提升增量数据流分析中中间表示映射的精度

Anushri Jana
anushri.jana@tcs.com TCS Research 和 DoT SPPU，浦那，印度

Kush Goradiya
kush.goradiya@tcs.com TCS Research，浦那，印度

Uday P. Khedker
uday@cse.iitb.ac.in
孟买印度理工学院，孟买，印度

arXiv:2611.32920v1 [cs.PL] 24 Nov 2026

摘要

静态分析工具基于中间表示（IR）运行——这是一种程序的内部模型，数据流分析在其上计算事实，并以 IR 对象标识符为索引存储这些事实。增量数据流分析会在连续程序版本间复用这些缓存事实，但这要求准确映射新旧版本中的 IR 对象。当映射不精确时，后果不仅仅是错失优化：未映射的 IR 对象会扩大变更集，增加分析器必须重新计算的程序点数量，并迫使其重复执行原本可通过更精确映射避免的工作。这种成本并非某一种 IR 对象所独有；只要 IR 级标识在不同版本间不稳定，就会产生这种成本。

任何会分解嵌套源表达式的中间表示，都会生成由方法局部计数器命名的临时变量；当方法中较早位置插入或删除语句时，计数器便会发生偏移，从而破坏任何基于名称或属性的映射。我们提出了一种系统化算法，可在连续版本之间为这些临时变量建立可靠的一对一映射，并对该映射进行形式化和可靠性证明，确保生成的每个映射在语义上都正确。由此得到的精确映射缩小了变更集，减少了增量分析必须重新计算的程序点数量。我们在 Soot 框架中针对 Jimple 中间表示实现了该方法，并使用 BUGSWARM 数据集中的真实修改，在 8 个真实 Java 项目上进行了评估。与 Reviser 图差分算法相比，我们的方法平均将中间表示变更集缩减了 82% 以上，同时每个项目的完整处理流程均在 250 毫秒内完成，直接驳斥了这样一种常见假设：更精确的变更集必然需要更多计算时间。

关键词： 增量数据流分析、IR 映射、临时变量、Jimple 中间表示。

ACM 引用格式：
Anushri Jana、Kush Goradiya 和 Uday P. Khedker。2026年。提升增量数据流分析中中间表示映射的精度。载于 . ACM，纽约，NY，美国，11页。https://doi.org/10.1145/nnnnnnn.nnnnnnn

允许出于个人或课堂用途免费制作本作品全部或部分内容的数字版或纸质副本，前提是不得为营利或商业利益制作或分发副本，且副本首页须附上此声明和完整引文。对于本作品中由作者以外人士拥有版权的部分，必须尊重其版权。注明出处后可以摘录。若要以其他方式复制、再版、发布到服务器或重新分发至列表，则须事先获得明确许可和/或支付费用。请向 permissions@acm.org 申请许可。*Conference'17*，美国华盛顿特区© 2026年 版权所有者/作者保留版权。出版权已授权给 ACM。ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

现代软件系统持续演进。每次提交后，分析工具都必须重新检查代码库，以检测错误、安全漏洞或正确性违规。对于代码频繁变更的大型代码库，在每次变更后从头开始重新分析的成本高得难以承受。增量数据流分析通过识别必须重置并重新计算的程序点，并在其他所有程序点复用缓存结果，而不是重新分析整个代码库 [3, 18, 23, 28, 29, 32, 33]，从而避免了这一成本。标准方法是识别变更集——即在两个程序表示之间被发现不匹配的程序点——并通过分析依赖结构，通常取所有可从变更集到达的程序点，以此推导出更大的受影响区域 [18] 。受影响区域中的节点会被重置并重新计算；其他节点则复用缓存结果。由于受影响区域源自变更集，不精确（膨胀的）变更集必然会产生相应更大的受影响区域。

中间表示与数据流分析。静态分析工具基于一种中间表示（IR）运行——这是一种与语言无关的程序内部模型，以便于分析的形式呈现程序的结构和数据流。Soot [31] 是一种广泛使用的 Java 字节码分析框架，其主要中间表示 Jimple 会将基于栈的 JVM 字节码转换为类型化、三地址、无栈的线性形式。为实现这一简化，Jimple 会将复杂的嵌套 Java 表达式分解为一系列简单语句，每条语句至多执行一个操作。临时变量—自动生成的局部变量以 $ 为前缀（例如，$i0、$stack1），用于保存方法调用的中间结果和引用。这些临时变量纯粹是中间表示降级过程产生的副产物；与源代码命名的局部变量不同，后者的标识符在不同版本间保持稳定，而这些临时变量在源代码中没有对应项，其名称由方法局部计数器分配，每当在方法中创建一个新的临时变量时，该计数器就会递增。

数据流分析会针对每个程序点计算程序状态的抽象描述，例如哪些变量可能为空，或者哪些表达式是常量。在 Jimple 中，每个程序点都直接对应一条 Jimple 语句，分析事实则以该程序点可见的变量（包括临时变量）表示。例如，空值分析可能会记录在某个程序点上 $stack7 非空，常量传播分析则可能会记录 $stack3 的值为 42。这些事实以变量名作为符号键进行存储和传播。因此，当不同版本间临时变量被重命名时——例如 $stack7 变成 $stack8——即使两个临时变量保存的是同一计算的结果，与旧名称关联的缓存事实也无法自动迁移到新版本。其后果不仅仅是错失一次复用机会

：未映射的 IR 对象会进入变更集——即在不同版本间被直接发现存在差异的程序点集合。由于其他位置的缓存事实可能依赖这些程序点，增量分析器会根据变更集推导出更大的受影响区域 [18]，并重置和重新计算该区域中的每个节点。因此，变更集膨胀并非只是多出几个节点：它会连带导致受影响区域相应膨胀，进一步加剧映射不精确带来的成本。这种成本并非某一种 IR 对象所独有——只要 IR 级标识在不同版本间不稳定，就会产生这种成本。因此，IR 映射的精确性会直接且可衡量地影响增量分析器必须重新计算的内容量。

映射的前提。 增量数据流分析的关键推动因素，是在程序的新旧版本之间建立准确的 IR 对象映射，从而将针对旧标识符索引的分析事实转移到新版本中的对应对象。这一要求已在先前工作中被明确提出并得到处理。Ghime 等人 [17] 将其形式化为面向 COBOL 程序的专有 IR 的*IR* 映射，并证明如果没有正确的映射，就无法安全地转移缓存结果。Nichols 等人 [22] 则通过对不同版本间的 IR 实体进行启发式匹配，处理了增量 JavaScript 分析中的同一要求。在这两种情况下，映射问题都是可处理的，因为待匹配的实体具有稳定且源代码派生的身份。本文针对 Java/Jimple 环境中的同一要求展开研究；在该环境中，生成的临时变量没有稳定的身份，因此基于名称或基于属性的匹配均不适用。

任何将嵌套源表达式分解为简单语句的中间表示，都必须将中间结果存储在某处，因此这类中间表示会生成临时变量：Jimple 创建以 $ 开头的局部变量，LLVM 创建未命名的%*n* 寄存器，而 WALA 等框架中的 SSA构造则会创建表示为 v1、v2 等的值编号。在每种情况下，计数器都会根据位置分配名称，因此插入一条语句后，其后的每个临时变量都会重新编号，基于名称的匹配也会对所有这些变量失效。我们研究 Soot/Jimple 场景下的这一问题。表1展示了其中的核心困难。当在 `x = x + y + z`（版本1）之前插入一条新语句时，版本2的 Jimple 中间表示会将临时变量从 $stack7 重新编号为 $stack8。尽管子表达式 x+y 在两个版本中在语义上完全相同，但基于字符串的比较会将这两条中间表示语句视为完全不同，错误地将第8行标记为已更改，尽管源代码并未发生变化。

对增量分析的影响。 由于生成的临时变量名称是按位置分配的，插入一条语句会导致方法局部计数器发生偏移，从而重命名插入点之后引入的每个临时变量。任何通过字符串匹配中间表示语句来识别变更代码的技术，都会将每个被重命名的临时变量归类为新的、无法匹配的语句——尽管底层计算完全相同——而且由于重命名会沿着后续所有使用该重命名临时变量作为操作数的语句级联传播，最终的变更集会远远超出实际变更。Reviser图差分算法 [2, 3]就是这样一种技术：在我们的实验中，对于一个实际只修改了13行的程序，它报告了最多219行变更的源代码行——膨胀了 17× 倍，这一结果仅通过直接

