通过大语言模型驱动的科学调试实现可解释自动调试

Sungmin Kang*
sung
min.kang@kaist.ac.kr
KAIST 韩国大田

Bei Chen
bei.chen
@microsoft.com 微软亚洲研究院 中国北京

Shin Yoo
s
hin.yoo@kaist.ac.kr
KAIST 韩国大田

Jian-Guang Lou
jlou@microsoft.com 微软亚洲研究院 中国北京

arXiv:2304.02195v1 [cs.SE] 5 Apr 2023

摘要

自动化调试技术有潜力减少开发者在调试中的工作量，并且已经足够成熟，可以被工业界采用。然而，现有技术的一个关键问题是，尽管开发者希望获得所提供的自动调试结果的理由，但现有技术不适合提供这些理由，因为它们的推理过程与人类开发者的推理过程有显著差异。受开发者在调试时与代码交互方式的启发，我们提出了自动化科学调试（AutoSD），这是一种技术，给定含缺陷代码和揭示缺陷的测试，提示大语言模型自动生成假设，使用调试器主动与含缺陷代码交互，从而在补丁生成之前自动得出结论。通过使自动化调试的推理更紧密地与人类开发者的推理对齐，我们旨在生成关于特定补丁如何生成的可理解解释，希望该解释能带来更高效和准确的开发者决策。我们对三个程序修复基准测试的实证分析表明，AutoSD 与其他程序修复基线相比具有竞争力，并且它可以指示何时对其结果有信心。此外，我们进行了一项包含 20 名参与者（包括六名专业开发者）的人类研究，以评估来自 AutoSD 的解释的实用性。有解释访问权限的参与者判断补丁正确性的时间与没有解释的参与者大致相同，但他们在所研究的六个真实世界中的缺陷中的五个上的准确性有所提高：70% 的参与者回答说他们在使用修复工具时想要解释，而 55% 的参与者回答说他们对科学调试的呈现感到满意。

CCS 概念

• **软件及其工程 → 软件测试与调试**.

关键词

自动化程序修复，机器学习

1 引言

自动化调试技术（例如故障定位（FL）或自动化程序修复（APR））旨在通过自动化部分调试流程来帮助开发者。鉴于其重要性开发者在调试上投入了大量精力，自动化调试已成为一个备受关注的研究课题 [41], : 每年都有许多论文发表 [20]，而且该领域已足够成熟，开始被业界采用 [25], 。

关于这些技术的实际应用，许多调查开发者对自动化调试期望的文献一再指出，软件工程任务的优异表现固然重要，能够帮助开发者判断结果的辅助信息也同样重要。例如，Kochhar 等人 [17] 对开发者在故障定位方面的期望进行了研究，发现超过 85% 的开发者认为提供理由的能力很重要。此外，Kirbas 等人 [16] 指出，一些开发者对自动程序修复的结果持负面态度，称这些结果会“凭空出现”。这些发现表明，自动化调试取得优异结果可能还不足以令人接受，还可能需要提供辅助信息来帮助解释 这些结果。

尽管人们一再要求自动化结果的生成过程具有可解释性，但据我们所知，可解释自动调试技术仍不常见。例如，在 Monperrus 于 2022 [25], 年 8 月更新的 APR 持续综述中，“解释”一词仅出现在一篇立场论文中 [24], ，这表明如何向开发者解释修复建议这一关键研究领域尚未得到充分探索。我们认为，部分原因在于现有自动化调试技术的推理方式与人类截然不同。现有自动化调试技术会缩小搜索空间 [10] ，并尝试多种解决方案，以找到与缺陷位置及其修复方式相关联的结果 [26], ；而人类开发者通常会使用调试器和打印语句与含缺陷代码交互，了解其行为，并据此根据观察结果生成补丁 [32]。也就是说，现有自动化调试过程的推理轨迹 [19] 与开发者的推理轨迹大相径庭，因此，提供这些轨迹可能无助于理解生成的补丁。

作为迈向能够生成有助于开发者的解释的自动化调试技术的一步，我们提出了 AutoSD，它弥合了人类与自动化调试过程之间的鸿沟。为此，AutoSD 利用大语言模型（LLM）和调试器接口，自动模拟 Zeller [41] 为开发者提出的科学调试（SD）过程。AutoSD 会提示大语言模型自动生成关于导致

*本工作是在微软亚洲研究院实习期间完成的。

缺陷成因的假设，并生成用于检验这些假设的调试器脚本。随后，AutoSD 执行建议的调试器命令，并将结果提供给大语言模型；基于此，大语言模型最终判断假设是否成立，并预测调试过程是否已完成，或是否需要进一步调查。由此生成的中间调试文本可以自然地作为 解释 呈现，说明 AutoSD 是如何得出结论的。模拟科学调试具有实现可解释调试的理想特性：值得注意的是，已有研究指出，即使没有接受过正式培训，开发者也会运用科学调试的原则进行调试 [32], ，因此这些解释有助于启发或辅助开发者的思考过程。

我们首先在三个程序修复基准测试上评估 AutoSD，并据此对其进行实证评估。结果表明，AutoSD 的修复效果可与不具备解释能力的自动程序修复技术相媲美。在实际使用中，精确度是一个重要因素 [39]；我们发现，对于 AutoSD 表示已收集到足够调试信息的情况，其修复性能确实更高。随着语言模型能力不断增强，AutoSD 的修复性能也迅速提升，展现出其潜力。此外，我们在真实的自动程序修复应用场景下，对 20 名 Python 开发者进行了用户研究，其中包括六名专业开发者；参与者需要审查补丁并决定是否接受。结果表明，对于本研究涉及的 83% 真实世界中的缺陷，AutoSD 生成的调试轨迹能提高开发者判断补丁是否正确的准确率，同时基本不影响开发者作出判断所需的时间；这些结果说明，自动生成的补丁解释对人类有所帮助。此外，70% 的参与者表示，在使用自动程序修复工具时，解释会是一个重要因素；55% 的参与者对 AutoSD 的科学调试表述方式表示满意。

总而言之，我们的贡献可概括如下：

- 我们发现，大语言模型通过模拟开发者的工作流程，可以实现可解释自动调试。作为示范，我们提出了 AutoSD，它利用大语言模型模拟科学调试 [41]；
- 我们在三个自动程序修复基准测试上进行了实证分析，结果表明，AutoSD 在自动程序修复方面表现出色，同时还能生成解释；
- 我们基于一个真实的补丁审查场景，对 AutoSD 开展了开发者研究，并证明 AutoSD 生成的解释有助于开发者进行决策；
- 我们还征求了用户对修复解释的反馈，并提出了改进解释的未来指导原则。

本文其余部分安排如下：第 2 节介绍本研究的技术背景，第 3 节介绍我们的技术 AutoSD。第 4 节介绍评估设置和研究问题，第 5 节呈现基于这些实验的实证结果。第 6 节讨论威胁与局限性，第 7 节总结全文。

2 背景

本节介绍我们工作的动机和背景。

2.1 可解释的自动化调试

自动化调试历史悠久，研究通常围绕故障定位 [14, 18, 26] 和自动化程序修复 [7]展开。如前所述，自动化调试技术的技术复杂度和性能不断提升 [12], ，其中包括将大语言模型用于自动程序修复 [11, 37],，但很难找到关于如何向开发者解释结果的实证研究。Monperrus 对自动程序修复的持续综述中仅有一篇论文提及解释 [25], ；Winter 等人 [36] 发现有 17 项人类研究评估了自动程序修复，但没有一项直接涉及自动程序修复工具提供的解释；Kochhar 等人 [17] 调查了当时的故障定位技术，发现有两种技术可以解释其结果 [22, 33]；遗憾的是，这两篇论文都没有开展人类研究。

这与日益增多的文献形成对比：这些文献表明，要在实践中采用自动化调试技术，人们乐于看到对结果的“解释”。开发者曾多次表达他们对解释的需求：除了前文提到的 Kochhar 等人的研究发现 [17] ，Noller 等人关于自动程序修复（APR）期望的开发者研究 [28]指出，“开发者最常提及、认为自动程序修复工具最有帮助的输出是<风格 id='3'>解释</风格>.....包括其<风格 id='5'>根因</风格>”。开发者的期望尤为重要，因为当自动化调试被业界采用时，自动生成的补丁总是会由开发者审查。在 Meta，自动程序修复系统与内部代码审查平台相连 [21]；Kirbas 等人 [16]写道，Bloomberg 认为“完全自动化远非理想”，因此他们会让软件工程师审查自动程序修复补丁。Noller 等人的研究结果也印证了这一点：“要获得开发者的充分信任，必须进行人工补丁审查。”

一种向开发者呈现解释的有效方式，可能是展示工具的推理轨迹 [19] ，即说明自动化调试工具如何得出建议：为联邦学习推荐某一行代码，或为自动程序修复推荐某个补丁。与提交消息生成等事后解释技术不同 [13],，此类推理轨迹能够回答开发者可能提出的关键问题，例如“为什么是这个补丁？”；事实上，人机交互（HCI）研究表明，解释应力求能够回答为什么某种方法会得出特定结果 [19]。

然而，当前的自动化调试技术不适合为其结果生成有用的解释，因为它们的推理轨迹与人类的推理轨迹显著偏离。以自动程序修复技术的一种常见分类 [8] 为例[1]，生成并验证（G&V）技术 [7] （其中包括基于学习的技术 [10, 38, 42]）会生成含缺陷代码的变体，直到测试通过。由于它们的推导过程只是简单地枚举更改并逐一尝试，该过程运行时不考虑任何“根因”。基于语义的自动程序修复技术，如 Angelix [23] ，使用变量值作为可满足性模理论（SMT）求解器的输入，以更有效地在补丁空间内搜索；因此它们本质上也没有识别任何“根因”。这并不是说这些技术在修复缺陷方面无效——大量关于自动程序修复的工作表明，现有的自动程序修复技术可以修复各种各样的缺陷。相反，我们认为，由于它们的推理轨迹与人类如此不同，因此很难

[1]联邦学习的可解释性将在附录中讨论。

对其结果做出令人满意的解释。另一方面，做出令人满意的解释的一种方法是开发一种以类似于人类的方式进行推导的自动化调试技术，以使决策过程透明 [4]。

2.2 科学调试

为了使自动程序修复（APR）的推理轨迹更贴近人类开发者的推理轨迹，我们必须了解开发者在实践中如何调试。关于开发者调试模式的先前研究，让我们得以初步了解调试的实际过程。

早期关于开发者调试的研究发现，开发者遵循一种“粗略描述模型”：他们先提出假设，再验证假设是否成立 [9]。Zeller [41], 将这一过程的正式版本命名为科学调试，并主张开发者维护调试日志，按迭代记录以下各项：

