软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 基于程序合成的 C/C++程序缺陷自动修复方法

周风顺, 王林章, 李宣东

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

**摘 要**: 程序缺陷不可避免的存在于计算机软件中。目前的程序缺陷自动修复方法大都遵循缺陷定位、修复候选项生成、选择及验证的流程，但在修复实际程序时存在修复率低，无法保证修复结果的正确性等问题。本文提出了一种基于程序合成的 C/C++程序缺陷自动修复方法，首先，从满足相同规约的程序集中，通过人工整理的方式总结错误模式及其对应的修复方法，使用重写规则表达错误模式，在此基础上实现了基于重写规则和基于程序频谱的缺陷定位方法，得到程序中可能的缺陷位置；其次，基于重写规则使用修复选项生成方法得到缺陷的修复选项，同时通过深度学习的方式学习正确程序的书写结构，帮助预测错误程序错误点的应有的语句结构，通过这两种方式提高候选项质量进而提高修复率；最后，在选择验证过程中我们使用程序合成的方法，将样例程序作为约束，保证合成后代码的正确性。基于上述方法实现了原型工具 Autograder，并在容易出错、缺陷典型的学生作业程序上进行了实验，结果显示，本文方法对学生作业程序中的缺陷有着较高的修复率，同时也能保证修复后代码的正确性。

**关键词**: 程序修复;程序合成;深度学习

## Automatic Program Repair for C/C++ Code Based on Program Synthesis

ZHOU Feng-Shun, WANG Lin-Zhang, LI Xuan-Dong

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

**Abstract**: Program defects are inevitably present in computer software. Most of the current automatic program repair methods follow the process of defect location, candidate generation, candidate verification. However, when the program is repaired, there is a problem that the repair rate is low and the repair result cannot be guaranteed. This paper proposes a method for automatic repair of defects in C/C++ program based on program synthesis. Firstly, we summarize the error mode and its corresponding repair methods from the assembly that satisfies the same specification, and use the rewrite rules to express the error mode and its corresponding repair methods. On this basis, we implement a defect-location method based on rewrite rules and program spectrum to obtain possible defect locations in the program. Secondly, we use the candidate-generation method to get the repair candidate based on the rewrite rule. At the same time, we learn the correct structure of the program through deep learning to help predict the correct sentence structure of the wrong program error point. These two ways improve the quality of the candidate and the repair rate. Finally, in the candidate-verification process, we use the method of program synthesis. The sample program is used as a constraint to ensure the correctness of the synthesized code. Based on the above methods, we implemented the prototype tool Autograder and experimented on student program. The experimental results show that our method has a high repair rate for the defects in the student program, and also ensures the correctness of the code after the repair .

**Key words**: Program Repair; Program Synthesis; Deep Learning

## 1 简介

随着信息技术的不断发展，计算机软件已经成为现代社会最重要的基础设施之一。但是，由于程序员的疏忽和软件系统复杂度的增加，这些程序中不可避免的存在缺陷。尽早的发现并排除这些潜在的缺陷是学术界和工业界普遍的共识。程序缺陷自动修复技术可以根据给定的错误程序，自动生成程序修复候选项，进而修复程序中的缺陷。其修复过程中产生的补丁可以直接用于修改程序，也可以帮助开发人员改进代码。因此，程序缺陷自动修复成为我们关注的重点。

程序缺陷自动修复的一般流程包括缺陷定位、候选项生成、验证及选择。程序修复方法首先通过缺陷定位方法将给定带有缺陷的程序的所有语句按某种规则排序，得到可疑语句的排列。然后将这些语句作为疑似缺陷，逐个传输给修复候选项生成算法。修复候选项生成算法会检测当前缺陷的位置，决定是否能够输出候选项，如果能够，则输出候选项给测试集进行验证，通过验证的候选项将作为结果输出。若该疑似缺陷的位置无法输出修复候选项,则从语句排列中获取下一个可疑语句,重新运行修复候选项生成算法以继续寻找潜在的修复候选项。

修复真实缺陷一直以来是程序缺陷自动修复研究的目标。目前学术界对现有的修复技术在真实缺陷上的修复能力存在一定争议[6-7]，关于缺陷修复的讨论也有很多。2015 年，Qi 等人[8]对 GenProg 和 AE 两个工作在 105 个真实缺陷上的修复进行了手工检查。其中 Genprog 的修复结果中只有两个是语义正确的，而 AE 的修复结果中只有三个是语义正确的。由此可见，现有的修复技术存在着修复率低，无法保证修复结果的正确性等问题。

而近年来，程序合成技术和机器学习、深度学习等 AI 技术不断进步，在软件工程中的有着广泛的应用。这两项技术的发展在解决程序自动修复问题上引起了我们新的思考。程序合成[1-2]是指从程序语言自动的构造程序，使其满足特定的规约。程序合成器通常是在程序空间内进行搜索以生成满足各种约束的程序。程序合成通常被使用在代码补全、软件开发、算法设计中[3]。近年来，程序合成除了规约外，还允许用户额外地提供可能的程序空间框架，一般是语法层面的。这样的做法有两点好处。首先，语法为假设空间提供了结构，这可以使得搜索的过程更加高效。其次，生成的程序也是可解释的，因为他们是从语法中衍生出来的。程序合成工具 SKETCH[4]首先提出了这个想法，它允许程序员编写部分程序框架（带有空白的程序），然后根据规约自动填补这些空白。深度学习中长短期记忆网络（LSTM）[5]是一种特定形式的循环神经网络（RNN）。LSTM 的关键是细胞状态， 细胞的状态类似于输送带，细胞的状态在整个链上运行，只有一些小的线性操作作用其上，信息很容易保持不变的流过整个链。同时 LSTM 有一种精心设计的称为门的结构，可以用来删除或添加信息到细胞，它是一种让信息选择式通过的方法。LSTM 的这些特性使得 LSTM 广泛用于序列预测问题。

