中国科学: 信息科学 2020 年 第 50 卷 第 11 期: 1612–1628
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
可成长软件专题 · 论文

意图敏感的日志自动增强

贾周阳, 李姗姗*, 刘晓东, 王戟, 廖湘科

国防科技大学计算机学院, 长沙 410072
* 通信作者. E-mail: shanshanli@nudt.edu.cn

收稿日期: 2020–04–26; 修回日期: 2020–06–15; 接受日期: 2020–08–05; 网络出版日期: 2020–11–09

国家重点研发计划 (批准号: 2017YFB1001802) 和国家自然科学基金 (批准号: 61872373, 61872375) 资助项目

摘要 高质量的日志代码是软件故障诊断的重要依据. 由于缺乏统一规范、开发人员不够重视等原因, 现有软件中的日志质量参差不齐. 目前, 已有工作致力于日志的自动化增强, 主要分为基于易错模式的方法和基于代码特征的方法, 这些方法通过总结软件易错代码模式或学习已有日志代码的代码特征, 进而在相应的代码段中自动添加日志语句. 但开发人员添加日志代码的意图往往难以用固定的模式或特征来描述, 导致日志增强的准确性不高. 因此, 本文探索了意图敏感的日志增强方法, 提出了一种日志意图描述模型 (log-intention description model, LIDM), 在此基础上设计和实现了自动化日志增强工具 SmartLog. SmartLog 利用 LIDM 提取日志代码意图, 挖掘日志增强规则, 进而实现意图敏感的日志自动增强. 本文在 6 款成熟且被广泛使用的开源软件上对 SmartLog 的有效性展开了评估. 评估结果显示, SmartLog 的准确性相比两个已有最好的日志增强工具分别提升 43% 和 16%. 此外, 本文收集了软件演化过程中 86 个开发人员增加日志的实例, 并使用 SmartLog 和两个已有工具分析每次日志演化的旧软件版本, 发现 3 个工具可自动在新软件版本添加的日志分别是 49, 10, 22 个, 软件演化效率相比已有工作显著增强.

关键词 日志增强, 故障诊断, 日志演化, 软件意图, 日志自动化

1 引言

软件系统在运行过程中经常会遇到故障. 当故障发生时, 开发人员必须在第一时间诊断故障并恢复系统. 目前, 大规模软件的故障诊断主要依靠人力排查, 定位问题不及时且效率低. 适当的日志信息可以极大帮助开发人员进行故障诊断并减少系统恢复时间 [1]. 但由于缺乏统一的被普遍接受的日志开发标准, 编程人员在日志开发过程中多数是依赖经验和个人习惯, 导致现有软件中的日志质量并不乐观, 尤其在多人开发的大规模软件中, 日志质量参差不齐. 此外, 由于日志代码不影响软件的功能性

引用格式: 贾周阳, 李姗姗, 刘晓东, 等. 意图敏感的日志自动增强. 中国科学: 信息科学, 2020, 50: 1612–1628, doi: 10.1360/SSI-2020-0111
Jia Z Y, Li S S, Liu X D, et al. Intention-aware log automation (in Chinese). Sci Sin Inform, 2020, 50: 1612–1628, doi: 10.1360/SSI-2020-0111

需求, 所以经常不被开发人员重视, 导致一些易错代码段缺少日志, 或者在软件演化过程中, 日志代码未能随着功能代码同步更新.

已有很多研究工作关注软件日志, 这些工作大致可分为日志自动增强 [1~10]、日志特性调研 [11~20], 以及日志信息分析 [21~28]. 日志自动增强辅助开发人员提升日志质量, 日志特性调研帮助理解软件日志特征, 日志信息分析研究如何使用日志进行故障诊断. 另一方面, 软件日志可被用于输出、调试、审计、故障诊断等不同用途. 其中, "输出" 指程序运行时输出目标变量的取值, "调试" 指开发、调试过程中帮助开发人员理解程序行为, "审计" 指帮助监控程序运行状态, 如运行路径、资源使用等. 上述类型的日志在程序正确运行时输出相应信息, "故障诊断" 指在程序运行发生错误时打印出错信息, 帮助用户和开发人员定位故障根源. 本文关注面向故障诊断的日志语句自动增强, 例如申请内存失败时, 添加错误日志语句 (error log statement, ELS) printf("out of memory").

已有面向故障诊断的日志自动增强技术可分为基于易错模式的日志增强技术和基于代码特征的日志增强技术. 基于易错模式的技术预定义代码中常见出错的模式, 自动识别软件中符合易错模式的代码段, 并在缺少日志的代码段中插入 ELS, 如 Errlog [1]. 基于代码特征的技术提取软件日志代码相关的特征, 并将此特征应用于机器学习算法中, 进而决策是否添加 ELS, 如 LogAdvisor [2]. 在实际软件代码中, ELS 通常位于分支语句之后, 例如 `if` 或 `switch`. 文本将上述分支语句中的判断条件以及相关的变量和函数调用统称为日志上下文 (log contexts), 并将日志上下文的语义称为日志意图 (log intentions). 本文避免使用日志语义 (log semantics) 这一术语, 因为日志语义可以指代日志语句本身的语义, 导致产生歧义. 合理的日志增强技术应当可以学习软件中的日志意图.

为深入理解日志自动增强技术的现状与局限, 本文调研了已有技术在 6 款开源软件中增强日志的效果. 以 Errlog 为代表的基于易错模式的技术可以达到 72% 的准确率和 16% 的查全率, 即 Errlog 添加的 72% 的日志是正确的 ELS, 但 84% 需要添加 ELS 的代码段未能成功添加. 另一方面, 以 LogAdvisor 为代表的基于代码特征的技术可以达到 62% 的准确率和 32% 的查全率. 但开发人员添加日志代码的意图往往难以用固定的模式或特征来描述, 导致已有日志增强技术的准确性不高. 调研显示, 当考虑日志意图时, 理论上添加 ELS 的准确率和查全率可以分别达到 84% 和 58%. 本文针对上述准确率和查全率的理论与实际差距, 设计并实现了可以理解日志意图的自动化日志增强工具 SmartLog. 相关调研细节详见第 2 节.

SmartLog 面临两个主要挑战. 一方面, 软件代码并非简单文本, 而是包含了复杂的逻辑结构, 如循环、分支、函数调用等. 上述逻辑结构反应了软件的意图. 自动化工具在深入理解软件意图前, 很难决定是否需要添加 ELS. 另一方面, 缺乏公认的标准指定在何种软件意图下添加日志. 即使给定软件意图, 自动化工具仍然难以决策是否加日志. 针对上述挑战, SmartLog 设计了以下主要技术. 首先, SmartLog 在传统基于关键字识别软件已有日志这一方法的基础上, 设计识别算法, 准确识别 ELS. 其次, SmartLog 设计了一种意图描述模型 (log-intention description model, LIDM), 在程序依赖分析技术的基础上, 从源码中提取日志意图. 最后, SmartLog 基于频繁项挖掘技术从软件已有的日志意图中挖掘日志添加规则, 进而添加 ELS. 本文的贡献包括以下 3 点:

• 本文研究了日志自动增强工具的现状与局限, 并且深入分析了导致上述局限的根本原因, 即已有工具没有考虑软件开发人员添加日志的意图, 导致影响自动日志增强的准确性.

• 本文设计并实现了 SmartLog, 并设计了针对日志代码的意图描述模型 LIDM, 意图描述模型可以普适地描述软件不同日志上下文之间语义的等价性.

• 本文系统评估了 SmartLog, 发现 SmartLog 的准确性相比两个已有最好的日志增强技术分别提升 43% 和 16%, 并且以不超过 1% 的性能损失覆盖 86 个日志演化历史中的 49 个.

表 1 不同技术的准确率和查全率

Table 1 Precisions and recalls of different techniques

Project	Recalls of different techniques			Precisions of different techniques		
	Errlog	LogAdvisor	Design target	Errlog	LogAdvisor	Design target
Httpd	5.3/50	19.5/50	30.7/50	7.2/8.2	14.9/28.0	13.7/18.1
Subversion	4.2/50	23.1/50	36.4/50	7.5/9.0	16.9/30.2	13.8/16.8
MySQL	10.5/50	6.0/50	24.1/50	2.4/6.3	7.4/24.0	6.7/8.8
PostgreSQL	9.6/50	14.3/50	33.3/50	9.4/10.7	23.1/25.3	21.9/23.0
GIMP	8.0/50	20.2/50	26.7/50	1.5/1.9	18.1/25.5	17.5/19.8
Wireshark	10.4/50	13.7/50	23.0/50	4.5/9.0	19.1/26.5	15.0/18.5
Total	48.0/300 (16%)	96.8/300 (32%)	174.2/300 (58%)	32.5/45.1 (72%)	99.5/159.5 (62%)	88.6/105 (84%)

本文是基于会议论文 [29] 的扩展版. 扩展内容包括但不限于以下 3 个主要方面: (1) 增加经验性调研, 研究已有工具不足的根源; (2) 增加设计, 帮助 SmartLog 确定新增日志的上下文和内容; (3) 增加实验, 评估 SmartLog 各模块的准确性.

