中文图书分类号：TP311
密　　级：公开
UDC：004
学　校　代　码：10005

硕士专业学位论文

PROFESSIONAL MASTER DISSERTATION

论　文　题　目：基于代码转换的 MC/DC 测试用例生成技术

论　文　作　者：

专业类别/领　域：计算机技术

指　导　教　师：

论文提交日期：2019 年 3 月

UDC：004
中文图书分类号：TP311
学校代码：10005
学　　号：S201607086
密　　级：公开

北京工业大学硕士专业学位论文

（全日制）

题　　目： 基于代码转换的 MC/DC 测试用例生成技术

英文题目： THE GENERATION TECHNOLOGY OF MC/DC TEST CASE BASED ON CODE TRANSFORMATION

论 文 作 者：

专业类别/领 域： 计算机技术

研 究 方 向： 计算机软件技术

申 请 学 位： 工程硕士专业学位

指 导 教 师：

所 在 单 位： 信息学部

答 辩 日 期： 2019 年 5 月

授予学位单位： 北京工业大学

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签　名：_____

日　期：　　年　月　日

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签　名：_____　　日　期：　　年　月　日

导师签名：_____　　日　期：　　年　月　日

摘 要

近年来随着计算机软件行业的飞速发展，计算机软件系统日益复杂，软件质量问题已经严重制约了计算机技术的发展。软件测试是保证软件质量和可靠性的主要手段，这对于航空航天、自动驾驶、医疗等领域中使用的安全关键软件系统尤为重要。这些系统需要极高的可靠性，只要发生故障就可能会导致灾难性的后果。因此人们制定了软件认证标准，对安全关键系统提出了一系列严格的要求，在软件测试时必须予以考虑。例如在航空软件认证标准中有一条要求仅适用于安全关键系统，即用于测试这些系统的测试用例必须满足修订的条件/判定覆盖（MC/DC）准则。

MC/DC 准则是一种比较严格的代码覆盖准则，旨在证明判定中涉及的所有条件都可以影响该判定的结果。目前，测试人员普遍采用手工方法设计测试数据。但是手工生成测试数据的效率低下、容易出错，导致测试成本昂贵，很难保证软件质量。动态符号执行技术能够针对被测程序自动生成测试用例，该技术降低了由手工测试而产生的高额成本，大幅提高了测试效率，近年来被广泛使用。但是，现有的动态符号执行技术以分支覆盖为目标进行测试生成，因此不能生成满足 MC/DC 准则的测试用例。

为了解决这个问题，本文提出了一种基于代码转换的测试用例生成方法，以优化现有的使用动态符号执行技术的测试生成过程，使其能够生成满足 MC/DC 准则的测试用例。本方法对待测程序的源代码进行分析，针对判定表达式生成条件约束，并通过代码转换的方式将条件约束添加到程序代码中。在使用动态符号执行技术进行测试用例生成时，这些条件约束会指导符号执行技术探索特定的路径，最终生成满足 MC/DC 准则的测试用例。本文基于 Clang 编译器设计并实现了一个代码转换工具，自动完成上述工作，并将该工具与动态符号执行工具 KLEE 相结合，生成满足 MC/DC 准则的测试用例。

本文将该方法应用于 14 个小型程序和 12 个 Coreutils 程序。实验结果表明，本方法能有效提高测试用例的 MC/DC 覆盖率。对于小型程序，应用本文方法生成的测试用例 MC/DC 覆盖率可以达到 100%；对于 Coreutils 程序，与未应用本方法相比，应用本文方法生成的测试用例 MC/DC 覆盖率平均提高了 14.66%，达到 98.01%。

关键字：测试数据生成；MC/DC；源代码转换

Abstract

With the rapid development of recent computer software industry, the computer software system is increasingly complex, and the quality issue of software has become a major restriction of the development of computer technology. Software testing is a major method to evaluate software quality and reliability, which is a vital process for the safety-critical software system used in the aerospace, autonomous driving, medical and other fields. These systems need extremely high reliability otherwise it can lead to disastrous consequences. Therefore, people formulated software certification standards and proposed a set of rigorous requirements for the safety-critical system, and these requirements have to be considered in software testing. For example, one requirement of the aviation software certification standard that is suitable for the safety-critical system means that the test cases for testing these systems must fulfill the modified condition/decision coverage (MC/DC) criterion.

The MC/DC criterion is a rigorous code coverage criterion that aims to identify all the involved conditions can affect the outcome of the decision. Currently, a majority of testers design test data manually. However, the manual generation of test data can be inefficient and error-prone, resulting in high test cost and difficulty in guaranteeing software quality. The dynamic symbolic execution technology can automatically generate test cases for the under-test software. This technology can reduce the high cost caused by manual testing and largely improves test efficiency, and this technology has been widely adopted in recent years. However, the existing dynamic symbolic execution technique performs test generation with a target of branch coverage, which cannot fulfill the MC/DC criterion.

To solve this problem, this thesis proposes a test case generation approach based on the code transformation, optimizing the current test generation process with dynamic symbolic execution technology, so that generating test cases that fulfill the criterion of MC/DC. This research analyses the source code of the program to be tested and generates conditional constraints for decisions, then adding these constraints to the source code by code transformation. When the dynamic symbolic executing technology generating test cases, these conditional constraints could guide the symbolic execution technology to discover specific routes and consequently generate the test cases that fulfill the MC/DC criterion. This thesis designs and implements a code

transformation tool based on Clang compiler. The code transformation tool could finish above-mentioned work automatically and combine this tool with the dynamic symbolic execution tool KLEE, finally generating the test cases that fulfill the MC/DC criterion.

This thesis applies the mentioned approach in 14 small programs and 12 Coreutils programs. The results of a serial of experiments indicate that the proposed approach can enhance the coverage ratio of MC/DC for test cases effectively. A coverage ratio of 100% was achieved when generating test cases for small programs with the proposed approach. Moreover, compared with pure dynamic symbolic execution, the MC/DC coverage of test cases generated by the proposed approach for Coreutils programs averagely increased by 14.66%, reached 98.01%.

Keywords: Test data generation, MC/DC, source code transformation

目 录

第1章　绪论

1.1 课题研究背景及意义

近年来随着计算机软件行业的飞速发展，社会各行业的生产方式也产生了巨大的变革。计算机软件在社会活动、经济产业中所起的作用越来越大。越来越多的行业领域依靠计算机软件来辅助实现安全控制、业务执行和生产管理等重要环节，软件质量问题已经严重制约了计算机技术的发展[1]。重要领域中运行的软件如果出现错误，就会给社会经济和人类生活造成巨大损失。尤其是对于关乎人类安全的关键应用，比如航空航天、医疗以及其它的控制系统等，软件一旦出现错误，其灾难性的后果不可估量，这使得对软件安全性的研究成为一项迫切的要求。目前针对软件漏洞检测的研究，主要有静态源代码检测、黑盒模糊测试、污点分析、符号执行几个方面[1]。国内外针对上述各种软件测试技术都进行了大量的研究，推动了软件测试技术的发展。目前普遍使用传统的手工测试和随机测试来生成测试用例[2]，然而这种方法只能检测程序所有可能路径中很小的一部分[3]。从目前的研究来看，动态符号执行技术[4-6]是比较有发展前景的一项软件自动化测试技术。

符号执行技术是 James C. King 在上世纪 70 年代提出的一种技术[4]，该技术在测试过程中分析源代码，将代码中的变量值替换为符号，模拟程序执行流程。符号执行技术最初应用于单元代码静态分析工具，受到当时硬件能力和约束求解能力不足的限制，符号执行技术的应用十分有限。随着计算机硬件能力的不断提升，以及约束求解技术的发展，更高效的约束求解工具被开发出来。符号执行技术发展出与程序实际执行相结合的动态符号执行技术，应用范围得到了很大的扩展，成为当前广泛应用的程序正确性检测技术。动态符号执行不再是静态的对代码进行分析、模拟执行，而是在程序实际运行中为程序提供约束求解器求解出来的具体值，并实时分析程序中可能存在的漏洞[7]。在遇到分支路径时，保存分支的约束条件，调用约束求解器求解约束，再通过路径调度算法，代入满足约束条件的值以进入该分支，并对执行过的程序路径进行记录，下一次执行到该分支时选择未进入的分支执行，这样逐步对程序中所有的可行路径进行遍历并结合漏洞检测工具进行漏洞检测。

近年来，动态符号执行测试生成技术成为了软件测试领域的研究热点，随之出现了一系列动态符号执行的工具，如伊利诺伊大学的 CUTE[6]、斯坦福大学的 KLEE[8]、加州大学伯克利分校的 CREST[9]以及微软公司的 SAGE[10]等等。随着

理论研究的深入和工业应用的发展，动态符号执行的理论基础和技术水平都逐渐成熟。KLEE 是一款自动化软件测试工具，其使用约束求解器 STP(Simple Theorem Prover)[11, 12]，采用符号执行技术。目前 KLEE 在测试复杂的系统软件上能够生成覆盖率很高的测试用例，取得了良好的表现。

在测试航空航天、自动驾驶、医疗等领域的那些只要出错就会带来灾难性后果的安全关键系统时，只考虑程序结构的测试技术往往是不够充分的。因此，人们提出了一些更加复杂的逻辑覆盖准则，包括条件覆盖准则、判定覆盖准则、条件/判定覆盖准则、多重条件覆盖准则和修订的条件/判定覆盖准则（MC/DC，Modified Condition/Decision Coverage）。

在基于代码的测试准则中，MC/DC 准则[13]是一种较严格的覆盖准则，其目的是涵盖所有程序逻辑语句，并分析每个条件对整个程序行为的影响。有实验结果显示，除了 MCC 以外，MC/DC 准则具有比上述所有准则更强的错误检测能力[14]。MC/DC 准则广泛运用在航空航天软件的检测方面，例如：要获得美国联邦航空管理局的认证[13, 15, 16]，航空软件必须遵循由航空无线电技术委员会颁布的《机载系统和设备合格审定中对软件的要求》（DO-178B，Software Considerations In Airborne Systems And Equipment Certification）文件[13]中定义的质量和安全标准。为了防止关键的 A 级软件（在这个级别的软件故障可能会导致灾难）发生故障，DO-178B 文件规定所有这些软件都必须符合 MC/DC 准则[13]。

现有的动态符号执行技术以分支覆盖为目标进行测试生成，因此不能生成满足 MC/DC 准则的测试用例。为了解决这个问题，本文提出了一种基于代码转换的测试用例生成方法，以优化现有的使用动态符号执行技术的测试生成过程，使其能够生成满足 MC/DC 准则的测试用例。本文提出的方法能提高航空航天领域中关键软件系统的测试效率，在软件测试自动化领域有着十分重要的现实意义。

1.2 相关研究

1.2.1 动态符号执行技术

随着对软件安全性日益提升的重视与软件规模的不断发展，传统的软件测试技术已经无法满足人们的需求。因此，将静态检测技术与动态执行相结合的动态符号执行技术得到了众多高校和企业的关注与研究。近年来，已研究提出了多种利用动态符号执行技术实现的软件工具，并在工业应用中取得了良好的效果。

2005 年，贝尔实验室的 Patrice Godefroid，Nils Klarlund 和 Koushik Sen 共同发表的文章提出了一种自动化软件测试工具 DART（Directed Automated Random

Testing）[5]。DART 融合了三种技术：一是采用静态分析源代码，自动抽取程序与外部环境交互的接口；二是为接口自动生成模拟外部环境操作的随机测试数据；三是动态分析程序在随机测试下的行为，自动生成进入不同分支的测试输入来引导测试程序中的不同路径。DART 工具使单元测试不再需要为测试一个程序功能而编写大量的模拟程序与环境交互的代码。

CUTE（Concolic Unit Testing Engine for C）是以伊利诺亚大学的 Koushik Sen 等人研究并提出的针对 C 语言的单元测试工具。针对 DART 中存在的动态数据结构指针操作（例如指针可能存在有别名）可满足性不确定的问题，Koushik Sen 等人提出“逻辑输入映射”（logical input map）技术。其核心是使用逻辑输入映射来表示包括内存图（memory graphs）在内的所有的输入，消除了指针别名的影响，使得 CUTE 支持自动化测试含有动态数据结构的程序[6]。

KLEE 及其前身 EXE（EXecution generated Executions），是由斯坦福大学的 Cristian Cadar，Daniel Dunbar 与 Dawson Engler 合作研究的自动化软件测试工具。EXE 使用约束求解器 STP，采用符号执行技术。在程序运行时，EXE 使用可被初始化为任意值的符号输入代替具体输入。在运行中，当 EXE 检查到程序在操作这些符号值时，EXE 使用输入约束来代替程序的操作。当程序在条件分支语句中检查符号的值时，EXE 为两个分支生成两个约束表达式[17]。当程序在某一分支出现错误时，EXE 对该路径下约束条件集进行求解，自动生成重现该错误的测试输入。KLEE 在 EXE 的基础上，对约束求解器进行了优化：简化约束集、符号数值具体化以及独立约束条件优化，进一步提高了约束求解器的工作能力，使得 KLEE 在测试复杂的系统软件上取得了更好的表现。从对 Unix Coreutils 套件的 89 个程序进行测试的结果看来，KLEE 的平均代码覆盖超过了 90%，且从中发掘出 3 个隐藏超过 15 年的程序错误。

对于符号执行技术的研究，还有许多科研团队开发了相关工具，如加州大学伯克利分校的 CREST 及 IBM 公司的 Apollo 等等。众多高校和科技企业从科学理论的研究到商业工具的研究与改进，使得符号执行技术得到不断的发展。

1.2.2　MC/DC 准则测试用例生成技术

MC/DC 准则是一种实用的软件结构覆盖率测试准则，广泛应用于软件验证和测试过程中。根据航空无线电技术委员会发布的 DO-178C 文件，MC/DC 覆盖准则有如下要求：程序的每个入口点和出口点至少被唤醒一次；每个条件的所有可能结果至少出现一次；每个判定本身的所有结果也至少出现一次；并且判定中的每一个条件能够独立地影响判定的结果，即在其它条件不变的前提下仅改变这

个条件的值就能使判定结果改变[16]。

随着 MC/DC 准则被广泛应用于高复杂性、高可靠性的关键软件测试中，为了提高测试效率与质量，减少软件测试的开销，国内外学者对基于 MC/DC 的软件测试用例生成展开了一系列研究。

1994 年 John Joseph Chilenski 和 Steven P. Miller 发表论文介绍了 MC/DC 的概念，并通过对 MC/DC 覆盖、语句覆盖、判定覆盖、条件/判定覆盖、多条件覆盖对复杂逻辑表达式的敏感度进行理论比较分析，提出 MC/DC 准则可以提高测试用例发现错误的机率[18]。

2008 年 Ajitha Rajan 和 Michael W. Whalen 使用六个民用航空软件和两个小程序作为被测程序，证明了 MC/DC 准则对程序结构的高敏感度[19]。所以，MC/DC 准则适用于程序结构复杂的软件测试。

由于 MC/DC 准则适用于结构复杂的软件测试，并且可以提高发现错误的机率，基于 MC/DC 准则的测试用例生成技术成为软件测试研究的热点。

NASA（美国国家航空航天局）提供了两种生成 MC/DC 最小测试用例集的方法。我们通常称之为唯一原因法、屏蔽法。唯一原因法即是在其它条件保持不变的前提下，改变其中一个条件的值即可影响整个判定的结果，强调的是其它条件必须保持一致[20]。屏蔽法指对一个条件的特定输入能隐藏该布尔表达式中其它条件的所有输入，两种方法都可借助真值表进行分析。然而，唯一原因法和屏蔽法要借助于真值表，人工设计时较为复杂，导致效率低下且出错率高，为了解决这个问题，学者们提出了基于图的 MC/DC 最小测试用例集快速生成算法[21]。首先将布尔表达式转换成语法树，其次，将语法树按一定规则转换为不带逻辑算子的图，最后，通过遍历给图中的每个顶点按规则设置默认值，生成关键路径集之后根据关键路径快速生成最小测试用例集。该方法生成最小用例集的过程简洁直观，更适合人工生成测试用例[22]。之后，又有人提出利用二叉树法，时序逻辑变换算法[23]、递归分块矩阵生成算法等生成 MC/DC 最小测试用例集。