- 假设：对缺陷的一种暂定描述，能够解释该缺陷并与已知观察相符；
- 预测：如果假设成立，预期会出现的结果；
- 实验：用于验证预测的方法；
- 观察：实验的结果；
- 结论：根据观察对假设作出的判断。

Siegmund 等人 [32]发现，即使没有接受过正式的调试技术培训，所有受访开发者也都会大致遵循科学调试的“先提出假设，再进行验证”流程。因此，科学调试可以视为对开发者调试时主导思维过程的正式描述。为此，我们力求在生成自动程序修复（APR）结果时模拟这一过程并给出解释。

2.3 大型语言模型

本文旨在通过大语言模型（LLM）模拟科学调试过程。我们认为，大语言模型能够模拟科学调试，原因如下。首先，它们在涉及推理的问答基准测试中表现日益出色 [2, 29], , 这也使它们有可能预测某个假设是否成立，以及接下来应调查哪个假设。虽然要人工收集大量符合科学调试格式、包含调试轨迹的数据并非易事，但已有研究表明，大语言模型也能够通过少样本或零样本方式解决问题：也就是说，只需给出少量示例，甚至只需通过自然语言提示描述待解决的任务，它们便能够完成该任务 [2]。基于人类反馈的强化学习（RLHF）训练能够提升这一能力 [30], ; 我们的任务所用的主要大语言模型（OpenAI 的 ChatGPT）便接受过此类训练。最后，科学调试要求与代码交互，因此需要使用外部工具。采用“思维链”（CoT）提示时 [34], , 大语言模型似乎也能够利用外部工具的结果来提升表现 [6, 40]。因此，我们认为大语言模型非常适合模拟科学调试过程，从而生成开发者能够理解的推理轨迹。

3 自动化科学调试

我们的方法整体流程如图1所示。首先，生成包含相关信息的提示（图1 ）：其中包括对科学调试的详细解释，以及对调试问题本身的描述，以便 AutoSD 能够执行以下步骤。准备好初始提示后，AutoSD 使用大语言模型生成关于代码问题所在或如何修复代码的假设，并提出用于验证该假设的具体实验（图1 ）。实验脚本将传递给后台调试器／代码执行进程，由该进程运行脚本并返回实际结果（图1 ）。根据观察到的信息，AutoSD 使用大语言模型判断该假设是否得到验证（图1 ）；根据结论，AutoSD 会提出新的假设，或选择终止调试过程并生成修复方案。与代码的交互结束后，AutoSD 会根据收集到的信息生成缺陷修复方案（图1 ）。与我们所知的其他自动化程序修复技术不同，由于执行了步骤（ - ）AutoSD 能够提供特定修复方案的理由，并可应开发者要求将其提供给开发者。

3.1 构建输入提示

为了构建初始提示，如图1 中所示的示例，我们首先手动编写了科学调试的详细描述，解释了假设、预测、实验、观察和结论是什么，并为每个类别提供了多个示例，以便大语言模型能够生成可理解的推理轨迹。完整描述可在附录中找到；在此，我们详细描述对AutoSD 流水线至关重要的描述方面。首先，提供了实验的具体示例，以允许大语言模型预测适当的实验脚本：给出了复合调试器命令（包括设置断点、运行代码和打印值）以及我们定义的用于允许编辑并执行命令的领域特定语言（DSL）。关于DSL，提示指定以下命令可用：
`REPLACE(line, old_expr, new_expr)`用于更改`line`处的表达式，`ADD(line, new_expr)` 用于在`line`上方添加新语句，以及`DEL(line, old_expr)` 用于删除一行中的表达式。多个命令可以用`AND` 连接器连接，最后可以通过`RUN` 命令在修改后执行揭示缺陷的测试。除了实验命令外，提示还指示如果已收集到足够信息以辨别补丁，则预测 `<DEBUGGING DONE>` 令牌（在本文其余部分简称为`<DONE>`），以便我们可以衡量AutoSD 对其补丁的置信度。提示足够详细，使我们的默认大语言模型ChatGPT能够零样本遵循指令，即无需完整过程的具体演示。在此科学调试描述之上，我们添加缺陷特定信息：具体来说，有缺陷的函数/方法、揭示缺陷的测试、执行缺陷时的错误消息，以及如果可用的缺陷报告。我们添加此信息是因为我们相信此类信息对于人类调试问题是必要的（即使不是充分的），因此也可能有助于自动化技术预测适当的假设并最终成功调试。

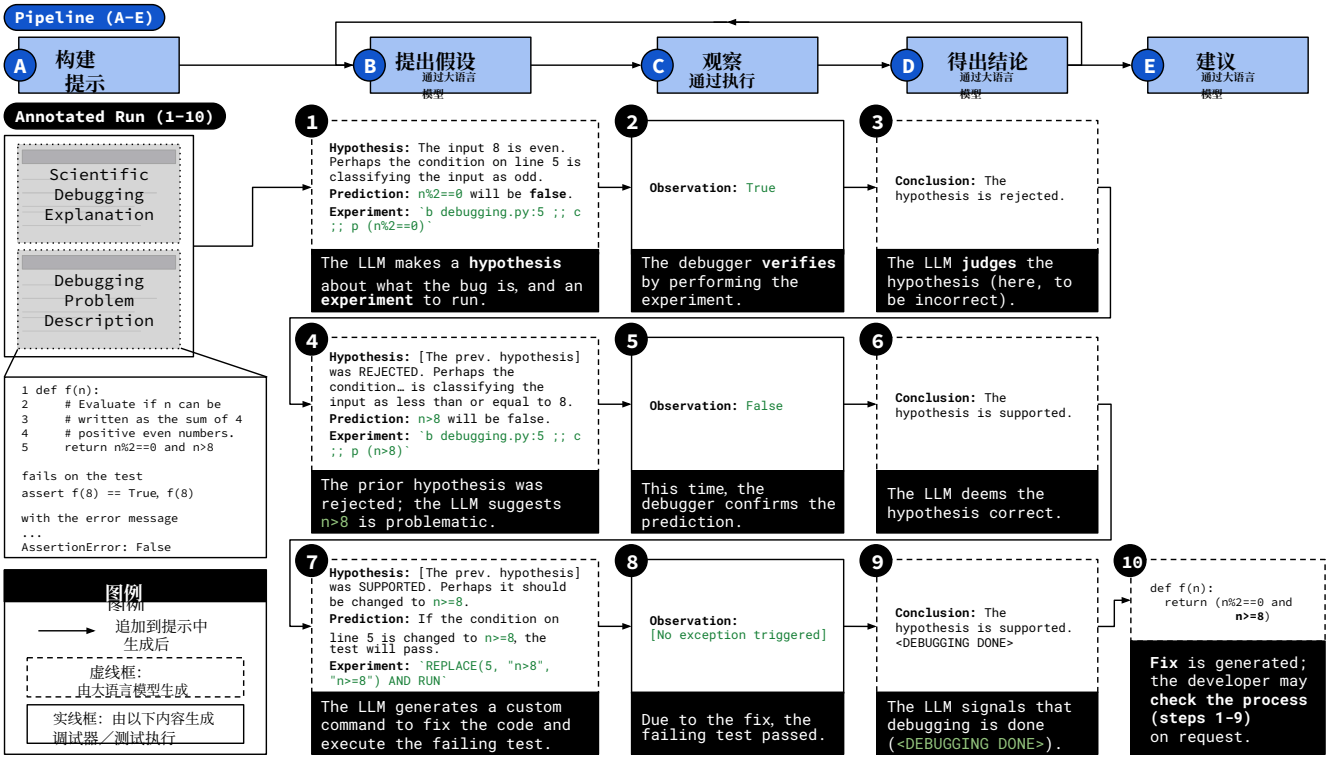


图1：AutoSD 的流程及一次真实运行示例，黑框中的注释为清晰起见经过了轻微编辑。给定科学调试概念的详细描述和缺陷描述（A），AutoSD 将生成关于缺陷内容的假设，并使用大语言模型（B）构建实验以验证该假设；随后使用调试器或代码执行来实际运行实验（C），再由大语言模型根据实验结果判断假设是否正确（D）。假设-观察-结论循环会重复进行，直至大语言模型得出调试结论或达到迭代次数上限；最后生成修复方案（E），并附上解释 （从（1）到（9）的白框），供开发者查看。

3.2 假设-观察-结论

有了初始提示，AutoSD开始迭代‘假设 -
如图1所示的观察-结论’循环（ - ）。每个迭 的 果都会附加到提示中，以便逐步预测假设；也就是说，在 中生成结论时，大语言模型会根据初始提示、 和 的拼接内容进行预测。我们将循环的每次迭代称为一个步骤：例如，图1中的 - 就构成一个步骤。

提出假设。在此，我们通过在提示中追加 tokenHypothesis:，引导语言模型生成一个假设，使其针对缺陷提出假设。我们观察到，由于提示中详细描述了科学调试过程，LLM 还会依次生成 Prediction: 和 Experiment: 行标题。下一步的关键是 Experiment 命令：语言模型要么生成可由调试器执行的调试器命令，要么生成自定义的代码修改与执行脚本，以便“测试”某项更改。由于文档采用 Markdown 格式，Experiment 脚本用反引号括起（`）；系统会从 LLM 输出中提取该脚本，以便在下一步获得具体的代码执行结果。

示例见图 1 、 和 ——注意A T D还会在提出假设的过程中定位故障，因此故障定位过程也具有可解释性。

观察。生成的实验脚本会被传递给一个基于传统软件工程工具的后台进程，该进程会将实际执行结果返回给语言模型，从而使我们能够依据真实结果来引导 AutoSD 的生成过程，并增强向开发者展示时的可信度。模型可以选择：（i）通过设置断点并打印一个值来调用组合调试器命令；或（ii）修改代码，并使用上述 DSL 运行失败测试。执行调试器命令时，系统会通过与语言相对应的调试器命令行界面来执行该命令，并返回组合命令中最后一个子命令的输出（假定为 `print` 命令），如图1 和 所示。当断点位于循环内时，调试器会收集执行过程中不同时间步的值，并将其一并返回，例如：“每次循环执行时，该表达式的值为： [v1、v2、...]”，最多可返回 100 个值。同时，通过编辑并执行 DSL 命令运行测试时，如果引发异常，则会将异常类型和消息作为观察结果返回；否则会附加“[未触发异常]”，如图1 所示。

