软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007108]

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

基于程序约束的细粒度 JVM 测试程序约简方法 *

杜义恒[1]，王赞[1]，赵英全[1]，陈俊洁[1]，陈翔[2,3]，侯德俊[4]，郑开[1]

[1](天津大学 智能与计算学部，天津 300350)

[2](南通大学 信息科学技术学院，江苏 南通 226019)

[3](中国科学院信息工程研究所,信息安全国家重点实验室，北京 100093)

[4](天津大学 信息与网络中心，天津 300072)

通讯作者：侯德俊，E-mail: hdj@tju.edu.cn

摘 要: 为了对 Java 虚拟机(JVM)进行测试，开发人员通常需要手工设计或利用测试生成工具生成复杂的测试程序，从而检测 JVM 中潜在的缺陷。然而，复杂的测试程序给开发人员定位及修复缺陷带来了极高的成本。测试程序约简技术旨在保障测试程序缺陷检测能力的同时，尽可能的删减测试程序中与缺陷检测无关的代码。现有研究工作基于 Delta 调试在 C 程序和 XML 输入上可以取得较好的约简效果，但是在 JVM 测试场景中，具有复杂语法和语义依赖关系的 Java 测试程序约减仍存在粒度较粗、约简效果较差的问题，导致约简后的程序理解成本依然很高。因此，针对具有复杂程序依赖关系的 Java 测试程序，本文提出一种基于程序约束的细粒度测试程序约简方法 JavaPruner。首先在语句块级别设计细粒度的代码度量方法，随后在 Delta 调试技术上引入语句块之间的依赖约束关系来对测试程序进行约简。以 Java 字节码测试程序为实验对象，通过从现有的针对 JVM 测试的测试程序生成工具中筛选出具有复杂依赖关系的 50 个测试程序作为基准数据集，并在这些数据集上验证 JavaPruner 的有效性。实验结果表明，JavaPruner 可以有效删减 Java 字节码测试程序中的冗余代码。与现有方法相比，在所有基准数据集上约减能力平均可提升 37.7%。同时，JavaPruner 可以在保障程序有效性及缺陷检测能力的同时将 Java 字节码测试程序最大约简至其原有大小的 1.09% ，有效降低了测试程序的分析和理解成本。

关键词: Java 虚拟机;JVM 测试;测试程序约简; Delta 调试

中图法分类号: TP311

中文引用格式: 杜义恒，王赞，赵英全，陈俊洁，陈翔，侯德俊，郑开. 基于程序约束的细粒度 JVM 测试程序约简方法. 软件学报. http://www.jos.org.cn/1000-9825/7108.htm

英文引用格式: Du YH, Wang Z, Zhao YQ, Chen JJ, Chen X, Hou DJ, Zheng K. A Fine-grained JVM Test Program Reduction Method Based on Program Constraints. Ruan Jian Xue Bao/Journal of Software (in chinese). http://www.jos.org.cn/1000-9825/7108.htm.

A Fine-grained JVM Test Program Reduction Method Based on Program Constraints

DU Yi-Heng[1], WANG Zan[1], ZHAO Ying-Quan[1], CHEN Jue-Jie[1], CHEN Xiang[2,3] , HOU De-Jun[4], ZHENG Kai[1]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

[2](School of Information Science and Technology, Nantong University, Nantong 226019, China)

[3](State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China)

[4](Network and Information Center, Tianjin University, Tianjin 300072, China)

Abstract: In order to test the Java Virtual Machine (JVM), developers often need to manually design or use test generation tools to create complex test programs to detect potential defects in the JVM. However, complex test programs bring high costs to developers in

* 基金项目:国家自然科学基金(62232001、62002256).
收稿时间: 2023-09-10; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05

terms of locating and fixing defects. Test program reduction techniques aim to minimize the amount of code in test programs that is unrelated to defect detection, while maintaining the program's defect detection ability. Existing research has achieved good reduction results based on Delta Debugging in C programs and XML inputs, but in the JVM testing scenario, Java test programs with complex syntax and semantic dependencies still have problems with coarse granularity and poor reduction effects, resulting in high comprehension costs even after reduction. Therefore, this paper proposes a fine-grained test program reduction method, called JavaPruner, based on program constraints for Java test programs with complex program dependencies. JavaPruner first designs a fine-grained code measurement method at the statement block level, and then introduces dependency constraint relationships between statement blocks based on Delta Debugging to reduce the test program. This work targets Java bytecode test programs, and selects 50 test programs with complex dependencies from existing test program generation tools for JVM testing as the benchmark dataset to evaluate the effectiveness of JavaPruner. Experimental results show that JavaPruner can effectively reduce redundant code in Java bytecode test programs. Compared to existing methods, the reduction capability can be improved by an average of 37.7% on all benchmark datasets. At the same time, JavaPruner can maximize the reduction of Java bytecode test programs to 1.09% of their original size while ensuring program effectiveness and defect detection ability, effectively reducing the analysis and comprehension costs of test programs.

Key words: Java Virtual Machine; JVM Testing; Test Program Reduction; Delta Debugging

Java 虚拟机（Java Virtual Machine，简称 JVM）作为 Java 生态系统的重要基石，其正确性及稳定性直接影响着所有经其编译、运行的上层应用软件的质量[1]。为了保障 JVM 的质量安全，通常需要开发人员针对性的设计或利用测试生成工具生成大量复杂的测试输入(即 Java 测试程序)，来对 JVM 进行更充分的测试。例如，基于程序变异的 JVM 测试程序生成工具 classfuzz[2]、classming[3]，以及基于程序合成的 J VM 测试程序生成工具 JavaTailor[4]、VECT[5]等。然而，这些生成工具主要通过叠加大量的变异操作，或在测试程序中引入大量其他测试程序的代码片段来生成新的测试程序，使得新生成的测试程序往往包含大量复杂的程序结构且较弱的内在逻辑性。因此，尽管新生成的测试程序有助于触发 JVM 缺陷，但测试程序复杂的分析成本为开发人员定位和修复缺陷带来了极大的困难。

为了提升开发人员定位以及修复缺陷的效率，现有研究多通过 Delta 调试技术(Delta Debugging)来对新生成的测试程序进行约简[6][7][8]。实验表明，该方法在 C 程序以及以 XML 文件的测试输入上可以取得较好的约简效果。然而，在具备复杂依赖关系的 Java 字节码测试程序上，现有方法依然存在以下不足：（1）Delta 调试技术无法处理具备复杂约束的测试程序，因为没有充分考虑程序中复杂的语法和语义结构，从而使得其约简的测试程序会产生大量不合法的测试程序输入，导致测试程序原有缺陷检测能力下降。（2）现有针对 Java 字节码测试程序约简的工作仍存在约简粒度较粗的问题，使得约简后的测试程序依然包含大量的冗余代码，开发人员仍需要花费大量时间来分析和理解测试程序。例如，Kalhauge 等人提出的针对 Java 字节码程序的约简方法 J-Reduce[9], J-Reduce 通过构建程序中不同类文件的依赖关系从而对冗余类文件进行约简。在 J-Reduce 的基础上，Kalhauge 等人进一步拓展并提出更细粒度的 Java 字节码程序约简工具，鉴于其为 J-Reduce 的扩展工作，本文以 J-ReduceEXT 指代该拓展方法。J-ReduceEXT 通过构建函数级别的依赖关系对 Java 测试程序进行进一步的约简。然而，函数级别的测试程序约简粒度依然较粗。已有工作[10][11][12]以及本文中的实验结果表明，80%的软件缺陷集中在约 20%的软件代码中，即绝大多数软件缺陷都集中在相对较少的模块和函数内部。尽管 J-ReduceEXT 可以将约简粒度降低至函数体级别，然而函数体中依然存在很多冗余的代码行。例如，现有针对 JVM 测试的测试程序生成工具往往会在同一个函数体内以迭代的方式引入不同程序的代码片段，导致函数体的代码庞大，如图 1 (a)中的 test 函数所示。利用 J-ReduceEXT 在此类测试程序上约简后的测试程序如图 1 (b)所示，从图中可以看出，J-ReduceEXT 仅约简了代码量较少且与缺陷无关的函数，然而可检测缺陷但包含大量代码的 test 函数依然存在，这依然需要开发人员花费大量的时间和程序去分析理解测试程序。

```
1  public class TestCharVect2 {
2    public static void main(String[ ] args){
3        int var1 = test();
4        if (var1 > 0) {
5          System.err.println("FAILED: " + var1 + " errors");
6          System.exit(97);
7        }
8    }
9    public static int test(){
         // hide method content
111      char[] var0 = new char[997];
112      char[] var1 = new char[997];
113      int var0 = test_srlc_add(var0, var1);
         // hide method content
1802   }
1803   public static int test_srlc_add(char[ ] var0, char[ ] var1){
         // hide method content
1904   }
       // hide 61 methods for saving space
2401   public static void verify(String var0, int var1){
         // hide method content
2411   }
2412 }
```

a) 待约简程序

```
1  public class TestCharVect2 {
2    public static void main(String[ ] args){
3        int var1 = test();
4        if (var1 > 0) {
5          System.err.println("FAILED: " + var1 + " errors");
6          System.exit(97);
7        }
8    }
9    public static int test(){
         // hide method content
111      char[] var0 = new char[997];
112      char[] var1 = new char[997];
113      int var0 = test_srlc_add(var0, var1);
         // hide method content
1802   }
1803   public static int test_srlc_add(char[ ] var0, char[ ] var1){
         // hide method content
1904   }
       // hide 61 methods for saving space
       // public static void verify(String var0, int var1){
           // hide method content
       // }
1905 }
```

b) J-ReduceEXT 约简结果

图 1 J-ReduceEXT 约简效果示例

为了进一步约简 Java 测试程序，减轻开发人员分析和理解测试程序的成本，从而更快定位和修复缺陷，本文提出了基于程序约束的细粒度 Java 测试程序约简方法 JavaPruner。具体来说，JavaPruner 首先设计了基于语句块（Block）粒度的代码片段度量方法，从而在约简效果和约简效率之间取得一个平衡。因为考虑太粗的代码度量方式（例如，函数级别），会导致约简的效果较差，约简后的测试程序依然包含大量冗余的测试代码。若考虑太细的代码度量方式（例如，代码行级别），则会导致约简的效率降低，需要构建庞大的程序间依赖关系使得约简的成本太高。随后，JavaPruner 构建不同语句块之间的依赖关系约束，并基于不同语句块之间的约束借助 Delta 调试对语句块进行拆分和删减，如果删减了特定的代码片段后依然能触发原有缺陷则在该约简程序的基础上进行进一步的约简，否则将还原该代码片段并选择其他代码片段进行约简。为了进一步提升测试程序约简效率，本文在 JavaPruner 中引入了未执行代码片段以及冗余函数预处理方法，从而减少约简代码片段的数量，提升约简效率。

与其它约简工作不同，Java 字节码测试程序约简的挑战在于如何保障约简后的测试程序的语法、语义正确性和缺陷检测能力。一方面，Java 程序内通常包含着复杂的语法、语义依赖关系，破坏程序的语法、语义约束容易引入新的缺陷从而导致原有缺陷的丢失。另一方面，本文考虑了现有 JVM 测试工作中往往通过差分测试(Differential Testing)来检测不同 JVM 实现中的缺陷[2][3][13]，其中包含对相同测试程序在不同 JVM 实现上有着不同输出的场景。该场景通常是通过输出测试程序运行过程中的运行状态，然后对比不同 JVM 实现上的输出来判断 JVM 是否存在缺陷，如果某一个 JVM 实现在一个测试程序上与其他 JVM 实现有着不同的输出，则表明该 JVM 实现存在缺陷。然而，现有工作均无法针对该场景的测试程序进行有效的约简。针对上述问题，本文提出了考虑程序语法和语义约束的细粒度约简方法，基于语句块级别的依赖约束关系，借助 Delta 调试来对 Java 字节码测试程序进行约简，在保证程序正确性的同时保障其原有的缺陷检测能力。其次，本文引入差分测试场景，通过在约简过程中跟踪测试程序在不同 JVM 实现上的差异信息来保证测试程序原有的缺陷检测能力，从而对该场景的测试程序进行有效的约简。

