软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3355−3376 [doi: 10.13328/j.cnki.jos.006925]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于变异的正则表达式反例测试串生成算法*

郑黎晓[1], 余李林[1], 陈海明[2], 陈祖希[1], 骆翔宇[1], 汪小勇[3]

[1](华侨大学 计算机科学与技术学院, 福建 厦门 361021)

[2](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

[3](卡斯柯信号有限公司, 上海 200070)

通信作者: 骆翔宇, E-mail: luoxy@hqu.edu.cn

摘　要: 正则表达式在计算机科学的许多领域具有广泛应用. 然而, 由于正则表达式语法比较复杂, 并且允许使用大量元字符, 导致开发人员在定义和使用时容易出错. 测试是保证正则表达式语义正确性的实用和有效手段, 常用的方法是根据被测表达式生成一些字符串, 并检查它们是否符合预期. 现有的测试数据生成大多只关注正例串, 而研究表明, 实际开发中存在的错误大部分在于定义的语言比预期语言小, 这类错误只能通过反例串才能发现. 研究基于变异的正则表达式反例测试串生成. 首先通过变异向被测表达式中注入缺陷得到一组变异体, 然后在被测表达式所定义语言的补集中选取反例字符串揭示相应变异体所模拟的错误. 为了能够模拟复杂缺陷类型, 以及避免出现变异体特化而无法获得反例串的问题, 引入二阶变异机制. 同时采取冗余变异体消除、变异算子选择等优化技术对变异体进行约简, 从而控制最终生成的测试集规模. 实验结果表明, 与已有工具相比, 所提算法生成的反例测试串规模适中, 并且具有较强的揭示错误能力.

关键词: 正则表达式; 正则语言; 字符串生成; 变异测试; 变异体约简

中图法分类号: TP311

中文引用格式: 郑黎晓, 余李林, 陈海明, 陈祖希, 骆翔宇, 汪小勇. 基于变异的正则表达式反例测试串生成算法. 软件学报, 2024, 35(7): 3355–3376. http://www.jos.org.cn/1000-9825/6925.htm

英文引用格式: Zheng LX, Yu LL, Chen HM, Chen ZX, Luo XY, Wang XY. Mutation-based Generation Algorithm of Negative Test Strings from Regular Expressions. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3355–3376 (in Chinese). http://www.jos.org.cn/1000-9825/6925.htm

Mutation-based Generation Algorithm of Negative Test Strings from Regular Expressions

ZHENG Li-Xiao[1], YU Li-Lin[1], CHEN Hai-Ming[2], CHEN Zu-Xi[1], LUO Xiang-Yu[1], WANG Xiao-Yong[3]

[1](College of Computer Science and Technology, Huaqiao University, Xiamen 361021, China)

[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[3](CASCO Signal Ltd., Shanghai 200070, China)

Abstract: Regular expressions are widely used in various areas of computer science. However, due to the complex syntax and the use of a large number of meta-characters, regular expressions are quite error-prone when defined and used by developers. Testing is a practical and effective way to ensure the semantic correctness of regular expressions. The most common method is to generate a set of character strings according to the tested expression and check whether they comply with the intended language. Most of the existing test data generation focuses only on positive strings. However, empirical study shows that a majority of errors during actual development are manifested by the fact that the defined language is smaller than the intended one. In addition, such errors can only be detected by negative strings. This study investigates the generation of negative strings from regular expressions based on mutation. The study first obtains a set of mutants by injecting defects into the tested expression through mutation and then selects a negative character string in the

* 基金项目: 国家自然科学基金 (61872339); 福建省自然科学基金 (2021J01316, 2021J01320); 中央高校基本科研业务费专项资金 (ZQN-1010); 厦门市自然科学基金 (3502Z20227191); 上海市自然科学基金 (22ZR1422200)
收稿时间: 2022-05-24; 修改时间: 2022-10-26, 2022-12-26; 采用时间: 2023-03-05; jos 在线出版时间: 2023-08-30
CNKI 网络首发时间: 2023-08-31

© 中国科学院软件研究所 http://www.jos.org.cn

complementary set of the language defined by the tested expression to reveal the error simulated by the corresponding mutant. In order to simulate complex defects and avoid the problem that the negative strings cannot be obtained due to the specialization of mutants, a second-order mutation mechanism is adopted. Meanwhile, optimization techniques such as redundant mutant elimination and mutation operator selection are used to reduce the mutants, so as to control the size of the finally generated test set. The experimental results show that the proposed algorithm can generate negative test strings with a moderate size and have strong error detection ability compared with the existing tools.

Key words: regular expression; regular language; string generation; mutation testing; mutant reduction

正则表达式具有较强的表达能力和灵活的定义方式, 在计算机科学的许多领域具有广泛应用. 其最早应用于编译器、浏览器、形式规约获取工具等软件中, 近年来应用范围更加广泛, 例如用于网络入侵或攻击检测[1]、MySQL 注入预防[2]、半结构化数据模式描述[3]、图数据库查询定义[4]等. 甚至在计算机领域以外也有正则表达式的应用, 例如, 在生物信息学领域使用正则表达式进行基因重组行为建模和序列联配[5]. 因此, 在实际开发中会经常使用到正则表达式. 调查研究表明[6,7], 有超过 1/3 的 JavaScript 和 Python 项目源码中包含至少 1 个正则表达式, 超过一半的被访问开发人员表示至少每周都会用到正则表达式进行项目开发.

然而, 由于正则表达式语法比较复杂, 并且使用大量元字符, 导致开发人员在定义和使用时容易出错[8,9]. 即便是对于非常短的正则表达式, 也很难读懂和理解其语义. 例如, 对于如下很短的正则表达式“\[([^\]]+)\]|\(([^\)]+)\)”, 用户很难立即明白它所定义的是什么样的语言. 对于可能包含 100 个字符或超过 10 个嵌套级别的复杂正则表达式来说, 难度会更大[10]. 实际上确实如此: 在一项真实访谈中, 开发人员普遍反映使用正则表达式时最大的困难就是编写和读懂它们[6]. 有学者对开发者在项目开发过程中如何编写正则表达式这一问题进行了调查[11], 发现大多数开发者都会通过搜索在线资源, 比如问答网站、在线论坛、正则表达式资源库等来解决问题. 这反映出编写正确的正则表达式通常是困难的, 开发人员可能更喜欢重用或修改现有的表达式, 而不是从头开始编写. 例如, 在开发人员学习和分享编程知识的热门网站 stackoverflow 上, 截至 2022 年 5 月 6 日, 有将近 25 万条 (随着时间还在增长) 与正则表达式相关的问题. 这也在一定程度上反映了开发者对正则表达式的高关注度以及在其使用上面临的困难.

错误的正则表达式会导致使用它们的应用程序出现缺陷. 例如, 假设一个程序接受允许带有正负号的非空数字串, 正确的表达式应该为“(+|−)?[0-9]$^+$”, 如果开发人员不小心写错为“[+|−]?[0-9]$^+$”, 则类似“|123”的非法串就有可能被接受. 这些含有缺陷的软件产品在部署后可能会产生无法预测的行为和结果, 甚至会对人们的生产和生活方式产生重大影响. 因此, 保证正则表达式的正确性是其可靠应用的重要前提条件. 然而, 调查显示开发者对正则表达式的测试程度要远低于其对程序代码的测试[6]. 更新的调查研究表明, 实际开发项目中使用的正则表达式约有 80% 没有经过测试, 而在被测试的正则表达式中, 约有一半只使用了 1 个测试串[12]. 因此, 亟需系统有效的方法和技术来确保正则表达式的正确性, 以提高相应软件系统的质量.

测试是检验软件系统或组件是否满足规定需求的常用和有效手段. 正则表达式测试的目的是检查所定义的语言是否符合用户的期望, 即满足用户的需求. 更具体地说, 如果正则表达式定义了所有预期要接受的字符串, 而没有定义任何预期要拒绝的字符串, 那么它在语义上是正确的. 一种常用的方法是根据被测试表达式自动生成一些字符串, 由开发者或者测试者检查它们是否符合预期. 生成的测试字符串可以是正例 (属于被测表达式定义的语言), 也可以是反例 (不属于被测表达式所定义的语言). 对于开发者或者测试者来讲, 判断给定的字符串是否属于预期所定义的语言要比编写语义正确的正则表达式容易得多. 如果用户发现生成的某些正例字符串应该被拒绝, 或者生成的某些反例字符串应该被接受, 则表示被测表达式的语义不正确. 因此, 正则表达式测试的核心问题是如何从待测试表达式中生成具有揭示错误能力的测试字符串.

假设待测试正则表达式定义的语言为 L, 预期语言为 L^I. 若 $L=L^I$, 则待测试表达式定义正确; 否则, 定义有误. 根据语言之间的包含关系可以将错误分为 3 类: ① $L\subsetneq L^I$; ② $L\supsetneq L^I$; ③ $L\not\subset L^I$ 且 $L\not\supset L^I$. 错误①在于待测试表达式定义的语言过小, 拒绝了所有应该被拒绝的字符串, 但同时也拒绝了一些应该被接受的字符串; 错误②在于待测试表达式定义的语言过大, 接受了所有应该被接受的字符串, 但同时也接受了一些应该被拒绝的字符串; 错误③则在于待测试表达式既接受了一些 (或全部) 应该被拒绝的, 又拒绝了一些 (或全部) 应该被接受的. 其中错误①只能通过生成被测试表达式的反例测试串发现, 因为对于每一个正例测试串 s, 都有 $s\in L$ 继而 $s\in L^I$, 也即每个正例

© 中国科学院软件研究所 http://www.jos.org.cn

测试串都符合预期. 同理, 错误②只能通过正例测试串发现, 错误③则有可能通过正例也有可能通过反例测试串发现.

近期一项对正则表达式错误类型的调查研究表明, 实际应用中出现的正则表达式语义错误类型中有大约 2/3 属于错误①, 即开发者往往编写过于受限 (比预期语言小) 的表达式[13]. 因此, 针对反例测试串的生成研究具有现实必要性和重要性. 实际上, 文献 [13] 也指出, 在正则表达式的测试生成研究中应该重点考虑生成被测表达式所定义语言之外的字符串, 也即反例字符串. 然而, 现有的大多数正则表达式字符串生成工具仅提供正例[14−16], 无法发现错误①这类占比较大的缺陷类型. 目前已知能够提供反例测试串的较为实用的两个工具是: EGRET[9]和 MutRex[17,18]. 下面对这两个工具进行介绍和分析.

EGRET 基于正则表达式底层自动机基本路径覆盖的方式生成字符串. 首先将一个正则表达式转换为一种特殊的自动机, 然后导出自动机上的一组基本路径, 最后从基本路径中创建正例或反例字符串. 其中反例串通过更改重复操作符的迭代次数等方式产生. 然而, 该算法的生成策略相对较为简单, 对于复杂的正则表达式, 其揭示错误能力相对较弱[19]. 在我们的实验中也发现 EGRET 的反例生成能力更加不足: 约 50% 的表达式产生的反例个数小于 4, 约 22% 的表达式甚至没有反例串产生.

MutRex 基于变异测试[20]的思想生成字符串. 变异测试是一种通过向被测试对象中注入缺陷来模拟软件中实际缺陷的技术, 被注入缺陷的测试对象称为变异体. MutRex 首先根据一组变异算子构造表达式的变异体, 然后对每个变异体生成一个能够区分该变异体与原始表达式的字符串. MutRex 能够同时提供正例和反例, 然而该算法在实际测试中存在以下几点不足之处. 首先, 变异算子的设计不全面, 我们在实证研究中发现有些错误类型无法使用这些算子模拟; 其次, 只采用了简单的一阶变异, 仅能模拟被测试表达式中的一个缺陷, 而研究表明, 实际项目当中存在的真实缺陷往往是由两个或者更多个缺陷一起造成的[21]; 再次, 算法通过计算变异体与原始表达式的对称差来产生字符串, 而 MutRex 一阶变异得到的许多变异体所定义语言是原始表达式所定义语言的子集, 这种情况下只可能生成正例字符串, 因此, 与 EGRET 一样存在反例测试串生成能力不足的问题.