得出结论。根据观察结果，AutoSD会调用大语言模型，检查假设与观察结果是否一致，让大语言模型判断该假设是被否定（例如 ）、得到支持（例如 ），还是因出现意外观察结果而无法确定。我们让大语言模型生成结论，以最大限度地提高对数值的解读灵活性。如前所述，如果大语言模型判断调试过程已完成，它还可以在此步骤预测一个单独的 <DONE> 标记；在这种情况下，AutoSD 对其输出会更有把握。图1 展示了一个例子：得知此前失败的测试现已通过后，大语言模型得出结论，认为调试已完成。如果预测出 <DONE> 标记，AutoSD 便会按照第 3.3 节生成修复方案；否则，循环会基于新获得的信息重新开始提出假设，直到达到最大迭代次数限制 s。如果届时仍未预测出 <DONE>，AutoSD 就未能识别缺陷原因，我们也可能需要对生成的补丁持更审慎的态度。

3.3 修复建议

当 AutoSD 完成与代码的交互后，我们会评估各项假设的结论，并在生成补丁前自动从提示中移除被否定的假设，因为我们的实验表明，这样做能提高程序修复的性能。即使在实际修复时不使用被否定的假设，仍可将其作为成功假设的背景信息呈现给开发者。随后，我们通过追加以下内容提示大语言模型利用现有信息生成修复方案：“修复后的代码（完整方法，不含注释）如下：\n```”。此提示会引导大语言模型根据问题描述和代码交互中可用的信息生成修复后的代码，如图1所示 。与其他自动程序修复技术一样，最终会生成一个补丁；而 AutoSD 的独特之处在于，它可以展示其中间推理步骤（ - ），作为一种解释，帮助开发者理解补丁的来源。

4 评估设置

本节介绍我们实证评估的设置。

4.1 研究问题

RQ1：可行性。虽然我们的工作主要关注为自动化调试结果生成推理链，但调试任务本身取得良好表现也很重要 [17, 28]。因此，我们试图回答以下问题：AutoSD 的表现是否达到与以往自动程序修复技术相当的水平？与提示大语言模型立即预测修复方案相比（本文其余部分将这一基线称为 LLM-Base），表现又如何？我们旨在证明，AutoSD 的可解释性不会带来显著的性能代价，尽管此前关于可解释人工智能的综述指出，可解释性与性能之间存在权衡 [1]。我们在构建的 Almost-Right HumanEval基准上评估 AutoSD，以缓解对数据泄露的担忧；同时也在由真实世界中的缺陷组成的 Defects4J v1.2 和 2.0 基准测试上进行评估 [15]。

RQ2：调试器消融。针对这一研究问题，我们首先评估：当 AutoSD 用 <DONE> 标记表示调试已完成时，其性能是否更好。由于精确性对于面向开发者的实用工具至关重要，如果 AutoSD 能在很可能正确时作出指示，便有助于开发者采用 AutoSD。基于我们关于置信度与 <DONE> 的实验，我们评估不使用调试器、而由大语言模型“臆造”观察结果（而非通过实际代码执行获得观察结果）时 AutoSD 的性能。我们评估在这种设置下，<DONE> 标记是否仍代表出色的性能。

研究问题 3：改变大语言模型。我们通过更换所用的大语言模型来评估 AutoSD 的性能。虽然我们通过实证发现，使用 ChatGPT 模型时性能最佳，因此在本文其余部分将其作为默认设置，但通过改变语言模型的规模并绘制性能变化，我们考察了随着模型参数规模扩大、训练方法日益复杂，自动修复性能如何变化。

研究问题 4：开发者受益情况。通过人类研究，我们评估开发者是否能从 AutoSD 自动生成的解释中实质性获益，即不论他们对解释持何种看法。在这项人类研究中，参与者会看到含缺陷代码、揭示缺陷的测试、候选补丁，并在一半情况下看到解释，然后判断该补丁是否正确解决了测试所揭示的问题。我们会衡量开发者判断补丁是否正确所用的时间和准确率，并询问他们“解释是否有助于作出判断？”。因此，我们希望评估向开发者提供解释是否能使他们受益。

研究问题 5：开发者接受度。我们通过向开发者提出六个问题，评估他们是否接受 AutoSD 的解释：他们是否愿意使用自动程序修复（APR），使用 APR 时是否希望获得解释，以及他们是否认可 AutoSD 及其解释的各个组成部分。我们希望借此衡量开发者是否愿意使用解释，而非衡量解释是否能提高他们的工作效率。此外，我们还进行了访谈，以了解开发者喜欢 AutoSD 解释的哪些方面，以及哪些方面有待改进。

研究问题 6：定性分析。在这一研究问题中，我们提供了开发者喜欢和不喜欢的补丁尝试及相应解释的示例，作为进一步的背景信息；此外，我们还随机抽取了 25 个案例进行分析。在这些案例中，AutoSD 将自己生成的所有假设都判定为错误。我们据此归纳了常见的失败原因。

4.2 环境

4.2.1 评估自动程序修复性能。为了评估 AutoSD 的性能，我们使用了四个程序修复基准测试。首先，我们采用了广泛使用的 Defects4J 基准测试 [15] 1.2 版和 2.0 版，这两个基准测试已被先前的研究用作比较自动程序修复技术的标准基准测试 [20],。此外，为了在人类研究中使用真实世界的 Python 缺陷进行评估，我们还采用了 BugsInPy 基准测试 [35]（本文简称为 BIP）。但我们不报告 AutoSD 在 BIP 上的性能，因为其中许多缺陷需要额外的环境配置，而 README 中并未对此进行说明。

此外，我们基于 Chen 等人提出的 HumanEval Python 单函数合成基准测试，构建了近乎正确的 HumanEval（ARHE）数据集 [3]。这样做是希望避免数据污染问题，因为 Chen 等人明确表示，创建 HumanEval 的目的之一是在评估其大语言模型时避免数据污染；该基准也被用于评估近期推出的

GPT-4 模型 [29]。ARHE 数据集通过修改 HumanEval 基准中的人工编写解答构建而成，使每个解答恰好有一个测试未通过，从而生成导致代码“几乎”正确的缺陷。我们使用七种变异器，最终得到 200 个缺陷用于评估；数据集按所用变异器划分的详细组成见附录。作为比较，我们还评估了一个基于模板的自动程序修复基线，该基线使用构建数据集时所用变异器的逆操作，并将其随机应用于含缺陷代码。由于该基线具有随机性，我们运行了 100 次。请注意，ARHE 中有 90 个缺陷是通过删除或字符串变异生成的，因此无法由该基线逆转；其余变异均可逆，因此只要有足够时间，我们的基于模板的基线就能修复这些缺陷。

关于具体的自动程序修复参数，对于每个数据集，我们向 AutoSD 提供有缺陷的方法，并生成 10 个补丁，以匹配 Jiang et al. [11],的大规模实证研究中的设置。该研究评估了多种大语言模型和更传统的基于学习的自动程序修复技术的修复性能。我们指出，我们的设置假设掌握的精确信息较少，因此难度更高：Jiang et al. 使用完美的语句级故障定位进行评估，而 AutoSD 使用完美的方法级故障定位和缺陷报告。评估生成的补丁时，我们针对每个缺陷运行各数据集提供的测试；通过所有测试的修复被视为看似合理的 补丁，并会对看似合理的补丁进行人工检查，以确定其在语义上是否等同于开发者补丁。在语义上等同的修复被视为正确。

AutoSD 需要使用大语言模型和调试器。对于大语言模型，我们测试了 CodeGen [27], Codex [3] (code- davinci-002) 和 ChatGPT（InstructGPT [30]的姊妹模型），其中 ChatGPT 大语言模型是默认模型。根据目标语言使用不同的调试器；对于 Java 基准测试（Defects4J v1.2 和 v2.0），我们使用 jdb 工具；对于 Python 基准测试（ARHE 和 BugsInPy），我们使用 pdb 工具。最大迭代次数限制 s 设为 3。

4.2.2 人类研究参数。为了评估 AutoSD 在现实世界中的影响，我们参考自动程序修复在实际应用中的情况 [16, 21]，邀请参与者审查补丁，以开展一项人类研究。我们专门抽取了 12 个缺陷，在这些缺陷中，AutoSD 生成的补丁使最初失败的测试通过：随机抽取.....

个此类 ARHE 数据集中的缺陷示例（其文档完整），以及 BugsInPy Python 数据集中的六个真实世界中的缺陷 [35]。在初步研究中，我们发现审查 12 个补丁可能需要很长时间，因此将 12 个缺陷分成两组，每组六个（各包含三个 ARHE 缺陷和三个 BugsInPy 缺陷），并随机分配参与者，要求他们解决其中一组的代码审查问题。图2展示了向参与者呈现的代码审查界面示意图；调查网站的截图见附录。我们的人类研究获得了 IRB 审查豁免（IRB-23-054）。

对于每个代码审查问题，参与者会获得含缺陷代码、能够揭示缺陷的（失败）测试以及补丁；在随机选出的六个案例中的三个案例里，还会提供解释。解释的每个步骤都有一个标题，用于概述解释该缺陷的假设；标题会根据预测的结论以颜色标示：得到支持、被否定和未定的假设分别标为绿色、红色和黄色，如图2所示。点击每个标题即可查看 AutoSD 的完整推理过程，如图1所示。参与者需要针对每个补丁回答三个问题：（Q1）该补丁是否正确，可回答是、否或不确定（作为代码审查过程中检查正确性的替代指标 [31]）；（Q2）简要说明其对 Q1 的判断，以筛除可能的不诚信回答；（Q3）如果提供了解释，该解释是否有助于作出判断，以衡量解释对不同补丁产生的不同影响。

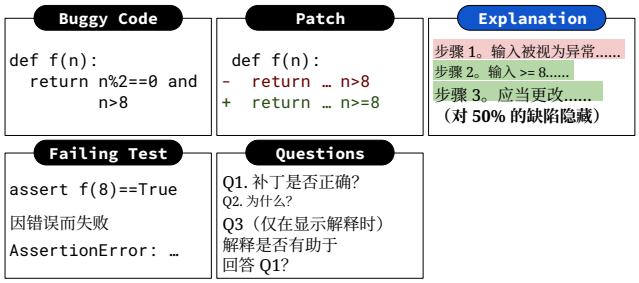


图2：人类研究界面示意图

为招募参与者，我们向本科生和研究生宣传了这项任务，要求他们至少拥有1年的Python经验；此外，我们还招募了一家专注于软件测试技术的公司的专业开发者。我们共招募了20名参与者：8名本科生、6名研究生，以及6名职业生涯跨度为3至10年的专业开发者。参与者首先听取研究任务说明，接着通过解决一道代码审查问题进行练习，然后以随机顺序在30至40分钟内完成6道代码审查题。这6项代码审查任务分别包含针对ARHE和BugsInPy基准测试的2个正确补丁和1个错误补丁。参与者填写事后问卷，回答有关其人口统计特征及对解释总体满意度的问题后，我们对他们进行了约5分钟的访谈，以定性分析他们对该工具的印象。