为了评估 JavaPruner 的有效性，本文构建了一个高质量基准数据集，该数据集通过运行现有针对 JVM 测试的测试程序生成工具，并在大量可以触发 JVM 缺陷以及输出不一致的测试程序中筛选出一组具有复杂依

赖关系的测试程序。在该数据集上的实验结果表明，JavaPruner 在约简效果上显著优于现有方法 J-ReduceEXT[14]，在保证程序正确性和原有缺陷检测能力的前提下最多可以将 Java 字节码测试程序约简至其原有规模的 1.09%此外，本文将 JavaPruner 约简的测试程序与人工约简的测试程序进行对比分析，发现 JavaPruner 可以有效地提升开发人员约简测试程序，从而定位缺陷位置的效率，定位时间最多缩短了 93.75%。

本文的主要贡献如下：

- 在语句块(Block)级别设计并提出了针对 Java 测试程序约简的细粒度代码片段表示方法，并基于此构建不同代码片段之间的依赖关系约束。
- 提出带程序语法、语义约束的 Delta 调试，通过对测试程序中的语句块进行拆分和删除，并在删除过程中基于程序约束保障测试程序的正确性和缺陷检测能力。
- 针对新的测试场景(差分测试场景)设计了新的约简策略，并基于现有测试程序生成工具构建一组高质量基准数据集。
- 设计并实现了基于程序约束的细粒度 Java 测试程序约简方法 JavaPruner，为了方便其他研究人员重现本文以及在此基础上提出更有效的方法，本文将对 JavaPruner 的源代码以及基准数据集进行共享[15]。

本文剩余内容结构安排如下：第 1 节介绍了测试程序约简的相关工作，包含测试程序约简特别是 Java 测试程序约简的相关工作，并突出本文相对现有工作的创新点。第 2 节介绍基于程序约束的细粒度 Java 测试程序约简方法 JavaPruner 的整体框架和实现细节。第 3 节介绍相关的实验设置以及基准数据集的构建，并通过 3 个实验问题评估、分析本文方法的有效性。第 4 节分析了对本方法实证研究有效性构成的潜在威胁以及相应的应对策略。第 5 节总结了全文的工作并对后续研究工作进行展望。

1 相关工作

本节主要分析现有的测试约简工作，包含测试套件约简以及测试程序约简，随后重点介绍现有针对 Java 测试程序约简的相关工作，最后强调本文的创新性。

1.1 测试约简

软件测试旨在通过运行大量测试程序，观察分析被测试程序的行为和运行结果，从而检测其是否包含潜在的软件缺陷。为了提升测试的充分性往往需要运行大量的测试程序，因此这些测试程序集合（测试套件）往往都很庞大。然而，并非所有测试程序都对检测软件缺陷有贡献，不仅如此，一些冗余和低质量的测试程序还会影响测试效率以及开发人员对缺陷的定位和修复[16]。因此，研究人员针对这类情况设计并提出了测试套件约简技术[17][18][19]。测试套件约简技术（Test-Suite Reduction）通过在原有的测试程序集合上，在保证测试覆盖的前提下，缩简测试程序集的规模，减少冗余测试用例，从而提高测试执行的效率。例如，顾庆等人[20]针对测试用例规模约简提出了选择性回归测试方法 HATS，通过删减无关需求覆盖，从而以最少的测试用例数量来保障软件缺陷检测的有效性。Fraser 等人[21]提出了一套约简测试套件的方法，通过识别基于模型检测技术生成的测试程序中的冗余代码，并对其进行删减，最终对测试程序进行转换从而形成新的测试套件等。

然而，仅对测试套件约简依然存在不足，开发人员在测试套件中定位到揭错的测试程序后往往仍需要花费大量的时间来定位产生缺陷的原因。例如，在编译器的测试工作中，为了向开发人员提交触发编译器缺陷的测试程序，研究人员往往需要花费大量的时间约简测试程序，从而提交一个代码数量较少但依然能触发编译器缺陷的测试程序。为此，研究人员提出了针对测试程序的约简技术。例如，由 Zeller 等人[22]提出的 Delta 调试测试输入约简算法，通过将原始的测试输入拆分为多个非空的子集，对每个子集做删减操作，若相应子集在删除后没有影响原程序的行为，则表明该子集可约简，否则将放回该子集。在 Delta 调试的基础上，Regehr 等人[23]提出了针对 C 编译器的测试程序约简工具 C-Reduce, C-Reduce 通过设计相应的代码转换规则和优化策略来对测试程序进行改写和简化，例如，死代码删除等，最终通过迭代的应用代码转换和优化策略将测试程序约简至最小的测试程序。Misherghi 等人[24]根据输入程序内部结构为启发来指导约简的进行以保证测试程序

的有效性同时提高约简效率，提出了 Hierarchical Delta Debugging 算法(HDD)，并对 XML 测试输入进行约简。

现有针对 C 程序以及 XML 测试输入上的约简已经取得了很好的效果，C-Reduce 通过 C 解释器进行语义检查很好的解决了 C 程序动态无效输入的问题[23]。然而，与现有工作不同，针对 Java 测试程序的约简面临的主要挑战为 Java 程序包含大量的内部依赖关系。例如，Java 复杂的面向对象机制使得一个类往往会依赖于其他类以及众多接口类，同时 Java 程序在编译和字节码验证时需要保证所有的依赖均是完整的，否则会抛出编译错误。因此，现有的工作无法直接迁移至 Java 测试程序的约简工作中。

1.2 Java测试程序约简

由于 Java 测试程序内部依赖的复杂性，现有针对 Java 测试程序约简的工作相对较少。Kalhauge 等人[9]首先提出了针对 Java 测试程序约简方法 J-Reduce，J-Reduce 通过构建 Java 测试程序依赖图，并在类文件(class)级别对测试程序进行约简。J-Reduce 很好的解决了 Java 测试程序类文件之间存在复杂依赖关系的问题，但由于其仅在类文件级别进行约简，导致约简后的测试程序依然很大。随后，Kalhauge 等人[14]在此基础上进一步提出了更细粒度的 Java 字节码程序约简工具 J-ReduceEXT，J-ReduceEXT 将测试程序中的依赖进一步细化为函数体之间的依赖，通过识别并删除冗余的函数体来达到进一步约简测试程序的信息。与此同时，为了验证 J-ReduceEXT 的完备性，Kalhauge 等人在 Featherweight Java 上[25]完成了相关的程序验证工作，证明了 J-ReduceEXT 的完备性。

尽管 J-ReduceEXT 将 Java 测试程序约简的粒度进一步降低为函数体级别，但经其约简的测试程序依然存在包含大量代码的函数体，且触发程序缺陷的代码往往存在于函数体内部。因此，仅在函数体级别对 Java 测试程序进行约简依然是不充分的，开发人员仍需要花费大量的时间来分析和理解约简后的测试程序。同时，在针对 Java 虚拟机测试的场景中，往往需要借助差分测试作为测试预言，即在不同的 JVM 实现上运行相同的测试程序，若某个 JVM 实现的输出结果与其他实现不一致，则该 JVM 实现可能存在缺陷。然而，现有工作无法针对此类致错场景的测试程序进行有效的约简。其主要原因在于现有工作均是对包含程序异常或崩溃的测试程序进行约简。然而在 JVM 测试中通常需要借助差分测试来检测因 JVM 缺陷导致的程序执行结果不一致问题，例如算术运算结果不一致，此类缺陷无法依赖测试程序本身的异常或崩溃来暴露。因此现有工作无法有效的对该场景下的测试程序进行有效的约简。本文首次引入了差分测试为测试场景，并为其设计了更细粒度的约简方法。

1.3 创新性分析

为了解决现有工作中存在的问题，本文首次在语句块级别设计了更细粒度的代码片段表示方式，在该级别进行 Java 测试程序的约简可以使约简的粒度更细，从而直接删除与缺陷无关的冗余代码，进一步提升约简的效果。为了解决更细粒度的约简方式会来带更复杂依赖分析的问题，本文首次提出了带程序语法、语义约束的 Delta 调试算法来保障约简测试程序的正确性以及缺陷检测能力。同时，本文对差分测试场景进行了适配，使得本文方法可以应用至 JVM 测试中导致不同 JVM 实现输出不一致的测试程序约简场景中。最后，为了评估本方法的有效性，在本文基于现有 Java 测试程序生成工具构建了一套基准数据集上。在该基准数据集上的实证研究表明，本文所提方法显著优于现有约简方法。

2 方法介绍

本文提出的方法 JavaPruner 通过识别 Java 测试程序中语句块级别的代码片段，并基于代码片段之间的依赖关系构建程序约束，最后借助带约束的 Delta 调试算法来对 Java 测试程序进行细粒度的约简。

JavaPruner 方法的整体流程图如图 2 所示。JavaPruner 针对 JVM 测试中可以触发 JVM 缺陷或不同 JVM 实现输出不一致的测试程序进行约简。考虑到复杂的测试程序中往往包含大量的测试代码，会导致提取到的代码片段数量非常庞大，导致构建约束以及约简的成本过高。为此，本文在提取代码片段前会对该测试程序进行预处理。JavaPruner 首先基于本文设计的 5 种代码片段类型提取测试程序中的代码片段，并删除测试程序中未执行的代码片段以及与缺陷无关的函数体，从而约简分析代码片段的数量，提升约简效率。随后，

JavaPruner 基于对剩下被执行的代码片段进行依赖关系分析。将代码片段集合与其依赖关系作为输入执行带约束的 Delta 调试算法，进行代码片段粒度的约简以及缺陷检测能力的检查。若在约简过程中，其缺陷检测能力丢失，则会进行代码片段的回溯；否则将在约简后的测试程序上进行迭代的约简，直至所有代码片段均无法被约简或满足终止条件。最终，输出约简后的测试程序，该约简程序可以用于提交给开发人员帮助其减少分析理解成本，快速定位缺陷并修复。接下来，本节将介绍 JavaPruner 各个部分的具体实现细节。

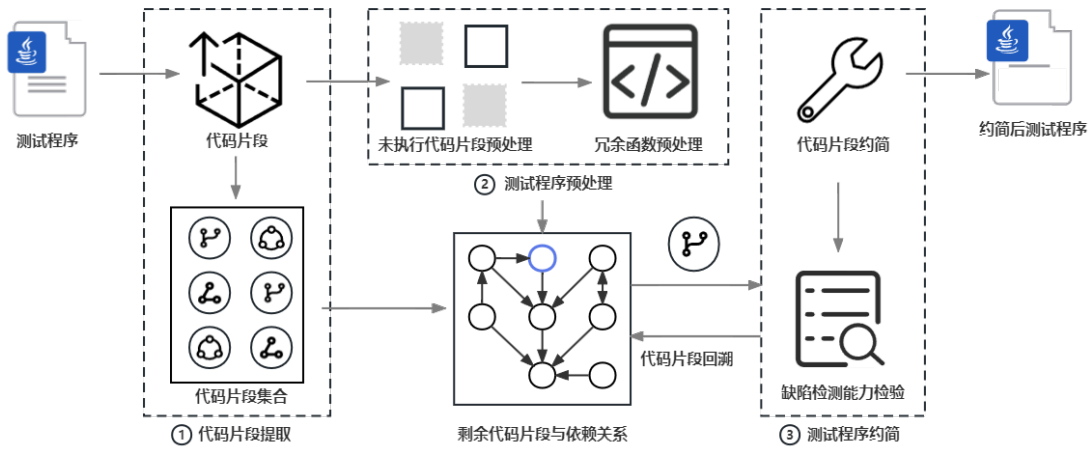


图 2　JavaPruner 方法整体流程图

2.1 代码片段的表示与提取

2.1.1 代码片段的表示

代码片段的表示方式会影响到约简工作的有效性和效率。因此，本文中的代码片段表示方式需要同时满足以下三个要求：(1) 基于该表示方式约简后的测试程序需具有足够低的复杂度和足够小的规模；(2) 基于该表示方式需具有较高的效率，即可以在合理的时间内生成约简后的测试程序；(3) 基于该表示方式需能够保证约简后测试程序的正确性以及缺陷检测能力。为此，本文从语句块粒度设计了 5 种代码片段表示方式用于测试程序约简:

- 顺序代码片段 (Sequential Code Snippet, SEQ).
- 分支代码片段 (If Code Snippet, IF).
- 循环代码片段 (Loop Code Snippet, LOOP).
- Switch 代码片段 (Switch Code Snippet, SWITCH).
- 异常代码片段 (Try-Catch Code Snippet, TRAP).