2 现状与局限

本节基于 6 款成熟且被广泛使用的开源软件对已有日志增强工具的准确率和查全率进行了调研, 对调研结果进行了分析, 并且揭示了导致上述结果的根源. 6 款软件包括 Httpd, Subversion, MySQL, PostgreSQL, GIMP 和 Wireshark. 本文选用 LogAdvisor 和 Errlog 作为已有工作的代表. LogAdvisor 是基于代码特征的日志增强工具, 通过提取软件日志代码相关的特征, 并将此特征应用于机器学习算法中, 进而决策是否添加 ELS. 常见的特征包括: 代码的结构特征、文本特征、语法特征等. Errlog 是基于易错模式的日志增强工具, 通过预定义代码中常见出错的模式, 自动识别软件中符合易错模式的代码段, 并在缺少日志的代码段中插入 ELS. 常见的模式包括: 特定库函数返回错误、异常信号处理、`switch` 语句默认分支等. 给定一个日志增强工具, 其准确率为所有新增日志语句中正确的 ELS 的比例, 其查全率为所有需要增加 ELS 的代码段中有新增 ELS 的比例.

2.1 已有技术查全率的局限性

为调研已有技术的查全率, 本小节从 6 款开源软件中各随机选取了 100 处面向故障诊断的 ELS (共 600 个 ELS) 以及相应的上下文, 再随机选取一半数据作为训练集另一半数据作为测试集. 由于不同的技术使用相同的数据集公平对比, 因此 600 个 ELS 足够评估出不同技术的准确性差距. 此外, Errlog 无需训练, 因此 Errlog 直接被应用于测试集. 上述过程独立重复 10 次的平均查全率如表 1 的左半部分所示. 例如 LogAdvisor 在 Httpd 中平均可以添加 19.5 个 ELS, 综合查全率为 32%, 同时 Errlog 的查全率为 16%.

LogAdvisor 的查全率为 32% 的原因在于学习的特征限于语法层次. 例如图 1 展示了关于函数 `proxy_ftp_command` 的日志行为的两个示例代码段, `proxy_ftp_command` 需要在不同的返回值下添加 ELS. 在第 1 个代码段中, 函数返回 −1, 421 或 550 时会触发 ELS. 在第 2 个代码段中, 同样当 `rc` 等于 −1, 421 或 550 时触发 ELS. 两个代码段中的日志上下文的语义是等价的, 因此出现了同样的日志行为, 但两个代码段在语法上除了函数名之外都是不同的. 因此基于语法特征的方法难以通过学习第 1 个代码段中的日志上下文, 进而在第 2 个代码段缺失日志时添加日志. 另一方面, Errlog 的查全率为

```
/*Case 1:httpd/…/mod_proxy_ftp.c*/
switch ( proxy_ftp_command (…)) {
  case −1:
  case 421:
  case 550:
    ap_proxyerror (…"Failed to …");
    break;
}
```

```
/*Case 2: httpd/…/mod_proxy_ftp.c*/
rc = proxy_ftp_command (…);
if (rc ==−1 || rc == 421) {
  return ftp_proxyerror ("Error …");
}
if (rc == 550) {
  ap_log_rerror (…,"RETR failed…");
}
```

图 1 语义等价代码示例, 两个代码段均为函数返回 −1, 421 或 550 时触发日志

Figure 1 Examples of semantic equivalent code snippets w.r.t. log placement. In Example 1, a log will be triggered when the return value equals to −1, 421 or 550. In Example 2, when `rc` equals to −1, 421 or 550, there is a log too

16%, 由于仅考虑特定的易错模式, 显然会造成漏报.

日志上下文语义等价性的影响. 语法等价的日志上下文必定语义等价, 相反, 语义等价的日志上下文可以有多种语法实现方法. 例如, 给定日志上下文的语义: "当函数 `foo` 返回 0 时应该加日志", 会有以下 4 种不同的实现方式:

$C1$: `rv = foo(); if(!rv) log();`　　$C3$: `if(rv = foo()) ... else log();`

$C2$: `rv = foo(); if(rv == 0) log();`　　$C4$: `if(tmp = foo()) ... else log();`

两个日志上下文的等价关系可以分为以下 3 类: (1) 语法等价. 两个日志上下文除空行、注释之外完全相同. (2) 语法不等价但语义等价. 两个日志上下文语法不等价, 但在相同的输入下会产生相同的日志行为. 例如两个日志上下文包括不同的变量名 ($C3$ and $C4$), 不同的判断条件 ($C1$ and $C2$), 或不同的分支 ($C2$ and $C3$). (3) 语义不等价. 例如针对函数 `atoi` 的日志: `if(atoi()==1) log();` 和 `if(atoi()==2) log();` 有不同语义的日志上下文. 基于语法规则的方法难以判断第 2 类日志上下文的等价性, 导致无法学到相应的日志规则, 最终产生漏报.

为定量研究上述缺陷, 本文分析了 6 个软件中的 202 个函数, 其中每个函数均被调用多次, 并且每次都伴有相应的 ELS (202 为目前找到的所有满足条件的函数数量). 通过人工分析每个函数在不同调用后的 ELS 的上下文的语义, 可按上述 3 种等价关系将所有函数分类. 分类结构如图 2 所示, 例如在 Wireshark 的 20 个函数中, 仅有 2 个函数有语法等价的日志上下文, 5 个函数有语义不等价的上下文, 其余 13 个均为语法不等价但语义等价, 包括 3 个有不同的变量名、4 个有不同的判断条件、6 个有不同的分支. 在所有 202 个函数的日志上下文中, 20.29% (41/202) 语法等价, 12.37% (25/202) 语义不等价, 其余 67.32% (136 /202) 为语法不等价但语义等价. 因此, 可以检测日志上下文语义等价性的日志增强工具可以极大提升查全率. 如表 1 所示, 人工分析发现考虑语义等价性时查全率可以提升至 58%.

2.2 已有技术准确率的局限性

本小节对 LogAdvisor 和 Errlog 的准确率进行调研. 软件日志可被用于不同用途, 除面向故障诊断的错误日志语句外, 还包括输出结果、调试、审计等非错误日志. 本小节利用 LogAdvisor 使用的关键字技术识别错误日志语句, 通过判定函数调用的函数名是否包含诸如 `log`, `trace`, `write` 等关键字, 来确定一个函数调用是否为错误日志语句. 通过在 6 个开源软件中各随机抽取 100 个函数调用代码段, 每个代码段包括一个错误日志语句或者非错误日志语句, 可得到 600 个代码段. 将上述代码段分为两个集合, 分别作为训练集和测试集. 实验移除测试集中的所有错误日志语句, 并测试 LogAdvisor

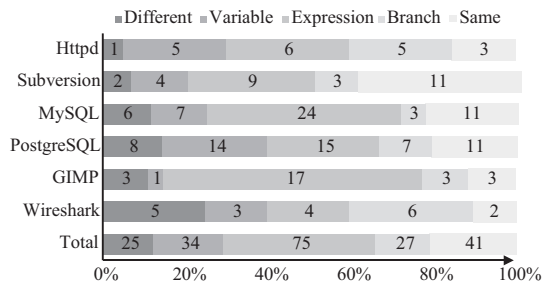


图 2　日志上下文等价性分布, 包括语义不等价 (“Different”)、语义等价但变量名不同 (“Variable”)、判断条件不同 (“Expression”)、分支不同 (“Branch”), 和语法等价 (“Same”)

Figure 2　Equivalences of log contexts, including semantic non-equivalent (“Different”), semantic equivalent but have different variables (“Variable”), check conditions (“Expression”) or branches (“Branch”), and syntax equivalent (“Same”)

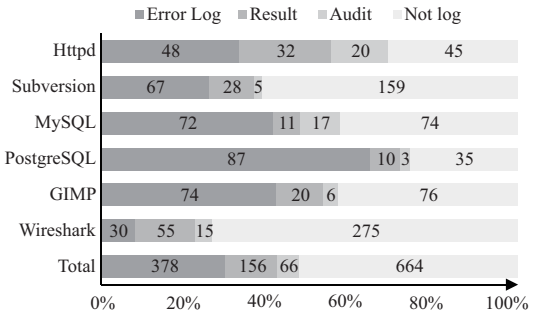


图 3　不同日志用途分布, 包括错误日志语句 (“Error Log”)、用于展示结果的日志 (“Result”)、用于程序审计的日志 (“Audit”), 以及非日志 (“Not log”)

Figure 3　Different usages of log statements, including error log statements (“Error Log”), showing results (“Result”), auditing (“Audit”), and non-log statements (“Not log”)

和 Errlog 能否正确添加被移除的错误日志语句. 最后将正确添加的错误日志语句除以添加的总日志数, 得到日志增强的准确率.

