分类号：_____ 密级：_____

UDC：_____ 编号：_____

工学硕士学位论文

白盒测试的方法研究

硕士研究生 ：綦 晶
指导教师 ：孙长嵩 教授
学位级别 ：工学硕士
学科、专业 ：计算机软件与理论
所在单位 ：计算机科学与技术学院
论文提交日期：2008 年 1 月
论文答辩日期：2008 年 3 月
学位授予单位：哈尔滨工程大学

摘　要

针对软件测试中的白盒测试技术，分析了当前白盒测试的主要方法及存在的测试用例繁多，测试不充分，效率低下等问题。为了解决这些问题，重点介绍了 DD 图、流程图等概念，并以构造 DD 图无约束边集合和程序流程树为基础，给出了一个基于程序流程树的测试模型，该模型通过五个步骤来对程序代码进行测试。第一个步骤是提取程序片段，包括重要度评价与程序切分。如果是在集成测试阶段，需要分析模块复杂度，选取当前最重要的模块进行测试。如果是在单元测试阶段，可认为当前的惟一模块就是最重要模块。然后，将复杂程序代码利用程序切片技术，截取为若干有逻辑意义的代码片段。第二个步骤是程序流图构造。按照一定规则将程序流程图转换为程序流图。第三个步骤是 DD 图生成。在生成的 DD 图中提取无约束边集合，用于精简测试路径。第四个步骤是流程树获取，按照保持程序流图逻辑意义不变的原则，将程序流图转换为流程树，目的是为了化解程序流图回路，构造清晰的可测试路径。每一条从树根节点出发到叶子节点为止的路径都是一条可测试的独立路径。所有这些路径的集合覆盖了整个程序段。第五个步骤是最佳测试路径筛选。在流程树所有可测试的独立路径中，遵循尽量充分地覆盖 DD 图无约束边的原则，通过循环计算当前未被选中路径中包含当前未被覆盖的无约束边的个数，选择最佳测试路径，直到无约束边全部被覆盖，精简可测试路径数目，利用最少的测试路径达到最充分的覆盖。此外，分析了基于流程树的白盒测试方法同传统白盒测试方法的区别。最后结合一个实际的软件进行了测试验证。

关键词：软件；白盒测试；路径；DD 图；无约束边；流程树

Abstract

This thesis is set focus on the white-box testing method in the software testing and the problems in the current white-box testing is analysed such as huge testing data, uncompleted testing, and low efficiency. To solve the problems, the conception of DD graph and procedure tree is introduced detailedly and the method model based on procedure tree is presented. The model based on procedure tree describes white-box testing in five steps. In the first step, slice of program codes is selected with analysing the important degrees of programs and slicing the program. If in the integrated testing, it will be necessary to analysing the important degrees of programs to select the most important program to test. If in the unit testing, the only program is the most important program. Then the complicated program is sliced with program slicing technology. In the second step, the program is transformed into the procedure graph. In the third step, free edges assemble in the DD graph is analogized to simplify testing paths. In the fourth step, the procedure tree is created according to logical meaning in the procedure graph. After that, the circle paths in the procedure graph are transformed into branches in the procedure tree. Each path whose starting point is the root and terminal point is a leaf in the tree, is a testing path. The whole program is covered by all the paths. In the fifth step, the best testing paths are selected according to covering more free edges. Furthermore, the conception is clarified from the white-box testing method based on procedure tree, and traditional white-box testing method. The theory is checked practically by software.

Keywords: software; white-box testing; path; DD graph; free edges; procedure tree

哈尔滨工程大学
学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除文中已注明引用的内容外，本论文不包含任何其他个人或集体已经公开发表的作品成果。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。本人完全意识到本声明的法律结果由本人承担。

作者(签字)：蔡晶

日期：2008年 3月 12日

第 1 章 绪论

1.1 课题的目的及意义

信息技术极大地促进了社会的进步和人类文明的发展，已成为当今社会发展的主要动力之一。软件是信息技术的重要组成部分。近年来，世界软件产业取得了突飞猛进的发展，软件产业在一些主要发达国家已被提到了国民经济的主导产业。目前，全球软件从业人员已接近 1000 万人，每年生产数百万个中等规模以上的软件，全球软件及信息服务业市场已达到了近 1 万亿美元。其中，软件销售额大约为 5000 亿美元，并以每年约 15%的速度在增加[1]。

虽然软件已应用到国民经济的各个领域，特别是在高科技领域和某些国家安全领域，软件技术显得更加重要，但是近些年来，软件的规模在大幅度提高，复杂性也在增加，软件不可靠的矛盾变得越来越突出，由于软件的错误所造成的损失也在大幅度增加，软件不可靠性的矛盾已经到了必须解决的地步。因此，需要大力发展软件测试技术。

软件测试是伴随着软件工程技术发展起来的。自 20 世纪 70 年代诞生至 20 世纪 90 年代初期，发展一直比较缓慢，主要原因是社会需求不足，认识不到位，软件结构测试难度大，表现为软件测试的从业人员不多，测试工具比较简陋[2]。近 10 年来，随着人们对软件认识的深入，软件危害的事例也越来越多，软件测试技术受到了广泛的重视。软件测试理论得到了快速发展，诞生了一些新的软件测试方法，参与软件测试的人员也在大幅度增加，高效率的软件测试工具也不断涌现。软件测试技术由于其重要的作用，已经发展成为软件工程中极其重要的环节。

目前软件测试技术分为黑盒测试与白盒测试两大类。黑盒测试技术着重于软件的功能测试，多用于软件测试中期与后期。而白盒测试技术着重于软件的结构测试，多用于软件测试前期与中期，能够较早地发现软件中存在的问题。

软件测试研究的结果表明，软件中存在的问题发现越早，其软件开发费

用就越低。在编码后修改软件缺陷的成本是编码前的 10 倍，在产品交付后修改软件缺陷的成本是是交付前的 10 倍。软件测试技术越精确，软件发布后的维护费用越低。另据对国际著名 IT 企业的统计，它们的软件测试费用占整个软件工程所有研发费用的 50%以上[3]。而白盒测试技术由于是对软件结构的测试，在单元测试阶段就可以应用白盒测试技术，可以更早地对软件程序代码进行测试，因此可以在软件开发的初级阶段就发现软件中存在的缺陷。由此可见，有效的白盒测试技术能够较早发现软件问题，提高软件的测试效率，从而节省软件开发的人力物力资源，是降低软件开发的成本的主要途径。

此外，有效的白盒测试技术可以促使单元测试顺利进行，能够推进软件的开发进度。有效的白盒测试技术由于是对软件的结构、软件的细节问题进行的测试，还可以更加保证软件产品的质量。

但是，就目前软件工程发展的状况而言，白盒测试技术还是相对于其他研究方向而言较为薄弱的一个方面。虽然传统的白盒测试技术，是现在软件结构测试的基础，但由于测试过程复杂且测试效率不高，使得软件结构测试技术在 20 世纪徘徊了 20 余年而进展不大[3]。虽然随着人们对软件结构及软件结构错误认识的不断深入，新的白盒测试技术应运而生。但是，随着计算机技术的发展，软件开发效率的提高，产品投入市场的周期缩短，必然对白盒测试技术提出更高的要求。因此，对白盒测试技术的研究需要不断优化，不断创新，不断发展。

1.2 白盒测试技术的国内外研究现状

早在 20 世纪 50 年代，英国著名的计算机科学家图灵就曾经给出了程序测试的原始定义，测试是正确性确认实验方法的一种极端。之后，随着人们对软件测试的深入认识，将软件测试技术分为白盒测试技术与黑盒测试技术。

