arXiv:2602.07742v1 [cs.PL] 8 Feb 2026

Gillian 调试：在（组合式符号执行）树间穿梭，扩展版

Nat Karmios, Sacha-Élie Ayoun, 和 Philippa Gardner

英国伦敦帝国理工学院
n.karmios@ic.ac.uk

摘要。近年来，组合式符号执行（CSE）工具日益受到关注，也越来越适用于真实世界的代码库。然而，时至今日，即使是专业用户，调试这些工具的输出仍然十分困难。为此，我们推出了一款面向符号执行工具的调试界面，该界面集成于 Visual Studio Code 和多语言 CSE 平台 Gillian 中，并着重强调可视化、交互性，以及对符号执行树的直观呈现。我们特别注重使该界面不依赖于特定工具，以便未来将其移植到其他符号分析工具中。我们通过一项用户研究对这项工作进行了实证评估。研究结果表明，该调试器有助于初级研究人员理解 CSE 的原理，并在 Gillian 中验证基本的数据结构算法。

关键词： 调试、符号执行、验证

1 引言

符号执行（SE）是一种静态程序分析技术，它通过在符号输入（而非具体输入）上执行代码来探索程序行为。尽管基于一阶逻辑的符号执行工具和框架（如 CBMC [5] 和 KLEE [4],）在学术界和工业界得到了广泛应用，但面对堆操作程序时，其推理能力难以扩展。

组合式符号执行 [13, 14] （CSE）通过为符号执行扩展以分离逻辑表达的函数规约，解决了这一局限，例如原始分离逻辑（SL） [21]和近期提出的不正确性分离逻辑（ISL）[23]。这使得函数能够脱离调用上下文单独分析，并将分析结果记录为简洁的规约，以供后续分析复用。这种组合性在保留对堆操作程序进行精确推理能力的同时，显著提升了可扩展性。

基于CSE的工具在学术界和工业界的开发与应用都在稳步增长。例如，VeriFast 使用分离逻辑为 C语言、Java 和 Rust 提供半自动组合验证；CN 支持使用分离逻辑验证 C语言程序；基于隐式

动态帧（与分离逻辑类似）的 Viper [2, 6, 25]框架，为 Go、Java、Python 和 Rust [2, 6, 25] 的验证工具提供了基础；基于 ISL 构建的 Infer-Pulse [24]，已部署于 Meta，用于大规模自动化多语言漏洞检测 [16]；Gillian [3, 8, 15] 则提供了一个统一框架，分别基于分离逻辑和 ISL 进行正确性与错误性推理，并应用于 C语言、JavaScript 和 Rust。这些工具已相当成熟，能够应用于真实软件，包括 AWS SDK（Gillian）、Google 的 pKVM 虚拟机监控程序（CN）、互联网路由器（Viper）、Meta 的生产代码库（Infer-Pulse）以及 Rust 标准库（Gillian、VeriFast）。其中一些工具还用于向本科生和研究生讲授程序分析原理（Gillian、VeriFast、Viper）。

随着这种日益广泛且多层面的适用性不断增强，人们也越来越需要清晰地呈现给定 CSE 工具的输出，并对其进行高效调试。对我们而言，这一需求包含三个方面：(1) 让工具开发者能够全面、细致地检查工具的内部运行机制，从而改进其实现；(2) 使工具用户能够通过将工具输出与所分析的代码联系起来，对其程序进行调试；(3) 指导作为下一代工具用户和开发者的学生理解 CSE 推理的基础知识。然而，这仍然是一个相对缺乏研究的主题，通常以临时性的方式加以处理，且不同工具的用户体验差异很大：有些工具只提供（令人不堪重负的）文本反馈；有些仅提供最低限度的基于语言服务器的集成；还有一些提供交互式探索，但将其限制在生成的反例上，或只能事后进行；而少数真正拥有交互式调试器的工具，要么以牺牲视觉表达为代价采用广泛的编辑器集成，要么专注于与单一编辑器集成，从而难以迁移到其他工具。关于最新进展的更详细概述见第2节。

为弥补这一空白，我们以 Gillian 为案例研究，推出了一款用于基于 CSE 的验证的交互式调试器。我们的目标是设计一款工具无关调试器，确定其哪些组件足够通用，能够支持其他 CSE 工具，哪些组件则是 Gillian 特有的。Gillian 是这项工作的理想测试平台，因为它以参数化方式处理内存，支持通过编译到通用中间表示（IR）来处理多种源语言，并在统一框架中实现多种分析（§3）。我们的调试器集成于 Visual Studio Code [17]，以利用这一广泛使用且广为熟悉的开发环境；我们再次采用通用化的方法，旨在便于今后集成 VSCode 之外的其他编辑器。

我们将在 §4中介绍调试体验，并在 §5中阐述调试器的设计原则，重点关注：

– 执行树表示，旨在通过树中树结构创建一种实用、直观且灵活的分析表示；– 执行树捕获，利用这种树中树结构，以一种不打扰的方式在引擎层记录分析信息；– 执行树提升，将低层级的树中树结构提升为用户可读的源代码层级结构；以及– 调试交互性，让用户能够全面掌控对符号分析的检查和实时引导，而不是等分析完成后再对其进行总结。

重要的是，我们尽可能避免将设计选择与 Gillian 的实现绑定，使其能够迁移到其他 CSE 工具，甚至有望迁移到更传统的符号执行工具。此外，在 §6中，我们重点介绍了该项目大量工程工作过程中获得的有趣经验和观察。

我们在一次 PL 暑期学校上进行了一项基础用户研究（§7），评估 Gillian 调试器在基于 CSE 的验证中的易用性。我们向初期研究人员介绍了分离逻辑、CSE 和 Gillian，随后让他们使用 Gillian 尝试验证若干数据结构算法。这些示例旨在要求使用半自动组合验证的主要功能：前置条件和后置条件的规约、循环不变式以及证明策略。之后，我们进行了定量分析，结果显示用户花费了大量时间使用调试器，表明该工具颇有用处。我们还通过李克特式问卷收集了定性反馈，汇总结果表明，调试器易于使用且直观，其反馈也具有信息价值和教育意义。

2 相关工作

我们简要回顾最新进展中的 CSE 工具所具备的调试能力。Gillian（在我们的工作之前）和 CN 仅提供基于文本的日志；在现实场景中的验证过程中，这些日志可能多达数千万行。这些日志将本质上非线性的分析线性化为扁平的文本流。尽管内容详尽，但细节过多，只有熟悉这些工具的专家才能理解。

Infer-Pulse 会生成 HTML 文件，其呈现的信息与文本日志类似，并将内容组织为与源代码位置相关联的分析“节点”。虽然这比纯文本日志更加清晰，但仍难以跟踪分析在程序中的流程，而且需要离开开发环境，这会打断用户的工作流程。

Viper 通过语言服务器协议（LSP） [19],与编辑器集成，在用户输入时重新运行分析，并以内联方式报告失败。这有助于快速获得反馈，但此类反馈通常仅限于简短的错误消息。我们认为，当需要深入了解分析时，提供调试器可以作为这种方式的补充。

最后，VeriFast 通过其 VSCode 扩展 [11], 支持对分析执行树进行交互式检查，类似于调试会话。不过，树节点没有标签，也难以对应回源代码。此外，由于分析会在遇到第一个断点或失败时终止，事后生成的树通常并不完整，导致部分执行分支未被探索。