表1：两个版本的 Java 代码及其对应的 Jimple 中间表示。尽管经过重命名，我们的算法仍会映射 红色 的临时变量。源代码行标签（以开头的数字表示）有助于对齐中间表示语句与其对应的 Java 行。

Java版本1	Java版本2（插入）
... 7: x = x + y + z; ...	... 7: System.out.println(y); 8: x =x +y +z; ...
Jimple 中间表示版本1	**Jimple 中间表示版本 2**
7: $stack7 =x +y; 7: x = $stack7 +z;	7: $stack7 = <系统: ...>; 7: **虚拟调用** ... 8: $stack8 = x +y; 8: x = $stack8 + z;

我们的 算法映射 $stack7(v1) ↦ $stack8 (v2)，正确识别出第 8 行为未 改变。

字符串比较得出，尚未进行任何基于可达性的传播（第2节将详细讨论Reviser的图差分算法及其设计理念）。由于Reviser通过可达性从该变更集中推导受影响区域，膨胀的变更集会产生相应更大——很可能大得多——的受影响区域，因此 17× 图中的数值只是无效重复计算真实成本的下界。每个虚假条目都会迫使系统重新计算底层逻辑未被修改的语句的分析事实，从而抵消增量分析的效率优势。重新计算成本并非唯一后果。增量分析会将其事实关联到中间表示标识符上，因此当映射无法将重命名的临时变量与其对应项关联起来时，以该标识符为键的事实不会被继承，分析便会将其报告为新增或更新事实。在关于指针分析的配套工作中，我们观察到了完全相同的现象：一次无关的编辑重命名了临时变量，而分析报告程序中编辑从未影响的部分存在变更事实。膨胀的变更集会损害增量分析的效率；膨胀的变更事实则会损害其可信度，因为报告将结果归因于提交从未影响的代码，审阅者也无法判断这些结果与提交实际影响的结果有何区别。我们的结果表明，常见的假设——更精确的变更集必然需要更多时间来计算——并不成立：我们的方法能够识别出严格更精确的变更集，同时完成计算所需的时间显著更少。

为便于本文进行比较，我们单独实现了 Reviser 的图差分组件 [2],，该组件负责变更集计算，也是直接受临时变量重新编号影响的部分。我们不使用 Reviser 的增量 IFDS/IDE 求解器或 Reviser 系统的任何其他组件。

我们的方法。 我们的关键洞见是，Soot 会在中间表示中嵌入源代码行标签，记录每条 Jimple 语句是由哪一行源代码生成的。对于经源代码级差异确认在文本上未发生更改的源代码行，Soot 会以确定性的方式生成相同顺序且类型相同的 Jimple 语句序列。在该序列中，临时变量最初只会出现在赋值语句的左侧；

它们的名称会因重新编号而在不同版本之间有所差异，但其右侧表达式在语义上是等价的。我们说的是语义等价而非文本等价，因为某些右侧表达式会使用之前引入的临时变量作为操作数，而该临时变量可能已经被重命名。不过，由于我们的算法按顺序处理语句序列并逐步构建映射，因此，右侧表达式中出现的任何临时变量，都已经在同一序列中更早的语句里建立了映射条目。在此映射下，右侧表达式可确认是等价的，因此左侧临时变量也能得到正确映射。这正是该方法可靠性的体现：生成的每一对映射都对应于同一逻辑程序点上计算出的相同值。

贡献。

（1）一种适用于 Jimple 中间表示的源锚定临时变量映射算法，可在连续程序版本之间建立临时变量的一一映射，从而直接减少不精确的变更集原本会迫使增量分析器进行的不必要重新计算（第 3 节）。（2）通过三个公理、三个辅助引理和一个主定理对该映射进行形式化可靠性证明，保证每个生成的映射对在语义上都是正确的（第 4 节）。（3）基于 BUGSWARM 数据集中 8 个包含真实编辑的 Java 项目进行实证评估，结果表明，与 Reviser图差分算法相比，我们的算法平均可将中间表示变更集减少 82% 以上（所有情况下的召回率为 = 1.0），同时在每个项目上于 250 毫秒以内完成完整流程——源代码差分、已编辑方法识别和临时变量映射——直接反驳了更精确的变更集所需时间多于保守变更集这一假设（第 5 节）。

论文组织结构。 第 2 节回顾相关工作。第 3 节介绍问题陈述和算法。第 4 节证明可靠性。第 5 节报告实验结果。第 6 节总结全文。

2 相关工作

2.1 增量程序分析

IR 级标识不精确这一普遍问题促使人们提出了一系列增量分析策略，而这些策略在如何（或是否）推导精确变更集方面各不相同。一些工具获取两个程序版本，并通过比较它们的中间表示或抽象语法树表示来推导变更集。Reviser [3] 将其图差分算法 [2] 应用于两个版本中每个方法的控制流图（控制流图），通过基于字符串的比较匹配节点的 Jimple 语句表示；语句字符串不匹配的节点构成变更集。随后，Reviser 通过沿控制流图后继关系从变更集出发，获取所有传递可达的节点，从而推导出 Reviser 的受影响区域；受影响区域中的节点会被重置并安排重新计算。Reviser 遵循一种清除并传播的理念 [3]，有意接受过近似变更集，以换取简单的基于字符串的差分步骤——其隐含假设是，识别精确变更集所需的计算量将多于采用保守方法所需的计算量。为了便于本文中的比较，我们单独提取并重新实现了 Reviser 的图差分组件 [2],；该组件负责变更集计算，也是直接受临时变量重新编号影响的部分。当源代码级编辑导致 Jimple 重新为临时变量编号时，基于字符串的语句匹配会将每条重新编号的语句都视为不匹配，从而使变更集本身膨胀——甚至在应用任何可达性步骤之前就已如此。由于 Reviser 的受影响区域是通过可达性从该变更集推导出来的，这种膨胀还会在后续处理中进一步叠加。我们的实验结果直接反驳了 Reviser 的设计假设：在全部 8 个评估项目上（第 5 节），我们基于映射的方法都产生了比 Reviser图差分算法严格更精确的变更集，同时完成时间显著更短。

truediff [13] 同样接收两个源程序版本，计算由新增和删除的抽象语法树节点组成的编辑脚本作为变更集，并将其传递给 IncA 增量 Datalog 框架 [29]，用于推导受影响区域并进行重新计算。

一些增量分析工具通过其他方式获得变更，因此完全避开了变更集识别。使用变更表达式 [32]或在可变程序表示上运行的工具——例如需求式抽象解释 [28],，它直接编辑实时的需求式抽象解释图（DAIG）——会将变更隐含在编辑操作本身中。经典的增量分析工作（如 Ryder 和 Paull [25]、Pollock 和 Soffa [24],）以及 IncIDFA [23],，都假定变更集由上游组件在外部提供。对于所有这些工具，根据两个程序版本推导变更集均不属于其明确的工作范围。

2.2 中间表示映射

Ghiya 等人 [17] 将中间表示映射形式化为增量静态分析的一项明确前提，并针对 COBOL 程序使用专有的中间表示。在他们的设定中，每个中间表示对象都对应一个有名称的源实体——变量、函数或表达式——并在不同程序版本中保留该名称。他们的匹配算法利用了这种稳定性：对于文本未发生变化的源代码行，中间表示对象按照名称、类型和出现顺序进行匹配，其正确性证明建立在这样一个事实之上：对于未发生变化的实体，这些属性保持稳定。

由于一个根本原因，该方法不适用于 Java/Jimple 场景：Jimple 生成的临时变量不带有源代码派生的名称。在 v_1 中，临时变量 $stack7 在 v_2中会变为 $stack8，用于执行相同的计算——这并不是因为程序发生了变化，而是因为方法局部计数器因方法其他位置的编辑而发生了偏移。基于属性的匹配会找不到名称匹配，从而无法生成映射；更糟糕的是，还可能因计数器的偶然重叠而生成错误的映射。Ghime 等人完全避开的这种不稳定性，恰恰是我们场景中的核心障碍。

