软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

基于动态测试的语义等价 API 函数检测方法*

姜艳杰, 刘 辉

(北京理工大学 计算机学院,北京 100081)

通讯作者: 刘辉, E-mail:liuhui08@bit.edu.cn

摘 要: 市场上出现了大量诸如 MFC、JDK 等体量庞大、使用广泛的 API 库。这些 API 代码库的出现大幅减少了重复性的开发工作，也提高了软件的质量。但是 API 函数数量巨大而且在持续演化，因此程序员很可能在不知情的情况下重复实现了某些 API 的功能。相较于经过广泛使用的 API 函数，这些语义等价的函数往往在性能、可靠性、可扩展性等方面都有一定的差距。因此，自动检测此类函数并及时替换为等价 API 函数具有重要意义。现有的克隆代码（函数）检测方法是通过比较源代码的文本相似性以识别克隆函数，因此它们难以检测功能相同但实现不同的函数代码。为此，本文提出了一种基于动态测试的方法以判定某个静态方法是否与现有 API 语义等价，进而判定是否需要进行替换。基于两个给定的函数自动生成测试用例。通过运行生成的测试用例可判定这两个函数是否语义等价。基于测试的语义等价函数检测方法通常需要较高的时间复杂度，因此将给定静态函数 m 与所有 API 函数都进行动态测试对比将导致极高的运行开销。为此，对于给定的方法 m，我们基于语法和文本相似性对海量 API 进行过滤，只筛选出少量 API 函数进行测试对比，从而大幅降低动态测试的运行开销。本文提出的方法在 10 个开源项目上进行了实验验证。实验结果表明该方法是有效的，其准确率高达 93.3%。

关键词: 克隆检测;语义等价;动态测试;静态方法

中图法分类号: TP311

中文引用格式: 姜艳杰，刘辉.基于动态测试的语义等价 API 函数检测方法.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Jiang YJ, Liu H. On Dynamic Testing Based Detection of Semantically Equivalent API Methods. Ruan Jian Xue Bao/Journal of Software, 2016 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

On Dynamic Testing Based Detection of Semantically Equivalent API Methods

JIANG Yan-Jie, LIU Hui

(Beijing Institute of Technology, Beijing 100081, China)

Abstract: There are a lot of large and extensive API libraries such as MFC and JDK. The emergence of these API code libraries significantly reduces repetitive development efforts and improves the quality of the software. But the number of API functions is huge and continues to evolve. So, the programmers are likely to repeat the implementation of certain API functions without knowing it. Compared to the widely used API functions, these semantic equivalent functions tend to differ in performance, reliability, scalability, and so on. Therefore, it is of great significance to automatically detect such functions and replace them with equivalent API functions. The existing methods to detect clone code(function) are to identify the cloning function by comparing the textual similarity of the source code, so they are difficult to detect the code functions with the same function but different implementation. In this paper, it proposes a method based on dynamic testing to determine whether a static method is equivalent to the existing API semantics. The test cases are generated based on two given functions, and then the two functions are determined whether is semantic equivalent based on the running results of the test cases. The detection of semantic equivalence function based on test usually requires a high time complexity, so the given static function m is compared dynamically to all the API functions, which results in extremely high operating overhead. So, for the given method m, we filter large amounts of API based on syntax and text similarity, and test and compare a few filtered API. The method proposed in this experiment is verified on 10 open source projects. Experimental results show that the accuracy of this method is 93.3%.

Key words: clone detection; semantically equivalent; dynamic testing; static methods

1 绪论

克隆代码是指那些相同或者相似的代码片段[1]。Roy 等人[13]将克隆代码分为四种类型，分别为：Type-1、Type -2、Type -3

* 基金项目: 国家重点研发计划资助 (2016YFB1000801)、国家自然科学基金 (61472034, 61772071,61690205) Foundation item: National Key R&D P rogram of China. (2016YFB1000801) and the National Natural Science Foundation of China (61472034, 61772071,61690205).

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

和 Type -4。Type -1 是指除了空格和注释外都相同的代码段；Type -2 是指除了标识符、类型、空格和注释外都相同的代码片段；Type -3 是指对语句做了增删改的复制代码段；Type -4 是指功能上相同但实现不同的代码段。

大量克隆代码的存在会降低软件的可维护性 [5]。为了降低克隆代码的影响，开发人员需要尽可能地消除克隆代码或者记录并跟踪克隆代码[2]。然而人工识别分布于不同文件中的克隆代码并不容易。为此，相关研究人员提出了多种自动或半自动方法以检测代码克隆[3,6]、消除克隆代码[7,8]、追踪和管理克隆代码 [4]。其中克隆检测是研究的重点。大多数克隆检测方法是针对前三种类型进行检测的。而 Type -4 的克隆检测因为涉及到复杂的语义分析，因此难度更大，目前还没有高效准确的检测方法。

作为对 Type4 克隆代码检测的一个初步尝试，本文提出了一种基于动态测试的语义等价 API 函数检测方法。API 代码库的出现大幅减少了重复性的开发工作，同时提高了软件的质量。市场上出现了大量诸如 MFC、JDK 等体量庞大且使用广泛的 API 库。鉴于 API 函数数量巨大而且在持续演化，程序员很可能在不知情的情况下重复实现了某些 API 的功能。相较于经过广泛使用的 API 函数，这些语义等价的函数往往在性能、可靠性、可扩展性等方面都有一定的差距。因此，自动检测此类函数并及时替换为等价 API 函数具有重要意义。文本提出的检测方法通过比较给定项目中的所有静态函数与 API 函数库，自动检测那些可以用 API 函数替换的静态函数。