我们的 Gillian 调试器扩展了调试适配器协议（DAP） [18] ，为其 CSE 分析提供交互式符号调试环境。其他一些符号分析工具，例如 GobPie [10] 和 SecC [7], ，也使用 DAP 与许多编辑器“免费”提供集成调试体验。不过，DAP 没有提供直观的机制来表示分支执行，因此出现了各种临时解决方案，例如挪用 DAP 的“线程”视图；该视图原本是为在具体执行中可视化并发线程而设计的。

或许与我们的调试器最相似的是 KeY 项目 [1]中功能丰富的符号执行调试器 [9] 。该调试器能够显示详细且带注释的符号执行树，但依赖非标准接口，并且与 Eclipse 集成开发环境及 KeY 的一阶 Java 分析紧密集成，因此难以移植到其他集成开发环境和分析框架。

3 背景

符号执行 是一种程序分析技术，它使用符号输入而非具体输入来执行给定程序。一个符号状态描述一组具体状态，从而能够对程序行为进行一般性推理。符号执行可以产生分支（例如通过 if-else 语句），每个分支都带有一个称为路径条件 的逻辑断言，用于描述导致执行进入该分支的符号变量所受的约束。

组合式 **符号执行** （CSE）扩展了符号执行，使其能够使用和创建以分离逻辑编写的函数规约。具体而言，这种推理允许使用函数规约来概括部分符号状态上的函数行为；函数规约是霍尔三元组，其含义由底层分离逻辑赋予。在验证中，前置条件描述符号状态中足以执行该函数的一部分；后置条件则描述所有可能的结果。

调用函数时，可以使用被调用函数的规约代替内联其函数体，从而实现组合性并提高可扩展性。执行规约意味着将当前符号状态与其前置条件进行匹配，从状态中消耗（或“吐出”）匹配的部分，并生成（或“吸入”）后置条件。验证函数只需从其前置条件开始对函数进行符号执行，直至终止，然后将每个最终状态与后置条件进行匹配。由于函数调用被规约的执行所取代，符号执行树的长度仅相当于单个函数中的语句数，与非组合式符号执行相比，大幅降低了推理复杂度。

Gillian 是一个多语言平台，支持三类分析：全程序符号测试，可提供与一阶工具 CBMC 类似的有界验证保证；半自动组合验证（采用 CN、VeriFast 的分离逻辑以及 Viper）；以及自动漏洞查找（类似于使用 ISL 的 Infer-Pulse）。Gillian 以所分析源语言的内存模型为参数；每次实例化时都需要提供该内存模型，以及一个将源语言编译为 Gillian 中间表示 GIL 的编译器。对于现实世界的语言，Gillian 已实例化支持 C语言、JavaScript 和 Rust。为便于讲解，本文重点介绍**WISL**，这是 Gillian 针对一种简单 while 语言的示范性实例化。该语言采用类 C 的块偏移内存模型，通常在本科四年级课程以及理学硕士学生课程中讲授。关于这种语言的更多细节，请参见附录第A 节。

4 Gillian 调试体验

我们通过使用 Gillian 调试器调试对 WISL`llen()` 函数的验证，来展示 Gillian 调试器的功能（图 1左图）。该函数计算以空指针结尾的单链表（SLL）的长度；每个节点由一个双单元块构成，其中包含该节点所携带的值以及指向下一节点的指针。此函数使用标准的分离逻辑 SLL 谓词进行规约，即 `list(x,alpha)`，该谓词说明，从标识符为 `x` 的块开始的 SLL 包含数学列表 `alpha`中的值。因此，前置条件中只有该链表，而后置条件还额外说明返回值等于链表长度。为展示源语言级（WISL）与中间表示级（GIL）的区别，我们还在图 `llen()` 到 GIL 的示意性编译结果，见图 1 右图。从中可以看到，WISL 的结构化控制流如何通过 GIL 跳转语句变为非结构化控制流，也可以看到 WISL 命令（例如指针解引用，$\mathtt{t} := [\mathtt{x} + 1]$）如何被拆解为多个 GIL 步骤。

```
1  谓词 列表(x, alpha) {
2     (x  == 空值)* (阿尔法  == 空列表);
3     (x -> #v, #z) * 列表(#z, #beta)
4                   * (阿尔法  == #v::#贝塔)
5  }
6
7  { (x ==#x)*列表(#x, #alpha)}
8  函数 llen(x) {
9     如果 (x  =空值) {
10       n := 0
11    } 否则 {
12       t: = [x+1]; n: =llen(t)
13    };
14    返回 n
15 }
16 {列表(#x,#alpha)*(返回  ==长度(#alpha))}
```

```
1  过程 llen(x) {
2           跳转? (x  =null) 然后 否则;
3     然后: n := 0;
4           跳转 end;
5     else:_var0 :=i_add(x, 1);
6           跳转? (_var0 is Ptr) cont fail;
7     fail: fail "无效指针";
8     cont: t: = load<_var0>;
9           n := llen(t);
10    end:  skip;
11          ret := n;
12          返回
13 };
```

图1。 略作简化的 SLL 谓词和指定的 WISL 递归列表长度函数（左），以及对应的简化版编译后 GIL 代码（右）。前缀 # 表示逻辑变量。

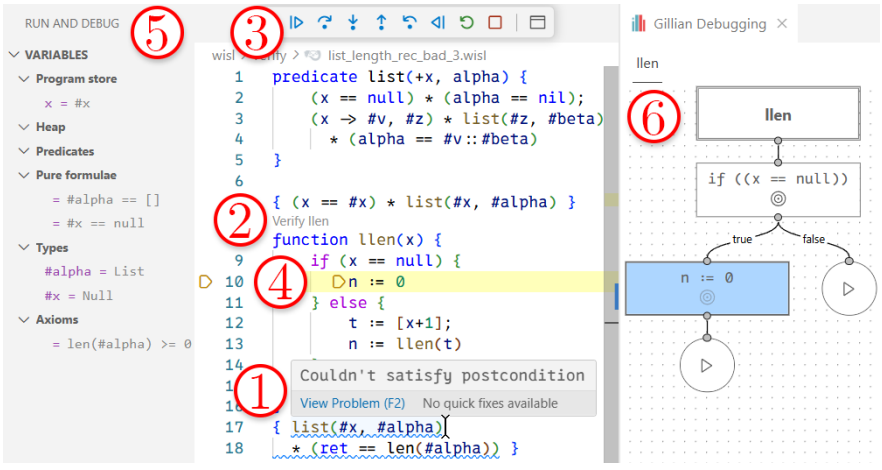

图2。 验证 llen() 时 Gillian 的调试和语言服务器界面。

尝试使用 Gillian 和 VSCode（参见图 `llen()` 进行验证时2），语言服务器会告知我们存在一个错误（1），指出无法满足函数后置条件。虽然其他现有的 LSP（例如 Viper 的 LSP）可能会提供稍微精确一些的信息，例如指出后置条件中的哪一部分未能满足，但 LSP 接口只能指出失败，却无法解释原因。此时，我们会转而使用 Gillian 调试器追查错误的根本原因，因为验证可能会因各种原因失败，例如规约有误、缺少证明注释，或程序本身存在错误。

Gillian 调试器会在每个函数上方显示一个“验证”按钮（2），用于启动该函数的交互式调试会话。在会话期间，界面会向用户呈现若干信息：用于逐步执行执行过程的控件（3）；当前正在执行的代码行高亮显示（4）；当前符号状态的详细分解（5）；以及在单独面板中以树视图呈现符号执行过程的树（6）。

