一种基于数据流分析的冗余变异体识别方法

孙昌爱[1)] 郭新玲[1)] 张翔宇[2)] 陈宗岳[3)]

[1)](北京科技大学计算机与通信工程学院 北京 100083)

[2)](普渡大学计算机科学系 西拉法叶 47906 美国)

[3)](斯文本大学计算机科学与软件工程系 墨尔本 3122 澳大利亚)

摘 要 变异测试是一种基于故障的软件测试技术,广泛用来评估测试用例集的充分性与软件测试技术的有效性.尽管变异测试具有较强的故障检测能力,但由于数量庞大的变异体导致了计算开销大的问题,阻碍了变异测试在实践中的广泛应用.为了增强变异测试的实用性,该文从减少变异体数量和缩短变异测试执行时间的角度出发研究变异测试的优化技术,提出冗余变异体的概念和一种基于数据流分析的冗余变异体识别方法.采用 11 个 C 程序以经验研究的方式评估了所提冗余变异体识别方法的可行性与有效性.实验结果表明,该文提出的冗余变异体识别方法不仅可以识别出大量的冗余变异体,有效地减少了变异测试执行时间,还提供了一种评价变异算子质量的方法.该文提出的冗余变异体概念及其识别方法可以有效地提高变异测试的效率,并为资源受限的情况下如何选择变异算子进行变异测试提供了指导方针.

关键词 软件测试;变异测试;冗余变异体;数据流分析;变异优化

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2019.00044

A Data Flow Analysis Based Redundant Mutants Identification Technique

SUN Chang-Ai[1)] GUO Xin-Ling[1)] ZHANG Xiang-Yu[2)] CHEN Tsong-Yueh[3)]

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

[2)](*Department of Computer Science, Purdue University, West Lafayette* 47906, *USA*)

[3)](*Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne* 3122, *Australia*)

Abstract Mutation testing is a fault-based software testing technique, which is widely used to evaluate the adequacy of a given test suite or the fault detection effectiveness of a given software testing technique. Although mutation testing has a strong fault detection capability, the high computation cost incurred by a huge number of mutants prevents mutation testing from being widely adopted in practice. In order to improve the practicability of mutation testing, we proposed the concept of redundant mutants with the purpose of reducing the number of mutants and the execution of mutation testing, and proposed a data flow analysis based technique for identifying redundant mutants. We conducted an empirical study where eleven C programs were used to evaluate the feasibility and effectiveness of the proposed technique. The experimental results show that our technique can effectively identify a large number of redundant mutants, reduce the execution time of mutation testing, and provide a way to evaluate the mutant generation quality of mutation operators. The concept of redundant mutants and its identification technique proposed in this paper can not only improve the efficiency of mutation testing, but also provide useful guidelines for selecting mutation operators in the context of constrained resources available for mutation testing.

收稿日期:2017-08-29;在线出版日期:2018-01-19.本课题得到国家自然科学基金(61872039,61370061)、北京市自然科学基金(4162040)、航空科学基金(2016ZD74004)、中央高校基本科研业务费专项资金(FRF-GF-17-B29)资助.**孙昌爱**,男,1974 年生,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为软件测试、程序分析、服务计算、软件体系结构.E-mail:casun@ustb.edu.cn.**郭新玲**,女,1994 年生,硕士研究生,主要研究方向为软件测试.**张翔宇**,男,1976 年生,博士,教授,博士生导师,主要研究领域为程序分析、软件测试.**陈宗岳**,男,1951 年生,博士,教授,博士生导师,主要研究领域为软件测试、程序调试.

Keywords　software testing；mutation testing；redundant mutant；data flow analysis；mutation optimization

1　引　言

变异测试是一种基于故障的软件测试方法[1].对于给定的程序 p,通过合乎语法的变动产生的错误程序版本 p',称为 p 的一个变异体.用某个测试用例集运行原程序 p 和变异体 p',如果输出结果不同,则称变异体被“杀死”,即变异体相关的故障能够被检测出来.针对给定的测试用例集,被“杀死”的变异体数量占所有变异体数量的比例,称为该测试用例集的变异得分.变异得分可以用来评估测试用例集的测试充分性,也可以用来评估不同测试技术的有效性.大量的实验结果表明,变异测试具有较强的故障检测能力[2];而且,与手动植入的故障相比,自动生成的变异体更接近软件中潜藏的真实故障[3].遗憾的是,变异测试技术并没有被广泛应用于工业界,主要原因包括变异测试的计算开销大(变异体数量大、测试时间长)、等价变异体识别困难、缺乏有效的自动化变异测试工具等[4].

近年来,在如何降低变异测试的计算开销方面已经出现了大量的研究成果[5].这些研究工作从两个方面对传统的变异测试进行改进或优化:(1)变异体数量精简方法从如何减少变异体的数量角度优化变异测试,包括变异体随机选择[6]、选择性变异算子[7]、高阶变异测试[8]、变异体聚类方法[9];(2)加速变异执行方法从如何缩短变异测试执行的时间角度优化变异测试,主要方法有变异体检测优化[10]、变异体编译优化[11]、并行执行变异体[12].上述两类变异测试优化方法中,变异体数量精简方法虽然能够大量减少变异体数量,但在一定程度上影响测试充分性的评估.

Sun 等人[13]提出了一种路径感知的变异体精简技术,利用程序结构信息选择变异体.在进行实验评估时,我们发现从全部变异体集合 M 中选择出 1%变异体子集 M_1,然后用能“杀死”M_1 中全部变异体的测试用例集 T 检测 M 中全部变异体,T 的变异得分大于 95%(即杀死 95%以上的变异体).由此我们推测变异体集合可能存在某些变异体,这些变异体是否被“杀死”可从其它变异体的测试结果中推理获得.如果在测试执行前能识别并删除这些变异体,不仅可以减少变异体的数量,还可以减少变异测试的执行时间.

基于上述猜想,本文从不同变异体之间在测试结果上的蕴涵关系的角度,探索如何减少变异体的数量和缩短变异测试的执行时间.我们提出一种基于数据流分析的冗余变异体识别方法,通过分析程序中变量定义与变量使用间的依赖关系,定义一组冗余变异体识别规则.在变异测试执行前,采用静态分析技术识别并删除冗余变异体,减少变异体的数量,缩短变异测试的执行时间.最后,我们采用 11 个 C 程序以经验研究的方式评估了所提冗余变异体识别方法的可行性与有效性.

本文第 2 节介绍变异测试原理、数据流分析及程序块相关概念;第 3 节提出冗余变异体的概念,介绍基于数据流分析的冗余变异体识别方法;第 4 节采用经验研究的方式对所提方法进行评估;第 5 节介绍相关的工作;最后总结全文.

2　背景介绍

介绍变异测试、数据流分析、程序块等相关概念.

2.1　变异测试

变异测试是一种基于故障的软件测试技术,借助变异算子模拟特定类型的故障、通过程序变换的方式将故障“植入”到程序中,然后用给定的测试用例集检测植入的故障,通过变异得分评估测试用例集对特定故障类型检测的充分性程度.给定待测程序 p,生成变异体集合 $M=\{m_1,\cdots,m_k\}$,测试用例集 $T=\varnothing$,传统变异测试流程[14]描述如下:

(1)增加一个新的测试用例 t,添加到当前测试用例集 $T(T=T\cup\{t\})$;

(2)用 t 运行原始程序 p,生成期盼的输出结果;

(3)用 t 运行 M 中的所有变异体 m_i,比较 p 和 m_i 的输出结果.如果不同,m_i 被“杀死”;否则,m_i 继续“活着”;

(4)计算当前测试用例集 T 的变异得分;

(5)如果变异得分满足要求,变异测试结束;否则,重复步骤 1～4.

图 1 所示的传统变异测试流程中,若存在某个测试用例能导致变异体和原始程序产生不同的输出,则称变异体被“杀死”.判断变异体被“杀死”必须同时满足如下 3 个条件:(1)可达性:变异点能够执

行到;(2)必要性:变异点执行后程序状态发生改变;(3)充分性:状态改变能够传播到程序输出[15].采用上述方法的变异测试称为强变异测试.弱变异测试[16]判断变异体被杀死只需满足上述条件(1)和(2).与强变异测试相比,弱变异测试缩短了变异测试的时间,本文在弱变异测试的基础上考虑变异测试执行时间的优化.

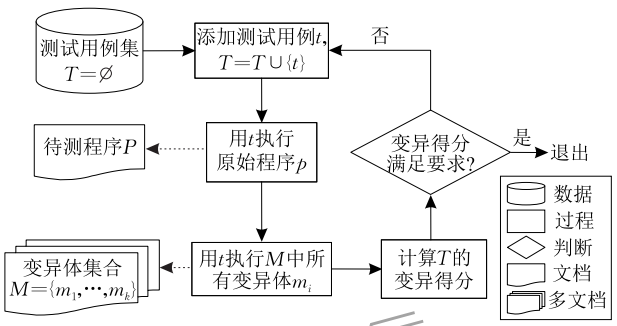

图 1 传统变异测试流程

2.2 数据流分析

数据流分析关注变量的定义和使用情况,在编译优化、程序验证、程序调试等领域应用广泛[17].我们介绍数据流分析相关的基本定义[18].

定义 1(程序依赖图 PDG). PDG 是一个有向图,表示为一个四元组$\langle N,E,entry,exit\rangle$.其中,$N$是非空节点集合,一个节点为一个或多个无分支的语句;E是有向边的集合,表示程序组成部分间的依赖关系,包括控制依赖和数据依赖;$entry$是图的唯一入口节点;$exit$是图的唯一出口节点.

定义 2(变量定义性出现). 设s是程序p中的一条赋值语句,v为s中被赋值的变量,则称v在s中定义性出现,记作$def(v,s)$.

定义 3(变量使用性出现). 设s是程序p中的一条语句,v为s出现的变量,且没有修改v的值,则称v在s中使用性出现,记作$use(v,s)$.

定义 4(定义-使用链). 假设 PDG 中存在一条从$def(v,s_1)$到$use(v,s_2)$的路径,且该路径上不存在$\mathrm{def}(v,s')(s_1<^+s'<^+s_2)$,则存在变量$v$的一个定义-使用链,记作$du(v,s_1,s_2)$.其中,$s_i<^+s_j$表示$s_i$在$s_j$之前执行.

