北京大学

本科学士学位论文

题目：一个双向变换C预处理器的实现与实验

姓　　名：吴逸鸣

学　　号：1100012807

院　　系：信息科学技术学院

专　　业：计算机科学与技术

研究方向：计算机技术

导　　师：熊英飞

2015.05.28

版权声明

摘要

现在有许多关于自动修改程序的研究，例如程序修复工具、程序迁移工具等等。我们把这类工具统称为*程序编辑工具*。同时，许多程序都需要使用C预处理器，比方说C，C++和Objective-C开发的程序。由于预处理器的处理十分复杂，许多程序编辑工具不是无法在有C预处理器的情况下给出可靠操作就是完全放弃处理预处理之前的代码。

本文中我们提出了一种轻量级的双向C预处理器算法。它可以帮助程序编辑工具免去处理C预处理器的烦恼。程序编辑工具现在只需关注预处理后的代码，而我们的双向预处理器可以自动地为编辑工具把修改反向映射到预处理前的代码上。我们在理论上和实践中证明了我们生成的修改时正确的且尽可能地保留源代码的完整性。我们在Linux内核代码上生成修改操作并通过实验验证了我们的方法。实验结果显示我们的方法可行且有效。

关键词：双向变换，预处理器，重写规则

Bidirectionalizing C Preprocessor

Yiming Wu (Computer Sciense & Engineering)
Directed by Prof. Yingfei Xiong

Abstract

Many tools directly change programs, such as bug-fixing tools, program migration tools, etc. We call them *program-editing tools*. On the other hand, many programing use the C preprocessor, such as C, C++, and Objective-C. Because of the complexity of preprocessors, many program-editing tools either fail to produce sound results under the presence of preprocessor directives, or give up completely and deal only with preprocessed code.

In this paper we propose a lightweight approach that enables program-editing tools to work with the C preprocessor for (almost) free. The idea is that program-editing tools now simply target the preprocessed code, and our system, acting as a bidirectional C preprocessor, automatically propagates the changes on the preprocessed code back to the unpreprocessed code. The resulting source code is guaranteed to be correct and is kept similar to the original source as much as possible. We have evaluated our approach on Linux kernel with a set of generated changes. The evaluation results show the feasibility and effectiveness of our approach.

Keywords: Bidirectional Transformation, Preprocessor, Rewriting Rule

目录

序言

0.1 Introduction

现在，许多程序分析工具都涉及代码修改功能。在这些工具中，有许多都是代码修复工具 [1–4]。通常来说，修复工具的输入是一段代码和一组测试，并不断修改代码直至代码能通过测试。另一些程序分析工具是API升级工具 [5–7]。当API升级时出现了不兼容情况时，这些工具可以自动更新相应的API调用让程序与API契合。我们把这类直接修改代码的工具称作*程序编辑工具*。

另一方面，许多程序语言的实现都带有预处理器 [8–10]。最常见的预处理器是C预处理器。许多程序语言也接受C预处理器，包括C，C++和Objective-C。同时，程序员也时常使用C++来写一些零散的小工具。这时就会使用到预处理器。例如Korpela [11]曾在文章中描述过用C++写一个HTML编辑工具：这个工具会把页面间相同的HTML代码转换成C的宏，而不是直接生成HTML页面。然后页面再利用这些宏最终生成HTML文件。

然而，程序编辑工具通常不会去修改程序的预处理指令。但是这并不代表他们能够处理有预处理器的程序。只有能够把修改反映射到预处理之前的代码的工具才算有用。仅仅在预处理后的代码中修复错误会导致原有程序再次编译的时候错误依然存在，这样毫无意义。这个问题具有挑战性，因为工具必须同时能理解预处理命令和目标程序语言，同时保证修改在两边能保持一致。事实上，现有的程序编辑工具往往无法正确处理预处理指令、或是直接不处理预处理指令，例如现有的C语言工具：GenProg [1, 2]，RSRepair [3]，和 SemFix [4]。这三个工具都只在预处理后的代码上工作。用户需要手动检查预处理后的代码变化，并自行修改源代码——而这又增加了新bug的可能。

代码重构是一个密切相关的领域 [12, 13]。在代码重构中，程序编辑工具有

时需要直接修改预处理指令。比如：用户有时想重命名一个宏，或者需要提取一个宏作为重构的一部分。在这种情况下，工具开发者别无选择，只好修改预处理指令。典型情况中，工具开发者会定义一种新的C语法使得原有的C语法和预处理指令能兼容。但是，如果我们考虑更一般的程序编辑工具，这种方法就捉襟见肘了。首先，工具开发者需要在真正设计工具之前把精力花费在学习语言的细节上。其次，学习新语言的努力并不能在其他语言中复用。

本文中我们提出了一个轻量级的支持C预处理器的程序编辑工具实现方法。该系统是一个双向C预处理器：原有的预处理过程可以背看作一个正向变换，考虑程序编辑工具对预处理后的代码做出修改双向C预处理器可以把这些修改反向映射回源代码。于是，程序编辑工具现在可以只关注于预处理后的代码，而不需要考虑预处理器带来的影响，并把映射修改的工作交给我们的自动工具[1]。

在这里我们列举一些例子：（1）上文中所提到的三个学术届认可的错误修复系统现在可以处理预处理前的代码；（2）API升级软件现在可以在有预处理代码的情况下更好地实现；（3）所有并不需要关心预处理过程的程序编辑工具都能够被改善。

如果要实现一个双向预处理器，首先会想到双向变换技术。虽然现在存在着几种双向变换的技术 [15–17]，但是它们都是为数据的转换设计的。给定一个源数据集`s`，一个变换程序`p`，和一个目标数据集`t = p(s)`，这些方法试图将`t`上的变化描述成`s`的变化。然而，C的预处理器与数据的转换不同，因为C的源代码不仅包含了作为数据的代码，还包含了作为变换程序的预处理指令。这就要求反向变换的机制能处理更复杂的情况：当目标数据发生变化时，我们可能要变化源数据、转换程序、或者是二者都变化。

本系统的一个创新处是在能够处理上文提到的复杂双向变换情况的同时，尽量最小化对变换程序`p`的修改。首先，该设计中从不引入新的宏定义或修改现有的宏定义，有效地限制反向变化的影响。其次，该设计只会移除/修改宏调用而并不会创造新的宏调用。再有，该设计只会在必要的时候移除宏调用。这样，我们就能尽可能保证源代码和修改之前的相似度。

实现这样的算法设计也是十分具有挑战的。典型的这类双向变换的方法 [15,

[1] 这个过程并不是全自动的。因为我们的工具暂时只支持程序编辑的基本操作。尽管任意程序修改步骤都可以用通用代码差分方法转换成基本的修改操作 [14]，但是如果工具能直接提供基本编辑操作可以有最好的效果

16, 18]是顺着程序的抽象语法树AST拆分双向变换。每一个子树对应着一个小的双向变换，组合起来就成为整个程序的双向变换。但是，一个未预处理的C++的程序并不能简单地解析成树状结构。比方说，在下列代码中，

```
#define inc(x) 1+x
#define double(x) 2*x
inc(double) 2
inc(double) (x)
```

第一个宏调用`inc(double)`会自动展开成为一个单独的语句（`statement`）。但是第二个宏调用不能单独展开成一个语句。它需要连上之后的`(x)`，循环展开后才算完整的句子。因此，我们不能把第二个宏调用`inc(double)`当作独立的一段并直接对他做双向变换的分析。为了克服这个困难，我们为描述类似C预处理程序的情况设计了新的模型。我们把预处理看作是对代码数据的重写规则（*rewriting rules*）集，而不是直接把代码解析成抽象语法树。这样的模型把程序的双向变换看作是重写规则的双向变换。

另外，该设计也可以被证明满足双向变换的正确性：（1）如果预处理后的程序没有变化，那么源代码也不会变化；（2）源代码在接受了反向映射的变化后，那么新的源代码预处理后和接受了修改操作的预处理后代码是等价的。这两个性质在双向变化领域被称作GETPUT和PUTGET [19]。

总结一下，该项目的贡献如下：

- 我们提出了一个轻量级的在程序编辑工具方面的双向C预处理器。我们分析了不同可能的设计并提出了五个反向变换应有的性质，包括GETPUT和PUTGET (Section 1.1)。
- 我们提出了一个能够符合五个性质的算法。该算法把C的预处理看作是重写规则的集合，并结构性地把预处理的双向变换转换到重写规则的双向变换上 (Section 2.1)。
- 我们在Linux内核上验证了我们的方法，并和另外两种基本的做法进行比较：一个是直接把整个修改的文件映射回去；另一个是只把修改了的代码行映射回源代码。实验的结果显示相较于其他方法，我们的方法破坏了相当少的宏调用，并且总是可以给出正确的修改，而其他方法有时不行(Section 3.1)。

本文接下的部分组织如下：首先我们会在 Section 1.1中描述项目背景和需要解决的问题。接着我们会在 Section 2.1中提出我们的算法，证明它满足双向预