5 实验结果

下面介绍实证评估的结果。

结果	基于模板	LLM-Base	AutoSD
合理	85.77 ± 4.20	179	189
正确	-	177	187

表1：ARHE基准测试上的修复结果。基于模板的性能基于100次重新运行，并显示均值和标准差修复性能。

5.1 RQ1：可行性

表1展示了 AutoSD 在 ARHE基准测试上的自动程序修复表现，并与 LLM-Base 和基于模板的基线方法进行了比较。需要注意的是，在相同条件下评估时，基于模板的基线方法的修复表现明显弱于 LLM-Base 和 AutoSD；因此，我们

由于正确性的上限是看似合理的补丁数量，因此未评估所生成的数千个补丁的正确性。此外，LLM-Base 和 AutoSD 的表现相近，这表明 AutoSD 在保留大语言模型修复性能的同时，还能够生成解释。

基准测试	Recoder	InCoder	LLM-Base	AutoSD
D4J v1.2	24	41	87	76
D4J v2.0	11	28	110	113

表 2：Defects4J 基准测试上的正确修复结果。Recoder 和 InCoder 的结果引自 Jiang et al. [11]。

表2展示了 AutoSD 在 Defects4J 基准测试上的自动程序修复表现，并与 LLM-Base 以及 Jiang et al. 的实证研究中表现最佳的技术进行了比较 [11]: Recoder（一种基于深度学习的自动程序修复技术） [42], 和经过微调的 InCoder [5], （Facebook 的一种语言模型，该模型使用完美的语句级故障定位结果进行了微调，因此使用的信息比 AutoSD 更精确）。我们发现，与其他基线方法相比，AutoSD 再次表现出竞争力，即使这些方法获得了更具体的信息。作为额外的参考，若与 Xia et al. [37] 报告的 Codex 在 Defects4J 上的修复结果进行比较，我们发现，在该论文的“补丁函数”设置下，AutoSD 在两个基准测试上的表现均优于 Codex；Codex 使用了 200 个候选补丁（而我们使用 10 个），且该设置假定的故障定位条件与我们的设置相同。

RQ1 的回答：与三种修复基准测试中的多种基线方法相比，AutoSD 的程序修复性能具有竞争力。

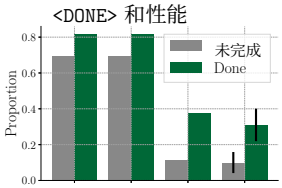


图 3：<已完成> 和性能。

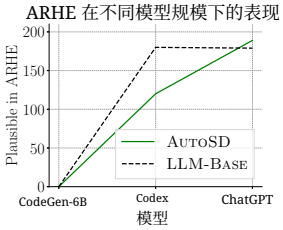


图 4：模型规模

5.2 研究问题 2：调试器消融实验

本研究问题首先探讨， <DONE> 标记预测所表示的结果置信度是否确实与更好的性能相关。结果见图 3。对于 Defects4J，由于无法对该数据集生成的 1045 个看似合理的补丁逐一进行人工标注，因此我们抽取了 100 个包含和不包含 <DONE> 的补丁进行分析。如图所示，对于 ARHE 和 Defects4J 两个数据集，当 <DONE> 标记作为结论的一部分生成时，AutoSD 的精确率都更高，这表明 AutoSD 确实能够提示何时可能生成看似合理或正确的补丁。此外，在生成了看似合理的补丁且预测出 <DONE> 标记的缺陷中，89% 得到了正确修复，而对于生成了看似合理的补丁但未预测出 <DONE> 标记的缺陷，82 % 得到了正确修复。

我们还针对 ARHE 数据集研究了这样一种情况下的性能：调试器/代码执行结果也由大语言模型预测，而非通过实际执行获得；此时，<已完成> 标记是否仍能预测出良好的性能？在这种“调试器幻觉”情形下， <已完成>预测的解决方案成为看似合理补丁的可能性实际低了 11 个百分点；这与使用实际代码执行结果的情况形成对比，后者使 <已完成>预测的解决方案成为看似合理补丁的可能性提高了 12.4 个百分点。此外，单次运行产生看似合理补丁的可能性也大幅降低：AutoSD 的单次运行中有 73% 会生成看似合理的补丁，而禁用调试器后这一比例仅为 63%。因此，纳入代码执行有助于提高 AutoSD 的可靠性；我们将在研究问题 5 中进一步展示，开发者也认为真实的代码执行结果很有用。

RQ2 的回答：AutoSD 可以通过 <DONE> 标记指出其答案更有可能正确，我们也利用这一点来验证使用调试器的效用。

5.3 RQ3：改变大语言模型

在图 4 中，我们展示了使用不同底层大语言模型时 AutoSD 的性能。 x 轴展示了不同的大语言模型，这些模型大致按照参数数量和训练技术的先进程度排序； y 轴则展示了在 ARHE 基准测试上使用相应大语言模型时 AutoSD 的性能。图中还展示了直接请求大语言模型修复缺陷的性能。结果表明，AutoSD 的性能迅速提升，最终与 LLM-Base 的性能相当，这说明使用更强大的语言模型时，AutoSD 的表现更好。对于 CodeGen-6B 等较小的模型，在零样本设置下，修复本身会失败，因为在我们的实验中，它只会返回原始的含缺陷代码。（我们还在少样本设置下评估了 CodeGen-6B，确认模型实现正常；在这种情况下，它能够修复 44 个缺陷。）因此，我们可以推测，随着语言模型不断进步，AutoSD 的性能也会进一步提升。

对研究问题 3 的回答：随着底层语言模型的改进，AutoSD 的性能也随之提升。

5.4 研究问题 4：开发者收益

在本节中，我们评估开发者是否能从解释中受益，同时尽量避免参与者对解释的看法影响评估结果。图 5 展示了代码审查时间、准确性，以及解释是否被评为有助于作出决策的测量结果。

首先，考察解决代码审查问题所需的时间，我们发现，无论是否提供解释，解决问题所需的时间总体上都相近。尽管 AutoSD 提供的解释比不提供解释时包含更多信息，因此可能需要开发者花费更多时间处理，但没有任何情况下的时间差异达到统计显著性。

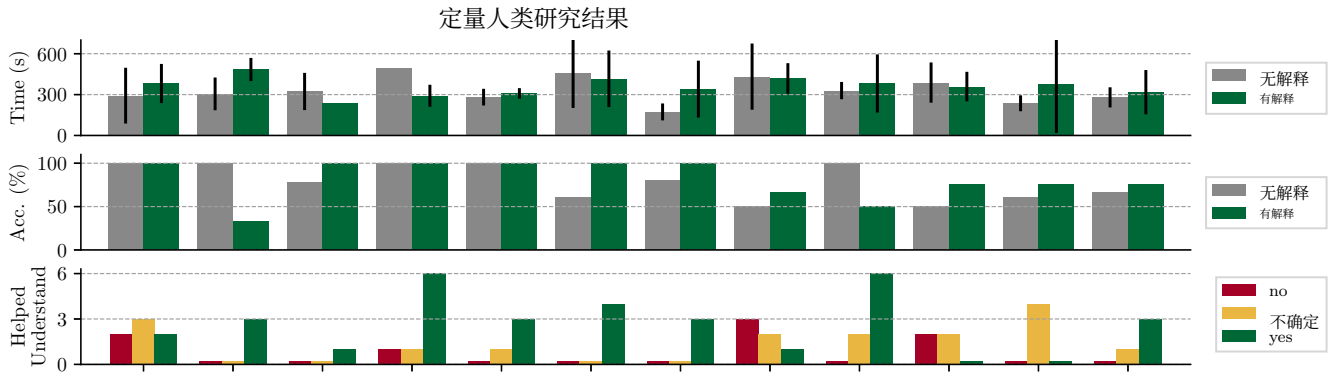


图 5：开发者在代码审查任务中的表现 ，比较使用和不使用 AutoSD 解释时的表现，以及解释评分 gs

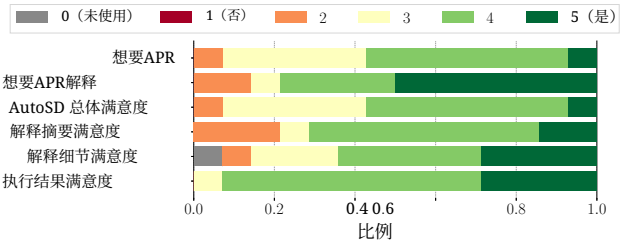
关于有无解释时的准确性，在七种情况下，参与者有解释时解决同一问题的准确性高于没有解释时，其中五种情况集中在真实世界中的 BugsInPy 基准测试中。这些结果表明，在使用自动程序修复时，AutoSD 可能对真实世界中的开发者生产力产生积极影响：评估真实世界中的缺陷时，判断质量有所提高，而开发者花费的时间大致相同。与此同时，有两种情况下，使用解释导致准确性下降：ARHE105 和 BIP003。对于 BIP003，我们发现，受访者查看解释后变得更加谨慎，并回答说他们需要更多信息才能作出判断。对于 ARHE105，回答错误的参与者则未加充分审视，便接受了 AutoSD 的推理。尽管这种情况较为罕见，只出现在随机抽取的 12 个问题之一，但它凸显了进一步研究潜在误导性推理的必要性。此外，对于 BIP 中两个错误的补丁（ BIP002 和 BIP004 ），开发者在有解释时的准确性有所提高，这说明开发者并不会盲目接受附有解释的补丁。

关于参与者是否认为解释有助于他们进行决策，在十二个问题中的八个问题上，开发者指出这些解释确实有助于他们得出结论，这凸显了为补丁提供解释所带来的心理益处。

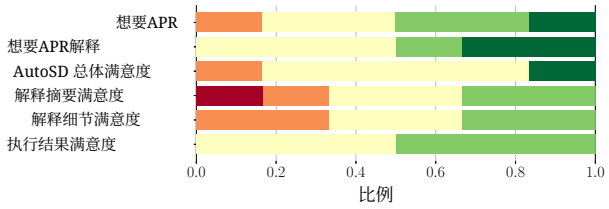
研究问题 4 的答案： 在看到 AutoSD 生成的解释后，人类参与者处理补丁所需的时间大致相同，但在六个真实世界中的缺陷里，有五个缺陷的准确率更高。他们还认为，在所有缺陷中，三分之二的解释具有帮助。

5.5 研究问题 5：开发者接受度

事后问卷的结果见图 6。令我们惊讶的是，学生与专业开发者对 AutoSD 的满意度存在差异：超过一半的学生对 AutoSD 感到满意，而六位开发者中只有一位感到满意。我们借此讨论 AutoSD 生成的解释的优势以及