2.3 程序块相关概念

本文在程序块层次上定义冗余变异体识别规则,下面介绍程序块相关概念[13,19-20].

定义 5(基本块). $BasicBlock=\{Statement_{i,\cdots,j}|(i\leq j)\wedge(\neg\exists n((n<i)\wedge(Statement_n\prec Statement_i)))\wedge(\neg\exists m((m>j)\wedge(Statement_j\prec Statement_m)))\}$;其中,$Statement_i$表示程序中的逻辑语句行$i$,$Statement_{i,\cdots,j}$表示由逻辑语句$i$到$j$构成的逻辑语句段;$Statement_x\prec Statement_y$当且仅当$Statement_x$执行后立即执行$Statement_y$.

定义 6(调用模块与被调用模块). 如果模块m调用了模块n(记作$m\rightarrow n$),称m是n的调用模块,记作$caller$;相应地,n是m的被调用模块,记作$callee$.若m调用n时有变量v传递,将m调用n记作$m\xrightarrow{v}n$.

定义 7(选择块). 表示谓词表达式φ及其控制的代码区域$Region_i$所组成的执行区域.$ChoiceBlock=(\varphi)\prec Region_i$.其中,$\varphi$表示谓词表达式.$Region_i$表示基本块、选择块、循环块、及其复合形成的一个控制依赖区域.

定义 8(循环块). 表示当循环谓词表达式满足时,选择重复执行的代码区域$Region_i$.$LoopBlock=((\varphi)\prec Region_i)^+$.其中,$\varphi$表示谓词表达式.$Region_i$表示基本块、选择块、循环块、及其复合形成的一个控制依赖区域.$^+$表示执行区域$(\varphi)\prec Region_i$执行一次或多次.仅当谓词表达式φ为 true 时,$Region_i$才会执行.

定义 9(上层块关系). b_j是b_i的上层块关系,记作$b_j=UpperBlock(b_i)$,当且仅当满足以下条件:

(1) $b_i\in BasicBlock\vee ChoiceBlock\vee LoopBlock$;

(2) $b_j\in BasicBlock\vee ChoiceBlock\vee LoopBlock$;

(3) 如果b_j执行可能引起b_i执行,而且b_i执行后b_j继续执行.

定义 10(顺接块关系). b_j是b_i的顺接块关系,记作$b_j=NextBlock(b_i)$,当且仅当满足以下条件:

(1) $b_i\in BasicBlock\vee ChoiceBlock\vee LoopBlock$;

(2) $b_j\in BasicBlock\vee ChoiceBlock\vee LoopBlock$;

(3) 如果b_i执行后b_j立即执行(即$b_i\prec b_j$),但b_j执行不会导致b_i执行.

3 基于数据流分析的冗余变异体识别

首先给出冗余变异体的定义,然后介绍基于数据流分析的冗余变异体识别方法,包括方法框架、识别规则、识别算法和方法示例.

3.1 冗余变异体

在变异测试中,通过变异得分评估测试用例集的充分性.变异得分指被"杀死"的变异体数量与全部变异体数量的比例.如果某个变异体能否被"杀死"

可由其它变异体的检测结果得知，那么我们可以不必重复执行这些变异体. 例如，图 2 示意了程序 p 和两个变异体 m_1 和 m_2. 其中，m_1 和 m_2 分别在 p 的第 2、3 行发生变异. 为了便于讨论，假设该程序仅包括变量 x 和 y，程序在某条语句执行后的状态则可以用变量 x 和 y 的值表示，那么 p，m_1 和 m_2 在第 1 行的状态为 $p^1(x,y)=m_1^1(x,y)=m_2^1(x,y)=\{x=0,y=0\}$；$p$ 在第 3 行的状态为 $p^3(x,y)=\{x=1,y=4\}$；m_1 在第 3 行的状态为 $m_1^3(x,y)=\{x=2,y=5\}$、m_2 在第 3 行的状态为 $m_2^3(x,y)=\{x=1,y=5\}$. 虽然 m_1 和 m_2 的变异位置不同，但它们对 p 的状态改变相似（即改变变量 y 的值）. 在进行变异测试时，任意的测试用例 t 如能触发上述代码的执行，则能杀死 m_1 和 m_2. 在计算变异得分时，由于 m_2 能否被杀死可由 m_1 的测试结果获得，因此仅需要执行 m_1，而不必执行变异体 m_2. 基于上述讨论，冗余变异体的定义如下：

原程序p	变异体m_1	变异体m_2
1：int x, y;	1：int x, y;	1：int x, y;
2：x=1;	**2：x=2;**	2：x=1;
3：y=x+3;	3：y=x+3;	**3：y=x+4;**
4：…	4：…	4：…

图 2　冗余变异体程序示例

定义 11（冗余变异体）.　给定待测程序 p 和测试用例集 T，变异体 m_2 是变异体 m_1 的冗余变异体，记作 $m_1 \mapsto m_2$（相应地，称 m_1 为 m_2 的源变异体），当且仅当如下命题恒真：

$$\forall t \in T(m_1(t) \neq p(t) \rightarrow m_2(t) \neq p(t)),$$

其中 $p(t)$、$m_1(t)$、$m_2(t)$ 分别表示用测试用例 t 执行待测程序 p、变异体 m_1、变异体 m_2 变异点后的程序状态. 特别地，为了进一步减少变异测试的执行时间，本文在弱变异的基础上定义冗余变异体. 为了减少变异体的数量和缩短变异测试的执行时间，可以将冗余变异体从变异体集合中删除.

3.2　基于数据流分析的冗余变异体识别方法

根据冗余变异体的定义可知，冗余变异体是否能被“杀死”可从其源变异体的测试结果中推理获得. 为了识别冗余变异体，则需要分析源变异体和冗余变异体植入的故障代码是否有相同的机会被触发（即源变异体和冗余变异体的变异点执行具有相同的执行路径），而且在变异点执行后程序状态都发生改变. 由于程序状态改变可以通过程序的状态变量进行刻画，数据流分析提供了一种识别冗余变异体的可行途径.

提出的基于数据流分析的冗余变异体识别方法（简称 ReMuDF）如图 3 所示. 首先，分析待测程序 p 的程序结构，得到 p 的块规则文件（记录每个程序块的编号、起止行号）；然后，逐个比较变异体程序和源程序，得到每个变异体的变异位置，根据块规则文件判断变异体的块类别；其次，对 p 进行数据流分析，生成变量定义、变量使用、定义-使用链等数据流信息；最后，根据预定义的冗余变异体识别规则，结合变异体的块类别和程序数据流信息，识别出冗余变异体集合.

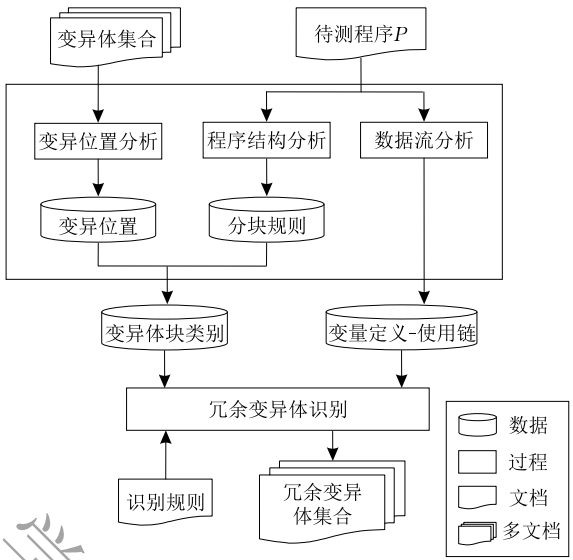

图 3　基于数据流的冗余变异体识别技术原理图

需要说明的是，前期开发的变异测试精简工具[13]支持程序结构分析和变异位置分析；借助静态分析工具 Frama-C 可以实现程序数据流分析[21]. 下面，我们讨论 ReMuDF 的关键问题，即如何定义和实现冗余变异体识别规则.

3.3　冗余变异体的识别规则

为了便于规则的描述，我们首先定义如下两个术语：

定义 12（定义变异集）.　假设变量 v 在语句 s 处定义性出现（$def(v,s)$），则将 s 处生成的变异体集合 M 称为变量 v 的定义变异集，记作 $M(def,v,s)$.

定义 13（使用变异集）.　假设变量 v 在语句 s 处使用性出现（$use(v,s)$），则将 s 处针对变量 v 生成的变异体集合 M 称为变量 v 的使用变异集，记作 $M(use,v,s)$.

通过静态分析识别冗余变异体时，首先应限定那些变异位置相近的变异体集合；其次，应限定那些具备相同的前置路径条件（即对任何测试用例而言，变异点要么都被执行，要么都不被执行）的变异体集

合;最后,应限定那些在变异点后发生类似的程序状态改变的变异体集合.在识别冗余变异体时,应将分析点设置在变量使用后,确保源变异体和冗余变异体的变异点之间存在从变量定义到变量使用的可达路径.根据上述原则,我们基于数据流分析技术在程序块的层次上,分别从块内、块间(顺接块、上层块)、模块间四个维度定义了如下冗余变异体识别规则.

识别规则 1(R_1). 假设存在一条定义-使用链 $du(v,s_1,s_2)$,变异体 m_1 和 m_2 分别为变量 v 的定义变异集 $M(def,v,s_1)$ 和使用变异集 $M(use,v,s_2)$ 的一个变异体,若 s_1 和 s_2 属于同一个基本块 $BasicBlock_i$,则 m_2 是 m_1 的冗余变异体(即 $m_1 \mapsto m_2$).

R_1 限定了源变异体 m_1 和冗余变异体 m_2 属于同一个基本块.具体说来,m_1 中变量定义性出现 $def(v,s_1)$ 所在的语句 s_1 与 m_2 中变量使用性出现 $use(v,s_2)$ 所在的语句 s_2 属于 $BasicBlock_i$,而且存在定义-使用链 $du(v,s_1,s_2)$.根据数据流分析技术,如果某个测试用例能够触发 m_1 中 s_1 的执行,也必然可以触发 m_2 中 s_2 的执行,而 $du(v,s_1,s_2)$ 则可以保证 s_1 处的错误状态能传播到 s_2.因此,根据冗余变异体的定义,将 m_2 识别为 m_1 的冗余变异体.