本文针对现有的修复技术存在的问题，结合程序合成技术和深度学习技术，提出了一种基于程序合成的代码缺陷自动修复方法。一方面人工整理常见的缺陷形成缺陷模式并总结修复方式，帮助修复常见的缺陷。另一方面通过深度学习的方法，从满足相同规约的正确程序集中学习正确程序的书写结构，帮助预测错误程序错误点的应有的语句结构。在选择验证的过程中我们使用程序合成的方法，将样例程序作为约束，保证合成后代码的正确性。本文的主要工作在于：

(1)本文提出了一种基于程序合成的 C/C++程序缺陷自动修复方法。总结了 C/C++程序中常见的错误模式，并建立了书写结构模型。使用缺陷定位方法：基于重写规则的定位方式和基于程序频谱的方式，得到程序中可能的缺陷位置。使用修复选项生成方法：基于重写规则的修复选项生成方法和基于预测的修复选项生成方法，得到缺陷的修复选项。

(2)提出了基于程序合成工具 SKETCH[4]的修复选项选择与确认方法。将带有修复选项的 C/C++程序转化为 SKETCH 代码，并进行预处理，将该程序的示例程序作为程序的规约并加入 SKETCH 代码。使用 SKETCH 执行程序合成，直到找到满足规约的修复，得到正确的修复结果。

(3)基于上述方法，本文实现了一个 C/C++程序缺陷修复的原型工具。

本文第 2 节主要介绍基于程序合成的代码缺陷修复方法的基本框架。第 3 节展示实验结果。第 4 节主要介绍相关工作，包括基于搜索的缺陷修复和基于约束求解的缺陷修复。第 5 节进行总结和展望。

## 2 基于程序合成的 C/C++程序缺陷自动修复方法

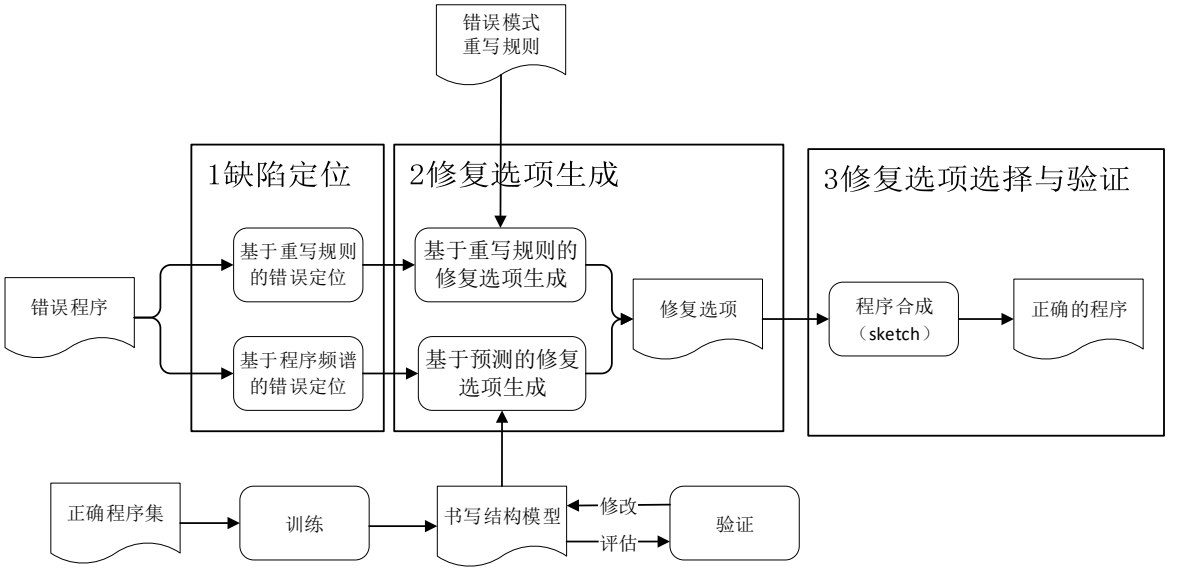


图 2-1　方法框架

本节我们提出基于程序合成的 C/C++程序缺陷自动修复方法的基本框架。程序自动修复的流程包括缺陷定位，候选项生成，候选项验证。本方法使用缺陷定位方法：基于重写规则的定位方法和基于程序频谱的定位方法，得到程序中可能的缺陷位置；使用修复选项生成方法：基于重写规则的修复选项生成方法和基于预测的修复选项生成方法，得到缺陷的修复选项。同时在修复选项验证步骤中使用程序合成的方法，保证合成后程序的正确性。该方法分为三个主要步骤：缺陷定位、修复选项生成、修复选项验证和一个准备步骤——重写规则总结和书写结构模型构建。

(1) 重写规则总结和书写结构模型构建。对于重写规则，我们对错误程序进行整理，总结其中常见的错误模式，并人工给出修复方法，将错误模式和修复方法配对形成重写规则。对于书写结构模型，我们收集满足相同规约的正确程序，提取其中的书写结构并序列化。将所得的序列化数据使用长短期记忆模型进行训练。使用梯度下降算法，当定义的损失函数在验证集上的变动小于设定的阈值时停止训练。将此时的模型用于书写结构预测。

(2) 缺陷定位。我们使用基于重写规则和基于程序频谱两种方式进行缺陷定位。在基于重写规则的定位中，我们将错误模式与错误程序的抽象语法树进行匹配，得到可能的缺陷节点。在基于程序频谱的定位中，我们在错误程序上执行测试用例，根据语句覆盖信息，计算程序中语句缺陷疑似度，按疑似度从高到低进行排序，将疑似度高的语句作为缺陷位置。

(3) 修复选项生成。对于使用重写规则确定的缺陷位置，我们直接使用重写规则中的修复方法生成修复选项。对于使用程序频谱确定的缺陷位置，我们将缺陷处前面的序列化数据作为书写结构模型的输入，预测缺陷处的书写结构，使用当前位置可见的变量进行扩展，得到多个修复候选项。

(4) 修复选项验证。首先将错误程序、缺陷定位信息以及修复选项结合，得到扩展的带有选择表达式的 C/C++程序，并转化为相应的 SKETCH 程序。然后将程序需要满足的规约作为约束，求解每一处选择表达式的选项，确认修复候选项。