(a) 学生的满意度。



(b) 专业开发者的满意度。

图 6：按组别划分的人类研究事后问卷结果。

潜在的改进空间。

学生觉得 AutoSD 的解释有哪些吸引人的地方？14 名学生参与者中，有 10 人指出，在无法看到解释时，他们会觉得「少了点什么」。当被问及为何在这些情况下想看到解释，以及在有解释时如何使用解释，学生描述了多种因解释的存在而得到帮助的思考过程。一种常见做法是先独自思考补丁，再将自己的想法与提供的解释进行比较；一位参与者认为解释很有用，因为它可以充当「橡皮鸭」。[2] 另一种常见的用法是查看解释，以判断应将精力集中在哪里，从而引导判断方向。还有一些学生会利用解释来更好地理解

[2]参见 https://en.wikipedia.org/wiki/Rubber_duck_调试。

代码原本打算实现什么。因此，我们认为 AutoSD 生成的解释有一项优势：**它们能够适应多种不同的思考过程**，从而可能帮助各种各样的开发者。

与此同时，另一种使用模式是查看解释中的实验和观察，以具体了解某些时点的取值，并利用这些值建立对缺陷成因的心理模型。这体现了 AutoSD 的另一项优势，即它**在解释中纳入了实际取值**：在图 6 (a) 中，我们注意到，超过 90% 的学生认为，加入执行结果提升了他们对解释的信任。

另一方面，专业开发者对 AutoSD 的解释则表现出较为复杂的态度。值得注意的是，开发者并不反对解释本身：半数受访者同意或强烈同意，在使用自动程序修复工具时，解释很重要（图 6 (b)），这凸显了该问题的重要性。当被问及为何认为 AutoSD 的解释仍有改进空间时，有人建议，如果能将当前的解释与“业务逻辑”或规范联系起来，解释会更有用；一位学生参与者也提出了类似建议。专业开发者认为，若缺少这类联系，就必须严格核验解释，而且即使核验之后，其价值仍然有限。因此，一个可能的改进方向是**将解释与规范文档等现有开发工件整合**。

另一个常见建议是改进该工具的界面：开发人员指出，如果该工具附加到集成开发环境中，他们可能会使用它，并且解释过于冗长。这一反馈表明，为了提高开发人员的满意度，我们可以考虑**将解释集成到开发人员经常使用的平台中**（正如 Kochhar 等人 [17] 所建议的那样），并进一步研究开发人员认为满意的解释的具体细节。

从总体统计来看，我们发现 70% 的参与者同意解释是使用程序修复时的一个重要因素，55% 的人认为科学调试细节（图 6 的解释细节满意度）令人满意，这表明大多数参与者同意 AutoSD 的总体动机和表述。

对研究问题 5 的回答：虽然 AutoSD 的解释能够适应多样化的思维过程，并通过使用具体的执行结果提高开发人员的信任，但可以通过增强界面和链接到规范来进一步改进。

5.6 研究问题 6：定性分析

AutoSD 生成的解释是什么样的？除了图1中嵌入的示例外，我们还提供了 AutoSD 生成的两条推理轨迹，它们分别在来自真实世界 BugsInPy 问题的人类研究中受到喜欢（BIP006 - 75% 喜欢）和不喜欢（BIP002 - 16% 喜欢）。在图7的左侧，我们展示了一个受欢迎的解释，以及一个精简的失败测试和生成的修复。查看补丁，开发人员会看到添加了一个 .lower() 调用；如果没有解释，这个修复可能看起来是虚假的。

相比之下，通过提供 AutoSD 为何关注此区域的理由，参与者可以迅速识别此修复是否与测试相关。例如，Student-6 说“我首先查看了解释，这帮助我确定了要查看代码的哪一部分”。随后的实验确认程序状态中有一个大写的‘分块’头，这是缺陷的来源。这些执行结果帮助参与者理解缺陷，例如 Student-11 指出“表达式值在做出决策时很有用”。总体而言，此补丁是正确的，解释有助于开发人员理解并建立信任。虽然我们提供了来自人类研究的一个简单示例，但我们也注意到 AutoSD 可以处理更复杂的缺陷，如第 5.1 节所示，并在附录中提供了额外示例。

提出假设的尝试也可能失败。图 7 右侧展示了 AutoSD 未能验证任何假设的案例。尽管 AutoSD 最初提出了一个关于以错误顺序追加内容的假设，但实验中建议的代码行实际上并未被覆盖；因此，调试器反馈称该断点未被覆盖。这是最常见的失败原因之一：我们对 25 个所有假设均被否决的案例进行分析后发现，其中 13 个案例里，AutoSD 建议的断点从未被命中，因此 AutoSD 无法获得所生成实验的结果。在 BIP002 的案例中，大语言模型没有寻找测试可能覆盖的新断点，而是开始提出测试有误。最终，尽管生成了一个修复方案，其解释与补丁几乎没有关联，因此人类研究的参与者认为该解释没有帮助；补丁本身看似合理，但同样不正确。尽管如此，这个例子也说明，糟糕的解释仍可能促使人们做出更好的判断：开发者或许会看出补丁依据薄弱，从而（正确地）对补丁更加存疑。在此背景下，值得注意的是，看过 BIP002 解释的开发者对其评估更为准确（图 5）。其他失败模式还包括生成无效的实验表达式（25 个案例中有 2 个），或在实验脚本中添加多条打印命令，而 AutoSD 的基础设施只允许一条打印命令，导致假设被错误否决（25 个案例中有 2 个）。

对 RQ6 的回答：AutoSD 能够针对其生成的补丁提供有帮助的解释，但推理过程也可能失败。常见的失败原因是无法识别正确的断点。

6 讨论

本节介绍我们工作的威胁与局限性。

6.1 对有效性的威胁

内部效度 关注分析是否支持关于因果关系的主张。潜在威胁包括实现错误、补丁正确性评估不准确，以及人类研究中回答可能存在偏倚的风险。为减轻前两项问题的影响，我们计划公开实现和修复结果，以供审查。在人类研究中，除了收集开发者对生成的解释的看法（其中偶尔收到负面反馈）外，我们还发现，参与者在六个 BugsInPy 问题中的五个上准确率有所提高，这一结果很难归因于偏倚。

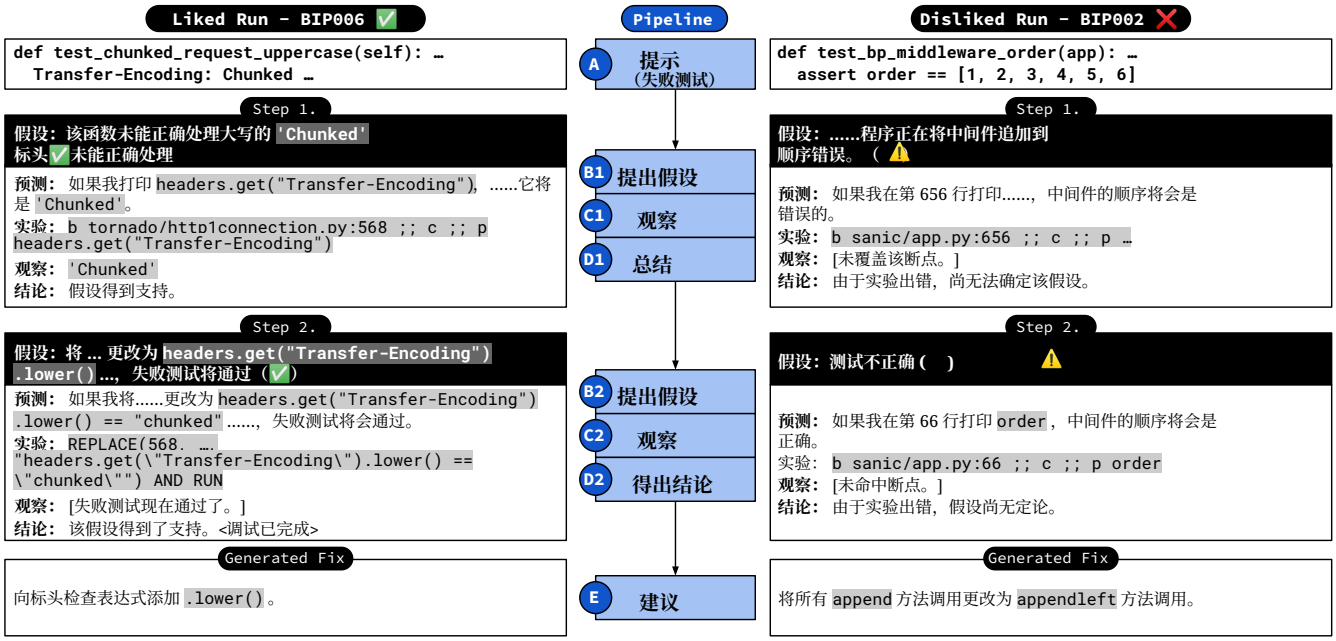


图 7：Exam　ple 成功和不成功的修复以及来自人类研究的 AutoSD 解释。

外部效度 关注本文所呈现的结果能否推广至其他情境。使用大语言模型时，一个特别值得关注的问题是，其训练数据可能包含评估数据中的片段。为缓解这一问题，我们专门构建了用于修复的 ARHE 数据集，并在该基准测试上评估 AutoSD。此外，我们的解释很可能从未出现在训练数据中，因为开发者通常描述代码时，所采用的结构不如科学调试所规定的那样严谨，即使他们的思路与之相近。

6.2 局限性

AutoSD 存在一些我们希望指出的局限性。首先，为了实现与代码的多步交互，必须多次调用语言模型和调试器，这会延长该技术的修复时间；在我们的实验中，与 LLM-Base 相比，AutoSD 生成补丁所需的时间可能约长五倍。尽管如此，鉴于图 6 显示开发者对自动生成补丁的解释有着显著需求，我们认为，为补丁生成解释所需的额外成本是合理的。其次，作为迈向可解释自动调试的一步，我们在已完成方法级故障定位的场景中进行了评估，随后由 AutoSD 以可解释的方式执行语句级故障定位。本文的主要目标是证明 AutoSD 能够生成有助于开发者实际工作的解释；我们希望未来研究可解释的方法级故障定位。另一个相关问题是，目前我们的技术只能处理单方法缺陷；如何整合更广泛的信息来处理更复杂的缺陷，也是一个值得探索的研究方向。最后，生成的解释偶尔可能会让错误补丁显得可信；通过使我们的技术能够指出其输出的置信度，并展示这种置信度与正确性之间的相关性，我们迈出了应对这一问题的第一步。此外，我们的解释包含具体的代码执行结果，有助于开发者进行决策（图 6）。

