北京大学

博士研究生学位论文

题目：概率化差异调试技术研究

姓　　名：王冠成

学　　号：1801111361

院　　系：计算机学院

专　　业：计算机软件与理论

研究方向：软件工程与软件开发方法学

导　　师：熊英飞 副教授

二〇二三 年 十 月

摘要

差异调试（Delta Debugging）问题是软件工程中的重要问题。差异调试是指，给定输入集合，在由输入集合的所有子集构成的庞大搜索空间中，自动化地搜索满足规定性质的尽可能小的输入集合的子集。通常，一个二值函数被用于检查某个集合是否满足规定的性质，若满足则返回通过（T）作为反馈，若不满足则返回失败（F）作为反馈，本文将这样的函数称作反馈函数。例如，在编译器开发过程中，给定一个揭露编译器缺陷的测试用例（一个程序），差异调试技术被用于找到一个较小规模的测试用例，在保证重现编译缺陷的同时，提高开发人员调试缺陷的效率。这里，假设以行粒度进行处理，输入集合是由该程序代码行组成的集合，搜索空间由这些代码行组成的程序的集合构成，反馈函数用于检查是否触发相同的编译缺陷。此外，差异调试问题还多存在于缓解软件膨胀、回归故障定位、测试集合约简等领域。

差异调试技术用于解决差异调试问题，其效率和效果是长期制约差异调试技术应用范围的首要因素。虽然现存多种算法来高效地解决差异调试问题，但是，目前最先进的算法的效率和效果仍然不令人满意。

本文观察到，现有的差异调试技术固定化地尝试删除输入对象中的元素，当反馈函数提供当前解具有规定性质的反馈时，当前解将作为新的输入用于后续的简化步骤。而当反馈函数提供当前解不具有规定性质的反馈时，现有技术无法利用这些反馈。而这种固定化存在于两个方面，一方面在于技术本身以固定顺序尝试删除元素导致的固定化，另一方面在于为达到最小不动点状态反复应用这样的固定顺序导致的固定化。

为了解决上述问题，本文提出概率化差异调试框架。根据给定的概率模型，指定待选择的集合，通过反馈函数提供的反馈信息更新模型，该过程直到模型收敛。概率化差异调试框架需要针对问题设计概率模型才能应用，本文针对集合作为输入差异调试和针对程序作为输入的差异调试的场景，分别提出了面向集合的概率模型和面向语法树的概率模型。最后，本文提出了基于细粒度反馈的概率化差异调试技术，探索精化反馈函数提供的反馈信息能否进一步带来概率化差异调试技术性能上的提升。本文所提出的一整套技术通过结合概率打破了现有技术固定化尝试删除元素带来的局限性，既能灵活地探索更大范围的搜索空间，又可以充分地利用反馈信息指导差异调试过程，最终达到提升解决差异调试问题技术的效率和效果的目的。

具体地，本文的主要研究工作及创新点如下：

1. **概率化差异调试框架**。本文提出了概率化差异调试框架。给定一个概率模型，该框架利用概率模型来估计每个元素被保留在最终结果中的概率。在每次迭代中，

根据概率模型指定一个最大化期望收益的元素集合，并通过反馈函数反馈该集合是否仍满足规定的性质。然后，根据反馈信息计算后验概率，并更新概率模型。直到达到收敛条件，算法停止并返回简化后的结果。进一步地，证明了当输入满足某些性质时，最小子集的存在性；同时，讨论了该框架产生结果的正确性。

2. **面向集合的概率模型**。概率化差异调试框架需要针对问题设计概率模型才能应用。大量差异调试应用场景中，输入可以看作是一个元素集合。为应用框架到这样的场景上，本文提出了面向集合的概率模型。为验证该模型的有效性，本文将该模型与框架结合形成面向集合的概率化差异调试技术进行实验验证，验证结果表明，面向集合的概率化差异调试技术显著提高了差异调试的效果和效率。进一步地讨论了面向集合的概率化差异调试技术保证结果极小性或最小性的情形和最坏情况下调用反馈函数的次数。
3. **面向语法树的概率模型**。简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。为了在差异调试中充分考虑语法结构上的依赖关系，本文提出了面向语法树的概率模型。为验证该模型的有效性，本文将该模型与框架结合形成面向语法树的概率化差异调试技术进行实验验证，验证结果表明，面向语法树的概率化差异调试技术显著提高了差异调试的效果和效率。
4. **基于细粒度反馈的概率化差异调试技术**。现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。基于这个观察，本文提出了基于细粒度反馈的概率化差异调试技术，探索精化反馈函数提供的反馈信息能否进一步带来概率化差异调试技术性能上的提升，支持在不修改概率模型的情况下引入细粒度反馈。由于反馈函数内部包含不同功能的子函数，根据子函数间的组织特点，进而提出两种变体技术。本文的实验验证结果表明，精化反馈函数提供的反馈信息能够进一步提升概率化差异调试技术的效率和效果。这些技术可在现有概率化差异调试技术引入细粒度反馈，以便进一步提升差异调试的效率和效果。

综上，本文提出了一系列技术，形成了一套概率化差异调试技术。该系列技术在公认的差异调试数据集和本文收集的编译器调试数据集上超过了最优技术。本文提出的技术已经成为后续工作的基础，已经有多个第三方团队在本文技术和方法的基础上构建了更先进的领域特定的差异调试技术。

关键词：差异调试，概率模型，抽象语法树，贝叶斯优化

Research on Probabilistic Delta Debugging

Guancheng Wang (Computer Software and Theory)
Supervised by Prof. Yingfei Xiong

ABSTRACT

The Delta Debugging problem is an important issue in software engineering. Delta Debugging refers to the automated search for the smallest possible subset of an input set, within the vast search space composed of all subsets of the input set, that satisfies a specified property. Typically, a binary function is used to check if a given set satisfies the specified property. If it does, the function returns True (T) as feedback; if not, it returns False (F). In this thesis, I refer to such a function as the feedback function. For example, in the context of compiler development, when provided with a test case that exposes a compiler defect (a program), Delta Debugging techniques are employed to find a smaller test case while ensuring the reproduction of the compilation defect. This process enhances the efficiency of developers in debugging defects. In this scenario, assuming line granularity, the input set consists of a collection of code lines from the program, and the search space is composed of collections of these code lines that make up different programs. The feedback function is used to check if the same compilation defect is triggered. Additionally, Delta Debugging problems are prevalent in areas such as mitigating software bloat, regression fault localization, and simplifying test sets.

Delta Debugging techniques are employed to address the Delta Debugging problem, and their efficiency and effectiveness have long been primary factors constraining the scope of their application. While several algorithms exist to efficiently tackle Delta Debugging problems, the efficiency and effectiveness of the most advanced algorithms currently available still leave much to be desired.

In this thesis, I observe that existing Delta Debugging techniques rigidly attempt to remove elements from the input objects. When the feedback function provides positive feedback indicating that the current solution satisfies the specified property, the current solution is used as the new input for subsequent simplification steps. However, when the feedback function provides negative feedback indicating that the current solution does not satisfy the specified property, existing techniques fail to utilize this feedback effectively. This rigidity is evident

in two aspects. Firstly, it arises from the fixed order in which elements are attempted to be removed by the technique itself. Secondly, it results from the repeated application of such a fixed order to achieve the fix-point state.

To address the aforementioned issues, this thesis proposes **The Probabilistic Delta Debugging Framework**. Using a given probabilistic model and specifying the set to be selected, this framework updates the model based on feedback provided by the feedback function, and this process continues until the model converges. The Probabilistic Delta Debugging Framework requires the design of probabilistic models tailored to the problem at hand. In this thesis, I introduce probabilistic models for scenarios involving set-based input Delta Debugging and program-based input Delta Debugging, referred to as **The Set-oriented Probabilistic Model** and **The Syntax-tree-oriented Probabilistic Model**, respectively. Finally, this thesis introduces **The Probabilistic Delta Debugging Technique Based on Fine-grained Feedback**, exploring whether refining the feedback provided by the feedback function can further improve the performance of Probabilistic Delta Debugging techniques. The entire set of techniques proposed in this thesis, by incorporating probability, breaks free from the limitations of existing techniques that rigidly attempt to remove elements. This approach allows for flexible exploration of a broader search space while effectively leveraging feedback information to guide the Delta Debugging process. Ultimately, these techniques aim to enhance the efficiency and effectiveness of solutions to Delta Debugging problems.

Specifically, the main research contributions and innovations of this thesis are as follows:

- **The Probabilistic Delta Debugging Framework**: This thesis introduces The Probabilistic Delta Debugging Framework. Given a probabilistic model, this framework utilizes the model to estimate the probability of retaining each element in the final result. In each iteration, it specifies a set of elements that maximizes the expected gain according to the probabilistic model. The feedback function is used to determine if this set still satisfies the specified property. Subsequently, posterior probabilities are calculated based on the feedback information, and the probabilistic model is updated. The algorithm continues until a convergence condition is met, at which point it stops and returns the simplified result. Furthermore, the thesis proves the existence of a minimal subset when certain properties are satisfied in the input. The correctness of the results generated by this framework is also discussed.
- **The Set-oriented Probabilistic Model**: The Probabilistic Delta Debugging Framework requires the design of probabilistic models tailored to the problem. In many Delta

Debugging application scenarios, the input can be considered as a set of elements. To apply the framework to such scenarios, this thesis introduces The Set-oriented Probabilistic Model. To validate the effectiveness of this model, the thesis combines it with the framework to create a Set-oriented Probabilistic Delta Debugging Technique and conducts experimental verification. The results indicate that the Set-oriented Probabilistic Delta Debugging Technique significantly improves the effectiveness and efficiency of Delta Debugging. Additionally, the thesis discusses scenarios where the set-oriented technique ensures the return of minimum or minimal results and the worst-case number of feedback function calls.

- **The Syntax-tree-oriented Probabilistic Model**: Program simplification is one of the most typical applications of Delta Debugging. While set-oriented models can also be used for programs, their effectiveness is limited because they do not fully consider dependencies in program structures. To address this limitation and account for syntax tree dependencies, this thesis introduces The Syntax-tree-oriented Probabilistic Model. To validate the effectiveness of this model, the thesis combines it with the framework to create a Syntax-tree-oriented Probabilistic Delta Debugging Technique and conducts experimental verification. The results demonstrate that the Syntax-tree-oriented Probabilistic Delta Debugging Technique significantly improves the effectiveness and efficiency of Delta Debugging.
- **The Probabilistic Delta Debugging Technique Based on Fine-grained Feedback**: Existing Delta Debugging techniques treat the feedback function as binary, providing only pass or fail feedback. However, practical feedback functions often consist of multiple smaller feedback functions, and the results of these smaller functions can offer finer-grained feedback information. Recognizing this, the thesis introduces The Probabilistic Delta Debugging Technique Based on Fine-grained Feedback, which explores whether refining the feedback provided by the feedback function can further enhance the performance of Probabilistic Delta Debugging Techniques. Two variant techniques are proposed based on the organization of sub-functions within the feedback function. Experimental results show that refining the feedback function's information can further improve the efficiency and effectiveness of Probabilistic Delta Debugging Techniques. These techniques can introduce fine-grained feedback into existing Probabilistic Delta Debugging Techniques to further enhance the efficiency and effectiveness of Delta Debugging.

In summary, this thesis presents a series of techniques that form a comprehensive set of Probabilistic Delta Debugging Techniques. These techniques outperform the state-of-the-art on recognized Delta Debugging datasets as well as a dataset of compiler debugging cases collected in this thesis. The technologies proposed in this thesis have become the foundation for subsequent research efforts, with several third-party teams building more advanced domain-specific Delta Debugging techniques based on the methods and techniques presented in this thesis.

KEY WORDS: Delta Debugging, Probabilistic Model, Abstract Syntax Tree, Bayesian Optimization

目录

第一章　引言

软件测试与调试在保障软件安全和软件质量等方面发挥着重要作用，随着软件规模的不断增长和复杂性不断增加，软件测试与调试也面临着新的挑战。一个典型的软件测试与调试流程是，开发人员利用自动化测试用例生成工具对被测试软件进行测试，再通过揭错的测试用例定位并修复被测试软件中的缺陷。然而，自动化生成的测试用例具有大规模和不易阅读等特点。例如，一个典型的编译器测试用例通常具有上万行代码的规模，包含数以万计随机生成的变量名、函数名等。开发人员通过这样的测试用例定位缺陷，需要耗费大量精力，且在调试过程中反复运行规模较大的测试用例通常消耗更多的计算资源。自动化测试用例简化工具能够将上万行揭错的测试用例简化到上百行甚至更小规模，开发人员通过简化后的测试用例能够更加方便地定位被测试软件中的缺陷。本质上，自动化测试用例简化是求解一个搜索问题，即在庞大的求解空间中，自动化地搜索满足规定性质的尽可能小的解，并在搜索过程中尽可能少地调用具有一定开销的反馈函数以检查当前解是否满足规定的性质，这属于差异调试问题（Delta Debugging）的范畴。此外，差异调试问题还多存在于缓解软件膨胀、回归故障定位、测试集合约简等领域。因此，研究差异调试问题来辅助软件开发具有重要的科学意义和实用价值。

差异调试问题最早在 1990 年代末引入[1]，Zeller 等人[2] 系统性地阐述了差异调试问题并提出了求解这一问题的算法。这一算法在后来几十年的差异调试相关研究中，成为大多数差异调试问题的基础。为了进一步提高差异调试的效率和效果，本文提出了一系列概率化差异调试技术。

具体来说，本章首先介绍本文所研究的问题，引出本文的主要工作和创新点，最后展示全文的结构。

1.1　问题的提出

差异调试是一种在保证某种特定性质的前提下，自动化减少一组元素的技术[3]。它在许多领域中得到了应用，例如编译器调试[4-8]、回归故障定位[1, 9, 10]、隔离故障的因果链[2, 11, 12]，以及缓解软件膨胀问题[13]。

差异调试可以形式化地定义如下。设 $\mathbb{X}$ 是所有感兴趣对象的集合，$\phi : \mathbb{X} \to \{F, T\}$ 是一个反馈函数，用于确定一个集合是否具有规定性质（T）或不具有（F），$|X|$ 表示集合 $X \in \mathbb{X}$ 的大小。给定一个满足 $\phi(X) = T$ 的集合 $X \in \mathbb{X}$，差异调试的目标是找到另一个集合 $X^* \in \mathbb{X}$，使得 $|X^*|$ 尽可能小且 $\phi(X^*) = T$，即 X^* 满足该性质。例如，在编译

器开发中，差异调试被用于找到一个较小的程序，以重现编译失败。在编译器开发过程中，给定一个揭露编译器缺陷的测试用例（一个程序），差异调试技术被用于找到一个较小规模的测试用例，保证重现编译缺陷以提高开发人员调试缺陷的效率。这里，$\mathbb{X}$ 是程序的集合，X 是一个可能很大的程序，导致编译失败，而 ϕ 则是用于判断是否触发相同编译缺陷的反馈函数。

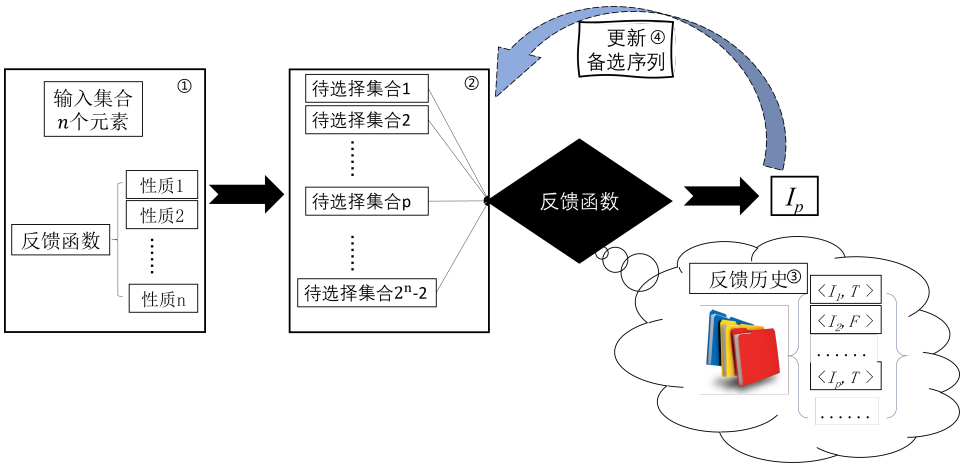


图 1.1　现有差异调试技术的工作流程示意图

图 1.1展示了现有差异调试技术的基本工作流程。差异调试技术接受输入集合和一段反馈函数作为输入（如图 1.1中①），输入正确的情况下，输入集合能够通过反馈函数，即反馈函数结果返回通过（T）。在差异调试迭代过程中，不断地从输入集合中寻找可能的解，每一个被选择的集合需要经过反馈函数的验证，理想情况下差异调试最终返回一个全局最优解，即在所有输入集合的子集合中通过反馈函数且元素数量最小的子集合。通过反馈函数的子集（如图 1.1中 I_P）作为差异调试迭代过程中新的输入集合，并产生图 1.1②新的待选择集合。使得反馈函数失败的子集会被反馈历史记录（如图 1.1中③），当待选择集合存在在反馈历史中时，会在反馈历史中查询反馈结果从而跳过执行反馈函数的步骤。当元素个数为 n 时，搜索空间为 2^n，即差异调试问题无法在多项式时间内求解。当 n 数值较小时，搜索空间较小，使得遍历每一个子集成为可能，这样的方式可以很容易地发现全局最优结果。但是，在实际应用场景下，n 的数值通常较大，使得搜索空间指数级爆炸增长，无法通过遍历的方式尝试每一个子集成为解的可能性。

现有的差异调试技术大都建立在 ddmin 算法[4] 的基础上。ddmin 算法输入集合 $X \in \mathbb{X}$。在每次迭代中，ddmin 将 X 分割为 n 个子集合，并依次尝试从 X 中删除每个子集及其补集。初始时，n 为 2，并在每次迭代中被翻倍。最终，当集合中每个元素被单独尝试删除一次后，ddmin 算法返回结果 X'。为了得到更小的满足规定性质的集合，在实际应用中 ddmin 算法通常会被反复调用，并将上一次 ddmin 算法返回的结果 X'

作为新的输入集合，直到 ddmin 算法无法约简集合中元素。后续基于 ddmin 技术的差异调试技术假设更复杂的领域特定结构，并将 ddmin 应用于从这些结构中产生的集合。例如，HDD[14] 假设输入具有树形结构，并仅将 ddmin 应用于同一层上所有兄弟节点组成的集合。CHISEL[13] 进一步考虑了 C 程序中元素之间的数据流和控制流的依赖关系，并以不破坏依赖关系的方式应用 ddmin 算法。然而，目前最先进的差异调试算法的效率和效果仍然不尽人意。例如，最先进针对 C 语言的差异调试技术 CHISEL[13] 可能需要调用上千次反馈函数、耗费长达 3 个小时来缩减一个包含一万多行代码的程序，而缩减后的程序可能会比理想的目标多出 2 倍的冗余代码。

本文观察到，现有的差异调试技术固定化地尝试删除输入对象中的元素，当反馈函数提供当前解具有规定性质的反馈时，当前解将作为新的输入用于后续的简化步骤。而当反馈函数提供当前解不具有规定性质的反馈时，现有技术仅将它们加入反馈历史中，无法利用这些反馈指导差异调试过程。而这种固定化存在于两个方面，一方面在于技术本身以固定顺序尝试删除元素导致的固定化，另一方面在于为达到最小不动点状态反复应用这样的固定顺序导致的固定化。

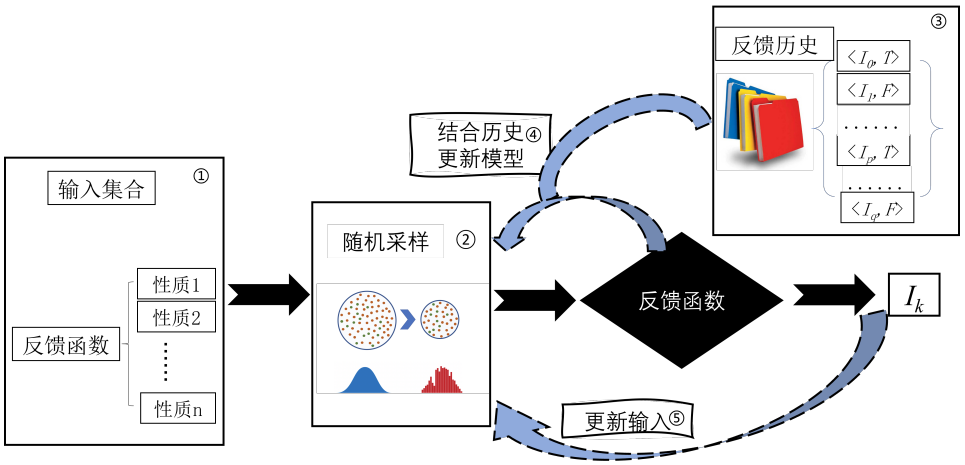


图 1.2　差异调试技术缺点的直观示例

相对应地，本文考虑用概率化的方法，进一步提高差异调试的效率和效果。通过结合概率化的方法，本文期望在差异调试过程中更加灵活地尝试删除元素，同时，在差异调试迭代过程中，充分地利用反馈信息，即当反馈函数提供当前解具有规定性质的反馈时，当前解将作为新的输入用于后续的简化步骤，而当反馈函数提供当前解不具有规定性质的反馈时，能够将反馈信息用于指导差异调试过程。一方面，如图 1.2中②所示，将现有差异调试技术以固定顺序尝试删除元素调整为结合概率进行随机采样的方式，帮助跳出局部最优解产生规模更小的简化结果；另一方面，如图 1.2④所示，将利用反馈函数提供的当前解不具有规定性质的反馈信息，更新概率，指导采样更优地尝试删除元素，更快地（更少地调用反馈函数）得到理想的简化结果。

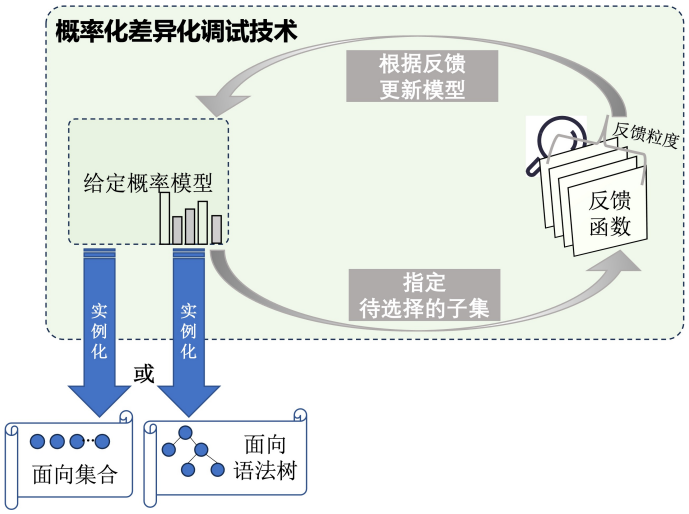


图 1.3　概率化差异调试技术

总的来说，针对差异调试问题的研究面临着严峻挑战。一方面，考虑到充分利用差异调试过程中积累的反馈信息指导差异调试过程，没有现有的方法能够直接应用于差异调试问题；另一方面，在现有技术的基础上，进一步利用反馈函数提供的当前解不满足规定性质的反馈信息指导差异调试过程，而不引入额外的开销是十分困难的。

1.2　本文主要工作和创新点

现有差异调试技术存在的性能问题，主要体现在两个方面。第一，差异调试的效果不理想，即返回的结果规模不够小，无法达到帮助开发人员定位缺陷、修复缺陷的要求；第二，差异调试的效率不理想，差异调试在一次运行过程中调用过多次反馈函数，导致运行缓慢，无法满足辅助开发人员高效地修复大量缺陷的需求。本文观察到，现有的差异调试技术固定化地尝试删除输入对象中的元素，当反馈函数提供当前解具有规定性质的反馈时，当前解将作为新的输入用于后续的简化步骤。而当反馈函数提供当前解不具有规定性质的反馈时，现有技术仅将它们加入反馈历史中，无法利用这些反馈指导差异调试过程。而这种固定化存在于两个方面，一方面在于技术本身以固定顺序尝试删除元素导致的固定化，另一方面在于为达到最小不动点状态反复应用这样的固定顺序导致的固定化。

为解决上述问题，本文采用概率化的方法，提出一系列技术，形成如图 1.3所示的一套**概率化差异调试技术**。首先，本文提出了**概率化差异调试框架**。给定概率模型，根据概率模型指定待选择的集合，将反馈函数用于检查该集合是否满足规定的性质。若满足规定的性质（通过的反馈），概率模型将该集合作为新的输入并指定其子集作为下一次选择的集合；若不满足规定的性质（失败的反馈），根据失败的反馈信息更新概率

模型。该过程直到模型收敛停止。概率化差异调试框架需要针对问题设计概率模型才能应用。大量差异调试应用场景中，由于输入被看作是一个元素集合，为应用框架到这样的场景上，本文提出了**面向集合的概率模型**。此外，简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。为了在差异调试过程中充分考虑语法结构上的依赖关系，本文提出了**面向语法树的概率模型**。针对输入是集合和程序的情况，可给定上述两种模型，分别被实例化为面向集合的概率化差异调试技术和面向语法树的概率化差异调试技术。

最后，本文观察到，现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。基于这个观察，本文提出了**基于细粒度反馈的概率化差异调试技术**，探索精化反馈函数提供的反馈信息能否进一步带来概率化差异调试技术性能上的提升，支持在不修改概率模型的情况下引入细粒度反馈。由于反馈函数内部包含不同功能的子函数，根据子函数间的组织特点，进而提出两种变体技术。这些技术可根据反馈函数的特性，在现有概率化差异调试技术引入细粒度反馈，以便进一步提升差异调试的效率和效果。

通过概率模型，在每一轮差异调试迭代过程中利用积累的反馈历史更新概率模型的概率化方式，本文提出的技术不再像现有技术那样使用固定的顺序尝试删除输入集合中的元素。另一方面，通过精化反馈函数提供的反馈信息，利用细粒度的反馈信息，进一步提高概率化差异调试技术的效率和效果。总之，本文提出了一套概率化差异调试技术提升差异调试的效果和效率，具体如下所述。

1.2.1　概率化差异调试框架

现有大多数差异调试技术采用固定顺序尝试删除输入集合中的元素，且在差异调试过程中没有充分利用历史的反馈信息，导致差异调试技术的效率和效果不够理想。理想情况下，差异调试技术返回的结果应该是满足规定性质的最小结果，然而，由于搜索空间和输入集合元素数量呈指数级关系，实际情况中很难确定某个结果是最小的。

鉴于本文的主要目标是利用差异调试迭代过程中积累的历史反馈信息指导差异调试过程，受到贝叶斯优化思想的启发，本文提出了一种概率化差异调试框架，旨在显著提升差异调试技术的效果和效率。给定概率模型，根据概率模型指定待选择的集合，将反馈函数用于检查该集合是否满足规定的性质。若满足规定的性质（通过的反馈），概率模型将该集合作为新的输入并指定其子集作为下一次选择的集合；若不满足规定的性质（失败的反馈），根据失败的反馈信息更新概率模型。该过程直到模型收敛停止。

为了达到上述目的，概率模型需要提供指定待选择子集，以及根据历史反馈结果计算后验概率的方式。在该框架中，本文定义了从待选择子集中获得的期望收益，在每一次迭代过程中，概率模型选择具有最高期望收益的子集。此外，本文讨论了在该框架中基于历史反馈结果计算后验概率并更新模型的方法。本文从理论上分析了最小子集的存在性，以及该框架下产生结果的正确性（满足规定性质）。

该框架既利用了反馈函数通过的反馈信息，也利用了反馈函数失败的反馈信息，并将这些反馈信息用于指导差异调试过程。然而，针对实际问题，该框架需要结合具体的概率模型才能应用。本文针对差异调试中两种常见场景，分别设计了两个概率模型，即面向集合的概率模型和面向语法树的概率模型。

1.2.2 面向集合的概率模型

概率化差异调试框架需要针对问题设计概率模型才能应用。大量差异调试应用场景中，输入可以看作是一个元素集合。为将框架应用到这样的场景上，本文提出了面向集合的概率模型。为验证该模型的有效性，本文将该模型与框架结合形成面向集合的概率化差异调试技术进行实验验证，验证结果表明，面向集合的概率化差异调试技术显著提高了差异调试的效果和效率。进一步地讨论了面向集合的概率模型保证结果极小性或最小性的情形和最坏情况下调用反馈函数的次数。

结合概率化差异调试框架，给定面向集合的概率模型，在差异调试迭代中充分利用产生的反馈信息直到差异调试。此外，该模型提出了一种模型推断的近似算法，能够高效地根据反馈信息计算后验概率。在最先进的针对 C 语言和树结构的差异调试技术上进行验证，验证结果表明，给定面向集合的概率模型，概率化差异调试技术提升了差异调试的效率和效果。

1.2.3 面向语法树的概率模型

简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。为了在差异调试中充分考虑语法结构上的依赖关系，本文提出了面向语法树的概率模型。为验证该模型的有效性，本文将该模型与框架结合形成面向语法树的概率化差异调试技术进行实验验证，验证结果表明，面向语法树的概率化差异调试技术显著提高了差异调试的效果和效率。

结合概率化差异调试框架，给定面向语法树的概率模型，考虑了输入集合中元素间关系，在差异调试过程中能够更加准确的估计每个元素在最终结果里的概率。此外，针对树结构的概率模型，提出了一种近似的模型推断算法，能够高效地根据反馈信息计算树结构概率模型中的后验概率。在最先进的针对上下文无关文法的差异调试技术

上进行验证，验证结果表明，给定面向语法树的概率模型，概率化差异调试技术提升了针对上下文无关文法的差异调试的效率和效果。

最后，为了进一步提高概率化差异调试技术的效果和效率，本文通过精化反馈函数提供的反馈信息，提出基于细粒度反馈的概率化差异调试技术，探究了利用细粒度反馈能否为概率化差异调试技术带来效率和效果的提升。

1.2.4　基于细粒度反馈的概率化差异调试技术

现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。基于这个观察，本文提出了基于细粒度反馈的概率化差异调试技术，支持在不修改概率模型的情况下引入细粒度反馈。进一步地，根据反馈函数中子函数的组织特点，基于该技术提出了两种变体技术。通过实验验证，证实了利用细粒度反馈信息确实能带来概率化差异调试技术性能的提升。

综上，本文提出了一系列技术，形成了一套概率化差异调试技术。该系列技术在公认的差异调试数据集和本文收集的编译器调试数据集上超过了最优技术。本文提出的技术已经成为后续工作的基础，已经有多个第三方团队在本文技术和方法的基础上构建了更先进的领域特定的差异调试技术。

1.3　论文的组织结构

本文的结构划分为五个主要部分，具体如下：

- **第一章 引言**。本章主要包括问题的提出，并介绍本文的研究目标和主要创新点。
- **第二章 相关研究现状**。本章深入探讨了与本研究相关的已有研究和文献。
- **第三章 概率化差异调试框架**。在这一章中，提出了概率化差异调试框架。形成了一种概率化方法，并在差异调试迭代过程中，充分利用了差异调试过程中积累的反馈信息。本章从理论上讨论了最小子集的存在性，以及概率化差异调试框架产生结果的正确性 (满足规定性质)。
- **第四章 面向集合的概率模型**。在这一章中，针对集合作为大部分差异调试的输入的场景，提出了面向集合的概率模型，并从理论上讨论了返回结果具有极小性或最小性的情形，以及最坏情况下的调用反馈函数的次数。
- **第五章 面向语法树的概率模型**。在这一章中，针对程序作为差异调试的输入的场景，为了在差异调试中充分考虑语法结构上的依赖关系，提出了面向语法树的概率模型，并从理论上讨论了最坏情况下调用反馈函数的次数。
- **第六章 基于细粒度反馈的概率化差异调试技术**。在这一章中，观察到构成反馈

函数的子函数可以提供细粒度的反馈，提出了基于细粒度反馈的概率化差异调试技术及其变体技术，进一步改进了概率化差异调试技术，同时验证了精化反馈函数的反馈信息对于提升差异调试技术性能的有效性。

- **第七章 结论及展望**。最后一章对本文的研究工作进行总结，并展望未来的研究方向。

第二章　相关研究现状

本文所研究的问题属于差异调试问题。开发人员通常利用差异调试技术自动化地简化揭错的测试用例，以提高开发人员理解缺陷、定位缺陷的效率。此外，根据多篇相关领域综述[15-17]，差异调试属于解决缺陷定位的重要手段之一。本章首先介绍缺陷定位的相关技术，然后介绍差异调试技术发展历程以及重要的差异调试技术。

2.1　缺陷定位技术

软件缺陷在软件开发过程中不可避免。为了提高程序的质量，开发人员需要在不引入新的缺陷的同时，尽可能多地消除程序中的缺陷。在程序调试过程中，缺陷定位是确定程序缺陷确切位置的重要过程，需要消耗大量时间和计算资源。其有效性取决于开发人员对正在调试的程序的理解、逻辑判断能力、过往程序调试经验，以及检查可疑代码的顺序等因素。

传统缺陷定位需要消耗大量人力，尽管逐渐诞生能够帮助开发人员进行调试的工具，开发人员仍然需要消耗大量精力。当程序执行出现异常行为时，一种直观的缺陷定位的方法是分析程序的内存状态。另一种方法是在可疑代码附近插入打印语句，以打印出一些变量的值。尽管前者现在很少使用，因为它可能需要对大量数据进行分析，但后者仍然被使用。然而，用户需要对程序在引起问题的给定测试用例下的执行过程有很好的理解，然后在适当的位置插入必要的（既不过多也不过少）打印语句。因此，这也不是一种理想的调试技术来确定缺陷的位置。为了解决上述问题，历史上诞生了诸如 DBX 和 Microsoft VC++ 调试器等调试工具。它们允许用户在程序执行过程中设置断点，并检查每个断点处的变量值和内存状态，可以将这种方法视为将打印语句插入程序中，但不需要实际插入任何打印语句。断点可以在执行之前预先设置，也可以由用户在程序执行过程中以交互方式动态设置。尽管调试工具的诞生一定程度上缓解了开发人员的调试压力，这种方法的一个主要缺点是需要用户预先设置好自定义的断点策略，并在调试过程中根据实际情况不断修正或反复运行程序获取新的调试信息。此外，这种方式不能根据给定执行路径上包含故障的可能性对代码进行优先排序来减少搜索缺陷的范围。

缺陷定位可以分为两个主要阶段。第一阶段是使用一种方法来识别可能包含程序缺陷的可疑代码；第二阶段是由开发人员检查已识别的代码，以确定它是否确实包含缺陷代码。以下介绍的经典缺陷定位技术大多侧重于第一阶段。

2.1.1 静态、动态和执行切片技术

程序切片是一种常用的调试技术[18-20]。给定一个变量和一个语句，静态程序切片[21]包含了所有可能影响该语句处变量值的语句。通过切片减少调试过程中的搜索空间的思想是基于以下观点：如果一个测试用例由于语句处的错误变量值而失败，那么缺陷应该在与该变量相关语句对应的静态切片中找到。因此，可以将搜索范围限制在切片上，而不是搜索整个程序。Lyle 和 Weiser 通过构建一个程序切片的差集（两组静态切片的差集）来扩展上述方法，以进一步减少可能故障位置的搜索范围[22]。这种方法的一个缺点是可能会生成包含某些不应包含的语句的切片。这是因为无法通过静态分析预测某些运行时值。为了从切片中排除这些额外的语句，需要使用动态切片[23, 24] 的技术，例如研究[25-31] 用于解决上述问题。

另一种技术是使用基于数据流测试的执行切片和切块来定位程序缺陷[32]，该缺陷存在于包含测试用例执行语句集合的给定测试用例的执行切片中。选择执行切片而不是静态或动态切片的原因是，静态切片侧重于找到可能对任何输入的感兴趣的变量产生影响的语句，而不是确实对特定输入影响这些变量的语句。这意味着静态切片不利用揭示缺陷的输入值。这明显违背了调试中的一个非常重要的直觉，即开发人员应该在失败的测试用例下分析程序行为，而不是在通用的测试用例下分析。使用动态切片的缺点是，即使已经提出了不同的技术[33-39] 来解决这些问题，收集它们可能会消耗过多的时间和空间。另一方面，如果知道测试的覆盖范围，那么对于给定的测试用例，可以非常容易地构建执行切片，因为可以通过将测试期间收集的覆盖数据转换为相应的执行切片，即不报告覆盖百分比，而是报告哪些语句被覆盖。

任何基于切片的技术都存在一个问题，即缺陷可能不在切片中。即使缺陷确实在切片中，仍然可能有太多需要检查的可疑代码。为了解决这些问题，Wong 等人提出了一种基于块间数据依赖的增强和细化方法[40]。根据当前正在检查的代码与其块间数据依赖关系，在搜索空间中包含额外找到的相关代码；通过使用额外成功测试的执行切片，从搜索空间中排除可疑程度较低的代码。

2.1.2 基于程序频谱的方法

程序频谱记录了程序的执行信息，例如条件分支的执行信息或无循环的内部过程路径的执行信息[41]。它可以用来跟踪程序的行为[42]。当执行失败时，这些信息可以用来识别导致失败的可疑代码。早期的研究[27, 43-45] 只使用失败的测试用例进行缺陷定位，尽管后来发现这种方法效果不佳[32, 46-48]。后续研究[49-52] 通过同时使用成功和失败的测试用例，并在它们之间进行对比，取得了更好的结果。

可执行语句命中频谱（Executable Statement Hit Spectrum，简称 ESHS）记录了哪

些可执行语句被执行。研究[53] 中提出了两种基于 ESHS 的缺陷定位技术，即集合并和集合交。集合并计算了一个失败测试的程序频谱与一组成功测试的程序频谱并集之间的差异。它关注的是由失败测试执行但没有任何成功测试执行的代码。这样的代码比其它代码更可疑。相对应地，集合交的技术排除了所有成功测试执行但失败测试未执行的代码。该研究也提出了一种基于程序频谱的技术，即最近邻技术。该技术将一个失败测试与一个与之最相似的成功测试进行对比，根据它们之间的“距离”来衡量相似度。在该技术中，测试的执行被表示为按执行次数排序的语句序列。如果一个缺陷在差异集中，它就被定位到了。对于不包含在差异集中的缺陷，该方法可以通过首先构建程序依赖图，然后逐步包含和检查图中未检查的相邻节点，直到检查完所有节点来进行缺陷定位。

另一种基于 ESHS 的缺陷定位技术是 Tarantula[46]，它使用覆盖率和执行结果来计算每个语句的可疑度，公式为 X / (X + Y)，其中 X =（执行该语句的失败测试数）/（失败测试总数），Y =（执行该语句的成功测试数）/（成功测试总数）。Tarantula 的一个问题是它不能区分一个失败测试用例与另一个失败测试用例的对于定位某个缺陷的贡献，或者一个成功测试用例与另一个成功测试用例的相关贡献。

Guo 等人试图回答一个问题：在缺陷定位过程中，如果将一个失败的测试用例与一个成功的测试用例进行比较，那么应该选择哪个成功的测试用例进行比较？该研究团队通过提出一种基于控制流的差异度量来解决这个问题，该度量考虑了两个测试用例执行中语句执行的顺序，而不仅仅是语句执行的集合[54]。给定一个失败的测试用例和一组成功的测试用例，基于差异度量选择执行序列与失败的测试用例最接近的成功测试。然后，通过返回失败测试和成功测试之间的序列差异生成一个缺陷报告。Wong 等人提出了一种更灵活的技术[40]，通过识别执行片段与失败测试尽可能相似的成功测试，以便尽可能过滤掉更多的可疑代码。

2.1.3　基于统计的技术

Liblit 等人提出了一种统计调试算法（称为 Liblit05），可以在程序的特定点上使用插桩谓词来隔离缺陷[55, 56]。这些插桩谓词生成反馈报告。对于每个谓词 P，算法首先计算 $Failure(P)$，即 P 为真意味着失败的概率，以及 $Context(P)$，即执行 P 意味着失败的概率。满足 $Failure(P) - Context(P) \leq 0$ 的谓词被丢弃。剩下的谓词根据它们的“重要性”得分进行优先排序，这给出了谓词与程序缺陷之间的关系。得分较高的谓词应该首先检查，以帮助开发人员找到错误。一旦找到并修复了一个缺陷，与该缺陷相关的反馈报告将被删除。这个过程持续进行直到所有的反馈报告被删除或所有的谓词被检查完毕。

Liu 等人提出了 SOBER 模型来对可疑谓词进行排序[57]。在一次运行中，谓词 P 可能被评估为真超过一次。计算 $\pi(P)$，即 P 在每次运行中被评估为真的概率，公式为 $\pi(P) = \frac{n(t)}{n(t)+n(f)}$，其中 $n(t)$ 是 P 在特定运行中被评估为真的次数，$n(f)$ 是 P 被评估为假的次数。如果 $\pi(P)$ 在失败的运行中的分布与成功的运行中的分布显著不同，则 P 与一个缺陷相关。

Wong 等人提出了一种基于交叉表（也称为交叉分类表）分析的技术（称为 Crosstab）来计算每个可执行语句的可疑性，即包含程序缺陷的可能性[58]。对于每个语句，构建一个交叉表，其中有两个列向分类变量“覆盖”和“未覆盖”，以及两个行向分类变量“成功执行”和“失败执行”。使用假设检验来提供执行结果与每个语句的覆盖之间的“依赖性/独立性”的参考。然而，每个语句的真实可疑性取决于其覆盖（覆盖它的测试数量）与执行结果（成功/失败执行）之间的关联程度（而不是假设检验的结果）。Yang 等人提出了一种基于数据流的缺陷定位技术[59]，结合统计信息重点考虑变量状态和数据流提升缺陷定位的效果。

简单来说，基于频谱和统计的技术均是使用某些特定的打分机制，对每个可疑语句打分，然后开发人员可以根据评分排序，依次检查可以语句。这样的技术依赖于特定的场景，很难泛化到更为通用的应用场景中。

2.1.4 基于程序状态的技术

程序状态由变量及其在执行过程中的特定取值组成。它可以作为定位程序缺陷的良好指标。在缺陷定位中使用程序状态的一般方法是修改一些变量的值，以确定哪个变量是导致程序执行错误的原因。

Zeller 等人提出了一种基于程序状态的调试方法，称为差异调试[2, 4]，通过对比成功测试和失败测试的内存图[60] 之间的程序状态，将缺陷的原因缩小到一小组变量。通过将变量的值从成功测试替换为失败测试中相同点的对应值，并重复程序执行来测试可疑性。除非观察到相同的缺陷，否则不再考虑该变量的可疑性。

Cleve 和 Zeller 将差异调试扩展为原因转换技术 (cause transition method)[11]，以识别缺陷原因从一个变量转换到另一个变量的位置和时间。该研究团队提出了一个名为 cts 的算法，用于快速定位程序执行中的缺陷原因转换。原因转换技术的一个潜在问题是执行成本相对较高；程序执行中可能存在数千个状态，并且在每个匹配点上进行差异调试需要额外的测试运行来缩小原因范围。另一个问题是确定的位置可能不是缺陷所在的地方。为了解决这个问题，Gupta 等人[61] 引入了缺陷诱导切片 (failure-inducing chop) 的概念作为原因转换技术的扩展。首先，使用差异调试来识别导致失败的输入和输出变量。然后，为这些变量计算动态切片，并将输入变量的前向切片和输出变量的

后向切片的交集中的代码视为可疑代码。

Zhang 等人提出的谓词切换 (predicate switching) 技术[62]，其中程序状态被改变以强制改变失败执行中的执行分支。一个能够使程序成功执行的谓词被标记为关键谓词。该技术首先找到变量中的第一个错误值。可以应用不同的搜索策略，如最后执行的先切换排序和基于优先级的排序，来确定关键谓词的下一个候选项。

Wang 和 Roychoudhury[63] 提出了一种自动分析失败测试的执行路径并改变该路径中分支的结果以产生成功测试执行的技术。成功通过测试并发生更改的分支语句被记录为可疑语句。

除上述技术外，还有很多基于机器学习的缺陷定位技术[64-73]，然而，随着深度学习大模型的发展，模型的表示能力逐渐提升[74-77]，越来越多的超越传统技术的基于深度学习大模型的缺陷定位技术被提出[78-81]。

传统的缺陷定位技术较为依赖开发人员针对应用场景设计合适的策略，而基于深度学习大模型的缺陷定位技术虽然在验证数据集上表现了出众的效果，实际应用中仍然存在提升的空间。可见，即使存在这么多不同的缺陷定位技术，针对缺陷定位的研究仍然远未完善。虽然这些技术不断发展，但软件也变得越来越复杂，这意味着缺陷定位所面临的挑战也在增加。因此，仍然有大量的研究需要进行，还有许多突破有待实现。本文重点关注差异调试技术。与大多数缺陷定位技术不同，差异调试技术关注简化揭错的测试用例、帮助开发人员理解缺陷发生的原因，而大多数缺陷定位技术关注被测试程序本身。在软件调试中，开发人员在使用差异调试时，不需要关注被测试软件的特征，只需要设计一个用于检查是否复现缺陷的反馈函数。差异调试技术能够自动化地简化缺陷，帮助开发人员理解缺陷发生的原因，提高调试的效率。此外，差异调试技术还可应用于缓解软件膨胀、回归故障定位、测试集合约简等领域。

2.2　差异调试技术

差异调试是基本的自动化调试手段之一，其效率和效果是长期制约差异调试应用范围的首要因素。二者的提升也非常困难：虽然涉及差异调试的论文已达数千篇，约 20 年前提出的 ddmin 算法仍然是几乎所有现代差异调试算法的核心。

差异调试的发展历程可以概括为以下三个主要阶段。首先是对 ddmin 算法的优化和改进，这一阶段着重于提升已有算法（ddmin 算法）的效果和效率。其次，是在 ddmin 算法基础上构建领域特定的差异调试技术，以满足不同应用领域的需求。最后，是以结合历史数据为代表的概率化差异调试技术的研究，通过分析以往的调试经验来改进差异调试技术。