Nichols 等人 [22] 通过对 IR 实体对之间进行基于编辑距离的评分，并将低于经验校准阈值的匹配判定为有效，解决了增量 JavaScript 分析中的 IR 实体映射问题。他们的方法能够处理函数重命名，但无法保证生成的映射正确，因为错误匹配会导致缓存事实从语义不同的程序点被转移到新版本中的错误位置。

本文探讨 Java/Jimple 环境下的 IR 映射，其中基于属性的匹配和基于阈值的启发式匹配均不适用。我们的解决方案以 Soot 的源代码行标签和 Jimple 的确定性中间表示生成为映射依据，从而能够在单条语句级别对生成的临时变量进行可靠且不依赖阈值的映射。

2.3 源代码级差分

Myers算法 [21], 由 java-diff-utils 实现，可高效计算文本行序列之间的最小编辑脚本。它将代码视为纯文本，忽略语法和语义结构，并构成我们源代码级差分阶段的基础。

基于令牌的工具（如 MOSS [1, 26]）通过令牌序列的哈希指纹检测结构相似性。基于AST的工具——GumTree [14],、GumTree Simple [15],、HyperAST [19], 、HyperDiff [20], 和 ChangeDistiller [16]——通过比较语法树结构，提供细粒度的源代码级变更检测。尽管这些工具能够生成精确的源代码级差异，但它们识别的是源代码构造级别的变更，并不建立 IR级对象对应关系。因此，它们无法直接支持复用在 Jimple 临时变量等 IR 实体上计算的数据流事实。

2.4 字节码级差分

SootDiff [12] 和 jNorm [27] 处理字节码相似性与等价性分析——通过比较两个字节码工件，确定它们在功能上是否等价，可用于代码克隆检测或编译器输出验证等任务。SootDiff 证明，在 Jimple 中间表示级别进行差分比直接处理原始 JVM 字节码更加稳健，并将 Jimple 用作规范化层。jNorm 实现了与编译器无关的字节码规范化，能够处理不同 Java编译器引入的差异。两种工具都将 Myers算法 应用于中间表示语句序列，因此会遇到相同的基于字符串的匹配脆弱性：临时变量重新编号会导致整个方法后缀被归类为已更改。两种工具都不是为增量分析而设计的，也都不提供版本之间的变量级映射。

3 方法

3.1 问题陈述

定义 1（临时变量映射）。给定 *Java*程序的两个连续版本 v_1 和 v_2 及其对应的 *Jimple* 中间表示 IR_1 和 IR_2, , 令$\mathcal{T}_1$ 和 $\mathcal{T}_2$ 分别表示 IR_1 和 IR_2 中的临时变量集合。临时变量映射是一个部分单射：

$$\text{tmpVarMap} : \mathcal{T}_1 \rightharpoonup \mathcal{T}_2$$

使得 $\text{tmpVarMap}(t_1) = t_2$ 当且仅当 t_1 和 t_2 在语义上等价时成立——它们在各自版本的同一逻辑程序点保存了同一子表达式的结果。

可靠映射使增量分析框架能够将与 t_1 在 v_1 中相关联的任意分析事实 f直接传递到 v_2 中的 $\text{tmpVarMap}(t_1)$，而无需重新计算。

我们要求映射是可靠的：tmpVarMap 中的每一对（t_1, t_2）都必须语义等价。映射若不可靠，增量分析器就会悄无声息地将为 t_1 计算的数据流事实传递给语义不同的临时变量 t_2，并将其传播到所有依赖的程序点，导致抽象状态出错。随后，增量分析将产生与对 v_2 进行从头分析所得结果不一致的结果，违反增量分析的一项基本正确性要求：从头分析一致性 [18]。

3.2 概述

我们的方法分为两个阶段：

阶段 *1*——源代码级差异分析。 我们使用源代码级差异工具（java-diff-utils[1], , 一个 Myers算法的 Java 实现 [21]）对旧版和新版源文件进行处理。该工具生成两项输出：（i）一个行号映射，记为 lineNoMap，将 v_2 中的每个未变更行号与其在 v_1中对应的行号关联起来；（ii）两组变更行号——删除列表，记为 *delList*，包含存在于 v_1 但不存在于 v_2中的行号；以及插入列表，记为 *insrtList*，包含存在于 v_2 但不存在于 v_1 中的行号。修改被视为先删除、后插入。

阶段 *2*——临时变量映射。 我们重新编译已编辑的源代码，以获取更新后的 Jimple 中间表示并识别已编辑方法。利用 lineNoMap 作为对齐锚点，我们并行遍历已编辑方法的中间表示语句，并根据源代码行标签对齐中间表示语句。对于经 lineNoMap 确认源自未变更源代码行的中间表示语句，我们尝试映射其左值临时变量。源自 删除列表、*insrtList* 中所列行或同时源自两者的中间表示语句将被跳过，不进行映射。

3.3 算法

算法 1 对阶段 2 进行了形式化描述。

算法 1 的执行过程如下。首先，它调用源文件 v_1 和 v_2上的 Compute- LineMapping，生成 lineNoMap。随后，它调用计算变更，将变更的行号分类到删除列表 和*insrtList*中，并调用获取已编辑方法，利用这两个列表识别已编辑方法。然后，从 IR_1 和 IR_2 中获取已编辑方法的中间表示语句，分别存入$stmts_1$ 和$stmts_2$, 并初始化两个索引 i 和 j，以并行遍历这些列表。

在 while 循环的每次迭代中，算法获取位置 i 和 j 处当前的中间表示语句 s_1 和 s_2，并通过 srcLineNo 获取其对应的源代码行号 l_1 和 l_2。核心逻辑分为四个分支：

- 未变更行——如果 $\text{lineNoMap}[\ell_1] = \ell_2$，则这两条语句源自文本上相同的源代码行。此时会对 (s_1, s_2) 调用 MapTemp- Vars，并同时推进 i 和 j。
- 已修改行——如果 $\ell_1 = \ell_2$ 和 $\ell_1 \in$ 删除列表 以及 $\ell_2 \in$ *insrtList*，则表示该行在两个版本之间已被修改。与该行号关联的所有中间表示语句——包括携带标签 -1 的任何语句（表示没有直接源代码行对应关系的编译器生成的语句）——都会在两侧分别通过独立推进 i 和 j 来跳过。
- 已删除行——仅存在于 $\ell_1 \in$ 删除列表 中时，该行存在于 v_1 而不存在于 v_2。索引 i 向前推进，以跳过旧侧与该行关联的所有中间表示语句。

[1] https://github.com/java-diff-utils/java-diff-utils

算法 1 通过源代码行锚点进行临时变量映射 要求：源文件 v_1、v_2；Jimple 中间表示 IR_1, IR_2确保：tmpVarMap：两个版本之间的临时变量映射 1：lineNoMap ← 计算行映射(v_1, v_2) 2：(delList, insrtList) ← 计算变更(v_1, v_2) 3：mthd ← 获取已编辑方法(delList, insrtList) 4：stmts$_1$ ← IR_1.获取 IR 语句(mthd) 5：stmts$_2$ ← IR_2.获取 IR 语句(mthd) 6：$i \leftarrow 0$；$j \leftarrow 0$7：当 $i <$ \|stmts$_1$\| 且 $j <$ \|stmts$_2$\| 时8 $s_1 \leftarrow$ $_1[i]$ $s_2 \leftarrow$ $_2[j]$：stmts；stmts：源代码行号()；源代码行号() 9 $\ell_1 \leftarrow s_1$. $\ell_2 \leftarrow s_2$. 10：如果 lineNoMap$[\ell_1] = \ell_2$ 则 11：{未变更行：映射临时变量} 12：MapTempVars(s_1, s_2) 13：$i \leftarrow i + 1$；$j \leftarrow j + 1$14：否则如果 $\ell_1 = \ell_2$ 且 $\ell_1 \in$ delList 且 $\ell_2 \in$ insrtList，则 15：{已修改行：跳过两侧} 16：当 $i <$ \|stmts$_1$\| 且 (stmts$_1[i]$.源代码行号() $= \ell_1$或 stmts$_1[i]$.源代码行号() $= -1$) 时 17：$i \leftarrow i + 1$ 18：结束当 19：当 $j <$ \|stmts$_2$\| 且 (stmts$_2[j]$.源代码行号() $= \ell_2$或 stmts$_2[j]$.源代码行号() $= -1$) 时 20：$j \leftarrow j + 1$ 21：结束当 22：否则如果 $\ell_1 \in$ delList，则 23：{已删除行：推进旧侧} 24：$i \leftarrow i + 1$25：否则如果 $\ell_2 \in$ insrtList，则 26：{插入行：推进新侧} 27：$j \leftarrow j + 1$28：结束如果 29：结束当 30：过程 MapTempVars(s_1, s_2)：31：如果 s_1 是赋值语句且 s_2 是赋值语句，则 32：lhs$_1$ ← s_1.获取左操作数().转换为字符串() 33：lhs$_2$ ← s_2.获取左操作数().转换为字符串() 34：如果 IsTempVar(lhs$_1$) 且 IsTempVar(lhs$_2$)，则 35：tmpVarMap[lhs$_1$] ←lhs$_2$36：结束如果 37：结束如果