7 结论

本文总结了先前研究所揭示的解释对于自动化调试结果的重要性，以及目前缺乏能够为人类提供充分解释的自动化技术。我们认为，这是因为现有自动化调试技术无法像人类一样进行推理。为弥合自动与人工调试实践之间的差距，我们使用大语言模型来模拟科学调试过程。我们证明，与其他修复基线相比，AutoSD 能够实现具有竞争力的修复性能，同时具备适合实际应用的优势，例如能够指示输出的置信度。随着语言模型能力增强，AutoSD 的修复性能也有所提升，这表明语言模型越强，性能和解释的可用性可能也会越好。最后，我们的人类研究表明，自动生成的解释可以帮助开发者评估补丁；大多数学生还表示，在无法获取解释时，他们会“想念”这些解释。我们进行的访谈显示，AutoSD 生成的解释能够辅助开发者的多种思维模式，并且可以通过更紧密地融入开发过程来加以改进，例如将其与书面规格说明联系起来。总的来说，我们认为，语言模型能力的快速提升可以用于显著降低开发者使用自动化技术的难度，并希望在未来进一步开发更便于人类使用的自动化调试技术。

参考文献

[1] Alejandro Barredo Arrieta、Natalia Díaz-Rodríguez、Javier Del Ser、Adrien Ben- netot、Siham Tabik、Alberto Barbado、Salvador García、Sergio Gil-López、Daniel Molina、Richard Benjamins 等。2020。可解释人工智能（XAI）：迈向负责任人工智能的概念、分类体系、机遇与挑战。载于《Information Fusion》58（2020），82–115。[2] Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。2020。语言模型是少样本学习器。载于《神经信息处理系统进展》33（2020），1877–1901。[3] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman 等。2021。评估基于代码训练的大语言模型。arXiv预印本 arXiv:2107.03374（2021）。[4] Hoa Khanh Dam、Truyen Tran 和 Aditya Ghose。2018。可解释软件分析。载于《第40届国际软件工程会议论文集：新思想与新兴成果》（瑞典哥德堡）（ICSE-NIER '18）。计算机协会，纽约，美国纽约州，53–56。[5] Daniel Fried、Armen Aghajanyan、Jessy Lin、Sida Wang、Eric Wallace、Freda Shi、Ruiqi Zhong、Wen-tau Yih、Luke Zettlemoyer 和 Mike Lewis。2022。InCoder：用于代码填充与合成的生成模型。arXiv预印本 arXiv:2204.05999（2022）。[6] Luyu Gao、Aman Madaan、Shuyan Zhou、Uri Alon、Pengfei Liu、Yiming Yang、Jamie Callan 和 Graham Neubig。2022。PAL：程序辅助语言模型。arXiv预印本 arXiv:2211.10435（2022）。[7] Luca Gazzola、Daniela Micucci 和 Leonardo Mariani。2019。自动软件修复：综述。《IEEE软件工程汇刊》45，1（2019），34–67。https://doi.org/ 10 1109 2017 2755013 。/TSE。/。Claire Le Goues、Michael [8] Pradel 和 Abhik Roychoudhury。2019。自动化程序修复。《Commun. ACM》62，12（2019年11月），56–65。[9] John D. Gould。1975。关于人们如何调试计算机程序的一些心理学证据。《人机研究国际期刊》7，2（1975），151–182。https://doi.org/ 10 1016 0020 7373 75 80005 8 。/S - ( ) - J. Jiang、Yingfei Xiong、H. Zhang、Q. Gao 和 X. Chen。2018。利用现有补丁和相似代码塑造程序修复空间。第27届ACM SIGSOFT国际软件测试与分析研讨会论文集（2018）。[10] [11] Nan Jiang、Kevin Liu、Thibaud Lutellier 和 Lin Tan。2023。代码语言模型对自动化程序修复的影响。arXiv:2302.05020 [cs.SE][12] Nan Jiang、Thibaud Lutellier、Yiling Lou、Lin Tan、Dan Goldwasser 和 Xiangyu Zhang。2023。KNOD：用于自动化程序修复的领域知识蒸馏树解码器。arXiv: 2302 01857 [ ] 。cs.SE Siyuan Jiang、Ameer Armaly 和 [13] Collin McMillan。2017。使用神经机器翻译根据差异自动生成提交说明。载于《2017年第32届IEEE/ACM国际自动化软件工程会议》（ASE）。 135 146 10 1109 2017 8115626 – 。https://doi.org/ 。/ASE。 。 [14] James A. Jones、Mary Jean Harrold 和 John Stasko。2002。测试信息可视化：辅助故障定位。载于《第24届国际软件工程会议论文集》（美国佛罗里达州奥兰多）（ICSE '02）。计算机协会，纽约，美国纽约州，467–477。[15] René Just、Darioush Jalali 和 Michael D. Ernst。2014。Defects4J：用于对 Java 程序进行受控测试研究的现有故障数据库。载于《2014年国际软件测试与分析研讨会论文集》（美国加利福尼亚州圣何塞）（ISSTA 2014）。计算机协会，纽约，美国纽约州，437–440。https://doi.org/10.1145/2610384.2628055[16] Serkan Kirbas、Etienne Windels、Olayori McBello、Kevin Kells、Matthew Pagano、Rafal Szalanski、Vesna Nowack、Emily Rowan Winter、Steve Counsell、David Bowes、Tracy Hall、Saemundur Haraldsson 和 John Woodward。2021。彭博社引入自动程序修复。《IEEE软件》38，4（ 2021 43 51 10 1109 2021 3071086 ），– 。https://doi.org/ 。/MS。[17] 。Pavneet Singh Kochhar、Xin Xia、David Lo 和 Shanping Li。2016。从业者对自动化故障定位的期望。载于《第25届国际软件测试与分析研讨会论文集》（德国萨尔布吕肯）（ISSTA 2016）。计算机协会，纽约，美国纽约州，165–176。https://doi.org/ 10 1145 2931037 2931051 。/ 。Xia Li、Wei Li、Yuqun Zhang 和 Lingming Zhang。2019。DeepFL：融合多个故障诊断维度以实现深度故障定位。载于《第28届ACM SIGSOFT国际软件测试与分析研讨会论文集》（中国北京）（ISSTA 2019）。计算机协会，纽约，美国纽约州，169–180。[18] [19] Brian Y. Lim、Anind K. Dey 和 Daniel Avrahami。2009。为何与为何不的解释如何提升上下文感知智能系统的可理解性。载于《SIGCHI计算机系统中的人为因素会议论文集》（美国马萨诸塞州波士顿）（CHI '09）。计算机协会，纽约，美国纽约州，2119–2128。[20] Kui Liu、Shangwen Wang、Anil Koyuncu、Kisub Kim、Tegawendé F. Bissyandé、Dongsun Kim、Peng Wu、Jacques Klein、Xiaoguang Mao 和 Yves Le Traon。2020。关于基于测试套件的程序修复效率：一项系统评估，16 Java程序自动修复系统。载于 ACM/IEEE 第 42 届国际软件工程会议论文集（韩国首尔）（ICSE '20）。计算机协会，纽约，美国纽约州，615–627。https: // doi.org/10.1145/3377811.3380338[21] Alexandru Marginean、Johannes Bader、Satish Chandra、Mark Harman、Yue Jia、Ke Mao、Alexander Mols 和 Andrew Scott。2019。SapFix：大规模端到端自动修复。载于 2019 IEEE/ACM 第 41 届国际软件工程大会：软件工程实践（ICSE-SEIP）。269–278。https://doi. org/ 10 1109 2019 00039 . /ICSE-SEIP. . Leonardo Mariani、Fabrizio [22] Pastore 和 Mauro Pezze。2011。用于诊断集成故障的动态分析。《IEEE软件工程汇刊》37，4（2011），486–508。10 1109 2010 ( ), 93 https://doi.org/ . /TSE. . [23] Sergey Mechtaev、Jooyong Yi 和 Abhik Roychoudhury。2016。Angelix：通过符号分析实现可扩展的多行程序补丁合成。载于 2016 IEEE/ACM 第 38 届国际软件工程大会（ICSE）。691–701。https: //doi.org/ 10 1145 2884781 2884807 . / . M. Monperrus。 . 可解释的软件机器人贡献：自动化错误修复案例研究。载于 2019 IEEE/ACM 第 1 届软件工程机器人国际研讨会（BotSE）。IEEE计算机学会，美国加利福尼亚州洛斯阿拉米托斯， 12 15 10 1109 2019 00010 – 。https://doi.org/ . /BotSE. . [24] [25] Martin Monperrus。 . 自动化程序修复综述。（2020 年 12 月）。 2019 2020 https://hal.archives-ouvertes.fr/hal-01956501 工作论文或预印本。[26] Seokhyeon Moon、Yunho Kim、Moonzoo Kim 和 Shin Yoo。2014。“询问变异体：变异故障程序以进行故障定位。”载于 2014 IEEE 第七届国际软件测试、验证与确认会议。153–162。https://doi.org/10.1109/ICST.2014.28[27] Erik Nijkamp、Bo Pang、Hiroaki Hayashi、Lifu Tu、Huan Wang、Yingbo Zhou、Silvio Savarese 和 Caiming Xiong。2022。CodeGen：一种用于代码及多轮程序合成的开放式大语言模型。arXiv 预印本（2022）。[28] Yannic Noller、Ridwan Shariffdeen、Xiang Gao 和 Abhik Roychoudhury。2022。程序修复中的信任增强问题。载于第 44 届国际软件工程会议论文集（宾夕法尼亚州匹兹堡）（ICSE '22）。计算机协会，纽约，美国纽约州，2228–2240。https://doi.org/ 10 1145 3510003 3510040 . / . OpenAI。 . GPT-[29] 技术报告。arXiv:2303.08774 [cs.CL][30] Long Ouyang、Jeff Wu、Xu Jiang、Diogo Almeida、Carroll L Wainwright、Pamela Mishkin、Chong Zhang、Sandhini Agarwal、Katarina Slama、Alex Ray 等。2022。训练语言模型以遵循人类反馈指令。arXiv 预印本 arXiv:2203.02155（2022）。[31] Caitlin Sadowski、Emma Söderberg、Luke Church、Michal Sipko 和 Alberto Bacchelli。2018。现代代码审查：Google 案例研究。载于第 40 届国际软件工程会议：软件工程实践论文集（瑞典哥德堡）（ICSE-SEIP '18）。计算机协会，纽约，美国纽约州， 181 190 10 1145 3183519 3183525 – 。 [32] https://doi.org/ . / . Benjamin Siegmund、Michael Perscheid、Marcel Taeumel 和 Robert Hirschfeld。2014。研究专业软件开发者调试实践的进步。载于 2014 IEEE 国际软件可靠性工程研讨会。 269 274 10 1109 2014 36 – 。 [33] https://doi.org/ . /ISSREW. . Chengnian Sun 和 Siau-Cheng Khoo。 . 挖掘简洁的谓词错误特征。载于 2013 年第 9 届软件工程基础联合会议论文集（俄罗斯圣彼得堡）（ESEC/FSE 2013）。计算机协会，纽约，美国纽约州，576–586。[34] Jason Wei、Xuezhi Wang、Dale Schuurmans、Maarten Bosma、Ed Huai hsin Chi、Quoc Le 和 Denny Zhou。2022。思维链提示能够激发大语言模型的推理能力。ArXiv abs/2201.11903（2022）。[35] Ratnadira Widyasari、Sheng Qin Sim、Camellia Lok、Haodi Qi、Jack Phan、Qijin Tay、Constance Tan、Fiona Wee、Jodie Ethelda Tan、Yuheng Yieh、Brian Goh、Ferdian Thung、Hong Jin Kang、Thong Hoang、David Lo 和 Eng Lieh Ouh。2020。BugsInPy：用于受控测试和调试研究的 Python 程序现有缺陷数据库。载于第 28 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（线上，美国）（ESEC/FSE 2020）。计算机协会，纽约，美国纽约州，1556–1560。[36] Emily Rowan Winter、Vesna Nowack、David Bowes、Steve Counsell、Tracy Hall、Sæmundur Haraldsson、John Woodward、Serkan Kirbas、Etienne Windels、Olayori McBello、Abdurah man Atakishiyev、Kevin Kells 和 Matthew Pagano。2022。迈向以开发者为中心的自动程序修复：来自 Bloomberg 的发现。载于第 30 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（新加坡，新加坡）（ESEC/FSE 2022）。计算机协会，纽约，美国纽约州， 1578 1588 10 1145 3540250 3558953 – 。 [37] https://doi.org/ . / . Chunqiu Steven Xia、Yuxiang Wei 和 Lingming Zhang。2022。大型预训练语言模型时代的实用程序修复。arXiv 预印本 arXiv:2210.14179（2022）。[38] Chunqiu Steven Xia 和 Lingming Zhang。2022。少训练，多修复：通过零样本学习重新审视自动化程序修复。载于第 30 届 ACM 欧洲软件工程会议与