20 世纪 70 年代中期以后是白盒测试技术发展最活跃的时期。Brooks 总结了开发 IBM OS/360 操作系统中的经验，阐明了软件结构测试技术即白盒测试技术在研制大系统中的重要意义。1975 年，美国黄荣昌教授在论文中讨论了测试准则、测试过程、路径谓词、测试数据及其生成问题，首次全面系统地论述了白盒测试的有关问题[4]。Hetzel 在 1975 年整理出版了《Program Test

Methods》一书，书中纵览了测试方法以及各种自动测试工具，这是专题论述软件测试也是白盒测试的第一本著作。Goodenough 和 Gerhart 首次提出了白盒测试的理论，从而把白盒测试这一实践性很强的技术提高到理论的高度，被认为是测试技术发展过程中具有开创性的工作。此后不久，著名测试专家 Howden 指出了上述理论的缺陷，并进行了新的开创性工作[5]。以后，Weyuker、Ostrand、Geller 和 Gerhart 等进一步总结原有的白盒测试理论并进一步加以完善，使白盒测试成为有理论指导的实践性技术[6]。

在白盒测试理论迅速发展的同时，各种白盒测试技术也应运而生。黄荣昌教授提出了程序插装测试技术。Howden 在路径分析的基础上，提出了代数测试的概念。Howden、Clarke 和 Darringer 等人提出了符号测试方法，并建立了 DISSET 符号测试系统。Demillo 提出了程序变异测试方法。Osterweil 和 Fosdick 提出了数据流测试方法。White 和 Cohen 提出了域测试方法。Richardson 和 Clarke 提出了划分测试方法[6]。

1979 年，Mark Weiser 先生在他的博士论文中建立起一种程序分解技术[6]。它通过寻找程序内部的相关性来分解程序，再通过对分解所得程序切片的分析达到对整个程序的分析和理解。它极大地提高了白盒测试技术中对程序的分析与理解的效率。

在软件测试技术的学术领域，1982 年，在美国北卡罗来纳大学召开了首届软件测试的正式学术会议，之后，该学术会议每两年召开一次，此外，国际上还有软件可靠性会议，从会议的规模与论文的数量与质量上看，研究软件白盒测试的人员在大幅度地增加。

总之，20 世纪 70 年代—20 世纪 80 年代，是白盒测试技术迅速发展的时期，数十种白盒测试方法被提出，白盒测试技术已经成为软件测试技术学科中最重要的组成部分之一。但总体来看，白盒测试技术的研究主要是在理论上。实用的白盒测试技术并不多见，少数的白盒测试技术由于测试效率不高，也难以进入市场。

我国软件与国际先进软件相比，在质量和成熟度上确实还有一定差距。尽管国外软件也存在不少 BUG(软件错误)和漏洞，但很少存在由于低级失误或大意而出现的软件产品质量问题。但国内软件由于低级错误而造成的严重产品质量问题却不时发生。究其原因有三：其一，国内 IT 行业相对欧美国家

起步较晚，经验积累不多，从业人员都是年轻的新生代，有经验的软件工程师不多，软件测试专家更是凤毛麟角；其二，众多软件开发企业软件产品开发周期短，测试不够精确；第三，国内大专院校基本上没有针对软件测试和质量保证岗位的专业实用课程，目前大多数的测试理论方面的书是翻译过来的，社会上缺乏系统培养专业软件测试人才的有效渠道。

2004 年 8 月，在青海省西宁市召开了“全国首届软件测试技术研讨会”[6]。白盒测试技术中的程序切片技术成为当前软件结构测试方面的研究热点。

可以预测，在未来的时间里，白盒测试技术将会得到更快地发展，主要可能的表现包括：白盒测试理论更加完善，白盒测试效率更加提高，更实用的白盒测试系统将会大量出现。

1.3 工作内容及论文结构

作者围绕提高白盒测试效率，结合实际工程，开展了以下研究：

(1)研究白盒测试相关技术，找出了优化覆盖路径的白盒测试方法。

(2)在分析程序代码结构中控制关系与数据依赖关系基础上，给出了白盒测试模型。

(3)提出了一种基于流程树模型的白盒测试流程。

(4)提出重要度测算，主张做为集成测试阶段模块测试顺序的考虑因素之一。

(5)对提出的流程树模型及测试流程进行实验验证。

全文共分 4 章。第 1 章是绪论，第 2 章是白盒测试相关知识，第 3 章是白盒测试相关技术研究，第 4 章是基于流程树的白盒测试方法，具体的内容为：

第 1 章介绍课题来源、研究意义、现状、主要内容及结构组织。

第 2 章介绍相关知识，先阐述了相关技术研究和测试方法实现中要涉及的基本术语，接着介绍白盒测试的基本方法，最后介绍有关白盒测试建模的理论。

第 3 章对白盒测试技术进行研究。通过结合程序切片、程序流图、路径覆盖原则与 DD 图无约束边理论，给出了一种基于流程树的白盒测试方法。

第 4 章首先对模块的重要度进行测算，按照一定规则对程序进行切片，然后分析 DD 图无约束边的结论，在流程树中选择最佳测试路径，完成对程序代码的白盒测试。

第 2 章 白盒测试相关知识

白盒测试又称结构测试，它允许测试人员利用被测程序内部的逻辑结构和有关信息设计或选择测试用例，对程序所有逻辑路径进行测试。由于白盒测试理论性比较强，因此本章主要介绍有关白盒测试的理论知识。

2.1 基本术语

(1)程序元素：程序元素是指在程序中有意义的，覆盖测试中指定要被执行的语法或语义成分，可以给出程序元素的集合，见式(2-1)：

$$E(P,C)=\{e_1,e_2,\cdots,e_n\} \tag{2-1}$$

式中：P —— 给定的程序

C —— 给定的测试覆盖准则[7]

(2)程序元素的覆盖率，见式(2-2)：

$$\text{程序元素的覆盖率}=\frac{E(P,C)\text{被执行元素的个数}}{E(P,C)\text{元素总数}}\times 100\% \tag{2-2}$$

式中：$E(P，C)$ —— 程序元素集合

P —— 给定的程序

C —— 给定的测试覆盖准则[7]

在许多情况下，特别是程序比较大、比较复杂、或者 $E(P，C)$包含的元素比较多时，要生成对 $E(P，C)$100%的覆盖不是一件很容易的事情。一般来讲，根据一定的测试结束规则(例如，测试时间、测试开销等)，当程序元素的覆盖率达到一定的数值时，测试用例的生成即可结束。

(3)故障覆盖率，见式(2-3)：

$$故障覆盖率=\frac{执行T时所能检测的F中的故障个数}{n}\times 100\% \qquad (2\text{-}3)$$

式中：F —— $F=\{f_1, f_2, \ldots, f_n\}$是 P 中存在的任意故障集合

T —— $T=\{t_1, t_2, \ldots, t_m\}$是测试用例的集合[7]

(4)有向图：$G=(V, E)$，V 是顶点的集合，E 是有向边(本文简称边)的集合。$e=(H(e), T(e))\in E$ 是一对有序的邻接节点，$H(e)$是头，$T(e)$是尾。$H(e)$是 $T(e)$的前驱节点，$T(e)$是 $H(e)$的后继节点。如果 $H(e)=T(e')$，则 e 和 e'是临界边[8]。

(5)**DD 图**：$G=(V, E)$有两个区分的边 e_0 和 e_k(惟一进入的边和惟一离开的边)，e_0 可以到达 E 的任何一个边，E 的任何一个边都可以到达 e_k，对任何 $n\in V$，$n\neq H(e_0)$，$n\neq T(e_k)$，$indegree(n)+outdegree(n)>2$，$indegree(H(e_0))=0$，$outdegree(T(e_0))=1$，$indegree(H(e_k))=1$，$outdegree(T(e_k))=0$[9]。