上述实验独立重复进行 10 次可计算得到平均准确率, 结果如表 1 的右半部分所示. Errlog 可以得到较高的准确率, 例如在 Httpd 中, Errlog 平均添加了 8.2 个日志语句, 其中 7.2 个错误日志语句, 因此准确率为 88%. 高准确率的原因在于 Errlog 有效定义了软件中的易错模式, 在易错模式中添加日志大多数为错误日志语句. 另一方面, LogAdvisor 在 Httpd 平均插入了 28.0 个日志语句, 其中 14.9 个错误日志语句. LogAdvisor 在 6 款软件的综合准确率为 62%. 这一准确率产生的原因在于使用关键字方法识别错误日志语句. LogAdvisor 使用机器学习技术, 因此准确率高度依赖于训练集中数据标记的准确性. 当使用关键字方法时, 训练集中的标记会出现错误, 因此影响了后续的学习过程.

识别 ELS 的影响. 在实际软件中, 准确识别 ELS 并非易事. 大规模软件中常有数百个日志函数, 人工输入所有日志函数并不可行. 而且一个日志函数可以被用于错误日志语句和非错误日志语句. 例如日志函数 `printf` 可以有调用 `printf("Memory page fault")` 和 `printf("Hello world")`, 前者为错误日志语句后者为非错误日志语句. 因此, 面向函数名的关键字方法容易导致误报.

为定量研究上述缺陷, 本小节首先提取 6 款软件中的函数调用语句, 再使用关键字方法从上述语句中各选出 100 个日志语句, 并人工分类每个日志语句的用途, 共计 600 个日志语句. 在筛选的过程中, 关键字方法得到 600 个日志语句的同时也得到了 664 非日志语句, 例如 `sprintf` 包含了关键字 `print`, 但并非日志语句. 分类结果如图 3 所示, 例如在 Httpd 中, 100 个日志语句包括 48 个错误日志语句和 52 个非错误日志语句. 非错误日志语句包括 32 个用于展示结果的日志, 20 个用于程序审计的日志. 此外, 还有 45 个非日志样本满足关键字要求. 由于软件发布时一般不包括调试日志 (如关闭宏开关), 且本实验分析软件源码时使用默认编译选项, 因此上述结果不包含调试日志. 在本实验中, 关键字方法共选出 1264 个日志语句, 其中 30% (378/1264) 为错误日志语句, 17.5% (222/1264) 为非错误日志语句, 其他 52.5% (664/202) 为非日志. 因此, 准确识别错误日志语句可以极大提升日志增强工具的准确率. 如表 1 所示, 人工分析发现考虑错误日志语句识别时准确率可以提升至 84%.

3 日志意图描述模型

本节将介绍日志意图描述模型 (LIDM), 用来帮助 SmartLog 提取代码中的日志意图. 开发人员添加错误日志语句的一般意图是 “在易错代码段发生错误时打印出错信息”, 本文将上述易错代码段称为日志上下文. 在软件开发中, 不同日志上下文可能蕴含相同的语义, 例如图 1 中两个不同的日志上下文有相同的语义. 因此, 本文定义日志意图如下:

定义1 (日志意图) 日志意图为日志语句所在的上下文的归一化形式, 语义相同的日志上下文有相同的归一化形式.

为准确描述日志意图, 本节首先定义日志上下文描述模型 (log-context description model, LCDM), LCDM 将进一步被归一化为日志意图描述模型 (LIDM), 用来帮助 SmartLog 评估日志上下文之间的等价性.

SmartLog 用于分析 C/C++ 程序, 其中主要错误处理机制为 “函数 – 返回值 – 检测”. 因此日志上下文主要涉及以下 4 个因素: 首先, ELS 所处理的错误经常是由一个函数所触发, 如 `fopen()`, 本文称之为易错函数. 其次, ELS 经常处于与易错函数相关的判断条件下, 如 `if(!fopen()) log();` 中 `if` 语句的判断条件. 第三, 判断条件可以发生于易错函数调用前的参数检测, 如 `if(!str) log(); fopen(str,...);`, 也可以发生于易错函数调用后的返回值检测, 如 `fp=fopen(); if(!fp) log();`, 因此判断条件位置是第 3 个因素. 最后, 在上述判断条件下是否应该加 ELS 的决策是第 4 个因素. 综上考虑, 本文定义日志上下文描述模型 (LCDM) 如下:

定义2 (日志上下文描述模型) 日志上下文描述模型是一个四元组: $\mathscr{M} = \langle \mathcal{F}, \mathcal{C}, \mathcal{P}, \mathcal{D} \rangle$, 其中 $\mathcal{F}$ 为日志上下文中的易错函数调用名; $\mathcal{C}$ 为与易错函数调用相关的判断条件; $\mathcal{P}$ 为判断条件的位置, $\mathcal{P}$ 的取值为 `B` (判断条件在函数调用之前) 或 `A` (判断条件在函数调用之后); $\mathcal{D}$ 为当前判断条件下是否需要加日志的决策, $\mathcal{D}$ 的取值为 `Y` (加日志) 或 `N` (不加日志).

其中判断条件 $\mathcal{C}$ 为分支语句 `if` 和 `switch` 中的判断条件, 其他不常用于出错处理的分支语句将被忽略, 如 `for` 和 `while`. 第 4.2 小节将介绍如何从代码中提取 LCDM 实例.

LCDM 可以用于描述日志上下文, 不同上下文的 $\mathcal{F}$, $\mathcal{P}$, $\mathcal{D}$ 容易判断等价性. 但其中的判断条件 $\mathcal{C}$ 可能较为复杂, 不同的判断条件可能表达相同的语义, 导致难以评估日志上下文之间的等价性. 因此, 本文定义日志意图描述模型 (LIDM) 如下:

定义3 (日志意图描述模型) 日志意图描述模型是一个四元组: $\hat{\mathscr{M}} = \langle \mathcal{F}, \hat{\mathcal{C}}, \mathcal{P}, \mathcal{D} \rangle$, 其中 $\mathcal{F}$ 为日志上下文中的易错函数调用名, $\mathcal{P}$ 为判断条件的位置, $\mathcal{D}$ 为当前判断条件下是否需要加日志的决策. $\hat{\mathcal{C}}$ 为布尔表达式且仅包含以下两个逻辑运算:

- $\neg$: 表达式与相应的取非形式, 如 a 和 $\neg a$;
- $\wedge$: 多个表达式 (或取非形式的表达式) 的交集, 如 $a \wedge \neg b \wedge d$.

LCDM 和 LIDM 的主要区别在于判断条件. 在 LCDM 中, 判断条件 $\mathcal{C}$ 可以为任何形式; 在 LIDM 中, 判断条件 $\hat{\mathcal{C}}$ 的形式则更加严格. 因此 LIDM 的形式更少, 相同的 LIDM 实例更有可能语义等价. 例如在第 2.1 小节中, 日志上下文语义 “当函数 `foo` 返回 0 时应该加日志” 有 4 个不同的 LCDM 实例, 可以被统一地描述为同一 LIDM 实例: $\hat{m}$ = ⟨`foo, ¬foo_0, A, Y`⟩. 第 4.3 小节将介绍如何从 LCDM 实例生成 LIDM 实例.

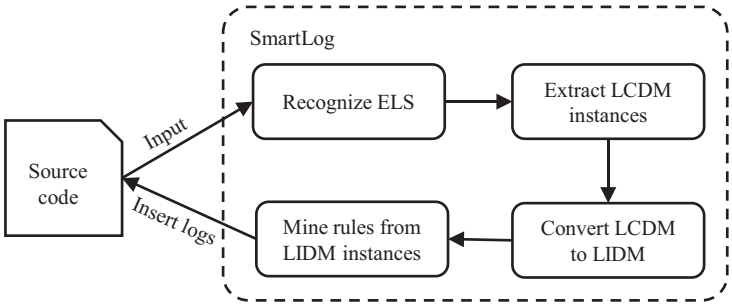


图 4 SmartLog 的工作流程, 首先从软件源码中识别并标记 ELS, 再从源码中提取 LCDM 实例, 并将其转化为 LIDM 实例, 之后基于 LIDM 实例的统计数据挖掘 ELS 添加规则, 最后在缺失错误日志语句的代码段中插入日志

Figure 4 The architecture of SmartLog. SmartLog recognizes ELS, extracts LCDM instances from code, and converts them to LIDM instances. We collect the statistics of LIDM instances to mine ELS rules, and inserts ELS for violations of the rules

4 SmartLog 设计与实现

本节将介绍 SmartLog 的设计与实现, 其工作流程如图 4 所示. 首先, SmartLog 从软件源码中识别并标记 ELS. 其次, 从源码中提取 LCDM 实例, 并将其转化为 LIDM 实例. 之后, 收集 LIDM 实例的统计数据, 并应用频繁项挖掘技术获取 ELS 添加规则. 最后, SmartLog 确定日志上下文和日志内容, 并在日志缺失的代码段中插入日志.

