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 25 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 **5**

1.1 变异测试 5

1.2 变异测试的瓶颈 6

1.3 相关工作 6

1.3.1 Weak Mutation 6

1.3.2 Mutation Sampling 6

1.3.3 Major Framework 6

1.3.4 Mutation Schemata 7

1.3.5 Test Prioritization 7

1.4 本论文的目标 7

二 变异测试的动态加速技术 **9**

2.1 加速原理 9

2.1.1 相关背景：系统调用 Fork 9

2.2 抽象模型 9

2.3 算法 9

2.3.1 静态算法 9

2.3.2 动态算法 9

三 工具实现 **10**

3.1 生成变异的 Pass 10

3.2 插桩的 Pass 10

3.3 动态分析算法的库 10

3.4 文件 IO 的支持 10

四 实验测试 **11**

4.1 实验对象 11

4.2 实验流程 11

4.3 实验结果 11

五 未来扩展：软件产品线测试 **12**

六 结论 **13**

致谢 **16**

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（插入或删除一条指令），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　变异测试的动态加速技术

.5

2.1　加速原理

不同的变异测试总的来说，通过动态的方法，加速了执行，X 的阶段。图 X 是本算法的一个示例。

2.1.1　相关背景：系统调用 Fork

XXX

2.2　抽象模型

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。

2.3 算法

2.3.1 静态算法

2.3.2 动态算法

核心算法

分类算法

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