编译器的五种性质。然后我们会在 Section 3.1中给出我们的实现与实验的过程、细节与结果，并进行讨论分析。最后我们会在 Section 4.1中讨论相关工作，并在 Section 5.1中总结。

第一章　背景介绍

1.1　Problem

1.1.1　C预处理器

表 *1.1:* 主流预处理器指令与操作

Directives	Functionality	Example	Result
`#progma`	Compiler options	`#progma once`	removed from the preprocessed file
`#include`	File Inclusion	`#include <stdio.h>`	the content of "stdio.h"

`#if, #ifdef, ...`	Conditional compilation	`#ifdef FEATURE1`
		`  x = x + 1;`
		`#endif`
			`x = x + 1;`

`#define X`	Object-like macro definition	`#define X 100` `a = X;`	`a = 100;`
`#define X(a, b)`	Function-like macro definition	`#define F(x) x * 100` `F(10);`	`10 * 100;`
`a ## b`	Concatenation	`#define X a_##100` `X`	`a_100`
`#b`	Stringification	`#define F(x) #x;` `F(hello);`	`"hello";`
`__FILE__, __DATE__, ...`	Predefined macros	`__FILE__`	main.c

表 1.1显示了主流预处理器的指令与操作。一条预处理指令在行首以#开头，在行末结束。宏可以被预处理指令定义，但是它们自身并不是预处理指令。本质上，我们认为现在有四种主要的预处理指令： `#progma` 提供了编译选项，`#include` 描述了包含的头文件， `#if` 提供了条件编译选项，`#define` 是宏定义指令。另外，在一个宏定义中，我们可以使用类似`##` 和`#` 的指令来连接两个变量、或字符化一

个变量。最后，还存在一些预定义的宏，如 __FILE__ ，会随着上下文的不同而被替换。

当C预处理器处理一个源文件的时候，它会依据以下的方法来转换源程序文件：

- 首先展开 #include 和 #if 指令，然后再重复扫描展开后的代码词序列
- 对于每个宏调用，预处理器先处理参数的展开，然后再展开宏调用。
- 对于含有 # 和 ##的参数，预处理器并不会处理这类参数。相反，预处理器会把参数直接文本拷贝到展开项中。
- 当一个宏调用被展开之后，这个被展开的程序词序列会被再次扫描。如果这时还有宏没有展开，预处理器会把宏调用展开。
- 为了避免循环展开宏调用，如果一个宏定义已经在展开过程中被展开，那么它就不会被再次展开。
- 如果在展开宏的过程中生成了新的预处理指令，该指令并不会被预处理器执行。

```
#define SAFE_FREE(x) if (x) vfree(x);
#define FREE(x) vfree(x);
#define RESIZE(array, new_size, postprocess) \
  g_resize_times++; \
  postprocess(array); \
  array = vmalloc(sizeof(int)*(new_size));
#define GARRAY(x) g_array##x;

RESIZE(GARRAY(2), 100, FREE);
```

图 *1.1:* 一个预处理的例子

这里举一个实例，让我们来考虑在图 1.1里的代码片段。这个例子向我们展示了许多实际项目中的宏定义与宏调用。这段代码中含有四段宏定义和一个宏调用。前两个宏定义被包含在一个用户自定义的空间释放函数里。第三个宏定义是为了用户自定义的内存空间管理和日志记录而重新调整数组的大小。最后一个宏是为了定义一组特殊的全局变量。当预处理器扫描这段代码时，第一个参数 RESIZE 将会被处理。此时， GARRAY(2) 会被展开成 g_array2 。尽管第三个参数 FREE 已经被定义成一个函数状的宏（*function-like macro*），但是并没有能够提供给 FREE 的参数，因此此时预处理器并不会把它当作一个宏调用来处理。然后 RESIZE 的宏调用被展开，于是我们得到了以下的代码：

```
g_resize_times++;
FREE(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

现在我们能看到在展开的宏中，我们已经给 FREE 提供了一个参数列表。因此接着系统会展开 FREE(g_array2)，我们得到以下代码：

```
g_resize_times++;
vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

换句话说，RESIZE 实际上是一个高阶宏（*high-order macro*），因为他的第三个参数也是宏。

1.1.2 反向变换的设计

现在让我们来考虑一下输入为预处理后代码的程序编辑工具们。比方在下面的例子中，程序编辑工具会发现 vfree 可能存在内存泄漏的可能，所以工具会在这句代码前加入一个保护语句，如下：

```
g_resize_times++;
if (g_array2) vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

现在让我们来考虑一下反向变换。反向变换一般来说输入是预处理后代码上的修改，然后产生预处理前代码上的修改。当生成的修改作用在预处理前的代码上后，新的代码在预处理后会得到与作用输入修改的预处理后代码相同的结构。对于之前例子里面程序编辑工具做出的修改，我们有两种处理办法：（1）我们可以修改 RESIZE 的宏定义，把加入的 IF 语句加入到宏定义中；（2）我们可以展开 RESIZE 的宏调用，并按照预处理后的修改把保护语句添加到原程序中。第二个选项只影响到了这一小段局部的代码，而第一个选项有会影响到全局的其他宏调用。但是，因为反向变换并不知道应该把修改作用到局部还是全局，一个更安全的做法是选择只影响局部的代码。在本例中，有很大可能我们并不需要对每一个vfree的调用都加上保护语句，因为这样会带来大量不必要运行时间。我们有理由相信程序编辑程序会根据需要选择是否在vfree前加上保护语句。进一步讲，局部的选项会尽可能小地修改原有代码，因为全局会影响程序的许多部分。这些可能性让我们想到一个双向预处理器的第一个性质。

Requirement 1 *反向变换不应该改变任何宏定义。*

根据我们定义的第一个性质，我们应该展开宏调用后再在源代码中加入保护语句。然而现在在映射修改时我们又面临着以下两个展开的选择。我们可以把所有的宏都依次展开：

```
g_resize_times++;
if (g_array2) vfree(g_array2);                    (1.1)
g_array2 = malloc(sizeof(int)*(100));
```

或者我们也可以只把宏展开一层：

```
g_resize_times++;
if (GARRAY(2)) FREE(GARRAY(2));                   (1.2)
GARRAY(2) = malloc(sizeof(int)*(100));
```

我们认为代码段 1.2比代码段 1.1更好，因为它保存了更多原有的结构，这使得代码更加易懂，重用或是维护。这就引入了我们的第二条性质。

Requirement 2 *反向变换应该尽可能保存现有的宏调用。*

也许有人会提议我们进一步地把这个保护语句缩减成一个新的宏，或者复用现有的某一个宏来实现保护语句的功能。在本例子中，我们的保护语句被 `SAFE_MACRO` 这个新宏定义包含。代码可能如下：

```
g_resize_times++;
SAFE_FREE(GARRAY(2));
GARRAY(2) = malloc(sizeof(int)*(100));
```

但是，这样做是十分危险的。因为宏定义并不是用来替换所有语义相同的代码片段。比方说，Ernst等人 [8]就在文章中描述过某一个宏定义， `#define ISFUNC 0`，定义了一个在系统调用中时常用到的常数。很明显，我们并不能把整个系统里的 `0`都替换成 `ISFUNC`。这就引入了我们的第三个性质。

Requirement 3 *反向变换不应该引入新的宏调用。*

除了已经提到的三个性质之外，我们还有两条从双向变换领域借鉴过来的性质，叫做 GETPUT 和 PUTGET [19]。令s是预处理前的源程序，t是预处理后代码，c_t是作用在t上的变化，c_s是完全的反向变换所提供的作用在s上的变化。

Requirement 4 (GETPUT) 如果c_t是空的，那么c_s也是空的。

Requirement 5 (PUTGET) 令$s' = c_s(s)$为新的预处理前的源代码。意为把生成的c_s作用在s上。令$t' = c_t(t)$为作用了修改操作的预处理后代码。对s'做预处理会得到t'。

以上五条性质一起定义了反向变换的行为：它应该通过修改宏调用参数、修改普通代码、展开宏调用并且展开尽可能少的宏调用的方法来把预处理后的修改映射到源代码上。

1.1.3 简单的方法

我们将在本节中讨论为什么简单的方法并不能满足我们提出的五条性质。

简单的方法 I (per-file). 第一种简单的方法是直接把修改过的文件不加处理地拷贝覆盖源代码。这种方法十分容易实现，我们称之为*per-file*，但是他有两大缺点。首先，原有的未预处理源代码可能含有宏定义，然而现在的做法会让这些宏定义都丢失掉，那么在其他地方，比如包含了该文件并调用了相应宏的其他代码，可能就会出现错误。其次，这会导致把文件中所有的宏调用都展开，甚至包括所有的 `#include` 指令。整个代码面目全非，破坏了完整性。

简单的办法 II (per-line). 第二个简单的办法是利用代码的性质，只把预处理后代码中被修改的行反向映射回源代码。我们把这个方法称作*per-line*。这个做法看起来是可行的，因为现代的预处理器会记录下预处理前后行间的对应关系。比方说，当GCC预处理我们之前的例子时，它会把1-7行替换成空行。同时它会把从第9行开始的几行展开压缩成一行。可以看到，现代预处理器都记录下了源代码与预处理代码行间的一一映射。

虽然第二种方法相较于第一种有了不删除宏定义的好处，它依然存在隐患。首先，依然有不少宏被不必要地展开了。在我们的例子中，如果我们把修改的

那一行复制回去，那么代码段 1.1就会时结果，但这与我们定义的第二条性质不符。甚至，如果我们考虑有的工具会在代码中挑选代码复制插入来修改，比如GenProg [1] 就会在代码中拷贝不同地方的代码来修改程序的错误。这样一来拷贝的行中所有的宏调用都会被展开，代码的完整性还是被破坏。另外，这还并不是最严重的。如果源文件中的宏调用是一个多行宏（在我们的调研与实验中也确实发现了这样的情况Section 3.1），那么只替换宏调用的第一行会直接带来错误的结果，反而引入了新的bug。比方说，如果在源代码的宏调用中插入了一个断行，如下：