4.1 错误日志语句识别

SmartLog 学习开源软件中的错误日志语句实践, 进而为缺失日志的代码段添加日志. SmartLog 首先需要识别软件中已有的 ELS, 每个 ELS 包括两个部分: 日志函数名和日志函数参数. SmartLog 提出一种两步识别算法: 首先使用关键字方法找出所有潜在日志函数, 再进一步识别用于错误日志语句的函数调用.

算法 1 描述了 SmartLog 识别错误日志语句的过程. 用于打印错误信息的日志函数通常包含 `error`, `exit` 或 `output` 的语义信息, SmartLog 首先基于 WordNet 工具建立包含上述语义的关键字列表 Ψ, 包含 `error`, `exit` 和 `output` 的同义词 (line 1). 之后, 针对软件中的每一个函数 (line 2), SmartLog 利用分词技术 (one-gram word segmentation) 分割函数名 (line 3). 当任一分割后的子函数名包含 Ψ 中的任一关键字时, SmartLog 将此函数认为潜在日志函数 (line 4). SmartLog 得到软件中所有满足上述条件的潜在日志函数集合 F_p (line 5).

一个潜在日志函数可能被用于错误日志语句, 如 `printf("Memory page fault")`, 或非错误日志语句, 如 `printf("Hello world")`. 为识别用于错误日志语句的函数调用, 本文发现此类函数调用的参数通常包含 `error`, `exit`, `limitation` 或 `negation` 语义. 因此 SmartLog 建立另一关键字列表 $\hat{\Psi}$, 包含上述关键字以及相应的同义词 (line 6). 之后, 对 F_p 集合中的每一个函数 f (line 7), 分析 f 的每一个调用 c (line 8). 当一个潜在日志函数的函数调用参数包含 $\hat{\Psi}$ 中的任一关键字时, 此调用被标记为错误日志语句 (line 9). 此外, SmartLog 考虑数据流依赖对错误日志语句识别的影响 (line 10), 如在代码段 `msg = "fatal:OOM"; printf("%s", msg);` 中, `printf` 的调用参数不包含 $\hat{\Psi}$ 中的任何关键字, 回溯数据流发现其参数 `msg` 的定义包括 $\hat{\Psi}$ 中的关键字 `fatal`. 因此上述 `printf` 同样被标记为错误日志语句.

算法 1 Error log statements recognition algorithm

Input: Software source code.

Build synonymous keyword list Ψ for {`error`, `exit`, `output`};
for each function f **do**
  Conduct word segmentation for f's name;
  Let $F_p = \{f \mid \exists g \in \text{one-gram}(f)$ for which $g \in \Psi\}$;
end for
Build synonymous keyword list $\hat{\Psi}$ for {`error`, `exit`, `limitation`, `negation`};
for $\forall f \in F_p$ **do**
  for $\forall c$ of f **do**
    Label c if its argument contains keywords in $\hat{\Psi}$;
    Label c if it has data dependence with another statement containing keywords in $\hat{\Psi}$;
  end for
end for

Output: Error log label.

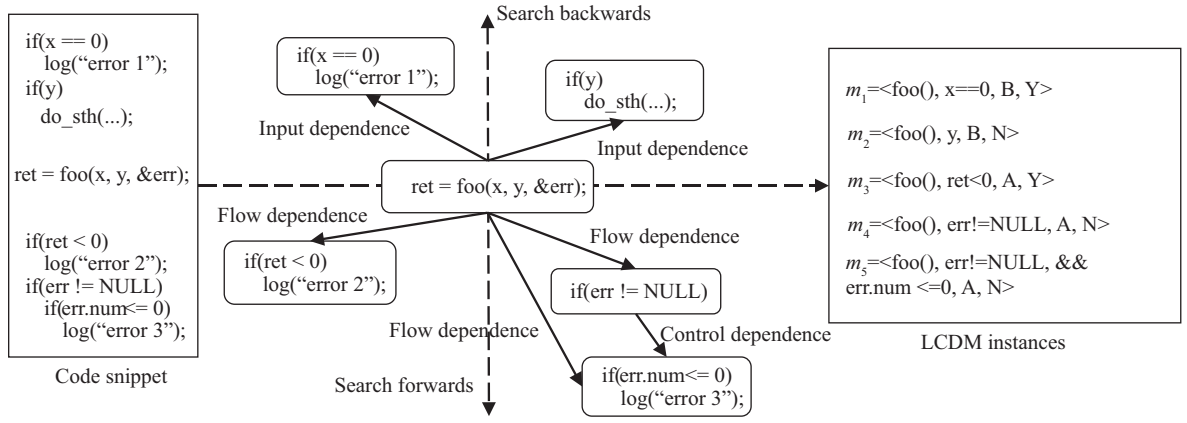


图 5 基于程序依赖图将示例代码段转化为 LCDM 实例

Figure 5 Extract LCDM instances form a code snippet by using program dependence graph

上述识别过程中, 关键字列表 Ψ 用于选择潜在日志函数, 关键字列表 $\hat{\Psi}$ 用于选择错误日志语句调用. 例如, 函数 `printf` 是一个潜在日志函数 (函数名包含 Ψ 中 `output` 的同义词 `print`), 函数调用 `printf("Memory page fault")` 是一个错误日志语句 (调用参数包含 $\hat{\Psi}$ 中 `error` 的同义词 `fault`).

4.2 LCDM 实例提取

SmartLog 通过从软件源码中提取 LCDM 实例, 可将日志意图从源码的形式转化为结构化数据. 如图 5 所示, SmartLog 将图中左侧代码段转换为右侧 LCDM 实例. 在图 5 的代码中, 首先对变量 x 和 y 判断, x 为 0 时执行错误日志语句. 之后调用函数 `foo`, `foo` 使用 x 和 y 作为参数, 返回 `ret` 并将错误信息写入参数 `err`. 函数调用后, 依次对返回值 `ret` 和参数 `err` 进行判断, 并在满足判断条件时执行错误日志语句.

SmartLog 首先提取软件源码中的函数调用, 例如 `foo`. SmartLog 一方面从 `foo` 进行后向搜索 (search backwards), 搜索输入依赖 (input dependent) 于 `foo` 的所有分支语句. 输入依赖表示分支语句和 `foo` 读取了同一变量 (或同一变量的别名). 如图 5 中分支语句 `if(x==0)` 和函数调用 `foo` 均读取了

变量 x, 因此 `if(x==0)` 输入依赖于 `foo`. SmartLog 另一方面从 `foo` 进行前向搜索 (search forwards), 搜索流依赖 (flow dependent) 于 `foo` 的所有分支语句. 流依赖表示分支语句读取了 `foo` 写入的变量. 如图 5 中分支语句 `if(ret<0)` 读取了函数 `foo` 写入的变量 `ret`, 因此 `if(ret<0)` 流依赖于 `foo`.

对于任意满足上述条件的分支语句 b, SmartLog 获取 b 中的判断条件 c, 并检测 b 是否包含 ELS. 如果包含, 得到 LCDM 实例 m = ⟨`foo`, c, `B/A`, `Y`⟩, 否则得到 m = ⟨`foo`, c, `B/A`, `N`⟩, 其中 `B/A` 取决于当前分支语句位于 `foo` 之前 (`B`) 或之后 (`A`). 例如, 在图 5 中, 从 `foo` 向后搜索得到满足条件的分支语句 `if(x==0)` 和 `if(y)`, 其中 `if(x==0)` 包含 ELS, `if(y)` 未包含 ELS. 因此, SmartLog 得到两个 LCDM 实例 m_1 和 m_2. 从 `foo` 向前搜索得到满足条件的分支语句 `if(ret<0)`, `if(err!=NULL)` 和 `if(err.num<=0)`, 其中 `if(ret<0)` 包含 ELS, `if(err!=NULL)` 未包含 ELS. 因此, SmartLog 得到两个 LCDM 实例 m_3 和 m_4. 当两个分支语句相互控制依赖时, SmartLog 合并两个判断条件. 如 `if(err.num<=0)` 控制依赖于 `if(err!=NULL)`, 因此 SmartLog 得到 LCDM 实例 m_5, 其中判断条件 $\mathcal{C}$ 为 `err!=NULL && err.num<=0`.

4.3 LCDM 归一化为 LIDM

LCDM 可以用于描述日志意图, 但其中的判断条件 $\mathcal{C}$ 可能较为复杂, 导致难以评估日志上下文之间的等价性. 因此本文设计日志意图模型 (LIDM), 并将 LCDM 实例的判断条件 $\mathcal{C}$ 归一化为 LIDM 实例的判断条件 $\hat{\mathcal{C}}$.