### 2.1 重写规则总结和书写结构模型构建

#### 2.1.1 重写规则

每一条重写规则由两部分组成，一部分表示缺陷模式，另一部分表示修复选项。我们对学生作业程序中常见的错误进行整理，总结了一些常见的错误模式及修复选项，具体如下

表 2-1 重写规则

| 序号 | 缺陷类型 | 修复方法 |
|---|---|---|
| 1 | 循环上界错误 | $\text{for}(e_1;e_2 op_{compare} e_3;e_4)\text{s} \rightarrow \text{for}(e_1;e_2 op_{compare} e_3 \pm 1;e_4)\text{s}$ |
| 2 | 浮点数直接比较缺陷 | $fe_1 == fe_2 \rightarrow abs(fe_1 - fe_2) < min$<br>$fe_1 != fe_2 \rightarrow abs(fe_1 - fe_2) > min$ |
| 3 | 整数相除缺陷 | $v_{float} = ie_1/ie_2 \rightarrow v_{float} = (float)ie_1/(float)ie_2$ |
| 4 | 条件判断中的赋值错误 | $\text{if(v = e)s} \rightarrow \text{if(v == e)s}$ |
| 5 | 位运算与逻辑运算误用 | $ie_1 \& ie_2 \rightarrow ie_1 \&\& ie_2$<br>$ie_1 \&\& ie_2 \rightarrow ie_1 \& ie_2$<br>$ie_1 \mid ie_2 \rightarrow ie_1 \mid\mid ie_2$<br>$ie_1 \mid\mid ie_2 \rightarrow ie_1 \mid ie_2$<br>$\sim ie_1 \rightarrow !ie_1$<br>$!ie_1 \rightarrow \sim ie_1$ |
| 6 | 与或运算误用 | $ie_1 \&\& ie_2 \rightarrow ie_1 \mid\mid ie_2$<br>$ie_1 \mid\mid ie_2 \rightarrow ie_1 \&\& ie_2$ |
| 7 | 模运算与除运算误用 | $ie_1/ie_2 \rightarrow ie_1 \% ie_2$<br>$ie_1 \% ie_2 \rightarrow ie_1/ie_2$ |
| 8 | 自增、自减运算误用 | $ie_1 op_u \rightarrow op_u ie_1$<br>$op_u ie_1 \rightarrow ie_1 op_u$ |

本节给出了目前已整理的 C/C++代码中常见的重写规则，这些重写规则比较通用，其中主要的缺陷是运算符误用和代码逻辑的错误。在一些特定的程序中。可能会出现某些特殊的缺陷，如果该缺陷存在特殊的模式，同时也有比较固定的修复方法，那么我们也可以针对这些缺陷，设计特定的重写规则，以实现对这些缺陷的修复。

#### 2.1.2 序列化

代码数据不同于普通的文本数据，它具有复杂的结构，若直接将代码视为序列化数据，便会失去其中的结构信息，长短期记忆网络也难以从这种缺乏关键信息的数据中学习书写结构。而抽象语法树既可以用来表示程序语义，同时也可以表示程序结构，因此我们采用抽象语法树对程序进行解析。并且我们在深度优先遍历抽象语法树的基础上，用“{”和“}”将每个中间节点包围起来，以此来加强序列化数据中的结构信息。

```
if (nums[i] > max) {
    max = nums[i];
}
```

图 2-2 代码片段

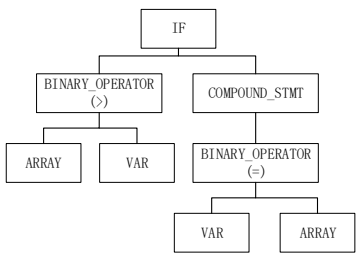


图 2-3 语法树结构

如图 2-2 一段简单的代码，其语法树形式为如图 2-3 所示，其序列化数据如图 2-4 所示。

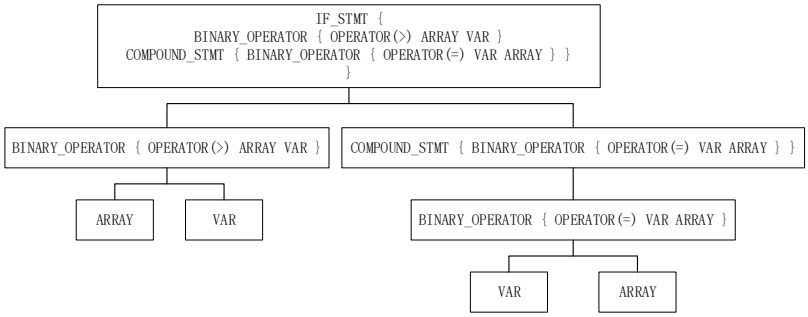


图 2-4 序列化数据

2.1.3 书写结构模型设计

长短期记忆网络广泛应用于序列预测，我们也使用 LSTM 来预测缺陷处的书写结构。我们使用的 LSTM 模型结构如图 2-5 所示：

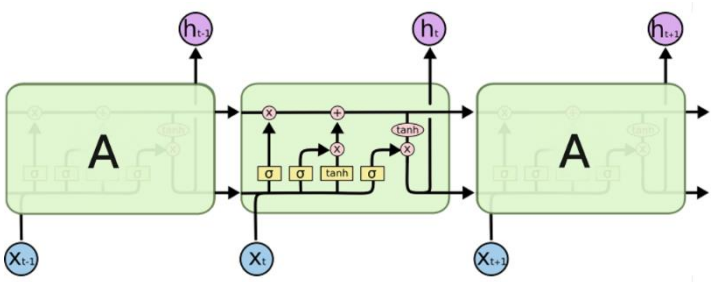


图 2-5 LSTM 模型

该模型的输入为 k 个 token。通过 one-hot 编码将 token 转化为特征向量，并输入到隐藏层的 LSTM cell。隐藏层的输入还包括上一时刻细胞状态$s_{t-1}$以及输出结果$o_{t-1}$。该层的输出为当前时刻细胞状态$h_t$以及输出结果$o_t$。将隐藏层的输出 o 输入模型的 softmax 层，得到预测结果的概率分布，向量中第 i 个元素的值代表结果为词汇表中第 i 个 token 的概率。

