小 型 微 型 计 算 机 系 统 2008年 月 第 期
Journal of Chinese Computer Systems Vol.28 No. 2008

使用抽象语法树和静态分析的克隆代码自动重构方法

于冬琦[1]，彭 鑫[1]，赵文耘[1]

[1]（复旦大学计算机科学与工程系 软件工程实验室，上海 200433）

E-mail ：062021113@fudan.edu.cn, pengxin@fudan.edu.cn, wyzhao@fudan.edu.cn

摘 要：单个软件系统中以及若干个相似系统之间的代码克隆给软件维护增加了很大困难。本文针对运用克隆侦测发现的相似代码片断，提出了一种基于抽象语法树和静态分析的代码自动重构方法。该方法首先为克隆代码分别构造抽象语法树，然后运用语句差异度指标建立起语法树之间流程控制语句的对应关系。在此基础上，该方法根据控制流程和基本语句块两个层次上的差异性分析，最终通过代码可变点提取实现克隆代码的自动合并。我们针对 Java 代码开发了克隆代码重构支持工具原型，并分别针对 JDK1.5 和一个业务系统进行了自动重构实验。初步的结果表明，该方法能够准确、有效地辅助开发者实现克隆代码的自动重构。

关键词：可变点提取；代码克隆；抽象语法树；再工程；逆向工程

An Automatic Refactoring Method of Cloned Code Using Abstract Syntax Tree and Static Analysis

YU Dong-qi[1]，PENG Xin[1]，ZHAO Wen-yun[1]

[1]（*Department of Computer Science and Engineering , Fudan University, Shanghai 200433, China*）

Abstract: Code clone in single software system or several similar systems makes software maintenance very difficult. This paper offers an automatic refactoring method of detected cloned code based on abstract syntax tree and static analysis. At first, this method builds abstract syntax trees for cloned code separately. Then the difference between statements is used to establish the relationship between flow control statements in abstract syntax trees. Based on these steps, this method analyses the difference between flow control statements and the difference between simple statement blocks, and finally combines the cloned code through extracting the variation points in source code. We have developed a prototype tool which supports refatoring of cloned code of Java. And experiments on automatic refactoring are taken on JDK1.5 and a business system .The initial result shows that this method can assist developer achieve the goal of automatically refatoring of cloned code both accurately and effectively.

Key words: Variation point extraction, Code clone, Abstract syntax tree, Reengineering, Reverse engineering

代码克隆是指源代码文件中多个相同或相似的代码片断[1]。在软件开发和维护过程中，开发者经常会由于复制/粘贴式的代码复用或者设计思想以及编码习惯上相似性而引入克隆。因此，代码克隆大量存在于大型软件系统以及若干相似的软件系统中。例如，有学者[1]对 JDK 1.3.0 中所有的源代码文件进行了克隆侦测实验，发现有 40%多的代码文件和 20%多的代码行中存在不同程度的代码克隆。代码克隆产生的主要原因包括[2]：维护者经常采用复制粘贴原有代码并加以修改的方式开发相似的功能，他们相信这样做引入的代码错误较少；开发者脑海中对于相似功能实现方法的固有印象使得他们常常会在不同的地方写出相似的代码。

代码克隆在大多数情况下是有害的，它增加了软件系统

收稿日期：2008-- 基金项目：国家自然科学基金（60703092）、国家 863 计划（2006AA01Z189、2007AA01Z125）资助 作者简介：**于冬琦，**男，1984 年生，硕士生，研究方向为软件再工程；**彭鑫，**男，1979 年生，博士，讲师，CCF 会员，研究方向为软件产品线、软件体系结构、软件维护与再工程；**赵文耘，**男，1964 年生，教授、博导，CCF 高级会员，研究方向为软件工程、电子商务.

代码的长度，从而带来软件理解和软件维护的负担。如果许多克隆的代码分散在软件系统中的不同地方，那么修改一处代码就要求其他克隆的代码也要被修改，而如果文档得不到及时的更新，那么想要保持源代码的一致性就十分困难[1]。修改了一部分克隆代码而忽略了其他的克隆代码，这样引入的错误很可能在运行时才会得以显现。由此可见，代码克隆是对程序结构的一种破坏，减少代码克隆就能在一定程度上降低软件系统维护的负担，降低错误出现的几率。另一方面，随着特定领域软件开发的逐渐发展，越来越多的领域需要在遗产系统逆向工程和重构的基础上构建软件产品线。此时面临的一个主要问题是如何对多个应用产品中的相似代码单元进行重构，以获得具有领域可复用性的核心资产。

本文对克隆代码片断的自动重构方法进行了研究。很多情况下，克隆的代码段并不是完全相同的，而是存在一定的差异性。本文的方法在克隆侦测工具检测出的多个相似代码片断基础上，首先为克隆代码分别构造抽象语法树。然后我们着重考虑流程控制语句，如 if 语句，while 语句，for 语句等，运用语句差异度指标建立起语法树之间流程控制语句的对应关系。能够建立起对应关系的两个流程控制语句是克隆的或者近似克隆的。在此基础上，该方法根据控制流程和基本语句块两个层次上的差异性分析，抽取公共部分，同时对差异性部分通过参数提取的方式进行重构。在该方法基础上，我们针对 Java 代码开发了克隆代码重构支持工具原型，并分别针对 JDK1.5[9]和一个业务系统中的克隆代码进行了自动重构实验。初步的实验结果表明，该方法能够准确、有效地辅助开发者实现克隆代码的自动重构。