引入两个区分的边只是为了分析问题方便，事实上，对于多个入口或多个出口的程序是常见的，通过增加一个虚的入口节点或出口节点总能使控制流图满足这两个约束。任何一个边 e 都是 e_0 可达且可以到达 e_k，此限制是必须的，否则，一块只有入而没有出的程序，或没有入只有出的程序是无法测试的。

(6)路径：如果 $P=e_1e_2\ldots e_q$，且满足 $H(e_i+1)=T(e_i)$，则称 P 为路径[9]。

(7)完整路径：如果 P 是一条路径，且满足 $e_1=e_0$，$e_q=e_k$，则 P 称为完整路径[8]。如果存在输入数据使得程序按照该路径完整运行，这样的路径称为可行完整路径，否则称为不可行完整路径。

程序设计要尽量避免不可行完整路径，因为这样的路径往往隐含错误。

(8)可达：如果 e_i 到 e_j 存在一个路径，则称 e_i 到 e_j 是可达的[9]。

(9)回路：路径 $P=e_1e_2\ldots e_q$ 满足 $H(e_1)=T(e_q)$，称为回路。除了第一个和最后一个节点外，其他节点都不同的回路成为简单回路[9]。

(10)主宰关系：设 $G=(V, E)$是一个 DD 图，e_0 和 e_k 分别是 G 的惟一进入的边和惟一离开的边。边 e_i 主宰边 e_j，当且仅当从 e_0 到 e_j 的任何一条路径都通过 e_i[10]。

(11)蕴涵关系：设 $G=(V, E)$是一个 DD 图，e_0 和 e_k 分别是 G 的惟一进

入的边和惟一离开的边。边 e_j 蕴涵边 e_i，当且仅当从 e_i 到 e_k 的任何一条路径都通过 e_i[10]。

(12)**程序切片**：将一个程序中用户所感兴趣的代码都抽取出来组成一个新的程序，这个新的程序称为源程序的切片[10]。

(13)**静态程序切片**：程序切片可以用 $S(V, N)$的形式表示。其中 V 表示程序中的某一个变量或是变量的集合，N 表示在程序中的某一个位置(变量 V 所在的语句)。$S(V, N)$的含义是"一个程序切片是由程序中的一些语句所组成的集合，这些语句可能会影响到在程序的某个位置 N 处所定义或引用的变量或变量的集合 V 的状态"[10]。

(14)**动态循环切片**是动态切片的一种，它可以定义为：$S(V, N, X, I)$。它表示程序在输入为 X 时，会影响到变量 V 在第 I 次执行语句 N 后的状态的所有语句的集合[10]。

2.2 基本思想

白盒测试技术的理论性比较强，用于测试证明每种内部操作和过程是否符合设计规格和要求，允许测试人员利用被测程序内部的逻辑结构和有关信息设计或选择测试用例，对程序所有逻辑路径进行测试。

白盒测试主要想对程序模块进行以下检查：

(1) 对程序模块的所有独立的执行路径至少测试一次。

(2) 对所有的逻辑判定，取 TURE 与取 FALSE 的两种情况都能至少测试一次。

(3) 在循环的边界和运行界限内执行循环体。

(4) 测试内部数据的有效性等。

白盒测试设计是以开发人员为主。

白盒测试技术在单元测试中的应用技术有逻辑覆盖法和基本路径测试法。这两种方法不考虑软件的功能实现。基于对模块内部结构的清晰了解，要求做到验证内部动作是否按照规格说明书的规定正常运行，按照程序内部的结构测试程序，检验程序中的每条通路是否都能按预定要求正确工作。

(1) 逻辑覆盖是以程序内部的逻辑结构为基础的测试用例设计方法。根

据覆盖测试的目标不同，逻辑覆盖测试可分为：语句覆盖，判定覆盖，判定-条件覆盖，条件组合覆盖及路径覆盖[11]。

(2) 基本路径测试是为了解决路径庞大难题，它是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法[12]。

白盒测试及其测试用例设计，要求测试人员对程序的逻辑结构有清楚的了解。

2.3 基本方法

白盒测试是针对软件内部结构的测试，主要是用覆盖的方式对程序代码进行测试。作者研究的基于流程树的白盒测试方法，其基本原则正是为了达到充分的路径覆盖。因为路径覆盖只有包含了语句覆盖、分支覆盖与谓词覆盖，路径覆盖的测试才能够更精确。也可以认为，语句覆盖、分支覆盖与谓词覆盖充分与否是衡量路径覆盖效率的重要准则之一。所以在本节需要介绍一下关于语句覆盖、分支覆盖与谓词覆盖的基本方法。

2.3.1 语句覆盖测试

语句测试是最简单的结构性测试方法之一。它要求在测试过程中，除了观察程序的输入、输出的正确性之外，还要观察程序中的语句是否都得到了运行，只有程序中的所有语句都得到了运行，才能称语句测试是充分的。在控制流图中，要求所有语句都能被运行的充分必要条件是覆盖图中的所有节点。因此，语句测试的充分准则可以定义为：让 T 为程序 P 的一个测试数据集，G_P 为 P 的控制流图，L_T 为与 T 相对应的 G_P 中的完整路径的集合。

定义 2.1 测试数据集 T 称为语句覆盖充分的，当且仅当 L_T 覆盖了 G_P 中的所有节点。语句覆盖测试的覆盖率定义，见式(2-4)：

$$\text{语句覆盖测试的覆盖率} = \frac{\|NODE(L_T)\|}{\|N_G\|} \times 100\% \qquad (2\text{-}4)$$

式中：$NODE(L_T)$ —— 路径集合 L_T 中所覆盖的 G_P 中的节点的集合

N_G —— G_P 中所有节点的集合[13]

2.3.2 分支覆盖测试

语句覆盖是最基本的测试技术，但就检验软件的错误而言，是远远不够的。分支覆盖要求在软件测试过程中，不仅观察软件输入、输出的正确性，还要观察软件执行过程中，是否所有的分支都得到了执行。只有当所有的分支都得到了测试，该测试才被认为是充分的。在控制流图模型中，分支表现为图中的一条有向边，因此，分支测试的充分准则定义如下。

定义 2.2 测试数据 T 称为分支覆盖充分的，当且仅当 L_T 覆盖了 G_P 中的所有有向边。语句覆盖测试的覆盖率定义，见式(2-5)：

$$\text{分支覆盖测试的覆盖率} = \frac{\|EDGE(L_T)\|}{\|E_G\|} \times 100\% \qquad (2\text{-}5)$$

式中：$EDGE(L_T)$ —— 路径集合 L_T 中所覆盖的 G_P 中的节点的集合

E_G —— G_P 中所有节点的集合[13]

定理 2.1 对任意一个程序 P 和任意一个 P 的测试数据集 T，若 T 满足分支覆盖准则，则 T 必满足语句覆盖准则，则称分支覆盖测试包含语句覆盖测试[13]。

2.3.3 谓词覆盖测试

一个分支的条件是由谓词组成的。单个谓词称为原子谓词，例如 a=0、mid>0 等都是原子谓词。而原子谓词通过逻辑计算可以构成符合谓词，常见的逻辑运算符包括“与”、“或”、“非”等。对于复合谓词而言，分之测试不是有效的。例如，对下列有符合谓词构成的语句：

```
if (math>=90|| lang>=80|| poli>=75)  x=1;
```

采用分支测试技术，只要原子谓词中的任何一个被满足，则该分之为真，而不管其他的两个是否满足。为此，提出了原子谓词覆盖准则、分子—谓词

覆盖准则和符合谓词覆盖准则。

1．原子谓词覆盖测试

定义 2.3 测试数据 T 称为原子谓词覆盖充分的(原子谓词覆盖准则)，如果对任意一个分支中的任意一个原子谓词，T 中存在一个测试数据使其在运行时为真、为假至少各一次。原子谓词的覆盖定义，见式(2-6)：

$$\text{原子谓词覆盖测试的覆盖率} = \frac{\|ATOMRUN(T)\|}{\|2 \times ATOM(P)\|} \times 100\% \qquad (2\text{-}6)$$