```
RESIZE(GARRAY(2),
  100, FREE);
```

在GCC中，这个宏调用会被展开成两行。其中第一行时全部的宏展开语句，而第二行是空行。因此，修改操作只会作用于第一行。如果反向变换仅仅把第一行替换成新的代码，那么就会留下不正确的程序。

第二章　算法设计

2.1　Approach

正如之前提到的，我们算法的基本思想是把C预处理器当作一组重写规则，而反向变换就是这些规则相应的逆向规则。在本章中，我们会先描述本项目的C预处理器的模型（Section 2.1.1）。然后我们会描述该系统所支持的修改操作。接着我们会集中阐述其中第一种操作：替换操作的处理方法（Section 2.1.2）。因为我们需要吧每一条重写规则反向应用，因此我们需要记录下预处理时使用重写规则的顺序（Section 2.1.2），然后按照顺序依次为这些规则生成反向变换（Section 2.1.3）。我们也会讨论这些步骤/过程为何能满足我们之前讨论的双向预处理器的五条性质（Section 2.1.4），并且给出一个更优化的算法。最终，我们会讨论如何把不同类型的修改操作都转换成替换操作（Section 2.1.6）。

为了读者可以更快理解我们算法设计的思想，我们暂时只考虑C预处理器指令的一个子集：去除 # 操作和 ##操作，去除 `#include` 操作，也不考虑宏出现循环调用的情况。我们会在之后的章节中（Section 2.1.5）讨论如何把子集上的模型扩充到支持全部C预处理指令的完整模型。

2.1.1　模拟正向变换：预处理

我们把C预处理程序需要处理的程序看作词（*token*）的一个序列。为了从词序列中识别出预处理指令，我们依赖于两个特征词：行首的#符号和该行最后的换行符。我们同时也假设当前环境中所有定义的宏都存储在环境变量`context`里。

我们把C预处理器语法看作是重写规则的一个集合。每个重写规则都有`guard` $\hookrightarrow$ `action`这样的形式。当`guard`是真时，`action`会把当前词序列的

前几项替换成指定的词，然后在替换的位置后继续下一条替换指令。

在模型中，guard 和 action 都可以被看作是函数。单独来看， guard 函数的输入为当前剩下的词序列和当前的上下文环境 context，它将会输出一个表示是否要把当前规则应用到现在的词序列的布尔值。与此同时，action 函数把当前还剩下的词序列，上下文环境当作输入，然后生成一个四元组 (finalized, changed, restIndex, newContext)。这个四元组中的变量含义如下： restIndex 表示在这变量之前的词都已经被重写规则处理过了。在这些被处理过的词序列中，finalized 表示了一串不需要再应用规则的词序列，而 changed 是可能还需要被扫描的序列。而最后一个变量 newContext 代表着更新过的上下文环境。

有了这些定义后，我们算法中的正向变换部分在算法 1中展示。该算法循环应用规则 R 直至整个词序列都被处理。

Input: token sequence src, rule list R
Output: new token sequence res
ctx ← {};
while src.length > 0 **do**
  for r ∈ R **do**
    if r.guard(src, ctx) **then**
      (finalized, hanged, rest, ctx′) ← r.action(src, env);
      break;
    end
  end
  res ← res + finalized;
  src ← changed + src.sub(rest);
  // sub(l) returns a subsequence starting from l
  ctx ← ctx′;
end

Algorithm 1: 正向预处理算法

在C预处理情况中，规则列表R中总共有四个规则。其中一个处理条件预处理指令，例如#if, #ifdef；一个处理其他的预处理指令，这样我们可以用它来清除预处理指令并且更新上下文环境；一个处理宏调用；最后一个处理普通字符文本。这四个规则如下定义：

- 规则1: 这个规则处理条件编译选项。guard函数将判定当前词序列的开头是否是一个独立的预处理条件指令，例如#if，#ifdef。如果为真， action函

数首先会在当前上下文环境中检验条件选项是否为真，然后根据是否为真选择使用真值分支或假值分支编译。选择分支后，用该分支替换原有的指令，并把替换成功的新词序列记录在changed里。而finalized是空的。

- 规则2: 这个规则处理其余所有预处理指令。guard函数将判定当前词序列的开头是否是一个独立的预处理指令。如果为真， action函数会解析这条指令，并对当前上下文环境做出必要的调整，比如添加一条宏定义。最后，该指令之后的词下标将会被记录成restIndex，而finalized和changed为空。
- 规则3: 这条规则会展开宏调用。guard函数将判定当前词序列的开头第一个词是否是一个对象宏调用（*object-like macro*）或者当前词序列的开头前两个非空白词是否是是一个函数宏调用（*function-like macro*）和一个开括号。如果为真， action函数会做以下两个步骤的操作：
  - 首先，我们会用规则3和规则4循环调用处理参数[1]。
  - 然后，我们把宏展开中各个参数出现的位置都替换成已经处理完了的宏参数。整个展开的部分都被记录在changed里，而finalized是空的，restIndex记录了宏调用之后的第一个词的位置。
- 规则4: 这个规则处理那些没有被其余规则处理的普通文本词。guard函数总是返回真。 action函数会把词序列中的第一个词放入finalized中，返回一个空的changed，并把下一个词的下标标记为restIndex。

让我们来看一个例子来理解这些规则的应用。考虑以下程序。

```
#define x 100
hello x
```
(2.1)

当我们的系统处理这段代码时，第一个能应用的规则是规则2。它会把第一个宏定义解析出来，存储在上下文中，并把接下来的下标移动到下一行。这时剩余的词序列为 `hello x`。接着应用规则4，并把 `hello`移动到finalized的序列中。然后应用规则3，把宏调用 `x` 展开成 `100`。最终，系统将会再次扫描一遍 `100` 并使用规则4把它移动到finalized序列中。当剩余词序列为空时，整个处理过程就停下了。

[1] 在C预处理器中未定义把预编译指令放在参数中的操作 [20]。在次我们认为在参数中不存在预处理指令

2.1.2 模拟修改

程序编辑工具可以通过各种各样的方式对一段程序进行修改。本文中我们考虑三种基本的修改：替换、插入和复制。这三种基本的修改是我们在分析了现有的主流代码修复工具例如GenProg [1] 和 SemFix [4] 之后总结而成。这些代码修复工具往往会拷贝或创造一个语句来替换现有语句或插入来修改程序错误。

这三种操作都直接地对词序列进行操作。一个替换操作描述为一个二元对`(l, s)`，其中`l`是要被替换的词的位置，而`s`是将要替换在`l`位置的一串词序列。一个插入操作也是一个二元对`(l, s)`，其中词序列`s`将会被插入到位置`l` 之后。一个拷贝操作时一个三元组(l, l_b, l_e)。它表示从位置l_b（包含）起至 l_e（不包含）的词序列将会被拷贝插入到位置`l`的词之后。

我们可以看出替换操作涵盖的删除操作。一个形式为`(l, [])` 的替换操作就代表删除了一个词，其中`[]`表示空序列。

本章接下来的部分，我们将讨论如何实现一个可以处理替换操作的反向变换。我们也会讨论如何把其他两种操作转换成替换操作。

2.1.3 重写步骤

正如在序言中介绍过的，我们的反向变换操作会逆向实施之前提到的重写规则。为了生成好的反向变换，我们设计了一个数据结构来记录下正向展开时使用了哪些规则，分别应用在代码的哪个部分。我们把这种记录的数据结构称作 *重写步骤*。每一个重写步骤都是一个四元组，`(src, i, ctx, r)`。其中`r`表示规则`r`已经被应用到一段子序列`src`。自序列位置在`i`，应用规则时的上下文环境为`ctx`。其中`src`总是代表了暂时的词序列，区别于源序列与最终序列，是应用了之前所有规则后生成的临时序列。

相应的，一个完整的正向序列会被记录成一个重写步骤序列。比方说，正向处理之前例子中的代码段 2.1 时，我们的算法会产生以下的重写步骤序列：`(P, 0, {}, R2)`，`(hello x, 0, {x}, R4)`，`(hello x, 1, {x}, R3)`，和`(hello 100, 1, {x}, R4)`。其中`p`是源程序。

2.1.4 考虑替换操作的反向变换

反向操作的基本思想是把预编译后代码上的修改通过重写规则序列一步步映射到预处理前的代码上。这样当我们再跑一次正向变换时，只会有两种情况：1. 程序会按照相同的重写规则再次正向展开 2. 程序会按照等价的重写规则序列展开，这个规则序列长度可以是0。我们设计的反向变换操作在算法 2中。其中backward函数沿着每一个重写步骤映射程序上的修改，或者是当它发现没有合适的修改可以映射时报错。

Input: a sequence of rewriting step `rs`
Input: a set of replacements `c`
Reverse sequence `r`;
for `r` ∈ `rs` **do**
    `c` ← `backward(rs, c)`;
    if `backward` *failed* **then**
        print "Changes cannot be applied.";
        return;
    end
end
return `c`;

Algorithm 2: 生成反向变换算法

我们从伪代码中可以看出，实现反向变换的关键在于如何实现 backward 函数。 backward 函数的行为随着规则的不同而改变。在此，我们根据复杂度，从简单到复杂讨论该函数的功能。

首先，如果当前重写步骤是依据规则2处理的，这意味着此处处理了一个非条件的预编译指令，正向预编译时，这条指令被预处理器删除。反向变换会根据预处理指令的位置来考虑是否要偏移操作。比方说，考虑以下的代码程序：