总的来说，这三个发展阶段相互交织，各自都有着重大的研究突破。然而，尽管

取得了显著进展，差异调试技术的发展潜力仍然巨大，其应用前景也是无限的。本章将从介绍 ddmin 算法开始，逐步展示各个阶段中具有重大意义的研究成果，为读者对差异调试领域的发展提供更深入的了解。

作为差异调试的基础算法，ddmin 是由 Zeller 和 Hildebrandt 提出的，用于简化导致失败的测试输入[4]。算法1展示了 ddmin 算法的伪代码（可能有多种实现方式，此处仅展示其中一种实现方式）。ddmin 算法将输入的元素集合视为一个集合，并通过两层嵌套的循环进行处理。外层循环用于减少表示要考虑的子集合长度的变量，n。n 从 2 开始（将集合均分为两个子集合），并在每次迭代时加倍，直到达到当前集合长度为止。内层循环首先测试这些子集合的补集, 然后将反馈函数用于所有连续和不相交的子集合。如果任何反馈函数返回通过，则只保留这个子集合。在调用反馈函数 ϕ 之前，如果一个子集合或其补集之前已经被反馈函数验证过，则跳过它（简略起见没有在伪代码中体现）。

图2.1所示为一段可能存在缺陷的函数 `foo`，利用字符串随机生成工具，开发人员找到了包含子串 `V"/+!aF-(V4EOz*+s/Q,7)2@0_` 的由 4,100 个字符组成的字符串导致

算法 1: ddmin 算法的伪代码

输入: 输入集合 X，反馈函数 ϕ
输出: 约减之后的集合 X

```
1  assert(φ(X)) ≠ F ;
2  // 初始化分割粒度（均分）;
3  n ← 2 ;
4  // 外层循环用于控制分割粒度 ;
5  while |X| >= 2 do
6      start ← 0 ;
7      subset_length ← int(|X|/n) ;
8      some_complement_is_passing ← False ;
9      // 内层循环用于连续且不相交的子集及其补集 ;
10     while start < |X| do
11         complement ← (X[: int(start)] + X[int(start + subset_length) :] ;
12         if φ(complement) == T then
13             X ← complement ;
14             n ← max(n − 1, 2) ;
15             some_complement_is_passing ← True ;
16             break ;
17         start ← start + subset_length ;
18         if not some_complement_is_passing then
19             if n == |X| then
20                 break;
21             n ← min(n ∗ 2, |X|) ;
22     end
23 end
24 return X;
```

```python
def foo(inp: str) -> None:
    x = inp.find(chr(0o17 + 0o31))
    y = inp.find(chr(0o27 + 0o22))
    if x >= 0 and y >= 0 and x < y:
        raise ValueError("Invalid input")
    else:
        pass
```

图 2.1　一个可能存在缺陷的函数

函数 `foo` 出现异常（简略起见，以该子串作为揭露异常的测试输入）。有了 ddmin 算法，开发人员可以利用它自动化地缩减揭露异常的输入串。

在应用 ddmin 算法之前，开发人员需要定义反馈函数 ϕ，用于检测当前的输入是否仍满足所规定的性质。在本例中，反馈函数 ϕ 被定义为，如果输入集合能够揭露异常 `ValueError("Invalid input")` 则返回通过（T），否则返回失败（F）。图2.2展示了 ddmin 算法在处理本例的主要流程（省略了部分）。根据该结果，可以知道只需要输入中包含两个匹配的括号（字符串 `()`）就足够触发 `foo` 异常。实际上，ddmin 算法在 25 步内确定了这一点，并返回一个极小的结果（1-极小），即从返回的结果中删除任意一个字符，都不会导致反馈函数失败。类似上面的简化测试用例具有许多优点：

```
mystery('V"/+!aF-(V4EOz*+s/Q,7)2@0_'): T (ValueError: Invalid input)
mystery('z*+s/Q,7)2@0_'): F
mystery('V"/+!aF-(V4EO'): F
mystery('F-(V4EOz*+s/Q,7)2@0_'): T (ValueError: Invalid input)
mystery('Oz*+s/Q,7)2@0_'): F
mystery('F-(V4EQ,7)2@0_'): T (ValueError: Invalid input)
mystery(',7)2@0_'): F
mystery('F-(V4EQ'): F
mystery('V4EQ,7)2@0_'): F
mystery('F-(Q,7)2@0_'): T (ValueError: Invalid input)
mystery('Q,7)2@0_'): F
mystery('F-()2@0_'): T (ValueError: Invalid input)
mystery('2@0_'): F
mystery('F-()'): T (ValueError: Invalid input)
mystery('()'): T (ValueError: Invalid input)
mystery(')'): F
mystery('('): F
'()'
```

图 2.2　ddmin 算法处理步骤示例

- 简化的测试用例降低了开发人员调试的负担。测试用例更短且突出重点，因此不会让开发人员陷入与缺陷无关的细节上的困扰。简化的输入通常也具有更短的执行时间和更小规模的程序状态的特点，这两者都会减少开发人员在理解缺陷时的思考成本。在本例中，消除了大量的无关输入，最终的返回结果仅包含两个字符。
- 简化的测试用例更容易记录或传达。在本例中，只需要记录“`foo` 函数在 `()` 上失败”即可，这比“`foo` 函数在一个 4,100 个字符的输入字符串上失败（附上该字符串）”要简略得多。
- 简化的测试输入有助于识别重复的问题。如果已经报告了类似的错误，并且所有这些错误都被简化为相同的原因（即输入包含一对匹配的括号），那么很明显，所有这些错误都是相同根本原因的不同症状，并且可以通过开发人员一次代码维护来同时解决所有这些错误。

差异调试技术 ddmin 算法在最佳情况下，反馈函数被调用的次数与输入长度的对数成正比，即 $O(\log_2 n)$（每当均分输入时反馈函数可以通过）；在最坏情况下，ddmin 算法的反馈函数调用次数为 $O(n^2)$（不得不反复尝试删除单独的字符）。

上述示例将差异调试技术 ddmin 应用于简化揭错的测试用例中，随着差异调试技术相关研究的深入的发展，差异调试技术被应用于越来越多的领域中。Zhou 等人将差异调试技术应用于调试微服务系统中有关于运行时系统、通信和协调机制的相关错误[82]。基于该项研究，Zhou 等人利用并行策略优化了应用于微服务系统调试的差异调试技术，使得调试效率进一步提升[83]。Pei 等人将差异调试技术应用于解决随机测试用例生成工具（Fuzzer）中模糊驯服问题（fuzzer taming problem），即通过对随机测试用例生成工具生成的测试用例进行排序以达到排名在前的测试用例尽早发现不同种类的缺陷的目的[84]。Rabin 等人通过利用差异调试技术缩减智能代码（Code Intelligence, CI）系统中的输入程序，来识别 CI 系统中模型的关键输入特征[85]。Tazl 等人将差异调试技术应用于解决超约束问题（over-constrained problem），通过约简指数数量级别的输入冲突，求解最小冲突，达到加速约束编程（constraint programming, CP）、布尔可满足性问题（Boolean Satisfiability Problem, SAT）和描述逻辑（Description Logic, DL）求解器等领域中基本算法的目的[86]。Brummayer 等人结合差异调试，提出新颖的模糊测试技术对 SMT 求解器进行测试，发现了若干 SMT 求解器中的缺陷[87]。

2.2.1 针对 ddmin 算法的改进技术

ddmin 算法被最早提出用于解决差异调试问题，随后部分研究团队基于 ddmin 算法提出了一系列改进技术，然而，大部分工作没有解决 ddmin 算法的本质缺陷，即固

定化地尝试输入集合中元素的删除。

伯克利大学的相关研究人员在 ddmin 算法提出后，实现了用于实验室研究的工具 Berkeley Delta[88]，该工具在差异调试技术研究的初期为科研人员提供了强有力的支撑。由于 ddmin 算法的输入必须满足特定性质，在其迭代过程中的目标是不断找到更小的满足同样性质的结果。然而，在真实调试场景中，这样的性质通常被其它程序所掩盖，无法找到满足 ddmin 算法要求的输入。Artho 等人通过在其它掩盖了这些性质的程序上迭代地执行 ddmin 算法，最终能够找到满足 ddmin 能够处理的输入，协助开发人员进行高效的调试工作[89]。ddmin 算法因其通用性和有效性被广泛使用，Vince 等人重点关注固定点迭代（fix-point）的 ddmin 算法[90]。通过精心设计迭代步骤，固定点迭代的 ddmin 算法能够返回比 ddmin 算法更小的局部最优解。在差异调试过程中，在 ddmin 算法迭代过程中，同一个缩减后测试用例可能会在迭代的不同阶段被多次选择，重复调用反馈函数会导致算法时间开销的增加。尽管目前已经有了比较成熟的缓存机制和缓存替换算法，如最不常用（LFU）、最常用（MFU）和最近最少使用（LRU）等，但这些经典的算法没有利用 ddmin 算法的特性，导致从缓存中替换后在后续迭代过程中会出现更多的不必要的替换步骤，Vince 等人通过利用 ddmin 等差异调试算法的特性来减少缓存占用，降低了差异调试技术的缓存需求[91]。针对其它的差异调试场景，Tian 等人[92] 和 Hodovan 等人[93] 也进行了相关的优化研究。考虑到 ddmin 算法提出的背景，当时并行化技术并没有成熟且硬件设备条件有限。Hodovan 等人针对 ddmin 算法研究出了一套并行的 ddmin 算法 Picire[94]，在提高效率的同时保证了返回结果的 1-最小性，即从返回结果中删除任意一个字符都无法满足所规定的性质。由于这套并行算法仅针对 ddmin 算法量身定制，在后来的差异调试技术发展历程中并没有被广泛采纳，不过这项研究工作也能够带来提升差异调试技术效率方面的启发。

2.2.2　领域特定的差异调试技术

为应对软件开发技术高速发展、软件迭代速度不断加快、新兴程序语言不断涌现的局面，现代化的差异调试技术主要为具体领域设计专门的算法，这类技术通常被统称为领域特定的差异调试技术。例如，针对 C 语言的差异调试技术[13, 95]，针对上下文无关文法的差异调试技术[96] 等。考虑到有结构的输入，如树结构等，ddmin 算法在迭代过程中总会产生错误的输入，导致时间开销增大。针对树结构的输入，Misherghi 等人提出了一种简单而有效的算法 HDD[14]。

算法2展示了 HDD 算法流程。HDD 算法接受一个树结构的输入 $input\_tree$ 和反馈函数 ϕ，并返回简化之后的树结构。对于树结构中的每一层节点组成的集合，HDD 使用 ddmin 算法对该集合进行简化，得到满足条件的子集合，并通过 $\mathbb{PRUNE}$ 函数从树

上删除相应的节点，得到新的简化之后的树结构。该过程直到处理完树中最后一层的节点集合终止。在该研究中，HDD 作者进而提出固定点迭代的 HDD 算法，即不断调用 HDD 处理输入树结构，直到某一次调用中无法删除节点终止。

在 HDD 算法被提出的后续若干年内，出现了许多针对 HDD 算法的改进。Hodovan 等人针对 HDD 算法，针对上下文无关文法解析的抽象语法树，从实现上提出了现代化 HDD（modernizing HDD）算法[97]。图2.3展示了现代化 HDD 算法的框架，其算法内部包括命令行接口模块、基于 ANTLRv4[98] 的词法和语法解析模块、基于 ANTLRv4 的输入程序解析模块、原生 HDD 算法和并行化的 ddmin 算法 Picire。基于此，现代化 HDD 算法接收 ANTLRv4 文法、传统的语法和词法解析、孤岛文法描述、输入程序和反馈函数作为输入，经过处理后返回简化后的程序。其中，由于像 HTML 格式的输入不能仅通过一个文法输入进行描述，孤岛文法描述是一种针对输入的补充文法。相比于原生 HDD 算法，在处理程序等上下文无关语言的输入时，由于采用了先进的解析格式和解析方法，现代化 HDD 可以更加高效地返回更小规模的简化程序。然而，HDD 算法在执行过程中调用 ddmin 算法处理语法树上每一层节点构成的集合，没能从根本上解决差异调试的问题。

Hodovan 等人提出支持过滤的（Coarsen）HDD 算法[99]，通过修改算法2中提取 ddmin 算法处理节点集合的 TAGNODES 函数，支持过滤的 HDD 算法可以作为 HDD 算法的预处理阶段技术，通过根据树结构中节点类型设计的过滤原则，将过滤后的子集传递给 HDD 算法中调用的 ddmin 算法进行处理，增大产生合法输入的比例，从而提升 HDD 算法的效率。

与 HDD 算法在每轮迭代时处理树结构同一层节点不同，Kiss 等人提出的 HDDr 算法[100]，在每一轮迭代时处理给定节点的孩子节点。算法2中的 TAGNODES 和 PRUNE 函

算法 2: HDD 算法

输入: 输入树结构 *input_tree*，反馈函数 ϕ
输出: 简化之后的树结构

```
1  level ← 0 ;
2  // TAGNODES 函数用于标记树中 level 层的所有节点并返回;
3  nodes ← TAGNODES(input_tree, level) ;
4  while nodes ≠ ∅ do
5      // ddmin 用于缩减树中每一层节点组成的集合;
6      minconfig ← ddmin(nodes, φ) ;
7      // PRUNE 函数用于从树上删除 minconfig 集合中的节点;
8      PRUNE(input_tree, level, minconfig) ;
9      // 进行树的下一层处理;
10     level ← level + 1 ;
11     nodes ← TAGNODES(input_tree, level) ;
12 end
```

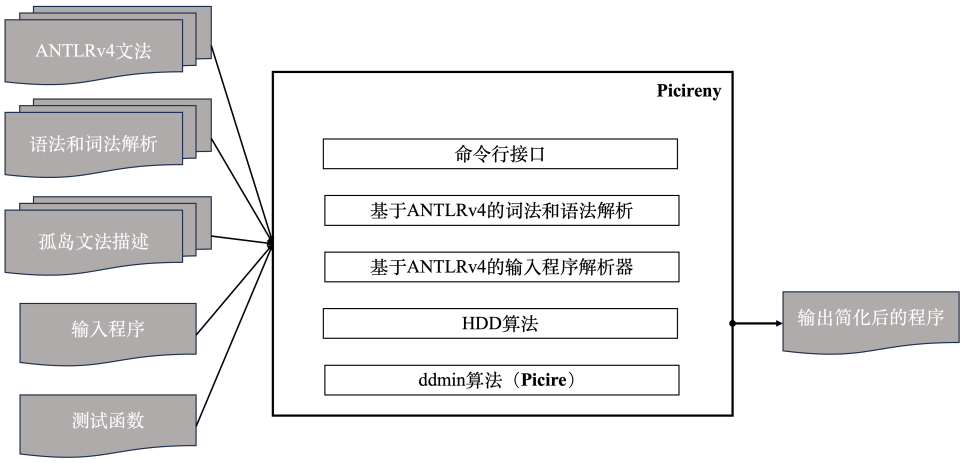


图 2.3　现代化 HDD 算法的框架

算法 3: HDDr 算法

输入: 输入树结构根节点 $root\_node$，反馈函数 ϕ
输出: 简化之后的树结构

```
1 queue ← ⟨root_node⟩ ;
2 while queue ≠ ⟨⟩ do
3     current_node ← POP(queue) ;
4     nodes ← TAGCHILDREN(current_node) ;
5     minconfig ← ddmin(nodes, φ) ;
6     // 调用 ddmin 算法 ;
7     PRUNECHILDREN(current_node, minconfig) ;
8     APPEND(queue, minconfig) ;
9 end
```

数根据处理输入的不同相应地变化为算法3中的 TAGCHILDREN 和 PRUNECHILDREN 函数。此外，由于 HDDr 算法中引入了队列数据结构，算法3中的 POP 和 APPEND 均是对队列的基本操作。从上述技术的算法中可以看出，在执行过程中大多依赖 ddmin 算法处理领域结构（语法树）预处理后给定的集合，没有从本质上解决差异调试的问题。

编译器是将高级编程语言代码转换为机器代码的关键工具，因此对其进行全面的测试至关重要。然而，编译器测试通常涉及大量的测试用例和代码，这使得手动分析和调试非常耗时和复杂。针对 C 语言编译器这样的大型软件，Regehr 等人提出专门用于简化编译器测试程序的差异调试技术 C-Reduce[101]。C-Reduce 技术首次定义了测试用例最小化问题（test-case minimization problem）：令 I 是某个被测试系统（SUT）的所有有效输入的集合。对于 $i \in I$，令 $|i|$ 表示根据适当度量标准来衡量的 i 的大小（字节级别或标记级别或行级别等）。令 $B \in I$ 为出发 SUT 中特定感兴趣故障的输入集合，如在 SUT 中特定位置的崩溃错误等。对于 SUT 中的特定故障，测试用例最小化问题是找

到 $iBmin$，其中 $iBmin \in B$ 且 $\forall i \in B$，$|i| \geq |iBmin|$。需要注意的是，$iBmin$ 可能不是唯一的，可能存在多个符合条件的最小大小的测试用例。然而，为了清晰起见，C-Reduce 技术假设 $iBmin$ 是唯一的。C-Reduce 旨在帮助编译器测试人员更高效地发现和修复编译器中的错误。算法4展示了 C-Reduce 的基本流程。该算法涉及三个主要的内部函数，这些内部函数用于维护和管理输入的转换操作集合 transformation。

具体地，C-Reduce 技术中的转换本质上是一种迭代器，它可以不断地对测试程序进行源代码到源代码级别的转换，该迭代器实现了以下功能：

- new：无参函数，返回一个新的用于表示转换状态的对象；
- transform：输入一个测试程序和一个转换状态，修改输入测试程序的同时返回修改后的结果状态，包括 **OK** 和 **STOP**，分别表示该转换被成功执行和没有新的转换可以被应用；
- advance：输入一个测试程序和转换状态，更新迭代器到下一个转换动作，并返回新的转换状态。

对于输入的转换操作集合 transformations，C-Reduce 作者根据长期编译器调试的经验，总结出了 30 种源-源转换的模板种类，并使用 LLVM[102] 的 Clang[103] 前端实现了这些转换，其中包括但不限于：

- 使用标量替代数组变量等；
- 从指针或数组类型的变量中去掉一层；
- 将函数调用从复杂的表达式中抽离；
- 将多个相同类型的变量合并为一个复合定义；
- 将函数作用域的变量改为全局变量；
- 从函数中移除一个参数及其所有的调用点，同时在函数作用域中添加一个同名且类型相同的变量；
- 移除未使用的函数或变量；
- 为函数、变量或是参数赋予新的、更短的名称；
- 将函数更改为返回 `void` 类型，并删除其中的返回语句；
- 将较小的函数定义为内联函数；
- 执行复制传播；
- 将联合体转换为结构体；

针对编译器调试应用场景，C-Reduce 技术通过预定义转换模板，简化揭错的测试用例。由于定义的转换模板大多涉及字符级别的转换，没有考虑程序本身的结构，因此会在简化过程中产生大量语法错误的程序，导致消耗过长的时间。在实际应用中，虽然 C-Reduce 能够返回足够小的结果，然而，较差的返回结果的可读性和较低的简化效率并

算法 4: C-Reduce 算法

输入: 待简化程序 $original\_test\_case$，反馈函数 ϕ，转换操作集合 transformations
输出: 简化之后的程序

```
1  current ← original_test_case ;
2  while not fixpoint do
3      foreach t ∈ transformations do
4          state ← t :: new() ;
5          while True do
6              variant ← current ;
7              result ← t :: transform(variant, state) ;
8              // 进行程序变换 ;
9              if result ← STOP then
10                 break ;
11             if φ(variant) == T then
12                 current ← variant ;
13             else state = t :: advance(current, state) ;
14             // 切换下一个状态 ;
15         end
16     end
17 end
```

不能帮助开发人员提高调试效率。此外，C-Reduce 技术是针对 C 语言编译器调试场景下专门设计的差异调试工作，其通用性受到了极大的限制。

针对程序的差异调试技术存在严重的性能问题，Sun 等人提出 Perses 技术，一种针对上下文无关文法的差异调试技术[96]。Sun 等人的关键洞察是，HDD、C-Reduce 等针对程序的差异调试技术，直接在树结构或程序文本上执行操作，很容易产生语法不正确的输入。针对该问题，Perses 技术在删除操作的基础上，设计了基于抽象语法树的转换模板。得益于现代化文法和解析工具的帮助，Perses 能够十分方便设计转换模板的结构上进行差异调试，保证产生的输入都是符合文法规则的。Perses 算法的总体流程如图2.4所示，Perses 由文法正则模块、文法转换模块、语法解析模块和程序简化模块组成。处理不同的程序或其它上下文无关语言时，Perses 接收不同种类的文法作为输入之一。此外，待简化程序和反馈函数 ϕ 也作为输入。当 Perses 终止时，简化后的程序作为结果被返回。实际上，得益于现代化的文法和解析工具，Perses 中的文法正则模块和文法转换模块已经被这些工具所囊括，本节不作为重点叙述。具体地，Perses 算法如算法5所示，Perses 不再像 HDD 一样，按照树结构层的顺序进行程序缩减。Perses 在算法迭代过程中，始终维护工作列表 *worklist*，每次从 *worklist* 中取出包含标记数最多的节点，根据取出节点的种类进行相应操作。这里不再像 HDD 一样只进行 ddmin 算法的简单调用，虽然 ddmin 算法的简单调用占据着 Perses 处理节点时的大多数情况，但是 Perses 设计的那些转换模板在快速缩减程序大小上也起到了巨大的作用。

与传统文法不同的是，Perses 利用 ANTLRv4 中设计的文法，包含了很多专为在抽

象语法树级别上操作而设计的节点类型。

- **Kleene Star（*）类型**。该节点类型能够产生 0 次或多次终结符或非终结符。例如，A* 能够产生空串、A、AA 等等。
- **Kleene Plus（+）类型**。该节点类型能够产生 1 次或多次终结符或非终结符。例如，A+ 与 A* 类似，不同的是 A+ 不会产生空串。
- **Optional（?）类型**。该节点类型能够产生 0 次或 1 次终结符或非终结符。例如，A? 可以产生空串或 A。

由于表达的简便性，上述三种类型被大量文法和解析工具广泛应用，如 ANTLR[98] 和 JAVACC[104]。在 Perses 中，上述三种类型的节点被统称为*数量节点*，其它节点被统称为*常规节点*。

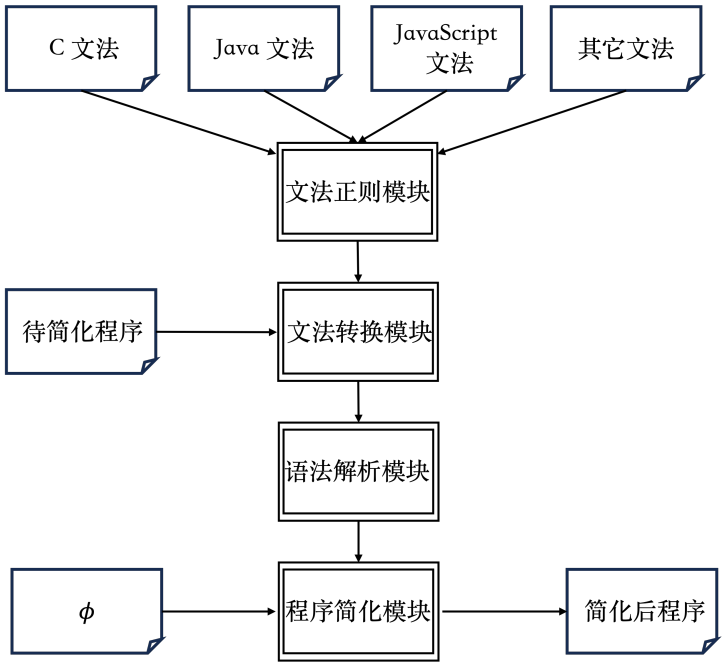


图 2.4　Perses 算法的总体流程

接下来，继续观察算法5。可以发现，Perses 根据当前处理节点类型的不同，调用不同的函数对节点进行处理。具体有以下三个函数，*ReduceStar*、*ReducePlus* 和 *ReduceRegular*。其中，*ReduceStar* 和 *ReducePlus* 直接调用 ddmin 算法对数量节点的子节点组成的集合进行缩减。Perses 在 *ReduceRegular* 函数中使用设计的抽象语法树转换模板，达到对相应子树缩减的目的。简单来说，对于某一个正在被处理的节点来说，从其子树中找寻找能够替代它，同时不违背与父亲连接规则的节点。考虑到时间开销，Perses 没有无限制地寻找可被用于替换的节点。每当处理完当前节点后，Perses 将未能删掉的孩子节点或剩余的子孙节点加入 *worklist* 中，直到 *worklist* 为空时，Perses 算法终止。

图2.5(a)所示为一个待缩减的示例程序，以该程序为例，本节展示 Perses 的算法流

算法 5: Perses 算法

输入: 待简化程序 P，反馈函数 ϕ
输出: 简化之后的程序 p

```
1  best ← ParseTree(P) ;
2  worklist ← {RootNode(best)} ;
3  while |worklist| > 0 do
4      largest ← GetAndRemoveLargestFrom(worklist) ;
5      if largest is Kleene-Star Node then
6          (best, pending) ← ReduceStar(best, φ, largest) ;
7          // 处理 Kleene-Star 节点
8      else if largest is Kleene-Plus Node then
         (best, pending) ← ReducePlus(best, φ, largest) ;
9      // 处理 Kleene-Plus 节点 ;
10     else if largest is Optional Node then (best, pending) ← ReduceStar(best, φ, largest);
11     // 处理 Optional 节点 ;
12     else if largest is Regular Rule Node then
         (best, pending) ← ReduceRegular(best, φ, largest);
13     // 处理 Regular 节点 ;
14     else continue;
15     worklist ← worklist ∪ pending ;
16 end
17 return best ;
```

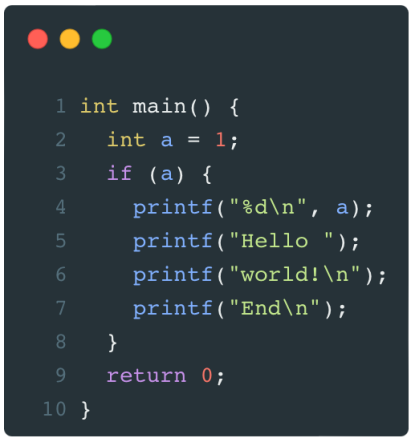
程以及相比于 HDD 和 C-Reduce 等工具的进步。该示例程序的执行结果是打印三行字符串，即“1”，“Hello world!”和“End”。假设规定属性为只打印出“Hello world!”，即 ϕ 函数用于检查程序的输出是否为“Hello world!”，如果是则返回成功，否则返回失败。Perses 的目标是找到能够只打印出“Hello world!”的最小程序。

与 HDD 算法不同的是，Perses 并不是按照语法树层层间的顺序来缩减程序。Perses 维护一个优先队列 Q 保存待处理的节点。每一轮迭代从 Q 中取出包含最多标记数的节点，并根据该节点类型来进行相应的缩减操作。当该节点被处理完后，将剩下的节点加入 Q 后进入下一轮迭代，直到 Q 为空算法终止。

首先，Perses 将示例程序解析成如图2.6(a)所示的语法树。然后在树上依次进行如下操作：

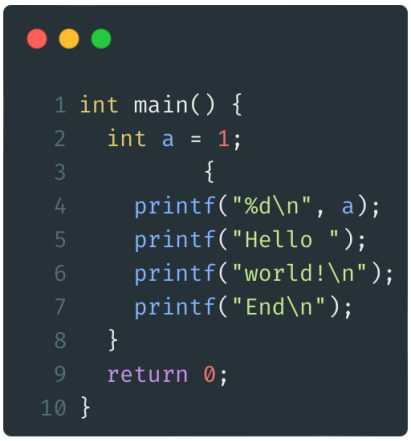
第一步（1. func_def）。Perses 尝试删除语法树的根节点，即 `1.func_def`。但是，删掉它后导致反馈函数返回失败。将其所有孩子节点加入 Q。

第二步（2. compound_stmt）。由于所包含的标记数最多，节点 `2.compound_stmt` 从 Q 中弹出，作为当前处理的结点。对于该函数体节点，由于删掉它会导致语法错误，因此它不能够被直接删除。但是，Perses 从它的子孙节点里找到了和它同类型的节点，用于替换它，即 `6.compound_stmt`。它代表源程序中第三行的语句部分‘if (a)’。在替换操作执行后，产生了新的输入。然而，由于在替换过程中，代表遍历‘a’定义语句的



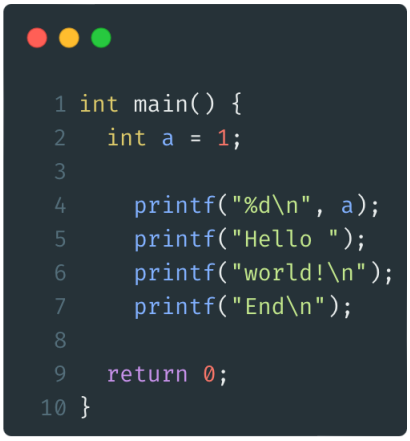
```c
int main() {
  int a = 1;
  if (a) {
    printf("%d\n", a);
    printf("Hello ");
    printf("world!\n");
    printf("End\n");
  }
  return 0;
}
```

(a) 原始示例程序



```c
int main() {
  int a = 1;
        {
    printf("%d\n", a);
    printf("Hello ");
    printf("world!\n");
    printf("End\n");
  }
  return 0;
}
```

(b) Perses 第一次成功后得到的程序

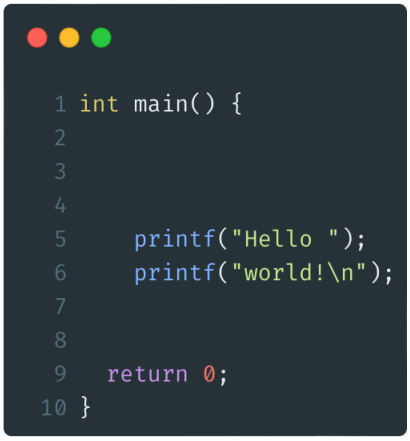


```c
int main() {
  int a = 1;

    printf("%d\n", a);
    printf("Hello ");
    printf("world!\n");
    printf("End\n");

  return 0;
}
```

(c) Perses 第二次成功后得到的程序



```c
int main() {



    printf("Hello ");
    printf("world!\n");


  return 0;
}
```

(d) Perses 最后一次成功后得到的程序

图 2.5　用于演示 Perses 算法的示例程序

节点被删除了，因此产生的程序是不合法的（使用了未定义的变量‘a’）。因此，三个孩子节点被加入队列 Q 中。

第三步（3. stmt_star）。该节点的后缀提示了当前处理节点是一个数量节点，表示其孩子可以在不违反语法的情况下被任意地删除。对于该类型节点，Perses 调用 ddmin 算法删除其孩子节点。不幸的是，ddmin 未能在该步骤删掉任意一个子节点，因此它们都被加入到了队列 Q 中。

第四步（4. if_stmt）。在队列 Q 中包含最多标记数的节点是 `4.if_stmt`。由于它的父亲节点是 `3.stmt_star`，可以在保证语法正确的前提下用任意一条语句替换它。Perses 在它所有孩子节点中搜索，并成功找到节点 `6.compound_stmt` 替换它。因此，Perses 第一次成功缩减了示例程序并得到如图2.5(b)中的程序。

第五步（5. compound_stmt）。为了尝试删除该节点，Perses 尝试使用它其中的子节点替换它。由于它替换了 `4.if_stmt`，它的父亲节点变为 `3.stmt_star`。然后，Perses

使用 `6.stmt_star` 替换它。该过程通过了反馈函数，并得到了如图2.5(c)中的程序。

第六步（6. stmt_star）。由于当前处理节点是数量节点，因此 Perses 使用 ddmin 算法删除其子节点。节点 `8.printf@4` 和 `11.printf@7` 被成功删除。

剩余步骤。在节点 `8.  printf@4` 被删除后，变量的依赖关系也不存在了。当 Perses 后续从 Q 中选择节点 `12.int a = 1;` 进行处理时，可以安全地删除它，并获得更小的程序。

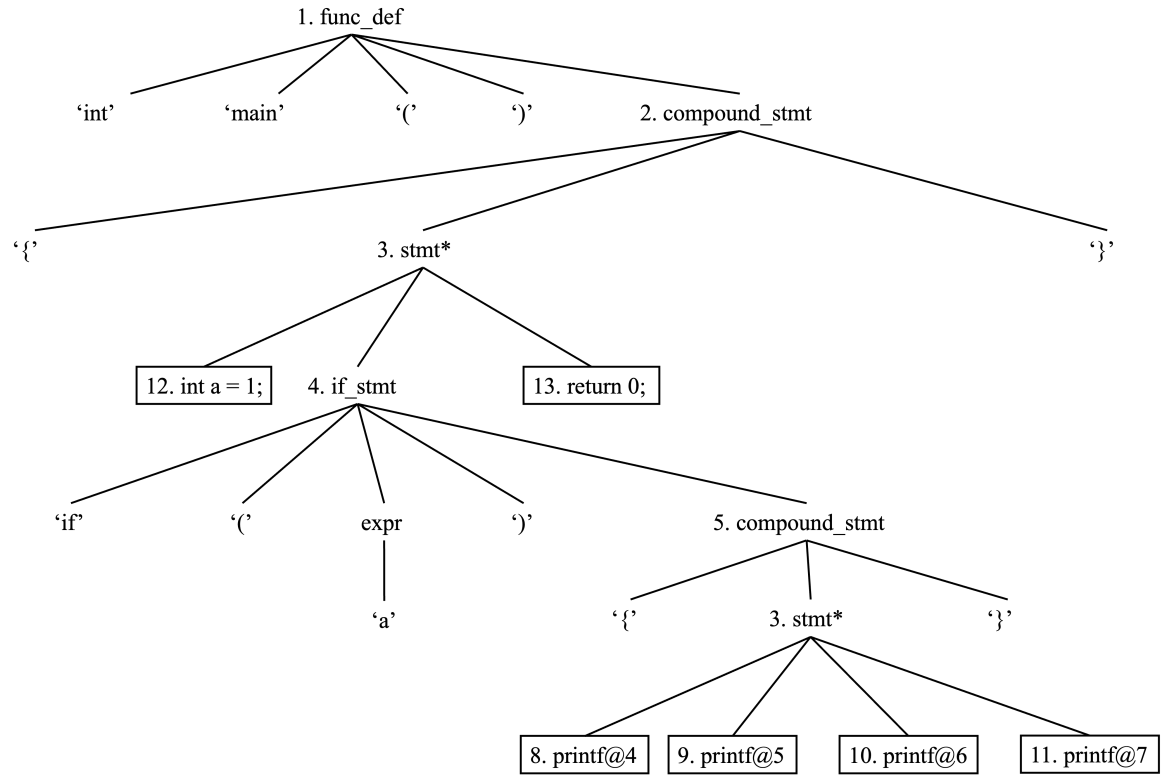
Perses 最终返回如图2.5(d)所示的简化后的程序，图2.6(b)展示了其对应的语法树。通过执行步骤可以看出，HDD 算法或是 ddmin 算法均无法产生 Perses 返回的结果。然而，尽管 Perses 技术提高了针对上下文无关文法的差异调试技术的效率和效果，依然没有从本质上解决差异调试的问题。可以看出，为了达到最小不动点状态，Perses 会被反复调用，在这个过程中，ddmin 算法可能被反复地应用于无法被成功删除的集合，导致时间的浪费。

除了上述列举的诸多领域特定的技术外，差异调试技术被更多的领域应用，并形成了相应的领域特定差异调试技术。Kalhauge 等人观察到差异调试技术在解决具有内部依赖关系的输入时，性能和可扩展性较差，于是提出面向依赖图的差异调试技术[105]。该项技术的实验证明能够达到 12 倍的加速比，并被应用于名为 J-Reduce 的工具中。

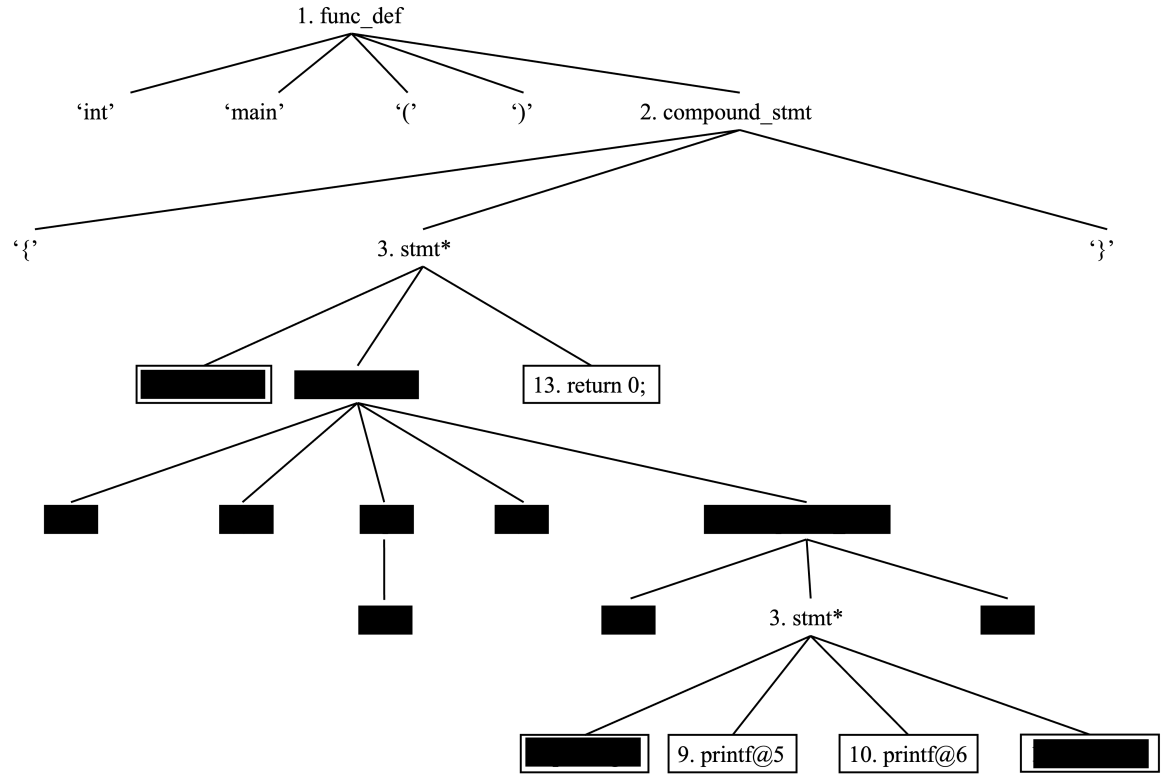
现代软件工程实践越来越多地将冗余代码带入软件产品，这引起了称为软件膨胀的现象，从而导致软件系统的维护，性能和可靠性问题以及安全性问题。随着智能设备的飞速发展和世界的互联，修剪过时的软件以提高互连软件和网络系统的精简性、敏捷性、可靠性、高效性和安全性变得越来越重要。针对已有技术通常不能完全自动化的缺点，Yufei 等人提出一种新的结合静态分析的软件精简技术[106]，该技术可以自动精简 Java 应用程序和 Java 运行时环境中未使用的代码，并基于 Soot 框架实现了一个名为 JRed 的原型工具。Hashim 等人提出一种基于循环展开、循环不变试消除进行程序缩减的技术 Trimmer[107]。Kostas 等人提出了一种新的程序缩减的技术叫做 Program Trimming[108]。其目标是提升安全性检查工具的扩展性和精度。该工作提出了一种针对程序 P 和其缩减版本 P’ 的属性 equi-safe。大量程序分析工具对于程序的执行路径是敏感的，通过使用 Program Trimming 技术，能够提高安全检查工具的效果。此外，该工具实现了一种轻型静态分析工具，作为保证 equi-safe 属性的工具，它扮演一个对程序进行预处理的角色。

2.2.3　概率化差异调试技术

现有的概率化差异调试技术主要针对程序领域。由于面向程序领域的差异调试技术在差异调试过程中往往产生语法错误或语义错误的程序，导致执行时间的严重浪费，



(a) 原始示例程序对应的语法树



(b) Perses 返回结果对应的语法树

图 2.6　用于演示 Perses 算法的示例语法树

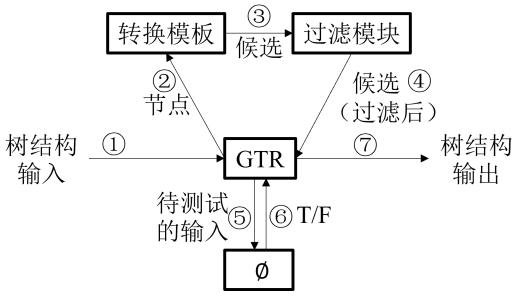


图 2.7　GTR 算法的总体流程

本发展历程中涉及到的极大部分研究工作考虑将机器学习应用于预测产生的程序是否合法，例如 Herfert 等人提出的 GTR 算法[109]、Heo 等人提出的 CHISEL 技术[13] 和 Gharachorlu 等人提出的类型批处理程序简化技术[110]。这类研究工作具有较大的局限性，如 GTR 算法仅针对 Python、XML 等语言训练了机器学习模型，以及类型批处理程序简化技术仅针对 C 语言程序，该技术与文法类型息息相关。此外，DEBOP[111] 是另一类的，基于马尔可夫链蒙特卡洛（MCMC）和 Metropolis-Hastings 采样的概率化方法。然而，DEBOP 是为不同的问题领域设计的：缓解软件膨胀问题，其涉及到的多元目标函数针对该领域特殊设计，并不具有通用性更不具有解决本文研究的含有二元反馈函数的问题的能力。

针对差异调试性能的不足，Herfert 等人提出的 GTR 算法[109]，能够高效地且通用地处理树结构的输入。得益于 GTR 算法中使用的机器学习模型，GTR 能够过滤掉那些具有很大概率出错的变体输入，不调用反馈函数检查它们的性质，从而达到提高效率的目的。另一方面，由于针对树结构，提出了在树上的转换模板，相比于仅有删除操作的 HDD 算法等技术，GTR 算法能够返回足够小的输入，从而提升开发人员的调试效率。图2.7展示了 GTR 算法的总体流程。给定一个输入的树结构（步骤①），GTR 算法通过从树的根节点到最底层节点的遍历，对每个节点应用可行的转换模板，从而缩减树结构中的节点。这些转换模板来自于预先设定好的模板（步骤②），它们能够产生相应的大量候选变体输入（步骤③）。针对其中较大概率出错的变体输入，GTR 算法利用预先准备的数据训练一个机器学习模型用来过滤这些变体输入，从而得到过滤后的候选输入（步骤④）。反馈函数检查候选输入是否满足规定的性质（步骤⑤和⑥）。当不再有新的树结构产生时，GTR 算法终止，并返回简化后的树结构（步骤⑦）。

GTR 算法针对树结构的每一层，依次应用预先设计好的树结构上的转换模板，包括，删除一个子树和用以孩子节点为根的子树替代父节点为根的子树。算法6展示了 GTR 算法的基本步骤。GTR 算法使用辅助函数 APPLYTEMPLATE，从根节点开始对树结构的每一层应用转换模板（算法 1-5 行）。当对某一层的节点应用转换模板时，GTR 算法会将只能应用删除模板和可以应用其它模板的情况区分开。在只能应用删除模板

算法 6: GTR 算法

输入: 待简化树 t，反馈函数 ϕ，模板集合 $\mathcal{L}$
输出: 简化之后的树

```
1  foreach i ∈ [0, depth(t)) do
2      foreach l ∈ L do
3          t ← APPLYTEMPLATE(t, i, φ, l) ;
4      end
5  end
6  Function APPLYTEMPLATE(t, i, φ, l):
7      levelNodes ← level(t, i) ;
8      if l最多返回一个转换 then
9          newNodes ← 应用 ddmin 使用 l 替换 levelNodes ;
10         return newNodes替换 levelNodes 后的树 ;
11     else return REDUCELEVELNODES(t, levelNodes, φ, l) ;
12 End Function
13 Function REDUCELEVELNODES(t, levelNodes, φ, l):
14     // 树 t，层节点结合 levelNodes，反馈函数 φ，模板函数 l;
15     conf ← 空映射;
16     foreach n ∈ nodes do
17         conf.put(n, n) ;
18     end
19     while not improvementFound do
20         improvementFound ← False ;
21         foreach n ∈ nodes do
22             currentRep ← conf.get(n) ;
23             foreach n′ ∈ l(n)其中size(n′) < size(currentRep) do
24                 t′ ← 每个 n 都被 n' 替换后的树 ;
25                 conf.put(n, n′) ;
26                 if φ(t′) then
27                     improvementFound ← True ;
28                     currentRep ← n′ ;
29                 else conf.put(n, currentRep) ;
30                 ;
31             end
32         end
33     end
34     return t'
35 End Function
```

的情况下，GTR 算法调用 ddmin 算法处理由这些节点组成的集合。当模板能够产生不止一个变体输入时，GTR 算法需要选择一个模板去产生某一个具体的变体输入。GTR 算法使用贪心算法搜索目标变体输入。算法 13-35 行列出了该过程的基本步骤。通过遍历节点和可应用的模板，GTR 算法将通过反馈函数且缩减树结构的变体记录在映射 $conf$ 中。当遇到使反馈函数失败的变体时，GTR 算法还原树结构并回到先前的状态。当不再有新的更小的结构被发现时，GTR 算法终止这一过程。

针对差异调试技术在迭代过程中产生大量不合法的程序，如语法错误、语义错误

算法 7: CHISEL 算法

输入: 待简化程序 P
输入: 反馈函数 ϕ
输入: 函数 $\mathcal{L}$
输入: 特征编码函数 F
输出: 简化之后的程序 P'