本文剩余部分将首先对克隆代码侦测和重构方面的一些相关工作进行探讨。接着，第 2 部分介绍基于抽象语法树和静态分析的克隆代码自动重构方法的主要过程，并在第 3 部分对代码可变性分析及提取的一些关键技术进行介绍。第 4 部分将介绍我们基于 Java 的重构工具原型，以及对 JDK1.5 和一个商业软件“网上考试报名系统”中的克隆代码进行的实验。第 5 部分结合实验结果，对本文方法的优缺点以及下一步的改进方向进行了探讨。最后，第 6 部分对全文进行了总结。

1 相关工作

在代码克隆侦测方面，已经有人提出了大量的方法，主要有基于标号（token）的方法和基于抽象语法树的方法等。基于标号的方法以Toshihiro Kamiya等[1]为代表，他们提出的侦测方法包括了将源文件进行一系列转换并进行token-by-token的比较。他们对此方法进行了优化并开发了一个叫做CCFinder的工具，该工具可以侦测C, C++, Java的源代码中的克隆。他们同时提出了度量克隆的一些手段。他们还在JDK等软件系统上面进行了一系列实验，定性与定量的评价了该方法的有效性。他们的方法着眼于克隆侦测，而没有对于侦测出来的克隆代码进行归并处理。而我们的方法着眼于基于语法树和克隆侦测的代码可变性提取。

基于抽象语法树的克隆侦测方法以Ira D. Baxter等[2]为代表，他们提出了一种简单实用的克隆侦测方法。他们开发了一个工具进行克隆侦测，该工具还能够为每处侦测到的克隆产生一个宏，并且用宏调用来替换原来的克隆代码。他们的方法不能侦测出由于插入或者删除一条语句而产生的近似的克隆，而我们的方法可以较好地处理这种情况。他们的方法只是简单的将克隆的代码用宏调用来替代，主要处理了传入函数的实参等简单的可变点。

面向可变性的克隆代码重构方面相关工作还不多，在静态分析方面，Yoshiki Higo等[4]提出了一种从面向对象软件系统中移除代码克隆的方法。他们使用已有的两种重构模式：抽取方法和将子类中的方法向上提取至父类中。他们开发了一个工具，这个工具利用了克隆侦测工具CCFinder和克隆分析环境Gemini来寻找可以进行重构的克隆代码。他们的方法对于在克隆代码中重命名的变量等有较好的处理。然而，CCFinder是基于token的克隆侦测工具，它只能侦测出语句结构完全相同的代码克隆片段，而不能侦测出由于插入或者删除一条语句而产生的近似的克隆。同时，考虑到某些在克隆代码中引用到的变量并不是在克隆代码中定义的，他们的方法并不保证能够移除所有侦测出来的克隆。相比他们的方法，我们的方法能够较好地处理克隆代码段中语句结构上的差异。

在动态分析方面，Bas Cornelissen等[5]提出了一种动态地检测软件系统中类似版本之间共性和可变点的方法。他们通过检测不同版本执行轨迹之间的分支和汇合来定位软件系统中任何抽象级别的可变点。这种动态检测的方法不可避免的具有一定的不精确性。他们只是检测出可变点，而没有对可变点进行归并处理。而我们采用的是静态分析的办法，我们寻找并且归并语句级的可变点。

2 方法概述

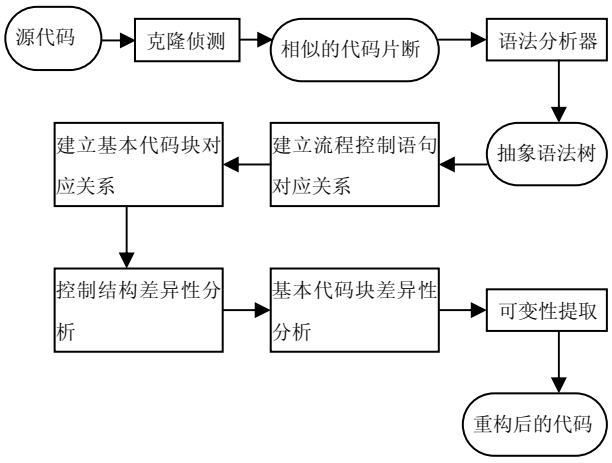

图 1: 方法概览

Fig1:Overview of the Method

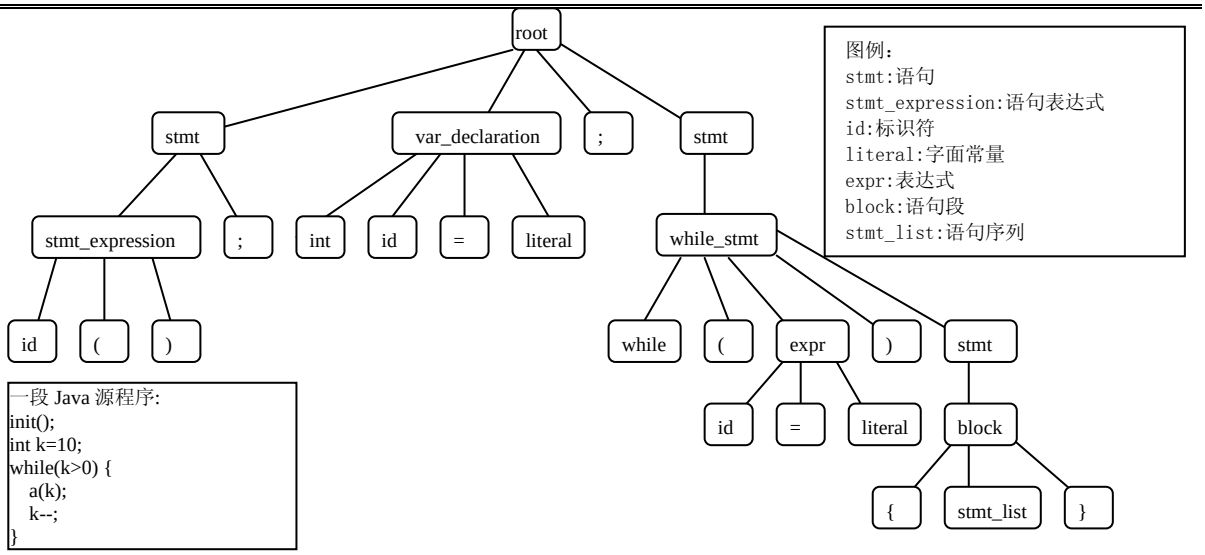

图 2: 抽象语法树示例

Fig2: An Example of Abstract Syntax Tree

本文的方法建立在代码克隆侦测工具识别出的克隆代码片断基础上，最终将在对克隆代码片断的差异性分析和可变性提取的基础上得到重构后的代码。整个方法的基本过程如图 1 所示，主要分为 6 个步骤，分别介绍如下。

