计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(1):376−384 [doi: 10.15888/j.cnki.csa.008884]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

# 抽象语义引导的空指针引用自动修复①

王　珣, 孙玉雪, 董玉坤, 位欣欣, 唐道龙

(中国石油大学 (华东) 计算机科学与技术学院, 青岛 266580)
通信作者: 董玉坤, E-mail: dongyk@upc.edu.cn

**摘　要**: 程序依赖图往往只能根据语句中变量的定义使用关系来判定数据依赖而无法从语义上精准判断, 从而容易引入虚假依赖关系, 使得缺陷修复的过程中使用错误信息造成修复失败. 因此, 本文将利用抽象属性对与空对象或空指针有关的虚假依赖进行剪枝, 提出基于抽象语义的程序依赖图减少与程序缺陷语义无关的依赖关系分析, 以完成空指针引用修复. 依据分析获取的依赖关系, 在空指针引用的不同修复策略的指导下实现一种多策略的修复方案, 在尽可能减小修复副作用的前提下完成空指针引用缺陷的修复. 本文利用 Defects4J 中的空指针引用对实现的修复工具 DTSFix 进行实验评估, 结果显示 DTSFix 的修复效果远远高于对比工具, 证明了方法的有效性.

**关键词**: 抽象语义; 程序依赖图; 程序自动修复; 空指针引用; 修复策略

引用格式: 王珣,孙玉雪,董玉坤,位欣欣,唐道龙.抽象语义引导的空指针引用自动修复.计算机系统应用,2023,32(1):376–384. http://www.c-s-a.org.cn/1003-3254/8884.html

## Automatic Repair for Null Pointer References Guided by Abstract Semantics

WANG Xun, SUN Yu-Xue, DONG Yu-Kun, WEI Xin-Xin, TANG Dao-Long

(College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China)

**Abstract**: Program dependency graph usually judges the data dependency according to definition-use relationships of variables in statements, and it cannot make an accurate judgment according to the semantics, which leads to the introduction of false dependency relationships and the repair failure caused by the use of error information in repairing defects. Therefore, this study will prune false dependencies related to null objects or null pointers by using abstract attributes and propose an abstract semantic-based program dependency graph to reduce the analysis of dependency relationships unrelated to the semantics of program defects and repair null pointer references. Based on the dependency relationships obtained from the analysis, a multi-strategies repair scheme is implemented under the guidance of different repair strategies for null pointer references, and the null pointer references are repaired with side effects minimized as much as possible. In addition, in this study, the null pointer references in Defects4J are adopted to evaluate the repair tool DTSFix through experiments. The results show that the repair effect of DTSFix is much better than that of other tools, which proves the effectiveness of the method.

**Key words**: abstract semantics; program dependency graph; automatic program repair; null pointer references; repair strategies

在软件开发及维护过程中, 人工进行程序缺陷修复极大消耗和降低软件开发人员的精力和效率. 空指针引用作为程序缺陷中一类隐蔽缺陷的存在, 不易察觉和修复. 据研究表明空指针引用在缺陷数量中所占比例超过 20%, 且修复平均时间达到 20 min. 为了提高修复效率、降低修复成本, 自从 2009 年开始, 程序自

① 基金项目: 山东省自然科学基金 (ZR2021MF058)
收稿时间: 2022-05-11; 修改时间: 2022-06-15, 2022-06-23; 采用时间: 2022-06-27; csa 在线出版时间: 2022-08-24
CNKI 网络首发时间: 2022-11-15

动修复技术[1–4] 成为研究热点, 根据总结分析已有的程序自动修复技术可分为基于启发式搜索、基于人工模板、基于语义约束和基于统计分析的修复技术[5]. 基于启发式搜索[6] 和基于统计分析[7] 的技术常被用来多类缺陷的通用修复策略的生成, 基于人工模板[8] 的修复技术适用于特定缺陷的补丁生成, 基于语义约束修复策略[9] 皆适用于两种情况. 但是由于造成程序缺陷的原因以及缺陷的影响范围存在不同, 对多类缺陷采取统一的修复策略可能会降低候选补丁的准确率, 所以针对某一特定的缺陷的修复更为有效.

为了实现空指针引用自动修复, 需要充分掌握程序间的依赖关系, 从而准确识别缺陷的修复位置, 采纳合理的修复方案. 而程序依赖图[10,11] 包含数据依赖和控制依赖两个不同层次的依赖关系的融合, 可以充分掌握程序中的依赖关系. 但是数据依赖是由程序语句中的定义使用关系来识别, 这就忽略了语义本身所赋予的依赖关系. 例如, (1) $x=a+a$; (2) $z=y+x\%2$, 根据语法中的定义和使用关系, $z$ 数据依赖于 $\{y, x\}$, 然而 $x\%2\equiv 0$, 在 (1) 的前提下, $x$ 的取值并不会影响表达式 $x\%2$ 的值, 也就是说, 实际上 $z$ 并不取决于 $x$ 的值. 从语义上来说 $y+x\%2$ 中的 $y$ 是唯一与 $z$ 相关的变量, 程序依赖图从语法上分析可能并不能计算出精确的依赖集, 因此并不能排除这种虚假依赖. 这就表明程序依赖图所识别的数据依赖关系超过了所需的依赖关系, 需要考虑真正的依赖关系及其传播.

本文做出了以下贡献.

(1) 提出了基于抽象语义的程序依赖图的空指针引用分析方法. 利用对抽象语义关联性的衡量, 忽略所有与抽象属性的不关联的语句节点进一步优化程序依赖图, 获取空指针引用的数据和控制依赖关系, 为选取修复位置及修复策略提供所需信息.

