北京大学信息科学技术学院

本科生毕业论文

变异测试的动态加速技术：设计与实现

1200012741
史杨勋惟

指导老师：熊英飞

(2016 年 4 月 27 日)

摘要

变异测试是一种在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。变异测试一个很大的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的加速技术，例如移除冗余的变异体等。然而，这些技术都是静态的，所以无法消除在变异体执行过程中的冗余部分。

本论文的目标是设计一个变异测试的动态加速技术：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。基于此技术，本论文在 LLVM 的框架上实现了一个 C 语言变异测试的加速工具 AccMut，并将此工具与现有的加速技术进行了对比和验证。实验表明动态加速技术加速效果显著，加速比是 Major Framework（目前最快的静态变异测试加速技术）的 X 倍。

关键词： 变异测试，动态加速，静态加速

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 to accelerate 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. Based on this techinque, we developed an acceleration tool "AccMut" on C programming language on top of LLVM and compared it with other techniques. Our experiment show that our approach can accelerate mutation analysis significantly, having a speedup up to x.xxX over Major Framework, a state-of-the-art tools of static acceleration.

Keyword: mutation analysis, dynamic acceleration, static acceleration

目录

一　引言

1.1　变异测试

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]。图 X 描述了变异测试的整个流程。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 [4]。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。变异测试还有其他的用途，例如缺陷定位 [8, 7, 14] 和缺陷修复 [11, 6, 9, 10] 等。

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）。变异测试生成的大部分变异体为一阶变异体（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（即变异体和原程序有几处不同）。高阶变异测试。

1.2 变异测试的瓶颈

变异测试有一个重要的瓶颈：可扩展性较差。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过程需要在 $n \times m$ 倍的程序执行时间。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时间就会显著增加。这也是变异测试在实践中只是被小规模采用的原因。

1.3 相关工作

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。这些方法可以分为有损加速和无损加速。

1.3.1 Weak Mutation

Weak Mutation[3] 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）。

1.3.2 Mutation Sampling

Mutation Sampling [12] 是另一种有损加速技术，它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。

1.3.3 Major Framework

Major Framework [5] 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，对剩变异体进行等价类划分。Major Framework 按照下面三个标准进行等价类划分：

- 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。
- 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。
- 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 逐一执行测试用例的每组测试用例。在一个等价类中任意选出一个一个变异体执行该测试用例，而不需要在其他变异体上执行了。这样就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。

1.3.4 Mutation Schemata

Mutation Schemata 从编译时间上加速了变异测试。由于每个变异体都是一个新的程序，所以编译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，只编译一次，节省了大量的编译时间。

1.3.5 Test Prioritization

Test Prioritization [13] 针对不同变异体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，这样就不用执行测试集中的其他测试用例了。此方法并没有具体工具的实现。

1.4 本论文的目标

以上的加速技术有一个共同的不足：它们都是静态加速的方式。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，而静态加速的方法无法消除程序执行过程中（图 X 的 X 部分）的冗余部分，导致这个过程被重复执行了两遍。

本论文的目标是设计一个变异测试的动态加速技术：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，此动态加速技术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。这个方法是无损的，这个方法有以下两个优点：

- 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。从而节省了大量的执行时间。
- 此方法将所有变异整合到了同一个程序中，无需编译多次，这是 Mutation Schemata 的优点。在动态执行过程中可以根据即时的结果进行等价类的划分，复用等价变异体的执行，这是 Major Framework 的优点。所以此方法集合了现有的两大静态加速技术，进一步提升了加速比。

本论文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，设计了动态变异测试的算法。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，并同时在 LLVM 的框架上复现了现有的加速工具（Schemata，Major Framework）用来进行横向对比和验证。实验表明动态加速技术加速效果显著，加速比是 Major Framework（目前最快的静态变异测试加速技术）的 X 倍。

说明

我的本科生科研的课题也是变异测试的加速，本论文和本科生科研论文的主要区别在于等价类划分算法的实现和 IO 支持上。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；另一方面，本论文实现的工具已经支持下涉及读写独立的文件 IO 的程序，而在本科生科研实现的工具中并不支持。在实验的规模上，本论文也超出了本科生科研时后的规模。总的来说，本论文在本科生科研的成果上做了许多扩展和修改工作。

二　方法

2.1　加速原理

图 X 描述了我们的动态加速技术如何消除执行过程中的冗余过程。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，是的系统状态转换为了两个不同的系统状态（状态 3 和状态 5）。其中，虽然其中两个变异体（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。这是可能发生的，比如当 $a == 2$ 的时候，语句 $a+ = 2$ 和语句 $a* = 2$ 执行后的效果是一样的。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，转换后的系统状态不想同。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。

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，只有其中一部分使可以消除的，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，动态加速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 X。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 H、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。根据不同地结果，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。在这个离子重，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，此新进程有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 X 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。

2.1.1 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 支持）。当我们调用 fork() 函数的时候，系统会创建一个新的子进程，这个子进程与创建它的进程（父进程）具有相同的系统状态（包括虚拟内存空间和栈帧空间）。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。我们也是基于这个返回值来确定当前执行的是父进程还是子进程。

POSIX 系统调用 fork() 通过 “写时拷贝” 机制来实现：当 fork() 被调用的时候，系统会创建一个新的进城对象并且赋予其一些基本信息，例如进程号等，但是并没有马上从父进程拷贝虚拟内存空间而是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，系统才会对所写页创建一份新的拷贝，并且更新子进程的页表信息。“写时拷贝” 机制被集成到了虚拟内存访问的过程中，所以系统调用 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距。