图 4 示例了 R_1 的应用.变量 $temp$ 在变异体 m_1 的定义性出现 $def(temp,4)$ 和变异体 m_2 的使用性出现 $use(temp,6)$ 处在同一个基本块(第 4~6 行).m_1 属于 $temp$ 的定义变异集 $M(def,temp,4)$,对程序状态的直接影响是变量 $temp$ 的值加 1,并通过 $temp$ 的使用将上述影响传播到第 6 行使变量 y 的值加 1.m_2 属于 $temp$ 的使用变异集 $M(use,temp,6)$,对程序状态的改变是变量 $temp$ 和变量 y 的值加 1.由于存在定义-使用链 $du(temp,4,6)$,两个变异体对程序在第 6 行的状态改变相同,因此变异体 m_2 是变异体 m_1 的冗余变异体.

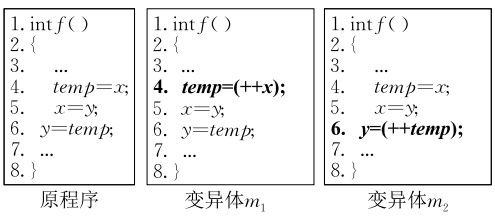

图 4 规则 1 示例程序

识别规则 2(R_2). 假设存在一条定义-使用链 $du(v,s_1,s_2)$,变异体 m_1 和 m_2 分别为变量 v 的定义变异集 $M(def,v,s_1)$ 和使用变异集 $M(use,v,s_2)$ 的一个变异体,若 s_1 属于程序块 b_i(b_i 的类型为基本块、选择块或循环块的三者之一),s_2 属于基本块 $BasicBlock_j$,b_i 和 $BasicBlock_j$ 满足顺接块关系(即 $s_1 \in b_i$,$s_2 \in BasicBlock_j$,$BasicBlock_j = (NextBlock(b_i))^+$),则 $m_1 \mapsto m_2$.

R_2 限定了源变异体 m_1 与冗余变异体 m_2 属于顺接块关系.m_1 属于 $M(def,v,s_1)$,而 m_2 属于 $M(use,v,s_2)$,变量定义性出现 $def(v,s_1)$ 所在块 b_i,变量使用性出现 $use(v,s_2)$ 所在的基本块 $BasicBlock_j$,b_i 和 $BasicBlock_j$ 满足直接或间接顺接块关系.由定义 10 可知,b_i 执行完一定会执行其顺接块 $BasicBlock_j$.$du(v,s_1,s_2)$ 则可以保证 s_1 处的错误状态能传播到 s_2,因此将 m_2 识别为 m_1 的冗余变异体.

图 5 示例了 R_2 的应用.变量 y 在变异体 m_1 中定义性出现 $def(y,3)$ 和在变异体 m_2 中一个使用性出现 $use(y,6)$.m_1 属于变量 y 的定义变异集 $M(def,y,3)$,对程序状态的影响是变量 x 和 y 的值改变,通过 y 的使用 $use(y,6)$ 将上述影响传播到第 6 行,使变量 z 的值发生改变.m_2 属于变量 y 的使用变异集 $M(use,y,6)$,对程序状态的影响是变量 y 和 z 的值改变.m_1 和 m_2 分别属于基本块 b_1 和 b_3,且 b_3 是 b_1 的间接顺接块(b_1 的直接顺接块为选择块 b_2,b_2 的直接顺接块为 b_3).由于存在定义-使用链 $du(y,3,6)$,m_1 和 m_2 对程序在第 6 行的状态改变相同,因此 m_2 是 m_1 的冗余变异体.

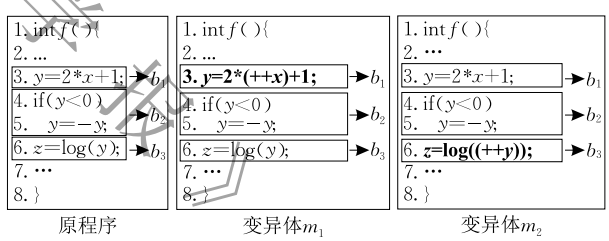

图 5 规则 2 示例程序

识别规则 3(R_3). 假设存在一条定义-使用链 $du(v,s_1,s_2)$,变异体 m_1 和 m_2 分别为变量 v 的定义变异集 $M(def,v,s_1)$ 和使用变异集 $M(use,v,s_2)$ 的一个变异体,若 s_1 属于程序块 b_i(b_i 的类型为基本块、选择块或循环块的三者之一),s_2 属于基本块 $BasicBlock_j$,$BasicBlock_j$ 是 b_i 的上层块的顺接块(即 $s_1 \in b_i$,$s_2 \in BasicBlock_j$,$BasicBlock_j = (NextBlock(UpperBlock(b_i))^+)^+$),则 $m_1 \mapsto m_2$.

R_3 限定了源变异体 m_1 与冗余变异体 m_2 属于上层块关系和顺接块关系的复合.m_1 属于 $M(def,v,s_1)$,而 m_2 属于 $M(use,v,s_2)$,变量定义性出现 $def(v,s_1)$ 所在块 b_i,变量使用性出现 $use(v,s_2)$ 所在的基本块 $BasicBlock_j$,b_i 和 $BasicBlock_j$ 满足直接或间接上层块和顺接块的复合关系.由定义 9 可知,b_i 执行完一

定会执行其上层块，然后执行顺接块 $BasicBlock_j$. $du(v,s_1,s_2)$则可以保证 s_1处的错误状态能传播到 s_2，因此将 m_2识别为 m_1的冗余变异体.

图 6 示例了 R_3的应用. 变量 sum 在变异体 m_1 中的定义性出现 $def(sum,4)$和在变异体 m_2中的一个使用性出现 $use(sum,6)$. m_1属于变量 sum 的定义变异集 $M(def,sum,4)$，对程序状态的影响是 sum 的值变为负值，通过 sum 的使用 $use(sum,6)$将上述影响传播到第 6 行，使变量 $result$ 的值变为负值. m_2 属于变量 sum 的使用变异集 $M(use,sum,6)$，对程序状态的影响是使变量 sum 和变量 $result$ 的值均变为负值. m_1和 m_2分别属于基本块 b_2和基本块 b_3，循环块 b_1是 b_2的上层块，且 b_3是 b_1的顺接块，所以程序在 b_1执行后，执行 b_2，然后再执行 b_1，最后执行 b_3. 由于存在定义-使用链 $du(sum,4,6)$，m_1和 m_2对程序在第 6 行的状态改变相同，因此 m_2是 m_1的冗余变异体.

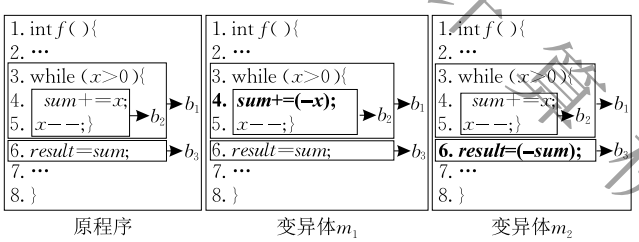

图 6　规则 3 示例程序

识别规则 4(R_4).　假设存在一条定义-使用链 $du(v,s_1,s_2)$，变异体 m_1和 m_2分别为变量 v 的定义变异集 $M(def,v,s_1)$和使用变异集 $M(use,v,s_2)$的一个变异体，若 s_1是模块 m 中以变量 v 为参数的一条函数调用语句 $m \xrightarrow{v} n$，s_2属于模块为 n 中的一个基本块 $BasicBlock_j$，且 $UpperBlock(BasicBlock_j)=\varnothing$，则 $m_1 \mapsto m_2$.

R_4限定了源变异体 m_1与冗余变异体 m_2属于跨模块关系. m_1属于 $M(def,v,s_1)$，而 m_2属于 $M(use,v,s_2)$，变量定义性出现 $def(v,s_1)$所在的块为 $b_i \in m$，变量使用性出现 $use(v,s_2)$所在的基本块 $BasicBlock_j \in n$，b_i和 $BasicBlock_j$之间存在调用关系 $m \xrightarrow{v} n$，且 $BasicBlock_j$在模块 n 中不存在上层块，因此，b_i执行完一定会执行 $BasicBlock_j$. $du(v,s_1,s_2)$则可以保证 s_1处的错误状态能传播到 s_2，因此将 m_2识别为 m_1的冗余变异体.

图 7 示例了 R_4的应用. 函数 $f()$在第 3 行调用函数 $area()$，并有参数 r 传递. 变量 r 在变异体 m_1 中定义性出现 $def(r,3)$和变异体 m_2中使用性出现 $use(r,8)$. m_1属于变量 r 的定义变异集 $M(def,r,3)$，对程序状态的影响是改变变量 r 的值，并通过函数调用和变量使用 $use(r,8)$，改变变量 ar 的值. m_2 属于变量 r 的使用变异集 $M(use,r,8)$，对程序状态的影响是改变变量 ar 的值. 由于 m_2使用性出现 $use(r,8)$属于基本块 $BasicBlock_j$且 $BasicBlock_j$没有上层块，因此函数 $area()$被调用时必然执行 $BasicBlock_j$. 由于存在定义-使用链 $du(r,3,8)$，m_1和 m_2对程序在第 8 行的状态改变相似，因此 m_2是 m_1的冗余变异体.

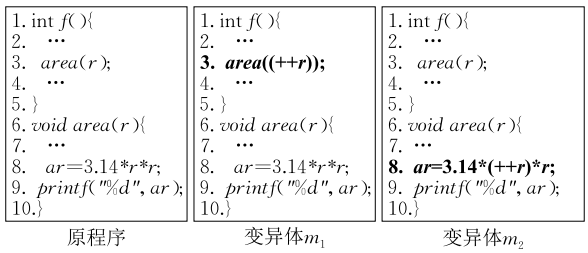

图 7　规则 4 示例程序

本文提出的冗余变异体识别方法基于静态程序分析技术，无需执行测试用例，因此冗余变异体识别不受测试用例集影响.