变量转换. (1) 如果变量是一个函数的返回值, 则将变量重命名为函数名加后缀 "0". 如 m_3 中的返回值 `ret` 被重命名为 `foo_0`. (2) 如果变量是一个函数的参数, 则将变量重命名为函数名加数字后缀, 数字含义为参数位置. 如 m_1 中变量 x 是函数 `foo` 的第 1 个参数, 因此被重命名为 `foo_1`. (3) 和易错函数不相关的变量将保持不变. (4) 常量表达式直接计算得到最终结果.

表达式转换. (1) 运算符 $>$ 和 $\geqslant$ 通过改变操作数的顺序分别转换为 $<$ 和 $\leqslant$ 的形式. (2) 三元条件运算符 $a?b:c$ 改变为等价的 $(a \wedge b) \vee (\neg a \wedge c)$ 形式. (3) 当运算符 $==$ 或 $!=$ 的一个操作数为 0 或 `NULL` 时, 转换为忽略此操作数的等价形式. 如 m_1 和 m_4 中的 `x==0` 和 `err!=NULL` 将变为 `!x` 和 `err`. (4) 赋值表达式直接用左值替代. (5) 二元运算符的两个操作数的顺序可变时按字典序排列, 如运算符 $+$, $==$, $||$ 等.

语句转换. (1) 统一分支语句, `switch` 语句将被转化为等价的 `if` 语句形式. 如 `switch(flag) case 1: log("error");` 转换为 `if(flag==1) log("error")`. (2) 当 ELS 位于 `if` 语句的 `else` 分支时, 将判断条件取反. 如 `if(foo())...else log();` 转化为 `if(!foo()) log();`.

一对多转换. 判断条件可能由复杂的逻辑连接词 (如 $\wedge$ 或 $\vee$) 连接而成. 这种情况下, SmartLog 将判断条件拆分为多个子条件: (1) 将 `if(a∨b)` 拆分为 `if(a)` 和 `if(b)`; (2) 将 `if(a∧(b∨c))` 拆分为 `if(a∧b)` 和 `if(a∧c)`. 通过拆分判断条件, 一个 LCDM 实例将生成多个 LIDM 实例. 多 LIDM 实例有共同的易错函数 $\mathcal{F}$, 判断条件位置 $\mathcal{P}$, 日志决策 $\mathcal{D}$, 以及各自独立的子判断条件 $\hat{\mathcal{C}}$.

SmartLog 重复使用上述转换规则到不动点, 即无法进一步应用任何转换规则. 完成转换后, 两个 LIDM 实例中, 若判断条件 $\hat{\mathcal{C}}$ 相同即为语义等价.

4.4 日志添加规则挖掘

基于 LIDM 实例, SmartLog 使用频繁项挖掘技术挖掘日志添加规则. 本小节首先定义挖掘过程中相关的符号: $\Pr(M)$ 为事件 M 发生的概率, $\Pr(M,N)$ 为事件 M 和事件 N 同时发生的概率, $\Pr(N|M)$

为已知事件 M 发生时事件 N 发生的条件概率, $\rho(M,N)$ 为事件 M 和事件 N 的相关性:

$$\rho(M,N)=\frac{\Pr(M,N)}{\Pr(M)\times\Pr(N)},$$

$\rho(M,N)$ 大于 1 时为正相关, $\rho(M,N)$ 小于 1 时为负相关, 否则事件 M 和事件 N 相互独立.

日志添加规则为一个三元组: $\mathscr{R}=\langle\mathcal{F},\hat{\mathcal{C}},\mathcal{P}\rangle$, 意为函数调用 $\mathcal{F}$ 应该在条件 $\hat{\mathcal{C}}$ 下, 在位置 $\mathcal{P}$ 添加日志. SmartLog 从 LIDM 实例中挖掘满足以下两条属性的日志添加规则:

$$\Pr(Y\mid\mathcal{F},\hat{\mathcal{C}},\mathcal{P})\geqslant\Pr_{\text{th}},\tag{1}$$

$$\rho(\hat{\mathcal{C}},\mathcal{P},Y\mid\mathcal{F})\geqslant 1,\tag{2}$$

其中 $\Pr_{\text{th}}$ 为预定义阈值. 属性 (1) 表示当 $\mathcal{F}$, $\hat{\mathcal{C}}$, $\mathcal{P}$ 同时出现时, 日志决策为 `Y` 的概率大于等于预定义阈值 $\Pr_{\text{th}}$, 即加日志的置信度大于等于预定义阈值 $\Pr_{\text{th}}$. 属性 (2) 表示当 $\mathcal{F}$ 出现时, $\hat{\mathcal{C}}$, $\mathcal{P}$ 和日志决策为 `Y` 不能为负相关, 即 $\hat{\mathcal{C}}$ 和 $\mathcal{P}$ 的出现不能使日志决策为 `Y` 的概率降低. 属性 (1) 和 (2) 本质上分别关注特定情况下日志决策为 `Y` 的绝对数量和相对数量.

例如下面给出了 4 个 LIDM 实例, 以及相应的出现次数:

`<foo, foo_0<0, A, Y>` : 10 次	`<foo, !foo_0, A, Y>` : 10 次
`<foo, foo_0<0, A, N>` : 9 次	`<foo, !foo_0, A, N>` : 1 次

表示对于函数 `foo`, 有 19 次调用后判断返回值是否小于 0 (`foo_0<0`), 其中 10 次加了 ELS, 9 次未加 ELS; 另有 11 次调用后判断返回值是否等于 0 (`!foo_0`), 其中 10 次加了 ELS, 1 次未加 ELS. 函数 `foo` 加 ELS 的概率为 $20/30=0.67$, 返回值小于 0 时加 ELS 的概率为 $10/19=0.53$, 即返回值小于 0 降低了加 ELS 的概率, 因此 SmartLog 应排除返回值小于 0 时加 ELS 的规则.

对于属性 (1), 返回值小于 0 和返回值等于 0 时加日志的置信度分别为 `Pr(Y|foo, foo_0<0, A)` $=0.52$ 和 `Pr(Y|foo, !foo_0, A)` $=0.91$. 假设 $\Pr_{\text{th}}$ 为 0.5 时, 两个判断条件均满足条件, 即两种情况下加 ELS 的绝对次数较高. 对于属性 (2), 两个判断条件与加日志的相关性分别为 `ρ(foo_0<0, A, Y|foo)` $=0.79$ 和 `ρ(!foo_0, A, Y|foo)` $=1.36$. 在这种情况下, 仅返回值等于 0 时满足条件, 而返回值小于 0 和加 ELS 负相关. 因此 SmartLog 可以正确得到唯一的日志添加规则: `<foo, !foo_0, A>`, 即 `foo` 的返回值等于 0 时应该加日志.

4.5 新增错误日志语句

SmartLog 插入 ELS 需要考虑两个主要因素: 新增日志的上下文以及新增日志的内容.

新增日志上下文. 得到日志添加规则之后, SmartLog 再次分析代码, 并在违反日志添加规则的代码段中添加日志. 给定日志规则 `r = <foo, !foo_0, A>`, SmartLog 可能在如图 6 所示的两种情况下加日志:

(1) 易错函数调用 (即 `ret=foo(a)`) 后跟随判断条件 (即 `if(ret==0)`), 但判断条件中缺失错误日志语句. 这种情况下, SmartLog 将直接在判断条件中添加日志.

(2) 易错函数调用 (即 `foo(a)`) 后没有任何判断条件, 并缺失错误日志语句. 这种情况下, SmartLog 首先声明函数返回值 (即 `int ret = foo(a)`), 之后依据日志添加规则中的 $\hat{\mathcal{C}}$ 增加针对函数返回值的判断条件 (即 `if(!ret)`), 最后在新增的判断条件中添加日志.

新增日志内容. 新增日志内容一般包含两个部分: 静态文本以及关键参数. SmartLog 应用文献 [3] 中提出的方法确定新增日志的关键参数. 对于静态文本, SmartLog 为每个需要添加日志的代码段生成如图 7 所示的补丁.

每条日志语句包括一个唯一的 ID, 日志语句在代码中的位置, 易错函数的函数名, 以及归一化之后的判断条件. 最后, SmartLog 从已有日志中学习需要添加的静态文本. SmartLog 收集已有的具有相同日志意图的日志, 并统计每条已有日志中静态文本的词频. SmartLog 选出所有重复出现词作为静态文本. 静态文本并非语法正确的错误信息, 但仍然可以为开发者提供故障诊断线索. 如图 7 所示, 易错函数 `init_dynamic_string` 在多次调用后均有日志, 并且已有日志中重复出现关键词 “Out”, “of”, “memory”. 因此 SmartLog 将上述关键词作为静态文本.

5 实验评估

本节使用同第 2 节相同的 6 款开源软件对 SmartLog 进行评估. 首先评估 SmartLog 模型中的参数对准确率和查全率的影响, 之后从有效性和性能两个方面评估 SmartLog 的结果, 最后分析 SmartLog 各主要模块的准确性.