其中，顺序代码片段为一组不包含任何分支结构的顺序指令序列，例如，变量的声明与赋值、函数调用以及输入输出等操作；分支代码片段为包含分支的语句结构，例如，if, if-else 等。分支代码片段中不仅需要包含分支语句的条件语句，同样需要不同分支的代码体；循环代码片段可以为 for、while、do-while 语法结构，该代码片段包含循环语句中相应的循环条件以及循环体；Switch 代码片段为常见的分支结构，用于处理多个条件判断，该代码片段包含 switch 语句中所有的分支条件以及对应的代码体；Try-Catch 代码片段为 try-catch 语法结构，包含了 try 代码体以及用于处理捕获到异常的代码体。

基于该代码片段粒度的表示方式首先有助于在约简效率和约简效果上取得一个平衡，即满足三个要求中第(1)和第(2)点。一方面，代码片段的粒度相比现有函数级别的粒度，可以直接对测试代码进行约简，这些代码中往往包含大量与缺陷无关的代码，因此可以使约简后的测试程序降低到更小的规模。另一方面，代码片段的粒度相比于更细粒度的代码行级别可以取得更高的约简效率。因为构建代码行的约束关系具有一定的复

杂性，同时基于代码行进行约简需要重复运行大量测试程序来保障其缺陷检测能力，导致约简效率的降低。相比之下，基于代码片段进行约简可以减少约简以及重复执行测试程序的次数。其次，基于代码片段粒度的表示方式同样能够满足三个要求中的第(3)点。一方面，该代码片段基于 Java 语法结构所设计，在保证其语法正确性的同时保障其语义的正确性，例如，变量的定义以及使用作用域等问题，从而保障约简后测试程序的正确性。另一方面，触发程序缺陷的代码片段往往仅集中在少量的代码片段中，即可以通过一个或多个代码片段的组合来触发相应的缺陷。

综上所述，基于该代码片段粒度的表示方式首先充分涵盖了 Java 测试程序中的语法特征，同时该表示方式可以迁移至其它相似编程语言场景中，具有一定的通用性。其次，基于该代码片段粒度的表示方式很好的满足了在 JVM 测试程序约简工作中的要求，在进一步提升约简效果的同时有效保证了约简效率。

2.1.2 代码片段的提取

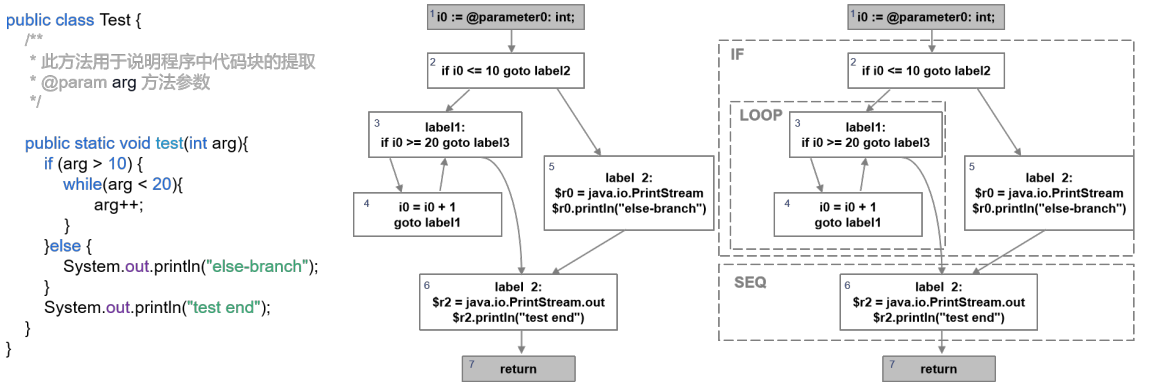
本文基于 Java 程序分析工具 Soot[26]来实现 JavaPruner 。Soot 是一款优秀的 Java 类文件分析工具，可用于静态分析、编译优化以及程序转换等领域[27][28][29]。因此，JavaPruner 在 Java 字节码文件（对应 Soot 中的 Jimple 指令）的基础上对 Java 测试程序进行约简而不是从源代码的级别进行约简。其主要原因在于：(1) Soot 中封装了大量对 Jimple 指令代码的操作[30]，且 Jimple 指令可以很好的涵盖本文中所设计的 5 种类型的代码片段，有利于 JavaPruner 的实现。(2) 所有的 Java 源程序均可以编译成字节码文件，且 Java 虚拟机支持编译和运行所有基于 Java 字节码的语言[31][32][33][34]，例如，Kotlin、Scala 等，均可以直接使用本工具进行测试程序的约简，其使用范围更广。为了对代码片段进行提取，JavaPruner 需要将测试程序转换为一个控制流图（Control Flow Graph, 简称 CFG）。控制流图常用于描述程序的控制流程或执行流程的有向图[35]，广泛应用于程序静态分析中[36][37],图中的节点代表程序中的基本块（Basic Block），每个基本块包含一组连续的指令序列，没有分支或跳转语句。图中的边表示控制流的转移关系，即执行时基本块之间的跳转关系，包含顺序执行、条件分支以及循环迭代等。本文定义的代码片段由一个或多个基本块所组成。

给定一个 Java 测试程序 P 作为输入，JavaPruner 首先将其转换为一个控制流图$CFG = < B, E >$，其中$B = \{b_0, b_1, \ldots, b_n\}$为该控制流图中所有基本块集合，$E = \{e_0, e_1, \ldots, e_n\}$为不同基本块之间的有向边集合。$e(b_i, b_j)$表示为基本块$b_i$指向$b_j$的有向边，因此，一个代码片段$s_0$应由一组基本块集合$B_0$以及其对应的有向边集合$E_0$所组成，记为$s_0 = < B_0, E_0 >$，其中$B_0 \subset B, E_0 \subset E$。一个测试程序所有的代码片段集合则记为$S = \{s_0, s_1, \ldots, s_n\}$。为了提取所有的代码片段，JavaPruner 随后从控制流图的根节点开始根据每个基本块中的指令类型来判断其是否是本文定义的后四种代码片段类型，因为顺序语句块没有明确的指令标识，而其他代码片段均包含相应的指令。例如，switch 指令在 Jimple 代码中被表示为 lookupswitch 以及 tableswitch。在分析基本块的时候，如果基本块b_i的指令中包含 lookupswich 或者 tableswitch，则该基本块即被视为一个 Switch 代码片段的起始节点，随后，JavaPruner 根据基本块之间的连接关系搜索该 switch 指令所有条件分支对应的基本块$\{b_{i+1}, b_{i+2}, \ldots, b_{i+n}\}$，最后将这些基本块集合$B_i = \{b_i, b_{i+1}, b_{i+2}, \ldots, b_{i+n}\}$以及其所有的有向边集合$E_i = \{e_i, e_{i+1}, e_{i+2}, \ldots, e_{i+n}\}$合并成一个 Switch 代码片段$s_i = < B_i, E_i >$，并将其放入代码片段集合 S 中。

需要注意的是，对与分支代码片段以及循环代码片段的提取需要进行特殊处理。在 Soot 中，循环结构被表示为一个 if 和 goto 指令的组合，因此，如果一个基本块b_j中包含 if 指令，则很难确定其为分支代码片段的起始节点还是循环代码片段的起始节点。因此，JavaPruner 需要进一步对b_j的后继基本块进行检查，分析其是否包含 goto 指令，如果后继节点中包含 goto 指令，且 goto 指令的目标基本块为b_j，则b_j将被识别为循环代码片段的起始节点，否则b_j将被识别为分支代码片段的起点。对于无法被识别为后四种代码片段的连续基本块，JavaPruner 则将其识别成一个顺序代码片段。需要注意的是，不同的代码片段之间具有包含关系，例如，条件代码片段的代码体可以包含任何类型的代码片段。因此，在识别出后四种代码片段之后，JavaPruner 会递归的对该代码片段内部包含的其它代码片段进行搜索，并将其添加至代码片段集合 S 中。

随后，我们基于一个实例，详细说明 JavaPruner 的提取过程。图 3 (a)为一个简单的 Java 程序示例，程序中为一个 if-else 分支包含 while 循环的代码结构。JavaPruner 首先使用 Soot 工具将其转化为 Jimple 中间代码并做静态分析，得到如图 3 (b)所示的程序控制流图，单个节点表示一个基本块。分析 2 号基本块，其包含两

个分支且两个分支于 6 号基本块汇合，因此该基本块为分支代码片段起始节点。同时 3 号基本块同样包含 if 语句，然而其后继节点 4 号基本块包含一条指向 3 号基本块的 goto 语句，因此 3 号基本块被识别为循环代码片段的起始节点。6 号基本块无法识别为后四种代码片段，因此识别为顺序代码片段。该测试程序的代码片段提取结果如图 3 (c)所示，各代码片段对应基本快集合为，分支代码片段 IF: {2,3,4,5}；循环代码片段 LOOP：{3,4}；顺序代码片段 SEQ：{4}, {5}, {6}。其中，由于顺序代码片段数量较多，图 3 (c)只标出一个作为示例。



a) Java 程序示例　　b) Jimple 代码控制流图　　c) 代码片段提取结果

图 3　代码片段提取示例

2.2 测试程序预处理

2.2.1 缺陷检测能力验证

为了提升约简效率，JavaPruner 在对测试程序约简前会对 Java 测试程序进行预处理，删除未执行代码片段以及冗余函数体。考虑到测试程序的预处理同样有可能导致测试程序缺陷检测能力的丢失，因此，在介绍测试程序预处理之前，本节将对如何验证测试程序是否保有缺陷检测能力进行介绍。

为了保证原有测试程序的缺陷检测能力，JavaPruner 在约简测试程序时会对测试程序进行验证，判断其致错原因是否依然存在。对于程序的致错原因，考虑到本文的研究目标不是缺陷定位，因此，本文仅基于测试程序致错表现是否丢失来判断测试程序的致错原因是否存在。因此定义两个函数来对测试程序进行执行以及执行结果的比较：（1）定义函数$O = Execute(P)$为测试程序执行函数，其中O为测试程序P的执行结果；考虑到程序执行往往包含不同的输出形式，例如，程序不正确执行导致的程序异常或程序崩溃。因此，执行结果的类型包含异常/崩溃(Exception/Crash)以及正常的程序输出(Output)。为此，执行结果 O 的定义为 $O = < Failures, Output >$，其中 $Failures$ 为程序异常/崩溃时的根因（例如，NullPointerException 或 SegmentationFault），$Output$ 则为程序正常执行结果的输出信息，包含控制台打印信息，输出的文件等。（2）定义函数$R = Compare(O_1, O_2)$为一致性比较函数，其中 R 为一致性比较结果，O_1为约简前测试程序的输出，O_2为约简后测试程序的输出；若返回值 R 为 $True$ 则表明约简前后结果一致；否则为不一致，即约简后测试程序缺陷检测能力丢失。具体说来，对于测试程序 P，其执行结果$O = Execute(P)$。在进行测试程序预处理以及后续的约简操作后得到约简后的测试程序P'，P'的执行结果$O' = \text{Execute}(P')$。如果 Compare(O, O')的返回值为 $True$ 则表明此次约简操作是正确的，JavaPruner 将在约简后的测试程序P'上进行更进一步的约简。否则，JavaPruner 将会对约简操作进行回溯，将P'还原为测试程序 P，并选择其它代码片段进行约简。