编号 3 处的控件继承自 VSCode 调试适配器协议接口。它们包括：一个“继续”按钮，沿当前分支继续执行，直到遇到下一个断点或执行结束；一个“单步跳过”按钮，执行下一条 WISL 语句；一个“单步进入”按钮，如果存在函数调用，则进入该函数调用；一个“单步跳出”按钮，继续执行，直到当前函数返回；一个“后退一步”按钮，撤销上一条语句；一个“向后继续”按钮，返回到上一个断点或执行起点；一个“重新启动”按钮，从头开始整个符号执行；以及一个“停止”按钮，用于结束调试会话。

使用这些控件浏览符号执行时，当前正在执行的代码行会高亮显示，并用一个小箭头指向正在求值的子表达式（4）。此外，屏幕左侧会显示当前符号状态，并使用 VSCode 内置的变量浏览器展示当前程序变量存储、堆、谓词、路径条件以及其他相关信息（5）。

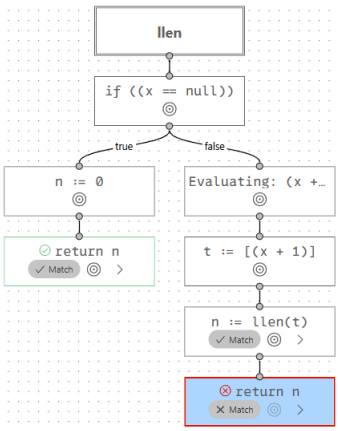

图 3. llen() 的完整符号执行树

最后，也是最重要的一点，树视图（6）支持对符号执行进行交互式可视化。语句在树结构中表示为节点，边则表示执行流程。点击某个节点即可跳转到执行过程中的相应位置，并更新
高亮显示的代码和符号状态提供了一种导航方式，可在不同分支之间切换，也可在同一分支内前后移动。如果某个分支尚未终止，该分支的叶节点处会显示一个“播放”按钮，点击后即可执行该分支上的下一条语句。随着示例复杂度提高、验证耗时变长，这一界面允许用户控制分支探索，避免长时间等待。

图3 展示了`llen()`的完整符号执行树。从中可以看出，导致验证失败的路径经过了`if` 语句的“false”分支，这一点由紧邻 return 语句的红色叉号标出。该节点还显示了一个向右的小箭头，表示节点中包含嵌套的信息。

点击该箭头会展开节点，显示失败的后置条件匹配步骤（图 4）；嵌套树中的放大镜图标则会将其“弹出”到新标签页中，以便单独查看。请注意，在匹配过程中，视觉上的分支仍然很有用：Gillian 已回溯，并尝试通过展开列表谓词来恢复。在这棵树中，我们看到 Gillian 未能匹配 `ret == len(#alpha)`（最左侧分支），于是回溯并应用了恢复策略（此处为“展开”策略），但匹配仍因相同原因失败（最右侧两个分支）。恢复策略由启发式规则触发，有时会令人困惑；这个例子展示了我们的调试器如何清晰地向用户呈现这一过程。

点击最左侧的失败节点，然后详细检查符号状态（图 4，右侧），我们看到 `ret`等于 `len(#lvar_4)`，而 `#alpha`是列表`([#lvar_7]@#lvar_4)`，因此 `ret` 的值差了 1。这是最终揭示错误根本原因的一步：我们忘记在程序中递增 `n`。具体来说，在图 1（左）第 12 行之后，必须满足 $\mathtt{n} := \mathtt{n} + 1$。虽然最后这一步仍需用户自行完成，但只需点击几下，就能找到所需信息。

需要指出的是，在顶层，执行树的最大深度等于正在验证的函数中的语句数量（不计表达式求值，这属于实现选择）；由于 CSE 允许单独验证函数，因此执行树的规模受到限制。现实条件——例如更大的函数、更复杂的源语言——自然会产生比本研究中遇到的更大的树，但我们预计其规模仍会比 KLEE 等非组合式符号分析所见的树小几个数量级。

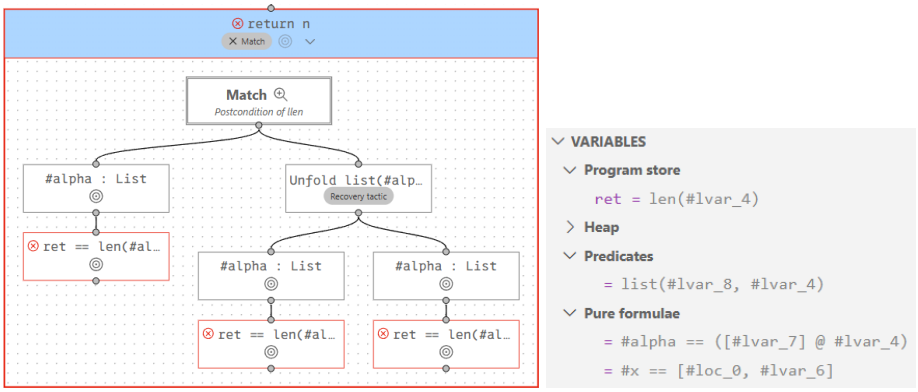

图 4. 展开的失败节点（左）以及失败匹配处的符号状态（右）。

5 条 CSE 调试原则

我们在图中概述了我们的三层架构。5 并讨论为构建实用的 CSE 调试器而组合的关键流程。我们将指 WISL 语句`t := [x+1]; n := llen(t)` 在 `llen()`第 12 行的执行；它们分别对应 GIL 代码的第 5–8 行和第 9 行（见图1）。为促进更广泛的适用性，我们将这些概念从 Gillian 特有的实现细节中抽象出来；第§6节将讨论 Gillian 特有的一些实现细节。

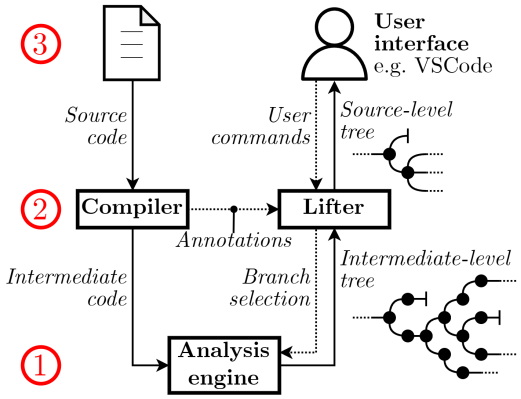

图 5. 概览：Gillian 调试器的架构。

在第 1 层，我们创建一种树结构，用于描述中间表示（IR）层面的分析过程。这是一棵低层级的树，包含分析过程中的所有执行步骤，理想情况下还包括任何相关的基础过程（例如，对于 Gillian，这些过程包括匹配（目前已支持）、表达式简化和状态简化（目前尚不支持））。这棵树必须准确体现人们对分析“形态”的直觉，并且能够随着交互式分析的开展不断扩展。只有工具专家开发者才需要单独使用这一层，从而微调工具性能，并理解工具本身出现的意外行为或错误。

为了向使用经分析的源语言的工具用户呈现易于理解且可采取行动的执行树，还必须弥合工具内部中间表示与该源语言之间的隔阂。这正是第 2 层的作用：调试器在这一层运用各种提升机制，将第 1 层的低层级树抽象为源代码级执行树。

