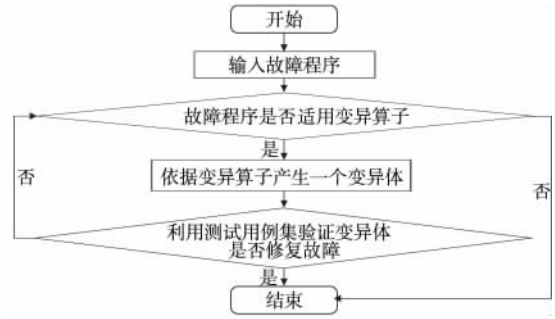

图 1 变异修复工作流程

1.2 故障程序修复步骤

使用基于变异技术的故障修复方法修复程序故障的操作步骤如下:

a) 确定修复的故障类型。程序变异技术是一种错误驱动测试技术，该技术主要是针对某类特定的程序故障，因此在进行故障变异修复之前，首先要明确变异修复主要针对的故障类型。变异修复的设计人员需要对程序中常见故障类型进行分析总结，从中选择发生概率高且对程序危害较大的故障类型进行修复设计。这样不仅可以取得较好的修复效果，还能降低修复的成本。

b) 设计合适的变异算子。变异算子的设计是基于变异技术的故障修复方法的核心环节，它是产生变异体的根本依据。使用丰富多元的变异算子能够使更多类型的故障得到修复，但与此同时也可能导致大量变异体的生成。一种有效的修复方法应该能够修复多种不同类型的故障，而一种高效的修复方法则要求变异修复的代价应该尽可能地小。因此，变异算子的设计对变异修复方法的效果和效率有很大的影响。变异修复的设计人员需要针对所要修复故障的类型，在修复效果与修复效率之间进行权衡，从而设计出合适的变异算子。

c) 设定变异修复停止条件。由于基于变异技术的故障修复方法针对的是某些特定类型的程序故障，因此并不是所有程序故障都能得到修复。当产生并检查一定数量的变异体后，如果仍然没有找到潜在修复版本的变异体，那么变异修复将会面临一个重要选择: 继续还是停止。如果选择继续，意味着除非找到一个潜在修复版本的变异体，否则直到故障程序所有可能变异体被检查完之前，修复过程不会停止。然而，当修复对象是一个大型程序时，生成变异体的数量通常特别大，如果选择继续，就可能导致大量资源的白白消耗。因此，为变异修复设定一个停止条件是非常必要且合理的。

d) 变异故障程序。根据所设计的变异算子与所设定的变异修复停止条件，开始对故障程序进行变异产生变异体。在实施本步骤之前，可以先利用故障定位技术对故障程序进行故障定位，然后再利用故障定位的结果指导变异修复工作的进行，从而降低变异修复的代价。

e) 确定潜在修复版本。通过对步骤 d) 变异产生的变异体进行检查验证，从而确定故障程序的潜在修复版本。

1.3 C 语言程序的故障修复方法

1.3.1 C 语言程序常见故障类型

通过对 SIR(software-artifact infrastructure repository) 知识库[10] 提供的故障系统案例的分析，本文总结了 C 程序四类常见故障类型:

a) 内存故障。内存分配和释放错误、数组越界错误、函数返回值错误(如返回局部变量) 、非法指针引用错误等都会引起内存故障。

b) 条件表达式故障。条件表达式故障主要发生在 if、while、for 结构的条件表达式中，主要是由表达式中关系运算符、逻辑运算符出错或者边界条件出现位移导致的。

c) 变量赋值故障。变量赋值故障主要发生在变量赋值语句中，多种原因可以导致这种故障。本文主要针对的是由运算符错误或者边界赋值错误(作为边界的变量，赋值过程中多加一点或少加一点造成的错误) 所导致的变量赋值故障。

d) 语句冗余或缺失故障。由于语句冗余或缺失造成程序流程与预计流程不同并最终导致程序执行失败的一类故障。

通过对 C 程序常见故障类型的分析总结，再结合程序变异技术自身的特点及优势，本文主要针对上述 C 程序常见故障类型中的 b) c) 两类故障以及语句冗余故障进行变异修复。