3.4　识别算法

基于冗余变异体识别规则 $R_1 \sim R_4$，我们提出了冗余变异体识别算法(算法 1). 该算法的输入为待测程序的函数集合 F、变异体集合 M、块结构信息 B 和定义-使用链集合 DU. 其中，函数 f_i指明了该函数的起止行号，m_i指明了变异发生的位置 s_i和变异操作(所属的定义变异集 $M(def,v,s)$或使用变异集 $M(use,v,s)$)，b_i指明了块的起止行号与类型，定义-使用链 $du(v_i,s_{1i},s_{2i})$描述了变量 v_i定义性出现所在的语句 s_{1i}和使用性出现所在的语句 s_{2i}. 算法输出为冗余变异体集合 M_{re}，其元素为源变异体和对应冗余变异体的二元组$\langle m_i,m_j\rangle$.

该算法首先初始化冗余变异体集合为空(1)；然后分析每个函数 f_i中各个变量的 $du(v,s_1,s_2)$，获取变量 v 的定义变异集 $M(def,v,s_1)$和使用变异集 $M(use,v,s_2)$，并判断其定义性出现所在语句 s_1和使用性出现所在语句 s_2所属的程序块(5～7)；依据识别规则 $R_1 \sim R_4$识别冗余变异体(8～38)；最后输出冗余变异体集合(42). 其中，$AddReMu(M(def,v,s_1),M(use,v,s_2))$将满足条件的定义变异集 $M(def,v,s_1)$中的变异体映射为源变异体 m_i，将使用变异集 $M(use,v,s_2)$中的变异体映射为冗余变异体 m_j，然后将二元组$\langle m_i,m_j\rangle$添加到变异体集合中.

算法 1. 基于数据流分析的冗余变异体识别算法.

输入:$M=\{m_1,m_2,\cdots,m_L\}$,$F=\{f_1,f_2,\cdots,f_U\}$,$B=\{b_1,b_2,\cdots,b_V\}$,$DU=\{du(v_1,s_{11},s_{21}),du(v_2,s_{12},s_{22}),\cdots,du(v_q,s_{1q},s_{2W})\}$

输出:$M_{re}=\{\langle m_{i1},m_{j1}\rangle,\langle m_{i2},m_{j2}\rangle,\cdots,\langle m_{in},m_{jn}\rangle\}$

```
1.  INITIALIZE M_re to an empty set
2.  FOR each f_j in F DO
3.   FOR each variable v in f_j DO
4.    FOR each du(v,s1,s2) of v in DU DO
5.     M(def,v,s1)←du(v,s1,s2)
6.     M(use,v,s2)←du(v,s1,s2)
7.     b_i←s1, b_j←s2
8.   IF ((b_i==b_j) && (type(b_j)==BasicBlock)
9.    AddReMu(M(def,v,s1),M(use,v,s2))
10. R1
11.   CONTINUE
12.   END IF
13.   IF (type(b_j)==BasicBlock)&&(b_i <+ b_j))
14.    DO
15.     IF (b_j==NextBlock(b_i))
16.      AddReMu(M(def,v,s1),M(use,v,s2))
17.      BREAK
18.     END IF
19.     b_i=NextBlock(b_i)
20.    WHILE (b_i <+ b_j)
21.   END IF
22.   IF (type(b_j)==BasicBlock) && (b_i <+ b_j))
23.    DO
24.     b_k=UpperBlock(b_i)
25.     DO
26.      IF (b_j==NextBlock(b_k))
27.       AddReMu(M(def,v,s1),M(use,v,s2))
28.       BREAK
29.      END IF
30.      b_k=NextBlock(b_k)
31.     WHILE (b_k <+ b_j)
32.     b_i=UpperBlock(b_i)
33.    WHILE (b_i <+ b_j)
34.   END IF
35.   f_i←b_i, f_j←b_j
36.   IF (f_i --v--> f_j && UpperBlock(b_j)==Φ
37.       && type(b_j)==BasicBlock))
38.  R4
39.    AddReMu(M(def,v,s1),M(use,v,s2))
40.    CONTINUE
41.   END IF
42.  END FOR
43. END FOR
44. END FOR
45. RETURN M_re
```

R1: R_1; R2: R_2; R3: R_3; R4: R_4

$AddReMu(M(def,v,s_1),M(use,v,s_2))$

```
1. FOR each m_j in M(use,v,s2) DO
2.  Randomly select m_i from M(def,v,s1)
3.  M_re=M_re ∪ {⟨m_i,m_j⟩}
4. ENF FOR
5. RETURN M_re
```

假设待测程序的变异体数量为 L,定义-使用链的个数为 W. 算法 1 中,针对每一个 $du(v,s_1,s_2)$,需要遍历整个变异体集合得到变量 v 的定义变异集和使用变异集,其时间复杂度为 $O(L)$;当 $du(v,s_1,s_2)$ 满足 $R_1\sim R_4$ 时需要遍历变量 v 使用变异集中的所有变异体,其时间复杂度为 $O(L)$. 算法 1 的时间复杂度是 $O(W\times L)$.

3.5 方法示例

采用一个程序实例(西门子程序集中的 tot_info 程序)示例本文提出的基于数据流分析的冗余变异体识别方法. 图 8 示意了 tot_info 程序实例的部分代码和使用 *Proteum* 生成的部分变异体集合. ReMuDF 识别变异体的过程简述如下.

```
80 double LGamma(x)
81 double x;
82 {
83    static const double cof[6]={
84    76.18009173, -86.50532033, 24.01409822,
85     -1.231739516, 0.120858003e-2, -0.536382e-5
86    };
87    double tmp=0.0, ser=0.0;
88    int j=0;
89    if (x<0.0){                                          → b2 → b1
90     double pix=3.1415926*x;
91     return log(pix/sin(pix))-LGamma(1.0-x);
       //return(log((pix/sin(pix)))-LGamma((5.5-x))); m2057
92    }
93    tmp=x+5.5;
       //(tmp-=(1.0+5.5));                                m2086
94    tmp-=(x+0.5)*log(tmp);                               → b3
       //(tmp-=((x+0.5)*log((10E-10))));                  m2016
95    ser=1.0;
96    for (j=0; j<6; ++j)                                  → b4
97     ser+=cof[j]/++x;                                    → b5
       //(ser+=(0.5/(++x)));                              m2107
98    return-tmp+log(2.50662827465*ser);                   → b6
       //return((10E10)+log((2.50662827465*ser)));        m2279
       //return((-tmp)+log((10E10)));                     m2022
99 }
```

图 8 tot_info 部分代码和变异体示例

(1)程序块结构分析:该程序的块结构信息如下:b_2、b_3、b_5、b_6 为基本块,b_1 为选择块,b_4 为循环块,b_1 是 b_2 的上层块,b_4 是 b_5 的上层块.

(2)变异位置分析:对比原程序和变异体,得到每个变异体的变异位置,判断变异体的变异位置的块类别. 例如,m_{2057} 的变异位置是 91 行,属于基本块 b_2;m_{2022} 的变异位置是 98 行,属于基本块 b_6.

(3)获取数据流信息:使用 Frama-C 获取程序中的变量定义、变量使用信息,并生成定义-使用链信息. LGamma()函数中有 x、tmp、ser 等变量. 其中,tmp 在 93、94 行是定义性出现($def(tmp,93)$、

$def(tmp,94)$)，在 94、98 行是使用性出现($use(tmp,94)$、$use(tmp,98)$)，构成两条定义-使用链：$du(tmp,93,94)$和 $du(tmp,94,98)$. 类似的，对于变量 ser 有 $def(ser,97)$和 $use(ser,98)$，构成定义-使用链：$du(ser,97,98)$.

(4) 识别冗余变异体：分别采用 $R_1 \sim R_4$ 进行冗余变异体识别.

① 识别规则 R_1：m_{2086} 和 m_{2016} 分别属于变量 tmp 的定义变异集 $M(def,tmp,93)$和使用变异集 $M(use,tmp,94)$，且二者的变异位置属于同一个基本块 b_3. m_{2016}是 m_{2086}的冗余变异体.

② 识别规则 R_2：m_{2016} 和 m_{2279} 分别属于变量 tmp 的定义变异集 $M(def,tmp,94)$和使用变异集 $M(use,tmp,98)$，m_{2016}的变异位置位于基本块 b_3，b_4 为 b_3的顺接块；m_{2279}的变异位置位于基本块 b_6，b_6是 b_4的顺接块. m_{2279}是 m_{2016}的冗余变异体.

③ 识别规则 R_3：m_{2107}和 m_{2022}分别属于变量 ser 的定义变异集 $M(def,ser,97)$和使用变异集 $M(use,ser,98)$，m_{2107}的变异位置位于基本块 b_5，其上层块是循环块 b_4；m_{2022}的变异位置位于基本块 b_6，b_6是 b_4的顺接块. m_{2022}是 m_{2107}的冗余变异体.

④ 识别规则 R_4：m_{2057}和 m_{2086}分别属于变量 x 的定义变异集 $M(def,x,91)$和使用变异集 $M(use,x,93)$. m_{2057}中函数 LGamma()在 91 行调用函数 LGamma(x)，而 m_{2086}中变量 x 使用性出现所处块 b_3的上层块为空. m_{2086}是 m_{2057}的冗余变异体.

4　实验评估

采用经验研究的方式评估基于数据流分析的冗余变异体识别方法的可行性和有效性.

4.1　研究问题

本文通过实验评估试图回答如下 5 个研究问题：

(1) 变异测试中是否存在冗余变异体？

通过对多个 C 程序进行变异测试，验证我们提出的存在冗余变异体的猜想是否正确.

(2) 提出的冗余变异体识别技术能否有效减少变异体的数量和缩短变异测试的执行时间？

验证所提的冗余变异体识别技术能否正确识别出多个 C 程序变异体集合中存在的冗余变异体. 通过比较删除识别出的冗余变异体前后的变异测试的执行时间，评估本文方法在减少变异体数量和缩短变异测试的执行时间方面的有效性.

(3) 提出的冗余变异体识别方法中不同识别规则的有效性如何？

本文提出的冗余变异体识别技术中定义了四种识别规则. 通过比较不同识别规则识别出的冗余变异体数量，评估他们的有效性. 借助评估结果，可以进一步推荐冗余变异体识别规则的应用优先级.