$L \leftarrow$ P 的每一条语句组成的列表 ;
$n \leftarrow 2$;
$\mathcal{D} \leftarrow \{\langle F(L)\rangle, T\}$;
while $n \leq |L|$ **do**
    $\mathcal{M} \leftarrow \mathcal{L}(\mathcal{D})$;
    建立$\hat{\pi}$;
    $\langle L', n'\rangle \leftarrow \hat{\pi}(\langle L, n\rangle)$;
    // $\Diamond \in \{T, F\}$;
    $\Diamond \leftarrow \phi(L')$;
    if $\Diamond == T$ ***or*** $n' == 2n$ **then**
        // 状态转移 ;
        $\langle L, n\rangle \leftarrow \langle L', n'\rangle$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle F(L'), \Diamond\rangle\}$;
end
return 简化后的语句集合 L 的 P'

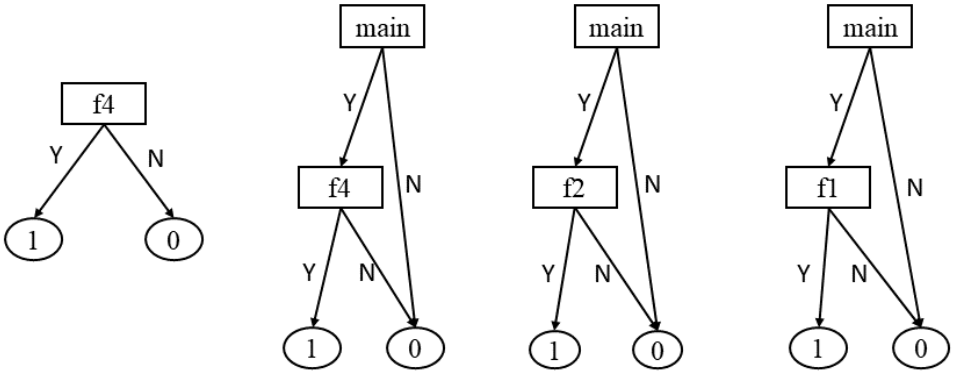
等，Heo 等人提出 CHISEL 技术，一种基于强化学习的差异调试技术，主要用于缓解软件膨胀问题[13]。根据 CHISEL 的实验验证情况，它是针对 C 语言程序的最先进的差异调试技术。由于 CHISEL 是一种基于强化学习的差异调试技术，在介绍算法之前，首先介绍 CHISEL 中定义的状态（State）空间、动作（Action）空间、状态转换（Transition Function）方程、奖励函数（Reward Function）等。CHISEL 以行粒度处理输入程序，将每一行作为列表 L 中的元素。由于 CHISEL 是基于 ddmin 算法设计的强化学习模型，因此其初始状态是 $\langle L, 2\rangle$。动作集合记为 $\mathcal{A}$，状态 s 对应的动作集合 $A(s)$ 可用下面的

```
1 int f1() {return 0;}
2 int f2() {return 1;}
3 int f3() {return 1;}
4 int f4() {return 1;}
5 int f5() {return 1;}
6 int f6() {return 1;}
7 int f7() {return 1;}
8 int main() {return f1();}
```

图 2.8　用于演示 CHISEL 算法的示例程序

	f1	f2	f3	f4	f5	f6	f7	main	√
1	f1	f2	f3	f4	f5	f6	f7	main	×
2	f1	f2	f3	f4	f5	f6	f7	main	×
3	f1	f2	f3	f4	f5	f6	f7	main	×
4	f1	f2	f3	f4	f5	f6	f7	main	√
5	f1	f2	f3	f4	f5	f6	f7	main	√
6	f1	f2	f3	f4	f5	f6	f7	main	×
7	f1	f2	f3	f4	f5	f6	f7	main	√
8	f1	f2	f3	f4	f5	f6	f7	main	√
9	f1	f2	f3	f4	f5	f6	f7	main	×
10	f1	f2	f3	f4	f5	f6	f7	main	×

(a) CHISEL 执行步骤示例



(b) 第 1、2、3、6 步对应的可视化决策树

图 2.9　CHISEL 算法的演示示例

公式计算。

$$A(s) = \{\langle\langle u_1\rangle, 2\rangle, \ldots, \langle\langle u_n\rangle, 2\rangle\} \quad \text{(子集)} \tag{2.1}$$

$$\cup \{\langle L\backslash u_1, n-1\rangle, \ldots, \langle L\backslash u_n, n-1\rangle\} \quad \text{(补集)} \tag{2.2}$$

$$\cup \{\langle L, 2n\rangle\} \quad \text{(更多粒度)} \tag{2.3}$$

其中，$u_1, u_2, \ldots, u_n$ 表示 L 的 n 个划分，就像 ddmin 算法中的那样。状态转换方程 T 被定义如下：

$$T(s'|s,a) = \begin{cases} 1, & (s' = \langle u_i, 2\rangle, \phi(u_i) = T) \\ 1, & (s' = \langle \backslash u_i, n-1\rangle, \phi(L\backslash u_i) = T) \\ 1, & (s' = \langle L, 2n\rangle, \nexists \phi(u_i) = T \vee \phi(L\backslash u_i) = T) \\ 0, & (\text{其它情况}) \end{cases} \tag{2.4}$$

CHISEL 定义反馈函数为二值的，当划分符合 1-最小性时返回 1，其它情况返回 0。然而，上述的定义并不能作为 ddmin 算法的优化，因为在划分 L 的过程中会产生大量不合法的划分。CHISEL 进一步地使用 $\hat{T}$ 和 $\hat{R}$ 来表示近似后的状态转换方程和激励函数。

$$\hat{T}(s'|s,a) = \begin{cases} \mathbb{M}(u_i)/K_{s,a}, & (s' = \langle u_i, 2\rangle) \\ \mathbb{M}(L\backslash u_i)/K_{s,a}, & (s' = \langle \backslash u_i, n-1\rangle) \\ (\prod\limits_{\langle L', n'\rangle \in A(s)\backslash s'} 1 - \mathbb{M}(L'))/K_{s,a}, & (s' = \langle L, 2n\rangle, 2n \le |L|) \\ 0, & (s' = \langle L, 2n\rangle, 2n > |L|) \end{cases} \tag{2.5}$$

其中，$K_{s,a}$ 是归一化因子。$\mathbb{M}$ 是对输入划分的合法性预测，返回输入是合法的概率。近似的激励函数定义为 $\hat{R}(\langle L, n\rangle = \prod_{1\le i\le |L|}(1 - \mathbb{M}(L\backslash u_i))$。

算法7展示了 CHISEL 算法的主要步骤。CHISEL 使用行级别粒度来切分输入程序（算法第 1 行）。类似 ddmin 算法，CHISEL 将初始分割粒度设置为 2（算法第 2 行），初始的输入可以通过反馈函数，CHISEL 将其作为 $\mathbb{M}$ 的第一个训练数据（算法第 3 行）。接下来，CHISEL 算法在主循环内缩减程序（算法 4-14 行），并根据简化后的语句集合返回简化后的程序 P'（算法第 15 行）。根据以下公式，算法第 6 行建立策略函数 $\hat{\pi}$。

$$\hat{\pi}(s) = \arg\min_{a\in A(s)} \sum_{s'} \hat{T}(s'|s,a)\hat{V}(s') \tag{2.6}$$

$$\hat{V}(s) = \hat{R}(s) + \gamma \sum_{s'} \hat{T}(s'|s,\hat{\pi}(s))\hat{V}(s'), (0 \le \gamma < 1) \tag{2.7}$$

策略函数根据当前状态返回下一个状态（算法第 7 行）。根据反馈信息，CHISEL 函数进行状态转移。当无法进一步划分集合的时候，CHISEL 算法终止。

图2.8展示了用于演示算法的示例 C 程序，包含 8 行语句。图2.9(a)展示了 CHISEL 算法处理该程序的迭代过程，可以看出 CHISEL 算法使用 10 次反馈函数的调用后返回简化后的程序。在 CHISEL 执行第一步时，选择了 $\langle f5, f6, f7, main\rangle$ 的子集作为反馈函数的输入，反馈函数返回失败。图2.9(b)中最左侧的子图显示了该次尝试后学习到的决策树模型。该决策树状态表示，当 $f4$ 函数存在时，反馈函数才可以通过。在第二步时，CHISEL 选择 $\langle f1, f2, f3, f4\rangle$ 进行删除尝试，反馈函数返回失败，同样地，该结果被加入决策树的学习数据中，学习到的决策树状态如图2.9(b)中第二个子图所示，该决策树预测只有当 $f4$ 和 $main$ 两个函数同时存在时，反馈函数才会通过。结合 ddmin 算法的执行步骤，划分粒度为 2 的所有可能都被尝试了，此时划分粒度变为 4，因此在下一步 CHISEL 选择除 $f1, f2$ 函数以外的部分作为反馈函数的输入。直到 CHISEL 第 10 步返回失败，所有的可能都被尝试了，CHISEL 将第 8 步中的结果返回，算法终止。

```c
double d = 0.10;
struct S {
  int f1;
  int f2;
};
void foo(struct S s, char str[]){
  double v = s.f2 + s.f2 * d;
  printf("%s %f\n", str, v);
}
int main(){
  unsigned int a = 1;
  char b[] = "first";
  char c[] = "second";
  if (a) {
    struct S s1;
    s1.f1 = 1;
    s1.f2 = 4000;
    struct S s2;
    s2.f1 = 2;
    s2.f2 = 2000;
    foo(s1,b);
    foo(s2,c);
  }
  printf("Hello world!\n");
  return 0;
}
```

图 2.10　用于演示 PARDIS 算法的示例程序

正像图2.9(b)中最后一个子图所示的决策树状态那样，仅当 $f1, main$ 两个函数同时存在时，反馈函数通过。然而，在实际中，由于仅利用在线数据进行训练，CHISEL 的决策树模型并不能达到理想的效果，与实际情况相差甚远。CHISEL 进一步加入了语法检查和定义-使用分析，跳过所有不合法的划分尝试，得以提升差异调试技术的性能。这样的改进具有极强的局限性，导致 CHISEL 只能应用于 C 语言程序，且并未改变按照固定尝试顺序进行尝试的缺陷。

可以看出，GTR 技术和 CHISEL 技术在算法执行过程中均离不开 ddmin 算法处理给定的集合，因此，它们没有从根本上解决差异调试的问题，在迭代过程中一定程度地依赖固定化尝试删除元素的顺序。尽管在这些技术中结合了机器学习的模型，从验证结果来看，这类使用机器学习模型的技术具有较为局限的泛化性。

Gharachorlu 等人发现了 Perses 技术中存在严重的优先级反置（priority inversion）

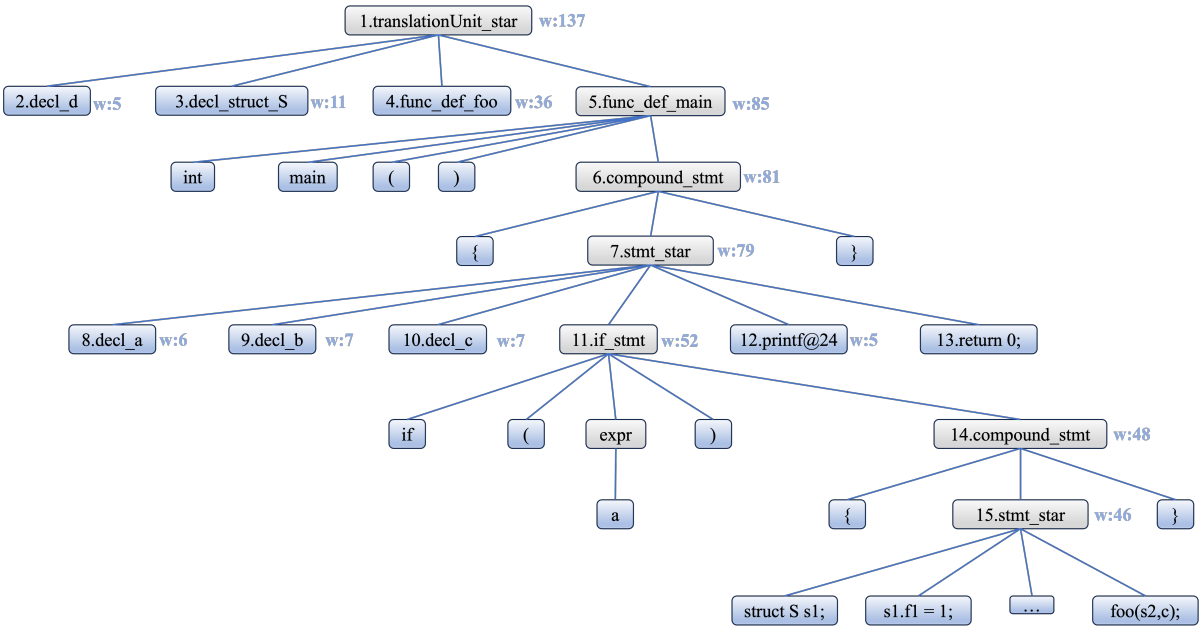


图 2.11　PARDIS 算法示例程序对应的语法树

问题，即尽管采取了措施优先处理那些具有高优先级的节点，在这一过程中往往低优先级的节点被大量地处理，导致时间开销增大。Gharachorlu 等人提出 PARDIS 技术，用于解决优先级反置的问题[112]；基于 PARDIS，研究[113] 引入了机器学习模型用于过滤那些包含语义错误的变体输入，进一步提升 PARDIS 技术的效率。

如图2.10所示为一个示例 C 程序，包含 26 行语句，图2.11是其对应的语法树，树中 w 意味着每个节点在 Perses 技术中被赋予的权重（所包含的标记数），为了简略起见，仅展示主要节点的权重。用这个例子，来展示 Perses 技术中存在的优先级反置问题。Perses 技术每次从工作列表中返回具有最大权重的节点进行处理。根据语法树中所

算法 8: PARDIS 算法

输入: 待简化程序 P
输入: 反馈函数 ϕ
输入: 节点排序器 ρ
输出: 简化之后的程序 p

```
1 p ← P ;
2 work ← MAXPRIORITYQUEUE({p.root}, ρ) ;
3 while not work.empty() do
4 |   node ← work.takeMax() ;
5 |   if node.isNullable and φ(p − node) then
6 |   |   p ← p − node ;
7 |   else work.insert(node.children) ;
8 end
9 return p;
```

展示的，节点 1.translationUnit_star 具有最高的优先级，因此它被从工作列表中提出并进行下一步处理。由于该节点是一个数量节点，因此 ddmin 算法被调用，用于缩减由其子节点构成的集合。然而，构成集合的子节点中，仅有一个节点具有 85 这样较高的权重值，其它子节点的权重值分别为 5、11、36，均比较小。Perses 技术的本意是优先处理权重值大的节点，现在反而优先处理了小权重值的节点，因此具有优先级反置的问题，极大影响了差异调试的性能。

PARDIS 技术引入节点排序器 ρ 来解决上述问题。算法8展示了 PARDIS 算法的主要步骤。可以看出，PARDIS 的主要流程遵循了 Perses 技术，唯一不同的是给节点赋权重的策略。通过引入节点排序器 ρ，PARDIS 可以方便地通过更换 ρ，达到使用不同权重策略进行差异调试的目的。在 PARDIS 初期的研究工作中，作者使用了一种包含检查在子节点中所处位置的策略为节点赋权重。后续，PARDIS 根据历史数据训练了一个能够判断是否语义合法的机器学习模型，为节点赋权重。

PARDIS 系列技术调整了 Perses 技术中的权重赋值策略，由于 Perses 并没有解决差异调试的本质问题，PARDIS 系列技术依然依赖 ddmin 算法中固定化的删除尝试序列。类似地，PARDIS 系列技术为了在差异调试迭代过程中尽可能地产生符合语义正确的程序，使用历史数据训练了机器学习模型。一方面，依赖训练模型的差异调试技术很难具有较强的泛化能力，仅在相应的验证集上效果较好，在实际应用中很难发挥作用；另一方面，在预测一个程序是否语义正确或能通过编译，机器学习模型需要额外的开销，在差异调试迭代过程中可能涉及上千次预测行为，产生的额外开销对于提高差异调试的效率而言是不可接受的。

2.3 讨论与小结

虽然针对差异调试问题的研究在二十多年的发展历程内涌现了大量新颖的技术，这些技术并没有从根本上解决现有技术遵循固定化尝试顺序删除元素的缺陷，差异调试问题仍然存在严重的效率和效果问题。尽管一些领域特定的差异调试技术在具体领域获得了巨大的成果，它们具有极强的局限性，不能被其它领域广泛使用。另一方面，一些基于机器学习的差异调试技术利用收集的训练数据，在相应应用领域获得了巨大的提升，它们仍然存在着十分明显的缺陷：即便是结合了大量的训练数据，得到的模型面临着准确性的问题。此外，模型的预测行为也会带来额外的开销。几乎所有的差异调试技术均基于 ddmin 算法，采用固定的尝试顺序简化输入集合，且没有充分利用在差异调试迭代过程中积累的历史反馈数据有效地指导差异调试过程。另一方面，现有差异调试技术将差异调试中的反馈函数视为二元函数，仅考虑通过或失败的二元反馈，没有考虑反馈函数的内部构成。

本文提出一系列概率化差异调试技术，在差异调试迭代过程中灵活地指定尝试删除的元素，并充分地利用差异调试过程中积累的反馈信息指导差异调试过程。此外，本文通过精化反馈函数提供的反馈信息，获得细粒度的反馈，并提出了基于细粒度反馈的概率化差异调试技术，进一步地提高了差异调试的效率和效果。从本文后续的实验验证可以看出，本文提出的概率化差异调试技术能够很好地解决上述问题，并提高差异调试的效率和效果。

第三章　概率化差异调试框架

差异调试问题涉及如何在满足特定性质的同时减小一个集合的大小。这个问题在许多应用中广泛存在，例如编译器开发、回归故障定位和缓解软件膨胀问题。差异调试问题本质上是一个搜索问题，即在庞大的求解空间中，搜索满足条件的最小解。然而，由于实际应用中，输入集合中包含大量的元素，无法在多项式时间内判定某个解是最小解，当无法继续简化集合的时候，将当前解作为结果返回。与大部分搜索问题不同的是，由于差异调试问题要求返回的结果必须满足特定的性质，在搜索过程中需要不断调用具有一定开销的反馈函数检查当前解是否满足规定的性质。因此，差异调试的目标是尽可能少地调用反馈函数的同时返回尽可能小的集合。

差异调试可以形式化地定义如下。设 $\mathbb{X}$ 是所有感兴趣元素组成的集合，$\phi : \mathbb{X} \to \{F, T\}$ 是一个反馈函数，用于确定一个集合满足给定的性质（T）或不满足给定的性质（F）并提供相应的反馈。$|X|$ 表示集合 $X \in \mathbb{X}$ 的大小。给定一个满足 $\phi(X) = T$ 的集合 $X \in \mathbb{X}$，差异调试的目标是找到另一个集合 $X^* \in \mathbb{X}$，使得 $|X^*|$ 尽可能小且 $\phi(X^*) = T$，即 X^* 满足规定的性质。例如，在编译器开发过程中，给定一个揭露编译器缺陷的测试用例（一个程序），差异调试技术被用于找到一个较小规模的测试用例，在保证重现编译缺陷的同时，提高开发人员调试缺陷的效率。这里，$\mathbb{X}$ 是程序的集合，X 是一个可能很大的程序，导致编译失败，而 ϕ 则是用于判断是否触发相同编译缺陷的反馈函数。

差异调试是基本的自动化调试手段之一，其效率和效果是长期制约差异调试应用范围的首要因素。虽然存在多种现有算法来高效地解决差异调试问题，但是，目前最先进的算法的效率和效果仍然不令人满意。本文旨在提高差异调试的效果和效率。本文的关键洞察是，大多数现有技术，遵循固定的顺序尝试从输入集合中删除元素，并未充分利用迭代过程中积累的反馈信息。为了解决上述问题，本章提出了概率化差异调试框架 ProbDD。该框架利用给定概率模型来估计输入集合中的元素被保留在最终结果里的概率，以合理的方法确定待选择的子集，利用迭代过程中积累的反馈信息更新模型，期望达到提升差异调试的效率和效果的目的。

3.1　启发性示例

本节使用一个程序最小化的例子来说明现有差异调试技术的工作原理。如图3.1所示代码，展示了来自 TensorFlow 教程的一个真实程序[114]。假设函数 `type()` 存在错误，任何对它的有效调用都会导致相同的错误。在减少程序的规模的同时，使得缩减后的程序仍然能够揭露相同的错误。程序中有 8 条语句，差异调试的目标是找到一个语句

```
import tensorflow as tf
x = tf.constant(3.0)
b = 1.0
with tf.GradientTape() as tape:
    tape.watch(x)
    y = x**2
    b = tape.gradient(y,x)
    print(type(b))
```

图 3.1　示例代码

的子集，仍然调用函数 `type()` 并产生错误。在本例中，s_i 表示第 i 行的语句。

差异调试的基础算法 ddmin 的输入是一个元素集合，并通过两个嵌套的循环进行处理。外部循环用于减少表示要考虑的子集长度的变量，n。n 从所有元素的一半开始，并在每次迭代时减半，直到达到 1。内部循环首先调用反馈函数检查长度为 n 的所有连续和不相交的子集是否满足规定的性质，然后调用反馈函数检查这些子集的补集。如

	s1	s2	s3	s4	s5	s6	s7	s8	
1	s1	s2	s3	s4	s5	s6	s7	s8	F
2	s1	s2	s3	s4	s5	s6	s7	s8	F
3	s1	s2	s3	s4	s5	s6	s7	s8	F
4	s1	s2	s3	s4	s5	s6	s7	s8	F
5	s1	s2	s3	s4	s5	s6	s7	s8	F
6	s1	s2	s3	s4	s5	s6	s7	s8	F
7	s1	s2	s3	s4	s5	s6	s7	s8	F
8	s1	s2	s3	s4	s5	s6	s7	s8	F
9	s1	s2	s3	s4	s5	s6	s7	s8	F
10	s1	s2	s3	s4	s5	s6	s7	s8	F
11	s1	s2	s3	s4	s5	s6	s7	s8	F
12	s1	s2	s3	s4	s5	s6	s7	s8	F
13	s1	s2	s3	s4	s5	s6	s7	s8	F
14	s1	s2	s3	s4	s5	s6	s7	s8	F
15	s1	s2	s3	s4	s5	s6	s7	s8	F
16	s1	s2	s3	s4	s5	s6	s7	s8	F
17	s1	s2	s3	s4	s5	s6	s7	s8	F
18	s1	s2	s3	s4	s5	s6	s7	s8	F
19	s1	s2	s3	s4	s5	s6	s7	s8	F
20	s1	s2	s3	s4	s5	s6	s7	s8	F
21	s1	s2	s3	s4	s5	s6	s7	s8	T
22	s1	s2	s3	s4	s5	s6	s7	s8	F
23	s1	s2	s3	s4	s5	s6	s7	s8	F
24	s1	s2	s3	s4	s5	s6	s7	s8	F
25	s1	s2	s3	s4	s5	s6	s7	s8	F
26	s1	s2	s3	s4	s5	s6	s7	s8	F
27	s1	s2	s3	s4	s5	s6	s7	s8	F
28	s1	s2	s3	s4	s5	s6	s7	s8	F

图 3.2　ddmin 算法的迭代步骤示例

果任何反馈函数的调用反馈通过，则只保留反馈通过的子集。如果一个子集或其补集之前已经被反馈函数检查过，则从反馈历史中查询反馈结果，跳过调用反馈函数的步骤。

ddmin 在这个例子中的执行流程如图3.2所示。每行末尾有一个 T 或 F，表示相同的错误是否仍然存在（T 表示存在，F 表示不存在）。首先，n 为 4，长度为 4 的两个子集在第 1 行和第 2 行调用反馈函数。两个子集都产生失败的反馈，在这个过程中它们互为补集，所以第一轮外部迭代结束。其次，n 减半为 2，长度为 2 的四个子集在第 3 行到第 6 行调用反馈函数。反馈函数均反馈失败，它们的补集也都反馈失败，所以第二轮外部迭代结束。

第三，n 减半为 1，长度为 1 的八个子集在第 11 行到第 18 行调用反馈函数，反馈函数都反馈失败。然后，反馈函数检查它们的补集，$\{s_3\}$ 的补集在第 21 行通过了反馈函数。由于反馈通过，补集中的元素被保留，需要检查这七个子集及其补集，所以算法继续以 n 为 1 进行。然而，在之前检查过的这七个子集中，没有一个通过反馈函数，它们的补集也没有一个通过反馈函数（第 22 行到第 28 行），所以第三轮外部迭代结束，算法返回 $\{s_1, s_2, s_4, s_5, s_6, s_7, s_8\}$。返回的集合无法通过删除任何单个元素来进一步减少它的大小，因此满足 1-最小性。从这个例子中可以看出，ddmin 的尝试顺序是预先定义好的固定顺序，并且不会从过去失败的反馈结果中学习。例如，在这个例子中，ddmin 算法仅利用了反馈函数反馈的检查集合满足规定性质的反馈信息（通过 T），如在第 21 次迭代时，相应的集合通过了反馈函数，并在后续过程中，基于该集合，指定待选择的结合并调用反馈函数检查；语句 s_8 不应该被删除。然而，按照固定的顺序，语句 s_8 已经尝试了 13 次删除，而且所有这些尝试都失败了。

事实上，正如 Zeller 和 Hildebrandt 在研究中所指出的[4]，ddmin 在最坏情况下调用反馈函数的渐进次数是 $O(n^2)$，其中 n 是初始集合的大小。此外，简化后的结果包含了七个语句，而最优结果是 $\{s_3, s_8\}$，只包含两个语句。由于本例中的集合元素数量较少，可以轻松地通过枚举遍历每一个子集，调用反馈函数来找到满足规定性质的最小的集合。然而，在实际应用中，差异调试输入集合的元素数量较大，无法在多项式时间内通过遍历的方式找到最小集合，只能尽可能地找到规模较小的集合。

3.2　框架 ProbDD 思想的来源

考虑到在差异调试过程中充分利用积累的反馈信息，贝叶斯优化[115-117] 的思想天然地适合用于差异调试问题，然而，已有的贝叶斯优化技术不能够直接应用于求解差异调试问题。给定一个概率模型，贝叶斯优化通过反馈结果计算后验分布来更新模型。ProbDD 可以被视为一种专门为差异调试问题设计的贝叶斯优化框架。与经典的为高斯

算法 9: 贝叶斯优化框架

输入: 优化目标 f，关于 f 的先验
输出: 使 f 最优的解或能够获得最大后验均值的解

```
// N 为观测数据总数；
while n ≤ N do
    根据所有观测数据更新后验分布；
    根据后验分布估计目标函数，并取最优的x_n作为下次观测目标；
    观测y_n = f(x_n)；
    将新的观测加入数据中，并处理下一个n；
end
return 一个满足要求的解
```

过程回归建模的贝叶斯优化算法不同[118]，ProbDD 针对的是具有二元反馈结果的差异调试问题。尽管最近一些针对二元目标函数的贝叶斯优化方法[119, 120] 被提出，但它们是为特定任务设计的。此外，还有一些用于其它调试任务的贝叶斯方法被提出，例如切片[121] 和故障定位[122]。据本文作者所知，目前还没有现有的贝叶斯优化方法可以应用于解决差异调试问题。此外，启发式搜索算法（如遗传搜索算法[123]）的思想也可以应用于解决差异调试问题，但与经典的贝叶斯优化类似，经典的启发式搜索算法依赖于连续的适应度函数。由于反馈结果是二元的，如何设计一个有效的适应度函数来指导这些算法是未来研究的一个开放问题。

此外，以 ChatGPT[124] 为代表的深度学习技术被广泛应用于软件工程领域的各个领域问题中。然而，由于训练领域深度学习模型需要大量数据，目前并不存在差异调试技术相关的大型数据集；另一方面，根据已有工作[13, 109, 112]，针对一种调试问题的数据并不能为另一种调试问题提供有效的经验，因此本文不考虑将深度学习作为概率模型，应用于求解差异调试问题。此外，有限的深度学习模型推理的性能也是导致其无法应用于差异调试领域的主要原因之一。针对程序的模型推理的时间消耗为秒级，在差异调试迭代过程中可能涉及数千次甚至上万次的推理，这对于提高差异调试的效率来说是不可接受的。未来，随着相关领域的发展，这或许会是一个颇具潜力的研究方向。

贝叶斯优化是一种机器学习优化算法，用于求解

$$\max_{x \in A} f(x), \tag{3.1}$$

其中，目标函数 f 是连续的，且需要花费巨大代价进行计算。此外，适用贝叶斯优化求解的问题，输入 x 所属的实数空间应具有较低的维度。然而，差异调试问题中的优化目标并不属于连续空间，且输入规模较为庞大。例如，一个典型的行粒度的程序简化问题的输入通常是十几万行的代码，产生的搜索空间是十分庞大的。可见，传统的贝叶

斯优化框架并不适用于求解差异调试问题。算法9展示了贝叶斯优化框架。贝叶斯优化框架在每次迭代中使用观测数据计算后验概率（算法第 3 行），然后利用后验分布估计目标函数并获取下一次观测目标（算法第 4 行）。在有限的迭代过程后，返回一个满足要求的解。本文从贝叶斯优化框架中获得启发，提出了概率化差异调试框架 ProbDD。

3.3　框架 ProbDD 的提出

本章提出如图3.3所示的概率化差异调试框架。给定概率模型，指定待选择的集合后调用反馈函数，根据反馈结果更新模型，该过程不断重复直到概率模型收敛。为了完成上述迭代过程，概率模型需要规定指定待选择集合的方法和根据反馈计算用于更新模型的后验概率的方法。通过结合概率，本框架支持使用更加灵活的方式指定待选择的集合，且充分地利用了迭代过程中积累的反馈信息。

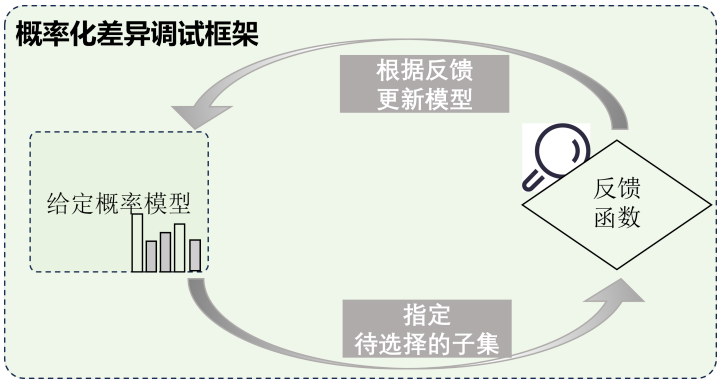


图 3.3　概率化差异调试框架

相应地，算法10展示了概率化差异调试框架求解差异调试问题的主要步骤。框架的输入包括概率模型 $\mathbb{M}$、与该模型相关的先验分布、待简化集合 X 与反馈函数 ϕ，算法将简化后的集合作为输出。其中，关于算法输入中的概率模型及其先验分布，将在第3.3.1节中介绍。差异调试迭代的过程主要在主循环内进行（算法 2-12 行），直到 $\mathbb{M}$ 收敛为止，即概率模型中的随机变量均为 0 或 1。算法第 3 行根据后验分布计算并指定待选择子集的过程将在第3.3.2节中介绍；算法 5-11 行关于根据反馈结果更新概率模型的过程将在第3.3.3节中介绍。

3.3.1　概率模型

符号定义。为方便后续章节的内容，先对涉及概率模型的符号进行定义。形式上，一般的差异调试定义如下：设 $\mathbb{X}$ 为所有感兴元素组成的集合。设 $\phi : \mathbb{X} \to \{F, T\}$ 为一个反馈函数，用于确定一个集合是否满足给定的性质（T）或不满足（F）。同时，设 $|X|$ 表示集合 $X \in \mathbb{X}$ 的大小。差异调试的目标是在给定满足 $\phi(X) = T$ 的集合 $X \in \mathbb{X}$ 的情

算法 10: 概率化差异调试算法

输入: 概率模型 $\mathbb{M}$，先验分布，待简化集合 X，反馈函数 ϕ
输出: 简化后的集合

使用先验分布初始化概率模型$\mathbb{M}$;
while $\mathbb{M}$未收敛 **do**
    根据后验分布计算并指定待选择子集X';
    // 如果 X' 被选择过，则 ϕ 会从历史记录中查找相应反馈结果；否则调用反馈函数后记录反馈结果;
    观测$Y' = \phi(X')$;
    if $Y' == T$ **then**
        // 提供通过反馈时;
        按照公式3.6计算后验概率;
        $X \leftarrow X'$;
    else 按照公式3.7计算后验概率 // 提供失败反馈时;
    使用后验分布更新模型参数;
end
return 简化后的集合

况下，发现另一个集合 $X^* \in \mathbb{X}$，使得 $|X^*|$ 最小化，同时保持 $\phi(X^*) = T$。换句话说，X^* 满足所规定的性质。

由于本章提出框架需要针对问题设计概率模型才能应用，第四章和第五章分别提出了面向集合和面向语法树的概率模型，为方便读者查看，本节一并定义了相关的符号。通常情况下，差异调试的输入是一个集合，并尝试找到一个子集使得反馈函数通过。全集 $\mathbb{X}$ 是一个 n 维布尔空间，而全集中的集合 X 可表示一个布尔向量 $X = \langle x_1, x_2, ..., x_n \rangle$，其中 $x_i \in \{0, 1\}$。这里，$x_i = 1$ 表示第 i 个元素包含在集合中，$x_i = 0$ 表示第 i 个元素不包含在集合中。为了简化表达，也将一个集合 X 视为一个包含了被包括元素索引的集合，即 $\{i \mid x_i = 1\}$，这样，集合运算符如 $\subseteq$ 就可以方便地得以应用。

针对输入是程序的情况，为充分考虑程序结构上的依赖关系，本文第五章提出面向语法树的概率模型，包含一个初始解析步骤，将输入集合 X 转换为抽象语法树 T。这棵树的节点，用 $N = \langle n_1, n_2, \ldots, n_k \rangle$ 表示，可以分为两个子集：叶节点集合 N_l 和非叶节点集合 N_{nl}。原始输入集合 X 中的每个元素对应于叶节点集合 N_l 中的一个元素。第五章所提出的技术旨在识别保留输入所规定的性质并最小化其大小的最优树状结构 T^*。通过遍历最优树状结构 T^* 的叶节点，可以得到最优子集 X^*。本质上，第五章所提出的技术利用输入集合的结构化表示来指导差异调试过程，确保所选择的集合符合语法结构，并实现所需的优化。

为了进一步提高概率化差异调试的效率和效果，本文第六章所提出的技术，精化反馈函数提供的反馈信息，利用细粒度的反馈信息更新模型。假设用于检查集合是否满足规定性质的反馈函数 $\phi : \mathbb{X} \rightarrow \{F, T\}$ 包括 m 个检查步骤，每个处理步骤由一个函

数 $\gamma_i(1 \le i \le m) : \mathbb{X} \to \{F, T\}$ 表示。函数 ϕ 的行为规则被定义如下：

$$\phi(X) = \begin{cases} T & \text{如果} \forall i, \gamma_i(X) = T \\ F & \text{如果} \exists i, \gamma_i(X) = F \end{cases} \tag{3.2}$$

3.3.1.1 最优子集的存在性

差异调试的目标是找到最优子集，即满足规定性质的最小子集。然而，由于实际应用场景中，输入集合包含大量的元素，无法在多项式时间内确定返回结果的最小性，通常将无法继续约简的子集作为结果返回。本节通过定义差异调试中集合的相关性质，讨论最优子集的存在性。为了简化概率分析，假设全集 $\mathbb{X}$ 具有两个性质，这些性质在现有的研究中经常被假设或讨论[1, 4]。请注意，假设这两个性质的目的是推导本文中概率模型的性质，而本文中相关方法的正确性和时间复杂度并不依赖于这两个假设。

单调性假设指的是如果 X 未通过反馈函数，那么 X 的任何子集也不能通过反馈函数。

定义 1 (单调性)**.** *对于任意的 $X, X' \in \mathbb{X}$，若 $X' \subseteq X$ 且 $\phi(X) = F$，则 $\phi(X') = F$。*

无歧义性假设指的是如果两个子集通过了反馈函数，它们的交集也会通过反馈函数。

定义 2 (无歧义性)**.** *对于任意的 $X, X' \in \mathbb{X}$, 如果 $\phi(X) = T$ 且 $\phi(X') = T$, 则 $\phi(X \cap X') = T$。*

本节证明了上述两个假设存在，意味着存在一个最优子集，当且仅当子集中的元素存在时，反馈函数才会通过。

定理 3.3.1. *如果一个集合 $\mathbb{X}$ 既是单调的又是明确的，那么存在一个最优子集 X^* 满足以下条件。*

$$\phi(X) = \begin{cases} T & X^* \subseteq X \\ F & \text{否则} \end{cases}$$

证明. 设 $X^* = \bigcap_{\phi(X)=T} X$。根据无歧义性假设，有 $\phi(X^*) = T$。根据单调性假设，得到 X^* 的任何超集 X 都使得反馈函数通过，即 $\phi(X) = T$。现在假设 X 是一个不是 X^* 的超集的子集。假设 $\phi(X) = T$，根据 X^* 的定义，有 $X \cap X^* = X^*$，这与 X 不是 X^* 的超集的事实相矛盾。因此，$\phi(X) = F$。 □

ProbDD 中的概率模型，用于估计输入集合中各个元素被保留在最终结果里的概率。该概率模型用于指定待选择的集合，并根据反馈结果更新概率模型，该过程直到概率模型中的参数收敛。

概率化差异调试框架需要针对问题设计概率模型才能应用。大量差异调试应用场景中，输入可以看作是一个元素集合。为应用框架到这样的场景上，本文提出**面向集合的概率模型**。简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。为了在差异调试中充分考虑语法结构上的依赖关系，本文提出**面向语法树的概率模型**本文第四章和第五章分别详细介绍了上述两种概率模型。

3.3.2 待选择集合的指定

根据给定的概率模型，概率化差异调试框架定义了选择集合的期望收益，在每一次迭代中指定具有最高期望收益的集合作为待选择的集合。首先，定义集合的收益。

3.3.2.1 集合的收益

当 X 通过反馈函数时，X 中被删除的元素的概率将被设为零，即这些元素不应再次被尝试删除（后续章节中讨论）。因此，每当反馈函数通过，都会从最终结果中删除一些元素。为了衡量反馈函数提供通过的反馈信息时能够删除多少元素，定义了集合 X 的收益。如果反馈函数通过，则收益为删除的元素数量，否则，收益为零。

$$gain(X, X_T) = \begin{cases} |ex(X, X_T)| & \phi(X) = T \\ 0 & \phi(X) = F \end{cases}$$

这里，X_T 表示最后一次通过反馈函数的集合，$ex(X, X_T)$ 表示当 X 通过反馈函数时新删除的元素集合，即 $ex(X, X_T)_i = 1$ 当且仅当 $x_i = 0$ 且 $p_i > 0$。在不会引起混淆的前提下，为了简化表达，这里省略参数 X_T，即用 $gain(X)$ 表示 $gain(X, X_T)$，用 $ex(X)$ 表示 $ex(X, X_T)$。

3.3.2.2 集合的期望收益

基于概率模型 $\langle p_1, p_2, \ldots, p_n \rangle$，可以计算集合 X 的期望收益。

$$\mathbb{E}[gain(X)] = |ex(X)|\, Pr(\phi(X) = T)$$

因此，确定一个待选择的子集的目标是选择一个子集 X，使得 $\mathbb{E}[gain(X)]$ 最大化。其中，计算某个子集通过反馈函数的概率涉及到模型推断算法，针对本文提出的两种概率模型，在后续章节分别介绍了如何进行模型推断以及选择具有最大化期望收益的子集的方法。

3.3.3 模型更新

在调用多次反馈函数之后，本节根据积累的反馈信息计算后验概率，并使用后验概率更新概率模型，指导差异调试过程。现在假设对集合 $X_1, X_2, \ldots, X_m$ 调用了多次反馈函数，反馈信息为 $R_1, R_2, \ldots, R_m$。将 X_i 的反馈结果为 R_i（即 $\phi(X_i) = R_i$）的事件表示为 T_i。然后可以按照以下方式计算 θ_i 的后验概率。

$$Pr(\theta_i = 1 | T_1, T_2, ..., T_n) = \frac{Pr(\theta_i = 1, T_1, T_2, ..., T_n)}{Pr(T_1, T_2, .., T_n)} \tag{3.3}$$

根据给定的 $T_1, T_2, ..., T_n$，当 $\theta_i = 1$ 时，事件的概率 $Pr(\theta_i = 1 | T_1, T_2, ..., T_n)$ 等于同时发生 $\theta_i = 1$ 和 $T_1, T_2, ..., T_n$ 的联合概率 $Pr(\theta_i = 1, T_1, T_2, ..., T_n)$ 与 $T_1, T_2, ..., T_n$ 的联合概率 $Pr(T_1, T_2, .., T_n)$ 的比值。计算上述两个联合概率的基本方法是枚举全集中的子集，并对符合条件的每个子集是最优的概率进行求和。如果子集 X 是与反馈结果 $T = \langle X', R \rangle$ 一致的，则当 X 是最优集合时，$\phi(X') = R$。更具体地，本节定义以下函数来判断一个子集是否与反馈结果一致。

$$\begin{aligned} con(X, \langle T_1, \ldots, T_m \rangle) &= \begin{cases} 1 & con'(X, T_1) \wedge \ldots \wedge con'(X, T_m) \\ 0 & \text{否则} \end{cases} \\ con'(X, \langle X', R \rangle) &= \begin{cases} \bigwedge_{x'_i=0} x_i = 0 & R = T \\ \bigvee_{x'_i=0} x_i = 1 & R = F \end{cases} \end{aligned} \tag{3.4}$$

基于这个函数，有以下结果。

$$\begin{aligned} &\frac{Pr(\theta_i = 1, T_1, T_2, ..., T_n)}{Pr(T_1, T_2, .., T_n)} \\ &= \frac{\sum_{X \in \mathbb{X}} \left(x_i * con(X, \langle T_1, \ldots, T_m \rangle) * \Pi_j p_j^{x_j} (1 - p_j)^{1 - x_j} \right)}{\sum_{X \in \mathbb{X}} \left(con(X, \langle T_1, \ldots, T_m \rangle) * \Pi_j p_j^{x_j} (1 - p_j)^{1 - x_j} \right)} \end{aligned} \tag{3.5}$$

计算上述公式是一个典型的**加权模型计数问题**[125]：为了求解该问题，需要对满足约束条件 $con(X, \langle T_1, \ldots, T_m \rangle)$ 的任何解 X 的权重进行求和，而解的权重是各个变量 x_i 的赋值的权重的乘积（即 p_i 或 $1 - p_i$）。然而，到目前为止，仍然缺乏一个高效的算法来解决加权模型计数问题：一个最先进的求解器通常需要数千秒来解决数千个变量的模型[125]，这在差异调试中是典型的情况。求解速度太慢，则无法加速差异调试。

为了提高效率，可以不基于所有反馈结果计算后验概率，而是在每个单独的反馈之后计算后验概率，并更新模型以进行后续的迭代过程。具体来说，在反馈 T 之后将 p_i 更新为 $Pr(\theta_i = 1 \mid T)$。这种方法忽略了不同反馈之间的相互作用，虽然不如前一种

方法精确，但可以高效地计算。针对本文提出的两种概率模型，在后续章节分别介绍了如何近似地计算后验概率的方法。下面，先介绍一般形式下的根据最近一次反馈结果更新每个 i 的 p_i 的方式。

一方面，如果反馈函数通过，则后验概率按照如下公式更新。

$$Pr(\theta_i = 1|\phi(X) = T) = \frac{Pr(\theta_i = 1)Pr(\phi(X) = T|\theta_i = 1)}{Pr(\phi(X) = T)} \tag{3.6}$$

另一方面，如果反馈函数失败，后验概率按照如下公式更新。

$$Pr(\theta_i = 1|\phi(X) = F) = \frac{Pr(\theta_i = 1)Pr(\phi(X) = F|\theta_i = 1)}{Pr(\phi(X) = F)} \tag{3.7}$$

由于公式3.6和公式3.7中涉及模型相关的计算，在后续章节中将详细讨论针对相应具体模型的模型推断算法。

3.3.4 模型的度量标准

为方便后续章节中讨论本文提出方法的性能，本节介绍了可能会被用于实验评估的度量标准。采用了三个在改进差异调试技术工作中常用的度量指标，这些指标是由之前的研究[13, 14, 96] 所确定的，包括：

i) 产生结果的大小：该度量指标衡量了最小化程序或代码片段的大小。使用结果中存在的标记数量来量化大小。

ii) 处理时间：该度量指标量化了差异调试技术所花费的时间。以秒为单位测量处理时间。

iii) 每秒删除的标记数量：该度量指标指示了在差异调试过程中标记被删除的速率。它提供了技术的效率信息。通过将总删除标记数除以处理时间来计算该度量指标。

结合相关的研究工作，考虑到用于差异调试技术评估的项目在上述三种指标上可能会存在着显著变化，为了确保结果的准确表示，当计算平均结果时，应使用了几何平均数而不是算术平均数。通过使用几何平均数，提供了一个更平衡和公平的度量指标，以衡量差异调试技术在不同项目上的整体效果。此外，针对发生超时情况的项目，生成结果的大小将是所有通过反馈函数的最小集合的大小，相应地处理时间将不作统计，用特殊的记号表示处理超时。

3.4 ProbDD 返回结果的正确性

本节证明了概率差异调试框架 ProbDD 的返回结果是正确的，有如下定理和证明过程。

定理 3.4.1. *PDD 返回的子集 X_O 将始终具有所规定的属性，即 $\phi(X_O) = T$。*

证明. 设 X^k 为一个子集，在第 k 次迭代后，所有概率为零的元素被删除，所有非零概率的元素被保留，即 $x_i^k = 1 \Leftrightarrow p_i \neq 0$，并且 X^0 是第一次迭代之前的集合。证明对于 $k = 0$ 和算法执行过程中的任意迭代 k，X^k 都通过了反馈函数，即 $\phi(X^k) = T$。

首先，很容易看出 X^0 是输入集合并通过了反馈函数。

假设 X^k 通过了反馈函数。如果在第 $k+1$ 次迭代中反馈函数失败，那么只有一些概率不为零的元素的概率会增加，因此 $X^{k+1} = X^k$ 仍然通过反馈函数。如果反馈函数通过，被移除元素的概率将被设为零，因此 X^{k+1} 与被调用反馈函数的子集相同，并通过反馈函数。