- 克隆侦测。我们利用某种克隆侦测工具检测出源代码中多个相似的代码片断。例如我们在工具实现中所使用的克隆侦测工具 Simian[6]。
- 生成抽象语法树。我们使用 Java Tree Builder(JTB)[7]和 JavaCC[8]为某种程序语言生成语法分析器，运用该语法分析器对源代码进行分析，从而为每段克隆代码生成一个抽象语法树。
- 建立起流程控制语句之间以及基本代码块之间的对应关系。
- 控制结构差异性分析。即在第三步的基础上标识出建立起对应关系的流程控制语句内部的克隆。
- 基本代码块差异性分析。即利用 Simian 侦测基本代码块之间的克隆。
- 可变性提取。即根据第四步和第五步标识出的克隆代码，针对除了克隆代码之外的每一处可变点，插入 if 语句和控制变量，将若干段代码合并为一段。

3 代码可变性分析及提取

3.1 抽象语法树生成

抽象语法树是将源代码经过词法分析和语法分析后的产物。例如，图 2 示例了一段 Java 源程序和其对应的抽象语法树：

抽象语法树全面反映了源代码的语法结构，它的叶子节点是标识符，字面常量等。而我们的方法则主要关注于抽象语法树中的流程控制语句。如果得到了源程序的抽象语法树，我们就能很容易地对于源程序的语法结构进行深入地分析，就能很容易地识别出每一个诸如 if 语句，while 语句，for 语句等流程控制语句的内部语法结构，从而为我们方法的后续步骤带来很大方便。

下面，本文就以重构两个相似的 Java 方法为例来说明我们的方法。本文方法这一步，我们将一个 Java 语法描述文件交给 JTB 修改，它会将生成抽象语法树的代码插入到语法描述文件中。然后我们将修改过的语法描述文件交给 JavaCC 生成一个语法分析器。这个语法分析器在进行语法分析的同时就会生成源程序的抽象语法树。我们将两个相似的 Java 方法输入到语法分析器中进行分析，就可以产生两个抽象语法树。在本文方法的后续步骤中，我们将利用这两个抽象语法树进行分析和处理。

3.2 代码块对应关系建立

在方法的这一步中，我们首先将代码划分为代码块，然后根据语句差异度指标建立起流程控制语句之间的对应关系，接着我们建立剩余的基本代码块之间的对应关系。

这里首先要说明什么是简单语句。我们可以把 Java 语言的语句分为两大类，第一类：流程控制语句，诸如 if 语句，for 语句，while 语句等。这些语句对于程序控制流有很大的影响，它们是建立抽象语法树之间控制流对应关系过程中所需要考虑的关键语句。第二类：除了流程控制语句之外的语句，我们将这种语句称为简单语句，诸如变量定义语句 int i=10; 赋值语句 i=j+k;以及方法调用语句 myMethod(k);等。简单语句在建立抽象语法树之间控制流对应关系过程中只起到次要作用，在方法的第五步中，我们把这些语句交给克隆侦测工具 Simian 处理。

下面，我们说明两个语句之间的差异度的三种取值。对于语句 a 和语句 b，它们的差异度 c 有三种取值：0：表示 a

和 b 没有差异；1：表示 a 和 b 有“一些”差异；2：表示 a 和 b 差异很大，认为两个语句完全不同。接下来我们定义相同种类语句的概念：

定义 1. 相同种类语句：对于语句 a 和语句 b，则有：

1. 如果 a 为流程控制语句且 b 也为流程控制语句且 a 和 b 是同种类型的流程控制语句，则定义语句 a 和语句 b 为相同种类的语句。

2. 如果 a 为简单语句且 b 也为简单语句，则定义语句 a 和语句 b 为相同种类的语句。

现在我们具体地描述两个语句之间差异度的计算方法。首先，对于语句 a 和语句 b，如果 a 和 b 不是相同种类的语句，比如，a 为 while 语句，b 为简单语句，或者 a 为 while 语句，b 为 for 语句等等，那么就定义 a 和 b 的差异度为 2。然后，对于相同种类的语句，下面以 while 语句，for 语句，if 语句以及简单语句为例，定义它们的差异度计算方法。

定义 2. 两个 while 语句 a,b 的差异度：

1. 如果 a 和 b 的循环控制条件不同，则 a 和 b 的差异度为 2。

2. 如果 a 和 b 的循环控制条件相同，则 a 和 b 的差异度就定义为它们循环体中语句的差异度。

例如，图 3 中 while 语句的循环体均为一个语句段（block），这里我们还需要定义两个语句段 block 的差异度。在定义 block 的差异度之前，我们首先定义两个 block 的差异百分比 percentC：