式中：$ATOMRUN(T)$ —— 对测试集 T 而言，运行过程中原子谓词为真和为假的个数

$ATOM(P)$ —— 程序 P 中所有原子谓词的集合[13]

这里，分母中的 2 是指真假各一次。注意：原子谓词覆盖准则和语句覆盖准则之间没有包含关系，和分支覆盖准则之间也没有包含关系，因为原子谓词强调的是原子谓词的真假，而不考虑语句或分支是否被执行。例如，下面的语句：

```
if ( x > 0 || y < 0 )    x++;
   else    x--;
```

当 x=1 且 y=1 和 x=-1 且 y=-1，满足原子谓词覆盖准则，但不满足语句覆盖准则，因为 x--这条语句没有被执行，同样也不满足分支覆盖准则。

因此，就原子谓词本身的测试而言，意义并不大。

2．分支—谓词覆盖测试

定义 2.4 测试数据集 T 称为分支—谓词覆盖充分的(分支—谓词覆盖准则)，如果对任意一个分支和所包含的任意一个原子谓词，T 中存在一个测试数据使其在运行时为真、为假至少各一次。分支—谓词的覆盖定义，见式(2-7)：

$$\text{分支－谓词覆盖测试的覆盖率} = \frac{\|BRATOMUN(T)\|}{\|2 \times BRATOM(P)\|} \times 100\% \qquad (2\text{-}7)$$

式中：$BRATOMRUN(T)$ —— 对测试集 T 而言，运行过程中分支和原子谓词为真和为假的个数

BRATOM(P) —— 程序 *P* 中所有分支及原子谓词的集合[5]

定理 2.2 对任意一个程序 *P* 和任意一个 *P* 的测试数据集 *T*，若 *T* 满足分支—谓词覆盖准则，则 *T* 必满足语句覆盖准则、分支覆盖准则和原子覆盖准则[13]。

3. 复合谓词覆盖测试

在许多情况下，如果分支的条件比较复杂，则上述几种覆盖准则是不够的，于是有提出复合谓词覆盖准则。

定义 2.5 测试数据集 *T* 称为复合谓词覆盖充分的(复合谓词覆盖准则)，如果任意一个分支，对该分支所包含的谓词的任意一个可行的真假组合，*T* 中都存在一个测试数据使该组合谓词运行时，原子谓词的取值恰好为该真假组合。复合谓词的覆盖定义，见式(2-8)：

$$\text{复合谓词覆盖测试的覆盖率} = \frac{\|COMATOMRUN(T)\|}{\|2 \times COMATOM(P)\|} \times 100\% \qquad (2\text{-}8)$$

式中：*COMATOMRUN(T)* —— 对测试集 *T* 而言，运行过程中复合谓词为真和为假的个数

COMATOM(P) —— 程序 *P* 中所有复合谓词的集合[14]

定理 2.3 对任意一个程序 *P* 和任意一个 *P* 的测试数据集 *T*，若 *T* 满足复合谓词覆盖准则，则 *T* 必满足语句覆盖准则、分支覆盖准则和原子覆盖准则[14]。

可以用图 2.1 覆盖准则关系图表示上面各种覆盖准则之间的关系，其中—>表示准则之间的包含关系。

2.4 白盒测试建模

随着计算机技术的不断发展，市场对计算机软件功能与性能的需求不断增长，软件程序代码结构也越来越复杂，因此对于大型程序代码，需要利用各种技术手段，逐步构造清晰的白盒测试模型，再进行白盒测试。本文主要研究先利用白盒测试的程序切片技术与路径分析技术，简化程序结构，再使用基于流程树的白盒测试方法进行测试。白盒测试模型是选取测试路径的基

础。构建白盒测试模型需要利用程序结构复杂度对程序切片，并基于路径覆盖的原则选取最佳测试路径。

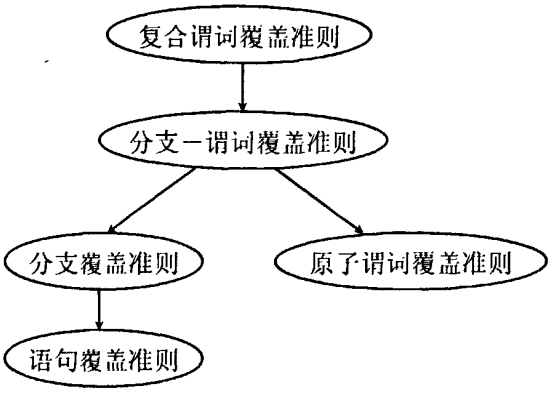

图 2.1 覆盖准则关系图

2.4.1 程序结构的复杂度

作者认为，对于由多个模块构成的程序代码段，应该按照一定原则排列模块测试顺序。对于模块内较复杂的程序代码段，应该按照一定原则先切分程序段为程序切片，然后再进行基于流程树的白盒测试，这样能够提高整个白盒测试过程的测试效率。而排列模块测试顺序与切分程序段的依据正是模块重要度与程序结构复杂度的概念。因此，在本节需要介绍一下模块重要度与程序复杂度的相关知识。

1．模块间的重要度

模块间的结构复杂性：模块间的结构复杂性是指模块之间的交联复杂性，可以用模块的扇入、扇出数量来表示。首先，可将程序以模块为结点表示为有向图，结点之间的关系表示为模块之间的调用关系，将结构复杂性定义为起点为顶结点到各个结点之间的路径数目之和再加 1。

根据模块间的有向图，可以定义模块的重要度。模块的重要度定义为

$$im(X)=\frac{C(X)}{\sum_{X}C(X)} \tag{2-9}$$

式中：$C(X)$ —— 从顶结点开始到达和经过 X 的路径数目[15]

2. 模块内部的复杂度

模块内的 McCabe 度量方法：McCabe 认为，程序的复杂性是由程序对应的控制流图中的回路个数决定的，回路个数越多，程序就越复杂。反之亦然。而程序中的回路基数可以根据下面的定理给出。

定理 2.4 强连通图 G 的回路基数 $V(G)$等于图中线形独立的回路数，它可由式(2-10)计算：

$$V(G)=e-n+1 \tag{2-10}$$

式中：e —— G 中的边数

N —— G 中的结点数[15]

实践表明，对于 $V(G)\geq 10$ 的控制图所对应的程序，一般就比较复杂了。对于 $V(G)\geq 10$ 的程序代码段，需要根据程序变量关系复杂度对程序代码段进行切片。

以下介绍了变量依赖关系的基本概念。

一个模块 P 的所有代码行的集合为 $G(P)$，所有变量的集合为 V，则有以下两个定义。

定义 2.6 结点 $n\in G(P)$是变量 $v\in V$ 的定义结点，记做 $DEF(v, n)$，当且仅当变量 v 的值由对应结点 n 的语句行定义[16]。

输入语句、赋值语句、循环控制语句和过程调用，都是定义结点语句的例子。如果执行对应这种语句的结点，那么与该变量关联的存储单元的内容就会改变。

定义 2.7 结点 $n\in G(P)$是变量 $v\in V$ 的使用结点，记做 $USE(v, n)$，当且仅当变量 v 的值由对应结点 n 的语句行处使用[16]。

输出语句、赋值语句、条件语句、循环控制语句和过程调用，都是使用结点语句的例子。如果执行对应这种语句的结点，那么与该变量关联的存储

单元的内容就保持不变。

基于以上的 *DEF*(*v*，*n*)和 *USE*(*v*，*n*)的定义，可以给出变量依赖的定义如下。

定义 2.8 同一语句行的变量中，有且仅有一个 *DEF*(*v*,*n*)，和多个 *USE*(*w*，*n*)，*USE*(*x*，*n*)...称作 *v* 变量依赖 *w*，*x* 等变量，*w*，*x* 等变量叫做原因变量，*v* 叫做结果变量[16]。