5.1 准确率和查全率评估

SmartLog 从源码中提取 LCDM 实例并进一步转化为 LIDM 实例, 之后应用频繁项挖掘技术获取日志添加规则. 对于每一条日志添加规则, SmartLog 要求当 $\mathcal{F}$, $\hat{\mathcal{C}}$ 和 $\mathcal{P}$ 同时出现时, 日志决策为 `Y` 的置信度高于特定阈值; 同时当 $\mathcal{F}$ 出现时, $\hat{\mathcal{C}}$, $\mathcal{P}$ 和 `Y` 不能为负相关. 本小节首先评估挖掘过程中的阈值对准确率和查全率的影响, 之后将 SmartLog 取最优阈值时的准确率和查全率与已有技术进行对比.

实验设置. 实验包括了 SmartLog 收集的所有 LIDM 实例, 每个实例都有一个标签 (`Y/N`) 表示加 ELS 或不加 ELS. 所有数据被平均分为两部分, 一部分用于训练一部分用于测试. 在每次实验中, 删掉用于测试的实例中已有的 ELS, 并评估不同技术恢复这些 ELS 的准确率和查全率. 为方便对比, 本小节进一步计算 F 值, 即准确率和查全率的调和平均数. 每次实验均将独立重复 10 次, 并且取平均的准确率、查全率, 以及 F 值.

阈值对准确率和查全率的影响. 在 SmartLog 设计中, 规则挖掘阶段有一个可调参数: 置信度阈值 (取值范围为 $[0, 1]$)). 如图 8 所示, 横轴为置信度设置为从 0 到 1 不同的阈值, 间隔为 0.05. 纵轴为设置相应阈值时的 F 值. 以 MySQL 为例, 当置信度阈值从 0 到 0.15 时, F 值持续上升直到最大值 0.52; 当置信度阈值大于 0.3 时, F 值开始下降. 所有目标软件均在置信度阈值为 0.3 到 0.35 时得到最优 F 值, 因此 SmartLog 选取 0.33 作为最优阈值.

此外, 训练集数据的占总数据集的比例也会影响评价准确率和查全率, 本小节进一步评估了训练数据占数据集 50%, 60%, 70%, 80% 和 90% 时, SmartLog 添加日志的 F 值. 实验结果如图 9 所示, 随着训练数据占比的提升 SmartLog 的 F 值保持不变或同步提升.

不同技术准确率和查全率的对比. 本小节将 SmartLog 使用最优阈值时的准确率和查全率和已有技术对比, 包括 Errlog 和 LogAdvisor. Errlog 直接应用预定义的代码模式在测试集中恢复 ELS, LogAdvisor 收集训练集中的代码特征, 并用机器学习算法决策测试集中的代码段是否需要加 ELS. 表 2 展示了不同技术的准确率 (P) 和查全率 (R), 实验发现 Errlog 针对 6 款软件中的准确率均高于 LogAdvisor, 但未能在大部分测试集代码段中添加 ELS, 因此查全率低于 LogAdvisor. 相比之下 SmartLog 可以达到更高的准确率和查全率, 一方面判断日志上下文的语义等价性提升了查全率, 另一

```
/* case 1 */
  int ret = foo(a);
  if(ret == 0){
+     printf(...);
      exit();
  }
```

```
/* case 2 */
−foo(a);
+ int ret = foo(a);
+ if(!ret){
+     printf(...);
+ }
```

图 6 日志补丁示例

Figure 6 Examples of log patches

```
/* mysql-5.6.17/client/mysql.cc.patch */
−init_dynamic_string(...);
+ bool init_dynamic_string_0 = init_dynamic_string(...);
+ if(init_dynamic_string_0)
+     smart_log (ID, LOCATION, " init_dynamic_string",\
+         "init_dynamic_string_0", "Out, of, memory");
```

图 7 新增日志的日志信息示例

Figure 7 An example of an additional log statement

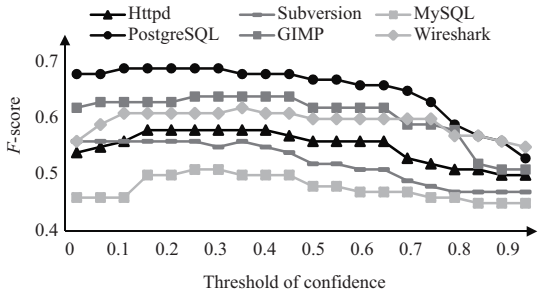


图 8 置信度阈值对挖掘算法准确性的影响

Figure 8 Impacts of the threshold in the mining process

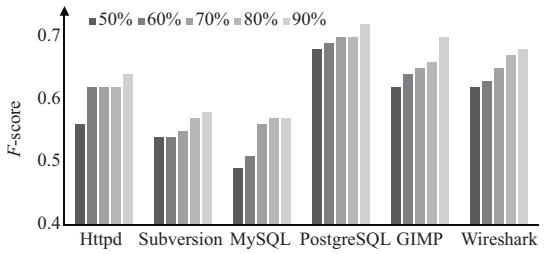


图 9 训练集占数据集比例对挖掘算法准确性的影响

Figure 9 Impacts of the proportion of training data

表 2 不同技术准确率和查全率的对比

Table 2 Comparison of different logging approaches

Project	Errlog		LogAdvisor		SmartLog	
	P	R	P	R	P	R
Httpd	0.73	0.08	0.53	0.44	0.72	0.49
Subversion	0.83	0.08	0.42	0.46	0.67	0.46
MySQL	0.40	0.05	0.29	0.37	0.73	0.40
PostgreSQL	0.83	0.12	0.65	0.48	0.81	0.60
GIMP	0.64	0.23	0.51	0.51	0.81	0.54
Wireshark	0.51	0.10	0.33	0.48	0.82	0.54
Average of F	0.18		0.45		0.61	

方面自动识别 ELS 提升了准确率.

尽管 SmartLog 可以提升准确率和查全率, 但仍存在误报和漏报. 本小节随机采样了 100 个误报和 100 个漏报, 并人工分析其中的原因. 分析显示 (1) 87% 的误报由 `return` 导致, 当一个代码段需要日志的的时候, 开发者经常选择返回一个错误码而非直接输出日志. 因此 SmartLog 建议添加 ELS, 但实际测试集代码段中没有 ELS. (2) 13% 的误报的代码段既无 ELS 也没有返回值, 这种情况可能是目标软件本身的日志缺失, 也可能是相应的日志上下文已通过其他手段处理, 例如内存被安全释放, 无需 ELS. (3) 42% 的漏报是由于日志上下文的判断条件出现无关变量, 导致无法判定语义等价性. (4) 12% 的漏报是由于 SmartLog 自身语义等价性判断能力不足, 如程序使用 `if(*str=='\0')` 和 `if(!str)` 检测 `str` 是否为空, SmartLog 无法获取 `*str=='\0'` 的语义. (5) 46% 的漏报是由于在训练集中, 相关易错函数没有加 ELS, SmartLog 无法学习, 因而没有成功添加 ELS.

表 3　SmartLog 的运行结果

Table 3　Results of SmartLog

Project	Numbers of logging contexts	Numbers of logging rules	Numbers of new logs
Httpd	21624	109	294
Subversion	24843	112	288
MySQL	48971	90	302
PostgreSQL	112736	180	818
GIMP	133127	76	242
Wireshark	250265	78	180
Total	591566	645	2124

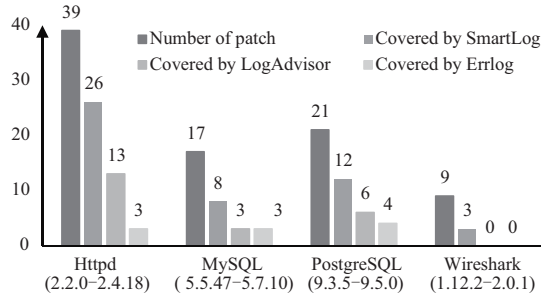


图 10　有效性评估

Figure 10　Effectiveness evaluation

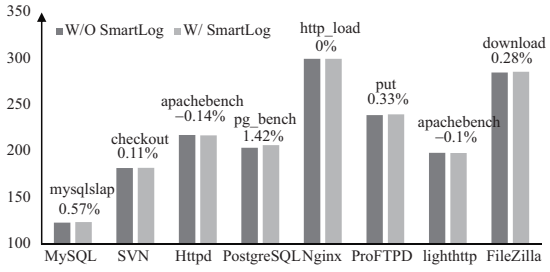


图 11　性能评估

Figure 11　Efficiency evaluation

5.2　有效性和性能评估

SmartLog 可以自动在源码中插入日志, 从而帮助开发人员提升日志质量, 但同时过多的新增日志可能会引起性能下降, 本小节将对 SmartLog 插入日志的有效性和性能进行评估, 并且与已有技术进行对比. 表 3 列举了 SmartLog 的运行结果, 包括在每个目标软件中提取 LCDM 的数量, 挖掘出日志规则的数量以及新增日志的数量. 例如在 Httpd 中, SmartLog 学到了 109 条日志添加规则, 并且在源码中找到了 294 处违反上述规则的代码段, 并添加日志.