最后，第 3 层涉及工具开发中一个重要但鲜少讨论的方面：用户与工具交互所依赖的基础设施。用户界面开发，尤其是以复用性、可维护性和长久适用为目标时，可能是一项艰巨的工程任务，因为这需要深入了解现有的开发者工具。

5.1 低级树中树结构

直观的交互式调试体验需要一种能够准确呈现分析过程“形态”的结构，而无需预先了解执行顺序。有人可能会认为，符号执行可以简单地表示为一棵树，其中拥有多个子节点的节点表示符号分支。我们使用树中树结构来表示中间表示层和源代码层的分析，其中每个节点都可以包含嵌套的树，以提供更多细节。这种结构的好处在于，它捕捉了将执行和分析拆解为更小步骤的直观思路，同时又能在用户需要之前隐藏过多细节。例如，嵌套可用于表示 Gillian 的匹配过程或内联函数调用的函数体；回想一下图 `llen()`中的执行树（图3），其中失败的 `return` 节点展开后，显示出图4所示的嵌套匹配树。

CSE 工具通常会实现特定的执行策略（例如深度优先），以遍历执行树。交互式调试要求记录一棵不受任何此类策略影响的执行树，使其能够处理中断并随后返回某条执行路径的情况。因此，工具的分析引擎需要进行一些修改，以提供构建树中树所需的信息。然而，跟踪符号执行中的非平凡控制流（例如进入和退出函数体及循环体）会大幅增加引擎的复杂性，使工具的一个关键部分变得更加

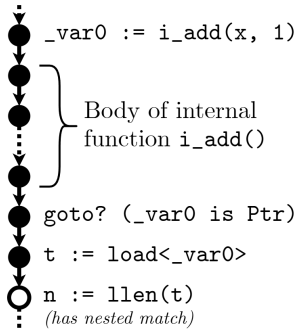

图 6. llen() 的 GIL 跟踪片段。

难以维护。我们发现，采用最简方案就足够了，而且可以直接应用于许多 CSE 工具：与大多数工具一样，仅写入描述每个步骤分析状态的文本日志不同，引擎可以生成包含类似信息和唯一标识符的机器可读报告。报告还会存储前一个执行步骤的标识符，以及包含该步骤的外层步骤（如适用）；这只需引擎进行少量记录，且很容易与执行上下文一并存储。在调试流程的后续阶段，可以自由访问这些报告；届时，无需引擎提供更多输入，即可完成构建更复杂树结构所需的大量工作。图 6 展示了第 12 行 WISL 语句对应的一段 GIL 跟踪；为保持引擎逻辑简单，诸如`i_add()` 这样的函数体不会嵌套在调用命令之下。在方便时，中间层仍会使用嵌套，例如对递归调用 `llen()`的匹配。

5.2 提升到源代码层

应用前述概念后，我们便得到了一个完全够用的交互式执行树，但它仅适用于相关工具的中间表示。为了能够在源语言层面进行调试，CSE 调试器需要包含一个

提升器，其职责是告知分析引擎如何将特定中间表示的树中树转换为更贴近特定源语言的树中树——在我们的调试器中，就是从 GIL 转换为 WISL。例如，提升器可以通过将多个中间表示级命令的节点合并为一个对应源级语句的节点来改变树的形状；也可以调整并合并分支情况，使其更加清晰；还可以在适当之处引入嵌套。提升器还可以改进与各个节点相关的信息，例如生成节点显示文本，并转换显示的状态，使其在源代码层面更易于识别。

构造源级树中树所需的源代码感知信息，最容易在编译期间收集，并可通过**注释**传递给提升器。这些中间表示代码上的注释可以包括源语句的显示文本、标记哪些中间表示语句组构成一个源语句的标志，或提示可能发生分支的原因。例如，WISL 语句 `t := [x+1]` 编译为 GIL 代码的第 5–8 行：我们将表达式求值视为一个独立步骤，因此第 5 行被标记为终止一个节点；第 6 行被标记为不终止节点；第 7 行和第 8 行则被标记为终止节点。在源级树中，这会形成一个用于求值`x+1`的步骤，以及一个用于执行查找的步骤。关于 WISL 注释及其在附录 A.2 中对 `llen()`的 GIL 代码所作赋值的更多细节，请参阅附录。

5.3 用户界面

说到用户界面开发，鉴于其所需的工程工作量，应尽量避免为了创建一个功能略有不同或运行环境略有不同的界面而“重复造轮子”。

如第§2节所述，一些工具会利用 DAP，以使用 VSCode 等集成开发环境提供的调试接口。这些功能包括突出显示当前代码步骤、控制执行过程中的单步执行，以及查看当前状态。然而，DAP 的表达能力不足以表示分支执行，更不用说导航完整的执行树了，因此人们不得不采用一些临时解决方案，例如使用“线程”视图列出执行分支，或者完全放弃 DAP，转而使用专门的分析后树查看器。

我们的方法通过扩展 DAP 来支持与编辑器中自定义树视图的交互，从而两全其美。参见图2，实现 DAP 的调试器会按标准提供标准调试 UI 元素；用户可以看到当前步骤的代码行（4）和状态（5），并使用（线性）单步控制（3）。定制树视图（6）由自定义事件`mapUpdate`支持，用以向树传递变更，并由自定义命令`jump` 跳转到树中的另一位置，以及`stepSpecific`在存在多个可用分支时沿特定分支向前单步执行。这确实削弱了 DAP 的编辑器无关性——因其需要 VSCode 扩展——但我们认为这在目前是值得的折中，并正在努力恢复 DAP 的编辑器独立性。

6 实现见解

虽然 Gillian 调试的实现细节并非本工作的主要贡献，但我们希望强调一些已经显现、并可能引起更广泛社区兴趣的观察。

结构化日志。 调试器所需的结构化日志必然比普通日志更复杂，难以维护。经过试验，我们决定将维护结构的负担降至符号执行引擎的最低程度，因为引擎本身已经是复杂的技术组件，并且具有严格的正确性要求。具体而言，我们将关注点分离开来：引擎仅执行最低限度的记录工作，将所有日志报告存储在 SQLite 数据库中，并额外添加 previous 和 parent 两个字段；提升器随后利用这些报告构建具有更复杂控制流的树。前一操作与语言无关，并已从 Gillian 实例中独立出来。

重建源代码级命令。 WISL 提升器将已执行的 GIL 步骤聚合为 WISL 命令。每个已执行的 GIL 步骤都会被分配给一个部分命令，该部分命令表示一个尚未完全探索的源代码级 WISL 命令。我们使用注释标记会终止 WISL 命令的 GIL 命令，使提升器能够在所有路径都探索完毕后，将部分命令最终确定为源代码级树节点。由于我们完全掌控 WISL 编译器，因此可以直接扩展它，为 GIL 代码添加此类注释。这种方法也适用于其他编译器：例如，我们成功地将相同模式应用于一个新的 Gillian-C 实例，该实例使用 CBMC 的解析器和中间表示。我们无法调整原始的 Gillian-C 实例，因为它所采用的 CompCert 编译器是用 Rocq 编写的，因此难以修改。这表明，如果前端在编译期间保留足够的源代码级信息，我们的方法就能够在不同的语言实现之间移植。

