基于变型空间代数的自动程序修复方法

徐 勇[1],毋国庆[2],袁梦霆[2],黄 勃[3]
(1. 广东肇庆学院数学与统计学院,广东肇庆 526061;2. 武汉大学计算机学院,湖北武汉 430072;
3. 上海工程技术大学电子电气工程学院计算机系,上海 201620)

摘 要: 基于代码枚举的自动程序修复方法借助变异算子对程序中错误语句进行变更操作,从而得到程序修复解. 由于缺乏文法制导及变异算子数量的有限性,该方法的有效性有待进一步提高. 本文提出一种基于变型空间代数的自动程序修复方法,即将回归测试用例集视为训练实例,通过归纳学习得到程序中出错语句的修复解. 具体而言,该方法包括以下特征:(1)从文法到变型空间的自动构造生成方法;(2)根据变型空间树中变型空间的不同类别,分别给出一致性定义;(3)结合静态及类型检查的变型空间代数运算. 实验结果表明:与基于代码枚举及基于搜索的修复方法相比,本文提出的方法在修复成功率方面更具优势;与此同时,方法中的静态及类型检查机制可以有效地削减假设空间的规模.

关键词: 自动程序修复; 变型空间代数; 归纳学习; 上下文无关文法; 生成树
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2017)10-2498-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2017.10.026

Automatic Program Repair Based on Version-Space Algebra

XU Yong[1], WU Guo-qing[2], YUAN Men-ting[2], HUANG Bo[3]
(1. *School of Mathematics and Statistics, Zhaoqing University, Zhaoqing, Guangdong* 526061, *China*;
2. *Computer School, Wuhan University, Wuhan, Hubei* 430072, *China*;
3. *School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai* 201620, *China*)

Abstract: Automatic program repair based on code enumeration exploits mutation operators to fix buggy programs by mutating the faulty statements. Its effectiveness is hindered by lack of grammar-directed mutation and limited number of mutation operators. This paper proposes a new automatic program repair method based on version space algebra, which uses inductive learning techniques to automatically produce repair solution for the faulty statement of buggy program. Specifically, the proposed method has the following features: (1) automatic derivation of version spaces from grammars, (2) defining consistency of version space according to its type, and (3) combining static and type checking with version space algebra. Experimental results show the proposed method outperforms other existing automatic program repair approaches in terms of repair success rate, and static and type-checking mechanism can prune the hypothesis space efficiently.

Key words: automatic program repair; version-space algebra; inductive learning; context-free grammar; derivative tree

1 引言

对于软件开发和维护者来说,软件调试中的错误更正(Bug Fixing)一直是件极其耗时、费力的日常性工作. 为了提高软件调试的效率,降低软件开发维护成本,人们提出自动程序修复方法(Automatic Program Repair)来改进目前纯人工方式的软件错误更正.

近几年来,人们对自动程序修复方法展开了广泛而深入的研究. 总的说来,这些方法大概可以分为以下几类:(1)基于搜索的方法[1,2]. (2)基于程序语义分析的方法[3,4]. (3)基于变异测试的方法[5,6]. 基于搜索的程序自动修复方法往往借助遗传编程或完全随机算法来搜索得到一个程序修复解. 由于基于搜索的方法中变异算子操作的随机性,可能得到许多无意义的修复结果[7,8]. 基于程序语义分析的方法需要在程序修复之前构造程序的语义模型,然后借助约束求解器对语义模型求解得到程序

收稿日期:2016-11-11;修回日期:2017-02-27;责任编辑:马兰英
基金项目:国家自然科学基金(No. 61640221, No. 61603242);上海高校青年教师培养资助计算专项基金(No. ZZGCD15088);肇庆学院科研基金(No. 201734);肇庆市科技创新指导类项目(No. 201704030409)

修复解[3,4]. 用于构造语义模型的符号执行本身的可扩展性问题使得此类方法仅能处理中等规模的带 bug 的程序[3]. 基于变异测试的自动程序修复方法引入变异测试中的变异算子,对带 bug 的程序生成变异体作为修复解[5,6]. 显然,此类方法缺少严格文法的指导,变异体可能不符合语法规范. Fan 等人提出了结合分阶段策略和条件合成的手段来削减修复搜索的空间,从而有效的提高程序修复的效率和成功率[9],该工作说明了通过减少搜索空间是改进程序修复方法的有效手段之一.

Tessa 扩展了 Mitchell 的变型空间[10],即允许变型空间进行代数运算[11]. 借助变型空间代数,人们提出基于演示的编程(Programming by Demonstration)思想,而且基于此思想能够通过归纳学习得到 shell 脚本[12],文本编辑代码[11]和 Python 程序片段[13]. 如果将回归测试用例集视为训练样例,而将用来替换错误语句的正确语句看作待求程序片段,则程序修复问题可以归结为归纳学习问题. 正是基于演示的编程和程序修复的共同点,激发了我们借鉴基于演示的编程思想来研究自动程序修复的问题.

2　相关概念与符号定义

定义 1[10]. 一个假设是一个函数 h,它将假设定义域 I_h 中的元素映射到假设值域 O_h 中的元素,即 $h(i)=o(i\in I_h, o\in O_h)$.