- 插入行——仅当 $\ell_1 \in insrtList$ 时，该行存在于 v_2 中而不存在于 v_1中。索引 j 向前推进，以跳过新侧与该行关联的所有中间表示语句。

在 MapTempVars 中，算法检查 s_1 和 s_2 是否都是赋值语句（AssignStmt）。如果是，则将它们的左侧操作数分别提取到 lhs_1 和 lhs_2中。如果 IsTempVar 确认二者都是临时变量（变量名以 $ 开头），则记录映射条目 tmpVarMap[lhs_1] ← lhs_2 。

语言结构的处理。该算法无需针对每种语言结构进行单独处理。由于它依据源代码行标签而非控制流结构来锚定对齐，同一遍遍历即可处理分支、循环和异常处理程序：它们的主体像其他语句一样携带行标签，因此当其源代码行未发生变化时，算法会对其进行映射，否则便跳过。

Lambda 表达式和匿名内部类在字节码中会分别成为独立的方法和类，每个方法作用域都可以直接应用算法 1。需要注意的是：计数器还会根据位置为这些合成方法和类分配名称，因此添加一个 lambda 会使其后续对象的索引发生偏移；而我们在识别经过编辑的作用域时，为临时变量解决的不稳定性也会再次出现。目前，我们尚未在不同版本之间映射合成方法标识；将源代码锚定方法扩展到这一标识是未来的工作。

MapTempVars 调用的正确性保证。MapTempVars 仅从由条件 lineNoMap$[\ell_1] = \ell_2$（算法 1 第 10 行）保护的分支中调用。只有当 l_1 和 l_2 经 java-diff-utils 确认为文本上相同时，该条件才会满足，正如 lineNoMap 中所记录的那样。其余所有分支——修改行分支会跳过行号同时出现在 *delList* 和 *insrtList*中的语句；删除行分支会跳过行号仅出现在*delList*中的语句；插入行分支会跳过行号仅出现在*insrtList* 中的语句——都会推进索引 i 和 j，而不会调用 MapTempVars，也不会在 tmpVarMap 中记录任何条目。因此，tmpVarMap 中记录的每一对（t_1，t_2）都唯一来源于已确认未发生变化的中间表示语句，这正是第 4 节可靠性论证的基础。

复杂度。该算法恰好遍历一次语句$_1$ 和语句$_2$，因此时间复杂度为 $O(|\text{语句}_1| + |\text{语句}_2|)$——与已编辑方法的中间表示大小呈线性关系。此界成立，是因为四个分支中的每一个都会至少推进 i 或 j，从而确保循环终止，且每条语句最多处理一次。源代码级差异（阶段 1）在最坏情况下通过 Myers算法运行于 $O(NM)$ 时间，但对于典型的 Java文件，实际运行通常只需数毫秒。

输出用途。生成的临时变量映射是增量分析框架的直接输入：对于与 v_1 中临时变量 t_1 相关联的任何数据流事实 f，该框架都会将 f 转移到 v_2 中的临时变量映射(t_1)，而无需重新计算。

4 映射的可靠性

现在我们证明，算法 1 生成的每一对 (t_1, t_2) 都是正确的映射——也就是说，t_1 和 t_2 在同一逻辑程序点处保存着同一子表达式的结果。

4.1 公理

公理 1（源代码差异正确性）。*java-diff-utils* 能正确识别未更改的源代码行。行 ℓ_2 v_2 中的ℓ_1 通过 lineNoMap$[\ell_1] = \ell_2$ 与 v_1 中的行相关联，当且仅当 ℓ_1 中的行的文本内容与 v_2 中的行 ℓ_2 的文本内容相同（不计装饰性空白）。

公理 2（Soot 行标签忠实性）。*Soot* 的源代码行标签忠实地记录了来源。如果一条 *Jimple* 语句 s 携带标签 ℓ，则 s 由源代码行 ℓ生成。

公理 3（未变更行的确定性中间表示生成）。对于在 v_1 和 v_2 中文本内容相同的任意源代码行，*Soot* 会生成相同的 *Jimple* 语句序列——数量相同，

语句类型相同，顺序也相同。在该序列中，每个临时变量都恰好在某个赋值语句的左侧被引入一次，随后可能作为同一序列中后续赋值语句右侧的操作数出现。每条语句中的所有非临时操作数（源代码命名变量、常量、方法签名）在不同版本之间都在文本上相同。版本之间唯一的差异在于临时变量的名称，这些名称由方法局部计数器分配，而该计数器可能因方法其他位置的编辑而发生偏移。

公理 1 是 java-diff-utils 的一项性质。公理 2 是 Soot 前端的一项性质。公理 3 源于编译流水线的确定性：使用相同的编译器和设置编译相同的源文本，会为相应的方法生成相同的 JVM 字节码；而 Soot 的 Jimple 生成器会将相同的字节码确定性地提升为相同的 Jimple 语句序列。唯一的变化来自方法局部临时变量索引计数器；该计数器的作用域限定于方法，可能会因方法体其他位置的编辑而偏移，从而产生不同的临时变量名称，而生成语句的所有其他方面均保持不变。

4.2 引理

引理 1（未更改行的结构同一性）。令 $\ell_1 \in v_1$ 和 $\ell_2 \in v_2$ 满足 $\text{lineNoMap}[\ell_1] = \ell_2$。令 $\mathcal{S}_{\ell_1}$ 和 $\mathcal{S}_{\ell_2}$分别表示由 ℓ_1 在中间表示$_1$ 中以及由 ℓ_2 在中间表示$_2$ 中生成的 ***Jimple IR*** 语句序列。那么：

(i) $|\mathcal{S}_{\ell_1}| = |\mathcal{S}_{\ell_2}|$——两个序列长度相同；**(ii)** 对于每个位置 k，$\mathcal{S}_{\ell_1}$ 和$\mathcal{S}_{\ell_2}$ 中第 k条语句具有相同的语句类型；**(iii)** 对于位置 k 上的每条赋值语句，左侧操作数仅在临时变量名称上有所不同；并且**(iv)** 对于位置 k上的每条赋值语句，右侧表达式在针对位置1、……、$k-1$，基于 ℓ_1 和 ℓ_2构建的部分映射临时变量映射 下在语义上等价：右侧表达式中的所有非临时操作数在不同版本间文本上相同，并且右侧表达式中出现的任何临时操作数 $t \in \mathcal{T}_1$ 都满足临时变量映射$[t] = t'$，其中 t' 是中间表示$_2$中对应的操作数。

证明。根据公理 1，第 ℓ_1 行和第 ℓ_2 行的文本内容完全相同。根据公理 2，所有标记为 ℓ_1（分别为 ℓ_2）的中间表示语句均由该相同源文本生成。根据公理 3，Soot 会根据相同的源文本确定性地生成相同的结构化 IR。

