北京大学

本科生毕业论文

题目：动态变异测试的设计与实现

Title: **Design and Implementation of Dynamic Mutation Analysis**

姓　　名：史杨勍惟

学　　号：1200012741

院　　系：信息科学技术学院

专　　业：计算机科学

研究方向：软件工程

导师姓名：熊英飞

二〇一六年五月

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则一旦引起有碍作者著作权之问题，将可能承担法律责任。

摘要

变异测试是一种通过细节改变源代码的软件测试方法，用来帮助测试者评估测试集的质量。变异测试的一个重要瓶颈在于其可扩展性。研究人员已经提出了各种不同的变异测试加速技术，这些方法的本质是是移除变异测试中的冗余部分。然而，这些技术都是静态的，所以无法消除变异体执行过程中的冗余部分。

本论文的目标是设计并实现一个动态变异测试技术：在变异测试的执行过程中对变异体进行动态分析，并且仅在变异体将会产生新的系统状态的时候，通过创建新的进程来执行变异体。本论文提出了一个可以用来实现动态编译测试的抽象模型，此模型支持不同变异算子的动态变异测试。本论文在 LLVM 的框架上实现了一个 C 语言变异测试加速工具 AccMut。我们的实验表明动态变异测试技术可以在 Major Framework（目前最快的静态变异测试加速技术）的基础上进一步加速变异测试，加速比达到 2.22X。

在本论文的最后，我们将我们的模型扩展到了软件产品线测试上。

关键词：变异测试，程序分析，软件产品线

Design and Implementation of Dynamic Mutation Analysis

Yangqingwei Shi Computer Science
Directed by Prof. Yingfei Xiong

Abstract

Mutation analysis is used to help evaluating the quality of existing software tests by modifying a program in small ways. One important bottleneck of mutation analysis is its scability. Researches have proposed different techniques to accelerate the mutation analysis, such as removing redundant computations in mutation analysis. However, all these techniques are static, and thus cannot remove redundancy that occurs in part of mutant execution.

The purpose of this thesis is to design a technique that accelerates the mutation analysis dynamically, which analyzes the mutants during the execution of the program and forks the execution only when a mutant leads to a new system state. We proposed an abstract model for implementing dynamic mutation analysis for different types of mutation operators. We developed an acceleration tool "AccMut" on C programming language on top of LLVM. Our experiment show that our approach can further accelerate mutation analysis with a speedup 2.22X over Major Framework, a state-of-the-art approach.

At the end of this thesis, we extend our model to software product line testing.

Keywords: Mutation analysis, Program analysis, Software product line

目录

序言

变异测试是一种软件测试方法，也是一种重要的程序分析技术[1,2]。给定一个程序，变异测试通过一些预定义的变异算子对程序进行微小的修改，产生一个程序集合。此集合中的每个程序和原程序都有细微的差别，这些程序称之为原程序的变异体。随后变异测试在这个程序集合的所有程序上对已有的测试集数据进行测试，并统计执行过程中的信息和执行结果进行下一步分析。

变异测试的主要用途是帮助测试员评价测试集的质量[3]。按照通常的理解：一个好的测试集能够检测出程序中所有潜在的错误。变异测试就是这个过程的逆向过程：每一个变异体可以看成一个有潜在错误的程序，如果一个测试集在任何一个变异体上都无法通过，那么这个测试集就可以视作是一个好的测试集。

除了检测测试集的质量，变异测试也被应用在了其他领域中。变异测试的其他的用途包括缺陷定位[4,5,6] 和缺陷修复[7,8,9,10] 等。

图1描述了变异测试的执行流程。首先变异测试通过变异算子对程序进行变异，生成变异体。在通常的情况下，一个程序经过变异算子变异后会生成很多变异体。随后，部分加速工具会对变异体进行筛选（传统的变异测试没有此步骤），得到一个更小的变异体集合。最后，变异测试在这个集合的每一个程序上执行测试集中的每一个测试用例，根据执行结果得到变异测试的最终结果，这个结果也是用来检验测试集质量的标准。

常见的变异算子包括：符号变异（逻辑符号变异，算数符号变异），数值变异，语句级变异（插入或删除一条语句），过程级变异（函数的替换或删除）。目前，绝大多数被使用变异算子均为一阶变异算子（即变异体和原程序只有一处不同）；有些时候根据需要，变异测试也会生成高阶变异体[11]（即变异体和原程序有几处不同），这种变异算子称为高阶变异算子。

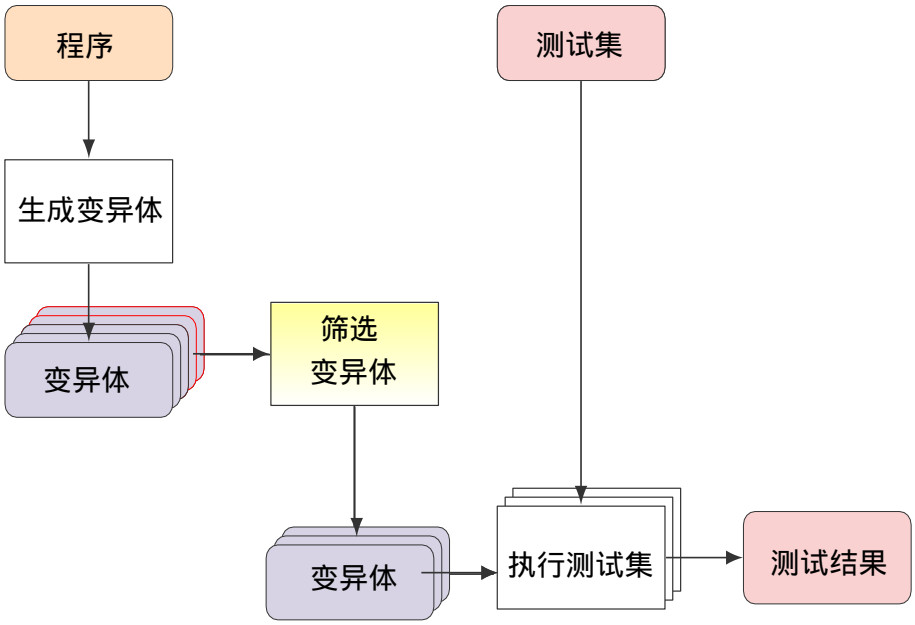


图 1: 变异测试流程概览

下面用一个例子来解释不同的变异算子。考察下面程序：

```
int triple(x):
  return x+x+x;
int zero(x):
  return 0;
main():
  int y = triple(x);
  print y;
```

下面给出不同变异算子对原程序变异后所产生的一个[1]变异体。

一阶操作符变异：

```
int triple(x):
  return x+x-x;
int zero(x):
  return 0;
main():
  int y = triple(x);
  print y;
```

一阶数值变异：

```
int triple(x):
  return x+x+x;
int zero(x):
  return 1;
main():
  int y = triple(x);
  print y;
```

[1] 实际中，一个变异算子会对原程序产生许多变异体，这里只选取有代表性的一个。

一阶语句级变异：

```
int triple(x):
  return x;
int zero(x):
  return 0;
main():
  int y = triple(x);
  print y;
```

一阶过程级变异：

```
int triple(x):
  return x+x+x;
int zero(x):
  return 0;
main():
  int y = zero(x);
  print y;
```

高阶操作符变异：

```
int triple(x):
  return x*x*x;
int zero(x):
  return 0;
main():
  int y = triple(x);
  print y;
```

高阶混合变异：

```
int triple(x):
  return x+x-x;
int zero(x):
  return 1;
main():
  int y = triple(x);
  print y;
```

变异测试有一个重要的瓶颈：可扩展性差。假设一个程序的测试集中有 m 组输入，而变异测试在这个程序上生成了 n 个变异体，那么整个变异测试的执行时间为原程序执行时间的 $n \times m$ 倍。虽然 m 是固定的，但是当我们对变异算子进行扩展的时候，变异体数量 n 就会随之膨胀，进一步导致整个变异测试的执行时间增长，导致巨大的时间开销。这也是变异测试在实践中只被小规模采用的原因。

为了解决可扩展性的问题，近年来研究人员已经提出了各种不同的方法来加速变异测试的执行。这些方法分为有损加速技术和无损加速技术。

主要的有损加速技术有：

- **Weak Mutation**

  Weak Mutation[12] 是一个典型的有损加速技术，Weak Mutation 认为只要变异体在某一个测试用例上产生了和原程序不同的系统状态，那么就认为这个这个测试用例无法通过此变异体。这显然是一个有损的方法（因为有些变异体即使产生了不同的系统状态，也不一定会产生错误的结果）。

- **Mutation Sampling**

  Mutation Sampling[13] 是另一个有损加速技术，此技术在所有变异体中选取一些具有代表性的变异体，并且只在这些变异体上进行测试，来减少执行时

间。

主要的无损加速技术有：

- **Major Framework**

  Major Framework[14] 是目前为止最快的无损加速工具。它通过静态分析的方式结合测试集中每组测试用例的数据对程序进行预处理，进而对变异体进行一次筛选和一次等价类划分。Major Framework 按照下面三个标准进行等价类划分：

  1. 覆盖等价：如果一个测试用例没有覆盖到某个变异体的变异点语句，那么这个变异体就可以认为是和原程序等价的。
  2. 传播等价：如果两个变异体所产生的变异点在同一个复合表达式上，而这个表达式的值是一样的，那么这两个变异体就可以认为是等价的。
  3. 结果等价：如果两个变异体在一个测试用例上在变异点语句上的所有执行结果都相同，那么这两个变异体就可以认为是等价的。

  划分完成后，Major Framework 逐一执行测试集中的每组测试用例。在一个等价类中任意选出一个变异体执行该测试用例，而不需要在同一等价类中的其他变异体上执行。这样每个测试用例就可以节省很多等价的重复执行，节省了时间。

- **Mutation Schemata**

  Mutation Schemata[17] 加速了生成变异体的过程。由于每个变异体都是一个新的程序，所以编译所有变异体需要大量的时间。Mutation Schemata 将所有的变异体整合到了同一个程序上，通过命令行参数或者宏定义来控制当前执行的变异体，所以生成变异体的过程中只需要进行一个程序的编译，节省了大量的编译时间，从而加速了整个变异测试。

- **Test Prioritization**

  Test Prioritization[15] 是一种从测试用例出发加速变异测试的方法，此方法针对不同变异体通过测试用例排序技术对测试集中的不同测试用例进行了重排，使得变异体尽可能早地被检测出[2]，这样就不用执行测试集中的其他测试用例了。

[2] 当变异体无法通过测试集中的某个测试用例，则可以认为这个变异体被检测出。

这些方法中，Mutation Schemata 和 Major Framework 为比较常用的加速技术，Major Framework 是目前最快的变异测试加速工具。

