软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(1):110−126 [doi: 10.13328/j.cnki.jos.005643]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

程序理解:现状与未来*

金 芝[1,2], 刘 芳[1,2], 李 戈[1,2]

[1](北京大学 信息科学技术学院,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 金芝, E-mail: zhijin@pku.edu.cn

摘 要: 程序理解是软件工程中的关键活动,在软件开发、维护、重用等任务中发挥着重要的作用.程序理解自软件工程出现以来,就一直是该领域的研究热点.随着软件应用的日益复杂和不断普及,程序理解研究的需求发生了新的变化,程序的自理解或自认知逐渐成为新的关注点,有必要对程序理解进行重新审视.从工程、学习和认知以及方法和技术这 3 个角度定位程序理解任务;随后,通过文献分析展示其研究布局,进而分别从认知过程、理解技术以及软件工程任务中的应用这 3 个方面,综合论述程序理解研究的发展脉络和研究进展.

关键词: 软件工程;程序理解;软件理解;程序静态分析;程序动态分析

中图法分类号: TP311

中文引用格式: 金芝,刘芳,李戈.程序理解:现状与未来.软件学报,2019,30(1):110−126. http://www.jos.org.cn/1000-9825/5643.htm

英文引用格式: Jin Z, Liu F, Li G. Program comprehension: Present and future. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1):110−126 (in Chinese). http://www.jos.org.cn/1000-9825/5643.htm

Program Comprehension: Present and Future

JIN Zhi[1,2], LIU Fang[1,2], LI Ge[1,2]

[1](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871, China)

Abstract: Program comprehension is a key activity in software engineering and plays an important role in software development, software maintenance, and software reuse. Since the advent of software engineering, program comprehension has always been a hot research hotspot issue in this field. With the increasing complexity and popularity of software, the needs for program comprehension have been changed. The program self-understanding and self-awareness have gradually become new focuses. Therefore, it is highly desired to re-examine the purposes, the tasks and the techniques of program comprehension. Firstly, this paper dicsusses the program comprehension from the 3 perspectives, namely, the engineering, the learning cognition, as well as the techniques. Then, it shows the degree of research attentions through literature analysis. Furthermore, it discusses the research progress from three aspects, i.e., the cognitive process, the methods and techniques, and the software engineering tasks. Finally, it discusses the development trend and challengs.

Key words: software engineering; program comprehension; software comprehension; static program analysis; dynamic program analysis

程序理解是软件工程中的一个经典话题,又叫作软件理解或系统理解.自软件出现以来,甚至在软件工程提

* 基金项目: 国家重点基础研究发展计划(973)(2015CB352201); 国家自然科学基金(61620106007, 61751210)

Foundation item: National Basic Research Program of China (973) (2015CB352201); National Natural Science Foundation of China (61620106007, 61751210)

本文由“软件学科发展回顾特刊”特约编辑梅宏教授、金芝教授、郝丹副教授推荐.

收稿时间: 2018-08-08; 修改时间: 2018-08-30; 采用时间: 2018-09-25; jos 在线出版时间: 2018-11-20

CNKI 网络优先出版: 2018-11-21 09:52:33, http://kns.cnki.net/kcms/detail/11.2560.TP.20181121.0952.001.html

出之前,就有了程序理解这一问题.1968 年第 1 次软件工程研讨会之后,程序理解成为软件工程中的关键活动,在进行软件重用、维护、迁移、逆向工程以及软件系统扩展等任务时,都要依赖于对程序的理解[1].比如,源码逆向工程通过分析目标系统来识别系统的成分及其相互关系,创建系统的更高抽象层次上的表示,包括构建目标系统的概念模型、抽取程序结构和控制信息以及进行数据抽取和系统抽象.程序理解是软件逆向工程的主要实现手段和行为活动,贯穿整个软件逆向工程,是决定软件逆向工程成败的关键[2].在软件维护和演化中,例如软件适应(adaptive)、软件修复(corrective)、软件复用(reuse)等任务,程序理解也是其首要活动,其他活动都是在程序理解的基础上进行的.例如:完成软件适应任务的流程包括理解系统、定义适应需求、制定适应策略、设计、代码修改、调试、回归测试;软件修复任务的流程包括理解系统、提出/评估问题假设、修复代码、回归测试.随着程序设计语言的不断变化和软件复杂性的不断提高,程序理解一直面临着挑战,新的研究问题不断涌现.

什么是“程序理解”呢?首先看看什么是“理解”.在 Wikipedia 中,“理解(comprehension 或 understanding)”(https://en.wikipedia/wiki/Comprehension)是指与抽象或物理对象相关的心理过程,比如:理解一个人、一件事或一段话,就是建立理解的主体和这个人、这件事或这段话之间的关系,表明这个主体能够想象出这个对象(具有心理表征),并且/或者能用一些概念恰当地刻画这个对象(https://en.wikipedia/wiki/Understanding).可以看出:理解是和学习相关的活动,需要建立从理解的对象所属的概念域到理解的主体所熟悉的概念域的(概念)映射.

从学习和认知的角度,可以把程序理解和自然语言文本理解进行类比.自然语言文本是用自然语言表述的一种认知,阅读和理解自然语言文本,就是从文本中获取知识的过程.程序通常看作是用程序设计语言表述的一种认知,因此可以抽象地把程序理解解释为从程序中获得关于程序所表达的知识的过程.与自然语言理解的语法、语义和语用等理解阶段相比较,程序理解可分为侧重文法理解的语法层上的程序分析、侧重抽象语义(如信息流、控制流、抽象解释等)提取的语义层上的程序分析以及侧重程序动态语义的程序行为分析.这体现了程序理解技术和结果的由浅入深的层次划分.

从方法和技术的角度,程序理解以程序分析为基础.通过对程序进行人工或自动分析,以验证、确认或发现软件性质.程序分析贯穿于软件开发、维护和复用阶段:在开发阶段,对正在开发的程序进行分析,以快速、高效地开发出高质量的软件;在维护阶段,对已经开发、部署或运行的程序进行分析,以准确理解并维护该程序,从而使其能够提供更好的服务;在复用阶段,对之前开发的程序进行分析,以复用其中有价值的成分.上述不同阶段的程序分析过程差异较大,需要的分析技术也各有不同.根据其分析过程是否需要运行程序,可以将程序分析分为静态分析和动态分析:静态分析不需要运行程序,直接对程序源代码进行分析,获取相关信息[3,4];动态分析则是分析程序运行时性质[5],通过程序的运行获取程序的输入/输出关系或程序内部状态等,以验证或发现程序的性质.

从工程应用的角度看,程序理解主要用于软件开发过程中涉及已有软件制品及其使用和变更的活动.比如:软件复用需要将已有软件制品用于当前软件开发;软件逆向工程从已有软件制品中获取该软件制品的高层抽象,所谓“逆向”是针对自顶向下的软件开发过程而言的;软件演化则在认识到当前软件已经不适合新的场景时,需要进行适当的软件调整和变更.在进行这些软件工程任务时,需要理解当前软件制品的含义,包括它所表达的软件能力及其实现方式,等等.

已有的工作从各自的研究侧重点出发,给出了程序理解的不同的说法.一些典型的说法比如:

- 程序理解是根据软件源代码去理解程序的过程[6];
- 程序理解的任务是构建软件的从代码模型到应用领域问题间各抽象层次上的模型,目的是支持软件维护、演化和再工程[7];
- 程序理解是利用领域知识以及语法语义知识来理解软件制品,构建软件制品与使用场景之间的心理认知模型(区别于物理模型)的过程[8];
- 程序理解是通过研究诸如源代码和文档等制品来理解软件系统内部工作原理,为软件维护任务提供足够的信息[9];
- 程序理解是理解整个软件系统或部分软件系统如何运行的活动[10];

- 程序理解的任务是解释软件行为,通常是通过阅读源代码,来判断它的作用和代码如何与整个软件进行交互[11].

当前,云计算和人机物融合应用模式不断涌现,加速了社会的信息化进程,使软件成为一种社会基础设施.软件的可成长性和可持续演化性成为软件工程的重要关注点[12].图灵奖得主Sifaki教授曾指出,(软件)系统工程的关注点正在发生转变,体现为:(1) 从小规模集中式不可进化的系统,到大型分布式可演化的系统;(2) 从严格控制的系统与外部环境的交互,到不可预测的动态变化环境;(3) 从设计正确性到通过适应性确保正确性[13].软件具有可成长和可持续演化能力的基本条件,是软件的可理解性甚至软件的可自我认知性.另一方面,软件自适应性和软件智能化等研究热点的形成,感知软件运行状态和运行环境,以及以此为基础的运行时决策和系统适应性变更和持续演化,已经成为研究的主要关注点,这更强调了运行时系统的可理解性将成为亟待解决的问题.