定义 3. 两个语句段（block）的差异百分比 percentC：

$$\text{percentC} = 1 - \frac{\text{差异度为0的语句数量}}{\text{语句的总数量}}$$

假设 thresholdC 为我们预先设定的一个语句差异度阈值，比如 0.5，那么就有如下定义：

定义 4. 两个语句段（block）的差异度：

1. 如果 percentC>thresholdC, 则两个语句段（block）的差异度为 2

2. 如果 0<percentC<=thresholdC, 则两个语句段（block）的差异度为 1

3. 如果 percentC=0，则两个语句段（block）的差异度为 0

其中 thresholdC<=1 且 thresholdC>=0

例如，图 3 中两个 while 语句的循环控制条件均为(k>0)，则需要进一步判断循环体中两个 block 的差异度。在 while 语句的循环体中，要借助 Simian 侦测其中的简单语句克隆，侦测出的克隆的简单语句的差异度定义为 0。在上面的例子中，借助 Simian，我们可以得到只有 a 中循环体里的 d(k)； 与 b 中循环体里的 c(k);这两个语句是不同

while 语句 a：
while(k>0)
{
a(k);
b(k);
d(k);
k--;
}
	while 语句 b：
	while(k>0)
	{
	a(k);
	b(k);
	c(k);
	k--;
	}

for 语句 a：
for(int p=0;p<10;p++)
{
first(p);
second(p);
third(p);
fourth(p);
fifth(p);
sixth(p);
}
	for 语句 b：
	for(int p=0;p<10;p++)
	{
	doFirst(p);
	doSecond(p);
	third(p);
	fourth(p);
	doFifth(p);
	doSixth(p);
	}

if 语句 a：
if (good)
{
first(p);
second(p);
third(p);
fourth(p);
fifth(p);
sixth(p);
}
else if(bad)
{
terminate();
}
	if 语句 b：
	if(good)
	{
	first(p);
	second(p);
	third(p);
	fifth(p);
	sixth(p);
	}
	else if(bad)
	{
	terminate();
	}

图 3：流程控制语句示例

Fig3:Example of Flow Control Statements

的。那么 percentC=2/8=0.25。如果我们预先设定 thresholdC=0.5，那么由于 0<0.25<=0.5， 最后可以得出两个语句 a 和 b 的差异度 c 为 1，就是说语句 a 和 b 基本相同，只存在少部分差异。对于这种差异度为 1 的语句，到后面我们会把循环体内部各处不同的代码分别视为可变点，而不是将整个 while 语句视为一个可变点。

定义 5. 两个 for 语句 a,b 的差异度：

1. 如果 a 和 b 的括号内的循环初始化或者循环控制条件或者循环更新语句三者之一不同，则 a 和 b 的差异度为 2。

2. 如果 a 和 b 的括号内的循环初始化以及循环控制条件以及循环更新语句三者均相同，则 a 和 b 的差异度就定义为它们循环体中语句的差异度。

例如，对于图 3 中两个 for 语句 a 和 b，它们括号内的循环初始化，循环控制条件等均为(int p=0;p<10;p++)，则继续判断两个语句的循环体。与上例类似，在本例中，借助 Simian，我们可以得到只有 a 中循环体里的 third(p);fourth(p); 与 b 中循环体里的 third(p);fourth(p); 这两组语句是分别相同的。那么 percentC=8/12=0.67。如果我们预先设定 thresholdC=0.5，那么由于 0.67>0.5， 最后可以得出两个语句 a 和 b 的差异度 c 为 2，就是说语句 a 和 b 差异很大，我们认为它们完全

不同。对于这样的语句，到后面我们会把它们作为一个整体视为可变点，而不再将循环体内部的各处不同的代码分别视为可变点。

下面，我们来看两个 if 语句之间的差异度计算。首先，这里约定 if 语句的语法结构为：
if(condition)statementUpper[else statementLower]，这里的 statementUpper 和 statementLower 均可以是任意的语句，其中的 else 部分是可选的。图 3 中的两个 if 语句的 statementUpper 是由花括号括起来的一个 block，statementLower 又是一个 if 语句。

定义 6. 两个 if 语句 a,b 的差异度：

1. 如果 a,b 两个 if 语句的条件 condition 不相同，那么 a,b 的差异度为 2
2. 如果 a,b 两个 if 语句的条件 condition 相同，则有：
   a. 如果 a，b 两个 if 语句都不包括 else 部分，那么定义 a,b 的差异度等于两个 statementUpper 的差异度
   b. 如果 a，b 两个 if 语句有且仅有一个包括 else 部分，那么定义 a，b 的差异度等于 2
   c. 如果 a，b 两个 if 语句都包括了 else 部分，那么还有如下规定：如果 a，b 的 statementUpper 的差异度为 0 且 statementLower 的差异度也为 0，那么我们定义 a，b 的差异度为 0；如果 a，b 的 statementUpper 的差异度为 2 且 statementLower 的差异度也为 2，那么定义 a，b 的差异度为 2；其余情况均定义 a，b 的差异度为 1

例如，对于图 3 中两个 if 语句 a 和 b，它们的条件部分相同，均为（good）且 a，b 均包含了 else 部分，而 else 部分又是一个 if 语句。可以得出 a，b 的 statementUpper 中 block 语句的差异百分比为 0<1/11<0.5,差异度为 1。statementLower 中 block 语句的差异百分比为 0/2=0,差异度为 0。于是可以得出 a，b 两个 if 语句的差异度为 1。

最后，对于除了流程控制语句之外的简单语句，我们把它们交给 Simian 判断它们是否为克隆，如果 Simian 判断简单语句 a,b 为克隆，那么就定义 a 和 b 的差异度为 0，否则定义 a 和 b 的差异度为 2。