对待测试函数 m，首先通过函数语法和文本相似性从海量 API 中筛选出一个可能与之等价的 API 函数子集(S)。利用现有的测试用例自动生成工具为待测试函数 m 生成测试用例（ta），并将 S 中的每个函数（a）测试生成的测试用例（ta），根据函数的测试结果比较函数 m 和 a 是否等价：如果两个函数运行完 ta 中的所有测试用例并具有相同的输出结果，则认为两个函数在语义上具有一致性。基于测试的克隆检测方法存在一个缺点是测试过程比较耗时：运行大量的测试用例通常是比较消耗时间和资源的。为了降低测试的复杂度，本文对基于语义进行的克隆代码检测进行了一个初步的尝试，只考虑静态方法。基于静态方法进行的克隆检测有以下优点。第一，应用程序中的静态方法的数量远远小于代码片段的数量，这极大地减少了候选等价代码段的数量。第二，相比与非静态方法而言，静态方法通常很少依赖于其他函数，因此容易运行和比较。第三，和任何的代码片段相比，静态函数明确定义了函数的输入和输出，便于测试用例的生成和运行结果的比较。除此之外，本实验还采用了以下方法来降低基于测试的克隆检测方法的复杂性。首先，对于待测试项目中的静态方法 m，基于语法筛选出和函数 m 语法匹配的 JDK 函数（若两个函数具有相同的返回类型和相同类型的形参和个数，则认为这两个函数语法匹配）。这一步筛选可以将要和 m 相比较的 JDK 函数从 9894（平均值）大幅降低至 0.24。其次，从语法匹配的 API 函数集中去除那些和 m 的函数名字相似性较低的方法。基于函数名字相似性的筛选方法进一步将与 m 相比较的 JDK 函数的数量减少 52%（平均）。最后，如果两个函数的语义一致时（大多数情况），则无须运行完所有的测试用例。当两个函数在同一测试用例上产生不同的运行结果时，即可判定它们语义不等价。在本文的实验中，高达 78%的情况下只需运行一个测试用例即可判定函数的语义等价性不成立。

本文的主要的贡献包括：

1）提出了一种基于动态测试的方法以检测源码中和 API 等价的静态函数。

2）基于开源应用程序对实验进行了验证，结果表明该方法是有效的。

本文的其它部分的结构如下：第二章描述了实验动机；第三章是实验流程；第四章对实验进行了验证；第五章讨论了相关工作；第六章是实验总结。

2 示例

DrJava(drjava-stable-20140826-r5761\src\edu\rice\cs\drjava\model\DrJavaFileUtils.java)**:**

```
1.  public static String removeExtension(String fileName)
2.  {
3.      int lastDotIndex=filename.lastIndexOf(".");
4.      if(lastDotIndex==-1)
5.        {
6.          return filename;
7.        }
8.      return filename.substring(0,lastDotIndex);
9.  }
```

API(Java\jdk1.8.0_121\src\com\sun\org\apache\xalan\internal\xsltc\compiler\util\Util.java)**:**

```
1.  public static String noExtName(String name)
2.  {
3.      final int index=name.lastIndexOf('.');
4.      return name.substring(0,index>=0?
5.                      index:name.length());
6.  }
```

Fig. 1 Motivating example

图 1 示例

本章将以图 1 所示的代码为例说明实验目的和实验的检测过程。图 1 的第一个函数 *removeExtension* 来自于一个知名的开源

应用程序 *DrJava*[1]。该函数的功能是去除形参输入的文件名 *fileName* 中的扩展名。如果 *fileName= "excel.exe "* 那么函数的输出将会是 *"excel "*。第二个函数是 Java API 函数 *noExtName*。*noExtName* 的功能也是去除形参输入的文件名 *name* 中的扩展名。因此，*removeExtension* 和 *noExtName* 在语义上是等价的。在程序开发中，用 JDK 方法来代替开发者的代码，可以提高软件的可维护性[3]，进行替换的好处是提高运行效率，节省设计代码的时间。我们通过测试的结果显示，API 函数 *noExtName* 效率明显优于语义等价函数 *removeExtension*：前者的运行时间只有后者的 50%左右。

但是要自动判定这两个方法是否语义等价（Type-4 克隆）却并不容易。目前常见的克隆检测工具，如 PMD[16]，就无法判定这两个函数是一个克隆代码。这是因为现有的克隆检测方法主要基于语法结构和文本相似性进行克隆检测。但图 1 中的两个方法在语法结构和文本上的相似度较低，因此现有克隆检测方法认为他们并不是克隆代码。

为了检测是否存在语义上等同于 *removeExtension* 的 JDK 函数，文本提出的基于测试的检测方法。首先根据 *removeExtension* 函数的形参特性，对 9894 个 JDK 中静态函数进行筛选，只保留与 removeExtension 具有相同的返回值类型、相同的形参类型和相等的形参个数的 JDK 静态函数（共 210 个）。对初步筛选出的 210 个候选函数再实行基于函数名字的过滤，只保留与 *removeExtension* 具有一定文本相似性的函数（共 22 个）。最后，对这 22 个候选函数中每一个函数 f 进行动态测试，以判定其与 *removeExtension* 是否语义等价。

动态测试过程如下：

1）基于 *removeExtension* 函数生成测试用例，并将其保存在测试用例集合 S 中。

2）基于测试用例集合 S 中的每个测试用例（输入），运行 22 个候选函数中的每个函数 *f*。如果在某个测试用例上他们的输出结果不一致，则 f 与 *removeExtension* 语义不一致。否则，f 与 *removeExtension* 语义一致。