高级嵌套。 树中树结构及其可视化嵌套（参见图 4）在具体呈现源代码级嵌套方面尤其有用，而中间表示级树中并不存在这种嵌套。例如，使用不变式注释的 WISL 循环会编译为带有前置条件和后置条件的 GIL 函数，但 WISL 提升器能够通过启动单独的验证，并将由此产生的树嵌套到外层函数的树中，重新整合循环体。这既展示了树中树结构的灵活性，也展示了提升器重构源代码级控制流的能力；这种控制流在编译过程中发生了根本性变化。

一切皆可交互。我们的方法与事后分析树查看器的一个关键区别在于，随着分析的推进，用户能够交互式地引导分析。为了实现这种交互性，我们修改了 Gillian 解释器，使其在每个 GIL 执行步骤后返回一个续延函数。在普通执行模式下，这些续延函数会被反复调用，直至执行完成，从而保留原有行为。而在调试模式下，调用续延函数时会传入一个分支标识符，以选择要继续的执行路径

。这种方法有望与其他已经采用续延传递风格的符号执行工具集成，例如 VeriFast 和 Viper；也有望与续延单子集成，例如 CN。

提升器通过在用户级命令与引擎操作之间进行转换，与这种引擎层面的交互协作。对于每条高层指令（“单步进入”“单步跳过”等），提升器都会确定扩展树所需的相应 GIL 步骤。

一种通用的 *UI* 工具包。Gillian 的调试 UI 在 VSCode 中通过结合 DAP 扩展与专用的树视图界面来实现，该界面通过网页视图面板呈现。调试器的早期迭代表明，DAP 扩展和自定义树视图界面在概念上都很少依赖 Gillian 或 VSCode，因此完全可以彼此分离；该界面处理的是抽象步骤，无需了解 Gillian、其中间表示或正在分析的语言，并且整个界面封装在单个基于网页的 UI 组件中。这一认识已通过名为 SEDAP（符号执行调试适配器协议）的原型得以实现。SEDAP 是一种 DAP 标准扩展的原型，配有库，可将交互式树视图作为可扩展的网页组件提供，并提供帮助程序，以便将此界面集成到 VSCode 调试会话中。正如 DAP 将编辑器与传统调试器连接起来，SEDAP 的进一步开发也应将编辑器与第2节讨论的分析工具连接起来；Gillian 在 VSCode 中的调试器如今已成为这一实践的首个示例。

基准测试	验证时间（毫秒）		
	无日志	File	数据库
单链表_iterative.wisl	242	680（2.8 倍）	487 (2.0x)
DLL_递归.wisl	91	259 (2.8x)	174 (1.9x)
单链表	303	390 (12.8x)	668 （2.2 倍）
双向链表	1875	81501（43.5 倍）	6335 （3.4 倍）
优先队列	292	5730（19.6 倍）	7815 (2.7倍)
排序	236	2369（10.0倍）	475 (2.0倍)
SLL.js	149	1192 (8.0倍)	1188 (8.0x)
DLL.js	202	1624 (8.1x)	1672 (8.3x)
ExprEval.js	246	6415 (26.1x)	4513（18.3 倍）

表1. 在 WISL、Gillian-C 和 Gillian-JS 中选取部分 Gillian 验证示例进行的基准测试结果。比较了不记录日志、详细文件日志记录和结构化数据库日志记录三种方式，并展示了后两种方式的减速因子。基准测试在一台配备 Intel i7-1065G7 处理器和 32 GiB 内存的笔记本电脑上进行。

性能影响。如表1所示，与不记录日志相比，将结构化日志记录到磁盘数据库会带来性能开销，但不会比 Gillian 默认的文本日志更差。有趣的是，在 C 验证中，数据库日志的表现远胜于文件日志，这是因为打印堆所用的逻辑十分复杂；这表明将日志记录关注点与引擎分离具有优势。我们注意到，目前尚未尝试优化结构化日志，而且还有许多

容易实现的性能改进空间：数据库查询可以异步执行或批量执行；也可以采用更合适的数据库模型（例如使用图数据库而非关系型数据库）；还可以使用 Protocol Buffers 等更高效的序列化方法来替代 JSON。

7 实证评估

我们在一所暑期学校中，对 Gillian 调试进行了形成性评估尝试。参加者是对形式化方法和编程语言研究感兴趣的高年级本科生和低年级博士生。在接受了 3 小时有关 SL、CSE 和 Gillian 的讲座后，参加者受邀参加两场各 90 分钟的实验课，尝试使用 Gillian 的语言服务器和调试器验证若干 WISL 程序。为此，我们设计了一套包含 13 个入门练习和 6 个进阶练习的题目：入门练习使用简单的链表程序介绍 Gillian 验证的各个方面；进阶练习则包括规模更大的程序，这些程序使用更复杂的数据结构（例如集合、双向链表和二叉搜索树），还包括 Collections-C 库部分内容的 WISL 翻译版本 [22]；完整练习列表见第B.1 节附录。

实验会话一开始采用引导式进行：我们与参与者一起完成了前几项练习，向他们介绍可用的调试器界面和 Gillian 证明策略，包括折叠和展开用户定义谓词、断言与应用证明策略，以及循环不变式。

活动结束时，我们邀请参与者填写一份匿名问卷，分享他们的体验，并可选择提交使用日志。这些日志在参与者的设备上本地生成并存储，记录了每次参与者与调试器及语言服务器交互时少量带时间戳的信息。在 ≈30名实验参与者中，19 人完成了问卷，其中 18 人提交了使用日志。我们利用这些数据评估 Gillian 调试在理解 CSE 和完成所提供练习方面的实用性。首先，我们通过分析一系列关于实验的李克特量表陈述的回答进行定性评估，以了解参与者对调试器实用性的主观印象。随后，我们分析所提交的使用日志，将这些回答与定量指标进行对照，从而了解参与者实际使用调试器和语言服务器的情况。

表 2. 中列出了所提供的李克特量表陈述及其汇总回答。总体而言，参与者对调试器实用性的评价压倒性地偏向正面（陈述 1/5/6）：63% 的回答表示非常赞同，另有 26% 表示比较赞同。此外，参与者认为调试器与验证过程集成良好（陈述 10），其他工具也能从类似的调试器中受益（陈述 11/12）。参与者还认可了调试器交互式、逐步操作特性所带来的更具体功能的实用性，包括探索单条执行路径（陈述 7），以及查看每个执行步骤的状态（陈述 9）。不过，他们对单独使用语言服务器的可行性（陈述 2/8）评价较低，回答从中立到略微负面不等。

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1 我不需要使用调试器来完成这些练习	11	6	1	1	0
2 语言服务器足以让我完成这些练习	1	6	5	6	1
3 调试器帮助我理解了 CSE	0	2	5	5	7
4 调试器帮助我理解了 Gillian 正在做什么	0	0	2	8	9
5 我发现调试器有助于完成这些练习	0	0	3	4	12
6 调试器帮助我理解了为什么验证失败	0	2	3	6	8
7 我发现逐一探索每条执行路径很有帮助，使用调试器	0	0	1	5	13
8 我理解了语言服务器向我显示的向我展示了	1	7	3	4	4
9 我发现查看执行过程中每个时点的状态很有帮助，使用调试器	0	0	3	5	11
10 将调试器作为验证工作流	0	1	2	4	12
11 这个调试界面也适用于我用过的类似工具	0	1	4	3	6
12 我会从这样的调试器中受益。我使用过的类似工具	0	0	1	4	9