因此,软件(程序)理解,特别是软件的自理解和自认知,已经被提到了前所未有的重要高度,有必要重新审视软件理解的内涵,分析目前的技术手段,结合系统工程的发展探索其新的需求,从而展望未来的技术挑战和趋势.本文首先通过文献分析展示长期以来软件工程领域对程序理解的研究布局,进而分别从学习和认知的角度、方法和技术的角度以及软件工程应用的角度这 3 个问题,综合分析程序理解研究的发展脉络和研究进展,最后,根据软件系统的发展趋势探讨程序理解面临的挑战和发展趋势.

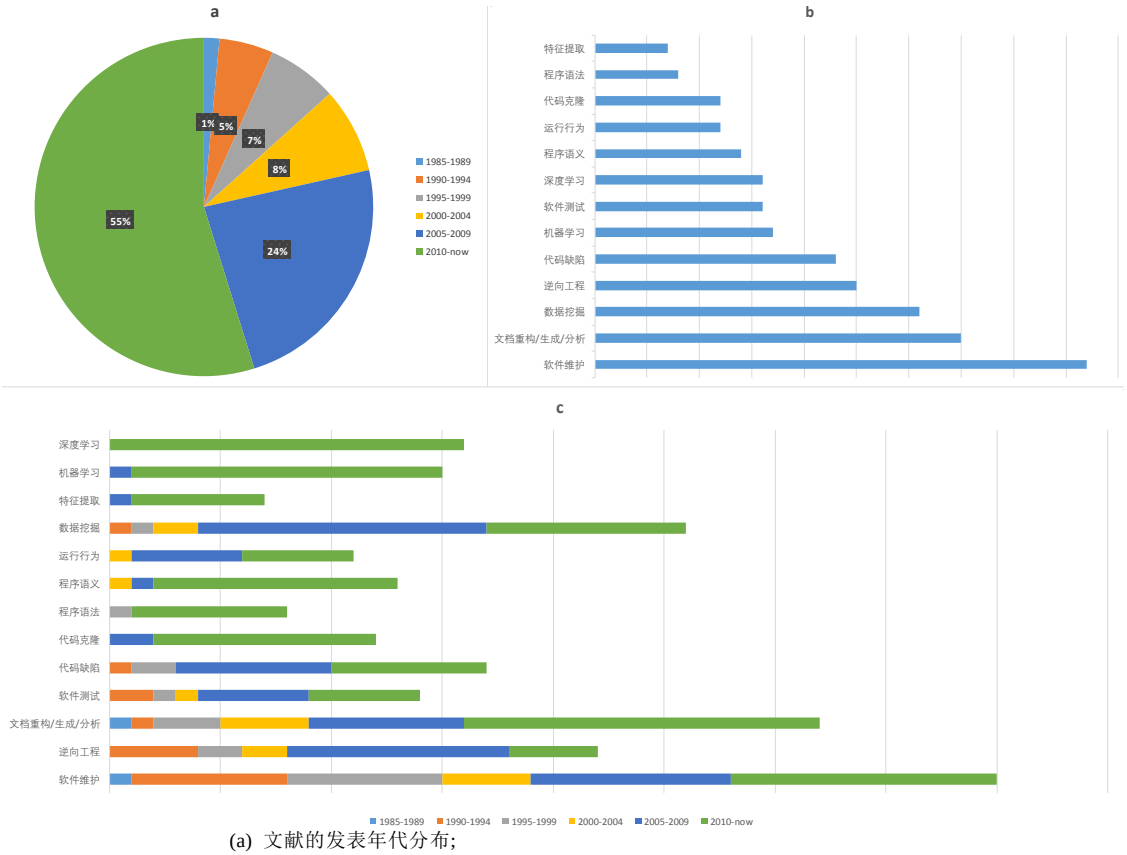
1 程序理解研究关注度分析

考察研究人员对程序理解研究方向的关注度,有一个维度是查看发表文章的强度.我们用“程序理解”/ “软件理解”等关键词在 IEEE Xplore Digital Library 上检索近年来发表的文献.使用程序理解(program comprehension)作为关键词,检索到 2 124 篇文章;使用软件理解(software comprehension)作为关键词,检索到 2 115 篇文章.这两个关键词的检索结果有很大部分是重合的.从检索结果中的文献来看,1985 年之前发表的文章数目较少,因此我们从 1985 年之后发表的文献开始统计.对总共 4 139 个检索结果进行人工筛选,去掉重合的文献,并通过人工阅读文章标题、文章摘要和关键词,选取出与程序理解的具体任务和理解技术相关的文献共 136 篇.对它们从文献发表年代、研究主题分布以及研究主题各年代分布等几个维度进行了统计,得到如图 1 所示的分布结果.可以看出:

(1) 自 1985 年起,对程序理解研究的关注度一直处于上升趋势(如图 1(a)所示);
(2) 软件维护任务和程序理解最相关,研究工作占有很大份额;其次是文档分析、重构和生成以及逆向工程(如图 1(b)所示);
(3) 机器学习和深度学习技术近年来大量进入程序理解领域(如图 1(c)所示);
(4) 文献所报道的研究主题完整地覆盖了程序理解研究的 3 个方面(如图 1(c)所示).
    a. 比如数据挖掘、特征提取、机器学习和深度学习等,是具有代表性的程序理解技术.其中,机器学习、特征提取、深度学习的程序理解方法,是最近 10 年出现的,并正日渐成为前沿热点;
    b. 对程序的语法分析、语义理解和程序的动态行为分析,表现了程序理解的认知层次.值得注意的是:程序语法分析的研究在 1995 年后出现,程序语义理解和运行行为分析的工作在 2000 年后出现,表现了程序理解从语法到语义再到语用的认知不断深入的过程;
    c. 软件维护、逆向工程、文档重构和生成、软件测试、代码缺陷检测和代码克隆等软件工程的经典任务,是程序理解的主要应用场景,除了软件测试和逆向工程外,应用热度一直处于上升趋势.

除此之外,本文还根据程序理解相关会议的投稿论文数,展示该领域的研究关注度,包括统计软件工程顶级会议 ICSE(Int’l Conf. on Software Engineering)近几年来各个主题提交文章数以及程序理解会议 ICPC(Int’l Conf. on Program Comprehension)近 10 年来录用文章数.其中,图 2 展示了 ICSE 2015~2018 各个主题的提交情况,列出了提交文章数排名前十的主题情况(2015 年和 2017 年未设置实证软件工程这个主题,2017 年未设置软件工程的人为和社会因素这个主题).其中,软件测试、程序分析、软件演化与维护、软件调试、错误定位和修复等主题与程序理解密切相关(用蓝色色系表示),这几个主题的文章数目一直占全部文章数的很大份额.可见,

程序理解在软件工程领域中占据着重要的地位.



(a) 文献的发表年代分布;
(b) 文献的研究主题分布;
(c) 研究主题和发表年代的相关分布(本文关键词统计,选取了出现频率较高并具有代表性的关键词,忽略了覆盖面过于宽泛的关键词)

Fig.1 Statistical analysis of program comprehension literature

图 1 程序理解研究论文的统计分析

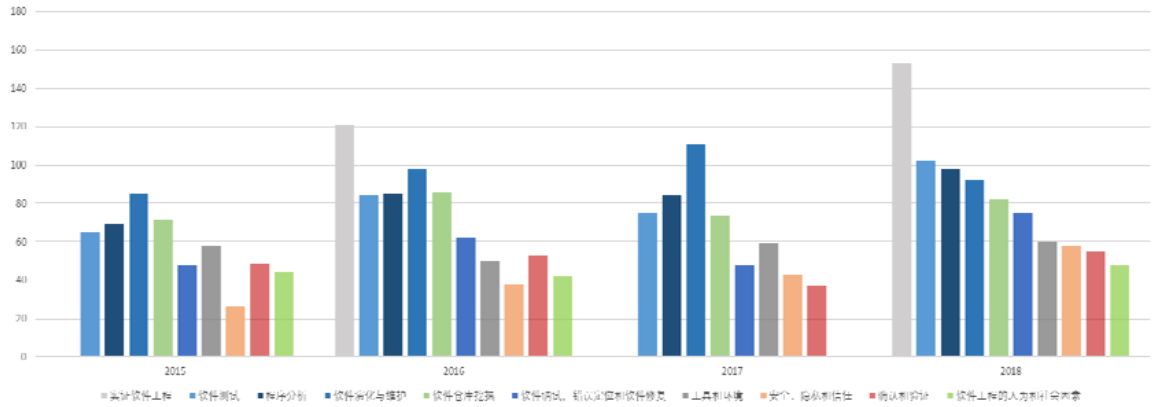


Fig.2 Top ten topics in the submissions of ICSE from 2015 to 2018

图 2 2015 年~2018 年 ICSE 提交文章数目前十的主题及其提交文献数目

ICPC 是专门以程序理解为主题的国际会议,该会议平均每年大概录用 40 篇文献,近 10 年来的录用文章数如图 3 所示,整体呈缓慢上升趋势.这表明,程序理解相关研究在近 10 年来一直都是较为热门的话题,可以推测,程序理解在未来仍然会是活跃的研究主题.