隐藏层中细胞状态$s_t$以及输出结果$o_t$计算公式如下：

$$\begin{bmatrix} fg \\ u \\ ig \\ og \end{bmatrix} = \begin{bmatrix} \sigma \\ tanh \\ \sigma \\ \sigma \end{bmatrix} P_{J,2J} \begin{bmatrix} x_t \\ o_{t-1} \end{bmatrix}$$

$$s_t = s_{t-1} * fg + u * ig$$

$$o_t = og * \tanh(s_t)$$

其中$x_t$为当前 token 的 one-hot 编码，$s_{t-1}$为上一时刻细胞状态，$o_{t-1}$为上一时刻输出结果。

损失函数定义为 softmax 的输出向量和样本实际标签交叉熵的平均值。

模型训练时，采用梯度下降优化算法，当损失函数的变化率小于设定值时停止训练。

## 2.2 缺陷定位

缺陷定位是程序修复的重要一步。本文使用了两种缺陷定位方法：基于重写规则的缺陷定位方法与基于程序频谱的缺陷定位方法。

2.2.1 基于重写规则的缺陷定位

在 2.1 中我们设计了重写规则，并给出了若干条重写规则实例。重写规则包括了缺陷模式和修复选项，其中缺陷模式描述了程序中可能的缺陷。我们可以将重写规则中的缺陷模式与源程序在语法树上进行模式匹配，得到程序中可能的缺陷位置。

**算法 1**. 基于重写规则的缺陷定位算法
输入:重写规则 Rule，源程序的抽象语法树 T
输出：可能的缺陷位置：抽象语法树上的节点集合 S

```
Begin
 1: Set S = ∅
 2: Queue Q = ∅
 3: parse Rule as DT → CTS
 4: ENQUEUE(Q, T.root)
 5: ndt = DT.root
 6: while Q is not EMPTY do
 7:         n = DEQUEUE(Q)
 8:         if nodeCompare(n, ndt) then
 9:             insert n to S
10:         else
11:             for each children in n as cn do
12:                 ENQUEUE(Q, cn)
13: return S
End
```

该算法的目的是将缺陷模式与抽象语法树进行匹配。算法的 6-12 行的广度优先遍历是主体部分。算法的第 4 行，将源程序抽象语法树的根节点加入队列 Q。第 7 行从队列 Q 中取出一个节点，将该节点与缺陷模式 DT 的根节点进行匹配。若完全匹配，那么该节点为可能的缺陷节点，算法第 9 行，将这个节点加入集合 S。若不匹配，算法 11-12 行，将该节点的所有子节点加入队列 Q。之后按照广度优先遍历的顺序，依次与缺陷模式进行比较。

2.2.2 基于程序频谱的缺陷定位

使用重写规则我们可以定位出程序中那些具有固定模式的缺陷，但对于程序中更加一般的缺陷，比如表达式计算错误，我们无法给出明确的重写规则。对于这样更加一般的缺陷，我们将采用基于程序频谱的定位方法。程序频谱从某些角度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程序路径。它可以用来跟踪程序的行为。当执行失败时，可以通过这些信息来识别导致缺陷的可疑代码。选用程序频谱的定位方式，虽然并不一定精确，但可以快速确定哪些语句与缺陷相关，缩小对缺陷语句的搜索范围，后期对修复候选项的选择与验证也能弥补这种定位带来的不精确。

对于一个错误程序，我们首先获得一组测试用例，其中包含通过的测试用例和失败的测试用例。这两类测试用例对程序语句的覆盖情况不同，失败的测试用例覆盖的语句更有可能存在缺陷。利用这两类测试用例的覆盖情况，使用特定的公式，我们就能计算出不同语句存在缺陷的疑似度。按照疑似度对语句进行排序，并依次尝试进行修复。本文计算疑似度使用的是 Tarantula 计算公式，具体如下：

$$\text{Suspiciousness} = \frac{N_{CF}/N_F}{N_{CF}/N_F + N_{CS}/N_S}$$

其中，$N_{CF}$表示覆盖语句的失败的测试用例数量，$N_F$表示所有失败的测试用例数量。$N_{CS}$表示覆盖语句的通过的测试用例数量，$N_S$表示所有通过的测试用例数量。

表 2-2 含有一个简单程序，其目的是求三个整型变量的中值。其中语句 6 存在缺陷，正确应为 m=x。表右侧是 6 个测试用例，其中有 5 个通过的测试用例，1 个失败的测试用例。表中圆点表示该测试用例覆盖的语句。根据语句覆盖情况，使用上述的公式，得到表 2-3 语句疑似度计算结果。从表 2-3 中看出，语句 6 存在缺陷的疑似度最高，同时语句 6 也正是存在缺陷的语句。

表 2-2 程序频谱示例

| | Source code | Test case | | | | | |
|---|---|---|---|---|---|---|---|
| | int mid(int x,int y,int z){ | 3,3,5 | 1,2,3 | 3,2,1 | 5,5,5, | 5,3,4 | 2,1,3 |
| 1 | m=z; | ● | ● | ● | ● | ● | ● |
| 2 | if(y<z) | ● | ● | ● | ● | ● | ● |
| 3 | if(x<y) | ● | ● | | | ● | ● |
| 4 | m=y; | | ● | | | | |
| 5 | else if(x<z) | ● | | | ● | | ● |
| 6 | m=y; //bug | ● | | | | | ● |
| 7 | else | | | ● | ● | | |
| 8 | if(x>y) | | | ● | ● | | |
| 9 | m=y; | | | ● | | | |
| 10 | else if(x>z) | | | | ● | | |
| 11 | m=x; | | | | | | |
| 12 | return m; | ● | ● | ● | ● | ● | ● |
| | } | pass | pass | pass | pass | pass | fail |

表 2-3 语句疑似度计算结果

| 语句序号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 疑似度 | 0.5 | 0.5 | 0.63 | 0 | 0.71 | 0.83 | 0 | 0 | 0 | 0 | 0 | 0.5 |
| 排名 | 4 | 4 | 3 | 7 | 2 | 1 | 7 | 7 | 7 | 7 | 7 | 4 |