(2) 提出赋值、约束、规避、转移 4 种修复策略. 根据不同空指针引用的引发原因及缺陷所影响的代码集合, 依靠上下文信息选择 4 种修复策略实现修复.

(3) 提出基于语义距离的候选补丁排序方法. 通过计算不同候选补丁与原程序之间语义距离, 比较不同补丁对原程序的影响, 把所获得的候选补丁按照语义距离由小到大排列, 推荐最优补丁.

本文的其余部分组织如下: 第 1 节阐述基于抽象语义的程序依赖图的构建; 第 2 节介绍空指针引用的 4 种修复策略; 第 3 节解释如何通过语义距离对候选补丁的排序; 第 4 节展开了一系列的实验对比讨论, 评估本文方法的有效性; 第 5 节对本文工作进行总结, 同时展望下一步的研究工作.

## 1 基于抽象语义的程序依赖图

研究表明, 数据依赖关系和控制依赖关系可以获取程序上下文详细信息, 然而程序依赖图中数据依赖关系的分析通过变量的定义使用链仅分析变量语法上之间的关系, 并不能识别变量之间是否真正具有语义之间的关联. 具体语义的关联性的计算繁琐且耗时, 并且在程序修复过程中往往只关注所需某一种语义属性. 基于抽象语义的程序依赖图恰恰可以采用抽象属性对程序依赖图进行简化, 将虚假的数据依赖关系筛除, 准确分析与空指针或空对象相关的数据和控制依赖关系. 并且在下文的空指针引用的 4 种修复策略中, 可利用优化后的数据和控制依赖关系精确识别程序上下文完成修复. 在此过程中, 需采取某种抽象属性的语义关联性对特定的语义相关度进行近似计算, 而这也是将程序依赖图转换为基于抽象语义的程序依赖图[12–14] 的关键. 以下是基于抽象语义的程序依赖图的相关概念.

定义 1. 控制依赖关系. 给定控制流图 $G=<N,E,Entry,Exit>$, $\exists n_1,n_2\in N$. $P$ 为 $G$ 中从 $n_1$ 到 $n_2$ 的执行路径, 在 $P$ 中除了 $n_1$ 和 $n_2$ 其他所有的节点都被 $n_2$ 后向支配, 并且 $n_1$ 并不被 $n_2$ 后向支配, 则 $n_2$ 控制依赖于 $n_1$, 被定义为 $n_1 \xrightarrow{CD} n_2$.

定义 2. 数据依赖关系. 给定控制流图 $G=<N,E,Entry,Exit>$, $\exists n_1,n_2\in N, P$ 为 $G$ 中从 $n_1$ 到 $n_2$ 的执行路径. 节点 $n_1$ 定义变量 $v$ 在节点 $n_2$ 中被使用, 且在 $P$ 中不存在对 $v$ 的重定义, 则 $n_2$ 数据依赖于 $n_1$, 也就是 $n_1 \xrightarrow{DD} n_2$.

定义 3. 程序依赖图. 给定一个有向图 $G=<N,E>$, $E=\{<n_i,n_j> \mid n_i\in N,n_j\in N,n_i \xrightarrow{DD} n_j \ or \ n_i \xrightarrow{CD} n_j\}$, 即所有的边均为控制依赖边或数据依赖边则称为程序依赖图.

定义 4. 抽象属性的数据依赖关系. 给定一个程序依赖图 $G=<N,E>$, 语句节点 $n_a$ 和抽象属性 $\rho$, 当 $n_a$ 在抽象状态 $\sigma$ 上执行时, 若抽象属性 $\rho$ 的状态发生了改变, 则表明 $n_a$ 与 $\rho$ 语义相关. 此时, $n_a$ 所连接的数据依赖边才可保留. 抽象数据依赖边可表示为 $E_{DD*}=\{e\in <n_a,n_b> \mid n_a,n_b\in N, n_a \xrightarrow{DD} n_b \cap \sigma^{\rho}\neq \sigma_a^{\rho}\}$, 其中 $\sigma^{\rho}$ 表示当前语句关于 $\rho$ 的抽象状态, $\sigma_a^{\rho}$ 表示执行完 $n_a$ 后关于 $\rho$ 的抽象状态.

把抽象语义代替具体语义分析程序语句关联性时, 可以将程序依赖图中包含的控制依赖关系和数据依赖关系进一步精炼, 获得基于抽象语义的程序依赖图$G_{pdg}=<N_{pdg},E_{pdg}>$, $N_{pdg}$表示语句的节点集合, $E_{pdg}$是表示节点间控制依赖边或抽象数据依赖边. 为了识别空指针引用的精确依赖, 将把识别空指针的相关信息作为抽象属性来进一步优化程序依赖图.

根据修复空指针引用的分析可知, 判定一个赋值语句或者方法调用中的所获得值是否是空值是追踪空指针的关键因素, 故设定抽象属性$\rho_1=\{v=e\,|e\in Exp, v\xleftarrow{\text{null}}e\}$, $\rho_2=\{m(\vec{r})\,|\vec{r}\leftarrow\text{null}\}$. 在$\rho_1$中, $v$ 为变量, $e$ 为表达式, 表示在语句 $v=e$ 中, $e$ 中传递的值为 null. 在$\rho_2$中$m(\vec{r})$为方法调用表达式, 且参数$\vec{r}$为一个空值. 通过利用抽象属性的关联性梳理程序依赖图中的语句节点是否具有空指针传递的数据依赖关系, 删除虚假的数据依赖, 获取抽象数据依赖边.