综上所述，计算两个语句差异度的算法伪代码如图 4 所示：

定义好了任意两个语句的差异度，我们就可以建立起两个抽象语法树之间流程控制语句的对应关系了。以图 6 中的两个 Java 方法体 v1 和 v2 为例：

首先我们将连续的简单语句视为一个基本代码块，而将每个 while 语句，for 语句，if 语句等流程控制语句各自视为单独的代码块，图 6 中的每一个方框为一个代码块。

```
int compare(statement a, statement b)
{
  if (a,b 非相同种类语句){
    retrun 2;
  }else if(a,b 均为简单语句){
    if(a,b 为克隆语句){
      return 0;
    }else{
      retrun 2;
    }
  }else if(a,b 均为 while 语句){
    if(a,b 的循环条件相同){
     return a,b 循环体的差异度;
    }else{
     retrun 2;
    }
  }else if(a,b 均为 for 语句){
    if(a,b 循环初始化及循环控制条件及循环更新语句均
          相同){
     return a,b 循环体的差异度;
    }else{
     retrun 2;
    }
  }else if(a,b 均为 if 语句){
    if(a,b 的判断条件相同){
     if(a,b 有且仅有一个包含 else 部分){
      return 2;
     }else if(a,b 均包含 else 部分){
      if(a,b 的 statementUpper 差异度为 0
          且 statementLower 差异度也为 0){
       return 0;
      }else if(a,b 的 statementUpper 差异度为 2 且
            statementLower 差异度也为 2){
       return 2;
      }else{
       retrun 1;
      }
     }
    }else{
     retrun 2;
    }
  }
}
```

图 4：计算两个语句差异度的算法

Fig4: Algorithem of Computing the Difference between Two Statements

然后就要为 v1 中每一个流程控制语句在 v2 中找一个相同类型的流程控制语句与其对应，寻找的标准就是两个流程控制语句的差异度 c<=1。对于没有在另一段代码中找到对应语句的流程控制语句，我们将其整体视为可变点，在方法的最后一步对其进行提取。对于剩余的基本代码块，我们简单地把它们对应起来，在方法地第五步我们将要把它们交给 Simian 进行克隆侦测。

图 6 中，两个代码块之间的直线表示它们的对应关系，而显示为斜体的代码是没有找到对应语句的流程控制语句。

3.3 控制结构差异性分析

在方法的这一步，我们要标识出两个对应的流程控制语句内部的克隆。对于在上一步得到的两个对应的流程控制语句，由于在上一步计算它们相似性的时候已经利用 Simian 对于它们的循环体（或者是 if 语句的 statementsUpper 与 statementsLower）进行了克隆侦测，这一步只需利用上一

步保存的结果标识出克隆的部分即可。

而对于在上一步未能配对成功的流程控制语句，我们在方法的最后一步会将其整体视为可变点进行提取。

3.4 基本代码块差异性分析

在方法的这一步，我们把在方法的第二步中对应起来的基本代码块交给 Simian 进行克隆侦测，根据侦测的结果标识出克隆的代码。

对于在第二步中未能配对成功的流程控制语句，也要和临近的简单语句块一起送交 Simian 进行克隆侦测，然后需要针对这种失配的流程控制语句对于克隆侦测结果进行调整：如果侦测出来的克隆语句出现在失配的流程控制语句中，那么这个克隆语句我们视为无效。图 5 是一个例子，我们将以下两个代码块 a,b 送给 Simian 进行克隆侦测，侦测出来的克隆代码为加重显示部分：

代码块 a:	代码块 b:

int j=10;
initA();
initB();
initC();
while(j>0)
{
a(j);
b(j);
j--;
}
k=10;
	int j=10;
	initA();
	initB();
	initC();
	for(j=10;j>0;j--)
	{
	a(j);
	b(j);
	}
	k=10;

图 5: 基本代码块差异性分析

Fig5: Difference Analysis of Basic Code Blocks

Simian 将 a 中 while 循环体中的两个语句 a(j);b(j); 和 b 中 for 循环体中的两个语句 a(j);b(j);视为克隆，这种克隆对于我们是无效的，因为它们分别处于两种不同类型的流程控制语句中，这种克隆对于我们提取代码可变点没有意义。在第二步中，a 中这个 while 循环和 b 中的这个 for 循环都是失配的流程控制语句，在我们的方法中，控制流结构是优先的。

对于我们例子中的两段代码 v1 和 v2，经过控制结构的差异性分析和基本代码块的差异性分析之后，就有图 6 所示的结果，图中加重的部分为有效的克隆代码：

3.5 可变性提取

在方法的这一步，我们根据上两步标识出的克隆代码，针对除了克隆代码之外的每一处可变点，插入 if 语句和控制变量，将两段代码合并为一段。合并的结果如图 7 所示。两段代码 v1 和 v2 共有 4 处可变点，用四个 if 语句和控制变量 arg1 至 arg4 分别控制。注意到，由于 v1 和 v2 中的两个 for 语句循环体只有少部分相同，大部分不同，差异度为 2，我们就将 for 语句整体视为可变点，这样做的好处是减少了一些不必要的可变点的数量，也就减少了控制变量的个数。