针对在差分测试的场景中，通过对比在不同 JVM 实现上执行同一个测试程序的输出来判定其是否会存在输出结果不一致的情况。因为所有的 JVM 实现都应符合 Java 虚拟机规范，因此如果不同的 JVM 实现在同一个测试程序上的输出不同，则表明其中一个 JVM 实现存在缺陷。具体来说，在该场景下，对于输入的测试程序 P，JavaPruner 首先通过$O = Execute(P)$函数在不同的 JVM 实现上进行执行。例如，在 Hotspot 虚拟机[38]上的执行结果为$O_{hotspot} = Execute(P)$，在 Openj9 虚拟机[39]上的执行结果为$O_{openj9} = Execute(P)$。若

$Compare(O_{hotspot}, O_{openj9})$返回为 $False$，则表明 Hotspot 和 Openj9 在测试程序 P 上的表现不一致，即其中一个 JVM 的实现可能存在潜在的缺陷。为此，测试人员需要对该测试程序进行约简，并提交给 JVM 的开发人员让其判定 JVM 的实现是否存在缺陷。因此，对于约简后的测试程序P'，对于不同 JVM 实现的执行结果 $O'_{hotspot}$、O'_{openj9}， 若$Compare(O'_{hotspot}, O'_{openj9})$的返回值依然为 $False$，则表明约简后测试程序缺陷检测能力依然存在，JavaPruner 将在约简后的测试程序上进行进一步的约简。

2.2.2 未执行代码片段预处理

大量冗余的未执行代码片段会增加约简时测试程序的验证成本，因此需要对其进行预处理。接下来，本小节将对未执行代码片段进行定义。对于测试程序 P，其所有代码片段集合为 S。集合 S 中包含了执行代码片段集合$ES = \{es_0, es_1, \ldots, es_n\}$， 以及未执行代码片段集合$NS = \{ns_0, ns_1, \ldots, ns_n\}$。根据定义，代码片段集合以及其对应的测试程序应满足以下性质：

$$ES \cup NS = S, ES \cap NS = \varnothing \tag{1}$$

$$Compare(Execute(S), Execute(ES \cup NS)) = True \tag{2}$$

根据上述性质，在对未执行代码片段进行处理时，若执行代码片段集合 ES 满足以下性质，则表明该未执行代码片段集合 NS 可以被有效的删除。

$$Compare(Execute(S), Execute(ES)) = True \tag{3}$$

需要注意的是，因为不同的 JVM 实现或其他类似的编译器中都包含对未执行代码区域的优化，例如，死代码优化。这些优化中同样有可能存在导致 JVM、编译器出错的缺陷。因此，如果在删除未知行代码片段后测试程序原有的缺陷检测能力丢失，则表明缺陷原因有可能与死代码优化有关。对于这种情况，JavaPruner 将放弃进行未执行代码片段预处理，直接进行后续的操作。在具体实现时，JavaPruner 通过程序插桩(Instrumentation)以及动态执行来实现未执行的代码片段的标记与识别。具体说来，对于所有提取到的代码片段，JavaPruner 首先在每个代码片段的开始插入标记代码。随后，JavaPruner 运行插桩后的测试程序，并根据插入的标记代码输出来识别未执行的代码片段。

2.2.3 冗余函数预处理

根据已有研究发现[14]，冗余函数同样会影响约简的效果和效率，同时这些函数中通常也会包含大量的代码片段。如果特定的函数被证明是冗余的，则表明该函数内所有的代码片段均可以被删除。同理，冗余函数的预处理同样需要保证测试程序原有的缺陷检测能力。对于测试程序 P，其所有声明函数集合为 M。集合 M 中包含了有效函数集合$MV = \{mv_0, mv_1, \ldots, mv_n\}$以及冗余函数集合$MR = \{mr_0, mr_1, \ldots, mr_n\}$。若有效函数集合 MV 以及冗余函数集合 MR 同时满足以下性质，则表明可以对冗余函数集合进行删除。

$$MV \cup MR = M, MV \cap ME = \varnothing \tag{4}$$

$$Compare(Execute(M), Execute(MV)) = True \tag{5}$$

为了对冗余函数进行处理，JavaPruner 针对每个测试程序构建其类内函数调用图(Call Graph，简称 CG)，函数调用图是描述程序中函数间调用关系的图形表示方法。其中节点表示函数，有向边表示函数之间的调用关系。图中的节点代表函数的定义或声明，而边表示一个函数调用另一个函数。如果函数 A 调用了函数 B，则在函数调用图中会有一条从节点 A 指向节点 B 的有向边。需要注意的是，JavaPruner 仅对类内的函数进行预处理，对于调用其它类或第三方 Jar 包的函数不做约简操作。

算法 1 展示了冗余函数的处理算法。对于输入的测试程序 P，JavaPruner 首先会获取 P 的所有函数集合 M 以及其对应的函数调用图 CG（行 1-2）。随后，对于集合 M 中函数，JavaPruner 将从起始节点开始，尝试依次删除 M 中的函数。对于不出现在函数调用图中的函数，即图中的孤立节点，JavaPruner 将直接对其进行删除；对于其它函数，JavaPruner 将会删除对应的函数以及调用该函数的语句（行 4-8）。同时，考虑到函数往往包含返回值，若直接删除对应的函数调用则会导致程序语义约束被破坏，例如,原有被函数返回值赋值的变量未定义或未初始化。为了保证测试程序语义的正确性，JavaPruner 会对被破坏的语义约束进行修复（行 8）。

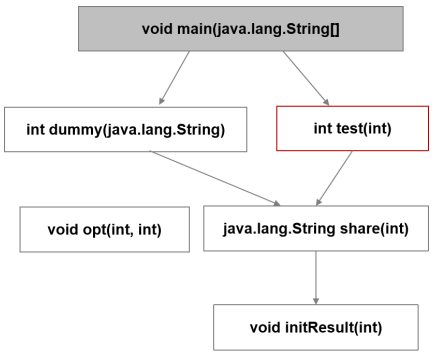
具体说来，JavaPruner 会根据函数返回值类型优先在当前类文件中寻找类型兼容的变量，并将其赋值给对应变量。若当前类文件中没有类型兼容的变量，JavaPruner 则会为相应的变量创建新的初始化语句。随后，JavaPruner 通过比较函数删减前后测试程序的输出来判断其缺陷检测能力是否存在，若存在，则将用P'覆盖 P 并在 P 的基础上删除其他冗余函数。最终该算法将返回删除冗余函数后的测试程序P'来进行后续的约简工作。

Algorithm 1: 冗余函数预处理

Require: 测试程序 P

1: $M \leftarrow$ identify all declared functions in P;
2: $CG \leftarrow$ *construct call graph od* P;
3: **for** m in M **do**
4: **if** $m \notin CG$ **then**
5: $P' \leftarrow P$ removes m;
6: **else**
7: $P' \leftarrow P$ removes m *and* the correspind invocation;
8: $P' \leftarrow P'$ fixed broken constraints;
9: **if** $Compare(Execute(P), Execute(P')) = True$ **then**
10: $P \leftarrow P'$;
11: **end if**
12: **end for**
13: **return** P

接下来，本小节将借助一个示例来展示具体的冗余函数预处理过程。如图 4 (a)所示，其中，main 函数为函数调用图的起始节点，节点 opt 为孤立节点。图 4 (b)展示了一个包含冗余函数调用的代码片段。在这个代码片段中，函数 dummy 的返回值被赋值给变量 dr。为了对冗余函数进行约简，JavaPruner 删除了函数 dummy 的定义，对于函数返回值赋值的变量 dr，鉴于程序上下文中并没有与变量 dr 类型兼容的变量，JavaPruner 直接通过随机初始化的方式为其赋一个新值。约简后的测试程序如图 4 (c)所示，由于该函数为冗余函数，约简后的测试程序依然能触发原有的缺陷，因此，JavaPruner 将在该测试程序的基础上进行进一步的约简。若约简后的测试程序无法触发原有的缺陷，则该函数将被还原，并选择其他函数进行约简。




```
public static void main(String[] args){
   int tr = test(2);
   int dr = dummy("dummy");
    // hide method content
}

public static int test(int a){
   share(a);
   // hide method content
   // this method would trigger error
   return test_result;
}

// hide some methods

public static int dummy(String str){
   share(str.length());
   // hide method content
   return dummy_result;
}
```

```
public static void main(String[] args){
   int tr = test(2);
   int dr = 2;
    // hide method content
}

public static int test(int a){
   share(a);
   // hide method content
   // this method would trigger error
   return test_result;
}

// hide some methods
```

a) 函数调用图　　b) 包含冗余函数测试程序　　c) 冗余函数约简结果

图 4　冗余函数预处理示例

2.3 测试程序约简

在测试程序预处理之后，JavaPruner 将对剩下的代码片段进行进一步的约简，直到测试程序中仅包含与

缺陷表现相关的代码片段。如前文所说，Java 测试程序约简工作中的最大挑战就是要处理程序内部复杂的依赖关系，为了解决这个挑战，现有工作 J-Reduce 通过构建项目内文件之间的依赖关系对 Java 测试程序进行约简。在 J-Reduce 的基础上，研究人员进一步通过构建函数间依赖关系来从函数的粒度上对测试程序进行约简。尽管现有工作部分解决了 Java 测试程序中内部依赖复杂的问题，但现有工作的约简粒度太粗，导致约简后的测试程序依然需要开发人员分析大量的源代码来定位到实际缺陷。本文引入了细粒度的代码片段度量方式，并在此基础上对其进行约简，从而取得更好的约简效率。然而，更细粒度的约简方式同样会引入更复杂的依赖关系，为此，本文引入了带程序约束的 Delta 调试约简方法。这里的程序约束指程序的语法和语义约束，例如，删减的代码片段必须符合语法规范同时在删减的过程中需保证程序语义的正确性。接下来，本小节将介绍如何基于程序的语法语义对测试程序进行约简。

2.3.1 程序约束

本文程序的正确性，主要是指程序语法、语义的正确性。为了满足程序语法、语义的正确性，则需满足正确程序语法、语义间的约束。具体来说，本文中语法的正确性保证主要为语法结构约束以及变量类型约束保证。语法结构约束包括条件语句、循环语句以及控制流语句（如 break 语句、return 语句）等。得益于本文设计的代码表示方式是基于程序语法结构的角度进行设计因此可以很好的保证其删减过程满足语法结构约束，从而保证语法结构的正确性。变量类型约束则是为了保障测试程序约简过程中，对涉及到变量类型匹配的操作，保证匹配的变量类型一定是相同的或类型兼容的。例如，为缺少初始化变量寻找类型相同或类型兼容的变量进行初始化操作。

语义的正确性保证则主要需要保证程序依赖约束以及变量声明约束。程序依赖约束为了保障测试程序在约简过程中原有的依赖结构不会发生改变，从而改变代码片段原有的程序逻辑。例如，在删除包含循环嵌套的代码片段中，若未保证原有的依赖关系则会导致循环体的改动甚至产生死循环，从而约简出与原始程序语义不同的约简程序。变量声明约束则是为了保障测试程序所有被使用到的变量均是被声明且初始化的。因为在程序约简过程中存在删减的代码片段中包含相关变量的声明，且这些变量被后续的代码片段所引用。因此，为了保证程序语法正确性而不在编译时抛出编译错误，需要保障所有被依赖的变量均不被约简。同时，变量声明约束需要保证所有变量均被初始化，从而防止约简测试程序因为变量 null 值而触发与缺陷无关的异常。例如，未进行初始化的引用类型变量值为 null，若测试程序包含对引用类型变量函数的调用则会触发空指针异常。

2.3.2 带程序约束的 Delta 调试约简方法

在本文定义的代码片段粒度进行测试程序约简时，传统的 Delta 调试算法无法处理不同代码片段之间复杂的依赖关系，因此需要在原有 Delta 调试的基础上添加程序依赖关系为约简约束以保证约简后测试程序的正确性。为了提取代码判断之间的依赖关系，JavaPruner 通过构建测试程序的数据流图(Data Flow Diagram，简称 DFD)，来获取不同代码片段中变量的 DEF-USE 集合。具体说来，对于代码片段s_i， s_i中所定义的变量为集合为DEF_{si}，s_i中所引用的变量集合为USE_{si}。若存在代码片段s_j以及变量 v，使得$v \in DEF_{si}$且$v \in USE_{sj}$，则表明代码片段s_j在变量上依赖于代码片段s_i。在对代码片段s_i进行约简时，则需要保留 v 对应的变量声明以及初始化语句。JavaPruner 通过定义方法 (s_i)来获取所有依赖代码片段 s 的代码片段集合，定义方法 $Vars(s)$来获取所有被依赖的变量集合。例如，在当前的例子中，$(s_i) = \{s_j\}, Vars(s_i) = \{v\}$。