表 2。用户调查中李克特量表的累计回答。

在定量方面，使用日志由带时间戳的记录构成，记录了每次参与者与调试器及语言服务器的交互；语言服务器记录显示练习文件的更改和分析结果（成功，或失败原因），调试器记录则标明用户是否启动或结束了会话，或在树中执行了一步操作。根据这些记录，我们计算每位参与者在每道题上花费的时间（并考虑在不同练习间切换以及长时间不活动的情况），以及其中有多少时间是在主动使用调试器（即最近 30 秒内与调试器发生过交互），从而粗略估算参与者使用调试器的时间占比。图 7 展示了这些占比的分布，结果表明，大多数参与者有 40% 至 60% 的时间近期与调试器进行过交互，这有力地说明参与者确实频繁且主动地使用了调试器。

尽管这项用户研究为评估 Gillian 调试提供了有力依据，但若干局限使我们无法获得更深入的见解。归根结底，实验室活动的首要目标是让参与者感到投入并有所收获，这意味着，正式用户研究所要求的受控条件和密切观察

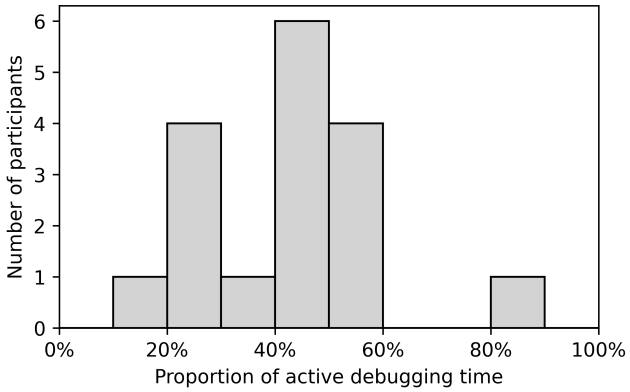

图 7. 分布：活跃调试会话中参与者时间所占的比例。

并不可行。参与者不必全神贯注于手头的练习，也不必独自完成练习，或全程参与。若不采用屏幕录制和日记等侵入性更强的手段，就难以收集关于参与者行为的更详细信息，而这些手段在此情境下或许并不合适。此外，学生对编程语言特定领域的教育仍处于早期阶段；至少 60% 的参与者表示，他们此前对证明助手、基于符号执行的工具、基于 CSE 的工具以及分离逻辑均无使用经验（见附录中的表 4 ）。相比之下，这套练习是根据此前面向一门本科四年级分离逻辑课程学员所做的 Gillian 调试演示设计的。因此，参与者完成的练习数量未达预期：他们最多完成了 13 道入门练习中的 6 道，且未能进行进阶练习。

与学生讨论并分析这些结果后，我们认为，Gillian 调试的潜力远不止于协助完成验证任务。我们提出了新的假设，认为它可以成为学习 CSE 及其实际应用的有力工具。例如，我们认为，相较于高级用户（他们会将调试器用于更复杂的调试任务），Gillian 的新用户会更广泛地使用调试器。尽管这一假设似乎与对语句 3、4 和 6 的积极回应相符，但仍需进一步研究，才能更有把握地断言这一点。同样，参与者对其他 CSE 工具缺乏使用经验，再加上他们对语句 11 和 12 的回应，表明交互式可视化也可能有助于定理证明或抽象解释等其他分析技术。

8 经验总结与未来工作

我们在开发和评估符号执行调试器的过程中，发现了让 CSE 工具更易于使用的潜力，同时也指出了仍需完成的工作。我们总结了通过这一项目获得的经验教训，并概述了未来研究方向。

用户研究。 在这一领域，用户研究并不常见，而本研究中开展用户研究的努力或许带来了意想不到的价值；尽管其首要目的是评估 Gillian 调试，但研究也引发了关于工具易用性本质，以及它如何因不同分析和用户经验水平而异的有趣讨论。除了展现教育价值外，未来的研究还可面向有经验的工具用户，让他们执行真实世界中的验证任务，并进行更细致的观察，从而针对特定设计选择开展聚焦评估。或许也应考察更客观的指标，例如生成的树的总体大小或分支数量。

Gillian. 由于 Gillian 具有高度模块化的特性，它成为探索此类调试方式的理想测试平台，并自然促成了编译器、分析引擎和提升器之间的职责分离。它与语言无关，因此无需重新实现调试器内部组件，就能探索为其他语言创建符号执行树的可能性；尤其可以利用 Gillian 对 JavaScript 和 C语言的实现，研究真实程序的调试分析。为此，我们已采用与 WISL 相同的技术，为第二个 Gillian-C 创建了一个提升器原型。除了半自动验证之外，Gillian 还能执行自动、有界的全程序符号测试（采用非组合式符号执行），并通过双向溯因真正查找漏洞；未来的工作将能够探索本研究所介绍的调试界面是否适用于这些分析。

扩展分析。 我们聚焦于小型、具有教学意义的验证示例，这自然引出了一个问题：如何将其扩展到真实世界的程序，包括针对真实世界语言提升器的复杂度，以及界面优雅处理更大执行树的能力。未来的工作将通过 Gillian 以及 CN 等其他分析工具，将我们的方法应用于对非平凡 C语言程序的分析。

呈现符号状态。 尽管本文主要关注如何直观地呈现分析形态，但符号状态的呈现方式也需要进一步关注。例如，WISL 提升器会以更接近 WISL 语法的形式打印引擎生成的 GIL 层级表达式，但状态中仍包含编译期间引入的中间变量。对于用户而言，这些变量可能令人困惑，因为它们看似与源程序毫无关联，但却是 Gillian 基于其中间表示进行分析所必需的。探究如何在源代码层有意义地呈现 CSE 状态，或许有助于改善用户体验。

错误消息。 错误消息是本实现中一个显著的痛点，用户调查也印证了这一点（见表 2，第 8 项）。验证错误消息很难实现，因为通常无法区分是实现有误、规约有误，还是证明注解不足。尽管如此，我们确实认为 Gillian 的错误消息仍有改进空间。交互式调试可以将一条不可更改的错误消息转变为与工具的对话，通过探索整棵分析树，将错误细化为更具体的信息。

界面通用性。 我们的 DAP 扩展和自定义界面已与 Gillian 和 VSCode 解耦，构成了 SEDAP。未来的工作将实现 SEDAP 的目标：为多种符号分析工具提供调试功能，并连接多种知名编辑器。我们希望这能进一步拓展 SEDAP 表示不同内部流程、分析类型或信息量的能力。SEDAP 能否适用于抽象解释？能否扩展为显示符号堆可视化？能否利用交互性，例如在实时调试会话中应用证明策略？

9 结论

我们为 Gillian 引入了一种新颖的调试界面，着眼于解决组合式符号执行工具领域中尚未得到充分探索的挑战：如何创建直观的调试界面。我们利用 VSCode 现有的调试界面提供标准 UI 元素，以带来熟悉的用户体验；同时开发了专门的自定义组件，以直观呈现组合式符号执行信息，并准确反映分析流程，包括分支、嵌套信息和状态匹配。

我们阐述了设计背后的概念原则，强调了这种方法向其他工具迁移的可能性，并讨论了实现过程中得出的有趣技术见解，希望这些经验能为该领域未来的工作提供参考。重要的是，我们尽可能地将 UI 与 Gillian 和 VSCode 解耦，从而构建了一个独立的库。我们希望继续开发这个库，并看到其他工具和编辑器采用它。