定义 2.9 模块 *P* 中的变量 $v \in V$，如果在该模块的第 *n* 行定义，则记做 v_n，称为时空绝对变量[16]。

定义 2.10 变量依赖关系图是一个有向图，用来描述模块 *P* 中所有时空绝对变量的拓扑关系。

作为原因变量必然先被定义，如果第 *r* 行中，*v* 是结果变量，*w*，*x* 是原因变量，则由最近定义的对应时空变量 w_n，x_m 分别画箭头指向当前的 v_r[16]。

以下介绍了变量度量定义和计算公式。

定义 2.11 缠结数(*Tn*)，见式(2-11)：

$$T_n = C_n \tag{2-11}$$

式中：*Cn* —— 依赖图中，时空绝对变量的非自身原因变量数[17]

定义 2.12 缠结距离(*Td*)，见式(2-12)：

$$Td_i = AL - CL_i \tag{2-12}$$

式中：*AL* —— 时空绝对变量所在行

CL_i —— 第 *i* 个非自身原因变量所在行[17]

定义 2.13 聚合缺乏度(*Tm*)，见式(2-13)：

$$Tm = \sum |Td_i| \quad (1 \le i \le Cn) \tag{2-13}$$

式中：*Td* —— 缠结距离

Cn —— 依赖图中，时空绝对变量的非自身原因变量数[17]

定义 2.14 变量自身距离(*Ad*)，见式(2-14)：

$$Ad = AL - VdL \tag{2-14}$$

式中：*AL* —— 时空绝对变量所在行

VdL —— 最近定义行[17]

定义 2.15 变量关系复杂度(*Vm*)，见式(2-15)：

$$Vm = \alpha \times Tm + \beta \times |Ad| \tag{2-15}$$

式中：*Tm* —— 聚合缺乏度

Ad —— 变量自身距离[17]

有了以上定义，现在给出变量度量复杂度的计算公式。所以，一个模块的变量度量的复杂度，见式(2-16)：

$$MVm = \sum Vm_i \tag{2-16}$$

式中：Vm_i —— *Dset* 中第 *i* 个元素的变量关系复杂度[18]

在不考虑其他因素的情况下，软件越复杂，其缺陷密度一般也会越高，本文研究复杂性的目的是通过这些理论设计一种程序切片技术，有利于简化软件测试。

2.4.2 程序切片建模

程序切片建模就是将一个程序中用户所感兴趣的代码都抽取出来组成一个新的程序。这个新的程序称为源程序的切片。根据切片规则的不同，生成的切片也各不相同。程序切片技术是一种分析和理解软件程序的技术。

程序切片技术是由Weiser在1979年首先提出的，他描述了一使用程序依赖图来实现过程内切片的技术。此后．在他的基础上又有许多人提出了不同的程序切片的定义和用于切片的算法．其中包括Horwitz提出的程序切片算

法，他将程序依赖图(PDG)扩展为系统依赖图(SDG)，解决了Weiser提出的算法中无法解决过程调用的问题，以及由Korel和Laski提出的动态切片的概念和Canfora提出的条件切片技术等等。 程序切片技术的发展经历了从静态到动态，从前向到后向，从单一过程到多个过程，从面向过程的程序到面向对象的程序。从非分布式到分布式的程序的发展过程。随着软件开发技术的不断进步，程序切片技术也会随之一同发展，并且广泛应用于代码调试和软件测试的过程中。

1．静态程序切片建模

程序切片可以用$S(V, N)$的形式表示。其中V表示程序中的某一个变量或是变量的集合，N表示在程序中的某一个位置(变量V所在的语句)。$S(V, N)$的含义是“一个程序切片是由程序中的一些语句所组成的集合，这些语句可能会影响到在程序的某个位置N处所定义或引用的变量或变量的集合V的状态”。$S(V, N)$是程序切片最基本的形态。任何形式的程序切片都可以通过对这个标准进行扩展而得到[19]。

静态程序切片是指在构造程序切片的时候使用静态的数据流和控制流的分析方法。从静态程序切片的定义$S(V, N)$。可以看出使用这种切片标准分析程序的时候，变量V的当前状态是无关紧要的，其值不会影响切片的结果。静态切片所作的分析完全是依赖于程序的静态信息。

静态切片方法需要对变量V所有的状态进行考虑。需要遍历程序中相应的所有轨迹，所以使用这种方法分析程序时工作量会非常的大。因此，由于静态切片技术的这些局限性，它主要应用于程序理解和软件维护领域。

2．动态程序切片建模

动态程序切片技术使用的是动态的数据流和控制流分析方法。它依赖于程序中某个变量的具体输人，其输人的不同可能会导致切片结果的不同。

静态切片技术强调的是在可以遍历到的所有轨迹中，对程序中某一点的变量状态造成影响的所有语句；而动态切片技术则是强调程序在一次特定的执行中，会影响变量在程序中某一点的状态的所有语句。可以看出，动态程序切片是相应的静态程序切片的一个子集，因此使用动态程序切片时的工作量要比使用静态程序切片的工作量来的小。更适合使用于程序测试与调试等本身工作量比较大的工作中。

动态循环切片是动态切片的一种，它可以定义为：$S(V, N, X, I)$。它表示程序在输入为X时，会影响到变量V在第I次执行语句N后的状态的所有语句的集合[20]。

3．条件切片建模

条件切片技术是介于静态切片技术与动态切片技术之间的一种切片技术，它既不是仅仅局限与只对程序的静态信息进行分析，也不是仅仅局限于只依赖外部的输入来获得程序的信息。在构造条件切片时，只有那些满足切片条件的语句才会被提取出来。

条件切片的定义可以表示为$S(V, N, X, W)$，它表示当输入X使得条件W为真的时候，所有影响变量V在N处状态的语句的集合。如果将程序中从满足一个切片条件的任何一个初始状态出发都不可能触发的语句除去，那么剩下的语句就是满足这个条件的一个切片。

2.4.3 测试路径建模

测试路径建模是基于流程树的白盒测试过程中关键的环节，因为路径中的程序代码能否得到充分覆盖，决定了测试是否达到了效率要求。路径建模主要是依靠控制流图、DD 图与计算机数据结构学科中“树”的概念，将这些概念结合与扩充，就形成的本文中“流程树”的模型。

1．控制流图

对于用结构化程序语言书写的程序，可以通过使用一系列规则从程序推导出其对应的控制流图。因此，控制流图和程序是一一对应的，但控制流图更容易使人们理解程序。

定义 2.16 有向图：$G=(V, E)$，V 是顶点的集合，E 是有向边(本文简称边)的集合。$e=(H(e), T(e))\in E$ 是一对有序的邻接节点，$H(e)$是头，$T(e)$是尾。$H(e)$是 $T(e)$的前驱节点，$T(e)$是 $H(e)$的后继节点。如果 $H(e)=T(e')$，则 e 和 e' 是临界边[21]。

定义 2.17 路径：如果 $P=e_1e_2...e_q$，且满足 $H(e_i+1)=T(e_i)$，则称 P 为路径[22]。

定义 2.18 完整路径：如果 P 是一条路径，且满足 $e_1= e_0$，$e_q= e_k$，则 P

称为完整路径。如果存在输入数据使得程序按照该路径完整运行，这样的路径称为可行完整路径，否则称为不可行完整路径[23]。

程序设计要尽量避免不可行完整路径，因为这样的路径往往隐含错误。

定义 2.19 可达：如果 e_i 到 e_j 存在一个路径，则称 e_i 到 e_j 是可达的[24]。

定义 2.20 简单路径：路径上所有的节点都是不同的成为简单路径[25]。

定义 2.21 基本路径：任意有向边都在路径中最多出现一次的称为基本路径[26]。

定义 2.22 子路径：路径 $A=e_ue_{u+1}...e_t$ 是 $B=e_1e_2...e_q$ 的子路径，如果满足 $1\le u\le t\le q$。一条路径被称为是不可行的，如果不存在可行的完整路径使其成为该完整路径的子路径[27]。