M Harman 等人在 2004 年提出了一种可测试性转换理论[24]：从转换的程序中生成测试数据，但在转换的过程中保证测试数据对于原始程序是足够的。可测试性转换只是达到目的的一种手段，一旦生成测试数据，转换后的程序就可以被丢弃。基于可测试性转换理论，R Pandita 等人[25]和 S.Godboley 等人[26]在 2010 年和 2013 年分别提出一种测试生成方法，用来协助以语句或分支覆盖为目标的现有测试生成方法，使其可以生成满足 MC/DC 准则的测试数据。但是他们没有给出正式的转换规则。

在 2014 年，T Wu 等人和 T Su 等人分别提出了一种新的测试生成方法，以找到合适的 MC/DC 测试数据[27, 28]。他们首先从目标程序中提取路径，然后找到

适当的测试数据来触发这些路径，并在路径提取过程中使用一个贪婪的策略来确定下一个选择的分支。他们的区别是前者使用的是静态符号执行技术[28]，而后者使用的是动态符号执行技术[27]。但是在他们的方法中，路径搜索算法的好坏直接决定了生成测试数据的质量，对于那些未能探索到的数据，这两种方法并不能生成对应的 MC/DC 测试用例。

据目前调研所知，现有的一些自动化测试生成工具，例如以 KLEE[29]为代表的动态符号执行工具，其生成的测试数据并不能达到 MC/DC 覆盖准则。随着对测试用例自动生成方法的不断研究，学者们正在探索基于 MC/DC 覆盖准则的测试用例自动生成算法。

1.3 课题来源

本工作得到国家自然科学基金（项目编号：61672505）、中国科学院前沿科学重点研究项目（项目编号：QYZDJ-SSW-JSC036）的资助。

1.4 本文研究工作和创新点

为了能够提高软件测试的质量和效率，本课题着重研究一种可以自动化生成符合 MC/DC 覆盖准则的测试生成技术。

本课题使用代码变换的方法，按照一系列代码转换规则，对待测程序的代码进行代码转换，将转换后的程序作为 KLEE 等现有符号执行工具的输入，自动生成测试用例，使得测试用例对于转换后程序的分支覆盖率达到 100%时，该测试用例对原待测程序的 MC/DC 覆盖率也能达到 100%。为了实现该目标，本课题的主要研究内容分为三部分：

1．如何将分支覆盖问题转换为 MC/DC 覆盖问题

为了使 KLEE 等分支覆盖测试生成工具生成的测试用例能够满足 MC/DC 覆盖准则，需要设计一套代码转换规则，通过在待测程序中插入若干代码使之变为一段新的程序，进而将针对待测程序的 MC/DC 覆盖测试用例生成问题转化为针对目标程序的分支覆盖测试用例生成问题。

2．如何按照指定的代码转换规则进行代码转换

在已有代码转换规则的条件下，需要实现一个代码转换器，分析代码并按照代码转换规则进行代码转换。代码转换器的输入为待测程序，输出为按照规则转换后的目标程序。

3．如何自动化生成 MC/DC 覆盖测试用例

本课题的目标是能自动化的生成符合 MC/DC 测试准则的测试用例，所以需要将代码转换器与自动化的测试生成工具 KLEE 集成。代码转换器转换后生成的目标程序应该能够作为 KLEE 的输入，并自动进行测试用例的生成。

本课题旨在克服现有技术中存在的问题，提供一种基于代码转换的符合 MC/DC 准则的软件测试数据生成方法。该方法通过规定一系列代码转换规则，对待测程序的代码进行转换，把转换后的程序作为符号执行测试生成工具的输入，使得测试数据集合对于转换后的程序满足分支覆盖准则时，测试数据集合对于待测程序满足 MC/DC 准则。

1.5 本文组织结构

本文组织结构如下：

第一章介绍了课题背景、研究目的，以及动态符号执行技术、测试用例生成技术等相关领域的研究现状。

第二章主要介绍了软件测试和 MC/DC 覆盖准则，符号执行技术等相关概念，并介绍了可测试性转换理论。

第三章介绍了基于代码转换的 MC/DC 测试用例生成方法，并通过一个示例展示整个过程。

第四章介绍了代码转换工具的实现方式，并对各个模块的功能做出说明。

第五章给出了针对本文所提出的基于代码转换的 MC/DC 测试用例生成技术的实验评估，包括实验方案、实验结果以及分析。

结论部分对本文的工作进行了总结，并给出对未来研究工作的展望与设想。

第2章　背景知识

2.1　MC/DC 覆盖准则

本节将会介绍软件测试和几个常用的代码覆盖率准则，提供一些基本概念的定义，介绍修订的条件/判定覆盖准则，最后介绍修订的条件/判定覆盖准则在航空航天等领域中的应用。

2.1.1　软件测试

软件测试是一个过程，在这个过程中对程序进行执行操作，并检查软件的实际功能与需求是否一致[30]。尽管软件测试永远不能保证程序中没有错误，也不可能用所有可能的输入值组合来执行程序并在每种情况下验证操作是正确的，但它是提高软件系统质量的一种实用且广泛使用的方法。软件测试分为黑盒测试和白盒测试。黑盒测试又叫功能测试，是一种从用户的角度出发，通过使用软件的功能进行测试的方法，测试者并不能检查程序的源代码或了解程序的设计细节。白盒测试也称为结构测试，测试者可以获得程序的源代码，了解设计细节，并根据程序的控制结构设计测试用例，这种方法主要用于软件或程序验证。由于本课题主要研究如何生成用于航空软件认证的 MC/DC 测试用例，需要对程序的逻辑结构进行分析，因此本课题的关注重点是白盒测试。

由于在软件测试的过程中覆盖一个程序所有的可执行路径通常是很困难的，因此可以使用代码覆盖率来衡量程序源代码被测试的程度。如果测试集中的执行覆盖了程序的所有可能执行的充分子集，则称测试过程（一组程序执行）实现某种代码覆盖。构成“充分”的内容实际上取决于所选的覆盖准则，从覆盖所有单个语句到覆盖程序中的所有执行路径，或是它们的组合。满足更复杂的覆盖准则需要在测试过程中进行更多的工作，但满足更复杂覆盖准则的测试用例，在测试过程中也会提高发现错误的可能性[31]。

2.1.2　代码覆盖准则

常用的代码覆盖准则大多属于基于控制流的覆盖准则，例如语句覆盖、分支覆盖、条件覆盖、路径覆盖、条件/判定覆盖、多条件覆盖和 MC/DC 覆盖等[32]。为了更好地说明代码覆盖准则，本节首先给出几个基本概念。

定义 2.1（布尔表达式）：布尔表达式是计算结果为 ture 或 false 的表达式，其中 true 也可以用 T 或者 1 表示，false 也可以用 F 或者 0 表示。在本文中分别使用∧、∨、¬和⊕表示布尔运算符与，或，非和异或。

定义 2.2（条件）：条件 C 是不包含布尔运算符的布尔表达式。一个条件不能再分解为几个更简单的布尔表达式。

定义 2.3（判定）：判定 D 是由一个或多个条件和零个或多个布尔运算符组成的布尔表达式。没有布尔运算符的判定是一个条件。

对于每个代码覆盖准则，本节将以图 2-1 中的示例程序为例，给出满足对应覆盖准则的一组测试用例。图 2-1 中的示例程序将四个整型输入值作为参数，如果第一个参数和第二个参数同时为正，或者第三个和第四个参数同时为正，则程序返回 true，否则程序返回 false。测试集中的每个单独测试用例均以输入参数 a，b，c，d 值的组合的形式给出。虽然该示例程序看起来非常简单，但是其判断逻辑足够复杂，足以体现本节中介绍的覆盖标准之间的差异。

由于输入参数是整数值，假设该示例程序应用于 32 位操作系统，则总共有 2^{128} 种不同的输入值组合，不可能将所有组合都一一进行测试，因此即使对于如此简单的程序，通过穷举测试（验证所有可能的执行）来验证所有正确的操作也是不可行的。

```
1    bool example(int a, int b, int c, int d) {
2        bool result = false;
3
4        if ((a>0 && b>0) || (c>0 && d>0)) {
5            result = true;
6        }
7        return result;
8    }
```

图 2-1 示例程序

Figure 2-1 The example program

2.1.2.1 语句覆盖准则

在进行测试时，如果程序中的每个可执行语句都被执行过至少一次，则该测试满足语句覆盖准则。对于图 2-1 中的示例程序，满足语句覆盖的测试用例如表 2-1 所示。

表 2-1 对于示例程序满足语句覆盖的测试用例

Table 2-1 A test case that fulfills statement coverage for the example program

a	b	c	d	return
0	0	0	0	false

语句覆盖又被称作行覆盖或者基本块覆盖，是常用覆盖准则中最简单也是检错能力最弱的覆盖准则。正如表 2-1 中所展现的，满足语句覆盖的测试用例甚至不需要包括让程序返回值为 true 的测试用例。

尽管语句覆盖的纠错能力很弱，但是它可以用于在测试过程中发现程序是否存在死代码。死代码是指在程序源代码中永远不能被执行到的部分代码，出现死代码说明程序在设计或实现时存在问题。如果一个测试不能满足语句覆盖准则，则意味着测试不够彻底，或者程序中存在死代码，无论是哪一种情况，都有助于发现测试或程序中存在的问题。

2.1.2.2　分支覆盖准则

在进行测试时，如果程序中的每个分支都被执行过至少一次，则该测试满足分支覆盖准则。对于图 2-1 中的示例程序，满足分支覆盖的测试用例如表 2-2 所示。

表 2-2 对于示例程序满足分支覆盖的一组测试用例

Table 2-2 A test-case set that fulfills branch coverage for the example program

a	b	c	d	return
0	0	0	0	false
1	1	0	0	true

分支覆盖又被称作判定覆盖，分支或判定是指控制代码结构的代码，例如 if、while，switch-case 等。对于示例程序，要想满足分支覆盖至少需要两个测试用例，其中一个使得 if 语句判定表达式的值为真，另一个使 if 语句判定表达式的值为假。

分支覆盖只关注分支的真假，而不关注决定分支真假的条件。如果该表达式是由几个条件组合起来的，则测试可能不会覆盖到每个条件的真假，例如在表 2.2 中，c 和 d 的取值始终为 0，也就是(c>0)&&(d>0)的结果始终为假。

2.1.2.3　条件覆盖准则

在进行测试时，如果程序中每个条件的取值都为 ture 和 false 至少各一次，则该测试满足条件覆盖。对于图 2-1 中的示例程序，满足条件覆盖的测试用例如

表 2-3 所示。

表 2-3 对于示例程序满足条件覆盖的一组测试用例

Table 2-3 A test-case set that fulfills condition coverage of the example program

a	b	c	d	return
0	1	0	1	false
1	0	1	0	false

条件覆盖只关注条件的取值，而不关注整个判定的结果，这会导致满足条件覆盖的测试可能并不能覆盖到全部的分支。例如表 2-3 中的测试用例满足条件覆盖，但是对于 if 语句的判定结果始终为假，导致程序的返回值始终为 false，并没有返回值为 true 的测试用例。

2.1.2.4 路径覆盖准则

在进行测试时，如果程序中每条可能的执行路径都被执行至少一次，则该测试满足路径覆盖准则。对于图 2-1 中的示例程序，满足路径覆盖的测试用例如表 2-4 所示。

表 2-4 对于示例程序满足路径覆盖的一组测试用例

Table 2-4 A test-case set that fulfills path coverage of the example program

a	b	c	d	return
0	0	0	0	false
1	1	0	0	true

路径覆盖的要求比语句覆盖、分支覆盖和条件覆盖要高，纠错能力也更强一些。对于图 2-1 中的示例程序来说，因为程序中只有一个 if 分支 ，所以路径覆盖和分支覆盖是等价的。但是假如程序中包含更多的分支，路径覆盖和分支覆盖之间的差异就会变得非常明显。例如图 2-2 所示的程序，其程序功能与图 2-1 中的示例程序相同，唯一的区别是将示例程序中的 if 语句拆成了两个。

```
1       bool example(int a, int b, int c, int d) {
2       bool result = false;
3
4           if (a>0 && b>0) {
5               result = true;
6           }
7           if (c>0 && d>0) {
8               result = true;
9           }
1           return result;
0
1       }
1
```

图 2-2 变体程序

Figure 2-2 The variant program

对于变体程序，仍然只需要两个测试用例就可以对其满足分支覆盖，如表 2-5 所示，其中第一个测试用例触发了第一个 if 语句的假分支和第二个 if 语句的真分支，第二个测试用例触发了第一个 if 语句的真分支和第二个 if 语句的假分支。但是要想满足路径覆盖，则至少需要 4 个测试用例，即两个 if 语句分别为假假、假真、真假、真真的测试用例，如表 2-6 所示。

表 2-5 对于变体程序满足分支覆盖的一组测试用例

Table 2-5 A test-case set that fulfills branch coverage of the variant program

a	b	c	d	if-1	if-2	return
0	0	1	1	false	true	true
1	1	0	0	true	false	true

表 2-6 对于变体程序满足路径覆盖的一组测试用例

Table 2-6 A test-case set that fulfills path coverage of the variant program

a	b	c	d	if-1	if-2	return
0	0	0	0	false	false	false
0	0	1	1	false	true	true
1	1	0	0	true	false	true
1	1	1	1	true	true	true

2.1.2.5 条件/判定覆盖准则

条件/判定覆盖准则是一个比较复杂的覆盖准则，也是 MC/DC 覆盖准则的基础。在进行测试时，如果程序中的每个条件和每个判定都取值真假至少各一次，则该测试满足条件/判定覆盖准则。对于图 2.1 中的示例程序，满足条件/判定覆盖的测试用例如表 2-7 所示。

表 2-7 对于示例程序满足条件/判定覆盖的一组测试用例

Table 2-7 A test-case set that fulfills condition/decision coverage of the example program

a	b	c	d	a>0	b>0	c>0	d>0	return
0	0	0	0	false	false	false	false	false
1	1	1	1	true	true	true	true	true

条件/判定覆盖准则结合了分支覆盖准则和条件覆盖准则，检错能力比两者更强。尽管条件/判定覆盖会让所有的条件和判定都取值真假至少一次，但它仍然无法检查每个条件对整个逻辑判断的影响。假设示例程序中 if 语句的逻辑表达式里，c>0 被错误的写成了 c<0，即 if((a>0&&b>0)||(c<0&&d>0))，如果仅通过条件/判定覆盖准则对该程序进行测试，那么使用表 2-7 中的测试用例仍然能够提供 100%的覆盖率，并且不能检查出错误（程序的返回值与预期相同）。

2.1.2.6 多条件覆盖准则

多条件覆盖准则要求在对程序进行测试时覆盖条件取值的所有可能组合，对于图 2-1 中的示例程序，满足多条件覆盖的测试用例如表 2-8 所示。由于会覆盖条件取值的所有可能组合，多条件覆盖的检错能力是上述几种常用覆盖准则中最强的，然而正如表 2-8 中展现的，满足多条件覆盖所需的测试用例数量极大，并且随着程序中条件数量的增多成指数型增长，如果程序中包含 n 个条件，则需要 2^n 个不同的测试用例才能满足多条件覆盖准则，这使得将该准则应用于现实生活中的程序测试是不切实际的。

