一种基于MAAT两步匹配的架构多层次变更检测方法

王 桐[1],李必信[2],王东东[2]
(1. 安徽工业大学计算机科学与技术学院,安徽马鞍山 243000; 2. 东南大学计算机科学与工程学院,江苏南京 210000)

摘 要: 掌握软件架构的变更对软件的持续演进具有十分重要的作用,然而目前的变更检测方法主要关注于细粒度的代码变更,忽略了对架构层级的检测. 为了检测架构层级的变更,本文提出一种基于MAAT(Multilevel Architecture Analysis Tree)两步匹配的架构多层次变更检测方法. 该方法包括三个步骤,分别是:构造MAAT;基于两个MAAT实施两步匹配算法检测变更;对变更进行分类和聚类. 基于以上算法,我们开发了工具ACAnalyzer. 实验结果证明,ACAnalyzer具有较好的准确性和性能.

关键词: 软件架构;变更检测;抽象语法树;软件演进;两步匹配
基金项目: 国家重点研发计划(No.2019YFE0105500); 国家自然科学基金(No.61872078); 安徽省自然科学基金(No.2108085QF263);安徽工业大学青年基金(No.QZ202013)
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2023)03-0694-07
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20210988

A Software Architecture Multiple-Level Change Detection Method Based on Two-Step MAAT Matching

WANG Tong[1], LI Bi-xin[2], WANG Dong-dong[2]
(1. *School of Computer Science and Technology, Anhui University of Technology, Maanshan, Anhui* 243000, *China*;
2. *School of Computer Science and Engineering, Southeast University, Nanjing, Jiangsu* 210000, *China*)

Abstract: Understanding the change of software architecture plays an important role in the continuous evolution of software. However, the current change detection methods mainly focus on fine-grained code and ignore the architecture level. In order to detect the change of architecture level, we propose a software architecture multiple-level change detection method based on two-step MAAT (Multilevel Architecture Analysis Tree) matching. The method includes three steps. Firstly, we construct an MAAT for each program. Secondly, a two-step matching algorithm is implemented to detect changes based on the two MAATs. Finally, we classify and cluster these changes. Based on the above algorithm, we develop the tool ACAnalyzer. And experimental results prove that ACAnalyzer has good accuracy and performance.

Key words: software architecture; change detection; abstract syntax tree; software evolution; two-step matching
Foundation Item(s): National Key Research and Development Program of China (No.2019YFE0105500); National Natural Science Foundation of China (No.61872078); Natural Science Foundation of Anhui Province of China (No.2108085QF263); Youth Foundation of AHUT (No.QZ202013).

1 引言

软件架构为开发者提供了一个较高抽象层次的视图,在软件演化过程中,检测和理解架构的变更对于软件的持续演进具有十分重要的意义[1]. 架构的变更信息便于开发人员快速地评估和理解软件演进情况,从而制定具有针对性的回归测试方案和演进方案,以较小的成本找到错误,并可依据变更信息判断当前版本是否与演进计划一致,最终达到提高软件质量、提升软件性能、节约开发成本等目的,因此获取软件架构的变更信息对软件的持续演进具有重要的意义[2,3].

变更检测是获取变更信息的有效方法,目前变更检测方法主要分为基于文本匹配的变更检测方法[4]和

收稿日期:2021-07-26;修回日期:2022-01-13;责任编辑:梅志强

基于抽象语法树(Abstract Syntax Tree, AST)匹配的变更检测方法.

基于文本匹配的变更检测方法将代码视为字符序列,然后基于文本行相似度计算两个版本之间的差异[5],该方法不考虑语法结构信息[6],因此这种方法不能有效的识别重命名操作和移动操作,此外,由于检测结果以文本行为单位,其检测结果的数据项数量非常庞大,不便于理解.

基于AST匹配的变更检测方法[7]是基于源代码的抽象语法树计算代码实体的相似度. 例如Fluri等[8]开发了一种变更检测工具ChangeDistiller,通过优化结点匹配算法提高了检测准确性,但AST仅可以展示两个代码片段或两个文件的语法结构,因此它不能用于检测架构层级的变更,并且AST中包含了大量的结点和结点间的关联信息,导致该类方法需要大量时间进行计算,因此该类方法的检测效率低于基于文本的变更检测方法.

另外,目前的变更检测方法主要关注于细粒度的变更,例如语句变更、方法变更等,没有解决软件架构层级变更检测的问题.