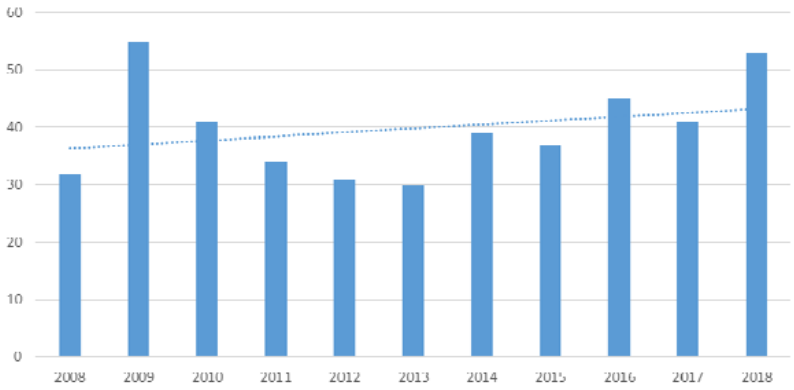


Fig.3 Number of accepted papers of ICPC in recent ten years

图 3 ICPC 近 10 年录用文章数目统计

2 程序理解的认知过程、技术方法和工程应用

2.1 从认知过程看程序理解

程序理解本身就是一个学习和认知的过程.程序是知识的编码,程序理解是指从程序中学习程序所表达的领域知识的过程.根据构建式学习的理论,学习和认知的过程是面向目标或模型驱动的归纳抽象过程:第一,学习是学习者个体从学习材料中通过提取或抽象等手段构建知识体的过程;第二,所构建的知识体除了依赖学习材料,很大程度上还依赖于学习者已有的知识背景和该次学习的目的.

早期很多对程序理解的探讨,都是从认知的角度出发进行的,它们大都侧重于研究程序员的学习过程和知识的构建过程.从构建式学习理论出发,可以有两种基本的程序理解策略.

- 自底向上的策略[1]:这种策略将程序理解看作是一个逐层聚合的过程.也就是从阅读程序源码开始,将这些源码中包含的信息逐层组合、抽象成更高层次的信息.例如,一种程序认知框架区分语法和语义两个层次:语法层关注程序的语句和基本单元;语义层与程序语言无关,是逐层抽象直到所抽象的知识能够描述应用领域[14].另一种认知框架区分程序模型(program model)和情景模型(situation model):程序模型对程序的控制流进行抽象;情景模型则是封装了关于数据流抽象和功能抽象的知识,表达了程序的目标层次[15].用传统的软件开发方式可以解释这种策略背后的原因,即,程序理解是为了还原自底向上的软件开发过程;
- 自顶向下的策略[1]:这种策略将程序理解看作是重新组织程序员自身已有的关于程序应用领域的知识,并建立这些知识到程序源码上的映射.比如,一个典型的过程是:从关于领域应用的假设出发,根据已有的领域知识进行逐层分解,同时与程序源码进行比对或验证,从而建立假设和源代码之间的关联.这种方式体现了模型驱动的思想,亦即:如果模型存在,程序理解就是将具体的程序片段关联到已知的模型上,通过模型的结构和语义去刻画程序的结构和语义.

对复杂一点的程序而言,仅仅只采用一种策略是不可行的,这两种策略常常需要结合起来交替地使用,因而产生了集成的策略.

- 集成式程序理解模型[16]:集成式程序理解模型结合了自底向上和自顶向下的策略,它认为:对比较熟悉的代码,可以采用自顶向下的方式进行理解,直接建立源码和应用知识之间的关联;而对不太熟悉的代码,则需要采用自底向上的策略从底层开始进行逐层抽象,以获得程序所表达的领域知识.比如,von

Mayrhauser 等人提出的集成式程序理解模型涉及 4 个成分.

(1) 自顶向下的策略:从领域知识出发产生假设,并不断细化假设,直到假设能关联到熟悉的代码片段上;
(2) 构建程序模型:从未知代码出发试图理解其控制流;
(3) 构建情景模型:从已构建的程序模型出发,发现数据流,得出功能抽象;
(4) 领域知识:包含程序员已经掌握的关于应用领域、应用目标、程序语言等的知识,用于支持其中的推理过程.

这些早期研究结果的获得,得益于对程序员认知过程的实验研究.这些实验性工作的目的是:研究程序员如何理解程序,进而找到能够帮助程序员理解程序或者提高理解效率的方式.这类实验工作一般采用诸如内省法(有声思维法)、记忆测试法等认知心理学的方法.比如,文献[17]中的有声思维法的实验显示:程序员在遇到熟悉的领域时,首先陈述领域假设(自顶向下法);而当遇到不熟悉的领域时,要结合源码作推断(自底向上法).文献[16]中的有声思维法实验也证实:程序员在理解比较大的软件系统的源码时会涉及多个不同的策略,用到多种模型,在理解的过程中经常在不同策略和不同模型之间切换(集成式方法).

记忆测试实验则希望展示源码理解和源码记忆之间的关系.文献[18]的记忆测试实验显示:按可执行过程排序的源码,比乱序源码更容易记忆.文献[19]中的实验发现:有经验的程序员主要是依靠命名惯例进行理解和记忆,当碰到不熟悉的命名方式时也和新手一样没有头绪.还有实验表明:当程序语句之间具有逻辑性时,程序员的反应更快[15].这些实验性研究都表明,源码的可理解性和可记忆性是正相关的.

这些关于程序可理解性的实验研究,在一定程度上引出了集成式开发环境的设计.比如,大部分 IDE 采用的缩进格式、合适的字符间隔宽度、用颜色标定不同类型的语句等策略,它们都是为了提高程序可理解性而引入的编程布局上的考虑[20].

高级语言以及 API 和领域特定语言(DSLs)也与这些早期程序理解研究同步发展.高级语言的出现和发展,对不断提升程序的可理解性起到重要的作用,方便了软件开发、软件重用、软件系统逆向工程和系统演化等.

值得注意的是:沉寂了多年的程序可理解性的实验研究,由于新的实验技术的出现,近年来又重新引起了学术界和工业界关于程序理解作为认知过程研究的关注.比如,研究者们利用功能磁共振成像、近红外光谱学、脑电图等实验手段,观察程序员在理解程序的过程中脑部区域的激活或活跃状态,试图建立不同的激活区域与编程任务的难度之间的相关性[21−23].

2.2 程序理解的方法

从程序理解方法的角度,现有工作可以分为基于分析的方法和基于学习的方法.

- 基于分析的方法以是否需要运行软件为准则,又可以分为静态分析方法和动态分析方法[24].
  - 静态程序理解直接分析程序源代码,从中获取相关信息,分析过程不需要执行程序[3,4].由于其不必运行系统,因而可进行系统的早期分析,在系统开发阶段就可以开始,分析结果具有普适性,也可以覆盖所有可能的执行路径;
  - 动态程序理解主要理解程序运行时性质[5],它通过执行程序,获取其输入输出关系或内部状态等,提取软件的性质.与静态方法相比,其特点是需要运行系统,通常需要具体的输入数据,因为有具体数据,其分析结果更为精确;但反过来,因为结果的精确是对特定输入而言的,不能保证分析结果对其他输入也适用,因此不具有普适性.
- 近年来,数据挖掘和机器学习技术在多个领域获得成功,特别是深度学习在图像处理、语音识别、自然语言处理等领域的多项任务上取得的成功.在软件工程领域,由于开源代码的大量出现,软件工程数据越来越容易获取,基于学习的技术近年来也开始得到关注.许多研究者利用数据挖掘、机器学习等技术进行程序理解,从源代码和代码文档中获取信息.深度学习技术越来越多地应用于程序理解,成为一种数据驱动的端到端的程序理解方法.

2.2.1 基于静态分析的程序理解

程序静态分析由 M.E.Fagan 于 1976 年提出[25],主要通过分析程序源码以发现其中的错误.Fagan 于 1976 年发表的文献[25]提到,使用静态分析技术发现了 30%~70%的逻辑和编码错误.后来,程序静态分析成为程序理解的重要方面.

按照理解的层次,程序静态分析可分为语法分析和语义分析.程序语法分析按照编程语言的语法和词法规则对程序源码进行分析,可以检查程序在语法上是否符合预定义的巴克斯范式(Backus normal form,简称 BNF),从而发现语法错误.程序语言的巴克斯范式一般属于上下文无关文法,程序语法分析技术则主要包括算符优先分析(自底向上)、递归下降分析(自顶向下)和 LR 分析(自左至右、自底向上)等,含有语法错误的程序无法解析为语法树.程序静态分析还可以进一步分析所生成的语法树,以完成程序理解的其他相关任务.例如在程序克隆检测中,通过对比程序的语法树之间的相似性来检测程序克隆[26].

程序静态分析与自然语言语法分析相似,但程序具有更强的结构性且存在大量的嵌套结构,程序语法树比自然语言语法树要复杂得多,分析难度也更大.而且自然语言即使存在语法错误,人仍可以根据其上下文来理解它的含义,但程序需要可执行才能体现其价值,含有语法错误的程序不能被正确执行,因此程序分析中语法约束更加严格.