例如, 示例 1 代表一个存在空指针引用的缺陷程序, cond 为 false, 且通过指针的传递在 L7 中造成空指针引用. 对于程序依赖图而言, 可以通过数据依赖边和控制依赖边构建出来如图 1 所示. 但是在分析过程中可以发现, 语句节点 2、4、5 和 6 并没有涉及空值的传递, 因此这 4 个语句节点与抽象属性$\rho_1$和$\rho_2$是语义无关的, 可将语句节点 2、4、5 和 6 及其数据依赖边和控制依赖边剪枝, 通过抽象数据依赖边的规则进一步识别精确的依赖关系, 最终获得基于抽象语义的程序依赖图 (图 2).

示例 1. 存在空指针引用的缺陷程序

```
L1  public Mode analyzeAction(Boolean cond, Parser p, Mode type){
L2    ConstructParser construct = new ConstructParser();
L3    Action action = p.getAction();
L4    if (cond) {
L5        Parser parser = construct.getParser(type);
L6        action = parser.changeAction(); }
L7    Mode mode = action.mode;
L8    return mode;
L9  }
```

为了获得关于抽象属性$\rho_1$和$\rho_2$的程序依赖图, 首先将分析抽象状态是否在程序中的赋值语句以及方法调用语句的语句点上发生变化, 忽略与抽象属性语义无关的赋值语句或者方法调用语句. 之后将对优化后的控制语句分为 3 类对其进行算法 1 所示的操作. 第 1 类是 while 语句, 在 L4 中$blk_{\text{control}}$与抽象属性语义无关可忽略. 第 2 类是 if 语句, 在 L6 中$blk_{\text{if}}$若与抽象属性语义无关可忽略. 第 3 类是 if…else…语句, 在 L9–L14 中将分情况讨论, 在 L9–L10 中, 若$blk_{\text{if}}$和$blk_{\text{else}}$与抽象属性语义无关皆可忽略, 在 L11–L12 中, 若$blk_{\text{else}}$与抽象属性语义无关则仅忽略$blk_{\text{else}}$, 在 L13–L14 中, 若$blk_{\text{if}}$与抽象属性语义无关则仅忽略$blk_{\text{if}}$. 最后依据所保留的与抽象语义相关的语句构建基于抽象语义的程序依赖图.

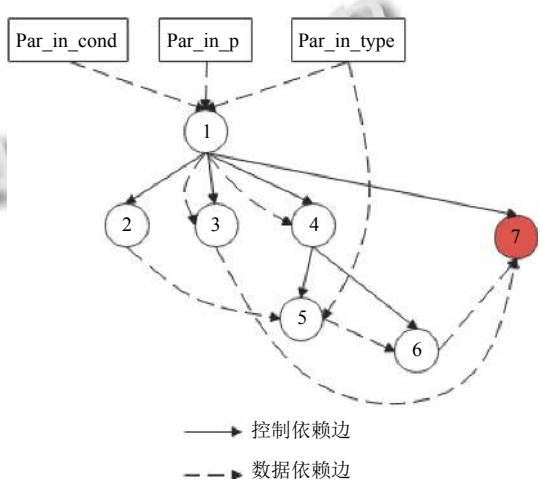


图 1 程序依赖图

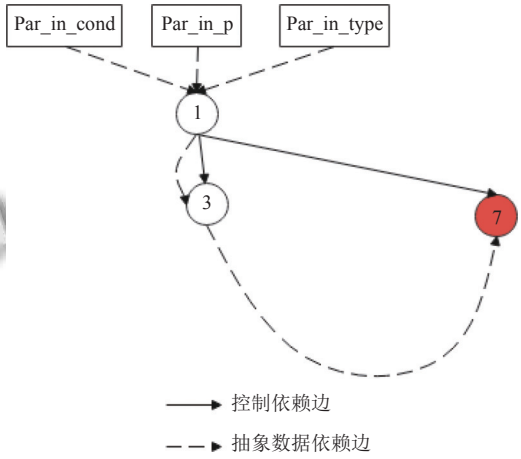


图 2 基于抽象语义的程序依赖图

算法 1. 构建基于抽象语义的程序依赖图的方法

1: **for** $s\in s_{\text{control}}$
2: **case1**: while (*cond*) then $blk_{\text{control}}$
3: **if** $\sigma(blk_{\text{control}})(\rho)=\sigma(blk_{\text{control}}')$
4: while (*cond*) then $blk_{\text{control}}\leftarrow$while (*cond*) then *skip*
5: **case2**: if (*cond*) then $blk_{\text{if}}$
6: **if** $\sigma(blk_{\text{if}})(\rho)=\sigma(blk_{\text{if}}')$
7: if (*cond*) then $blk_{\text{if}}\leftarrow\phi$

8: **case3**: if (*cond*) then $blk_{if}$ else $blk_{else}$
9: **case3a**: $\sigma(blk_{if})(\rho)=\sigma(blk_{if}')\cap\sigma(blk_{else})(\rho)=\sigma(blk_{else}')$
10: if (*cond*) then $blk_{if}$ else $blk_{else}\leftarrow\phi$
11: **case3b**: $\sigma(blk_{else})(\rho)=\sigma(blk_{else}')$
12: if (*cond*) then $blk_{if}$ else $blk_{else}\leftarrow$if (*cond*) then $blk_{if}$
13: **case3c**: $\sigma(blk_{if})(\rho)=\sigma(blk_{if}')$
14: if (*cond*) then $blk_{if}$ else $blk_{else}\leftarrow$
if (*cond*) then *skip* else $blk_{else}$
15: **end for**

## 2 空指针引用的修复

通过基于抽象语义的程序依赖图获取程序内精确的依赖关系后, 将结合空指针引用修复策略进行修复. 为了避免对程序其他功能造成隐患, 减小缺陷修复所带来的副作用, 本文尽力在最小的程序代码范围内实现精确修复, 因此在修复过程中把对程序执行路径的影响程度作为评价标准, 根据其影响程度由小到大依次提出了赋值策略、约束策略、规避策略和转移策略 4 种修复策略.