定义 2[10]. 一个假设 h 与一个训练样例集 D 一致,当且仅当 D 中的每一个训练样例 $t<i_t,o_t>$ 都有 $h(i_t)=o_t$,即 $Consistent(h,D)=(\forall t\in D)h(i_t)=o_t$.

定义 3[10]. 一个假设空间 H 是由相同定义域和值域的假设组成的集合.

定义 4[10]. 给定假设空间 H,训练样例集 D,一个变型空间是 H 中所有与 D 一致的所有假设构成的集合,记为 $VS_{H,D}$,即 $VS_{H,D}=\{h\,|\,h\in H\wedge Consistent(h,D)\}$.

文献[11]提出变型空间代数运算,即转换、并、交操作. 这样,变型空间呈现树状结构,称为变型空间树. 变型空间树中根变型空间也是目标变型空间,而非终端变型空间和叶子变型空间只是目标变型空间的构件.

定义 5[14]上下文无关文法由四元组表示,记作 $G(V_N,V_T,P,S)$,其中 V_N 是变元集,V_T 终结符号集,P 是产生式集,S 是开始符.

定义 6　设有 $G(V_N,V_T,P,S)$ 和它的某生成树 τ,则树 τ 的大小是树中所出现的运算符数目,记为$|\tau|$.

那么,文法 G 推导出的所有大小等于 i 的生成树集,记作 $\Psi_G^i=\{\tau\,|\,|\tau|=i\}$,当约定了生成树大小,则可简记为 Ψ_G. ,若约定了生成树的根 $\alpha(\alpha\in V_N)$,则记作 $\Psi_a^i(\alpha)$. 一定大小范围之内文法 G 的所有生成树集用 $\Sigma_G^n=\{\bigcup_{i\leqslant n}\Psi_G^i\}$ 表示. 而指定了根 $\alpha(\alpha\in V_N)$的生成树集,则记作 $\Sigma_G^n(\alpha)$.

定义 7　当程序运行时,程序中某个位置 i 的可见变量与其值的映射称为程序状态,记为 σ_i.

```
1.  #include <stdio.h>
2.  #define OLEV       600
3.  #define MAXALTDIFF 600
4.  #define MINSEP     300
5.  #define NOZCROSS   100
6.  int Cur_Vertical_Sep;
7.  bool High_Confidence;
8.  bool Two_of_Three_Reports_Valid;
9.  int Own_Tracked_Alt;
10. int Own_Tracked_Alt_Rate;
11. int Other_Tracked_Alt;
.....
13. bool Own_Above_Threat()
14. {return (Other_Tracked_Alt < Own_Tracked_Alt);}
15. int alt_sep_test()
16. {bool enabled, tcas_equipped, intent_not_known;
17. bool need_upward_RA, need_downward_RA;
18. int alt_sep;
19. enabled = High_Confidence && (Own_Tracked_Alt_Rate
    <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
20. tcas_equipped = Other_Capability == TCAS_TA;
21. intent_not_known = Two_of_Three_Reports_Valid || Other_RAC
    == NO_INTENT; /* logic change */
    alt_sep = UNRESOLVED;
.....
}
```

图1　一个带bug的程序: TCAS v3

为了叙述的方便,下面用带 bug 的程序(见图 1,记作 P_{bug})作为实例来说明基于变型空间代数的程序修复方法,且设它有测试用例集 $T=T_{pass}\cup T_{fail}$,其中 T_{pass},T_{fail} 分别表示失败和成功测试用例集. 图 2 给出了一个赋值语句的简单文法,记作 G. 设文法 G 中有产生式 $A\rightarrow\beta\in P$,则$|\beta|$表示其长度.

3　基于变型空间代数的修复方法

图 3 给出的是基于变型空间代数的程序修复方法(简记为 SmartRepair)的总体框架,它的输入包括:带有 bug 的程序,测试用例集 T,文法 G,生成树大小 k 和阈值 ω,λ (ω,λ 分别表示非终端及叶子变型空间一致性阈值). SmartRepair 包含有五个组件,即错误定位、程序状态对收集、枚举生成树集、构造变型空间树和变型空间更新,下面分别介绍.

3.1　错误定位

错误定位是要准确地定位到带 bug 的程序中引发程序失效的语句,原则上任何错误定位的方法都可用. 在具体的实现中,采用了基于频谱的错误定位方法[15]. 例如,程序 P_{bug} 中第 21 行为错误语句.