表 2-8 对于示例程序满足多条件覆盖的一组测试用例

Table 2-8 A test-case set that fulfills multiple condition coverage of the example program

a	b	c	d	return
0	0	0	0	false
0	0	0	1	false
0	0	1	0	false
0	0	1	1	true
0	1	0	0	false
0	1	0	1	false
0	1	1	0	false
0	1	1	1	true
1	0	0	0	false
1	0	0	1	false
1	0	1	0	false
1	0	1	1	true
1	1	0	0	true
1	1	0	1	true
1	1	1	0	true
1	1	1	1	true

2.1.3 MC/DC 覆盖准则的定义

MC/DC 准则由两个相关准则组成，即条件覆盖（CC）准则和判定覆盖（DC）准则。但是条件覆盖只考虑条件的取值而不考虑判定结果，判定覆盖只考虑判定结果而不考虑条件的取值，这两种准则不足以充分测试嵌入了复杂逻辑判断的软件，因此有人提出了 MC/DC 准则，其定义为：程序中的每个入口点和出口点至少被调用一次；每个判定的所有可能结果至少出现一次；判定中每个条件的所有可能取值至少出现一次；每个条件都能独立地影响判定的结果[33]。

由于前三个要求直接由判定覆盖和条件覆盖组成，并已经在 2.1.2 节中介绍过，因此这里主要讨论 MC/DC 覆盖准则的第四个要求，即每个条件都能独立地影响判定的结果。如果在其它所有条件值不变的情况下，仅通过改变某一条件的值，使得判定的结果发生改变，则称该条件独立地影响了判定的结果。能体现某条件独立影响判定结果的两个测试用例的组合被称为该条件的 MC/DC 对[34]。要达到 MC/DC 覆盖准则，一组测试用例需要包含所有条件的 MC/DC 对，这意味

着达到 MC/DC 覆盖准则所需要的测试用例随着条件的增加呈线性增长，远远小于多条件覆盖所需要的呈指数增长的测试用例数。对于拥有 n 个条件的程序来说，MC/DC 覆盖准则所需要的测试用例数在 n+1 到 2n 之间[28]。

对于图 2-1 中的示例程序，满足 MC/DC 覆盖的测试用例如表 2-9 所示。由于示例程序只有一个入口点和出口点，并且每个条件取值真假至少各一次，每个判定也取值真假至少各一次，所以该组测试用例满足 MC/DC 覆盖准则的前三项要求。为了满足第四项要求，测试用例必须包含四个条件的 MC/DC 对。我们用 $P_{(a>0)}$表示条件 a>0 的 MC/DC 对，$P_{(b>0)}$表示 b>0 的 MC/DC 对，以此类推，

表 2-10 中显示了测试用例中对于每个条件的 MC/DC 对。对于测试用例 1 和测试用例 2 而言，由于其他三个条件的真假均没有改变，只有条件 a>0 的真假发生了改变，并且整个判定的结果也发生了改变，因此这两个测试用例是条件 a>0 的 MC/DC 对。基于相同的原因，测试用例 2 和 3 是条件 b>0 的 MC/DC 对，测试用例 4 和 5 是条件 c>0 的 MC/DC 对，测试用例 3 和 4 是条件 d>0 的 MC/DC 对。因此该组测试用例满足 MC/DC 覆盖准则的第四项要求。

表 2-9 对于示例程序满足 MC/DC 覆盖的一组测试用例

Table 2-9 A test-case set that fulfills modified condition/decision coverage of the example program

a	b	c	d	a>0	b>0	c>0	d>0	decision
0	1	1	0	false	true	true	false	false
1	1	1	0	true	true	true	false	true
1	0	1	0	true	false	true	false	false
1	0	1	1	true	false	true	true	true
1	0	0	1	true	false	false	true	false

表 2-10 该组测试用例中的 MC/DC 对

Table 2-10 MC/DC pairs in the test-case set

#	a>0	b>0	c>0	d>0	decision	$P_{(a>0)}$	$P_{(b>0)}$	$P_{(c>0)}$	$P_{(d>0)}$
1	false	true	true	false	false	2	-	-	-
2	true	true	true	false	true	1	3	-	-
3	true	false	true	false	false	-	2	-	4
4	true	false	true	true	true	-	-	5	3
5	true	false	false	true	false	-	-	4	-

2.1.4 MC/DC 覆盖准则的应用

上文提到的代码覆盖准则通常用于软件测试，但是本文主要关注 MC/DC 准

则，因为该准则经常用于航空航天等生命攸关领域的软件认证。在航空航天领域进行软件认证时，使用何种软件通常由软件认证机构与飞机开发商共同确定，这意味着软件系统始终被认为是整个飞机的一部分。飞机开发商必须提供确保软件开发过程能够满足软件认证的方法。航空软件认证的过程主要基于标准，并且该方法在生产安全关键的软件系统方面非常有效[35]。

飞机设计过程中的一个重要步骤就是安全分析。由于飞机上的每个系统都要执行特定的功能，这意味着只要发生系统故障，就会使飞机发生故障，导致意外功能或功能丧失。安全分析会根据故障的严重程度进行分级，从“灾难性”（即飞机失事）、“危险”、“主要”、“次要”到“无效”，取决于它们对飞机，机组人员和乘客安全的影响。安全分析的主要目标是使人们在进行系统设计的过程中，最大限度地减少可能发生的故障的数量和严重程度。机载系统和设备认证中的软件考虑因素（DO-178B，Software Considerations in Airborne Systems and Equipment Certification）[13]中定义了确保实施正确性的指导原则，该指导可以被视为航空软件开发和认证的标准[35]。

DO-178B 概述了一些软件开发和分析过程，航空软件开发人员可以使用这些过程来获得软件系统的批准。DO-178B 确定了从 A（最高）到 E（最低）一共五个不同的设计保证级别(或软件级别)，以及软件开发和分析过程中对应每个级别应该满足的一组目标。软件系统的级别由与系统相关的故障严重程度决定，例如级别为 A 的软件对应于“灾难性”故障，级别为 B 的软件对应“危险”故障。为了防止最关键的 A 级软件发生故障，DO-178B 规定所有这些软件在进行测试时都必须满足 MC/DC 准则。同时，我国的载人航天工程要求级别为 A 级和 B 级的软件在进行单元测试时，MC/DC 覆盖率应达到 100%，并对因测试条件限制而覆盖不到的语句进行逐一分析和确认[36]。

2.2 符号执行技术

本节将介绍动态符号执行技术和符号执行工具 KLEE，它们是本文所提出的方法的基础。本节首先介绍自动化软件测试，并提供有关该领域相关工作的概述，然后将通过一个例子介绍动态符号执行是如何工作的，最后介绍符号执行工具 KLEE 和一些使用细节。

2.2.1 自动化软件测试

为了能够正确的理解自动化软件测试，我们首先应该知道“自动化软件测试”和“测试自动化”之间的区别。虽然人们常常将两者混用，但通常来讲“测试自

动化”指的是测试过程的自动化，即在已经有一组测试用例的前提下，自动去运行这些测试用例，而“自动化软件测试”是指从测试用例生成到运行测试用例的整个过程自动化完成[37]。由于测试自动化在实际应用中已经非常普遍了，因此本节主要讨论测试用例生成部分。

符号执行等软件测试技术在对软件进行测试之前，需要先对程序进行分析[38]。程序分析主要分为静态分析和动态分析。程序静态分析基于程序的结构，其主要对程序的源代码（有时是目标代码）进行分析，而程序动态分析基于程序的实际运行。这两种手段并不是排他性的，本节介绍的大部分工具所使用的方法都是两者的结合。

自动化软件测试过程大致可以分为三个部分。第一步是提取程序的接口并对需要的测试输入进行标记，这一步决定了要生成的测试用例的格式。第二步是对需要的测试输入选择输入值，也就是生成测试用例。这些输入值应该尽可能多的执行程序的可执行路径，这是自动化软件测试中最困难的部分，也是不同方法之间差异最大的部分。第三步是使用所选的输入值实际执行程序，也就是对程序进行测试。这一步通常还会在程序运行时进行额外的分析，以便于对后续选择输入值进行指导。

最简单的自动化软件测试方法就是随机产生输入值，被称为随机测试或者模糊测试[39]。虽然方法的原理非常简单，随机测试仍然能够很有效的发现实际错误[40]，并且它适用于所有类型的程序，同时也适用于黑盒测试。随机测试最大的问题在于它很难触发一些常见的分支，因此并不能达到很高的代码覆盖率[41]。假设有一个 if 语句对整数变量和一个特定整数进行比较（例如 a==1），由于整数的可能取值非常多，所以产生 a 的输入值时，随机到特定整数的概率几乎是不可能的。一个名为 DART[5]的工具通过将程序的具体执行和符号执行相结合来解决上述问题。DART 使用的方法被称为动态符号执行，该方法将通过下一节的示例进行更详细的说明。简单来说，该方法会先根据具体执行时遇到的条件对输入值生成相应的符号约束，将它们组合成路径约束。然后使用符号执行的约束求解器对路径约束进行求解，从而得到执行特定路径的输入值。这一思想与传统的符号执行[4]非常相似，但区别在于，使用该方法可以确保每次生成的输入值总是会遵循具体执行的路径[42]。

传统的符号执行[4]，也就是动态符号执行的基础，是自动生成测试用例的另一种常见方法。如前文所述，动态符号执行和非动态符号执行之间的主要区别在于动态符号执行会实际执行程序来探索程序的不同路径。虽然这两种方法的差异看起来相对较小，但实际上，这个差异对两种方法的实现方式有着很大的影响。具体来讲，对于传统的符号执行，程序必须从一开始就以符号的方式执行，这需

要对符号进行更细致的管理，因为如果出现问题，执行过程中没有具体的值可以回退。在符号执行刚开始时，每个输入的符号值都是没有约束的，但随着程序的执行，输入的符号值会根据遇到的操作添加对应的约束。如果程序存在分支，那么符号执行也必须一分为二并分别考虑每种情况。有一些工具基于该方法设计，其中使用比较广泛的是 KLEE[8]和 Java PathFinder[43]。KLEE 对程序的 LLVM[44]的 bitcode 进行处理，而 Java PathFinder 对 Java[45]字节码进行处理。这些工具使用了巧妙的约束和环境管理方法，以大幅提高 KLEE 的性能，并使用模型检查为 Java PathFinder 中的高级编程语言功能提供支持。

2.2.2 动态符号执行技术

动态符号执行是一种自动化软件测试方法，其目的是通过自动探索尽可能多的执行路径来发现程序中的错误。具体来说，该技术是通过将具体执行程序和象征性的执行程序相结合来完成的。在进行测试时，该技术首先会随机产生输入值用于程序执行，在程序执行的过程中收集关于程序结构的相关信息，这些信息（以符号约束的形式体现）将会用于生成新的输入值，并且这些新的输入值会将程序引导到之前未执行过的特定路径[46]。

本节使用一个例子程序来具体说明动态符号执行技术如何对程序的可执行路径进行探索。图 2-3 展示了该例子程序的源代码和程序执行流程示意图。该例子程序包含一个整形变量输入和两个可能会被触发的错误，并在程序的最后返回一个整形变量。为了简洁易懂，我们使用 input()表示变量 x 的值是从外界输入的，并且直接使用注释来表示错误（第 6 行和第 10 行），从而省略掉会触发错误的实际代码（例如断言失败等）。对于该程序来说，第 6 行的错误实际上是不可达的，而第 10 行的错误只有在输入值 x 为某个特定值的时候才会被触发，这是为了说明动态符号执行如何处理这些情况，并且展示与随机测试相比，动态符号执行如何轻易的触发那些需要特定输入值才能触发的错误。

```
1   bool example() {
2       int x = input();
3       if (x > 0) {
4           int y = 2 * x + 1;
5           if (y < 0) {
6               // error
7           }
8
9           if (y == 7) {
10              // error
11          }
12          return y;
13      }
14      return x;
15  }
```

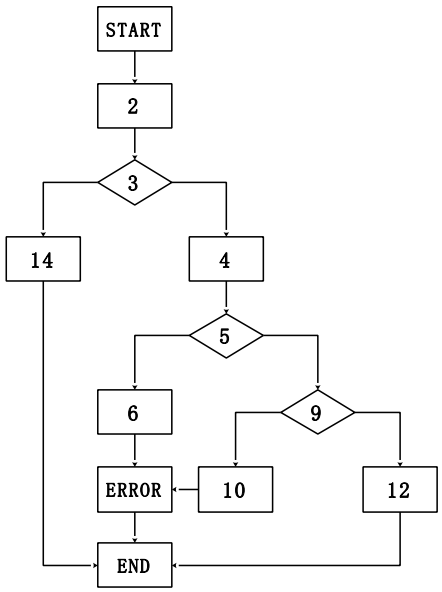
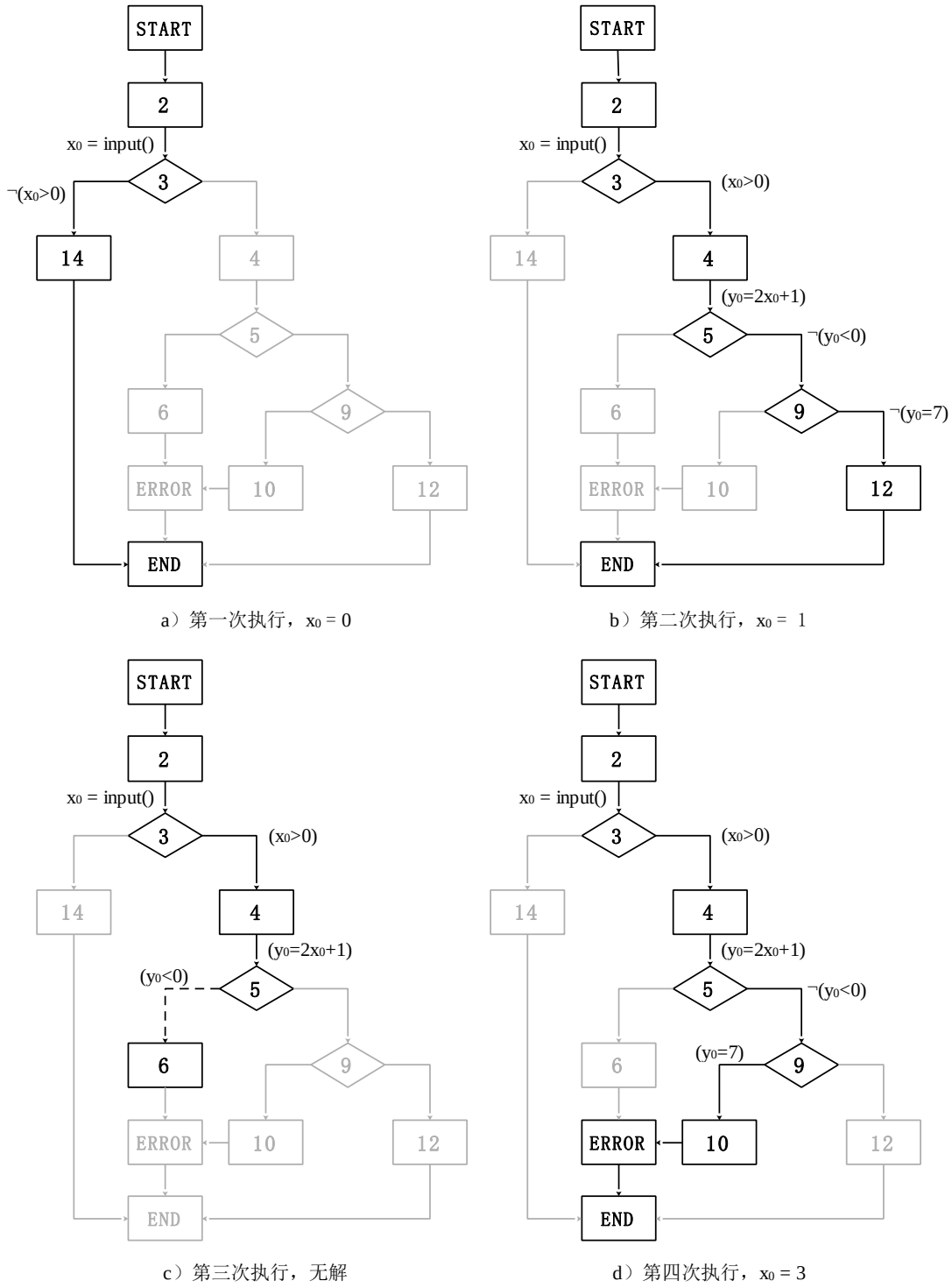