### 2.1 赋值策略

赋值策略主要是对缺陷变量通过类型适配进行赋值, 其修复方式分为两类: 第 1 类, 寻找缺陷变量的定义位置, 在定义位置上利用非空的值替换错误赋值; 第 2 类, 在缺陷变量使用之前, 预先进行判断缺陷变量是否为空, 若是则进行恰当赋值操作. 式 (1) 和式 (2) 分别代表这两类修复方式:

$$\frac{DU(l_{def},l_{use},v)\ \ l_{def}:v=\_\ \ l:v=new\_value;}{l_{def}\rightarrow l} \tag{1}$$

$$\frac{DU(l_{def},l_{use},v)\ \ c:v==\text{null}\ \ l:v=new\_value;}{l_{use}\rightarrow \text{if}(c)\{l\}\ l_{use}} \tag{2}$$

其中, $DU(l_{def},l_{use},v)$代表变量 $v$ 的定义使用关系, $l_{use}$表示变量 $v$ 的使用的语句, $l_{def}$表示变量 $v$ 的定义的语句. $c$ 表示 if 语句中的条件表达式, $l$ 表示需要插入的语句. 式 (1) 则借助依赖关系寻找到缺陷变量的定义位置 $l_{def}$重新赋值. 在式 (2) 中将在缺陷变量使用语句$l_{use}$之前增加 if 判断及重新赋值语句 $l$.

替换缺陷变量的值 *new_value* 的选取分为 4 种: 第 1 种, 若缺陷点存在与返回值类型相同的活跃变量, *new_value* 为该活跃变量; 第 2 种, 若 $v$ 的类型为第三方 API 库, 则将修复位置的活跃变量与构造方法的形式参数的类型相比较, 选取最佳的构造方法的实例对象作为 *new_value*; 第 3 种, 若 $v$ 的类型为自定义类型且只有一个构造方法, 则 *new_value* 为当前构造方法实例对象, 若有多个构造方法, 则可以根据当前程序点的活跃变量与构造方法的参数匹配选择最佳构造方法的实例对象; 第 4 种, 若 $v$ 为有子类的自定义类型. 根据变量名与子类名之间相似度选择最佳匹配, 则该子类实例对象被选择为 *new_value*.

### 2.2 约束策略

约束策略是通过对缺陷变量的提前进行条件判断来约束后续程序的执行, 其修复方式分为两类: 第 1 类, 将受到缺陷变量影响的代码集合预先放入条件判断中, 保证在缺陷变量不为空的情况下代码集合才可执行; 第 2 类, 将受到缺陷变量影响的代码集合放入 try 语句中, 在缺陷变量出现异常时抛出空指针异常. 式 (3) 和式 (4) 分别表示这两类情况:

$$\frac{DU(l_{def},l_{use},v)\ \ c:v!=\text{null}\ \ ds(v)=\{l_1,l_2,\cdots,l_n\}}{l_{use}\rightarrow \text{if}(c)\{ds(v)\}} \tag{3}$$

$$\frac{\begin{array}{c}DU(l_{def},l_{use},v)\ \ ds(v)=\{l_1,l_2,\cdots,l_n\}\\ l:\text{try}\{ds(v)\}\ \text{catch}(\text{NullPointerException}\ e)\{\cdots\}\end{array}}{l_{use}\rightarrow l} \tag{4}$$

其中, $ds(v)$ 代表由缺陷点传递的错误信息影响之后执行的语句集合, 可通过基于抽象语义的程序依赖图所获取的依赖关系识别出缺陷变量的影响语句集合.

### 2.3 规避策略

规避策略根据不同情况添加适当中断语句, 转移当前方法的执行来避免空指针引用触发, 其修复方式可分为 3 类: 第 1 类, 若被缺陷变量所影响的代码集中包含 return 语句, 为了避免约束受影响代码集合时出现无返回值的情况, 则需在空条件判断中增加恰当返回值; 第 2 类, 若被缺陷变量所影响的代码集合中包含 return 语句, 也可将代码块放入 try 语句中, 将在 catch 语句中增加适当返回值; 第 3 类, 若缺陷变量在循环的某次循环过程中出现, 可添加 continue 提前结束当前这次异常循环过程的执行. 其中恰当的返回值$value_{return}$分为 5 种, 第 1 种, 若返回值类型为基本数据类型, $value_{return}$为基本数据类型的默认值; 其余 4 种与 *new_value* 获取方式相同. 式 (5)–式 (7) 分别表示这 3 类修复方式:

$$\frac{\begin{array}{c}DU(l_{def},l_{use},v)\ \ c:v==\text{null}\\ ds(v)=\{l_1,l_2,\cdots,l_n\}\ \ l:\text{return}\ \ value_{return};\end{array}}{l_{use}\xrightarrow{\text{type}(l_1\sim l_n)\in\text{return}}\text{if}(c)\{l\}\ l_{use}} \tag{5}$$

$$\frac{DU(l_{\text{def}},l_{\text{use}},v)\ \ ds(v)=\{l_1,l_2,\cdots,l_n\}\ \ l:\text{try}\{ds(v)\}\ \text{catch(NullPointerException}\ e)\{\text{return}\ value_{\text{return}};\}}{l_{\text{use}}\xrightarrow{\text{type}(l_1\sim l_n)\in\text{return}}l} \tag{6}$$

$$\frac{DU(l_{\text{def}},l_{\text{use}},v)\ \ c:v==\text{null}}{l_{\text{use}}\xrightarrow{\text{type}(d\_dep(v))\in\text{loop}}\text{if}(c)\{\text{continue};\}\ l_{\text{use}}} \tag{7}$$