本文借鉴 MutRex 思想研究基于变异的正则表达式反例测试串生成算法. 基本方法是通过变异得到一组模拟待测试表达式中可能存在的缺陷或错误的变异体, 然后在待测试表达式所定义语言之外、变异体所定义语言之内的范围中选取字符串, 这些字符串是被测表达式的反例并且能够揭示相应变异体所模拟的错误. 为了模拟复杂缺陷类型, 以及避免变异体特化 (定义语言比原始表达式小) 而无法获得反例串的问题, 引入二阶变异机制. 同时利用冗余变异体消除、变异算子选择等优化技术对变异体进行约简, 从而控制生成的测试集规模. 本文的主要贡献总结如下.

(1) 提出并实现两种新的正则表达式变异算子, 这些算子模拟的错误是实际中经常出现而 MutRex 没有考虑到的, 新算子的引入丰富已有的正则表达式变异算子体系, 能够更全面地模拟实际应用中常见的缺陷类型.

(2) 首次引入正则表达式的二阶变异机制, 相对于一阶变异而言, 二阶变异一方面能够模拟更加复杂的缺陷类型, 另一方面有助于减少等价和特化的变异体比例, 从而提高反例测试串的生成能力.

(3) 提出一阶变异体的冗余消除、二阶变异体的变异算子选择和双轮随机选择策略等变异优化技术对正则表达式的变异体进行约简, 在不降低揭示错误能力的同时控制最终生成的测试集规模, 提高算法的实用性.

(4) 实现了基于变异的正则表达式反例串生成算法, 由于实际开发中存在的大多数错误只能通过反例串揭示, 本文强调反例串的生成, 具有重要的现实意义和应用价值. 实验表明, 本文算法生成的反例串规模适中, 能有效地帮助发现已有工具无法发现的错误.

本文第 1 节介绍正则表达式和自动机相关的预备知识. 第 2 节给出变异算子的设计和实现. 第 3 节描述一阶变异体和二阶变异体的约简策略. 第 4 节介绍基于变异的反例测试串生成算法. 第 5 节给出实验结果及分析. 第 6 节介绍相关工作. 第 7 节总结全文.

1 预备知识

1.1 正则表达式

令 Σ 是一个字母表 (字符的集合). Σ 上的一个正则表达式 (简称表达式) 递归定义如下: ① 空集 $\varnothing$、空字符串 ε、任意字符 $a \in \Sigma$ 是正则表达式; ② 如果 R 和 S 是正则表达式, 则 $R{\cdot}S$, $R|S$ 和 R^* 是正则表达式. 其中运算符·、|和*分

© 中国科学院软件研究所 http://www.jos.org.cn

别表示连接、并和 0 次或多次重复 (Kleene 闭包) 运算. 为了方便起见, 本文有时省略连接运算符, 将 $R{\cdot}S$ 写为 RS. 为了便于区分, 正则表达式中出现的字母表上的字符通常被称为普通字符, 运算符号等则被称为元字符.

正则表达式 E 所定义的语言 $L(E)$ 是 Σ^* 的一个子集, 其中 Σ^* 表示字母表 Σ 上的所有字符串的集合. $L(E)$ 递归定义如下: $L(\varnothing)=\varnothing$; $L(\varepsilon)=\{\varepsilon\}$; 对于任意字符 $a\in\Sigma$, $L(a)=\{a\}$; 若 $E=RS$, 则 $L(E)=\{vw \mid v\in L(R),\ w\in L(S)\}$; 若 $E=R|S$, 则 $L(E)=L(R)\cup L(S)$; 若 $E=R^*$, 则 $L(E)=\{v_1v_2\ldots v_n|n\geqslant 0$ 且 $v_1,v_2,\ldots,v_n\in L(R)\}$. 若字符串 $s\in L(E)$, 则称 s 是表达式 E 所接受的字符串, 也称作是 E 的正例字符串; 否则称 s 是表达式 E 所拒绝的字符串, 也称作是 E 的反例字符串. 称两个表达式 R 与 S 等价, 当且仅当 $L(R)=L(S)$.

除了上述标准正则运算符, 许多特定应用对所支持的正则表达式进行扩展以增强定义的灵活性和便捷性. 例如, 可扩展标记语言 (extensible markup language, XML) 的文档类型定义 (document type definition, DTD) 中允许附加运算符+、?分别表示 1 次或多次重复、0 次或 1 次重复; XML 模式定义 (XML schemas definition, XSD)[22]进一步支持运算符{m, n}表示重复次数至少 m 次至多 n 次; 而 Relax NG[23]甚至还允许使用&&运算符来指定字符串的无序连接. 特别地, 在字符串模式匹配领域中, 正则表达式具有更紧凑的语法和更加灵活的定义方式. 例如允许使用“[]”来定义字符集、使用“\d”表示 0–9 之间任意一个数字字符等. 不同的正则表达式模式匹配引擎所支持的语法和特性也不尽相同, 因此, 实际应用中存在着大量的正则表达式方言. 一些正则表达式引擎甚至包含反向引用 (back references) 等非正则运算符. 本文算法支持字符串模式匹配中常见的正则运算符, 同时由于底层基于自动机实现, 因此不支持反向引用等非正则运算. 图 1 给出了本文算法所支持的正则表达式的语法描述.

名称		规则	描述
regexp	::=	unionexp	
unionexp	::=	interexp | unionexp	并运算
	|	interexp	
interexp	::=	concatexp & interexp	交运算
	|	concatexp	
concatexp	::=	repeatexp concatexp	连接运算
	|	repeatexp	
repeatexp	::=	repeatexp ?	零次或一次重复
	|	repeatexp *	零次或多次重复
	|	repeatexp +	一次或多次重复
	|	repeatexp {n}	n 次重复
	|	repeatexp {n,}	最少 n 次重复
	|	repeatexp {n, m}	最少 n 次最多 m 次重复
	|	complexp	
complexp	::=	~complexp	补运算
	|	charclassexp	
charclassexp	::=	[charclasses]	字符集合
	|	[^charclasses]	负值字符集合
	|	simpleexp	
charclasses	::=	charclass charclasses	
	|	charclass	
charclass	::=	charexp - charexp	字符范围
	|	charexp	
simpleexp	::=	charexp	
	|	.	任意单个字符
	|	#	空语言
	|	@	任意字符串
	|	“<Unicode string without double-quotes>”	不含双引号的字符串
	|	( )	空串
	|	(unionexp)	
charexp	::=	<Unicode character>	单个非保留字符
	|	\w	字母、数字和下划线
	|	\d	数字字符
	|	\s	空格字符
	|	\ <Unicode character>	转义成单个普通字符

图 1　本文算法支持的正则表达式语法

© 中国科学院软件研究所 http://www.jos.org.cn

1.2 自动机

一个非确定性有限自动机 (non-deterministic finite automaton, NFA) 是一个五元组 $A=(\Sigma,Q,q_0,F,\delta)$, 其中 Σ 是字母表, Q 是状态的有限集合, q_0 是初始状态, F 是终止状态集, $\delta: Q\times\Sigma\to 2^Q$ 是转换函数, 该函数将每个状态和字符对 (q,a) 映射到一个状态集上. 称一个字符串 $w=a_1a_2\ldots a_n$ 被非确定性有限自动机 A 接受当且仅当存在一个状态序列 $q_0,q_1,\ldots,q_n$ 使得 $q_n\in F$ 并且对每一个 $i\in[1,n]$, $q_i\in\delta(q_{i-1},a_i)$. 非确定性有限自动机 A 所接受的所有字符串的集合称为 A 定义的语言, 记作 $L(A)$. 若 NFA 中的转换函数 δ 仅将每个状态和字符对映射到单个状态上, 则称其为确定性有限自动机 (deterministic finite automaton, DFA). 本文中确定性和非确定性自动机统称为自动机.

已知正则表达式、NFA 和 DFA 三者的表达能力等价, 即对于任意正则表达式 E, 存在非确定性自动机 A 和确定性自动机 A' 使得 $L(R)=L(A)=L(A')$. 例如, 图 2 所示就是与表达式“(a|b)c*”等价的确定性有限自动机, 其中 q_0 是初始状态, q_1 是终止状态; 该自动机接受所有字母 a 或字母 b 开头且后面有零个或多个字母 c 构成的字符串.

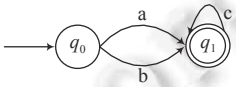


图 2 与正则表达式“(a|b)c*”等价的确定性有限自动机

1.3 变异测试

在软件测试技术中, 变异测试是一种通过向源程序中人工注入缺陷来模拟软件中实际缺陷的技术, 该技术可用于衡量测试数据集的揭错能力、评估和改进测试充分性, 也可用于指导测试数据的设计和生成. 缺陷的注入通过变异算子来完成, 变异算子定义了对被测程序做微小的合乎语法的变动规则, 例如将关系运算符“>”改为“<”. 被注入缺陷的程序称为变异体. 根据执行变异算子的次数, 可以将变异体分为一阶变异体和高阶变异体. 在原有程序上执行单一变异算子形成的是一阶变异体. 在原有程序上依次执行多次变异算子形成的变异体称为高阶变异体, 特别地, 若依次执行 k 次变异算子得到的变异体称 k 阶变异体.

对于给定的测试数据集 T, 若存在测试数据 $t\in T$ 使得变异体在 t 上的运行结果和原程序在 t 上的运行结果不同, 则称该变异体被测试用例集 T 杀死. 若变异体与原有程序在语法上存在差异, 但在语义上完全一致, 则称该变异体是一个等价变异体. 变异得分统计杀死的变异体数量占全部非等价变异体数量的百分比, 变异测试最终通过变异得分来评估测试用例集的错误检测能力, 变异得分越高说明测试数据集的揭错能力越强.

2 变异算子

2.1 变异算子介绍

变异测试的关键是如何设计变异算子产生变异体. 在程序变异测试领域, 针对许多编程语言如 Java、C、C# 等都有一套较为完善的变异算子和变异体生成工具. 而针对正则表达式的变异研究较为欠缺. MutRex[17]根据开发人员在编写正则表达式时容易犯的错误总结出 14 种变异算子. 本文在进行实际调研时发现有些常见错误无法被这些算子覆盖到, 因此提出 2 种新的变异算子.

2.1.1 原有变异算子

MutRex 中的 14 种变异算子分为 3 大类: 单个字符类、字符集合类和其他类. 单个字符类主要模拟表达式中单个字符上容易出现的错误, 如误将元字符作为普通字符使用等; 字符集合类主要模拟与字符集合定义相关的错误, 如集合中的字符范围定义过大或过小等; 其他类模拟与字符定义无关的错误, 如重复次数有误等.

(1) 单个字符类变异算子包括 4 种

• 改变大小写 CC: 模拟字母大小写错误, 该算子将每个大 (或小) 写字母字符改变为小 (或大) 写.

• 增加大小写 CA: 模拟字母大小写定义缺失错误, 该算子对每个大 (或小) 写字母字符增加小 (或大) 写定义.

© 中国科学院软件研究所 http://www.jos.org.cn

• 元字符转换普通字符 M2C: 模拟将元字符误用作普通字符的错误, 该算子对每个元字符增加转义符使其变为普通字符.

• 普通字符转换元字符 C2M: 模拟将普通字符误用作元字符的错误, 该算子去除每个转移符使待转义的普通字符变为元字符.

(2) 字符集合类变异算子包括 8 种

• 字符集合构造 CCC: 模拟用户在定义含有字符范围的字符集合时与方括号“[]”有关的错误, 该算子对表达式中形如“c_1-c_2”的子表达式构造字符集合“[c_1-c_2]”.

• 字符集合增加 CCA: 模拟用户在定义字符范围时可能遗漏某些情况的错误, 该算子对表达式中的每个字符集合“[c_1-c'_1…c_n-c'_n]”增加一个字符范围“c_{new}-c'_{new}”, 其中“c_{new}-c'_{new}”可以是“a-z”“A-Z”或“0-9”且在原始字符集合中没有出现过.

• 字符集合修改 CCM: 模拟用户在定义字符范围时与元字符“-”有关的错误, 该算子对形如“[c_1c_2]”的子表达式增加元字符“-”修改为“[c_1-c_2]”, 或者相反, 对形如“[c_1-c_2]”的子表达式删除元字符“-”修改为“[c_1c_2]”.

• 区间修改 RM: 模拟用户定义字符范围时区间过小或过大等错误, 该算子对形如“[c_1-c_2]”字符范围中区间两端的字符“c_1”和“c_2”进行编码递增或递减 (如果增减后仍是合法字符的话).