图 2-3 动态符号执行示例程序及其执行流程示意图

Figure 2-3 The example program for dynamic symbol execution and its execution process graph

使用动态符号技术的测试过程如图 2-4 所示。由于在第一次执行前对程序一无所知，因此初始的输入值是随机产生的，并且初始的路径约束为空。假设初始输入值为 0，这意味着第一次的执行路径是 2-3-14，并且返回值是 0，如图 2-4 a 所示，在执行期间获得的路径约束显示在图中对应边的旁边。在第一次执行结束后，动态符号执行技术会发现有一条分支未覆盖，即 x>0 为真的分支。为了覆盖该分支，动态符号执行技术会将$\{x_0>0\}$作为路径约束进行求解，并得到一个输入值（如果有解的话）。在这里假设求解结果为 1，则第二次测试将使用 1 作为输入值，如图 2-4 b 所示，执行路径为 2-3-4-5-9-12，返回值为 3。在第二次执行期间，动态符号执行技术又会发现两条分支未被覆盖，并会尝试依次覆盖它们。首先是覆盖 y<0 为真的分支，动态符号执行技术会将$\{(x_0>0)\wedge(y_0=2*x_0+1)\wedge(y_0<0)\}$作为路径约束进行求解，即执行路径为 2-3-4-5-6，并发现该约束无解，这意味着不管提供怎样的输入值都不可能触发该分支，这样的分支将被忽略。然后是覆盖 y==7 的真分支，动态符号执行技术会将$\{(x_0>0)\wedge(y_0=2*x_0+1)\wedge(y_0=7)\}$作为路径约束进行求解，并求得输入值 $x_0=3$，使用该输入值进行测试，如图 2-4 d 所示，执行路径为 2-3-4-5-9-10，该次执行会触发程序中第 10 行的错误，并终止该次执行。



a）第一次执行，$x_0 = 0$　　b）第二次执行，$x_0 = 1$

c）第三次执行，无解　　d）第四次执行，$x_0 = 3$

图 2-4 动态符号执行技术的执行过程

Figure 2-4 Execution process of the dynamic symbol execution

动态符号执行技术最大的问题是如果符号执行由于某种原因无法跟踪具体执行的情况，则测试过程就不能保证完全覆盖程序中的所有执行路径。无法跟踪的原因包括程序执行了符号执行不支持的操作，或者运行到了符号执行尚未检测的库函数。动态符号执行技术通常可以识别这些情况，并且使用户意识到哪些地

方没有被完整的覆盖。另外，如果程序中存在未被视作输入的非确定性变量（例如生成随机数等），则可能会存在在测试期间不可达的路径在实际应用中可达的情况。但是如果整个程序都在符号执行的范围内并且所有的输入值都已经被正确的定义，那么动态符号执行技术能找到执行程序的每条可执行路径的不同输入值，从而为程序提供详尽的测试。

动态符号执行技术的另一个严重问题是路径爆炸问题[47]。程序中不同的执行路径可能会非常庞大，这意味着即使不受上面那些问题的限制，对于一个非常复杂庞大的程序来说，动态符号执行技术要想对其进行完全覆盖仍然是不可行的。使用启发式算法可以在一定程度上解决这个问题，这些方法可以在众多未覆盖的可执行路径中优先选择增加代码覆盖率最多的执行路径，但这些方法仍然无法从根本上解决路径爆炸的问题。

2.2.3 符号执行工具 KLEE

KLEE 是一款由斯坦福大学的 Cristian Cadar 等人研究设计的，使用动态符号执行技术的自动化软件测试工具[8, 29]。KLEE 构建在 LLVM 编译器的基础结构之上，其使用 LLVM 的中间表示，即 bitcode 作为输入，并在其上应用动态符号执行技术来对程序进行测试，同时使用用户指定的约束求解器来进行约束求解。

研究人员在对 KLEE 进行实验评估时发现，如果能运行足够长的时间，KLEE 可以对于 Linux 实用工具[48]生成语句覆盖率达到 90%以上的测试用例，并且研究人员在实验中使用 KLEE 发现了三个在 Linux 系统中存在了十年以上的漏洞[8]。但是，由于真实世界中的程序通常比较复杂，对其进行完全的符号执行可能需要非常长的时间，因此在使用 KLEE 时通常会增加各种限制条件，比如限定时间、限制最大状态数或者限制最大内存使用量等[49]。

2.3 可测试性转换

本节将会介绍可测试转换理论，提供一些基本概念的定义，并详细说明可测试性转换与传统转换之间的区别。

2.3.1 可测试性转换的定义

可测试性转换旨在转换程序，以便更容易为其生成测试数据（以改进原始程序的“可测试性”）[24]。但这种可测试性转换的概念存在一个矛盾，即结构测试是基于结构定义的测试充分性准则，可测试性转换通过转换程序从而更容易地生

成测试数据，但转换同时也会改变程序的结构，因此导致生成的测试数据对于转换前的程序可能不满足测试充分性准则。

这个矛盾的解决方案是允许可测试性转换在转换代码的同时改变测试充分性准则，使得测试数据对于转换后的程序满足特定测试充分性准则时，可以确保测试数据对于转换前的程序满足原先的测试充分性准则[50]。测试充分性准则是在测试期间要覆盖的任何语法结构集。测试充分性准则确定一组测试数据是否满足测试者想要达到的标准，例如对于语句覆盖准则，当针对程序 P 应用测试数据，使 P 中的每条语句都被至少执行过一次时，该准则为真。我们给出下列定义用以更精准的描述以上内容。

定义 2.4（面向测试的转换）：设 P 是一组程序，Cr 是一组测试充分性准则，程序转换是 $P \to P$ 中的部分函数。面向测试的转换是 $(P \times Cr) \to (P \times Cr)$ 中的部分函数[51]。

其中，测试充分性准则 Cr 指的是整体标准，其可以由一系列特定的测试充分性准则 cr 组成。例如测试充分性准则 Cr 可以由分支覆盖准则 cr_1 和语句覆盖准则 cr_2 组合而成。

定义 2.5（可测试性转换）：面向测试的转换 τ 是可测试性转换，当且仅当对于所有的程序 p，充分性准则 cr 和测试集 T，如果 T 对于 p'满足充分性准则 cr'，则 T 对于 p 满足充分性准则 cr，即 $\tau(p,cr) = (p',cr')$[51]。

可测试性转换是面向测试的转换，根据转换后的程序生成的测试数据可以为原始程序提供足够的测试数据。在某些情况下，测试充分性标准可以与被测试程序一起被转换，并且定义允许这样做。例如，MC/DC 测试充分性准则[52]可以转换为分支充分性准则，前提是程序中的谓词被展开成布尔逻辑运算符项。这种转换意味着转换后程序的分支覆盖与转换前程序的 MC/DC 覆盖相同（具有相同的能够满足测试充分性准则的测试输集）。可测试性转换仅保证生成足够的测试数据以满足原始测试充分性标准。它不要求生成的测试数据不包含冗余的测试用例。如果可以在不影响原始测试充分性标准的满足的情况下移除某个测试用例，则该测试用例是冗余的。

2.3.2　可测试性转换与传统转换之间的区别

可测试性转换在三个重要的方面与传统转换[53-55]不同。

首先，传统的程序转换仅转换程序。对于可测试性转换来说，就像定义 2.4 中所描述的，程序的可测试性转换可以伴随着对测试充分性标准的转换，使得为转换后的程序（和转换后的标准）生成的测试数据对于原始程序和原始标准是足

够的[56]。

其次，传统的程序转换的目标是得到转换后的程序，原始程序将会被转换后的程序所取代。可测试性转换所产生的转换后程序本身只是一种“达到目的的手段”，而不是“最终目的”，转换后的程序用于生成测试数据后就可以被丢弃[57]。

最后，由于传统代码转换的目标是得到转换后的程序，所以在进行转换时必须保证转换后程序的语义和原始程序的语义是等价的。但是，可测试性转换的目标是从转换后的程序生成测试数据，使其针对原始程序进行测试时可以满足某些测试充分性准则，这意味着对应的转换规则不需要确保转换后程序的语义和原始程序的语义等价，使得定义新的程序转换规则和算法成为了可能[51]。

2.4 本章小结

本章主要介绍了一些与本文方法有关的背景知识，包括软件测试，代码覆盖准则，符号执行技术和可测试性转换理论等。第一节介绍了软件测试和作为软件测试标准的几个代码覆盖准则，定义并介绍修订的条件/判定覆盖，并讨论其在航空航天等领域中的应用；第二节介绍了本文所用到的动态符号执行技术和符号执行工具 KLEE，为自动化测试生成过程提供背景知识；第三节介绍了可测试性转换理论，并阐述了其与传统转换之间存在的区别。

第3章　MC/DC 测试用例生成方法

3.1 概述

现有的动态符号执行技术以分支覆盖为目标进行测试用例生成。MC/DC 准则对于判定中的每一个条件都需要生成特定组合值的测试用例，这并不总是使被测程序能够执行一个新的分支，导致现有的动态符号执行技术不能生成满足 MC/DC 准则的测试用例。

本文方法的主要思想是针对程序中的每个条件生成一组条件约束，通过代码转换将条件约束添加到待测程序的源代码中，并将转换后的程序用于测试用例生成。条件约束的本质是用于生成特定组合值的测试用例的一组 if 语句，即提供一些额外的分支以供符号执行技术探索。当针对程序中某个条件所生成的条件约束被动态符号执行技术探索并满足时，生成的测试用例即可对该条件满足 MC/DC 覆盖，也就是该条件的 MC/DC 对。由于动态符号执行技术以分支覆盖为目标进行测试用例生成，这意味着由本方法生成的条件约束至少会被覆盖到一次，也就是在测试用例生成时会对每个条件生成对应的 MC/DC 对，最终满足 MC/DC 准则。

本文方法主要分为三步。首先，本方法将待测程序源代码作为输入并将其解析为 AST。然后对程序的 AST 进行分析，识别每个分支语句的判定，并获取每个判定的条件集合。在获取到条件集合后，本方法针对每个条件生成一组条件约束，并将其添加到待测程序源代码中该条件所属的判定之前，将待测程序转换为一段新的程序。最后，本方法使用转换后的程序作为基于分支覆盖的动态符号执行工具的输入，生成测试用例。图 3-1 给出了本文方法的架构。

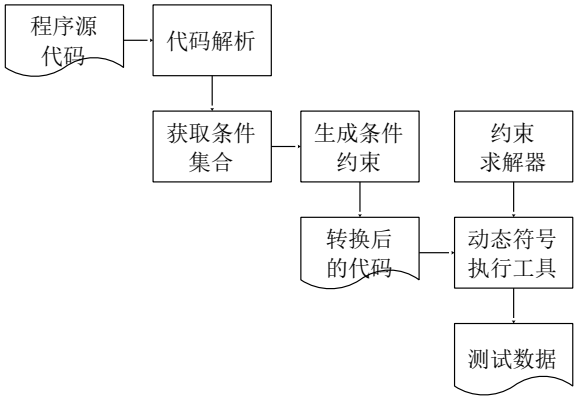


图 3-1 方法的架构图

Figure 3-1 The architecture of approach

本方法是针对转换后的代码来生成测试用例，而不是针对待测程序源代码来生成测试用例，这意味着用于生成测试用例的代码在程序结构上可能与待测程序源代码不同。另外，动态符号执行工具不是针对程序源代码，而是针对程序的目标代码进行测试生成。这中间可能存在问题，因为我们是对程序源代码的覆盖范围感兴趣，而在编译过程中，程序的结构会发生很大变化。对于以上两点，本方法生成的测试用例最终将用于针对待测程序源代码进行测试，这意味着即使用于测试生成的代码与待测程序源代码结构不同，并且动态符号执行工具在目标代码上工作，生成的测试用例都将提供待测程序源代码上的 MC/DC 覆盖。

我们不保证本方法一定可以生成能够达到 100%覆盖率的测试用例，有几个原因会导致这一点。最重要的是我们的方法所基于的动态符号执行技术在程序执行期间工作，这意味着如果程序中存在死代码，我们将不能针对死代码中的条件和判定生成测试用例[58]。此外，由于动态符号执行技术的约束求解和路径爆炸问题，完全生成测试用例的时间可能会过长。比较明智的做法是在较高覆盖率和适当的测试生成时间之间取得平衡。我们可以对测试过程中获得的 MC/DC 覆盖率进行计算，相关内容详见第 5 章。

3.2 获得条件集并生成条件约束

正如 2.1.3 节中定义的，给定某一个判定 D，如果对于 D 中的每个条件 C，都存在至少一组 MC/DC 对，使得 C 被 MC/DC 覆盖，则判定 D 被 MC/DC 覆盖。所以本方法的主要目标就是针对判定中的每个条件，生成一组约束，使符号执行工具在执行过程中可以根据约束生成对应条件的 MC/DC 对。为了达到这个目的，我们首先要确定一个判定中都有哪些条件。

由于条件的副作用可能会对测试生成技术产生影响[59, 60]，同时消除程序中的副作用也被认为有助于更好的理解程序[61]，所以我们假设程序不包含带有副作用的条件，也就是说条件不会改变任何变量的值。如果程序中包含具有副作用的条件，可以先将其转换成不包含副作用的程序[61, 62]。

根据 MC/DC 准则最初的定义[58]，如果某个条件在同一个判定中出现不止一次，则每次出现都算作是一个新的条件。但在真实世界的程序中，当计算一个判定的真假时，相同的条件只能计算出相同的结果。因此在本文中，我们把在一个判定中多次出现的相同条件视为同一个条件。例如，对于判定 D=(a∧b)∨(a∧c)，条件 a 出现了两次。如果根据 MC/DC 准则最初的定义，该判定应该有四个条件，但是在本文中，我们认为该判定只有三个条件，即 a、b 和 c。

对于要测试的原始代码，本方法首先获取代码中的每个判定的条件集合，然

后生成对应的约束。约束主要分为两部分，第一部分是确保判定中的其它条件不会影响该判定的结果，第二部分是保证对应的条件会触发真假各一次，用来影响整个判定的真假。在进行代码转换时，约束会以 if 语句的形式体现在转换后的版本上，当符号执行工具探索这些 if 语句时，会把对应的约束添加到条件路径中，最终生成对应条件的 MC/DC 对。

获得判定的条件集合的具体算法如表 3-1 所示。该算法首先确定给定判定是否为一个二元运算表达式，如果不是，那么该判定是一个条件；如果是一个二元运算表达式，再判断它的二元运算符是不是布尔运算符∧或者∨，如果不是则该判定是一个条件，如果是则把该二元运算表达式的左操作数和右操作数看成是两个子判定，分别获取它们的条件集合。

表 3-1 条件集合获取算法

Table 3-1 Algorithm for get the set of conditions

输入：表达式 *D*
输出：*D* 的条件集合 CS_D

```
1   function GetCS(D)
2       CS_D ← ∅
3       while D 是一元运算表达式
4           D ← D 的操作数
5       if D 是二元运算表达式 then
6           op ← D 的二元运算符
7           if op 为“&&”或“||” then
8               ld ← D 的左操作数
9               rd ← D 的右操作数
10              CS_D ←CS_D∪GetCS(ld)∪GetCS(rd)
11          else
12              CS_D ←CS_D∪{D}
13          end if
14      end if
15      return CS_D
16  end function
```