有效性评估.　由于缺乏统一的针对日志的评价标准, 因此难以直接评价 SmartLog 新增日志的有效性. 成熟开源软件中的已有日志可以作为很好的经验, 但已有日志本身也经常会被开发人员修改, 因此难以直接作为标准. 本文收集软件演化历史中, 开发人员针对日志的修改补丁, 并将历史上开发人员添加的错误日志语句作为标准. 针对历史上每个添加日志的补丁, 本文将不同的日志添加技术应用于补丁的旧版本, 并评估不同技术是否能够自动添加补丁中的日志. 本文使用 diff 工具收集日志演化补丁, 经人工确认得到 86 个补丁. 图 10 展示了假设使用不同的日志增强技术, 可以自动帮助开发人员添加的日志数量. 例如在 PostgreSQL 中, 从版本 9.3.5 到 9.5.0 识别到 21 个新增日志, 其中 12 个可以被 SmartLog 添加, 而 LogAdvisor 和 Errlog 可以分别添加 6 和 4 个. SmartLog, LogAdvisor 和 Errlog 针对所有补丁的综合覆盖率分别为 57% (49/86), 26% (22/86) 和 12% (10/86). SmartLog 同时有 43% 漏报, 主要原因包括: (1) 相应易错函数在旧版本中没有 ESL (33%); (2) 相应易错函数在旧版本中有 ESL, 但日志上下文的语义不等价 (10%).

性能评估. SmartLog 添加的所有 ELS 均处于特定判断条件之下, 新增日志仅在出错情况下会被触发, 因此新增日志导致的性能下降非常有限. 图 11 展示了评价 SmartLog 性能使用的软件, 以及对应的负载. 每个目标软件包含两个时间: T_{wo} 和 T_{w} 分别表示没有新增日志和有新增日志时, 软件在当前负载下不同的性能. 新增日志产生的性能损失可以通过 $(T_{\mathrm{w}} - T_{\mathrm{wo}})/T_{\mathrm{wo}}$ 计算. 上述过程独立重复运行 5 次, 得到的平均性能损失如图 11 所示. 以 MySQL 为例, T_{wo} 和 T_{w} 分别为 123.3 和 124 s, 因此性能损为 0.57%. 所有目标软件的平均性能损失低于 1%, 其中最高的为 PostgreSQL, 下降 1.4%. 此外, 性能测试中包含了少数 "负损失" (如 Httpd), 可能的原因是软件运行受到环境影响 (如网络延时). 已有技术中, Errlog 的性能损失为 1.1%~1.4%, 和 SmartLog 基本相同. LogAdvisor 被用于在线日志推荐插件, 并没有实际插入日志, 因此没有评价性能损失.

5.3 SmartLog 各主要模块准确性评估

本小节通过人工分析对 SmartLog 设计中的各个模块的准确性进行了评估. 包括了识别错误日志语句的准确性, 提取 LCDM 实例的准确性, 以及归一化后 LIDM 实例的准确性.

识别错误日志语句. SmartLog 使用两步识别算法自动识别错误日志语句 (第 4.1 小节). 为评价 SmartLog 识别过程的准确率和查全率, 本文随机选取了 100 个用于错误日志语句和 100 个其他语句 (包括非错误日志语句和非日志语句). 结果显示 SmartLog 识别的准确率和查全率分别为 98% 和 91%. 产生漏报和误报的原因为识别算法关键字的例外情况.

提取 LCDM 实例. SmartLog 从软件源码中提取 LCDM 实例, 可将源码转化为结构化数据 (第 4.2 小节). 为评价 SmartLog 提取过程的准确率和查全率, 本文随机选取 100 个包含 LCDM 实例的代码段和 100 个不包含 LCDM 实例的代码段. 结果显示 SmartLog 提取的准确率和查全率分别为 100% 和 85%. 产生漏报的主要原因为 LCDM 实例处于复杂控制/数据流中.

归一化 LIDM 实例. LCDM 实例难以评估日志意图之间的等价性, 因此 SmartLog 将 LCDM 实例归一化为 LIDM 实例 (第 4.3 小节). 为评价 SmartLog 转换过程的准确率和查全率, 文本随机选取了 100 对语义等价的日志上下文, 和 100 对语义不等价的日志上下文. 结果显示 SmartLog 转换的准确率和查全率分别为 100% 和 77%. 产生漏报的主要原因为判断条件包含复杂的数据结构、数组、指针或无关变量.

6 相关工作

日志自动增强. 日志自动增强辅助开发人员提升日志质量. Yuan 等 [1] 设计了 Errlog, 通过总结软件易错代码模进而自动添加 ELS. Zhu 等 [2] 设计了 LogAdvisor, 通过学习软件已有日志代码的代码特征进而自动添加 ELS. 本文在第 5 节详细对比了 SmartLog, Errlog 和 LogAdvisor. Yuan 等 [3] 设计了 LogEnhancer, 自动为软件已有日志添加重要的变量, 从而使日志可以更好监控程序运行. Zhao 等 [4] 设计了 Log20, 可以在给定性能约束的情况下自动为程序生成日志, 以便监控程序运行. LogEnhancer 和 Log20 关注面向程序运行路径监控的日志, 而 SmartLog 关注面向故障诊断的日志. Yao 等 [5] 设计了 Log4Perf, 推荐日志位置帮助网页系统监控性能. Ding 等 [6] 设计了 Log2, 一种代价敏感的日志机制帮助程序进行性能诊断. Log4Perf 和 Log2 关注面向性能的日志, 而非面向故障诊断的日志. Li 等 [7] 和 Kim 等 [8] 设计了面向程序日志级别的自动化方法. Cinque 等 [9] 在软件设计阶段形定义日志规则, 而 SmartLog 是直接面向代码自动学习日志规则. Li 等 [10] 通过学习日志的演化历史指导日志变更, 但并非针对故障诊断日志.

日志特征调研. 日志特性调研帮助理解软件日志特点. Yuan 等 [11] 系统研究了日志修改的特征. Fu 等 [12] 组织了调查问卷发现日志是面向故障诊断的主要依据. Kabinna 等 [13] 调研了软件日志语句的稳定性. Chen 等 [14] 研究了代码中的反模式日志行为. Pecchia 等 [15] 评估了软件开发中事件日志的工业实践. Liao 等 [16] 调研了大规模软件系统日志研究进展. Barik 等 [17] 介绍了 Microsoft 的日志与监测实践. Chen 等 [18] 研究了 Java 软件中的日志实践. He 等 [19] 研究了日志语句中的自然语言描述. Zeng 等 [20] 调研了移动应用中的日志实践. 上述工作分析已有日志的特点, 以帮助开发人员更好地了解日志. SmartLog 可以自动添加日志, 以提升已有日志质量.

日志信息分析. 日志信息分析研究如何使用日志进行故障诊断. Xu 等 [21] 通过分析系统内置的控制台日志检测程序错误. Kadav 等 [22] 帮助系统管理人员利用日志修复或替换失效硬件. Yuan 等 [23] 分析软件失效情况下的日志, 以帮助开发人员诊断错误. Pecchia 等 [24] 分析了影响利用日志进行软件失效诊断的准确率的因素. Duan 等 [25] 提出了一种利用日志划分从复杂日志中挖掘块结构过程的方法. Tang 等 [26] 提出了针对大规模软件系统中日志的采集、汇集与存储的方案. Zhong 等 [27] 利用机器学习技术设计并实现了日志解析系统. Zhang 等 [28] 设计了面向应用软件运行日志的收集与服务处理框架. 上述工作分析已有日志语句产生的日志信息, SmartLog 可以提升已有日志质量.

7 结论

当软件系统失效时, 日志是开发人员第一时间进行故障诊断的重要手段, 但已有日志实践常常不足以提供足够的诊断信息. 日志增强工具可以提升日志质量从而缓解上述矛盾. 本文系统调研了面向故障诊断的日志增强工具的现状与局限, 以及导致相关局限的根源: 无法理解软件日志的意图. 因此本文设计了日志意图描述模型, 以判断不同日志意图的等价性. 基于上述模型, 本文设计并实现了自动化工具 SmartLog, 可以理解日志意图, 并在代码中适当的地方插入日志. 通过将不同技术应用于 6 款开源软件, 实验结果发现 SmartLog 相比已有技术可以达到更高的准确性. 同时, 本文收集了软件演化历史中开发人员添加日志的实例, 发现 SmartLog 可以自动添加 57% 的日志, 且带来的性能损失不超过 1%.