3.2　收集程序状态对

首先确定可见变量,它是程序中某个位置可引用的变量. 借助静态程序分析工具 LLVM-Clang[16] 可以获得某位置的可引用变量. 例如,对于程序 P_{bug} 中的第 21 行,可以确定 18 个变量和 6 个函数作为可变见变量.

```
<Primary_Expr> ::= <IDENTIFIER> | <CONSTANT>

<Unary_expr> ::= <Primary_Expr> |<Prefix_Oper><Unary_Expr>
| <Not_Oper><Unary_expr> | <Unary_oper><Unary_expr>

<Not_Oper> ::= '!'

<Prefix_Oper> ::= '++' | '--'

<Unary_Oper> ::= '+' | '-'

<Multiplicative_Expr> ::= <Unary_Expr>
| <Multiplicative_Expr><Multip_Oper><Unary_Expr>

<Multip_Oper> ::= '*' | '/'

<Additive_Expr> ::= <Multiplicative_Expr>
| <Additive_Expr><Add_Oper><Multiplicative_Expr>

<Add_Oper> ::= '+' | '-'

<Relational_Expr> ::= <Additive_Expr>
| <Relational_Expr><Relation_Oper><Additive_Expr>

<Relation_Oper> ::= '>' | '>=' | '<' | '<='

<Equality_Expr> ::= <Relational_Expr>
        | <Equality_Expr><Equ_Oper><Relational_Expr>

<Equ_Oper> ::= '!=' | '=='

<Logical_and_Expr> ::= <Equality_Expr>
        | <Logical_and_Expr><And_Oper><Equality_Expr>

<And_Oper> ::= '&&'

<Logical_or_Expr> ::= <Logical_and_Expr>
        | <Logical_or_Expr><Or_Oper><Logical_and_Expr>

<Or_Oper> ::= '||'

<Assignment_Expr> ::= <logical_or_Expr>
        | <Unary_Expr><Assign_Oper><Assignment_Expr>

<Assign_Oper> ::= '='
```

图2 一个赋值语句的简单文法G

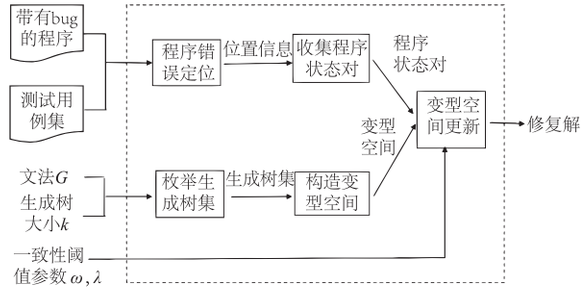

图3 SmartRepair总体框架

然后,在出错语句的前、后位置插装代码,功能是能输出程序执行位置i的程序状态σ_i,接着,对插装后的程序执行T_{pass}中的测试用例,这样就可以得到程序状态对集,记作S.

而主变量是可见变量中可以作为赋值语句的左值候选者的变量. 给定程序状态对集,考察程序状态对中哪些变量所对应的值是否发生了变更来确定主变量.

3.3 枚举文法生成树集

文献[17]提出一种文法生成树的枚举方法,与之不同的是,此处则根据定义6来构造生成树集,即Σ_G^n. 正因如此,可以将一些变元作为终结符来对待. 而且,对于某生成树的结果,不关心其中出现的变量、常量和函数等. 为叙述方便,称文法中这些语法变元组成的集合为扩展终结符集,记$\Sigma_{terminal}$. 若某生成树的结果中的所有符号都属于扩展终结符,则称完整生成树.

从文法构造生成树集由Enumerate算法完成,由于它需要调用Generate算法(见算法1)生成大小为0或者1范围之内的生成树集合,即Ψ_G^0和Ψ_G^1,因此先对后者进行说明.

首先引入两个函数IsOper和IsCompleted,前者的功能是判断变元是否属于运算符变元,而后者则判断一棵生成树是否是完整生成树. 另外BackTrack函数的功能是按要求进行出栈操作.

算法1 Generate算法:构造生成树大小为0或1的生成树集

输入:$G(V_N,V_T,P,S)$:一个文法;$start$:开始变元;$\Sigma_{terminal}$:G的扩展终结符集;k:生成树大小

输出:大小为i的生成树集Ψ_G^i,$(i\leqslant 1)$

stack:一个堆栈,记录推导过程中应用的产生式

Ψ_G:生成树集,其中每一个元素表示一棵生成树

```
1.  if start ∈ Σ_terminal and k = 0 then
2.      if IsCompleted(stack, Σ_terminal) = true then
3.          Ψ_G ← Ψ_G ∪ stack;      // 此时 stack 中的产生式就代表一棵完整生成树
4.          BackTrack(stack);
5.          return;
6.      else
7.          return;
8.      end if
9.  end if
10. if start ∈ Σ_terminal and k > 0 then
11.   stack.pop();
12.   return;
13. end if
14. for each start→β ∈ P // 设 β = β0β1…βi (i≥0)
15.   if |β| = 1 then
16.   stack.push(start→β);
17.   Generate(G, β, Σ_terminal, k);
18.   else
19.       if  k < 1 then
20.       continue;
21.       end if
```

```
22.            stack. push(start→β);
23.            for each βi ∈β
24.            if IsOper(βi) = false then
25.            Generate(G,βi,Σterminal,k-1);
26.            end if
27.            end for
28.        end if
29.    end for
```

算法 Generate 维护两个数据结构:堆栈 stack 和生成树集 Ψ_G. 前者用于保存推导过程中所应用的产生式,后者,则保存所构造的生成树。需要注意的是,生成树的保存是将生成树并入 Ψ_G(即通过∪操作完成的);另外,调用 BackTrack 函数是对推导过程进行回溯,即对 *stack* 进行出栈操作,直到最近一个含有运算符变元的产生式为止.

定理 1　给定文法 G,开始变元 *start*,扩展终结符集 $\Sigma_{terminal}$ 和生成树大小 $k(k=0,1)$,Generate 一定生成 k 大小的生成树集.