用于指导符号执行针对目标条件生成 MC/DC 对的约束主要分为两部分，第一部分是确保判定中的其它条件不会影响该判定的结果，第二部分是保证目标条件会触发真假各一次，用来影响整个判定的真假。

为了确保判定中的其它条件不会影响判定结果，我们需要分析整个判定的结构，使得通过布尔运算符与与目标条件相连的条件保持为真，通过布尔运算符或与目标条件相连的条件保持为假。例如，对于判定 D=a∧b∨c，我们想要生成条件 b 的 MC/DC 对时，需要保证 a 为真的同时 c 为假，即生成约束{(a=true)∧(c=false)}。对于一个复杂的，带有多层括号嵌套的逻辑表达式，分析整个判定的

结构是一件非常复杂的事情，所以我们使用现有的符号执行优化技术来完成这部分工作。符号执行的表达式重写[63]优化可以将原始的约束转换成更容易求解的形式，这使得我们可以通过生成更加“直观”的约束来保证判定中的其它条件不会影响判定结果。同样以判定 D=a∧b∨c 为例，我们只需要生成约束{(a∧true∨c)≠(a∧false∨c)}，符号执行的表达式重写优化将自动把该约束化简为{(a=true)∧(c=false)}。

在上述约束的基础上，我们需要能够使目标条件的真假各触发一次的约束，对于上面的例子来说就是{b=true}和{b=false}。将两部分约束合并，就能得到最终的约束，即{((a=true)∧(c=false))∧(b=true)}和{((a=true)∧(c=false))∧(b=false)}。当符号执行对该约束进行约束求解时，就可以得到条件 b 的 MC/DC 对。该约束实际上并不能保证其它条件不变，只能保证其他条件的真假性不变。然而符号执行技术为了加快求解速度而使用了约束缓存技术[63]，该技术的主要思想是新的约束通常不会使现有子集的解失效，可在子集的解的基础上验证可解性，由于我们针对条件 b 生成的约束并不会影响到条件 a 和 c，所以符号执行会复用约束{(a=true)∧(c=false)}的解，这也意味着我们可以满足其它条件不变。

本方法并不会直接生成约束，而是通过代码转换在程序源代码中插入一组 if 语句，用来引导符号执行工具自行生成对应的约束，如表 3-2 所示。由于某些符号执行的优化技术会自动忽略掉不涉及任何变量改变的分支，所以本方法会在不需要包含任何语句的 if 分支中添加一个变量自增的操作，该操作没有实际意义，只是为了确保符号执行会探索该路径。

表 3-2 条件约束生成算法

Table 3-2 Algorithm for generate conditional constraint

输入：判定表达式 *D* 和 *D* 的条件集合 CS_D
输出：*D* 的条件约束

```
1   instrumentations ← [ ]
2   D ← AddParens(D)
3   for each C∈CS_D do
4       ins ← " "
5       D_CT ← 将 D 中的 C 替换为 true
6       D_CF ← 将 D 中的 C 替换为 false
7       ins ← ins + "if (D_CT ≠ D_CF) { \n"
8       ins ← ins + "    int i = 0; \n"
9       ins ← ins + "    if (C) {i++;} else {i++;} \n"
10      ins ← ins + "} \n"
11      instrumentations.add(ins)
12  end for
13  return instrumentations
```

3.3 测试用例生成

本节通过一个具体的示例介绍基于代码转换的 MC/DC 测试用例生成的具体过程。示例程序如图 3-2 所示，该程序包含两个输入参数（省略了初始化和输入部分）。程序中没有包含可能会触发的错误，因为我们的主要目标不是找到程序中的错误，而是生成符合 MC/DC 准则的测试用例。整个测试生成过程主要分为两部分，分别是代码转换（用来添加条件约束）和符号执行（用来生成测试用例）。

```
1  int add_positive(int x, int y) {
2      if ((x>0) && (y>0)) {
3          return x + y;
4      }
5      return -1;
6  }
```

图 3-2 测试生成示例程序

Figure 3-2 Test generation example program

生成条件约束的目的是让动态符号执行技术可以根据条件约束生成满足 MC/DC 准则的测试用例。MC/DC 准则要求程序中所有的条件必须采取所有可能的结果至少一次，所有判定必须采取所有可能的结果至少一次，并且每个条件要独立的影响判定结果至少一次。我们只需要针对第三点要求生成条件约束，因为满足第三点要求的约束也能保证在测试过程中程序中的每个条件和判定都触发真假各一次。

```
1  int add_positive(int x, int y) {
2      if (((1) && (y>0)) != ((0) && (y>0))) {
3          int i = 0;
4          if (x>0) { i++; } else { i++; }
5      }
6      if (((x>0) && (1)) != ((x>0) && (0))) {
7          int i = 0;
8          if (y>0) { i++; } else { i++; }
9      }
10     if ((x>0) && (y>0)) {
11         return x + y;
12     }
13     return -1;
14 }
```

图 3-3 转换后的测试生成示例程序

Figure 3-3 The transformed test generation example program

本示例使用在 3.2 节中提到的方法递归地分析判定中的条件，针对每个条件

生成可以得到 MC/DC 对的条件约束，并将其添加到代码中。转换后的代码如图 3-3 所示，其中第 2 到 5 行是针对条件 x>0 生成的条件约束，6 到 9 行是针对条件 y>0 生成的条件约束。10 到 13 行是程序原本的代码，不需要进行改动。

为了更好的展示条件约束如何指导符号执行技术，我们首先看看未经转换的代码如何进行测试生成。由于符号执行技术基于 LLVM 的中间表示文件而非源代码文件，所以在进行测试生成之前，程序会经过预处理，并对程序中的分支结构进行改写。本示例通过改写程序源代码来达到类似的效果，经过 LLVM 预处理后的代码如图 3-4 所示。

```
1  int add_positive(int x, int y) {
2      if (x>0) {
3          if (y>0) {
4              return x + y;
5          }
6      }
7      return -1;
8  }
```

图 3-4 预处理后的测试生成示例程序

Figure 3-4 The preprocessed test generation example program

图 3-5 展示了对于未经转换的代码进行测试生成时的动态符号执行过程，其中 σ 为符号表，PC 为条件约束。由于具体的执行细节已经在 2.2 节中有过描述，因此这里略过细节，直接展示结果。对程序进行探索后，动态符号执行技术会产生三条路径约束，分别是{¬(x_0>0)}、{(x_0>0)∧¬(y_0>0)}和{(x_0>0)∧(y_0>0)}，并且会随机生成满足这三条路径约束的输入值，也就是测试用例。这里假设根据路径约束生成的测试用例为(0,0)、(1,0)和(1,1)。根据定义，这组测试用例不能满足 MC/DC 准则，因为变量 x 未能单独影响整个判定的结果。

对于转换后的代码，动态符号执行过程如图 3-6 所示。在动态符号执行技术对图 3-3 中条件 x>0 的条件约束（2 到 5 行）进行探索时，会生成路径约束{(x0>0)∧(y0>0)}和{¬(x0>0)∧(y0>0)}；在对条件 y>0 的条件约束（6 到 9 行）进行探索时，会重复使用路径约束{(x0>0)∧(y0>0)}，并生成新的条件约束{(x0>0)∧¬(y0>0)}。当动态符号执行技术执行到程序原本的代码（10 到 13 行）时，会发现已有的条件约束已经可以满足剩下的所有路径，所以会重复使用之前的路径约束，不会生成新的路径约束。假设根据路径约束生成的测试用例为(1,1)、(0,1)和(1,0)，因为其中的变量 x 和变量 y 均单独影响了整个判定的结果，所以该组测试用例可以满足 MC/DC 准则。

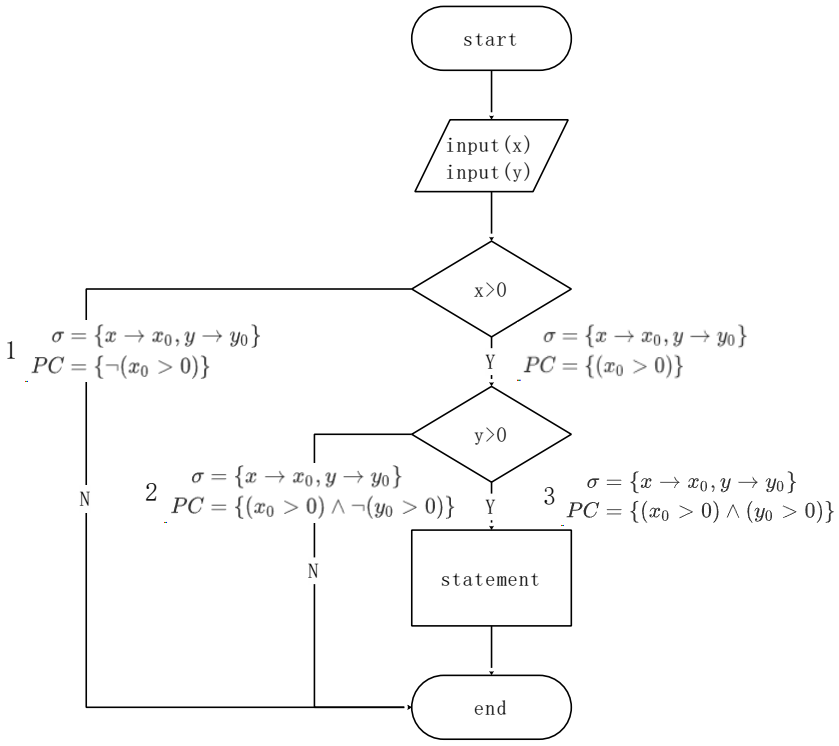


图 3-5 测试生成示例程序的执行过程

Figure 3-5 Execution process of the test generation example program

满足 MC/DC 准则的测试用例，其数量在 n+1 到 2n 之间，其中 n 是判定中条件的个数[34]。通过上述示例可以发现，通过一个条件的条件约束最多可以生成两个不同的路径约束，即生成两个不同的测试用例。而本方法会针对判定中的每个条件，生成条件约束，所以在最坏情况下，测试用例数为条件数的两倍。一般情况下，由于条件路径可以重复使用，生成的测试用例数会比条件数的两倍要低，即处于 n+1 到 2n 之间。

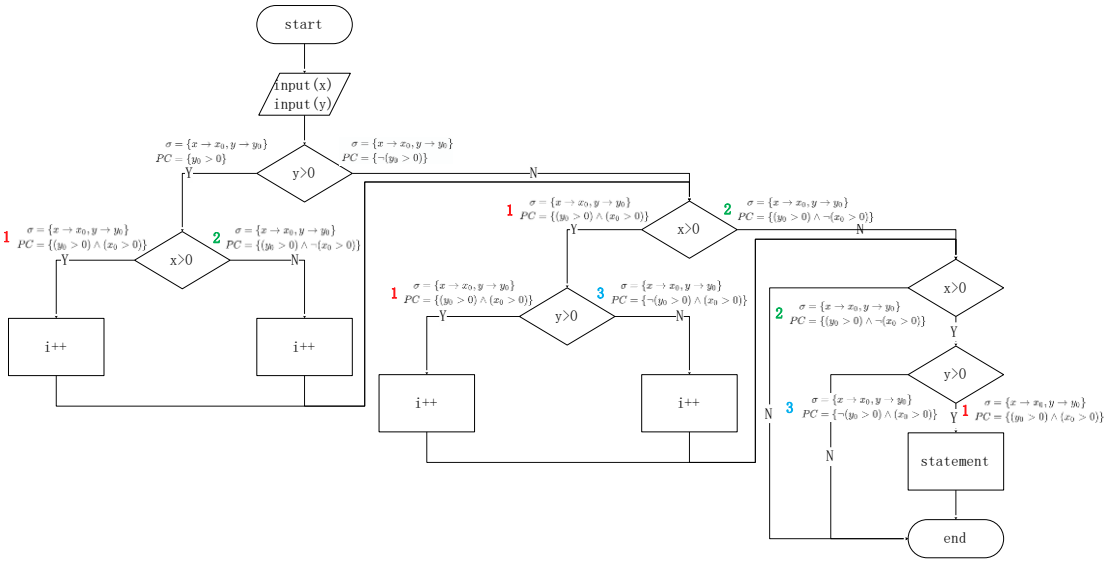


图 3-6 转换后的测试生成示例程序的执行过程

Figure 3-6 Execution process of the transformed test generation example program

3.4 本章小结

本章主要介绍了基于代码转换的 MC/DC 测试用例生成方法。第一节介绍了怎样的测试用例能够满足 MC/DC 准则，并讨论了再生成这些测试用例时的一些问题；第二节介绍了通过代码转换生成条件约束的具体方法；第三节使用一个示例程序完整展示从代码转换到生成 MC/DC 测试用例的整个过程。

第4章　基于 Clang 的用于 MC/DC 测试用例生成的代码转换工具

4.1 概述

4.1.1 Clang 编译器

LLVM 项目始于伊利诺伊大学 2000 年的一个开源项目，是一系列模块化和可重复使用的编译器和工具链的集合。LLVM 现在已经发展成一个大型项目，包括从编译器到静态分析等各种开源项目。Clang 是 LLVM 的编译前端，其主要支持 C、C++、Objective-C、Objective-C++和 Swift 等语言。Clang 良好的模块化设计，使其成为相关研究领域的理想选择。Clang 拥有定义良好的接口，并且提供了多种访问抽象语法树的方式，允许使用者以各种方式遍历抽象语法树。

Clang 的抽象语法树中有两个基本类，其中 Stmt 为语句类的基本类，而 Decl 是所有声明类型的基本类，其大致结构如图 4-1 所示。由于构建过程的基本部分是翻译单元的编译，因此根节点的类型是 TranslationUnitDecl。Clang 通过 RecursiveASTVisitor 模板类提供了一种非常有效的抽象语法树遍历方法，用该方法可以遍历树的每个节点，并在遍历到特定类型的节点时执行操作。由于在 Clang 中抽象语法树是不可变的，源代码转换基于字符串操作，为此，Clang 的抽象语法树节点包含了源代码实际文本存储位置的格式化信息。源代码转换操作由 Replacement 对象组成，其中包含位置信息和要替换的字符串。因此在代码转换时，除了 Replacement 中要替换的字符串外，源代码的其他部分均不会被更改。

```
`-FunctionDecl 0x55cc013d6f88 <test.c:5:1, line:35:1> line:5:5 main 'int (void)'
  `-CompoundStmt 0x55cc013d8d78 <col:16, line:35:1>
    |-DeclStmt 0x55cc013d70c0 <line:6:3, col:12>
    | `-VarDecl 0x55cc013d7040 <col:3, col:11> col:7 used a 'int' cinit
    |   `-IntegerLiteral 0x55cc013d70a0 <col:11> 'int' 0
    |-DeclStmt 0x55cc013d7170 <line:7:3, col:12>
    | `-VarDecl 0x55cc013d70f0 <col:3, col:11> col:7 used b 'int' cinit
    |   `-IntegerLiteral 0x55cc013d7150 <col:11> 'int' 1
    |-DeclStmt 0x55cc013d7220 <line:8:3, col:12>
    | `-VarDecl 0x55cc013d71a0 <col:3, col:11> col:7 used c 'int' cinit
    |   `-IntegerLiteral 0x55cc013d7200 <col:11> 'int' 0
    |-DeclStmt 0x55cc013d72d0 <line:9:3, col:12>
    | `-VarDecl 0x55cc013d7250 <col:3, col:11> col:7 used d 'int' cinit
    |   `-IntegerLiteral 0x55cc013d72b0 <col:11> 'int' 1
    |-IfStmt 0x55cc013d8708 <line:11:3, line:17:3>
    | |-<<<NULL>>>
    | |-<<<NULL>>>
    | |-BinaryOperator 0x55cc013d8500 <line:11:8, col:29> 'int' '&&'
    | | |-BinaryOperator 0x55cc013d7348 <col:8, col:10> 'int' '<'
    | | | |-ImplicitCastExpr 0x55cc013d7330 <col:8> 'int' <LValueToRValue>
    | | | | `-DeclRefExpr 0x55cc013d72e8 <col:8> 'int' lvalue Var 0x55cc013d7040 'a' 'int'
    | | | `-IntegerLiteral 0x55cc013d7310 <col:10> 'int' 5
    | | `-ParenExpr 0x55cc013d84e0 <col:15, col:29> 'int'
    | |   `-BinaryOperator 0x55cc013d84b8 <col:16, col:28> 'int' '||'
    | |     |-BinaryOperator 0x55cc013d73d0 <col:16, col:19> 'int' '>='
    | |     | |-ImplicitCastExpr 0x55cc013d73b8 <col:16> 'int' <LValueToRValue>
    | |     | | `-DeclRefExpr 0x55cc013d7370 <col:16> 'int' lvalue Var 0x55cc013d70f0 'b' 'int'
    | |     | `-IntegerLiteral 0x55cc013d7398 <col:19> 'int' 0
    | |     `-BinaryOperator 0x55cc013d8490 <col:24, col:28> 'int' '!='
    | |       |-ImplicitCastExpr 0x55cc013d7440 <col:24> 'int' <LValueToRValue>
    | |       | `-DeclRefExpr 0x55cc013d73f8 <col:24> 'int' lvalue Var 0x55cc013d71a0 'c' 'int'
    | |       `-IntegerLiteral 0x55cc013d7420 <col:28> 'int' 0
```