断言 *(i)* 和 *(ii)* 直接由公理 3 得出：相同的源代码文本会以相同的顺序生成数量相同且类型相同的语句。

断言 *(iii)* 由公理 3 得出：左侧唯一的变化是方法局部临时变量计数器，它会因方法其他位置的编辑而发生偏移，但不会影响语句结构。

断言 *(iv)* 通过对 S_{ℓ_1} 和 S_{ℓ_2} 中的位置 k 进行归纳法证明。

基本情况 ($k = 1$)：第 ℓ_1 行第一个赋值语句的右侧不包含此前在该行引入的临时变量。因此，所有操作数都是源代码命名变量、常量或方法签名；根据公理 3，它们在文本上相同。因此，该右侧与空的临时变量映射语义等价。

归纳步骤 ($k > 1$ $k' < k$)：假设对于所有位置，命题 (iv) 均成立。考虑第 k - 个赋值语句的右侧。根据公理 3，任何非临时变量操作数在文本上都相同。该右侧中的任何临时变量操作数$t \in \mathcal{T}_1$，都是在同一源代码行上的某个位置 $k' < k$ 的左侧引入的 ℓ_1（因为 Jimple临时变量具有方法级作用域，并由赋值语句引入）。根据在 k' 处应用的命题 (iii)，中间表示$_2$ 中的对应左侧临时变量是临时变量映射$[t] = t'$。根据在 k'处的归纳假设，t 和 t' 在语义上等价——它们保存的是同一子表达式的结果。因此，通过临时变量映射将t 替换为 t'，可得到在语义上等价的中间表示$_2$, 右侧，从而证明位置 k处的命题 (iv)。□

引理 2（映射仅限于未变更行）。对于算法 *1* 中调用 *MapTempVars* 的每一对 （s_1、s_2），s_1 和 s_2 均源自经 lineNoMap 确认为未变更的源代码行。

证明。检查算法 1。MapTempVars 仅在第 12 行被调用，而该行只能从第 10–13 行的分支到达，该分支受条件 $\text{lineNoMap}[\ell_1] = \ell_2$ 约束。根据公理 1，当且仅当第 ℓ_1 行和第 ℓ_2 行的文本完全相同时，该条件成立。所有其他分支——修改行分支（第 14–21 行）、删除行分支（第 22–24 行）和插入行分支（第 25–27 行）——都会推进索引 i 和/或 j，但不会调用 MapTempVars。□

引理 3（并行遍历保留位置对应关系）。调用 *MapTempVars*（s_1、s_2）时，对于相同的索引 k，s_1 和 s_2 分别是其对应（映射的）源代码行所生成的第 k 个 ***Jimple*** 语句。

证明。每当遇到映射行时，算法会在第 13 行以锁步方式推进 i 和 j。非映射分支则独立推进 i 和 j，以跳过已删除、插入和修改的区域。完成所有跳过操作后，当第 10 行的条件再次满足时，i 和 j 都指向各自下一个映射行对中尚未处理的第一个语句。

根据引理 1，对于映射行对（ℓ_1、ℓ_2），两个中间表示序列具有相同的长度和顺序。由于两个索引分别从 ℓ_1 和 ℓ_2 对应序列的开头开始，并在第 13 行同步递增，因此，对于该映射行对，MapTempVars 的第 k 次调用会从两侧分别接收第 k 个语句。□

4.3 主定理

定理 1（临时变量映射的可靠性）。对于算法 *1* 生成的每个映射（$t_1 \mapsto t_2$） $\in$临时变量映射，t_1 和 t_2 在语义上等价：在 v_1 和 v_2 中对应的逻辑程序点处，它们持有相同子表达式的结果。

证明。设 $(t_1 \mapsto t_2) \in$临时变量映射。此映射是在过程 MapTempVars(s_1, s_2) 的第 30–37 行中生成的，因此 t_1 = 左侧$_1$ 且 t_2 = 左侧$_2$。

步骤 *1*。根据引理 2，只有因为 $\text{lineNoMap}[\ell_1] = \ell_2$，($s_1$, s_2) 才会被传递给 MapTempVars；也就是说，行 ℓ_1 和 ℓ_2 在文本上完全相同。

步骤 *2*。根据引理 1(ii)，s_1 和 s_2 是同一类型的赋值语句（在 MapTempVars 中的 IsTempVar 检查处也得到了确认）。根据引理 1(iii)，它们的左侧操作数仅在临时变量

名称上有所不同，因此 t_1 和 t_2 是结构上对应的赋值中的左值临时变量。根据引理 1(iv)，在为 ℓ_1和 ℓ_2上所有前置位置构造的部分临时变量映射下， s_1的右侧表达式与 s_2 的右侧表达式语义等价：非临时操作数在文本上相同，并且 s_1 右侧表达式中的任何临时操作数都已通过临时变量映射映射到 s_2 中语义等价的对应项。因此，s_1 和 s_2 计算出相同的值，并且 t_1 和 t_2 在对应的程序点持有该值。

步骤 3. 根据引理 3， s_1 和 s_2 分别是各自映射后的行序列中索引相同的语句，即它们表示同一个计算步骤。

结论。 t_1 保存了 v_1中程序点p_1 处表达式 e 的结果； t_2 保存了 v_2中对应程序点 p_2 处相同表达式 e 的结果。因此， t_1 和 t_2 语义等价，且该映射是可靠的。

由于这一论证适用于临时变量映射中的每个条目，因此整个映射都是可靠的。□

推论1（安全的分析事实转移）。设 f 是与临时变量 t_1 相关联的分析事实，存在于针对 v_1的数据流分析结果中。在定理 *1* 的条件下，增量分析框架无需重新计算，即可在 v_2 中将 f 与临时变量映射(t_1) 相关联，且这一转移在语义上是正确的。

证明。由定理 1 可直接得出：由于临时变量映射 tmpVarMap(t_1) 在对应的程序点计算出的值与 t_1 相同，因此，任何基于该值推导出的分析事实（例如非空性、常量性、类型边界）同样适用于二者。□

4.4 范围与局限性

可靠性证明以公理1–3为前提。实际中：

- 对于文本上相同但因上下文不同而语义不同的代码行，公理 1 可能不成立（例如，两个不同方法中都出现的 return x 行被归类为未更改）。由于我们的映射仅适用于已编辑方法，这一问题得到缓解。
- 对于某些会合并或拆分源代码行的 Soot转换，公理 2 可能不够精确。在评估中，我们没有发现此类情况，但这仍然是一项理论局限。
- 对于标准的 Soot 编译，公理 3 成立。执行非确定性转换的第三方Soot插件可能会违反该公理。
- 新增或删除方法。当在 v_2中新增一个方法时，它在 v_1中没有对应项；算法 1 不会在临时变量映射中为该方法生成条目，增量分析器会从头计算其事实，这是安全的。当在 v_2中删除一个方法时， v_1中该方法的缓存事实将变得无法访问，因此不会尝试进行转移。源代码级别的方法重命名会被视为删除旧方法并新增新方法，并根据同一论据以保守的方式处理。在这三种情况下，算法都不会生成错误的映射——对于受影响的方法，临时变量映射为空这一保守结果能够保证从头分析一致性。同样，重构期间在类之间移动的方法会被视为从源类中删除并在目标类中新增；不会尝试进行映射，并且能够保持从头分析一致性。

该算法采用保守策略：只要存在任何疑问（缺少行标签、存在非确定性中间表示），相关语句就不会建立映射，而是触发重新计算，以避免错误地复用结果。

5 评估

5.1 实验设置

我们使用 Soot[2] 框架实现了算法 1，以生成中间表示。我们采用了 Soot 的 Jimple 中间表示，并使用标准编译设置；未使用任何 Soot 插件或非标准转换。对于源代码级差异比较，我们使用 java-diff-utils[3], , 这是一个以 Java 编写的 Myers算法 [21]实现。每项实验最多分配 2GB 堆内存，并在一台配备 4 个 Intel Core i5-1145G7 核心（8 个逻辑处理器）、运行 Windows 11 和 OpenJDK 17 且拥有 16 GB RAM 的机器上执行。

我们在 8 个来自 BUGSWARM 数据集 [30], 的真实开源 Java 项目上进行了评估。该数据集提供程序的成对失败（有错误）和修复后 版本，以及自动化构建脚本。项目规模从 285 个类到 2345 个类不等（表 2）。我们选择 BUGSWARM，是因为其中包含真实世界的修改，而非人为构造的修改，因而能够提供符合实际的变更类型分布，包括单条语句插入、条件修改和空值检查添加 [33]。我们确认，Soot 会为所有受测方法中分支、循环和异常处理程序的主体生成源代码行标签。对于那些修改涉及添加或删除 Lambda表达式或匿名内部类的版本对，我们予以排除，因为目前我们尚未跨版本映射合成方法标识。