定义 2.23 回路：路径 $P=e_1e_2...e_q$ 满足 $H(e_1)=T(e_q)$，称为回路。除了第一个和最后一个节点外，其他节点都不同的回路成为简单回路[28]。

定义 2.24 无回路路径：一条路径中不包含有回路子路径的称为无回路路径[29]。

定义 2.25 A 连接 B：若 $A=e_ue_{u+1}...e_t$，$B=e_ve_{v+1}...e_q$ 为两条路径，$H(e)$是边 e 的头节点，$T(e)$是边 e 的尾节点，如果 $T(e_t)=H(e_v)$且 $e_ue_{u+1}...e_te_ve_{v+1}...e_q$ 为路径，则称 A 连接 B，记为 $A*B$[30]。

当一条路径是回路时，它可以和自己连接，记 $A^1=A$，$A^{K+1}=A*A^K$。

定义 2.26 路径 *A* 覆盖路径 *B*：如果路径 B 中所含的有向边均在路径 A 中出现，则称路径 A 覆盖路径 B[31]。

2. DD 图

程序和控制流图是一一对应的，经过适当的变换，控制流图可以一一对应地转化为 DD 图，因此，程序和 DD 图也是一一对应的。

定义 2.27 DD 图：$G=(V, E)$有两个区分的边 e_0 和 e_k(惟一进入的边和惟一离开的边)，e_0 可以到达 E 的任何一个边，E 的任何一个边都可以到达 e_k，对任何 $n\in V, n\neq H(e_0), n\neq T(e_k)$, $indegree(n)+outdegree(n)>2$, $indegree(H(e_0))=0$, $outdegree(T(e_0))=1$，$indegree(H(e_k))=1$，$outdegree(T(e_k))=0$[32]。

引如两个区分的边只是为了分析问题方便，事实上，对于多个入口或多个出口的程序是常见的，通过增加一个虚的入口节点或出口节点总能使控制流图满足这两个约束。任何一个边 e 都是 e_0 可达且可以到达 e_k，此限制是必

须的，否则，一块只有入而没有出的程序，或没有入只有出的程序是无法测试的。

定义 2.28 主宰关系：设 $G=(V, E)$是一个 DD 图，e_0和 e_k分别是 G 的惟一进入的边和惟一离开的边。边 e_i主宰边 e_j，当且仅当从 e_0到 e_j的任何一条路径都通过 e_i[33]。

通过应用主宰关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_0，称为主宰树 $DT(G)$。树 $T=(V, E)$是一个图，拥有一个根节点，该节点的入度为 0。除根节点外，T 中的其他节点的入度都为 1，并且存在从根节点到其他节点惟一一条路径，T 中出度为 0 的节点称为叶子节点。$e=(m, n)$是 T 的一条边，则 n 是 m 的父节点，m 是 n 的子节点。

定义 2.29 直接主宰：对任意 DT 的一对邻接节点(e_i, e_j)，$e_i=Parent(e_j)$称为 e_j的直接主宰。显然，若 e_i是 e_j的直接主宰，则 e_i也是 e_j的主宰，也表明，任何节点如果主宰 e_i，则必然主宰 e_j。除了根节点外，任何节点都有惟一一个直接主宰[34]。

定义 2.30 主宰路径：在 $DT(G)$中的一个主宰路径 $P_{DT}=e_1e_2...e_q$，$e_i=Parent(e_i+1)$，$1\le i\le q$。主宰路径一般不是 G 中的路径[34]。

定义 2.31 蕴涵关系：设 $G=(V, E)$是一个 DD 图，e_0和 e_k分别是 G 的惟一进入的边和惟一离开的边。边 e_i蕴涵边 e_j，当且仅当从 e_j到 e_k的任何一条路径都通过 e_i[35]。

通过应用蕴涵关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_k，称为蕴涵树 $IT(G)$。

定义 2.32 直接蕴涵：对任意 IT 的一对邻接节点(e_i, e_j)，$e_j=Parent(e_i)$称为 e_i的直接蕴涵。显然，若 e_j是 e_i的直接蕴涵，则 e_j也是 e_i的蕴涵，也表明，任何节点如果蕴涵 e_j，则必然蕴涵 e_i。除了根节点外，任何节点都有惟一一个直接蕴涵[36]。

定义 2.33 蕴涵路径：在 $IT(G)$中的一个蕴涵路径 $P_{IT}=e_1e_2...e_q$，$e_i=Parent(e_i+1)$，$1\le i\le q$。蕴涵路径一般不是 G 中的路径[36]。

定义 2.34 子路径：设 $P=e_1e_2...e_q$是 G 的一个路径，$P'=e_{i1}e_{i2}...e_{ir}$，如果满足$\{i_1, i_2, \ldots, i_r\}\subset\{1, 2, \ldots, q\}$并且 $i_j<i_{j+1}(j=1, 2, \ldots, r-1)$，则称 P'是 P 的子路径[36]。

3．树

定义2.35 树：是$n(n>=0)$个结点的有限集。在任意一棵非空树中：①有且仅有一个特定的称为根的结点；②当$n>1$时，其余结点可分为$m(m>0)$个互不相交的有限集T_1，T_2，…T_m，其中每一个集合本身又是一棵树，并且称为根的子树[37]。

树的结点包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为结点的度。度为0的结点称为叶子或终端结点。度不为0的结点称为非终端结点或分支结点[37]。

定义2.36 二叉树：二叉树是另一种树型结构，它的特点是每个结点至多只有二棵子树(即二叉树中不存在度大于2的结点)，并且，二叉树的子树有左右之分，其次序不能任意颠倒[37]。

一棵深度为k且有$2(k)$-1个结点的二叉树称为满二叉树，为每个结点编号，如果有深度为k的，有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称之为完全二叉树[37]。

2.5 本章小结

本章介绍白盒测试相关知识，先阐述了白盒测试设计实现中要涉及的基本术语，接着根据基于流程树的白盒测试建模需要，引入程序复杂度、程序切片概念的目的是根据切片规则化简程序段，引入控制流图、DD 图、树的相关理论，目的是为构造流程树测试路径提供基本理论依据，为后面各章节的展开打下了基础。

第 3 章 流程树的相关技术研究

本文提出的基于流程树的白盒测试方法，不仅用于软件单元测试阶段，而且也可用于软件集成测试阶段。本章第 3.1 节是关于模块复杂度的研究，因为在集成测试阶段，首先要分析模块复杂度，再根据分析结果规定程序模块的测试顺序，然后对选定模块进行基于流程树的白盒测试。本章 3.2 节是对程序切片技术的研究，因为在单元测试阶段，如果一个模块内部的程序段非常复杂，应该先考虑对程序段进行切片，再对具有重要测试意义的程序片段进行基于流程树的白盒测试。本章第 3.3 节是对基于 DD 图的路径构造的研究，因为充分覆盖 DD 图无约束边是基于流程树的白盒测试重要原则之一。在本章第 3.4 节，作者提出了一套从程序流图到流程树的转换规则，是构造基于流程树的白盒测试模型的关键步骤。

3.1 模块复杂性的度量

自顶向下的集成测试方法，桩模块个数比较多，低层模块的测试被延迟，并且如果低层模块的数量较多，可能导致测试不够充分；自底向上的集成测试方法是从程序的低层模块开始测试，由于高层的测试被推迟到最后，因此，设计上的错误不能及时发现。