```
#undef hello
hello
```

现在如果我们有一个`(0, x)`的替换操作，那么在这个例子中， `hello`会被替换成`x`。 `backward`函数会把该替换操作位移到`(4, x)`来保证当前替换操作依然是作用在`hello`这个词上。

第二，如果当前重写步骤是依据规则1处理的，这意味着此处预处理器处理一个条件预处理指令。预处理器当时把这一段替换成了条件指令的一个分支。在

反向变换中，我们把在分支上发生的修改操作映射到原来的条件指令中，同时计算必要的偏移量等。因为我们不会改变宏的定义，此处的条件预处理指令在再次预处理时，还是会选择相同的分支进行替换。

第三，如果当前重写步骤是依据规则4处理的，这意味着此处预处理器处理了一段非指令非宏调用的普通文本字符。正向预处理器应该把一个词放到finalized里去了。反向变换需要保证在修改过后的词序列上，依然会应用规则4。举例来说，如果用户把hello c 中的 hello 替换成了 a b，我们需要检查，从a开始，是否唯一的预处理办法就是应用两次规则4来替换当前序列。在这个例子中，只有当规则4的guard函数能够在要么 a b c 或者 b c上返回真才行。这时候就会发生反向变换失败的情况。比方说，如果程序编辑程序给出的修改时把hello修改成x，其中hello并不是宏调用，但x是宏调用。那么此时应该报错。另一个情况是程序编辑程序把GARRAY hello替换成了GARRAY (hello)。此时GARRAY是一个含有一个参数函数式宏调用。在这种情况下，之前GARRAY被当作文本使用规则4重写因为它之后并不是括号。而在新的序列中，GARRAY变成了一个宏展开。此时很有可能造成程序错误。以上两种情况backward函数都应该报错，自动程序并不能轻易地把修改映射到代码中去。

最后，如果当前重写步骤是依据规则3处理的，这意味着此处预处理器处理了一个宏调用。这种情况是最复杂的。规则3包含了两个步骤。我们首先试图沿着两个步骤逆向重写代码。如果两个步骤中有一个失败了，那我们只能试着展开宏调用。

在第二个小步骤中，宏展开中各个参数出现的位置都已经用展开的参数替换了。如果有必要，反向变换需要把作用在展开后的参数上的修改都映射到参数上去。这时有两种情况会导致逆向重写失败，也就是说我们不得不展开宏调用：（1）被修改的词并不是宏的参数出现的位置中的；（2）同一个参数在多个地方出现，被多次修改，但不同地方修改的内容不一样。比方说，考虑以下代码段：

```
#define x 1
#define plus(a) a+a
plus(x);
```
(2.2)

如果我们把 1 + 1 变换成 1 - 1 或者 1 + 2，我们在反向时就得展开宏调用。

在第一个小步骤中，我们预处理了宏的参数。反向变换应该循环地调用backward来先把参数上的修改通过反向的重写步骤应用到每个参数上。最终，我们需要对整个代码段在映射修改时做一次安全检查：如果被修改的参数在最后含有了一个逗号，并且这个逗号没有被单独的括号包裹住；或者是存在括号不匹配的情况，我们还是必须展开宏调用。这是因为一个顶层的逗号或者不匹配的括号会破坏原有的参数列表结构，影响程序的正确性。

如果在上述两个反向步骤中，反向变换失败，我们不得不展开宏调用。展开宏调用时，会产生作用在宏展开式上的相应修改操作。举个例子，我们不妨假设在上面的代码例子 2.2中1 + 1中的第二个1被替换成2。此时，映射来的修改会把 plus(x) 中的 plus 替换成 x + 2 然后删除掉 (x)。

在展开宏调用的过程中，关键是如何构造替换序列，在刚才的例子里时x + 2。从正向变换我们可以看出，没有变过的预处理词序列为 1 + 1，而且这两个 1 都来自参数 x。所以我们把参数 x 的重写步骤拷贝到两个 1 的词上。然后使用这些重写步骤来反向映射作用在这两个 1 上的修改操作。第一个 1 并没有变化，所以原来的 x 可以继续保留。但第二个 1 被替换成了 2，因此该宏调用需要被展开，x 也需要被展开。于是我们得到了最终的替换文本 x + 2。

但是，我们不能总是把参数的重写步骤原封不动地拷贝到每个参数在展开式中出现的地方。基本上说，当我们拷贝一个参数的重写步骤时，我们其实是在假设每一个参数出现的位置可以用参数的未预处理形式来替换。而且替换后每个地方参数的重写规则与它单独展开是一致的。比方说，当展开 plus(x) 时，我们可以把两个 a 出现的地方替换成 x。此时生成的展开式 x + x 中两个参数出现的地方的展开重写规则序列和 x是一样的。然后就能得到 1 + 1。但是，情况并不总是这么理想，我们可以考虑以下的代码例子。

```
#define p (x)
#define pplus(x) plus x hello                                    (2.3)
pplus(p)
```

加上之前代码段 2.2 中定义的宏，这里的例子最终会预处理为1 + 1 hello。但是，如果我们需要展开pplus(p)，我们并不能把它直接展开成plus p hello，因

为这只会展开成 plus (1) hello。这是因为在这里使用 p 而不是 (x) 破坏了原有的宏调用。

因此，我们需要在拷贝重写步骤的时候做一次安全检查。我们当且仅当以下条件都不满足时，才会拷贝重写步骤。

- 预处理参数起始字符是一个左括号。
- 预处理参数在顶层包含一个逗号。
- 预处理参数中存在括号不匹配情况。
- 如果我们重新预处理该参数，结果会不同。

前三个条件对应着之前例子中类似的情况：预处理中参数在展开式中被用于其他的宏调用，替换它会导致展开式中的宏调用出错，给程序引入错误。最后一个条件对应着以下情况。