加上程序约束的 Delta 调试约简方法流程如算法 2 所示。与传统 Delta 调试算法不同的是，在约简每个代码片段时，本方法均会分析代码片段的依赖关系，在保障测试程序依赖关系的同时，修复其它删减过程中被破坏的约束，例如，对于未初始化变量，JavaPruner 会对其进行初始化以满足变量声明约束。对于输入的测试程序 P 以及预处理后的代码片段集合 ES，JavaPruner 首先会对代码片段集合 ES 进行逆序处理如算法 2 第 1 行所示。其主要原因在于，Java 测试程序一般为顺序程序，其变量的依赖关系一般表现为后定义的变量依赖于先定义的变量，且控制流图中末端的基本块通常会存在大量引用初始基本块中定义的变量。如果从初识语句块进行约简则会导致在约简的过程中需要分析大量被依赖的代码片段，且会导致大量变量声明语句无

法被约简。如果从末端代码语句块进行约简，则可以尽可能的删除与初始语句块中定义变量的依赖关系。

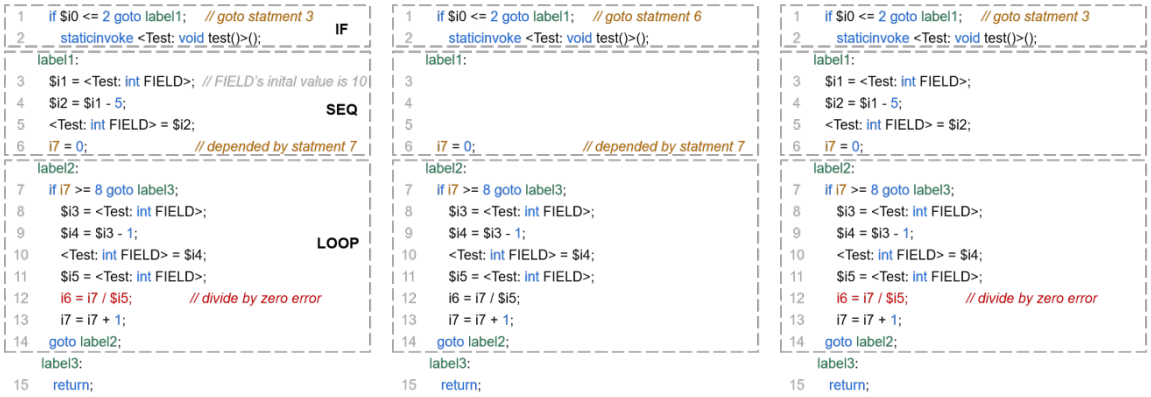
在逆序集合的基础上，JavaPruner 将遍历集合中所有的代码片段。对于集合ES'中的每个代码片段es，JavaPruner 首先通过 (es)函数判断是否存在依赖该代码片段的其他代码片段集合，如果 (es)的返回值不为空，则表明存在其他代码片段依赖代码片段es中的变量。JavaPruner 将会通过$Vars(es)$获取被依赖的变量，并在代码片段es中删除其声明语句（行 3-5）。随后，JavaPruner 将代码片段es从测试程序 P 中删除，并基于本文定义的程序约束修复删除过程中被破坏的程序约束。接着，JavaPruner 将修复后的测试程序P'的执行结果与原始测试程序 P 的执行结果进行对比，如果两个程序的执行结果一致，则表明约简成功，JavaPruner 则用P'覆盖 P，并在其基础上进行进一步的约简。否则，则对P'进行回溯，将删除的代码片段重新添加至P'，并赋值给 P（6-12 行）。需要注意的是，在不满足约简条件时，JavaPruner 是使用回溯的方式来还原代码片段而不是用原始程序 P 覆盖P'。其主要原因在于预处理后的代码片段集合 ES 是在 P 的基础上获得的，若用覆盖的方式则需要重新对其进行加载以及重新识别预处理后的代码片段集合，从而导致约简效率的降低。最后，在所有的代码片段遍历完成之后，JavaPruner 便可以返回约简后的测试程序 P。

Algorithm 2: 带程序约束的 Delta 调试

Require: 测试程序 P；预处理后代码片段集合 ES

1: $ES' \leftarrow$ reverse ES;
2: **for** es in ES' **do**
3: **if** $Depends(es) \neq \emptyset$ **then**
4: $es \leftarrow es$ removes declared variables in $Vars(es)$;
5: **end if**
6: $P' \leftarrow P$ removes es;
7: $P' \leftarrow P'$ fixed broken constraints;
8: **if** $Compare(Execute(P), Execute(P')) = True$ **then**
9: $P \leftarrow P'$;
10: **else**
11: $P \leftarrow P'$ adds es;
12: **end if**
13: **end for**
14: **return** P

接下来，本小节通过一个示例来进一步说明 JavaPruner 的约简过程。图 5(a)为一个 jimple 代码片段，其在第 12 行会触发除 0 错误。该代码片段由一个 IF 代码片段，一个顺序代码片段和一个循环代码片段组成，所使用的全局变量 FIELD 初始值为 10。其基本语义是在循环语句中不断对 FIELD 进行减法和除法运算，最终出现除数为零的情况，造成程序异常。该测试程序中主要的语义语法依赖约束主要有两个，首先，第 3 行语句是第 1 行 goto 语句的目标语句，所以有第 1 行的 goto 语句依赖于第 3 行语句。其次，第 7 行语句使用了 i7 变量，而 i7 变量于第 6 行语句进行了初始化，所以有第 7 行语句依赖于第 6 行语句。约简过程中，JavaPruner 将按照 LOOP，SEQ，IF 的顺序自下而上约简。由于 LOOP 中触发了除 0 错误，其在约简过程中会被回溯保留。我们以约简 SEQ 为例，通过遍历依赖关系可以得到$LOOP \in Depends(SEQ), i7 \in Vars(SEQ)$，因此在约简过程中将第 6 行过滤，删除除第 6 行以外的 SEQ 语句。由于将第 1 行 goto 语句的目标语句删除，为了保证程序结构正确性，我们将目标修改为第 6 行语句以修复该程序约束。SEQ 语句块的删除结果如图 5(b)所示。对约简后的代码片段进行执行验证后可得除 0 错误丢失，我们需要进行对代码片段回溯，包括代码语句的回溯和依赖约束的回溯。图 5}(c)为回溯结果，将删除的语句插入代码片段，并将 goto 语句目标修改为原目标语句。



a）待约简代码片段　　b）删除无关语句并修复约束　　c）回溯代码片段

图 5　带程序约束的 Delta 调试约简示例

3　实验设计与结果分析

本节利用真实的缺陷程序对本文设计的约简策略和约简工具进行实证研究，来检验本文方法和工具的有效性。

3.1 实验问题

为了验证本文提出的基于程序约束的细粒度 Java 测试程序约简方法的有效性，本文设计了以下三个实验问题并分析这些问题的设计动机。

RQ1：相比于现有工作，本文提出的 JavaPruner 方法是否能取得更显著的约简效果？

针对现有 Java 测试程序约简工作的约简粒度依然较粗，约简后测试程序理解成本依然较高的问题。本文提出了基于程序约束的细粒度 Java 测试程序约简方法 JavaPruner。JavaPruner 从语句块级别设计了更细粒度的代码片段度量方式来用于测试程序约简，并针对细化的代码片段设计了相应的程序约束作为 Delta 调试的输入，保障约简测试程序的正确性，有效解决了 Java 测试程序约简工作中依赖复杂的问题。同时，本文首次引入针对差分测试场景的测试程序约简问题。差分测试是被广泛使用的一种测试方法，有多种应用场景如编译器测试[16][40]、形式化验证[41]等。为了分析本方法的有效性，本文设计了该实验问题来分析 JavaPruner 是否能比最新的研究工作 J-ReduceEXT[14]取得更显著的约简效果。

RQ2：本文设计的测试程序预处理方法能否有效提升测试程序约简的效率？

鉴于本文基于代码片段进行测试程序约简，同时使用的算法为带约束的 Delta 调试，在约简的过程中对于每个代码片段都需要进行缺陷检测能力验证操作。然而，Java 测试程序复杂多变，其内部解析出的代码片段数量庞大，从而导致在约简过程中进行高频次的缺陷检测能力验证操作，从而严重影响测试程序约简的效率。测试程序预处理的目的则是通过删除未执行代码片段以及冗余函数的操作，减少待分析的代码片段数量，从而减小缺陷检测能力验证频次，提高约简效率。为此，本文设计本实验，将对有预处理和无预处理的效率情况进行对比分析，判断预处理工作是否能有效提升约简效率。

RQ3: JavaPruner 是否有助于提升开发人员的缺陷定位效率？

本文主要研究目的是为了减轻开发人员分析、定位程序缺陷的工作量，提升整体的工作效率。为此，本文将对约简后的测试程序进行人工分析，通过和人工约简的效果效率进行对比，判断本文所用约简策略的有效性和实用性，以评估其是否能够满足当前测试工作的要求。

3.2 实验设置

3.2.1 数据集

为了收集真实测试场景中产生的测试程序，本文通过现有工作 JavaTailor[4]和 VECT[5]来进行数据集的构建。JavaTailor 和 VECT 是当前最新的 JVM 测试工作，其生成的测试程序为 JVM 开源社区提供了数十个真实的 JVM 缺陷。基于这两个测试工具，本文一共整理了 50 个 Java 测试程序作为本文实验的数据集。这些测试程序的选择标准包括: (1)该测试程序能否触发 JVM 异常与缺陷；(2)该测试程序是否拥有较高的程序复杂度。具体来说，本文从两个维度衡量测试程序的复杂度：（1）基于指令数量衡量测试程序的程序规模S；（2）基于圈复杂度(Cyclomatic Complexity，CC)[42]衡量测试程序的程序逻辑复杂度C。测试程序P的复杂度$Comp(P)$则表示为$Comp(P) = 0.5 \times S + 0.5 \times C$。圈复杂度是一种衡量程序结构复杂性的指标，用于评估程序中控制流程的复杂性，圈复杂度越高，程序的控制流程越复杂，人工分析约简的难度也会相应增加。本文统计分析了 Vect[5]从 JVM 仓库中挖掘的测试程序，并基于其圈复杂度以及程序规模计算复杂度。最终计算得出所有测试程序的平均复杂度为 56.17，本文以此来作为开发人员对测试程序复杂度的接受程度。其次，本文对现有 JVM 测试工作中借助程序变异和合成方式所生成程序的复杂度进行分析，若其复杂度大于历史测试程序的平均复杂度，则表明该测试程序的复杂度较高。

表 1 展示了这 50 个程序的基本信息，其中索引为测试程序的编号；名称为测试程序的 Java 类名；指令数量为测试程序包含的 Jimple 代码行数量；语句块数量表示测试程序按照上述的拆分规则拆分出的语句块数量；函数数量表示测试程序中定义的函数数量，其中不包括来自其父类的函数；程序复杂度表示该测试程序相应的复杂度取值，其取值越高表明测试程序的复杂度越高。致错类型包含程序异常/崩溃(Exception/Crash)和程序输出不一致(Inconsistent Output)两种；来源则表示生成测试程序的现有工具，分为 Javatailor 和 VECT。从表中可以看出，本文中收集的数据集具有较强的多样性，即包含不同指令数量以及致错类型的测试程序，有助于全面评估论文所提方法的有效性。