然而，以上的加速技术有一个共同的不足：它们都是静态的加速技术。事实上，任意给定两个变异体，在发生变异的变异点之前，这两个程序在同一个输入上的执行过程是完全相同的，也就是说在变异体执行的过程中存在大量的冗余计算。所有静态的加速技术只能加速变异测试流程（图 1）中的生成变异体和筛选变异体部分，但无法加速执行测试集部分。也就是说，静态加速技术无法消除程序执行过程中的冗余部分。

本论文的目标是设计一个动态变异测试技术：在变异测试的执行过程中对变异体进行分析，仅在变异体产生新的系统状态的时刻创建新进程来执行变异体。与静态加速技术不同，动态变异测试从一个包含了所有变异体的程序开始执行，每遇到一个包含了变异体的语句，就对此语句以及所有的变异语句作动态分析，根据分析结果进行等价类划分。当且仅当有新的等价类诞生的时候（表示此变异体集合会产生新的系统状态），原程序会创建一个新的进程，在这个进程中执行新的等价类的变异体，而并不是从头开始执行每个变异体。

动态变异测试是无损的加速技术，并且有以下两个优点：

- 不同的变异体在变异点之前共享同一个执行过程，从而消除了执行过程中的冗余部分，节省了大量的执行时间。
- 由于动态变异测试的加速部分和其他相关工作的加速部分不同，所以动态变异测试可以兼容所有静态的加速技术。在我们的抽象模型中（第二章），我们会详细解释如何将动态变异测试和两大现有技术，Mutation Schemata 和 Major Framework 结合起来，进一步提高加速比。

本论文将给出动态变异测试的方法概述和加速原理（第一章），并详细定义了支持变异测试的抽象模型，并且在这个模型上给出了静态变异测试的算法并设计了动态变异测试的算法（第二章）。我们在 LLVM 的框架上实现了一个 C 语言变异测试的加速工具 AccMut（第三章），并同时在 LLVM 的框架上复现了现有的加速工具 Mutation Schemata 和 Major Framework，通过实验用来进行横向对比和验证（第四章）。实验表明动态变异测试技术加速效果显著，加速比是目前

最快的变异测试加速工具 Major Framework 的 2.22 倍。最后，本论文简单提出了动态变异测试技术的抽象模型在软件产品线测试上的扩展方法（第五章）。

说明: 我的本科生科研的课题也是变异测试的加速，但是本论文的内容和本科生科研论文的内容有了较大的区别，本论文和本科生科研论文的主要区别有：

1. 本科生科研时我采用的等价类划分使用的是复现的 Major Framework 的划分方法，而本论文则使用的是纯动态的自行设计的等价类划分方法；简而言之，本科生科研的主要内容为 “变异测试的动态执行”，而本论文的主要内容为 “基于动态分析的变异测试”。
2. 本论文实现的工具已经支持涉及文件 IO 的程序，而在本科生科研实现的工具中并不支持。
3. 在变异算子的规模上，本论文超出了本科生科研所使用的变异算子规模。
4. 本论文的抽象模型支持高阶变异算子和软件产品线测试，而本科生科研的工作并不支持。

第一章　方法概述

在这一章节中，我们将概述动态变异测试技术，并通过一个简单的例子解释动态变异测试的加速原理。

1.1　启发：记忆化搜索

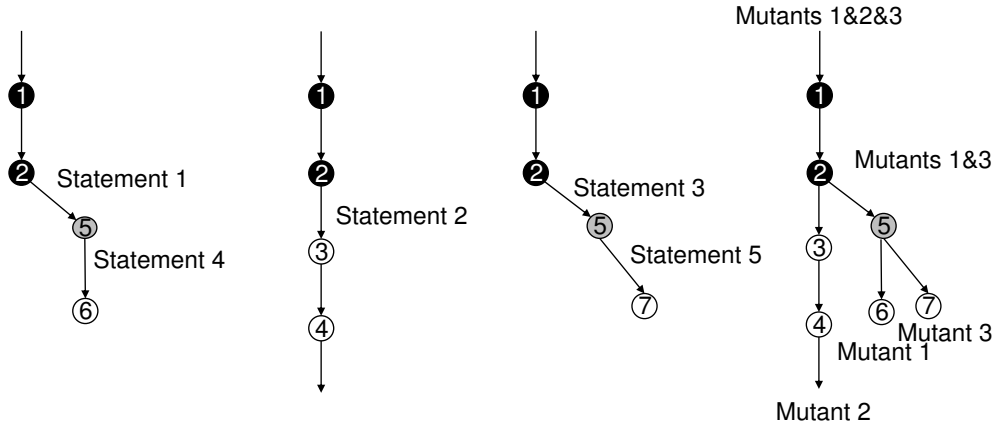
动态变异测试技术由记忆化搜索启发。记忆化搜索是一种结合了动态规划的搜索方法，也是一种在搜索问题求解中被广泛使用的策略。朴素的动态规划算法需要遍历所有的状态但是会保存状态，而搜索可以算法可以通过剪枝来排除一些无效状态。记忆化搜索结合了动态规划算法和搜索算法的优点，在求解的时候按自顶向下的顺序进行搜索，但是每求解一个状态，就将它的解保存下来，这样在再次遇到这个状态的时候，就不必重新求解了。

记忆化搜索可以减少搜索过程中大量的重复计算，而动态变异测试的目标也是去除程序的执行中的重复计算。受记忆化搜索启发，动态变异测试技术的初始方法为记忆化程序执行的方法。这个方法在每个变异点的位置记忆当前程序状态，在运行变异体的时候直接从已记忆的状态开始执行，避免了变异点之前部分的重复执行。随后我们发现，POSIX 系统调用 fork() 可以帮我们简化这个过程。通过在变异位置创建新进程的方式，我们可以免去保存当前程序状态的这个步骤，从而可以节省大量的时间和空间。所以，动态变异测试使用创建新进程的方式代替了记忆化程序执行的方式。

具体来说，动态变异测试在变异测试的执行过程中对变异体进行动态分析，并且仅在变异体将会产生新的系统状态的时候，通过创建新的进程来执行变异体。

1.2　动态变异测试的加速原理

如序言所说，动态变异测试通过共享不同变异体中重复的执行过程来加速变异测试的执行过程。



(a) Mutant 1 Execution　(b) Mutant 2 Execution　(c) Mutant 3 Execution　(d) Dynamic Mutation Analysis

图 1.1: 基于进程创建的动态变异测试

图1.1通过一个例子详细描述了基于进程创建的动态变异测试技术如何消除执行过程中的冗余计算。图中的每个圆圈代表了一个系统状态，数字相同的圆圈代表了相同的系统状态。箭头代表了在执行一句或者多句语句（具体情况根据变异算子而定）之后的系统状态的转换过程。在状态 2，图中的三个变异体执行了三句不同的语句或者语句块，使得系统状态转换到了两个不同的新系统状态（状态 3 和状态 5）。其中，虽然有两个变异体（变异体 1 和变异体 3）的语句是不同，但它们产生的结果是相同的，即转换后的系统状态是相同的[1]。在状态 3，变异体 1 和变异体 3 再次执行了不同的语句，而这次两个变异体执行的语句产生了不同的结果，转换后的新系统状态不同。所以，状态 1，状态 2，以及之前的一系列状态对于这三个变异体来说是完全相同的，这就导致了冗余的执行过程。同理，状态 5 以及之前的一系列状态对于变异体 1 和变异体 3 来说也是完全一样的，也是冗余的执行过程。根据重复度的不同，我们把冗余的执行过程标注成了黑色的或者是灰色的圆圈。

[1] 这是可能发生的，比如当 $a == 2$ 的时候，语句 $a = a + 2$ 和语句 $a = a * 2$ 执行后的效果是一样的。

因为这三个变异体最终产生的系统状态是不同的，所以在它们的执行过程中，只有其中一部分的执行是可以共享的，这就意味着任何静态加速技术无法消除这部分冗余的执行过程。相比之下，动态变异测试技术可以通过在执行过程中创建新进程的方法来复用这部分的重复执行。变异测试的动态执行方法从一个代表了所有变异体的主进程开始执行，如图1.1(d)。在每个系统状态，我们检查在这个系统状态下所有变异体将要执行的语句。如果所有变异体将要执行的语句是一样的，那么我们就简单地执行这条语句；如果存在不同的可能需要执行的语句，那么我们就对不同的执行过程可能产生的结果进行分析，并且在需要的时刻创建新的进程。在状态 2，三个变异体有三个不同的语句需要执行，所以我们需要分析这三条语句可能产生的执行结果。

分析过程中，我们依次模拟执行每条语句并且收集它们的结果。根据不同的结果，我们将这些变异体划分成不同的等价类。两个变异体在某个系统状态是等价的当且仅当这两个变异体在这个系统状态下所需要执行的语句是一样的或者这两个变异体在这个系统状态下所执行的语句产生的结果是一样的。例如，状态 2 下，变异体 1 和变异体 3 虽然执行的语句不一样，但是产生的结果是一样的，那么变异体 1 和变异体 3 在状态 2 下就被认为是等价的，归入同一个等价类。

如果在某个状态下，我们分析并生成了 n 个等价类（$n > 1$），那么我们就要创建 $n - 1$ 个新进程。每个新创建的进程都代表一个等价类，在这个进程中，我们在其代表的等价类的变异体中任意选取一个语句（在这个系统状态下的）并执行该语句。在这个例子中，在状态 2 下，我们创建一个子进程代表变异体 1 和变异体 3 并执行其中任何一个的语句，进而转换为状态 5。到了状态 5，此新进程又创建出一个孙子进程代表变异体 3, 孙子进程执行变异体 3 的语句转换为状态 7，子进程执行变异体 1 的语句转换为状态 6，而原进程在状态 2 执行变异体 2 的语句，转换为状态 3，进而转换为状态 4。

从图 1.1(d) 中我们可以看到，状态 1 之前的转换，状态 1 到状态 2 的转换，以及状态 2 到状态 5 的转换都只执行了一次，所以不会有冗余的执行过程。这就是是动态变异测试的加速原理。

1.2.1　相关背景：系统调用 fork()

我们的方法使用系统调用 fork() 来创建新线程。fork() 是 POSIX 系统中的一个系统调用，在其他操作系统例如 Windows，也有 fork() 系统调用的实现（通过 Cygwin [2] 支持）。

POSIX 系统调用 fork() 用于从已存在进程中创建一个新进程，新进程称为子进程，而原进程称为父进程。得到的子进程是父进程的一个拷贝，它从父进程处继承了整个进程的地址空间，包括进程上下文、进程堆栈、内存信息、进程优先级等。父进程和子进程最大的区别在于 fork() 函数的返回值，fork() 向子进程返回 0，向父进程返回子进程的进程号。根据这个返回值我们可以确定当前执行的是父进程还是子进程。