### 2.2.3 两种定位方式对比

重写规则是我们在调研南京大学《程序设计》课程中连续 2 年 11 题课程作业、316 个学生作业提交后，人工总结常见的错误类型（详见表 2-1）及针对该类错误的修复方式形成的，基于重写规则的缺陷定位方式主要也是定位这些人工总结出来的具有固定模式的缺陷。但人工总结难免有所遗漏，基于程序频谱的缺陷定位针对的就是重写规则遗漏的部分，与基于重写规则的缺陷定位方式互为补充。

## 2.3 修复选项生成

在 2.2.1 节中我们介绍了基于重写规则的缺陷定位方法和基于程序频谱的缺陷定位方法。在得到可能的缺陷位置后，我们将分别使用两种修复选项生成方法：基于重写规则的修复选项生成和基于预测的修复选项生成，对缺陷位置生成修复选项。

### 2.3.1 基于重写规则的修复选项生成

在 2.2.1 中我们通过重写规则得到了可能的缺陷节点。重写规则中包含了修复选项，我们可以利用重写规则，对由重写规则得到的缺陷节点，生成实际的修复选项。

该算法在第一行匹配标识符（变量名）与语法树节点的对应关系，保存在字典 NDict 中。之后，在 6-12 行按照广度优先的方式，遍历每一个修复选项 ct 的所有节点，在第 9 行将其中的标识符节点替换。替换掉标

识符节点的 ct 就是针对缺陷节点 node 的实际修复选项，最后在第 13 行将 ct 加入集合 S 中。

**算法 2**. 基于重写规则的修复选项生成算法
输入:重写规则 Rule，抽象语法树上的缺陷节点 node
输出:针对缺陷节点 node 的修复选项集合 S

```
Begin
 1:                                   Dict < ID, Node > NDict =
identifierNodeMatch(Rule, node)
 2: parse Rule as DT → CTS
 3: for each candidate ct in CTS do
 4:         Queue Q = ∅
 5:         ENQUEUE(Q, ct.root)
 6:         while Q is not EMPTY do
 7:             n = DEQUEUE(Q)
 8:             if n.type == identifier then
 9:                 n ← NDict[n]
10:             else
11:                 for each children cn in n do
12:                     ENQUEUE(cn, Q)
13:         add ct to S
14: return S
End
```

2.3.2 基于预测的修复选项生成

在 2.2.2 中，我们介绍了基于程序频谱的缺陷定位，通过该方法定位到缺陷位置后，我们通过预测的方法来生成修复候选项。在深度学习中，长短期记忆网络能解决长期依赖问题，适合处理序列预测问题，因此我们选用长短期记忆网络作为是写结构模型。针对每一处缺陷位置，我们按照之前序列化的方法，将缺陷位置前的正确代码序列化，并截取固定长度作为书写结构模型的输入。反复迭代产生预测直到匹配到完整的一对“{”“}”。将模型产生的预测结果作为该处缺陷可能的书写结构。

由于在序列化的时候我们对变量名进行了抽象，在实际生成候选项时，我们根据书写结构，使用当前可见的变量名替换结构中的对应部分，产生候选项集。

### 2.4 修复选项验证

传统的验证方法多使用测试的方法，通过所有的测试用例就认为修复正确，但这样并不能保证真正的正确。2015 年，Qi 等人[8]对 GenProg 和 AE 两个工作在 105 个真实缺陷上的修复进行了手工检查。其中 Genprog 的修复结果中只有两个是语义正确的，而 AE 的修复结果中只有三个是语义正确的。我们采用程序合成的方法，并使用程序合成工具 SKETCH 来进行候选项验证，确保合成后程序的正确性。我们将错误程序、缺陷定位信息以及修复选项结合，得到扩展的带有选择表达式的 C/C++程序，并转化为相应的 SKETCH 程序。然后将正确的样例程序作为 SKETCH 程序合成器的约束，在规定时间内求解 SKETCH 程序。如果在设定的时间内能够求解得到每一处选择表达式的选项，就以此确定 C/C++程序中对应的选择表达式选项，得到正确的 C/C++程序。如果超出规定时间或无解则认为没有正确的修复选项，修复失败。

# 3 工具与评估

本文实现了一个基于程序合成工具 SKETCH 的 C/C++程序缺陷自动修复原型工具 AutoGrader，该工具可以修复 C/C++程序上的常见缺陷。为了验证工具在缺陷修复上的有效性，本文以学生作业程序为对象，设计了相关实验，并对比了 GenProg 和 AE 两个工具在缺陷修复上的效果。

## 3.1 工具框架

图 3-1 展示了我们工具的框架，其中底层的是工具的输入，包括待修复的 C/C++程序、重写规则和正确程序集，中间部分是工具主要实现的模块，包括解析模块、模型构建模块、缺陷定位模块、修复选项生成模块、SKETCH 程序合成模块，最上层是工具的输出。