证明　当 $k=0$ 时,分两种情况,即:(a) $start \in \Sigma_{terminal}$,此时,若 isComplete 判断为假,生成空集结束. 否则,则得到一棵生成树,将其并入 Ψ_G^0. (b) $start \notin \Sigma_{terminal}$,显然,此时,Generate 仅能选择不含有运算符号的产生式来推导,并且继续以该产生式的右端调用 Generate 进行递归推导. 由于文法 G 中存在终结符(即变量和常量),因此,递归调用的过程必定会满足(a)情况中的完整生成树条件. 结合(a),(b)两种情况,当 $k=0$ 时,算法正确.

再考虑 $k=1$ 的情况. 由于 Generate 算法的递归定义,事实上,计算 $k=1$ 最终要递推到计算 $k=0$ 的情况,所以,可以将 $k=0$ 看作归纳基础步.

当 $k=1$ 时,也分两种情况:(a) $start \in \Sigma_{terminal}$,显然,当前应用的产生式无法得到大小为 1 的生成树,需要回溯到上一个产生式. (b) $start \notin \Sigma_{terminal}$. 此时考察文法中的每一个以 *start* 为左端的产生式 $start \to \beta$,β 可分两种形式:(Ⅰ)未含有运算符变元,则以递归方式调用 Generate 构造以 β 为根大小为 1 的生成树集. (Ⅱ)含有运算符变元,则构造其中以非运算符变元为根的大小为 0 的生成树集,即 $\Psi_G^0(\beta_i)$. 由于根据文法 G 的开始变元,一定能够通过推导到达 P 中的每一个产生式且 P 中除了 $\Sigma_{terminal}$ 之外的所有变元都存在产生式的右端含有运算符. 这样,对于(Ⅰ)情况来说,就确保了递归调用能够结束. 对于(Ⅱ)情况,计算 $\Psi_G^0(\beta_i)$ 等价于 Generate 算法当 $k=0$ 时的情况. 总之,(Ⅰ)、(Ⅱ)这两种情况都保证 Generate 能够构造以 *start* 为根的大小为 1 的生成树集. 证毕.

假定文法 G 中含有运算符产生式个数为 $|N|$,产生式的总数量 $|M|$,明显地,$|N| < |M|$. Generate 算法时间复杂度分析如下:算法的运行时间和文法 G 中运算符产生式的个数有关,而栈的操作次数决定了 Generate 总的时间开销. 栈的操作次数又正比于生成树的高度,而生成树高度的上界 $|M|$. 显然,Generate 算法的最坏时间复杂度应该是它生成大小为 1 的生成树集的时间. 文法 G 所能生成大小为 1 的生成树总数不超过文法 G 中含有运算符产生式个数为 $|N|$. 在算法第 3 行,获得一棵完整生成树时,就要做一次堆栈复制操作(也就是树的复制),而复制的最坏时间为 $O(|M|)$,所以时间总的复杂度为 $O(N*M)$.

算法 2　Enumerate 算法:构造大小范围为 k 的生成树集

输入:$G(V_N,V_T,P,S)$,一个文法;*start*,开始变元;$\Sigma_{terminal}$,G 的扩展终结符集;k,生成树大小
输出:大小范围为 k 的生成树集,$\Sigma_G^k(start)$

```
1.    Σ_G^k(start) ← ∅
2.      if start ∈ Σterminal then
3.        return Σ_G^k(start);
4.    end if
5.    if k = 0 or k = 1 then
6.      Σ_G^k(start) ← Generate(G,start,Σterminal,k);
7.      return Σ_G^k(start);
8.    end if
9.    for each start→β ∈ P
10.       if |β| = 1 then
11.       Σ_G^k(β) ← Enumerate(G,β,Σterminal,k);
12.       Σ_G^k(start) ← Σ_G^k(start) ∪ BuildUTree(start→β,Σ_G^k(β));
13.       end if
14.       if |β| = 2 and k < 1 then
15.          continue;
16.       end if
17.       if |β| = 2 then   // 设 β = αβ0 且 α 为运算符变元
18.          Σ_G^{k-1}(β0) ← Enumerate(G,β0,Σterminal,k-1);
19.          Σ_G^k(start) ← Σ_G^k(start) ∪ BuildUTree(start→β, Σ_G^{k-1}(β0));
20.       end if
21.       if |β| = 3 then // 设 β = β0αβ1 且 α 为运算符变元
22.          for z = 0;z < k-1;z++
23.             Σ_G^z(β0) ← Enumerate(G,β0,Σterminal,z);
24.             Σ_G^{k-z-1}(β1) ← Enumerate(G,β1,Σterminal,k-z-1);
25.             for each <τ1,τ2> ∈ Σ_G^z(β0) × Σ_G^{k-z-1}(β1)
26.                Σ_G^k(start) ← Σ_G^k(start) ∪ BuildBTree(start→β, τ1,τ2);
27.             end for
28.          end for
29.       end if
30.    end for
31.    return Σ_G^k(start);
```