综上所述,目前软件架构的变更检测方法存在以下问题:

(1)基于文本的变更检测方法检测效率较高,但是该类方法会将重命名和移动操作识别为原有的代码实体被删除,在新版本中增加了新的代码实体.

(2)基于AST的变更检测方法虽然可以识别更多类型的变更操作,但是由于AST仅可展示一个文件块或一个文件的信息,所以该方法不能用于检测两个程序间的变更,且检测效率较低.

(3)当前变更检测方法主要关注于细粒度的代码实体,无法用于检测软件架构层级的变更检测.

为了解决以上问题,本文提出了一种基于MAAT(Multilevel Architecture Analysis Tree)两步匹配的架构多层次变更检测方法. 该方法中,我们首先构造程序的MAAT,然后基于MAAT执行两步匹配算法发现变更,最后将变更进行分类与聚类.

本文主要贡献主要体现在以下三个方面:

(1)本方法基于MAAT实现多层次的变更检测,可以从细粒度的代码行理解架构变更原因,也可以从高级抽象层次掌握架构变更效果.

(2)相比于基于文本匹配的变更检测方法,可识别更多的变更操作类型,例如重命名和移动,提高检测准确性.

(3)相比于基于AST匹配的变更检测方法,本方法采用两步匹配算法,通过排除大量的未变更结点,降低所需匹配的结点数,提高变更检测效率.

2　架构变更检测方法

本文提出的架构变更检测方法主要包含以下三个步骤.

步骤1:构造MAAT. 基于变更前后的软件源码及架构分别构造MAAT. 根结点为架构结点,根结点自顶向下依次为组件级结点、模块级结点、文件级结点和语句级结点.

步骤2:检测变更. 该步骤基于MAAT执行两步匹配算法发现变更,该算法包括自顶向下匹配和自底向上匹配.

步骤3:聚类变更. 将具有关联性的变更进行聚类,以便更贴近开发者实施变更时的意图,便于理解.

该方法的具体流程如图1所示.

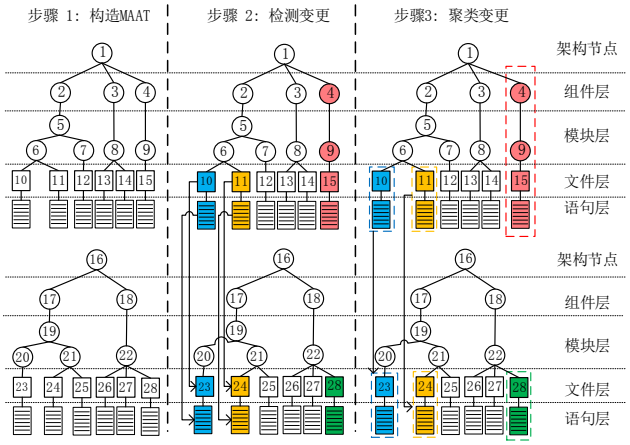


图1　基于MAAT的变更检测流程

2.1　构造MAAT

软件架构是软件源代码的高级抽象,因此,软件架构的变更需由源代码的变更逐步抽象得出. 即,架构层次的变更检测准确度取决于其对应的细粒度代码实体的变更检测准确度. 因此,我们首先实现精准检测细粒度的源代码变更,然后由源代码逐步抽象至架构层次.

为了便于分析细粒度变更与组件变更的关联关系,我们提出了MAAT. MAAT是一个用来表示架构多层次的树形结构,该树形结构的根结点代表软件整体架构,根结点之下划分为四个层级,分别为:组件级、模块级、文件级和语句级,其中组件级用于表示该架构所包含的组件,模块级为文件级和组件级之间的过渡层级[9],文件级为该架构所对应的源代码文件,语句级分别代表各个文件中的语句.

MAAT中,每个结点具有三个属性,分别是结点层次、结点类型和结点值. 结点层次是指MAAT中结点所属的抽象层次;结点类型是指语法规则中代码元素的类型;结点值是指该结点相对应的变量的字符串值.

MAAT的构建包括三个步骤:首先,构造架构结构

树;然后,为每个文件构造AST;最后,基于文件路径和文件名,将架构结构树中的叶子结点与各个文件的AST相连接.MAAT详细的构造见算法1.