其中, $d_dep(v)$表示缺陷变量 $v$ 数据依赖的代码集合, $\text{type}(d_dep(v))\in\text{loop}$表示$d_dep(v)$的语句类型是否包含循环语句类型. 式 (5) 和式 (6) 分别采用了 if 和 try 对缺陷变量影响的代码集合进行约束, 并在出现空对象时将恰当的返回值返回. 式 (7) 首先判断缺陷对象的数据依赖代码集合中是否包含循环语句, 若是则添加 continue 结束本次循环, 进入下次循环.

### 2.4 转移策略

若空指针的来源作为参数传递进入方法中, 转移策略通过在空指针被引用之前添加判断条件, 在判断条件为真时将空指针异常抛给外部调用方法处理. 式 (8) 在缺陷变量为空时添加 $l$ 语句将异常抛出, 由外部调用方法捕获处理.

$$\frac{DU(l_{\text{def}},l_{\text{use}},v)\ \ c:v!=\text{null}\ \ ds(v)=\{l_1,l_2,\cdots,l_n\}\ \ l:\text{throw new NullPointerException}(\cdots);}{l_{\text{use}}\rightarrow\text{if}(c)\{ds(v)\}\ \text{else}\{\ l\ \}} \tag{8}$$

### 2.5 修复策略的匹配

考虑修复程序需在最小范围内的改动来消除空指针引用, 提出算法 2 来描述 4 种修复策略的匹配方法. Step 1 获取当前程序缺陷的上下文信息, 之后判定是否满足修复策略所需条件并应用对应的修复操作. 因赋值策略的修复是从根本上改变缺陷变量的值完成修复, 所以首先在 Step 4 中判断的赋值策略的条件是否满足. 在此之后为了避免空指针的触发, Steps 5−6 通过提取特定的条件选取约束策略或规避策略. 当之前 3 种策略均不可获得正确补丁时, Step 7 判定是否满足转移策略的修复条件, 实施对应的修复操作. 最后在 Step 10 中验证获得的补丁是否通过所有的测试用例获取候选补丁.

算法 2. 4 种修改策略的匹配方法

输入: $G$: semantic-based abstract program dependency graph
输出: $p_{\text{can}}$: the candidate patches; bugs: the information of defects; repair_oper (m): the repair operation

```
Begin
1.  bugs←extract_infor_for_bugs(G)
2.      p_app,p_can←ϕ
3.  for bugs_infor ∈ bugs do
4.      case1: assignment_infor⊂bug_infor then
            p_app←repair_oper(assignment)
5.      case2: evading_infor⊂bug_infor then
            p_app←repair_oper(evading)
6.      case3: restraint_infor⊂bug_infor then
            p_app←repair_oper(restraint)
7.      case4: transfer_infor⊂bug_infor then
            p_app←repair_oper(transfer)
8.  end for
9.  for p ∈ p_app do
10.     if p passes test cases
11.         p_can←p
12.     end if
13. end for
14. return p_can
END
```

## 3 基于语义距离的补丁排序

经过匹配修复模板实施修复操作之后, 为了优先获取高质量的候选补丁, 将定义一些的评判规则对补丁进行排序. 在基于静态分析的补丁评价规则中一般分为语义距离[15] 和语法距离两种[16]. 但是由于基于修复模板的缺陷修复本身就是通过不同的修复方式总结出的不同修复策略[17], 通过语法距离评判修复补丁并不具有客观性, 因此在本文中选择基于语义距离对候选补丁进行排序.

补丁程序 $P'$是对缺陷程序 $P$ 进行有限的程序修改获得的, 在$T_{\text{sat}}\subseteq T$包含 $P$ 和 $P'$满足的所有的测试用例的执行过程中, 给定一个$t\subseteq T_{\text{sat}}$, $\pi(t)=(\eta_0,\eta_1,\cdots,\eta_n)$和$\pi'(t)=(\eta'_0,\eta'_1,\cdots,\eta'_n)$分别表示 $P$ 和 $P'$在 $t$ 的执行, 其中$\eta_n$是$(l_n,v_n)$形式的元组, $l_n$为语句位置, $v_n$为相应的变量取值, 则 $P$ 和 $P'$二者的语义距离可定义为:

$$d_\zeta(P,P',T)=\begin{cases}\infty, & \text{if}\ \forall LOC:P'\notin R_{LOC}(P)\ \text{or}\ T_{\text{sat}}\ \text{is empty}\\ \sum\limits_{t\in T_{\text{sat}}}(|M-K|+\sum_{h=0}^{min}diff(\eta_n,\eta'_n)), & \text{otherwise}\end{cases} \tag{9}$$

其中, $M$ 和 $K$ 分别表示程序 $P$ 和 $P'$在 $T$ 上执行路径的长度, $min$ 表示 $M$ 和 $K$ 中的较小值, $diff(\eta_h,\eta'_h)$表示在两次执行中比较执行位置和变量值, 如果$\eta_h=\eta'_h$, $diff(\eta_h,\eta'_h)$的值为 0, 否则为 1.