算法 2 给出的是 Enumerate 算法,算法维护堆栈数组 $\Sigma_G^k(start)$,存放枚举得到的以根 *start* 的生成树集. Enumerate 算法第 1 行初始化 $\Sigma_G^k(start)$,当 $start \in \Sigma_{terminal}$ 返回当前的结果. 当 $k=0$ 或者 $k=1$ 时,调用 Generate 算法,枚举以 k 为大小的生成树集. 否则,考察以及 *start* 为左端的所有产生式 $start \rightarrow \beta$. 当 $|\beta|=1$,即没有符号运算符,就以 β 为起始元进一步枚举 k 大小的生成树集. 当 β 存在运算符号且 $k<1$ 时,此时,枚举下一候选产生式. 当 $|\beta|=2$,则先枚举 β 中非运算符号变元(设 β_0)为根,大小为 $k-1$ 的生成树集,然后调用 BuildUTree 函数构造以 *start* 为父结点,$\Sigma_G^{k-1}(\beta_0)$ 中的元素为儿子的生成树集,即 $\Sigma_G^k(start)$. 而当 $|\beta|=3$ 时,与 $|\beta|=2$ 的情况类似不再赘述. 需要注意的是:(1)$\Sigma_G^e(\beta_0) \times \Sigma_G^{k-z-1}(\beta_1)$ 表示 $\Sigma_G^e(\beta_0)$ 与 $\Sigma_G^{k-z-1}(\beta_1)$ 的笛卡尔积(见算法第 25 行). (2)BuildBTree 的功能是构造以 *start* 为根,生成树 τ_1 和 τ_2 为儿子的生成树.

它的时间复杂度分析如下:其最坏时间复杂度在算法的第 20 – 31 行,该处的运行时间依赖于 k, M, N. 当 $k=1$ 或 $k=0$ 时,Enumerate 就等价为 Generate 算法,此时的时间复杂度就为 Generate 的时间复杂度. 当 $k \geqslant 2$,其时间复杂度为 $O(k * N^k * M)$.

3.4 变型空间的构造

算法 3 是根据生成树构造变型空间的 BuildVS 算法,它需要 3 个辅助函数,BuildLeaf、BuildUnary 和 BuildBinary. Buildleaf 用来创建叶子变型空间,而 BuildUnary 和 BuildBinary 用来创建非终端变型空间. 明显地,BuildVS 通过对存放生成树的堆栈逐步出栈来构造对应的变型空间,并且将结果存放在一个堆栈 VS_Stack,而结果是非终端变元 V 与变型空间 VS 对 (V, VS). 若设生成树的高度为 M,则 BuildVS 算法的时间复杂度为 $O(M)$.

算法 3 BuildVS:变型空间的构造

```
输入:存放生成树的堆栈,stack
1.  while stack is not empty
2.    A→β←stack.pop();
3.    if |β| = 1 then
4.        if β ∈ Σ_terminal then
5.        VS ← BuildLeaf(β);
6.        VS_Stack.push(A,VS);
7.      else
8.        (β,VS)←VS_Stack.pop();
9.        VS_Stack.push(A,VS);
10.       end if
11.   end if
12.   if |β| = 2 then  // 设 β = αβ0 且 α 为运算符变元
13.       (β0,VS0)← VS_Stack.pop();
14.       VS1← BuildUnary(α,VS0);
15.       VS_Stack.push(A,VS1);
16.   end if
17.   if |β| = 3 then  // 设 β = β0αβ1 且 α 为运算符变元
18.     (β0,VS0)←VS_Stack.pop();
19.     (β1,VS1)←VS_Stack.pop();
20.     VS2← BuildBinary(A,VS0,α,VS1);
21.     VS_Stack.push(A,VS2);
22.   end if
23. end while
```

经过代数运算后的变型空间可能存在假设不符合语法规范. 例如"+ +5". 为了克服此问题,引入静态及类型检查机制,即对文法中的产生式定义相关静态及类型检查语义规则(见表 1). 注意,限于篇幅,表中仅给出了部分规则.

表 1 类型检查语义规则

文法中的产生式	类型检查语义规则
〈Primary_Expr〉::=〈IDENTIFIER〉	{Primary_Expr.type:=IDENTIFIER TYPE}
〈Primary_Expr〉::=〈CONSTANT〉	{Primary_Expr.type:=CONSTANT.type}
〈Unary_Expr〉::=〈Primary Expr〉	{Unary_Expr.type:=Primary_Expr.type}
〈Unary_Expr〉::=〈Prefix_Oper〉〈Unary_Expr1〉	{Unary_Expr.type:=if Unary_Expr1.type! =int error else Unary_Expr1.type}
〈Unary Expr〉::=〈Not_Oper〉〈Unary_Expr1〉	{Unary_Expr.type:=if Unary_Expr1 type! =bool error else bool}
〈Unary_Expr〉::=〈Unary_oper〉〈Unary_Expr1〉	{Unary_Expr.type:=if Unary_Expr1.type ∉ {int,float} error else Unary_Expr1.type}

3.5 变型空间的更新

(1)变型空间树中根结点的定义

变型空间树中的根结点变型空间是所有目标概念集合,即修复解集. 测试用例集 T 为训练样例集,而假设空间为所有符合给定文法 G(见图 2)的语句集合记为 H. 若 $P_{bug}[h/s_n]$ 表示将用假设 h 替换程序 P_{bug} 中第 n 行的 s 语句后的程序,而 $\mathrm{Test}(P_{bug}, t)$ 表示对程序 P_{bug} 执行某测试用例 t 的结果,包括 pass(通过)或 fail(失败),则某假设 h 与 T 的一致性定义应该是 h 是否能够使 $P_{bug}[h/s_n]$ 通过 T 中的全部测试用例,即:$\mathrm{Consistent}(h,T) = (\forall t \in T)\mathrm{Test}(P_{bug}[h/s_n], t) = \mathrm{pass}$.