(4) 冗余变异体识别如何影响整个变异测试的效率？

本文提出的冗余变异体识别技术需要在变异测试之前进行静态程序分析. 通过比较基于数据流分析的冗余变异体识别时间与删除冗余变异体后的变异测试的执行时间，进一步评估冗余变异体识别对变异测试效率的影响.

(5) 从冗余变异体角度，如何度量和比较变异算子的质量？

通过分析识别出的冗余变异体在不同变异算子上的分布情况，评估变异算子质量. 该评估结果可为资源受限的情况下如何选择高质量变异算子(即减少冗余变异体)提供一种参考依据.

4.2　度量指标

本文采用变异体冗余率 RR 和执行加速比 TA 衡量冗余变异体识别技术的有效性. 其中，变异体冗余率表示所识别出的冗余变异体占所有变异体的比例，其计算公式如下：

$$RR=\frac{RN}{TN-EN}\times 100\%,$$

其中，RN 表示识别的冗余变异体的数量；TN 表示变异体总数；EN 表示等价变异体数量. 冗余率越高，意味着变异体集中存在越多的冗余变异体，变异测试的效率提升空间较大.

执行加速比表示采用冗余变异体识别前后，相对传统变异测试的执行时间而言，变异执行效率的提升百分比，其计算公式如下：

$$TA=\frac{TMT-NMT}{TMT}\times 100\%,$$

其中，NMT 表示识别冗余变异体后变异测试的执行时间；TMT 表示传统变异测试的执行时间. 执行加速比越大，意味着本文提出的方法优化效果越明显.

变异算子的冗余率 $MORR$ 表示某个变异算子 op 生成的冗余变异体与其生成的所有非等价变异体的比例. $MORR$ 可以用来评价某个变异算子 op 的质量，计算公式如下：

$$MORR(op)=\frac{RN_{op}}{TN_{op}-EN_{op}}\times 100\%,$$

其中，RN_{op}表示变异算子 op 生成的冗余变异体数量；TN_{op}表示 op 生成的变异体总数；EN_{op}表示 op

生成的等价变异体数量.变异算子的冗余率越高,意味着该变异算子生成冗余变异体的几率越高,生成的变异体的质量较低.

为了衡量冗余变异体识别对变异测试效率的影响.我们引入了冗余变异体识别能效比 *CER*. *CER* 表示引入冗余变异体识别技术后变异测试的执行时间与冗余变异体识别时间的比例,其计算公式如下:

$$CER=\frac{NMT}{RIT},$$

其中,*NMT* 表示识别冗余变异体后变异测试的执行时间;*RIT* 表示识别冗余变异体所用时间.冗余变异体识别能效比越大,意味着冗余变异体识别对变异测试效率的影响越小.

4.3 实验对象

采用三组 C 程序集作为实验对象[13]:(1)科学计算程序集:minmax 获取一组数据的最大值和最小值;bubble 实现冒泡排序算法;nextdate 计算下一天的日期;(2)西门子程序集[22]:tcas 为一个飞行器防碰撞程序;schedule 和 schedule2 为两个调度程序;tot_info 针对指定的输入数据生成统计信息;print_tokens 和 print_tokens2 为两个词法分析器;replace 完成模式匹配和替换;(3)space 程序集:针对数组定义语言(ADL)的一个解释器.表 1 总结了实验对象的基本信息.需要说明的是:由于 space 程序规模较大,选取该程序被调次数最多的三个函数进行了变异测试.

表 1 实验程序的基本信息

实验对象	代码行数	测试用例数量	变异体总数	单变体数量	等价变异体数量	适用变异体数量
minmax	33	60	286	256	30	225
bubble	24	75	336	304	24	279
nextdate	83	377	774	714	135	579
tcas	137	1052	4308	2488	442	2044
tot_info	281	1608	6478	4439	678	3654
schedule	296	2650	3516	1648	204	1441
schedule2	263	2710	5794	2521	471	2045
print_tokens	343	4130	10881	4238	583	3619
print_tokens2	355	4115	9369	4046	545	3478
replace	513	5542	21703	9361	1556	7805
space	5982	13467	9380	9139	1079	7986

本文实验评估所用测试用例集的构造分为两步:首先根据等价类划分、边界值分析设计部分测试用例;然后采用语句覆盖、分支覆盖、定义-使用覆盖等补充测试用例,针对每一个覆盖标准至少设计 30 个测试用例[13].表 1 中"测试用例数量"一栏列出了每个实验程序的测试用例集的规模.

4.4 实验步骤

按照以下步骤进行实验评估:

(1)针对一个给定实验对象程序 p,选取所有非等价、单一变异体作为被验证变异体集合 M(即表 1 "适用变异体数量"栏中变异体).

(2)使用分析工具 Frama-C[21] 获取实验对象程序的数据流信息.

(3)采用我们前期开发的变异测试精简工具[13]分析程序块结构,获取实验对象程序 p 的块规则文件.

(4)针对被验证变异体集合 M 每个变异体 m_i,获取 m_i 的变异位置信息.

(5)采用 3.3 节的识别规则集(R_1~R_4)识别 M 中的冗余变异体,统计冗余变异体数量,记录冗余变异体识别时间.

(6)用测试用例集运行所有变异体,记录每个测试用例在每个变异体上的运行结果(若变异体被杀死,则计为"1",否则记为"0").

(7)验证识别正确性(即比较所有冗余变异体的运行结果与对应变异体的运行结果).

(8)依据 *RR* 计算公式,计算冗余率.从变异体数量角度评估基于数据流分析的冗余变异体识别技术的有效性.

(9)在测试用例集上分别执行所有变异体和删除冗余变异体后的变异体集合,统计两者的执行时间(即变异测试的执行时间).依据 *TA* 计算公式,计算执行加速比.从变异测试的执行时间角度评估基于数据流分析的冗余变异体识别技术的有效性.根据 *CER* 计算公式,评估冗余变异体识别对变异测试效率的影响.

(10)依据 *MORR* 计算公式,计算变异算子的冗余率.进一步从冗余变异体的角度定量评估和比较各种变异算子的质量.

4.5 实验结果与分析

表 2 总结了不同识别规则在不同实验对象程序中识别出的冗余变异体情况.

表 2 不同实验对象程序的冗余变异体情况

实验对象	冗余变异体数量				冗余变异体总数
	R_1	R_2	R_3	R_4	
minmax	0	0	38	0	38
bubble	10	0	32	0	42
nextdate	0	0	53	0	53
tcas	0	0	73	23	96
tot_info	52	7	87	150	296
schedule	31	0	12	20	63
schedule2	12	3	18	33	66
print_tokens	59	10	57	123	249
print_tokens2	19	0	41	69	129
replace	88	0	160	94	342
space	872	73	521	0	1466

(1)冗余变异体的存在性验证：实验结果表明，不同 C 程序集生成的变异体集合中均存在冗余变异体，尽管不同程序的冗余率有所不同. 冗余变异体的存在部分解释了变异测试中少量测试用例能杀死大量变异体的原因.

特别地，我们对表 2 中列出的所有冗余变异体都进行了验证(即实验步骤 6 和 7)：在我们的实验方案中，变异体执行结果文件中第 i 行记录第 i 个测试用例能否杀死变异体("1"表示杀死；"0"表示未杀死). 根据冗余变异体的定义可知，若变异体 m_2 是变异体 m_1 的冗余变异体(即 $m_1 \mapsto m_2$)，那么在变异体 m_1 执行结果为"1"的行上，变异体 m_2 执行结果都必须为"1"(能够杀死变异体 m_1 的任意测试用例必然能杀死变异体 m_2).

(2) ReMuDF 的有效性分析：采用冗余率 RR 和执行加速比 TA 评估 ReMuDF 的有效性. 图 9 比较了不同实验对象程序的冗余率. 由表 2 和图 9 可知：

采用三个程序集进行变异测试时，ReMuDF 从总计 33155 个变异体集合中识别出 2840 个冗余变异体，识别出的变异体冗余率为 8.57%. 这意味着，在进行变异测试时约 9%的冗余变异体无需执行，这些变异体能否被给定的测试用例集杀死可从其它变异体的测试结果获得.

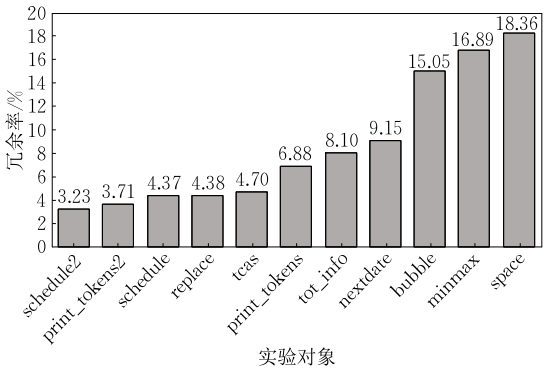

图 9　不同实验对象程序的变异体冗余率比较

针对三个程序集的变异体集合，ReMuDF 识别出的冗余率不同. 其中，space 程序识别出的冗余率最高(高达 18.36%)、科学计算程序集(minmax、bubble、nextdate)识别出的冗余率次之(平均超过 10%)、西门子程序识别的冗余率最低(平均在 5%左右).

由于识别规则依赖于特定的程序结构，通过分析程序的结构解释不同的冗余率的原因如下：① space 程序中(三个函数)顺序块较多，可应用的识别规则较多，识别出的冗余率高；② 科学计算程序集规模小，生成的变异体数量少，不存在变量在判断分支上使用性出现的情况(R_2、R_3 不适用)，因此冗余率较高；③ 西门子程序集识别的冗余率偏低，主要原因有：该程序集存在较多的变异体被过滤掉的情形(即变量定义在基本块中、而变量使用在分支块中)；该程序集存在较多在变量定义的同时进行初始化的语句，采用 Proteum 生成变异体时，不对这些语句进行处理.

图 10 比较了不同实验对象程序的变异测试执行加速比结果(即删除冗余变异体后变异测试执行效率的提升百分比). 由图 10 知：

通过识别并删除冗余变异体，可以有效地减少变异测试的执行时间. 采用 ReMuDF 技术后，变异测试减少了约 10%的执行时间.