POSIX 系统调用 fork() 对父子进程的内存管理通过 “写时拷贝” 机制[16] 来实现：当 fork() 被调用的时候，系统会创建一个新的进程对象并且赋予其一些基本信息，而没有马上从父进程拷贝整个虚拟内存空间。相反地，子进程在创建后的伊始与其父进程共享虚拟内存空间。当且仅当某一个进程（父进程或者子进程）的虚拟内存空间发生写操作的时候，系统才会对所写内存地址所在的页面创建一份新的拷贝，并且更新子进程的页表信息。由于 “写时拷贝” 机制的使用，系统调用 fork() 本身的执行速度很快，同时被创建的子进程的执行速度和普通进程（不是从另一个进程创建出来的进程）执行速度并没有太大差别。

POSIX 系统调用 fork() 的另一个特性是所有由父进程打开的文件描述符都被复制到子进程中。父、子进程中相同编号的文件描述符在内核中指向同一个 file 结构体。也就是说父子进程的文件读写操作是对彼此有影响的，这也给动态变异测试对文件 IO 的支持带来了挑战。本文的 3.4 章节将会详细解释这个挑战和我们的解决方法。

[2] https://www.cygwin.com/faq.html

第二章　算法设计

为了更具体地说明动态变异测试技术，本章节将给出支持动态变异测试的抽象模型以及详细的算法设计。

2.1　抽象模型

为了更好地支持不同种类的变异算子，尤其是一阶和高阶的变异算子，我们定义了一个基础结构的抽象模型来支持动态变异测试技术。这样做的好处在于，当变异算子发生改变的时候，我们只需要修改抽象模型的实现就可以了，而不需要修改整个抽象模型。进一步，我们发现这个抽象模型不仅可以用于动态变异测试，还可以扩展到其他领域，如软件产品线测试（第五章）。

2.1.1　变异过程的抽象模型

给定一个程序，变异测试首先通过变异算子生成不同的变异体。因为不同的变异算子所产生变异的粒度不一定相同，例如表达式级变异，语句级变异，语句块级变异等，所以我们使用一个抽象的概念，位置（location）来代表变异算子所发生变异的单元。同时，我们认为每个不同的变异体都有一个专有的变异号（mutant ID）。

更具体一点，一个程序可以被视作是一个位置的集合，一个变异过程 p 是一个从位置到变元（variant）集合的映射。此映射结果由变异算子（mutation operator）决定。一个变元 v 由一个可执行的代码块 $v.code$ 和一个变异体集合（实际上是变异号的集合）$v.I$ 组成。其中，变异体集合包含了所有包含 $v.code$ 的变异体。而变异号集合符合以下几个特性：

1. 对同一个过程 p 和任意两个不同的程序位置 l_1 和 l_2，这两个位置具有相同的 “映射后变元集的变异体集合的并集”, 也即 $\bigcup_{v \in p(l_1)} v.I = \bigcup_{v \in p(l_2)} v.I$ ，而这个并集就是 p 所产生的所有变异体。
2. 对于同一个位置经过 p 过程映射后变元集中的任意两个不同变元 v_1 和 v_2，它们包含的变异体集合的交集是空集，也即 $v_1, v_2 \in p(l) \Rightarrow v_1.I \cap v_2.I = \emptyset$。
3. 给定一个变异号 i，从每个位置映射后的变元集中，取出变异体集合中包含 i 的变元，并将这些变元组合起来构成一个新的程序，这个程序就是变异体 i。

举一个例子，下面是一个两行程序：

```
a = a + 1;
b = b + 1;
```

假设我们仅有一个变异算子：将加号替换为减号和乘号，并且变异粒度为语句级变换。那么此程序就有两个位置：第一行和第二行。在第一行，这个变异算子生成了三个变元: (a) `a = a + 1` ，(b) `a = a - 1` ，和 (c) `a = a * 1` 。其中 (a) 是原语句，(b) 和 (c) 是变异语句。同理在第二行，变异算子生成了三个变元：(d) `b = b + 1` , (e) `b = b - 1` ，和 (f) `b = b * 1` 其中 (d) 是原语句，(e) 和 (f) 是变异语句。

进而这个变异算子产生了四个变异体，变异号为 1-4,，并且有：

(a). $I = \{3, 4\}$, (b). $I = \{1\}$, (c). $I = \{2\}$, (d). $I = \{1, 2\}$, (e). $I = \{3\}$, (f). $I = \{4\}$.

特别地，这个抽象模型是支持高阶变异的（在程序的多个语句产生变异）。

2.1.2　程序执行过程的抽象模型

程序的执行可以视作为一系列的系统状态的相互转换。一份特殊的函数 Φ 将每个系统状态（system state）映射到位置（location）上，代表在这个系统状态下需要下一步执行的语句块。特别地，当系统状态为 s 时，并且 $\phi(s) = \perp$，那么

程序到达终止态。给定一个系统状态 s 和一个语句块 c[1]，execute(s, c) 操作得到的结果为：系统状态 s 下执行语句块 c 之后的状态。进一步，可以分解 execute 为 `try` 和 `apply` 。`try`(s, c) 操作在系统状态 s 下执行语句块 c，并返回一个系统状态的改变 x 但不直接改变系统状态 s，`apply`(x, s) 操作将这个改变 x 真是应用到状态 s 上并转换到新的系统状态。

为了更高效地实现动态变异测试技术，我们定义了以下两个方法：

- filter_variants(V, I). 这个方法基于一个变异号集合，从一个变元集合中筛选出一个子集，这个子集中的每个变元的变异号集合中至少出现一个 I 中的变异号，即将 V 更新为 $\{v \mid v \in V \wedge v.I \cap I \neq \emptyset\}$. 这些变元是从同一个位置（location）映射过来的。
- filter_mutants(I, V). 这个方法基于一个变元集合，从一个变异号集合中筛选出一个子集，这个子集中的每个变异号都至少出现在其中一个变元的变异号集合中，即将 I 更新为 $\{i \mid i \in I \wedge \exists v \in V.i \in v.I\}$。同样地，这些变元是从同一个位置映射过来的

接下来，我们首先将在这个抽象模型上给出静态变异测试的算法，随后将设计动态变异测试的核心算法，并将与静态变异测试的算法进行比较。

2.2 算法

2.2.1 静态算法

基于抽象模型，我们首先给出传统静态变异测试的算法（算法1）。

给定系统中所有变异体的变异号，静态变异测试的算法逐一执行每个变异体（第 1 行）。每个变异体的执行可以看做是一系列的系统状态的转换直到没有需要执行的语句块（第 3 行）。在每个系统转换过程中，系统根据此刻的变异号选择相应的变元（第 4 行），然后执行这个变元的语句（第 5 行）。最后通过调用 save(s, i) 将执行过程中所需要的信息记录起来（第 7 行）。

以上算法通过解释器可以高效地实现。当我们使用编译器实现时，可以首先

[1] 这里我们认为在一个位置的语句块的中间不会有跳转指令，即语句块是顺序执行的一块。如果语句块中存在跳转指令，如 while 或者 if，那么我们以可能发生跳转的节点为边界，将其分为两个位置。

Input: p: 变异算子
Data: s: 现在的系统状态
for 变异号集合中的每个变异号 i **do**
    $s \leftarrow$ 系统初始状态
    while $\phi(s) \neq \perp$ **do**
        $\{v\} \leftarrow \text{filter\_variants}(p(\phi(s)), \{i\})\ \text{execute}(v.code, s)$
    end
    $\text{save}(s, i)$
end

算法 1: 静态变异测试算法

通过一次插桩把变异算子 p 可能生成的所有变元插入原始程序中，并通过命令行参数或者宏定义来控制当前执行的变异体，Mutation Schemata[17] 是基于这个原理实现了算法1 。

2.2.2　动态算法

Input: p: 变异算子
Data: s: 现在的系统状态
Data: I: 当前进程所代表的变异号集合
$I \leftarrow$ 所有的变异号的集合
$s \leftarrow$ 系统的初始状态
while $\phi(s) \neq \perp$ **do**
    $\text{proceed}(p(\phi(s)))$
end
for $i \in I$ **do**
    $\text{save}(s, i)$
end

算法 2: 动态变异测试的主循环

和静态变异测试不同，动态变异测试从一个代表了所有变异体的进程开始，当且仅当遇到某个变异体改变了原有的系统状态的时候才创建新的进程。算法2展示了动态变异测试的主循环，这里我们可以看到三个动态变异测试和静态变异测试不一样的地方。

1. 在测试的一开始，变异号集合 I 包含了所有的变异号。
2. 在所有变异体执行结束的时候，我们用一个循环逐一统计每个变异体的结果和信息。
3. execute() 函数被换成了 proceed() 函数，proceed() 函数除了负责系统状态的转换，还负责新进程的创建。

动态变异测试的核心算法是 proceed() 函数（算法3）。

首先如果这个位置只有一个变元需要执行，那么我们就直接执行然后返回（第 2-6 行）。否则，我们首先尝试执行每个变元的语句块，然后根据尝试执行的结果对这些变元进行等价类的划分（第 7-11 行）。这里，同一个等价类中的不同的变元的语句块在当前系统状态下执行后的系统状态是一样的。如果划分结果为当前有多个等价类，那么我们为其中随机选出一个等价类（第 12 行），并且为剩下的等价类创建新的进程（第 13-15 行）。创建出的每个进程都代表了一个等价类（第 17 行），我们从这个等价类中随机选出一个变元，执行它的语句块，更新系统状态（第 18-19 行）。最后，由于原进程本身也代表了一个等价类，所以我们对原进程进行同样的操作（第 24-27 行）。

如果划分出的等价类特别多，那么我们就需要同时创建出很多个新进程。从操作系统的角度来看，过多的进程并行执行会给操作系统带来调度的负担，因此我们需要限制创建的进程数。为了简化进程并行的管理，在我们的算法中，我们选择限制同时运行的进程数为 1，这意味着父进程在创建完子进程后会挂起等待子进程结束后再继续执行。我们使用 waitpid() 系统调用限制进程数（第 21 行），这个系统调用会挂起父进程知道子进程结束后才继续。

这里将进程数限制为 1 并不意味着完全放弃并行执行。在支持并行的体系结构上，动态变异测试可以在测试用例级别并行执行，即每个进程代表的输入为测试集中的一个测试用例，这样在限制创建出新进程数量的同时，我们也可以利用并行提高动态变异测试的执行效率。

为了防止子进程有时可能不会终止（例如变异引发了无限循环等），我们在创建子进程的时候设置了一个计时器，当子进程执行的 CPU 时间超过了某个阈值的时候，我们强制终止子进程。对于超时变异体控制的详细讨论，在 4.1.3 节给出了超时变异体的详细讨论。