表 1　测试程序信息

索引	类名	指令数量	语句块数量	函数数量	程序复杂度	来源	致错类型
c1	TestShortVect	3138	596	43	1684	JavaTailor	Exception/Crash
c2	InvokeStaticSuccessTest	362	3	3	183	JavaTailor	Exception/Crash
c3	TestIntBoxing	1398	27	76	773	JavaTailor	Exception/Crash
c4	subcommon03	435	35	7	226	JavaTailor	Exception/Crash
c5	TestDoubleVect2	2952	192	20	1515	JavaTailor	Exception/Crash
c6	TestIntVect	1278	370	48	716	JavaTailor	Exception/Crash
c7	TestGCOld	529	108	20	293	JavaTailor	Exception/Crash
c8	Loops03	369	23	4	190	JavaTailor	Exception/Crash
c9	TestDoubleBoxing	1444	270	76	797	JavaTailor	Exception/Crash
c10	TestDoubleVect	2952	192	20	1572	JavaTailor	Exception/Crash
c11	subcommon04	460	41	7	239	JavaTailor	Exception/Crash
c12	demo11	1034	176	10	714	JavaTailor	Exception/Crash
c13	Test6196102	452	9	2	229	JavaTailor	Exception/Crash
c14	TestShortBoxing	2085	288	76	1124	JavaTailor	Exception/Crash
c15	TestCharVect2	2412	729	65	1351	JavaTailor	Exception/Crash
c16	TestBooleanVect	2168	614	43	1474	JavaTailor	Exception/Crash
c17	TestIntVect2	2340	689	61	1338	JavaTailor	Exception/Crash
c18	Test6726999	2745	281	57	1431	JavaTailor	Exception/Crash
c19	TestByteShortVect	1718	328	23	1139	JavaTailor	Exception/Crash
c20	dead03	1386	60	5	1116	JavaTailor	Exception/Crash

续表 1　测试程序信息

索引	类名	指令数量	语句块数量	函数数量	程序复杂度	来源	致错类型
c21	BasicFunctionality	220	69	7	147	VECT	Exception/Crash
c22	Test6798726	2898	278	57	1510	VECT	Exception/Crash
c23	Test6443505	1006	48	8	536	VECT	Exception/Crash
c24	BadPredicateAfterPartialPeel	1254	36	7	638	VECT	Exception/Crash
c25	TestShortFloatVect	1674	312	23	897	VECT	Exception/Crash
c26	TestShortVect2	2433	732	66	1360	VECT	Exception/Crash
c27	TestIntVect	1802	387	49	1218	VECT	Exception/Crash
c28	TestDoubleVect3	1903	616	43	1082	VECT	Exception/Crash
c29	stringvalueof001	1071	60	4	563	VECT	Exception/Crash
c30	TestIntAtomicOrdered	1535	590	42	881	VECT	Exception/Crash
c31	TestIntAtomicCAS	1546	600	43	944	VECT	Exception/Crash
c32	TestCRC32	790	87	10	448	VECT	Exception/Crash
c33	TestFloatBoxing	1998	277	76	1076	VECT	Exception/Crash
c34	TestIntAtomicCAS	2245	608	43	1244	VECT	Exception/Crash
c35	Test6855164	1468	24	4	742	VECT	Exception/Crash
c36	DivideMcTests	17476	36	6	10189	VECT	Exception/Crash
c37	Supplementary	1426	238	27	772	VECT	Exception/Crash
c38	HypotTests	3895	31	5	2066	VECT	Exception/Crash
c39	CubeRootTests	2359	59	5	1990	VECT	Exception/Crash
c40	TestLimitLoadBelowLoopLimitCheck	892	26	3	452	VECT	Exception/Crash
c41	Test6826736	255	52	4	140	VECT	Inconsistent Output
c42	checkarray	2880	15	6	1445	VECT	Inconsistent Output
c43	Test6892265	114	28	5	66	VECT	Inconsistent Output
c44	Test6368267	200	51	4	112	VECT	Inconsistent Output
c45	demo3	401	58	8	221	VECT	Inconsistent Output
c46	demo8	621	110	9	352	VECT	Inconsistent Output
c47	demo1	926	116	7	504	VECT	Inconsistent Output
c48	demo4	1018	135	13	684	VECT	Inconsistent Output
c49	demo13	621	110	9	352	VECT	Inconsistent Output
c50	ChecksumTest	175	49	5	102	VECT	Inconsistent Output

3.2.2 对比方法

本文将 JavaPruner 方法与最新的 Java 测试程序约简方法 J-ReduceEXT 进行对比。J-ReduceEXT[14]是 Kalhauge 等人在 J-Reduce 上的扩展工作。J-Reduce 通过构建 Java 程序依赖图，在类文件粒度对 Java 程序进行约简，J-ReduceEXT 则是在 J-Reduce 的基础上从函数的角度设计了更细粒度的 Java 测试程序约简方法。同时，Kalhauge 等人在 Featherwight Java[25]上对 J-ReduceEXT 进行了形式化的验证，证明了其方法的完备性。为了将 JavaPruner 与 J-ReduceEXT 进行公平的对比，本文在整理的数据集上以相同的实验配置运行两个方法，并记录其约减的效果以及运行时间用于后续的对比分析。

3.2.3 工具实现及实验环境

本文基于 OpenJDK8 以及 Soot 在 Jimple 指令级别实现 JavaPruner。其中，OpenJDK8 是当前主流的 OpenJDK 版本之一。Soot 则是当前较成熟的 Java 程序分析工具，其中封装了大量对 Java 类文件进行操作的功能函数。同时，在 Jimple 指令级别实现 JavaPruner 则有助于提升本工具的通用性，可以直接迁移至所有基于 Java 字节码的编程语言。为了验证约简过程中测试程序的缺陷检测能力是否丢失，本文收集了 4 个不同型

号版本的 Java 虚拟机，分别为：jdk8u361_hotspot、jdk8u_362-b09_openj9、jdk11.0.18_hotspot 和 jdk-11.0.18+10_openj9。通过在这些虚拟机上运行约简后的测试程序，从而验证在单个测试程序场景(测试程序直接触发了 JVM 异常或崩溃)以及差分测试场景(相同测试程序在不同 JVM 实现上有不同的输出)下测试程序的缺陷检测能力是否丢失。本文实验运行的服务器信息：12th Gen Intel(R) Core(TM) i5-12400F 2.50GHz、16GB 内存和 Windows11 操作系统。

3.3 结果分析

3.3.1 相针对 RQ1 的结果分析

为了回答 RQ1，本文将 JavaPruner 和已有工作在 50 个数据集上进行约简结果对比。评价指标为约简后 Jimple 指令集数量占原文件指令数量的比例，对比结果如表 2 所示。根据表 2 中的结果，可以得出以下结论。首先在约简效果上，本文基于语句块粒度的 Java 测试程序约简技术对比于已有工作，JavaPruner 在 45 个测试程序上可以取的更显著的约简效果，尤其是在 c36 测试程序上，约简后的测试程序仅占原程序的 1.09%，约简比例 98.01%。J-ReduceEXT 可以将测试程序平均约简为原大小的 57.8%,JavaPruner 可以将测试程序平均约简为原大小的 20.1%，该数据集上本文工作约简效果提升了 37.7%，且有 16 个测试程序约简效果提升大于 50%。为了进一步衡量本文方法提升的显著性，本文采用 Wilcoxon 秩和校验[43]对本文效果与已有工作效果做显著性检验，以两者约简后剩余代码量的占比作为输入，计算得到 P 值等于2.39×10^{-9}，可以得出结论本文对比于已有工作效果提升显著。其次，J-ReduceEXT 在 JavaTailor 和 VECT 生成的实验数据集上有 4 个测试程序约简率为 0%，8 个测试程序约简率小于 10%，结合表 1 中内容和测试程序的实际结构分析发现，这些测试程序函数数量非常少且致错代码片段位于的函数调用链的底层，约简掉任何一个函数都会导致致错代码不被执行，因此在这些程序中 J-ReduceEXT 的约简效果较差。相应的，若测试程序函数数量较大，程序内代码较为分散，则 J-ReduceEXT 的约简效果会比较可观。以 c3 为例，该测试程序共定义了 76 个函数，各个函数中代码含量较为均衡，没有出现代码量比例极高的函数，且致错函数的调用栈非常浅，大部分函数均以致错无关。经 J-ReduceEXT 约简后 c3 仅剩 4 个函数和 30%的代码量，效果较为可观。

需要注意的是，在部分测试程序上 JavaPruner 和 J-ReduceEXT 的效果均不理想。例如，在测试程序 c38 上两个方法约简后的测试程序均占原有程序的 86%。通过分析其程序结构可以知道其致错代码片段所在的函数只包含了一个代码片段，而该代码片段包含的程序指令数量占全程序的 86%，为顺序代码片段，导致两个方法的约简效果均不理想。通过对较长的顺序代码片段进行拆分并进行进一步的约简有助于处理这种罕见的测试程序，本文将在未来研究工作中对其进行讨论。综上所述，本文提出的 JavaPruner 方法相比于现有方法可以显著提升测试程序约简的效果。

表 2 J-ReduceEXT 与 JavaPruner 约简效果对比

索引	J-ReduceEXT	JavaPruner	索引	J-ReduceEXT	JavaPruner
c1	0.66	**0.03**	c2	1.00	**0.05**
c3	0.30	**0.01**	c4	0.38	**0.05**
c5	0.87	**0.06**	c6	0.47	**0.03**
c7	0.44	**0.06**	c8	0.18	**0.12**
c9	0.33	**0.12**	c10	0.97	**0.12**
c11	0.36	**0.32**	c12	0.14	0.13
c13	1.00	**0.25**	c14	0.23	**0.04**
c15	0.79	**0.01**	c16	0.70	**0.26**
c17	0.55	**0.09**	c18	0.42	**0.10**
c19	0.51	**0.18**	c20	0.76	**0.23**
c21	0.26	**0.11**	c22	0.46	**0.10**
c23	0.83	**0.03**	c24	0.63	0.62
c25	0.81	**0.43**	c26	0.52	**0.01**
c27	0.62	**0.29**	c28	0.67	**0.18**
c29	0.46	0.46	c30	0.65	**0.03**
c31	0.65	**0.04**	c32	0.54	**0.09**

续表 2 J-ReduceEXT 与 JavaPruner 约简效果对比

索引	J-ReduceEXT	JavaPruner	索引	J-ReduceEXT	JavaPruner
c33	0.51	**0.32**	c34	0.76	**0.02**
c35	0.99	**0.65**	c36	1.00	**0.01**
c37	0.36	**0.28**	c38	0.86	0.86
c39	0.85	**0.01**	c40	1.00	**0.77**
c41	0.93	**0.41**	c42	0.02	0.02
c43	0.84	**0.54**	c44	0.98	**0.39**
c45	0.33	**0.18**	c46	0.23	**0.14**
c47	0.14	**0.08**	c48	0.22	**0.12**
c49	0.31	**0.21**	c50	0.42	**0.37**

3.3.2 相针对 RQ2 的结果分析

为了回答 RQ2，本文进行了消融实验，从而验证本文设计的测试程序预处理方法是否能有效提升测试程序约简的效率。对比结果如表 3 所示。其中，JavaPruner--代表删除本文测试程序预处理方法的实现。JavaPruner--/JavaPruner 列为两个方法在测试程序上约简所花费的时间，单位为秒。由于两个方法均是在语句块粒度上进行测试程序约简，因此两者在程序程序最终约简效果上没有差别，均可以将其约简至相同大小，差别仅在不同的约简效率上。减少的约简时间表示 JavaPruner 相比于 JavaPruner--减少的约简时间，单位为秒。效率提升表示对比去除预处理的约简效率，保留预处理提升效率的倍数，单位为 times。