不同实验对象程序集的变异测试加速比有所差别. 其中，space 程序的规模大，测试用例数量大且不易杀死变异体，冗余率最大，因此 space 程序的变异测试加速比最高(21.12%)；科学计算程序集的程序规模小，测试用例集易于杀死变异体，变异体被杀死耗费的时间短，因此其变异测试加速比较低.

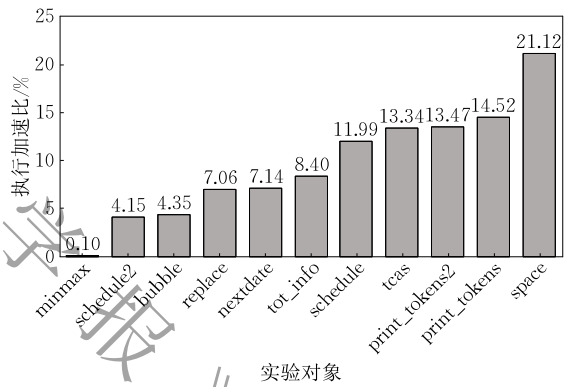

图 10　不同实验对象程序的执行加速比结果比较

上述实验结果表明：① 在冗余变异体数量方面，ReMuDF 可以有效识别出约 9%的冗余变异体；② 在变异测试的执行时间方面，ReMuDF 可以减少约 10%的执行时间；③ 通过比较不同规模的实验对象程序的冗余率和执行加速比可以发现，ReMuDF 在 space 程序的变异测试中识别出变异体冗余率最高、变异测试的执行加速比也最高. 换言之，ReMuDF 更适用于规模较大的实验对象程序.

(3)冗余变异体识别规则的有效性分析：图 11 比较了不同识别规则识别出的冗余变异体的比例. 由图 11 和表 2 可以看出：

识别规则 R_1 和 R_3 识别出的冗余变异体数量较多，分别占冗余变异体总数的 40%和 39%；而 R_2 和 R_4 识别出的冗余变异体数量较少，分别占冗余变异体总数的 3%和 18%.

不同识别规则在不同实验对象程序上识别出的

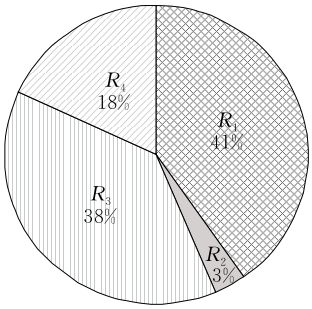

图 11 R_1～R_4识别的冗余变异体比例

冗余变异体数量差异大.具体说来,识别规则 R_1 在 space 程序中识别出 872 个冗余变异体,而在科学计算程序集中仅识别出 10 个冗余变异体;识别规则 R_2 仅在 tot_info、schedule2、print_tokens、space 程序中识别出少量的冗余变异体;识别规则 R_4 在西门子程序集识别出 512 个冗余变异体,但在 space 程序和科学计算程序集都未能识别出冗余变异体.进一步分析实验对象程序的结构发现:space 程序中存在较多的顺序块,R_1 识别的冗余变异体数量多;tot_info、schedule2、print_tokens、space 程序中存在变量定义后在其直接或间接顺接块中使用变量的程序结构,导致 R_2 识别出少量冗余变异体;西门子程序集中分支结构和循环结构占比大,分支/循环结构中定义的变量在其后的顺接块中使用情形多,R_3 识别的冗余变异体相对较多;西门子程序集中存在大量的函数调用,而且每个函数中的代码并不多.某个变量使用所对应的变量定义大多来自于函数调用传进来的变量值,导致 R_4 所识别的冗余变异体较多.

上述实验结果表明,不同规则识别冗余变异体的有效性因程序结构特征而异:① 存在较多的顺序结构程序中,R_1 较为有效;② 存在较多的分支/循环结构程序中,R_3 较为有效;③ 存在较多函数调用的程序中,R_4 较为有效.

(4)冗余变异体识别对变异测试效率的影响分析:表 3 比较了不同实验对象程序的冗余变异体识别能效比.由表 1 和表 3 可知:

总体来说,程序规模越大,实验对象程序的冗余变异体识别能效比越大.当程序规模增加时,变异体的数量和测试用例数增加,导致变异测试的执行时间急剧增加.但冗余变异体识别只需静态扫描程序的定义-使用链及变异体集合,与程序规模成线性关系.因此,当实验程序规模增加时,程序的冗余变异体识别能效比越大,冗余变异体识别对变异测试效率的影响越小.

表 3 不同实验对象程序的冗余变异体识别能效比

实验对象	变异测试的执行时间/s	冗余变异体识别时间/s	冗余变异体识别能效比
minmax	10.772	1.695	6.36
bubble	16.168	2.815	5.74
nextdate	153.796	7.790	19.74
tcas	755.581	11.349	66.58
tot_info	178.449	15.921	11.21
schedule	442.672	35.653	12.42
schedule2	1606.126	29.971	53.59
print_tokens	3924.337	56.979	68.87
print_tokens2	3508.264	53.940	65.04
replace	8632.458	104.979	82.23
space	42813.138	54.072	791.78

上述实验结果表明:虽然识别冗余变异体需要耗费一定的时间,但随着程序规模的扩大,程序的冗余变异体识别能效比不断提高,space 程序的冗余变异体识别能效比达到 791.78.这意味着,实验对象程序规模越大,冗余变异体识别对变异测试效率的影响越小;ReMuDF 更适用于规模较大的实验对象程序.

(5)基于冗余变异体的变异算子质量分析:实验中我们采用 Proteum 生成变异体,该工具支持 108 种 C 程序变异算子[23].本次实验应用了 60 种变异算子,其中 24 种变异算子产生的变异体中存在冗余变异体.图 12 和表 4 分别比较了这些变异算子生成的冗余变异体数量和冗余率.具体说来:

表 4 不同变异算子的冗余率比较

变异算子	冗余变异体数量	变异体总数	冗余率/%
STRI	1	443	0.23
OCOR	1	17	5.88
Oido	2	62	3.23
ArgStcDif	3	3	100.00
ArgDel	5	31	16.13
VLPR	6	34	17.65
VLAR	6	23	26.09
ArgStcAli	18	129	13.95
ArgBitNeg	19	160	11.88
ArgLogNeg	19	154	12.34
VSCR	27	904	2.99
ArgAriNeg	35	227	15.42
VGSR	41	835	4.91
SRSR	92	988	9.31
SSDL	102	779	13.09
STRP	120	926	12.96
ArgIncDec	148	1517	9.76
VDTR	170	1512	11.24
ArgRepReq	193	1096	17.61
Ccsr	195	2980	6.54
VTWD	257	1646	15.61
FunCalDel	302	1944	15.53
VLSR	481	3329	14.45
CRCR	597	4659	12.81

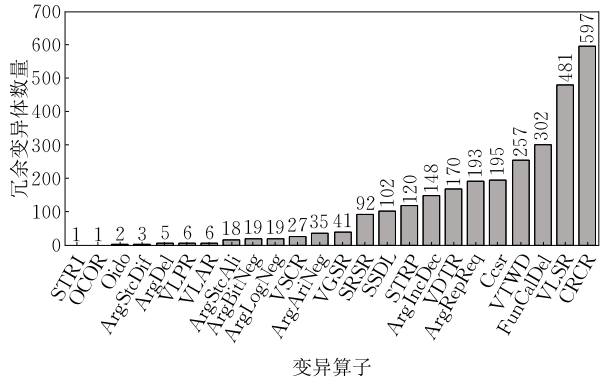

图 12　不同变异算子生成的冗余变异体数量比较

从生成的冗余变异体数量看来，CRCR(Required Constant Replacement)、VLSR(Mutate Local Scalar References)、FunCalDel(Removes function call)、VTWD(Twiddle Mutations)等 4 种变异算子均超过了 200；Ccsr(Constant for Scalar Replacement)、ArgRepReq(Replaces arguments by set R)、VDTR(Domain Traps)、ArgIncDec(Argument Increment and Decrement)、STRP(Trap on Statement Execution)、SSDL(Statement Deletion)、SRSR(Return Replacement)等生成的冗余变异体数量次之；其它 13 种变异算子生成的冗余变异体数量较少.

从生成的变异体的冗余率看来，ArgStcDif(Switch argument of non-compatible type)的冗余率为 100%，意味着该变异算子生成的变异体均是冗余的；VLAR(Mutate Local Array References)、VLPR(Mutate Local Pointer References)、ArgRepReq(Replaces arguments by set R)、ArgDel(Remove argument)等变异算子易于产生冗余变异体(约 17%).

上述实验结果表明，部分变异算子易于产生冗余变异体，在测试资源受限的情形下(如限定测试时间或变异体数量)，应尽量优先使用那些不易生成冗余变异体的变异算子. 变异算子生成的冗余变异体数量与变异算子冗余率作为评价变异算子质量的一种新手段，为选择与使用变异算子提供了参考依据.

5　相关工作

变异测试是一种具有较强故障检测能力的测试技术[2]. 近年来，人们围绕如何降低变异测试的计算开销和提高变异测试的效率开展了大量的研究工作. 下面简要介绍减少变异体数量与减少变异执行时间两方面的变异测试优化技术研究进展.

一类降低变异测试计算开销的方法是减少变异体数量. Acree[6]提出从所有变异体中随机选择一定比例的变异体进行变异测试，称为“随机选择”方法. Mathur 和 Wong[24]研究了不同比例下随机选择变异测试的有效性，实验结果表明随机选择 10%的变异体时构造的测试用例集的测试充分度比用所有变异体构造的测试用例集低 16%. Zhang 等人[25]采用经验研究评估了实验对象程序规模对选择变异测试可扩展性的影响. 实验结果表明，当程序规模或非等价变异体数量增加时，选择的变异体数量缓慢增加，但所选变异体比例减少；为取得比较充分的变异得分(例如 90%～99%)，当程序的可执行代码行数为 E 时，选择的变异体数量为 E^c，选择的变异体比例为 E^b(b 和 c 小于 1). Papadakis 等人[26]提出一种将变异前后的语句转化为变异分支的方法，将原程序的弱变异测试问题转化为新程序的分支覆盖问题. 在此基础上，Gong 等人[27]进一步提出了基于占优关系的变异体精简技术，通过分析测试用例对变异分支的覆盖情况识别变异分支之间的占优关系，将不被占优变异分支对应的变异体作为变异体子集进行变异测试. 实验结果表明该技术能有效减少变异体数量，但在识别变异分支间的占优关系方面引入了较大的时间开销.