同时可以注意到，这个动态变异测试的算法已经包括了 Mutation Schema-

Input: V: 当前位置所映射的变元集合
Data: s: 当前的系统状态
Data: I: 当前进程所代表的变异号集合

filter_variants(V, I)
if $|V| = 1$ **then**
    $v \leftarrow V$ 中唯一的变元
    execute (v.code, s)
    return
end
$X = \emptyset$
for V 中的每个变元 v **do**
    $X \leftarrow X \cup \{\texttt{try}(v.code, s)\}$
end
$\mathbb{X} \leftarrow$ 将 X 划分成等价类
$X_{cur} \leftarrow \mathbb{X}$ 中的任意一个等价类
for $\mathbb{X} - \{X_{cur}\}$ 中的每个等价类 X **do**
    $V \leftarrow X$ 中的任意一个改变所对应的变元
    $pid \leftarrow$ fork()
    if $pid = 0$ **then**
        // 子进程
        filter_mutants(I, V)
        $x \leftarrow X$ 中的任意一个为变异的改变
        apply(x, s)
    else
        // 父进程
        waitpid(pid)
    end
end
$V \leftarrow X_{cur}$ 中的任意一个变元
filter_mutants(I, V)
$x \leftarrow X_{cur}$ 中的任意一个变元的改变
apply(x, s)

算法 3: proceed(s) 算法

ta[17] 和 Major Framework[14] 的加速部分。我们的程序也可以通过插桩将不同的变元插入到原程序中，所以也只需要编译一次；在 process() 的第 11 行，我们通过 try 的结果进行等价类的划分，这就是 Major Framework 的加速技术。由此我们证明了动态变异测试的第二个优点，可以和现有的静态加速技术结合，进一步提高加速比。

动态算法的正确性

这里我们证明了动态算法的正确性。为了证明正确性，我们只需要说明动态算法在程序执行完的产生的结果和静态算法一样的。

定理 1 *Algorithm 2* 会调用 $save(s, i)$ 当且仅当 *Algorithm 1* 会用相同的参数调用 $save(s, i)$ 。

证明 1 要证明上述定理，只需要证明动态算法和静态算法所产生的系统状态转换序列是一样的即可。在每个为止，动态算法和静态算法选择的都是完全一样的变元，如果 `try`, `apply`, 以及等价类划分的算法被正确地实现了，那么转换后的系统状态也是完全一样的。

2.2.3 一阶变异算子的筛选算法

为了更进一步说明动态变异测试算法的实现，我们首先给出一阶变异测试的筛选算法的实现，这也是我们目前所实现的变异算子和筛选算法。目前绝大多数的变异测试都只使用一阶变异，所以本实现已经适用于大部分的变异测试工具。当然，之前给出的抽象模型是支持高阶变异算子的，针对高阶变异算子有不同的实现方法，在本章节的后面有对高阶变异算子筛选算法的实现和讨论。

首先，一阶变异测试需要满足下述条件：

- 变异算子是作用在语句上的（语句级变异）；
- 每个变异体只包含一个变异语句；
- 每个位置的经过变异算子映射后的变元数量比较少（有上界 u）；
- 变异体的总数量 m 可以很大并且和程序的规模有关。

特别地，第三个条件在源码级变异上往往是不成立的，因为源码中的一句语句通常可以包含很多个操作符。但是我们可以对程序做一次分解，将程序分解成

一系列三地址码语句，这样每个语句只有一个操作符，每个语句产生的变元数量就比较少了，所以在程序的中间码而不是源码上应用变异算子是一个更好的方法，这在后续工具实现部分（第三章）会提及。

实现过程中，最主要的挑战在于如何为两个筛选算法选择合适的高效的数据结构。因为 filter_variants 会被应用在每个位置上，而 filter_mutants 会在每次创建一个新进程的时候被调用。所以这两个算法的复杂度不能很高。然而这两个算法的实现过程中，都需要计算集合的交集，而标准的集合交集计算的时间复杂度 $O(n \log n)$，其中 n 是集合中的元素个数。对变异测试来说，这个集合中的元素个数是所有变异体的个数，由一阶变异测试所满足条件的第四条，这个数量是十分庞大的，所以整个筛选算法的开销会非常大。

为了更高效地实现筛选算法，我们需要利用一阶变异测试所满足条件的第三条，即每个位置经过变异算子映射后的变元数量有上界 u。我们定义了一下三个数据结构：

- `VectorSet`: 这个数据结构用通过位表实现集合。位表中 1 所在的位置表示集合中包含的元素，0 所在的位置表示集合中不包含的元素。在我们的实现中，原进程当前所代表的变异号集合使用 `VectorSet` 数据结构
- `ListSet`: 这个数据结构通过链表实现集合。链表中包含的元素就是集合的元素。在我们的实现中，每个位置映射后的变元集合以及每个被创建的新进程的所代表的变异号集合均使用 `ListSet` 数据结构
- `DefaultSet`: 这个数据结构仅仅是一个占位符。他所代表的集合只有一个元素，这个元素就是在这个占位符表示的位置的元素。在我们的实现中，每个位置的原程序语句被包装在了 `DefaultSet` 数据结构中。

基于这三个数据结构，我们设计了如下的筛选算法。

算法4 是 filter_variants 。其中，`DefaultSet` 只有一个元素（原语句），其他的变元通过 `ListSet` 表示，所有的这些变元均发生在同一个位置中。算法首先对 `ListSet` 进行筛选 (第 3-6 行)，把所以包含变异号集合中变异号的变元筛选出来，随后判断是否需要把原语句加入集合（第 8-10 行）。由于是一阶变异测试，每个变元至多代表一个变异体，所以如果筛选出来的变元集合的变元个数小于变异号集合个数，那么一定有某个变异号并没有出现再变元集合中的任何一个变元中，那么这个变异体在这个位置的语句一定是原语句，所以需要把原语句加入变元集

Input: V: 需要筛选的变元集合（`ListSet` ）
Input: I: 用来筛选 V 的变异号集合（`ListSet` 或者 `VectorSet` ）
1 $V' \leftarrow \emptyset$
2 $v' \leftarrow$ `DefaultSet` 中 V 的元素（可能没有）
3 **for** 每个 $v \in V$ 且 $v \neq v'$ **do**
4 　**if** $I.contains(v.I.first)$ **then**
5 　　$V' \leftarrow V' \cup \{v\}$
6 　**end**
7 **end**
8 **if** $V'.size < I.size$ **then**
9 　$V' \leftarrow V' \cup \{v'\}$
10 **end**
11 $V \leftarrow V'$

算法 4: filter_variants

合；相反地，如果变元个数不小于变异号个数，那么变元和变异号就是一一对应的，故这个位置的原语句就不需要加入变元集合了。

算法5 是 filter_mutants 。如果当前的变元集合包含了 `DefaultSet` 的元素，那么表示当前的进程代表的变异体中包含原程序，也就意味着当前为原进程，而此时的变元集合的数据结构是 `VectorSet`, 所以我们只需要将其中不在 V 中的变元删除即可，即相应的位置设 0（第 1-4 行）；否则表示当前为创建出的新进程，不包含原程序，此时我们新建一个 `ListSet` 把每个变元所代表的变异号加入集合即可。

复杂度分析

在初始化过程中，`ListSet` 和 `DefaultSet` 的初始化时间为 $O(1)$，`VectorSet` 的初始化时间为 $O(n)$, 但是它只在原进程中初始化，一共只初始化一次。通过均摊分析，我们知道初始化操作在均摊到每个变异体后的复杂度仍然是 $O(1)$ 的。

在我们的算法中，所有的 `ListSet` 的大小都是有上界 u 的，这个 u 是一个常数，所以所有对 `ListSet` 的操作都是 $O(1)$ 的。`DefaultSet` 仅仅是一个占位符，表示原语句，所以对其的操作也都是 $O(1)$ 的。对 `VectorSet` 的操作只有 filter_mutants 算法的第 1-4 行，这里第 2 行的 $p(L)$ 的大小是有上界 u 的，而

Input: I: 需要筛选的变异号集合（`ListSet` 或者 `VectorSet` ）
Input: V: 用来筛选 I 的变元集合（`ListSet` ）
Input: L: 当前位置
1 **if** V 包含 `DefaultSet` 中的元素 **then**
2 　**for** 每个 $v \in p(L) - V$ **do**
3 　　$I.remove(v.I.first)$
4 　**end**
5 **else**
6 　$I \leftarrow$ 新的空 `ListSet`
7 　**for** 每个 $v \in V$ **do**
8 　　$I.add(v.I.first)$
9 　**end**
10 **end**

算法 5: filter_mutants

`VectorSet` 删除一个元素也是 $O(1)$ 的，所以对 `VectorSet` 的操作也是 $O(1)$ 的。

综上，两个筛选算法的时间复杂度均为 $O(1)$，整个变异测试的初始化复杂度为 $O(n)$，在筛选过程中总开销的时间复杂度为 $O(1) * n = O(n)$。所以动态变异测试均摊到每个变异体上的时间开销是 $O(n)/n = O(1)$。这个开销是很小的，所以动态变异测试在传统静态变异测试的基础上不会带来额外的是时间开销。

2.2.4　高阶变异算子的筛选算法

目前主要的高阶变异算子主要有两种，第一种高阶变异算子虽然会对程序的几处发生变异，但是这些变异发生在同一个语句块内，对于这种变异算子，我们只需要将位置的范围扩大到语句块即可，而不需要改变算法。第二种高阶变异算子会在程序的几处发生变异，并且发生变异的位置距离较远，也没有很大的关联。对于这种变异算子，我们需要重新定义我们的筛选算法 filter_mutants 和 filter_variants 。

算法6是针对第二类高阶变异算子的 filter_mutants 算法。这个算法和一阶变异算子的 filter_mutants 算法基本相同。唯一的不同在于在由于高阶变异算子的一个变元可能包含多个变异号，所以我们需要循环对变元的每个变异号进行添加（第 8 行）。

Input: I: 需要筛选的变异号集合（`ListSet` 或者 `VectorSet` ）
Input: V: 用来筛选 I 的变元集合（`ListSet` ）
Input: L: 当前位置
if V 包含 `DefaultSet` 中的元素 **then**
  for 每个 $v \in p(L) - V$ **do**
    $I.remove(v.I.first)$
  end
else
  $I \leftarrow$ 新的空 `ListSet`
  for 每个 $v \in V$ **do**
    for $v.I$ 中的每个 i **do**
      $I.add(v.I.first)$
    end
  end
end

算法 6: 高阶 filter_mutants

Input: V: 需要筛选的变元集合（`ListSet` ）
Input: I: 用来筛选 V 的变异号集合（`ListSet` 或者 `VectorSet` ）
$size \leftarrow 0$
$V' \leftarrow \emptyset$
$v' \leftarrow$ `DefaultSet` 中 V 的元素（可能没有）
for 每个 $v \in V$ 且 $v \neq v'$ **do**
  for $v.I$ 中的每个 i **do**
    if $I.contains(i)$ **then**
      $V' \leftarrow V' \cup \{v\}$
      $size \leftarrow size + 1$
    else
      $v.I.remove(i)$
    end
  end
end
if $size < I.size$ **then**
  $V' \leftarrow V' \cup \{v'\}$
end
$V \leftarrow V'$