图 4-1 Clang 的抽象语法树

Figure 4-1 Abstract syntax tree of clang

Clang 作为 LLVM 的编译前端，为开源项目提供了三种不同的开发接口，分别是 LibClang、Clang Plugins 和 Clang LibTooling[64]。其中 LibClang 可以为开发者提供 C/C++语言之外的 clang 接口，并且对抽象语法树进行更高级的抽象，可以使开发者不用了解其中具体的细节就能使用，但是该接口不支持完全控制 Clang 的抽象语法树，也就是无法进行源代码的转换。Clang Plugins 允许开发者编写附加在 Clang 上的插件，该插件将作为 Clang 编译器的一部分，在编译器工作时增加一些附加操作，该方法虽然能够完全控制抽象语法树，但是其作为插件并不能脱离 Clang 单独运行。Clang LibTooling 是为 C++开发提供的接口，可以让开发者编写独立的 Linux 命令行工具对代码进行操作，其缺点是开发者必须熟悉 Clang 的抽象语法树结构。本工具基于 Clang LibTooling 进行开发。

4.1.2 总体框架

为使工具拥有良好的易修改性和可扩展性，用于 MC/DC 测试用例生成的代码转换工具主要分为四个模块，分别为代码解析模块，约束生成模块，代码重写模块和用户选项模块，具体如图 4-2 所示。其中代码解析模块负责解析从 Clang 得到的抽象语法树，对其进行遍历并寻找要处理的结点；约束生成模块负责分析找到的控制语句，对其进行拆分并生成要插入的约束语句；代码重写模块负责将

生成的约束语句插入到代码中的指定的位置，并将最终结果输出为文件；可选参数模块负责解析用户从命令行传入的参数，并供其它模块使用。

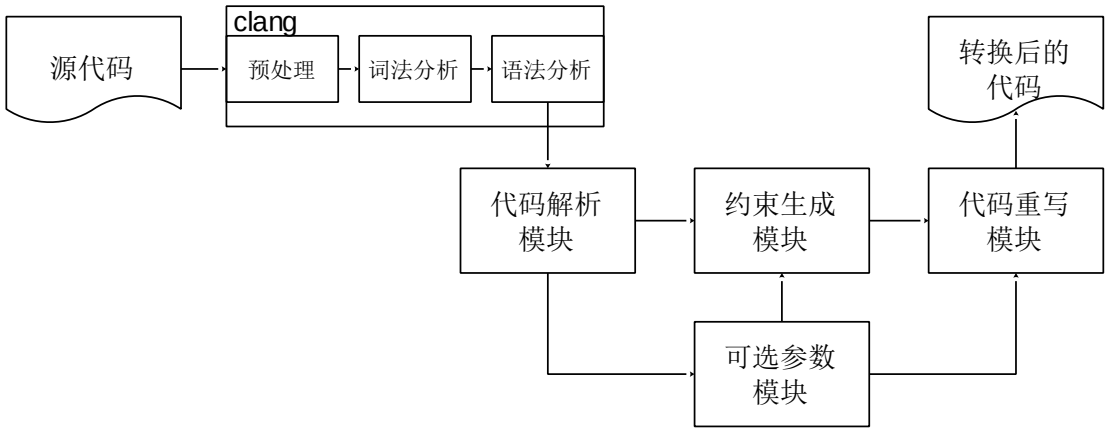


图 4-2 工具的整体架构

Figure 4-2 The framework for tool

工具的总体流程如下：首先使用 clang 编译器解析源代码并获得程序的抽象语法树，然后代码解析模块负责递归的遍历抽象语法树并找到源代码中所有的控制语句，这些被识别的语句将交给约束生成模块并针对每个条件生成适当的约束语句，最后代码重写模块会将约束语句插入到特定的位置，并生成代码文件。如果使用者在执行工具时添加了额外的参数，则可选参数模块会分析这些参数并控制其他模块执行不同的行为。

4.2 代码解析模块

代码解析模块主要的功能是接受 Clang 编译器生成的抽象语法树，对其进行遍历和分析，找到要转换的控制语句，并将其提供给约束生成模块做进一步处理。

Clang 编译器在编译 C 源文件时，会自动读取它所包含的头文件，并将所有包含的头文件看作是一份完整的代码进行处理。由于我们只希望对用户从命令行输入的源代码文件进行代码转换，而不希望将所有头文件一并进行转换，所以该模块还需要在找到控制语句时，判断该语句所在的位置是否属于用户从命令行输入的文件，如果不属于则该控制语句是头文件中的语句，不需要进行转换。

本模块主要基于 Clang 的 CompilerInstance 类，即编译器实例类。该类的对象表示一次编译的编译器实例，其中关联了进行一次编译所需要的所有信息，如编译选项，头文件路径等。

代码分析模块中的主要方法和相关功能说明如表 4-1 所示。

表 4-1 代码分析模块的主要方法

Table 4-1 The main method of the code analysis module

函数返回值	函数名及参数	功能
const LangOptions &	getLangOpts() const	获取源代码语言选项，包括语言类型和语言标志等
ASTContext &	getASTContext() const	从翻译单元获取抽象语法树的上下文
SourceManager &	getSourceManager()	获取源代码的管理器实例
FileID	getMainFileID () const	获取由命令行传入的源文件的 FileId
bool	isWrittenInMainFile (Source-Location Loc) const	判断给定位置是否在主文件缓冲区内
bool	TraverseStmt (Stmt *S)	根据 Clang 提供的动态类型递归的访问抽象语法树
bool	VisitIfStmt(IfStmt *stmt)	访问节点类型为 IfStmt 的节点
bool	VisitForStmt(ForStmt *stmt)	访问节点类型为 ForStmt 的节点
bool	VisitWhileStmt(WhileStmt *stmt)	访问节点类型为 WhileStmt 的节点
bool	VisitDoStmt(DoStmt *stmt)	访问节点类型为 DoStmt 的节点
SourceRange	getSourceRange () const	获得指定语句在源代码中的位置区间
SourceLocation	getBeginLoc () const	获得指定语句第一个字符在源代码中的位置
SourceLocation	getEndLoc () const	获得指定语句最后一个字符在源代码中的位置
unsigned	getExpansionLineNumber (SourceLocation Loc, bool *Invalid=nullptr) const	获得指定位置在源代码中所在的行数
unsigned	getExpansionColumnNumber (SourceLocation Loc, bool *Invalid=nullptr) const	获得指定位置在源代码中所在的列数

4.3 约束生成模块

约束生成模块的主要功能是根据代码解析模块找到的控制语句，获取其中的判定表达式，对其进行分析并生成对应约束语句，其功能是当动态符号执行工具对程序的可执行路径进行探索时，会根据约束语句生成对应的路径约束，并根据路径约束生成包含 MC/DC 对的测试用例，最终达到 MC/DC 覆盖。

为了达到这个目的，约束生成模块需要对判定中的每个条件生成对应的约束。正如在 2.1.2 中所提到的，条件是是不包含布尔运算符的布尔表达式，所以该模

块首先会递归的对判定进行拆分，获得判定的条件集合。然后，约束生成模块会针对条件集合里的每个条件生成 if 语句形式的约束，该约束主要分为两部分，第一部分是确保判定中的其它条件不会影响该判定的结果，第二部分是保证对应的条件会触发真假各一次，用来影响整个判定的真假。最后，约束生成模块会将属于该判定的所有条件的约束语句拼接在一起，并传递给代码重写模块进行代码重写。

约束生成模块中的主要方法和相关功能说明如表 4-2 所示。

表 4-2 约束生成模块的主要方法

Table 4-2 The main method of constraint generation module

函数返回值	函数名及参数	功能
std::string	getRewrittenText (SourceRange Range) const	获得缓冲区中指定源代码区间内的文本信息
Expr *	getCond()	获得控制语句的条件表达式
Expr *	IgnoreParens()	消除围绕表达式的所有括号
StringRef	getOpcodeStr () const	获得二元运算表达式的二元运算符
Expr *	getLHS() const	返回二元运算表达式的左表达式
Expr *	getRHS() const	返回二元运算表达式的右表达式

4.4 代码重写模块

代码重写模块的主要功能是将约束生成模块生成的约束插入到对应的控制语句之前，并将修改后的代码输出成 c 代码文件保存到磁盘上。代码重写工作主要由以下几个步骤组成：

1. 对于代码解析模块找到的控制语句，获取它所表示的文本在源代码文件中的起始位置。
2. 将约束生成模块生成的约束语句添加到源文件输入缓冲区中对应的控制语句之前。
3. 将输入缓冲区中的所有内容输出到指定输出流上。根据参数不同，还可能将输入缓冲区中所有的更改保存到磁盘上，或将修改后的缓冲区内容打印到终端上。

Clang 的抽象语法树是不可变的，源代码转换基于字符串操作。由于这个原因，在 Clang 编译的过程中，抽象语法树的每个节点都会记录该节点所表示的代码在源代码中的起始位置和结束位置。

代码重写模块中的主要方法和相关功能说明如表 4-3 所示。

表 4-3 代码重写模块中的主要方法

Table 4-3 The main method in the code rewriting module

函数返回值	函数名及参数	功能
FileID	getMainFileID () const	获取由命令行传入的源文件的 FileId
RewriteBuffer &	getEditBuffer (FileID FID)	获取指定文件的输入缓冲区
bool	InsertText (SourceLocation Loc, StringRef Str, bool InsertAfter=true, bool indentNewLines=false)	将指定的字符串插入输入缓冲区中的指定位置
bool	ReplaceText (SourceRange range, StringRef NewStr)	用新字符串替换输入缓冲区中指定区间的字符串
bool	IncreaseIndentation (SourceRange range, SourceLocation parentIndent)	增加缓冲区中指定源代码区间的缩进
raw_ostream &	write (raw_ostream &Stream) const	将缓冲区中所有的内容输出到指定输出流中
bool	overwriteChangedFiles ()	将缓冲区中所有修改保存到硬盘上

4.5 可选参数模块

由于本工具是一个独立的 linux 命令行工具，并不依赖于 clang 编译器，因此提供一些 linux 风格的命令行参数。本模块的作用是对命令行进行解析，并改变工具的行为。该模块基于 Clang 的 CommandLine 库进行实现，其中每一个参数对应一个全局变量，当使用者从命令行输入相关参数时，可选参数模块会捕获到这些参数，并改变对应全局变量的值。其它模块在运行过程中会对与其相关的全局变量值进行判断，并做出对应的行为。

其中命令行参数和相关说明如表 4-4 所示。

表 4-4 命令行参数列表

Table 4-4 Command line argument list

命令行参数	说明	可选性	默认可见性
-help	显示帮助信息	可选	可见
-help-hidden	显示全部帮助信息	可选	隐藏
-version	查看版本信息	可选	可见
-o=<filename>	指定输出文件名	必选	可见
-I=<dir>	添加头文件搜索路径	可选	隐藏
-p=<path>	指定输出路径	可选	隐藏
-d	显示详细的转换内容	可选	可见
-r	将修改保存到源文件	可选	可见
-t	将缓冲区内容输出到终端	可选	可见
-extra-arg=<string>	提供给编译器的附加参数	可选	隐藏

4.6 本章小结

本章介绍了用于 MC/DC 测试用例生成的代码转换工具的设计与实现。该工具主要基于 Clang 编译器实现，在第一节中给出了 Clang 编译器的基本介绍，并介绍了工具的总体框架；第二到第四节分别介绍了该工具不同模块的主要功能及常用函数；第五节介绍了使用该工具时可以选择的参数。

第5章　实验评估

本章介绍了基于本文中方法的实验评估。在 5.1 节中将会介绍实验的整体流程。在 5.2 节中将会介绍用于评估的程序，以及在实验过程中针对不同程序进行测试生成时所使用的参数。在 5. 3 节中将会展示实验的结果，并对结果进行分析。

5.1　整体流程

本实验的测试内容主要是基于代码转换的 MC/DC 测试用例生成技术与传统的基于动态符号执行的测试生成技术的比较。实验主要分为三部分，整体流程如图 5-1 所示。实验的第一部分直接使用动态符号执行工具 KLEE 对待测程序进行测试生成，该部分也是目前已有的测试生成方法。实验的第二部分首先对待测程序的源代码进行代码转换，然后使用动态符号执行工具 KLEE 对转换后的代码进行测试生成，并生成对应的测试用例，该部分基于本文中提出的方法。实验的第三部分使用代码插装器对待测程序的源代码进行插桩，并将插桩后的程序编译并分别执行实验第一部分和第二部分所生成的测试用例，输出插桩信息并统计 MC/DC 覆盖率，最后对前两部分生成的测试用例所能达到的 MC/DC 覆盖率进行对比。用于实验评估的程序及使用动态符号执行工具 KLEE 进行测试生成时的参数会在 5.2 节详细介绍。

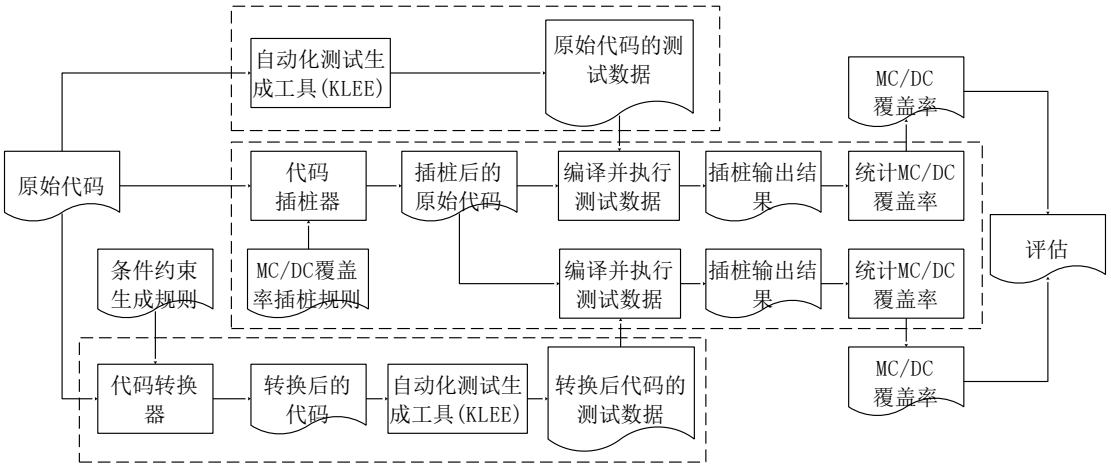


图 5-1 实验流程

Figure 5-1 Experiment process

为了研究本文方法的有效性，实验主要关注两个方面。第一个方面是 MC/DC 覆盖率，也就是本方法能不能比现有方法更有效的生成符合 MC/DC 准侧的测试用例。第二个方面是测试生成的效率，也就是通过本文方法对待测程序源代码添