软件系统用于求解特定领域问题,软件系统的语义本质上就是问题的求解过程.比如,软件系统的控制流从操作的角度表达问题求解过程,它包括过程的细分,表达操作的顺序步骤以及包含 if-then-else 条件、循环和/或分支路径等操作的控制转移.软件系统的数据流从数据传递和加工角度表达了系统的逻辑功能,即数据在系统内部的逻辑流向和逻辑变换过程.通过各种程序分析的手段抽象出系统的控制流和/或数据流的过程,可以看作是从问题求解过程视角出发的程序语义理解.

更深一个层次是程序的形式语义,它描述程序执行时所遵循的规约,是关于程序执行性质的抽象.比如通过描述程序(程序语句)的输入输出关系,或者解释程序的执行步骤来创建其计算模型.关于这类语义的刻画依赖于编程语言的形式语义.编程语言形式语义有 3 种:一是操作语义,它通过规约程序在抽象机上的执行过程来描述语言的语义;二是指称语义,它认为程序(或者程序成分)的语义是执行该程序(或成分)所得到的最终效果,这个最终结果才是程序成分所要表达的含义,它通过构造可描述语句含义的数学概念(称为指称)来表达语言的语义;三是公理语义,它使用公理化方法,通过描述程序语句在程序状态断言上的效果来定义程序语句的语义.

形式语义精确地表达了程序的执行过程或执行效果,但构造形式语义相当困难,目前只有一些非常有限的辅助手段.比如:符号执行使用抽象符号来表达程序变量的值,用一个解释器来跟踪程序,从而确定哪些输入会引起程序各个部分的执行[27];然后,根据程序中表达式和变量的符号来获得表达式,根据每个条件分支的可能结果获得这些符号的约束.由于需要穷举各种可能执行的路径,其搜索空间将随程序规模的扩大呈指数级增长.

2.2.2 基于动态分析的程序理解

程序动态分析用于分析程序运行时性质,分析时需要采集程序运行时信息,通过分析这些运行时信息来获得程序运行时属性.程序动态分析的原因包括:一方面,动态链接库被广泛使用,软件只有在完成部署之后才能完整地表达程序的功能,这种情况下,传统的仅依赖源码分析的静态分析获得的结果由于可能缺少涉及动态链接库的信息而不够精确[28];另一方面,面向对象语言(尤其是 Java)被广泛使用,这类语言具有动态绑定、多态、线程等特征,这些特性也使静态分析的结果受到限制.

基于插装的方法,将监测代码插装到程序中,以捕获程序执行路径等相关信息[29].监测代码可以插装在源代码、二进制码以及字节码中.源代码插装在程序编译前进行,它在需要监测的地方直接加上监测代码,例如增加输出信息语句、增加日志语句等.二进制码插装需要修改或重写编译代码来添加监测代码,分静态插装和动态插装:静态二进制码插装直接编写可执行二进制码重写应用程序;动态二进制插装在程序加载到内存后,在执行之前进行.字节码插装在已编译的代码中进行追踪,同样也可分为静态的或者动态的:静态字节码插装在程序执行之前离线更改已编译代码,创建已插装的中间代码的副本;动态字节码插装则在程序运行时进行.

基于虚拟机分析的方法通过使用特定虚拟机提供的分析(profiling)和调试(debugging)机制来执行动态分

析[30],以深入了解程序的内部操作,特别是与内存和使用堆的相关操作.

从某种意义上说,基于动态分析的程序理解可以类比于自然语言理解中的一种语用性理解,亦即在实际的程序运行过程中捕捉程序的含义,建立程序的执行及输入输出之间的依赖关系.

2.2.3 基于学习的程序理解

随着大量开源代码的出现,程序相关数据越来越容易获得.同时,软件是知识的载体,软件开发是人类表达知识的行为,对大量代码进行统计建模来理解程序获取知识具有合理性.因此,许多研究开始采用机器学习技术,学习代码数据和代码文档数据等其中蕴含的知识[31].这些研究通过运用统计学习和机器学习相关技术来挖掘程序的特征,进行程序理解.

统计语言模型用于学习代码中的统计特性.*N*-gram[32]是使用最为广泛的模型,它基于的假设是:第 N 个词的出现只与前 $N-1$ 个词相关,因此可以用前 $N-1$ 个连续的词来预测下一个词出现的概率.Hindle 等人首先采用 *N*-gram 为程序语言建模[32],并用于代码补全任务.Nguyen 等人对 *N*-gram 模型进行了扩展[33],在构建模型时考虑了代码中的语义信息.Tu 等人观察到代码具有局部性特征[34],因此在 *N*-gram 模型的基础上增加了缓存机制来记录代码的局部特性.但这些都属于语句一级的理解,仅能用于进行特定的编码和程序分析.

一些经典机器学习算法,如决策树、条件随机场等,被用于进行程序理解.其中,条件随机场(conditional random field,简称 CRF)是一类判别式无向概率图模型[35],用于编码观察间的已知关系并构建与其一致的解释.在程序理解中,该模型可以对多个变量在给定观测值后的条件概率进行建模,构建代码结构以及代码中元素间的依赖关系.比如,文献[36]利用条件随机场进行程序元素的属性预测,包括预测变量名或变量类型.它首先将程序表示为一个依赖网络(dependency network),以表达程序中各元素间的关系;然后构建条件随机场模型,从而对程序中元素属性进行预测.其优点是在预测过程中能够充分考虑程序中各元素间的依赖关系和程序的结构,可提高预测的准确率.决策树是基于树结构进行决策的方法,文献[37]将学习代码概率模型的问题看作是基于抽象语法树学习领域特定语言上的决策树的问题,从而支持在动态计算上下文中调节程序元素的预测.这些工作支持了程序中跨语句依赖关系的理解.

近年来,人们开始探索深度学习在程序理解上的应用.以往的程序理解研究需要对程序数据进行分析,获取程序中包含的特征信息,然后根据这些先验知识人为地制定启发式规则,这一过程非常耗时耗力.深度学习是一种数据驱动的端到端的方法,将深度学习用于程序理解,目的是希望根据已有程序及其相关数据,自动地学习程序数据中蕴含的特征,取代繁琐的人工特征提取过程.目前,基于深度学习的程序分析大多是静态的,即直接对程序源代码或者其他抽象表示(如抽象语法树、数据流图等)进行学习.根据不同表示方式,可以分为如下几个方面.

- 基于词素(token)序列的理解:基于词素序列的理解将程序表示为词素序列,用深度神经网络对词素序列进行建模,学习词素序列中包含的信息.这样,当给定部分词素序列时,模型可以预测下一个最有可能出现的词素.这样的模型可用于代码补全任务中[38,39].这个学习过程也可以获得整个程序词素序列的特征向量表示,因此也可以完成程序注释生成[40]、代码检索[41]等任务;
- 基于应用程序接口(API)序列的理解:基于 API 序列的程序理解认为程序中用到的 API 序列可以在一定程度上反映程序的功能特性,它将程序中包含的 API 序列作为对象进行建模,或者将 API 序列作为额外信息输入学习网络.这类模型可用于如代码注释生成[42]、代码检索[41]等任务;
- 基于抽象语法树(AST)的理解:抽象语法树表示了程序的语法结构.利用深度神经网络对程序语法树进行建模,可学习到特定程序代码数据集的语法和结构信息.这类模型可用于包括代码补全[43]、程序自动生成[44,45]、代码克隆检测[46]等任务中;
- 基于图的理解:程序还可以用基于图的结构来表示,例如控制流图和数据依赖图.把程序的图表示作为学习模型的输入,使得模型能更好地捕捉程序中变量间或活动间的依赖关系.这类模型可用于包括程序验证[47]、程序变量名预测、误用检测[48]等任务中;
- 基于程序执行动态轨迹的理解:对程序的动态执行过程中产生的数据进行程序建模,可以捕获程序运

行时的性质,比如程序执行轨迹(execution trace).这些执行过程数据可以包含程序执行时产生的中间结果,如执行过程中调用的子程序及其参数,或程序各变量值的变化.这类研究工作目前主要集中在程序综合上[49−51].此外,文献[52]还提出了基于程序执行轨迹构建程序表示模型,获得程序的特征向量表示,并将此特征向量表示用于对程序错误进行分类.

2.3 程序理解与软件工程任务

程序理解与软件工程中的很多任务相关,本小节选择几个典型的任务,讨论程序理解与它们之间的关系以及程序理解对解决这些问题的支撑作用.

2.3.1 程序理解与软件维护

软件维护是指根据需求变化或硬件环境变化对应用程序的部分或全部进行修改.实践表明,50%~80%的软件预算消耗在对现有系统的维护上[53].软件维护包括如下几个活动:与修改请求相关的软件理解、修改影响分析和修改实施(包括重构以及修改传播分析)以及修改后系统调试与测试[54,55].其中,程序理解主要是分析并确定已有系统成分间的关联关系、分析维护需求与目标代码间的映射关系等,还可以辅助维护人员理解已有系统代码.比如,Snelting 等人采用程序理解技术进行类与接口层次的识别,以辅助维护人员理解软件系统的结构[56].一些程序理解工具,如程序切分器、静态分析器、动态分析器[57]等,可提示程序员选择最值得关注的程序片段,显示数据链和相关特征,使程序员能够准确跟踪更改影响.静态分析器还能帮助程序员快速提取模块、过程、变量、数据元素、对象与类、类层次结构等信息.这些工具可以用清晰可读、可理解的方式组织源代码,从而把程序员从烦躁的代码阅读中解放出来.