定义 8 关于假设空间 H 和测试用例集 T 的变型空间,记为 *VS_ROOT*,是 H 中与 T 一致的所有假设构成的

子集. 即 $VS\_ROOT_{H,T} = \{h \mid h \in H \wedge \text{Consistent}(h,T)\}$.

(2)非终端变型空间的定义

在变型空间树中,非终端变型空间(记作 VS_NT)是其他变型空间的构件. 显然,此时的假设空间 H 为所有符合文法的表达式集. 因此,其假设的一致性可以借助文献[18]中的思想,利用两个变量的 Spearman 相关系数来刻画变型空间中假设的一致性. 即:

$$Corr(\boldsymbol{D},\boldsymbol{D}') = \sum_{i=1}^{n}(p_i-\bar{p})(q_i-\bar{q}) \Big/ \sqrt{\sum_{i=1}^{n}(p_i-\bar{p})^2 \cdot \sum_{i=1}^{n}(q_i-\bar{q})^2} \tag{1}$$

那么给定主变量 Y、假设 h 和程序状态对集 $\mathcal{S}$,可以根据 $\mathcal{S}$ 构造两个向量 $\boldsymbol{D}$,$\boldsymbol{D}'$,其中 $\boldsymbol{D}$ 和 $\boldsymbol{D}'$ 中元素分别来自于 Y 和 h 所对应的程序状态. 则假设 h 与 $\mathcal{S}$ 一致性定义可以描述为与主变量 Y 的相关性系数是否大于等于某个阈值 ω,即 $\text{Consistent}(Y,h,\mathcal{S}) = Coor(\boldsymbol{D},\boldsymbol{D}') \geqslant \omega$.

定义 9　给定主变量 Y,关于假设空间 H 与程序状态对集 $\mathcal{S}$ 的非终端变型空间,记为 VS_NT,是 H 中与 $\mathcal{S}$ 一致的所有假设构成的子集. 即 $VS\_NT_{H,\mathcal{S}}(Y) = \{h \mid h \in H \wedge \text{Consistent}(Y,h,\mathcal{S})\}$.

作为示例,表 2 列出的是变量或方法的变型空间及其中每个假设与主变量 intent_not_known 之间相关系数值. 若设 ω 为 0.3,则变型空间中的假设缩减至 Two_of_Three_Reports_Valid,Other_RAC.

表 2　主变量与其他变量相关性系数值

可见的方法或变量	相关性系数值	可见的方法或变量	相关性系数值
		Cur_Vertical_Sep	0.036
Own_Below_Threat()	0.134	Other_Tracked_Alt	0.153
enabled()	0.043	Alt_Layer_Value	0.067
Inhibit_Biased_Climb()	0.001	Other_RAC	0.520
ALlM()	0.067	Up_Separation	0.005
Other_Capability	0.081	Non_Crossing_Biased_Climb()	0.001
alt_sep	0.076	tcas_equipped	0.081
Own_Tracked_Alt	0.004	need_downward_RA	0.076
Climb_Inhibit	0.016	High_Confidence	0.048
need_upward_RA	0.076	Two_of_Three_Reports_Valid	0.599
Own_Above_Threat()	0.134	Non_Crossing_Biased_Descend()	0.117
Own_Tracked_A1t_Rate	0.093	Down_Separation	0.027

(3)叶子变型空间定义

叶子变型空间包括布尔常量、可见变量和整数常数变型空间. 由于程序中的可见变量,布尔常量值有限,因此对于这两个变型空间采用枚举的方法来表示.

对于整数常数变型空间,若采用与文献[13]中基于边界集的表示方法则不合适,原因是,程序状态中的变量值呈现的是离散的状态,并不趋向于某一个区间. 考虑到程序状态对都是通过收集运行成功的测试用例时程序出错位置前后的程序状态组成,因此,本文采用基于统计频率的方法来描述一致性. 假设空间 H 为程序状态对集中出现的套数集,则假设 h 与 $\mathcal{S}$ 一致性定义可描述为它的统计频率值是否大于等于某个阈值 λ,即 $Consistent(h,\mathcal{S}) = Frequency(\mathcal{S}) \geqslant \lambda$.

定义 10 关于假设空间 H 与程序状态对集 $\mathcal{S}$ 的整数常数变型空间,记为 VS_CONST,是 H 中与 $\mathcal{S}$ 一致的所有假设构成的子集. 即 $VS\_CONST_{H,S} = \{h \mid h \in H \wedge \text{Consistent}(h,\mathcal{S})\}$.