算法1 MAAT构造算法

```
输入:程序源代码的根目录路径 programPath
输出:MAAT
Node root
architecture = analyzeArchitecture (programPath)
foreach component in architecture do
    Node comNode = createNode(component)
    root.addChild(comNode)
    foreach module in component do
        Node modNode = createNode(module)
        comNode.addChild(modNode)
        foreach file in module do
            AST=creatAST(file)
            ASTs.add(AST)
            Node fileNode = createNode(file)
            modNode.addChild(fileNode)
        end foreach
    end foreach
end foreach
foreach AST in ASTs do
    foreach fileNode in fileNodes
        if (fileNode.path = AST.path)
            fileNode.addChild(ASTNode)
    end foreach
end foreach
return root
```

2.2 检测变更

当构建MMAT后,基于MMAT实施两步匹配检测算法[10],该算法包含自顶向下匹配和自底向上匹配.通过自顶向下匹配算法快速排除没有发生变更的代码,然后通过自底向上算法检测剩余代码.

(1)自顶向下匹配.自顶向下的匹配过程可以快速识别两棵树之间是否具有完全同构的子树,通过快速排除大量的同构子树所包含的结点,来降低所需要匹配的结点数,从而提高整体的变更检测效率.

自顶向下的匹配过程中,使用贪婪算法匹配根结点中的子树.首先,使用MD5(Message Digest algorithm 5)算法[11]计算各个子树的哈希值.若哈希值相同,则相应的源代码也相同.若不同,则进一步比较它们的子树.为了提高检测效率,当子树的高度高于阈值时,才对其进行匹配,若低于阈值,采用自底向上算法检测未匹配的子树.

(2)自底向上匹配算法.首先计算子树相似度,然后匹配剩余结点.

子树相似度计算公式如下:

$$\mathrm{sim}_{\mathrm{contn}}(m,n)=\frac{2\times\mathrm{common}(m,n)}{\mathrm{size}(m)+\mathrm{size}(n)}$$

其中,m和n是两个结点,如果m和n的结点层次相同,且相似度大于最低阈值,则m和n是一对匹配结点.$\mathrm{common}(m,n)$表示m和n之间相同匹配结点的数量,$\mathrm{sim}_{\mathrm{contn}}(m,n)$是$m$和$n$之间的相似度,$\mathrm{size}(n)$表示以结点$n$为根结点的子树的结点总数.

当子树的根结点的抽象层次不同时,子树相似度为0;当子树的根结点类型不同时,子树相似度为0;当子树具有相同的MD5值,则两棵子树的是完全相同的.

在自顶向下的匹配过程中,优先匹配下层结点.未被匹配的结点采用Chawathe的树结构匹配方法[12],首先根据叶结点相似度计算匹配情况,然后根据非叶结点相似度计算方法,自底向上获取非叶结点的匹配结果.

2.3 变更分类与聚类

2.3.1 变更脚本分类

从物理结构的角度来看,变更脚本可以分为添加、删除、更新和移动;从层次结构来看,变更脚本可以分为组件变更、模块变更、文件变更、语句变更等.

在判定变更脚本类别时,我们设定高阈值和低阈值用于判断组件相似度与变更操作类型之间的关联.组件相似度定义如下.

假定A_1是变更前的原始架构,A_2是变更后的当前架构.假定C_i和C_j分别是A_1的A_2中的组件.组件相似度是指C_i和C_j的相似程度,其数值说明了C_j是由C_i演进得来的可能性.

组件是由多个文件组成的,组件相似度具体计算公式如下:

$$\mathrm{Sim}(i,j)=\frac{2\times|S_i\cap S_j|}{|S_i|+|S_j|}$$

其中,$\mathrm{Sim}(i,j)$是组件C_i和组件C_j的组件相似度,S_i和S_j分别是C_i和C_j所包含的文件构成的集合,$|S_i|$和$|S_j|$分别是S_i和S_j中所包含的文件数.

当两个组件之间的组件相似度高于高阈值时,可以认定这两个组件为完全相同的组件,即该组件在演进过程中并未发生变化.当两个组件之间的组件相似度高于低阈值且低于高阈值时,认定为该组件经历了更新操作.当两个组件之间的组件相似度低于低阈值时,认定这两个组件不构成一组变更对,即原始架构中的组件被删除,当前架构中的组件为新增加的组件.

同理,基于模块相似度和文件相似度的阈值判定模块层和文件层的变更操作类型.