• 字符范围受限 CCR: 模拟用户定义字符范围时过于宽泛的错误, 该算子对形如“c_1-c'_1…c_n-c'_n”的字符集合依次去除其中的每个字符范围“c_i-c'_i”.

• 前缀增加 PA: 模拟所定义语言中对字符串首位字符有特殊限制相关的错误, 例如编程语言中标识符要求不能以数字开头. 该算子针对形如“[c_1-c'_1…c_n-c'_n] λ” (λ 是闭包或者{m, n}等重复运算符) 的字符集合增加一个去除“c_i-c'_i”区间的前缀, 例如将表达式“[a-zA-Z0-9]*”变异为“[a-zA-Z][a-zA-Z0-9]*”“[A-Z0-9][a-zA-Z0-9]*”以及“[a-z0-9][a-zA-Z0-9]*”.

• 字符集合负值 CCN: 模拟与负值字符集合定义相关的错误, 该算子对字符集合“[c_1-c'_1…c_n-c'_n]”增加一个负值运算符以及对每一个区间“c_i-c'_i”增加一个负值运算符.

• 负值字符集合转换为可选 NCCO: 模拟负值字符集合与可选重复运算符?搭配使用时有关的错误. 该算子在每个负值字符集合后增加运算符?. 例如, 用户希望匹配结尾处是字符“a”且后面没有“b”的所有字符串, 他可能会写成“(.*a [^b])”, 但是该表达式要求“a”之后必须有一个字符, 类似“China”“banana”这样的字符串将会被拒绝, 而 NCCO 算子将构造可以接受这些字符串的变异体“(.*a [^b])?”.

(3) 其他类变异算子包括 2 种

• 增加求补运算 NA: 模拟用户忘记写求补运算符号导致的错误, 该算子在表达式或子表达式所有可能的地方增加一个补运算符.

• 重复量词改变 QC: 模拟用户定义重复次数时可能存在的错误, 该算子将表达式中每一个简单重复运算符*、+、?分别替换为另外两个简单重复运算符, 将用户自定义重复运算符{m, n}中的 m 和 n 分别进行加 1 和减 1, 将重复运算符{n}替换为{n–1}和{n+1}等.

2.1.2 新变异算子

(1) 字符集合转组算子 CC2G

正则表达式中由小括号括起来的子表达式也称作组 (group), 而中括号括起来的是字符集合. 用户可能会不小心将一个组写成字符集合, 或者没有意识到小、中括号的不同功能. 以一个错误的正则表达式“[AM|PM|am|pm]”为例, 中括号括起来的是一个字符集合, 在该集合中, 字符“|”不表示并运算而是表示普通字符“|”本身. 因此, 上述表达式实际上等价于“[AMPMampm|]”. 这类常见错误在文献 [9] 和文献 [24] 中均有指出. 本文提出一种新的变异算子 CC2G 来模拟此类错误, 该算子将表达式中出现的描述字符集合的中括号替换为描述组的小括号. 例如, 将“[AM|PM|am|pm]”变异为“(AM|PM|am|pm)”, 后者是描述上下午时间标志的正确表达式.

CC2G 算子从某种程度上可以看作是 CCC 算子的逆操作, CCC 对表达式中形如“c_1-c_2”的子表达式增加中括号构造字符集合“[c_1-c_2]”, 而 CC2G 将删除“[c_1-c_2]”中的中括号取而代之为小括号. 但是 CC2G 变异范围更广泛, 并

© 中国科学院软件研究所 http://www.jos.org.cn

不局限于形如“c_1-c_2”含有字符范围操作的子表达式, 例如, “[abc]”不含有字符范围定义, CC2G 仍会将其变异为“(abc)”.

(2) 并运算受限算子 UR

并运算是正则表达式中最常用的特征之一[6]. 并运算的优先级比较低, 往往需要配合小括号来提升其运算顺序, 随之可能会带来优先级问题而导致的语义错误. 例如, 用户想定义正则表达式匹配除“cat”和“dog”之外的单词, 正确写法应该是“~(cat|dog)”, 但有可能不小心或者没有注意到优先级问题而误写成“(~cat|dog)”, 该表达式实际上会匹配“dog”和所有除“cat”之外的以“at”结尾的单词, 与预期不相符. 并且, 当并运算中嵌套的小括号越来越多, 用户通过匹配一对小括号来识别和理解并运算的左右操作数也变得越来越困难, 因此更容易出现错误.

文献 [9] 指出由类似的优先级问题导致的错误在实际中是很常见的. 因此本文提出新的变异算子 UR 来模拟此类错误. 该算子针对含有并运算的子表达式, 通过限制并运算的左右操作数来修改相应子表达式的运算顺序. 例如, 对“(~cat|dog)”使用 UR 算子可以变异为“~(cat|dog)”“~c(at|dog)”“~ca(t|dog)” “~cat|(dog)”“~cat|d(og)”“(~cat|do)g” “(~cat|d)og”等. 其中变异体“~(cat|dog)”是符合期望的正确表达式.

表 1 总结出本文使用的所有变异算子, 包括 MutRex 中原有的 14 种和本文新提出的 2 种 (使用加粗体标注). 表格第 1 列给出模拟缺陷所属类别, 第 2 列给出每一大类涵盖的变异算子, 第 3 列给出算子的英文描述, 第 4、5 列给出变异示例. 本文提出的 CC2G 算子归类为字符集合类, UR 算子归类为其他类.

表 1 本文算法使用的正则表达式变异算子

缺陷类别	变异算子	英文解释	原始表达式	变异体
单个字符类	CC	Case change	a[a-z]	A[a-z]、a[A-Z]、a[A-z]
	CA	Case addition	a[a-z]	(a|A)[a-z]、a[a-zA-Z]
	M2C	Metachar to char	.{3}	\.{3}
	C2M	Char to metachar	\.{3}	.{3}
字符集合类	CCC	Character class creation	(0-9)+	([0-9])+
	CCA	Character class addition	[a-z]	[a-zA-Z]、[a-z0-9]
	CCM	Character class modification	[az]或[a-z]	[a-z]或[az]
	RM	Range modification	[f-m]	[e-m]、[g-m]、[f-l]、[f-n]
	CCR	Character class restriction	[a-zA-Z0-9]	[A-Z0-9]、[a-z0-9]、[a-zA-Z]
	PA	Prefix addition	[a-z0-9]*	[0-9][a-z0-9]* 、[a-z][a-z0-9]*
	CCN	Character class negation	[a-zA-Z]	[^a-zA-Z]、[^a-z]|[A-Z]、[a-z]|[^A-Z]
	NCCO	Negated character class to optional	.*q[u]	.*q[^u]?
	CC2G	Character class to group	[a|b|c]	(a|b|c)
其他类	NA	Negation addition	[A-Z][a-z]	~([A-Z][a-z])、(~[A-Z])[a-z]、[A-Z](~[a-z])
	QC	Quantifier change	[0-9]*	[0-9]+、[0-9]?
	UR	Union restriction	a\d|A.	a(\d|A.)、(a\d|A).、a(\d|A).

2.2 变异算子实现

MutRex 基于 brics 库[25]实现了原来的 14 种变异算子. brics 是一个 Java 语言实现的正则表达式与自动机工具包, 支持表达式与自动机相互转换以及自动机上的交并差补等基本操作. MutRex 首先利用 brics 中的正则表达式解析器构造原始表达式的抽象语法树, 然后在语法树结构上做相应的修改操作, 最后遍历修改之后的抽象语法树得到原始表达式的变异体. 本文采用类似的方法实现新的变异算子 CC2G 和 UR, 具体实现过程描述如下.

2.2.1 CC2G 算子实现

CC2G 算子针对表达式中的字符集合进行变异, 字符集合可以分为一般的字符集合、仅含字符范围的字符集合和负值字符集合. brics 在解析时对不同类型的字符集合构造不同的语法树结构, 因此 CC2G 在基于语法树变异时相应地区分 3 种情况. ① 对于仅含字符范围的字符集合, 如“[0-9]”, brics 将其解析为单个的字符范围, 对于这种

© 中国科学院软件研究所 http://www.jos.org.cn

类型, CC2G 将该范围的最小字符“0”、元字符“-”与最大字符“9”做连接运算得到变异体“(0-9)”. ② 对于负值字符集合, 如“[^a-c]”, brics 解析时将其等价地转换为匹配任意字符的通配符“.”与一般字符集合补运算的交集, 即“.&~[a-c]”. CC2G 算子在实现时首先检查带有交运算的正则表达式是否为负值字符集合, 若是, 则将普通字符“^”与变异后的一般字符集合, 如该例中“[a-c]”变异后的“(a-c)”做连接运算来获得变异体. ③ 对于一般的字符集合, brics 将其解析为等价的并运算, 如“[AM|PM|am|pm]”解析为“(A|M|\\|P|M|\\|a|m|\\|p|m)”, 对应的语法树实际上是并运算的语法树. 对这样的字符集合实现 CC2G 算子的基本思想是通过连接解析后并运算的每个操作数, 将字符集合的内部成分转换为普通字符串, 然后利用 brics 将该字符串解析为变异表达式, 具体实现方式如算法 1 所描述.

算法 1. 一般字符集合的 CC2G 变异算法.

输入: 一般字符集合解析后的并表达式 E;
输出: 该字符集合的一个 CC2G 变异体.

1. $Elements \leftarrow getOperands(E)$ //获取 E 的操作数列表
2. **if** $\exists e \in Elements$ s.t. e is not a character or a character range **then return** $\varnothing$ // E 是普通并运算表达式
3. **else** // E 是一般字符集合
4. $s \leftarrow$ “ ” // s 初始化
5. **for each** $e \in Elements$ **do**
6. $s \leftarrow s + toString(e)$
7. **end for**
8. **return** $toRex(s)$ //将串 s 解析为变异体表达式
9. **end if**

以“[AM|PM](~ab?c|de*)”为例, brics 解析得到的语法树如后文图 3(a) 所示. CC2G 针对虚线框里面的子树进行变异操作时将并运算的操作数进行连接得到字符串“AM|PM”, 进而对该字符串解析得到如图 3(b) 所示的语法树, 即目标变异体“(AM|PM)(~ab?c|de*)”.

(1) 首先通过遍历 E 的子表达式 (即语法树中的子树) 生成子表达式列表 $Elements$, 其中 E 是待变异的一般字符集合解析后的并表达式 (第 1 行). 子表达式顺序与其在并表达式 E 中的出现顺序相同.

(2) 若存在有操作数既不是普通字符也不是字符范围, 表明 E 表示的不是一个字符集合, 而是一个如“(ab|cd|ef)”的普通并运算 (第 2 行), 此时不进行变异处理. 否则, 对于列表中的每个子表达式 e 按照顺序添加到字符串 s 中 (第 4–7 行), 当 e 是字符范围时, $toString(e)$ 会按顺序返回包含 3 个部分的字符串: 该字符范围的最小值、字符“-”和最大值.

(3) 最后, 函数 $toRex(s)$ 将字符串 s 解析为语法树, 得到相应的正则表达式, 即目标变异体 (第 8 行).

2.2.2 UR 算子实现

UR 算子针对表达式中含有并运算的子表达式. 直观上, 该算子通过增加或修改小括号的位置来实现运算符优先级方面的变异, 本质上则是改变抽象语法树中对应子树的语法结构. UR 算子主要考虑并运算左右操作数涉及有连接或者交运算的情况. 算法 2 以连接运算为例给出 UR 算子的实现方法. 首先排除并运算是由字符集合解析的情况 (第 1 行). 对于一般的并运算表达式 E 做如下处理.

(1) 首先, 取出 E 左右操作数对应的子表达式 (第 3 行).

(2) 然后, 使用函数 $getOrderPairs()$ 求出各操作数中按连接顺序出现的元素所构成的有序偶对 (第 4 行), 其中有序偶对定义如下: 设 s 是一个连接运算表达式, $<x, y>$是 s 的一个有序偶对当且仅当 x 与 y 进行普通字符串上的连接运算之后与 s 相同, 称 x、y 为该有序偶对的左部和右部, 并且规定左操作数的左部允许为空串, 右操作数的右部允许为空串. 例如, 若左操作数子表达式为“~ab?c”, 则其有序偶对为{<ε, ~ab?c>, <~, ab?c>, <~a, b?c>, <~ab, ?c>, <~ab?, c>}.