软件工程基础研讨会（新加坡，新加坡）（*ESEC/FSE 2022*）。计算机协会，纽约，美国纽约州，959–971。[39] Yingfei Xiong、王杰、闫润发、张家晨、韩适、黄刚和张路。2017。程序修复的精确条件合成。载于*2017* 年 *IEEE/ACM* 第 *39* 届国际软件工程大会（*ICSE*）。416–426。

[40] 姚顺雨、Jeffrey Zhao、余典、杜楠、Izhak Shafran、Karthik Narasimhan 和曹元。2022 年。React：协同语言模型中的推理与行动。*arXiv* 预印本 *arXiv:2210.03629*（2022 年）。[41] Andreas Zeller。2009 年。程序为何会失败：系统化调试指南。爱思唯尔。[42] 朱启豪、孙泽宇、肖元安、张文杰、袁康、Yingfei Xiong 和张路。2021 年。神经程序修复的语法引导编辑解码器。计算机协会，纽约，纽约州，美国，341–353 页。

arXiv:2304.02195v1 [cs.SE] 5 Apr 2023

基于大语言模型驱动的科学调试实现可解释自动调试的附录

1 可解释故障定位的讨论

在本节中，我们将论证故障定位同样需要解释；尽管有多种故障定位技术声称具有可解释性并能帮助开发者，但相关的人类研究却很少。首先，正如论文中所述，Kochhar 等人 [7] 的调查显示，85% 的开发者希望获得故障定位的解释。事实上，故障定位对解释的需求更为迫切：对于自动化程序修复，建议的补丁具有可操作性（用户可以接受、拒绝或检查它）；而故障定位的结果却可能让人不清楚该采取什么行动，这与缺陷预测结果的情况类似 [9]。值得注意的是，在 Kochhar 等人的调查中，开发者还将解释需求与修复及“可操作性”联系起来：例如，一位开发者指出：“.....为了决定如何修复缺陷，我想*确切地*知道自动化工具为何认为代码存在缺陷 [（原文如此）]。”

一些常用的故障定位技术包括基于频谱的故障定位（SBFL）和基于信息检索的故障定位（IRFL）。SBFL 分析失败测试和通过测试的覆盖模式，并使用各种公式来推荐与故障最相关的语句 [5]。与此同时，IRFL 使用缺陷报告或失败测试，分析这些材料与源代码之间的文本相似度，以确定与故障描述最相关的文件或方法 [8]。虽然这些技术的推理轨迹可以呈现给开发者，但 Kochhar 等人指出，“这些基本原理不太可能足以帮助实践者”，并引用了 Parnin and Orso [13], ，后者通过一项人类研究质疑故障定位技术的实用性。故障定位技术的近期改进包括基于变异的故障定位（MBFL） [12], ，该方法通过变异语句并观察测试行为的变化来确定可能的故障位置；以及基于机器学习的故障定位 [10], ，该方法利用各种联邦学习技术的特征，并通过机器学习预测哪些位置可能存在故障。与我们对自动化程序修复的观察类似，这些技术都不会以类似人类的方式进行推理，因此很难认为呈现这些技术的推理轨迹会有助于理解其结果（对于基于机器学习的故障定位，甚至不清楚它是否一开始就存在推理轨迹）。

与此同时，我们认为，与程序修复领域相比，可解释故障定位方面的研究仍然更多。我们参考了三项综述 [1, 14, 16] ，以确定相关研究。Groce 提出了 explain [4],，该方法通过将失败测试与最相似的通过测试进行比较，来找出程序值发生分歧的位置。Zeller 的早期工作也沿着相关方向展开：他将 Delta调试应用于程序内部状态，以进行故障定位 [2, 17]。例如，Cleve and Zeller [2], 在论文的结论中指出，开发者“将不仅知道测试失败了，还会知道它为何以及在何处失败”，这也表明他们有兴趣解释缺陷及故障定位结果。近来，Sumner and Zhang [15] 利用切片技术，使 Zeller 的状态替换方法更加精确，从而更准确地解释差异。尽管我们受到这类研究的启发，但上述文献并未针对所提供的解释开展用户研究，因此我们无法据此比较其效果。此外，与 AutoSD 相比，调试方式存在显著差异：在 AutoSD 中，大语言模型会自动生成关于代码问题所在的假设，并据此提取值；而在上述研究中，研究者会检查所有变量的值，以找出导致缺陷的变更。Whyline [6] 允许

开发者向调试系统提出“为什么”和“为什么不”的问题，并获得答案；与 AutoSD 不同，它的重点并非自动化调试，假设仍由人类开发者提出。作为一种事后分析技术，Bugsplainer [11],由 Mahbub 等人提出，用于解释某个位置为何可能存在缺陷（但无法实际说明工具为何将该位置定位为特定位置）。该方法使用神经机器翻译（NMT）训练 Transformer 模型，将已识别的位置转换为可能的提交说明。

总的来说，Alipour [1] 十多年前提出的观察似乎至今仍然成立：“解释故障最合适的抽象层次尚不明确”；我们希望本文能为这一问题提供一些线索。

2 ARHE基准测试变异器分类

表1展示了按生成各个缺陷所用的变异器分类的 ARHE基准测试。我们还在此介绍各个变异器的具体细节，并将它们与 PIT [3], 这一广泛使用的变异测试工具中的变异器进行比较。“整数常量更改器”会将字面量常量 0 改为 1，反之亦然，其行为类似于 PIT 的“内联常量变异器”。“If 移除器”会移除 if 语句的 then 块或 else 块；若移除后不再包含任何子项，则 if 语句本身也会被移除，这与 PIT 的“移除条件语句变异器”类似。“字符串字面量更改器”会将字符串字面量改为空字符串、小写或大写；将其改为空字符串后无法还原，而大小写转换能否反向应用以恢复原始代码，则因问题而异。生成空字符串的行为类似于 PIT 的“空返回值变异器”。“运算符更改器”会将加号改为减号等，类似于 PIT 的“数学运算变异器”。“二元运算符移除器”会移除二元运算符，只保留一个操作数，类似于 PIT 的“算术运算符删除变异器”。“增强赋值更改器”会将 += 改为 -=，反之亦然，等等，类似于 PIT 的“递增变异器”。“条件否定器”会在 if 条件中添加 not，类似于 PIT 的“条件取反变异器”。

在表1中，If Remover 产生的24个缺陷和 Binary Operator Remover 产生的24个缺陷都不可逆；此外，我们人工判定 String Literal Changer 产生的63个缺陷中有42个不可逆，因此总共有90个缺陷无法通过应用相同的变异集来修复。

变异器	数量
整数文字更改器 ∘	45
If Remover □	24
字符串文字更改器 Δ	63
运算符更改器 ∘	40
二元运算符移除器 □	24
增强赋值更改器 ∘	3
条件否定器 ∘	1

表1. ARHE基准测试细分。可逆变异器用 ∘ 标记，不可逆变异器用 □标记，偶尔可逆的变异器用 Δ 标记。

3 网站截图

人类研究界面的截图见图1。请注意，原始含缺陷代码和错误消息显示在左栏，补丁建议显示在中间，解释显示在右侧，问题则显示在网页底部中央，与我们论文中的图2类似。

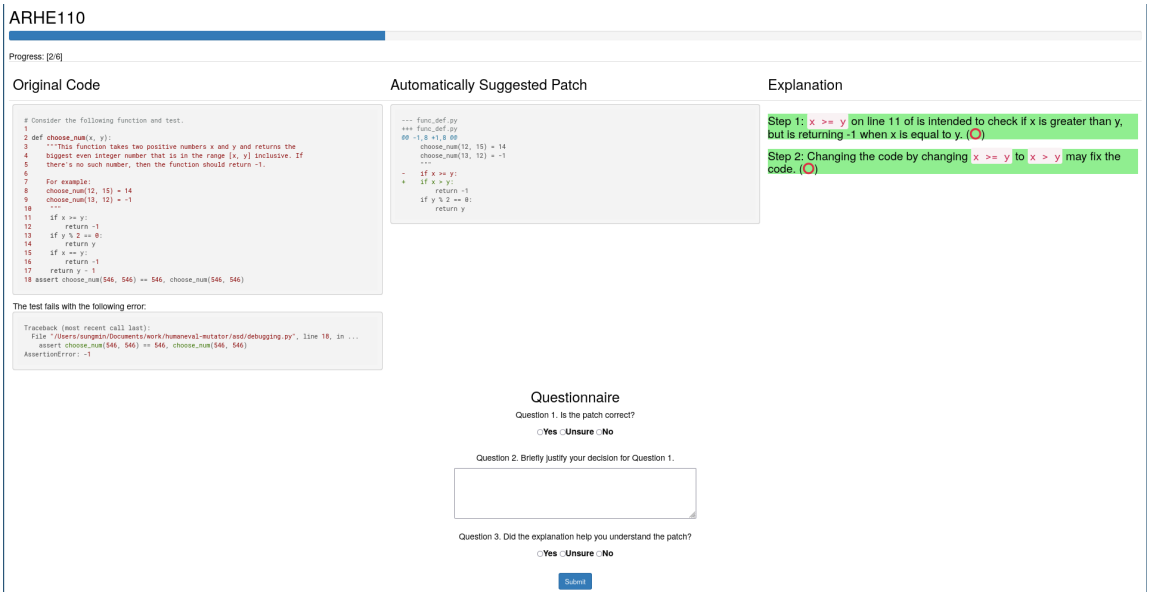


图 1. 人类研究网页截图。

4 科学调试提示词

下方框中展示了用于 Defects4J 基准测试的科学调试描述提示 rk.