2.3.2 变更脚本聚类

由于变更的代码实体数量较大,不便于理解,因此我们将具有一定关联性的变更进行聚类,减少变更结果

的数据项,提高易理解性[13]. 变更脚本的聚类规则如下:

(1)如果被变更的语句的相邻语句也发生了同类型的变更,则将这些具有相同类型变更的连续语句聚类为一个语句块的变更.

(2)如果结点t与其父结点的变更脚本同为增加或删除,则将t的变更聚类到其父结点的变更中.

算法2为变更脚本的分类与聚类算法,定义函数parent(n)返回结点n的父结点,定义函数getMatchNode(n)返回与结点n所匹配的结点.

算法2　变更脚本的分类与聚类算法

```
输入:
   变更前MAAT T1
   变更后MAAT T2
输出:
    变更脚本集合 editSet
//变更脚本分类
editSet={}
foreach y in bfsOrder(T2) do
   y^p=parent(y)
   x=getMatchNode(y)
   if (x = null)
       edit = add(y, y^p, i, label(y), value(y))
       editSet = editSet.add(edit)
   else
      if (value(x) != value(y))
         if (level(x) is statement-level)
            edit = update(x,value(y))
         else
            edit = rename(x,value(y))
         endif
         editSet = editSet.add(edit)
      endif
      x^p = parent(x)
      if (y^p!= x^p)
         edit = move(x,y^p,i)
         editSet = editSet.add(edit)
      endif
   endif
end foreach
foreach x in postOeder(T1) do
    y = getMatchNode(x)
    if (y = null)
       edit = delete(x)
       editSet = editSet.add(edit)
    endif
end foreach
//变更脚本聚类
foreach edit in editSet do
    if (edit is add or delete)
        node=getNode(edit)
        if (parent(node).edit = edit)
            cluster(parent(node).edit,edit)
        endif
    else
        if (node is Statement & node.next.edit =edit)
            cluster(node.next.edit, edit)
        endif
    endif
end foreach
foreach edit in editSet do
    get level of edit;
    if level is Component-level
        classifyComponentChange(edit)
    endif
    if level is Module-level
        classifyModuleChange(edit)
    endif
    if level is File-level
        classifyFileChange(edit)
    endif
    if level is Statement-level
        classifyStatementChange(edit)
    endif
end foreach
```

3　实验研究

3.1　实验设计

我们开发了一个名为ACAnalyzer的工具来实现本文方法. 为了准确分析ACAnalyzer的准确性和性能,我们基于以下三个原则从GitHub中随机选取实验用例:(1)在GitHub中所获得的星数需大于300,保障项目具有一定的认可度;(2)项目具有10个以上的发布版本,保障项目经历了一个长期的演进过程;(3)选取不同规模的实验用例,保障检测对象的广泛度.

基于以上原则,我们选取的实验用例如表1,其中第1列为项目名称,第2列为所选取的用于检测变更的演进前后版本,例如检测paperwork项目从1.2.4版本演进至1.2.7版本过程中发生的变更,第3列为该项目在GitHub中收到的星数,第4列为该项目代码规模.

由于各项目中没有关于软件架构的相关文档,所以实验中所使用的软件架构均通过基于源代码和目录的架构恢复方法[9]获取其架构.

从准确性和性能以及粗粒度和细粒度角度出发,我们探究以下三个问题.

问题1:在细粒度的代码层级上,ACAnalyzer变更

表1 实验用例信息

项目名称	版本号	星数	规模(KLOC)
paperwork	1.2.4→1.2.7	341	0.5
Pury	1.0.2→1.0.3	475	4.3
blitz4j	1.37.0→1.37.1	523	4
pocketknife	3.2.0→3.2.1	410	6.3
spark	2.7.0→2.7.1	7776	20.4
La4j	0.5.5→0.6.0	312	22

检测的准确性如何?

问题2:在粗粒度的架构层级上,ACAnalyzer变更检测的准确性如何?

问题3:ACAnalyzer检测效率如何?

3.2 细粒度代码层级的变更检测准确性分析

变更检测算法主要分为两类,分别是基于AST匹配的变更检测算法和基于文本匹配的变更检测算法.我们各选取一个具有代表性的工具进行对比与分析,分别是ChangeDistiller[14]和BeyondCompare(http://www.scooter software.com).ChangeDistiller在变更检测的相关研究中,常被用于作为分析和对比的对象,具有较高的认可度,BeyondCompare被广泛应用于工业界,具有较高的广泛应用度.