1.3.2 变异算子的设计

1) 常用的变异算子　在基于变异的测试技术中，常用的变异算子主要有算术运算符的变异、关系运算符的变异以及逻辑运算符的变异。它们都是针对程序中出现的运算符进行相应的变异，能够解决部分由运算符错误引起的变量赋值故障或条件表达式故障。由于大部分程序设计语言都含有算术、关系和逻辑运算符，因此这三个变异算子具有很好的通用性。表 1 对这三个变异算子进行了描述。

2) 新变异算子的设计　本文针对 1.3.1 节总结的 C 程序条件表达式故障、变量赋值故障以及语句冗余故障进行变异算子的设计。在设计变异算子的过程中，每一类待修复类型的故障可能会包含许多子类型故障，这时需要依据这种故障在实际程序开发过程中发生概率的高低进行有针对性的设计，优先为发生概率高的故障设计相应的变异算子。本文针对 C 程序故障设计的变异算子集如表 2 所示。

表 1 常用变异算子

变异算子名	变异规则	修复的故障类型
算术运算符变异	遇到一个算术运算符则选择其他的算术运算符来替换它	变量赋值故障
关系运算符变异	遇到一个关系运算符则选择其他的关系运算符来替换它	条件表达式故障
逻辑运算符变异	遇到一个逻辑运算符则选择其他的逻辑运算符来替换它	条件表达式故障

表 2 C 程序变异算子集

变异算子名	变异规则	故障类型
运算符变异	遇到一个运算符则选择其他同类的运算符将其替换产生变异体	变量赋值故障 条件表达式故障
条件表达式变异	遇到一个条件表达式则对其进行相关修改产生变异体	条件表达式故障
边界值变异	遇到 for 循环初始化语句则对其初始化变量的边界值进行修改产生变异体	变量赋值故障
常量浮点变异	遇到常量则用与其等价的浮点常量将其替换产生变异体	变量赋值故障
类型扩展变异	遇到可执行语句中含有较低级别的基本数据类型则将其提高一级进行变异	变量赋值故障
删除变异	遇到一条可执行语句则直接删除	语句冗余故障

各类变异算子具体描述如下:

a) 运算符变异。针对可疑语句中可能出现的算术运算符、逻辑运算符、关系运算符、一元运算符或赋值运算符从同一

类型的运算符中选取一个不同的运算符来替换原有运算符进行变异。具体类型划分如下(注: 括号括起来的属于同一类型) :

(a) 算术运算符　(+ 　- 　* 　/ 　%) ;
(b) 逻辑运算符　(&& 　||)　(! 无) ;
(c) 关系运算符　(> 　> = 　< 　< =) 　(= = 　! =) ;
(d) 自增/自减运算符　(前 ++ 　后 ++) (前 - - 　后 - -) (++ - -) ;
(e) 位操作运算符: (& 　^ 　|) ;
(f) 单目运算符　(- 　无) ;
(g) 赋值运算符　(= 　+ = 　- = 　* = 　/ = 　% =) ;
(h) 易混淆运算符　(&& 　&)　(| 　||) 。

该类变异算子能够修复的故障类型主要包括变量赋值故障及条件表达式故障。当运算符错误发生在变量赋值语句中时,则该变异算子可以修复变量赋值故障; 而当运算符错误发生在 if、while、for 结构中的条件表达式时,则该变异算子可以修复条件表达式错误。

b) 条件表达式变异。针对可疑语句中可能出现的条件表达式(如 if、while、for 结构中的条件表达式) ,可将条件表达式整体取反进行变异,或者通过对条件表达式整体加 1 或减 1 进行变异。本变异算子可以解决部分由于分支条件写反或者分支条件出现位移所导致的条件表达式故障。