最后，我们通过对早期研究人员在一次实验室会议中使用该调试器的情况进行定量和定性分析，开展了初步评估，从而验证了它在理解和使用组合式符号执行方面的实用性。

尽管要开展关于提升组合符号执行工具可用性的正式研究，需要投入大量工程工作——过去也曾有过几次尝试，但后来都被放弃了——我们仍希望这项工作能成为该领域未来研究的垫脚石。

数据可用性声明

一个包含 Gillian 二进制文件、调试器 VSCode 扩展以及 WISL 示例程序（包括评估中使用的练习）的工件可在 [12] 获取。

致谢

我们衷心感谢 OpaleSjöstedt、Simon Park 和 Diego Cupello 不辞辛劳地为用户研究设计练习，也感谢参与研究的各位使我们的评估得以完成；感谢 Jessica Shi 和 Alastair Donaldson 分别就用户研究和本文的构思提出建议；感谢 Petar Maksimović 在项目启动之初给予支持；还要感谢 Radu Lacraru 和 Matthew Ho，他们的工程硕士项目为本项目奠定了工程基础。

本研究获得了 Gardner 的 UKRI 研究金“经验证的可信软件规约”以及 Meta 和亚马逊提供的资助。

参考文献

[1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.（编）：《演绎式软件验证——KeY 指南》，《计算机科学讲义》，第 10001 卷。施普林格国际出版，Cham（2016），ISBN 978-3-319-49811-9 978-3-319-49812-6，https://doi.org/10. 1007/978-3-319-49812-6[2] Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.：《Prusti 项目：Rust 的形式化验证》（特邀报告）。载于：NASA 形式化方法（第 14 届国际研讨会），第 88–108 页，Springer（2022），网址 https://link.springer.com/chapter/10. 1007/978-3-031-06773-0_5[3] Ayoun, S., Denis, X., Maksimović, P., Gardner, P.：《一种半自动 Rust 验证的混合方法》。《ACM 程序设计语言汇刊》**9**（PLDI），970–992（2025），https://doi.org/10.1145/3729289，网址 https://doi. org/10.1145/3729289[4] Cadar, C., Dunbar, D., Engler, D.：《KLEE：为复杂系统程序生成高覆盖率测试的非辅助式自动方法》。载于：第 8 届 USENIX 操作系统设计与实现会议论文集，第 209–224 页，OSDI'08，USENIX 协会，美国（2008 年 12 月）[5] Clarke, E., Kroening, D., Lerda, F.：《用于检查 ANSI-C 程序的工具》。载于：Jensen, K., Podelski, A.（编），《系统构建与分析工具及算法》（TACAS 2004），《计算机科学讲义》，第 2988 卷，第 168–176 页，Springer（2004），ISBN 3-540-21299-X[6] Eilers, M., Müller, P.：《Nagini：Python 静态验证器》。载于：《计算机辅助验证：第 30 届国际会议 CAV 2018，作为联合逻辑会议 FloC 2018 的一部分举行，英国牛津，2018 年 7 月 14–17 日，论文集，第一部分 30》，第 596–603 页，Springer（2018）[7] Ernst,G., Blau,J.,Murray, T.：《通过调试适配器协议进行演绎验证》。《理论计算机科学电子论文集》**338**，89–96（2021 年 8 月），ISSN 2075-2180，https://doi.org/10.4204/eptcs. 338.11，网址 http://dx.doi.org/10.4204/EPTCS.338.11[8] Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.：《Gillian，第一部分：用于符号执行的多语言平台》。载于：程序设计语言设计与实现大会，PLDI（2020），ISBN 9781450376136，https://doi.org/10.1145/3385412.3386014，网址 https://doi.org/10.1145/3385412.3386014[9] Hentschel, M., Bubel, R., Hähnle, R.：《符号执行调试器（sed）：用于交互式符号执行、调试、验证等的平台》。《国际软件工具与技术转移杂志》（2019），ISSN 1433-2787，https://doi.org/10.1007/s10009-018-0490-9，网址 https://doi.org/10.1007/s10009-018-0490-9[10] Holter, K., Hennoste, J.O., Saan, S., Lam, P., Vojdani, V.：《使用 GobPie 进行抽象调试》。载于：第 2 届 ACM 国际研讨会论文集

载于《未来调试技术》，第 32–33 页，DEBT 2024，美国计算机协会，纽约州纽约（2024），ISBN 9798400711107，https://doi.org/10.1145/3678720.3685320，URL https://doi.org/10.1145/3678720.3685320[11]Jacobs, B.: verifast-vscode (2023), https://github.com/verifast/ verifast-vscode [访问日期：2025/10/17][12] Karmios, N., Ayoun, S.E., Gardner, P.: [工件] Gillian 调试：穿梭于（组合符号执行）树中（2026 年 1 月），https://doi.org/10.5281/zenodo.18311740，URLhttps://doi.org/10.5281/zenodo.18311740[13] Lööw, A., Nantes-Sobrinho, D., Ayoun, S.E., Cronjäger, C., Maksimović, P., Gardner, P.: 用于正确性与错误性推理的组合符号执行。载于：Aldrich, J., Salvaneschi, G.（编）第 38 届欧洲面向对象编程大会（ECOOP 2024），信息学莱布尼茨国际会议论文集（LIPIcs），第 313 卷，第 25:1–25:28 页，达格斯图尔城堡—莱布尼茨信息学中心，德国达格斯图尔（2024），ISBN 978-3-95977-341-6，ISSN 1868-8969，https://doi. org/10.4230/LIPIcs.ECOOP.2024.25，URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.25[14]Lööw, A., Park, S.H., Nantes-Sobrinho, D., Ayoun, S.E., Sjöstedt, O., Gardner, P.: 面向未来 700 种内存模型的组合符号执行。《ACM 程序设计语言汇刊》**9**(OOPSLA2)（2025 年 10 月），https: //doi.org/10.1145/3763151，URL https://doi.org/10.1145/3763151[15]Maksimović, P., Ayoun, S.E., Santos, J.F., Gardner, P.: Gillian，第二部分：面向 JavaScript 和 C语言的真实世界验证。载于：计算机辅助验证（CAV）（2021），ISBN 978-3-030-81687-2，https://doi. org/10.1007/978-3-030-81688-9_38，URL https://doi.org/10.1007/ 978-3-030-81688-9_38[16]Meta：Infer 静态分析器（2025），https://fbinfer.com/ [访问日期：2025/10/17][17] 微软：Visual Studio Code（2024），https://code.visualstudio.com[访问日期：2025/10/17][18] 微软：调试适配器协议（2025），https://microsoft.github.io/debug-adapter-protocol/ [访问日期：2025/10/17][19] 微软：语言服务器协议（2025），https://microsoft.github. io/language-server-protocol/ [访问日期：2025/10/17][20]Müller, P., Schwerhoff, M., Summers, A.J.: Viper：基于权限推理的验证基础设施。载于：Jobstmann, B., Leino, K.R.M.（编）《验证、模型检查与抽象解释》，第 9583 卷，第 41–62 页，施普林格·柏林·海德堡，柏林，海德堡（2016），ISBN 978-3-662-49121-8 978-3-662-49122-5，https://doi.org/10.1007/ 978-3-662-49122-5_2[21] O'Hearn, P.: 分离逻辑。《ACM 通讯》**62**(2), 86–95（2019 年 1 月），ISSN 0001-0782，https://doi.org/10.1145/3211968，URL https:// doi.org/10.1145/3211968