算法 7: 高阶 filter_variants

算法7是针对第二类高阶变异算子的 filter_variants 算法。在第 5 行，由于高阶变异算子的一个变元可能包含多个变异号，所以我们需要循环对每个变异号进行筛选。同时，需要维护所有筛选过的变异号的数量 $size$，并根据这个值来决定是否要加入 `DefaultSet` （第 14 行）。

除此之外，由于一个变异号 i 可能关联多个变元，所以每个变元包含的变异号有可能在其他位置的 filter_mutants 中被筛去。所以在 filter_variants 的过程中，我们需要对 $v.I$ 进行维护，删去其中不在 I 中的变异号（第 10 行）。

复杂度分析

和一阶变异算子的筛选算法相比，高阶变异算子主要增加的内容为每个非原程序语句的变元的变异号集合的遍历（ filter_variants 的第 5 行和 filter_mutants 的第 8 行）。所以高阶变异算子的筛选算法的复杂度取决于该算子所生成变异体后，每个变元所包含的变异号集合的大小。特别地，如果该算子在生成变异体后，每个变元所包含的变异号集合存在一个与变异对象程序规模无关的上界 u 的时候，高阶变异算子的筛选算法的复杂度也是 $O(1)$ 的。

第三章　工具实现:AccMut

为了更好地验证我们的设计，我们将上述抽象模型和动态变异测试算法实现成了一个变异测试工具 AccMut。为了公开本工具的实现，本工具的所有代码及实验数据已上传至 GitHub 的开源项目上[1]。本章节将给出 AccMut 所针对的程序对象，变异算子的信息，以及 AccMut 的实现细节。本章节的最后将讨论 AccMut 如何处理文件 IO 的相关挑战。

3.1　变异对象的选取

目前的大量相关工作是选择 Java 作为对象程序的，在 Java 语言中，每个 Java 虚拟机都是一个进程，Java 只支持线程级并行而不支持进程级并行。而且，我们使用的是 POSIX 系统 fork()，所以我们所实现的工具所针对的是 C 语言程序的变异测试。我们的实现是基于 clang+LLVM 的。其中，clang 是编译器前端，将 C 语言编译成 LLVM 的 IR，而 LLVM 是一个被广泛使用的编译器后端框架，LLVM 通过一系列的 Pass 对 IR 码进行遍历、优化和编译，最终将 LLVM 的 IR 编译成可执行的程序。我们设计的变异算子是 LLVM 的 IR 级别的变异，因为 LLVM 的 IR 是静态单赋值代码，在 IR 上实现变异算子比在源码级别实现方便很多，相比于源码级变异，IR 级变异省去了词法分析和语法分析的步骤，而且可以控制操作数个数。

[1] https://github.com/shiyqw/accmut

表 3.1: 变异算子

Name	Description	Example
AOR	替换一个算术操作符	`a+2` → `a-2`
ROR	替换一个逻辑操作符	`a==2` → `a>=2`
LVR	替换一个数值	`0` → `1`
LOR	替换一个移位操作符	`a>>2` → `a<<2`
STD	删除一个函数调用	`y=f();` → `;`

3.2　变异算子的设计

表3.1描述了我们目前所支持的支持的动态变异算子。这些变异算子都是 Major Framework 所设计使用的变异算子[18]。我们并没有使用的 Major Framework 所使用的所有变异算子，主要理由为：

1. Major Framework 所针对的是 Java 的变异测试，有些 Java 源码级的变异算子在 LLVM-IR 上并不适用。例如 Major Framework[18] 的 COR 变异算子在条件运算符上进行变异，而在 LLVM-IR 上，所有的布尔型都被编译转换成整型，所以不存在条件运算符。
2. Major Framework 的一些变异算子在我们所选取的实验对象上没有用处，例如 Major Framework 的 LOR 中有关于位运算的变异算子，而我们选取的实验对象中没有出现位运算。

所有的这些变异算子均为一阶变异算子。所以我们的工具也是针对一阶变异算子实现的。本论文并没有实现高阶变异算子的动态变异测试工具，但是在在本章节的后面会讨论分析如何将我们的实现扩展到高阶变异算子上。

3.3　实现细节

具体的实现过程中，我们首先用 C 语言设计了一个动态分析算法库，这个库的主要作用是在动态分析的过程中应用动态变异测试算法对变异体进行等价类划分和新进程创建。同时，我们在 LLVM 上设计了新的 Pass。我们一共为动态变异测试设计了两个 Pass，第一个 Pass 是根据已经设定好的变异算子生成相关的

变异体信息，第二个 Pass 是根据这些变异体信息在指定的位置修改 IR 码，插入库调用语句。

3.3.1 生成变异的 Pass

在这个 Pass 中，AccMut 在访问 IR 码的同时，根据上面设定好的变异算子生成变异体。我们顺序访问源代码的每条 IR 语句，如果发现这条 IR 语句符合某个变异算子的条件，那么就把这条 IR 语句所映射后的变元集保存下来，输出到一个文件中。

这个 Pass 的输出为一个记载了变异体信息的文件[2]。文件的每一行代表了一个变元，其中记录了变元的信息。由于一阶变异测试的变元和变异体是一一对应的，所以我们可以认为每一行代表的就是一个变异体，行号就是变异号。下表详细描述了变异文件每行的结构。

ID	Type	Func	Index	oldOp	oldVal	newOp	newVal

其中，从左到右的表项分别表示变异号，变异算子类型，变异所在函数，变异所在 IR 语句的索引，原语句的操作符，原语句的参数，变异后的操作符以及变异后的参数。

在生成变异体的过程中，我们把变元位于同一个位置的变异体放在了变异文件的相邻行号中。也就是说，同一个语句上的不同变元所代表的变异体的变异号是连续的。

3.3.2 程序插桩的 Pass

在这个 Pass 中，AccMut 在访问 IR 码的同时，对 IR 码进行修改。如果发现这条 IR 语句中出现了变异号，那么我把这条 IR 语句替换为对我们编写的库的一个库调用 process。

根据 IR 语句的不同的操作符类型和参数类型，我们提供了不同的库调用接口。比如针对 64 位整型的算数运算 IR 语句，我们提供的是 process_arith_i64()

[2] 输出一个文件并不需要对变异体进行编译，不需要额外的编译时间。

接口，针对 32 为整型的逻辑运算 IR 语句，我们提供的是 process_logical_i32() 接口，对函数调用语句，我们提供的是 process_call() 接口。process 函数的详细接口为：

$$\texttt{process\_XX}(left, right, from, to)$$

其中，$left$ 和 $right$ 分别表示运行时刻此语句的左右操作数，$from$ 和 to 表示此语句所对应的变元的变异号的闭区间。

3.3.3　动态分析算法库

为了更好的实现抽象模型和动态变异测试算法，我们设计了一个 AccMut 动态分析算法库。这个算法库的入口为 process() 函数，这个函数是通过程序插桩的 Pass 植入 IR 的。而 AccMut 动态分析算法库的本质就是动态变异测试算法中的 proceed()， filter_variants， filter_mutants 在 C 语言上的实现。

具体的实现代码可参考附录 A。

3.3.4　AccMut 对于高阶变异算子的扩展讨论

AccMut 并不支持高阶变异算子。但是如果要使 AccMut 支持高阶变异算子，需要上述每步进行如下修改：

- 在变异 Pass 上，由于每一个变异体可能含有多个变元，而每个变元有可能激活多个变异体，所以变异体和变元之间的映射关系不满足双射关系。因此，除了生成变元信息的文件，还需要生成一个变异体和变元的关联文件，统计每一个变异体包含了哪些变元。
- 在插桩 Pass 上，由于高阶变异算子同一个位置的以变元所包含的变异号不一定连续，所以每条语句所对应的变异号集合应该是几段区间，不能简单地使用 $from$ 和 to 来表示变元范围，应替换为 $from[]$ 和 $to[]$。
- 在动态分析算法库上，需要修改筛选算法为高阶变异算子的筛选算法。

3.4 AccMut 对于文件 IO 的支持

AccMut 实现上的重要挑战在于对文件 IO 的支持。从上述的实现中，我们知道新进程是通过 POSIX 系统调用 fork() 创建的。而 fork() 所创建的进程有着独立的虚拟内存空间，但是子进程与父进程的打开文件列表中的文件指针是共享的。也就是说子进程和父进程在文件 IO 上并不是独立的。

考察下面的程序片段:

```
FILE *fp = fopen("tmp", "r");
int x, y = 1;
y++;
fscanf(fp, "%d", &x);
```

假设在第 2 行有一个变元 `y++` → `y--` ; 并且动态变异测试在动态分析的过程中发现这个需要创建新进程代表这个变元。那么在子进程中就会率先执行第 4 行的 fscanf 语句，这就导致了 fp 的文件内偏移量发生了变化，由于子进程和父进程的文件指针是共享的，所以父进程的 fp 的文件内偏移量也发生了变化，进而当回到父进程后 fscanf 就会发生错误。

一种可行的解决方法是在创建子进程的同时备份父进程所打开的每个文件的文件内偏移量，然后在子进程结束的时候将每个文件文件内的偏移量恢复。然而在一些特殊的程序中会同时对文件进行读写，这样子进程很有可能会改变原文件导致无法完全恢复创建子进程的之前的文件信息，所以此方法并不是一个通用的方法。

为了通用地解决这个问题，我们需要对将 fork() 对内存所使用的 “写时拷贝” 机制实现到文件 IO 上，即每当子进程对文件进行写操作的时候，我们为子进程创建一个新的文件，将父进程的文件内容完全拷贝到这个文件，之后子进程的对文件的读写操作就在这个新文件上进行。实现文件级别的 “写时拷贝” 是一个十分复杂的过程，需要对系统调用 fork() 进行重写。为了更简化地实现，我们使用了一个更朴素的解决方法：将打开的文件映射到一块内存中。在第二个 Pass 插桩的时候，AccMut 将文件相关的系统调用转换为我们自己写的函数，例如 `fscanf()` → `acc_fscanf()` ，在我们设计的函数中，所有的文件读写操作都

被替换为对内存区域的读写操作。由于 fork() 对于内存是遵守 “写时拷贝” 机制的，所以我们的实现等价于实现了文件 IO 的 “写时拷贝”。

当然，这个实现的方法的不足之处在于：如果对于文件 IO 密集的程序或者打开了很多或者很大的文件的程序，我们的方法就会需要额外的大量的内存空间用来映射文件。这样就会对系统的执行带来较大的额外内存的负担。对于文件 IO 密集的程序，最合适的处理办法还是重写 fork()，使其支持文件 IO 级别的 “写时拷贝”。在我们的实验对象中并不存在文件 IO 密集的程序，所以 AccMut 在文件 IO 上的支持算法对动态变异测试的加速比并无太大影响，所以在这个版本中，我们没有实现重写的 fork()。

3.5 Major Framework 的复现

为了更好地和 Major Framework 和 Mutation Schemata 进行对比，我们需要在 C 语言和 clang+LLVM 框架上复现这两个技术的实现。对于 Mutation Scemata，这个复现很简单，只需要在程序的一开始创建出 m 个进程（m 是变异体个数），每个进程执行一个变异体即可。