```
#define id(x) x
id(plus p)
```

这段代码展开后得到1 + 1，但是如果我们把 id(plus p) 展开成 plus p，另外一个宏展开就会被阻塞。换句话说，在宏调用中，一个参数会被扫描两次。但是当宏调用被展开时，一个参数只会被扫描一次。我们需要确认这之间的区别不会影响变换的正确性。

正确性

根据之前一节对反向变换的描述，我们可以简单推导出我们的反向变换满足性质1、3、4和5。性质2将会在实验部分证明。该反向变换满足性质1因为我们从不把修改操作映射到宏定义重。该反向变换满足性质3因为如果我们通过修改操作引入了一个新的宏调用，我们需要把一个普通的文本词变成宏调用的一部分。但是这个文本词本来是通过规则4被预处理的。而规则4中反向变换的安全检查会阻止错误的发生。该反向变换满足性质4因为我们只能在不得不展开某个宏调用时才会引入新的修改操作，而该处没有修改操作，我们不会展开宏调用。该反向变换满足性质5可以这样推导。为了正式地证明性质5，我们需要诱导性地证明每一个重写步骤只能要么保持同样行为，要么被一串等价的重写步骤序列替换（该序列长度可以为0）。所幸，当我们拷贝参数的重写步骤到展开式中时，拷贝后的重写步骤可能会和之后的重写步骤混合起来。我们可以指出，由于这些重写步骤

序列作用位置不同且是覆盖性的，调换重写步骤序列的位置并不会对结果产生影响。

优化算法

在现有的算法中，我们需要在每一步重写步骤后把整个程序的状态都记录下来，即使只是很小的一部分发生了改变。在反向变换中，每一步我们都需要调整所有修改操作的位移。这样的算法使我们的系统十分笨重。

为了优化该算法，我们首先把源程序中的词序列切分成一组序列，序列有自己独立的一套重写步骤。这样，我们就能把每个序列当作单独的程序来做反向变换，再把修改融合到一起。给定一个重写步骤 `(src, i, ctx, r)`，如果在 `i` 位置的词不是由之前的重写规则生成的，也就是说在原来的代码中就有，那么我们就可以认为位置 `i` 可以是一个*切分点*。有了切分点的定义后，由于词是程序的最小单位，我们可以认为没有重写规则会同时作用在切分点的两边。我们依据切分点把程序切分成序列的集合，然后依次在序列上做反向变换，再融合这些修改。当然，再合并两个子序列时，我们还需要特殊检查两边程序在预处理后，合并起来不会形成新的宏调用。这个检查方法与之前提到的规则4的安全检查方法类似。

比方说，之前的代码段 2.1可以被切分成三个子序列：宏定义， `hello`，和 `x`。假设之前定义过宏 `plus`。如果程序编辑工具把 `hello` 修改成了 `plus`，把 `100` 修改成了 `(x)`，那么两个子序列在预处理后，合并起来就在边界上形成了一个新的宏调用。我们应该报错。

2.1.5 扩展到完整的C预处理器

本节中我们讨论如何把我们的算法扩展到支持完整的C预处理。由于篇幅限制，我们只讨论核心思想而非细节。

为了在正向预编译中支持 # 和 ##操作，我们需要在之前提到的规则3的正向变换中添加两个新的小步骤。一个为了支持#可以字符化一个词，一个支持 ##可以连接两个词。另外，如果在参数中出现这两类操作，我们会直接使用未预处理的形式来替换展开式中出现的位置。

在反向变换中，我们需要添加3个新的扩展。第一，我们需要为原有的规则3依据之前所添加的小步骤，设计两个新的反向步骤来支持#和##操作。第二，我们

需要检查使用这两个操作的代码在预处理之前和预处理之后是否有同样的修改。最后，当展开一个宏的时候，我们并不需要恢复#和## 操作，因为他们不能出现在没有宏定义指令的地方。

为了在正向变换中支持 #include指令，我们需要为 #include指令添加新的一条重写规则。其内容为删除这行宏指令并引入新的文件内容。在反向变换中，我们需要记录加入了哪些文件和内容，并根据不同的 #include指令，把这些文件引入都一一返回回去，这样才能保持一致。

最后，一个完整的C预处理器不允许宏的循环调用。为了支持这个功能，我们需要在正向重写规则中添加一些细节的考察。当我们在不断重写由宏 m 展开的词时，我们需要检查是否有再次使用这个宏展开的情况。如果出现，应该报错。

2.1.6 扩展到其他修改操作上

在上文中我们详细地讨论了如何处理修改操作。现在让我们来讨论如何处理插入和拷贝操作。我们可以注意到，插入操作可以直接被转换为修改操作。如果我们要在词 x 之前插入一个词 y，那我们可以把这个插入操作转换为一个将 x 变为 y x 的替换操作。但是，这样的转换有时会引入不必要的宏调用展开。比方说，如果我门在之前的代码段 2.1中在 100 之前插入了一个词 y，那么我们在反向变换中并不需要拆开 x 这个宏。但如果我们把这个插入操作转换成一个替换操作的话，比如把 100 变成 y 100，那么我们就需要把宏 x 拆开。

刚才的例子是在切分点上插入了一段词，而这样的插入可以简单证明，不会影响到宏，也不需要再反向编译时展开宏。为了减少不必要的反向宏展开，我们把插入在切分点的修改操作看作是添加了一个独立的子序列，然后直接在上面检查是否可以应用规则4。如果其他规则可以在这个插入序列上应用，那说明这段插入是错误的，将会由我们的安全检查报错。这样一来，其余的代码依然可以使用之前的算法处理。

拷贝操作的处理和插入操作类似。二者之间唯一的区别是，拷贝操作引入的代码段中可能包含宏调用，而我们需要尽量不展这些宏调用。为此，我们需要对这段词序列做一次反向转换。这次反向转换算法如下：（1）首先，我们假定对这段代码以外的所有预处理后代码做删除操作。（2）然后，我们剔除掉那些没有在目标代码段定义的宏并做一次反向变换。这样我们就能保证只有在目标代码段的

上下文环境中定义的宏会被正确地反向变换回去。最后我们可以把已经做好反向变换的这段代码插入到预处理前的相应位置中。

第三章　实验与分析

3.1　Evaluation

3.1.1　研究问题

本章中，我们将围绕以下几个研究问题来展开讨论。

- **研究问题1: 宏调用保留**。 针对我们提出的双向预编译器性质2，我们的系统希望尽可能地保留宏调用。我们在实验中想知道在实际应用场景中，我们的方法能保留多少宏？和其他方法比较，我们的方法是否更优？
- **研究问题2: 正确性**。 前文中我们已经证明过，我们提出的双向算法满足性质4与性质5。我们在实验中想知道保证这样的正确性在实际应用中究竟有多重要？我的方法相比较于那些没有保证正确性的方法，是否在实际场景中更优？
- **研究问题3: 报错功能**。 在之前的章节中提到，当我们的方法发现没有正确的映射方法时，系统将会报错。在实验中我们想知道现实中这种情况发生频率多少？会不会存在误报的情况（存在一种可以正确映射的方法，但我们的系统没有找到）?

为了回答以上几个研究问题，我们在一组生成的Linux内核源代码上做了一组控制实验。另外，我们将我们的算法和前文Section 1.1.3中提到的两个简单方法进行比较。本章的余下部分，我们会介绍实现与实验的细节。

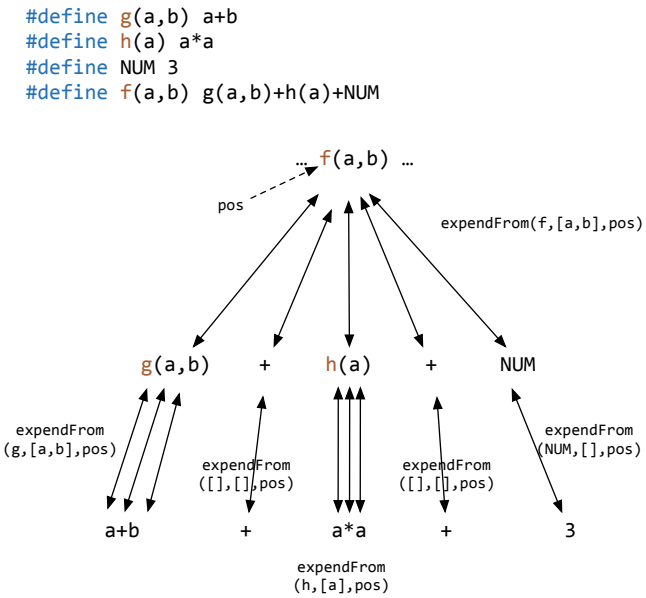

图 *3.1: 数据结构示例*

3.1.2 环境搭建

JCPP

我们基于一个开源的C预处理器，JCPP[1] 实现了我们的双向预处理器，BXCPP。 JCPP是一个完整的，兼容性的，独立的，纯Java实现的C预处理器。他也可以处理 Apple Objective C 库。

JCPP项目中内置了基本的依照GCC规则的C处理器，其中定义了代码词 `Token`类，宏 `Macro`类，以及识别预处理指令并序列化代码词的预处理器类 `Preprocessor`。开发中学习了JCPP预处理器的思想，在此感谢作者之前的工作。

数据结构

实现系统时，采用的是前文（2.1.4）优化过的算法思想。优化过的算法会依据切分点，把代码切分成许多小段，并独立进行预处理和反向预处理。本小节中我们重点介绍一下实现过程中的数据结构，并通过数据结构简单介绍算法。在这里我们可以看一个例子：

[1] http://www.anarres.org/projects/jcpp/

图 3.1展示了宏定义和宏调用展开后的数据结构。构造过程如下：

- 预处理器扫描源代码发现宏定义指令，将宏定义及其展开式纪录在数据结构中。
- 当预处理器扫描到位置 pos 时，根据算法规则3，发现函数式宏（*function-like macro*）调用，分析参数数量，确认宏调用
- 确认宏调用后，在 pos 位置建立切分点，认为 f(a,b) 是一个独立代码段单元。在实现中，这些单元用Unit类及其子类实现
- 独立对f(a,b)递归正向展开。正向展开算法在前文中提到（算法 1）。纪录展开时响应参数位置。

图 3.1中的双向箭头表示了词和代码段单元之间的关联关系。expandedFrom中的三个参数表示在下一层的词在正向展开时的来源。第一个参数描述了该词来源于哪一个宏定义；第二个参数描述了展开宏时的参数列表，[] 代表空，即没有参数；第三个参数描述了顶层宏展开在源代码中的位置。在本例中，第二层中的 NUM 指向第一层的箭头中，f 表示该词被名为 f 的宏展开，[a,b]表示参数列表，pos表示展开在源代码中的位置。

我们可以看到，记录下了这些信息和前文提到的重写步骤的信息可以互换。因此我们就能在这样的数据结构上应用我们的反向算法，实现双向预处理器。

BXCPP框架

我们基于JCPP实现了自己的双向预处理器BXCPP。所有的项目代码和实验数据都可以在我们的项目网站[2]上找到。在此我们简介一下BXCPP框架中主要的几个类。

图 3.2是代码中主要类的类图。小箭头表示包含关系，空心箭头表示继承关系。我们重写了JCPP中的预处理器 Preprocessor类，构造自己的双向C预处理器 MyPreprocessor。该双向预处理器可以识别预处理指令，建立上下文环境纪录和宏定义索引。同时，双向预处理器把源程序词序列化后，将会把序列保存在类型 MySegment里。 MySegment封装了词序列和代码段单元序列，同时包含识别拆分代码段单元的函数，类似算法中的guard函数。

Unit及其子类描述了各种不同类型的代码段单元，他的子类中：StringUnit表

[2] https://github.com/harouwu/BXCPP

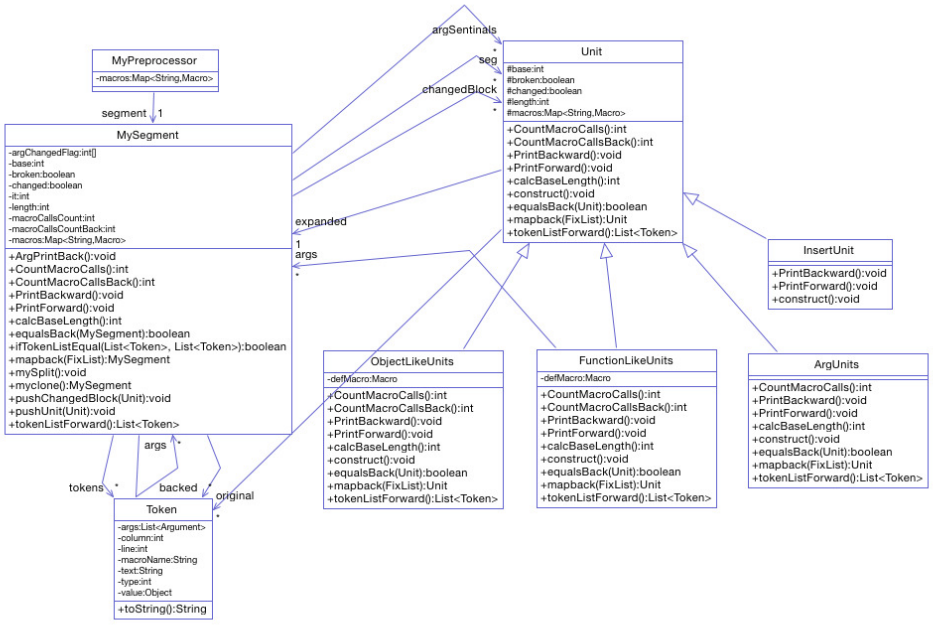

图 *3.2:* 主要类类图。

示普通文本代码段单元， `ObjectLikeUnits`表示非函数宏调用代码段单元，`FunctionLikeUnits`表示函数宏调用代码段单元， `ArgUnits`表示宏调用参数代码段单元， `InsertUnit`表示展开式中插入宏参数的代码段单元。

从类图中可以看到，`MySegment` 和 `Unit` 类互相包含。 `MySegment` 中含有 `Unit` 代码段单元的序列，而`Unit`在展开时，又包含新的代码段`MySegment`。这样循环的关系模拟了上一节 Section 3.1.2中的数据结构，实现了算法。

另外，项目中还定义了许多其他类，如`Token`类描述了词的文本类型等。具体的项目文档和程序细节，可以访问我们的项目网站了解。同时我们也实现了前文中提到的 *per-line* 和 *per-file* 两种简单的反向预处理操作。至此，实现部分基本搭建完成。

实验基准

我们的实验在Linux内核版本3.19上执行。我们选择Linux内核源代码因为Linux是现在最广泛的C语言项目。这之中有许多程序员贡献的代码，包含了不同

风格的代码，也含有大量的预处理指令和宏调用。