4　工具实现与实验评价

4.1　工具实现

为了对本文方法的有效性进行评价,实现了工具原型 SmartRepair. 鉴于 Siemens 基准库在同类研究工作中被广泛使用[4,3,19],本文选用它为实验对象(见表 3). 表 4 给出了实验相关参数及其含义. 实验的软件环境是 Ubuntu 10.04;硬件环境:处理器 Intel(R) Core(TM) i5 - 3210M,内存 4G.

表 3　实验对象程序

程序名称	错误版本数	测试用例集规模	程序行数
tcas	41	1520	173
schedule	9	2650	412
schedule2	9	2710	307
replace	32	5542	563
tot_info	23	1052	406

表 4　实验参数取值及意义

参数名称	参数取值	意义
tnum	100	用于训练的程序状态对数目
ω	0.35	非终端变型空间的一致性阈值
λ	0.05	整型常数变型空间一致性阈值
k	4	生成树的大小

4.2　实验结果

表 5 列出了本文方法与相关修复方法在修复成功率上的比较结果. 需要注意的是:(1)表中第三 ~ 五列的修复结果数据是直接从其文献中获得的. (2)在文献[3]中,对于 Replace 实验对象仅选择其中的 29 个错

误版本,且没有考察实验对象 tot_info,因此,在表中以“-”标识出来. 从表中可以看出,文献[19]中的方法的修复成功率为 18.4%(21/114),而本文方法的修复成功率为 35.4% (40/114);若不考虑实验对象 tot_info,则本文方法的修复成功率 34.07%(31/91),而 Genprog 的修复成功率为 18.18%(16/88). 因此,与文献[20,19]中的方法相比,本文的方法在修复成功率方面有一定的优势. 产生优势的原因在于,由于缺少文法指导和仅通过变异操作,这样就可能无法构造出新的符合语法的语句. 例如,在文献[19]中,无法对实验对象 schedule 进行修复的原因是变异操作符无法实现常量的替换.

表 5 SmartRepair 与文献中其它修复方法的修复成功率上的比较

程序名称	SmartRepair	Debroy and Wong 的方法[19]	Semfix[3]	Genprog[20]
tcas	48.78%(20/41)	21.95%(9/41)	82.93%(34/41)	26.83%(11/41)
schedule	33.33%(3/9)	0%(0/9)	44.44%(4/9)	0%(0/9)
schedule2	33.33%(3/9)	11.11%(1/9)	22.22%(2/9)	11.11%(1/9)
replace	15.62%(5/32)	9.38%(3/32)	20.69%(6/29)	13.79%(4/29)
tot_info	39.13%(9/23)	34.78%(8/23)	-	-
Σ	35.0%	18.4%	52.27%	18.18%

另一方面,与 Semfix 相比,总体来说并没有明显优势,有以下几个方面的原因:(1)Semfix 利用约束求解来获得修复表达式中的常量,而本文方法中叶子变型空间定义可能将正确的常量值排除在外. (2)生成树的大小设置也影响了某些实验对象的成功修复. 与此同时,对于实验对象 Schedule2,本文的方法在修复成功率上要优于 Semfix,这也说明,需要进一步地用更多实验对象与 Semfix 进行比较研究.

表 6 SmartRepair 与不同试验对象进行修复所需的时间花费

程序名称	平均时间(min)	最小时间(min)	最大时间(min)
tcas	15.2	12.5	16.8
schedule	13.4	11.6	15.3
schedule2	7.8	6.9	8.2
replace	6.9	5.0	7.2
tot_info	10.6	8.8	12.1

表 6 给出的是 SmartRepair 在不同实验对象上的修复花费时间,其包括构造生成树,构造变型空间和变型空间的更新训练时间. 文献[19]并未对修复花费时间展开研究,因此无法与之展开直接比较. 根据文献[3]中修复时间花费实验结果,结合表 6 可以看出本文方法所需的修复时间花费的方法是 Genprog 所需时间花费的 2~4 倍(当前后所采用的测试用例集规模大小为 100 和 50 时). 同时,基于表达式枚举方式修复所需最大时间大约不超过 16.6(min). 而本文方法对所有实验对象的平均时间花费不超过 16(min). 总体而言,本文修复方法与 SemFix 所给出的基于表达式枚举方法在时间花费上大致相当.

图 4 给出了在变型空间构造过程静态及类型检查机制所发现的语法不合格假设的比率. 由于每一个实验对象程序采用不同的错误版本,因此,语法不合格假设的比率是取同一实验对象程序中不同错误版本的平均值. 从图中可以看出,静态及类型检查机制有效地发现语法不合格假设,最小的不合格率为 35%,也就是将

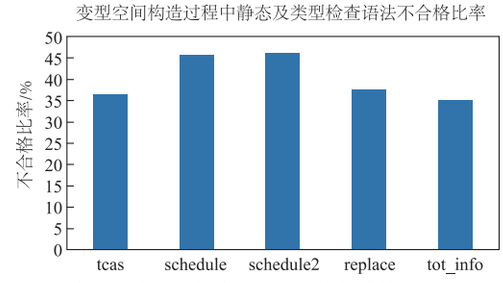

图4 变型空间中语法不合格假设的比率

变型空间的假设空间规模进行了有效地缩减.