对于 Major Framework，我们仿照 Major Framework 的 Java 实现，在 C 语言上也实现了一个 Major Framework 等价类划分的算法库。这里，我们简单解释一下 Major Framework 的算法。

Major Framework 首先对程序进行一次静态分析，在静态分析的过程中生成变异体和程序修改，将需要变异的语句替换为库调用，并同时进行数据流分析。在这里，需要数据流分析是因为在判断结果等价的过程中，如果两个变元的结果是等价的但与原程序不等价，并且程序执行流程与这两个变元存在数据依赖的时候，不能将这两个变元划分为一个等价类。随后，Major Framework 执行修改的程序进行等价类划分。在这个执行过程中，Major Framework 的分类库会记录每条语句每次执行的结果以及每次执行时变元的结果，进行等价类划分。主要划分依据为：

- 如果一个变异体的变元没有运算结果，那么这组测试用例就没有覆盖到这个变异体，那么这个变异体和原程序就是覆盖等价的。
- 如果两个变异体所在表达式相同，并且这个表达式的结果和原程序的运行结

果一致，那么这个变异体和原程序就是传播等价的。

- 如果两个变异体每次执行的结果都相同，那么他们就是结果等价的。

注意到，Major Framework 的筛选方法并不适用于高阶变异算子[3]。

在我们的复现过程中，我们首先通过一个 LLVM 的 Pass 将发生变异的 IR 码替换成这个算法库的库调用。随后我们执行一遍程序就可以通过动态分析得到等价类划分的结果。最后根据等价类划分的结果，针对测试集的每个测试用例，结合 Mutation Schemata 的技术，我们在程序的一开始创建出 m' 个进程（m' 是等价类的个数），每个进程执行一个等价类即可。

这里，覆盖等价和结果等价只需要维护一个数组保存每个表达式在每次运行后的结果即可实现。而传播等价需要记录每句 IR 所处的表达式信息。这个信息只有通过修改编译器前端 clang 才能获得。在我们的实现中，我们在 clang 访问抽象语法树的过程中记录了源码表达式和 LLVM-IR 的映射关系。在 IR 遍历过程中，如果发现两句 IR 是同一个表达式映射的，那么我们就向上遍历，这样就可以在 IR 级别简单地建立抽象语法树[4]。在执行过程中，我们只需要比较不同变异体所对应的变元的最顶层 IR 的所有运行结果即可。

3.5.1 AccMut 和 Major Framework 的筛选效果比较

这里我们比较我们的动态变异测试算法和 Major Framework 在筛选变异体的效果上的比较。我们针对 Major Framework 的筛选算法的三种不同的等价类划分策略分别进行比较。

- **覆盖等价：**

  Major Framework 通过覆盖等价筛去的所有变异体均为测试用例没有覆盖到的变元所对应的变异体，而动态变异测试是在动态执行过程中进行创建新进程的，测试用例没有覆盖到的变元将不会被执行，更不会被创建新进程。所以 Major Framework 通过覆盖等价筛去的变异体也同时会被 AccMut 筛去。

- **传播等价：**

  由于 AccMut 是基于纯动态分析的，所以无法在分析过程中判断 IR 所处的

[3] 本章节之前的部分已经讨论过了 AccMut 扩展到高阶变异算子的可能性。
[4] 这个过程被整合到了 Major Framework 的插桩 Pass 中，不需要新增一个 Pass。

表达式。所以 AccMut 无法筛去 Major Framework 通过传播等价筛去的变异体。

- **结果等价：**

  Major Framework 会根据静态分析时的数据流分析结果来决定每个变元是否要进行结果等价。如果在某个语句上发生了状态改变，那么 Major Framework 仅在与程序执行流程无关的变元上进行结果等价的筛选。相反地，AccMut 是在动态执行的时刻进行筛选的，所以无论变元是否与程序执行流程发生数据相关，都可以通过 AccMut 进行筛选和分类。也就是说，Major Framework 所有能通过结果等价够筛去的变异体都能被 AccMut 筛去，并且 AccMut 能够在程序执行流程发生改变后继续筛去等价的变异体，这是 Major Framework 无法筛去的。所以，在结果等价的筛选中，AccMut 能够筛去的变异体集合是 Major Framework 能够筛去的变异体集合的超集，具有更强的筛选力。

所以，相比于 Major Framework 的筛选效果，动态变异测试在传播等价的筛选能力弱于 Major Framework，在结果等价的筛选能力强于 Major Framework。根据 Major Framework 的测试结果[14]，三个等价筛选中，结果等价能筛选出的变异体数量是最多的，远高于传播等价的筛选数量。所以，动态变异测试放弃了一个非核心筛选部分，但是在核心筛选部分比 Major Framework 更有提高。

为了更公平的横向对比，在我们的实验评估测试 Major Framework 的性能过程中，我们用动态变异测试的结果等价的筛选结果替换了 Major Framework 的筛选结果。

第四章　实验评估

4.1　性能预测

首先，我们从理论上对动态变异测试的性能进行预测分析。总体上，动态变异测试与传统的静态变异测试相比，使用额外的时间进行动态分析，从而节省了重复计算的时间。下面我们就从这两个角度来对性能进行分析。

4.1.1　运行时的额外开销

根据动态变异测试算法，运行时的额外开销主要由 process() 函数引起。而 process() 函数中的时间开销主要有以下三个部分：

- 选择变元执行的时间
- 等价类划分的时间
- 集合筛选和创建新进程的时间

这其中，第一个时间开销在 Mutation Schemata 中会出现，第二个时间开销在目前最快的变异测试加速工具 Major Framework 中会出现。这两个时间开销已经被证实是很小的（相比于其节省的时间）。所以动态变异测试主要可能的运行时额外开销就在于集合筛选和创建新进程的时间上。

对于集合筛选，在之前的算法复杂度分析上我们已经证明集合筛选的时间复杂度为 $O(1)$，也就是说所有的集合筛选可以在常数时间内完成；对于创建新进程，我们知道 POSIX 的系统调用 fork() 的额外开销很小，并且动态变异测试技术执行新进程创建的次数不会超过变异体的数量。所以总的来说，动态变异测试运行时的额外开销是很小的。

4.1.2　节省的重复计算

由于动态变异测试集成了 Mutation Schemata 和 Major Framework 的加速技术，所以这两个技术所节省的重复计算我们也都可以节省，所以这里我们只考虑在 Major Framework 的基础上动态变异测试技术还可以节省哪些重复计算。对于每一个变异体，在它的变异点之前的程序执行过程是被共享的，而之后的执行过程是单独的，所以对一个变异体来说，它在执行过程中由于动态变异测试技术节省的重复计算的时间应该是程序开始到变异点之间的程序执行时间。对所有变异体来说，总的节省时间所占的比例应该是程序开始到平均变异点位置的执行时间和整个程序的执行时间之比。假设每个变异点出现在程序的 a 处（$0 < a < 1$），那么动态变异测试的平均时间即为：

$$\bar{t} = \overline{1 - a} = 1 - \bar{a}$$

从经验上来看，这个平均位置应该在程序的中部附近，即 $\bar{a} \approx 0.5$，所以总共可以节省的重复计算的时间因为总时间的一半。进一步，动态变异测试技术的理论加速比为：

$$S = \frac{1}{1 - \bar{a}}$$

在 $\bar{a} \approx 0.5$ 的情况下，$S \approx 2$。

但是，实际上节省的时间是不到总时间的一半的。也就是 $\bar{a} < 0.5$。程序中通常会有很多的循环结构，而如果在循环体中存在变异点的话，那么在大多数情况下，会在循环体的第一次执行过程中就创建出新进程执行变异体。所以加速比应该是大于 1 小于 2 的一个数，至于这个数的具体值是很难计算的，只能通过实验来测试这个值。

4.1.3　超时变异体的影响

正如之前所提到的，有些变异体会产生无法终止的程序。对于这种变异体，即使变异点是在距离程序结束很近的地方，动态变异测试依然无法节省其执行时间。我们计时器所设定的超时阈值对每个变异体都是一样的，也就是说如果出现了超时的变异体，那么这个无论什么时候创建的新进程，这个进程总共执行的时

间都近似于我们所设定的超时阈值。如果这个阈值设的很长，那么整个动态执行变异测试的大部分时间都会花在超时变异体的执行上，这就导致了整个变异测试的大部分时间使无法得到加速的，加速比很低。我们假设非超时变异体的执行时间在所有变异体的执行时间中的比例为 p，并且在非超时变异体上动态变异测试有 S 的加速比，根据 Amdahl 法则[1]，整个变异测试过程的加速比 S' 为：

$$S' = \frac{1}{1 - p + \frac{p}{S}}$$

在 p 很小的时候，即使 S 很大，S' 也接近于 1。这就意味着如果超时阈值过长导致超时变异体所执行的时间过长，即使我们的算法在能够在非超时变异体上有很大的加速比，也无法提高总加速比。所以超时阈值不能设置的过长。

如果这个阈值设的很小，那么很多可以终止的变异体也会被错误的报成超时变异体，这就会导致错误的测试结果。

实现中，我们首先从实验对象中随机选取了 300 个测试用例执行了原程序并统计了执行时间，然后将它们的平均执行时间的五倍时间（5ms）作为超时阈值。经过结果验证，我们发现这个时间是一个比较合适的时间，也是不会产生错误结果的时间段中比较短的时间段。

当然，超时变异体的问题在变异测试的研究领域中是长期存在的，Mutation Schemata 和 Major Framework 也会遇到相同的问题。只不过从他们的技术的来看，超时变异体是不会对他们的加速效果产生关键性影响的。而对于动态变异测试技术，超时变异体的影响是比较关键的。但可以肯定的是，即使超时变异体会产生性能上的影响，动态变异测试在 Major Framework 的基础上还是可以加速的，并不会产生副作用。

4.2 实验对象

我们从 SIR 仓库（software artifact infrastructure repository）[2]中选取了 5 个程序。SIR 仓库被广泛的使用在当前的变异测试研究中。SIR 仓库之所以被广泛

[1] https://en.wikipedia.org/wiki/Amdahl%27s_law
[2] http://sir.unl.edu/portal/index.html.