加的条件约束，是否会降低符号执行工具的效率。

为了确保实验结果的准确，实验的第一部分和第二部分均在相同的环境下进行测试生成，并使用相同的参数进行设置。在比较执行时间时，由于试验的第二部分先进行代码转换，再进行测试生成，因此应该用这两项的时间之和与第一部分进行比较，但是基于本文方法所实现的代码转换器在进行代码转换时，用时均为毫秒级别，所以本实验在对试验用时进行比较时，只比较测试用例生成部分所花费的时间。

我们统计的 MC/DC 覆盖率只考虑待测程序本身的代码，而不对库文件代码进行统计，这是基于两点考虑：首先，由于许多应用程序通常会调用相同的库函数，对库文件进行 MC/DC 覆盖率统计会对许多判定进行重复计算，这可能导致不同程序的覆盖率几乎相同；其次，对库文件进行统计会大幅度的降低所得到的 MC/DC 覆盖率，因为应用程序通常只会用到库文件中很少一部分的代码，例如调用 printf 函数打印整形变量时，程序不会执行到有关打印浮点数或字符串的代码。

实验在具有 30GB 内存和 1.70GHz Intel(R) Xeon(R) Bronze 3104 CPU 的工作站上进行，该 CPU 具有 6 个核心和 8.25MB L3 缓存。

5.1.1 MC/DC 覆盖率统计工具

由于现有的动态符号执行方法旨在实现较高的行覆盖率和分支覆盖率，因此有许多可靠的工具可以测量这些覆盖率，例如 Gcov[8]等。但是本文的目标是提供一种能够基于现有动态符号执行技术生成满足 MC/DC 准则测试用例的技术，我们需要对生成的测试用例所能达到的 MC/DC 覆盖率进行测量。目前有一些工具，例如 vectorCAST[65]等，能够测量 MC/DC 覆盖率并且足够可靠，但是这些工具都不能免费获得。由于开发和验证这种工具所需的高昂代价，这些软件均为商业软件，并且不提供给个人研究使用。基于这种原因，本文实现了一个简单的插桩工具用于统计程序执行时每个条件和判定的取值，并编写对应的 python 脚本对 MC/DC 覆盖率进行统计。

对于相同的程序和相同的测试用例集，其 MC/DC 覆盖率可能不同，这是因为目前没有一个被普遍认可的计算 MC/DC 覆盖率的定义。不同的工具会使用不同的原理：有些工具会根据单独影响判定结果的条件数与总条件数之比计算 MC/DC 覆盖率；另一些工具则通过当前的测试用例数量与满足 MC/DC 覆盖所需的测试用例数量之比计算 MC/DC 覆盖率。对于一个能够达到 100%覆盖率的 MC/DC 测试用例集来说，上述两种方法的计算结果是一样的。虽然这两种计算

方式都有一些道理，但是本文的任务并不是评价这两种计算方式的好坏。在本文中，我们使用前一种方法，也就是通过“能够单独影响判定结果的条件数与总条件数之比”来计算 MC/DC 覆盖率。

```
1   #include <stdio.h>
2   int main(void) {
3           a = input();
4           b = input();
5           c = input();
6
7           if (a<5 && b>=0) {
8                   a++;
9           } else if (c == 1) {
10                  b++;
11          } else {
12                  c++;
13          }
14          while (a < 5) {
15                  a++;
16          }
17          return 0;
18  }
```

图 5-2 插桩示例程序

Figure 5-2 Example program for instrumentation

为了获得某一个判定的 MC/DC 覆盖率，我们需要知道在程序执行过程中该判定的真假，以及判定中每个条件的真假。对于 if 语句来说，插桩工具会在 if 语句之前进行插桩，将每个条件真假以及整个判定的真假输出到指定流文件中，以图 5-2 中的程序为例，对于 if 语句具体的插桩结果如图 5-3 所示。

```
1   fprintf(fp,"%s\n","mcdc statistics mark");
2   fprintf(fp,"%s\n","1");
3   fprintf(fp,"%s\n","2");
4   int tempBool0 = (a<5);
5   fprintf(fp,"%s%d\n","(a<5):",tempBool0);
6   int tempBool1 = (b>=0);
7   fprintf(fp,"%s%d\n","(b>=0):",tempBool1);
8   int tempBool2 = tempBool0 && tempBool1;
9   fprintf(fp,"%s%d\n","(a<5 && b>=0):",tempBool2);
10  fprintf(fp,"%s\n","mcdc statistics mark");
11  fprintf(fp,"%s\n","2");
12  fprintf(fp,"%s\n","1");
13  int tempBool3 = (c == 1);
14  fprintf(fp,"%s%d\n","(c == 1):",tempBool3);
15  int tempBool4 = tempBool3;
16  fprintf(fp,"%s%d\n","(c == 1):",tempBool4);
17  if (a<5 && b>=0) {
18      a++;
19  } else if (c == 1) {
20      b++;
21  } else {
22      c++;
23  }
```

图 5-3 插桩后的 if 语句

Figure 5-3 The instrumented if statement

对于 for 语句，while 语句和 do-while 语句，由于再循环过程中会对判定的真假进行多次判定，所以需要在循环体的最开始和循环语句之后进行插桩，以 while 语句为例，具体的插桩结果如图 5-4 所示。当 for 语句、while 语句和 do-while 语句的循环体只有单行语句且没有使用大括号括起来时，在对其循环体内进行插桩时需要给整个循环体的开头和结尾加上大括号。

插桩的语句会多次计算条件的真假值，如果条件中存在副作用，则会触发这些副作用。例如对于判定((++i<0) && flag)，其中条件(++i<0)存在副作用，不应该被重复计算。对于这种含有副作用的程序，可以先将其转换成不包含副作用的程序再进行插桩[61, 62]。

```
1  while ( a < 5 ) {
2        fprintf(fp,"%s\n","mcdc statistics mark");
3        fprintf(fp,"%s\n","3");
4        fprintf(fp,"%s\n","1");
5        int tempBool5 = (a < 5);
6        fprintf(fp,"%s%d\n","(a < 5):",tempBool5);
7        int tempBool6 = tempBool5;
8        fprintf(fp,"%s%d\n","(a < 5):",tempBool6);
9        a++;
10 }
11 fprintf(fp,"%s\n","mcdc statistics mark");
12 fprintf(fp,"%s\n","3");
13 fprintf(fp,"%s\n","1");
14 int tempBool5 = (a < 5);
15 fprintf(fp,"%s%d\n","(a < 5):",tempBool5);
16 int tempBool6 = tempBool5;
17 fprintf(fp,"%s%d\n","(a < 5):",tempBool6);
```

图 5-4 插桩后的 while 语句

Figure 5-4 The instrumented while statement

使用经过插桩的程序运行测试用例，可以在程序每次运行时得到当次执行各个判定即条件的真假值，输出结果片段如图 5-5 所示。可以看到该插桩输出结果记录了程序中每个判定的真假和判定中每个条件的真假。以判定(a<5&&b>=0)为例，第 1 至 6 行记录了该次执行时条件 a<5 的计算结果为真，条件 b>=0 的计算结果为真，并且整个判定的计算结果也为真；第 12 至 17 行记录了该次执行时条件 a<5 的计算结果为假，条件 b>=0 的计算结果为真，整个判定的计算结果为假。由此可见在条件 b>=0 的真假性不变的情况下，条件 a<5 的真假性改变，并单独影响了判定结果，导致整个判定的真假性发生改变，即对于该输出结果片段而言，只有条件 a<5 被 MC/DC 覆盖条件 b>=0 和 c==1 未被 MC/DC 覆盖，MC/DC 覆盖率为 33.33%。

```
1  mcdc statistics mark
2  1
3  2
4  (a<5):1
5  (b>=0):1
6  (a<5 && b>=0):1
7  mcdc statistics mark
8  2
9  1
10 (c == 1):0
11 (c == 1):0
12 mcdc statistics mark
13 1
14 2
15 (a<5):0
16 (b>=0):1
17 (a<5 && b>=0):0
```

图 5-5 插桩输出结果片段

Figure 5-5 Fragment of the instrumentation output result

5.2 用于评估的程序及参数

5.2.1 用于评估的程序

我们使用 26 个程序来评估本文所给出的方法，包括 14 个小型程序和 12 个 Coreutils 程序。14 个小型程序每个只包含简单的条件或循环语句，用于验证本文方法的可行性，而 Unix 的 Coreutils[48]是用于研究符号执行的标准程序组[8, 66, 67]。

对于动态符号执行工具，我们使用基于 LLVM3.4 的 KLEE 来进行测试用例生成工作，并且使用 STP 作为 KLEE 的约束求解器。

5.2.2 参数设置

对于小型程序，我们不使用任何限制条件，让 KLEE 对程序进行完全的符号执行，并生成测试用例。具体来说，我们使用以下参数对小型程序进行测试用例生成：

clang –I klee/include -emit-llvm -c -g example.c

klee --libc=uclibc --posix-runtime --only-output-states-covering-new example.bc --sym-args 0 4 4

对于 Coreutils，由于 KLEE 的参数众多并且非常复杂，改变任何一个参数的

值都有可能会影响整个测试生成的结果，所以我们决定使用 KLEE OSDI'08 论文中所使用的参数，以确保测试用例生成过程的正确性。Coreutils 的大部分功能都可以通过不超过两个的短参数，一个长参数和两个输入流（stdin 和文件）来触发。具体来说，我们使用以下参数对 Coreutils 进行测试用例生成：

klee --simplify-sym-indices --write-cvcs --write-cov --output-module --max-memory=1000 --disable-inlining --optimize --use-forked-solver --use-cex-cache --libc=uclibc --posix-runtime --allow-external-sym-calls --only-output-states-covering-new --environ=test.env --run-in=/tmp/sandbox --max-sym-array-size=4096 --max-instruction-time=30. --max-time=3600. --watchdog --max-memory-inhibit=false --max-static-fork-pct=1 --max-static-solve-pct=1 --max-static-cpfork-pct=1 --switch-type=internal --search=random-path --search=nurs:covnew --use-batching-search --batch-instructions=10000 ./example.bc --sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdin 8 --sym-stdout

由于动态符号执行技术会实际运行程序，在对某些程序进行测试用例生成时，程序可能会做出某些不可预估的行为并破坏实验环境（例如 rm 命令）。因此我们使用一个简单的环境设置和一个“沙箱”目录，来避免测试用例生成期间，程序做出的行为对实验环境产生影响。

另外，KLEE 还提供了一些在符号执行过程中对符号变量可能采用的值进行约束的功能，可以让用户自己定义程序输入空间的边界，或者是让 KLEE 在随机生成测试用例时优先使用某些值。但是为了提高实验的可重复性和一般性，在本实验中我们不使用类似的约束。

5.3 评估结果

在 14 个小型程序上进行试验的结果如表 5-1 和表 5-2 所示。其中表 5-1 是包含两个条件的程序的实验结果，可以看到未经过转换，直接使用 KLEE 进行测试生成时，测试用例的 MC/DC 覆盖率从 50%到 100%不等，这是因为 KLEE 所使用的动态符号执行技术会根据特定的模式生成测试用例，而条件的结构会影响所生成的测试用例的 MC/DC 覆盖率。KLEE 对经过转换后的程序所生成的测试用例的 MC/DC 覆盖率均能达到 100%。对于执行时间而言，增加约束条件意味着符号执行所需要探索的路径增多，由于 if 语句处理起来较为简单，所以执行时间并没有明显差别，而动态符号执行技术处理 while 语句的性能较差，所以可以看到执行时间有所增加。对于包含两个条件的程序来说，由于插入额外约束，导致执行时间平均增加了约 2%。

表 5-1 包含两条件程序的实验结果

Table 5-1 Experimental results of program that contain two conditions

控制语句类型	未经转换		经过转换	
	MC/DC 覆盖率	执行时间(s)	MC/DC 覆盖率	执行时间(s)
if (a && b)	50.00%	0.02	100.00%	0.02
if (a || b)	100.00%	0.02	100.00%	0.02
while (a && b)	50.00%	3.07	100.00%	3.14
while (a || b)	100.00%	3.07	100.00%	3.14

包含三个条件的程序的实验结果如表 5-2 所示。由于条件的组成结构不同，直接由 KLEE 生成的测试用例所能达到的覆盖率从 33.33%到 100%不等，而 KLEE 对经过转换后的程序所生成的测试用例的 MC/DC 覆盖率均能达到 100%。同样，由于在转换时增加了额外的约束，经过转换的版本在进行测试生成时所用的执行时间会高于未经转换的版本，这点在动态符号执行技术处理性能较差的 while 语句上体现尤为明显，平均而言，由于插入额外约束导致执行时间增加了 38.15%

表 5-2 包含三条件程序的实验结果

Table 5-2 Experimental results of program that contain three conditions

控制语句类型	未经转换		经过转换	
	MC/DC 覆盖率	执行时间(s)	MC/DC 覆盖率	执行时间(s)
if (a&&b&&c)	33.33%	0.03	100.00%	0.03
if (a||b||c)	100.00%	0.03	100.00%	0.03
if (a&&(b||c))	66.67%	0.03	100.00%	0.03
if ((a||b)&&c)	33.33%	0.03	100.00%	0.03
if (a||(b&&c))	66.67%	0.03	100.00%	0.03
while(a&&b&&c)	33.33%	132.41	100.00%	182.92
while(a||b||c)	100.00%	132.41	100.00%	182.92
while(a&&(b||c))	66.67%	132.41	100.00%	182.92
while((a||b)&&c)	33.33%	132.41	100.00%	182.92
while(a||(b&&c))	66.67%	132.41	100.00%	182.92

针对 Coreutils 程序的实验结果如表 5-3 所示。从表中可以发现未经转换时，KLEE 提供的分支覆盖测试用例可以提供很高的 MC/DC 覆盖率。这是因为如果程序中不存在包含复杂布尔表达式的判定，那么根据 2.1 节中提到的定义，MC/DC 覆盖与分支覆盖是等价的，在动态符号执行技术进行测试用例生成时，会生成满足 MC/DC 准则的测试用例。例如对于 pwd 程序，源码中包含 26 个判

断语句，其中均只包含一个条件，所以使用 KLEE 直接对其进行测试用例生成，测试用例的 MC/DC 覆盖率就可以达到 100%。经过转换的程序同样可以生成满足 MC/DC 准则的测试用例，这证明本方法不会影响动态符号执行技术，降低原本能够达到的 MC/DC 覆盖率。

在表 5-3 中，有些程序经过转换后仍然不能达到 100%的 MC/DC 覆盖率，这是因为有些条件无法被 MC/DC 覆盖，也就是该条件无法独立的影响整个判定的结果。这是有可能发生的，例如 while(true)等无限循环，或者某些防御性编程风格中含有的包括“不可能发生”的安全性检查。统计的覆盖率不包含死代码，因为本方法是使用插桩的方式，在程序运行时输出对应的信息，而死代码永远都不会被执行到，也就不会输出任何相关的信息。