程序理解技术不仅能够提取软件元素间的依赖关系,还能支持软件演化性分析.比如,文献[58]从代码演化过程中,提取开发过程和演化过程中的模式,从而提高源程序的可理解性;文献[59]利用基于概念格的程序理解技术,提取出协同修改模式,为软件演化所需的维护活动提供有效的度量.

2.3.2 程序理解与软件测试

软件测试是指在特定条件下对程序进行操作,以发现程序错误,衡量软件质量,并评估其是否满足设计要求,目的在于检验程序是否满足需求,或弄清预期结果与实际结果间的距离.软件测试既耗时又耗力,测试成本通常超过软件首次发布后总成本的 50%[60].

无论采用什么测试方法,测试员都需要了解测试对象的功能.软件设计和实现文档提供理解软件的基本功能说明,但没有提供代码级别的测试信息,测试员只有阅读源代码才能知道函数级或语句级的功能,因此能全面反映程序功能的程序理解技术,可以极大地提升软件测试的效率和效果.

程序理解技术目前已经用于软件智能调试、软件自动测试和软件安全漏洞发掘等任务.如,IEEE 软件测试标准要求对能独立测试的软件成分进行辨识[61],Cellier 等人提出融合关联规则与基于形式概念分析的程序理解技术,帮助进行错误定位[62].通过概念格,很容易获得错误的测试和正确的测试间的特征差异,从而实现错误定位.Ammons 等人将类似的方法应用于调试时态规约[63],大大提高了效率,只需检查原有轨迹数量的 1/3,就可分类检查出错误与正确的规约.

符号执行和抽象解释等新方法,可以从外部和内部自动识别程序的运行时特征.与此相关的其他技术,如基于序列[64]和统计的软件验证技术、模型验证技术、可携带证明代码等,都成为当前研究的热点.

2.3.3 程序理解与软件重构

软件重构是指:为了改善软件的结构,提高软件的清晰性、可扩展性和可重用性,在不改变软件功能和外部可见性的情况下,对其进行的改造,使程序的设计模式和架构更趋合理,从而提高软件的可扩展性和可维护性[65].主要研究包括类层次结构重构、模块结构、“坏味道”检测和代码重构,由低层次语言规范向高层次的语言规范进行重构等.

程序理解在不同层次的软件重构中都能发挥作用.如,Arévalo 等人采用形式概念分析,将对象类结构区分为好的层次设计、坏的层次设计和不正常的层次设计,指导维护人员理解类层次以及修改类层次中存在的“坏味道”[66].Snelting 等人提出了基于概念格进行类层次重构的方法[67],经过重构后的类层次更加合理.Bhatti 等人

对过程式面向对象代码进行分析,识别程序代码中有用的类层次和结构,并进行重构[68].Al-Ekram 等人将概念分类、程序切片及形式概念分析结合起来进行重构[69],使重构后的模块能够自包含、模块间有很小的冲突以及较少的代码复制.Kim 等人提出采用面向对象的概念分析方法进行模块重构[70],该方法除保持了传统形式概念分析方法的优点外,还可直接粗粒度地识别模块,适用于大规模软件的模块重构.

重构过程中,对软件进行的修改会对软件的其他部分造成潜在影响,可能导致软件不一致.软件修改影响分析可用于识别这些潜在影响[71].如,Tonella 将概念分析与分解切片结合起来,提出一种基于分解切片概念格的程序理解方法[72],用于过程内针对某语句或者变量修改的影响分析.

2.3.4 程序理解与软件复用

软件复用是指重复使用已有软件的各种制品,包括各类文档和代码,来开发新的软件,减少软件开发和维护的开销.程序理解技术支持软件制品的理解,是软件复用的关键技术.

对可复用软件制品的理解,特别是代码功能的理解,是进行复用的基础.如:Tonella 等人通过程序理解从源代码中提取设计模式[73],进而指导程序的演化;Hill 等人[74]通过分析与代码相关的文档、错误报告等,猜测并标注代码功能;Vinz 等人[75]分析代码中的自然语言注释,进行软件代码的功能分析;Falleri 等人[76]分析程序代码中的变量名、方法名、类名等类自然语言标识符,推测程序代码的功能;Nguyen 等人[77]通过建立程序语言的概率模型(如词袋模型),进而根据关键字或表达式的出现概率对代码功能进行推测;Shepherd 等人发现程序代码的动作可以用动词和名词的组合来表示,因此提出一组规则和算法,根据程序代码方法中的自定义标识符生成简短的自然语言描述,作为对代码功能的表达[78].

更细粒度的理解包括 Robillard 等人在特征定位方面的工作[79,80],他们提出基于概念图的方法,利用概念图表示代码不同成分间的关系,从而通过代码中的成分进行特征定位.

2.3.5 程序理解与需求追踪

需求跟踪是指跟踪需求生命周期全过程,包括建立每个需求和系统元素之间的关联,比如和其他需求、体系结构、系统构件、源代码模块、测试案例之间的关联等.程序理解在需求追踪中的作用体现在特征定位方面,即,将自然语言形式的修改请求映射到代码层的修改上,识别哪些代码实现需求中的哪些功能(或者非功能)特性[81].如,Zhao 等人提出了一种基于分支调用图(branch reserving call graph)的程序理解方法[82],采用信息检索技术建立关键词与代码功能单元之间的关系,利用分支调用图表示不同功能单元间的关系;Chen 等人基于特征定位技术,提出一种基于抽象系统依赖图(ASDG)的代码表示方法[83],将函数或全局变量作为点,它们之间的控制依赖关系作为边,利用 ASDG 识别代码的功能;Poshyvanyk 等人引入形式概念分析,对代码中不同对象的属性进行聚类分析,根据同类对象的共同属性,建立层次概念模型,找出与指定特征最相关的类或方法,方便维护人员进行特征定位[84].

程序理解技术在非代码层次的需求分析方面的应用包括:Ivkovic 等人利用概念格聚集需求层次的各个模型元素,通过聚集得到的模型元素集合被认为存在依赖关系[85];Niu 等人通过形式概念分析来识别软件演化产品线的模块化和交互性需求,进而检查其功能和质量需求[86].

2.3.6 程序理解与逆向工程

逆向工程是指分析目标系统,确定系统的成分及其相互关系,用高层抽象或其他形式来表达目标系统[87].Muller 等人将逆向工程,具体地说是信息系统逆向工程,细化为数据抽取、数据分析和概念抽象这 3 个阶段[88]:数据抽取利用适当的抽取机制从目标系统的执行过程中收集原始数据;数据分析从原始数据中发现结构完整语义一致的逻辑数据模型;概念抽象则将数据分析中获得的逻辑数据模型映射为等价的概念模型.

程序理解在逆向工程的数据抽取和数据分析活动中都能发挥作用.例如:Harandi 等人提出一种将代码中与数据处理相关的部分组织为有层次结构的事件模型[89],其中,底层为数据定义、声明等,中间层为栈、队列、树等数据结构,高层为算法,然后将该事件模型与预定义的事件库相比对,找出可能与代码功能相对应的概念;Viljamaa 等人基于程序理解技术从软件框架中提取可重用性接口的规约和代码[90];Marcus 等人提出一种基于潜在语义索引(LSI)的方法[91],对代码中的自定义标识符进行拆解,用拆解后的字段集合作为对代码功能的表

示,在此基础上建立关于代码文本描述的索引,以帮助在特征定位任务中确定代码段的功能;Carey 等人采用支持向量机对人工选择的代码特征进行分类[92],将代码划分至不同的概念类别,建立代码与表达功能的概念类别之间的关联;His 等人提出一种基于图匹配技术的代码功能表示和识别方法[92],该方法首先建立代码接口调用关系图,然后通过接口调用图的匹配定位实现指定功能的代码段.

3 基于程序理解的延伸任务

近年来,深度学习方法的引入,使得程序理解逐步趋向于自动化,也延伸了程序理解在软件开发过程中的作用,基于理解的程序或软件制品的自动处理成为可能.本节介绍这个方面的一些最新进展.

3.1 代码补全

代码补全是 IDE,例如 Eclipse、IntelliJ、Visual Studio 等中使用频率几乎最高的功能[93],其本质是基于关于代码的普适知识,预测当前上下文中缺失的代码.目前,大多数工作通过构建语言模型来学习代码 token 的概率分布,根据已有部分代码(context)来预测接下来最可能出现的 token.前面曾经提到,Hindle 等人[32]使用了 *n*-gram 的语言模型,在 token 级别完成代码补全.随着深度学习的发展,深度神经网络模型,例如 RNN、CNN 等,都被应用于代码补全任务中,通过构建深度神经网络模型,学习已有代码的特征,基于这些特征完成代码补全任务,包括 token 级别、AST 级别以及各类图级别的补全.