表 4.1: 实验对象

Name	LOC	Tests	Mutants
grep	10068	339	953
printtoken	726	4130	390
replace	564	5542	864
schedule	412	2650	137
tcas	173	1608	369
Total	11943	13729	2713

使用[19,20]，主要原因在于：

1. 程序的规模并不是很大，变异测试的时间开销处在一个可以接受的范围。
2. 程序的功能多种多样，可以获得更普适的测试结果。

在我们选取的这 5 个程序中，grep 是 Unix 系统中用来进行文本搜索的工具、printtoken 是一个迷你词法分析器、replace 是一个模式匹配和替换工具、schedule 是一个基于优先级的调度器、tcas 是飞行器防撞系统。

这些程序的数据如表4.1中所示。5 个程序的总行数达到了 11943 行，总测试数达到了 13729 行，总变异体数量达到了 2713 个。这三项数据的规模均大于其他大部分 C 语言的变异测试相关工作所采用的实验数据集合。

4.3　实验流程

对于每个实验对象，我们分别测试了 Mutation Schemata 加速后的变异测试，Major Framework+Mutation Schemata 加速后的变异测试以及动态变异测试的执行时间。在实验过程中，为了获得更稳定的结果，在限制同时运行的进程数为 1 的同时，我们也没有将测试用例并行化地执行，而是将不同的测试用例串行地逐一执行，统计总共的执行时间。

实验环境为 Ubuntu 14.10 的笔记本电脑，处理器为 Intel i7-4710MQ 2.5GHz，内存为 12G。

表 4.2: 实验结果

Subjects	DMA	MF	MS	MF/DMA	MS/DMA	MS/MF
grep	2m23s	2m46.06s	8m43.71s	1.160X	3.658X	3.154X
printtoken	20m22.79s	45m14.68s	101m0.74s	2.220X	4.956X	2.233X
replace	11m59.59s	19m13.21s	55m17.08s	1.603X	4.610X	2.877X
schedule	7.38s	9.13s	23.31s	1.237X	3.159X	2.553X
tcas	5.38s	5.48s	1m33.59s	1.020X	17.422X	17.088X
Total/Avr	34m58.28s	67m28.50s	166m58.42s	1.929X	4.775X	2.475X

4.4 实验结果

实验结果如表4.2表示：

其中 DMA 为我们的动态变异测试技术（同时结合了 Mutation Schemata 和 Major Framework 的加速技术）的执行时间，MS 为朴素的变异测试经过 Mutation Schemata 优化后的的执行时间，MF 为复现后的 Major Framework（已经过 Mutation Schemata 优化）的执行时间。不同技术的时间之商即为不同技术之间的加速比。通过实验结果，我们可以得到以下结论：

- 动态变异测试技术的执行时间少于朴素的变异测试和 Major Framework（目前最快的加速工具）。这个结果和我们在性能预测中提到的 “动态变异测试在执行过程中的额外开销很小” 是一致的。
- 动态变异测试技术在大多数实验对象上的加速效果显著。在 Major Framework 的基础上平均加速 1.929X，最大加速 2.220X；在朴素的变异测试的基础上平均加速 4.775X，最大加速 17.422X。
- 动态变异测试技术仅在 tcas 程序上的加速效果[3]不显著。我们调研了 tcas 的程序代码，发现 tcas 为命令行工具，所以它的每个测试占用一个独立的线程。进而，程序的执行时间大多数都被进程的创建和销毁占用，只有一小部分是在测试的执行上的。动态变异测试是通过加速测试的执行而加速整个变异测试的，所以动态变异测试在 tcas 程序上的加速效果不显著，加速比较低。然而，Major Framework 通过筛选变异体在 tcas 上的获得了极高的加速效果，而动态变异测试中包含了 Major Framework 的筛选部分，所以相

[3] 这里的加速效果指的是动态变异测试在 Major Framework 的基础上的进一步加速效果。

比于朴素的变异测试，动态变异测试已经在 tcas 上有了显著地加速效果。

- 只对比 Major Framework 和朴素的变异测试，我们发现 Major Framework 相比于朴素的变异测试已有了很大的加速比。这个结果和 Major Framework 介绍论文[18] 里的信息是一致的。

4.5　有效度的威胁因素分析

本实验主要的内部效度威胁因素在于工具的实现可能是错误的。为了避免这个威胁，我们人工地检查了动态分析的结果和变异测试的结果，发现我们的最终结果和朴素的变异测试在每个实验对象上的结果都是一致的。

本实验主要的外部效度威胁因素在于实验中使用的变异算子。尽管本实验采用的变异算子是从 Major Framework 中迁移过来的，但是 C 语言的变异测试所使用的变异算子集合通常都比我们所使用的大很多。使用更大的变异算子集合会使得变异点分布更均匀，进而会使加速比在不同实验对象上的方差更小。

本实验主要的结构效度威胁因素在于我们测量出来的性能可能不够准确。为了降低这个风险，我们在限制同时运行的进程数为 1 的同时，没有将测试用例并行化地执行，而是将不同的测试用例串行地逐一执行，以获得更稳定的测试结果。

第五章　扩展：软件产品线测试

我们的动态变异测试技术是一项通用的技术，它的应用领域不仅限于变异测试。一个和变异测试紧密相关的领域是软件产品线测试。本章节讲阐述如何将我们的动态变异测试技术扩展到软件产品线测试上。

5.1　软件产品线测试简介

软件产品线是指具有一组可管理的公共特性的软件密集性系统的合集，这些系统满足特定的市场需求或任务需求，并且按预定义的方式从一个公共的核心资产集开发得到。软件产品线测试就是对软件产品线上的所有软件进行测试。软件产品线的软件集合和高阶变异算子生成的变异体集合有相似之处。在软件产品线测试中，有一组布尔型的配置项（可以看做是一个布尔向量），在程序的某些位置惠根据配置项的不同选择不同的变元。对所有配置项的一个赋值称为一个配置。我们的目标需要对同一个软件在所有配置下的产品统一进行测试。

软件产品线测试的一个重要的研究方向就是如何快速地判断软件在所有配置下的产品都通过了测试。现有的技术中，主要使用的方法是变化监听执行技术（variability-aware execution）[21,22,23]。它通过复用不同产品之间的冗余执行来加速软件产品线测试。现有的变化监听执行技术所使用的主要方法都是重量级的，有些方法使用了一个特定的解释器[22]，另一些方法则将一组产品的执行编码成了一个模型检查器（model checker）[23]。

虽然软件产品线测试和变异测试有很大的关系，但是变化监听执行技术并不适用于变异测试。变异测试和软件产品线测试的不同之处在于规模大小。软件产品线的配置项是一个布尔向量，对于一个大小为 n 的布尔向量，其不同的取值可以有 2^n 个，也就是说软件产品的个数是指数级的，而变异测试中变异体仅在一

```
#if A && !B
  x++;
#else
  x--;
#endif
```

图 5.1: 软件产品线示例程序

```
#if A || B
...
#endif
#if A
...
#endif
```

图 5.2: 对 SPLat 的示例程序

处发生变异，即使是高阶变异算子其发生变异的变异点数量也不是一个很大的值，所以我们可以近似认为变异测试产生的变异体个数是 $O(n)$ 的，这个规模远小于软件产品线产品个数的规模。所以，变化监听执行技术所使用的方法在额外开销比较大，这对于大规模的软件产品线测试的加速效果并没有太大的影响，但是对于相对较小规模的变异测试，这个额外开销就会严重影响加速比。

与变化监听执行技术相比，动态变异测试技术更加轻量级，这个轻量级体现在以下两个方面：

1. 动态变异测试技术的额外开销很小，只需要创建新进程来执行变异体即可。即使变异体数量很小也可以适用。
2. 动态变异测试技术实现起来更加方便，不需要实现特定的解释器或者模型检查器。

所以，一个有意思的问题是，是否可以将动态变异测试扩展到软件产品线测试上？在接下来的内容中，我们将详细讨论这个问题。

5.2　动态变异测试在软件产品线测试上的扩展

由于我们已经为动态变异测试设计了抽象模型和动态算法，所以为了扩展动态变异测试技术到软件产品线测试，我们只需要说明我们所设计的抽象模型同样适用于软件产品线测试以及给出在软件产品线测试上的抽象模型的实现方法即可。

首先，我们说明动态变异测试的抽象模型同样适用于软件产品线测试。我们考察下面一个软件产品线程序的例子。

在图 5.1 的示例程序中，A 和 B 是程序的两个配置项，根据 A 和 B 的值，程序在某一个位置选择的语句是 `x++` 或者 `x--` 。和变异测试相同，有些语句中

可以有多个位置可以发生变元，我们也可以用现有的技术将这些语句分解为三地址语句[24,25]，所以我们可以假设在 `#if` 语句的两个分支下都只有完整的语句。

这个模型和我们的抽象模型是吻合的。每一个配置（布尔向量）可以看做是一个变异号，每一个配置对应的产品都可以看做是一个高阶变异体，每个 `#if` 语句所在的位置都可以看做一个位置和一组变元。每个变异体所对应的变元通过 `#if` 条件表达式定义。在示例程序中，第 1-5 行一起组成了一个位置，即变异点，其中第 2 行和第 4 行是两个变元。第 2 行的变元包含所有变异号满足 $\mathtt{A} \wedge \neg \mathtt{B}$，第 4 行的变元所包含的所有变异号满足 $\neg(\mathtt{A} \wedge \neg \mathtt{B})$，显然这两组变异号是交集为空集的。

接下来，我们讨论如何根据软件产品线测试的特点来调整 filter_variants 和 filter_mutants 算法。这里我们用算法 8 替代原有的 filter_variants 算法。

Input: V: 需要筛选的变元集合
Input: I: 用来筛选 V 的变异号集合
1 $V' \leftarrow \emptyset$
2 **for** 每个 $v \in V$ **do**
3 　**if** *satisfiable*$(I \wedge v.I)$ **then**
4 　　$V' \leftarrow V' \cup \{v\}$
5 　**end**
6 **end**
7 $V \leftarrow V'$

算法 8: 软件产品线 filter_variants 算法

首先，由于每个变元代表的变异号集合均为符合某个逻辑表达式的布尔向量的集合，所以我们可以用这个逻辑表达式来代表整个布尔向量的集合。所以在这里，$v.I$ 就是一个逻辑表达式。在第 3 行，我们借助 SAT 求解器来检查变异号是否满足变元的逻辑表达式，如果满足，那么就将此变元添加进去。

对于 filter_mutants 算法，我们只需要将需要筛选的变异号集合 I 更新为 $I \wedge \neg(v_1 \wedge \ldots \wedge v_n)$ 即可，其中 $V = \{v_1, \ldots, v_n\}$。

有趣的是，在软件产品线测试的加速技术中，SPLat（目前最著名的软件产品线测试加速技术）[26] 的思想与动态变异测试的思想有相似之处。SPLat 在执行软件产品的过程中对比当前的配置和当前的条件表达式，在执行过程中不去执行仅在与当前条件表达式无关的配置项存在不同的配置。而动态变异测试在执行过

程中，也仅对使得当前表达式的值发生改变的配置项进行新进程的创建。除此之外，相比于 SPLat，动态变异测试技术可以在此基础上进一步加速软件产品线测试的执行。

我们考察图 5.2 中的示例程序。我们假设程序在执行过程中访问到了这段代码。我们首先考察配置项 {A=1, B=1} 和 {A=1, B=0}，在 SPLat 技术中，这两个配置项会同时被执行，因为程序片段的两个表达式中包含了 A 和 B。然而虽然 B 出现在了第一个表达式中，但是当 $A = 1$ 的时候表达式的值是一样的，即这两个配置项是等价的，所以动态变异测试技术是不会新建进程执行第二个配置项，而会用同一个进程代替 A 和 B。