5 结语

针对目前基于代码枚举的自动程序修复方法的一些不足,本文提出一种基于变型空间代数的自动程序修复方法,该方法将程序修复问题归结为归纳学习问题. 与基于变异算子的和基于搜索的修复方法相比,本文方法在修复成功率上更具优势. 与此同时,引入静态及类型检查机制后,可以有效地削减假设空间的规模. 下一步的工作包括:更广泛地对本文所提方法进行实证研究,进一步研究回归测试用例集的规模对该方法的影响等.

参考文献

[1] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer. Genprog:A generic method for automatic software repair[J]. Software Engineering, IEEE Transactions on,2012,38(1):54-72.

[2] Qi Yuhua, Mao Xiaoguang, Lei Yan, Dai Ziying, Wang Chengsong. The strength of random search on automated program repair[A]. In Proceedings of the 36th International Conference on Software Engineering[C]. Hyderabad, India:ACM,2014. 254-265.

[3] Hoang Duong Thien Nguyen, Qi Dawei, Abhik Roychoudhury, Satish Chandra. Semfix:Program repair via se-

mantic analysis[A]. In Proceedings of the 2013 International Conference on Software Engineering[C]. San Francisco,CA,USA:IEEE Press,2013. 772 – 781.
[4] R Konighofer, Roderick Bloem. Automated error localization and correction for imperative programs[A]. In Formal Methods in Computer-Aided Design (FMCAD)[C]. Austin,Texas:FMCAD Inc, 2011. 91 – 100.
[5] Vidroha Debroy,W Eric Wong. Using mutation to automatically suggest fixes for faulty programs[A]. In Software Testing,Verification and Validation (ICST),2010 Third International Conference on[C]. Paris:IEEE,2010. 65 – 74.
[6] Westley Weimer,Zachary P Fry,Stephanie Forrest. Leveraging program equivalence for adaptive program repair:Models and first results[A]. In Automated Software Engineering (ASE),2013 IEEE/ACM 28th International Conference on[C]. Silicon Valley,CA:IEEE,2013. 356 – 366.
[7] Dongsun Kim, Jaechang Nam, Jaewoo Song, Sunghun Kim. Automatic patch generation learned from human-written patches[A]. In Proceedings of the 2013 International Conference on Software Engineering[C]. San Francisco,CA:IEEE Press,2013. 802 – 811.
[8] Qi Zichao,Long Fan,Sara Achour,Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems[A]. In Proceedings of the 2015 International Symposium on Software Testing and Analysis[C]. New York,NY,USA:ACM,2015. 24 – 36.
[9] Long Fan,Martin Rinard. Staged program repair with condition synthesis[A]. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering[C]. New York,NY,USA:ACM,2015. 166 – 178.
[10] Tom M. Mitchell. Generalization as search[J]. Artificial Intelligence,1982,18(2):203 – 226.
[11] Tessa A. Lau,Pedro Domingos,Daniel S. Weld. Version space algebra and its application to programming by demonstration[A]. In Seventeenth International Conference on Machine Learning[C]. San Francisco,CA,USA:Morgan Kaufmann Publishers Inc. ,2000. 527 – 534.
[12] Tessa Lau, Lawrence Bergman, Vittorio Castelli, Daniel Oblinger. Programming shell scripts by demonstration[A]. In Workshop on Supervisory Control of Learning and Adaptive Systems[C]. San Jose, California: AAAI Press,2004. 44 – 47.
[13] Tessa Lau, Pedro Domingos, Daniel S Weld. Learning programs from traces using version space algebra[A]. In Proceedings of the 2nd international conference on Knowledge capture[C]. Sanibel Island, USA: ACM , 2003. 36 – 43.
[14] JohnE. Hopcroft,RajeevMotwani,JeffreyD. Ullman,Hopcroft,Motwani,Ullman. Introduction to automata theory,languages, and computation[M]. TBS,2006. 173 – 175.
[15] Rui Abreu,Peter Zoeteweij,Rob Golsteijn,Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization[J]. Journal of Systems and Software,2009, 82(11):1780 – 1792 .
[16] Clang:a C language family frontend for LLVM[R/OL]. http://clang. llvm. org. 2016.
[17] Dong Yunmei. Linear algorithm for lexicographic enumeration of cfg parse trees[J]. Science in China Series F:Information Sciences,2009,52(7):1177 – 1202.
[18] Shalini Kaleeswaran,Varun Tulsian,Aditya Kanade,Alessandro Orso. Minthint:Automated synthesis of repair hints[A]. In Proceedings of the 36th International Conference on Software Engineering[C]. Hyderabad, India: ACM, 2014. 266 – 276.
[19] Vidroha Debroy,W Eric Wong. Combining mutation and fault localization for automated program debugging[J]. Journal of Systems and Software,2013,45 – 60.
[20] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, Claire Le Goues. A genetic programming approach to automated software repair[A]. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation[C]. Montreal,Canada:ACM,2009. 947 – 954.

作者简介

徐　勇　男,1977 年生,博士,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

黄　勃(通讯作者)　男,1985 年出生,博士,讲师,主要研究方向为软件工程、需求工程、形式化方法,人工智能.
E-mail:huangbosues@ sues. edu. cn