3.2 代码注释生成

自动生成给定代码片段的自然语言描述,是程序理解的一个重要应用场景.早期关于代码注释生成的研究,一般用启发式规则提取代码关键信息,再合成自然语言描述[94,95].深度学习的机制,比如基于 seq2seq 框架的模型,可以大大提高代码注释生成的能力.比如,Iyer 等人[40]采用了 seq2seq 模型,并引入了注意力机制,完成根据代码片段生成自然语言描述的任务.Hu 等人[42]在基于 seq2seq 的程序注释模型的基础上,进一步集成了代码所调用的 API 序列,将 API 调用序列蕴含的知识输入到网络模型中,以辅助注释的生成,使模型生成的注释能够更好地描述代码的功能.

3.3 代码模式检测、克隆检测和缺陷检测

代码模式检测[96]常用于进行错误检测或代码克隆检测.Mou 等人[96]首次提出了基于树结构的卷积神经网络,对程序 AST 进行建模,学习程序代码的结构信息,用于程序分类和代码模式检测任务.White 等人[97]和 Wei 等人[46]通过深度神经网络学习代码中隐含的特征,根据得到的特征向量表示来判断两个代码是否属于一对克隆对,改进了克隆检测的效果.

利用语言模型对代码建模,就是对代码段赋予其概率值.频繁出现的代码段获得较高的概率值,而出现概率很低的代码段则被认为不太可能出现,很可能是错误代码.一些研究基于这个假设构建语言模型,进行代码缺陷检测.Wang 等人[98]使用深度信念网络学习 token 级别的代码特征,用于预测代码中的缺陷.Murali 等人[99]结合了主题模型和循环神经网络,对代码中调用的 API 序列进行建模,检测代码中是否包含不常用的 API 序列,以判断代码中是否包含错误.

3.4 代码风格修正

除了程序语言自身的语法约束之外,代码风格也是约束程序的一种方式.例如命名风格等,风格规范的代码有利于代码的理解[100].对代码风格的研究主要包括变量名、函数名和类名的预测(重命名)以及代码格式修正等.这些工作一般利用深度神经网络得到代码的特征向量表示,然后根据这些向量表示完成风格修正任务.Allamanis 等人[101]针对程序中的关键语句以及层次结构性等特性,构建基于注意力机制的卷积神经网络,学习程序的这一特性,并用于程序函数名的预测.Allamanis 等人[48]采用图结构来表示程序,对程序中的数据依赖关系进行建模,用于程序变量命名和变量误用错误检测中.

4 趋势与展望

软件系统已经成为信息社会的基础设施,它面临其软硬件环境及外部资源不断变化的挑战.增强软件系统的自适应和持续演化能力,使其能够长期生存并不断成长,是当前学术界研究的新热点.前面的技术分析显示,软件理解技术将在软件系统自适应和持续演化中扮演重要的角色,是支撑软件系统能够长期生存和不断成长的不可或缺的基础技术.

同时,在开放网络环境下,“万物互联”的网络空间推动了人机物的深度融合,促进应用场景的普适化,软件系统在规模化、复杂度和异构性上不断提升,这给软件系统理解带来了新的需求和研究挑战,有必要重新认识现有的程序理解技术,提炼未来软件系统理解的关键需求,从而确定进一步的研究思路和可以突破的方向.我们同样从技术的角度、工程的角度和认知的角度,分析和讨论程序理解的技术需求,目的是希望能把握未来研究选题的方向.

- 在理解的对象上:自然语言理解技术能够理解的元素粒度从词到句子再到段落再到篇章.相应地,以从元素到合成物、从微观向宏观、从具体关联到抽象关联为维度建立谱系,程序理解应该建立在软件系统完整生命周期上的全谱系理解,包括从词素和变量及其属性值等的建模,到程序代码语句结构理解,再到程序调用关系建模和数据依赖关系建模,需要具备从程序片段的功能理解到程序系统的能力理解.总之,程序理解仅仅局限在源码级的理解是远远不够的,我们需要能够理解:(1) 软件系统构件间的关系(架构具有什么含义?);(2) 特定软件系统中对应现实世界问题表达的数据类型结构(如何表达问题空间?);(3) 特定软件系统问题求解过程调用关系的层次(如何求解现实问题?).更具挑战性的是:在一定抽象层次上建立对大型软件系统及其软件体系结构的某个方面(如良构性、鲁棒性、健康性等)的系统级量度及其度量方法,支持大型程序的系统级概述的获得;
- 在理解的技术上:动态分析技术将凸显其重要性.动态分析根据收集到的软件系统运行时数据进行软件系统的分析,它可以揭示软件系统的真实行为,有可能建立软件系统的准确的表示.这个表示可以是语句层的调用关系、信息处理的变化过程,甚至体系结构的视图.随着软件系统规模的扩大、复杂度的增长以及软件系统运行上下文或运行场景不确定性的提高,系统的动态理解技术将越来越重要.同时,在人机物互联应用环境下,动态理解的范畴有可能不仅需要软件系统本身的状态感知,还需要结合互联环境中的上下文感知和运行场景感知.这种情况可以称其为程序理解扩大到系统理解;
- 在理解的层次上:从表达知识的角度看,程序理解具有与自然语言理解相似的特性,依据自然语言理解的层次,程序理解也可以遵循从语法分析到语义分析再到语用分析这样一个由浅入深的过程.目前的技术大部分侧重在程序语言语法分析上,部分技术涉及从信息处理视角出发的程序语义分析.但是从软件系统逆向工程、软件复用以及程序自适应演化等的需求来看,程序或软件系统的语用分析的重要性将逐步凸显,这涉及对软件系统的运行时上下文、应用场景或软件系统作用环境的分析,从而决定软件系统在当前现实应用场景中的适配性,决定在特定现实应用场景下,软件制品是否适合复用,或在应用场景变化的情况下,软件系统是否适合继续使用(即是否需要调整和演化)等;
- 在理解的作用域上:从前面的分析中可以看到,程序理解的方法和理解结果可用于软件工程的多个方面.目前有很多程序理解的工作,其目标一方面是为了提高程序员理解程序的效率,提取一些具体的程序特征,辅助程序员去理解程序;另一方面是提取出程序中表达的知识,支持逆向工程和知识复用.未来软件系统越来越需要具有在线适应、自主演化和自我成长的能力,这要求对系统的理解过程要能完全自动化,支持软件系统的自我认知或自我动态认知能力.关于这方面的一个值得关注的问题与运行时模型相关,软件系统的自我认知可以看作是它能掌握运行时模型,并确保系统运行状态和运行时模型的双向同步.

5 结束语

程序理解是软件工程中的一个重要活动,在完成软件重用、维护、迁移、逆向工程以及软件系统扩展等任务时,都要依赖于对程序的理解.在软件工程的整个发展过程中,程序理解一直都占据着重要地位.随着社会信息化进程的不断加快,程序理解研究的内涵与需求等也发生了新的变化.因此,本文对程序理解进行了重新审视.

本文首先给出了程序理解的定义,分别从学习和认知角度、工程角度以及方法的角度对程序理解进行了解释.然后,通过文献分析展示了长期以来软件工程领域中程序理解的研究布局.分析结果表明:程序理解一直都是软件工程中的研究热点,并且随着软件工程的发展,程序理解的研究侧重点也在随之发生变化.然后,分别从认知的角度、软件工程的视角以及理解技术这 3 个方面综合论述程序理解研究的发展脉络和研究进展.最后,对程序理解的发展趋势进行了探讨.

作为软件工程领域研究的核心内容之一,程序理解技术正在随着互联网各项技术的发展处于不断推进中,具有广阔的研究前景和研究价值.随着近几年来人工智能和深度学习技术的发展,越来越多的学者开始研究如何运用深度学习的方法实现程序分析的智能化与自动化,程序理解的自动化值得期待.

References:

[1] Storey MA. Theories, methods and tools in program comprehension: Past, present and future. In: Proc. of the 13th Int'l Workshop on Program Comprehension (IWPC). 2005. 181−191. [doi: 10.1109/wpc.2005.38]

[2] Rugaber S. Program comprehension for reverse engineering. In: Proc. of the AAAI Workshop on AI and Automated Program Understanding. 1992. 106−110.

[3] Nielson F, Nielson HR, Hankin C. Principles of Program Analysis. Springer-Verlag, 2015. 1−3. [doi: 10.1007/978-3-662-03811-6]

[4] Kirkov R, Agre G. Source code analysis—An overview. Cybernetics and Information Technologies, 2010,10(2):60−77.

[5] Ball T. The concept of dynamic analysis. ACM SIGSOFT Software Engineering Notes, 1999,24(6):216−234.