将上述内容综合起来，上述性质成立。由于 X_O 是最后一次迭代 k 的 X^k，可知 X_O 通过了反馈函数。 □

3.5　讨论与小结

为考虑充分利用差异调试迭代过程中积累的反馈信息，本章受到贝叶斯优化思想的启发，提出了概率化差异调试框架 ProbDD。该框架利用给定的概率模型，指定待选择的集合，并通过反馈结果来更新模型[115]，可以被视为一种专门为差异调试问题设计的贝叶斯优化框架。与经典的为高斯过程回归建模的贝叶斯优化算法不同[118]，概率化差异调试框架针对的是具有二进制反馈结果的差异调试问题。尽管最近提出了一些针对二进制目标函数的贝叶斯优化方法[119, 120]，但它们是为特定任务设计的。此外，还有一些贝叶斯方法被提出用于其它调试任务，例如切片[121] 和故障定位[122]。据作者所知，目前还没有现有的贝叶斯优化方法可以应用于解决差异调试问题。此外，启发式搜索算法（如遗传搜索算法[123]）被广泛用于解决黑盒优化问题，但与经典的贝叶斯优化类似，经典的启发式搜索算法依赖于连续的适应度函数。由于反馈结果是二进制的，如何设计一个有效的适应度函数来指导这些算法是未来研究的一个开放问题。根据本文作者的了解，目前唯一可以应用于差异调试的启发式搜索方法是 ACTIVECOARSEN[126]。该方法旨在使用启发式搜索在静态分析领域找到最小的抽象。在第四章的实验评估中，将 ACTIVECOARSEN 作为基准，得出的结果表明，在这两个技术中，基于 ProbDD 框架的技术优于 ACTIVECOARSEN，能够改进代表性方法。最近，Xin 等人[111] 还提出了一种基于 MCMC 和 Metropolis-Hastings 采样的软件简化技术 DEBOP，这是一种典型的启发式技术。然而，DEBOP 针对的是一个与标准问题不同的软件精简问题：优化目标是最大化一组连续的目标函数。换句话说，该方法的目标问题不涉及二元的反馈函数，因此可以方便地应用 MCMC 算法求解。

本章提出了概率化差异调试框架 ProbDD，通过给定的概率模型来估计输入集合中每个元素被保留在最终结果里的概率，在差异调试迭代过程中，利用积累的历史反馈信息，计算后验概率并更新概率模型。区别于现有技术使用固定顺序尝试删除元素，且没有充分利用历史反馈信息的缺点，本章提出的框架期望提升差异调试技术的效率和效果。由于本章提出的框架依赖于具体的概率模型，实验验证只在实例化的技术章节体现。本文第四章提出面向集合的概率模型，实例化为面向集合的概率化差异调试技术；第五章提出面向语法树的概率模型，实例化为面向语法树的概率化差异调试技术；第六章提出基于细粒度反馈的差异调试技术，支持在不修改具体概率模型的条件下，进一步地提升差异调试技术的效率和效果。

第四章　面向集合的概率模型

概率化差异调试框架需要针对问题设计概率模型才能应用。现有的差异调试技术大都建立在 ddmin 算法的基础上[4]。ddmin 算法将一个输入 $X \in \mathbb{X}$ 视为一个集合。在每次迭代中，ddmin 将 X 分割为 n 个子集，并依次尝试从 X 中删除每个子集及其补集。初始时，n 为 2，并在每次迭代中被翻倍。后续基于 ddmin 技术的差异调试技术假设更复杂的领域特定结构，并将 ddmin 应用于从这些结构中产生的集合。例如，HDD[14] 假设输入具有树形结构，并仅将 ddmin 应用于同一层上所有兄弟节点组成的集合。CHISEL[13] 进一步考虑了 C 程序中元素之间的数据流和控制流的依赖关系，并以不破坏依赖关系的方式应用 ddmin 算法。然而，目前最先进的差异调试算法的效率和效果仍然不尽人意。例如，正如后续本章的评估中展示的那样，最先进的差异调试工具 CHISEL[13] 可能需要长达 3 个小时来缩减一个包含 14,092 行代码的程序，而缩减后的程序可能会比理想的目标程序多出 2 倍的冗余代码。

可见，通常情况下差异调试的输入是一个集合。为应用框架到这样的场景上，本章提出了面向集合的概率模型。给定面向集合的概率模型，概率化差异调试框架可被实例化为面向集合的概率化差异调试技术 PDD。PDD 根据面向集合的概率模型来估计每个元素保留在最终结果中的概率。在每次迭代中，PDD 根据概率模型选择一个最大化下一次期望收益的元素子集，并调用反馈函数判断该子集是否仍满足规定的性质。然后，PDD 根据反馈结果计算后验概率并更新概率模型。本章从理论上证明了返回的结果是极小的或最小的情形，并进一步分析了最坏情况下调用反馈函数的渐近次数。在最坏情况下，渐近次数为 $O(n)$，这比 ddmin 的渐近次数 $O(n^2)$ 要小，其中，n 为输入集合中的元素个数。

此外，通过两个应用领域（即树和 C 程序）中两种代表性技术（HDD[14] 和 CHISEL[13]）中的 ddmin 替换为 PDD，使用 40 个项目对本章所提技术进行了评估。根据已有知识，本章中用于评估的项目数量超过了所有近期在软件工程顶级会议上关于差异调试的工作[1, 2, 4, 13, 14, 96, 97, 99-101]。结果表明，在这两个应用领域中，PDD 显著提高了代表性技术的效率和效果。平均而言，在将 ddmin 替换为 PDD 后，HDD 和 CHISEL 分别在规定时间内产生了 59.48% 和 11.51% 更小的结果。在两个版本都在规定时间内完成的项目中，将 ddmin 替换为 PDD 后，HDD 和 CHISEL 分别使用了 63.22% 和 45.27% 更少的时间。总的来说，本章的主要贡献如下：

- 提出了面向集合的概率模型，并结合概率化差异调试框架，实例化为面向集合的概率化差异调试技术 PDD，该技术改进了约 20 年前提出的差异调试基础算

法 ddmin。

- 从理论上分析并证明了返回结果具有极小性和最小性的情形以及在最坏情况下调用反馈函数的渐近次数。
- 在两个应用领域中评估了 PDD，结果表明 PDD 在效率和效果上显著优于现有技术。

4.1 面向集合的概率模型

概率化差异调试框架需要针对问题设计概率模型才能应用。通常情况下差异调试的输入是一个集合。为应用框架到这样的场景上，本章提出了面向集合的概率模型。本节描述了 (1) 所提出的概率模型，(2) 如何根据反馈结果更新该模型，以及 (3) 如何使用该模型来指导差异调试过程。此外，为了方便读者更好地理解面向集合的概率模型，本节重新审视了本文3.1节中的启发性示例。

4.1.1 模型定义

4.1.1.1 模型

下面，将定义面向集合的概率模型。根据本文第3.3.1.1节中介绍的最优子集 X^* 的存在性，差异调试的目标是找出所有 X^* 中的元素。因此，为每个索引 i 分配一个伯努利随机变量 θ_i，用于表示第 i 个元素是否在 X^* 中。使用参数 p_i 来表示第 i 个元素在 X^* 中的概率，即 $Pr(\theta_i = 1) = p_i$。因此，该概率模型是一个 n 维参数向量 $\langle p_1, p_2, \ldots, p_n \rangle$。

进一步地，假设随机变量 θ 是相互独立的。这个假设是合理的，因为大部分差异调试技术已经考虑了输入集合所涉及到的领域特定结构，如果两个元素相互依赖，例如它们可以一起被删除但不能单独被删除，这样的依赖关系很可能被集成了 ddmin 算法的差异调试技术捕捉到。当 ddmin 应用于一个集合时，这个集合中的大多数元素是相互独立的。基于这个假设，向量 X 等于 X^* 的概率是 $\prod_i p_i^{x_i}(1-p_i)^{1-x_i}$。[①] 这也意味着 PDD 的差异调试过程在每个 p_i 等于 1 或 0 时停止。

使用上述概率模型，可以很容易地计算反馈结果的概率。即，X 通过反馈函数的概率是 X^* 中没有任何元素被删除在 X 之外的概率，即 $Pr(\phi(X) = T) = \prod_i (1-p_i)^{1-x_i}$。

4.1.1.2 模型的先验分布

由于最初用户对各个元素没有任何了解，可以将所有的 p_i 均匀地设置为 σ，其中 $0 < \sigma < 1$ 是面向集合的概率模型中的一个超参数。根据问题领域的特性，有多种方法可以确定 σ 的值。如果应用领域中通常具有固定的减少比例，将 σ 设置为该比例。如

① 在本文中，假设 $0^0 = 1$。

果约简后的子集通常具有固定的长度 m，可以将 σ 设置为 m/n，其中 n 是输入集合的长度。

4.1.2　待选择集合的指定

回顾第3.3.2.1节中定义的集合收益：

$$gain(X, X_T) = \begin{cases} |ex(X, X_T)| & \phi(X) = T \\ 0 & \phi(X) = F \end{cases}$$

在不会引起混淆的前提下，为了简化表达，这里省略参数 X_T，即用 $gain(X)$ 表示 $gain(X, X_T)$，用 $ex(X)$ 表示 $ex(X, X_T)$。根据面向集合的概率模型 $\langle p_1, p_2, \ldots, p_n \rangle$，可以计算一次集合 X 的期望收益。

$$\mathbb{E}[gain(X)] = |ex(X)| Pr(\phi(X) = T) = |ex(X)| \Pi_i (1 - p_i)^{1-x_i}$$

因此，指定一个待选择集合的目标是选择一个集合 X，使得 $\mathbb{E}[gain(X)]$ 最大化。

请注意，仅仅选择具有最大概率等于 X^* 的子集并不一定能够实现最大化期望收益的目标，因为它通过反馈函数的概率可能很低。为了理解如何最大化期望收益，首先考虑一个简单的情况，即所有概率 p_i 都相等。在这种情况下，任何相同大小的集合都会导致相同的期望收益。图4.1展示了当任意 p_i 为 0.1 时，期望收益 $\mathbb{E}[gain(X)]$ 和删除元素数量 $|ex(X)|$ 之间的关系。从图中可以看出，当更多元素被删除时，期望收益首先增加然后减少，在拐点处达到最大值。这是因为 $\mathbb{E}[gain(X)]$ 是两个因素的乘积，即 $|ex(X)|$ 和 $\Pi_i (1 - p_i)^{1-x_i}$。第一个因素单调增加，但增长速率逐渐减小。第二个因素单调减少，但减少速率保持不变。因此，必然存在一个点，在该点上减少速率超过增加速率，从而最大化期望收益。

现在，考虑概率不同的情况。第一个因素 $|ex(X)|$ 不受此变化的影响。对于第二个因素 $\Pi_i (1 - p_i)^{1-x_i}$，可能会出现相同长度的不同集合具有不同值的情况。为了选择具有最大值的集合，需要删除那些在 X^* 中具有最低概率的元素。基于上述分析，使用以下步骤来找到具有最大期望收益的集合。

1. 按照集合中元素的概率 p_i，将 X_T 中的元素概率按升序排序；
2. 根据上述顺序，逐个删除 X_T 中的元素，直到期望收益开始下降；
3. 返回具有最高期望收益的集合。设 $\hat{X}$ 是上述过程返回的集合。下面的定理表明，$\hat{X}$ 具有最大的期望收益。

定理 4.1.1. 对于任意 $X \subseteq X_T$，有 $\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(X)]$。

证明. 首先，使用 $S(k)$ 表示在步骤（2）中删除 k 个元素后获得的集合。首先，证明对于任意 $X \subseteq X_T$，集合 $S(|ex(X)|)$，该集合排除了与 X 相同数量的元素，但按照增加概

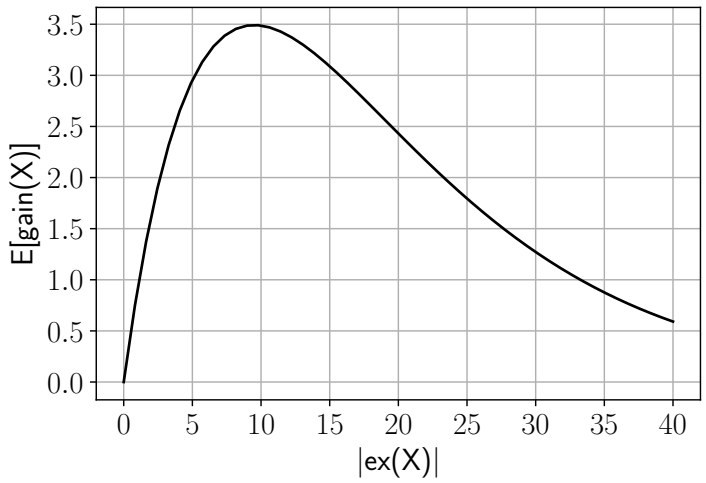


图 4.1　$\mathbb{E}[gain(X)]$ 与 $|ex(X)|$ 之间的关系

率的顺序选择元素，不能具有更差的期望收益。其次，证明算法返回的集合在 $S(k)$ 中具有最高的期望收益，其中 $1 \leq k \leq |X_T|$。因此，$E[gain(\hat{X})] \geq E[gain(S(|ex(X)|))] \geq E[gain(X)]$。详细证明见第4.2.3节。 □

4.1.3 模型更新

结合本章提出的面向集合的概率模型，以及第3.3.3节中介绍的计算后验概率的方法，本节介绍模型更新的方法。

下面将描述如何更新每个 i 的 p_i。首先有以下引理。

引理 4.1.2. *给定一个集合 X，其中 $x_i = 1$，即第 i 个元素在 X 中被保留，则 $\phi(X) \perp \theta_i$，即 $\phi(X)$ 和 θ_i 是独立的。*

证明. 将从 X 中被删除的元素的索引表示为 $j_1, j_2, ..., j_k$。那么

$$Pr(\phi(X) = F) = Pr(\theta_{j_1} = 1 \cup \theta_{j_2} = 1 \cup ... \cup \theta_{j_k} = 1)$$

和

$$Pr(\phi(X) = T) = Pr(\theta_{j_1} = 0 \cap \theta_{j_2} = 0 \cap ... \cap \theta_{j_k} = 0).$$

$\theta_{j1}, \theta_{j2}, ..., \theta_{jk}$ 和 θ_i 之间的独立性意味着 $\phi(X)$ 和 θ_i 之间的独立性。 □

鉴于上述引理，描述如何更新每个 i 的 p_i。一方面，如果反馈函数反馈失败，公

式3.7可被改写为：

$$
\begin{aligned}
&Pr(\theta_i = 1|\phi(X) = F) \\
&= \frac{Pr(\theta_i = 1)Pr(\phi(X) = F|\theta_i = 1)}{Pr(\phi(X) = F)} \\
&= \begin{cases}
\frac{Pr(\theta_i=1)\cdot 1}{1-\Pi_j(1-Pr(\theta_j=1))^{1-x_j}} = \frac{p_i}{1-\prod_j(1-p_j)^{1-x_j}} & x_i = 0 \\
\\
\frac{Pr(\theta_i=1)Pr(\phi(X)=F)}{Pr(\phi(X)=F)} = Pr(\theta_i = 1) = p_i & x_i = 1
\end{cases}
\end{aligned}
\tag{4.1}
$$

另一方面，如果反馈函数反馈通过，公式3.6可被改写为：

$$
\begin{aligned}
&Pr(\theta_i = 1|\phi(X) = T) \\
&= \frac{Pr(\theta_i = 1)Pr(\phi(X) = T|\theta_i = 1)}{Pr(\phi(X) = T)} \\
&= \begin{cases}
\frac{Pr(\theta_i=1)\cdot 0}{Pr(\phi(X)=T)} = 0 & x_i = 0 \\
\\
\frac{Pr(\theta_i=1)Pr(\phi(X)=T)}{Pr(\phi(X)=T)} = Pr(\theta_i = 1) = p_i & x_i = 1
\end{cases}
\end{aligned}
\tag{4.2}
$$

根据上述方程，按照以下规则在反馈函数每次反馈结果 $\phi(X) = R$ 后更新每个 i 位置元素对应的参数 p_i。

1. 如果第 i 个元素包含在 X 中，则 p_i 保持不变。
2. 如果第 i 个元素从 X 中删除，并且 $R = T$，即反馈通过，则 p_i 被设置为零。
3. 如果第 i 个元素从 X 中删除，并且 $R = F$，即反馈失败，则 p_i 被设置为 $\frac{p_i}{1-\prod_j(1-p_i)^{1-x_j}}$。

4.1.4　重新审视启发性示例

在本文第3.1节中展示了一个示例，用以说明现有差异调试技术存在的缺陷。为了展示概率化差异调试框架的优势，以及结合面向集合的概率模型，实例化后产生的面向集合的概率化差异调试技术（如图4.2所示）的优势，本节结合3.1节中的示例，展示如图4.3所示的一个可能的 PDD 执行序列。在图4.3 中，每个奇数行代表指定一个选择的集合并调用一次反馈函数，选中的元素显示在颜色较深的单元格中。每个奇数行的最后一个单元格显示了反馈函数反馈的结果。每个偶数行表示每个元素在模型更新后的概率。产生变化的概率值显示在颜色较深的单元格中。

假设期望的减少比率为 0.25，并且最初所有的概率都设置为 0.2500（设置模型先验的步骤可参考第4.1.1.2 节）。在每次迭代中，PDD 会持续删除概率最低的元素，直到期望的收益开始下降。由于初始概率都相等，第一次迭代的选择实际上是随机的。PDD 在删除了 s_1、s_2、s_3 和 s_8 后，反馈函数失败了。因此，在第4.1.3节的最后，PDD 根据

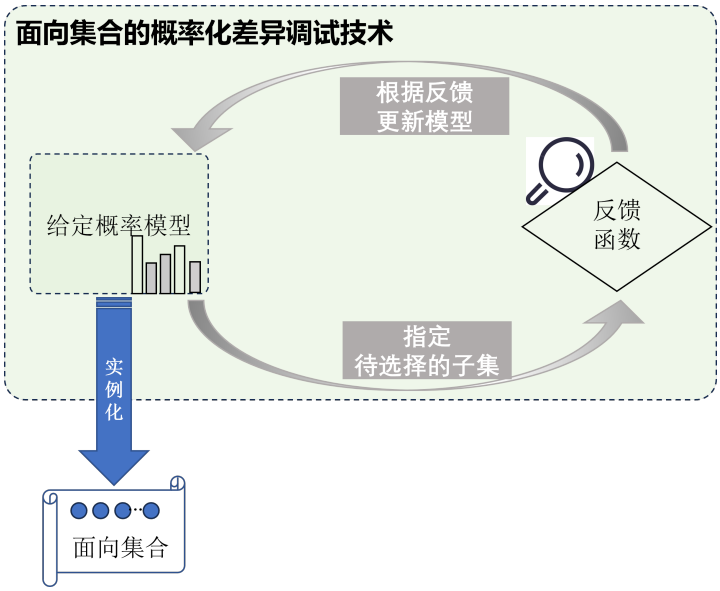


图 4.2　面向集合的概率化差异调试技术

规则（3）更新了被删除元素的概率。在第二次迭代中，PDD 选择了概率最低的四个元素 s_4、s_5、s_6 和 s_7 进行删除。在这种情况下，反馈函数通过了，所以 PDD 根据规则（2）直接将被删除元素的概率设置为零。在第三次和第四次迭代中，PDD 分别选择了 $\{s_2, s_3\}$ 和 $\{s_1, s_8\}$ 调用反馈函数。它们都未通过反馈函数，因此被删除元素的概率相应地进行了更新。在剩余的迭代中，剩余元素的概率已经提升到一个水平，以至于每次只能删除一个元素，并且被删除元素的概率将根据反馈信息相应地被设置为 0 或 1。最后，当每个元素的概率为 1 或 0 时，PDD 停止，并返回 $\{s_3, s_8\}$。

从上述过程中可以看出，PDD 从反馈历史中学习：当删除 s_8 失败时，s_8 的概率会增加，因此不会重复选择 s_8 进行删除。此外，在上述过程中，PDD 返回的结果 $\{s_3, s_8\}$ 比 ddmin 返回的结果要小得多。为了得到这个结果，需要删除 s_4、s_5、s_6 和 s_7，而且这四个元素需要一起被删除，否则无法重现错误。由于 ddmin 使用固定顺序来划分集合，它永远不会将这四个元素一起删除。另一方面，PDD 不使用固定的划分，理论上可以产生任何属于全集 $\mathbb{X}$ 中的集合。此外，在实际运行中，面向集合的概率化差异调试技术不会因为相较于现有差异调试技术额外利用了反馈函数提供的集合不满足规定性质的反馈信息，产生较大的开销。这是因为，本章通过提出一种近似的模型推断方法，面向集合的概率模型解决了概率化差异调试框架中涉及到的模型推断无法在多项式时间内解决的问题。由于 ddmin 和本章提出技术均是面向通用的差异调试的场景，本例中 ddmin 和 PDD 均把程序语句集合当做输入。实际上，从本章后续的实验部分可以看出，广泛被使用的差异调试技术使用某些方法分离出元素间相对独立的集合作为输入，如 HDD 技术中将语法树每一层的节点集合作为输入。

4.2　PDD 的性质

本节讨论 PDD 的效率和产生结果具有极小性或最小性的情形。关于正确性，已在第3.4节中进行了讨论。

4.2.1　效率

由于反馈函数的执行具有一定开销（消耗时间和计算资源），在差异调试迭代过程中调用反馈函数的次数主要反映了差异调试技术的效率，针对效率的讨论有如下定理。

定理 4.2.1. *对于输入大小为 n 的情况，PDD 在最坏情况下调用反馈函数的次数渐近地受到 $O(n)$ 的限制。*

证明. 首先，反馈函数每次反馈通过时，都会将至少一个元素的概率设为 0，因此最多可能有 n 个通过的反馈函数调用。其次，可以证明，在所有元素的概率要么为零，要么大于 0.5 之前，最多可能有 $O(n)$ 个失败的反馈函数调用。当剩余元素的概率都大于 0.5 时，算法将调用反馈函数检查逐个元素删除的情况，因此最多可能还剩下 $O(n)$ 个失败的反馈函数调用。具体细节请参见第4.2.3节。 □

请注意，这个定理意味着 PDD 总是在有限步内终止的。

4.2.2　极小性和最小性

定理 4.2.2. *如果满足单调性，PDD 的输出 X_O 是极小的，即对于任意 $X \subset X_O$，$\phi(X) = F$。*

	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	T
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	F
	0	0	1	0	0	0	0	1	

图 4.3　本章所提技术的迭代步骤示例

证明. 设 s_i 是 X_O 中但不在 X 中的元素。由于 X_O 是输出的结果，有 $p_i = 1$。很容易看出，只有当 $\forall k \neq i, p_k = 0 \vee x_k = 1$ 时，$\frac{p_i}{1-\Pi_j(1-p_j)^{1-x_j}} = 1$，即 X' 上存在一个失败的测试，只有 s_i 被新删除。由于 X_O 是输出，有 $X \subset X_O \subseteq X' \cup \{s_i\}$。由于 s_i 不在 X 中，有 $X \subseteq X'$。由于 $\phi(X') = F$，根据单调性，有 $\phi(X) = F$。 □

定理 4.2.3. 如果满足单调性和无歧义性，则 *PDD* 的输出 X_O 是最小的，即对于任意的 X，如果 $|X| < |X_O|$，则 $\phi(X) = F$。

证明. 根据定理3.3.1 的证明，最小的 $X^* = \bigcap_{\phi(X)=T} X$。由于 $\phi(X_O) = T$，有 $X^* \subseteq X_O$。假设 $X^* \subset X_O$。然后根据定理4.2.2，有 $\phi(X^*) = F$，这与 X^* 的定义相矛盾。因此，$X^* = X_O$，即 X_O 是最小的。 □

4.2.3 PDD 性质的补充证明

在本节中，给出了前文所涉及到的部分定理（定理4.1.1和4.3.1）的详细证明。

定理 4.2.3. 对于任意 $X \subseteq X_T$，有 $\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(X)]$。

证明. 为了证明这个定理，首先证明两个引理。将步骤（1）后的元素排序后的集合表示为 $\langle \hat{s}_1, \hat{s}_2, ..., \hat{s}_m \rangle$，其中 $\hat{s}_i$ 表示原始集合 X_T 中第 i 个元素的索引。进一步使用 $S(k)$ 表示在步骤（2）中删除 k 个元素后得到的集合，即从 X_T 中删除索引为 $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ 的元素得到的集合。

引理 1. 对于任意的 $X \subseteq X_T, \mathbb{E}[gain(S(|ex(X)|))] \geq \mathbb{E}[gain(X)]$.

证明. 回顾一下 $\mathbb{E}[gain(X)] = |ex(X)|\Pi_i(1-p_i)^{1-x_i}$。为了比较这两个期望收益，分别比较期望收益的两个组成部分，$|ex(X)|$ 和 $\Pi_i(1-p_i)^{1-x_i}$。

- 首先，很容易看出 $|ex(S(|ex(X)|))| = |ex(X)|$。
- 其次，将 $ex(X)$ 中元素的索引按概率升序排列，记为 $s_1, s_2, ..., s_{|ex(X)|}$。那么有 $p_{\hat{s}_1} \leq p_{s_1}, p_{\hat{s}_2} \leq p_{s_2}, ..., p_{\hat{s}_{|ex(X)|}} \leq p_{s_{|ex(X)|}}$，这意味着 $\prod_i(1-p_{\hat{s}_i}) \geq \prod_i(1-p_{s_i})$

因此，引理成立。 □

引理 2. 对于任意的 $1 \leq i < j \leq m-1$，有

$$\mathbb{E}[gain(S(j))] \cdot \mathbb{E}[gain(S(i+1))] \geq \mathbb{E}[gain(S(i))] \cdot \mathbb{E}[gain(S(j+1)]$$

。

证明.

$$
\begin{aligned}
& i < j \\
\Rightarrow & 1 - p_{\hat{s}_{i+1}} \geq 1 - p_{\hat{s}_{j+1}} \\
\Rightarrow & (ij + j)(1 - p_{\hat{s}_{i+1}}) \geq (ij + i)(1 - p_{\hat{s}_{j+1}}) \\
\Rightarrow & j(i+1)(1 - p_{\hat{s}_{i+1}}) \geq i(j+1)(1 - p_{\hat{s}_{j+1}}) \\
\Rightarrow & j\Pi_{k \leq j}(1 - p_{\hat{s}_k}) \cdot (i+1)\Pi_{k \leq i+1}(1 - p_{\hat{s}_k}) \\
& \geq i\Pi_{k \leq i}(1 - p_{\hat{s}_k}) \cdot (j+1)\Pi_{k \leq j+1}(1 - p_{\hat{s}_k}) \\
\Rightarrow & \mathbb{E}[gain(S(j))] \cdot \mathbb{E}[gain(S(i+1))] \\
& \geq \mathbb{E}[gain(S(i))] \cdot \mathbb{E}[gain(S(j+1)]
\end{aligned}
$$

□

基于这两个引理，证明了定理4.2.3。设 X 是任意的子集，满足 $X \subseteq X_T$。根据引理1，有 $\mathbb{E}[gain(S(|ex(X)|))] \geq \mathbb{E}[gain(X)]$。下面，将证明 $\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(S(|ex(X)|))]$。

- 如果 $|X| \geq |\hat{X}|$，那么根据算法的步骤(2)，有 $E[gain(S(|\hat{X}^{ex}|))] \geq \mathbb{E}[gain(S(|X^{ex}|))]$，因为只有在期望收益仍然增加的情况下，算法才会删除更多的元素。
- 现在考虑 $|X| < |\hat{X}|$ 的情况。根据步骤 (2)，有

  $$E[gain(\hat{X})] = E[gain(S(|ex(\hat{X})|))] > \mathbb{E}[gain(S(|ex(\hat{X})| + 1))].$$

  根据引理2，对于 $\forall k > |ex(\hat{X})|$，有

  $$
  \begin{aligned}
  & \mathbb{E}[gain(S(k))] \cdot \mathbb{E}[gain(S(|ex(\hat{X})| + 1))] \\
  \geq & \mathbb{E}[gain(S(|ex(\hat{X})|))] \cdot \mathbb{E}[gain(S(k+1)]
  \end{aligned}
  $$

  因此，对于 $\forall k > |ex(\hat{X})|$，有

  $$\mathbb{E}[gain(S(k))] \geq \mathbb{E}[gain(S(k+1))]$$

  。结果就是，

  $$
  \begin{aligned}
  & \mathbb{E}[gain(S(|ex(\hat{X})|))] \\
  > & \mathbb{E}[gain(S(|ex(\hat{X})| + 1))] \\
  \geq & \ \ldots \\
  \geq & \mathbb{E}[gain(S(|ex(X)|))]
  \end{aligned}
  $$

综上所述，可知原定理成立。 □

定理 4.3.1. 对于输入大小为 n 的情况，*PDD* 在最坏情况下调用反馈函数的次数渐近地受到 $O(n)$ 的限制。

证明. 回顾第4.1.3节中的更新规则，在每次迭代中，调用一次反馈函数，

- 如果反馈通过，至少会将一个元素的概率设置为 0。
- 如果反馈失败，至少会将一个元素的概率增加 $\frac{p_i}{1-\Pi_j(1-p_j)^{1-x_j}} - p_i = \frac{p_i * \Pi_j(1-p_j)^{1-x_j}}{1-\Pi_j(1-p_j)^{1-x_j}}$

为了证明总调用反馈函数的次数受 $O(n)$ 的限制，将分别考虑上述两种情况。

首先，由于总共有 n 个元素，将概率设置为 0 的操作最多可以执行 n 次。

其次，将展示可能的概率增加数量也受到 $O(n)$ 的限制。 □

使用定理4.1.1证明中的符号，下面首先证明三个引理。

引理 3. $|ex(\hat{X})| \leq \frac{1}{\sigma}$，其中 σ 是初始概率，$0 < \sigma < 1$。

证明. 首先，如果 $|ex(\hat{X})| = 1$，那么 $1 < \frac{1}{\sigma}$ 成立。

其次，如果 $|ex(\hat{X})| \geq 2$，

$$\begin{aligned} &\mathbb{E}[gain(S(|ex(\hat{X})|))] \geq \mathbb{E}[gain(S(|ex(\hat{X})| - 1))] \\ \Rightarrow &(1 - p_{|ex(\hat{X})|}) * |ex(\hat{X})| \geq |ex(\hat{X})| - 1 \\ \Rightarrow &|ex(\hat{X})| \leq \frac{1}{p_{|ex(\hat{X})|}} \end{aligned}$$

因此，当 $p_{|ex(\hat{X})|} \geq \sigma$ 时，$|ex(\hat{X})| \leq \frac{1}{\sigma}$。 □

引理 4. 在所有元素的概率变为零或大于 *0.5* 之前，概率增加的次数受到 $O(n)$ 的限制。

证明. 首先证明概率增加有一个下界。假设存在概率大于 0 且小于等于 0.5 的元素。其中最小的概率为 $0.5 - \epsilon$，其中 $\epsilon \geq 0$。

然后根据引理3，

$$\begin{aligned} &\mathbb{E}[gain(\hat{X})] \geq E[gain(S(1))] = (1 - (0.5 - \epsilon)) * 1 \geq 0.5 \\ \Rightarrow &\Pi_i(1 - p_i)^{1-x_i} \geq \frac{0.5}{|ex(\hat{X})|} \geq 0.5\sigma \end{aligned}$$

因此，有概率增量

$$\begin{aligned}&\frac{p_i * \Pi_j(1-p_j)^{1-x_j}}{1-\Pi_j(1-p_j)^{1-x_j}}\\ \geq&\frac{\sigma}{\frac{1}{\Pi_j(1-p_j)^{1-x_j}}-1}\\ \geq&\frac{\sigma}{\frac{1}{0.5\sigma}-1}\end{aligned}$$

因此，如果至少有一个元素的正概率小于等于 0.5，则最小概率增量为 $\Delta = \frac{0.5\sigma^2}{1-0.5\sigma}$，这是一个常数。因此，在所有元素的概率变为零或大于 0.5 之前，概率增加的次数最多为 $\frac{0.5n}{\Delta} = O(n)$。 □

引理 5. *当 $p_i > 0.5 \vee p_i = 0$ 对于任意 i 时，$\mathbb{E}[S(1)] > \mathbb{E}[S(2)]$。*

证明. 设 p_a 为最小的正概率，p_b 为第二小的正概率，那么有 $\mathbb{E}[S(1)] = (1-p_a) > 2(1-p_b) * (1-p_a) = \mathbb{E}[S(2)]$。

因此，根据引理4，在所有剩余元素的概率大于 0.5 之前，需要进行 $O(n)$ 次概率增加。根据引理5，在所有剩余元素的概率大于 0.5 之后，算法每次只删除一个元素并调用反馈函数。由于删除一个元素后，根据失败反馈会将该元素的概率设置为 1，在所有剩余元素的概率大于 0.5 之后，最多会有 $O(n)$ 次概率增加。因此，概率增加的数量受到 $O(n)$ 的限制。

综上所述，可知原定理成立。 □

综合以上证明，在最坏情况下，PDD 调用反馈函数的次数渐进地受到 $O(n)$ 次的限制；此外，当输入满足单调性时，PDD 的输出具有极小性，当输入满足单调性和无歧义性时，PDD 的输出具有最小性。

4.4 实验设计

本章提出面向集合的概率模型，结合概率化差异调试框架，实例化为面向集合的概率化差异调试技术。如前所述，许多现有的差异调试技术都是基于 ddmin 算法的，且它们大多是面向特定领域的。具体而言，典型的特定领域的差异调试技术考虑了领域中的约束，并将 ddmin 应用于元素的集合，以确保不违反特定领域的约束。由于本章所提出技术的目标是改进 ddmin，在评估实验中希望了解在不同的应用领域中，PDD 相比 ddmin 的性能提升情况，即在将 ddmin 替换为 PDD 时，特定领域技术的性能是否有所改善。此外，本节还研究了 PDD 与 ACTIVECOARSEN[126] 的比较，后者是目前本文作者所了解到的唯一可以应用于差异调试的随机搜索算法。总之，本节的评估涉及以下研究问题（Research Question，简称 RQ）。

RQ1: PDD 在不同应用领域与 ddmin 算法相比表现如何?

RQ2: 参数如何影响 PDD 的效率和效果?

RQ3: PDD 与 ACTIVECOARSEN 相比表现如何?

4.4.1 实验设置

评估考虑了以下两个应用领域，在每个应用领域中，选择了基于 ddmin 的代表性差异调试技术作为目标技术。将目标技术中的 ddmin 组件替换为 PDD，并与原始目标技术使用 ddmin 进行性能比较。

- **树结构**. 树结构的代表性差异调试技术是 HDD[14]，通常应用于程序中抽象语法树（AST）可获取的情况。
- **C 程序**. C 程序的代表性调试工具是 CHISEL[13]，它依赖于 C 语言的语法和程序元素之间的依赖关系（如变量的定义-使用等依赖关系）。

考虑到现有的差异调试研究中对上述两种应用领域研究较为广泛，且有基于 ddmin 的代表性技术的公开实现，本章着重选取了这两个应用领域进行评估。

为了方便表述，将原始的集成 ddmin 算法的 HDD 技术称为 *d-HDD*，将把 HDD 技术中 ddmin 替换为 PDD 的版本称为 *p-HDD*。类似地，两个版本的 CHISEL 分别称为 *d-CHISEL* 和 *p-CHISEL*。在不引起歧义的前提下，也使用 *d-*版本和 *p-*版本。

4.4.2 数据集

本章的实验评估中主要选择了已有研究工作中原始技术的评估项目，以避免选择偏差等因素导致的不公平对比。具体而言，选择了以下项目。

- **树结构**. 在树结构领域使用了 30 个项目。使用了 20 个公开可用的项目作为与 HDD 和 CHISEL 技术进行比较的项目。这些项目是触发 GCC 和 Clang 中崩溃和编译错误的 C 语言程序，在差异调试过程中要满足的属性是在没有任何未定义行为的情况下重现报告的错误。由于这些项目都属于 C 程序的应用领域，添加了 10 个 XML 缩减任务以增加多样性。从公开可用的互联网 XML 文件存储库中爬取了超过 1000 个 XML 文件的语料库，过滤掉了 73 个无法解析的文件，并随机选择了 10 个 XML 文件作为项目。要满足的属性是至少保持在 XML 解析器 xmllint[127] 上的原始测试覆盖率。没有使用 HDD[14] 的原始工作中的基准数据集，因为它不是公开可用的。
- **C 程序**. 在 C 程序领域使用了 30 个项目。其中包括 CHISEL[13] 技术中使用到的 10 个 C 语言程序的项目（应用在嵌入式系统中进行精简的任务），以及与树结构相同的 20 个项目。对于这 10 个 CHISEL 中使用过的项目，要满足的属性是

成功编译、通过给定的测试用例，并保留特定的软件功能。对于这 20 个项目要满足的属性与树结构应用领域中使用的属性相同。

在评估中，总共使用了 40 个项目，其中有 20 个项目同时用于两个应用领域。根据已有的认知，本章评估中的项目数量超过了所有近期在顶级会议上关于差异调试的研究工作[1, 2, 4, 13, 14, 96, 97, 99-101]。充分利用了服务器的 16 个核心，整个评估过程平均每个核心耗时约 90 小时（总计 1,441 小时）。

4.4.3 实验过程

为了回答 RQ1，记录了每个项目的原始大小。然后，对每个项目应用了 d 版本和 p 版本的技术，并记录了产生结果的大小和处理时间。接下来，计算了每秒删除的标记数。此外，计算了配对样本 Wilcoxon 符号秩检验的 p 值，给出了产生结果的大小、每秒删除的标记数以及没有超时的项目的处理时间，以回答本章所提出的技术是否在效果和效率上相对于原始技术都取得了显著的改进。

为了回答 RQ2，调整了 PDD 中唯一使用的参数的值，即概率 σ 的初始值。由于这个实验非常耗时，对一部分项目进行了采样，使用采样的项目组成的子数据集。考虑到减少比率的多样性（即最小返回大小与原始大小的比率），对所有项目的减少比率进行了排序，并均匀选择了 14 个项目，它们的减少比率范围从 0.005 到 0.899。这 14 个项目按照减少比率的升序分别是 xml-10、xml-5、xml-6、xml-3、clang-27747、gcc-64990、clang-27137、gcc-65383、gcc-71626、chown-8.2、mkdir-5.2.1、date-8.21、sort-8.16 和 grep-2.19。在树结构应用领域运行了前半部分项目，剩余的项目在 C 程序应用领域运行。将概率 σ 的初始值分别更改为 0.01、0.05、0.1、0.15、0.2、0.25 和 0.3。对于每个设置，使用所有度量标准来衡量结果。由于一些项目超时，没有提供处理时间的结果，而是使用每秒删除的标记数作为效率的主要指标。在这个研究问题中，考虑到时间开销等因素，没有对所有项目进行实验。

为了回答 RQ3，选择了 ACTIVECOARSEN[126] 作为代表性的随机搜索算法，并使用了 ACTIVECOARSEN 的默认设置。然后，通过将 ddmin 替换为 ACTIVECOARSEN 来创建了两个版本的 HDD 和 CHISEL，分别称为 *a-HDD* 和 *a-CHISEL*。接下来，在所有领域的所有项目中比较了 a 版本和 p 版本。PDD、CHISEL 和 ACTIVECOARSEN 的结果也受到随机性的影响。为了减少随机性的影响，对所有受随机性影响的版本运行了 5 次，并计算了平均结果。选择了 5 作为重复实验的次数，因为每个项目和每种技术的 5 次运行结果的标准差已经小于其相应平均结果的 1%。在 RQ1 和 RQ3 中，将 PDD 中的 σ 设置为 0.1。

4.4.4 实现细节

本节介绍了两个应用领域的实现，以及整体评估的实验环境。

- **树结构**. 采用了最新的 Python 版本 HDD 实现[97, 128] 作为 d-HDD，并在此基础上实现了 p-HDD 和 a-HDD。
- **C 程序**. 采用了原作者在 C++ 中实现的 CHISEL[13] 作为 d-CHISEL，并在此基础上实现了 p-CHISEL 和 a-CHISEL。特别地，CHISEL 的实现包括自动死代码消除（DCE）和依赖分析（DA）的组件，通过使用三个命令行选项（即`-skip_local_dep`，`-skip_global_dep` 和`-skip_dce`）禁用了这些组件，原因如下。首先，DCE 是为 CHISEL 中的程序精简而设计的，在其它应用领域的大多数测试中都会失败。例如，由于无法访问的代码而触发的编译器错误。其次，发现在某些情况下，例如当函数调用作为参数传递时，DA 组件会产生错误的结果，而评估中使用的项目中存在这种情况。请注意，对于本评估中使用到的所有版本的 CHISEL，DCE 和 DA 都被禁用了。

在 HDD 和 CHISEL 技术的具体实现中，分别利用语法解析器，结合变量的定义-使用分析工具的语法解析器获取相对较为独立的元素构成的集合传递给 ddmin 算法进行处理。虽然 HDD 和 CHISEL 中涉及 ddmin 算法的具体实现不同（分别利用 Python 和 C++ 语言实现），但其本质上均是 ddmin 算法的思想。从本质上说，两种技术对于 ddmin 算法的实现，均涉及一个特定的类定义，该类中定义了 ddmin 算法的执行函数以及相关必要的类成员。为了应用本章提出的技术，作者分别使用 Python 语言和 C++ 语言，设计并实现了相应的类。通过将原技术实现中涉及 ddmin 算法的类替换为本章技术的类实现，本章技术能够结合 HDD 和 CHISEL 技术实现中的预处理模块，即提取传递给 ddmin 算法集合的模块，成功应用于差异调试中。类似地，作者还实现了 ACTIVECOARSEN 算法的类，并结合 HDD 和 CHISEL 形成相应的对比技术的具体实现。

此外，为了跑通本章实验验证中涉及的数据集中的项目，需要将项目中实现反馈函数的脚本中的相关路径，以及涉及到项目编译的脚本中的路径，设置为实验环境中的相应值。为了在验证过程中不影响服务器的本地环境，推荐使用 Docker 技术进行实验验证。可能影响本地环境的风险主要在于，来自评估 CHISEL 技术的数据集中的项目会在本地根目录下新建大量文件夹或者复制大量文件等操作，这主要是具体的项目决定的，比如 mkdir 项目会在根目录下建立大量新的文件夹。

本章的评估是在一台配备 16 核 32 线程的 Intel(R) Xeon(R) Gold 6130 CPU（3.7GHz）、128GB 内存和 Ubuntu Linux 16.04 操作系统的 Linux 服务器上进行的。

表 4.1　PDD 和 ddmin 之间的比较

总览	R_i	p-版本			d-版本								
		R_p	S_p	T_p	R_d	S_d	T_d	$\uparrow_R$	$p-value_R$	$\times S$	$p-value_S$	$\uparrow_T$	$p-value_T$
树结构	31,533	376	9	778	928	4	2,115	59.48%	0.0000	2.25	0.0000	63.22%	0.0015
C 程序	64,782	8,791	31	874	9,935	17	1,597	11.51%	0.0012	1.82	0.0000	45.27%	0.0000

在这个表格和本节其余的表格中，R 代表结果的大小；S 代表每秒删除的标记数；T 代表处理时间（以秒为单位）；R_i 代表输入的大小；p 代表 p 版本；d 代表 d 版本；↑表示改进，其中 $\uparrow_X = (X_d - X_p)/X_d$；$\times S$ 表示加速比，其中 $\times S = S_p/S_d$。在这个表格中，所有度量标准的统计数字都是几何平均数，并且处理时间的几何平均数仅在 p 版本和 d 版本都在时间限制内完成的项目中计算。

4.5　实验结果与分析

4.5.1　PDD 和 ddmin 的比较

表4.1展示了 p 版本和 d 版本在三个度量标准上的整体表现。从表4.1可以看出，p 版本在所有度量标准上的表现都优于 d 版本。平均而言，在树结构和 C 程序的应用领域中，p 版本每秒删除 5 和 14 个额外的标记，获得比 d 版本分别小 59.48% 和 11.51% 的结果。

在那些 p 版本和 d 版本都在时间限制内完成的项目中，p-HDD 和 p-CHISEL 分别使用了 63.22% 和 45.27% 更少的时间。所有的 p 值都是显著的（< 0.05）。然后研究了两个应用领域中每个项目的详细结果。表4.2展示了 p 版本和 d 版本之间的比较结果。从表4.2可以看出，在 60 个项目中，p 版本在 58 个项目上表现优于 d 版本。这里，定义 p 版本在某个项目上优于 d 版本，即如果 p 版本在任何一个指标上有更好的结果，并且在任何一个指标上没有更差的结果。只有在两个项目 mkdir-5.2.1 和 grep-2.19 上，p 版本的表现不如 d 版本。对这两个项目进行了分析，并发现为了保持目标属性，必须在返回结果中保留连续的子集。换句话说，如果知道索引 i 处的元素在 X^* 中，那么索引 $i-1$ 和 $i+1$ 处的元素也很可能在 X^* 中。这与 PDD 独立性假设相矛盾，因此 p-CHISEL 的性能并没有更好。然而，即使在独立性假设不成立的这两个项目上，p-CHISEL 的性能也只比 d-CHISEL 稍差一些。此外，考虑了表4.2中 $\uparrow_R$ 和 $\times S$ 列中大于 0 和 1 的值，并使用箱线图展示其分布，如图4.5(a)中的左右子图所示。在每个箱子中，将箱子分为两部分的线表示数据的中位数，箱子的两端分别表示上四分位数（Q3）和下四分位数（Q1），第一四分位数和第三四分位数之间的差异称为四分位距（IQR），极端线显示的是 Q3+1.5xIQR 到 Q1-1.5xIQR 之间的范围，异常值被省略。

RQ1: 平均而言，PDD 通过每秒删除 5 个和 14 个额外的标记，使得 HDD 和 CHISEL 的结果分别减小了 59.48% 和 11.51%。在两个版本都在时间限制内完成的项目中，PDD 将 HDD 和 CHISEL 的执行时间分别减少了 63.22% 和 45.27%。