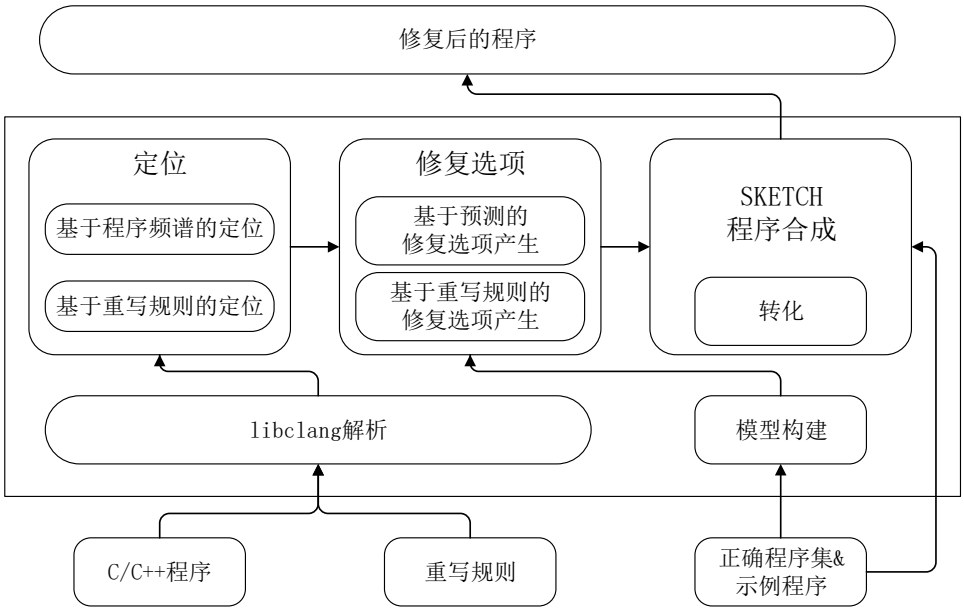


图 3-1 具体实现的框架图

## 3.2 实验对象

SKETCH 合成工具仅能处理小规模的问题，一方面由于大规模程序过于复杂，SKETCH 求解器的能力不足，另一方面由于大规模程序往往包含过多的函数库，使用 SKETCH 手工实现过于复杂。因此，我们的实验对象主要是学生作业程序。学生作业程序都是小规模程序，同时可能会包含一些相似的错误。

我们收集了 2016-2017 年间，南京大学《程序设计》课程的作业，累积 11 个任务、316 个学生提交。经过分析整理我们选取了两个任务的学生作业。任务 1 是求一个整型数组在除去最大值和最小值之后的平均值，任务 2 是对一个正整数进行质因子分解。除去编译发生错误的情况，共有 19 题错误作业，这些错误程序规模都在 20-30 行左右，包含 1-3 个错误。错误类型分布如表 3-1。

表 3-1 错误类型分布

| 错误类型 | 数量 |
|---|---|
| 循环上界错误 | 6 |
| 浮点数直接比较缺陷 | 5 |
| 整数相除缺陷 | 3 |
| 条件判断中的赋值错误 | 3 |
| 位运算与逻辑运算误用 | 2 |
| 与或运算误用 | 1 |

| | |
|---|---|
| 模运算与除运算误用 | 1 |
| 自增、自减运算误用 | 2 |
| 语句类型错误 | 7 |

### 3.3 实验设计

基于本文实现的 C/C++程序缺陷自动修复工具 AutoGrader，选取了学生的作业程序进行实验。作业程序有着明确的实验要求，我们提前给出了示例程序，使用示例程序作为 SKETCH 合成的规约。在实验中我们主要讨论以下几个问题：

**（1）AutoGrader 在作业程序上的修复效果如何？**

**（2）AutoGrader 对比其他工具有哪些优势？**

**（3）AutoGrader 所采用的缺陷定位方法能否定位到实际的缺陷位置？**

实验的运行环境如下：CPU Intel Core i7-4790GHz，内存 16GB，系统 Ubuntu16.04

#### 3.3.1 修复效果

表 3-2 展示了我们的实验结果。表中 Question 表示程序的问题，Program 表示程序编号，LOC 表示程序的代码行数，Defect 表示程序的缺陷数量，其余列表示对应方法修复缺陷的时间(以秒为单位)，X 表示未能完成修复。我们使用了 GenProg 和 AE 这两个工具进行了对比实验。

表 3-2 修复结果

| Question | Program | LOC | Defect | AutoGrader | GenProg | AE |
|---|---|---|---|---|---|---|
| Q1 | 001 | 20 | 3 | 39.8 | X | X |
| Q1 | 012 | 22 | 2 | 34.9 | X | X |
| Q1 | 014 | 19 | 1 | 19.6 | 19.2 | 14.6 |
| Q1 | 015 | 33 | 2 | X | X | X |
| Q1 | 019 | 24 | 1 | 18.4 | 18.5 | 15.1 |
| Q1 | 020 | 22 | 2 | 35.2 | X | X |
| Q1 | 021 | 21 | 1 | 36.5 | 16.3* | 14.2 |
| Q1 | 042 | 20 | 2 | 28.8 | X | X |
| Q1 | 045 | 19 | 1 | 19.9 | 19.7 | 12.6 |
| Q1 | 053 | 25 | 3 | X | X | X |
| Q1 | 057 | 22 | 1 | 19.5 | 18.1* | 13.6* |
| Q1 | 058 | 23 | 2 | X | X | X |
| Q2 | 006 | 34 | 1 | 45.6 | X | X |
| Q2 | 009 | 39 | 2 | X | X | X |
| Q2 | 017 | 31 | 1 | 25.1 | 22.8 | 17.8 |
| Q2 | 029 | 38 | 1 | 44.7 | X | X |
| Q2 | 034 | 37 | 1 | 44.9 | 26.1* | 20.6* |
| Q2 | 047 | 31 | 1 | 31.3 | 24.4 | 19.2 |
| Q2 | 065 | 32 | 2 | X | X | X |

通过实验数据表明，我们的工具对作业程序的修复有如下优点：

（1）对这 19 个错误程序，我们的工具可以对其中的 14 个进行修复，修复率为 73.7%，修复时间都在 60s 以下。GenProg 和 AE 仅能对其中的 8 个程序进行修复，尽管修复所需的时间更短，但我们的工具具有更高的修复率。

（2）GenProg 和 AE 仅能修复只含有一个缺陷的程序，我们的工具可以修复多个缺陷。

（3）由于我们在 SKETCH 合成中引入了示例程序，这保证了我们的工具修复后的程序是正确的。而对于 GenProg 和 AE，修复的程序仅能通过测试用例。表格中带*的数据表明，修复后的程序通过了测试用例，但不是一个正确的修复。

3.3.2 定位与修复分析

我们的工具使用了基于重写规则和基于程序频谱的缺陷定位技术，为了验证缺陷定位技术是否能够定位到程序中实际的缺陷，我们人工对比了程序中实际的缺陷与工具定位的可能的缺陷位置。