候选补丁 $P'$和候选补丁 $P''$为示例 1 的两个候选补丁, 分别代表两种不同的修复方式. 表 1 记录了一次测试用例中原程序与两个候选补丁的语义距离计算, 假设 $cond$=true , $p$=$p$, $type$=$type$ 为输入参数, 因为参数

和 *construct* 的值未改变, 故并未将其添加到表 1 中. 其中, loc 为执行位置, 其余均为程序内变量. 从表 1 可以看出, 在示例 1 缺陷程序 $P$ 中 action 和 mode 分别在代码行 3 和 7 被定义且赋值, 且 action 在代码行 6 中被重新赋值. 根据候选补丁 $P'$和候选补丁 $P''$中的代码来标记 action 和 mode 定义和赋值情况. 通过比较执行位置 loc 和程序变量状态的一致性, 在$diff(P,P')$中可以发现 Steps 6−8 这 3 列中 $P$ 和 $P'$程序变量状态并不一致标记为 1, 同理, 在$diff(P,P'')$中 Steps 3−5 中 $P$ 和 $P''$程序变量状态也不一致, 因此, $\sum diff(P,P') = \sum diff(P,P'') = 3$. 但在表 1 中 Steps 9−10, 候选补丁 $P'$比 $P$ 的执行距离多 2 步. 综上, 根据式 (9), $d_\zeta(P,P',T)=3+2=5$, $d_\zeta(P,P'',T)=3$. 因此可以判定候选补丁 $P''$为更加高质量的补丁.

代码 1. 候选补丁 $P'$

```
L1   public Mode analyzeAction(Boolean cond, Parser p, Mode type){
L2     ConstructParser construct = new ConstructParser();
L3     Action action = p.getAction();
L4     if (cond) {
L5       Parser parser = construct.getParser(type);
L6       action = parser.changeAction(); }
L7(+) if (acyion == null) {
L8(+)   action = new Action(t); }
L9     Mode mode= action.mode
L10    return mode;
L11  }
```

代码 2. 候选补丁 $P''$

```
L1   public Mode analyzeAction(Boolean cond, Parser p, Mode type){
L2     ConstructParser construct = new ConstructParser();
L3(−)    Action action = p.getAction();
L3(+)    Action action = new Action (t);
L4     if (cond) {
L5       Parser parser = construct.getParser(type);
L6       action = parser.changeAction(); }
L7     Mode mode = action.mode;
L8     return mode;
L9   }
```

表 1 执行过程中的语义距离计算

| 程序 | 参数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 缺陷程序$P$ | loc | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | — | — |
| | action | — | — | null | null | null | $O$_action | $O$_action | $O$_action | — | — |
| | mode | — | — | — | — | — | — | $O$_mode | $O$_mode | — | — |
| 候选补丁$P'$ | loc | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | action | — | — | null | null | null | null | null | $O$_action | $O$_action | $O$_action |
| | mode | — | — | — | — | — | — | — | — | $O$_mode | $O$_mode |
| 候选补丁$P''$ | loc | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | — | — |
| | action | — | — | $O$_action | $O$_action | $O$_action | $O$_action | $O$_action | $O$_action | — | — |
| | mode | — | — | — | — | — | — | $O$_mode | $O$_mode | — | — |
| $diff(P, P')$ | — | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | — | — |
| $diff(P, P'')$ | — | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | — | — |

## 4 实验

基于本文方法, 我们实现了一个缺陷自动修复工具 DTSFix, 图 3 为其界面展示. 为了验证该工具在空指针引用的有效性, 实验 1 在公开数据集 Defects4J[18] 中空指针引用上设计了相关实验, 与其他工具的修复效果进行对比. 实验结果表明 DTSFix 多修复 9 个缺陷, 平均节省人工修复时间 180 min. 实验 2 则通过在开源的数据库管理工具 DBeaver 的修复过程中成功修复了 67% 的空指针引用, 成功证明在 DTSFix 在实际工程上的修复能力.

### 4.1 与其他修复工具的对比

在实验 1 中, 将 DTSFix 与同样专门针对空指针引用的工具 NPEFix[19] 进行对比, 所采用数据集为 Defects4J 中空指针引用如表 2 所示, 其中 Chart 包含 7 个, Lang 包含 5 个, Math 包含 3 个. 通过表 3 中的实验结果中可以看出 NPEFix 仅能正确修复 2 个缺陷, 而 DTSFix 正确修复 11 个缺陷修复成功率为 73.3%, 共产生候选补丁 19 个, 正确补丁个数为 14 个, 准确率为 73.7%.

为了更好地获得正确补丁, 在 DTSFix 中加入了候选补丁的排序算法. 在表 3 中的 CAP 和 COP_F 分别为 DTSFix 中候选补丁的数量和候选补丁中第一个正确补丁的排名, 从 COP_F 中可以看出, 通过经过对候选补丁的进行语义距离排序后, 在已修复的 12 个缺陷

中, 其中 10 个正确补丁位于候选补丁的排名第一位. 在含有多个候选补丁的 5 个缺陷中, 4 个正确补丁经过排序后排名上升, 这表明基于语义距离的候选补丁排序能够较好地区分正确补丁与错误补丁, 从而有助于开发人员选择正确的补丁.