作者认为，软件度量应该成为指导集成测试与单元测试过程的重要内容之一。软件度量方法可以测量代码模块内部或外部的逻辑关系或是依赖关系，对解剖软件本质有着指导意义。

在集成测试中结合模块重要度的分析，决定模块测试的先后顺序，能够及时测试到对全局影响较大的模块，能够使测试尽量充分，可以改进传统的集成测试方法中存在的不足。

首先，可将程序以模块为结点表示为有向图，结点之间的关系表示为模块之间的调用关系。然后根据模块间的有向图，用公式计算各个模块的重要度。再结合模块重要度的分析，决定模块测试的先后顺序，对模块逐级进行

测试。

3.2 程序切片技术

软件测试是保证软件质量的一个重要环节，在现代软件工程中，软件测试在整个软件生命周期中所处的地位越来越高。随着软件规模和复杂性的不断提高，采用传统的软件测试技术发现和定位软件中的错误也越来越难。基于程序切片的软件测试按照一定的切片准则对软件进行分解和剪裁，从而简化程序，同时，使得程序切片代码仍能反映源程序的部分特征。使得开发人员能够将注意力集中到相关代码上，达到快速错误定位的目的。

程序切片的概念是由Mark weiser于1979年首次提出的。一个程序切片是由程序P中那些可能影响在某个感兴趣的程序点n计算的变量V的值的部分组成的，其中(n, V)称为切片标准，对程序P进行切片称为程序切片，切片后得到的剩余程序称为程序片。

一种重要的切片方法是基于程序依赖图PDG(Porgram Dendency Graph)的切片方法。PDG是CFC的一个变形，是CFG中去掉了控制流边，增加了控制依赖和数据依赖以后生成的图。PDG是一个有向图，其中的节点表示程序语句，边用来表示节点间的控制依赖和数据依赖关系，更直接地表达了控制流和到达定义的效果：如果一个语句控制着另一个语句的执行，则两个语句间存在着控制依赖关系；如果一个语句所定义的变量能够到达另一个语句对同一变量的使用，则两语句间存在着数据依赖关系。

3.2.1 代码依赖关系

为了便于说明代码依赖关系，设定一个模块的所有代码行的集合为 G，所有变量的集合为 V。

1．控制依赖

如果节点 a，b 满足下面条件，则称 b 控制依赖于 a，记作 $b \xrightarrow{cd} a$：①G 上存在一条由节点 a 到 b 的路径 p，对于路径上的其他节点 $c \in p - \{a,b\}$，b 是 c 后的必经节点；②b 不是 a 后的必经节点；③如果 a 是 $CDG=(N, E, s,$

e，v)唯一的开始节点s，则a后的所有必经节点都控制依赖于a[38]。

由此可见，处于分支上的节点都控制依赖于判断节点，而顺序语句则不存在这种控制依赖关系。控制依赖关系关注的是*if*，*case*，*while*等含有分支路径的语句。

2. 数据依赖

要研究程序代码的数据依赖关系，首先就要理解对于“变量定义”与“变量使用”的概念与区别。

当且仅当变量v的值由对应结点a的语句行定义，结点$a\in G$是变量$v\in V$的定义结点，记做$v\in def(a)$。输入语句、赋值语句、循环控制语句和过程调用，都是定义结点语句的例子。如果执行对应这种语句的结点，那么与该变量关联的存储单元的内容就会改变。

当且仅当变量v的值由对应结点b的语句行使用，结点$b\in G$是变量$v\in V$的使用结点，记做$v\in ref(b)$。输出语句、赋值语句、条件语句、循环控制语句和过程调用，都是使用结点语句的例子。如果执行对应这种语句的结点，那么与该变量关联的存储单元的内容就保持不变。

基于以上的$def(a)$和$ref(b)$的说明，可以给出数据依赖的理解。

如果节点a，b满足下面条件，则称b数据依赖于a，记作$b\xrightarrow{dd}a$：①如果存在一个变量v满足$v\in def(a)\cap ref(b)$；②G上存在一条由节点a到节点b的路径p，对于路径上的其他节点$c\in p-\{a, b\}$，v不$\in def(c)$[38]。

由此可见，满足数据依赖的充分条件是，对于某一变量v，在程序的第a行处定义，在程序的第b行处使用，但是在处于第a行和第b行之间的第c行不可以再对变量v有第2次定义，否则就不能称之为a数据依赖于b，而要称之a数据依赖于c了。

3. 传递依赖

如果节点a，b满足下面条件，则称b传递依赖于a，记作$b\xrightarrow{*}a$：①存在一条路径$p=<n_1, \ldots, n_k>$，其中$n_1=a$，$n_k=b$，并且每个n_{i+1}都控制依赖或数据依赖于n_i，$1<i<k$；②p是控制依赖图中的一个可实现路径[38]。

3.3 基于 DD 图的路径构造

目前关于 DD 图的理论研究只限于提炼无约束边，只说明了覆盖无约束边的路径集合就能覆盖整个路径，没有阐明如何利用覆盖无约束边的路径集合去覆盖整个路径的问题，没有解决精简覆盖无约束边的路径集合，使其数目能够达到充分测试整个路径的最小值的问题。作者对 DD 图无约束边在路径构造方面的使用方法进行了扩充，解决了精简覆盖无约束边的路径集合，使其数目能够达到充分测试整个路径的最小值的问题。在本节有简单叙述，在第 4 章有详细实现过程。

3.3.1 主宰与蕴涵关系

提炼无约束边，首先要根据 DD 图边与边之间的主宰与蕴涵关系，建立主宰树与蕴涵树，然后研究他们的叶子节点。

1．主宰树

设 $G=(V, E)$是一个 DD 图，e_0和 e_k分别是 G 的惟一进入的边和惟一离开的边。边 e_i主宰边 e_j，当且仅当从 e_0到 e_j的任何一条路径都通过 e_i。通过应用主宰关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_0，称为主宰树 $DT(G)$。树 $T=(V, E)$是一个图，拥有一个根节点，该节点的入度为 0。除根节点外，T 中的其他节点的入度都为 1，并且存在从根节点到其他节点惟一一条路径，T 中出度为 0 的节点称为叶子节点。$e=(m, n)$是 T 的一条边，则 n 是 m 的父节点，m 是 n 的子节点。在 $DT(G)$中的一个主宰路径 $P_{DT}=e_1e_2...e_q$，$e_i=Parent(e_i+1)$，$1\le i\le q$。主宰路径一般不是 G 中的路径[39]。

2．蕴涵树

设 $G=(V, E)$是一个 DD 图，e_0和 e_k分别是 G 的惟一进入的边和惟一离开的边。边 e_i蕴涵边 e_j，当且仅当从 e_j到 e_k的任何一条路径都通过 e_i。通过应用蕴涵关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_k，称为蕴涵树 $IT(G)$。在 $IT(G)$中的一个蕴涵路径 $P_{IT}=e_1e_2...e_q$，$e_i=Parent(e_i+1)$，$1\le i\le q$。蕴涵路径一般不是 G 中的路径[39]。

3．DD 图的路径

G 是一个 DD 图，$DT(G)$和 $IT(G)$是 G 对应的主宰树和蕴涵树。对 G 中的

任意两个边 e_i 和 e_j，如果 e_i 和 e_j 在 $DT(G)$和 $IT(G)$中分别是直接主宰关系或直接蕴涵关系，虽然在 G 中不是邻接的，但是对于 DT 或 IT 树中存在的路径，出于该路径是 G 中的路径，但可以通过增加边来弥补其中的不连续，是其成为 P 中的路径。

如果 $DT(G)$中存在一个路径，$P_{DT}=e_1e_2...e_qe$，则在 G 中存在一个从 e_0 到 e 的路径，并具有下列形式 $P=e_0\ P_0e_1P_1...e_q\ P_qe_{q+1}(=e)$，$P_j$(可能是空的)是从 $H(e_j)$到 $T(e_{j+1})$的路径[39]。