平均来说，在使用相同的时间进行测试用例生成时，应用本方法所生成的测试用例可以将 MC/DC 覆盖率提高 14.66%，达到 98.01%，并且除了个别无法被 MC/DC 覆盖的条件外，应用本方法所生成的测试用例可以达到 100%MC/DC 覆盖率。

表 5-3 Coreutils 的实验结果

Table 5-3 Experimental results of Coreutils

程序名称	未经转换		经过转换		
	MC/DC 覆盖率	测试用例数	MC/DC 覆盖率	测试用例数	额外满足的条件数
base64	77.50%	79	100.00%	87	9
basename	81.82%	38	100.00%	44	2
cat	80.60%	66	94.03%	73	9
cp	92.65%	112	100.00%	112	5
csplit	87.20%	114	96.00%	114	11
date	91.67%	215	97.22%	256	2
du	79.69%	54	95.31%	88	10
echo	73.91%	35	100.00%	37	6
env	69.23%	37	100.00%	39	4
expand	90.91%	71	95.45%	79	2
ls	87.09%	176	98.08%	178	40
uname	87.88%	44	100.00%	46	4

5.4 本章小结

本章主要介绍了用于评估本文方法的实验设计和结果分析。第一节介绍了实验的整体流程，并介绍了一个本文为了统计 MC/DC 覆盖率而设计并实现的工具；第二节详细介绍了用于实验的程序以及用于生成测试用例的参数设置；第三节展示了实验结果，并对结果进行了分析。

实验结果表明，对于小型程序，应用本文方法生成的测试用例均能达到 100%MC/DC 覆盖率；对于 Coreutils 程序，应用本文方法生成的测试用例，MC/DC 覆盖率平均提高了 14.66%，达到 98.01%。

结 论

本文提出了一种基于代码转换的测试用例生成技术，该技术基于可测试性转换理论，在待测程序源代码中添加额外的条件约束，并使用动态符号执行技术进行测试用例生成，最终使得生成的测试用例满足 MC/DC 准则。本文基于 clang 设计并开发了一种代码转换工具，用于实现本文中的方法，在程序源码中添加条件约束，使用现有的动态符号执行工具 KLEE 进行测试生成，设计并开发了一种代码插桩器用于统计测试用例所能达到的 MC/DC 覆盖率。本文将方法应用于小型程序和 Coreutils 程序进行评估，实验结果显示本方法可以使 MC/DC 覆盖率提高，并在刨除不能被 MC/DC 覆盖的条件后，能够达到 100%MC/DC 覆盖率。

本文的另一主要贡献在于提出了一种在软件测试时提高测试用例质量的新方法，即根据可测试性转换理论在程序源代码中添加条件约束，并使用动态符号执行技术进行测试用例生成。本文已经证明了通过该方法生成的测试用例可以满足 MC/DC 准则这种要求非常严格的覆盖准则。由于大多数常用覆盖准则对条件的约束要求比 MC/DC 准则低，并且相对容易定义，因此可以使用相同的基于动态符号执行技术的方法来生成满足其他准则的测试用例。

目前，本文只将文中提出的方法应用于 C 语言程序并做了相关实验评估，这主要是因为本文所选的动态符号执行工具 KLEE 对 C 语言有较好的支持。未来可以将该方法与更多的动态符号执行工具组合使用，将本文中的方法扩展到除了 C 语言之外的其他编程语言。

应用本文中的方法会在程序中加入额外的条件约束，会使动态符号执行技术进行测试用例生成时探索更多的指令，导致测试生成时间有所增加。有一些方法可以改善这种状况，例如新的动态符号执行优化技术，或者将动态符号执行搜索算法与条件约束相关联。由于条件约束本质上描述了一系列测试人员感兴趣的变量值，因此与单纯的使用深度优先算法或启发式算法相比，使用尽可能优先探索条件约束的搜索策略应该能在更短的时间内生成同样数量的测试用例。这也是未来工作的重点。

参考文献

[1] YU J. Review of Panorama and Key Technical Prospect on Software Testing. Proceedings of the 2018 7th International Conference on Software and Computer Applications, Kuantan, Malaysia, 2018[C]. ACM, 2018: 61-5.

[2] BOUNIMOVA E, GODEFROID P, MOLNAR D. Billions and billions of constraints: whitebox fuzz testing in production. Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013[C]. IEEE Press, 2013: 122-31.

[3] FERGUSON R, KOREL B. The chaining approach for software test data generation [J]. ACM Trans Softw Eng Methodol. 1996, 5(1): 63-86.

[4] KING J C. Symbolic execution and program testing [J]. Commun ACM. 1976, 19(7): 385-94.

[5] GODEFROID P, KLARLUND N, SEN K. DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, IL, USA, 2005[C]. 1065036, ACM, 2005: 213-23.

[6] SEN K, MARINOV D, AGHA G. CUTE: a concolic unit testing engine for C. Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal, 2005[C]. 1081750, ACM, 2005: 263-72.

[7] GODEFROID P, LEVIN M Y, MOLNAR D A. Automated Whitebox Fuzz Testing. 2008[C]. 2008.

[8] CADAR C, DUNBAR D, ENGLER D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008[C]. USENIX Association, 2008: 209-24.

[9] BURNIM J, SEN K. Heuristics for Scalable Dynamic Test Generation. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008[C]. IEEE Computer Society, 2008: 443-6.

[10] GODEFROID P, LEVIN M Y, MOLNAR D. SAGE: whitebox fuzzing for security testing [J]. Commun ACM. 2012, 55(3): 40-4.

[11] GANESH V, DILL D L. A decision procedure for bit-vectors and arrays. Proceedings of the 19th international conference on Computer aided verification, Berlin, Germany, 2007[C]. Springer-Verlag, 2007: 519-31.

[12] "STP constraint solver." http://stp.github.io/. 2018.

[13] RIERSON L. Developing safety-critical software : a practical guide for aviation software and DO-178c compliance [M]. CRC Press, 2013: 610.

[14] FANG C R, CHEN Z Y, XU B W. Comparing logic coverage criteria on test case pr ioritization [J]. Science China(Information Sciences). 2012, 12): 2826-40.

[15] DUPUY A, LEVESON N. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. 19th DASC 19th Digital Avionics Systems Conferen ce Proceedings (Cat No00CH37126), 2000[C]. 2000: 1B6/1-B6/7 vol.1.

[16] J. H K, S. V D, J. C J, et al. A Practical Tutorial on Modified Condition/Decision C overage [R]: NASA Langley Technical Report Server, 2001.

[17] CADAR C, GANESH V, PAWLOWSKI P M, et al. EXE: automatically generating in puts of death. Proceedings of the 13th ACM conference on Computer and communica tions security, Alexandria, Virginia, USA, 2006[C]. 1180445, ACM, 2006: 322-35.

[18] CHILENSKI J J, MILLER S P. Applicability of modified condition/decision coverage to software testing [J]. Software Engineering Journal. 1994, 9(5): 193-200.

[19] RAJAN A, WHALEN M W, HEIMDAHL M P E. The effect of program and model structure on mc/dc test adequacy coverage. Proceedings of the 30th international conf erence on Software engineering, Leipzig, Germany, 2008[C]. ACM, 2008: 161-70.

[20] 岳海, 任慧敏. 基于修正条件判定覆盖的软件测试技术研究和方法应用 [J]. 航天控制. 2012, 03): 69-72.

[21] MITRA P, CHATTERJEE S, ALI N. Graphical analysis of MC/DC using automated s oftware testing. 2011 3rd International Conference on Electronics Computer Technolog y, 2011[C]. 2011: 145-9.

[22] 黄孝伦. 基于图的MC/DC最小测试用例集快速生成算法 [J]. 计算机系统应用. 2012, 1 1): 145-8.

[23] 郑平, 许胜. 基于MC/DC准则的确认测试用例生成方法 [J]. 现代电子技术. 2007, 16): 114-7.

[24] HARMAN M, HU L, HIERONS R, et al. Testability Transformation [J]. IEEE Trans Softw Eng. 2004, 30(1): 3-16.

[25] PANDITA R, XIE T, TILLMANN N, et al. Guided test generation for coverage criter ia. Proceedings of the 2010 IEEE International Conference on Software Maintenance, 2010[C]. IEEE Computer Society, 2010: 1-10.

[26] GODBOLEY S, PRASHANTH G S, MOHAPATRO D P, et al. Increase in Modified Condition/Decision Coverage using program code transformer. 2013 3rd IEEE Internati onal Advance Computing Conference (IACC), 2013[C]. 2013: 1400-7.

[27] SU T, PU G, FANG B, et al. Automated Coverage-Driven Test Data Generation Usin g Dynamic Symbolic Execution. 2014 Eighth International Conference on Software Se curity and Reliability (SERE), 2014[C]. 2014: 98-107.

[28] WU T, YAN J, ZHANG J. Automatic Test Data Generation for Unit Testing to Achie ve MC/DC Criterion. Proceedings of the 2014 Eighth International Conference on Soft ware Security and Reliability, 2014[C]. IEEE Computer Society, 2014: 118-26.

[29] "KLEE LLVM Execution Engine." http://klee.github.io/. 2017.

[30] 俞祥贤. 基于遗传算法的 MC/DC 测试用例自动生成方法研究 [D]. 南昌航空大学. 2015.

[31] PELED D A, GRIES D, SCHNEIDER F B. Software reliability methods [M]. Springer-Verlag, 2001: 331.

[32]苏亭. 基于覆盖准则的软件测试用例自动化生成方法的研究与实现 [D]. 华东师范大学. 2016.

[33] 陈明珠, 孙志安. 基于 MC/DC 覆盖的白盒测试用例设计方法研究. 中国造船工程学会电子技术学术委员会 2017 年装备技术发展论坛, 中国浙江杭州, 2017[C]. 2017: 3.

[34] YANG L, YAN J, ZHANG J. Generating minimal test set satisfying MC/DC criterion via SAT based approach. Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau, France, 2018[C]. ACM, 2018: 1899-906.

[35] RUSHBY J. New challenges in certification for aircraft software. 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT), 2011[C]. 2011: 211-8.

[36] 李树芳, 安金霞, 刘洋, 等. 采用 Clang/LLVM 的 C++源代码覆盖率分析插装方法 [J]. 计算机科学. 2017, 44(11): 191-4.

[37] ARCURI A. An experience report on applying software testing academic results in industry: we need usable automated test generation [J]. Empirical Software Engineering. 2018, 23(4): 1959-81.

[38] BALDONI R, COPPA E, D'ELIA D C, et al. A Survey of Symbolic Execution Techniques [J]. ACM Comput Surv. 2018, 51(3): 1-39.

[39] BIRD D L, MUNOZ C U. Automatic generation of random self-checking test cases [J]. IBM Systems Journal. 1983, 22(3): 229-45.

[40] FORRESTER J E, MILLER B P. An empirical study of the robustness of Windows NT applications using random testing. Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4, Seattle, Washington, 2000[C]. USENIX Association, 2000: 6-.

[41] OFFUTT A J, HAYES J H. A semantic model of program faults. Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis, San Diego, California, USA, 1996[C]. ACM, 1996: 195-200.

[42] SEN K. Scalable automated methods for dynamic program analysis [D]. University of Illinois at Urbana-Champaign. 2006: 129.

[43] VISSER W, PASAREANU C S, KHURSHID S. Test input generation with java Path Finder. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA, 2004[C]. 1007526, ACM, 2004: 97-107.

[44] LATTNER C, ADVE V. LLVM: a compilation framework for lifelong program analys

is transformation. International Symposium on Code Generation and Optimization, 2004 CGO 2004, 2004[C]. 2004: 75-86.

[45] GOSLING J, JOY B, GUY L. STEELE J, et al. The Java Language Specification, Java SE 7 Edition [M]. Addison-Wesley Professional, 2013: 672.

[46] 孙溢, 阳小华, 刘杰, 等. 基于布尔表达式约束的测试用例生成技术 [J]. 计算机与现代化. 2019, 01): 86-94.

[47] 叶志斌, 严波. 符号执行研究综述 [J]. 计算机科学. 2018, 45(S1): 28-35.

[48] "Coreutils-GNU core utilities." https://www.gnu.org/software/coreutils/. 2019.

[49] 黄琦, 蔡爱华, 吕慧颖, 等. 基于 KLEE 的软件漏洞测试用例自动生成技术 [J]. 计算机工程与设计. 2016, 06): 1515-9+25.

[50] CONVERSE H, OLIVO O, KHURSHID S. Non-Semantics-Preserving Transformations for Higher-Coverage Test Generation Using Symbolic Execution. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017[C]. 2017: 241-52.

[51] HARMAN M. We Need a Testability Transformation Semantics. Software Engineering and Formal Methods, Cham, 2018[C]. Springer International Publishing, 2018: 3-17.

[52] YU Y T, LAU M F. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions [J]. Journal of Systems and Software. 2006, 79(5): 577-90.

[53] BURSTALL R M, DARLINGTON J. A Transformation System for Developing Recursive Programs [J]. J ACM. 1977, 24(1): 44-67.

[54] PARTSCH H A. Specification and transformation of programs: a formal approach to software development [M]. Springer-Verlag, 1990: 493.

[55] WARD M P. Reverse Engineering through Formal Transformation: Knuth's ‘Polynomial Addition’ Algorithm [J]. The Computer Journal. 1994, 37(9): 795-813.

[56] HARMAN M. Refactoring as Testability Transformation. 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011[C]. 2011: 414-21.

[57] HARMAN M, BARESEL A, BINKLEY D, et al. Testability Transformation – Program Transformation to Improve Testability [M]//HIERONS R M, BOWEN J P, HARMAN M. Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers. Berlin, Heidelberg; Springer Berlin Heidelberg. 2008: 320-44.

[58] 左泽轩, 薛战东. 基于 DO-178 的机载软件结构覆盖分析 [J]. 科技视界. 2016, 15): 1-2+25.

[59] MCMINN P. Search-based software test data generation: a survey: Research Articles [J]. Softw Test Verif Reliab. 2004, 14(2): 105-56.

[60] BARESEL A, STHAMER H, SCHMIDT M. Fitness function design to improve evolutionary structural testing. Proceedings of the 4th Annual Conference on Genetic and E

volutionary Computation, New York City, New York, 2002[C]. Morgan Kaufmann Pub lishers Inc., 2002: 1329-36.

[61] HARMAN M, MUNRO M, LIN H, et al. Side-effect removal transformation. Proceed ings 9th International Workshop on Program Comprehension IWPC 2001, 2001[C]. 20 01: 310-9.

[62] DOLADO J J, HARMAN M, OTERO M C, et al. An empirical investigation of the influence of a type of side effects on program comprehension [J]. IEEE Transactions on Software Engineering. 2003, 29(7): 665-70.

[63] ZHANG Z, CHEN Z, GAO R, et al. An empirical study on constraint optimization t echniques for test generation [J]. Science China Information Sciences. 2016, 60(1): 01 2105.

[64] "Choosing the Right Interface for Your Application - Clang 6 documentation." http://r eleases.llvm.org/6.0.1/tools/clang/docs/Tooling.html. 2018.

[65] "VectorCAST." http://www.vectorcast.cn/. 2019.

[66] SIDDIQUI J H, KHURSHID S. Scaling symbolic execution using ranged analysis. Pr oceedings of the ACM international conference on Object oriented programming syste ms languages and applications, Tucson, Arizona, USA, 2012[C]. ACM, 2012: 523-36.

[67] DONG S, OLIVO O, ZHANG L, et al. Studying the influence of standard compiler optimizations on symbolic execution. 2015 IEEE 26th International Symposium on Sof tware Reliability Engineering (ISSRE), 2015[C]. 2015: 205-15.

攻读硕士学位期间发表的学术论文

[1] 用于生成 MC/DC 测试用例的代码转换工具软件 V1.0. 软件著作权登记号：2019SR0153185

[2] MC/DC 覆盖率统计工具软件 V1.0. 软件著作权登记号：2019SR0165794

致　谢