Evosuite[14]基于 *removeExtension* 函数生成了 8 个测试用例，而且在每个测试用例上 *noExtName* 与 *removeExtension* 的输出都保持一致。因此可以判定 *noExtName* 与 *removeExtension* 语义等价。

3 语义等价 API 函数的检测方法

本章提出了一种基于测试的语义等价函数的检测方法，用以检测 Java 应用程序中可以被 JDK 函数替代的静态方法。在 3.1 小节中,简要介绍了该方法的框架结构。在 3.2-3.4 小节中,详细阐述了该方法的每个关键步骤。

3.1 方法概览

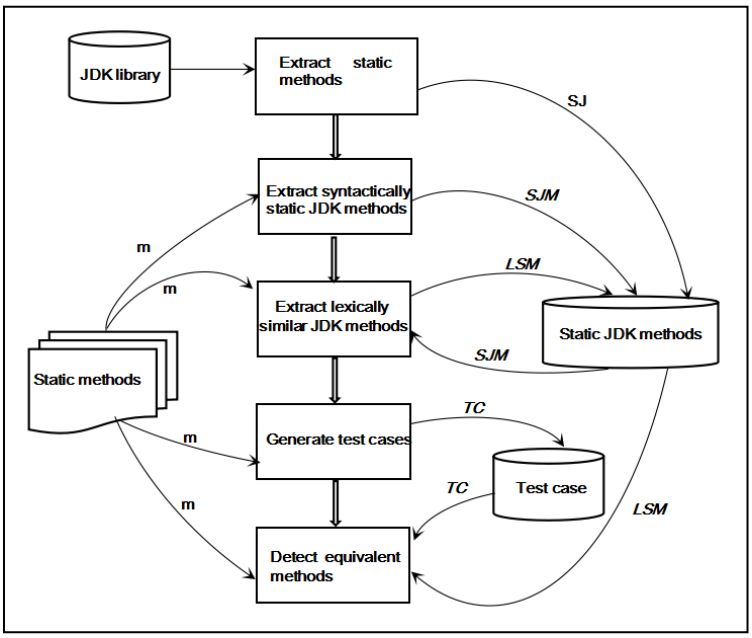

Fig. 2 Overview of the approach.

图 2 方法流程

该方法的整体流程如图 2 所示。其中，输入是 JDK 库和一个 Java 应用程序(包含静态方法)；输出是一个列表(包含与输入的静态方法语义等价的 JDK 方法)。

该方法的核心思想包括两个方面。（1）语义等价的函数可以通过运行测试用例来判定；（2）对于一个给定的函数 m,如果仅有少量 API 函数需要通过动态测试来判定其是否与 m 语义等价，那么基于测试语义等价函数检测并不需要消耗太多的计算资源。

1 https://github.com/DrJavaAtRice/drjava

为了从数据量巨大且不断增长的 JDK 库中筛选与函数 m 可能等价的 JDK 函数，该方法通过函数的语法相似性(形参)和文本相似性（函数名称）进行了两次筛选和过滤。

对于 Java 应用程序中的每一个静态函数 *m*，处理步骤如下:

(1) 首先，从 JDK 库中检索出所有与函数 *m* 语法匹配的静态 JDK 函数(标记为集合 *SJM*)。

(2) 其次，从 *SJM* 中检索出所有与函数 *m* 函数名相似的函数(标记为集合 *LSM*)。

(3) 最后，基于函数 m 生成测试用例（标记为 TC）。

(4) 基于测试用例集合 TC 中的每个测试用例，运行函数 m 及 LSM 中的每个 API 函数，从而判断是否存在某个 API 函数与 m 具有语义等价性。

后续章节将对其中的每个关键步骤进行详细介绍。

3.2 预处理

预处理包括两个部分。在第一部分中，从输入的应用程序中解析出所有的静态方法，步骤如下：

(1) 移除所有不包含静态成员（属性或方法）的类（或接口）。

(2) 从剩余的类中，移除所有的非静态方法。

(3) 移除所有会造成语法错误的 import 声明。

在第二部分中，解析出 JDK 的 API 中所有的静态函数（记为 SJ）。将这些函数的返回值类型、形参类型、函数名和其在 JDK 中存放的路径存入数据库，方便后续函数的筛选，有效地减少函数的测试时间。

3.3 候选JDK方法的初始选择

对于应用程序中的每一个静态函数(记为 *m*)，从 SJ 中筛选出可能与 m 语义等价的静态 JDK 函数。候选 JDK 方法的筛选对于本实验的性能（时间复杂度）是至关重要的。从 JDK 中，总共解析出 9894 个静态函数。对于函数 m，若用 m 生成的测试用例来检测 JDK 中所有的静态函数，将是非常消耗时间和资源的。

初始筛选包括两个步骤。在第一步中，选取和函数 m 语法匹配的静态 JDK 函数，并记为 SJM。两个函数语法匹配的条件是：

(1) 返回类型相同。

(2) 参数列表相同。

参数列表相同的形式化的定义如下。假设两个函数的参数列表分别是<type11,p11,type12,p12,…type1k,p1k>和<type21,p21,type22,p22,…type2m,p2m>.如果 k=m 且 type1j=type2j(1<=j<=k)。候选的 JDK 函数必须与 m 语法匹配。如果两个函数语法不匹配，则无法在 JDK 函数和 m 上运行相同的测试用例，因此也就不能通过测试来判断它们是否语义等价。

在初始选择的第二步中，从 SJM 中选取与待测函数的函数名相似的函数。函数名通常传达丰富的语义，并且好的函数名应该揭示函数的功能。因此，语义等价的函数通常有相似的函数名。基于这一假设，从 SJM 中选取与函数 m 词法相似度超过给定阈值的 JDK 方法，并记为 LSM。