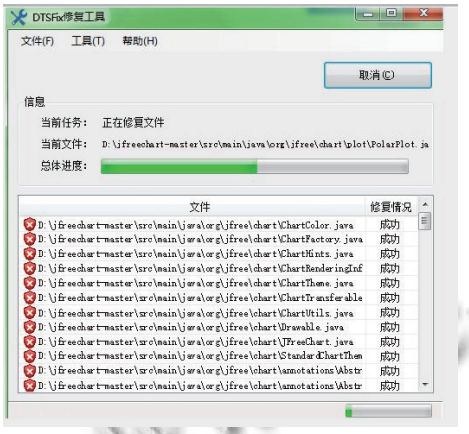


图 3 DTSFix 界面

表 2 Defects4J 中空指针引用

| Project | Defect ID | Defects |
|---|---|---|
| Chart | 2, 4, 14, 15, 16, 25, 26 | 7 |
| Lang | 20, 33, 39, 47, 57 | 5 |
| Math | 4, 70, 79 | 3 |
| Total | — | 15 |

### 4.2 在实际工程上验证 DTSFix 的修复能力

为了验证 DTSFix 对实际 Java 工程中空指针引用缺陷的修复能力, 我们选取免费开源的数据库管理工具 DBeaver 进行实验. DBeaver 可支持任何具有 JDBC 驱动程序数据库, 为开发人员和数据库管理人员提供支持. DTSJava[3] 作为检测工具对 DBeaver 进行缺陷检测, 其中含有 9 个空指针引用缺陷. 给定 COP 为正确补丁的数量, CAP 为候选补丁的数量, DEN 为缺陷数量. 从表 4 可以看出, DTSFix 成功修复了 6 个缺陷, 其中约束策略修复 4 个, 规避策略修复 2 个. 未修复的 3 个缺陷中其中 2 个缺陷并未成功获得候选补丁, 原因在于修复模板提出的不够充分, 需要进一步细化空指针引用缺陷的修复模板, 但 DTSFix 对 DBeaver 的修复结果仍然可以确定 DTSFix 对于实际应用软件具有非常有效的修复能力.

表 3 实验 1 的对比结果

| Project | Defect ID | NPEFix | DTSFix | CAP | COP_F |
|---|---|---|---|---|---|
| Chart | 2 | — | — | — | — |
| | 4 | × | √ | 1 | 1 |
| | 14 | × | √ | 1 | 1 |
| | 15 | × | √ | 2 | 1 |
| | 16 | × | — | — | — |
| | 25 | × | √ | 2 | 1 |
| | 26 | × | √ | 1 | 1 |
| Lang | 20 | × | — | — | — |
| | 33 | √ | √ | 1 | 1 |
| | 39 | — | √ | 1 | 1 |
| | 47 | — | √ | 3 | 2 |
| | 57 | √ | √ | 1 | 1 |
| Math | 4 | × | √ | 2 | 1 |
| | 70 | × | √ | 3 | 1 |
| | 79 | × | × | 1 | — |

表 4 实验 3 修复结果

| 文件 | 缺陷数 | 修复策略 (COP/CAP/DEN) | | | | 总数 (COP/CAP/DEN) |
|---|---|---|---|---|---|---|
| | | 赋值策略 | 约束策略 | 规避策略 | 转移策略 | |
| org.jkiss.dbeaver.ext.mysql | 1 | 0/0/0 | 1/1/1 | 0/0/0 | 0/0/0 | 1/1/1 |
| org.jkiss.dbeaver.registry | 1 | 0/0/0 | 0/0/0 | 1/3/1 | 0/0/0 | 1/3/1 |
| org.jkiss.dbeaver.ui.editors.hex | 2 | 0/0/0 | 1/2/1 | 0/0/1 | 0/0/0 | 1/2/2 |
| org.jkiss.dbeaver.erd.ui | 1 | 0/0/0 | 1/1/1 | 0/0/0 | 0/0/0 | 1/1/1 |
| org.jkiss.dbeaver.core | 3 | 0/0/0 | 1/2/2 | 1/2/1 | 0/0/0 | 2/4/3 |
| org.jkiss.dbeaver.data.office | 1 | 0/0/0 | 0/0/1 | 0/0/0 | 0/0/0 | 0/0/1 |
| 总数 | 9 | 0/0/0 | 4/6/6 | 2/5/3 | 0/0/0 | 6/11/9 |

代码 3 展示了其中一个空指针引用修复示例, 经过缺陷检测后缺陷开始行为 224 行, 缺陷发生行为 226 行, 缺陷变量为 *forTable*. 经过基于抽象语义的程序依赖图分析 *forTable* 的缺陷影响的代码集合为{225, 226, 227, 228}, 其中 228 为 return 语句. *forTable* 来源于外部参数通过赋值策略无法获取合适的值, 且缺陷感染域中包含 return 语句不满足约束策略的使用条件, 最后通过规避策略, 寻找方法中含有与返回值同类型同名变量的值, 增添修复语句如代码 4 进行修复, 并通过 DTSJava 进行回归测试结果证明缺陷已被成功

修复.

代码 3. DBeave 缺陷程序示例

```
…
223  protected JDBCStatement prepareChildrenStatement(@NotNull
JDBCSession session,
224  @NotNull OceanbaseMySQLCatalog owner, @Nullable
MySQLTableBase forTable) throws SQLException {
225          if (forTable instanceof OceanbaseMySQLView) {
226              JDBCPreparedStatement dbStat = session
227                  .prepareStatement("desc" + owner.getName()
+ "." + forTable.getName());
228              return dbStat; }
…
```

代码 4. DBeave 缺陷示例修复补丁

```
…
223  protected JDBCStatement prepareChildrenStatement(@NotNull
JDBCSession session,
224  @NotNull OceanbaseMySQLCatalog owner, @Nullable
MySQLTableBase forTable) throws SQLException {
+            if (forTable==null){
+                return (JDBCPreparedStatement)session.pre
pareStatement(sql.toString());
+            }
225          if (forTable instanceof OceanbaseMySQLView) {
226              JDBCPreparedStatement dbStat = session
227                  .prepareStatement("desc" + owner.getName()
+ "." + forTable.getName());
228              return dbStat; }
…
```