Mathur[7]提出了选择部分变异算子进行变异测试，采用少量变异算子产生较少的变异体的方法，称为“约束变异”. Offutt 等人[28]进一步提出了“选择变异”方法，通过选择部分变异算子降低变异测试的代价. Zhang 等人[29]对随机选择和选择变异进行了实验评估，发现“选择变异”并不比“随机选择”更有效. 基于上述的结论，Zhang 等人[30]提出了将“随机选择”和“选择变异”方法结合起来的想法，在“选择变异”的基础上设计了 8 种随机采样策略，并验证了将这两种方式结合起来运用能更加有效地降低变异测试的开销. Yao 等人[31]研究发现一些变异算子易产生等价变异体，但生成的顽固变异体(很难被检测出来，但不是等价变异体)很少；而另一些变异算子易产生顽固变异体，但生成的等价变异体很少. 因此在生成变异体时，可对变异算子设置不同的选取优先级. Delamaro 等人[32]研究了只使用一个变异算子进行变异测试的有效性. 实验结果表明 SDL(语句删除)变异算子可能是最有效的变异算子. Kondbhar 和 Emmanuel[33]研究了不同操作符变异算子生成等价变异体的可能性. 实验结果表明，算术操作符对应的变异算子更容易产生等价变异体. Just 和 Schweiggert[34]指出变异算子之间存在包含

关系,并通过实验验证了只使用精简的变异算子集合生成变异体能在不明显影响变异测试有效性的条件下减少 20%的执行时间.与上述变异算子评价方法不同,本文根据不同变异算子生成的冗余变异体的数量和变异算子冗余率两个方面评估变异算子的质量.

Sun 等人[13]提出一种基于路径感知的变异体精简技术,在程序结构分析的基础上设计了基于深度优先的变异体选择启发式规则,通过赋予启发式规则不同的优先级,提出了变异体精简策略,通过选取不同比例的变异体子集进行实例验证.实验结果表明基于路径感知的变异体精简技术可以在不显著降低变异得分的情况下提高变异测试效率,而且在选取同样比例的变异体前提下,深度优先的变异体精简策略比"随机选择"和"选择变异"更高效.

Jia 和 Harman[8]提出了高阶变异测试的概念,即一次高阶变异由多个单阶变异构成,采用高阶变异体代替单阶变异体,可以有效减少变异体的数量.Polo 等人[35]研究了二阶变异测试,提出将一阶变异体组合成二阶变异体的三种算法.实验结果表明,采用二阶变异体可以有效减少测试开销,但没有显著降低测试的有效性.Harman 等人[36]研究了强包含高阶变异体(SSHOM)的有效性.实验结果表明,使用强包含高阶变异体能减少高达 45%的变异体数量,测试效率提高了 15%.Ghiduk[37]提出了两种基于进化算法的高阶变异体生成算法,并使用遗传算法生成测试用例.

Hussain[9]提出了将具有相似特征的变异体归为一类,再从每一类中随机选择一部分变异体进行变异测试,称为"变异体聚类"方法.实验结果表明,聚类方法能够在基本不影响测试有效性的情况下,很好地实现了变异体数量的精简.

上述变异体精简技术虽然能减少变异体数量,但在一定程度上影响了变异体集合的充分性评估能力.本文使用程序静态分析技术,依据变量定义和变量使用间的依赖关系识别冗余变异体,由于冗余变异体与源变异体被"杀死"的可能性相似,所提方法能在不影响变异体集合的充分性评估能力的基础上减少变异体的数量.

减少变异测试代价的另一类方法是减少变异测试的执行时间.弱变异测试不需要执行完所有的程序就可以判断变异体是否被杀死,从而提高了变异测试的效率.Girgis 和 Woodward[10]针对 Fortran77 程序语言实现了一个弱变异测试系统,通过分析程序内部状态来判断变异体是否被检测,结果表明弱变异在计算开销上要小于强变异,在多数情况下弱变异可以替代强变异.Horgan 和 Mathur[38]验证了在特定条件下基于弱变异生成的测试用例与基于强变异生成的测试用例具有相同的测试充分性.Offutt 等人[39]开发了一个弱变异平台,验证了弱变异是一种有效的变异测试方法,比强变异测试节省很大的计算开销.Kim 等人[40]将强变异和弱变异结合起来,选择能被弱变异杀死的变异体进行强变异测试,并将此方法应用于非解释性 Java 程序.在此工作基础上,Ma 等人[41]提出将弱变异测试结果相同的变异体进行聚类,再从每一类中选择一个变异体进行强变异测试,进一步提高了变异测试效率.

编译优化相关技术被用来减少变异测试的执行时间.King 和 Offutt[11]提出一种基于中间代码的变异测试方法,首先将原程序编译为中间代码,然后对中间代码进行变异和解释执行.当程序规模急速增加时,该方法的执行开销也会快速增长.Delamaro 等人[23]提出基于编译器的优化技术,将每个变异体编译为可执行程序,然后用测试用例运行可执行程序.上述技术可以加快变异体执行速度,但是如果变异体的编译时间大于执行时间,则限制了该技术的适用性.Untch 和 Offutt[42]提出一种变异模式生成方法,即采用一个包括所有可能变异体的测试系统,统一编译变异体而不是单独地一个个编译变异体.实验结果表明,该方法比传统的基于编译器的优化技术更为高效.DeMillo 等人[43]对传统的基于编译器的变异测试进行改进,通过对编译器进行插桩,使得编译原程序时输出两个对象:原有程序的目标代码和变异体对应的一系列补丁,应用这些补丁可以高效地将原有目标代码转化为变异体目标代码.Bogacki 和 Walter[44]提出了一种面向方面的方法来缩减编译开销,该方法需要执行两次补丁,第一次获知结果和原有程序的上下文信息,第二次生成并执行变异体.Ma 等人[45]提出了字节码翻译技术,在原程序的目标代码上变异生成变异体,生成的变异体不需要编译即可运行.Offutt 等人[46]尝试将字节码翻译技术应用于 Java 程序的变异测试.

人们还尝试了采用并行计算等技术提高变异测试的效率.Krauser 等人[12]提出一种单指令多数据流计算机上并行执行变异体的方法.Offutt 等人[47]提出一种在多指令多数据流计算机上实现了并行执行变异体的方法.Mateo 等人[48]通过减少主机和节点之间的通信开销进一步缩短并行执行变异体的时

间，提高了变异测试的效率. Cañizares 等人[49]利用集群系统的共享资源来解决变异测试的高计算开销问题，但网络延迟可能导致所提方法的性能下降.

上述并行执行变异体方法虽然能提高变异测试效率，但依赖于特定的运行环境. 本文所提冗余变异体识别方法通过识别冗余变异体，减少变异体的数量和缩短变异测试的执行时间，不依赖于特定的运行环境.

Kurtz 等人[50]依据测试用例集上变异体的黑盒测试行为提出了变异体之间的包含关系，并将其表示为变异体包含关系图. 通过不断执行测试用例运行变异体，动态获取变异体包含关系图[51]. 由于采用所有测试用例执行完所有变异体后才能得到包含关系图，该方法并没有减少变异体的数量，也没有减少变异测试的执行时间. Kurtz 等人[52]使用符号执行技术比较两个变异体在每条可执行路径上的输出结果，通过静态的方式获取变异体包含关系图，但该方法存在准确率低的缺点.

本文提出的冗余变异体识别方法，采用静态的数据流分析方法寻找具有相同“被杀死”行为的变异体，在测试执行前删除识别出的变异体，因而可以减少变异体的数量和缩短变异测试的执行时间. 通过度量与比较变异算子生成的冗余变异体的数量与冗余率，提供了一种评价变异算子质量的新途径.

6　结　论

变异测试广泛用于评价测试用例集充分性和评价软件测试技术有效性. 但由于计算开销大等原因，变异测试难以在软件测试实践中广泛使用. 本文从减少变异测试执行时间的角度出发，提出了冗余变异体的概念，结合程序块结构和数据流分析技术，在弱变异测试的基础上定义了一组冗余变异体识别规则. 采用 11 个 C 程序评估了提出的冗余变异体识别技术的有效性. 实验表明，使用本文方法可以识别出大量的冗余变异体，不仅减少了变异体的数量，而且缩短了变异测试的执行时间；提出的识别规则的有效性与程序结构有关；从生成冗余变异体数量与比例角度定量评价了变异算子的质量，在测试资源受限的情况下优先使用那些不易生成冗余变异体的变异算子.

本文工作开辟了变异测试优化技术的一个研究方向. 下一步的工作包括：(1) 探索两个变异体对多个变量的定义和使用关系识别冗余变异体(本文的方法仅从两个变异体对同一变量的定义和使用关系识别冗余变异体)；(2) 采用更大规模的工业应用实例程序评估本文方法的有效性；(3) 探索其他更高效的冗余变异体识别技术.