函数名相似度的计算公式如下：

$$lexSim(\mathrm{m}_1,m_2)=Sim(\mathrm{name}_1,\mathrm{name}_2)=\frac{comterms(\mathrm{name}_1,\mathrm{name}_2)\times 2}{\mathrm{terms}(\mathrm{name}_1)+terms(\mathrm{name}_2)}$$

其中，m1 和 m2 是两个函数，name1 和 name2 是它们的函数名。comterms(name1,name2)是在 name1 和 nama2 中同时出现的单词的数目。terms(name1)是 name1 被分割的单词的数目。例如：两个相似的词（比如相同单词的单数形式和复数形式），如果基于字符的词法相似度超过 0.25，那么它们被认为是相同的。

3.4 测试用例的生成与等价方法的检测

对于一个给定的静态函数 m，如果不存在语法和词法均相似的静态 JDK 函数（也就是说，SJM 是空的），那么终止程序；否则，对于给定的函数 m，通过语法和词法筛选，得到候选语义等价的 API 函数。此时，生成函数 m 的测试用例，候选匹配的 API 函数运行 m 生成的测试用例，根据测试结果判断语义是否一致。测试用例的覆盖率越高，通过在两个函数上运行测试用例来发现其不同的可能性就越大。

对于每一个待测试的静态函数 m，通过以下步骤判断 LSM 中是否存在与其语义等价的 API 函数。

(1)对于静态函数 m，通过词法和语法对 JDK 中函数进行过滤，匹配到相应的函数 T，自动生成函数 m 的测试用例并运行得出测试结果。此时，让 LSM 中的每一个函数 f 调用 m 生成的测试用例，将测试结果和 m 的测试结果进行对比，当发现一个结果不同时，说明 m 和 T 不等价，并立即终止。

（2）如果所有测试用例的运行结果均相等，说明 m 和 T 等价。

4 实验验证

4.1 研究问题

- RQ1：给定一个项目源码，本文所提出的方法能否从中检测出与某个 JDK 函数语义等价的函数？检测算法的查准率如何？是否会导致大量的误报（false positive）？
- RQ2：本文提出的基于语法相似性的过滤方法是否有效？它能在多大程度上减少需要通过动态测试进行对比的 JDK 函数?
- RQ3：本文提出的基于函数名称相似性的过滤方法是否有效？它能在多大程度上减少需要通过动态测试进行对比的 JDK 函数?
- RQ4：基于本文给定的方法，要判定两个函数是否语义等价平均需要运行几个测试用例？

研究问题 RQ1 主要关注检测方法的查准率。查准率对于该方法的成功至关重要。如果有大量的误报产生，可能导致错误的函数替换并引入软件缺陷。研究问题 RQ2 和 RQ3 主要考察所采用的过滤方法（用于减少候选的等价 JDK 函数的数量）的有效性。这些过滤方法可以使大量的 JDK 函数免于进一步的动态比较（测试），从而使所提出方法消耗较少的时间。因此，这些过滤方法的有效性对于本方法的运行性能（时间复杂度）具有重要意义。研究问题 RQ4 意在揭示所提出方法的平均性能。

4.2 原型实现

为了对所提出的方法进行试验验证，我们实现了一个原型系统 Tisem (Test-based Identification of Semantically Equivalent Static Methods)。Tisem 是 Eclipse 的一个插件。选择 Eclipse 的原因在于它拥有强大的静态源代码分析工具，例如 JDT。Tisem 通过 EvoSuite (http://www.evosuite.org/)的 API 来生成和运行测试用例。选择 EvoSuite 是因为如下几个原因：（1）它是一个用于生成测试用例的非常流行也非常有效的工具；（2）它由 Sheffield 和其他大学联合开发，是免费开源的；（3）EvoSuite 生成的测试用例可以直接在 JUnit 上运行。Tisem 解析 JDK 静态函数的信息，并将其存入数据库。其中，数据库由 Navicat（http://www.navicat.com/）管理。

4.3 应用程序选择

为了验证所提出的方法，实验从 GitHub 中选择了 10 个有名且开源的 Java 应用程序。所选应用程序如表 1 所示。实验选择这些项目的原因是：首先，这些应用程序的源代码是公开的，这允许其他研究人员重复本实验。其次，这些应用是具有较高的知名度，代码质量较高。如果这些高质量的应用程序也包含语义等价的静态函数，那么在其他应用程序中将有更多的静态函数。

Table 1 Evaluation Results

表 1 实验结果

Applications	Static methods	Detected equivalent methods N_1	Confirmed equivalent methods N_2	N_2/N_1
Areca	656	4	4	100%
iText	812	1	1	100%
JabRef	481	3	3	100%
Vuze	1110	2	2	100%
PMD	291	3	3	100%
Jsch	87	1	1	100%
uniCentaoPos	178	0	0	/
DavMail	175	0	0	/
JasperReports	63	2	2	100%
Mondrian	720	3	3	100%
Total	4573	19	19	100%

为了将所选项目的静态函数和 JDK 静态函数进行对比，本实验从 JDK 的最新版本（1.8 版本）中解析出所有静态方法。总共解析出 9,894 个静态 JDK 函数。