表 3-3 缺陷定位结果

| Question | Program | LOC | Real Defect | Defect Location | Repaired |
|---|---|---|---|---|---|
| Q1 | 001 | 20 | 5,8,10 | 5,8,10,16 | √ |
| Q1 | 012 | 22 | 6,19 | 5,6,7,19 | √ |
| Q1 | 014 | 19 | 18 | 11,18 | √ |
| Q1 | 015 | 33 | 16,31 | 7,8,10,31 | X |
| Q1 | 019 | 24 | 18 | 10,18 | √ |
| Q1 | 020 | 22 | 9,17 | 9,14,17,20 | √ |
| Q1 | 021 | 21 | 19 | 8,12,14,19 | √ |
| Q1 | 042 | 20 | 14,16 | 8,14,16 | √ |
| Q1 | 045 | 19 | 6 | 5,6,9 | √ |
| Q1 | 053 | 25 | 6,7,18 | 4,5,6,7,18 | X |
| Q1 | 057 | 22 | 7 | 7,10 | √ |
| Q1 | 058 | 23 | 5,7 | 7,10 | X |
| Q2 | 006 | 34 | 17 | 6,17,22,23 | √ |
| Q2 | 009 | 39 | 23,27 | 5,7,22 | X |
| Q2 | 017 | 31 | 18 | 4,18 | √ |
| Q2 | 029 | 38 | 22 | 18,22,31 | √ |
| Q2 | 034 | 37 | 19 | 7,19,30,36 | √ |
| Q2 | 047 | 31 | 26 | 18,26 | √ |
| Q2 | 065 | 32 | 12,19 | 12,23,30 | X |

表 3-3 展示了程序中实际的缺陷位置与定位技术得到的缺陷位置。Real Defect 表示程序的实际缺陷行号，Defect Location 表示定位得到的缺陷行号。从表中看到，程序中实际缺陷一共 30 个，定位技术得到的缺陷个数为 59 个。所有实现修复的程序中，定位技术都覆盖了实际的缺陷。对于未实现修复的程序，除 Q1 053 外，由于定位得到的缺陷未覆盖实际缺陷，因此无法实现修复。对于程序 Q1 053，定位覆盖了实际缺陷，而由于错误比较特殊，未产生正确的修复选项，因此无法修复。

# 4 相关工作

## 4.1 基于搜索的缺陷修复

在基于搜索的方法中，生成程序补丁的过程被看作是从搜索空间中寻找最优解的过程。对于待修复的程序来说该方法将对程序的修改作为潜在补丁，获得补丁空间。在搜索过程方面，多使用启发式方法、演化算法等方法寻找补丁。

GenProg 方法[9]由 Weimer 和 Le Goues 于 2009 年提出。GenProg 将程序看作抽象语法树，将代码段看作子树。GenProg 的核心算法是遗传算法，其中变异操作包括：复制别处语句替换当前语句，在当前语句后插入

其他语句，删除当前位置的语句。遗传操作为选择两个适应度高的程序，交换其中的变异操作。如果修改后的程序通过的测试用例越多，程序的适应度就越高。2012 年，Le Goues 等对 Genprog 方法进行了大型实验[17]，选择了 105 个大型程序中的缺陷进行修复，成功修复了 105 个缺陷中的 55 个。GenProg 是缺陷修复技术兴起的代表性工作。

2013 年，Weimer 等人对 GenProg 方法进行改进，提出了 AE 方法[10]。他们认为 GenProg 方法生成的候选补丁数量太多。为了尽可能的减少生成的候选补丁数量，AE 方法提出了等价类的概念。通过设置一系列规则，快速检查特定类型的等价变换，在修改时，避免产生等价变换。验证结果表明，修复的时间开销为原来的三分之一。

2013 年，Kim 等人提出了 PAR 方法[11]。该方法从开源项目 Eclipse JDT 中分析了 6 万个开发者人工修复的补丁，总结了常见的 10 种代码修复模板，并将模板与 GenProg 相融合生成补丁。PAR 方法生成的补丁具有更好的可读性。

2014 年，Qi 等人认为 GenProg 方法中的遗传算法存在计算开销大，难以识别有效补丁的问题，并提出了 RSRepair 方法[12]，将遗传算法替换为随机搜索。实验表明，RSRepair 和 GenProg 有着相近的修复能力，并能有效减少生成补丁的时间。

2016 年，Fan 提出了 Prophet 方法[13]。通过使用机器学习方法，对开源项目中正确代码的特征进行学习。在实际修复过程中，Prophet 方法对可能的修复补丁，分析补丁特征和代码上下文，通过概率模型，计算补丁可能修复程序的概率。最后根据概率排序，对补丁依次进行验证。

### 4.2 基于约束求解的缺陷修复

基于约束求解的缺陷修复，将在搜索空间寻找补丁的过程转换为约束求解问题，通过约束求解器获得可行解并转换为修复补丁。由于约束求解的特性,这类修复方法往往能够获得精确的结果，但也会消耗更多的时间。

该类算法的先驱是 2012 年的 SemFix 方法[14]，一种 C 语言的基于约束的修复算法。该算法使用基于组合的程序合成方法生成补丁。将测试用例转换为约束，使用 SMT 求解器求解，得到最终补丁。

2014 年的 Nopol 方法[15]是一种专门修复 if 条件缺陷的方法，针对条件错误或者条件语句缺失这两种常见缺陷进行修复。在实际修复过程中，使用天使定位算法，获得可能的缺陷位置。与 SemFix 方法不同，Nopol 仅针对 if 条件表达式，搜索空间小，可应用于大规模程序。

2015 年，Mechtaev 等人提出了 DirectFix 方法[16]，该方法为了提高 SemFix 方法生成的补丁质量，用语法树上被改动的元素个数定义差别，最终生成语法树上差别最小的补丁。

### 4.3 其他方法

2010年起，Pei等人提出一种基于契约的方法Autofix[18-21]。Autofix是一种基于契约的修复方法。该方法需要程序契约中的规约信息作为输入，例如函数的前置和后置条件等。通过从程序中抽取执行序列来对程序行为特征进行抽象，并根据语法特点和程序执行信息来定位疑似缺陷语句。随后选择不同的修复模式，运用值替换、表达式替换等方式生成补丁代码。该方法可用于 Effiel 语言的 bug 修复。