[6] Deimel L, Naveda J. Reading computer programs: Instructor's guide and exercises educational materials CMU. In: Proc. of the SEI-90-EM, Vol.3. 1990. 9−11. [doi: 10.21236/ada228026]

[7] Müller HA, Tilley SR, Wong K. Understanding software systems using reverse engineering technology perspectives from the RIGI project. In: Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research: Software Engineering, Vol.1. IBM Press, 1993. 217−226. [doi: 10.1142/9789812831163_0016]

[8] O'brien MP. Software comprehension—A review & research direction. Technical Report, Ireland: Department of Computer Science & Information Systems University of Limerick, 2003. 1−2.

[9] Cornelissen B, Zaidman A, Van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. on Software Engineering, 2009,35(5):684−702. [doi: 10.1109/tse.2009.28]

[10] Maalej W, Tiarks R, Roehm T, Koschke R. On the comprehension of program comprehension. ACM Trans. on Software Engineering and Methodology (TOSEM), 2014,23(4):31.

[11] Armaly A, Rodeghero P, McMillan C. A comparison of program comprehension strategies by blind and sighted programmers. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). 2017. 788. [doi: 10.1145/3180155.3182544]

[12] Hughes J, Sparks C, Stoughton A, Parikh R, Reuther A, Jagannathan S. Building resource adaptive software systems (BRASS): Objectives and system evaluation. ACM SIGSOFT Software Engineering Notes, 2016,41(1):1−2. [doi: 10.1145/2853073.2853081]

[13] Sifakis J. System design in the era of IoT—Meeting the autonomy challenge. arXiv Preprint arXiv: 1806.09846, 2018. [doi: 10.4204/eptcs.272.1]

[14] Shneiderman B, Mayer R. Syntactic/Semantic interactions in programmer behavior: A model and experimental results. Int'l Journal of Computer & Information Sciences, 1979,8(3):219−238. [doi: 10.1007/bf00977789]

[15] Pennington N. Stimulus structures and mental representations in expert. Comprehension of Computer Programs, 1987,19(3): 295−341.

[16] Von Mayrhauser A, Vans AM. From program comprehension to tool requirements for an industrial environment. In: Proc. of the 2nd Int'l Workshop on Program Comprehension (IWPC). 1993. 78−86. [doi: 10.1109/wpc.1993.263903]

[17] Shaft TM, Vessey I. The relevance of application domain knowledge: The case of computer program comprehension. Information Systems Research, 1995,6(3):286−299. [doi: 10.1287/isre.6.3.286]

[18] Shneiderman B. Exploratory experiments in programmer behavior. Int'l Journal of Computer & Information Sciences, 1976,5(2): 123−143. [doi: 10.1007/bf00975629]

[19] Soloway E, Ehrlich K. Empirical studies of programming knowledge. IEEE Trans. on Software Engineering, 1984,10(5):595−609. [doi: 10.1109/tse.1984.5010283]

[20] Siegmund J. Program comprehension: Past, present, and future. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). 2016. 13−20.

[21] Siegmund J, Kästner C, Apel S, Parnin C, Bethmann A, Leich T, Saake G, Brechmann A. Understanding understanding source code with functional magnetic resonance imaging. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). 2014. 378−389. [doi: 10.1145/2568225.2568252]

[22] Nakagawa T, Kamei Y, Uwano H, Monden A, Matsumoto K, German DM. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In: Companion Proc. of the 36th Int'l Conf. on Software Engineering. 2014. 448−451. [doi: 10.1145/2591062.2591098]

[23] Fritz T, Begel A, Müller SC, Yigit-Elliott S, Züger M. Using psycho-physiological measures to assess task difficulty in software development. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). 2014. 402−413. [doi: 10.1145/2568225.2568266]

[24] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697−1710 (in Chinese with English abstract).

[25] Fagan M. Design and code inspections to reduce errors in program development. In: Proc. of the Software Pioneers. Berlin, Heidelberg: Springer-Verlag, 2002. 575−607. [doi: 10.1007/978-3-642-59412-0_35]

[26] Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Proc. of the 1998 Int'l Conf. on Software Maintenance (ICSM). 1998. 368−377. [doi: 10.1109/icsm.1998.738528]

[27] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394. [doi: 10.1145/360248.360252]

[28] Mock M. Dynamic analysis from the bottom up. In: Proc. of the WODA 2003 ICSE Workshop on Dynamic Analysis. 2003. 13.

[29] Larus JR, Ball T. Rewriting executable files to measure program behavior. Software: Practice and Experience, 1994,24(2):197−218. [doi: 10.1002/spe.4380240204]

[30] Gosain A, Sharma G. A survey of dynamic program analysis techniques and tools. In: Proc. of the 3rd Int'l Conf. on Frontiers of Intelligent Computing: Theory and Applications (FICTA). 2015. 113−122.

[31] Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. arXiv Preprint arXiv: 1709.06182, 2017. [doi: 10.1145/3212695]

[32] Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). 2012. 837−847. [doi: 10.1109/icse.2012.6227135]

[33] Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (FSE). 2013. 532−542. [doi: 10.1145/2491411.2491458]

[34] Tu Z, Su Z, Devanbu P. On the localness of software. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). 2014. 269−280. [doi: 10.1145/2635868.2635875]

[35] Lafferty J, McCallum A, Pereira FC. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proc. of the 8th Int'l Conf. on Machine Learning (ICML). 2001. 282−289.

[36] Raychev V, Vechev M, Krause A. Predicting program properties from big code. ACM SIGPLAN Notices, 2015,50(1):111−124. [doi: 10.1145/2775051.2677009]

[37] Raychev V, Bielik P, Vechev M. Probabilistic model for code with decision trees. ACM SIGPLAN Notices, 2016,51(10):731−747. [doi: 10.1145/3022671.2984041]

[38] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. ACM SIGPLAN Notices, 2014,49(6):419−428. [doi: 10.1145/2666356.2594321]

[39] Bhoopchand A, Rocktäschel T, Barr E, Riedel S. Learning python code suggestion with a sparse pointer network. arXiv Preprint arXiv: 1611.08307, 2016.

[40] Iyer S, Konstas I, Cheung A, Zettlemoyer L. Summarizing source code using a neural attention model. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol.1: Long Papers). 2016. 2073−2083. [doi: 10.18653/v1/p16-1195]

[41] Gu X, Zhang H, Kim S. Deep code search. In: Proc. of the 40th Int'l Conf. on Software Engineering (ICSE). 2018. 933−944. [doi: 10.1145/3180155.3180167]

[42] Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing source code with transferred API knowledge. In: Proc. of the 27th Int'l Joint Conf. on Artificial Intelligence (IJCAI). 2018. 2269−2275. [doi: 10.24963/ijcai.2018/314]

[43] Li J, Wang Y, King I, Lyu MR. Code completion with neural attention and pointer networks. arXiv Preprint arXiv: 1711.09573, 2017. [doi: 10.24963/ijcai.2018/578]

[44] Yin P, Neubig G. A syntactic neural model for general-purpose code generation. arXiv Preprint arXiv: 1704.01696, 2017. [doi: 10.18653/v1/p17-1041]

[45] Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. arXiv Preprint arXiv: 1704.07535, 2017. [doi: 10.18653/v1/p17-1105]

[46] Wei HH, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence (IJCAI). 2017. 3034−3040. [doi: 10.24963/ijcai.2017/423]

[47] Li Y, Tarlow D, Brockschmidt M, Zemel R. Gated graph sequence neural networks. arXiv Preprint arXiv: 1511.05493, 2015.

[48] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. arXiv Preprint arXiv: 1711.00740, 2017.

[49] Reed S, De Freitas N. Neural programmer-interpreters. arXiv preprint arXiv: 1511.06279, 2017.

[50] Cai J, Shin R, Song D. Making neural programming architectures generalize via recursion. arXiv Preprint arXiv: 1704.06611, 2017.

[51] Balog M, Gaunt AL, Brockschmidt M, Nowozin S, Tarlow D. Deepcoder: Learning to write programs. arXiv Preprint arXiv: 1611.01989, 2016.

[52] Wang K, Singh R, Su Z. Dynamic neural program embedding for program repair. arXiv Preprint arXiv: 1711.07163, 2017.

[53] Boehm BW. Software Engineering Economics. Englewood Cliffs (NJ): Prentice-Hall, 1981.