4 工具实现及实验

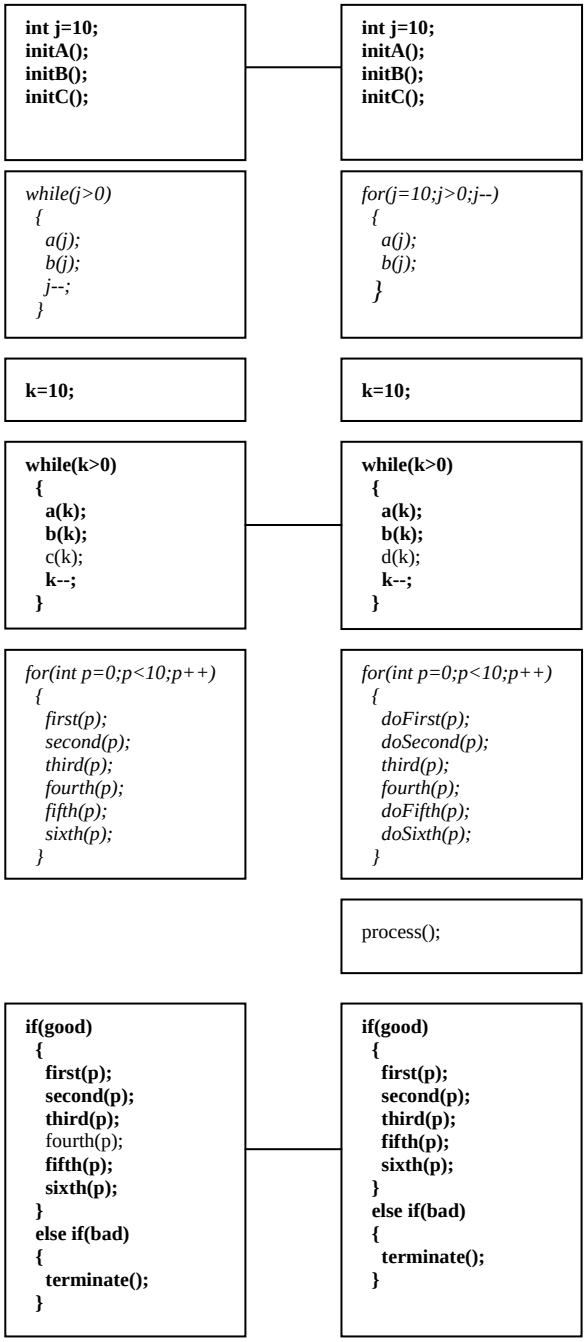

图 6: 两个 Java 方法体 v1,v2

Fig6: Two Java Method Bodies v1,v2

4.1 原型工具

我们使用 JTB 来生成抽象语法树 AST。JTB 与 JavaCC 一起使用，它可以为 JavaCC 语法描述文件中的每一条产生式中的每一个非终结符生成相应的一个 Java 类。它还改写了 JavaCC 的语法描述文件，以便语法分析器在进行语法分析的同时，能够生成一棵抽象语法树。我们将此文件输入 JavaCC，就会得到一个语法分析器 JavaParser。我们再将一段代码作为该语法分析器的输入，就会得到一棵抽象语法树的根节点。然后我们就可以在这个语法树上进行一系列操作。

```
result(int arg1,int arg2,int arg3, int
arg4)
{
 int j=10;
 initA();
 initB();
 initC();
 if (arg1==0)
 {
    while(j>0)
    {
      a(j);
      b(j);
      j--;
    }
 }
 else if (arg1==1)
 {
    for(j=10;j>0;j--)
    {
      a(j);
      b(j);
    }
 }
 k=10;
 while(k>0)
 {
  a(k);
  b(k);
  if (arg2==0)
  {
     c(k);
  }
  else if (arg2==1)
  {
     d(k);
  }
  k--;
 }
```

```
//(接左边)
if (arg3==0)
{
 for(int p=0;p<10;p++)
 {
   first(p);
   second(p);
   third(p);
   fourth(p);
   fifth(p);
   sixth(p);
 }
}
else if (arg3==1)
{
 for(int p=0;p<10;p++)
 {
   doFirst(p);
   doSecond(p);
   third(p);
   fourth(p);
   doFifth(p);
   doSixth(p);
 }
 process();
}
if(good)
{
   first(p);
   second(p);
   third(p);
   if (arg4==0)
   {
      fourth(p);
   }
   fifth(p);
   sixth(p);
}
else if(bad)
{
   terminate();
}

}
```

图 7：可变性提取结果

Fig7: Result of Variation Extraction

本文使用 Simian 来侦测简单代码块中的克隆，Simian 可以检测出 Java，C#，C，C++，COBOL 等编程语言源文件中的重复代码，具有可变程接口和多种选项。

我们开发了一个原型工具，该系统支持输入两个相似的 Java 方法，并可设定语句差异度阈值 thresholdC，系统的输出是合并后的 Java 方法，（如图 8 所示）

4.2 实验 1：JDK1.5

Sun 公司提供的 Java 开发包 JDK1.5 中存在大量的相似代码，我们首先选取了 JDK1.5 中 java.math 包中的大量的相似的方法进行了实验，选取 java.math 包是因为里面的方法都是与数学计算有关，我们容易读懂这些代码，从而容易判断实验结果的好坏。图 9 是一个实验结果示例：图中的方法 1 和方法 2 是 java.math 包中 BigInteger 类的两个方法，BigInteger 类用来表示任意精度的整数。setBit 和 clearBit 都是用来对 BigInteger 进行位操作的方法。其中

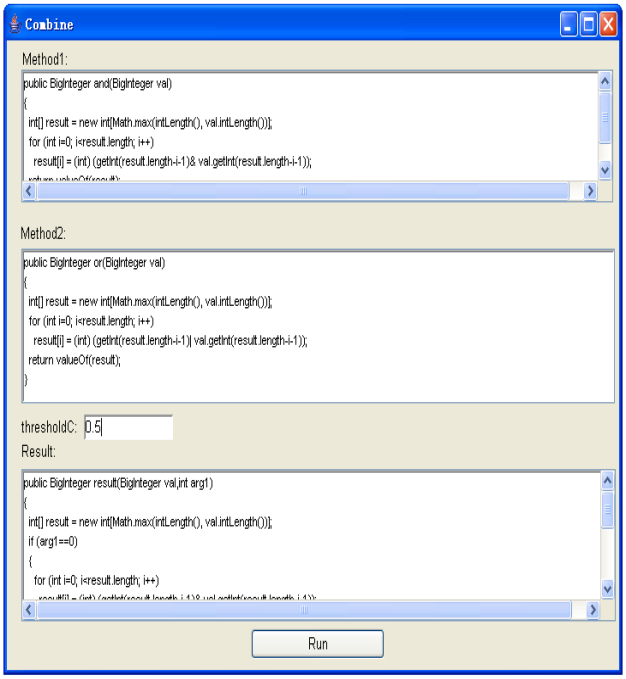

图 8：原型工具

Fig8: Prototype Tool

setBit 方法将 BigInteger 的二进制表示中第 n 位置为 1，而 clearBit 方法将 BigInteger 的二进制表示中第 n 位置为 0。这两个方法的处理逻辑是相似的，可以通过添加参数来重构为一个方法。容易看出，上述实验中，我们的原型系统提取的可变点是比较准确的。