对于每个失败/通过配对，我们都运行了我们的映射器和 Reviser图差分算法；该算法依据 [2] 中的伪代码实现。该算法以旧版和新版程序版本中每个方法的控制流图作为输入，并对二者进行并行遍历。两个控制流图中的节点通过对其 Jimple 语句表示进行基于字符串的比较来匹配：当且仅当两个节点的语句字符串在文本上相同时，才认为它们等价。未匹配的节点会被分类为新增或删除，同时也会记录新增/删除的边。为了进行比较，我们将新增/删除节点的行号存储为变更集；该变更集仅通过直接的基于字符串的节点匹配来确定，在推导任何基于可达性的受影响区域之前完成，以确保与我们的映射器生成的变更集进行公平比较。所有报告的时间均不包括字节码加载和控制流图构建。我们的映射时间涵盖整个流程：在项目范围内应用 java-diff-utils、识别经过编辑的方法，以及执行映射本身。

5.2 研究问题

RQ1（效率）： 我们的映射算法是否足够快速，能够作为增量分析工作流中的实际组件？我们将运行时间与 Reviser图差分算法进行比较，以其作为基准。

RQ2（变更集大小缩减）： 与 Reviser 的图差分算法相比，我们的映射能将被错误报告为已更改的源代码行数量减少多少？变更集越小

[2]https://github.com/soot-oss/soot [3] https://github.com/java-diff-utils/java-diff-utils

表 2：数据集特征描述。按项目-版本对报告编辑类型、已编辑方法作用域以及已编辑方法中的临时变量数量。

项目	#方法数 已编辑	编辑类型	方法大小 （中间表示语句）	临时变量数 在方法中
一个 [4]	1	空值检查插入	48	31
b [8]	1	语句修改	14	7
c [5]	1	条件修改	116	70
d [7]	1	空值检查插入	113	50
e [6]	1	空值检查插入	494	366
f [9]	2	空值检查插入	11	4
		语句删除	52	27
g [11]	1	空值检查插入	27	12
h [10]	1	空值检查插入	282	170

#mthds 已编辑：包含实际编辑的方法数量；方法大小：已编辑方法作用域中的中间表示语句总数；*#tmps*：已编辑方法中的临时变量数量。

就意味着需要重新分析的中间表示语句越少，也能安全复用更多分析事实，而无需重新计算。

5.3 指标

- **映射器时间（秒）：** 完整映射流程（java-diff-utils、已编辑方法识别和临时变量映射）的实际运行时间总计。
- **Reviser图差分算法时间（秒）：** Reviser图差分算法的实际运行时间总计。
- **效率提升（%）：**

$$\frac{\text{Reviser图差分算法时间} - \text{映射器时间}}{\text{Reviser图差分算法时间}} \times 100$$

- **映射器变更集（M-CS）：** 我们的算法报告为已更改的源代码行数。
- **Reviser变更集（R-CS）：** Reviser图差分算法报告为已更改的源代码行数。
- **变更集大小缩减（%）：**

$$\frac{|\text{R-CS}| - |\text{M-CS}|}{|\text{R-CS}|} \times 100$$

由于我们的方法和 Reviser图差分算法都会遍历所有确实发生变更的行，且二者都不会产生假阴性（如第 5.5 节所述，经人工验证），因此，这一缩减直接对应于假阳性的消除——即由于临时变量重新编号而被错误报告为已变更的源代码行。因此，召回率 = 1.0 对于所有项目中的两种方法均为 1.0。我们不将 F1分数作为主要指标报告，因为在两种方法的召回率均固定为 1.0 的情况下，F1 会单调地退化为仅由精确性决定的函数，除了变更集大小缩减之外不再提供任何额外信息。我们直接测量变更集大小；不会单独测量增量分析器根据变更集推导出的受影响区域。由于受影响区域是从变更集推导出来的（例如通过可达性），更小的变更集必然保证更小的受影响区域，因此，所报告的变更集缩减是对无效重复计算实际缩减幅度的保守下界。

5.4 结果

表3展示了逐项目结果。

表3：效率（RQ1）和变更集大小缩减（RQ2）结果。在两种方法的召回率 = 1.0 均相同的情况下，变更集大小缩减直接对应于假阳性消除。

项目	#cl.	#m.	映射（秒）	修订（秒）	EG（%）	M-CS	R-CS	CSR%
a [4]	782	30	0.241	1.243	81	3	52	94
b [8]	288	2	0.210	0.397	47	1	2	50
c [5]	930	24	0.259	1.864	86	1	36	97
d [7]	616	40	0.211	1.088	81	4	7	43
e [6]	285	199	0.155	0.882	82	13	219	94
f [9]	332	25	0.244	0.700	65	7	65	89
g [11]	2345	4	0.223	9.380	98	2	30	93
h [10]	2326	121	0.289	6.978	96	1	25	96
Mean			**0.229**	**2.816**	**80**	**4.0**	**54.5**	**82**

#cl.：类数量；*#m.*：映射数量；*Map.*：映射器时间；*Rev.*：Reviser图差分算法时间；*EG*：效率提升；*M-CS*/*R-CS*：映射器/Reviser图差分算法的变更集大小；*CSR%*：变更集大小缩减比例。

研究问题*1*——效率。 我们的平均映射耗时为每个项目 0.229 秒，而 Reviser图差分算法为 2.816 秒，平均效率提升达 80%。项目 g（2345 个类）的运行时间节省高达 98%。值得注意的是，我们的运行时间基本上不受应用程序规模影响：对于项目 g 和 h（每个项目均超过 2300 个类），我们的耗时分别为 0.223 秒和 0.289 秒，而 Reviser图差分算法分别需要 9.38 秒和 6.978 秒。这种独立性源于算法 1 仅遍历已编辑方法作用域内的中间表示语句（通过删除列表和*insrtList*识别），而非整个类层次结构。虽然我们规模最大的项目包含 2345 个类，但我们的方法的运行时间取决于已编辑方法作用域的规模，而非代码库总规模，这表明该方法可扩展至更大的项目；在企业级代码库上进行实证验证仍是未来的工作。

研究问题*2*——变更集大小缩减。 我们的方法平均可将变更集缩减 82%（Reviser图差分算法生成的变更集为 54.5 行，而我们的映射器生成的变更集为 4.0 行）。由于两种方法的召回率均为 = 1.0，这一缩减意味着直接消除了临时变量重新编号引入的假阳性。在项目 e 中，对于仅有 13 行实际修改的程序，Reviser图差分算法报告了 219 行变更，膨胀了 17 倍；而我们的方法恰好报告了 13 行。

我们平均 4.0 行的变更集，相较于 Reviser图差分算法的 54.5 行，意味着未更改区域的分析状态复用率有望提升 >90%。我们生成的所有变更集都是 Reviser图差分算法生成的变更集的真子集，这证实不存在假阴性。手动验证协议详见第 5.5 节。

分析 *Reviser*图差分算法 的过度报告。Reviser图差分算法过度报告的来源包括：(a) 临时变量重新编号（这是主要原因，我们的映射已解决此问题），以及 (b) Reviser图差分算法归类为编辑的表面元数据或构建编号更新。我们的源代码级差异过滤器也会消除后一类情况，因为元数据更改通常发生在单独的文件中，不会出现在 lineNoMap 中。

表 4：项目 c 中的代表性示例。唯一的语义变化是第 118 行插入了空值检查。Reviser 的图差分算法报告了 36 行变更，这是由于临时变量重新编号所致；我们的方法报告了 1 行变更。为便于阅读，省略了类路径。

Java 代码——失败版本

```
...
101  返回      转换为规则匹配数组(ruleMatches);
...
117  消息.获取字符串("简短拼写描述"));
118  if(
       用户配置.获取最大拼写建议数()                    == 0
       || 规则匹配.size()
       <= 用户配置.获取最大拼写建议数())
{
119 列表    建议             = 获取建议(word);
...
```