表 4.2　PDD 和 ddmin 之间的比较：详细数据

D	项目名	p-版本			d-版本			↑$_R$	×S	↑$_T$
		R_p	T_p	S_p	R_d	T_d	S_d			
树结构	clang-22382	355	998	20.755	355	4,915	4.214	0.0%	4.9249	79.7%
	clang-22704	1,540	-	16.936	1,826	-	16.909	15.7%	1.0016	-
	clang-23309	1,327	-	3.456	13,782	-	2.302	90.4%	1.5009	-
	clang-23353	325	1,780	16.782	344	3,932	7.592	5.5%	2.2104	54.7%
	clang-25900	634	7,458	10.502	723	-	7.244	12.3%	1.4498	-
	clang-26760	397	-	19.369	624	-	19.348	36.4%	1.0011	-
	clang-27137	206	-	16.142	238	-	16.139	13.4%	1.0002	-
	clang-27747	227	4,256	40.792	315	-	16.067	27.9%	2.5389	-
	clang-31259	1,010	-	4.425	3,800	-	4.167	73.4%	1.0620	-
	gcc-59903	538	-	5.282	1,550	-	5.188	65.3%	1.0181	-
	gcc-60116	8,420	-	6.186	16,658	-	5.423	49.5%	1.1407	-
	gcc-61383	957	-	2.916	1,636	-	2.853	41.5%	1.0220	-
	gcc-61917	322	8,393	10.132	378	-	7.869	14.8%	1.2876	-
	gcc-64990	1,451	-	13.656	41,104	-	9.984	96.5%	1.3677	-
	gcc-65383	710	8,119	5.325	42,583	-	0.126	98.3%	42.3275	-
	gcc-66186	1,010	-	4.303	46,993	-	0.045	97.9%	95.1969	-
	gcc-66375	551	-	6.013	10,668	-	5.076	94.8%	1.1846	-
	gcc-70127	428	-	14.295	659	-	14.274	35.1%	1.0015	-
	gcc-70586	17,969	-	17.990	49,488	-	15.071	63.7%	1.1936	-
	gcc-71626	179	103	57.806	179	397	14.998	0.0%	3.8544	74.1%
	xml-1	30	486	11.152	170	3,731	1.415	82.4%	7.8804	87.0%
	xml-2	181	2,572	2.703	181	3,612	1.925	0.0%	1.4043	28.8%
	xml-3	236	2,508	3.462	236	3,378	2.571	0.0%	1.3469	25.8%
	xml-4	293	2,515	3.697	325	3,768	2.459	9.8%	1.5034	33.3%
	xml-5	46	509	11.493	230	2,765	2.049	80.0%	5.6086	81.6%
	xml-6	177	422	18.860	200	2,346	3.383	11.5%	5.5753	82.0%
	xml-7	54	382	12.421	54	807	5.880	0.0%	2.1126	52.7%
	xml-8	50	273	21.000	50	921	6.225	0.0%	3.3736	70.4%
	xml-9	179	2,707	1.768	195	3,240	1.472	8.2%	1.2009	16.5%
	xml-10	39	431	17.629	58	822	9.220	32.8%	1.9120	47.6%
C程序	mkdir-5.2.1	8,321	1,429	18.530	8,291	1,417	18.709	-0.4%	0.9905	-0.8%
	rm-8.4	7,427	3,232	11.458	7,427	4,192	8.834	0.0%	1.2970	22.9%
	chown-8.2	7,445	3,704	9.834	8,286	5,297	6.718	10.1%	1.4639	30.1%
	grep-2.19	114,754	-	1.197	113,155	-	1.345	-1.4%	0.8899	-
	bzip2-1.05	51,123	-	1.797	64,686	-	0.541	21.0%	3.3208	-
	sort-8.16	51,848	-	3.354	55,336	-	3.031	6.3%	1.1066	-
	gzip-1.2.4	16,548	-	2.720	30,074	-	1.468	45.0%	1.8531	-
	uniq-8.16	14,045	9,242	5.390	14,201	-	4.598	1.1%	1.1723	-
	date-8.21	20,219	9,920	3.349	33,541	-	1.843	39.7%	1.8175	-
	tar-1.14	58,374	-	9.715	130,328	-	3.053	55.2%	3.1825	-
	clang-22382	4,028	150	113.600	4,028	453	37.616	0.0%	3.0200	66.9%
	clang-22704	1,224	2,917	62.811	1,742	3,749	48.733	29.7%	1.2889	22.2%
	clang-23309	7,267	1,006	31.193	7,272	2,365	13.266	0.1%	2.3513	57.5%
	clang-23353	7,698	694	32.418	7,741	1,911	11.750	0.6%	2.7589	63.7%
	clang-25900	2,988	1,270	59.821	3,016	2,343	32.413	0.9%	1.8456	45.8%
	clang-26760	4,970	2,286	89.504	5,241	3,345	61.087	5.2%	1.4652	31.7%
	clang-27137	14,400	2,907	55.087	15,373	4,995	31.865	6.3%	1.7288	41.8%
	clang-27747	6,270	717	233.710	6,324	1,011	165.693	0.9%	1.4105	29.1%
	clang-31259	4,166	633	70.510	4,166	1,197	37.287	0.0%	1.8910	47.1%
	gcc-59903	6,602	737	69.171	7,614	1,342	37.233	13.3%	1.8578	45.1%
	gcc-60116	9,453	886	74.234	9,611	1,750	37.493	1.6%	1.9799	49.4%
	gcc-61383	9,702	1,109	20.511	9,932	1,265	17.800	2.3%	1.1523	12.3%
	gcc-61917	13,691	947	75.679	13,691	1,607	44.597	0.0%	1.6969	41.1%
	gcc-64990	6,970	882	160.953	7,532	1,891	74.775	7.5%	2.1525	53.4%
	gcc-65383	5,065	397	97.927	5,065	1,509	25.763	0.0%	3.8010	73.7%
	gcc-66186	6,447	622	65.971	6,447	1,196	34.309	0.0%	1.9228	48.0%
	gcc-66375	5,169	854	70.631	5,169	2,243	26.892	0.0%	2.6265	61.9%
	gcc-70127	12,452	1,346	105.768	12,452	2,043	69.684	0.0%	1.5178	34.1%
	gcc-70586	8,383	908	224.533	8,383	1,677	121.572	0.0%	1.8469	45.9%
	gcc-71626	639	14	392.429	639	33	166.485	0.0%	2.3571	57.6%

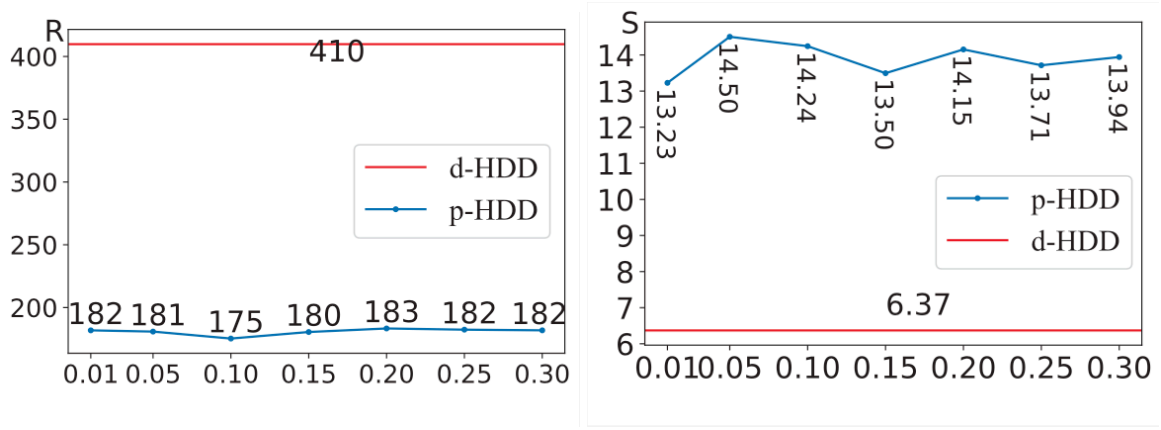
4.5.2　参数的影响

随后，基于在两个应用领域中选择的 14 个项目（如第4.4.1节所述），研究了 p-版本中唯一的参数，即每个元素的初始概率 σ。结果如图4.4所示。图4.4(a)和图4.4(b)中的左子图显示了生成大小的几何平均值，而右子图则描述了每秒删除的标记数的几何平均值。在每个子图中，蓝线标记了 PDD 的性能。此外，使用红线标记了使用 ddmin

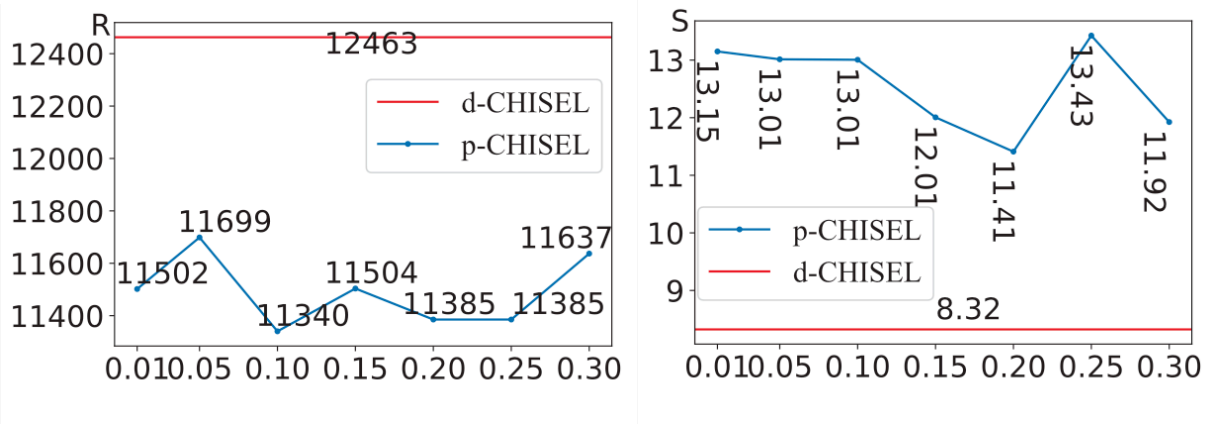
表 4.3　PDD 和 ACTIVECOARSEN 的对比

总览	R_i	p-版本			a-版本			$\uparrow_R$	$p-value_R$	$\times S$	$p-value_S$	$\uparrow_T$	$p-value_T$
		R_p	S_p	T_p	R_a	S_a	T_a						
树结构	31,533	376	9	778	910	6	1,887	58.68%	0.0000	1.5	0.0000	58.77%	0.0015
C 程序	64,782	8,971	31	874	12,597	9	2,788	27.03%	0.0000	3.33	0.0000	68.65%	0.0001

的原始技术的性能，以便进行清晰的比较。可以观察到，尽管不同的 σ 值会导致性能上的偏差，但在所有研究的 σ 值中，p-版本始终优于 d-版本。此外，不同 σ 值之间的性能差异明显小于 p-版本和 d-版本之间的差异。



(a) 在树结构领域中 σ 对结果的影响

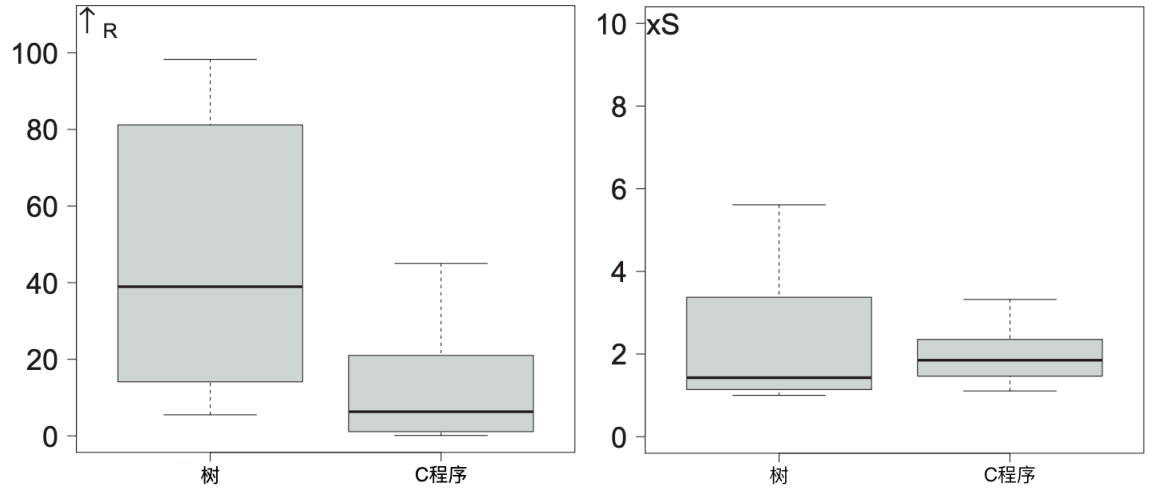


(b) 在 C 程序领域中 σ 对结果的影响

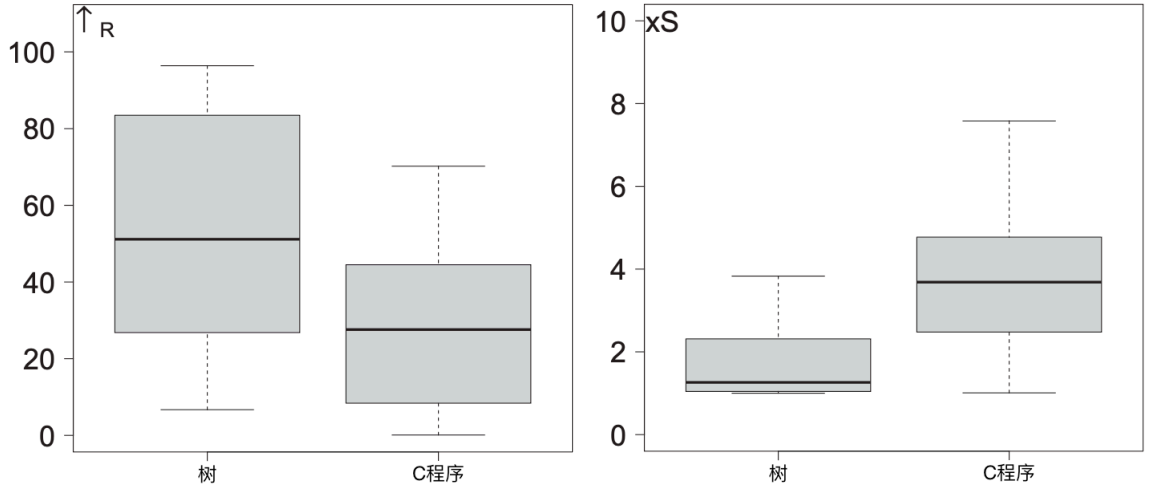
图 4.4　PDD 中参数 σ 对结果的影响

表 4.4 PDD 和 ACTIVECOARSEN 之间的比较：详细数据

D	项目名	p-版本			a-版本			↑$_R$	×S	↑$_T$
		R_p	Th_p	S_p	R_a	T_a	S_a			
树结构	clang-22382	355	998	20.755	452	4,250	4.851	21.5%	4.2786	76.5%
	clang-22704	1,540	-	16.936	9,342	-	16.213	83.5%	1.0446	-
	clang-23309	1,327	-	3.456	2,836	-	3.316	53.2%	1.0422	-
	clang-23353	325	1,780	16.782	394	7,578	3.933	17.5%	4.2672	76.5%
	clang-25900	634	7,458	10.502	1,248	-	7.196	49.2%	1.4595	-
	clang-26760	397	-	19.369	544	-	19.355	27.0%	1.0007	-
	clang-27137	206	-	16.142	1,280	-	16.042	83.9%	1.0062	-
	clang-27747	227	4,256	40.792	945	-	16.009	76.0%	2.5481	-
	clang-31259	1,010	-	4.425	1,380	-	4.391	26.8%	1.0078	-
	gcc-59903	538	-	5.282	1,461	-	5.196	63.2%	1.0165	-
	gcc-60116	8,420	-	6.186	10,880	-	5.958	22.6%	1.0382	-
	gcc-61383	957	-	2.916	6,652	-	2.389	85.6%	1.2208	-
	gcc-61917	322	8,393	10.132	8,969	-	7.073	96.4%	1.4325	-
	gcc-64990	1,451	-	13.656	35,791	-	10.476	95.9%	1.3035	-
	gcc-65383	710	8,119	5.325	2,201	-	3.865	67.7%	1.3777	-
	gcc-66186	1,010	-	4.303	1,716	-	4.237	41.1%	1.0154	-
	gcc-66375	551	-	6.013	7,095	-	5.407	92.2%	1.1121	-
	gcc-70127	428	-	14.295	629	-	14.277	32.0%	1.0013	-
	gcc-70586	17,969	-	17.990	27,172	-	17.138	33.9%	1.0497	-
	gcc-71626	179	103	57.806	204	728	8.144	12.3%	7.0978	85.9%
	xml-1	30	486	11.152	70	1,116	4.821	57.1%	2.3134	56.5%
	xml-2	181	2,572	2.703	263	3,408	2.016	31.2%	1.3409	24.5%
	xml-3	236	2,508	3.462	236	6,971	1.246	0.0%	2.7795	64.0%
	xml-4	293	2,515	3.697	314	4,288	2.163	6.7%	1.7088	41.3%
	xml-5	46	509	11.493	46	1,950	3.000	0.0%	3.8310	73.9%
	xml-6	177	422	18.860	377	500	15.518	53.1%	1.2154	15.6%
	xml-7	54	382	12.421	54	603	7.869	0.0%	1.5785	36.7%
	xml-8	50	273	21.000	50	1,669	3.435	0.0%	6.1135	83.6%
	xml-9	179	2,707	1.768	207	2,900	1.641	13.5%	1.0776	6.7%
	xml-10	39	431	17.629	263	505	14.602	85.2%	1.2073	14.7%
C 程序	mkdir-5.2.1	8,321	1,429	18.530	8,398	-	2.445	0.9%	7.5798	-
	rm-8.4	7,427	3,232	11.458	15,867	-	2.647	53.2%	4.3280	-
	chown-8.2	7,445	3,704	9.834	18,120	-	2.384	58.9%	4.1245	-
	grep-2.19	114,754	-	1.197	114,871	-	1.186	0.1%	1.0091	-
	bzip2-1.05	51,123	-	1.797	55,790	-	1.365	8.4%	1.3166	-
	sort-8.16	51,848	-	3.354	71,271	-	1.555	27.3%	2.1563	-
	gzip-1.2.4	16,548	-	2.720	35,799	-	0.938	53.8%	2.9003	-
	uniq-8.16	14,045	9,242	5.390	47,209	-	1.542	70.2%	3.4958	-
	date-8.21	20,219	9,920	3.349	37,061	-	1.517	45.4%	2.2080	-
	tar-1.14	58,374	-	9.715	132,849	-	2.819	56.1%	3.4460	-
	clang-22382	4,028	150	113.600	5,626	496	31.133	28.4%	3.6488	69.8%
	clang-22704	1,224	2,917	62.811	2,288	6,425	28.351	46.5%	2.2155	54.6%
	clang-23309	7,267	1,006	31.193	7,937	5,553	5.530	8.4%	5.6403	81.9%
	clang-23353	7,698	694	32.418	7,850	3,062	7.298	1.9%	4.4421	77.3%
	clang-25900	2,988	1,270	59.821	4,462	1,971	37.797	33.0%	1.5827	35.6%
	clang-26760	4,970	2,286	89.504	5,489	5,653	36.103	9.5%	2.4792	59.6%
	clang-27137	14,400	2,907	55.087	14,456	6,597	24.266	0.4%	2.2701	55.9%
	clang-27747	6,270	717	233.710	6,322	4,072	41.139	0.8%	5.6810	82.4%
	clang-31259	4,166	633	70.510	5,894	3,089	13.890	29.3%	5.0765	79.5%
	gcc-59903	6,602	737	69.171	9,292	4,378	11.030	28.9%	6.2712	83.2%
	gcc-60116	9,453	886	74.234	12,786	4,505	13.860	26.1%	5.3561	80.3%
	gcc-61383	9,702	1,109	20.511	9,884	4,093	5.513	1.8%	3.7205	72.9%
	gcc-61917	13,691	947	75.679	14,705	4,455	15.860	6.9%	4.7718	78.7%
	gcc-64990	6,970	882	160.953	9,940	2,842	48.906	29.9%	3.2911	69.0%
	gcc-65383	5,065	397	97.927	9,127	1,377	25.283	44.5%	3.8732	71.2%
	gcc-66186	6,447	622	65.971	8,990	2,749	14.002	28.3%	4.7116	77.4%
	gcc-66375	5,169	854	70.631	6,672	3,515	16.733	22.5%	4.2211	75.7%
	gcc-70127	12,452	1,346	105.768	13,725	4,407	32.015	9.3%	3.3037	69.5%
	gcc-70586	8,383	908	224.533	11,625	3,010	66.656	27.9%	3.3685	69.8%
	gcc-71626	639	14	392.429	699	88	61.750	8.6%	6.3551	84.1%



(a) 表4.2的数据分布情况



(b) 表4.4的数据分布情况

图 4.5　实验结果分布

RQ2：参数 σ 对 PDD 的性能影响较小，并且在测试的所有参数值中，PDD 的效率和效果稳定地优于 HDD 和 CHISEL 的效率和效果。

4.5.3　PDD 和 ACTIVECOARSEN 的对比

表4.3展示了在树结构和 C 程序应用领域中，p-版本和 a-版本在所有项目上的整体比较结果。从该表中可以看出，在树结构和 C 程序应用领域中，p-版本平均每秒删除 3 个和 21 个额外的标记，使得生成结果的大小分别比 a-版本小 58.68% 和 27.03%。在两个版本都在限时内完成的项目中，p-版本分别使用了 58.77% 和 68.65% 更少的时间。每个项目的详细比较结果可以在表4.4中找到，PDD 实现的改进 ($\uparrow_R$) 和加速度 ($\times S$) 的分布在图4.5(b)中被报告。

RQ3: 平均而言，在树和 C 程序的应用领域中，p-版本通过每秒删除 3 个和 22 个额外的标记，分别获得了 58.68% 和 27.03% 更小的结果，明显优于 a-版本。在两个领域中，如果两个版本都在时间限制内完成，p-版本分别使用了 58.77% 和 68.65% 更少的处理时间。

4.5.4 对有效性的威胁

对于内部有效性的威胁主要在于 p 版本的实现正确性和实验脚本的正确性。为了减少这种威胁，本文作者仔细审查了实现的代码。

对于外部有效性的威胁主要存在于项目和目标技术之中。关于本章研究中使用的项目，采用了现有研究工作中用于两个应用领域（即树结构和 C 程序）的项目。此外，为了增加树结构领域中的项目多样性，还从爬取的语料库中随机选择了 10 个 XML 文件，并对本章提出的技术进行了评估。在未来，将在更多项目上评估 PDD。关于目标技术，采用了树结构和 C 程序领域的两种代表性技术，即 HDD 和 CHISEL，如第4.4.1节所述。

构建有效性面临的威胁主要来自随机性。随机性可能会影响 p-版本、a-版本和 d-CHISEL 的性能。为了减少这种威胁，对每个项目分别运行了 5 次，并计算了平均结果，如第4.4.1节所示。

4.6 讨论与小结

作为差异调试的基本算法，ddmin 是由 Zeller 和 Hildebrandt 提出的，用于最小化导致失败的测试输入[4]。该算法已在2.2节和3.1节中进行了描述。此外，该团队还提出了称为 dd 技术的扩展版本，旨在获得通过测试输入和失败测试输入之间的最小差异，而不是最小的导致失败的测试输入[4]。随后，一些技术将差异调试应用于不同的领域特定结构。Misherghi 和 Su[14] 提出了 HDD，用于更有效地对树状数据进行差异调试，该技术已在第二节中进行了描述。受 HDD 的启发，还提出了现代化的 HDD[97]、支持过滤的的 HDD[99] 和 HDDr[100]，以进一步提高 HDD 的性能。例如，HDDr 是 HDD 的递归变体。Sun 等人[96] 提出了 Perses，它利用编程语言的形式语法来指导缩减，并始终生成语法上有效的子集。对于每次缩减的迭代，Perses 调用 ddmin 来修剪量化节点的解析树中的节点，并为常规节点提出替换策略。基于 Perses 实现的 CHISEL[13] 引入了依赖分析，以了解哪些元素需要一起删除。CHISEL 还改进了 ddmin，并构建了一个决策树模型，在缩减过程中修剪 ddmin 的固定的序列。与大多数现有技术将 ddmin 应用于不同领域的技术不同，本章的工作旨在提出面向集合的概率化差异调试技术，解决 ddmin 算法暴露的缺陷。与 ddmin 不同，结合 ProbDD 框架，PDD 通过概率模型来

指导差异调试过程，并根据反馈结果更新模型。本章的研究表明，通过将 ddmin 替换为 PDD，在不同的应用领域中，PDD 显著提高了基于 ddmin 构建的代表性技术的性能。在现有的技术中，CHISEL 还利用统计模型来改进 ddmin，因此与本章的工作密切相关。然而，CHISEL 仍然依赖于 ddmin 中预定义的尝试序列，并且仅使用统计模型来优先处理序列中的尝试。与之不同的是，PDD 直接根据学习到的分布选择元素。本章的实验评估结果表明，PDD 能够显著提高 CHISEL 的性能。

与现有差异调试算法按照预定义的顺序搜索不同，本章提出面向集合的概率模型，结合概率化差异调试框架，实例化了面向集合的概率化差异调试技术 PDD，利用该概率模型估计生成结果中每个元素被保留的概率。该模型的一个基本假设是，每个元素与要保留的属性是独立相关的。然而，由于目标领域中的结构约束，元素之间可能会相互依赖。例如，在树结构中，子节点的存在取决于其父节点的存在。在本章中，通过将 PDD 集成到现有技术中来确保这些结构约束，使得现有技术仅将 PDD 应用于不违反约束的子集。更直接的方法是在概率模型中直接建立这些约束。例如，在一个包含两个元素的树中，可以使用两个随机变量来表示父节点的概率和当父节点存在时子节点的条件概率。当父节点不存在时，不需要子节点的条件概率，因为知道概率为零。本文第五章介绍面向语法树的概率模型，考虑了上述问题。

总的来说，本章提出了一种面向集合的概率模型，并结合概率化差异调试框架，实例化为面向集合的概率化差异调试技术 PDD，该技术利用概率模型来估计每个元素在简化结果中被保留的概率。本章从理论上分析并证明了返回结果具有极小性或最小性的情形和最坏情况下的调用反馈函数的渐近次数。此外，本章在两个应用领域，即树结构和 C 程序中评估了 PDD。平均而言，在将 ddmin 替换为 PDD 后，HDD 和 CHISEL 分别在时间限制内产生了 59.48% 和 11.51% 更小的结果。在两个版本都在时间限制内完成的项目中，HDD 和 CHISEL 使用 PDD 的时间分别减少了 63.22% 和 45.27%。结果表明，面向集合的概率模型能够显著提升差异调试的效果和效率。

第五章　面向语法树的概率模型

差异调试技术发展的一个显著趋势是用于处理具有特定领域结构的数据，例如程序。然而，尽管有这些进展，特定领域的差异调试算法的效率和有效性仍然存在挑战。如 Sun 等人[96] 所强调的，许多针对程序输入的领域特定技术在差异调试过程中经常生成不符合语法结构的程序。由于生成的程序可能不是有效的或可执行的程序，这导致了大部分现有技术效率上的问题。由此可见，简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。

从实验评估的的结果来看，本文作者在第四章中实例化的面向集合的概率化差异调试技术 PDD 可被认为是该领域的最先进的差异调试算法之一。PDD 通过面向集合的概率模型来估计每个元素在结果输出中保留的可能性。在每次迭代中，利用概率模型提供的信息，PDD 指定具有最大的期望收益的集合。然后，调用反馈函数检查该集合是否满足规定的性质，并根据反馈结果更新概率模型。与 ddmin 相比，后者遵循固定的顺序从输入集合中尝试删除元素，PDD 根据历史反馈结果的来指导差异调试过程。通过将 ddmin 算法替换为 PDD，原先依赖 ddmin 的技术可以获得更好的性能。在第四章中报告的实验结果展示了显著的时间节省和输入大小的减少。

同时，领域特定的差异调试技术也有了显著的进展，尤其是在程序领域。这些技术，如 Perses[96]，重点设计基于抽象语法树的转换。这些基于抽象语法树的转换用于确保每一轮迭代中产生对象的语法正确性，并为进一步缩减结果大小和提高效率提供了可能。本文作者观察到，像 Perses 这样的领域特定技术[96]，尽管它们很有效，但也依赖于固定的尝试序列，将定义的基于抽象语法树的转换应用于原始程序。此外，像 PDD 这样的概率方法假设元素之间是独立的，这可能限制了它们捕捉语法关系的性能。为了克服这些限制，本章提出了面向语法树的概率模型，并结合概率化差异调试框架，实例化为面向语法树的概率化差异调试技术 T-PDD。T-PDD 利用从抽象语法树构建的贝叶斯网络，既利用了现有的反馈结果，又更加精准地捕获了元素之间的语法关系，以估计每个元素在结果中保留的概率。在每次迭代中，T-PDD 根据概率模型选择一个最大化后期望收益的元素子集。然后，调用反馈函数检查该子集是否满足所规定的性质。此外，T-PDD 根据反馈结果更新概率模型，进一步精细化概率模型对元素保留在最终结果中概率的估计。从抽象语法树构建概率模型也使 T-PDD 能够有效地执行概率推断，因为抽象语法树的结构复杂性相对简单。因此，本章预计 T-PDD 将在减少生成结果的大小和提高效率方面能够超越现有的领域特定的差异调试技术。

```c
int main() {
  int* g_1 = safe_arr(config_1(),config_2());
  while(1) {
    g_1[0] = safe_val(config());
  }
  return 0;
}
```

图 5.1　示例程序片段

本章使用了包含 107 个项目的数据集对 T-PDD 进行了全面评估。根据现有的知识，本章所使用的评估数据集规模比所有在软件工程顶级会议上发表的关于差异调试的论文中使用的数据集都大[13, 96, 111, 129]。这个数据集包括了从以前的工作中[96, 129] 选出的 20 个被广泛采用的 C 语言项目。此外，本章构建了一个包含 C 和 Rust 程序的 87 个项目的语法多样化的数据集，用于评估 T-PDD。这个数据集能够更全面地评估 T-PDD 的有效性和效率。本章的实验验证结果表明，T-PDD 显著提高了 Perses 这种现代差异调试技术的效率。例如，在一个典型的案例中，Perses 花了高达 7 小时的时间来减少一个有 371,277 个标记的 C 程序，而 T-PDD 只用了 1.5 小时就完成了同样规模的程序缩减。平均地，T-PDD 达到了与 Perses 相同大小的结果，同时减少了 26.95% 的时间消耗。

总之，本章的主要贡献如下：

- 本章提出了面向语法树的概率模型，并结合概率化差异调试框架，实例化为面向语法树的概率化差异调试技术 T-PDD。
- 相较于已有工作，本章额外构建了一个专门为评估本章所提出的技术而设计的包含 87 个项目的数据集。这个数据集能够更加全面地评估 T-PDD 等差异调试技术的有效性和效率。
- 本章使用现有的数据集和额外构建的数据集对 T-PDD 进行了广泛的评估，总共包括了 107 个项目。实验验证结果表明，T-PDD 显著提高了针对上下文无关文法的最先进的差异调试技术的效率，同时也产生了在最好情况下小 3.4 倍的结果。

5.1　启发性示例

本节使用一个程序简化的例子来展示领域特定的差异调试技术的功能。考虑到 Perses 是为通用目的设计的上下文无关语言的最先进的技术，本节在可用的技术中选

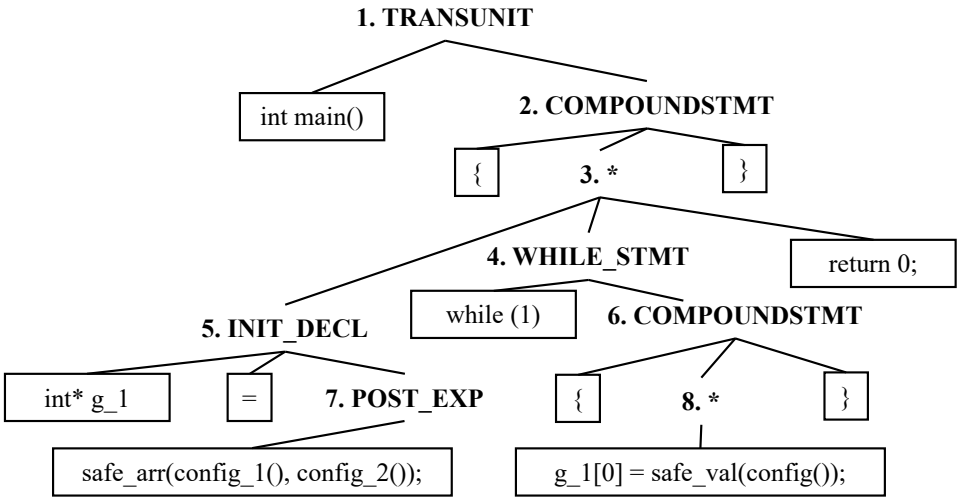


图 5.2　图5.1中程序片段对应的抽象语法树，清晰起见做了简化处理

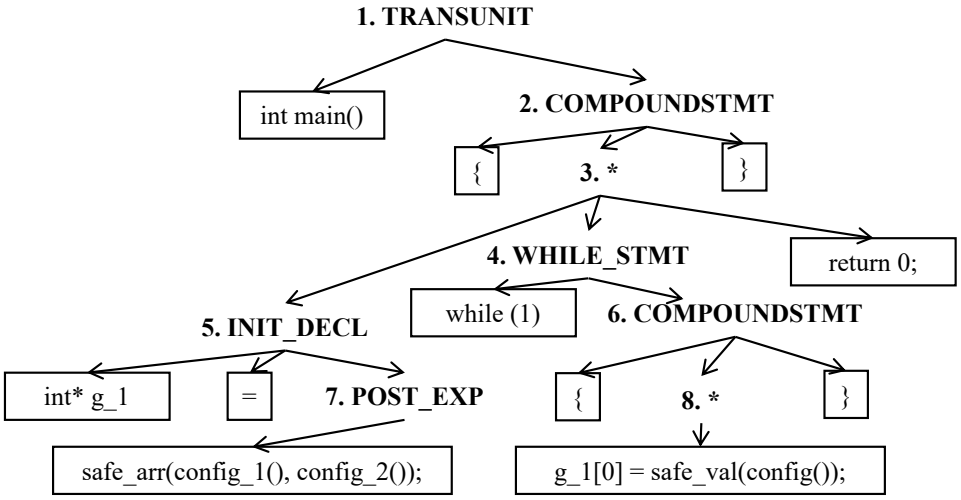


图 5.3　图5.2中抽象语法树对应的概率模型，清晰起见省略了随机变量的标注

择 Perses 作为代表技术。代码片段展示在图5.1中，展示了需要简化的函数。为了清晰起见，本节示例省略了被调用函数的定义。在这种情况下，本节假设 `while` 循环有可能触发编译器崩溃的错误。此外，本节假设任何包含 `while` 关键字的合法程序都会触发上述错误。

在这个例子中，Perses 在图5.2中描述的抽象语法树上操作。Perses 的算法首先初始化一个优先队列（Q）来维护等待处理的节点。在每次迭代中，Perses 选择 Q 中标记数量最多的节点进行进一步处理。Perses 还区分 Kleene-Star 节点和常规节点。Kleene-star 节点有一个可变长度的子节点列表，例如函数定义中的参数列表。当遇到这样的节点时，Perses 应用 ddmin 算法尝试删除其子节点。在本章中，像 Kleene-Star 这样的节点被称为*数量节点*，其它的被称为*常规节点*。对于常规节点，Perses 首先尝试删除节点本

	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	
	1.0	1.0	0.5	0.5	0.5	1.0	1.0	0.5	
1	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	$\varphi(X^1) = F$
	1.0	1.0	**0.89**	0.5	0.5	1.0	1.0	0.5	
2	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	$\varphi(X^2) = F$
	1.0	1.0	0.89	**1.0**	0.5	1.0	1.0	0.5	
3	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	$\varphi(X^3) = T$
	1.0	1.0	0.89	1.0	0.5	1.0	1.0	**0.0**	
4	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	$\varphi(X^4) = T$
	1.0	1.0	0.89	1.0	**0.0**	1.0	0.0	0.0	
5	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	$\varphi(X^5) = F$
	1.0	1.0	**1.0**	1.0	0.0	1.0	0.0	0.0	

图 5.4　本章提出算法的步骤

身。如果在删除节点后反馈失败，Perses 继续寻找可以替代已处理节点且不违反语法的任何子节点。处理完一个节点后，其子节点被添加到 Q 中等待未来的处理。当 Q 变为空时，开始新的迭代。该过程被重复直到在单次迭代中没有可以被删除的节点为止。

按照算法规定的步骤，Perses 尝试按照序列 `(1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 4.WHILE_STMT, 5.INIT_DECL, 6.COMPOUNDSTMT, 8.*, 7.POST_EXP, ......)` 中的节点顺序，在第一次迭代中删除图5.2中的节点。当处理常规节点时，即除了 `3.*` 和 `8.*` 外图5.2中的所有节点，Perses 首先尝试删除节点本身。如果在删除节点后反馈失败，Perses 继续寻找可以替代已处理节点而不违反语法的任何子节点。在本节的示例中，节点 `2.COMPOUNDSTMT` 和 `4.WHILE_STMT` 可以分别被它们的子节点 `6.COMPOUNDSTMT` 替代。当处理数量节点，即 `3.*` 和 `8.*` 时，Perses 使用 ddmin 最小化由其子节点组成的集合。这样，在处理预定义序列中的所有前置节点后，`8.*` 的子节点可以在这次迭代中被成功删除。在第二次迭代中，Perses 按照 `(1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 5.INIT_DECL, 4.WHILE_STMT, ......)` 的顺序尝试删除节点。在此次迭代中，节点 `5.INIT_DECL` 在节点 `1.TRANSUNIT`、`2.COMPOUNDSTMT` 和 *3.** 被处理后可以被成功删除。在最后一次迭代中，Perses 按照 `(1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 4.WHILE_STMT, ......)` 的顺序尝试删除节点。由于在此次迭代中没有更多的节点可以被删除，算法 Perses 终止迭代。

从上述过程中可以看出，Perses 坚持按照预定的顺序进行节点删除，并且在这一过程中历史的反馈结果没有被利用。例如，节点 `4.WHILE_STMT` 在三次调用 ddmin 过程中都被尝试删除，但都没有成功。如果上述的简化过程可以从历史的反馈结果中学习，正如在图5.4中展示的那样，节点 `4.WHILE_STMT` 只会被尝试删除两次，进而其父节点被保留在最终结果中的概率变为 1.0。虽然 PDD 利用一个概率模型，基于历史反馈结

果来确定下一次迭代中调用反馈函数的子集，但它并没有解决上述的缺陷。

一方面，PDD 假设元素之间是独立的，将其直接应用于抽象语法树叶子节点时可能会有问题。这种技术将抽象语法树由叶子节点组成的集合作为输入，可能会删除某些节点，导致产生语法错误的程序。例如，PDD 使用两个独立的伯努利随机变量来表示 `2.COMPOUNDSTMT` 的子节点，即节点 `{`和节点`}` 是否应该包括在最终结果中。但是，如果节点 `{`不应该在最终结果中保留，那么节点`}` 也不应该保留。这表明这两个节点是相互依赖的，它们应该一起被删除或保留。这突出了 PDD 在处理抽象语法树结构中相互依赖的节点时的一个缺陷。

另一方面，Perses 使用 ddmin 算法尝试删除如 `3.*` 这样的数量节点的子节点。在每次迭代中都重复应用 ddmin 算法，尝试不同的子节点的删除组合，直到最终尝试删除每一个独立的子节点。作为一个比 ddmin 算法更高效的算法，PDD 可被用来替换 ddmin 算法来优化 Perses。虽然 PDD 在利用概率模型指导搜索过程上表现出潜力，与 ddmin 类似，它也在每次迭代中被重复应用来处理 `3.*` 节点。总之，无论采用 ddmin 还是 PDD，Perses 在不同的迭代中都根据其预定义的删除尝试序列删除抽象语法树上的节点。

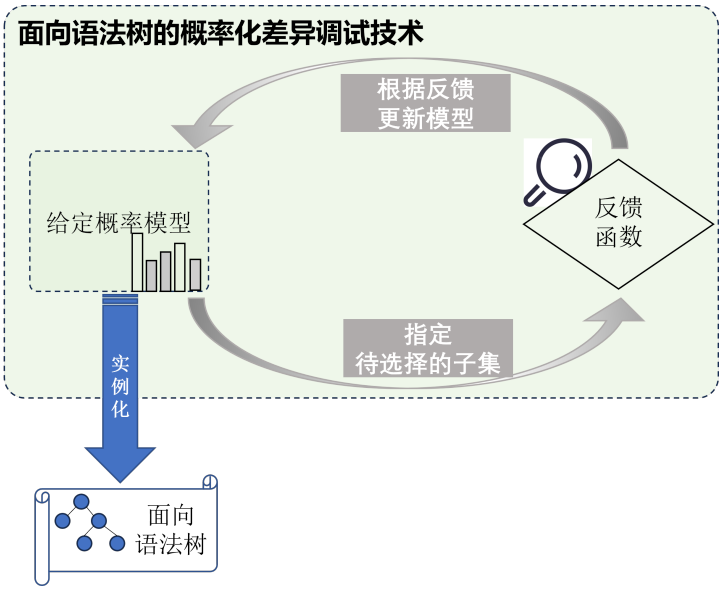


图 5.5　面向语法树的概率化差异调试技术

相比之下，本章提出的技术 T-PDD（如图5.5所示）通过直接从抽象语法树中构建概率模型。抽象语法树的结构简单性使 T-PDD 能够高效地执行概率推断并有效地指导搜索过程。通过利用这种轻量级的概率模型，T-PDD 能够实现更高效的差异调试。它利用已有的反馈结果并捕获元素之间的关系，从而提高了效率和效果。

按照后续模型定义中的步骤，图5.3描述了从图5.2中显示的抽象语法树构建的概率

模型。图5.3中的每个节点代表一个随机变量，该变量表示在其父节点给定的情况下，图5.2中的相应节点是否包含在最终结果中。为了增强可读性，本节在此图中省略了随机变量的标注。图5.4演示了本章提出算法在简化图5.1中程序的步骤。标签 n_1 到 n_8 表示在图5.2中，节点按从 1 到 8 顺序编号。在图5.4中，每个奇数行代表调用一次反馈函数，被删除的元素用颜色较深的单元格描绘。每个奇数行的最后一个单元格显示相应反馈的结果。每个偶数行显示了每个节点在其父节点被保留在最终结果中的情况下，最终被保留在最终结果中的概率。

对于每个数量节点及其子节点，包含一个条件概率表（CPT）来表示当父节点在最终结果中时，该节点在最终结果中的概率。这些 CPTs 根据第5.2.1.3节中描述的步骤进行初始化。这个概率模型捕捉了节点之间的语法关系，并在程序简化过程中确保语法的有效性，从而将 T-PDD 与 PDD 区分开来。在算法的每个步骤中，T-PDD 选择并删除具有最高预期收益的节点及其后代节点，如第5.2.2节所述。T-PDD 首先删除 n_3 及其后代元素 n_4 到 n_8。然而，反馈失败，导致对 n_3 的 CPT 进行更新。随后，T-PDD 继续在第二步删除 $\{n_4, n_6, n_8\}$，该步骤未通过反馈函数，需要对相应的 CPT 进行更新。在第三和第四步中，T-PDD 分别删除 $\{n_8\}$ 和 $\{n_5, n_7\}$ 并调用反馈函数。在这两种情况下，反馈通过，将 n_8 和 n_5 被保留在最终结果中的概率（假设它们的父节点被保留在最终结果中）设置为零。在第五次反馈函数调用中，T-PDD 关注 $\{n_1, n_2\}$ 的预期收益，这是唯一剩下需要调用反馈函数的集合。然而，它也未通过反馈函数。在相应地更新 CPT 之后，CPT 中的每个项要么是 0，要么是 1，促使 T-PDD 开始应用转换模板来进一步地简化程序。在这个特定的例子中，只有两个模板被应用，即用 `6.COMPOUNDSTMT` 替换 `2.COMPOUNDSTMT` 和用 `6.COMPOUNDSTMT` 替换 `4.WHILE_STMT`。然而，这两个转换都未通过反馈函数。最后，简化过程终止并返回简化后的程序。请注意，T-PDD 不定义任何转换模板。作为对 T-PDD 应用于 Perses 的示例，在 T-PDD 收敛后处理示例程序时遍历了 Perses 中定义的转换模板。

本章所提出的技术 T-PDD 在这个例子中展示了一种高效的程序简化流程。通过优先删除具有最大期望收益的元素集合，T-PDD 在差异调试过程中尽可能少的调用反馈函数。相比较之下，领域特定的差异调试技术如 Perses，按照固定的序列尝试删除元素，并需要应用 3 次 ddmin 算法来处理上述示例中的单个节点 `3.*`。尽管在第四章中，面向集合的概率化差异调试技术 PDD 相对于 ddmin 显示出更先进的性能，但它无法捕获抽象语法树中节点之间的语法关系。即使在 Perses 中用 PDD 替代 ddmin，仍然需要三次应用 PDD 算法来处理节点 `3.*`。此外，无论使用 ddmin 还是 PDD，Perses 都在不同迭代中按照固定的顺序尝试删除节点。这种差异突显了 T-PDD 在减少简化程序所需的调用反馈函数次数方面的优势。