© 中国科学院软件研究所 http://www.jos.org.cn

(3) 最后, 遍历左右操作数中的有序偶对构造目标变异体 (第 5–13 行).

① 选取左操作数有序偶对的右部、右操作数有序偶对的左部作为变异后并运算的操作数 (第 9 行).

② 修改后的并运算与有序偶对中剩余部分进行连接 (第 10 行).

③ 若修改之后并运算左操作数的第 1 个元素是重复运算符或者右操作数的最后一个元素是补运算符, 则会导致产生的变异体不符合正则表达式语法规范, 因此需要事先排除这种情况 (第 7, 8 行).

以“[AM|PM](~ab?c|de*)”为例, brics 解析得到的语法树如图 3(a) 所示. UR 算子针对实线框里面的并运算子树进行变异操作时根据有序偶对<~a, b?c> 和<d, e*> 重新组织语法树, 变异结构如图 3(c) 所示, 得到原始表达式的一个 UR 变异体“(A|M|\\|P|M)(~a(b?c|d)e*)”.

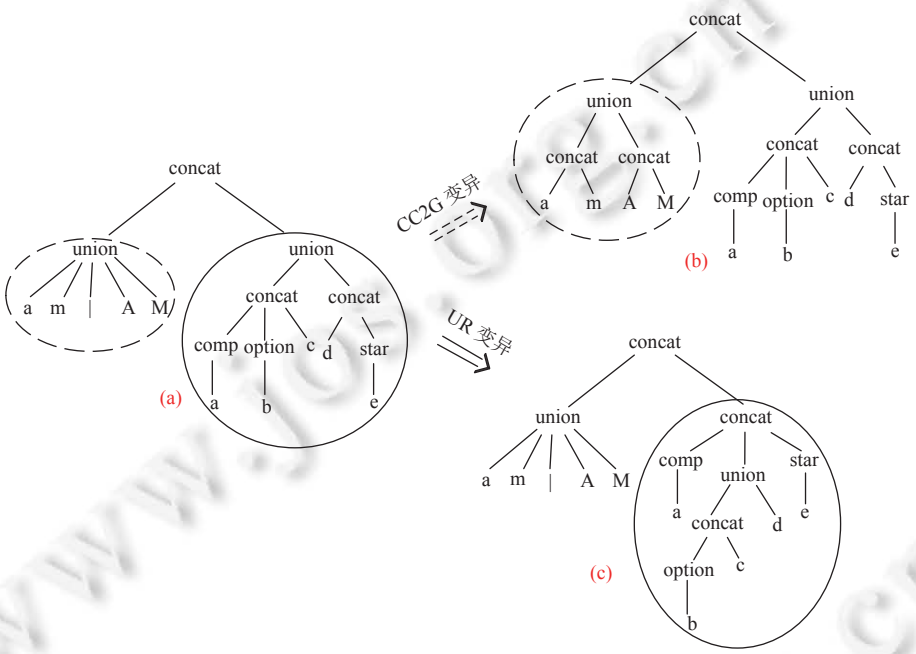


图 3 CC2G 与 UR 变异举例

算法 2. 并运算表达式的 UR 变异算法.

输入: 解析后的并运算表达式 E;

输出: E 的 UR 变异体集合 $Muts$.

1. **if** E is a character class **then return** $\varnothing$ //E 是字符集合则不变异
2. $Muts \leftarrow \varnothing$ //初始化
3. $l \leftarrow leftOperandOf(E)$, $r \leftarrow rightOperandOf(e)$ //获取并运算的左、右操作数
4. $lCombs \leftarrow getOrderPairs(l)$, $rCombs \leftarrow getOrderPairs(r)$ //获取左、右操作数中的序偶对
5. **for each** $lc \in lCombs$ **do**
6. **for each** $rc \in rCombs$ **do**
7. **if** $rightOf(lc)$ starts with repetition operator **then continue**
8. **if** $leftOf(rc)$ ends with complement operator **then continue**
9. $u \leftarrow union(rightOf(lc), leftOf(rc))$
10. $E' \leftarrow concatenate(leftOf(lc), u, rightOf(rc))$

© 中国科学院软件研究所 http://www.jos.org.cn

11. $Muts \leftarrow Muts \cup \{toRex(E')\}$
12. **end for**
13. **end for**
14. **return** *Muts*

2.3 变异体分类

根据所定义语言的大小, 可以将正则表达式 E 的变异体 M 分为如下 4 类, 如图 4 所示.

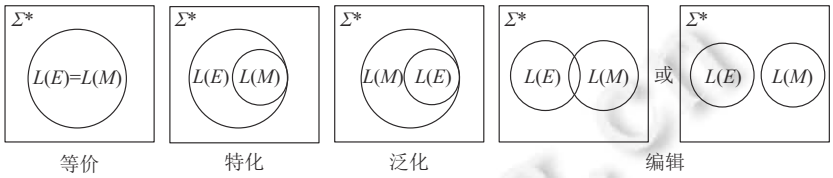


图 4 变异体分类

(1) 等价: 若 $L(M) = L(E)$, 即变异体 M 没有修改原始语言, 则称 M 是 E 的等价变异体.

(2) 特化: 若 $L(M) \subsetneq L(R)$, 即变异体 M 从原始语言中删除一些字符串, 则称 M 是 E 的特化变异体.

(3) 泛化: 若 $L(M) \supsetneq L(R)$, 即变异体 M 向原始语言中添加一些字符串, 则称 M 是 E 的泛化变异体.

(4) 编辑: 若 $L(M) \not\subset L(R)$ 并且 $L(M) \not\supset L(R)$, 即变异体 M 从原始语言中删除一些或删除全部字符串, 同时添加一些新的字符串, 则称 M 是 E 的编辑变异体.

例如, 表达式“[0-9]⁺”使用 QC 算子得到的变异体“[0-9]?”是原始表达式的编辑, 而使用 QC 算子得到的另一个变异体“[0-9]*”则是原始表达式的泛化; 表达式“[0-9].[0-9]”使用 M2C 算子得到的变异体“[0-9]\.[0-9]”是一个特化; 表达式“[0-9]{2, 2}[0-9]*”使用 QC 算子得到的变异体“[0-9]{2, 3}[0-9]*”则是一个等价变异体.

不同类型的变异体能够揭示的缺陷类型不同, 例如, 特化类变异体只能揭示定义的语言比预期语言大的缺陷, 即接受一些应该被拒绝的字符串, 而泛化类变异体只能揭示定义的语言比预期语言小的缺陷, 即拒绝一些应该被接受的字符串. 变异体类型受原始表达式结构、变异算子种类、变异位置等一系列因素影响. 有些变异算子只能产生一种类型的变异体, 例如前缀增加 PA 算子为形如“$[c_1\text{-}c_1' \ldots c_n\text{-}c_n']\lambda$” ($\lambda$ 是闭包等重复运算符) 的正则表达式添加前缀, 只能减少可接受的字符串, 因此总是产生特化类变异体.

3 变异约简

3.1 一阶变异体约简

变异测试具有较强的故障检测能力, 但是存在变异体数量较多的问题. 例如, 有统计发现, 对只包含 137 行代码的程序仅应用一阶变异就可以生成近 5 000 个变异体. MutRex 针对正则表达式的部分变异实验数据也显示, 使用 14 个变异算子进行一阶变异时平均每个表达式会生成 400 多个变异体[17]. 大量变异体的生成使得变异测试和分析的开销极为高昂, 因此变异约简是变异测试中一个重要问题. 本文对 MutRex 原有的变异算子进行深入分析发现部分算子会产生大量冗余性较高的变异体, 因此针对这些算子进行冗余消除和变异体约简.

(1) CC 和 CA 算子约简策略

CC 和 CA 属于单个字符类变异算子, 主要模拟字符大小写方面的错误. CC 算子将每个大 (或小) 写字母字符改变为小 (或大) 写, 而 CA 算子对每个大 (或小) 写字母字符增加小 (或大) 写定义. 对于含有字母较多的表达式就会产生大量冗余, 例如对于描写网址后缀的表达式“com|org|net|mil|edu”使用 CC 算子得到“Com|org|net|mil|edu”“cOm|org|net|mil|edu”等 15 个变异体, 使用 CA 算子同样得到“(c|C)om|org|net|mil|edu”“c(o|O)m|org|net|mil|edu”等 15 个变异体. 实际上, 当用户通过一个变异体发现表达式中存在大小写错误时, 他一般会同时检查其他地方以消除此类错误, 因此该类变异体只需要保留少数几个即可. 鉴于此, 本文对表达式中每个连续出现的

© 中国科学院软件研究所 http://www.jos.org.cn

字母字符序列以及每个字母字符区间只作用一次 CC 变异和一次 CA 变异.

(2) RM 算子约简策略

RM 属于字符集合类变异算子, 对形如“[c_1-c_2]”字符范围中区间两端的字符 c_1 和 c_2 进行编码递增或递减. 常见的字符集合主要包括小写字母“[a-z]”、大写字母“[A-Z]”以及数字字符“[0-9]”, 同一个字符区间在表达式中可能会出现多次, 例如描述邮箱地址的表达式“[a-z0-9][a-z0-9_\.-]{0, }[a-z0-9]\@[a-z0-9][a-z0-9_\.-]{0, }[a-z0-9][\.][a-z0-9]{2, 4}”以及描述科学记数法的表达式“[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?”, 代表小写字母和数字字符的“[a-z]”和“[0-9]”重复出现多次. RM 对字符区间的每次出现都进行变异, 变异体数量随着区间出现次数增长. 实际上, 当用户通过一个变异体发现某一字符区间定义有误时, 他一般会同时检查该字符区间出现的其他地方以消除此类错误. 因此, 与 CC 和 CA 算子冗余消除策略类似, 本文只选择同一个字符区间在表达式中的一次出现进行 RM 变异, 其他出现的地方不再重复处理.

(3) NA 和 CCN 算子约简策略

NA 和 CCN 都模拟与字符串集合上的求补运算有关的缺陷. NA 在整个表达式和所有子表达式上增加一个补运算符, CCN 对表达式中的形如“ [c_1-c_1' … c_n-c_n'] ”的字符集合以及每个区间“ c_i-c_i' ”增加一个负值运算符. 对于一些含有字符集合的表达式, 这两类算子产生的变异体之间会存在包含关系. 例如, 表达式“[a-z]”经过 NA 变异得到“~[a-z]”, 经过 CCN 变异得到“[^a-z]”. 前一个变异体匹配 a, b,…, z 字符之外的任意字符串, 后一个变异体匹配 a, b,…, z 之外的任意字符. 显然, “~[a-z]”包含“[^a-z]”, 任何能够区分后者的反例测试字符串都可以区分前者, 所以考虑前者冗余. 本文对原始表达式的每个 NA 变异体 M_{NA} 进行包含关系判定, 如果发现存在某个 CCN 变异体 M_{CCN} 使得 $L(M_{NA}) \supseteq L(M_{CCN})$, 则舍弃变异体 M_{NA} .

3.2 二阶变异体约简

一阶变异一次仅注入 1 个错误, 因此仅能模拟被测试对象中的 1 个缺陷. 实际项目中存在的真实缺陷可能是由两个或者更多个缺陷一起造成的, 所以高阶变异在模拟复杂缺陷方面有着重要意义, 可以用来解决复杂缺陷的问题. 另外, 有研究表明高阶变异能够有效避免等价变异体的出现. 例如, 在程序变异测试中, 一阶变异体中包含约 10% 的等价变异体, 而二阶变异体中仅包含 1% 的等价变异体[26]. 特别地, 在正则表达式测试中, 等价变异体对测试串的生成无法产生贡献, 因此可以通过高阶变异去除等价变异体从而提高测试串的生成能力. MutRex 只考虑一阶变异, 其作者也强调了在后续工作中引入高阶变异的重要性和必要性[18].

本文重点考虑二阶变异. 首先求出原始表达式的所有一阶变异体, 然后对每个一阶变异体再次做一阶变异. 理论上, 若一阶变异体数量为 N 个, 则二阶变异体数量大约为 N^2 . 急剧增长的变异体数量会导致最终产生的测试字符串集合庞大, 因此如何有效约简二阶变异体数量是一个关键问题. 针对程序代码的高阶变异体约简已有许多研究成果[27,28], 例如采用选择变异算子组合、减少变异位置或者对变异体进行聚类[29]等方式进行约简. 本文借鉴程序变异中的相关研究成果对正则表达式的二阶变异体采取如下 3 种约简方案.