4.3 实验 2：网上报考系统

网上考试报名系统是一个基于 Web 的公共考试课程报名系统，主要功能包括课程选择、报考费用支付、报考查询等。通过克隆侦测工具 Simian 的检查，我们发现该系统代码中存在超过 70 处克隆代码片断。经系统开发者确认，其中部分代码克隆是由于相似功能单元编码时的代码复制粘贴造成的，其余一些则是由于一些编码习惯造成的（例如在处理业务信息查询时的一些固有编程模式）。我们选取了其中两个关于账单处理的 Java 方法，相似的代码片断出现在两个不同的地方。由于文档的缺失，以及对于方法命名的随意性，开发者已经回忆不出它们的区别。经过我们原型系统的分析，得出了两个方法的 3 个可变点。我们将结果交给开发者评价，开发者很快地回忆起了两者的区别，并认为我们的原型系统很准确地提取了两者的可变点。

图 10 和图 11 分别是合并后 Java 方法的经过精简的源代码以及相应的经过简化的抽象语法树，源代码中的省略号“...”以及抽象语法树中标识“...”的结点表示省略了的代码部分。在抽象语法树中，作为示例，第 1 处可变点中 block 节点所对应的源代码被详细地标识出来。

方法1:

```
public BigInteger setBit(int n)
{
  if (n<0)
    throw new ArithmeticException("Negative bit address");
  int intNum = n/32;
  int[] result = new int[Math.max(intLength(), intNum+2)];
  for (int i=0; i<result.length; i++)
    result[result.length-i-1] = getInt(i);
  result[result.length-intNum-1] |= (1 << (n%32));
  return valueOf(result);
}
```

方法2:

```
public BigInteger clearBit(int n)
{
  if (n<0)
    throw new ArithmeticException("Negative bit address");
  int intNum = n/32;
  int[] result = new int[Math.max(intLength(), (n+1)/32+1)];
  for (int i=0; i<result.length; i++)
    result[result.length-i-1] = getInt(i);
  result[result.length-intNum-1] &= ~(1 << (n%32));
  return valueOf(result);
}
```

重构结果:

```
public BigInteger result(int n, int arg1, int arg2)
{
  if (n<0)
    throw new ArithmeticException("Negative bit address");
  int intNum = n/32;
  if (arg1==0){
    int[] result = new int[Math.max(intLength(), intNum+2)];
  }
  else if (arg1==1){
    int[] result = new int[Math.max(intLength(),
                                    (n+1)/32+1)];
  }
  for (int i=0; i<result.length; i++)
    result[result.length-i-1] = getInt(i);
  if (arg2==0){
    result[result.length-intNum-1] |= (1 << (n%32));
  }
  else if (arg2==1){
    result[result.length-intNum-1] &= ~(1 << (n%32));
  }
  return valueOf(result);
}
```

图 9: 实验结果示例

Fig9: Example of Experiment Result

5 讨论

5.1 代码重构中的参数命名

由于我们的方法目前仅从语法层面进行分析,所以对于控制变量的命名采用的是一种简单的命名方式，即命名为:arg1,arg2,arg3 等等。这样的名称本身并没有表达出该控制变量控制的内容,有时候会带来对于程序理解上面的困难。而如果要能够对这些变量自动赋予让人容易理解的名字,则需要综合考虑该控制变量所在的程序上下文语义信息等，这将是一项非常复杂的工作。

5.2 语句差异性指标

在计算由花括号括起来的某两个语句段(block)的相似度的时候，我们使用的是差异百分比 percentC 这个指标。

结果:

```
public static void result(String bNo,String bdate,String amt,boolean isSucc
      ,String ip,int arg1,int arg2,int arg3) throws SQLException {
  …
  if(isSucc){
     …
     if(rs.next()){
        …
        if (arg1==0){
         sql = "update billInfo set state='5',operator='Z' where billno = '"
           +bNo+"' and billdate='"+bdate+"'";
        }
        else if (arg1==1){
           sql = "update billInfo set state='5' where billno = '"
           +bNo+"' and billdate='"+bdate+"'";
        }
        …
        if (arg2==1){
           String billLog = "insert into log (LOGNO, TYPE, UNINO"
           +", NAME, DESP, SPEC1, SPEC2, LOGTIME, IP)"
           +"values(logno.nextval,'11','"+uniNo+"','"+sname
           +"','成功处理帐单信息','"+bdate+":"+bNo+"','"
           +orderDate+":"+orderNo+"',sysdate,'"+ip+"')";
            st2.executeUpdate(billLog);
        }
        …
     }
     else{
        …
     }
  }
  else {
     …
     if(rs.next()){
        if(arg3==0){
         String unConfirmedBillSql = "update billInfo"
           +" set state='1',operator='Z' where billno = '"
           +bNo+"' and billdate='"+bdate+"'";
        }
        else if(arg3==1){
           String unConfirmedBillSql = "update billInfo"
           +" set state='1' where billno = '"
           +bNo+"' and billdate='"+bdate+"'";
        }
         …
     }
  }
  …
}
```