4.4 实验过程

对于每个应用程序，实验工作按如下步骤进行。

（1）首先，在所选项目上运行 Tisem。Tisem 会报告一组与某些静态 JDK 方法语义等价的静态方法 SM。

（2）第二，如果 SM 为空，则终止对该项目程序的检测。

（3）第三，手工检查 SM 中的每个方法 m，以确定 m 和 JDK 方法是否真的语义等价。

（4）最后，计算出方法的查准率。

在实验过程中，如果 SM 集合为空，不一定就没有语义等价的静态方法，原因为：第一，经过函数名称相似性的筛选，可能会损失一些语义等价的函数。第二，存在形参类型和数量不一致但函数语义一致的方法，但在本实验中不能对此类方法进行

有效地检测。所以，当 SM 为空时，终止对该项目的检测，认为没有语义等价的函数。

在步骤（3）中，手工检查的具体步骤为：对集合 SM 中的每一个函数，根据程序记录的存储路径，在 API 中找到对应的函数体，手工地检测两个函数的语义是否一致，对于较简单的函数，可以根据经验直接判断语义是否相同；对于复杂函数，可以编写测试用例对两个函数进行检测，从而判断函数语义是否一致。

4.5 实验结果

1）RQ1：实验方法的查准率

实验方法的评估结果见表 1。从表 1 的结果来看，本文提出的检测方法具有较高的查准率。总共检测出 19 个与对应 JDK 方法语义等价的函数，经过手工确认，这 19 对函数确实是语义等价的，即检测方法的查准率高达 100%。此外，从检测结果看，大部分应用程序（80%= 8/10）确实包含了可以用 JDK 方法替代的静态函数。而且本文提出的方法已经成功检测到这些函数。

检测语义等价函数的另一个作用是可以发现某些软件缺陷。图 3 所示的这两个方法的功能（语义）是等价的，都是比较两个 byte 类型的数组内容是否相同。基于这两个函数 Evosuite 自动生成了 9 个测试用例。在 8 个测试用例上，这两个函数的输出结果始终保持一致。但是在最后一个测试用例（a= byteArray0,b=null）上，则左边的函数返回 false 而右边的函数将抛出空指针异常（代码缺陷）。虽然本文提出的方法会基于测试用例的运行结果判定它们在语义上并不等价，但它发现只有一个测试用例不一致，因此它会警告提醒这两个函数虽然仅有细微差异，并建议程序员检查是否是软件缺陷。

Jsch(jsch-0.1.51\src\com\jcraft\jsch\ChannelX11.java)**:**

```
1.  public static boolean equals(byte[] a, byte[] a2)
2.  {
3.     if(a==null ||a2==null)
4.     return false;
5.     if(a==a2)
6.     return true;
7.     int length=a.length;
8.     if(a2.length!=length)
9.       return false;
10.    for(int i=0;i<length;i++)
11.     {
12.       if(a[i]!=a2[i])
13.         return false;
14.     }
15.    return true;
16. }
```

API(Java\jdk1.8.0_121\src\com\sun\org\apache\bcel\internal\classfile\Utility.java)**:**

```
1.  public static boolean equals(byte[] a, byte[] b)
2.  {
3.     int size;
4.     if((size=a.length)!=b.length)
5.       return false;
6.     for(int i=0;i<size;i++)
7.      {
8.        if(a[i]!=b[i])
9.        return false;
10.     }
11.    return true;
12. }
```

Fig.3 False positive

图 3 检测错误的例子

为了测试现有的克隆检测工具是否也能检测到这些语义等价的函数，我们利用克隆检测工具 PMD 对这 19 对等价函数进行检测。测试结果如表 2 所示，它们中的大部分(78.9%=15/19)都不能被 PMD 所检测到。

通过以上分析，高质量和众所周知的项目程序包含与 JDK 函数语义等价的静态函数，并且，所提出的方法在检测这些函数时，也是有效的。

2）RQ2：语法相似性筛选的有效性。

为了降低检测方法的计算开销，本文提出的检测方法从海量 JDK 函数中只选择了一小部分进行基于动态测试的语义等价性检测。选择过程主要分成两个部分。首先，它选择语法匹配的 JDK 函数；其次，从所得到的函数中根据名字相似性进一步过滤筛选。基于语法相似性筛选的结果（第一步）如表 3 所示。第一列为应用程序，第二列为每个应用程序中静态函数的数量，第三列显示了语法匹配函数对（每一对中包括一个来自应用程序的函数和一个 JDK 函数）。从表 2 可知基于语法的 API 函数过滤方法是非常有效的。理论上每个静态函数需要和每个 JDK 静态函数进行对比，因此总的函数对是 N = N1*N2=4,573×9,894==45,245,262。其中 N1 是应用程序中的静态函数数量，N2 是 JDK 静态函数的数量。基于语法的过滤方法将 N 大幅降低到 1,107，降幅高达 88.8%(8787/9894)，从而大幅减少了动态运行测试的时间。

Table 2 Detection Results of PMD

表 2 PMD 检测结果

Static Method	JDK Method	Containing Clones	Identical
int Get16(byte b[],int off)	int get16(byte b[],int off)	True	True
boolean equals(Object o1,Object o2)	boolean equals(Object a, Object b)	False	False
boolean equals(byte[] o1,byte[]o2)	boolean equals(byte[]a, byte[]a2)	True	False
boolean equals(int[]o1,int[] o2)	boolean equals(int[]a, int[]a2)	False	False
Int relativeCCW(double x1,doubley1…)	int relativeCCW(double x1,y1…)	True	True
boolean isAlphaLower(char check)	boolean isLowerCase(char ch)	False	False
boolean isAlphaUpper(char check)	boolean isUpperCase(char ch)	False	False
String rtrim(String s)	String rtrim(String value)	False	False
int ByteArrayToInt(byte[]b, int offset)	int BytesToInt(byte[]array, int offset)	False	False
Window getWindow(Component p)	getWindowForComponent(Component *)	True	False
int compare(String s1,String s2)	int compare(String s, String t)	False	False
boolean areEqual(Object v, Object t)	boolean equals(Object a ,Object b)	False	False
Long getLongValue(String a, long b)	Long getLong(String nm, long val)	False	False
Long intFlagToLong(int flag)	Long ToUnsignedLong(int x)	False	False
String byteArrayToHexString(byte[]a)	String toHexString(byte[]bytes)	False	False
boolean isAbstract(Class c)	boolean isAstractBase(Class clazz)	False	False
boolean equals(byte[]foo, byte[]bar)	boolean equals(byte[]a, byte[]b)	False	False
XMLInputFactory getXmlInputFactory()	XMLInputFactory newFactory()	False	False
String removeExtension(String fn)	String noExtName(String name)	False	False