5.2　面向语法树的概率模型

在本节中，将描述本章所提出的模型的关键组成部分，包括贝叶斯网络、待选择集合的指定以及基于历史反馈结果的更新过程。

i) 贝叶斯网络：本章所提出的技术基于抽象语法树构建了一个贝叶斯网络，以指导差异调试过程。该贝叶斯网络捕捉了元素之间的语法关系，并估计了每个元素在最终结果中被保留的概率。

ii) 指定待选择的集合：在概率化差异调试框架中定义了选择下一次调用反馈函数的集合的标准，即期望收益。该期望收益根据基于贝叶斯网络估计的概率计算得出。该部分描述了选择具有最大期望收益的子集，同时满足规定性质的方法。

iii) 基于历史反馈结果的更新过程：贝叶斯网络根据差异调试过程中获得的历史反馈结果进行更新。每个反馈结果提供了关于删除或保留某些元素的影响的信息。通过分析这些结果，更新贝叶斯网络以更新元素概率的估计。

5.2.1　模型定义

5.2.1.1　模型

本节定义的概率模型本质上是一种树状贝叶斯网络，该网络被定义为一个有向无环图（DAG），表示为 $G = (\Theta, E)$。这里，Θ 表示图中的节点集合，而 E 表示连接节点的有向边的集合。在该贝叶斯网络中，每个节点 $\theta_i \in \Theta$ 对应一个伯努利随机变量，表示相应的元素 n_i 是否包含在 T^* 中。因此，θ_i 和 n_i 是一一对应的。对于叶节点 $n_i \in N_l$，$\theta_i = 1$ 表示所代表的叶节点包含在 T^* 中。对于非叶节点 $n_i \in N_{nl}$，$\theta_i = 0$ 表示该非叶节点不包含在 T^* 中，因此其对应子树中的所有节点都被删除。边集 E 是从抽象语法树中导出的。如果在 T 中存在从 n_i 到 n_j 的边，则在 E 中存在从 θ_i 到 θ_j 的边。

设 S_{n_i} 表示 T 中 n_i 的父节点，S_{θ_i} 表示 G 中 θ_i 的父节点。根据贝叶斯网络的定义，为每个节点分配一个条件概率表（CPT）$P(\theta_i|S_{\theta_i})$，指定在其父节点条件下 θ_i 的概率。由于 $S_{\theta_i} = 0$ 意味着相应子树中的所有节点都不在 T^* 中，有 $P(\theta_i = 1|S_{\theta_i} = 0) = 0$。因此，$P(\theta_i|S_{\theta_i})$ 可以用一个单一的概率值 $p(\theta_i = 1|S_{\theta_i} = 1)$ 表示，记为 p_{θ_i}。

通过提出的贝叶斯网络模型，图 G 的联合分布可定义为条件概率分布的乘积。

$$P(\theta_1, ..., \theta_k) = \prod_{i=1}^{k} P(\theta_i|S_{\theta_i}) \tag{5.1}$$

如果 θ_i 是 G 的根节点，则定义 $P(\theta_i|S_{\theta_i}) = P(\theta_i)$。设 Θ_I 是包括 θ_i 及其所有祖先（从根节点到 θ_i 的路径）的集合，第 i 个变量的边际概率为：

$$P(\theta_i = 1) = \prod_{\theta \in \Theta_I} p_\theta \tag{5.2}$$

5.2.1.2 模型推断

在本章提出的技术中，贝叶斯网络被用来预测集合 X 是否通过反馈函数的概率。其核心假设是，只有当 X 包含最优子集 X^* 中的所有元素时，X 才能通过反馈函数。根据这个假设，X 通过反馈函数的概率等于删除子树 T_d 不包含 T^* 中任何叶节点的概率。给定一个删除子树 T_d，可以通过包括所有祖先节点直到根节点来扩展它为 T_{d-EX}，相应的概率子图可以表示为 $G_{d-EX} \subset G$。删除子树 T_d 不包含 T^* 中任何叶节点的概率可以递归地定义为 $Q(\theta_R)$，其中 θ_R 是 G_{d-EX}（也是 G）的根节点，Q 的定义如下。

$$Q(\theta) = \begin{cases} (1-p_\theta), \theta \text{ 为叶节点} \\ (1-p_\theta) + p_\theta \cdot \prod\limits_{\theta_c \in \Theta(G_{d-EX}), S_{\theta_c}=\theta} Q(\theta_c), \text{其他情况} \end{cases} \tag{5.3}$$

根据公式 5.3，$Q(\theta)$ 的计算方式如下：当 θ 是叶节点时，$Q(\theta)$ 等于 $(1-p_\theta)$。否则，$Q(\theta)$ 等于 $(1-p_\theta)$ 加上 p_θ 乘以所有满足 $S_{\theta_c}=\theta$ 的 $\theta_c \in \Theta(G_{d-EX})$ 的 $Q(\theta_c)$ 的乘积。在这个方程中，$Q(\theta)$ 代表着当 θ 的所有祖先节点都为 1 时，θ 的子树中所有节点都为 0 的概率。而 p_θ 则表示 $P(\theta=1|S_\theta=1)$，正如之前所描述的那样。

在 T_{d-EX} 中的每个节点上，可以以线性时间计算 $Q(\theta_R)$，因为函数 Q 只被调用一次。这使得对通过反馈函数的子集的概率进行高效地预测。

5.2.1.3 模型的先验分布

贝叶斯网络中每个节点被赋予的先验概率用于在差异化过程中融入用户对输入语法结构或输入中元素间关系的假设和知识。设 n_R 表示 T 的根节点的节点，定义 $P(\theta_R=1)=1.0$，表示根节点始终保留在最优子集中。对于 T 中的其他节点 n_i，根据以下情况为 G 中对应节点定义先验概率：

i) 如果 S_{n_i} 是一个数量节点，令 $p_{\theta_i}=\sigma$，其中 σ 是技术中的超参数。这个超参数可以调整以控制将节点 n_i 保留在最优子集中的概率。通过调整 σ，可以影响 n_i 的子节点在差异调试过程中被保留或删除的可能性。

ii) 如果 S_{n_i} 是一个常规节点，令 $p_{\theta_i}=1.0$。在这种情况下，如果父节点 S_{n_i} 存在于最优子集 T^* 中，节点 n_i 也必须被保留。

5.2.2 待选择集合的指定

回顾第3.3.2.1节中定义的收益：

$$gain(X, X_T) = \begin{cases} |ex(X, X_T)| & \phi(X) = T \\ 0 & \phi(X) = F \end{cases}$$

当通过反馈函数时（$\phi(X') = T$），收益等于集合大小的减少量；当反馈函数失败时（$\phi(X') = F$），收益为零。在不会引起混淆的前提下，为了简化表示，本章使用 $gain(T_d)$ 代替 $gain(X', X)$、用 $ex(T_d)$ 表示 $ex(X, X_T)$，因为在选择过程中 X 保持不变，而 T_d 逐个对应于 X'。这个收益函数有助于量化反馈结果对集合大小的影响，并在差异调试过程中指导选择最优子集。

通过考虑删除子树 T_d 中叶节点的数量和反馈通过 ($\phi(X') = T$) 的概率，可以计算收益函数 $gain(T_d)$ 的期望。该期望可以表示为：

$$E(gain(T_d)) = |T_d|_l \cdot P(\phi(X') = T) \tag{5.4}$$

在这个方程中，$|T_d|_l$ 代表被删除子树 T_d 中的叶节点数量。有关如何计算 $P(\phi(X') = T)$ 的详细信息，请参阅第5.2.1.2节。

选择的目标是找到一个被删除的子树 T_d，以最大化 $E(gain(T_d))$。通过枚举 T 中所有剩余的子树，找到产生最大的期望值时所删除的子树，删掉这颗子树以得到用于下一次调用反馈函数的 X'。

5.2.3　模型更新

结合本章提出的面向语法树的概率模型，以及第3.3.3节中介绍的计算后验概率的方法，本节介绍模型更新的方法。

更新概率的步骤如下：在选择调用反馈函数的集合 X 之后，获得了它是否通过反馈函数的结果。根据这个结果，更新贝叶斯网络中的条件概率表（CPT）。具体而言，是更新变量 $\theta_i \in G_{d-EX}$ 的后验概率，如第5.2.1节所述。

一方面，如果反馈函数通过，对于 $\theta_i \in G_{d-EX}$，公式3.6可被改写为：

$$P(\theta_i = 1|\phi(X) = T) = \frac{P(\theta_i = 1) \cdot P(\phi(X) = T|\theta_i = 1)}{P(\phi(X) = T)} \tag{5.5}$$

这里，$P(\theta_i = 1)$ 表示先验边缘概率，可以使用公式5.2 计算得到。$P(\phi(X) = T)$ 是反馈函数通过的概率，可以通过计算公式5.3 中的 $Q(\theta_R)$ 得到。$P(\phi(X) = T|\theta_i = 1)$ 可以通过将 θ_i 和其所有祖先设置为 1 后计算 $Q(\theta_R)$ 得到，此过程中无需重新计算其它未设置为 1 的 θ 的 Q 值，因为在计算第5.2.2 节中的公式5.4 时已经计算了这些值。最后，后验条件概率 $\hat{p_{\theta_i}}$ 计算如下：$\hat{p_{\theta_i}} = \frac{P(\theta_i=1|\phi(X)=T)}{P(S_{\theta_i}=1|\phi(X)=T)}$。

另一方面，如果反馈函数失败，情况类似。公式3.7可被改写为：

$$P(\theta_i = 1|\phi(X) = F) = \frac{P(\theta_i=1)\cdot(1-P(\phi(X)=T|\theta_i=1))}{1-P(\phi(X)=T)} \tag{5.6}$$

计算过程与反馈函数通过时相同。

在输入大小为 n 的情况下，面向语法树的概率化差异调试技术调用反馈函数的渐

近次数在最坏情况下受 O(n) 的约束。在处理已删除的子树时，如果删除后反馈函数通过，则该子树对应的元素将从集合中删除。相反，如果反馈函数反馈失败，沿着从根节点到子树根的路径的节点的每个 p_θ 都将被设置为 1，从而确保该子树不会被考虑再次删除。因此，将进行最多 n 次尝试。更新过程按照拓扑顺序进行，以实现线性时间的高效计算。然而，在实践中，需要更新的条件概率表（CPT）的数量庞大，这给计算时间带来了显著的挑战，促使寻求提高效率的方法。理论上，在单个失败的反馈之后，从根节点到被删除子树根节点的路径上的节点的概率理想情况下应该为 1.0。这意味着这些节点在最终结果中被保留的概率是 100%。然而，现实场景往往由于语义依赖等因素与理想情况有所偏差，而本章中提出的面向语法树的模型模型，无法捕捉到这些偏差。为了解决这种差异，本节采用了一种优化策略，更新只删除的子树根节点的 CPT。这样可以使本应在最终结果中保留的元素在未来反馈函数的调用中有可能会被删除，从而得到更小的结果大小。具体而言，对于 $\theta_i \in G_{d-EX}$，当反馈函数通过时，令 $\hat{p_{\theta_i}} = 0$。相反，当反馈函数失败时，令 $\hat{p_{\theta_i}} = \frac{p_{\theta_i}}{P(\phi(X)=F)}$。此外，如果这种调整后的 $\hat{p_{\theta_i}}$ 超过 1.0，则将其设置为 1.0。尽管这种优化可能会稍微影响模型的准确性，但实验结果表明其有效性。未来可以进一步研究其它优化方法，以提高更新过程的效率同时保持结果的准确性。

在本章中，提出了面向语法树的概率模型，结合概率化差异调试框架，实例化为面向语法树的概率化差异调试技术 T-PDD。本章所提出技术的核心步骤包括从抽象语法树构建贝叶斯网络，基于产生的期望收益选择集合并调用反馈函数，并在每次反馈后更新网络。这个迭代过程会一直进行，直到每个 p_θ 要么为 0 或 1，要么当前的期望收益低于预定义的阈值（设置为 1.0）。与领域特定的差异调试技术相比，在 T-PDD 中，转换模板是在上述步骤收敛后应用的。终止条件是当所有模板都被应用一次时，此时的结果被返回作为最终结果。

5.3 实验设计

在本节的评估中，将本章提出的技术与编译器调试领域中广泛使用的领域特定的差异调试技术 Perses 进行比较。Perses 建立在 ddmin 算法的基础上，是针对上下文无关文法的最先进的差异调试技术。本节的目标是通过与 Perses 进行比较来评估本章提出的技术的性能。鉴于面向集合的概率化差异调试技术 PDD 相对于 ddmin 算法表现出更好的性能，本节假设将 PDD 集成到 Perses 中可能会增强其性能。因此，本节还通过将 PDD 技术集成进 Perses 得到 Perses 的变体来评估本章提出的技术。在本节的评估中，将这个变体称为 p-Perses。此外，本节还研究了本章所提技术中参数设置的影响。这个分析可以帮助了解不同的参数配置如何影响本章所提技术在程序缩减任务中的性

能和效果。通过这些比较和参数评估，旨在展示本章提出的技术的效果和效率，展示其改进领域特定差异调试技术的潜力。具体而言，本节的评估旨在回答以下研究问题：

- **RQ1.** 在处理时间和结果大小方面，T-PDD 与 Perses 相比如何？
- **RQ2.** 在处理时间和结果大小方面，T-PDD 与将 PDD 技术集成其中的 Perses 变体（p-Perses）相比如何？
- **RQ3.** T-PDD 中的参数对实验结果有何影响？

5.3.1 实验设置

由于面向语法树的概率化差异调试技术主要针对的是程序，评估考虑了编译器调试领域中简化测试用例，并选择了针对上下文无关文法的最先进的差异调试工具 Perses 作为对比技术。具体地，评估考虑到了 C 语言编译器调试领域和 Rust 语言编译器调试领域中简化测试用例。此外，考虑到面向集合的概率化差异调试技术 PDD 相对于 ddmin 算法表现出更好的性能，假设 PDD 集成到 Perses 中可能会增强其性能。因此，考虑将 PDD 技术集成进 Perses 得到 Perses 的变体 p-Perses，并将其作为另一个对比技术。

5.3.2 数据集

本章的评估数据集包括 20 个使用 C 编程语言编写的实验项目。这些实验项目是从现有的研究中选择的，在软件工程领域被广泛认可和使用。它们作为代表性示例，用于评估本章所提技术的有效性。

除了上述提到的项目外，本节还整理了一个包含 87 个项目的数据集。其中，82 个项目是用 C 编程语言编写的，而剩下的 5 个项目是用 Rust 编程语言编写的。这些项目涵盖了具有多样性的程序代码和程序语法结构。

为了生成 C 语言的测试项目，本节使用了一个专门为 C 编译器设计的模糊测试工具 Csmith。通过 Csmith，本节生成了大量的 C 程序，并在接下来的一周内用这些程序来测试 15 个不同稳定版本的 GCC 和 LLVM 编译器。在这个广泛的测试过程中，共发现了 133 个揭错的测试用例，这些用例会导致编译器崩溃或生成错误的代码。这些揭错的测试用例分布在 7 个不同的编译器稳定版本中。为了确保本节的评估具有高效性和相关性，进一步过滤了 51 个编译时间过长的测试用例。经过这一步骤，得到能够代表真实场景的测试项目。最终，本节的评估数据集包含了 82 个测试项目，这些项目成功地在 7 个不同的 GCC 和 LLVM 编译器的稳定版本中触发了错误。其中，有 17 个测试对象在特定版本的 GCC 编译器（gcc-4.4.0 和 gcc-4.6.0）中触发了崩溃错误，而其余的 65 个测试对象在各个版本的 GCC 和 LLVM 编译器（gcc-4.4.3、gcc-4.5.0、gcc-4.6.0、gcc-6.2.0、gcc-7.1.0 和 LLVM-6.0.1）中触发了生成错误代码的缺陷。

关于 Rust 项目，本节特别从 Rust 的缺陷跟踪系统[130] 中选择了 5 个项目。在这 5 个项目中，有一个项目在特定版本的 Rust 编译器（rust-nightly-20191029）中触发了一个崩溃错误。剩下的 4 个项目分别在不同版本的 Rust 编译器（rust-1.20.0、rust-1.34.0、rust-nightly-20200210 和 rust-nightly-20200922）中触发了错误代码生成错误。

通过结合已有的和生成的测试样例，本节的评估数据集为评估本章所提技术的有效性提供了一组全面且具有多样性的数据集。

5.3.3 实验过程

为了回答 RQ1，在评估中使用了所有 107 个项目。首先，记录了每个项目的原始大小。然后，对每个项目分别应用了 Perses 和 T-PDD，并记录了生成结果的大小以及处理时间。此外，计算了每秒删除的标记数。为了确定本章所提技术在效果和效率方面相对于对比技术所取得的改进的统计显著性，进行了配对样本 Wilcoxon 符号秩检验，并针对生成结果的大小、每秒删除的标记数和处理时间计算了相应的 p 值。

为了研究 RQ2，考虑到运行所有项目所需时间巨大，本节在数据集中随机选择了 25 个项目，用于比较 T-PDD 和 p-Perses 的性能。

为了探索 RQ3，进行了一系列实验来评估参数 σ 在 T-PDD 中的影响。对于这个实验，使用了与 RQ2 相同的数据集子集，并将 σ 的值（即条件概率的初始值）变化为 0.3、0.4、0.5、0.6 和 0.7。考虑到运行所有项目的时间开销，没有对所有的项目进行实验。

T-PDD 和 p-Perses 的结果都受到随机性的影响。为了减轻随机性的影响，对这两种技术进行了五次执行，并计算了平均结果。选择了五次作为重复次数，因为每个项目和技术的五次运行结果已经低于其各自平均结果的 1%。对于 RQ1 和 RQ2，在 T-PDD 中将 σ 设置为 0.5。

5.3.4 实现细节

对于比较的技术，本节选择了评估开始时可用的最新版本的 Perses（v1.5）作为实验评估的基准技术。此外，实现了一个名为 p-Perses 的变体，它集成了 PDD 算法。在这个变体中，用 PDD 替换了 Perses（v1.5）的 ddmin 组件，并使用了与 PDD 中使用的相同的超参数值。这两种技术都是在 Perses 的默认设置下执行的。

类似地，为了实现面向语法树的概率化差异调试技术，需要实现面向语法树的概率模型的类，定义指定待选择集合的方法和模型推断的方法，并定义其它必要的类成员。与面向集合的概率模型不同的是，面向语法树的概率模型需要将语法树转换为贝叶斯网。为了达到这一目的，实现了必要的树节点类、树状贝叶斯网的类。关于本章所提技术 T-PDD，实现了从抽象语法树构建贝叶斯网络所需的函数，更新后验概率以

及选择下一个调用反馈函数的集合的必要函数。为了确保公平比较，基于 Perses（v1.5）实现了这些函数。然而，与 Perses 相比，对 T-PDD 的终止条件进行了修改，因此在运行 T-PDD 时，添加了一个额外的命令行选项`-fixpoint false`。这个选项反映了 T-PDD 和 Perses 的不同终止条件。

本节的评估是在一台拥有 16 核 32 线程的 Intel(R) Xeon(R) Gold 6130 CPU（3.7GHz）、128 千兆字节 RAM 和 Ubuntu Linux 16.04 操作系统的 Linux 服务器上进行的。

5.4　实验结果与分析

5.4.1　T-PDD 与 Perses 的比较

表5.1展示了 T-PDD 和 Perses 的整体性能，突出了它们在三个指标上的效率。结果清楚地表明，T-PDD 在效率方面优于 Perses。平均而言，在所有 107 个项目中，T-PDD 每秒能够删除 12 个以上的标记，而保持产生结果的大小不变。重要的是，这种性能改进在统计上是显著的，这可以通过 $p-value_S$ 和 $p-value_T$ 都小于 0.05 来证明。

此外，图5.6中展示了改进的详细分布，重点关注生成结果的大小和处理时间。每个子图展示了 T-PDD 在每个项目上相对于 Perses 所取得的改进，蓝线表示实际改进，橙线作为参考点（刻度为 0）。T-PDD 优于 Perses 的情况在橙线上方表示。

关于生成结果的大小，T-PDD 在 24 个项目上的表现不如 Perses。这种差异可以归因于 T-PDD 生成的输出中存在未使用的变量定义。与 Perses 生成的结果相比，这些残留部分导致平均增加了 22 个标记。由于 T-PDD 在单次遍历转换模板后终止，它可能会保留与已删除的调用函数或变量相关联的定义。关于处理时间，T-PDD 在 6 个项目上的表现相对较差，其中 4 个项目在使用 T-PDD 时生成了较小的结果。值得注意的是，只有两个项目在生成结果的大小和处理时间两方面都不如 Perses。在这种情况下，观察到 T-PDD 难以高效地消除大块连续的死代码，这在实践中是一个罕见的情况，而 Perses 使用的 ddmin 算法可以更有效地处理这种情况。

> **RQ1：**总结来说，对 T-PDD 的评估表明其在效率上相比 Perses 有显著的改进。在一个包含 107 个项目的数据集上，T-PDD 相比 Perses 的时间消耗减少了 26.95%，每秒删除的标记数量增加了 12 个，同时产生的结果大小相同。根据 p 值，这些发现具有统计学意义。

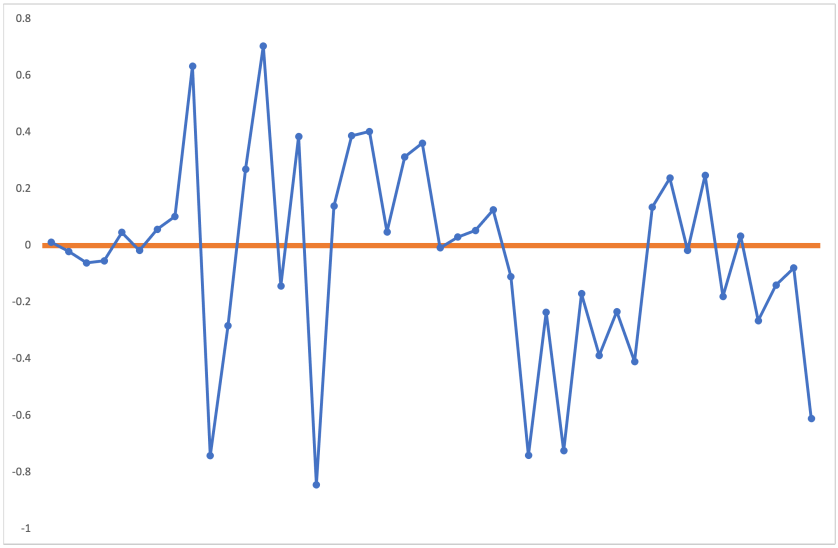
5.4.2　T-PDD 和 p-Perses 的比较

表5.2展示了 T-PDD 和 p-Perses 在三个指标上的详细结果和整体性能。从表5.2可以看出，T-PDD 在所有指标上表现优于 p-Perses。平均而言，T-PDD 每秒删除 8 个更多的

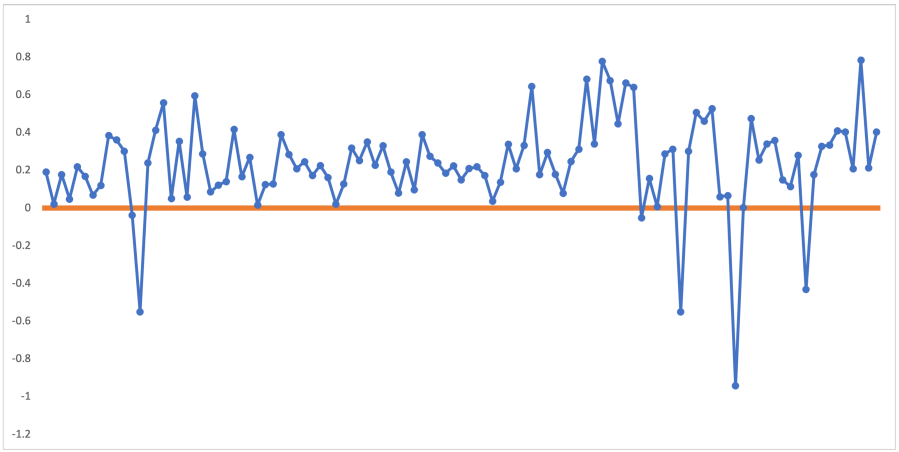
表 5.1　T-PDD 和 Perses 之间的比较

总览	R_i	T-PDD			Perses			$\uparrow_R$	$p-value_R$	$\times S$	$p-value_S$	$\uparrow_T$	$p-value_T$
		R_t	S_t	T_t	R_p	S_p	T_p						
数据集	45,612	48	43	1,000	48	31	1,369	00.00%	0.9736	1.37	0.0000	26.95%	0.0006

在这个表格和本节其余表格中，R 代表结果的大小；S 代表每秒删除的标记数；T 代表处理时间（以秒为单位）；R_i 代表输入的大小；t 代表本章提出的面向语法树的概率化差异调试技术；p 代表 Perses 技术；↑表示改进，其中 $\uparrow_X = (X_p - X_t)/X_p$；$\times S$ 表示加速比，其中 $\times S = S_t/S_p$。在这个表格中，涉及度量指标的数字都是几何平均数。



(a) T-PDD 在各项目上返回结果大小的改进分布



(b) T-PDD 在各项目上处理时间的改进分布

图 5.6　T-PDD 在各项目上的改进分布

标记，使结果比 p-Perses 小 17.5%，这是从数据集中随机选择的 25 个项目进行评估的结果。值得注意的是，由于预定义的尝试删除元素的技术（如第5.1节所述），p-Perses 的表现不如 T-PDD，因为它没有利用历史反馈结果。然而，在两个项目中，T-PDD 在生成结果的大小和处理时间方面表现不如 p-Perses。对于第 20 个项目，T-PDD 的结果中只多了一个标记，但时间消耗少了 8.077%。对于第 25 个项目，T-PDD 所需时间比

表 5.2　T-PDD 和 p-Perses 之间的比较：详细数据

D	项目序号	T-PDD			p-Perses					
		R_t	S_t	T_t	R_p	S_p	T_p	$\uparrow_R$	$\times S$	$\uparrow_T$
包含25个项目的子数据集	1	20	63.008	858	20	60.001	901	0.0%	1.05	4.772%
	2	20	87.628	494	20	62.827	689	0.0%	1.395	28.302%
	3	20	87.05	536	20	48.351	965	0.0%	1.8	44.456%
	4	20	208.426	1,075	20	89.053	2,516	0.0%	2.34	57.273%
	5	20	171.498	1,459	20	162.689	1,538	0.0%	1.054	5.137%
	6	20	110.01	630	20	103.288	671	0.0%	1.065	6.11%
	7	145	44.273	1,849	267	27.374	2,986	45.693%	1.617	38.078%
	8	103	14.41	1,769	189	13.095	1,940	45.503%	1.1	8.814%
	9	82	19.823	1,677	203	15.833	2,092	59.606%	1.252	19.837%
	10	72	35.444	1,041	79	22.549	1,636	8.861%	1.572	36.369%
	11	229	33.23	1,834	395	27.036	2,248	42.025%	1.229	18.416%
	12	351	23.326	2,077	410	20.443	2,367	14.39%	1.141	12.252%
	13	237	23.637	2,426	281	17.243	3,323	15.658%	1.371	26.994%
	14	260	74.224	2,003	301	42.2	3,522	13.621%	1.759	43.129%
	15	20	104.099	696	20	38.849	1,865	0.0%	2.68	62.681%
	16	20	87.099	497	20	62.555	692	0.0%	1.392	28.179%
	17	20	86.888	537	26	73.935	631	23.077%	1.175	14.897%
	18	20	108.886	1,038	20	63.425	1,782	0.0%	1.717	41.751%
	19	55	17.026	574	116	14.693	661	52.586%	1.159	13.162%
	20	60	4.166	808	59	3.83	879	-1.695%	1.088	8.077%
	21	66	4.051	884	74	2.975	1,201	10.811%	1.362	26.395%
	22	54	27.526	1,549	55	16.107	2,647	1.818%	1.709	41.481%
	23	4,539	0.239	15,090	4,677	0.111	31,216	2.951%	2.153	51.659%
	24	99	0.127	55	99	0.047	150	0.0%	2.702	63.333%
	25	141	37.69	4,890	172	54.118	3,405	18.023%	0.696	-43.612%
	总览	66	27	1,104	80	19	1,572	17.5%	1.421	29.771%

p-Perses 更多。正如第5.4.1节所讨论的，T-PDD 在处理包含大量死代码的项目时并不特别有效，而这在程序简化任务中是一个罕见的情况。

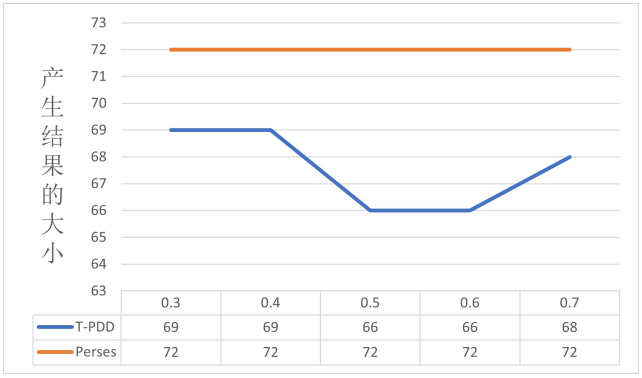
RQ2：对于从数据集中随机选择的 25 个项目，T-PDD 平均每秒删除 8 个更多的标记，以获得比 p-Perses 小 17.5% 的结果，从而显著提高了效率。

5.4.3　T-PDD 中 σ 的影响

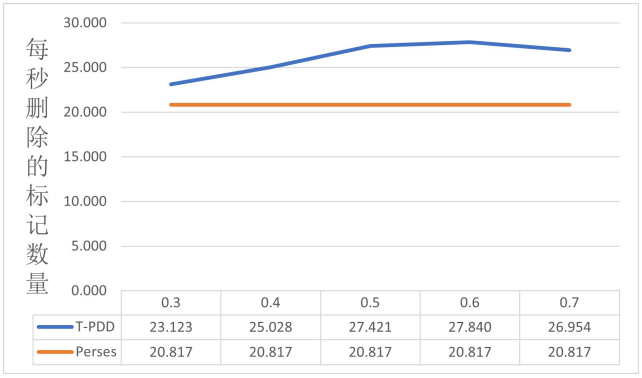
本节对 T-PDD 中初始条件概率参数 σ 的影响进行了评估。该评估基于数据集中随机选择的 25 个项目。结果如图5.7所示，其中包含多个子图，展示了产生结果的大小、每秒删除的标记数以及 T-PDD 在不同 σ 值下的处理时间。

在每个子图中，蓝色线表示具有相应 σ 值的 T-PDD 的性能，而橙色线表示 Perses 的性能以进行比较。值得注意的是，尽管不同的 σ 值可能导致性能的变化，但 T-PDD 在所有研究的 σ 值下始终优于 Perses。

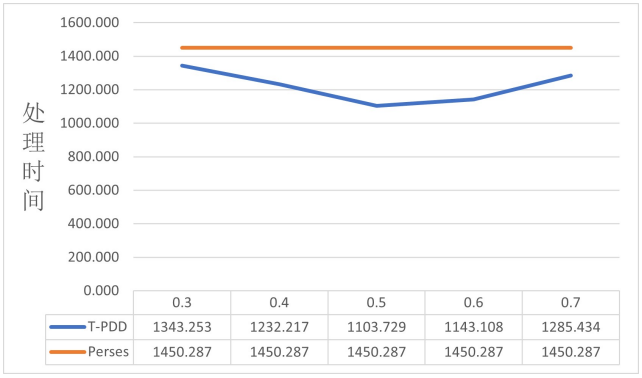
此外，可以观察到，不同 σ 值之间的性能差异相对于 T-PDD 和 Perses 之间的性能



(a) σ 对产生结果的大小的影响



(b) σ 对每秒删除的标记数量的影响



(c) σ 对处理时间的影响

图 5.7　σ 对 T-PDD 的影响

差异来说要小得多。这表明选择 σ 对 T-PDD 的整体性能影响相对较小。

RQ3: 本部分实验结果表明，参数 σ 对 T-PDD 的性能影响较小。无论参数在规定范围内如何取值，T-PDD 始终优于 Perses。

5.4.4　可能威胁有效性的因素

为了解决内部有效性的威胁，本章努力确保 T-PDD 的实现和实验脚本的正确性。作者进行了彻底的代码审查和测试，以验证实现的准确性和可靠性。通过仔细检查代

码，旨在最小化可能影响本章研究内部有效性的错误或不一致性。

外部有效性的威胁主要与本章研究中使用的对象和比较的技术有关。为了解决这个问题，采用了一种结合了现有出版物中常用和公认的对象的技术。此外，为了增加对象的多样性，本章扩展了评估范围，包括 5 个 Rust 语言项目和一些 C 语言项目。这种更广泛的项目范围使得实验验证结果更能体现本章所提出技术的普适性。在未来，计划通过加入来自各种类型的上下文无关文法的其它项目来进一步扩展评估。关于比较的技术，正如之前所讨论的，本章选择了 Perses 作为编译器调试领域的代表性技术。

对于构建有效性的威胁主要来自于实验设置中固有的随机性。随机性可能会导致 T-PDD 的性能以及与之进行比较的 Perses 变体的性能出现变异。为了减轻这种威胁，对每个项目多次运行所涉及的技术，具体地，进行了 5 次重复实验。通过计算平均结果，旨在尽可能地降低或消除随机性的影响并获得更可靠的性能指标。

通过认识和解决这些对有效性的威胁，旨在提高本章所涉及研究的严谨性和可信度。通过仔细的实现、项目选择和随机性影响消融策略，努力确保本章中发现的有效性，并为与 Perses 相比，T-PDD 的性能提供有价值的分析和讨论。

5.5　讨论与小结

本章提出一种面向语法树的概率模型，结合概率化差异调试框架 ProbDD，实例化为面向语法树的概率化差异调试技术 T-PDD，本章实验验证结果表明，本章所提出的技术在效率方面明显优于针对上下文无关文法的最先进的差异调试技术。本章所提出的技术 T-PDD 与 PDD 有着类似的动机，即针对差异调试中的不同应用场景，设计概率模型并结合概率化差异调试框架，利用历史反馈结果来指导差异调试。然而，PDD 和 T-PDD 之间存在关键差异。PDD 提出了面向集合的概率模型，假设元素之间相互独立，而 T-PDD 提出面向语法树的概率模型，利用领域特征，即抽象语法树（AST），构建概率模型，捕捉元素之间的关系。这种区别使得 T-PDD 能够有效解决特定领域的差异调试问题，而 PDD 则更关注更一般的情况。另一个相关工作是 CHISEL[13]，该工作设计了一个马尔可夫决策模型来优化预定义的 ddmin 算法序列。然而，CHISEL 并不能直接应用于改进领域特定的差异调试技术。Pardis[113] 是一种利用机器学习技术增强 C 程序的差异调试过程的技术。它训练一个模型来预测对象的语义有效性，从而提高调试过程的效率和效果。DEBOP[111] 是另一种基于马尔可夫链蒙特卡洛（MCMC）和 Metropolis-Hastings 采样的概率方法。然而，DEBOP 是为不同的问题领域设计的：软件精简，其目标是最大化一组连续的目标函数，而不是使用二元反馈函数。因此，DEBOP 不适用于解决具有二元反馈结果的领域特定差异调试问题，而解决具有二元反馈结果的差异调试问题正是本文关注的重点。

与 ddmin 和 PDD 等通用的差异调试技术不同，存在一些针对特定编程语言或问题领域定制的领域特定的差异调试技术。Berkeley Delta[88] 利用 *topformflat* 来识别嵌套结构，然后调用 ddmin 算法进行处理。分层差异调试（Hierarchical Delta Debugging，简称 HDD）[14] 是一种利用语法树作为基础的程序简化技术，建立在 ddmin 算法之上。自从它被引入以来，已经提出了几种 HDD 的变体，以进一步增强其功能。这些变体包括支持过滤的 HDD（Coarse HDD）[99]、HDDr[100] 和现代 HDD（Modern HDD）[97, 128]。这些变体旨在通过引入新的策略和优化来改进原始的 HDD 算法，以实现更有效的程序简化。C-reduce[101] 是一种利用从 Clang 获取的 C/C++ 语义的启发式方法，用于高效的程序缩减。类似于 C-reduce，领域中还有其他利用专家知识指导缩减过程的技术。这些技术的例子包括 Trimmer[107]、uTrimmer[131] 和 PRAT[132]。J-reduce 及其改进版本是由 Kalhauge 和 Palsberg 提出的用于 Java 字节码的缩减工具，它将缩减任务建模为依赖图缩减问题[105, 133]。GTR*[109] 为树形数据定义了转换模板，并过滤掉在收集的示例数据语料库中不存在的模板。Storm[134] 专门为概率程序简化设计了转换模板。

尽管这些基于转换模板的差异调试技术在各自的领域内显示出了简化效果的改进，但它们经常遇到效率问题，正如现有研究所报道的那样[96]。另一方面，Perses[96] 是一种领域特定的差异调试技术，它在抽象语法树（AST）级别上操作。然而，Perses 遵循预定义的程序简化尝试序列，并不利用现有反馈结果的信息。相比之下，本章所提出的技术 T-PDD 利用从 AST 构建的贝叶斯网络，结合现有的反馈结果，并捕捉元素之间的关系，估计每个元素在结果中被保留的概率。正如第5.3节中的实验验证结果所表明的，T-PDD 在效率方面明显优于 Perses。最近，Vulcan[129] 提出，它使用程序文法来进行更为激进的程序转换。虽然 Vulcan 可以产生更小规模结果，但它被应用于差异调试工具的后处理步骤，并且为了达到产生更小规模结果的目的，过程异常耗时。

本章提出了面向语法树的概率模型，结合概率化差异调试框架，实例化为面向语法树的概率化差异调试技术 T-PDD，旨在提高以程序作为输入的差异调试技术的性能。通过从抽象语法树（AST）构建贝叶斯网络，T-PDD 利用现有的反馈结果并捕捉元素之间的关系，以估计每个元素在结果中被保留的概率。在 T-PDD 中，通过贝叶斯网络的指导，选择具有最大化期望收益的集合调用反馈函数。然后，根据反馈结果更新网络。本章对 T-PDD 的有效性和效率进行评估，使用包括之前研究中广泛使用的 20 个项目以及专门为差异调试问题生成的 87 个额外项目的数据集。实验结果表明，T-PDD 相较于针对上下文无关文法的最先进的领域特定差异调试技术 Perses 具有显著优势。平均而言，T-PDD 在处理时间上减少了 26.95%，同时在最佳情况下产生的结果大小比 Perses 小 3.4 倍。

第六章　基于细粒度反馈的概率化差异调试技术

回顾第三章中给出的差异调试定义：设 $\mathbb{X}$ 是所有感兴趣元素的集合，$\phi : \mathbb{X} \to \{F, T\}$ 是一个反馈函数，用于确定一个集合是否满足规定性质（T）或不满足（F），$|X|$ 表示集合 $X \in \mathbb{X}$ 的大小。给定一个满足 $\phi(X) = T$ 的集合 $X \in \mathbb{X}$，差异调试的目标是找到另一个集合 $X^* \in \mathbb{X}$，使得 $|X^*|$ 尽可能小且 $\phi(X^*) = T$，即 X^* 满足规定的性质。例如，在编译器缺陷调试的应用场景下，差异调试技术通常用于简化揭错的测试用例，帮助编译器开发人员更高效地定位编译器中的缺陷。开发人员通过将揭错的测试用例和一个用于检查是否满足规定性质的反馈函数作为输入，差异调试技术在迭代过程中根据反馈函数的通过（T）或失败（F）的反馈，自动地简化揭错的测试用例，最终返回被简化的测试用例方便开发人员高效地调试。

在软件调试、缓解软件膨胀等领域，输入集合需要满足的性质较为多样，因此反馈函数通常较为复杂，它们往往包含若干步骤的检查。当某一步的检查失败后，反馈函数返回失败（F）；只有当所有的检查都成功后，反馈函数返回通过（T）。现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。例如，在上述调试场景中，检查是否满足给定性质的反馈函数往往包含以下步骤：1）使用最新版本的 Clang 和 GCC 编译器检查编译信息中是否存在未定义行为等相关提示；2）使用 Clang 命令行工具检查是否存在未定义行为、内存泄漏等问题；3）使用最新版本的 Clang 或 GCC 编译器编译并运行程序，检查是否不同于错误的运行结果；4）使用被测试版本的 Clang 或 GCC 编译器编译或运行程序，检查是否揭露了相同的错误。可以观察到，现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。

本文第三章提出的概率化差异调试框架引入概率模型，根据给定反馈函数的反馈结果更新每个元素是否在最终结果中的概率。直觉上地，在删除某些元素后反馈函数在第一步检查失败并反馈失败的情况与在删除某些元素后反馈函数在最后一步检查失败并反馈失败的情况不应相同，而在概率化差异调试框架中，相应被删除的元素在上述两种情况中以同样的方式更新概率。本章旨在通过精化反馈函数的反馈信息，在第三章所提出的概率化差异调试框架的基础上，进一步提升概率化差异调试的效果和效率，提出了基于细粒度反馈的概率化差异调试技术 covProbDD，支持在不修改给定概率模型的情况下引入细粒度反馈。本章视反馈函数由多个子函数构成，对反馈函数中的每一

个子函数（每一步检查）应用概率化差异调试框架中给定的概率模型，每个子函数对应的概率模型用于估计在仅有相应的检查情况下，每个元素保留在最终结果中的概率。在每次迭代中，covProbDD 根据每个子函数（每步检查）对应的概率模型计算出每个元素保留在最终结果的概率，并得到每个元素保留在最终结果里的概率，根据该概率分布选择一个最大化期望收益的元素集合，并调用反馈函数检查该子集是否仍满足规定的性质。然后，covProbDD 根据每一个子函数的反馈结果更新相应的概率模型。

本章深入探讨了精化反馈函数的反馈信息对概率化差异调试性能的影响。基于本文第三章提出的概率化差异调试框架，提出基于细粒度反馈的概率化差异调试技术 covProbDD，该技术利用细粒度反馈更新模型。针对反馈函数中子函数的不同组织方式，进一步提出了基于子函数并行和基于子函数去冗余化的两种变体技术 P-covProbDD 和 D-covProbDD。在实验评估部分，为了验证本章技术对于面向集合的概率模型的提升，采用了包含 30 个项目的公开数据集，这个数据集在差异调试研究中被广泛使用。为了验证本章提出的技术适用于不同种类的反馈函数，根据不同反馈函数的特性，分别在编译器测试和缓解软件膨胀领域验证了本章提出的技术。为了将这三种技术分别应用到数据集的项目中，实现了 Oc-CHISEL、Pc-CHISEL 和 Dc-CHISEL。实验结果表明，精化反馈函数提供的反馈信息，结合细粒度反馈确实能够显著提升差异调试性能。总体而言，三种技术中，D-covProbDD 表现最为出色，与面向集合的概率化差异调试技术相比，它能够将返回结果的大小减小 7.03%，并且节省了 17.58% 的处理时间。为了验证本章技术对于面向语法树的概率模型的提升，采用了包含 20 个项目的公开数据集，这个数据集在差异调试研究中被广泛使用，在编译器测试领域验证了本章提出的技术。为了将这三种技术分别应用到数据集的项目中，实现了 Oc-T-PDD、Pc-T-PDD 和 Dc-T-PDD。实验结果表明，精化反馈函数提供的反馈信息，结合细粒度反馈确实能够显著提升差异调试性能。总体而言，三种技术中，D-covProbDD 表现最为出色，与面向语法树的概率化差异调试技术相比，它能够将返回结果的大小减小 41.18%，并且节省了 43.89% 的处理时间。另外，在两种概率模型的验证中，实验结果表明 covProbDD 和 P-covProbDD 也能够在一定程度上提高差异调试性能，尽管在某些项目上可能不如 D-covProbDD 表现出色。

总之，本章的主要贡献如下：

- 提出了基于细粒度反馈的概率化差异调试技术 covProbDD，该技术基于概率化差异调试框架，精化反馈函数提供的反馈信息，结合细粒度反馈动态地更新概率模型。
- 根据反馈函数中子函数组织特点，进而提出 covProbDD 的变体技术 P-covProbDD 和 D-covProbDD，探索不同的子函数组织形式对差异调试性能的影响。

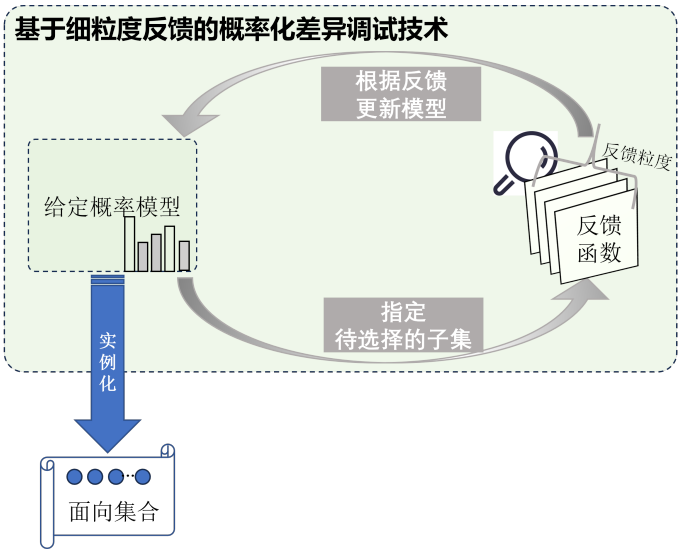


图 6.1　基于细粒度反馈的概率化差异调试技术（面向集合的）

- 分别在编译器测试和缓解软件膨胀领域中评估了本章提出的三种技术，结果表明通过结合细粒度反馈能够带来概率化差异调试性能的进一步提升。

6.1　基于细粒度反馈的概率化差异调试技术及其变体

基于现有差异调试技术仅利用了反馈函数二元反馈信息的观察，为探究精化反馈函数提供的反馈信息对差异调试性能的影响，本章提出了基于细粒度反馈的概率化差异调试技术。通常，差异调试的输入是集合，因此，考虑到通用性，本节以第四章中提出的面向集合的概率化差异调试技术的应用情形，介绍本章所提出的技术如何基于现有概率化差异调试技术，进一步提升概率化差异调试性能的方法。本节基于面向集合的概率模型，介绍如图6.1所示的基于细粒度反馈的概率化差异调试技术。从图6.1中可以看出，本章技术的重点在于精化反馈函数提供的反馈信息，利用细粒度反馈更新概率模型，本章提出的技术视为对于概率化差异调试框架的提升。实际上，本章提出的基于细粒度反馈的概率化差异调试技术并不会修改依赖的概率模型。因为，概率模型中已经定义好模型推断等涉及概率模型部分的具体细节，本章技术是结合细粒度反馈，对概率化差异调试技术的进一步提升，主要关注如何利用细粒度反馈更新模型的过程。