c) 边界值变异。针对可疑语句中可能出现的 for 循环初始化语句,可将初始化变量的值加 1 或减 1 进行变异。本变异算子主要针对 for 循环结构中初始化语句经常出现赋值偏差的情况,通过对初始化变量的赋值进行加 1 或减 1 的校正从而修复部分变量赋值故障。这里选择用加 1 或减 1 进行校正而非加 2 或减 2 的原因主要是赋值偏差为 1 的情况较为普遍,因此优先为这种子类型的故障设计变异算子。

d) 常量浮点变异。针对可疑语句中可能出现的整型常量,可将其变异成与之等值的浮点型常量。本变异算子可以修复部分变量赋值故障。如赋值语句 float $a = 1/2$; 假设程序要求 a 的期望赋值结果是 0.5,而根据 C 语言的语法,由于分子分母都是整型常量,因此除法运算的结果依旧为一个整型常量 0,从而引发变量赋值故障。将该句中的整型常量 1 变异成 1.0 即可修复此故障。

e) 类型扩展变异。针对可疑语句中可能出现的 float、int 等类型关键字,可将其变异成高一级的类型。本变异算子通过类型扩展变异可以修复部分由于精度缺失造成的变量赋值故障。

f) 删除变异。将某条可执行语句直接删去进行变异,解决部分语句冗余错误。

当检查一个可疑语句时,若找到能够适用上述变异算子集的位置时则可依据相应的变异算子进行变异产生变异体。需要注意的是,有时在一个位置上使用一个变异算子可能会产生多种不同的变异体,因此在实现该变异算子的过程中可以依据不同故障发生的可能性大小,适当调整不同变异体产生的先后顺序,以减少变异体产生及检测的总数量,从而在一定程度上降低故障修复的代价。例如,当一个语句中包含逻辑运算符 && 时,依据上述的运算符变异算子可以得到两种变异体,一种是将 && 变异成||,另一种则是将 && 变异成 &。倘若根据以往的经验了解到||写成 && 的故障比较常见,则在实现该变异算子时可先产生第一种变异体。

1.3.3　变异修复停止的条件

由于基于变异技术的故障修复方法并不能修复所有类型的程序故障,因此存在找不到潜在修复版本的情况。这就要求变异修复的设计人员为变异修复设定一个停止条件。可以给变异修复所花费的代价设置一个上限,当变异修复所花费的代价达到这个上限但仍未找到潜在修复版本的变异体时,就会停止修复过程的继续进行。假设一个变异体从产生到执行完已知测试用例集 T 中的所有用例所花费的代价为 e; 倘若在找到一个潜在修复版本的变异体之前需要检查 n 个变异体,那么此次变异修复所花费的代价为

$$\text{effort}(E) = ne$$

由上式可知,变异修复所花费的代价主要取决于 n 值的大小,即变异修复过程中产生并检查的变异体数量越少所花费的代价就越少,反之亦然。因此可以为变异修复过程中被检查变异体的数量设置一个上限。然而,直接把这个上限作为变异修复的停止标准并不实际。因为被检查变异体的数量直接取决于变异算子的选择,如果变异修复过程中使用了一个较大的变异算子集,那么每个语句都可能有大量的变异体产生,这样变异修复可能会迅速突破该上限。此外,从一个变异修复设计人员角度来说,设置一个大小合适的上限通常比较困难。本文采用的停止标准是为故障程序设置一个代码检测行数的上限,即在故障修复过程中,当变异检查可执行代码的行数达到该上限但仍未找到潜在修复时就立即停止修复过程的继续进行。

1.3.4　潜在修复版本的确定

为了确定一个变异体是否是潜在修复版本,本文假定如果一个变异体执行完所有可用的测试用例后它还活着,那么该变异体即为一个潜在修复版本,所以变异修复要求程序开发人员除了提供故障程序之外,还需提供一个测试用例集 T 来帮助验证一个变异体是否是故障程序的一个潜在修复版本。同时还要求测试用例集 T 应至少包括一个失败测试用例(即故障程序执行该用例后的输出结果与期望的不同) ,否则依据定义故障程序本身将是一个潜在修复,而这是不合实际的。但是,基于变异技术的故障修复方法并没有对测试用例集 T 中所包含的测试用例的数量进行限制。然而 T 中所包含的测试用例的数量越多则每次验证变异体是否为潜在修复版本的代价就越大。因此测试用例集的选取会对变异修复所花费的代价造成影响。如果程序开发人员提供的测试用例集比较大,可以先对测试用例集进行优化,再进行故障程序的变异修复,从而有效地降低变异修复的代价。