Table 3 Effect of syntax and Lexical Similarity Based Selection

表 3 词法和语法相似性筛选的结果

Applications	Static methods N_1	Pairs of syntactically matching methods N_2	Pairs of lexically similar methods N_3	N_2/N_1	N_3/N_1	N_3/N_2
Areca	656	141	24	0.21	0.04	17%
iText	812	164	112	0.20	0.14	68%
JabRef	481	509	310	1.06	0.64	61%
Vuze	1110	59	5	0.05	0	8%
PMD	291	19	16	0.07	0.05	84%
Jsch	87	24	18	0.28	0.21	75%
uniCentaoPos	178	24	15	0.13	0.08	63%
DavMail	175	18	13	0.10	0.07	72%
JasperReports	63	29	5	0.46	0.08	17%
Mondrian	720	120	29	0.17	0.04	24%
Total	4573	1107	547	0.24	0.12	49%

3）RQ3：基于函数名称相似性的过滤方法

基于函数名称相似的过滤方法，将函数名字差异较大（相似度小于阈值 η）的函数对过滤掉从而减少需要通过动态测试进行等价性验证的函数对数，降低计算开销。阈值 η 的设置对过滤效果有比较大的影响：如果 η 过大，可能导致很多语义等价的函数对被过滤掉，进而降低检测算法的查全率；如果 η 过小，则可能过滤效果并不明显，不能有效降低检测算法的计算复杂度。阈值对检测方法的影响如图 4 和图 5 所示。其中图 4 展示了阈值对检测方法最终发现的语义等价函数的数量（N1）的影响。图 5 展示了阈值对候选语义等价函数的数量（N2）的影响（即需要通过动态测试进行语义等价性判定的函数对的个数）。从图中可见，当 $\eta < 0.25$ 时，随着 η 的增长，N1 有轻微下降而 N2 则迅速下降。但 $\eta > 0.25$ 时，随着 η 的增长，N1 快速下降，而 N2 的下降速度明显变缓。因此，我们将 η 的默认值设置为 0.25。

基于函数名称相似性的过滤方法，其效果如表 3 所示。此方法虽然可以明显地减少候选等价函数的数量，但在实验过程中也造成一定的损失，如可能会过滤掉某些确实语义等价但函数名称差异较大的函数对。这部分函数对的统计结果如表 4 所示。

在所选的 10 个项目中，检测等价的函数总数为 19 个，损失函数为 2 个，损失函数占总函数的 9.5%。在损失代价为 9.5%的基础上，大幅的减少了候选等价函数的数量，这可以有效地缩短函数的测试时间，节省了测试资源。

4）RQ4：运行测试用例的数量：运行测试用例对待测试方法和筛选后 API 函数进行语义一致的检测是本实验中最为消耗时间和资源的。虽然通过函数语法特性和函数名称相似性进行筛选后，有效地缩小了函数的测试集，但是在函数调用生成的全部测试用例时，实验过程仍然是耗时的。此时，将 API 函数生成的测试用例和运行的实验结果进行存储，可以有效地缩短实验的测试验证时间。