参考文献

1 Yuan D, Park S, Huang P, et al. Be conservative: enhancing failure diagnosis with proactive logging. In: Proceedings of Symposium on Operating Systems Design and Implementation, Hollywood, 2012. 293–306

2 Zhu J M, He P J, Fu Q, et al. Learning to log: helping developers make informed logging decisions. In: Proceedings of International Conference on Software Engineering, Florence, 2015. 415–425

3 Yuan D, Zheng J, Park S, et al. Improving software diagnosability via log enhancement. Trans Comput Syst, 2011, 46: 3–14

4 Zhao X, Rodrigues K, Luo Y, et al. Log20: fully automated optimal placement of log printing statements under specified overhead threshold. In: Proceedings of Symposium on Operating Systems Principles, Shanghai, 2017. 565–581

5 Yao K D, Padua G, Shang W Y, et al. Log4Perf: suggesting logging locations for web-based systems' performance monitoring. In: Proceedings of International Conference on Performance Engineering, Berlin, 2018. 127–138

6 Ding R, Zhou H C, Lou J G, et al. Log2: a cost-aware logging mechanism for performance diagnosis. In: Proceedings of Annual Technical Conference, Santa Clara, 2015. 139–150

7 Li H, Shang W Y, Hassan A E. Which log level should developers choose for a new logging statement? Empir Softw Eng, 2017, 22: 1684–1716

8 Kim T, Kim S, Yoo C, et al. An automatic approach to validating log levels in Java. In: Proceedings of Asia-Pacific Software Engineering Conference, Nara, 2018. 623–627

9 Cinque M, Cotroneo D, Pecchia A. Event logs for the analysis of software failures: a rule-based approach. IEEE Trans Softw Eng, 2013, 39: 806–821

10 Li S S, Niu X, Jia Z Y, et al. Guiding log revisions by learning from software evolution history. In: Proceedings of Conference on Program Comprehension, Gothenburg, 2018. 178–188

11 Yuan D, Park S, Zhou Y Y. Characterizing logging practices in open-source software. In: Proceedings of International Conference on Software Engineering, Zurich, 2012. 102–112

12 Fu Q, Zhu J M, Hu W L, et al. Where do developers log? An empirical study on logging practices in industry. In: Proceedings of International Conference on Software Engineering, Hyderabad, 2014. 24–33

13 Kabinna S, Shang W, Bezemer C, et al. Examining the stability of logging statements. In: Proceedings of International Conference on Software Analysis, Evolution, and Reengineering, Osaka, 2016. 326–337

14 Chen B Y, Jiang Z M. Characterizing and detecting anti-patterns in the logging code. In: Proceedings of International Conference on Software Engineering, Buenos Aires, 2017. 71–81

15 Pecchia A, Cinque M, Carrozza G, et al. Industry practices and event logging: assessment of a critical software development process. In: Proceedings of International Conference on Software Engineering, Florence, 2015. 169–178

16 Liao X K, Li S S, Dong W, et al. Survey on log research of large scale software system. J Softw, 2016, 27: 1934–1947 [廖湘科, 李姗姗, 董威, 等. 大规模软件系统日志研究综述. 软件学报, 2016, 27: 1934–1947]

17 Barik T, DeLine R, Drucker S, et al. The bones of the system: a case study of logging and telemetry at Microsoft. In: Proceedings of International Conference on Software Engineering Companion, Austin, 2016. 92–101

18 Chen B Y, Jiang Z M. Characterizing logging practices in Java-based open source software projects — a replication study in apache software foundation. Empir Softw Eng, 2017, 22: 330–374

19 He P J, Chen Z B, He S L, et al. Characterizing the natural language descriptions in software logging statements. In: Proceedings of International Conference on Automated Software Engineering, Montpellier, 2018. 178–189

20 Zeng Y, Chen J F, Shang W Y, et al. Studying the characteristics of logging practices in mobile Apps: a case study on F-Droid. Empir Softw Eng, 2019, 24: 3394–3434

21 Xu W, Huang L, Fox A, et al. Detecting large-scale system problems by mining console logs. In: Proceedings of Symposium on Operating Systems Principles, Big Sky, 2009. 117–132

22 Kadav A, Renzelmann M, Swift M. Tolerating hardware device failures in software. In: Proceedings of Symposium on Operating Systems Principles, Big Sky, 2009. 59–72

23 Yuan D, Mai H H, Xiong W W, et al. SherLog: error diagnosis by connecting clues from run-time logs. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, 2010. 143–154

24 Pecchia A, Russo S. Detection of software failures through event logs: an experimental study. In: Proceedings of International Symposium on Software Reliability Engineering, Dallas, 2012. 31–40

25 Duan R, Fang H, Zhan Y. Approach for mining block structure process from complex log using log partitioning. Comput Sci, 2019, 46: 334–339 [段瑞, 方欢, 詹悦. 一种利用日志划分从复杂日志中挖掘块结构过程的方法. 计算机科学, 2019, 46: 334–339]

26 Tang W X, Wang J H, He L J, et al. Design and implementation of log collection service platform for large-scale software system. Comput Appl Softw, 2018, 35: 173–178 [汤网祥, 王金华, 赫凌俊, 等. 大规模软件系统日志汇集服务平台设计与实现. 计算机应用与软件, 2018, 35: 173–178]

27 Zhong Y, Guo Y B. Design and implementation of log parsing system based on machine learning. J Comput Appl, 2018, 38: 352–356 [钟雅, 郭渊博. 基于机器学习的日志解析系统设计与实现. 计算机应用, 2018, 38: 352-356]

28 Zhang X, Ying S, Zhang T. Collection and service processing framework of application running log. Comput Eng Appl, 2018, 54: 81–89 [张骁, 应时, 张韬. 应用软件运行日志的收集与服务处理框架. 计算机工程与应用, 2018, 54: 81–89]

29 Jia Z Y, Li S S, Liu X D, et al. SMARTLOG: place error log statement by deep understanding of log intention. In: Proceedings of International Conference on Software Analysis, Evolution and Reengineering, Campobasso, 2018. 61–71

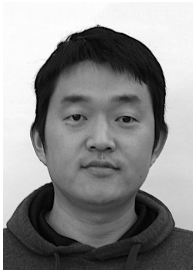
Intention-aware log automation

Zhouyang JIA, Shanshan LI*, Xiaodong LIU, Ji WANG & Xiangke LIAO

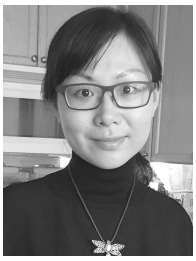
College of Computer Science, National University of Defense Technology, Changsha 410072*, China*
* Corresponding author. E-mail: shanshanli@nudt.edu.cn

Abstract Log automation is a technique that helps developers write high-quality log code. When software systems fail, log code can ease the failure diagnosis process and reduce system recovery time. Existing log automation tools can be roughly classified into two groups: feature- and pattern-based tools. These existing tools define log placement rules by either extracting syntax features or summarizing code patterns, but they are hard to understand the source code's intention. In this paper, we design and implement SmartLog, which can create log statements based on deep understanding of log intention. To achieve this, we propose a log intention description model to describe the intention of log statements. SmartLog then explores the intentions of existing logs and mines log rules from those intentions. We evaluated SmartLog on six mature open-source projects. Compared with two state-of-the-art projects, i.e., Errlog and LogAdvisor, SmartLog improved the accuracy of log placement by 43% and 16% respectively. SmartLog could cover 49 out of 86 real-world patches aimed to add logs, while the state-of-the-art works could cover 10 and 22 patches, respectively.

Keywords log enhancement, failure diagnosis, log evolution, software intention, log automation



Zhouyang JIA was born in 1994. He received his B.S. and M.S. degrees from the College of Computer Science at National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively. He was a visiting student at University of Kentucky, USA, from 2018 to 2020. He is currently working toward a Ph.D. degree at National University of Defense Technology. His research interests include software reliability and software availability.



Shanshan LI was born in 1980. She received her M.S. and Ph.D. degrees from the College of Computer Science at National University of Defense Technology, Changsha, China, in 2003 and 2007, respectively. She was a visiting scholar at The Hong Kong University of Science and Technology in 2007. She is currently a professor at National University of Defense Technology. Her research interests include software quality enhancement, defect prediction, and misconfiguration diagnosis.

Xiaodong LIU was born in 1985. He received his B.S. and Ph.D. degrees from the College of Computer Science at National University of Defense Technology, Changsha, China, in 2007 and 2013, respectively. He is currently an assistant professor at National University of Defense Technology. His research interests include parallel computing, operating systems, and software quality enhancement.

Ji WANG was born in 1969. He received his Ph.D. degree from the College of Computer Science at National University of Defense Technology, Changsha, China, in 1995. He is a professor at National University of Defense Technology. He currently serves as the vice director of the Academic Board of the State Key Laboratory of High-Performance Computing in China. His research interests include programming methodology, program verification, and formal methods.