## 5 结论与展望

本文提出了一种基于抽象语义的程序依赖图的空指针引用的自动修复方法. 通过抽象属性对程序依赖图的剪枝构建一种基于抽象语义的程序依赖图, 在其掌握精确的控制依赖关系和基于抽象语义的数据依赖关系的基础上, 采用总结出来的赋值策略、约束策略、规避策略和转移策略 4 种策略完成修复. 最后利用基于语义距离的排序方法对候选补丁进行排序, 将高质量的候选补丁优先排列. 实验证明基于抽象语义的程序依赖图的空指针引用的自动修复方法针对空指针引用的修复能够保证在较好的准确率的情况下提供正确的修复补丁, 并且在实际工程中具有非常有效修复能力. 在未来的工作中, 将在基于抽象语义的程序依赖图的分析上进一步总结空指针引用的修复模板, 提升空指针引用的修复精度.

## 参考文献


1 Ye H, Martinez M, Durieux T, *et al*. A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software, 2021, 171: 110825. [doi: 10.1016/j.jss.2020.110825]

2 Klieber W, Martins R, Steele R, *et al*. Automated code repair to ensure spatial memory safety. Proceedings of the 2021 IEEE/ACM International Workshop on Automated Program Repair (APR). Madrid: IEEE, 2021. 23–30. [doi: 10.1109/APR52552.2021.00013]

3 Campos D, Restivo A, Ferreira HS, *et al*. Automatic program repair as semantic suggestions: An empirical study. Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). Porto de Galinhas: IEEE, 2021. 217–228. [doi: 10.1109/ICST49551.2021.00032]

4 李斌, 贺也平, 马恒太. 程序自动修复: 关键问题及技术. 软件学报, 2019, 30(2): 244–265. [doi: 10.13328/j.cnki.jos.005657]

5 Villanueva OM, Trujillo L, Hernandez DE. Novelty search for automatic bug repair. Proceedings of the 2020 Genetic and Evolutionary Computation Conference. Cancún: Association for Computing Machinery, 2020. 1021–1028. [doi: 10.1145/3377930.3389845]

6 White M, Tufano M, Martínez M, *et al*. Sorting and transforming program repair ingredients via deep learning code similarities. Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou: IEEE, 2019. 479–490.

7 Liu K, Koyuncu A, Kim D, *et al*. TBar: Revisiting template-based automated program repair. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing: Association for Computing Machinery, 2019. 31–42. [doi: 10.1145/3293882.3330577]

8 Afzal A, Motwani M, Stolee KT, *et al*. SOSRepair: Expressive semantic search for real-world program repair. IEEE Transactions on Software Engineering, 2021, 47(10): 2162–2181. [doi: 10.1109/TSE.2019.2944914]

9 Kanemitsu T, Higo Y, Kusumoto S. A visualization method of program dependency graph for identifying extract method opportunity. Proceedings of the 4th Workshop on Refactoring Tools. Waikiki: Association for Computing Machinery, 2011. 8–14. [doi: 10.1145/1984732.1984735]

10 Agarwal S, Agrawal AP. An empirical study of control dependency and data dependency for large software systems.

Proceedings of the 2014 5th International Conference—Confluence The Next Generation Information Technology Summit (Confluence). Noida: IEEE, 2014. 877–879. [doi: 10.1109/CONFLUENCE.2014.6949230]

11 Noda K, Yokoyama H, Kikuchi S. Sirius: Static program repair with dependence graph-based systematic edit patterns. Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). Luxembourg: IEEE, 2021. 437–447. [doi: 10.1109/ICSME52107.2021.00045]

12 Halder R, Cortesi A. Abstract program slicing on dependence condition graphs. Science of Computer Programming, 2013, 78(9): 1240–1263. [doi: 10.1016/j.scico.2012.05.007]

13 Sukumaran S, Sreenivas A, Metta R. The dependence condition graph: Precise conditions for dependence between program points. Computer Languages, Systems & Structures, 2010, 36(1): 96–121. [doi: 10.1016/j.cl.2009.04.001]

14 Dong YK, Wu M, Zhang L, *et al*. Priority measurement of patches for program repair based on semantic distance. Symmetry, 2020, 12(12): 2102. [doi: 10.3390/sym12122102]

15 D'Antoni L, Samanta R, Singh R. QLOSE: Program repair with quantitative objectives. Proceedings of the 28th International Conference on Computer Aided Verification. Toronto: Springer, 2016. 383–401. [doi: 10.1007/978-3-319-41540-6_21]

16 Liu K, Kim D, Bissyandé TF, *et al*. Mining fix patterns for FindBugs violations. IEEE Transactions on Software Engineering, 2021, 47(1): 165–188. [doi: 10.1109/TSE.2018.2884955]

17 Martinez M, Durieux T, Sommerard R, *et al*. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. Empirical Software Engineering, 2017, 22(4): 1936–1964. [doi: 10.1007/s10664-016-9470-4]

18 Cornu B, Durieux T, Seinturier L, *et al*. NPEFix: Automatic runtime repair of null pointer exceptions in Java. arXiv:1512.07423, 2015.

19 王淑栋, 尹文静, 董玉坤, 等. 面向顺序存储结构的数据流分析. 软件学报, 2020, 31(5): 1276–1293. [doi: 10.13328/j.cnki.jos.005949]

(校对责编: 孙君艳)