Gao 等人针对 C 程序中与内存泄露相关的缺陷，提出了 Leakfix 方法[22]。该方法能够修复大量内存泄漏问题。其主要修复方式是在程序合适的位置插入存储单元分配语句。

Su 等人提出了 SRAFGEN 方法[23]。该方法将大量程序按照控制流结构分类。从错误程序所属的类别中，根据“程序间距离”算法，找出与错误程序最相似的程序。以此程序为基础，根据对应的结构，匹配相应的语句，产生修补方案。

Gulwani 等人提出了 CLARA 方法[24]。该方法根据控制流结构聚类相似的程序，并为每一类选出一个代表程序。对于错误程序，根据控制流结构划分到相应的类，并尝试与该类的代表程序匹配，产生条件匹配对，所有条件匹配对中的条件作为可能的修改操作。最终使用 0-1 线性规划选取最终的修改操作。

## 5 总结与展望

本文提出了一种基于程序合成的C/C++程序缺陷自动修复方法。使用缺陷定位方法：基于重写规则和基于程序频谱，得到程序中可能的缺陷位置。使用修复选项生成方法：基于重写规则和基于预测，得到缺陷的修复选项。并使用程序合成的方法进行修复选项验证，保证合成后程序的正确性。

在本文方法的实验中,我们也遇到了一些问题和挑战,这也是我们未来的研究方向：(1)目前本文关注的是单条语句的修改，并没有考虑增加语句或删除语句的情况，这是我们下一步研究的方向 (2)目前本文采用了基于重写规则和程序频谱的缺陷定位，基于重写规则的缺陷定位依赖一定的先验知识，而基于程序频谱的缺陷定位与精确定位还有一定的差距，下一步将尝试加入更多的程序缺陷定位方法，以处理如指针错误等更多类型的缺陷 (3)目前本文在通过预测的修复选项生成过程中，使用当前可见的全部变量替换语句结构中的相应部分，可以考虑提前过滤掉部分无关的修复选项。(4)目前数据集程序数量较少，下一步将继续收集更多的学生作业程序，在更多的学生作业程序上验证我们方法的有效性。

References:


[1] Manna Z, Waldinger R. A Deductive Approach to Program Synthesis.[J]. Readings in Artificial Intelligence & Software Engineering, 1980, 2(1):90-121.A

[2] Manna Z, Waldinger R J. Toward Automatic Program Synthesis.[J]. Communications of the Acm, 1971, 14(3):151-165.

[3] Gulwani S. Dimensions in program synthesis[C]// International ACM Sigplan Symposium on Principles and Practice of Declarative Programming. ACM, 2010:13-24.

[4] Bodik R, Solar-Lezama A. Program synthesis by sketching[J]. Dissertations & Theses - Gradworks, 2008.

[5] Graves A. Long Short-Term Memory[M]// Supervised Sequence Labelling with Recurrent Neural Networks. Springer Berlin Heidelberg, 2012:1735-1780.

[6] Kong X, Zhang L, Wong W E, et al. Experience report: How do techniques, programs, and tests impact automated program repair?[C]// IEEE, International Symposium on Software Reliability Engineering. IEEE, 2015:194-204.

[7] Kong X, Zhang L, Wong W E, et al. The Impacts of Techniques, Programs and Tests on Automated Program Repair: An Empirical Study[J]. Journal of Systems & Software, 2017.

[8] Qi Z, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems[C]// International Symposium on Software Testing and Analysis. ACM, 2015:24-36.

[9] Weimer W, Nguyen T V, Goues C L, et al. Automatically finding patches using genetic programming[C]// IEEE, International Conference on Software Engineering. IEEE, 2009:364-374.

[10] Forrest S, Forrest S, Forrest S. Leveraging program equivalence for adaptive program repair: models and first results[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE Press, 2013:356-366.

[11] Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches[C]// International Conference on Software Engineering. IEEE Computer Society, 2013:802-811.

[12] Qi Y, Mao X, Lei Y, et al. The strength of random search on automated program repair[C]// Proceedings of the 36th International Conference on Software Engineering. ACM, 2014:254-265.

[13] Long F, Rinard M. Automatic patch generation by learning correct code[J]. Acm Sigplan Notices, 2016, 51(1):298-312.

[14] Nguyen H D T, Qi D, Roychoudhury A, et al. SemFix: Program repair via semantic analysis[J]. 2013, 8104:772-781.

[15] Xuan J, Martinez M, Demarco F, et al. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs[J]. IEEE Transactions on Software Engineering, 2017, 43(1):34-55.

[16] Mechtaev S, Yi J, Roychoudhury A. DirectFix: Looking for Simple Program Repairs[C]// Ieee/acm, IEEE International Conference on Software Engineering. IEEE, 2015:448-458.

[17] Goues C L, Dewey-Vogt M, Forrest S, et al. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each[C]// International Conference on Software Engineering. IEEE, 2012:3-13.

[18] Yi W, Yu P, Furia C A, et al. Automated fixing of programs with contracts[C]// International Symposium on Software Testing and Analysis. ACM, 2010:61-72.

[19] Yu P, Yi W, Furia C A, et al. Code-based automated program fixing[J]. 2011, abs/1102.1059:392-395.

[20] Pei Y, Furia C A, Nordio M, et al. Automatic Program Repair by Fixing Contracts[M]// Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2014:246-260.

[21] Pei Y, Furia C A, Nordio M, et al. Automated Fixing of Programs with Contracts[J]. IEEE Transactions on Software Engineering, 2014, 40(5):427-449.

[22] Gao Q, Xiong Y, Mi Y, et al. Safe Memory-Leak Fixing for C Programs[C]// Ieee/acm, IEEE International Conference on Software Engineering. IEEE, 2015:459-470.

[23] Wang K, Singh R, Su Z. Data-Driven Feedback Generation for Introductory Programming Exercises[J]. 2017.

[24] Gulwani S, Radiček I, Zuleger F. Automated Clustering and Program Repair for Introductory Programming Assignments[J]. 2016.