1.3.5　结合故障定位的变异修复方法改进

由于一个变异算子可能应用在程序中大量的位置上,如果对于从何处开始检查没有线索,那么为了找到一个潜在修复版本,可能要检查大量的变异体。下面通过表 3 展示一个例子对如何将变异技术和故障定位技术结合以便高效地修复程序故障进行研究。已知一个故障程序 P' 和一个测试用例集 T(共有三个测试用例 t_1, t_2, t_3) 。在这个例子中故障语句是语句 5, t_3 是失败测试用例。使用变异算子(如简单的算术运算符: +、-、* 和/) ,可以发现将该变异算子运用到故障程序 P' 中将产生六个变异体。变异程序 P' 中的语句 3 将产生三个变异体(P''_1: sum = $a - b$, P''_2: sum = $a * b$ 以及 P''_3: sum = a/b) ,而变异语句 5 将产生另外三个变异体(P''_4: sum = sum - 2, P''_5: sum = sum* 2 以及 P''_6: sum = sum + 2) 。其中只有 P''_5 成功执行所有用例。如果使用变异技术,则变异修复过程会先产生语句 3 的三个变异体并进行验证,再产生语句 5 的变异体直至找到潜在修复版本 P''_5 为止。此过程一共产生并验证五个变异体。

如果基于频谱的故障定位技术[11],首先计算出例子中各条可执行语句的可疑度系数,可疑度高的语句首先进行变异,则可降低修复的代价。例如,在一个语句对应的行中黑点代表这个语句被该测试用例覆盖,反之,未被该用例覆盖(如语句 5 被 t_1, t_3 覆盖但未被 t 覆盖) 。基于语句的覆盖信息及程序的

执行结果(成功/失败),根据文献[11]中的排名公式"O"计算出的语句可疑度在表3的最后一列,可以发现语句5的可疑度最高,因此变异修复过程将会更早地找到潜在修复(P''_5)而不必检查语句3产生的变异体。在最坏的情况下必须在P''_5之前先检查P''_4和P''_6,但是在最好的情况下,将会在其他变异体之前先检查P''_5。因此在最坏的情况下,必须检查三个变异体来修复故障,也即50%的总变异体数。但是在最好的情况下,仅需检查一个变异体就能成功修复故障,也即只需检查16.67%的总变异体数。

表3 变异修复结合故障定位举例

行号	期望程序(P)	故障程序(P')	覆盖情况			可疑度
			t_1	t_2	t_3	
1	read(a);	read(a);	·	·	·	0
2	read(b);	read(b);	·	·	·	0
3	sum = a + b;	sum = a + b;	·	·	·	0
4	if(sum < 100)	if(sum < 100)	·	·	·	0
5	sum = sum * 2;	sum = sum / 2;	·		·	1
6	print(sum);	print(sum);	·	·	·	0
t_1	输出:0	输出:0				
a=0						
b=0		执行成功				
t_2	输出:100	输出:100				
a=50						
b=50		执行成功				
t_3	输出:40	输出:10				
a=10		执行失败				
b=10						

结合故障定位技术和变异技术不仅能够自动化地修复故障,而且有效地降低了修复的代价,且故障定位技术的精度及效率越高修复的代价就越低。因此使用何种故障定位技术对程序自动化修复的效率有很大的影响。为了能很好地配合变异技术进行故障修复工作,本文实现的故障定位器采用了Naish等人[11]提出的基于频谱的故障定位技术。基于频谱的故障定位技术是目前最有效的自动化定位技术之一,排名公式是基于频谱的故障定位的关键,它计算各个语句可能是故障语句的排名,一个语句的排名越高表明该语句是故障语句的可能性越大。在基于频谱的故障定位中,已经有35种排名公式被提出,如最早提出的排名公式Tarantula[12,13]和最近提出的排名公式"0"[11],到目前为止,从理论和实践分析的结果来看,Naish等人提出的排名公式"0"是最优的,所以本文选择排名公式"0"对程序语句的可疑度进行排名。