为了执行我们的实验，我们需要在Linux内核代码上生成一组修改。因为我们想看到不同反向预处理算法对程序的影响如何，所以我们需要的修改应该尽可能出现在有宏调用的函数中。同时我们在观察中发现，在实际代码中，非函数式宏调用的出现频率远远高于函数式宏调用。这样导致如果随机生成修改，大多数修改都会修改非函数式调用，而少数会修改函数式调用。所以我们根据概率模型，控在非函数式和函数式宏调用的比例为1.5：1。基于上面的这些计算和宏调用，最终我们从项目中抽取了8000行代码，这之中一共出现了133次宏调用。

接着我们为这抽取的8000行代码生成一组修改。为了模拟现实工程项目中出现的修改，我们随机性地生成两类修改操作。第一种是词级别的修改操作，我们随机地替换、删除或插入一个次。第二种是语句级别的修改，我们随机地删除一句语句、或者从别处随机抽取一句话插入当前位置。这两种修改的选择是取自现在主流的代码修改工具的方法 [1, 3, 21]。其中，GenProg [1] 和 RSRepair[3]会直接食用语句级别的修改。而第一种词级别饿修改则模拟了 PAR [21] 中使用的例如替换参数、修改操作符等细微操作。

具体的修改生成过程如下：我们以概率p选择在每个词上进行操作。这个操作可以是插入、替换或删除，三者等概率。替换操作随机调换一个词中不同字符的位置。插入操作随机从别处拷贝一个词插入。类似的，对于每个句子，我们也有味q的概率对它生成修改。修改内容可以是复制或者删除。复制操作直接把之前一句语句复制过来。我们根据C语法，根据分号来划分句子。

不同的程序编辑工作会有不同的修改模式：一个移动工具可能会修改程序中的许多未知，但一个代码修复工具只会修改程序的少数地方。为了模拟不同工具之间给出的修改模式，我们设计了两套不同概率的修改操作集。一套是高修改密度的操作集合，我们设置`p = 0.33` 和 `q = 0.1`。另一套是低修改密度的操作集合，我们设置`p = 0.1` 和 `q = 0.05`。

我们生成了10组修改操作集合，其中5个是高密度的，5个是低密度的。具体生成的修改数量在表 3.1中。

自变量

实验中我们认为以下变量是自变量：（1）*Techniques*，我们认为我们的算法和

表 *3.1: C*实验中生成的修改操作

Low Density	Set	1	2	3	4	5
	Changes	952	885	956	967	884
High Density	Set	6	7	8	9	10
	Changes	3133	3136	3088	3123	3048

另外两种简单的算法， per-file和per-line不同。（2）修改密度，不同组之间，修改出现的密度不同。我们的实验中同时出现高密度和低密度。

因变量

实验中我们认为有两个因变量。（1）剩余宏调用的数量。我们在反向预处理后再次调用正向预处理，依次纪录有多少宏调用被展开。因为实验的方法中没有一种会引入新的宏调用，所以纪录下的就是剩余宏调用的数量。为了减少包含系统头文件中的宏调用带来的干扰，我们只会记录当前文件中的宏调用展开次数。（2）错误的数量。这里的错误是指当我们在反向预处理后再次调用正向预处理，比较新生成的预处理后代码和之前应用修改的预处理后代码之间是否不同。这里我们使用Unix系统提供的文件比较工作 `fc`。每当`fc`给出一处不同时，我们认为是一个错误。（3）报错。我们的方法会检测修改能否被映射回去，我们对于生成的修改操作集合，我们也会记录报错的数量。

3.1.3 影响实验可信度的因素

一个影响实验外部可信度的因素是我们生成的这些修改操作能不能被一般化为真实项目中产生的程序修改。为了减轻该因素的影响，我们使用了从现有工具分析得来的不同类型的修改，同时控制修改出现的频率，这样可以更好地模拟现实项目中的程序修改。

一个影响实验内部可信度的因素是我们实现的三种双向预处理器可能实现错误。为了减轻该因素的影响，我们通过在Linux内核，自写测试集上不断修改错误来保证我们的实现是可信可靠的，不会对实验产生负面影响。

表 3.2: 实验结果

Low Density	Set	1	2	3	4	5
Our Approach	Macros	73	75	72	80	81
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	23	25	23	20	26
	Errors	6	7	6	7	7
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0
High Density	Set	6	7	8	9	10
Our Approach	Macros	47	51	53	48	44
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	9	7	7	8	10
	Errors	6	6	7	6	6
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0

“Macros” 行表示剩余的宏调用数量。“Errors” 行表示修改造成的错误数量 “Failure” 行表示反向预处理器发生了多少错误。

3.1.4 实验结果

我们实验的结果在表 3.2中。接着我们将会讨论本章开始时提到的三个研究问题的解答。

研究问题1

我们从表中可以看出，我们的方法尽可能地保存了宏调用。 Per-line方法保留了一些宏调用，而per-file方法，不出所料的，完全不保留宏调用。我们进一步考察了为什么per-line方法保存了这么少的宏调用。其中一个主要原因是我们发现在Linux内核代码中，时常出现一行中有多个宏调用的情况。而一旦修改发生在该行， per-line方法会破坏该行中的所有宏调用。

研究问题2

从表中看出，我们的方法和per-file方法都不会产生错误。但per-line方法产生了一些错误。我们认为这是因为存在一些宏调用形式是多行宏调用。这些宏调用

的参数一般是一个表达式，甚至一个句子，以至于一行中无法写完而需要换行。

研究问题3

正如之前讨论过的，我们的方法会在反向变换时对不合适的修改操作进行报错。这是因为有时一个修改会意外地在预处理后的代码中引入一个新的宏调用。这样就无法满足正确性性质的PUTGET性质。但是，在我们的实验中没有看到类似的情况。这是因为宏定义的名字之间差别比较大，而宏调用的位置在整个代码中很零散，通过调换字符位置或者拷贝并不能引入不合适的修改操作。同时我们知道另外两种方法并不能检测这种不合适的修改操作，所以在表 3.2关于报错的部分都留空了。

尽管出现不合适的修改操作而导致报错的情况在实际工作中也不常见，但理论上我们的方式是可以对这种情况作出报错的。

第四章　相关工作

4.1　Related Work

4.1.1　双向变换领域

我们工作的灵感来自于双向变换领域的研究。双向变换领域中一个经典的使用情况就是数据库设计中的 *view-update problem* [22–26]：一个域表示一个为一条查询指令计算好的数据库，研究工作主要关注于如何把在这个域上做的修改操作反向升级到源数据库中。这个工作就像模块转化在软件进化中一样重要 [27, 28]。

同时，针对这种双向变换的应用，学术界也设计了许多语言使得工作更加方便。其中广为人知晓的是 透镜理论框架 [19, 29–37]。在这套框架下涵盖了许多为双向变换提供连接因子的语言架构。另一种比较主流的思想是自动化地为现存的非双向变换的程序找到对应的反向程序。这类的研究称为 双向变换化*(Bidirectionalization)* [15–17, 38–44]。在软件工程的模块转换领域，需要被转换的数据通常以图（并非是树）的结构来保存。此时主流会使用一个关系型，而非函数型，的方法来描述不同模块之间的双向的不同的关系 [45–50]。但是，我们工作的要求现在并没有一种双向变换的技术可以满足。这是因为在我们的模型中，不仅作为数据的代码会出现变化，作为转换程序本身的宏、预处理指令，也会有相应的变化。因此，我们也为这类数据和转换程序都会变化的双向变换模型，提出了自己见解。

4.1.2　分析和修改未预处理的C代码

C的预处理器给静态程序分析带来了巨大的挑战。由于C的预处理器支持预处理变量，导致分析时，场景数量组合型快速增长。这使得传统的每次只处理

一个变量的程序分析方法变得不可行。也仅直到最近，通过*family-based analyses* [51–53] 的方法才能对未预处理的C代码进行可靠的解析与分析。之前的许多工具都时常出现不可靠的操作，或者只能限制在非常严格的使用情况下使用 [54–56]。

类似的，为了处理这种多变量的情况，在C代码重构的研究中学术界也花了很多功夫。大多数方法 [13, 57–59]都试图找到一个能够同时表示C语法和预处理器指令的合适模型。最近有一项工作 [60]提出了另外一种方法：为某一个变量做重构，但如果这对其他变量产生影响就阻止重构。这项工作是基于预处理变量修改相互之间往往影响很小的观测而设计出来的。我们可以看到，解决预处理前代码的分析现在并没有优秀的解决方案。

需要承认的是，我们的项目现在暂时也只考虑一个变量。在未来我们可能会把我们的工作和多变量的算法相结合。但是，即使只能处理一个变量，我们的项目在许多方面都十分有用：（1）许多实际工作中的程序，尽管有许多条件编译选项，没有许多预处理变量。（2）Overbey等人 [60]在他们的论文中指出，现实中修改一个变量往往不会对其他变量造成影响。

4.1.3 C预处理器上的实例调查

许多年来，学术界不乏各类对C预处理器十分有见解的学术实例调查工作 [8, 61, 62]。同时也有一些C预处理的替代品被提出，比如语法预处理器 [12, 63] 或者面向侧面开发方法 [64–66]。但是直至今日，业界并没有什么采用这类替代预处理器的迹象， C预处理器依然是主流的工具。这也说明C预处理器带来的麻烦依然广泛存在。

第五章　总结

5.1　Conclusion

为程序编辑工具处理C预处理器是十分困难的事情。因此，所以许多工具不是只能给出不可靠的结果或者完全放弃处理预处理器。本文中我们使用双向变换的方法把预处理器带来的麻烦从程序编辑工具中分离出来单独解决。这样程序编辑工具就能专注于预处理之后的代码，达到更模块化的设计效果。

C预处理器涉及到的问题也可以看作是转换系统问题中特殊的一类。在这类问题中，转换程序和数据可能同时被转换。这类问题还有类似双向变换的PHP [67]。现有的方法 [67]往往使用十分复杂且专门的方法并陷入双向变换性质的定义和正确性的讨论中去。本文所提出的方法对这一类的问题都有适用价值：该方法把转换程序当作数据，并在更高一层的操作语义上双向变换。我们也希望未来能看到一个通用的理论能够完美解决这一类问题。

参考文献

[1] C. Le Goues *et al.* "*GenProg: A generic method for automatic software repair*". *Software Engineering, IEEE Transactions on*, **2012**, *38*(1): 54–72.

[2] C. Le Goues *et al.* "*A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each*". In: *ICSE*, **2012**: 3–13.

[3] Y. Qi *et al.* "*The strength of random search on automated program repair*". In: *ICSE*, **2014**: 254–265.

[4] H. D. T. Nguyen *et al.* "*SemFix: Program repair via semantic analysis*". In: *ICSE*, **2013**: 772–781.

[5] J. Li *et al.* "*SWIN: Towards Type-Safe Java Program Adaptation between APIs*". In: *PEPM*, **2015**: 91–102.