[22]Panić, S.: Collections-C（2025），https://github.com/srdja/ Collections-C [访问日期：2025/10/17][23] Raad, A., Berdine, J., Dang,H.H., Dreyer, D., O’Hearn,P., Villard, J.: 关于错误存在性的局部推理：不正确性分离逻辑。载于：Lahiri, S.K., Wang, C.（编）《计算机辅助验证》，第 225–252 页，施普林格国际出版，Cham（2020），ISBN 978-3-030-53291-8, https://doi.org/10.1007/978-3-030-53291-8_14[24] Meta 的 Infer 团队：Infer Pulse（2025），https://fbinfer.com/docs/ checker-pulse [访问日期：2025/10/17][25] Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra：Go 程序的模块化规约与验证（扩展版）（2021）

A WISL

A.1 语法

值

$$v \in \mathcal{V}_W \stackrel{\text{def}}{=} \texttt{null} \mid n \in \mathbb{N} \mid b \in \mathsf{Bool} \mid p \in \mathcal{L} \times \mathbb{N}$$

表达式

$$E \in \mathcal{E}_W \stackrel{\text{def}}{=} v \in \mathcal{V}_W \mid \texttt{x} \in \mathsf{PVar} \mid \ominus E \mid E \oplus E$$

命令

$$\begin{array}{lll} C \in \mathcal{C}_W \stackrel{\text{def}}{=} & \texttt{skip} \mid \texttt{x} := E \mid \texttt{x} := f(\vec{E}) & \text{Basic} \\ & \mid \texttt{x} := [E] \mid [E] := E \mid \texttt{x} := \texttt{new}(E) \mid \texttt{free}(E) & \text{Memory management} \\ & \mid C; C \mid \texttt{if}\ (E)\ C\ \texttt{else}\ C \mid \texttt{while}\ (E)\ C & \text{Control flow} \end{array}$$

函数

$$\texttt{function}\ f(\vec{E})\,\{C; \texttt{return}\ E\} \in \mathcal{F}unc_W$$

A.2 注释

*(** 此命令如何映射到 WISL 语句？ *)*类型 语句_类型 ⫤普通的 布尔值*(** 构成 WISL 语句一部分的命令，以及此命令是否终止该语句 *)*返回的 布尔值*(** 与普通相同，但专用于 WISL return 语句 *)*隐藏 *(** 不映射到特定 WISL 语句的命令 *)*循环前缀 *(** 循环体函数初始命令的特殊情况 *)*

*(** 此命令处可以出现哪种分支？ *)*类型分支_类型 ⫤ 如果-否则| 当循环

*(** 这条命令中可能出现哪种嵌套？ *) type nest_kind* ⫤ *LoopBody of string (** WISL 循环体抽象的特殊情况 *)*| *FunCallof string (** 源代码级函数调用；用于在以非组合方式调用函数时告知嵌套信息 *) type annot* = *{ source_loc: Location.t option; stmt_kind : stmt_kind; branch_kind : branch_kind option; nest_kind : nest_kind option; }*

Line	语句	stmt_kind	分支 _kind
2	跳转? （x = null）...	普通 true	某个如果-否则
3	n := 0	普通 true	None
4	跳转 end	隐藏	None
5	_var0: = i_add(x , 1)	普通 true	None
6	跳转? (_var0 是指针) ...	普通 false	None
7	失败 "无效指针"	普通 true	None
8	t := 加载< _var0>	普通 true	None
9	n := llen(t)	普通 true	None
10	skip	隐藏	None
11	ret := n	返回 false	None
12	返回	返回真	None

表 3。每个 GIL 命令的注释 `llen()`（参见图 1）。

B 进一步评估详情

Area	None	<1yr	1–3年	>3年
证明助手	12	5	1	1
基于符号执行的工具	13	5	1	0
基于 CSE 的工具	18	1	0	0
分离逻辑	16	3	0	0

表4. 实验参与者报告的相关领域先前经验情况。

B.1 实验练习

这里列出了我们评估中所涉及实验室的练习，并统计了报告尝试和／或完成每项练习的参与者人数。

1. **入门（自动）** — 19 人尝试，19 人完成 递归列表长度，有一个小错误；与图 1 类似。这个示例曾在现场简要演示，之后作为一个简单示例提供给参与者，让他们自行尝试并熟悉 Gillian 和调试器。2. **入门（手动）** — 19 人尝试，19 人完成 递归列表长度，现已修正，但启用了手动模式。这个示例也曾在现场演示；它介绍了证明策略，并展示 Gillian 如何通过折叠与展开处理谓词。

3. **折叠**— 16 人尝试，2 人完成 手动折叠的一个有趣边界情形：即使链表的基本情形不会消耗状态中的任何内容，也必须手动折叠。4. **函数调用**— 15 人尝试，1 人完成 一个示例，展示如何使用折叠/展开，根据函数调用的结果归纳地构造谓词。5. **分支与递归**— 9 人尝试，4 人完成 展示了在不同分支中应用不同证明策略的必要性。6. **断言**— 0 人尝试，0 人完成 介绍断言策略，用于确保条件成立，并绑定逻辑变量以供后续引用。7. **应用**— 0 人尝试，0 人完成 介绍引理，以及如何使用应用策略。8. **列表删除**— 0 人尝试，0 人完成 一组关于 list_remove() 的练习中的第一个。用于测试对折叠与展开的理解。9. **自动模式**— 0 人尝试，0 人完成 接续上一道练习。要求参与者粘贴之前的解答，然后查看在禁用手动模式后可以省略哪些策略。10. **更强的规约**— 0 人尝试，0 人完成 接续上一道练习。练习指出 list_remove() 的规约较弱，并要求编写一个谓词来强化该规约。11. **规约变更**— 0 人尝试，0 人完成 接续上一道练习。练习描述了对 list_remove() 行为的一项预期变更；参与者必须对函数体进行必要修改，并相应更新后置条件。12. **不变式**— 0 人尝试，0 人完成 WISL 中循环不变式的入门练习。练习给出了循环不变式，参与者必须插入策略，以确保循环体内外的证明都能成功。13. **斐波那契**— 0 人尝试，0 人完成 最后一项入门练习。通过使用迭代式斐波那契函数，进一步熟悉循环不变式，同时展示“纯”谓词（以及用于折叠它们的引理）可以重复使用。14. **列表转集合**— 0 人尝试，0 人完成 第一项进阶练习。内容是创建一个 “ 集合列表 ”：一种不含重复元素、可作为集合使用的列表，并用谓词描述这一性质。附加挑战要求找出给定规约中的一个弱点，然后强化它。15. **双向链表**— 0 人尝试，0 人完成 验证双向链表的 concat()；其中涉及复杂的引理应用。

16. **括号匹配** —已尝试 0 次，已完成 0 次 给定一个用于检查字符串括号是否匹配的函数，需要编写一个描述括号匹配属性的谓词。17. **二叉搜索树** —已尝试 0 次，已完成 0 次 验证若干二叉搜索树函数，需要大量断言和引理应用。18. **Collections-C 单向链表**—已尝试 0 次，已完成 0 次 通过将 Collections-C 中的一些单向链表函数转换为 WISL，初步体验如何验证真实世界风格的代码 [22]。19. **更多循环不变式** —已尝试 0 次，已完成 0 次 以迭代方式而非递归方式呈现前面的一些单向链表算法。