(1) 选择变异算子组合

首先, 避免一阶变异和二阶变异时使用相同的变异算子组合. 由于同一变异算子模拟的是同类缺陷, 如果该类缺陷在一阶变异时已经能够体现出来, 则无需进行二阶变异. 例如, 如果一阶变异体 M 是使用大小写相关的 CC 算子得到的, 则对 M 做第 2 次变异时不再使用 CC 算子. 其次, 去除产生不合理变异体的算子组合. 进行二阶变异时, 变异算子的使用顺序会影响到最终变异体的合理性. 例如, 第 1 次变异使用 CA 算子将“[a-z]”修改为“[a-zA-Z]”来模拟大写字母缺失方面的缺陷, 第 2 次变异时若使用 CC2G 会将修改后的字符集合“[a-zA-Z]”转换成组的形式“(a-zA-Z)”, 则会掩盖掉字母大小写定义方面的缺陷. 再次, 排除互逆的变异算子组合. 二阶变异过程中在同一位置连续使用两种互逆的变异算子得到的二阶变异体与原始表达式完全相同, 因此互逆的变异算子组合可能会产生冗余. 例如对“[0-9].[0-9]”中元字符“.”使用 M2C 算子得到一阶变异体“[0-9]\.[0-9]”, 二阶变异时在相同位置使用 C2M 算子又变回原始表达式“[0-9].[0-9]”. 鉴于此, 本文对不同变异算子组合进行人工分析, 最终筛选出 66 种组合用于构造二阶变异体.

© 中国科学院软件研究所 http://www.jos.org.cn

(2) 双轮随机选择

随机选择法尝试从生成的大量变异体中随机选择一定比例的变异体. 相对于变异算子选择法, 随机选择是一种简单并且行之有效的优化方法[30]. 在二阶变异中, 一阶变异体数量和二阶变异位置都是导致大量变异体产生的重要因素. 本文采用一种类似文献 [30] 中的双轮随机选择法来减少正则表达式二阶变异体的数量. 首先, 对于每种变异算子随机选择一定比例的一阶变异体用于二阶变异; 其次, 对于二阶变异时的每种变异算子, 在一阶变异体中随机选择一定比例的位置进行二阶变异. 关于程序变异的实证研究表明当选择比例超过 10% 时可以达到比较好的效果[31]. 双轮随机选择中的随机选择比例由用户决定. 本文算法在实现时考虑到两轮随机比例之和尽可能达到 100%, 并尽量使较多的一阶变异体能够参与二阶变异, 因此默认情况将第 1 轮和第 2 轮的随机比例分别设为 75% 和 25%.

(3) 去除冗余二阶变异体

NA 和 CCN 算子可能会导致另外一种冗余. 在第 3.1 节对一阶变异体进行分析时发现 NA 和 CCN 算子产生的变异体可能存在包含关系. 若变异体 M 包含 M', 则任何能够区分 M' 的反例测试串都可以区分 M, 因此可以将 M 舍弃. 同样地, 二阶变异时若采用与 NA 和 CCN 相关的变异算子组合也可能会产生具有包含关系的二阶变异体. 本文对每个涉及 NA 算子和 CCN 算子的二阶变异体进行包含关系判定从而消除冗余.

4 反例串生成算法

基于变异的测试数据生成研究如何根据一组变异体构造能够杀死这些变异体的测试数据集. 对于程序而言, 杀死意味着测试数据在原始程序和变异体上的运行结果不同. 由于一个正则表达式定义一个正则语言, 因此当被测试对象是正则表达式时, 变异体被杀死意味着测试数据在原始表达式和变异体上的接受或拒绝行为不同, 具体阐述如下.

对于正则表达式 E、变异体 M 和测试字符串 s, 若 $s \in L(E)$ 且 $s \notin L(M)$ 或者 $s \notin L(E)$ 且 $s \in L(M)$, 则称变异体 M 被字符串 s 杀死. 若 $s \in L(E)$, 则称 s 是杀死变异体 M 的正例测试串; 若 $s \notin L(E)$, 则称 s 是杀死变异体 M 的反例测试串. 显然, 正、反例测试串的生成与变异体和原始表达式之间的关系有关.

如图 5 所示, 若变异体 M 是原始表达式 E 的一个特化, 即 $L(M) \subsetneq L(E)$, 则只能生成杀死 M 的正例测试串; 若变异体 M 是原始表达式 E 的一个泛化, 即 $L(M) \supsetneq L(E)$, 则只能生成杀死 M 的反例测试串; 若变异体 M 是原始表达式 E 的一个编辑, 即 $L(M) \not\subset L(E)$ 并且 $L(M) \not\supset L(E)$, 则既可以生成杀死 M 的正例测试串, 也可以生成反例测试串; 若变异体 M 与原始表达式 E 等价, 则无法生成任何测试串.

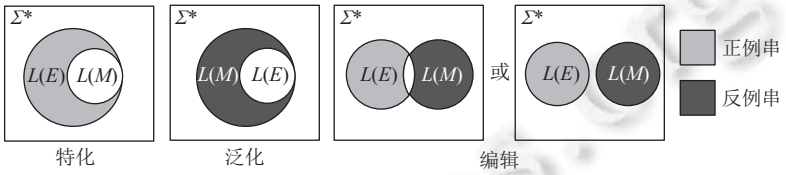


图 5 测试串生成与变异体分类之间的关系

基于上述分析, 反例测试串生成的基本框架如下. 首先排除特化类和等价类变异体, 其次对于每个泛化类和编辑类变异体, 先求出变异体所定义语言与原始表达式所定义语言的差集, 然后从差集中选取一个字符串, 该字符串即是能够杀死变异体的反例测试串. 考虑到一个反例串可能会杀死多个变异体, 因此在遍历时若发现某一变异体可以被已经生成的某个反例串杀死, 则对该变异体不再重复生成测试串, 从而降低最终的测试集规模. 算法 3 给出具体实现.

(1) 首先对原始表达式 E 进行一阶和二阶变异得到变异体集合 *Muts* (第 1–3 行).

(2) 然后将原始表达式转换成自动机表示 (第 5 行).

(3) 对每个变异体 M, 求 M 和 E 的差自动机 A_{Diff}.

① 若 A_{Diff} 定义语言为空, 说明 M 是 E 的等价或者特化变异体, 无法生成反例测试串, 略过 (第 9 行).

② 否则, 若存在一个已经生成的反例串被 M 接受, 说明 M 能够被该串杀死, 同样略过 (第 11 行).

© 中国科学院软件研究所 http://www.jos.org.cn

③ 否则, 根据差自动机 A_{Diff} 生成一个新的反例串来杀死 M (第 13, 14 行).

算法使用 brics 库实现正则表达式到自动机的转换、自动机的求差等操作. 同时, 为了使得产生的字符串方便用户进行判断, 通过调用 brics 库中的 *getshortestExample* 函数获取能够杀死 M 的长度最短的一个反例字符串.

算法 3. 反例串生成算法.

输入: 正则表达式 E;

输出: E 的反例串集合 *NStrs*.

1. $FstMuts \leftarrow fstMutation(E)$ //获取一阶变异体
2. $SecMuts \leftarrow secMutation(FstMuts)$ //获取二阶变异体
3. $Muts \leftarrow FstMuts \cup SecMuts$
4. $NStrs \leftarrow \varnothing$ //初始化
5. $A_E \leftarrow toAutomaton(E)$
6. **for each** $M \in Muts$ **do**
7. $A_M \leftarrow toAutomaton(M)$
8. $A_{\text{Diff}} \leftarrow minus(A_M, A_E)$ //计算差自动机
9. **if** $L(A_{\text{Diff}})$ is empty **then continue** //变异体 M 与 E 等价或是 E 的特化
10. **else**
11. **if** $\exists s \in NStrs$ s.t. $accept(A_M, s)$ is True **then continue** //变异体 M 被已经生成的字符串杀死
12. **else**
13. $s \leftarrow getShortestExample(A_{\text{Diff}})$ //获取杀死变异体 M 的字符串
14. $NStrs \leftarrow NStrs \cup \{s\}$
15. **end if**
16. **end if**
17. **end for**
18. **return** *NStrs*

5 实 验

第 5.1 节描述实验所使用的正则表达式数据集; 第 5.2 节分析本文算法 (源码链接: https://github.com/DamonYu97/Test-String-Generator-For-Regex) 产生的变异体特性和分类情况, 并和 MutRex 中变异体做了对比; 第 5.3 节从生成的反例测试串规模、揭示错误能力、算法运行效率等方面考察本文算法在实际测试中的应用情况, 并与现有工具进行了对比. 实验运行环境: CPU 为 Intel i5, 1.6 GHz, 内存大小为 8 GB. 算法中双轮随机选择比例按照默认值, 变异分析及变异得分等统计 25 次的平均值, 算法运行时间统计 100 次的平均值.

5.1 数据集

5.1.1 来 源

我们共收集了 297 个正则表达式用于实验, 分别来源于: 正则表达式库 RegExLib.com[32]、正则表达式网站 Regular-expressions.info[33]、GitHub 开源项目、MutRex 实验中提供的正则表达式. 具体介绍如下.

• RegExLib.com 是一个正则表达式在线库, 里面收集了来自世界各地开发人员所贡献的包括描述邮箱类、网址类、电话类、时间类、日期类、邮政编码类等在内的正则表达式. 我们从每一类中筛选出提交日期较新、不包含非正则运算符并且语法结构差异较大的若干正则表达式, 最终收集了 111 个用于实验.

• Regular-expressions.info 是一个正则表达式在线学习网站, 该网站提供了正则表达式相关的教程、书籍、部分正则表达式示例等资源, 我们从示例部分筛选出 RegExLib.com 中未覆盖到的包括描述浮点数类、数值范围类、

© 中国科学院软件研究所 http://www.jos.org.cn

Java 注释类等在内的 13 个较为复杂的正则表达式.

• GitHub 开源社区中通过搜索函数 re.compile() 和 Pattern.compile() 收集近期提交的 Python 与 Java 项目中的正则表达式, 去除较为简单的和支持非正则特性的, 最终筛选出上面两类数据集未覆盖到的包括描述编程语言预定义类、网关类、超链接类、IPv6 地址类等在内的 23 个正则表达式.

• MutRex 实验数据集中去除与上述重复出现的共计 150 个正则表达式.

我们在收集时对部分与本文算法所支持语法有出入的正则表达式做了预处理, 例如, 通过添加转义标记将本文语法中的某些特殊元字符如电子邮件中使用的字符“@”转换为普通字符等.

5.1.2 表达式特性

我们统计了所收集的正则表达式的长度等特性, 如表 2 所示. 其中, 运算符个数统计并、连接等常规运算符出现次数; 元字符个数统计除上述常规运算符之外的特殊符号如字符集合、通配符、转义符等的出现次数, 这类符号对表达式的可读性和可理解性有较大影响; 括号嵌套深度指小括号的最大嵌套层数; 星高度和一元嵌套高度解释如下.

表 2 实验所使用正则表达式数据集的特性

名称	最小	最大	平均数	中位数
长度	1	1 082	93.8	67
运算符个数	0	114	14.1	9
元字符个数	0	60	6.8	4
括号嵌套深度	0	115	3.9	2
星高度	0	3	0.9	1
一元嵌套高度	0	4	1.1	1

• 正则表达式 E 的星高度 $h(E)$ 递归定义为[34]: 若 E 是空集、空串或单个字符, 则 $h(E)=0$; 若 $E=R\lambda S$, 其中 λ 是二元运算符, 则 $h(E)=\max\{h(R), h(S)\}$; 若 $E=R^{\tau}$, 其中 τ 是*、+或者最多重复无限次的$\{n,\}$, 则 $h(E)=h(R)+1$; 若 $E=R^{\tau}$, 其中 τ 是?或者有限次重复$\{n\}$和$\{n, m\}$, 则 $h(E)=h(R)$.

• 正则表达式 E 的一元嵌套深度 $d(E)$ 递归定义为[35]: 若 E 是空集、空串或单个字符, 则 $d(E)=0$; 若 $E=R\lambda S$, 其中 λ 是二元运算符, 则 $d(E)=\max\{d(R), d(S)\}$; 若$E=R^{\tau}$, 其中 τ 是一元运算符, 则 $d(E)=d(R)+1$.