6.1.1　模型定义

给定概率模型 $\mathbb{M}$（以面向集合的概率模型为例），对于每个子函数 γ_i，covProbDD 实例化一个概率模型 $\mathbb{M}_i$。类似第四章中提出的面向集合的概率化差异调试技术，对于包含 n 个元素的输入 $X = (x_1, x_2, ..., x_n)$，在每个 $\mathbb{M}_i$ 中，为 X 中的每个元素索引 j 分配一个伯努利随机变量 θ_{ij}，用于表示仅考虑子函数 γ_i 作为反馈函数时第 j 个元素是否

在 X^* 中。使用参数 p_{ij} 来表示第 j 个元素在 X^* 中的概率，即 $Pr(\theta_{ij}=1)=p_{ij}$。因此，该概率模型是一个 $n\times m$ 维参数向量：

$$\mathbb{M}=\begin{bmatrix}\mathbb{M}_1\\ \mathbb{M}_2\\ \vdots\\ \mathbb{M}_m\end{bmatrix}=\begin{bmatrix}p_{11} & p_{12} & \cdots & p_{1n}\\ p_{21} & p_{22} & \cdots & p_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ p_{m1} & p_{m2} & \cdots & p_{mn}\end{bmatrix}$$

在每个模型 $\mathbb{M}_i$ 中，随机变量 θ_{ij} 是相互独立的。该假设的合理性已经在第四章中讨论。类似地，给定输入集合 $X=(x_1,...,x_n)$ 和反馈函数 $\phi=(\gamma_1,...,\gamma_m)$，$X$ 通过反馈函数 γ_i 的概率是 X^* 中没有任何元素被删除在 X 之外的概率，也就是

$$Pr(\gamma_i(X)=T)=\prod_j(1-p_{ij})^{1-x_j}$$

。下面，进一步地讨论子函数之间的关系。按照现有差异调试中反馈函数的执行流程，当反馈函数中某一个检查失败时，反馈函数直接反馈失败信息，后续的检查均被认为是失败的。设 ϕ 在第 k 个检查处失败，有 $\gamma_i(X)=1, 1\le i\le k$ 且 $\gamma_j(X)=0, k\le j\le m$。进一步考虑 γ 是相互独立的。这个假设是合理的。例如在缓解软件膨胀场景下，所需满足的性质中包括一系列软件功能，它们由不同的代码块实现，这些功能通常是相互独立的。基于这个假设，有 $Pr(\phi(X)=T)=\prod_i Pr(\gamma_i(X)=T), 1\le i\le m$。给定输入集合 $X=(x_1,...,x_n)$ 和反馈函数 $\phi=(\gamma_1,...,\gamma_m)$，随机变量 θ_i 表示第 i 个元素在最终结果中的概率，每个元素应当保留在最终结果中的概率 $Pr(\theta_i=1)=\prod_{j=1}^{m}Pr(\theta_{ji}=1)$。

6.1.1.1 模型的先验分布

实际上，本章所提出的技术并不涉及模型的先验分布，模型的先验分布应在给定概率模型时确定好，为了读者能够更好地理解本章技术，以基于面向集合的概率化差异调试技术为例，介绍本章技术中模型先验分布的设置方法。

鉴于最初用户对各个元素缺乏详细了解，对于每一个子模型 $\mathbb{M}_i$，可以考虑将所有的 p_{ij} 设定为一个参数为 σ_i 的均匀分布，其中 $0<\sigma_i<1$ 为 covProbDD 的可调参数。根据问题领域的特性，可以采用多种方法来确定合适的 σ_i 值。如果应用领域通常伴随着固定的减少比例，那么可以选择将 σ_i 设置为该比例。另一方面，如果在减少后的子集通常具有一个固定的长度 m，那么可以将 σ_i 设定为 m 与输入集合长度 n 之比，即 $\sigma_i=\frac{m}{n}$。根据第4.5.2节的实验评估，可以得出如下结论，参数 σ_i 对子模型 $\mathbb{M}_i$ 的性能影响较小。可以看出，基于细粒度反馈的概率化差异调试技术中模型的先验分布设置方法仅与所使用的概率模型本身有关。

6.1.2　待选择集合的指定

回顾第3.3.2.1节中定义的集合收益：

$$gain(X', X) = \begin{cases} |X| - |X'|, & \phi(X') = T \\ 0, & \phi(X') = F \end{cases}$$

基于给定的概率模型 $\mathbb{M}$（本章考虑基于面向集合的概率模型），可以计算出集合 X 的期望收益。

$$\begin{aligned} \mathbb{E}[gain(X)] &= |ex(X)|\, Pr(\phi(X) = T) \\ &= |ex(X)| \prod_{i}^{m} Pr(\gamma_j(X) = T) \\ &= |ex(X)| \prod_{i} \prod_{j} (1 - p_{ij})^{1-x_j} \end{aligned} \tag{6.1}$$

根据第四章的结论，使用以下步骤来找到具有最大期望收益的集合，并将其作为下一次迭代的输入。

1. 根据每个子模型 $\mathbb{M}_j$ 中的概率分布，计算出 $\tau_i = \prod_{j=1}^{m}(1 - p_{ji})$；
2. 按照 τ_i 的降序顺序，组成元素下标索引集合；
3. 根据上述顺序，逐个删除集合中的元素，直到期望收益开始下降；
4. 返回具有最高期望收益的集合。设 $\hat{X}$ 是上述过程返回的子集。

6.1.3　模型更新

结合第3.3.3节中介绍的计算后验概率的方法，本节介绍模型更新的方法。

更新概率的步骤如下：对于每个子模型 $\mathbb{M}_i$，在指定待选择的集合 X 之后，获得了它是否通过反馈函数的反馈信息。根据这个反馈，对于每个元素 x_j，更新子模型中的参数 p_{ij}。具体而言，是更新变量 θ_{ij} 的后验概率。

一方面，如果子函数 γ_i 失败，公式3.7可被改写为

$$\begin{aligned} & Pr(\theta_{ij} = 1 | \gamma_i(X) = F) \\ &= \frac{Pr(\theta_{ij} = 1) Pr(\gamma_i(X) = F | \theta_{ij} = 1)}{Pr(\gamma_i(X) = F)} \\ &= \begin{cases} \frac{Pr(\theta_i=1)\cdot 1}{1-\prod_j (1-Pr(\theta_j=1))^{1-x_j}} = \frac{p_{ij}}{1-\prod_j (1-p_{ij})^{1-x_j}} & x_i = 0 \\ \\ \frac{Pr(\theta_i=1)\, Pr(\gamma_i(X)=F)}{Pr(\gamma_i(X)=F)} = Pr(\theta_i = 1) = p_{ij} & x_i = 1 \end{cases} \end{aligned} \tag{6.2}$$

另一方面，如果子函数 γ_i 通过，公式3.6可被改写为

$$
\begin{aligned}
&Pr(\theta_{ij}=1|\gamma_i(X)=T) \\
&=\frac{Pr(\theta_{ij}=1)Pr(\gamma_i(X)=T|\theta_{ij}=1)}{Pr(\gamma_i(X)=T)} \\
&=\begin{cases}
\frac{Pr(\theta_i=1)\cdot 0}{Pr(\gamma_i(X)=T)}=0 & x_i=0 \\
\frac{Pr(\theta_i=1)Pr(\gamma_i(X)=T)}{Pr(\gamma_i(X)=T)}=Pr(\theta_i=1)=p_{ij} & x_i=1
\end{cases}
\end{aligned}
\tag{6.3}
$$

根据上述方程，根据以下规则在每次反馈 $\gamma_i(X)=R_i$ 后更新每个 j 的参数 p_{ij}。

1. 如果第 j 个元素包含在 X 中，则 p_{ij} 保持不变。
2. 如果第 j 个元素从 X 中删除，并且 $R_i=T$，反馈函数通过，则 p_{ij} 被设置为零。
3. 如果第 j 个元素从 X 中删除，并且 $R_i=F$，反馈函数失败，则 p_{ij} 被设置为 $\frac{p_{ij}}{1-\Pi_j(1-p_{ij})^{1-x_j}}$。

通过上述在给定面向集合的概率模型的情况下，对本章提出的基于细粒度反馈的概率化差异调试技术的介绍，不难发现，本章技术不依赖于具体给定的概率模型。在将细粒度反馈用于更新模型时，可以直接利用概率模型中相关部分的结论。本章技术是在现有概率化差异调试技术的基础上引入细粒度反馈（本章以面向集合的概率化差异调试技术为基础），以便进一步提升差异调试的效率和效果。

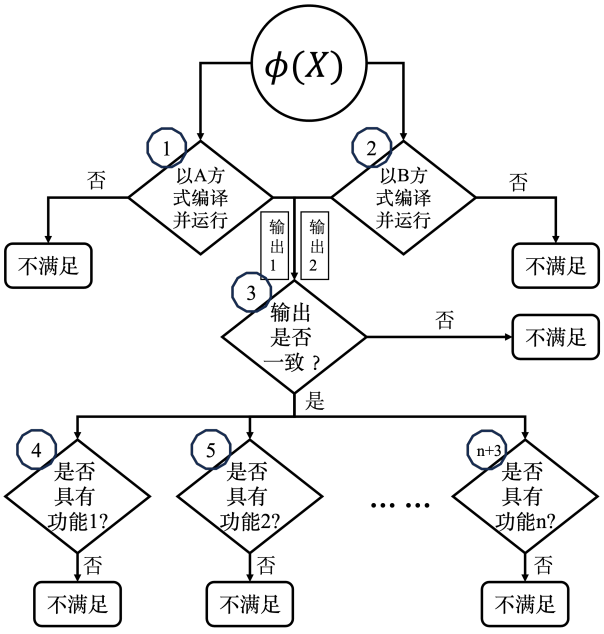


图 6.2　一个典型的反馈函数流程示例

6.1.4　covProbDD 的变体技术

本章技术精化差异调试输入之一的反馈函数提供的反馈信息，主要研究了结合细粒度反馈信息后能否为差异调试技术带来性能的提升。这使得本章可以深入讨论反馈函数中各子函数的组织结构和潜在的可以提升差异调试性能的技术。图6.2展示了一个典型的反馈函数示例，为了简略起见，省略了全部子函数通过、反馈函数返回成功的流程示意；在本例中，仅以是否有某一处“不满足”被覆盖到为依据，判断反馈函数是否通过。图中带有标号的菱形框依次表示子函数 $\gamma_1, \gamma_2, \ldots, \gamma_{n+3}$，标号也表明了子函数由第 1 步被执行到第 $n+3$ 步被执行的串行执行的顺序。矩形方框及附近的箭头表示数据流，即 γ_1 和 γ_2 产生的中间结果“输出 1”和“输出 2”作为 γ_3 中某步骤的输入。从图中可以清晰地看出，covProbDD 中的反馈函数执行某一个子函数失败后会立即返回失败。请注意，虽然这样的执行方式蕴含子函数之间并不独立，但为了计算的简便性考虑，covProbDD 中依旧假设了子函数之间的独立性。下面，根据与 covProbDD 不同的子函数间组织结构，本章进一步提出 covProbDD 的变体技术 P-covProbDD 和 D-covProbDD。

6.1.4.1　基于子函数并行的 P-covProbDD

从图6.2可以看出，子函数 $\gamma_4, \gamma_5, \ldots, \gamma_{n+3}$ 之间没有数据依赖。然而，covProbDD 用串行的方式执行子函数 $\gamma_4, \gamma_5, \ldots, \gamma_{n+3}$。进一步地，子函数 γ_1 和 γ_2 由于没有依赖，也可以使用并行的方式。该例是一个典型的反馈函数，意味着上述改进适用于大部分差异调试的场景。本节提出基于子函数并行的 P-covProbDD 技术。由于大部分的子函数间没有依赖，它们在 P-covProbDD 中被并行执行，相互之间不会产生影响，更符合本章中的假设，即子函数之间是独立的。

6.1.4.2　基于子函数去重的 D-covProbDD

进一步地，本节发现，大部分反馈函数中的子函数存在冗余的情况，因此提出基于子函数去重的 D-covProbDD 技术，探究子函数去重能够为差异调试性能带来多大的提升。本节发现，在实际场景中，应用差异调试技术的反馈函数质量参差不齐，由于对目标软件功能理解的局限性，大多存在子函数冗余的情况。例如，在缓解软件膨胀的应用场景中，结合图6.2例，子函数 $\gamma_4, \ldots, \gamma_{n+3}$ 对目标软件功能的检查存在相互包含的情况，如子函数 γ_5 对功能 2 的检查中包含 γ_4 的检查内容，那么 γ_4 作为冗余子函数，可以不被执行。然而，自动化地检测冗余子函数是复杂且困难的，一方面需要对目标软件具有很强的了解；另一方面，通过程序分析等手段的冗余性处理很难保证准确性。因此，在本章的试验评估中，采取人工的方式，确保反馈函数经过冗余处理的前后能

够检查相同类型的属性。

6.2 实验设计

给定输入集合和用于检查是否满足规定性质的反馈函数，差异调试技术能够自动化地缩减输入集合中的元素，并返回最小的、依然满足给定属性的结果。现有的差异调试技术仅利用反馈函数返回结果的通过（T）或失败（F）作为反馈，本章探索了精化反馈函数的反馈信息，结合细粒度反馈，对差异调试技术性能的改变。本文第四章提出的面向集合的概率化差异调试技术改进了 ddmin 算法，成为最先进的差异调试技术之一。本节将面向集合的概率化差异调试技术 PDD 作为基准技术，探索结合细粒度反馈信息是否能够进一步提升差异调试的性能（RQ1~RQ3）。此外，为了验证本章提出方法的通用性，本节进而将面向语法树的概率化差异调试技术 T-PDD 作为基准技术，探索结合细粒度反馈信息能否进一步提升差异调试的性能（RQ4~RQ6）。在不同的比较场景下，本节统一使用本章提出的方法名指代对应的技术，如在与 PDD 比较的场景中，covProbDD 指代针对面向集合的概率模型结合细粒度反馈的技术（下文的 Oc-CHISEL），又如在与 T-PDD 比较的场景中，covProbDD 指代针对面向语法树的概率模型结合细粒度反馈的技术（下文的 Oc-T-PDD）。最后，总结了本章的三种技术在解决差异调试问题中表现的区别（RQ7）。总之，本节的评估涉及以下研究问题。

RQ1: covProbDD 在差异调试技术不同的应用领域中与 PDD 相比表现如何?

RQ2: P-covProbDD 在差异调试技术不同的应用领域中与 PDD 相比表现如何?

RQ3: D-covProbDD 在差异调试技术不同的应用领域中与 PDD 相比表现如何?

RQ4: covProbDD 与 T-PDD 相比表现如何?

RQ5: P-covProbDD 与 T-PDD 相比表现如何?

RQ6: D-covProbDD 与 T-PDD 相比表现如何?

RQ7: covProbDD、P-covProbDD 和 D-covProbDD 在解决差异调试问题中表现的区别?

6.2.1 实验设置

对于 RQ1~RQ3，评估考虑了以下两个应用领域，选择了基于 PDD 的代表性差异调试技术作为目标技术。将目标技术中的 PDD 组件替换为 covProbDD，并与原始目标技术使用 PDD 进行性能比较。

- **编译器调试**. 该应用领域主要是缩减揭露 C 编译器（GCC 和 LLVM）缺陷的测试用例，它们都是由一种随机 C 程序生成器 Csmith[95] 生成的，专门用于编译器测试。

- **缓解软件膨胀**. 该应用领域主要是缩减由 C 语言编写的软件，使其能够在嵌入式设备上运行。

以上两个领域的目标语言是 C 语言，C 程序的代表性调试工具是 CHISEL[13]，它依赖于 C 语言的语法和程序元素之间的依赖关系（如变量的定义-使用等依赖关系）。考虑到现有的差异调试研究中对上述两种应用领域研究较为广泛，且有基于 PDD 的代表性技术的公开实现，本章着重选取了这两个应用领域进行评估。

为了方便表述，将原始的集成 PDD 的 CHISEL 技术称为 *p-CHISEL*，将把 CHISEL 技术中 PDD 替换为 covProbDD、P-covProbDD 和 D-covProbDD 版本分别称为 *Oc-CHISEL*、*Pc-CHISEL* 和 *Dc-CHISEL*。

由于 T-PDD 所依赖的工具没有完全支持缓解软件膨胀应用领域中部分项目的语法解析，对于 RQ4~RQ6，评估只考虑编译器测试应用领域。为了方便表述，将 T-PDD 技术的 covProbDD、P-covProbDD 和 D-covProbDD 对应版本分别成为 *Oc-T-PDD*、*Pc-T-PDD* 和 *Dc-T-PDD*。

6.2.2　数据集

本章的实验评估中主要选择了已有研究工作中原始技术的评估项目，以避免选择偏差等因素导致的不公平对比。具体而言，选择了以下 30 个项目。

- **编译器测试**. 在编译器测试领域使用了 20 个公开可用的项目。这些项目是触发 GCC 和 Clang 中崩溃和编译错误的 C 语言程序，在差异调试过程中要满足的属性是在没有任何未定义行为的情况下重现报告的错误。
- **缓解软件膨胀**. 在缓解软件膨胀领域使用了用于评估 CHISEL[13] 的基准数据集，包括 10 个 C 程序项目，用于在嵌入式系统中进行精简。对于这 10 个 CHISEL 中使用过的项目，要满足的属性是成功编译、通过给定的测试用例，并保留特定的软件功能。

6.2.3　实验过程

本章提出了 covProbDD 技术，根据反馈函数中子函数不同的组织特点，进而提出了基于子函数并行的 P-covProbDD 和基于子函数去重的 D-covProbDD 技术。为了验证本章所提出的技术在不同种类的应用领域中的效果，选取了差异调试技术被广泛使用的两个应用领域：编译器调试和缓解软件膨胀。

为了回答 RQ1-RQ3，记录了每个项目的原始大小。然后，对两个应用领域中的每个项目应用了 Oc-CHISEL、Pc-CHISEL、Dc-CHISEL 和 p-CHISEL 的技术，并记录了产生结果的大小和处理时间。接下来，计算了每秒删除的标记数。对于运行超时的项目

(超时时间被设置为 3 小时，即 10,800 秒)，使用标记 "-" 来表示其处理时间。对于 RQ4-RQ6，情况类似，唯一区别在于，T-PDD 不支持缓解软件膨胀应用领域中某些项目的语法特性，仅在编译器测试领域中的每个项目应用了 *Oc-T-PDD*、*Pc-T-PDD*、*Dc-T-PDD* 和 T-PDD 技术。通过比较产生结果的大小、每秒删除的标记数以及没有超时的项目的处理时间，以分别回答本章所提出的 covProbDD、P-covProbDD 和 D-covProbDD 技术是否在效果和效率上相对于面向集合的概率化差异调试技术都取得了显著的改进。

为了回答 RQ7，基于 RQ1-RQ6 的实验结果，着重分析了本章提出的三种技术的优缺点。

6.2.4 实现细节

将 C++ 实现的 CHISEL[13] 中的 ddmin 算法替换为 PDD 技术形成的技术实现作为 p-CHISEL，并在此基础上实现了 Oc-CHISEL、Pc-CHISEL 和 Dc-CHISEL。特别地，CHISEL 的实现包括自动死代码消除（DCE）和依赖分析（DA）的组件，通过使用三个命令行选项（即`-skip_local_dep`，`-skip_global_dep` 和`-skip_dce`）禁用了这些组件，原因如下。首先，DCE 是为 CHISEL 中的程序精简而设计的，在其它应用领域的大多数调用反馈函数中都会失败。例如，由于无法访问的代码而触发的编译器错误。其次，发现在某些情况下，例如当函数调用作为参数传递时，DA 组件会产生错误的结果，而评估中使用的项目中存在这种情况。请注意，对于本评估中使用到的所有版本的 CHISEL，DCE 和 DA 功能都被禁用了。

本章提出基于细粒度反馈的概率化差异调试技术，可以方便得移植到任意概率化差异调试技术的具体实现中。对于 Oc-CHISEL，本文第四章介绍了面向集合的概率化差异调试技术具体实现的过程，通过将涉及表示单个模型中各个元素的概率的常量扩展为 m 维向量（m 为反馈粒度，由用户根据反馈函数的具体性质进行设置，在本章中使用子函数个数作为 m 的取值)，同时，在每一步差异调试迭代过程中，需要实现利用该 m 维向量，计算最终第 i 个元素在最终结果里的概率值 p_{ji}，其中 $1 \leq j \leq m$，相关计算式在第6.1.2节中给出。此外，为了精化反馈函数提供的反馈信息，获得细粒度的反馈，需要在实现反馈函数的脚本中进行插桩，输出各个子函数的反馈结果，并统一记录到文件中。在相应概率化差异调试技术的实现中，通过读取该文件，获得细粒度反馈的信息并保存到 m 维向量中。至此，通过修改反馈函数的实现脚本并记录细粒度反馈、将原有概率化差异调试技术的具体实现中的两个常量有依据地转换为向量、实现真正表示元素在结果中的概率等有限步骤，概率化差异调试技术就可以轻松地转换为基于细粒度反馈的概率化差异调试技术。在面向语法树的概率化差异调试技术的实现中，利用上述步骤也可以转换为相应的基于细粒度反馈的概率化差异调试技术。可

以类比地，可以将使用任意概率模型的概率化调试技术方便地转换为基于细粒度反馈的概率化差异调试技术的实现。

对于 Pc-CHISEL，涉及到对每个项目反馈函数的并行化处理。主要涉及将原反馈函数根据子函数间依赖拆分成若干个独立的子函数，并实现并行执行独立子函数的 bash 脚本，将该 bash 脚本作为差异调试技术的输入之一。

对于 Dc-CHISEL，涉及到对每个项目反馈函数的子函数去重。主要涉及将原反馈函数根据子函数检查功能是否重叠，去除检查功能被其它子函数完全包含的子函数，并将处理后的反馈函数作为差异调试技术的输入之一。在本章的实验评估的项目中，对于编译器测试领域，冗余的子函数主要体现在利用最新的编译器对输入程序的重复编译；对于缓解软件膨胀领域，冗余的子函数主要体现在对目标软件冗余的功能检查。

Oc-T-PDD、Pc-T-PDD 和 Dc-T-PDD 的实现方式与上述 Oc-T-PDD、Pc-T-PDD 和 Dc-T-PDD 技术的实现类似，进一步印证了本章所提出的技术可以很容易地迁移到使用不同概率模型的场景。具体地，针对每一个细粒度的反馈函数，都对应生成一个概率模型。在模型推断之前，通过将每个对应边所表示的条件概率相乘的方式合并所有的细粒度概率模型，在合并后的概率模型上使用第五章中相同的方法进行模型推断。

本章的评估是在一台配备 16 核 32 线程的 Intel(R) Xeon(R) Gold 6130 CPU（3.7GHz）、128GB 内存和 Ubuntu Linux 16.04 操作系统的 Linux 服务器上进行的。

6.3　实验结果与分析

6.3.1　covProbDD 和 PDD 之间的比较

表6.1显示了 covProbDD 与 PDD 之间比较的详细数据。表格第一列是应用领域，编译器测试应用领域中对应 20 个项目的数据条目，缓解软件膨胀应用领域中对应 10 个项目的数据条目。表格第二列是项目名称。表格第三-五列是基于面向集合的概率模型的 covProbDD 技术对应的结果，表格第六-八列是 ProbDD 技术的结果。表格的最后三列代表了两种技术在三个指标上的比例数值。从整体上看，将 CHISEL 中的相关算法替换为 covProbDD 后，Oc-CHISEL 获得了平均 31 个标记每秒的缩减速度，平均处理时间 1,054 秒，平均返回结果大小 8,499。相比于 PDD，处理时间缩减了 0.38%，获得 5.26% 更小的返回结果大小，在缩减速度上获得 1.03 加速比。

具体地，在编译器测试领域，在返回结果大小上 covProbDD 有两个项目不如 PDD，然而 covProbDD 都使用了更短的处理时间；covProbDD 在 8 个项目上消耗了更长的时间，其中有三个项目 covProbDD 返回了更小的结果，有五个项目 covProbDD 消耗了更长的时间返回了与 PDD 相同大小的结果。

在缓解软件膨胀领域，在不超时的项目中，covProbDD 在返回结果上有一个项目

不如 PDD，然而 covProbDD 使用了更短的处理时间；covProbDD 在该领域全部项目上都比 covProbDD 更高效，即均使得差异调试过程加速。在超时的项目中，covProbDD 在固定的三小时时间限制内在三个项目上的返回结果大小不如 PDD，然而差距十分有限，即较 PDD 返回的结果分别差 145、105 和 164 个标记数量，仅占原始大小的约 0.1~0.33%。

RQ1：总的来说，对 covProbDD 的评估结果表明，其在效率和效果上改善了 PDD，消耗了 5.26% 更少的时间同时返回了 0.38% 更小的结果。从整体上来看，不改变反馈函数的情况下，利用细粒度反馈能够进一步提高概率化差异调试技术的效率和效果。

6.3.2 P-covProbDD 与 PDD 之间的比较

表6.2显示了 P-covProbDD 与 PDD 之间对比的详细数据。从总体上看，P-covProbDD 获得了平均 7,807 个标记数的返回结果，平均耗时 1,727 秒，产生 20 个标记每秒钟的缩减速度，相比于 PDD，在缩减大小上返回了比 PDD 小平均约 12.98% 的结果，然而却要消耗多 63.23% 的时间。结果表明，P-covProbDD 以并行处理子函数的方式并不能够带来效率的提升，这种方式的时间瓶颈在于独立的子函数中耗时最久的子函数。然而，这样的子函数在串行执行的方式中，当前驱的子函数返回失败时，往往会被直接设置为失败并返回整个反馈函数。

具体地，在编译器测试领域中，P-covProbDD 仅在六个项目上获得了更小的返回结果，以及两个项目上消耗了了更短的处理时间，在两个项目上使用了更短的时间返回了更小的结果。可以发现，在编译器测试领域中，P-covProbDD 与 PDD 差距较大。由于输入通常是随机产生的较大的 C 程序，编译、运行的时间通常较长，导致并行子函数后的瓶颈拖累了整体的效果。

在缓解软件膨胀领域，P-covProbDD 在两个项目上获得了更小的返回结果，在四个项目上获得了与 PDD 一样大小的返回结果，在两个项目上消耗了更多的时间。然而，相比于编译器领域中 P-covProbDD 的表现，在缓解软件膨胀领域，P-covProbDD 在表现差的项目上与 PDD 的差距较小，同时 P-covProbDD 也能够在大部分项目上获得更好的效率和效果。在未来工作中，更多属于缓解软件膨胀领域的项目应当被用来实验评估，进一步证明 P-covProbDD 在该领域中可能的提升。

RQ2：总的来说，对 P-covProbDD 的评估结果表明，其仅在效果上改善了 PDD。相比于 PDD，P-covProbDD 能够返回 12.98% 更小的结果，但要消耗 63.23% 更多的时间。从整体上来看，将反馈函数中的子函数调整为并行方式，虽然更符合本章

表 6.1　covProbDD 和 PDD 之间的比较：详细数据

D	项目名	Oc-CHISEL			p-CHISEL			↑$_R$	×S	↑$_T$
		R_{oc}	S_{oc}	T_{oc}	R_p	S_p	T_p			
编译器测试	clang-22382	4,028	32.56	183	4,028	39.73	150	0.00%	0.82	-22.00%
	clang-22704	1,225	62.85	2,915	1,224	62.81	2,917	-0.08%	1.00	0.07%
	clang-23309	7,267	22.79	1,142	7,267	25.87	1,006	0.00%	0.88	-13.53%
	clang-23353	7,432	32.34	704	7,698	32.41	694	3.46%	1.00	-1.41%
	clang-25900	2,988	51.61	1,472	2,988	59.78	1,270	0.00%	0.86	-15.83%
	clang-26760	5,241	103.72	1,970	4,970	89.49	2,286	-5.45%	1.16	13.83%
	clang-27137	14,365	54.91	2,917	14,400	55.08	2,907	0.24%	1.00	-0.32%
	clang-27747	6,124	245.20	684	6,270	233.68	717	2.33%	1.05	4.62%
	clang-31259	4,166	66.95	666	4,166	70.50	633	0.00%	0.95	-5.31%
	gcc-59903	6,602	77.22	660	6,602	69.09	737	0.00%	1.12	10.53%
	gcc-60116	9,445	76.67	858	9,453	74.18	886	0.08%	1.03	3.23%
	gcc-61383	9,702	15.00	1,516	9,702	20.50	1,109	0.00%	0.73	-36.66%
	gcc-61917	13,691	80.35	892	13,691	75.67	947	0.00%	1.06	5.82%
	gcc-64990	6,870	150.35	944	6,970	160.92	882	1.43%	0.93	-7.11%
	gcc-65383	5,065	99.18	392	5,065	97.78	397	0.00%	1.01	1.41%
	gcc-66186	6,447	66.72	615	6,447	65.96	622	0.00%	1.01	1.14%
	gcc-66375	5,169	71.47	844	5,169	70.57	854	0.00%	1.01	1.26%
	gcc-70127	2,527	154.00	988	12,452	105.71	1,346	79.71%	1.46	26.57%
	gcc-70586	8,383	212.81	958	8,383	224.41	908	0.00%	0.95	-5.45%
	gcc-71626	639	282.56	13	639	262.80	14	0.00%	1.08	6.99%
缓解软件膨胀	mkdir-5.2.1	8,321	20.06	1,320	8,321	18.53	1,429	0.00%	1.08	7.63%
	rm-8.4	7,400	11.83	3,132	7,427	11.46	3,232	0.36%	1.03	3.11%
	chown-8.2	7,682	9.88	3,662	7,445	9.83	3,704	-3.18%	1.00	1.14%
	grep-2.19	110,025	1.63	-	114,754	1.20	-	4.12%	1.37	-
	bzip2-1.0.5	51,123	1.80	-	51,123	1.80	-	0.00%	1.00	-
	sort-8.16	51,993	3.34	-	51,848	3.35	-	-0.28%	1.00	-
	gzip-1.2.4	16,653	2.71	-	16,548	2.72	-	-0.63%	1.00	-
	uniq-8.16	14,045	5.42	9,190	14,045	5.39	9,242	0.00%	1.01	0.56%
	date-8.21	20,211	3.80	8,736	20,219	3.35	9,920	0.04%	1.14	11.94%
	tar-1.14	58,538	9.70	-	58,374	9.71	-	-0.28%	1.00	-
	总览	8,499	31	1,054	8,971	30	1,058	5.26%	1.03	0.38%

的技术假设，产生了更小的结果，但由于子函数并行的瓶颈在于正常情况下执行不到的十分耗时的子函数，导致效率上并没有改善现有概率化差异调试技术。

6.3.3　D-covProbDD 与 PDD 之间的比较

表6.3显示了 D-covProbDD 与 PDD 之间比较的详细数据。从整体平均来看，D-covProbDD 获得了 8,340 个标记数的返回结果，消耗了 872 秒处理时间，获得了 37 个标记数每秒的缩减速率，相比于 PDD 产生了 7.03% 更小的返回结果，消耗了 17.58% 更短的处理时间，获得了平均 1.23 的缩减加速比。

具体地，在编译器测试领域，D-covProbDD 在五个项目上产生了比 PDD 更大的返回结果，但在其中三个项目上消耗了更短的时间；D-covProbDD 只在两个项目上产生了

表 6.2　P-covProbDD 和 PDD 之间的比较：详细数据

D	项目名	Pc-CHISEL			p-CHISEL					
		R_{pc}	S_{pc}	T_{pc}	R_p	S_p	T_p	$\uparrow_R$	$\times S$	$\uparrow_T$
编译器测试	clang-22382	4,028	33.69	176	4,028	39.73	150	0.00%	0.85	-17.93%
	clang-22704	3,051	17.94	10,108	1,224	62.81	2,917	-149.26%	0.29	-246.54%
	clang-23309	7,267	13.99	1,861	7,267	25.87	1,006	0.00%	0.54	-84.93%
	clang-23353	7,722	12.60	1,784	7,698	32.41	694	-0.31%	0.39	-157.00%
	clang-25900	3,012	30.92	2,455	2,988	59.78	1,270	-0.80%	0.52	-93.24%
	clang-26760	4,311	126.99	1,616	4,970	89.49	2,286	13.26%	1.42	29.30%
	clang-27137	3,864	78.37	2,177	14,400	55.08	2,907	73.17%	1.42	25.10%
	clang-27747	6,334	116.49	1,437	6,270	233.68	717	-1.02%	0.50	-100.52%
	clang-31259	492	70.30	687	4,166	70.50	633	88.19%	1.00	-8.55%
	gcc-59903	7,614	42.78	1,168	6,602	69.09	737	-15.33%	0.62	-58.29%
	gcc-60116	9,445	48.38	1,359	9,453	74.18	886	0.08%	0.65	-53.36%
	gcc-61383	9,726	3.24	7,012	9,702	20.50	1,109	-0.25%	0.16	-532.05%
	gcc-61917	11,008	34.59	2,149	13,691	75.67	947	19.60%	0.46	-126.98%
	gcc-64990	7,971	71.72	1,965	6,970	160.92	882	-14.36%	0.45	-122.80%
	gcc-65383	5,065	47.45	819	5,065	97.78	397	0.00%	0.49	-106.09%
	gcc-66186	6,447	32.72	1,254	6,447	65.96	622	0.00%	0.50	-101.59%
	gcc-66375	5,169	38.59	1,562	5,169	70.57	854	0.00%	0.55	-82.84%
	gcc-70127	2,527	82.33	1,849	12,452	105.71	1,346	79.71%	0.78	-37.35%
	gcc-70586	8,497	96.90	2,102	8,383	224.41	908	-1.36%	0.43	-131.46%
	gcc-71626	633	97.77	38	639	262.80	14	0.94%	0.37	-169.23%
缓解软件膨胀	mkdir-5.2.1	8,321	20.06	1,320	8,321	18.53	1,429	0.00%	1.08	7.63%
	rm-8.4	7,461	10.14	3,650	7,427	11.46	3,232	-0.46%	0.88	-12.91%
	chown-8.2	7,682	9.78	3,700	7,445	9.83	3,704	-3.18%	0.99	0.11%
	grep-2.19	110,025	1.63	-	114,754	1.20	-	4.12%	1.37	-
	bzip2-1.0.5	52,234	1.69	-	51,123	1.80	-	-2.17%	0.94	-
	sort-8.16	51,800	3.36	-	51,848	3.35	-	0.09%	1.00	-
	gzip-1.2.4	16,653	2.71	-	16,548	2.72	-	-0.63%	1.00	-
	uniq-8.16	14,045	5.34	9,322	14,045	5.39	9,242	0.00%	0.99	-0.87%
	date-8.21	20,219	3.68	9,019	20,219	3.35	9,920	0.00%	1.10	9.08%
	tar-1.14	58,374	9.71	-	58,374	9.71	-	0.00%	1.00	-
	总览	7,807	20	1,727	8,971	30	1,058	12.98%	0.67	-63.23%

更大的返回结果同时消耗了更长的处理时间。在冗余子函数被去掉后，面对相同的输入，在一次失败的情况下，D-covProbDD 中的反馈函数可以执行更多种类的检查，导致处理时间更长。同时，由于在不同类型的检查上导致了更多的失败情况，相应元素的概率会被更快速地增加，导致最终返回结果大小变大。值得注意的是，虽然结果变大了，但从绝对数值上来看是可以接受的，比如，在处理项目 gcc-70586 时，D-covProbDD 仅产生了 114 个更多的标记数，仅占原始大小的 0.05%。考虑到处理时间，D-covProbDD 在五个项目上消耗了更长的处理时间，其中四个项目上 D-covProbDD 返回了一样或更小的结果。D-covProbDD 最大可以获得 2.86 的加速比。

在缓解软件膨胀领域，D-covProbDD 仅在一个项目上，即 chown-8.2，产生了更大的返回结果（237 个标记数），但节约了 0.84% 的处理时间。D-covProbDD 在该领域中所有项目上都获得了更短的处理时间，最大可获得 2.12 的加速比。

表 6.3　D-covProbDD 和 PDD 之间的比较：详细数据

D	项目名	Dc-CHISEL			p-CHISEL					
		R_{dc}	S_{dc}	T_{dc}	R_p	S_p	T_p	$\uparrow_R$	$\times S$	$\uparrow_T$
编译器测试	clang-22382	3,416	41.07	160	4,028	39.73	150	15.19%	1.03	-6.67%
	clang-22704	1,823	52.30	3,492	1,224	62.81	2,917	-48.94%	0.83	-19.71%
	clang-23309	4,038	36.65	798	7,267	25.87	1,006	44.43%	1.42	20.66%
	clang-23353	7,285	29.17	785	7,698	32.41	694	5.37%	0.90	-13.12%
	clang-25900	2,845	66.83	1,138	2,988	59.78	1,270	4.79%	1.12	10.39%
	clang-26760	5,241	96.60	2,115	4,970	89.49	2,286	-5.45%	1.08	7.48%
	clang-27137	8,855	157.25	1,053	14,400	55.08	2,907	38.51%	2.86	63.76%
	clang-27747	6,260	225.48	743	6,270	233.68	717	0.16%	0.96	-3.64%
	clang-31259	3,835	74.79	601	4,166	70.50	633	7.95%	1.06	5.04%
	gcc-59903	6,657	73.80	690	6,602	69.09	737	-0.83%	1.07	6.49%
	gcc-60116	9,029	94.28	702	9,453	74.18	886	4.49%	1.27	20.81%
	gcc-61383	9,066	28.81	811	9,702	20.50	1,109	6.56%	1.41	26.85%
	gcc-61917	12,524	108.94	668	13,691	75.67	947	8.52%	1.44	29.41%
	gcc-64990	4,032	330.44	438	6,970	160.92	882	42.15%	2.05	50.29%
	gcc-65383	4,327	137.89	287	5,065	97.78	397	14.57%	1.41	27.74%
	gcc-66186	5,639	183.12	228	6,447	65.96	622	12.53%	2.78	63.27%
	gcc-66375	4,572	180.33	337	5,169	70.57	854	11.55%	2.56	60.48%
	gcc-70127	12,975	149.10	951	12,452	105.71	1,346	-4.20%	1.41	29.37%
	gcc-70586	8,497	149.12	1,366	8,383	224.41	908	-1.36%	0.66	-50.40%
	gcc-71626	639	152.76	24	639	262.80	14	0.00%	0.58	-72.03%
缓解软件膨胀	mkdir-5.2.1	8,300	18.62	1,423	8,321	18.53	1,429	0.25%	1.01	0.42%
	rm-8.4	7,400	12.77	2,902	7,427	11.46	3,232	0.36%	1.11	10.23%
	chown-8.2	7,682	9.85	3,673	7,445	9.83	3,704	-3.18%	1.00	0.84%
	grep-2.19	100,324	2.53	-	114,754	1.20	-	12.57%	2.12	-
	bzip2-1.0.5	51,123	1.80	-	51,123	1.80	-	0.00%	1.00	-
	sort-8.16	51,848	3.35	-	51,848	3.35	-	0.00%	1.00	-
	gzip-1.2.4	16,548	2.72	-	16,548	2.72	-	0.00%	1.00	-
	uniq-8.16	14,045	5.53	9,006	14,045	5.39	9,242	0.00%	1.03	2.55%
	date-8.21	20,211	4.59	7,237	20,219	3.35	9,920	0.04%	1.37	27.05%
	tar-1.14	58,209	9.73	-	58,374	9.71	-	0.28%	1.00	-
	总览	8,340	37	872	8,971	30	1,058	7.03%	1.23	17.58%

RQ3: 总的来说，对 D-covProbDD 的评估结果表明，其在效率和效果上改善了 PDD，消耗了 7.03% 更少的时间同时返回了 17.58% 更小的结果。从整体上来看，结合细粒度反馈，并将反馈函数中的子函数去冗余化，能够进一步提升概率化差异调试技术的效率和效果。

6.3.4　covProbDD 和 T-PDD 之间的比较

表6.4显示了 covProbDD 与 T-PDD 之间比较的详细数据。从整体上看，Oc-T-PDD 获得了平均 36 个标记每秒的缩减速度，平均处理时间 1,777 秒，平均返回结果大小 215。相比于 T-PDD，处理时间缩减了 12.46%，获得 2.72% 更小的返回结果大小，在

表 6.4　covProbDD 和 T-PDD 之间的比较：详细数据

D	项目名	Oc-T-PDD			T-PDD					
		R_{oc}	S_{oc}	T_{oc}	R_T	S_T	T_T	$\uparrow_R$	$\times S$	$\uparrow_T$
编译器测试	clang-22382	182	17.38	564	242	15.92	612	24.79%	1.09	7.84%
	clang-22704	141	37.69	4,890	128	35.62	5,174	-10.16%	1.06	5.49%
	clang-23309	587	9.9	3,305	515	9.05	3,625	-13.98%	1.09	8.83%
	clang-23353	169	27.22	1,103	257	24.32	1,231	34.24%	1.12	10.4%
	clang-25900	296	58.75	1,339	272	51.43	1,530	-8.82%	1.14	12.48%
	clang-26760	161	91.33	2,293	263	76.06	2,752	38.78%	1.20	16.68%
	clang-27137	222	21.03	8,287	200	19.65	8,873	-11.0%	1.07	6.6%
	clang-27747	148	150.77	1,152	182	131.46	1,321	18.68%	1.15	12.79%
	clang-31259	351	23.33	2,077	351	21.35	2,269	0.0%	1.09	8.46%
	gcc-59903	237	23.64	2,426	216	19.88	2,886	-9.72%	1.19	15.94%
	gcc-60116	463	14.67	5,097	436	13.67	5,470	-6.19%	1.07	6.82%
	gcc-61383	374	9.27	3,461	368	8.45	3,796	-1.63%	1.10	8.83%
	gcc-61917	177	38.27	2,226	286	35.23	2,415	38.11%	1.09	7.83%
	gcc-64990	260	74.22	2,003	255	66.05	2,251	-1.96%	1.12	11.02%
	gcc-65383	181	37.69	1,161	170	33.04	1,325	-6.47%	1.14	12.38%
	gcc-66186	375	24.74	1,904	343	17.69	2,664	-9.33%	1.40	28.53%
	gcc-66375	475	18.34	3,545	444	16.79	3,874	-6.98%	1.09	8.49%
	gcc-70127	29	199.47	776	22	170.29	909	-31.82%	1.17	14.63%
	gcc-70586	339	60.86	3,482	291	47.26	4,485	-16.49%	1.29	22.36%
	gcc-71626	51	108.65	40	51	88.69	49	0.0%	1.23	18.37%
	总览	215	36	1,777	221	32	2,030	2.72%	1.13	12.46%

缩减速度上获得 1.13 加速比。

具体地，在编译器测试领域，在返回结果大小上 covProbDD 有 13 个项目不如 T-PDD，然而 covProbDD 都使用了更短的处理时间。虽然在某些项目上，covProbDD 返回结果增大的百分比较大，但绝对数值的差异较小。例如，在项目 gcc-70127 中，covProbDD 返回 29 个标记数的结果，T-PDD 返回 22 个标记数的结果，相差了 7 个标记数，百分比却达到了 31.82%。

RQ4: 总的来说，对 covProbDD 的评估结果表明，其在效率和效果上改善了 T-PDD，消耗了 12.46% 更少的时间同时返回了 2.72% 更小的结果。从整体上来看，不改变反馈函数的情况下，利用细粒度反馈能够进一步提高概率化差异调试技术的效率和效果。

6.3.5　P-covProbDD 和 T-PDD 之间的比较

表6.5显示了P-covProbDD与T-PDD之间对比的详细数据。从总体上看，P-covProbDD获得了平均 126 个标记数的返回结果，平均耗时 2,191 秒，产生 29 个标记每秒钟的缩

表 6.5　P-covProbDD 和 T-PDD 之间的比较：详细数据

D	项目名	Pc-T-PDD			T-PDD			↑$_R$	×S	↑$_T$
		R_{pc}	S_{pc}	T_{pc}	R_T	S_T	T_T			
编译器测试	clang-22382	179	13.81	710	242	15.92	612	26.03%	0.87	-16.01%
	clang-22704	118	12.43	14,829	128	35.62	5,174	7.81%	0.35	-186.61%
	clang-23309	27	9.67	3,441	515	9.05	3,625	94.76%	1.07	5.08%
	clang-23353	148	14.24	2,110	257	24.32	1,231	42.41%	0.59	-71.41%
	clang-25900	287	51.12	1,539	272	51.43	1,530	-5.51%	0.99	-0.59%
	clang-26760	191	78.81	2,657	263	76.06	2,752	27.38%	1.04	3.45%
	clang-27137	79	19.03	9,166	200	19.65	8,873	60.5%	0.97	-3.3%
	clang-27747	188	113.42	1,531	182	131.46	1,321	-3.3%	0.86	-15.9%
	clang-31259	347	19.71	2,458	351	21.35	2,269	1.14%	0.92	-8.33%
	gcc-59903	230	26.54	2,161	216	19.88	2,886	-6.48%	1.34	25.12%
	gcc-60116	133	14.61	5,139	436	13.67	5,470	69.5%	1.07	6.05%
	gcc-61383	368	8.32	3,854	368	8.45	3,796	0.0%	0.98	-1.53%
	gcc-61917	183	36.75	2,318	286	35.23	2,415	36.01%	1.04	4.02%
	gcc-64990	86	66.48	2,239	255	66.05	2,251	66.27%	1.01	0.53%
	gcc-65383	156	33.42	1,310	170	33.04	1,325	8.24%	1.01	1.13%
	gcc-66186	337	20.84	2,262	343	17.69	2,664	1.75%	1.18	15.09%
	gcc-66375	32	16.93	3,866	444	16.79	3,874	92.79%	1.01	0.21%
	gcc-70127	22	166.27	931	22	170.29	909	0.0%	0.98	-2.42%
	gcc-70586	123	74.23	2,858	291	47.26	4,485	57.73%	1.57	36.28%
	gcc-71626	51	48.83	89	51	88.69	49	0.0%	0.55	-81.63%
	总览	126	29	2,191	221	32	2,030	42.99%	0.91	-7.93%