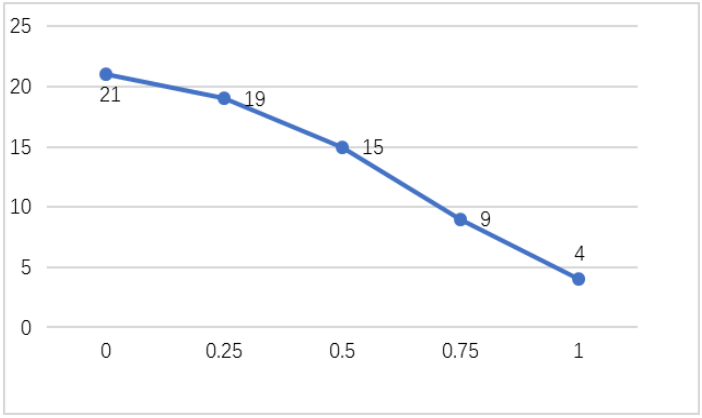

Fig.4 The Number of Detected Equivalent Functions（N1）

图 4 方法检测到的等价函数个数 N1

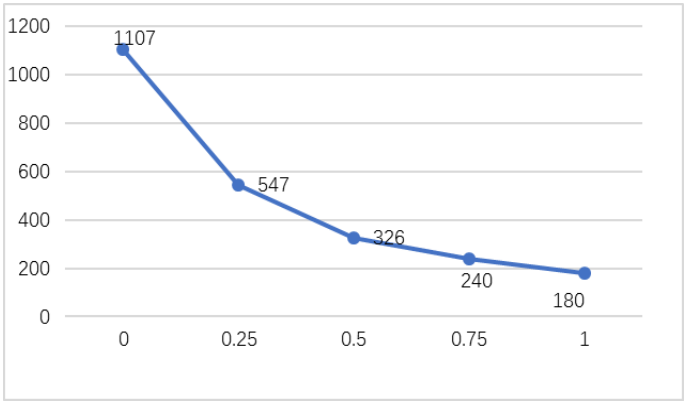

Fig.5 The Number of Equivalent Functions Need to Test（N2）

图 5 需要动态测试语义等价性的函数对的数量 N2

Table4 The number of detected equivalent methods and Lost equivalent methods

表 4 等价函数和损失函数数量的统计

Applications	**Detected equivalent methods** N_1	**Lost equivalent methods** N_2	$N_2/(N_1+N_2)$
Areca	4	1	20%
iText	1	0	0%
JabRef	3	1	25%
Vuze	2	0	0%
PMD	3	0	0%
Jsch	1	0	0%
uniCentaoPos	0	0	0%
DavMail	0	0	0%
JasperReports	2	0	0%
Mondrian	3	0	0%
Total	19	2	9.5%

生成测试用例的数量和实际运行的测试用例的数量如表 5 所示。第一列是应用程序的名称，第二列为需要通过动态测试判定是否语义等价的函数对，第三列是被第一个测试用例杀死的函数对，第四列表示生成的测试用例的数量，第五列是实际运行的测试用例的数量。

通过表 5 可以观察到以下结果：首先，在大多数情况下（79%= 430/547）通过运行一个测试用例即可判断两个函数在语义上是否等价。这在很大程度上降低了需要运行的测试用例数量。其次，平均而言，每一对函数只需要执行 1.16 = 637/547 个测试用例即可判定其是否语义等价。因此，本文提出的基于动态测试的语义等价函数检测方法相对快速高效，并不需要耗费太多的计算资源。在普通 PC 机上（INTER i7， 8G 内存，Win7），10 个项目上的总的匹配检测时间为 43.3 分钟。

Table 5 Generated and Run Test Cases

表 5 生成和运行的测试用例数量

Applications	Pairs of methods to be tested N_1	Pairs killed by the first test case N_2	Generated test cases N_3	Run test cases N_4	N_2/N_1	N_4/N_3	N_4/N_1
Areca	24	13	116	49	54%	42%	2.04
iText	112	88	312	148	79%	47%	1.32
JabRef	310	260	972	289	84%	30%	0.93
Vuze	5	1	38	16	20%	42%	3.2
PMD	16	10	65	36	63%	55%	2.25
Jsch	18	14	63	19	78%	30%	1.05
uniCentaoPos	15	8	73	21	53%	29%	1.4
DavMail	13	10	98	21	77%	21%	1.62
JasperReports	5	1	5	1	20	20%	0.2
Mondrian	29	25	146	37	86%	25%	1.28
Total	547	430	1888	637	79%	34%	1.16

4.6 实验结果有效性分析与方法的局限性

针对本实验的结果，存在以下不确定因素。首先实验只测试了 10 个应用程序。如果使用其他的测试项目可能会有不同的实验结果。为了提高实验结果的一般性，我们从 GitHub 中选择属于不同领域的、应用广泛、具有较高质量的应用程序。实验结果表明本文提出的方法在不同领域项目上都取得了较好的结果，因此本实验的结果具有较高的一般性。在高质量实验项目上的实验结果表明本文提出的方法依然可以找出与 JAVA API 语义等价的静态函数，说明与 API 语义等价的函数具有一定的广泛性。其次，在手动检查两个函数是否语义等价具有一定的主观性。为了降低这个干扰因素，我们让具有丰富开发经验的多个高级开发人员一起检查，并通过进一步的手动设计的附加测试用例进行比较。

由于本实验是对动态检测语义等价的一个初步尝试，所以选择了静态方法，静态方法较独立，可方便高效地生成其对应的测试用例，易于检测。非静态方法较复杂，今后将进一步扩展，实现对非静态方法语义等价的检测。同时，本实验只对 JAVA API 进行了检测。为了更好的验证方法的通用性，应对多种编程语言的 API 进行检测。在生成方法的测试用例部分，仅仅使用了 Evosuite 一种插件。为了提高测试用例的覆盖率，应选择多种工具来生成待测试函数的测试用例，当方法运行完所有的测试用例，结果均一致时，则认为两个方法是语义等价的。在实际开发中，存在形参类型和数量不一致但语义功能一致的方法，本实验所提出的方法对这类方法不能进行有效地检测，造成了一定的损失。

5 相关工作

在软件系统的开发过程中，及时检测出克隆代码是重要的，因为大量的克隆代码会给系统带来很多危害，比如：增大了系统源代码的规模、提高了对资源的要求和扩大了错误的传播等，因此，需要对克隆代码进行有效的检测、重构和管理。Lague 等人[29]基于克隆管理做了大量的研究，采用的主要方法为：第一，尽量保证在系统加入一个新的函数时，其在程序的源代码中没有复本。第二，保证克隆代码的一致性修改。克隆代码的检测是克隆研究领域的基础性工作[14]，克隆检测的大致步骤为预处理、转换、匹配检测和生成克隆报告[10,11]。目前，多种克隆检测方法已被提出，比如：基于文本的检测方法[9]、基于词法的检测方法[8]和基于语法的检测方法[25]等。根据目前的研究现状来说，Type-1 和 Type-2 类型的克隆代码可以被有效地检测，对 Type-3 类型的克隆代码的检测还需要深入的研究，而 Type-4 类型克隆代码处于初始阶段[12,13]。流行的克隆检测工具包括 Dup[19], Duploc [20], NICAD [24,25], CCFinder [15,26]和 PMD 等。这些检测工具只可以有效地识别 Type-1、Type-2 和 Type-3 的克隆代码。由于可发了多种克隆代码检测的方法和工具，所以开发人员将不同的检测方法进行对比，这将极大地推动克隆代码检测技术的发展，有利于更好的进行软件的维护。