直观上, 星高度体现正则表达式闭包运算的嵌套深度, 对应于自动机中的环路嵌套次数, 是表达式语义结构复杂度的一种衡量标准. 一元嵌套深度则统计一元运算符号的最大嵌套层次. 相比于星高度, 它反映了正则表达式语法结构复杂性. 例如, “((a?|b|c)*|d)?”与“a*”的星高度都是 1, 但是一元嵌套深度分别是 3 和 1, 前者的语法结构比后者更为复杂.

从表 2 可以看出, 实验使用的数据集包含不同复杂程度的正则表达式: 从长度小于 10、运算符个数和括号嵌套深度为 0 的简单正则表达式到长度和运算符个数超过 100、括号嵌套层数超过 100 层的复杂正则表达式. 此外, 星高度和一元运算符嵌套高度都不超过 4, 平均值和中位数大约为 1, 括号嵌套深度平均为 4 层, 中位数为 2, 反映出实际中使用的表达式中大部分只有 1 层闭包运算或 1 层一元运算, 但是含有至少 2 层括号嵌套.

5.2 变异分析

5.2.1 变异体数量和分类

我们考察了本文变异策略产生的变异体数量和变异体分类等, 并与 MutRex 对了对比. 图 6 给出本文算法和 MutRex 对每个正则表达式生成的变异体个数, 其中表达式按照长度递增排序.

• 总体而言, 两种算法生成的变异体数量都随着表达式长度的增加而递增, 也有部分例外, 这是由于变异体数量与表达式中的运算符、语法结构等诸多因素相关, 与长度并不呈现绝对的正比关系.

• 两种算法生成的变异体数量大部分都不超过 300 个, 但是 MutRex 对其中个别表达式产生高达约 700 甚至超过 1 000 个变异体, 经分析发现这些表达式中含有大量字母字符和字母区间, 每个字母字符作用 CC 和 CA 算子

© 中国科学院软件研究所 http://www.jos.org.cn

或者字符区间作用 RM 算子导致大量冗余变异体产生, 而本文算法由于采取相应冗余消除策略, 避免出现变异体数量过多的极端情况.

• 本文虽然引入新的变异算子, 但是一阶变异时进行了冗余消除, 并且二阶变异时采取若干约简策略, 因此变异体总数和平均数相比 MutRex 均有所下降.

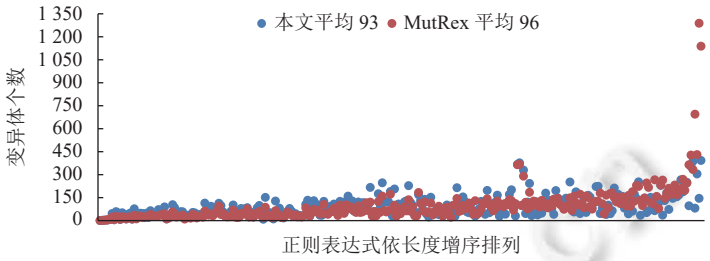


图 6 本文算法与 MutRex 生成的变异体数量

表 3 给出所生成变异体的分类情况, 其中第 3 列给出变异策略所模拟的缺陷种类, 第 4–7 列给出等价、特化、泛化和编辑的 4 类变异体占比, 第 8 列给出能够产生反例的泛化类和编辑类变异体总体占比; 第 2, 3 行分别统计本文一阶变异和二阶变异情况, 第 4 行统计 MutRex 变异情况.

表 3 生成变异体的分类情况

算法	变异策略	缺陷类型数	各类变异体占比 (%)				生成反例变异体占比 (%)
			等价	特化	泛化	编辑	
本文	一阶	16	10	20	29	41	70
	二阶	66	4	10	9	77	86
MutRex	一阶	14	7	20	35	38	73

• 本文一阶变异使用了 16 种变异算子, 二阶变异使用了 66 种不同的算子组合, 一共模拟缺陷类型共计 82 种, 而 MutRex 只使用 14 种变异算子进行一阶变异, 仅能够模拟 14 种单个缺陷.

• 从一阶变异体分类角度来看, 由于本文新增的 CC2G 和 UR 算子主要产生编辑类变异体, 约简的 CA、CC 等算子主要产生泛化类变异体, 因此本文一阶变异体中编辑类比例稍高、泛化类比例稍低. 特化类比例基本与 MutRex 一致. 等价类变异体没有太大变化, 由于一阶变异约简后总数比 MutRex 降低, 所以等价类比例相比 MutRex 稍高.

• 对比一阶和二阶变异可以看出, 二阶变异减少了等价、特化和泛化类变异体比例, 大大提高了编辑类变异体比例. 二阶变异体中泛化和编辑类总体占比将近 90%, 而这两类保证了反例测试串的生成, 说明二阶变异能够有效提高反例串的生成能力.

5.2.2 变异体分布情况

图 7 给出本文和 MutRex 一阶变异中各变异算子产生的变异体分布情况, 图 8 给出本文二阶变异中不同变异算子组合产生的变异体分布情况.

• MutRex 中不同算子产生的变异体数量差距较大, 尤其是 CC、CA、NA 等算子的变异体数要远远高于其他算子. 本文针对这些极端算子进行了冗余消除, 相比 MutRex 有较大幅度降低, 其他算子的变异体与 MutRex 相同.

• MutRex 中有部分算子产生的变异体很少, 比如 CCC 和 NCCO 算子对于 297 个正则表达式分别只生成 34 和 90 个变异体, 说明这些算子在实际中用到较少. 例如, NCCO 算子对负值字符集合增加一个?运算符, 而负值字符集合在实际中并不常见. 本文新引入的 CC2G 和 UR 算子没有出现这种极端情况, 说明与之相关的运算符如并运算以及字符集合等在实际中有广泛应用, 因此, 这两种算子的提出是具有实际意义的.

• 二阶变异时变异算子组合中的两种算子会直接影响到最终生成的二阶变异体数量. 例如, 一阶变异时 RM 和 CCN 算子的变异体相对较多, 则这两种组合对应的二阶变异体数量相对也较多; 相反, 与 CCC 和 NCCO 有关的算子组合对应的二阶变异体数量较少.

© 中国科学院软件研究所 http://www.jos.org.cn

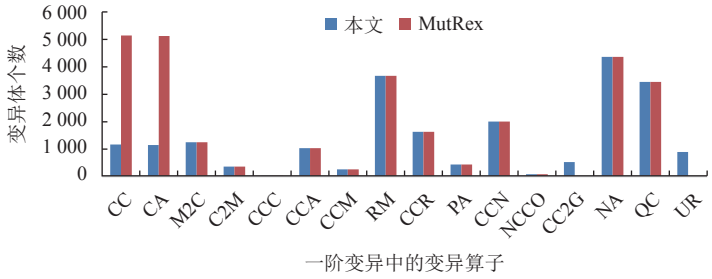


图 7 一阶变异体分布情况

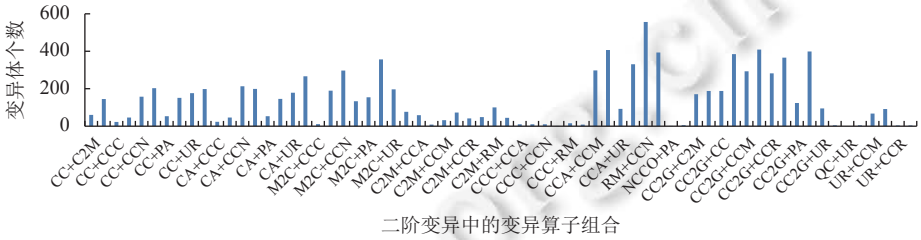


图 8 二阶变异体分布情况

5.3 反例串分析

5.3.1 反例串规模

测试数据集的规模将影响到测试活动的效率. 在正则表达式测试中, 测试字符串主要由人工来判断是否与预期相符合, 因此, 测试串的规模需要控制在合理的范围之内以降低测试开销. 目前能够生成正则表达式反例测试串的较为实用的工具主要有 MutRex 和 EGRET, 我们从反例字符串个数、分布情况等方面进行对比. MutRex 提供 3 种生成方式: 随机生成、正例优先和反例优先. 为了公平起见, 在实验中选择反例优先方式.

(1) 反例串平均数量对比

图 9 给出 3 种算法生成的反例字符串的数量对比情况, 其中横轴对应实验中的 297 个正则表达式, 按照长度增序排列, 纵轴分别显示 3 种算法生成的反例字符串的个数.

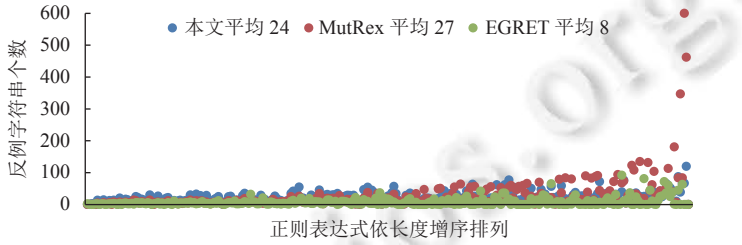


图 9 3 种算法生成反例字符串的个数

• 理论上, 字符串个数与变异体数量相关, 因此图 9 中本文和 MutRex 算法的字符串数量总体变化趋势与图 6 中变异体数量变化趋势基本一致.

• 本文算法对于大多数表达式 (约 70%) 生成的反例串要多于 MutRex, 但是由于 MutRex 对个别表达式产生的字符串较为极端 (最多 609 个), 因此平均值比本文算法高. 我们分析发现, 这些较多的字符串中 80% 以上都是由 CC、CA 等字母大小写相关的变异算子导致. 例如, 对于正则表达式“com|org|net|mil|edu”, MutRex 会生成“Com”“cOm”“coM”等大量改变字母大小写的反例, 而本文算法进行了冗余消除, 仅保留少数此类反例, 使得字符串个数大幅度降低.

• EGRET 生成的反例个数较少, 平均仅有 8 个. EGRET 中有极个别的反例个数比本文和 MutRex 多. 例如, 对

© 中国科学院软件研究所 http://www.jos.org.cn

于表达式“[a-z]$^+$([a-z0-9-]*[a-z0-9]$^+$)?(\.([a-z]$^+$([a-z0-9-]*[a-z0-9]$^+$)?)$^+$)*”, 本文和 MutRex 分别产生 29 和 8 个反例串, EGRET 生成 37 个, 原因在于该表达式中含有多个字符集合, 而字符集合中又包含多个字符范围, EGRET 构造自动机时每个字符范围对应一条边, 通过修改字符范围来构造反例, 因此产生的反例相对较多, 而本文和 MutRex 采用基于变异的生成策略, 对于此类表达式生成的反例串没有 EGRET 多.

(2) 反例串数量区间分布情况

图 10 给出 3 种算法生成的反例字符串数量区间的分布情况, 其中横轴表示字符串个数区间, 纵轴表示该区间对应的正则表达式在全部 297 个表达式中的占比.

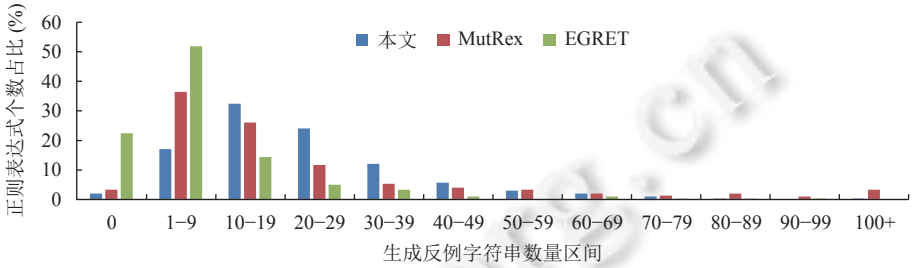


图 10 3 种算法生成的反例串数量分布情况

• 本文算法对超过一半 (51%) 的正则表达式生成的反例串少于 20 个, 近 90% 的正则表达式生成的反例串少于 40 个, 仅有 1 个表达式生成了超过 100 个反例串. 总体而言, 本文算法生成的测试串规模较为适中, 一方面避免测试数据过少导致测试不充分, 另一方面避免测试数据过多导致测试代价过高.