图 10: 网上报考系统实验结果源代码示例

Fig10: Example of Source Code of an Online Examination Entering System

当 percentC 超过某个预先设定的阈值的时候，我们就认为这两个语句段完全不同，从而将整个语句段视为一个可变点，而不再深入到语句段内部定位各个可变点。这样做的局限性在于忽略了语义层面的信息。当一个程序员判断某两个语句段是否应该整体作为可变点的时候,他会综合考虑程序上下文的语义信息,所以就有可能产生即使 percentC 很大，也不该把语句段整体视为可变点的情况。这种判断结合了语义层面的因素，而我们的方法没有考虑程序语义的信息，所以目前只能利用一个阈值来控制。

5.3 语句顺序对于重构的影响

由于我们的方法没有考虑程序语义的信息,对于可以互换顺序的一些语句的处理可能会导致产生冗余的控制变量。比如，在比较 int i=0; int j=0; 与 int j=0; int i=0; 的时候，我们的方法会识别出两个可变点，合并的结果如图 12 所示：

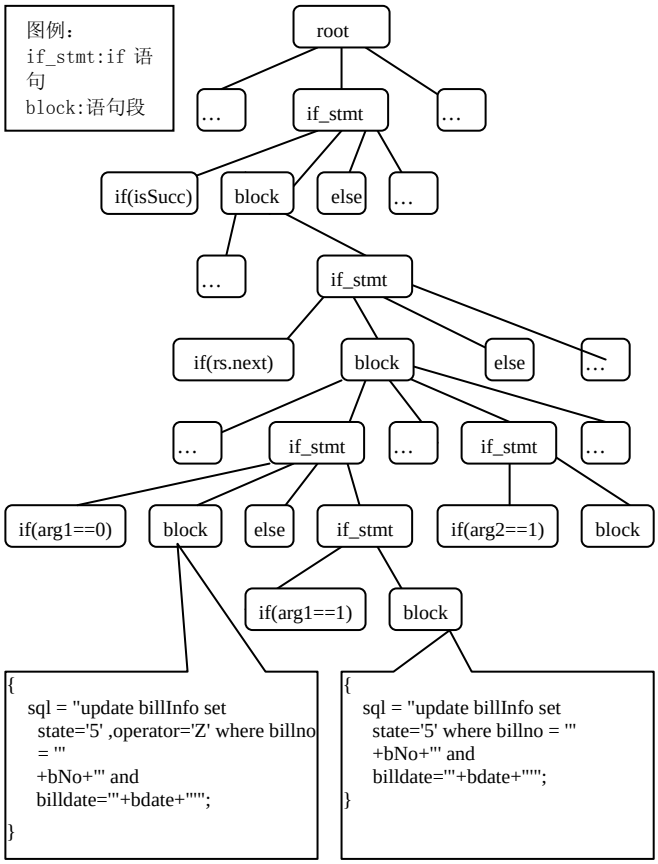

图 11：网上报考系统实验结果的抽象语法树

Fig11: Abstract Syntax Tree of the Result of the Experiment on an Online Examination Entering System

```
if (arg1=1){
  int j=0;
}
int i=0;
if (arg2=0){
  int j=0;
}
```

图 12：一个重构结果的例子

Fig12: An Example of the Result of Refactoring

容易看出，这两条定义变量的语句是可以互换顺序的，这里并不存在语义层面上的可变点，从这个角度来说， 这两个控制变量是冗余的。

所以，我们未来的工作可以从程序语义层面入手，给出自动命名的控制变量的上下文信息，以辅助程序员对其命名。同时，通过考虑上下文语义信息，优化语句之间差异度的计算方法。还有，通过分析程序中变量的引用链，从而消除一些冗余的可变点。

6　总结和展望

针对大型软件系统中普遍存在的代码克隆现象，以及特定领域开发中多个遗产系统变体中大量存在的相似代码片断，本文提出了一种基于抽象语法树和静态分析的代码自动重构方法。该方法通过控制流程以及基本代码块两个层面上的共性和差异性分析进行代码重构，抽取公共代码片断，并通过参数提取将差异性部分归纳为统一的可变性代码。在该方法基础上，我们实现了一个基于 Java 的克隆代码自动重构工具，并分别针对 JDK1.5 和一个商业软件系统进行了实验，结果表明该方法提供的差异性分析结果以及重构结果能够有效地实现克隆代码重构。进一步的研究工作主要包括对重构过程中的可变性参数进行优化和整合，并在数据可变性以及可变性参数命名上提供更多的自动或半自动支持。

References:

[1] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, JULY 2002, VOL.28, NO.7: 654-670.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura et al. Clone Detection Using Abstract Syntax Trees. Proceedings of the International Conference on Software Maintenance, 1998, 368-377.

[3] R. Komondoor, S. Horwitz. Using slicing to identify duplication in source code. Proceedings of the 8th International Symposium on Static Analysis, 2001, 40–56.

[4] Y Higo, T Kamiya, S Kusumoto et al. Refactoring Support Based on Code Clone Analysis. Proceedings of 5th International Conference on Product Focused Software Process Improvement, April 2004, 220-233.

[5] Bas Cornelissen, Bas Graaf, Leon Moonen. Identification Of Variation Points Using Dynamic Analysis. Proceedings of First International Workshop on Reengineering towards Product Lines, November 2005, 9-13.

[6] Simian, http://www.redhillconsulting.com.au/products/simian/, Accessed October 2007

[7] JTB, http://compilers.cs.ucla.edu/jtb/, Accessed October 2007

[8] JavaCC, https://javacc.dev.java.net/, Accessed October 2007

[9] JDK1.5, http://java.sun.com/javase/downloads/index.jsp, Accessed March 2008