由此，我们已经说明了我们的动态变异测试技术是可以扩展到软件产品线测试上的，并且动态变异测试技术可以比现有软件产品线测试技术进一步加速软件产品线测试的执行。然而，具体的工具实现和实验评估，以及动态变异测试技术和变化监听执行技术在软件产品线测试上的效果对比等内容已经超出了本论文的范围，在此就不继续展开了。

结论与展望

本论文提出了动态变异测试技术，通过共享不同变异体之间的重复执行过程来加速变异测试的执行。我们的实验表明动态变异测试技术可以在目前最快的变异测试加速工具 Major Framework 上最高可进一步加速 2.220X。

本论文的主要贡献包括：

1. 提出了动态变异测试技术，这是一个全新的变异测试加速技术；
2. 设计了支持动态变异测试技术的抽象模型，并在此基础上提供了静态和动态变异测试技术的算法和一阶和高阶变异算子的筛选算法的实现；
3. 实现了动态变异测试工具 AccMut，并通过实验与现有变异测试加速工具 Major Framework 和 Mutation Schemata 进行了对比，进一步验证了加速效果；
4. 基于抽象模型，提出了动态变异测试技术在软件产品线测试上的扩展方法。

未来的工作包括：

1. 进一步完善工具，重写系统调用 fork()，真正实现文件 IO 的 “写时拷贝”；
2. 进一步扩大实验规模，从更多的领域寻找更多的实验对象；
3. 扩展工具功能，是 AccMut 支持高阶变异算子的变异测试；
4. 扩展工具功能，使 AccMut 支持软件产品线测试，并进行实验评估。

参考文献

[1] Richard A DeMillo, Richard J Lipton, Frederick G Sayward. Hints on test data selection: Help for the practicing programmer[J]. Computer. 1978, **11**(4):34–41

[2] Richard G Hamlet. Testing programs with the aid of a compiler[J]. Software Engineering, IEEE Transactions on. 1977, **SE-3**(4):279–290

[3] Yue Jia, Mark Harman. An analysis and survey of the development of mutation testing[J]. Software Engineering, IEEE Transactions on. 2011, **37**(5):649–678

[4] Mike Papadakis, Yves Le Traon. Using mutants to locate" unknown" faults[C]. ICST. 2012, 691–700

[5] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization[C]. ICST. 2014, 153–162

[6] Lingming Zhang, Lu Zhang, Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software[C]. Proc. OOPSLA. 2013, 765–784

[7] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, Stephanie Forrest. Automatically finding patches using genetic programming[C]. ICSE '09. 2009, 364–374

[8] Dongsun Kim, Jaechang Nam, Jaewoo Song, Sunghun Kim. Automatic patch generation learned from human-written patches[C]. ICSE '13. 2013, 802–811

[9] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, Chengsong Wang. The Strength of Random Search on Automated Program Repair[C]. Proceedings of

the 36th International Conference on Software Engineering. ICSE 2014, 2014, 254–265

[10] W. Weimer, Z.P. Fry, S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results[C]. Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. 2013, 356–366

[11] Mark Harman, Yue Jia, William B Langdon. Strong higher order mutation-based test data generation[C]. Proc. FSE. 2011, 212–222

[12] William E. Howden. Weak mutation testing and completeness of test sets[J]. IEEE Transactions on Software Engineering. 1982, **SE-8**(4):371–379

[13] W Eric Wong, Aditya P Mathur. Reducing the cost of mutation testing: An empirical study[J]. Journal of Systems and Software. 1995, **31**(3):185–196

[14] René Just, Michael D Ernst, Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states[C]. ISSTA. ACM, 2014, 315–326

[15] Lingming Zhang, Darko Marinov, Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction[C]. Proc. ISSTA. 2013, 235–245

[16] Randal E Bryant, O'Hallaron David Richard, O'Hallaron David Richard. Computer systems: a programmer's perspective[M]. Prentice Hall Upper Saddle River, 2003

[17] Roland H Untch, A Jefferson Offutt, Mary Jean Harrold. Mutation analysis using mutant schemata[C]. Proc. ISSTA. 1993, 139–148

[18] Rene Just, Franz Schweiggert, Gregory M Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler[C]. ASE. 2011, 612–615

[19] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, Hong Mei. Is operator-based mutant selection superior to random mutant selection?[C]. Proc. ICSE. 2010, 435–444

[20] Akbar Siami Namin, James H Andrews, Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness[C]. Proc. ICSE. 2008, 351–360

[21] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, Klaus Ostermann. Toward Variability-Aware Testing[C]. FOSD. 2012, 1–8

[22] Chang Hwan Peter Kim, Sarfraz Khurshid, Don Batory. Shared execution for efficiently testing product lines[C]. ISSRE. IEEE, 2012, 221–230

[23] Jens Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications[Z], 2014

[24] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, Thorsten Berger. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation[C]. OOPSLA. 2011, 805–824

[25] Paul Gazzillo, Robert Grimm. SuperC: Parsing All of C by Taming the Preprocessor[C]. PLDI. 2012, 323–334

[26] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, Marcelo D'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems[C]. ESEC/FSE 2013. 2013, 257–267

附录 A　AccMut 动态分析算法库核心部分代码实现

```
...
Mutation* ALLMUTS[MAXMUTNUM + 1];
int forked_active_set[21];
int forked_active_num;
int default_active_set[MAXMUTNUM + 1];
int recent_set[21];
int recent_num;
long temp_result[21];

typedef struct Eqclass {
    long value;
    int num;
    int mut_id[21];
} Eqclass;
Eqclass eqclass[21];
int eq_num;

// Algorithms for Dynamic mutation anaylsis
void __accmut__filter__variant(int from, int to) {
    recent_num = 0;
    int i;
    if (MUTATION_ID == 0) {
        recent_set[recent_num++] = 0;
        for(i = from; i <= to; ++i) {
            if (default_active_set[i] == 1) {
                recent_set[recent_num++] = i;
```

```
27              }
28          }
29      } else {
30          for(i = 0; i < forked_active_num; ++i) {
31              if (forked_active_set[i] >= from && forked_active_set[i] <= to) {
32                  recent_set[recent_num++] = forked_active_set[i];
33              }
34          }
35          if(recent_num == 0) {
36              recent_set[recent_num++] = 0;
37          }
38      }
39
40  }
41
42  void __accmut__divide__eqclass() {
43      eq_num = 0;
44      int i;
45      for(i = 0; i < recent_num; ++i) {
46          long result = temp_result[i];
47          int j;
48          int flag = 0;
49          for(j = 0; j < eq_num; ++j) {
50              if(eqclass[j].value == result) {
51                  eqclass[j].mut_id[eqclass[j].num++] = recent_set[i];
52                  flag = 1;
53                  break;
54              }
55          }
56          if (flag == 0) {
57              eqclass[eq_num].value = result;
58              eqclass[eq_num].num = 1;
59              eqclass[eq_num].mut_id[0] = recent_set[i];
60              ++eq_num;
61          }
62      }
63  }
64
65  void __accmut__filter__mutants(int from, int to, int classid) {
```

```
66      /** filter_mutants **/
67      int j;
68      if(eqclass[classid].mut_id[0] == 0) {
69          for(j = from; j <= to; ++j) {
70              default_active_set[j] = 0;
71          }
72          for(j = 0; j < eqclass[classid].num; ++j) {
73             default_active_set[eqclass[classid].mut_id[j]] = 1;
74          }
75      } else {
76          forked_active_num = 0;
77          for(j = 0; j < eqclass[classid].num; ++j) {
78              forked_active_set[forked_active_num++] = eqclass[classid].mut_id[j];
79          }
80      }
81  }
82
83  long __accmut__fork__eqclass(int from, int to) {
84      if(eq_num == 1) {
85          return eqclass[0].value;
86      }
87      int result = eqclass[0].value;
88      int id = eqclass[0].mut_id[0];
89      int i;
90
91      /** fork **/
92      for(i = 1; i < eq_num; ++i) {
93           int pid = 0;
94           pid = fork();
95           if(pid == 0) {
96               int r = setitimer(ITIMER_PROF, &tick, NULL);
97               __accmut__filter__mutants(from, to, i);
98               MUTATION_ID = eqclass[i].mut_id[0];
99               return eqclass[i].value;
100          } else {
101              waitpid(pid, NULL, 0);
102          }
103     }
104     __accmut__filter__mutants(from, to, 0);
```

```
105     return result;
106 }
107
108 int __accmut__process_i32_arith(int from, int to, int left, int right){
109     int ori = __accmut__cal_i32_arith(ALLMUTS[to]->op , left, right);
110     __accmut__filter__variant(from, to);
111     // generate recent_set
112     int i;
113     for(i = 0; i < recent_num; ++i) {
114         if(recent_set[i] == 0) {
115             temp_result[i] = ori;
116         } else if (ALLMUTS[recent_set[i]]->type == LVR) {
117             int op = ALLMUTS[recent_set[i]]->op;
118             int t_con = ALLMUTS[recent_set[i]]->t_con;
119             if(ALLMUTS[recent_set[i]]->op_index == 0) {
120                 temp_result[i] = __accmut__cal_i32_arith(op, t_con, right);
121             } else {
122                 temp_result[i] = __accmut__cal_i32_arith(op, left, t_con);
123             }
124         } else {
125             temp_result[i] =
126                 __accmut__cal_i32_arith(ALLMUTS[recent_set[i]]->t_op,
127                                         left, right);
128         }
129     }
130     if(recent_num == 1) {
131         if(MUTATION_ID < from || MUTATION_ID > to) {
132             return ori;
133         }
134         return temp_result[0];
135     }
136     /* divide */
137     __accmut__divide__eqclass();
138     /* fork */
139     int result = __accmut__fork__eqclass(from, to);
140     return result;
141 }
142 ...
```

致谢

首先我要感谢熊英飞老师给我提供了如此有意思的研究课题，以及在研究和论文写作过程中对我悉心的指导。同时，我要感谢王博师兄在环境配置和工具使用中给我带来的便利，以及软工所“程序语言与开发环境研究小组”所有组员在我研究中给予我各种有用的建议。没有你们的帮助，我的研究也不可能达到现在的高度。

感谢北京大学，北京大学信息科学技术学院，以及北京大学软件研究所为我的研究提供的良好环境和资源。感谢未名 BBS 的 thesis 版面所提供的论文 LaTeX 模板。

感谢在大学四年间所有给过我帮助，给我带来快乐的同学，尤其是 39 号楼 628 的三位室友刘敏行，陈庆英和姜双，以及 520 宿舍的黄茗。你们为我的大学生活增添了不少色彩。

最后，感谢我的家人给我精神上和财产上的支持和关心。感谢身处异地的女友对我的坚持和鼓励。大学四年一路走来，你们是我最坚实的后盾。

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