2.2 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变异测试的动态加速技术。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。

2.2.1 变异过程的抽象模型

给定一个程序，变异测试首先通过一些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 是一个从位置到位变异（variant）集合的映射。一个位变异 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。这个变异体集合包含了所有包含 $v.code$ 的变异体。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 “映射后位变异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。同时，对于同一个位置经过 p 过程映射后位变异集中的任意两个不同位变异 v_1 和 v_2，它们包含的变异体集合的交集是空集，也即 $v_1, v_2 \in p(l) \Rightarrow v_1.I \cap v_2.I = \emptyset$。给定一个变异号 i，从每个位置映射后的位变异集中，取出变异体集合中包含 i 的位变异，并将这些位变异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：

```
1: a = a + 1;
2: 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位变异:

(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位变异：

(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）。

2.2.2 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，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 `dtry` 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。

说明：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。

此处将给出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。

Input: p: a mutation procedure

Data: s: the current system state

1 **for** *each mutant ID* i *in all mutant IDs* **do**
2 $s \leftarrow$ the initial system state
3 **while** $\phi(s) \neq \perp$ **do**
4 $\{v\} \leftarrow$ filter_variants$(p(\phi(s)), \{i\})$ execute$(v.code, s)$
5 **end**
6 save(s, i)
7 **end**

Algorithm 1: Static Mutation Analysis (with Mutant Schemata)

2.3 算法

2.3.1 静态算法

2.3.2 动态算法

核心算法

一阶变异算子的分类算法

2.3.3 算法分析

算法的正确性

算法的复杂度

Input: p: a mutation procedure
Data: s: the current system state
Data: I: a set of mutant IDs represented by the current process

```
1  I ← all mutant IDs
2  s ← the initial system state
3  while φ(s) ≠ ⊥ do
4  |  proceed(p(φ(s)))
5  end
6  for i ∈ I do
7  |  save(s, i)
8  end
```

Algorithm 2: Main Loop of Dynamic Mutation Analysis

Input: V: a set of variants at current location
Data: s: the current system state
Data: I: a set of mutation IDs represented by the current process

```
1  filter_variants(V, I)
2  v_cur ← a random element in V
3  for each v in V − {v_cur} do
4  |  pid ← fork()
5  |  if pid = 0 then
   |  |  // Child process
6  |  |  filter_mutants(I, {v})
7  |  |  execute(v.code)
8  |  else
   |  |  // Parent process
9  |  |  filter_out_mutants(I, {v})
10 |  end
11 end
12 execute(v_cur.code)
```

Algorithm 3: Basic Forking of proceed(s)

Input: V: a set of variants at current location

Data: s: the current system state

Data: I: a set of mutation IDs represented by the current process

```
1  filter_variants(V, I)
2  if |V| = 1 then
3      v ← the only variant in V
4      execute (v.code, s)
5      return
6  end
7  X = ∅
8  for each v in V do
9      X ← X ∪ {try(v.code, s)}
10 end
11 𝕏 ← group changes in X into equivalent classes
12 X_cur ← any one of the equivalent classes in 𝕏
13 for each equivalence class X in 𝕏 − {X_cur} do
14     V ← the variants corresponding to changes in X
15     pid ← fork()
16     if pid = 0 then
           // Child process
17         filter_mutants(I, V)
18         x ← a random change in X
19         apply(x, s)
20     else
           // Parent process
21         filter_out_mutants(I, V)
22     end
23 end
24 x ← a random change in X_cur
25 apply(x, s)
```

Algorithm 4: Adding Equivalence Analysis to proceed(s)

Input: V: a set of variants to be filtered
Input: I: a set of mutation IDs used to filter V

```
1  V' ← ∅
2  v' ← the variant in V with DefaultSet
3  for each v ∈ V where v ≠ v' do
4  |  if I.contains(v.I.first) then
5  |  |  V' ← V' ∪ {v}
6  |  end
7  end
8  if V'.size < I.size then
9  |  V' ← V' ∪ {v'}
10 end
11 V ← V'
```

Algorithm 5: filter_variants

Input: I: a set of mutation IDs to be filtered
Input: V: a set of variants used to filter I
Input: L: the current location

```
1  if V contains the variant with DefaultSet then
2  |  for each v ∈ p(L) − V do
3  |  |  I.remove(v.I.first)
4  |  end
5  else
6  |  I ← a new empty ListSet
7  |  for each v ∈ V do
8  |  |  I.add(v.I.first)
9  |  end
10 end
```

Algorithm 6: filter_mutants

三　工具实现

选择 C 语言，LLVM 上进行了实现。新增两个 Pass，插桩，库，执行，图 X

3.1　生成变异的 Pass

3.2　插桩的 Pass

3.3　动态分析算法的库

3.4　文件 IO 的支持

四　实验测试

4.1　实验对象

4.2　实验流程

4.3　实验结果

五　未来扩展：软件产品线测试

抄写熊老师

六　结论

重写一下 Intro

参考文献

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

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

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

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

[5] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*, pages 315–326. ACM, 2014.

[6] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, pages 802–811, 2013.

[7] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162, 2014.

[8] M. Papadakis and Y. Le Traon. Using mutants to locate" unknown" faults. In *ICST*, pages 691–700, 2012.

[9] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, 2014.

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

[11] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374, 2009.

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

[13] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*, pages 235–245, 2013.

[14] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proc. OOPSLA*, pages 765–784, 2013.

致谢