由于ChangeDistiller基于AST实现变更检测,无法检测文件级变更,所以在文件级对比实验中,ACAnalyzer只与BeyondCompare进行对比.由于BeyondCompare基于文本实现变更检测,无法准确检测更新和重命名操作,所以在语句级对比实验中,ACAnalyzer只与ChangeDistiller进行对比.

语句级检测结果如表2所示,其中第2列和第3列分别表示检测工具所检测到的变更,例如ACAnalyzer在blitz4j项目中检测到了1个更新变更.在Pury的变更检测中,ChangeDistiller检测到Pury.java文件中删除了较多语句,但通过对该文件的分析,并非真实变更,同时,ChangeDistiller在blitz4j和spark中没有检测到更新.

表2 语句级变更操作个数

项目名	ACAnalyzer	ChangeDistiller
paperwork	3个更新,4个增加	2个更新,4个增加
Pury	25个更新,6个增加,3个删除	22个更新,9个增加,24个删除
blitz4j	1个更新	0
pocketknife	4个更新	4个更新
spark	1个更新	0
La4j	18个更新,8个增加,1个移动	15个更新,8个增加,2个删除

文件级检测结果如表3所示.ACAnalyzer检测ProfileMethodAspect.java文件重命名为MethodProfilingAspect.java文件,但BeyondCompare将其检测为该文件被删除后又增加了一个新的文件.通过GitHub中的变更记录发现,ACAnalyzer所提供的变更信息更贴近开发者的变更意图.

表3 文件级变更操作个数

项目名	ACAnalyzer	ChangeDistiller
paperwork	0	0
Pury	3个增加,1个重命名	4个增加,1个删除
blitz4j	0	0
pocketknife	0	0
spark	4个增加	4个增加
La4j	2个增加	2个增加

我们基于GitHub中的变更记录验证检测结果的准确性,并采用查准率(Precision)、查全率(Recall)和F度量值(F-score)作为衡量检测准确性指标.

鉴于BeyondCompare和ChangeDistiller的检测粒度的局限性,因此只分析BeyondCompare在文件级和ChangeDistiller在语句级的检测准确性.

ACAnalyzer、BeyondCompare和ChangeDistiller的度量结果如图2所示.在文件级和语句级的变更检测中,ACAnalyzer的检测准确性均高于BeyondCompare和ChangeDistiller.

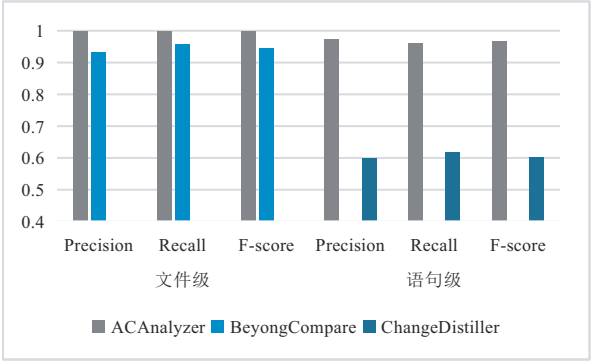


图2 各工具的检测准确性

3.3 粗粒度架构层级的变更检测准确性分析

由于目前缺少软件架构层次的变更检测工具,无法进行工具对比实验,因此本实验将变更检测结果与真实的架构图的变化和开发者所提交的更新日志进行对比,验证架构层次变更检测的准确性.

本实验以Apache-httped项目的2.4.16版本、2.4.17版本和2.4.18版本为例,分析粗粒度的架构变更检测准确性.在2.4.16版本演进至2.4.17版本和2.4.17版本演进至2.4.18版本的演进过程中,ACAnalyzer所检测到的变更如表4所示.其中,第1列演进历程中包含演进前版本和演进后版本,第2列中列出架构层次的变更,即组件变更.然后,我们结合真实架构图和更新日志汇总架构真实变更情况.