Java 代码——修复版本

```
...
101  返回      转换为规则匹配数组(ruleMatches);
...
117  消息.获取字符串("简短拼写描述"));
118  if(
       用户配置       == null    ||
       用户配置.获取最大拼写建议数()                    == 0
       || 规则匹配.大小()
       <= 用户配置.获取最大拼写建议数())
{
119 列表    建议   =          获取建议(word);
...
```

Jimple 中间表示 — 失败版本

```
...
101  $stack79 = 虚拟调用
         $r3.转换为规则匹配数组($stack13)
101  返回      $stack79
...
116 $stack32 =$r11.消息
117  $stack35 = 虚拟调用
         $stack32.获取字符串("拼写")
117 $stack33 =$r12.消息
117  $stack34 = 虚拟调用
         $stack33.getString("简短拼写描述")
117  特殊调用          $u1.<初始化 >(
         $r13, 句子      , len,
         $stack36,  $stack35,  $stack34)
118 $stack37 =$r14.用户配置
118  $stack38 = 虚拟调用
         $stack37.获取最大拼写建议数()
118 如果 $stack38 == 0  goto
118  $stack56 = 接口调用              $stack13.size()
118 $stack54 =$r15.用户配置
118  $stack55 = 虚拟调用
         $stack54.获取最大拼写建议数()
118 如果 $stack56 >$stack55      goto
119  $stack63 = 虚拟调用
         $r16.获取建议(单词)
...
```

Jimple 中间表示 — 修复版本

```
...
101  $stack80 = 虚拟调用
         $r3.转换为规则匹配数组($stack13)
101  返回      $stack80
...
116 $stack32 =$r11.消息
117  $stack35 = 虚拟调用
         $stack32.获取字符串("拼写")
117 $stack33 =$r12.消息
117  $stack34 = 虚拟调用
         $stack33.获取字符串("简短拼写描述")
117  特殊调用          $u1.<初始化 >(
         $r13, 句子      , len,
         $stack36,  $stack35,  $stack34)
118 $stack37 =$r14.用户配置
118如果 $stack37 == 空      goto
118 $stack53 =$r15.用户配置
118  $stack54 = 虚拟调用
         $stack53.获取最大拼写建议数()
118 如果 $stack54 == 0  goto
118  $stack67 = 接口调用              $stack13.size()
118 $stack65 =$r16.用户配置
118  $stack66 = 虚拟调用
         $stack65.获取最大拼写建议数()
118 如果 $stack67 >$stack66      goto
119  $stack57 = 虚拟调用
         $r17.获取建议(word)
...
```

图差分 CS: {101，.....118，119，120，.....}，总大小 = 36**映射器 CS:** {118}，总大小 = 1 **映射:** {.....，$stack79 → $stack80，$stack32 → $stack32，$stack33 → $stack33，$stack34 → $stack34，$stack35 → $stack35，$stack63 → $stack57，.....}

5.5 手动验证协议

为验证我们的方法没有遗漏任何真实的代码变更，两位作者分别独立检查了每一对失败/通过版本。对于每一对版本，他们通过检查两个版本在源代码级别上的 Git 差异，确定真实变更集，找出所有存在语义修改的行（排除空白、注释和仅涉及元数据更改的情况）。随后，将映射器报告的变更集与这一真实变更集进行比较。

在全部 8 个项目中，映射器的变更集严格包含于 Reviser图差分算法的变更集，并且包含所有真实语义变更，从而确认不存在假阴性（召回率 = 1.0）。未发现两名检查人员之间存在任何分歧。此外，我们还手动核验了这 8 个项目中由我们的算法生成的全部 445 个映射并在每种情况下确认两个临时变量都保存了同一个子表达式的结果。

5.6 工作示例

表 4 展示了项目 c 中一个具有代表性的示例。固定版本仅引入了一处语义变更：在第 118 行对 userConfig 进行空值检查。尽管这是唯一有意义的编辑，但文件其他位置的元数据更改导致 Jimple 在整个方法中重新为临时变量编制索引。

我们的方法正确识别出第 118 行是唯一发生更改的行。对于由 lineNoMap 对齐的第 101 行和第 116–117 行，已建立变量映射：第 117 行的 $stack32、$stack33、$stack34 和 $stack35 分别与修复版本中

语义等价的右侧表达式相对应。对于第 119 行（同样已对齐），尽管插入空值检查导致索引发生偏移，映射仍成功将失败版本中的 $stack63 与修复版本中的 $stack57 配对。

Reviser图差分算法缺少这一映射，因此由于临时变量重新编号的级联效应，报告了包含 36 行的变更集——该结果是在执行任何可达性分析之前通过字符串比较直接发现的。由于 Reviser 的受影响区域是通过控制流图可达性从变更集中推导出来的，膨胀至 36 行的变更集（而非真正的 1 行）直接抵消了增量分析的效率优势；一旦推导出受影响区域，浪费的重新计算所带来的实际成本很可能会超过 36 行这一数值。我们的方法不仅生成了精确的变更集 {118}，而且耗时显著少于图差分算法（表3），从而反驳了这样一种普遍假设：计算精确的变更集必然比计算保守的变更集需要更多计算量。

5.7 有效性威胁

内部有效性。 主要的内部有效性威胁是重新实现风险：我们重新实现了 Reviser图差分算法，而不是使用原始工件。我们尝试构建公开可用的 Reviser 实现，但无法使其在当前工具链版本下成功构建，因此无法直接与原始系统进行比较。因此，我们严格遵循了 [2] 中的伪代码，使用默认设置且不进行参数调优，并将比较限制在变更集计算上——这是受临时变量重新索引影响的组件——而不是完整的增量求解器。

外部有效性。 我们的比较使用 BUGSWARM 数据集，而不是原始 Reviser 评估中的目标程序。无法按照本比较所需的粒度复现该评估：目标应用虽已列出名称，但未注明具体的版本对，而且报告的测量涵盖求解器更新时间，却未单独说明图差分步骤的开销。由于变更集大小和图差分时间取决于编辑发生在方法中的位置，而不仅仅取决于代码库本身，因此，在这些项目上使用不同提交运行我们的映射器，并不能构成基于原始基准的比较；同时，也没有已发表的图差分时间可供对照。因此，我们重新实现了 Reviser图差分算法的差分组件，并在相同的版本对和相同条件下测量了两种方法。

构念效度。 我们使用变更集大小缩减作为增量复用潜力的替代指标；实际端到端加速取决于客户端分析，本文未对此进行测量。这个替代指标并非凭空设想：在一项配套工作中，我们针对若干相同的研究对象提交开展了指针分析，观察到未映射的临时变量会增加变更事实，并使其出现在编辑从未影响的代码部分，浪费代码审查者的检查精力。因此，精确映射既决定了增量分析所执行的重新计算量，也决定了其报告的变更事实的准确性。此外，我们关于召回率 = 1.0 的主张是通过人工检查而非形式化判定标准得出的，且受第 5.5 节所述验证协议的局限性影响。

6 结论

我们提出了一种系统且可靠的算法，用于在连续程序版本之间映射 Jimple 中间表示中的临时变量，这是 Java 生态系统实现高效增量数据流分析的基本前提。核心挑战在于，Jimple 会自动生成临时变量，而这些变量的名称本质上并不稳定：任何语句的插入或删除都会引发连锁重编号，因此基于字符串的中间表示匹配不适用于这种场景，因为它会将名称发生变化但语义未变的语句判定为已修改。

我们的算法通过以 Soot 嵌入的源代码行标签为锚点来解决这一问题：当确认源代码行未发生变化时，就将其关联的中间表示语句视为对应语句，并根据左侧对齐建立临时变量映射。我们形式化地提出了三条公理，用以刻画 java-diff-utils、Soot 的行标签生成以及 Soot 的确定性中间表示降级的性质；证明了三个辅助引理，并确立了一条可靠性定理，保证生成的每个映射在语义上都是正确的。一个推论进一步确认，分析事实可以沿映射安全传递，而无需重新计算。