减速度，相比于 T-PDD，在缩减大小上返回了比 T-PDD 小平均约 42.99% 的结果，然而却要消耗多 7.93% 的时间。结果表明，P-covProbDD 以并行处理子函数的方式并不能够带来效率的提升，这种方式的时间瓶颈在于独立的子函数中耗时最久的子函数。然而，这样的子函数在串行执行的方式中，当前驱的子函数返回失败时，往往会被直接设置为失败并返回整个反馈函数。

具体地，在编译器测试领域中，P-covProbDD 在 14 个项目上获得了更小的返回结果，以及 10 个项目上消耗了了更短的处理时间，在 9 个项目上使用了更短的时间返回了更小的结果。可以发现，在编译器测试领域中，P-covProbDD 与 T-PDD 差距较大。由于输入通常是随机产生的较大的 C 程序，编译、运行的时间通常较长，导致并行子函数后的瓶颈拖累了整体的效果。

RQ5：总的来说，对 P-covProbDD 的评估结果表明，其仅在效果上改善了 T-PDD。相比于 T-PDD，P-covProbDD 能够返回 42.99% 更小的结果，但要消耗 7.93% 更多的时间。从整体上来看，将反馈函数中的子函数调整为并行方式，虽然更符合本章的技术假设，产生了更小的结果，但由于子函数并行的瓶颈在于正常情况下执

表 6.6　D-covProbDD 和 T-PDD 之间的比较：详细数据

D	项目名	Dc-T-PDD			T-PDD					
		R_{dc}	S_{dc}	T_{dc}	R_T	S_T	T_T	$\uparrow_R$	$\times S$	$\uparrow_T$
编译器测试	clang-22382	137	21.79	452	242	15.92	612	43.39%	1.37	26.14%
	clang-22704	73	160.88	1146	128	35.62	5,174	42.97%	4.52	77.85%
	clang-23309	30	22.37	1488	515	9.05	3,625	94.17%	2.47	58.95%
	clang-23353	96	42.76	704	257	24.32	1,231	62.65%	1.76	42.81%
	clang-25900	352	78.92	996	272	51.43	1,530	-29.41%	1.54	34.9%
	clang-26760	97	130.93	1,600	263	76.06	2,752	63.12%	1.72	41.86%
	clang-27137	82	40.05	4,356	200	19.65	8,873	59.0%	2.04	50.91%
	clang-27747	177	180.9	960	182	131.46	1,321	2.75%	1.38	27.33%
	clang-31259	319	39.51	1,227	351	21.35	2,269	9.12%	1.85	45.92%
	gcc-59903	281	32.37	1,770	216	19.88	2,886	-30.09%	1.63	38.67%
	gcc-60116	44	34.55	2,176	436	13.67	5,470	89.91%	2.53	60.22%
	gcc-61383	268	14.15	2,275	368	8.45	3,796	27.17%	1.67	40.07%
	gcc-61917	134	54.25	1,571	286	35.23	2,415	53.15%	1.54	34.95%
	gcc-64990	111	132.76	1,121	255	66.05	2,251	56.47%	2.01	50.2%
	gcc-65383	162	53.13	824	170	33.04	1,325	4.71%	1.61	37.81%
	gcc-66186	304	70.94	665	343	17.69	2,664	11.37%	4.01	75.04%
	gcc-66375	31	113.05	579	444	16.79	3,874	93.02%	6.73	85.05%
	gcc-70127	329	34.65	4,459	22	170.29	909	-1395.45%	0.20	-390.54%
	gcc-70586	329	68.28	3,104	291	47.26	4,485	-13.06%	1.45	30.79%
	gcc-71626	51	96.58	45	51	88.69	49	0.0%	1.09	8.16%
	总览	130	57	1,139	221	32	2,030	41.18%	1.78	43.89%

行不到的十分耗时的子函数，导致效率上并没有改善现有概率化差异调试技术。

6.3.6　D-covProbDD 和 T-PDD 之间的比较

表6.6显示了 D-covProbDD 与 T-PDD 之间比较的详细数据。从整体平均来看，D-covProbDD 获得了 130 个标记数的返回结果，消耗了 1,139 秒处理时间，获得了 57 个标记数每秒的缩减速率，相比于 T-PDD 产生了 41.18% 更小的返回结果，消耗了 43.89% 更短的处理时间，获得了平均 1.78 的缩减加速比。

具体地，在编译器测试领域，D-covProbDD 在四个项目上产生了比 T-PDD 更大的返回结果，但在其中三个项目上消耗了更短的时间；D-covProbDD 只在一个项目上产生了更大的返回结果同时消耗了更长的处理时间。在冗余子函数被去掉后，面对相同的输入，在一次失败的情况下，D-covProbDD 中的反馈函数可以执行更多种类的检查，导致处理时间更长。同时，由于在不同类型的检查上导致了更多的失败情况，相应元素的概率会被更快速地增加，导致最终返回结果大小变大。考虑到处理时间，D-covProbDD 仅在一个项目上消耗了更长的处理时间。D-covProbDD 最大可以获得 6.73 的加速比。

表 6.7　covProbDD、P-covProbDD 和 D-covProbDD 之间的比较：面向集合

D	项目名	Oc-CHISEL			Pc-CHISEL			Dc-CHISEL		
		R_{oc}	S_{oc}	T_{oc}	R_{pc}	S_{pc}	T_{pc}	R_{dc}	S_{dc}	T_{dc}
编译器测试	clang-22382	4,028	32.56	183	4,028	33.69	176	**3,416**	**41.07**	**160**
	clang-22704	**1,225**	**62.85**	**2,915**	3,051	17.94	10,108	1,823	52.30	3,492
	clang-23309	7,267	22.79	1,142	7,267	13.99	1,861	**4,038**	**36.65**	**798**
	clang-23353	7,432	**32.34**	**704**	7,722	12.60	1,784	**7,285**	29.17	785
	clang-25900	2,988	51.61	1,472	3,012	30.92	2,455	**2,845**	**66.83**	**1,138**
	clang-26760	5,241	103.72	1,970	**4,311**	**126.99**	**1,616**	5,241	96.60	2,115
	clang-27137	14,365	54.91	2,917	**3,864**	78.37	2,177	8,855	**157.25**	**1,053**
	clang-27747	**6,124**	**245.20**	**684**	6,334	116.49	1,437	6,260	225.48	743
	clang-31259	4,166	66.95	666	**492**	70.30	687	3,835	**74.79**	**601**
	gcc-59903	**6,602**	**77.22**	**660**	7,614	42.78	1,168	6,657	73.80	690
	gcc-60116	9,445	76.67	858	9,445	48.38	1,359	**9,029**	**94.28**	**702**
	gcc-61383	9,702	15.00	1,516	9,726	3.24	7,012	**9,066**	**28.81**	**811**
	gcc-61917	13,691	80.35	892	**11,008**	34.59	2,149	12,524	**108.94**	**668**
	gcc-64990	6,870	150.35	944	7,971	71.72	1,965	**4,032**	**330.44**	**438**
	gcc-65383	5,065	99.18	392	5,065	47.45	819	**4,327**	**137.89**	**287**
	gcc-66186	6,447	66.72	615	6,447	32.72	1,254	**5,639**	**183.12**	**228**
	gcc-66375	5,169	71.47	844	5,169	38.59	1,562	**4,572**	**180.33**	**337**
	gcc-70127	**2,527**	**154.00**	988	2,527	82.33	1,849	12,975	149.10	**951**
	gcc-70586	**8,383**	**212.81**	**958**	8,497	96.90	2,102	8,497	149.12	1,366
	gcc-71626	639	**282.56**	**13**	**633**	97.77	38	639	152.76	24
缓解软件膨胀	mkdir-5.2.1	8,321	**20.06**	**1,320**	8,321	**20.06**	**1,320**	**8,300**	18.62	1,423
	rm-8.4	**7,400**	11.83	3,132	7,461	10.14	3,650	**7,400**	**12.77**	**2,902**
	chown-8.2	**7,682**	**9.88**	**3,662**	7,682	9.78	3,700	7,682	9.85	3,673
	grep-2.19	110,025	1.63	-	110,025	1.63	-	**100,324**	**2.53**	-
	bzip2-1.0.5	**51,123**	**1.80**	-	52,234	1.69	-	**51,123**	**1.80**	-
	sort-8.16	51,993	3.34	-	**51,800**	**3.36**	-	51,848	3.35	-
	gzip-1.2.4	16,653	2.71	-	16,653	2.71	-	**16,548**	**2.72**	-
	uniq-8.16	**14,045**	5.42	9,190	14,045	5.34	9,322	14,045	**5.53**	**9,006**
	date-8.21	**20,211**	3.80	8,736	20,219	3.68	9,019	**20,211**	**4.59**	**7,237**
	tar-1.14	58,538	9.70	-	58,374	9.71	-	**58,209**	**9.73**	-
	总览	8,499	31	1,054	**7,807**	20	1,727	8,340	**37**	**872**

RQ6：总的来说，对 D-covProbDD 的评估结果表明，其在效率和效果上改善了 T-PDD，消耗了 43.89% 更少的时间同时返回了 41.18% 更小的结果。从整体上来看，结合细粒度反馈，并将反馈函数中的子函数去冗余化，能够进一步提升概率化差异调试技术的效率和效果。

6.3.7　covProbDD、P-covProbDD 和 D-covProbDD 之间的比较

针对面向集合的概率模型，表6.7显示了 covProbDD、P-covProbDD 和 D-covProbDD 在各个项目上的详细结果。从整体来看，P-covProbDD 获得了 7,807 个标记数的平均返回结果，是三种技术中效果最好的。时间上来看，D-covProbDD 获得了 872 秒的平均处理时间，平均每秒缩减 37 个标记数，是三种技术中效率最高的。此外，在表6.7中，

表 6.8　covProbDD、P-covProbDD 和 D-covProbDD 之间的比较：面向语法树

D	项目名	Oc-T-PDD			Pc-T-PDD			Dc-T-PDD		
		R_{oc}	S_{oc}	T_{oc}	R_{pc}	S_{pc}	T_{pc}	R_{dc}	S_{dc}	T_{dc}
编译器测试	clang-22382	182	17.38	564	179	13.81	710	**137**	**21.79**	**452**
	clang-22704	141	37.69	4,890	118	12.43	14,829	**73**	**160.88**	**1,146**
	clang-23309	587	9.9	3,305	**27**	9.67	3,441	30	**22.37**	**1,488**
	clang-23353	169	27.22	1,103	148	14.24	2,110	**96**	**42.76**	**704**
	clang-25900	296	58.75	1,339	**287**	51.12	1,539	352	**78.92**	**996**
	clang-26760	161	91.33	2,293	191	78.81	2,657	**97**	**130.93**	**1,600**
	clang-27137	222	21.03	8,287	**79**	19.03	9,166	82	**40.05**	**4,356**
	clang-27747	**148**	150.77	1,152	188	113.42	1,531	177	**180.9**	**960**
	clang-31259	351	23.33	2,077	347	19.71	2,458	**319**	**39.51**	**1,227**
	gcc-59903	237	23.64	2,426	**230**	26.54	2,161	281	**32.37**	**1,770**
	gcc-60116	463	14.67	5,097	133	14.61	5,139	**44**	**34.55**	**2,176**
	gcc-61383	374	9.27	3,461	368	8.32	3,854	**268**	**14.15**	**2,275**
	gcc-61917	177	38.27	2,226	183	36.75	2,318	**134**	**54.25**	**1,571**
	gcc-64990	260	74.22	2,003	**86**	66.48	2,239	111	**132.76**	**1,121**
	gcc-65383	181	37.69	1,161	**156**	33.42	1,310	162	**53.13**	**824**
	gcc-66186	375	24.74	1,904	337	20.84	2,262	**304**	**70.94**	**665**
	gcc-66375	475	18.34	3,545	32	16.93	3,866	**31**	**113.05**	**579**
	gcc-70127	29	**199.47**	**776**	**22**	166.27	931	329	34.65	4,459
	gcc-70586	339	60.86	3,482	**123**	**74.23**	**2,858**	329	68.28	3,104
	gcc-71626	**51**	**108.65**	**40**	**51**	48.83	89	**51**	96.58	45
	总览	215	36	1,777	**125**	29	2,191	130	**57**	**1,139**

对于每一个项目中的每一个指标，最好的数值被加粗显示。从整体来看，D-covProbDD 在所有项目中获得了最多的最好指标，然而在某些项目上，covProbDD 和 P-covProbDD 也能在某些指标上获得最好的效果。可见，三种技术各有优劣，平均来看，推荐使用 D-covProbDD 作为最好的差异调试工具，然而，需要人工分析反馈函数，去除冗余的子函数再进行差异调试。如果期望在待处理的项目上获得更小的结果，推荐使用 P-covProbDD，同样也需要分析独立的子函数并作并行处理，但根据本章实验评估的经验来看，人工消耗要比 D-covProbDD 小得多。值得注意的是，平均来看，covProbDD 和 D-covProbDD 均获得了比 PDD 更好的效果和效率；P-covProbDD 虽然效率上不如 PDD，但能够获得比 PDD 产生结果小 12.98% 的返回结果，在三种技术中其在效果上的提升最为明显。

考虑针对面向语法树的概率模型，表6.8显示了 covProbDD、P-covProbDD 和 D-covProbDD 在各个项目上的详细结果。由于面向语法树的差异调试技术无法被应用于缓解软件膨胀应用领域中的部分项目，本章仅在编译器测试应用领域中进行了验证，可以得出与面向集合的概率模型的类似结论，即 P-covProbDD 技术可以获得最小的返回

结果大小，D-covProbDD 可以获得最高效率，综合返回结果大小和效率来看，covProbDD 和 D-covProbDD 均获得了比 PDD 和 T-PDD 更好的效果和效率。

RQ7：本章提出的基于细粒度反馈的概率化差异调试技术能够进一步提升概率化差异调试技术的效率和效果。此外，针对反馈函数中子函数的不同组织形式，本章验证结果表明，将子函数去冗余化的基于细粒度反馈的概率化差异调试技术整体上的性能表现最好。从效果上来看，将子函数并行化的基于细粒度反馈的概率化差异调试技术的提升最为明显，因为子函数的这种执行方式更符合本章技术的假设。

6.3.8　可能威胁有效性的因素

为了解决内部有效性的威胁，本章努力确保 covProbDD 及其变体的实现和实验脚本的正确性。本文作者进行了彻底的代码审查和测试，以验证实现的准确性和可靠性。通过仔细检查代码，旨在最小化可能影响本章研究内部有效性的错误或不一致性。

外部有效性的威胁主要与本章研究中使用的项目和比较的技术有关。为了解决这个问题，采用了现有研究工作中常用和公认的项目。在未来，计划通过加入来自各种类型的上下文无关文法的其它项目来进一步扩展评估。关于比较的技术，由于本章提出技术仅和具体概率化差异调试技术中使用到的概率模型相关，本章分别验证了本章技术在面向集合的概率模型和面向语法树的概率模型上的改进效果。

对于构建有效性的威胁主要来自于实验设置中固有的随机性。随机性可能会导致 covProbDD、P-covProbDD 和 D-covProbDD 的性能以及与之进行比较的面向集合的概率化差异调试技术的性能出现不可控制的波动。为了减轻这种威胁，对每个项目多次运行所涉及的技术，具体地，进行了 5 次重复实验。通过计算平均结果，旨在尽可能地降低或消除随机性的影响并获得更可靠的性能指标。

通过认识和解决这些对有效性的威胁，旨在提高本章所涉及研究的严谨性和可信度。通过仔细的实现、项目选择和随机性影响消融策略，努力确保本章中发现的有效性，并为与面向集合的概率化差异调试技术相比，covProbDD 及其变体技术的性能提供有价值的分析和讨论。

6.4　讨论与小结

现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。本章旨在精化反馈函数提供的反馈信息，结合细粒度反馈，以期望此举能够为概率化差异调试带来性能上的进一步提升。

结合本文第三章提出的概率化差异调试框架，本章通过精化反馈函数提供的反馈信息，提出了基于细粒度反馈的概率化差异调试技术 covProbDD。根据反馈函数中子函数的不同组织方式，进而提出了基于子函数并行的变体技术 P-covProbDD 和基于子函数去冗余化的变体技术 D-covProbDD。在实验评估环节，本章利用了包含 30 个项目的、被差异调试研究中大量使用的公开数据集。根据反馈函数的种类，本章分别在编译器测试领域和缓解软件膨胀领域对所提出的技术进行了验证。为了验证本章技术在第五章提出的面向集合的概率模型上的效果，根据反馈函数的原有执行、并行执行和去冗余执行，分别实现了 Oc-CHISEL、Pc-CHISEL 和 Dc-CHISEL，并将它们分别应用到两个应用领域的 30 个项目中。为了验证本章技术在第六章提出的面向语法树的概率模型上的效果，根据反馈函数的原有执行、并行执行和去冗余执行，分别实现了 Oc-tech、Pc-T-PDD 和 Dc-T-PDD，并将它们应用到编译器测试应用领域的 20 个项目中。实验表明，结合细粒度反馈确实能够带来概率化差异调试技术性能的进一步提升，从整体平均来看，本章提出的三种技术中，D-covProbDD 表现最好，相比于面向集合的概率化差异调试技术，能够减小 7.03% 的返回结果大小，节省 17.58% 的处理时间；相比于面向语法树的概率化差异调试技术，能够减小 41.18% 的返回结果大小，节省 43.89% 的处理时间。covProbDD 和 P-covProbDD 对于差异调试性能也有一定程度的提高，然而，具体在某些项目上的表现不如 D-covProbDD。

第七章　总结和展望

7.1　本文工作总结

差异调试问题是软件工程中的重要问题，其核心目标在于在满足规定性质的同时，减小一个集合的大小。该技术被广泛应用于多个领域，包括编译器开发、回归故障定位以及缓解软件膨胀问题。在本质上，差异调试问题可以看作是一个搜索问题，即在庞大的解空间中寻找一个满足特定性质的最小解。与一般的搜索问题不同的是，差异调试问题要求返回的解必须满足特定的性质，因此在搜索过程中需要反复调用反馈函数以验证当前解是否满足规定性质。更好的差异调试技术意味着尽可能少地调用反馈函数的同时返回尽可能小的集合。

在第3.3.2节中，本文通过一个实际场景的示例，展示现有差异调试技术通过固定化方式尝试删除输入集合中元素的缺点。固定化的方式导致现有技术无法利用反馈函数提供当前集合不具有规定性质的失败反馈。此外，固定化的方式还容易导致现有技术陷入局部最优解中。实际上，从5.1节中的示例可以看出，这种固定化尝试删除元素还体现在对于差异调试反复调用的过程中，即为了得到更小规模的输出，反复调用差异调试达到最小不动点状态。

为了解决上述问题，本文提出概率化差异调试框架。根据给定的概率模型，指定待选择的集合，通过反馈函数提供的反馈信息更新模型，该过程直到模型收敛。通过结合概率的方式，概率化差异调试能够利用失败的反馈更新概率模型，更好地指导差异调试过程。由于概率化差异调试框架需要针对具体问题设计概率模型才能应用，本文针对集合作为输入和针对程序作为输入的差异调试的场景，分别提出了面向集合的概率模型和面向语法树的概率模型。有了上述两种概率模型，在应用中，概率化差异调试框架能够分别被实例化为面向集合的概率化差异调试技术和面向语法树的概率化差异调试技术。本文的实验验证结果证明了，它们在相应场景中均提高了差异调试的效率和效果。最后，本文提出了基于细粒度反馈的概率化差异调试技术，探索精化反馈函数提供的反馈信息能否进一步带来概率化差异调试技术性能上的提升。至此，本文所提出的一整套技术通过结合概率打破了现有技术固定化尝试删除元素带来的局限性，既能灵活地探索更大范围的搜索空间，又可以充分地利用反馈信息指导差异调试过程，最终达到提升解决差异调试问题技术的效率和效果的目的。

具体地，本文的主要研究工作及创新点如下：

1. **概率化差异调试框架**。本文提出了概率化差异调试框架。给定一个概率模型，该框架利用概率模型来估计每个元素被保留在最终结果中的概率。在每次迭代中，

根据概率模型选择一个最大化期望收益的元素集合，并通过反馈函数反馈该集合是否仍具有规定的性质。然后，根据反馈信息计算后验概率，并更新概率模型。直到达到收敛条件，算法停止并返回简化后的结果。进一步地，证明了当输入满足某些性质时，最小子集的存在性；同时，讨论了该框架产生结果的正确性。

2. **面向集合的概率模型**。概率化差异调试框架需要针对问题设计概率模型才能应用。大量差异调试应用场景中，输入可以看作是一个元素集合。为应用框架到这样的场景上，本文提出了面向集合的概率模型。为验证该模型的有效性，本文将该模型与框架结合，实例化为面向集合的概率化差异调试技术进行实验验证，验证结果表明，面向集合的概率化差异调试技术显著提高了差异调试的效果和效率。进一步地讨论了面向集合的概率化差异调试技术保证结果极小性或最小性的情形和最坏情况下调用反馈函数的次数。
3. **面向语法树的概率模型**。简化程序是差异调试最典型的应用之一。虽然面向集合的模型也可以用在程序上，但因为模型没有充分考虑程序结构上的依赖关系，应用效果有限。为了在差异调试中充分考虑语法结构上的依赖关系，本文提出了面向语法树的概率模型。为验证该模型的有效性，本文将该模型与框架结合，实例化为面向语法树的概率化差异调试技术进行实验验证，验证结果表明，面向语法树的概率化差异调试技术显著提高了差异调试的效果和效率。
4. **基于细粒度反馈的概率化差异调试技术**。现有差异调试技术将反馈函数看作二元的，只提供通过和失败两种反馈信息，但实践中的反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。基于这个观察，本文提出了基于细粒度反馈的概率化差异调试技术，探索精化反馈函数提供的反馈信息能否进一步带来概率化差异调试技术性能上的提升，支持在不修改概率模型的情况下引入细粒度反馈。由于反馈函数内部包含不同功能的子函数，根据子函数间的组织特点，进而提出两种变体技术。本文的实验验证结果表明，精化反馈函数提供的反馈信息能够进一步提升概率化差异调试技术的效率和效果。这些技术可在现有概率化差异调试技术引入细粒度反馈，以便进一步提升差异调试的效率和效果。

综上，本文提出了一系列技术，形成了一套概率化差异调试技术。该系列技术在公认的差异调试数据集和本文收集的编译器调试数据集上超过了最优技术。本文提出的技术已经成为后续工作的基础，已经有多个第三方团队在本文技术和方法的基础上构建了更先进的领域特定的差异调试技术。

7.2　未来工作展望

本文提出的概率化差异调试框架及相关技术虽然在一定程度上改善了差异调试技术，但仍存在着较大的提升空间。特别的是，本文提出基于细粒度反馈的概率化差异调试技术，为后续针对差异调试的研究提供了更多的借鉴。具体地，未来研究可以从以下几个方面展开。

1. **与其它概率化方法的结合**。除了概率模型，现代化的概率化方法的主要代表是深度学习，具体地说是以 OpenAI 公司所研发的 ChatGPT 技术为代表的大模型技术。前文中已提到，由于差异调试问题的特殊性，暂不存在供深度学习模型使用的大数据；根据相关工作中的实证研究结果，针对一种缺陷的历史数据未必对处理另一种缺陷有增益性的帮助；此外，深度学习模型针对大规模输入的模型推理较为耗时。因此，结合现阶段差异调试技术的应用背景，深度学习不适用于处理此类问题。然而，软件开发正飞速发展，软件相关的研究也突飞猛进。相信随着相关领域的快速发展，差异调试问题即将出现应用深度学习大模型应用的曙光。
2. **概率模型领域的优化算法**。本文的技术框架极大地依赖于概率编程领域，具体涉及概率模型的设计和概率模型推断等相关领域。现有概率模型相关领域中的算法还有较大缺陷，如本文第三章中提及的推断效率问题，本文根据差异调试问题，提出相对应的近似优化算法，使得本文提出的技术不会受到上述缺陷的限制。随着该领域中相关算法的演化和成熟，相信本文所提出的技术也会相应被提升或具有较大提升空间。
3. **结合细粒度反馈的差异调试技术**。差异调试技术相关的研究仅把反馈函数作为二元函数。本文观察到，反馈函数往往由多段更小的反馈函数复合而成，这些更小函数的执行结果可以提供更细粒度的反馈信息。本文第六章的验证结果表明，结合细粒度反馈，差异调试技术的效率和效果能够被进一步提高。期待本文更够启发出更多的基于细粒度反馈的差异调试技术。
4. **新的实践场景**。差异调试技术在测试用例简化、测试集合简化、缓解软件膨胀问题等领域有了成功实践。未来工作还可以在新的领域中应用差异调试技术，用于更多更复杂的软件调试环境，有针对性地对不同的应用场景提出适应性的技术，有效地应对不同应用场景中面临的挑战。

参考文献

[1] A. Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: *ACM SIGSOFT Software Engineering Notes* 24.6 (**1999**): pp. 253-267.

[2] A. Zeller. “Isolating cause-effect chains from computer programs”. In: *ACM SIGSOFT Software Engineering Notes* 27.6 (**2002**): pp. 1-10.

[3] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, **2009**.

[4] A. Zeller, R. Hildebrandt. “Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (**2002**): pp. 183-200.

[5] J. Chen, G. Wang, D. Hao, et al. “History-guided configuration diversification for compiler test-program generation”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2019**: pp. 305-316.

[6] J. Chen, G. Wang, D. Hao, et al. “Coverage prediction for accelerating compiler testing”. In: *IEEE Transactions on Software Engineering* (**2018**).

[7] A. F. Donaldson, P. Thomson, V. Teliman, et al. “Test-case reduction and deduplication almost for free with transformation-based compiler testing”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. **2021**: pp. 1017-1032.

[8] G. Ye, Z. Tang, S. H. Tan, et al. “Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing”. In: *arXiv preprint* (**2021**). arXiv: 2104.07460 [cs.PL]. URL: https://arxiv.org/abs/2104.07460.

[9] A. Christi, M. L. Olson, M. A. Alipour, et al. “Reduce before you localize: Delta-debugging and spectrum-based fault localization”. In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. **2018**: pp. 184-191.

[10] J. Chen, J. Han, P. Sun, et al. “Compiler bug isolation via effective witness test program generation”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2019**: pp. 223-234.

[11] H. Cleve, A. Zeller. “Locating causes of program failures”. In: *Proceedings of 27th International Conference on Software Engineering, 2005. ICSE 2005.* **2005**: pp. 342-351.

[12] S. Kim, T. Zimmermann, K. Pan, et al. “Automatic identification of bug-introducing changes”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. **2006**: pp. 81-90.

[13] K. Heo, W. Lee, P. Pashakhanloo, et al. “Effective program debloating via reinforcement learning”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. **2018**: pp. 380-394.

[14] G. Misherghi, Z. Su. “HDD: hierarchical delta debugging”. In: *Proceedings of the 28th International Conference on Software Engineering*. **2006**: pp. 142-151.

[15] 李征, 吴永豪, 王海峰, 等. 软件多缺陷定位方法研究综述[J]. 计算机学报, **2022**, 45(2).

[16] W. E. Wong, R. Gao, Y. Li, et al. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (**2016**): pp. 707-740.

[17] M. A. Alipour. "Automated fault localization techniques: a survey". In: *Oregon State University* 54.3 (**2012**).

[18] F. Tip. *A survey of program slicing techniques*. Centrum Voor Wiskunde En Informatica Amsterdam, **1994**.

[19] M. Weiser. "Programmers use slices when debugging". In: *Communications of the ACM* 25.7 (**1982**): pp. 446-452.

[20] 李必信. 程序切片技术及其应用[M]. 科学出版社, **2006**.

[21] M. Weiser. "Program slicing". In: *IEEE Transactions on Software Engineering* 4 (**1984**): pp. 352-357.

[22] R. Lyle. "Automatic program bug location by program slicing". In: *Proceedings 2nd International Conference on Computers and Applications*. **1987**: pp. 877-883.

[23] H. Agrawal, J. R. Horgan. "Dynamic program slicing". In: *ACM SIGPlan Notices* 25.6 (**1990**): pp. 246-256.

[24] B. Korel, J. Laski. "Dynamic program slicing". In: *Information Processing Letters* 29.3 (**1988**): pp. 155-163.

[25] H. Agrawal, R. A. DeMillo, E. H. Spafford. "Debugging with dynamic slicing and backtracking". In: *Software: Practice and Experience* 23.6 (**1993**): pp. 589-616.

[26] R. A. DeMillo, H. Pan, E. H. Spafford. "Critical slicing for software fault localization". In: *ACM SIGSOFT Software Engineering Notes* 21.3 (**1996**): pp. 121-134.

[27] B. Korel. "PELAS-program error-locating assistant system". In: *IEEE Transactions on Software Engineering* 14.9 (**1988**): pp. 1253-1260.

[28] X. Zhang, S. Tallam, N. Gupta, et al. "Towards locating execution omission errors". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. **2007**: pp. 415-424.

[29] X. Zhang, H. He, N. Gupta, et al. "Experimental evaluation of using dynamic slices for fault location". In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. **2005**: pp. 33-42.

[30] C. D. Sterling, R. A. Olsson. "Automated bug isolation via program chipping". In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. **2005**: pp. 23-32.

[31] C. Liu, X. Zhang, J. Han, et al. "Indexing noncrashing failures: A dynamic program slicing-based approach". In: *2007 IEEE International Conference on Software Maintenance*. **2007**: pp. 455-464.

[32] H. Agrawal, J. R. Horgan, S. London, et al. "Fault localization using execution slices and dataflow tests". In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. **1995**: pp. 143-151.

[33] B. Korel, S. Yalamanchili. “Forward computation of dynamic program slices”. In: *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*. **1994**: pp. 66-79.

[34] Á. Beszédes, T. Gergely, Z. M. Szabo, et al. “Dynamic slicing method for maintenance of large C programs”. In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. **2001**: pp. 105-113.

[35] T. Gyimóthy, A. Beszédes, I. Forgács. “An efficient relevant slicing method for debugging”. In: *ACM SIGSOFT Software Engineering Notes* 24.6 (**1999**): pp. 303-321.

[36] X. Zhang, R. Gupta, Y. Zhang. “Precise dynamic slicing algorithms”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* **2003**: pp. 319-329.

[37] X. Zhang, R. Gupta, Y. Zhang. “Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams”. In: *Proceedings of 26th International Conference on Software Engineering*. **2004**: pp. 502-511.

[38] 曹鹤玲, 姜淑娟, 鞠小林等. 基于动态切片和关联分析的错误定位方法[J]. 计算机学报, **2015**, 38(11): 2188-2202.

[39] 王新平, 顾庆, 陈翔等. 基于执行轨迹的软件缺陷定位方法研究[J]. 计算机科学, **2009**, 36(10): 168-171.

[40] W. E. Wong, Y. Qi. “Effective program debugging based on execution slices and inter-block data dependency”. In: *Journal of Systems and Software* 79.7 (**2006**): pp. 891-903.

[41] M. J. Harrold, G. Rothermel, K. Sayre, et al. “An empirical investigation of the relationship between spectra differences and regression faults”. In: *Software Testing, Verification and Reliability* 10.3 (**2000**): pp. 171-194.

[42] T. Reps, T. Ball, M. Das, et al. “The use of program profiling for software maintenance with applications to the year 2000 problem”. In: *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. **1997**: pp. 432-449.

[43] A B. Taha, S. M. Thebaut, S S. Liu. “An approach to software fault localization and revalidation based on incremental data flow analysis”. In: *Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. **1989**: pp. 527-534.

[44] B. Korel, J. Laski. “STAD-A system for testing and debugging: User perspective”. In: *Workshop on Software Testing, Verification, and Analysis*. **1988**: pp. 13-14.

[45] H. Agrawal, R. A. De Millo, E. H. Spafford. “An execution-backtracking approach to debugging”. In: *IEEE Software* 8.3 (**1991**): pp. 21-26.

[46] J. A. Jones, M. J. Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. **2005**: pp. 273-282.

[47] W. E. Wong, J. Li. “An integrated solution for testing and analyzing Java applications in an industrial setting”. In: *12th Asia-Pacific Software Engineering Conference (APSEC’05)*. **2005**: pp. 8-16.

[48] M. Zeng, Y. Wu, Z. Ye, et al. “Fault localization via efficient probabilistic modeling of program semantics”. In: *Proceedings of the 44th International Conference on Software Engineering*. **2022**: pp. 958-969.

[49] 陈翔, 鞠小林, 文万志等. 基于程序频谱的动态缺陷定位方法研究[J]. 软件学报, **2015**, 26(2): 390.

[50] 伍佳, 洪玫, 万莹等. 基于程序频谱的两阶段缺陷定位方法.[J]. 计算机应用研究, **2021**, 38(3).

[51] 王赞, 樊向宇, 邹雨果等. 一种基于遗传算法的多缺陷定位方法[J]. 软件学报, **2016**, 27(4): 879-900.

[52] 舒挺, 黄明献, 丁佐华等. 基于条件概率模型的缺陷定位方法[J]. 软件学报, **2017**, 29(6): 1756-1769.

[53] M. Renieres, S. P. Reiss. “Fault localization with nearest neighbor queries”. In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* **2003**: pp. 30-39.

[54] L. Guo, A. Roychoudhury, T. Wang. “Accurately choosing execution runs for software fault localization”. In: *Proceedings of 15th International Conference on Compiler Construction*. **2006**: pp. 80-95.

[55] B. Liblit, M. Naik, A. X. Zheng, et al. “Scalable statistical bug isolation”. In: *Acm Sigplan Notices* 40.6 (**2005**): pp. 15-26.

[56] 李伟, 郑征, 郝鹏等. 基于谓词执行序列的软件缺陷定位算法[D]. 2013.

[57] C. Liu, L. Fei, X. Yan, et al. “Statistical debugging: A hypothesis testing-based approach”. In: *IEEE Transactions on Software Engineering* 32.10 (**2006**): pp. 831-848.

[58] E. Wong, T. Wei, Y. Qi, et al. “A crosstab-based statistical method for effective fault localization”. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. **2008**: pp. 42-51.

[59] 杨波, 吴际, 刘超. 基于数据链的软件故障定位方法[J]. 软件学报, **2015**, 26(2): 254-268.

[60] T. Zimmermann, A. Zeller. “Visualizing memory graphs”. In: *Software Visualization: International Seminar Dagstuhl Castle, Germany, May 20–25, 2001 Revised Papers*. **2002**: pp. 191-204.

[61] N. Gupta, H. He, X. Zhang, et al. “Locating faulty code using failure-inducing chops”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. **2005**: pp. 263-272.

[62] X. Zhang, N. Gupta, R. Gupta. “Locating faults through automated predicate switching”. In: *Proceedings of the 28th International Conference on Software engineering*. **2006**: pp. 272-281.

[63] T. Wang, A. Roychoudhury. “Automated path generation for software fault localization”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. **2005**: pp. 347-351.

[64] F. Wotawa. “Debugging hardware designs using a value-based model”. In: *Applied Intelligence* 16 (**2002**): pp. 71-92.

[65] C C. Lee, P C. Chung, J R. Tsai, et al. “Robust radial basis function neural networks”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29.6 (**1999**): pp. 674-685.

[66] S. Nessa, M. Abedin, W. E. Wong, et al. “Software fault localization using n-gram analysis”. In: *Wireless Algorithms, Systems, and Applications: Third International Conference, WASA 2008, Dallas, TX, USA, October 26-28, 2008. Proceedings 3*. **2008**: pp. 548-559.

[67] P. Cellier, M. Ducassé, S. Ferré, et al. “Formal concept analysis enhances fault localization in software”. In: *International Conference on Formal Concept Analysis*. **2008**: pp. 273-288.

[68] P. Cellier, M. Ducassé, S. Ferré, et al. “Multiple Fault Localization with Data Mining.” In: *SEKE*. **2011**: pp. 238-243.

[69] T. Denmat, M. Ducassé, O. Ridoux. “Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. **2005**: pp. 396-399.

[70] W. E. Wong, Y. Qi. “BP neural network-based effective fault localization”. In: *International Journal of Software Engineering and Knowledge Engineering* 19.04 (**2009**): pp. 573-597.

[71] 张芸, 刘佳琨, 夏鑫等. 基于信息检索的软件缺陷定位技术研究进展[J]. 软件学报, **2020**, 31(8): 2432-2452.

[72] 常佩佩, 赵逢禹. 基于代码结构信息的软件缺陷定位方法研究.[J]. 计算机应用研究, **2016**, 33(8).

[73] 岳雷, 崔展齐, 陈翔等. 基于历史缺陷信息检索的语句级软件缺陷定位方法[J]. 软件学报, **2023**: 1-20.

[74] W. X. Zhao, K. Zhou, J. Li, et al. “A survey of large language models”. In: *arXiv preprint* (**2023**). arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[75] N. M. S. Surameery, M. Y. Shakor. “Use chat gpt to solve programming bugs”. In: *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3.01 (**2023**): pp. 17-22.

[76] 张军阳, 王慧丽, 郭阳等. 深度学习相关研究综述.[J]. 计算机应用研究, **2018**, 35(7).

[77] 卢经纬, 郭超, 戴星原等. 问答 ChatGPT 之后: 超大预训练模型的机遇和挑战[J]. 自动化学报, **2023**, 49(4): 705-717.

[78] Y. Li, S. Wang, T. Nguyen. “Fault localization with code coverage representation learning”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. **2021**: pp. 661-673.

[79] Y. Lou, Q. Zhu, J. Dong, et al. “Boosting coverage-based fault localization via graph-based representation learning”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2021**: pp. 664-676.

[80] 薄莉莉, 朱程, 李斌等. 文件信息增强的方法级软件缺陷定位[J]. 电子学报, **2023**, 51(3): 613.

[81] 郭肇强, 周慧聪, 刘释然等. 基于信息检索的缺陷定位: 问题, 进展与挑战[J]. 软件学报, **2020**, 31(9): 2826-2854.

[82] X. Zhou, X. Peng, T. Xie, et al. “Delta debugging microservice systems”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. **2018**: pp. 802-807.

[83] X. Zhou, X. Peng, T. Xie, et al. “Delta debugging microservice systems with parallel optimization”. In: *IEEE Transactions on Services Computing* 15.1 (**2019**): pp. 16-29.

[84] Y. Pei, A. Christi, X. Fern, et al. “Taming a Fuzzer Using Delta Debugging Trails”. In: *2014 IEEE International Conference on Data Mining Workshop*. **2014**: pp. 840-843.

[85] M. R. I. Rabin, V. J. Hellendoorn, M. A. Alipour. “Understanding neural code intelligence through program simplification”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2021**: pp. 441-452.

[86] U. Junker. “Quickxplain: Preferred explanations and relaxations for over-constrained problems”. In: *Proceedings of the 19th national Conference on Artifical intelligence*. **2004**: pp. 167-172.

[87] R. Brummayer, A. Biere. “Fuzzing and delta-debugging SMT solvers”. In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. **2009**: pp. 1-5.

[88] *Berkeley Delta*. Accessed: 2023. URL: http://delta.tigris.org/.

[89] C. Artho. “Iterative delta debugging”. In: *International Journal on Software Tools for Technology Transfer* 13.3 (**2011**): pp. 223-246.

[90] D. Vince. “Iterating the minimizing delta debugging algorithm”. In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. **2022**: pp. 57-60.

[91] D. Vince, Á. Kiss. “Cache Optimizations for Test Case Reduction”. In: *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. **2022**: pp. 442-453.

[92] Y. Tian, X. Zhang, Y. Dong, et al. “On the Caching Schemes to Speed Up Program Reduction”. In: *ACM Transactions on Software Engineering and Methodology* (**2023**).

[93] R. Hodován, Á. Kiss, T. Gyimóthy. “Tree preprocessing and test outcome caching for efficient hierarchical delta debugging”. In: *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. **2017**: pp. 23-29.

[94] R. Hodován, Á. Kiss. “Practical improvements to the minimizing delta debugging algorithm”. In: *International Conference on Software Engineering and Applications*. Vol. 2. **2016**: pp. 241-248.

[95] X. Yang, Y. Chen, E. Eide, et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. **2011**: pp. 283-294.

[96] C. Sun, Y. Li, Q. Zhang, et al. “Perses: syntax-guided program reduction”. In: *Proceedings of the 40th International Conference on Software Engineering*. **2018**: pp. 361-371.

[97] R. Hodován, Á. Kiss. “Modernizing hierarchical delta debugging”. In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. **2016**: pp. 31-37.

[98] *ANTLR4*. Accessed: 2021. URL: https://www.antlr.org.

[99] R. Hodován, Á. Kiss, T. Gyimóthy. “Coarse hierarchical delta debugging”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. **2017**: pp. 194-203.

[100] Á. Kiss, R. Hodován, T. Gyimóthy. “HDDr: a recursive variant of the hierarchical delta debugging algorithm”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. **2018**: pp. 16-22.

[101] J. Regehr, Y. Chen, P. Cuoq, et al. “Test-case reduction for C compiler bugs”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. **2012**: pp. 335-346.

[102] *The LLVM Compiler Infrastructure*. Accessed: 2023. URL: https://llvm.org.

[103] *Clang: a C language family frontend for LLVM*. Accessed: 2023. URL: https://clang.llvm.org.

[104] *JAVACC*. Accessed: 2023. URL: http://javacc.java.net/.

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

[106] Y. Jiang, D. Wu, P. Liu. “Jred: Program customization and bloatware mitigation based on static analysis”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. **2016**: pp. 12-21.

[107] H. Sharif, M. Abubakar, A. Gehani, et al. “TRIMMER: application specialization for code debloating”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. **2018**: pp. 329-339.

[108] K. Ferles, V. Wüstholz, M. Christakis, et al. “Failure-directed program trimming”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. **2017**: pp. 174-185.

[109] S. Herfert, J. Patra, M. Pradel. “Automatically reducing tree-structured test inputs”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. **2017**: pp. 861-871.

[110] G. Gharachorlu, N. Sumner. “Type Batched Program Reduction”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. **2023**: pp. 398-410.

[111] Q. Xin, M. Kim, Q. Zhang, et al. “Program debloating via stochastic optimization”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. **2020**: pp. 65-68.

[112] G. Gharachorlu, N. Sumner. “PARDIS: Priority Aware Test Case Reduction”. In: *International Conference on Fundamental Approaches to Software Engineering*. **2019**: pp. 409-426.

[113] G. Gharachorlu, N. Sumner. “Leveraging Models to Reduce Test Cases in Software Repositories”. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. **2021**: pp. 230-241.

[114] *Tensorflow tutorials*. Accessed: 2021. URL: https://www.tensorflow.org/guide.

[115] M. Pelikan, D. E. Goldberg, E. Cantú-Paz, et al. “BOA: The Bayesian optimization algorithm”. In: *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*. Vol. 1. **1999**: pp. 525-532. URL: https://dl.acm.org/doi/10.5555/2933923.2933973.

[116] P. Congdon. *Bayesian statistical modelling*. John Wiley & Sons, **2007**.

[117] B. Shahriari, K. Swersky, Z. Wang, et al. “Taking the human out of the loop: A review of Bayesian optimization”. In: *Proceedings of the IEEE* 104.1 (**2015**): pp. 148-175.

[118] P. I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. arXiv: 1807.02811 [stat.ML]. URL: https://arxiv.org/abs/1807.02811.

[119] Y. Zhang, Z. Dai, B. K. H. Low. "Bayesian optimization with binary auxiliary information". In: *Uncertainty in Artificial Intelligence*. **2020**: pp. 1222-1232. arXiv: 1807.02811 [stat.ML]. URL: https://arxiv.org/abs/1906.07277.

[120] K. Swersky, Y. Rubanova, D. Dohan, et al. "Amortized bayesian optimization over discrete spaces". In: *Conference on Uncertainty in Artificial Intelligence*. **2020**: pp. 769-778. URL: http://proceedings.mlr.press/v124/swersky20a.html.

[121] S. Lee, D. Binkley, R. Feldt, et al. "MOAD: Modeling Observation-based Approximate Dependency". In: *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. **2019**: pp. 12-22.

[122] R. Abreu, A. Gonzalez-Sanchez, A. J. van Gemund. "Exploiting count spectra for bayesian fault localization". In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. **2010**: pp. 1-10.

[123] M. Srinivas, L. M. Patnaik. "Genetic algorithms: A survey". In: *computer* 27.6 (**1994**): pp. 17-26. URL: https://dl.acm.org/doi/10.1109/2.294849.

[124] *CHATGPT*. Accessed: 2023. URL: https://chat.openai.com.

[125] S. Chakraborty, D. Fried, K. S. Meel, et al. "From Weighted to Unweighted Model Counting". In: *IJCAI*. **2015**: pp. 689-695. URL: https://dl.acm.org/doi/10.5555/2832249.2832345.

[126] P. Liang, O. Tripp, M. Naik. "Learning minimal abstractions". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. **2011**: pp. 31-42.

[127] *xmllint*. Accessed: 2021. URL: http://xmlsoft.org/xmllint.html.

[128] *The implementation of modernized HDD*. Accessed: 2021. URL: https://github.com/renatahodovan/picireny.

[129] Z. Xu, Y. Tian, M. Zhang, et al. "Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (**2023**): pp. 636-664.

[130] *Rust*. Accessed: 2023. URL: https://github.com/rust-lang/rust/issues.

[131] H. Zhang, M. Ren, Y. Lei, et al. "One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. **2022**: pp. 255-270.

[132] R. Williams, T. Ren, L. De Carli, et al. "Guided feature identification and removal for resource-constrained firmware". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (**2021**): pp. 1-25.

[133] C. G. Kalhauge, J. Palsberg. "Logical bytecode reduction". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. **2021**: pp. 1003-1016.

[134] S. Dutta, W. Zhang, Z. Huang, et al. “Storm: program reduction for testing and debugging probabilistic programming systems”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. **2019**: pp. 729-739.