• MutRex 对约 65% 的表达式生成的反例串数量少于 20 个, 近 40% 不到 10 个, 没有反例产生的表达式比本文多, 说明反例生成能力较为不足. 这也反映了本文引入的新算子以及二阶变异有效提高了反例生成能力. 此外, 有 10 个表达式生成超过 100 个反例串, 有个别甚至超过 600 个, 这会给测试人员进行是否符合预期的判断带来较大压力. 如前所述, 这些数量较多的反例串大多是由 CC、CA 等字母大小写相关的变异算子导致, 实际上存在大量冗余.

• EGRET 对约 74% 的表达式生成反例个数少于 10 个, 近 90% 的表达式产生的反例少于 20 个, 大约 22% 的表达式没有反例生成. 表明 EGRET 的反例生成策略较为简单, 在实际应用中存在测试不够充分的问题.

(3) 反例串数量与正则表达式长度之间的关系.

此外, 我们还考察了本文算法生成的字符串数量与正则表达式长度之间的关系, 如图 11 所示.

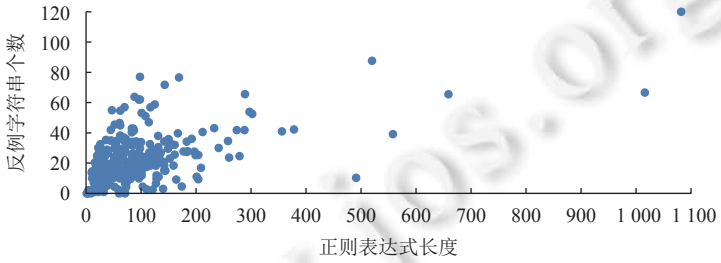


图 11 反例字符串个数与正则表达式长度之间的关系

• 正则表达式长度与字符串个数大致成正比, 长度较短的表达式产生的反例字符串也较少, 相反,较长的表达式其反例字符串规模较大. 这与理论分析相符合: 复杂的正则表达式会导致较多的变异体, 进而会生成规模较大的反例测试串.

• 正则表达式语法结构、运算符类型等诸多因素也会影响测试字符串的数量. 例如, 表达式“..\:..\:..\:..”和“3[47][0-9]{13}”长度相同, 但是前者运算符较为单一、结构比较简单, 构造出的变异体个数较少, 最终只得到 3 个反例串, 而后者可以生成 13 个反例串.

© 中国科学院软件研究所 http://www.jos.org.cn

5.3.2 反例串揭错能力

(1) 变异得分情况

变异得分是一种衡量测试数据集的揭错能力、评估测试充分性的重要指标. 变异得分越高说明测试数据的揭错能力越强. 我们使用如下公式来计算反例测试串集合 S 对于正则表达式 E 的变异得分.

$$MutationScore(S,E)=\frac{\#KilledMTs(S,E)}{\#TotalNonEqMTs(E)}\times 100\%,$$

其中, $\#KilledMTs(S, E)$ 表示由集合 S 杀死的表达式 E 的变异体总数, $\#TotalNonEqMTs(E)$ 表示 E 的不等价变异体总数. 在程序变异测试中, 根据给定的测试数据运行变异体, 通过比较变异体的运行结果和原程序的运行结果是否相同来判断变异体是否被杀死. 对于正则表达式 E 和其反例测试集 S, 由于 E 拒绝 S 中的所有字符串, 因此若变异体 M 接受 S 中的某个或者某些字符串, 则称测试集 S 杀死变异体 M; 反之, 若 M 同样拒绝 S 中的所有字符串, 则称测试集 S 没有杀死变异体 M. 我们对比了本文算法、MutRex 和 EGRET 生成的反例测试集的变异得分, 如图 12 所示. 变异体的生成采用本文的变异策略.

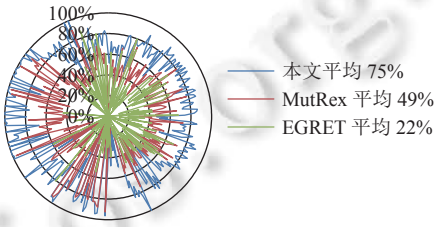


图 12 3 种算法生成反例字符串的变异得分

• 本文算法生成反例测试串的平均变异得分比 MutRex 高, 这与理论分析相符合: 本文增加了二阶变异, 同时引入更多的变异算子, 因此生成测试数据的揭示错误能力更强.

• 本文部分变异得分可以达到 100%, 然而由于测试集中只含有反例串, 而反例串不能杀死特化类变异体, 因此, 不能够达到全部 100% 的变异率.

• EGRET 的变异得分较低, 一方面由于 EGRET 的反例生成能力不足, 大约 1/4 的表达式没有反例产生; 另一方面, EGRET 的反例生成策略较为简单, 导致测试数据的揭示错误能力相对较弱.

(2) 实例分析情况

我们考察了生成的反例串在正则表达式测试中的实际应用情况. 特别地, 重点考察了本文新引入的变异算子以及二阶变异策略的有效性. 由于实验收集的正则表达式大多缺乏明确的规约说明, 无法准确判断预期语言. 例如, 有些描述浮点数的正则表达式允许含有小数点的串例如“.0”表示一个合法浮点数, 但拒绝不含小数点的串例如“0”, 而有一些则刚好相反. 从收集的数据中无法获得足够信息来确定生成的测试串符合预期还是一个错误. 因此, 此处只列举一些通过测试串发现的较为明显的常规错误.

① 新变异算子 CC2G 的有效性

正则表达式“([51|52|53|54|55]{2})([0-9]{14})”本意是描述以数字 51 到 55 开头后面由 14 个数字组成的万事达信用卡卡号, 但是错误地使用了字符集合“[]”和并“|”运算符. 实际上, 此时的“|”并不表示并运算而是表示字符“|”本身 (即, 该表达式等价于“[12345|]{2}[0-9]{14}”). 本文算法由 CC2G 算子产生的一条反例串“5151000000000000000”是合法卡号, 应该被接受, 因此帮助发现该错误. MutRex 对该错误表达式生成的 8 条反例“1A00000000000000”等均是非法卡号, 不能揭示该错误. EGRET 生成的 5 条反例也都是非法卡号, 同样不能揭示该错误.

② 新变异算子 UR 的有效性

正则表达式“($?\d{1, 3}(\, \d{3})*|(\d)+(\.\d{2}))”本意是描述含有两位小数点的美元格式, 子表达式“(\, \d{3})*|(\d)+”本应该表示整数部分的数字可用也可不用逗号隔开, 由于并运算的优先级问题导致定义不符合预期. 本文算法由 UR 算子生成一条反例“00.00”, 该反例符合预期的美元格式, 应该被接受,因而能够揭示原始表达式中

© 中国科学院软件研究所 http://www.jos.org.cn

的错误. 而 MutRex 对该表达式生成的 10 条反例“0.000”“0A00”“0000”“00.00”, 该反例符合预期的美元格式, 应该被接受, 因而能够揭示原始表达式中的错误. 而 MutRex 对该表达式生成的 10 条反例“0.000”“0A00”“0000”“?0”“0.0”“$$0”“0.”“0.000*”“.00”“0+.00”确实是应该被拒绝的字符串, 均不能帮助发现错误. 另外一条描述浮点计数法的正则表达式“([+-]?[0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*([eE][+-]?[0-9]+)?)”也存在优先级导致的错误, 该错误可以通过本文算法生成的反例“+1E0”揭示, 而 MutRex 生成的 16 条反例如“++0”“0E0E0”等均是非法的浮点计数格式, 不能揭示错误. EGRET 的反例在测试这两条表达式时同样没有能够揭露定义中的缺陷.

③ 二阶变异的有效性

二阶变异能够同时模拟表达式中的两处错误, 这种情况在实际中也经常出现. 例如, 用户想要描述整数部分是一位数字、小数部分最多 3 位数字的浮点数格式, 正确的表达式应该是“\d\.\d{1, 3}”, 但是在定义时误将元字符“.”当作普通字符, 并且不小心将“\d”的重复次数写错, 使用了错误的表达式“\d.\d{3}”. 本文算法组合 M2C 和 QC 两个变异算子构造二阶变异体“\d\.\d{2}”, 进而生成一条应该被接受的反例串“0.00”揭示原表达式中的两处错误. MutRex 对该错误表达式只做 1 次变异, 生成的 7 条反例“AA000”“0A0000”“0A00a”“0A00A”“0A”“aA000”“A000”都应该被拒绝, 同样, EGRET 生成的 2 条反例“0a00”“0a000”也不能揭示这两处错误.

5.3.3 算法运行时间

图 13 给出 3 种算法的运行时间对比情况, 其中横轴的正则表达式按照长度增序排列.

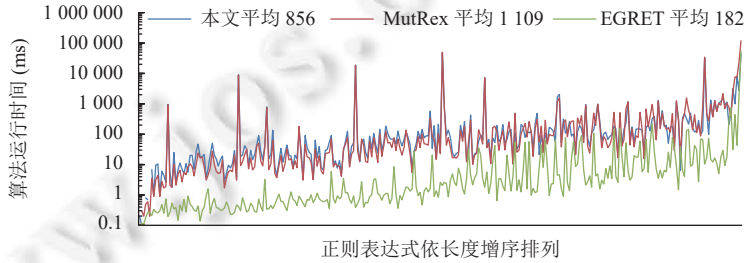


图 13 3 种算法生成反例字符串的运行时间

• 本文算法和 MutRex 算法的时间耗费主要在于构造变异体和根据变异体生成字符串, 这两部分都与变异体数量有关. 由于本文进行了变异约简, 平均变异体数量比 MutRex 少, 所以运行速度稍快. 本文产生测试集的时间平均不超过为 1 s, 最长约 49 s, 对于测试人员来讲在可接受范围之内.

• EGRET 的时间耗费主要在于自动机的构造和遍历, 自动机的大小与表达式长度有关, 因此运行时间基本上随着长度递增而增加. 由于 EGRET 不需要产生大量变异体, 因此运行速度最快.

6 相关工作

6.1 正则表达式测试串生成

正则表达式的测试串生成本质上归约为正则语言的生成问题. 在形式语言理论中, 抽样和枚举是语言生成的两种基本方式. 抽样旨在随机生成长度为 n 的字符串, 并且使得该语言中所有长度为 n 的字符串都具有相同的生成概率[36]. 枚举尝试按某种字典顺序列举语言中所有不同的字符串或一定范围内 (如长度为 n) 的所有字符串[37]. 现有一些正则表达式字符串生成工具如 Exrex[14]等采取的就是随机或者枚举的方式. 然而, 随机生成和枚举生成不适合用于探测错误为目的的正则表达式测试中. 首先, 随机算法一次只能产生 1 个字符串, 测试人员很难决定何时终止测试活动, 而枚举算法通常会产生大量的字符串, 这给测试任务带来比较大的开销; 其次, 这两种生成方式比较单一, 产生的字符串的错误检测能力不很令人满意, 文献 [9,18,19] 的对比实验也验证了这一点.

文献 [19] 提出一种基于正则表达式覆盖准则的测试串生成方法. 该文献借鉴软件测试中组合测试思想提出针对正则表达式的成对覆盖准则. 该准则要求正则表达式中连接的任意两个子表达式的所有可能组合都被测试到. 此外, 为了避免生成无限多个字符串, 成对覆盖仅限制闭包*的 3 种典型重复次数: 0 次、1 次和多次重复. 该文献实现了一

© 中国科学院软件研究所 http://www.jos.org.cn

种字符串生成算法, 根据给定的正则表达式生成一组满足成对覆盖准则的字符串. 算法接收符合传统语法定义的正则表达式, 即只支持连接、并、闭包等基本的正则算子. 该算法可用于测试 XML 模式的内容模型, 但不适合直接用于测试其他特殊应用如字符串模式匹配中的正则表达式, 这些应用中的表达式往往具有不同的语法形式.

上述算法都只能生成正则表达式的正例字符串. 目前已知能够同时提供正例和反例串的较为实用的工具主要有 EGRET[9]和 MutRex[17,18]. EGRET 基于正则表达式底层自动机基本路径覆盖, 重点在于生成正例, 反例生成策略比较简单. MutRex 基于变异的方式, 根据构造的一阶变异体产生相应的字符串. 本文延续 MutRex 的变异思想重点研究反例测试串生成算法, 通过提出新的变异算子、引入二阶变异以及采取变异约简等策略提高算法的反例生成能力, 生成具有较强揭错能力且规模适中的反例测试串.