参　考　文　献

[1] DeMillo R A, Lipton R J, Sayward F G. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 1978, 11(4): 34-41

[2] Andrews J H, Briand L C, Labiche Y. Is mutation an appropriate tool for testing experiments?//Proceedings of the 27th International Conference on Software Engineering. Saint Louis, USA, 2005: 402-411

[3] Offutt J, Untch R H. Mutation 2000: Uniting the orthogonal//Proceedings of the Mutation 2000: Mutation Testing in the 20th and the 21st Centuries. San Jose, USA, 2000: 45-55

[4] DeMillo R A, Guindi D S, McCracken W M, et al. An extended overview of the Mothra software testing environment//Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis. Piscataway, USA, 1988: 142-151

[5] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 2011, 37(5): 649-678

[6] Acree A T. On Mutation [Ph. D. dissertation]. Georgia Institute of Technology, Atlanta, USA, 1980

[7] Mathur A P. Performance, effectiveness, and reliability issues in software testing//Proceedings of the 5th International Computer Software and Applications Conference. Tokyo, Japan, 1991: 604-605

[8] Jia Y, Harman M. Constructing subtle faults using higher order mutation testing//Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation. Beijing, China, 2008: 249-258

[9] Hussain S. Mutation Clustering [Ph. D. dissertation]. King's College, London, UK, 2008

[10] Girgis M R, Woodward M R. An integrated system for program testing using weak mutation and data flow analysis//Proceedings of the 8th International Conference on Software Engineering. London, UK, 1985: 313-319

[11] King K N, Offutt A J. A Fortran language system for mutation based software testing. Software Practice and Experience, 1991, 21(7): 685-718

[12] Krauser E W, Mathur A P, Rego V J. High performance software testing on SIMD machines. IEEE Transactions on Software Engineering, 1991, 17(5): 403-423

[13] Sun C, Xue F, Liu H, Zhang X. A path-aware approach to mutant reduction in mutation testing. Information & Software Technology, 2017, 81(C): 65-81

[14] Sun Chang-Ai, Wang Guan. MujavaX: A distribution-aware mutation generation system for Java. Journal of Computer Research and Development, 2014, 51(4): 874-881(in Chinese)
(孙昌爱，王冠. MujavaX:一个支持非均匀分布的变异生成系统. 计算机研究与发展，2014，51(4)：874-881)

[15] Demillo R A, Offutt A J. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 1991, 17(9): 900-910

[16] Howden W E. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering, 1982, 8(4): 371-379

[17] Li Hui-Xian, Liu Jian. Method of data flow analysis. Computer Engineering and Applications, 2003, 39(13): 142-144(in Chinese)
(李慧贤，刘坚. 数据流分析方法. 计算机工程与应用，2003，39(13)：142-144)

[18] Zeng J, Ju S, Song X. Construct information flow graph based on PDG//Proceedings of the 8th International Symposium on Computer Science and Computational Technology. Shanghai, China, 2008: 756-759

[19] Sun Chang-Ai, Jin Mao-Zhong, Liu Chao, Jin Ruo-Ming. An approach to static prediction and visual analysis of program execution time. Journal of Software, 2003, 14(1): 68-75(in Chinese)
(孙昌爱，金茂忠，刘超，靳若明. 程序执行时间的静态预估与可视化分析方法. 软件学报，2003，14(1)：68-75)

[20] Sun Chang-Ai, Liu Chao, Jin Mao-Zhong. Effective wove algorithm for software structure graph. Journal of Beijing University of Aeronautics and Astronautics, 2000, 26(2): 705-709(in Chinese)
(孙昌爱，刘超，金茂忠. 一种有效的软件结构图的布图算法. 北京航空航天大学学报，2000，26(2)：705-709)

[21] Cuoq P, Kirchner F, Kosmatov N, et al. Frama-C: A software analysis perspective//Proceedings of the 10th International Conference on Software Engineering and Formal Methods. Thessaloniki, Greece, 2012: 233-247

[22] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria//Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, 1994: 191-200

[23] Delamaro M E, Maldonado J C. Proteum—A tool for the assessment of test adequacy for C programs//Proceedings of the Conference on Performability in Computing Systems. New Jersey, USA, 1996: 79-95

[24] Mathur A P, Wong W E. An empirical comparison of data flow and mutation-based test adequacy criteria. Software Testing, Verification and Reliability, 1994, 4(1): 9-31

[25] Zhang J, Zhu M, Hao D, Zhang L. An empirical study on the scalability of selective mutation testing//Proceedings of the 25th International Symposium on Software Reliability Engineering. Naples, Italy, 2014: 277-287

[26] Papadakis M, Malevris N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal, 2011, 19(4): 691-723

[27] Gong D, Zhang G, Yao X, Meng F. Mutant reduction based on dominance relation for weak mutation testing. Information & Software Technology, 2017, 81(C): 82-96

[28] Offutt A J, Rothermel G, Zapf C. An experimental evaluation of selective mutation//Proceedings of the 15th International Conference on Software Engineering. Baltimore, USA, 1993: 100-107

[29] Zhang L, Hou S, Hu J, et al. Is operator-based mutant selection superior to random mutant selection?//Proceedings of the 32nd International Conference on Software Engineering. Cape Town, South Africa, 2010: 435-444

[30] Zhang L, Gligoric M, Marinov D, Khurshid S. Operator-based and random mutant selection: Better together//Proceedings of the 28th International Conference on Automated Software Engineering. Palo Alto, USA, 2013: 92-102

[31] Yao X, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence//Proceedings of the 36th International Conference on Software Engineering. Florence, Italy, 2014: 919-930

[32] Delamaro M E, Deng L, Li N, Offutt J. Experimental evaluation of SDL and one-op mutation for C//Proceedings of the 7th International Conference on Software Testing, Verification and Validation. Cleveland, USA, 2014: 203-212

[33] Kondbhar B B, Emmanuel M. An analysis of mutation operator in mutation testing. International Journal of Computer Science and Information Technologies, 2016, 7(4): 1701-1704

[34] Just R, Schweiggert F. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. Software Testing, Verification and Reliability, 2015, 25(5-7): 490-507

[35] Polo M, Piattini M, García-Rodríguez I. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification and Reliability, 2009, 19(2): 111-131

[36] Harman M, Jia Y, Mateo P R, Polo M. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation//Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. Vasteras, Sweden, 2014: 397-408

[37] Ghiduk A S. Using evolutionary algorithms for higher-order mutation testing. International Journal of Computer Science Issues, 2014, 11(2): 92-104

[38] Horgan J R, Mathur A P. Weak mutation is probably strong mutation. Purdue University, West Lafayette: Technical Report SERC-TR-83-P, 1990

[39] Offutt A J, Lee S D. An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering, 1994, 20(5): 337-344

[40] Kim S W, Ma Y S, Kwon Y R. Combining weak and strong mutation for a noninterpretive Java mutation system. Software Testing, Verification and Reliability, 2013, 23(8): 647-668

[41] Ma Y S, Kim S W. Mutation testing cost reduction by clustering overlapped mutants. Journal of Systems & Software, 2016, 115(2): 18-30

[42] Untch R H, Offutt A J, Harrold M J. Mutation analysis using mutant schemata//Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis. Cambridge, USA, 1993: 139-148

[43] DeMillo R A, Krauser E W, Mathur A P. Compiler-integrated program mutation//Proceedings of the 15th Annual International Computer Software and Applications Conference. Tokyo, Japan, 1991: 351-356

[44] Bogacki B, Walter B. Evaluation of test code quality with aspect-oriented mutations//Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering. Berlin, Germany, 2006: 202-204

[45] Ma Y S, Offutt J, Kwon Y R. MuJava: An automated class mutation system. Software Testing, Verification and Reliability, 2005, 15(2): 97-133

[46] Offutt J, Ma Y S, Kwon Y R. An experimental mutation system for Java. ACM SIGSOFT Software Engineering Notes, 2004, 29(5): 1-4

[47] Offutt A J, Pargas R P, Fichter S V, Khambekar P K. Mutation testing of software using a MIMD computer// Proceedings of the International Conference on Parallel Processing. Chicago, USA, 1992: 257-266

[48] Mateo P R, Usaola M P. Parallel mutation testing. Software Testing, Verification and Reliability, 2013, 23(4): 315-350

[49] Cañizares P C, Merayo M G, Núñez A. EMINENT: Embarrassingly parallel mutation testing//Proceedings of the 29th International Conference on Computational Science. San Diego, USA, 2016: 63-73

[50] Kurtz B, Ammann P, Delamaro M E, et al. Mutant subsumption graphs//Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops. Cleveland, USA, 2014: 176-185

[51] Ammann P, Delamaro M E, Offutt J. Establishing theoretical minimal sets of mutant//Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops. Cleveland, USA, 2014: 21-30

[52] Kurtz B, Ammann P, Offutt J. Static analysis of mutant subsumption//Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops. Graz, Austria, 2015: 1-10

SUN Chang-Ai, born in 1974, Ph.D., professor, Ph.D. supervisor. His research interests include software testing, program analysis, software architecture, and service-oriented computing.

GUO Xin-Ling, born in 1994, M.S. candidate. Her research interest is software testing.

ZHANG Xiang-Yu, born in 1976, Ph.D., professor, Ph.D. supervisor. His research interests include program analysis and software testing.

CHEN Tsong Yueh, born in 1951, Ph.D., professor, Ph.D. supervisor. His research interests include software testing and program debugging.

Background

Mutation testing is a fault-based software testing technique, which is widely used to evaluate the adequacy of a given test suite or the fault detection effectiveness of a given software testing technique. Although mutation testing has a strong fault detection capability, the high computation cost incurred by a huge number of mutants prevents mutation testing from being widely adopted in practice. Researchers have investigated how to reduce the number of mutants or shorten the execution time of mutation testing from different perspectives. Typical mutant reduction techniques include random mutant selection, selective mutation operators, high order mutation, and mutant clustering. These techniques are effective in reducing the number of mutants; however, the fault detection capability of the reduced mutants is affected. Typical mutation optimization techniques include weak mutation, compilers-based or parallel architecture-based mutation optimizations. These techniques are useful to shorten the execution time of mutation testing, while most of them rely on specific compiler or architecture techniques.

In our previous work, we have proposed a new mutant reduction technique by making use of program structure, conducted an empirical study on mutation testing of WS-BPEL

programs, and proposed a distribution-aware mutation analysis technique.

In this paper, we address the mutation optimization issue from a new angle. In order to improve the practicability of mutation testing, we propose the concept of redundant mutants with the purpose of reducing the number of mutants and the execution of mutation testing, and propose a data flow analysis based technique for identifying redundant mutants. We conduct an empirical study where eleven C programs are used to evaluate the feasibility and effectiveness of the proposed technique. The experimental results show that our technique can effectively identify a large number of redundant mutants, reduce the execution time of mutation testing, and provide a way to evaluate the mutant generation quality of mutation operators. The concept of redundant mutants and its identification technique proposed in this paper can not only improve the efficiency of mutation testing, but also provide useful guidelines for selecting mutation operators in the context of constrained resources available for mutation testing.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61872039 and 61370061, the Beijing Natural Science Foundation of China under Grant No. 4162040, the Aeronautical Science Foundation of China under Grant No. 2016ZD74004, and the Fundamental Research Funds for the Central Universities under Grant No. FRF-GF-17-B29.