通过对表 3 的分析可以得出以下结果。首先，去除测试程序预处理方法 JavaPruner--在 50 个测试程序的平均约简时间为 306 秒，JavaPruner 的平均约简时间则为 82 秒。相比之下，加上预处理的 JavaPruner 方法约简效率平均提升了 2.73 倍。其次，在 Jimple 指令集数量大于 1500 的测试程序上，JavaPruner--的平均约简时间为 474 秒，JavaPruner 的平均约简时间为 93 秒，JavaPruner 相比于 JavaPruner--约简效率提升了 4.09 倍；在 Jimple 指令集介于 500 至 1500 的测试程序上，JavaPruner--的平均约简时间为 200 秒，JavaPruner 的平均约简时间为 61 秒，JavaPruner 相比于 JavaPruner--约简效率提升了 2.27 倍；在 Jimple 指令集小于 500 的测试程序上，JavaPruner--的平均约简时间为 131 秒，JavaPruner 的平均约简时间为 91 秒，JavaPruner 相比于 JavaPruner--约简效率提升了 0.44 倍。由此可以得出结论，随着测试程序大小的增长，预处理对于约简效率的提升会变得更加显著，而当原程序体量很小时预处理带来的提升并不可观，甚至在部分数据上没法抵消其本身带来的成本。以测试程序 c2 举例，由表 1 可知其函数数量与语句块数量极少，该程序一共定义了 3 个函数，每个函数内部只包含一个语句块，说明其原本需要进行的缺陷检测能力验证操作频次就很低，预处理带来的效益甚微。与此同时，预处理步骤中未执行代码片段预处理与冗余函数预处理所需的成本甚至要大于其带来的提升，所以在这类测试程序中加了预处理其效率反而会更低。

表 3 去除预处理和保留预处理约简效率对比

索引	JavaPruner--(s) /JavaPruner(s)	减少约简时间(s) /效率提升(times)	索引	JavaPruner--(s) /JavaPruner(s)	减少约简时间(s) /效率提升(times)
c1	359 / **32**	**327 / 10.22**	c2	5 / 22	-17 / -3.4
c3	158 / **23**	**135 / 5.87**	c4	46 / **11**	**35 / 3.18**
c5	380 / **157**	**223 / 1.42**	c6	246 / **219**	**27 / 0.12**
c7	77 / **23**	**54 / 2.35**	c8	18 / **15**	**3 / 0.2**
c9	167 / **33**	**134 / 4.06**	c10	338 / **117**	**221 / 1.89**
c11	32 / **15**	**18 / 1.29**	c12	93 / **20**	**73 / 3.65**
c13	11 / 16	-5 / -0.45	c14	1319 / **529**	**790 / 1.49**
c15	457 / **52**	**405 / 7.79**	c16	706 / **37**	**669 / 18.08**
c17	1447 / **61**	**1386 / 22.72**	c18	279 / **175**	**104 / 0.59**

续表 3 去除预处理和保留预处理约简效率对比

索引	JavaPruner--(s) /JavaPruner(s)	减少约简时间(s) /效率提升(times)	索引	JavaPruner--(s) /JavaPruner(s)	减少约简时间(s) /效率提升(times)
c19	208 / **19**	**189 / 9.95**	c20	897 / **47**	**850 / 18.09**
c21	551 / **388**	**163 / 0.42**	c22	408 / **68**	**340 / 5**
c23	33 / **17**	**16 / 0.94**	c24	40 / **26**	**14 / 0.54**
c25	269 / **47**	**222 / 4.72**	c26	558 / **47**	**511 / 10.87**
c27	493 / **126**	**367 / 2.91**	c28	687 / **39**	**648 / 16.62**
c29	92 / **12**	**80 / 6.67**	c30	711 / **105**	**606 / 5.77**
c31	562 / **102**	**460 / 4.51**	c32	112 / **61**	**51 / 0.84**
c33	254 / **131**	**123 / 0.94**	c34	308 / **31**	**277 / 8.94**
c35	77 / **74**	**3 / 0.04**	c36	531 / **95**	**436 / 4.59**
c37	144 / **34**	**110 / 3.24**	c38	44 / **25**	**19 / 0.76**
c39	43 / **22**	**21 / 0.95**	c40	25 / **18**	**7 / 0.39**
c41	133 / **93**	**40 / 0.43**	c42	137 / **42**	**95 / 2.26**
c43	71 / **62**	**9 / 0.15**	c44	129 / **81**	**48 / 0.59**
c45	285 / **158**	**127 / 0.8**	c46	339 / **66**	**273 / 4.14**
c47	172 / **53**	**119 / 2.25**	c48	372 / **166**	**206 / 1.24**
c49	360 / **161**	**199 / 1.24**	c50	165 / **147**	**18 / 0.12**

综上所述，预处理操作对 JavaPruner 效率的提升是非常可观的，在指令数量较多的测试程序上预处理带来的效率提升格外显著，这是由于预处理可以提前删除大量无关代码，从而减少了后续约简中缺陷检测能力检验的频次。而在指令数量较少的测试程序上的效率提升较小，且会出现效率下降的情况。考虑到此类测试程序本身所需的约简时间很少，其效率下降在可接受范围内，如表 3 中 c13 程序中虽然效率下降了 45%，实际增加的时间成本只有 5 秒。同时，在后续的工作中，可以设置相应的指令或声明函数数量阈值从而判断是否需要对其进行预处理。

3.3.3 相针对 RQ3 的结果分析

为了分析 JavaPruner 在实际约简工作的效果，本文从 50 个样例中选择 10 个具有代表性的测试程序进行人工的分析和约简。这 10 个测试程序覆盖了数据集中不同的生成工具，致错类型和代码量大小，得到不同生成来源、不同致错类型和不同代码量大小情况下 JavaPruner 和人工分析约简的结果，使得实验结果更具有一般性。通过比较 JavaPruner、J-ReduceEXT 和人工约简的差异衡量对比于 J-RedeuceEXT 和人工约简本文工作的优势所在。本文提出本文方法属于自动化软件测试技术，目的是降低人工测试的成本，为此本文将从约简效果和约简所用时间两个方面把 JavaPruner 和 J-ReduceEXT、人工约简进行对比，结果如表 4 所示。其中编号表示所用的测试程序。指令数量表示约简前测试程序的 Jimple 指令数量，JavaPruner 表示用本文方法约简后 Jimple 文件的指令数。J-ReduceEXT 表示用 J-ReduceEXT 约简后 Jimple 文件的指令数。人工约简表示人工约简后 Jimple 文件的指令数。JavaPruner 缺陷定位时间表示用本文语句块粒度约简后定位单个测试程序缺陷位置所需的时间。J-ReduceEXT 缺陷定位时间表示用 J-ReduceEXT 约简后定位单个测试程序缺陷位置所需的时间。人工缺陷定位时间表示不使用任何约简工具，人工定位单个测试程序缺陷位置所用的时间。

表 4 使用 JavaPruner 和人工约简缺陷定位时间对比

索引	指令数量	JavaPruner	J-ReduceEXT	人工	JavaPruner 缺陷定位时间	J-ReduceEXT 缺陷定位时间	人工 缺陷定位时间
c1	3138	94	2071	18	2min	68min	92min
c5	2952	177	2568	25	4min	70min	89min
c12	1034	124	144	14	3min	4min	47min
c16	2168	563	1517	28	7min	42min	79min
c21	220	22	59	11	1min	3min	16min
c26	2433	32	1265	26	4min	51min	86min
c31	1546	61	927	36	10min	42min	73min
c34	2245	44	1706	24	5min	46min	76min
c37	1426	399	527	15	5min	14min	49min
c46	621	86	149	45	2min	13min	32min

通过对表 4 分析可得出以下结论。首先，在经过 JavaPruner 约简后，人工定位缺陷的时间成本会显著缩减，平均节省了 93.0%的缺陷定位时间，大大提高了测试工作的效率。相比于现有工作 J-ReduceEXT，本文工作可以平均可以节省 77.27%的缺陷定位时间。对于部分测试程序，如 c12，其缺陷定位时间仅带来了 2 min 左右的提升，通过分析发现其原因在于该类测试程序本身函数数量较大，且函数体内部代码较为分散，J-ReduceEXT 在此类测试程序上可以取得较优的约简效果，因此在这种情况下，JavaPruner 对约简效果提升较小。这里我们以一个能够真实触发 JVM 缺陷并已被 OpenJ9 官方确认的程序 c21 为例[44]。该程序暴露的 JVM 缺陷为，在 OpenJ9 的 JDK8 版本虚拟机上运行会导致虚拟机 Crash，而在其余版本的虚拟机上则不会。如图 6 所示，(a)为经过 JavaPruner 约简后的 Java 测试程序，(b)为在(a)的基础上进一步进行人工约简后提交给 OpenJ9 官方的测试程序。通过对比两个约简结果可以发现，本文工作对于实际的测试工作有着很好的辅助作用，通过删除大量与缺陷表现无关的代码，大大降低了开发人员进行缺陷定位的成本。然而，基于现有的其他测试程序约简方法 J-ReduceEXT 在该测试程序上的约简，仅能删除冗余的函数调用，对于函数内部的冗余语句块约简效果很差。值得注意的是，对比于人工约简的结果，本文工作还有着很大的约简空间，如程序 c16 所示在经过 JavaPruner 约简后还需要人工约简超过 500 行的指令，这是由于人工约简的粒度为单个指令。然而由于采用单个指令为粒度进行 Delta 调试自动化约简，会导致约简过程中程序执行验证的频次变得极高，从而效率进一步大幅下降，从而失去实用性。同时，以 c16 为例，JavaPruner 已经约去了大部分复杂结构，所剩余的 563 行指令集中只包含数个结构简单的语句块，所需的人工约简成本非常低。

综上所述，JavaPruner 能够对现有测试工作起到很好的辅助优化作用，有效减少了开发人员分析测试程序进行缺陷定位的时间，满足测试工作的要求。

```
public static void main(String[] var0) throws Exception {
   factory = ProviderFactory.getDefaultFactory();
   if (factory != null) {
      try {
         throw new OutOfMemoryError();
      } catch (OutOfMemoryError var1) {
         long var10000 = TRAPCOUNT;
         bp.probeWithArgs(-2, 2.0F, "1 ^", new Long(-2L));
         TRAPCOUNT = var10000 + 1L;
      }
   }
}
```

a) JavaPruner 约简结果

```
public class Case {
   public static void main(String[] args) {
      // this statment will cause OpenJ9 JDK8 Crash
      ProviderFactory.getDefaultFactory();
   }
}
```

b) OpenJ9 缺陷报告代码示例

图 6 真实致错程序的约简示例

4 讨论

本文提出基于程序约束的细粒度 Java 测试程序约简技术 JavaPruner。JavaPruner 通过构建测试程序内部的依赖关系约束，实现了对程序语句块的精细化约简。与现有工作相比，本文进一步细化了约简的粒度，从而实现了更高程度的简化。为了解决细粒度带来的约简效率下降问题，本文还设计并实现了未执行代码片段以及冗余函数预处理功能。通过消融实验证明，本文设计的预处理操作可以有效提升约简的效率。实验结果表明，在个别测试程序中，JavaPruner 效果与现有工作相当。同时在个别测试程序中预处理操作会带来约简效率的下降，为了进一步深入探讨这些问题，本节将从测试程序结构和程序执行方面进行详细的讨论和分析。

JavaPruner 约简效果受测试程序结构影响. 相比于已有工作，JavaPruner 约简效果的优势来自于其更细的粒度，但是当测试程序内部出现单个语句块包含大量代码的情况时，JavaPruner 的粒度优势会被削减，从而影响它约简的效果。根据表 1 与表 2 可知 JavaPruner 的约简效果会随着原程序语句块数量的增长呈现出增长趋势。一方面语句块数量庞大的测试程序其所包含的致错无关代码比例较高，JavaPruner 会有更多可约简的代码；另一方面语句块的数量可以说明测试程序内部代码内容的分布情况，当语句块数量很小时，原始程序内部可能存在大量代码内容较多的语句块，从而降低语句块约简的粒度优势，影响约简效果。