2 程序故障修复工具的设计与实现

2.1 修复工具的设计

图2给出了程序故障自动化修复工具的框架图,其包含:

a) 故障定位模块。该模块利用代码覆盖工具gcov统计各条可执行语句的代码覆盖情况,收集给定测试用例集执行的程序代码的情况,对于每个程序语句而言,收集的信息包括:执行该语句并且执行结果成功的测试用例数(用a_{ep}表示);执行该语句并且执行结果失败的测试用例数(用a_{ef}表示);没有执行该语句并且执行结果成功的测试用例数(用a_{np}表示);没有执行该语句并且执行结果失败的测试用例数(用a_{nf}表示)。根据可疑度排名公式"0"(见式(1))计算出各条语句的可疑度,并根据计算结果对语句进行可疑度降序排序,生成语句可疑度排序列表文件。

$$O(a_{np},a_{nf},a_{ep},a_{ef}) = \begin{cases} -1 & a_{nf}>0 \\ a_{np} & \text{otherwise} \end{cases} \quad (1)$$

b) 故障修复模块。该模块依据故障定位模块得到的语句可疑度排序列表文件,每次选择未检查过的可疑度排序最靠前的可执行语句进行检查。对正在被检查的语句进行词法分析,发现适用变异算子的位置时则进行一次变异产生一个变异体。最后,在该变异体上执行成功测试用例和失败测试用例并将执行结果与测试用例的期望输出结果进行对比,以确定该变异体是否为一个潜在修复版本。若在停止前找到一个潜在修复版本则结束变异,则返回这个潜在修复版本程序,否则修复失败退出程序。

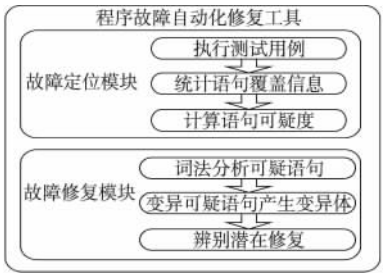

图2 修复工具框架结构

程序故障自动化修复工具执行流程如图3所示。首先用户分别输入故障C程序文件地址、成功测试用例集输入参数文件、期望输出文件地址、失败测试用例集、输入参数文件及期望输出文件地址;其次执行成功测试用例和失败测试用例,并记录执行中各语句的代码覆盖情况,根据代码覆盖信息计算各条语句的可疑度,生成语句可疑度排序文件,然后选取一个未检查过可疑度最高的语句进行词法分析,选择该条语句适用变异算子集,依据变异算子集产生一个变异体,最后生成潜在修复。