[54] Poshyvanyk D, Gethers M, Marcus A. Concept location using formal concept analysis and information retrieval. ACM Trans. on Software Engineering and Methodology (TOSEM), 2012,21(4):23. [doi: 10.1109/icpc.2007.13]

[55] Koschke R, Quante J. On dynamic feature location. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, 2005. 86−95. [doi: 10.1145/1101908.1101923]

[56] Snelting G, Tip F. Understanding class hierarchies using concept analysis. ACM Trans. on Programming Languages and Systems (TOPLAS), 2000,22(3):540−582. [doi: 10.1145/353926.353940]

[57] Grubb P, Takang AA. Software maintenance: Concepts and practice. In: Proc. of the World Scientific. 2003. [doi: 10.1142/9789812564429]

[58] Glorie M, Zaidman A, Van Deursen A, Hofland L. Splitting a large software repository for easing future software evolution—An industrial experience report. Journal of Software Maintenance and Evolution: Research and Practice, 2009,21(2):113−141. [doi: 10.1109/csmr.2008.4493310]

[59] Gîrba T, Ducasse S, Kuhn A, Marinescu R, Daniel R. Using concept analysis to detect co-change patterns. In: Proc. of the 9th Int'l Workshop on Principles of Software Evolution. 2007. 83−89. [doi: 10.1145/1294948.1294970]

[60] Runeson P, Andersson C, Höst M. Test processes in software product evolution—A qualitative survey on the state of practice. Journal of Software Maintenance and Evolution: Research and Practice, 2003,15(1):41−59. [doi: 10.1002/smr.265]

[61] Freedman RS. Testability of software components. IEEE Trans. on Software Engineering, 1991,17(6):553−564. [doi: 10.1109/32.87281]

[62] Cellier P, Ducassé M, Ferré S, Ridoux O. Formal concept analysis enhances fault localization in software. In: Proc. of the Int'l Conf. on Formal Concept Analysis. 2008. 273−288. [doi: 10.1007/978-3-540-78137-0_20]

[63] Ammons G, Mandelin D, Bodík R, Larus JR. Debugging temporal specifications with concept analysis. ACM SIGPLAN Notices, 2003,38(5):182−195. [doi: 10.1145/780822.781152]

[64] Prowell SJ, Poore JH. Foundations of sequence-based software specification. IEEE Trans. on Software Engineering, 2003,29(5):417−429. [doi: 10.1109/tse.2003.1199071]

[65] Mens T. A survey of software refactoring. IEEE Trans. on Software Engineering, 2004,30(2):126−139. [doi: 10.1109/tse.2004.1265817]

[66] Arévalo G, Ducasse S, Gordillo S, Nierstrasz O. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. Information and Software Technology, 2010,52(11):1167−1187. [doi: 10.1016/j.infsof.2010.05.010]

[67] Snelting G, Tip F. Reengineering class hierarchies using concept analysis. ACM SIGSOFT Software Engineering Notes, 1998,23(6):99−110. [doi: 10.1145/291252.288273]

[68] Bhatti MU, Ducasse S, Huchard M. Reconsidering classes in procedural object-oriented code. In: Proc. of the 15th Working Conf. on Reverse Engineering (WCRE). 2008. 257−266. [doi: 10.1109/wcre.2008.58]

[69] Al-Ekram R, Kontogiannis K. Source code modularization using lattice of concept slices. In: Proc. of the 8th European Conf. on Software Maintenance and Reengineering (CSMR). 2004. 195−203. [doi: 10.1109/csmr.2004.1281420]

[70] Kim HH, Bae DH. Object-Oriented concept analysis for software modularisation. IET Software, 2008,2(2):134−148. [doi: 10.1049/iet-sen:20060069]

[71] Li B, Sun X, Leung H, Zhang S. A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability, 2013,23(8):613−646. [doi: 10.1002/stvr.1475]

[72] Tonella P. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. on Software Engineering, 2003,29(6):495−509. [doi: 10.1109/tse.2003.1205178]

[73] Tonella P, Antoniol G. Inference of object-oriented design patterns. Journal of Software Maintenance and Evolution: Research and Practice, 2001,13(5):309−330.

[74] Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of NL-queries for software maintenance and reuse. In: Proc. of the 31st Int'l Conf. on Software Engineering (ICSE). 2009. 232−242. [doi: 10.1109/icse.2009.5070524]

[75] Vinz BL, Etzkorn LH. Improving program comprehension by combining code understanding with comment understanding. Knowledge-Based Systems, 2008,21(8):813−825. [doi: 10.1016/j.knosys.2008.03.033]

[76] Falleri JR, Huchard M, Lafourcade M, Nebut C, Prince V, Dao M. Automatic extraction of a wordnet-like identifier network from software. In: Proc. of the 18th Int'l Conf. on Program Comprehension (ICPC). 2010. 4−13. [doi: 10.1109/icpc.2010.12]

[77] Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (FSE). 2013. 532−542. [doi: 10.1145/2491411.2491458]

[78] Fry ZP, Shepherd D, Hill E, Pollock L, Vijay-Shanker K. Analysing source code: Looking for useful verb—Direct object pairs in all the right places. IET Software, 2008,2(1):27−36. [doi: 10.1049/iet-sen:20070112]

[79] Robillard MP, Murphy GC. Representing concerns in source code. ACM Trans. on Software Engineering and Methodology (TOSEM), 2007,16(1):3. [doi: 10.1145/1189748.1189751]

[80] Robillard MP, Shepherd D, Hill E, Vijay-Shanker K, Pollock L. An empirical study of the concept assignment problem. Technical Report, SOCS-TR-2007.3, School of Computer Science, McGill University, 2007. 1−4.

[81] Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: A taxonomy and survey. Journal of Software: Evolution and Process, 2013,25(1):53−95. [doi: 10.1002/smr.567]

[82] Zhao W, Zhang L, Liu Y, Sun J, Yang F. SNIAFL: Towards a static noninteractive approach to feature location. ACM Trans. on Software Engineering and Methodology (TOSEM), 2006,15(2):195−226. [doi: 10.1145/1131421.1131424]

[83] Chen K, Rajlich V. Case study of feature location using dependence graph. In: Proc. of the 8th Int'l Workshop on Program Comprehension (IWPC). 2000. 241−247. [doi: 10.1109/icpc.2010.40]

[84] Poshyvanyk D, Gueheneuc YG, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. on Software Engineering, 2007,33(6):420−432. [doi: 10.1109/tse.2007.1016]

[85] Ivkovic I, Kontogiannis K. Towards automatic establishment of model dependencies using formal concept analysis. Int'l Journal of Software Engineering and Knowledge Engineering, 2006,16(4):499−522. [doi: 10.1142/s0218194006002902]

[86] Niu N, Easterbrook S. Concept analysis for product line requirements. In: Proc. of the 8th ACM Int'l Conf. on Aspect-Oriented Software Development. 2009. 137−148. [doi: 10.1145/1509239.1509259]

[87] Pinzger M, Gall H. Pattern-Supported architecture recovery. In: Proc. of the 10th Int'l Workshop on Program Comprehension (IWPC). 2002. 53−61. [doi: 10.1109/wpc.2002.1021318]

[88] Müller HA, Jahnke JH, Smith DB, Storey MA, Tilley SR, Wong K. Reverse engineering: A roadmap. In: Proc. of the Conf. on the Future of Software Engineering. 2000. 47−60. [doi: 10.1145/336512.336526]

[89] Harandi MT, Ning JQ. Knowledge-Based program analysis. IEEE Software, 1990,7(1):74−81. [doi: 10.1109/icsm.1988.10182]

[90] Viljamaa J. Reverse engineering framework reuse interfaces. ACM SIGSOFT Software Engineering Notes, 2003,28(5):217−226. [doi: 10.1145/949952.940101]

[91] Marcus A, Sergeyev A, Rajlich V, Maletic JI. An information retrieval approach to concept location in source code. In: Proc. of the 11th Working Conf. on Reverse Engineering. 2004. 214−223. [doi: 10.1109/wcre.2004.10]

[92] Carey MM, Gannod GC. Recovering concepts from source code with automated concept identification. In: Proc. of the 15th IEEE Int'l Conf. on Program Comprehension (ICPC). 2007. 27−36. [doi: 10.1109/icpc.2007.31]

[93] Amann S, Proksch S, Nadi S, Mezini M. A study of visual studio usage in practice. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). 2016. 124−134. [doi: 10.1109/saner.2016.39]

[94] Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K. Automatic generation of natural language summaries for Java classes. In: Proc. of the 21st IEEE Int'l Conf. on Program Comprehension (ICPC). 2013. 23−32.

[95] McBurney PW, McMillan C. Automatic source code summarization of context for Java methods. IEEE Trans. on Software Engineering, 2016,42(2):103−119. [doi: 10.1109/tse.2015.2465386]

[96] Mou L, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language. In: Proc. of the 30th AAAI Conf. on Artificial Intelligence (AAAI). 2016. 1287−1293.

[97] White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). 2016. 87−98.

[98] Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE). 2016. 297−308. [doi: 10.1145/2884781.2884804]

[99] Murali V, Chaudhuri S, Jermaine C. Finding likely errors with bayesian specifications. arXiv Preprint arXiv: 1703.01370, 2017.

[100] Allamanis M, Barr ET, Bird C, Sutton C. Learning natural coding conventions. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). 2014. 281−293. [doi: 10.1145/2635868.2635883]

[101] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proc. of the Int'l Conf. on Machine Learning (ICML). 2016. 2091−2100.

附中文参考文献:

[24] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1710.

金芝(1962－),女,安徽休宁人,博士,教授,博士生导师,CCF 会士,主要研究领域为需求工程,知识工程,基于知识的软件工程.

刘芳(1994－),女,博士生,CCF 学生会员,主要研究领域为深度学习,程序分析.

李戈(1977－),男,博士,副教授,CCF 专业会员,主要研究领域为深度学习,程序分析,自然语言处理.