在采用 BUGSWARM 数据集中真实编辑的 8 个真实世界 Java 项目上进行评估后，我们的方法相比 Reviser图差分算法，平均将有效变更集缩减了 82%，同时在 250 毫秒内完成整个流程。由于增量分析器必须重置并重新计算的受影响区域是根据变更集得出的，因此，更精确的变更集会带来相应更小、且很可能缩小幅度更大的受影响区域，从而进一步提升精确 IR 映射的实际效益。这些结果也驳斥了这样一种假设：精确的变更集比保守的变更集需要更多计算。我们的方法同时实现了更高的精确性和更短的计算时间。这表明，精确、可靠且轻量的 IR 映射并非只是实现细节，而是高效增量分析的重要基础。作为未来工作，我们计划测量端到端增量分析加速比。由于目前无法获得可正常运行的 Reviser 原始实现，这需要实现一种可集成该映射的增量数据流分析，我们正在另行开展这项工作。

参考文献

[1] Alex Aiken. 1994年。《用于检测软件抄袭的系统——MOSS》。http: //theory.stanford.edu/~aiken/moss/。在线资源；访问于 2026-04-20。[2] Steven Arzt 和 Eric Bodden。2013年。响应增量程序变更，高效更新基于 *IDE* 的数据流分析。技术报告 TUD-CS- 2013-0253。达姆施塔特工业大学。https://www.bodden.de/pubs/ TUD-CS-2013-0253.pdf [3] Steven Arzt 和 Eric Bodden。2014年。Reviser：响应增量程序变更，高效更新基于 IDE-/IFDS 的数据流分析。载于第 *36* 届国际软件工程会议（*ICSE '14*）论文集。ACM，288–298。doi:10.1145/2568225.2568254[4] BugSwarm。2026。工件 Adobe-Consulting-Services-acs-aem-commons-374231969。https://bugswarm.org/dataset BugSwarm 数据集。访问于 2026-04-20。[5] BugSwarm。2026。工件 languagetool-org-languagetool-393031702。https://bugswarm.org/dataset BugSwarm 数据集。访问于 2026-04-20。[6] BugSwarm。2026。工件 orbit-orbit-237244974。https://bugswarm.org/ dataset BugSwarm 数据集。访问于 2026-04-20。

[7] BugSwarm。2026。工件 petergeneric-stdlib-290369132。https://bugswarm.org/dataset BugSwarm 数据集。访问于 2026-04-20。[8] BugSwarm。2026。工件 SpigotMC-BungeeCord-130330788。https://bugswarm.org/dataset BugSwarm 数据集。访问于 2026-04-20。[9] BugSwarm。2026。工件 square-okhttp-99350732。https://bugswarm.org/dataset BugSwarm 数据集。访问于 2026-04-20。[10] BugSwarm。2026。工件 yamcs-yamcs-186324159。https://bugswarm.org/ dataset BugSwarm 数据集。访问于 2026-04-20。[11] BugSwarm。2026。工件 yamcs-yamcs-218364663。https://bugswarm.org/ dataset BugSwarm 数据集。访问于 2026-04-20。[12] Andreas Dann 和 Eric Bodden。2019。SootDiff：跨不同 Java编译器的字节码比较。载于第 8 届 ACM SIGPLAN 程序分析前沿国际研讨会论文集（SOAP '19）。ACM，13–18。doi:10.1145/3315568.3329964[13] Sebastian Erdweg、Tamás Szabó 和 André Pacak。2021。简洁、类型安全且高效的结构差分。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集。406–419。[14] Jean-Ré my Falleri、Morane Gruhier、Marc Dirix 和 Xavier Blanc。2014。细粒度且精确的源代码差分。载于第 29 届 ACM/IEEE 自动化软件工程国际会议论文集（ASE ' *14* 313 324 10 1145 2642937 2642982 ）。ACM，–。doi: 。/ 。[15] Jean-Remy Falleri 和 Matias Martinez。。2024。细粒度、精确且可扩展的源代码差分。载于 IEEE/ACM 第 46 届国际软件工程会议论文集。1–12。[16] Beat Fluri、Michael Wuersch、Martin Pizka 和 Harald C. Gall。2007年。变更提炼：用于提取细粒度源代码变更的树差分。《IEEE软件工程汇刊》33，11（2007），725–743。doi:10.1109/ TSE.2007.70731[17] Vaidehi Ghime、Ankita Khadsare、Anushri Jana 和 Bharti Chimdyalwar。2022年。IR 映射：基于中间表示（IR）的映射，以支持增量静态分析。载于第 15 届软件工程创新会议论文集。1–5。[18] Anushri Jana 和 Uday Khedker。2025。增量数据流分析的共同脉络：综合综述。《计算调查》58，4（2025），1–36。[19] Quentin Le Dilavrec、Djamel Eddine Khelladi、Arnaud Blouin 和 Jean-Marc Jézéquel。2022。HyperAST：支持大规模高效分析软件历史。载于第 37 届 IEEE/ACM 自动化软件工程国际会议论文集。1–12。[20] Quentin Le Dilavrec、Djamel Eddine Khelladi、Arnaud Blouin 和 Jean-Marc Jézéquel。2023。HyperDiff：大规模计算源代码差异。载于第 31 届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集。288–299。[21] Eugene W. Myers。1986年。一种 O(ND) 差分算法及其变体。《算法》 1, 1 1986 251 266 10 1007 01840446 ( )，–。doi: 。/BF[22] Lawton Nichols、Mehmet Emre 和 Ben Hardekopf。。2019。用于增量 JavaScript 分析的定点复用。载于第 8 届 ACM SIGPLAN 程序分析前沿国际研讨会论文集。2–7。[23] Aman Nougrahiya 和 V Krishna Nandivada。2025。IncIDFA：一种高效通用的增量迭代数据流分析算法。《ACM编程语言与系统汇刊》9，OOPSLA1（2025），617–648。[24] Lori L Pollock 和 Mary Lou Soffa。1989年。一种迭代数据流分析的增量版本。《IEEE软件工程汇刊》15，12（1989），1537–1549。[25] Barbara G Ryder 和 Marvin C Paull。1988年。增量数据流分析算法。《ACM编程语言与系统汇刊》（TOPLAS）10，1（1988），1–50。[26] Saul Schleimer、Daniel S Wilkerson 和 Alex Aiken。2003年。Winnowing：用于文档指纹识别的局部算法。载于 2003 年 ACM SIGMOD 数据管理国际会议论文集。76–85。[27] Stefan Schott、Serena Elisa Ponta、Wolfram Fischer、Jonas Klauke 和 Eric Bodden。2024年。用于代码相似性分析的 Java 字节码规范化。载于第 38 届欧洲面向对象编程会议（ECOOP 2024）。Schloss Dagstuhl–Leibniz-Zentrum für Informatik，1–37。[28] Benno Stein、Bor-Yuh Evan Chang 和 Manu Sridharan。2021。需求式抽象解释。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集。282–295。[29] Tamás Szabó、Sebastian Erdweg 和 Gábor Bergmann。2021。基于格的 Datalog 增量全程序分析。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集。1–15。[30] David A Tomassi、Naji Dmeiri、Yichen Wang、Antara Bhowmick、Yen-Chuan Liu、Premkumar T Devanbu、Bogdan Vasilescu 和 Cindy Rubio-González。2019。BugSwarm：挖掘并持续扩充可复现故障与修复数据集。载于 2019 年 IEEE/ACM 第 41 届国际软件工程会议（ICSE）。IEEE，339–349。[31] Raja Vallée-Rai、Phong Co、Etienne Gagnon、Laurie Hendren、Patrick Lam 和 Vijay Sundaresan。2010。Soot：Java 字节码优化框架。载于 CASCON 首个十年高影响力论文集。214–224。[32] Jens Van der Plas、Quentin Stiévenart、Noah Van Es 和 Coen De Roover。2020 年。通过计算依赖具体化实现增量流分析。载于*2020 IEEE* 第*20*届国际源代码分析与操作工作会议（*SCAM*）。IEEE，第25–36页。[33] Xizao Wang、Xiangrong Bin、Lanxin Huang、Shangqing Liu、Jianhua Zhao 和 Lei Bu。2025 年。实际环境中的增量程序分析：关于现实世界程序变更的实证研究。载于*2025* 年第*40*届 *IEEE/ACM* 国际自动化软件工程会议（*ASE*）。IEEE，第2350–2362页。