6.2 正则表达式正确性保障

正则表达式正确性保障手段除了测试之外, 还有静态检查、形式化验证等. 文献 [38] 对相关工作进行了较为全面的调研. 与程序的静态分析类似, 可以通过对正则表达式进行静态检查以发现一些隐藏的语义错误或定义缺陷. 文献 [39] 调查了开发人员在定义用于字符串模式匹配的正则表达式时经常犯的错误, 总结出 11 条静态检查规则. 例如, 检查表达式中是否存在可疑范围的字符集, 或者检查表达式中的某些特殊字符如括号、引号等是否存在不配对的情况. 然而, 静态检查具有局限性, 提示的错误有限且可能出现误报的情况. 文献 [40] 提出一种正则表达式的验证框架, 该框架允许用户使用自然语言描述其预期定义的正则语言, 然后将自然语言描述的需求翻译成特定形式的形式化规约, 最后分别将规约与待验证表达式转换成自动机进行等价性判断, 从而完成表达式的语义正确性验证. 然而, 该方法在具体应用中存在困难. 首先, 不能保证用户给出的需求描述是正确和完备的, 其次, 正如文献 [40] 中指出, 从自然语言描述到形式规约的转换可能会引入错误, 导致最终验证结果的不准确性.

此外, 还有一些可视化解释工具[41,42]通过将正则表达式的语法结构以图形或高亮显示的方式展现, 提高表达式可读性和可理解性, 作为辅助性工具帮助用户发现定义中一些较为明显的缺陷, 但是揭示错误能力有限. 还有一些工作尝试从给定的字符串集合中自动合成正则表达式, 但是合成结果的正确性仍旧需要开发人员最终确认[43].

7 总 结

测试是保证正则表达式语义正确性的实用和有效手段. 现有的测试数据生成大多只关注正例串, 而研究表明实际开发中存在的错误大部分在于正则表达式所定义的语言比预期语言小, 即拒绝了一些应该被接受的字符串, 这类错误只能通过反例串才能发现. 本文提出一种基于变异的正则表达式反例串生成算法. 在原有变异算子基础上补充新的变异算子来更全面地模拟实际应用中的缺陷类型; 并引入二阶变异来模拟表达式中存在的复杂缺陷, 提高算法的反例生成能力; 同时采取冗余消除、变异约简等策略控制最终生成的测试集规模. 实验结果表明, 算法生成的测试数据具有较强的揭示错误能力, 而且测试集规模适中, 在保障正则表达式语义正确性方面有较高的应用价值.

本文只考虑了一阶变异和二阶变异, 在后续工作中将研究基于更高阶变异的测试生成. 同时, 将考虑丰富变异算子类别, 比如增加针对 CA、PA 等的逆算子以及增加二元运算符间相互替换的算子等. 变异体约简的双轮随机选择策略中不同随机比例对最终结果的影响也值得进行深入研究. 另外, 本文算法仅支持表达能力为正则语言的正则表达式. 在实际应用中存在许多含有非正则操作符如反向引用、递归子模式 (recursive subpattern) 等的扩展表达式. 含有非正则操作符的扩展表达式的测试串生成问题也将是未来的一个研究方向.

References:

[1] Zeek. The Zeek network security monitor. 2022. https://zeek.org/

[2] Gu HF, Zhang JN, Liu T, Hu M, Zhou JL, Wei TQ, Chen MS. DIAVA: A traffic-based framework for detection of SQL injection attacks and vulnerability analysis of leaked data. IEEE Trans. on Reliability, 2020, 69(1): 188–202. [doi: 10.1109/TR.2019.2925415]

[3] Murata M, Lee D, Mani M, Kawaguchi K. Taxonomy of XML schema languages using formal language theory. ACM Trans. on Internet Technology, 2005, 5(4): 660–704. [doi: 10.1145/1111627.1111631]

[4] Libkin L, Martens W, Vrgoč D. Querying graphs with data. Journal of the ACM, 2016, 63(2): 14. [doi: 10.1145/2850413]

[5] Navarrete LRR, Telles GP. Practical regular expression constrained sequence alignment. Theoretical Computer Science, 2020, 815:

© 中国科学院软件研究所 http://www.jos.org.cn

95–108. [doi: 10.1016/j.tcs.2020.02.017]

[6] Chapman C, Stolee KT. Exploring regular expression usage and context in Python. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 282–293. [doi: 10.1145/2931037.2931073]

[7] Davis JC, Coghlan CA, Servant F, Lee D. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 246–256. [doi: 10.1145/3236024.3236027]

[8] Spishak E, Dietl W, Ernst MD. A type system for regular expressions. In: Proc. of the 14th Workshop on Formal Techniques for Java-like Programs. Beijing: ACM, 2012. 20–26. [doi: 10.1145/2318202.2318207]

[9] Larson E, Kirk A. Generating evil test strings for regular expressions. In: Proc. of the 2016 IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). Chicago: IEEE, 2016. 309–319. [doi: 10.1109/ICST.2016.29]

[10] Erwig M, Gopinath R. Explanations for regular expressions. In: Proc. of the 15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE). Tallinn: Springer, 2012. 394–408. [doi: 10.1007/978-3-642-28872-2_27]

[11] Bai GR, Clee B, Shrestha N, Chapman C, Wright C, Stolee KT. Exploring tools and strategies used during regular expression composition tasks. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Montreal: IEEE, 2019. 197–208. [doi: 10.1109/ICPC.2019.00039]

[12] Wang PP, Stolee KT. How well are regular expressions tested in the wild? In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 668–678. [doi: 10.1145/3236024.3236072]

[13] Wang PP, Brown C, Jennings JA, Stolee KT. Demystifying regular expression bugs. Empirical Software Engineering, 2022, 27(1): 21. [doi: 10.1007/s10664-021-10033-1]

[14] Exrex. 2022. https://github.com/asciimoo/exrex

[15] Generex. 2022. https://github.com/mifmif/generex

[16] Regldg. 2022. https://regldg.com/

[17] Arcaini P, Gargantini A, Riccobene E. MutRex: A mutation-based generator of fault detecting strings for regular expressions. In: Proc. of the 2017 IEEE Int'l Conf. on Software Testing, Verification and Validation Workshops (ICSTW). Tokyo: IEEE, 2017. 87–96. [doi: 10.1109/ICSTW.2017.23]

[18] Arcaini P, Gargantini A, Riccobene E. Fault-based test generation for regular expressions by mutation. Software Testing, Verification and Reliability, 2019, 29(1–2): e1664. [doi: 10.1002/stvr.1664]

[19] Zheng LX, Ma S, Wang YY, Lin G. String generation for testing regular expressions. The Computer Journal, 2020, 63(1): 41–65.

[20] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978, 11(4): 34–41. [doi: 10.1109/C-M.1978.218136]

[21] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009, 51(10): 1379–1393. [doi: 10.1016/j.infsof.2009.04.016]

[22] Thompson HS, Mendelsohn N, Beech D, Maloney M. W3C XML schema definition language (XSD) 1.1 part 1: Structures. The World Wide Web Consortium (W3C), W3C Working Draft, 2009.

[23] Clark J, Makoto M. Relax NG tutorial. OASIS Committee Specification. 2001. http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html

[24] Pan R, Hu QHP, Xu GW, D'Antoni L. Automatic repair of regular expressions. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 139. [doi: 10.1145/3360565]

[25] Møller A. Dk.brics. automaton—Finite-state automata and regular expressions for Java. 2022. http://www.brics.dk/automaton/

[26] King KN, Offutt AJ. A fortran language system for mutation-based software testing. Software: Practice and Experience, 1991, 21(7): 685–718. [doi: 10.1002/spe.4380210704]

[27] Chen X, Gu Q. Mutation testing: Principal, optimization and application. Journal of Frontiers of Computer Science and Technology, 2012, 6(12): 1057–1075 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.2012.12.001]

[28] Ghiduk AS, Girgis MR, Shehata MH. Higher order mutation testing: A systematic literature review. Computer Science Review, 2017, 25: 29–48. [doi: 10.1016/j.cosrev.2017.06.001]

[29] Song L, Liu J. Second-order mutant reduction based on SOM neural network. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1464–1480 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5723.htm [doi: 10.13328/j.cnki.jos.005723]

[30] Zhang L, Hou SS, Hu JJ, Xie T, Mei H. Is operator-based mutant selection superior to random mutant selection? In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering. Cape Town: ACM, 2010. 435–444. [doi: 10.1145/1806799.1806863]

© 中国科学院软件研究所 http://www.jos.org.cn

[31] Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. Journal of Systems and Software, 1995, 31(3): 185–196. [doi: 10.1016/0164-1212(94)00098-0]

[32] Regular Expression Library. 2022. http://www.regexlib.com

[33] Goyvaerts J. Regex Tutorial, Examples and Reference. 2022. https://www.regular-expressions.info/

[34] Bala S. Intersection of regular languages and star hierarchy. In: Proc. of the 29th Int'l Colloquium on Automata, Languages and Programming. Malaga: Springer, 2002. 159–169. [doi: 10.1007/3-540-45465-9_15]

[35] Li YT, Zhang XL, Peng FF, Chen HM. Practical study of subclasses of regular expressions in DTD and XML schema. In: Proc. of the 18th Asia-Pacific Web Conf. on Web Technologies and Applications. Suzhou: Springer, 2016. 368–382. [doi: 10.1007/978-3-319-45817-5_29]

[36] Bernardi O, Giménez O. A linear algorithm for the random sampling from regular languages. Algorithmica, 2012, 62(1–2): 130–145. [doi: 10.1007/s00453-010-9446-5]

[37] Ackerman M, Shallit J. Efficient enumeration of words in regular languages. Theoretical Computer Science, 2009, 410(37): 3461–3470. [doi: 10.1016/j.tcs.2009.03.018]

[38] Zheng LX, Ma S, Chen ZX, Luo XY. Ensuring the correctness of regular expressions: A review. Int'l Journal of Automation and Computing, 2021, 18(4): 521–535. [doi: 10.1007/s11633-021-1301-4]

[39] Larson E. Automatic checking of regular expressions. In: Proc. of the 18th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). Madrid: IEEE, 2018. 225–234. [doi: 10.1109/SCAM.2018.00034]

[40] Liu X, Jiang YF, Wu DH. A lightweight framework for regular expression verification. In: Proc. of the 19th IEEE Int'l Symp. on High Assurance Systems Engineering (HASE). Hangzhou: IEEE, 2019. 1–8. [doi: 10.1109/HASE.2019.00011]

[41] Budiselic I, Srbljic S, Popovic M. RegExpert: A tool for visualization of regular expressions. In: Proc. of the 2007 Int'l Conf. on "Computer as a Tool". Warsaw: IEEE, 2007. 2387–2389. [doi: 10.1109/EURCON.2007.4400374]

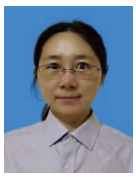
[42] Beck F, Gulan S, Biegel B, Baltes S, Weiskopf D. RegViz: Visual debugging of regular expressions. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 504–507. [doi: 10.1145/2591062.2591111]

[43] de Almeida Farzat A, de Oliveira Barros M. Automatic generation of regular expressions for the Regex Golf challenge using a local search algorithm. Genetic Programming and Evolvable Machines, 2022, 23(1): 105–131. [doi: 10.1007/s10710-021-09411-x]

附中文参考文献:

[27] 陈翔, 顾庆. 变异测试: 原理、优化和应用. 计算机科学与探索, 2012, 6(12): 1057–1075. [doi: 10.3778/j.issn.1673-9418.2012.12.001]

[29] 宋利, 刘靖. 基于SOM神经网络的二阶变异体约简方法. 软件学报, 2019, 30(5): 1464−1480. http://www.jos.org.cn/1000-9825/5723.htm [doi: 10.13328/j.cnki.jos.005723]



郑黎晓(1983－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件测试, 形式语言与自动机.

陈祖希(1981－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为形式化方法.

余李林(1997－), 男, 博士生, 主要研究领域为软件工程, 数据科学, 自然语言处理.

骆翔宇(1974－), 男, 博士, 教授, CCF 专业会员, 主要研究领域为形式化方法, 模型检测, 时态逻辑, 多智能体系统.

陈海明(1966－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为软件设计方法和形式规约, 程序语言.

汪小勇(1976－), 男, 博士, 高级工程师, 主要研究领域为交通运输系统信息与控制.

© 中国科学院软件研究所 http://www.jos.org.cn