JavaPruner 约简效率受缺陷检测能力验证的限制. 本文的约简策略基于 Delta 调试算法，通过在每一步约简操作后执行约简后测试程序来验证其缺陷检测能力是否丢失，这导致算法的执行效率受到缺陷检测能力操作执行时间以及次数的限制。以表 3 中的 c21 为例，其执行结果会触发 OpenJ9 虚拟机的 Crash，Crash 导出日志文件时间过长，导致单次执行的成本高昂，致使 Delta 调试运行过程效率低下。考虑到测试约简工作往往更注重约简效果评估，而非约简效率。并且，约简的效率并不能反映约简效果的好坏，即使某种方法具有较高的约简效率，如果其约简效果不佳，约简后的测试程序仍然需要开发人员花费大量的分析成本来定位缺陷。本文 3.3.3 节证明，和细粒度带来的约简效率降低相比，JavaPruner 带来的缺陷定位效率的提升是更加显著的，即更优的约简效果可以抵偿效率的降低。由于 JavaPrunner 在更细粒度的代码片段级别进行约减，需要执行更多的缺陷能力检测操作，可能导致相对较低的效率。为此，本文设计了测试程序预处理方法，尽可能减少约简的成本，从而将约减效率提升至可接受范围内，如 RQ2 中的结果所示。同时，目前存在一些研究工作对 Delta 调试做出改进以提升其运行效率[45][46]，在后续的工作中可以尝试引入更先进的 Delta 调试算法来进一步提升 JavaPrunner 的效率。

预处理对效率的提升受测试程序结构影响. 本文实现了未执行代码片段以及冗余函数的预处理操作，前者通过检测与删除程序中不被执行的代码片段从而减少约简的验证成本，后者在函数粒度上过滤掉致错无关的函数。两者都能有效减少执行 Delta 调试算法的频次进而提高约简效率。为了观察预处理对约简效率的提

升情况，本文设计了消融实验，比较去除和保留预处理两种情况下 JavaPruner 约简单个测试程序所需的时间。通过对表 1 与表 2 深入分析，预处理对约简效率的提升随着原程序函数数量和语句块数量的增长呈现出增长趋势，一方面，语句块数量与函数数量庞大的测试程序冗余代码片段与函数所占比例较高，预处理由更多可约简的代码；另一方面，预处理涉及到的未执行代码片段解析和冗余函数分析会带来一定的时间成本，因此在某些语句块数量和函数数量很小的程序上预处理操作会导致约简效率的下降。

4.1 有效性影响因素分析

本小节对可能会影响到本文工作有效性的因素进行分析。

内部有效性分析. 本文方法的实现较为复杂，可能会影响到方法有效性的内部因素主要为代码实现是否正确。为了减少代码实现给本文工作带来的影响，本文的代码部分均使用了成熟的第三方框架，如 Soot 等。同时本文参考了已有工作的代码实现，如 J-Reduce，C-Reduce 等，同时本工作的三位开发人员均对编码实现进行多次验证，最大程度地保证了本文代码实现的正确性。

外部有效性分析. 影响本文实验结果的结论是否具有一般性的因素主要有三个。第一个因素为本文所使用的数据集是否具有代表性，为保证本文实验结论的有效性，本文使用目前最新的 JVM 测试工具 JavaTailor 和 VECT 生成实验数据集，该数据集可覆盖 Java 程序常见的结构类型和 JVM 测试常见的缺陷类型。第二个影响因素为实验平台是否具有一致性，为了减少实验平台对实验结果的影响，本文的所有实验均在同一个实验平台上运行，同时所有实验运行多次以消除可能的随机性对本文实验的影响。第三个因素为对比方法是否具有代表性，考虑到 J-ReduceEXT 与本文工作的实验对象均是 Java 程序，且现有实验已验证了 J-ReduceEXT 方法的有效性和先进性，具有一定的代表性，故本文工作选择与 J-ReduceEXT 进行对比。

综合有效性分析. 影响本文实验结果是否有效的主要因素为实验所选择的评价指标是否合理。为此，本文实验采取了约简工作中最常用的两项指标来评估本文工作，即约简后的代码量所占原始程序代码量的比例和约简单个测试程序所用的时间。同时，为了评估本文工作是否符合当前测试工作要求，本文引入了缺陷定位时间这一指标，用于衡量定位单个测试程序缺陷位置的时间，进一步提升了实验结论的有效性。

5 总结与展望

本文提出了基于程序约束的细粒度 JVM 测试程序约简方法 JavaPruner。JavaPruner 首先在语句块粒度设计了用于 Java 测试程序约简的细粒度代码片段度量方式，并基于此构建不同代码片段之间的依赖关系。随后通过不同代码片段之间的依赖关系设计带语义语法约束的 Delta 调试算法。同时，为了提升约简效率，JavaPruner 从未执行代码片段以及冗余函数两个纬度对测试程序进行预处理，提升约简效率。本文在 50 个测试程序上对 JavaPruner 的约简效果进行了实验研究。实验结果表明 JavaPruner 的约简效果显著优于已有工作。在保证测试程序正确性与缺陷检测能力的前提下，能够有效降低测试程序的代码复杂度，最高可以将测试程序约简至原有程序大小的 1.09%，大大减小了测试人员缺陷定位分析的成本。

在后续研究工作中，一方面，考虑到目前 JavaPruner 是基于遍历的方式对代码片段进行约简，其效率可以有进一步的提升空间。在后续的工作中，可以通过引入程序后向切片分析与测试程序致错根因相关的测试代码，从而对测试程序进行进一步的预处理；以及在约简过程中设计更先进的待约简代码片段选择算法，从而进一步提升减约的效率。

References：

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. ACM Comput. Surv. 53, 1 (2020), 4:1–4:36.

[2] Chen Y, Su T, Sun C, et al. Coverage-directed differential testing of JVM implementations[C]//proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2016: 85-99.

[3] Chen Y, Su T, Su Z. Deep differential testing of JVM implementations[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 1257-1268.

[4] Zhao Y, Wang Z, Chen J, et al. History-driven test program synthesis for JVM testing[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1133-1144.

[5] Gao T, Chen J, Zhao Y, et al. Vectorizing Program Ingredients for Better JVM Testing [C]. In Proceedings of the 32st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023: 526–537.

[6] Zhang Y, Wang Z, Jiang J, et al. Toward Improving the Robustness of Deep Learning Models via Model Transformation[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-13

[7] Yang C, Chen J, Fan X, et al. Silent Compiler Bug De-duplication via Three-Dimensional Analysis [C]. In Proceedings of the 32st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023: 677–689.

[8] Hammoudi M, Burg B, Bae G, et al. On the use of Delta Debugging to reduce recordings and facilitate debugging of web applications[C]//Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015: 333-344.

[9] Kalhauge C G, Palsberg J. Binary reduction of dependency graphs[C]//Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019: 556-566.

[10] McDermid J. Software Engineering: A Practitioner's Approach [Book Review][J]. Software Engineering Journal, 1995, 10(6): 266-266.

[11] Moller K H, Paulish D J. An empirical investigation of software fault distribution[C]//[1993] Proceedings First International Software Metrics Symposium. IEEE, 1993: 82-90.

[12] Zeller A. Where Do Bugs Come from? [C/OL] // Yorav K. In Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007.

[13] Yang Y, Zhou Y, Sun H, et al. Hunting for bugs in code coverage tools via randomized differential testing[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 488-499.

[14] Chen Y, Su T, Su Z. Deep differential testing of JVM implementations[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 1257-1268.

[15] JavaPruner. https://github.com/sail-repos/JavaPruner.

[16] Gong D, Su X, Wang T, et al. A Survey of Test Cases Reduction Approach Oriented to Software Fault Localization. Intelligent Computer and Applications. 2014, 4(3): 39-41.(in chinese)

[17] Yu Y, Jones J A, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization[C]//Proceedings of the 30th international conference on Software engineering. 2008: 201-210.

[18] Zhong H, Zhang L, Mei H. An experimental study of four typical test suite reduction techniques[J]. Information and Software Technology, 2008, 50(6): 534-546.

[19] Zhang L, Marinov D, Zhang L, et al. An empirical study of junit test-suite reduction[C]//2011 IEEE 22nd International Symposium on Software Reliability Engineering. IEEE, 2011: 170-179.

[20] Gu Q, Tang B, Chen D X. A test suite reduction technique for partial coverage of test requirements[J]. Jisuanji Xuebao(Chinese Journal of Computers), 2011, 34(5): 879-888. (in chinese)

[21] Fraser G, Wotawa F. Redundancy based test-suite reduction[C]//International Conference on Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007: 291-305.

[22] Zeller A. Isolating Failure-Inducing Input[J]. IEEE Transactions on Software Engineering. Citeseer, 2001.

[23] Regehr J, Chen Y, Cuoq P, et al. Test-case reduction for C compiler bugs[C]//Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. 2012: 335-346.

[24] Misherghi G, Su Z. HDD: hierarchical Delta Debugging[C]//Proceedings of the 28th international conference on Software engineering. 2006: 142-151.

[25] Igarashi A, Pierce B C, Wadler P. Featherweight Java: a minimal core calculus for Java and GJ[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2001, 23(3): 396-450.

[26] Vallée-Rai R, Co P, Gagnon E, et al. Soot: A Java bytecode optimization framework[M]//CASCON First Decade High Impact Papers. 2010: 214-224.

[27] Lam P, Bodden E, Lhoták O, et al. The Soot framework for Java program analysis: a retrospective[C]//Cetus Users and Compiler Infastructure Workshop (CETUS 2011). 2011, 15(35).

[28] Vallée-Rai R, Gagnon E, Hendren L, et al. Optimizing Java bytecode using the Soot framework: Is it feasible?[C]//Compiler Construction: 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9. Springer Berlin Heidelberg, 2000: 18-34.

[29] Miecznikowski J. New algorithms for a Java decompiler and their implementation in Soot[J]. 2003.

[30] Bartel A, Klein J, Le Traon Y, et al. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot[C]//Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. 2012: 27-38.

[31] Canales F, Hecht G, Bergel A. Optimization of java virtual machine flags using feature model and genetic algorithm[C]//Companion of the ACM/SPEC International Conference on Performance Engineering. 2021: 183-186.

[32] Grimmer M, Rigger M, Schatz R, et al. Trufflec: Dynamic execution of c on a java virtual machine[C]//Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools. 2014: 17-26.

[33] Rigger M, Grimmer M, Wimmer C, et al. Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle[C]//Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages. 2016: 6-15.

[34] Smith J, Nair R. Virtual machines: versatile platforms for systems and processes[M]. Elsevier, 2005.

[35] Allen F E. Control Flow Analysis [C/OL]. In Proceedings of a Symposium on Compiler Opti- mization, New York, NY, USA, 1970: 1–19.

[36] Bruschi D, Martignoni L, Monga M. Detecting self-mutating malware using control-flow graph matching[C]//Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3. Springer Berlin Heidelberg, 2006: 129-143.

[37] Nandi A, Mandal A, Atreja S, et al. Anomaly detection using program control flow graph mining from execution logs[C]//Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. 2016: 215-224.

[38] Kotzmann T, Wimmer C, Mössenböck H, et al. Design of the Java HotSpot™ client compiler for Java 6[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2008, 5(1): 1-32.

[39] IBM. Ibm developer kits. https://developer.ibm.com/javasdk/.

[40] Groce A, Holzmann G, Joshi R. Randomized differential testing as a prelude to formal verification[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 621-631.

[41] McKeeman W M. Differential testing for software[J]. Digital Technical Journal, 1998, 10(1): 100-107.

[42] McCabe T J. A complexity measure[J]. IEEE Transactions on software Engineering, 1976 (4): 308-320.

[43] Wilcoxon F, Katti S K, Wilcox R A. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test[J]. Selected tables in mathematical statistics, 1970, 1: 171-259.

[44] eclipse-openj9 issue 17389. https://github.com/eclipse-openj9/openj9/issues/17389.

[45] Wang G, Shen R, Chen J, et al. Probabilistic Delta Debugging[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 881-892.

[46] Cleve H, Zeller A. Finding failure causes through automated testing[J]. arXiv preprint cs/0012009, 2000.

附中文参考文献:

[16] 龚丹丹, 苏小红, 王甜甜, 等. 面向软件错误定位的测试用例约简技术综述[J]. 智能计算机与应用, 2014, 4(3): 39-41.

[20] 顾庆, 唐宝, 陈道蓄. 一种面向测试需求部分覆盖的测试用例集约简技术[D]. , 2011.