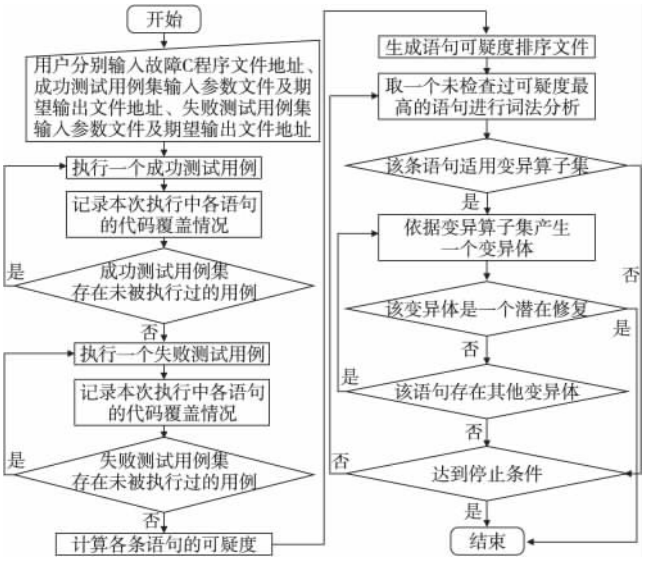

图3 程序故障自动化修复流程

2.2 修复工具的实现

本文采用C++编程语言在集成开发环境VC++ 6.0下编程实现了故障定位模块和故障修复模块。故障定位模块包括测试用例执行模块、语句覆盖信息统计模块以及语句可疑度计算模块。故障定位模块中几个主要函数的功能描述如表4所示。故障修复模块包括词法分析模块、变异模块以及变异体验模块。故障修复模块中几个主要函数功能描述如表5所示,工具的故障修复界面如图4所示。

表4 故障定位实现的主要函数说明

函数名	功能描述
findLineNo	解析gcov文件的一行数据,获取该行语句在故障程序中的行号
statisticSuccCase	统计执行成功测试用例集后各条可执行语句的覆盖情况,并返回总的成功测试用例数
statisticFailCase	统计执行失败测试用例集后各条可执行语句的覆盖情况,并返回总的失败测试用例数
caculateSuspi	根据可疑度计算公式计算各条可执行语句的可疑度

表 5　故障修复模块实现的主要函数说明

函数名	功能描述
fix	对可疑语句进行词法分析,发现适用变异算子的位置时则调用 mutate 函数进行变异及验证
mutate	应用参数指定的变异算子变异可疑语句产生可能的变异体,并验证这些变异体是否为潜在修复,需要调用 generateMutation 及 testMutation 函数
generateMutation	按参数指定的方式变异可疑语句产生一个变异体
testMutation	验证当前产生的变异体是否为一个潜在修复

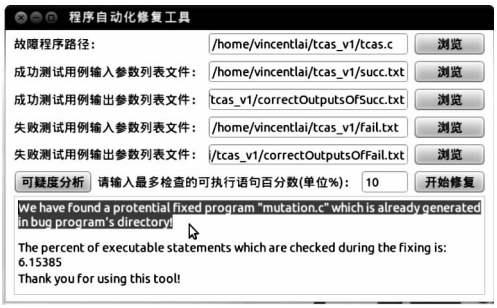

图 4　故障修复执行结果

3　实例验证

本章采用著名的 SIR 知识库[14]提供的实验对象,通过实际案例程序阐释了程序故障自动化修复方法,从而验证本文程序故障自动化修复方法的有效性。

3.1　实验案例

本节采用的七个研究案例均来自于西门子套件。这七个西门子套件 C 程序在故障定位及修复的研究中被广泛地使用。每个案例程序的正确版本和错误版本以及各自可用的测试用例集都可在文献[14]中下载得到。表 6 简要概述了实例验证过程中所使用的每个案例的程序名、功能、故障版本数、测试用例数以及正确版本的源码行数。

表 6　故障案例介绍

程序名	功能描述	故障版本数	测试用例数	代码行数
print_tokens	词法分析器	7	4 130	539
print_tokens2	词法分析器	10	4 115	489
schedule	优先级调度器	9	2 650	397
schedule2	优先级调度器	10	2 710	299
replace	模式替换	32	5 542	507
tcas	空中防撞系统	41	1 608	174
tot_info	信息统计	23	1 052	398

3.2　修复效果

修复工具的修复效果可以通过研究案例中 132 个故障程序最终被成功修复的数量来展现。各个案例修复结果的统计信息和修复耗时如表 7 所示。

表 7　案例修复结果统计

案例名	故障修复百分数	修复耗时(分′秒″)	案例名	故障修复百分数	修复耗时(分′秒″)
print_tokens	0% (0/7)	3′10″	replace	15.63% (5/32)	4′8″
print_tokens2	0% (0/10)	3′20″	tcas	24.39% (10/41)	45″
schedule	22.22% (2/9)	2′13″	tot_info	39.13% (9/23)	5′3″
schedule2	10% (1/10)	1′20″			