[6] Y. Padioleau *et al.* "*Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers*". In: *PLOS*, **2006**.

[7] N. Meng, M. Kim and K. S. McKinley. "*Systematic Editing: Generating Program Transformations from an Example*". In: *PLDI*, **2011**: 329–342.

[8] M. D. Ernst, G. J. Badros and D. Notkin. "*An empirical analysis of C preprocessor use*". *Software Engineering, IEEE Transactions on*, **2002**, *28*(12): 1146–1170.

[9] E. Kohlbecker *et al.* "*Hygienic macro expansion*". In: *LISP and functional programming*, **1986**: 151–161.

[10] B. Lee *et al.* "*Marco: Safe, expressive macros for any language*". In: *ECOOP*. Springer, **2012**: 589–613.

[11] J. Korpela. *Using a C preprocessor as an HTML authoring tool*, **2000**.

[12] B. McCloskey and E. Brewer. "*ASTEC: A New Approach to Refactoring C*". In: *ESEC/FSE-13*, **2005**: 21–30.

[13] A. Garrido and R. Johnson. "*Embracing the C preprocessor during refactoring*". *Journal of Software: Evolution and Process*, **2013**, *25*(12): 1285–1304. http://dx.doi.org/10.1002/smr.1603.

[14] B. Fluri *et al.* "*Change distilling: Tree differencing for fine-grained source code change extraction*". *Software Engineering, IEEE Transactions on*, **2007**, *33*(11): 725–743.

[15] Ed. by R. Hinze and N. Ramsey. "*Bidirectionalization transformation based on automatic derivation of view complement functions*". In: *ICFP*. ACM, **2007**: 47–58.

[16] Ed. by Z. Shao and B. C. Pierce. "*Bidirectionalization for free! (Pearl)*". In: *POPL*. ACM, **2009**: 165–176.

[17] J. Voigtländer *et al.* "*Combining syntactic and semantic bidirectionalization*". In: *ICFP*, **2010**: 181–192.

[18] Ed. by A. D. Gordon. "*A Grammar-Based Approach to Invertible Programs*". In: *ESOP*. Springer, **2010**: 448–467.

[19] J. N. Foster *et al.* "*Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem*". *ACM Trans. Program. Lang. Syst.* 2007-05.

[20] ISO/IEC JTC 1, SC 22, WG 14. *Programming languages – C*, 2011-04.

[21] D. Kim *et al.* "*Automatic patch generation learned from human-written patches*". In: *ICSE*, **2013**: 802–811.

[22] F. Bancilhon and N. Spyratos. "*Update Semantics of Relational Views*". *ACM Trans. Database Syst.* **1981**, *6*(4): 557–575.

[23] U. Dayal and P. A. Bernstein. "*On the Correct Translation of Update Operations on Relational Views*". *ACM Trans. Database Syst.* **1982**, *7*(3): 381–416.

[24] Ed. by S. Abiteboul and P. C. Kanellakis. "*Foundations of Canonical Update Support for Closed Database Views*". In: *ICDT*. Springer, **1990**: 422–436.

[25] Y. Cui, J. Widom and J. L. Wiener. "*Tracing the Lineage of View Data in a Warehousing Environment*". *ACM Trans. Database Syst.* 2000-06: 179–227. http://doi.acm.org/10.1145/357775.357777.

[26] L. Fegaras. "*Propagating updates through XML views using lineage tracing*". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, 2010-03: 309–320.

[27] Ed. by M. Glinz, G. C. Murphy and M. Pezzè. "*Maintaining invariant traceability through bidirectional transformations*". In: *ICSE*. IEEE, **2012**: 540–550.

[28] Ed. by R. E. K. Stirewalt, A. Egyed and B. Fischer. "*Towards automatic model synchronization from model transformations*". In: *ASE*. ACM, **2007**: 164–173.

[29] Ed. by N. Heintze and P. Sestoft. "*A programmable editor for developing structured documents based on bidirectional transformations*". In: *PEPM*. ACM, **2004**: 178–189.

[30] Ed. by W.-N. Chin. "*An Algebraic Approach to Bi-directional Updating*". In: *APLAS*. Springer, **2004**: 2–20.

[31] Ed. by G. C. Necula and P. Wadler. "*Boomerang: resourceful lenses for string data*". In: *POPL*. ACM, **2008**: 407–419.

[32] Ed. by J. Hook and P. Thiemann. "*Quotient lenses*". In: *ICFP*. ACM, **2008**: 383–396.

[33] Ed. by C. Bolduc, J. Desharnais and B. Ktari. "*Gradual Refinement: Blending Pattern Matching with Data Abstraction*". In: *MPC*. Springer, **2010**: 397–425.

[34] Ed. by J. Whittle, T. Clark and T. Kühne. "*From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, **2011**: 304–318. http://dx.doi.org/10.1007/978-3-642-24485-8_22.

[35] M. Hofmann, B. Pierce and D. Wagner. "*Edit Lenses*". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Philadelphia, PA, USA: ACM, **2012**: 495–508. http://doi.acm.org/10.1145/2103656.2103715.

[36] Ed. by J. Gibbons. "*Three Complementary Approaches to Bidirectional Programming*". In: *SSGIP*. Springer, **2010**: 1–46.

[37] R. Rajkumar *et al. Lenses for Web Data*, **2013**.

[38] Ed. by M. M. T. Chakravarty, Z. Hu and O. Danvy. "*Incremental updates for efficient bidirectional transformations*". In: *ICFP*. ACM, **2011**: 392–403.

[39] J. Voigtländer *et al.* "*Enhancing semantic bidirectionalization via shape bidirectionalizer plugins*". *J. Funct. Program.* **2013**, *23*(5): 515–551.

[40] M. Wang *et al.* "*Refactoring pattern matching*". *Sci. Comput. Program.* **2013**, *78*(11): 2216–2242.

[41] Ed. by M. Felleisen and P. Gardner. "*FliPpr: A Prettier Invertible Printing System*". In: *Programming Languages and Systems*. Springer Berlin Heidelberg, **2013**: 101–120. http://dx.doi.org/10.1007/978-3-642-37036-6_6.