图3分别展示了该项目2.4.16版本、2.4.17版本和

表4　Apache-httped变更检测结果

架构演进历程	架构级变更检测结果
2.4.16→2.4.17	更新(modules\lua、modules) 新增(modules\http2) 删除(modules\cache)
2.4.17→2.4.18	更新(modules、modules\lua、modules\http2) 新增(modules\cache)

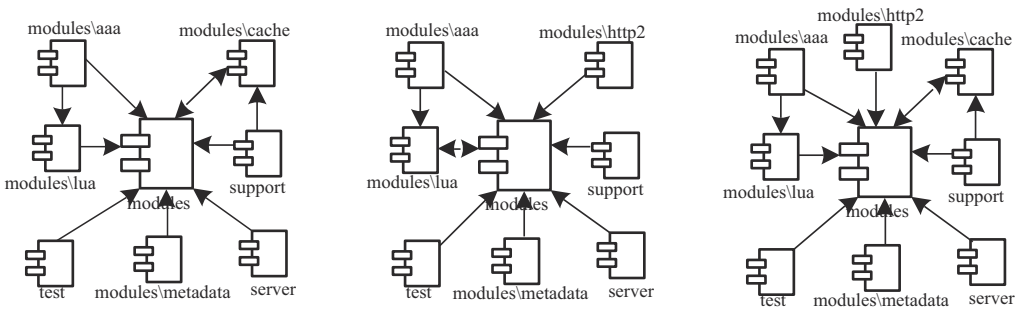
2.4.18版本的组件图.

如图3所示,Apache-httped在这三个版本的演进过程中,组件图的变更均是以组件modules为核心,这三个版本的组件信息如表5所示.

基于Apache-httped的更新日志,我们提取了该项目的功能变更及其对应的架构变更内容,具体信息如表6所示.

表5　Apache-httped组件信息

组件名	组件规模(KLOC)		
	版本2.4.16	版本2.4.17	版本2.4.18
modules	110	122	112
modules\lua	10	6	10
modules\aaa	10	10	10
server	8	8	8
support	8	8	8
modules\metadata	4	4	4
test	0.8	0.8	0.8
modules\http2	/	7	9
modules\cache	6	/	6



（a）2.4.16版本组件图　　（b）2.4.17版本组件图　　（c）2.4.18版本组件图

图3　Apache-httped组件图

表6　功能变更和架构变更信息

版本2.4.16→2.4.17		版本2.4.17→2.4.18	
功能变更	架构变更	功能变更	架构变更
开始支持http2	新增组件modules\http2	优化mod_http2	组件modules\http2规模增加
优化mod_proxy和mod_ssl	组件modules规模增加	优化mod_ssl和core	组件modules规模减小
调整mod_cache	消除组件modules\cache	调整mod_cache	恢复组件modules\cache

将表4中ACAnalyzer检测到的架构变更内容与表5的架构信息和表6中的功能变更及架构变更信息进行对比,发现ACAnalyzer检测到的架构变更与实际相符,因此ACAnalyzer能有效检测粗粒度的架构层次变更.

3.4　变更检测效率分析

基于保障检测准确性,我们选择与检测准确率较高的ChangeDistiller进行检测效率的对比.

由于ChangeDistiller只能检测文件级及其以下层级,而不能检测项目整体的变更,因此我们首先从GitHub的变更记录提取发生变更的文件,组成变更文件对,然后用ChangeDistiller逐个检测这些变更文件对,将检测所有的变更文件对的总耗时作为检测项目整体变更的时间. 两个工具检测实验用例的耗时见图4.

如图4所示,与ChangeDistiller相比,ACAnalyzer变更检测耗时更短,这表明与传统的基于AST的变更检测方法相比,ACAnalyzer检测效率更高.

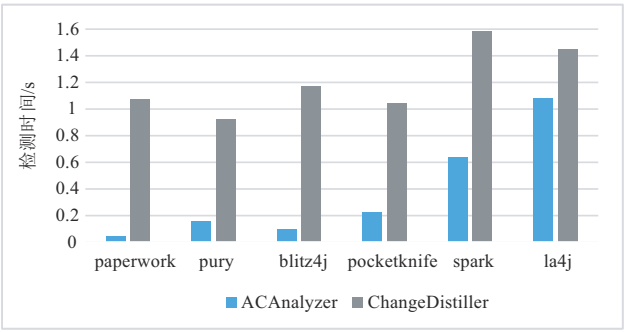


图4　ACAnalyzer和ChangeDistiller检测耗时对比

检测效率差异的主要原因在于传统的基于AST匹配的变更检测算法采用的是Chawathe的结构化信息差异分析算法[12],即从下到上计算叶结点和非叶结点的相似度,最后获得匹配集合. 而ACAnalyzer采用的是自顶向下和自底向上相结合的两步匹配算法,通过自顶向下匹配算

法快速排除未发生变更的源代码,减少需要一一匹配的结点数,从而提高整体检测效率. 尤其对于两个相邻版本而言,通常情况下,未变更的源代码的数量远远大于变更的源代码的数量. 然后,通过自底向上匹配算法检测剩余的少量源代码的变更,避免了全局匹配效率较低的问题.