分析表 7 中的案例故障修复结果可以发现,修复工具能够对 20.45% 的故障程序进行自动化修复并产生它们所对应的期望程序。然而,从表中数据也可以观察到各个案例被修复故障的比例并不是一致的,有些案例修复的故障比较多,而有一些则比较少,甚至有的案例一个故障也没能修复。通过比较发现,tot_info 案例故障修复比例最大,达到 39.13%,修复效果主要在于待修复的故障类型以及修复工具所使用的变异算子集。例如,案例 print_tokens 中存在很多语句缺失故障,但是本文实现的修复工具所使用的变异算子集中并不包含针对语句缺失故障的变异算子,从而也就不能对这类故障进行修复,可以通过后续扩展变异算子集来解决。另外,修复效率较高,程序的代码行数为 100 ~ 600 之间,修复耗时的最小时长为 45 s,最大修复时长仅为 5 min。

4　结束语

本文提出了一种结合故障定位技术的程序变异修复新方法,并采用该方法阐释了 C 程序故障的变异修复机制,而且研制了 C 程序的程序故障自动化修复辅助工具,取得了一定的研究成果,通过对实际故障系统案例的分析证明了方法的有效性。通过本文研究成果,用户可以实现对 C 程序部分故障类型的自动化修复。同时,本文工作为大型 C 程序故障的自动化修复奠定了基础,为实现由局部到全局逐步解决整个应用系统故障的自动化修复问题创造了条件。未来,笔者将致力于变异算子的扩展以解决更多的故障类型,同时研究多故障 C 程序的修复问题。

参考文献:

[1] NIST. The economic impacts of inadequate infrastructure for software testing, planning report 02-3 [R]. 2002.

[2] BALLOU M C. Improving software quality to drive business agility [R/OL]. (2008-06). http://www.coverity.com/library/pdf/IDC_Improving_Software_Quality_June_2008.pdf.

[3] DALLMEIER V, ZELLER A, MEYER B. Generating fixes from object behavior anomalies [C]//Proc of International Conference on Automated Software Engineering. Washington DC: IEEE Computer Society, 2009: 550-554.

[4] PERKINS J H, KIM S, LARSEN S, *et al*. Automatically patching errors in deployed software [C]//Proc of the 22nd ACM SIGOPS Symposium on Operating Systems Principles. New York: ACM Press, 2009: 87-102.

[5] ARCURI A, YAO Xin. A novel co-evolutionary approach to automatic software bug fixing [C]//Proc of IEEE Congress on Evolutionary Computation. Washington DC: IEEE Computer Society, 2008: 162-168.

[6] FORREST S, WEIMER W, NGUYEN V, *et al*. A genetic programming approach to automated software repair [C]//Proc of Genetic and Evolutionary Computing Conference. New York: ACM Press, 2009: 947-954.

[7] WEIMER W, NGUYEN T, GOUES C, *et al*. Automatically finding patches using genetic programming [C]//Proc of the 31st International Conference on Software Engineering. Washington DC: IEEE Computer Society, 2009: 364-374.

[8] WEIMER W, FORREST S, GOUES C, *et al*. Automatic program repair with evolutionary computation [J]. Communications of the ACM, 2010, 53(5): 109-116.

[9] SCHULTE E, FORREST S, WEIMER W, *et al*. Automated program repair through the evolution of assembly code [C]//Proc of the 25th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM Press, 2010: 313-316.

[10] 何加浪,张琨,孟锦,等. 基于不变量的程序修复进化扩展模型[J]. 计算机应用研究, 2010, 27(12): 4533-4535.

[11] NAISH L, LEE H J, RAMOHANARAO K. A model for spectra-based software diagnosis [J]. ACM Trans on Software Engineering and Methodology, 2011, 20(3): 32.

[12] JONES J A, HARROLD M J, STASKO J. Visualization of test information to assist fault localization [C]//Proc of the 24th International Conference on Software Engineering. New York: ACM Press, 2002: 467-477.

[13] JONES J A, HARROLD M J. Empirical evaluation of the tarantula automatic fault-localization technique [C]//Proc of the 20th International Conference on Automated Software Engineering. 2005: 273-282.

[14] http://sir.unl.edu/portal/index.html [EB/OL].