我将使用科学方法调试下面的问题（根据 Zeller，2009 年的写法），并描述假设/预测/实验/观察/结论。具体如下：假设：对缺陷行为的解释。假设是该方法的关键，应详细阐述并谨慎撰写。假设应建立在所有先前信息的基础上，因此强烈不建议重复先前的假设。下面提供了一些示例。- 假设示例 1：“鉴于 [信息]„ 该方法 [的整体错误行为]。具体来说，我认为这是因为方法‘foo’第 4321 行的‘c>b’本应实现 [预期行为]„ 却出现了 [错误行为]。” - 假设示例 2：“由于先前的假设被否定，我们现在知道，方法‘foo’第 4321 行的‘c>b’很可能不是问题所在。看看其他地方，也许‘x.append(y)’本应实现 [预期行为], ，却出现了[错误行为]。” - 假设示例 3：“由于先前的假设得到支持，我认为将代码中的‘c>b’改为‘c>b && a <= d’可能会修复代码。” - 假设示例 4：“看来上一次实验以错误告终，我们可能需要尝试不同的实验。也许可以通过 [新实验]来改进实验。”预测：如果假设正确，将观察到的特定值或症状。根据假设，可以预测某项测试会通过。对于循环中的表达式，应进行具体分析。

- 预测示例 1：如果我使用调试器打印 [表达式]，，根据给定的输入及其预期用途，该值应为 [预期值]，；但实际并非如此。也就是说，当我在第 lineno 行暂停调试器时，‘[验证_表达式]’将为真。- 预测示例 2：如果我将 [表达式] 改为 [新_表达式]，，测试就会通过。- 预测示例 3：如果我修改代码以使用新变量，测试就会通过。实验：用于检查预测是否成立的特定 ‘jdb’ 脚本。请确保行号指向实际语句（而不是括号）。- 示例 1：（pdb 脚本）：‘stop at org.not.a.test.class.file:lineno ; run ; print [验证_表达式]’ - 示例 2：（编辑脚本，可使用 REPLACE/ADD/DEL）：‘REPLACE(4321, "c>b", "c>b && a <= d") AND ADD(4323, "a+=1; ") AND RUN’观察：‘jdb’ 脚本的输出。示例：‘[值]’结论：根据观察结果判断假设是否成立。当假设得到确认并带来具体的程序修复时，还要添加 <调试已完成>。- 示例：该假设（得到支持/被否定/因实验错误而无法确定）。 [测试通过时，添加<调试已完成>。]

5 Defects4J AutoSD 示例

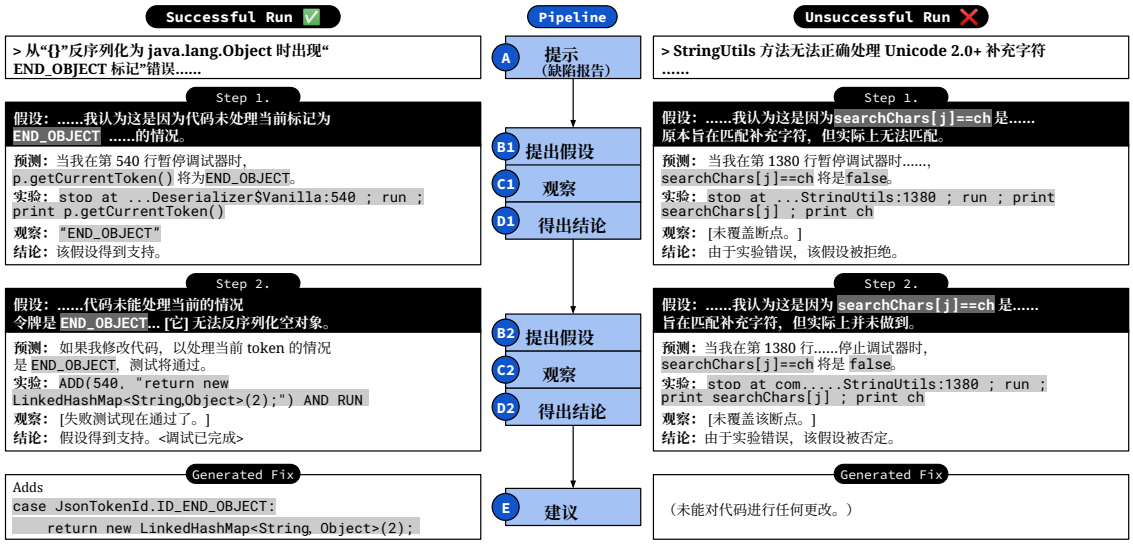


图 2. AutoSD 在 Defects4J 缺陷上的运行示例。

图2展示了一条最终得出正确补丁的推理轨迹，以及一条未能得出正确补丁的推理轨迹。在左侧的例子中，AutoSD 假设缺陷是在当前令牌为 END_OBJECT 时发生的，并设计了一项实验来验证这一点。由于实际情况确实如此，它便在尝试 2 中继续寻找能使失败测试通过的行为。将这两个步骤结合起来后，它生成了一个与开发者补丁完全相同的补丁（就此方法而言），使测试套件中的所有测试均通过。与此同时，右侧展示了另一个未能识别正确断点的例子

。在这个例子中，它重复了相同的假设和实验，没有带来任何改进。

参考文献

[1] Mohammad Amin Alipour。2012。自动化故障定位技术：一项综述。俄勒冈州立大学 54，3（2012）。[2] Holger Cleve 和 Andreas Zeller。2005。定位程序故障原因。载于第 27 届国际软件工程会议论文集（美国密苏里州圣路易斯）（ICSE '05）。计算机协会，美国纽约州纽约，342–351。[3] Henry Coles、Thomas Laurent、Christopher Henard、Mike Papadakis 和 Anthony Ventresque。2016。PIT：一种实用的 Java 变异测试工具（演示）。载于第 25 届国际软件测试与分析研讨会论文集（德国萨尔布吕肯）（ISSTA 2016）。计算机协会，美国纽约州纽约，449–452。[4] Alex Groce、Sagar Chaki、Daniel Kroening 和 Ofer Strichman。2006。基于距离度量的错误解释。8，3（2006 年 6 月），229–247。[5] James A. Jones、Mary Jean Harrold 和 John Stasko。2002。测试信息可视化以辅助故障定位。载于第 24 届国际软件工程会议论文集（佛罗里达州奥兰多）（ICSE '02）。计算机协会，美国纽约州纽约，467–477。[6] Amy J. Ko 和 Brad A. Myers。2008。使用 Whyline 进行源代码级调试。载于 2008 年软件工程中的协作与人为因素国际研讨会论文集（德国莱比锡）（CHASE '08）。计算机协会，美国纽约州纽约，69–72。[7] Pavneet Singh Kochhar、Xin Xia、David Lo 和 Shanping Li。2016。从业者对自动化故障定位的期望。载于第 25 届国际软件测试与分析研讨会论文集（德国萨尔布吕肯）（ISSTA 2016）。计算机协会，美国纽约州纽约，165–176。https://doi.org/10.1145/2931037.2931051[8] Anil Koyuncu、Kui Liu、Tegawendé F. Bissyandé、Dongsun Kim、Martin Monperrus、Jacques Klein 和 Yves Le Traon。2019。IFixR：由缺陷报告驱动的程序修复。载于 2019 年第 27 届欧洲软件工程会议与软件工程基础研讨会 ACM 联合会议论文集（爱沙尼亚塔林）（ESEC/FSE 2019）。计算机协会，美国纽约州纽约，314–325。[9] Chris Lewis、Zhongpeng Lin、Caitlin Sadowski、Xiaoyan Zhu、Rong Ou 和 E. James Whitehead Jr. 2013。缺陷预测能否支持开发人员？来自 Google 案例研究的发现。载于 2013 年国际软件工程会议论文集（美国加利福尼亚州旧金山）（ICSE '13）。IEEE 出版社，372–381。[10] Xia Li、Wei Li、Yuqun Zhang 和 Lingming Zhang。2019。DeepFL：整合多个故障诊断维度，实现深度故障定位。载于第 28 届 ACM SIGSOFT 国际软件测试与分析研讨会论文集（中国北京）（ISSTA 2019）。计算机协会，美国纽约州纽约，169–180。[11] Parvez Mahbub、Ohiduzzaman Shuvo 和 Mohammad Masudur Rahman。2023。利用神经机器翻译中的代码结构解释软件缺陷。arXiv:2212.04584 [cs.SE][12] Seokhyeon Moon、Yunho Kim、Moonzoo Kim 和 Shin Yoo。2014。询问变异体：通过变异故障程序进行故障定位。载于 2014 年 IEEE 第七届软件测试、验证与确认国际会议。153–162。https://doi.org/10.1109/ICST.2014.28[13] Chris Parnin 和 Alessandro Orso。2011。自动化调试技术真的能帮助程序员吗？载于 2011 年国际软件测试与分析研讨会论文集（加拿大安大略省多伦多）（ISSTA '11）。计算机协会，美国纽约州纽约，199–209。[14] Alexandre Perez、Rui Abreu 和 Eric Wong。2014。故障定位技术综述。（2014）。[15] William N Sumner 和 Xiangyu Zhang。2013。比较因果关系：解释执行过程之间的差异。载于 2013 年第 35 届国际软件工程会议（ICSE）。IEEE，272–281。[16] W. Eric Wong、Ruizhi Gao、Yihao Li、Rui Abreu 和 Franz Wotawa。2016。软件故障定位综述。《IEEE软件工程汇刊》42，8（2016），707–740。https://doi.org/10.1109/TSE.2016.2521368[17] Andreas Zeller。2002。从计算机程序中分离因果链。载于第 10 届 ACM SIGSOFT 软件工程基础研讨会论文集（美国南卡罗来纳州查尔斯顿）（SIGSOFT '02/FSE-10）。计算机协会，美国纽约州纽约，1–10。