[42] Ed. by R. Peña and T. Schrijvers. "*Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem*". In: *PPDP*. ACM, **2013**: 297–308.

[43] Ed. by W.-N. Chin and J. Hage. "*Semantic bidirectionalization revisited*". In: *PEPM*. ACM, **2014**: 51–62.

[44] K. Matsuda and M. Wang. "*"Bidirectionalization for Free" for Monomorphic Transformations*". *Science of Computer Programming*, **2014**.

[45] I. (Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*.

[46] P. Stevens. "*Bidirectional model transformations in QVT: semantic issues and open questions*". *Software & Systems Modeling*, **2010**, *9*(1): 7–20. http://dx.doi.org/10.1007/s10270-008-0109-9.

[47] Ed. by E. Mayr, G. Schmidt and G. Tinhofer. "*Specification of graph translators with triple graph grammars*". In: *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, **1995**: 151–163. http://dx.doi.org/10.1007/3-540-59071-4_45.

[48] Ed. by H. Ehrig *et al.* "*15 Years of Triple Graph Grammars*". In: *Graph Transformations*. Springer Berlin Heidelberg, **2008**: 411–425. http://dx.doi.org/10.1007/978-3-540-87405-8_28.

[49] Ed. by P. Hudak and S. Weirich. "*Bidirectionalizing graph transformations*". In: *ICFP*. ACM, **2010**: 205–216.

[50] S. Hidaka *et al.* "*GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations*". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, **2011**: 480–483. http://dx.doi.org/10.1109/ASE.2011.6100104.

[51] C. Kästner *et al.* "*Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation*". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Portland, Oregon, USA: ACM, **2011**: 805–824. http://doi.acm.org/10.1145/2048066.2048128.

[52] P. Gazzillo and R. Grimm. "*SuperC: Parsing All of C by Taming the Preprocessor*". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, China: ACM, **2012**: 323–334. http://doi.acm.org/10.1145/2254064.2254103.

[53] J. Liebig *et al.* "*Scalable Analysis of Variable Software*". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg, Russia: ACM, **2013**: 81–91. http://doi.acm.org/10.1145/2491411.2491437.

[54] I. D. Baxter and M. Mehlich. "*Preprocessor Conditional Removal by Simple Partial Evaluation*". In: *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, **2001**: 281–. http://dl.acm.org/citation.cfm?id=832308.837146.

[55] A. Garrido and R. Johnson. "*Analyzing Multiple Configurations of a C Program*". In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, **2005**: 379–388. http://dx.doi.org/10.1109/ICSM.2005.23.

[56] Y. Padioleau. "*Parsing C/C++ Code Without Pre-processing*". In: *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. York, UK: Springer-Verlag, **2009**: 109–125. http://dx.doi.org/10.1007/978-3-642-00722-4_9.

[57] A. Garrido and R. Johnson. "*Challenges of Refactoring C Programs*". In: *Proceedings of the International Workshop on Principles of Software Evolution*. Orlando, Florida: ACM, **2002**: 6–14. http://doi.acm.org/10.1145/512035.512039.

[58] M. Vittek. "*Refactoring browser with preprocessor*". In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, 2003-03: 101–110.

[59] D. Spinellis. "*Global Analysis and Transformations in Preprocessed Languages*". *IEEE Trans. Softw. Eng.* 2003-11: 1019–1030. http://dx.doi.org/10.1109/TSE.2003.1245303.

[60] J. L. Overbey, F. Behrang and M. Hafiz. "*A Foundation for Refactoring C with Macros*". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, **2014**: 75–85. http://doi.acm.org/10.1145/2635868.2635908.

[61] H. Spencer and G. Collyer. *#ifdef Considered Harmful, or Portability Experience With C News*, **1992**.

[62] J. Liebig, C. Kästner and S. Apel. "*Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code*". In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. Porto de Galinhas, Brazil: ACM, **2011**: 191–202. http://doi.acm.org/10.1145/1960275.1960299.

[63] D. Weise and R. Crew. "*Programmable Syntax Macros*". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico, USA: ACM, **1993**: 156–165. http://doi.acm.org/10.1145/155090.155105.

[64] D. Lohmann *et al.* "*A Quantitative Analysis of Aspects in the eCos Kernel*". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. Leuven, Belgium: ACM, **2006**: 191–204. http://doi.acm.org/10.1145/1217935.1217954.

[65] B. Adams *et al.* "*Can We Refactor Conditional Compilation into Aspects?*" In: *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*. Charlottesville, Virginia, USA: ACM, **2009**: 243–254. http://doi.acm.org/10.1145/1509239.1509274.

[66] Q. Boucher *et al.* "*Tag and Prune: A Pragmatic Approach to Software Product Line Implementation*". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Antwerp, Belgium: ACM, **2010**: 333–336. http://doi.acm.org/10.1145/1858996.1859064.

[67] X. Wang *et al.* "*Automating presentation changes in dynamic web applications via collaborative hybrid analysis*". In: *FSE*, **2012**: 16.

致谢

pkuthss 文档模版最常见问题:

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在 □ 一年 / □ 两年 / □ 三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日