总之,实验结果表明,与 ChangeDistiller 相比,ACAnalyzer的检测效率更高.

4 总结

我们提出了一种基于MAAT两步匹配的架构多层次变更检测方法,实现了语句、文件、模块和组件的变更检测,并基于该方法开发了架构变更检测工具ACAnalyzer. 实验结果表明,与一些代表性的检测工具相比,ACAnalyzer无论在细粒度的代码层级还是粗粒度的架构层级中,都具有较好的准确性,并且检测效率更高.

参考文献

[1] WANG T, WANG D D, LI B X. A multilevel analysis method for architecture erosion[C]//Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering. Lisbon: KSI Research Inc and Knowledge Systems Institute Graduate School, 2019: 443-448.

[2] SUN X B, ZHOU T C, WANG R C, et al. Experience report: Investigating bug fixes in machine learning frameworks/libraries[J].Frontiers of Computer Science, 2021, 15(6): 1-16.

[3] 王桐, 廖力, 李必信. 一种基于演进原则度量的软件架构持续演进效果评估方法[J]. 电子学报, 2019, 47(7): 1475-1481.
WANG T, LIAO L, LI B X. An approach to evaluate the sustainable evolution effect of software architecture based on the measurements of evolution principles[J]. Acta Electronica Sinica, 2019, 47(7): 1475-1481. (in Chinese)

[4] CANFORA G, CERULO L, DI PENTA M. Ldiff: An enhanced line differencing tool[C]//2009 IEEE 31st International Conference on Software Engineering. Piscataway: IEEE, 2009: 595-598.

[5] CANFORA G, CERULO L, PENTA M D. Identifying changed source code lines from version repositories[C]// Fourth International Workshop on Mining Software Repositories. Minneapolis: IEEE, 2007: 14-14.

[6] AYINALA K T, CHENG K S, OH K, et al. Tool support for code change inspection with deep learning in evolving software[C]//2020 IEEE International Conference on Electro Information Technology. Piscataway: IEEE, 2020: 13-17.

[7] DOTZLER G, PHILIPPSEN M. Move-optimized source code tree differencing[C]//2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE, 2016: 660-671.

[8] FLURI B, WURSCH M, PINZGER M, et al. Change distilling: Tree differencing for fine-grained source code change extraction[J]. IEEE Transactions on Software Engineering, 2007, 33(11): 725-743.

[9] WANG T, ZHANG Y L, LI B X. Recover and Optimize Software Architecture based on Source code and Directory Hierarchies[C]//The 31st International Conference on Software Engineering and Knowledge Engineering. Lisbon: KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019: 469-472.

[10] FALLERI J R, MORANDAT F, BLANC X, et al. Fine-grained and accurate source code differencing[C]//Proceedings of the 29th ACM/IEEE International Conference On Automated Software Engineering. New York: ACM, 2014: 313-324.

[11] MIRAKHORLI M, CLELAND-HUANG J. Detecting, tracing, and monitoring architectural tactics in code[J]. IEEE Transactions on Software Engineering, 2016, 42(3): 205-220.

[12] CHAWATHE S S, RAJARAMAN A, GARCIA-MOLINA H, et al. Change detection in hierarchically structured information[J]. ACM SIGMOD Record, 1996, 25(2): 493-504.

[13] HIGO Y, OHTANI A, KUSUMOTO S. Generating simpler AST edit scripts by considering copy-and-paste[C]// 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE, 2017: 532-542.

[14] FRICK V, GRASSAUER T, BECK F, et al. Generating accurate and compact edit scripts using tree differencing[C]// 2018 IEEE International Conference on Software Maintenance and Evolution. Piscataway: IEEE, 2018: 264-274.

作者简介

王 桐 女,1990年出生于黑龙江绥化市. 博士. 现为安徽工业大学计算机科学与技术学院讲师. 主要研究方向为智能化软件架构维护与演进.
E-mail: tongwang_se@163.com

李必信 男,1969年出生于安徽庐江县. 现任东南大学计算机科学与工程学院教授、博士生导师. 主要研究方向为智能软件开发、软件测试和缺陷检测.

王东东 男,1993年出生于山东滕州. 硕士研究生. 主要研究方向为软件架构恢复.