测试用例的自动生成是软件测试领域的研究热点之一，相关研究人员已经提出多种方法[16]。Weyuker 等人[21]提出了一种基

于布尔规范的方法。Coward 等人[22]提出了一种基于符号执行的测试用例生成方法，其按照程序的执行顺序将变量表示成符号，从而得到变量的值。Bird[19]提出了一种生成随机测试用例的方法，它可以在较短时间内生成大量的测试用例。Korel[30]提出了一种动态方法，它在执行程序时采用了步进的方法，一次前进一个分支谓词。除此之外，M.Gallagher 和 Neelan Gupta[31]分别介绍了程序插装和迭代松弛的动态方法。目前，存在很多工具可以自动生成测试用例，如 EvoSuite 和 Milu[23]。这些工具的研发有助于比较两个方法之间的语义等价性，从而成为实验中提出方法的实现基础。

6 总结与展望

作为基于语义进行克隆代码检测的一个初步尝试，我们采用了一种基于测试的方法来检测在语义上等价于 JDK 函数的静态函数。该方法的关键是通过语法和文本相似性进行筛选，有效地减少了需要通过动态测试进行比较的函数对。在开源的项目上的实验结果表明，开源项目中的很多静态方法可以被 JDK 中的静态方法进行替换，而且本文提出的检测方法可以快速准确地将这些函数检测出来。

在未来的工作中，我们将考虑如何将将本文的方法扩展到非静态函数。另外，本文的方法目前主要专注于 Java 项目，后续将考虑在 C 语言等其他语言上进行试验，明确该方法是否能适用于不同的编程语言。

References:

[1] 任浩,史庆庆,张丽萍,刘东升.克隆代码检测方法综述.电脑编程技巧与维护，2011(20):19-23.

[2] 曹羽中，金茂忠，刘超.克隆代码检测技术综述.计算机工程与科学，2006（z2）:9-14.

[3] 郭颖,陈锋宏,周明辉.大规模代码克隆的检测方法.计算机科学与探索，2014(04):417-426.

[4] 张刚.代码克隆扩展分析及管理技术研究[D] 复旦大学.2013.

[5] 袁悦.基于复制粘贴操作的克隆代码一致性维护需求预测方法[D].2016.

[6] 郭婧，吴军华.一种新的检测结构克隆的方法.计算机工程与科学 2007(09):78-83.

[7] 李亚军,徐宝文，周晓宇.基于 AST 的克隆序列与克隆类识别.东南大学学报.2008:228-232.

[8] 梅宏，王千祥，张路，王戟.软件分析技术发展.计算机学报.2009:1697-1710.

[9] 叶青青.软件源代码中的代码克隆现象及其检测方法.计算机应用与软件 2008(09):147-159.

[10] Roy C K, Cordy J R. A survey on software clone detection research[J]. Queen’s School of Computing TR, 2007, 541(115): 64-68.。

[11] Sheneamer A, Kalita J. A Survey of Software Clone Detection Techniques[J]. International Journal of Computer Applications, 2016: 0975-8887.

[12] Sudhamani M, Rangarajan L. Structural similarity detection using structure of control statements[J]. Procedia Computer Science, 2015, 46: 892-899.

[13] Roy C K, Cordy J R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization[C]//Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. IEEE, 2008: 172-181.

[14] http://www.evosuite.org/

[15] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.

[16] https://pmd.github.io/

[17] Kamimura M, Murphy G C. Towards generating human-oriented summaries of unit test cases[C]//Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, 2013: 215-218.

[18] Chang C H, Lin N W. Constraint-Based Test Case Generation for White-Box Method-Level Unit Testing[C]//Computer Symposium (ICS), 2016 International. IEEE, 2016: 601-604.

[19] Baker B S. Parameterized pattern matching: Algorithms and applications[J]. Journal of computer and system sciences, 1996, 52(1): 28-42.

[20] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code[C]//Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. IEEE, 1999: 109-118.

[21] Göde N, Koschke R. Frequency and risks of changes to clones[C]//Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011: 311-320.

[22] Roy C K, Cordy J R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization[C]//Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. IEEE, 2008: 172-181.

[23] Jia Y, Harman M. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language[C]//Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference. IEEE, 2008: 94-98.

[24] Baxter I D, Yahin A, Moura L, et al. Clone detection using abstract syntax trees[C]//Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998: 368-377.

[25] Agrawal A, Yadav S K. A hybrid-token and textual based approach to find similar code segments[C]//Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on. IEEE, 2013: 1-4.

[26] CCFinderX, http: //www.ccfinder.net/.

[27] Johnson J H. Identifying redundancy in source code using fingerprints[C]//Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1. IBM Press, 1993: 171-183.

[28] Baker B S. On finding duplication and near-duplication in large software systems[C]//Reverse Engineering, 1995., Proceedings of 2nd Working Conference on. IEEE, 1995: 86-95.

[29] Lague B, Proulx D, Mayrand J, et al. Assessing the benefits of incorporating function clone detection in a development process.

[30] Korel B. Automated software test data generation[J]. IEEE Transactions on software engineering, 1990, 16(8): 870-879.

[31] Gupta N, Mathur A P, Soffa M L. Generating test data for branch coverage[C]//Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on. IEEE, 2000: 219-227.