如果 $IT(G)$中存在一个路径，$P_{IT}=ee_2...e_k$，则在 G 中存在一个从 e 到 e_k 的路径，并具有下列形式 $P=(e_1=)eP_1e_2P_2...e_q\ P_qe_k$，$P_j$(可能是空的)是从 $H(e_j)$到 $T(e_{j+1})$的路径[39]。

3.3.2 DD 图的路径覆盖

$G=(V, E)$是一个DD图，$\varepsilon=\{P_1, P_2, ..., P_n\}$一个路径集合，如果对任意$e\in E$，存在$P_i\in P$，其中$P_i$包含$e$，那么路径集合$\varepsilon$是$G$的路径覆盖。而如果对任意一个$G$的覆盖$\varepsilon'$都满足$\|\varepsilon'\|>\|\varepsilon\|$，则路径集合$\varepsilon$是$G$的最少路径覆盖。

对于DD图路径覆盖的研究，可以引入无约束边的概念与一个重要的定理。

定义3.1 无约束边：如果一个边e_u不主宰其他的节点也不被其他的节点所蕴涵，则e_u称为无约束边[39]。

作者分析，在主宰树$DT(G)$中不主宰其他节点的边是$DT(G)$中的叶子节点，而在蕴涵树$IT(G)$中不蕴涵其他节点的边是$IT(G)$中的叶子节点。根据这个定义，DD图无约束边的集合为：

$UEdge(G)=DTLeaves(G)\cap ITLeaves(G)$

其中，$UEdge(G)$的无约束边的集合，$DTLeaves(G)$是主宰树$DT(G)$中叶子节点的集合，$ITLeaves(G)$是蕴涵树$IT(G)$中叶子节点的集合。

定理 3.1 $G=(V, E)$是一个 DD 图，e_0和 e_k分别是 G 中的惟一进入的边和惟一离开的边，有①覆盖所有无约束边的路径集合也覆盖了所有的路径。②在所有满足①的边的集合中，无约束边是最少的[39]。

作者认为，可以在选取覆盖路径时，要使每一条从 e_0 到 e_k 的路径尽可能

充分地覆盖 *UEdge*(*G*)集合中的所有节点，由此可以构造出 G 的最少路径覆盖集合 ε。实现方法在第 4 章有详细介绍。

3.4 程序流程图向流程树的转换规则

白盒测试是以程序结构为基础的一类测试方法，目的在于使测试充分地覆盖软件的结构，并以软件结构中的某些元素是否都已得到测试为准则来判断测试的充分性。充分性是以被测元素占总元素的百分比来衡量的。从直观上看，对某个确定的元素，测试的越充分，其发现错误能力就应该越强。的确，测试的充分度是和错误的发现率相关的。在解决实际问题时，如果遇到一个不太复杂的程序，使用传统的白盒路径测试方法可以使程序得到充分的测试，但是如果遇到一个庞大而又复杂的程序时，无论是画程序流图，还是设计测试用例，都需要花费大量的时间与精力，此时追求测试的充分性而忽视高效率的传统白盒路径测试方法就显得有些束手无策了。充分性并不等价于高效率，只能是高效率要求的一部分。但是随着软件开发效率的提高，产品投入市场的周期缩短，必然对软件测试的效率提出更高的要求。

在传统的白盒路径测试方法中，只是将程序段转换到程序流图为止，这就需要用到深度遍历和广度遍历等复杂的算法，实现起来比较浪费时间。所以本文提出了一种基于流程树的白盒测试方法。先根据程序片段得到一个程序流图，再用一系列的规则把程序流图转换成相应的二叉树的形式，当遍历二叉树叶子结点到二叉树根结点的一条路径时，即生成与该路径相对应的一组测试用例。这种方法的优点在于，二叉树形式的图不但表达更加清晰，而且在分析和算法方面实现起来也比较容易。

从程序流程图向流程树转换，需要经历 2 个阶段。第 1 个阶段是从程序流程图向程序流图转换。第 2 个阶段是从程序流图向流程树转换。

3.4.1 程序流程图向程序流图的转换规则

程序流程图向程序流图的转换规则如下。

规则 1. 如果是顺序语句，则实行如图 3.1 顺序语句转换图所示的转换规

则。

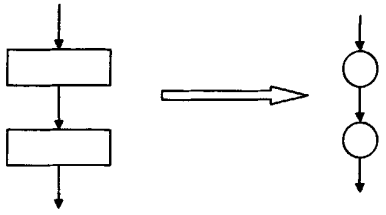

图 3.1 顺序语句转换图

规则 2. 如果是 if 语句，则当逻辑为真或假时需执行的语句分别作为两个不同的结点，如图 3.2 条件语句转换图所示。

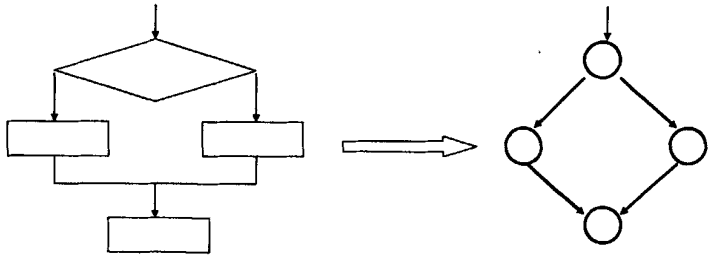

图 3.2 条件语句转换图

规则 3. 如果是 case 语句，则假设每个判定条件为真时的执行语句对应一个结点，当 case 条件全部为假时，执行的 default 语句作为一个结点，如图 3.3case 语句转换图所示。

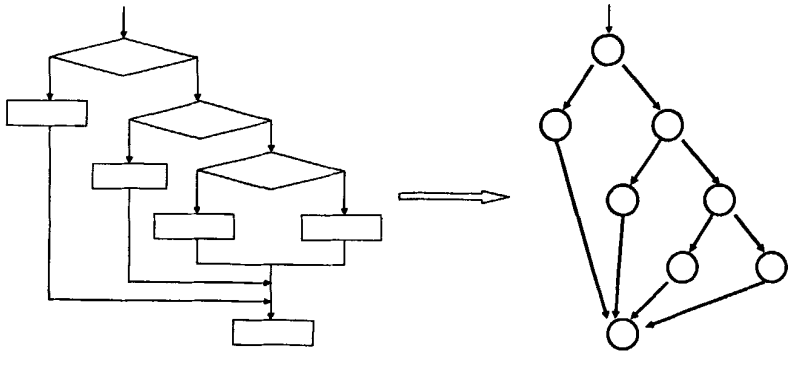

图 3.3 case 语句转换图

规则 4. 如果是 while 语句，则考虑其循环执行情况画出其它结点，如图 3.4while 型循环语句转换图所示。for 语句的情况与 while 语句类似。

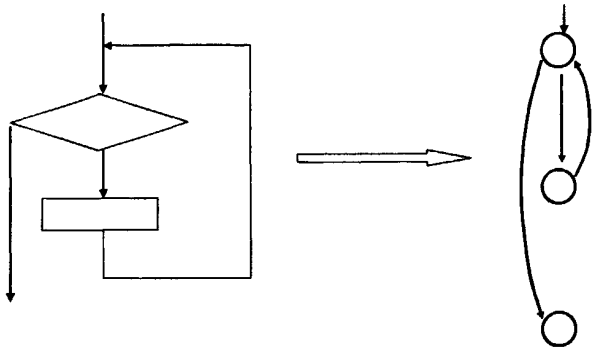

图 3.4 while 型循环语句转换图

规则 5. 如果是 do-until 语句，则考虑其循环执行情况画出其它结点，如图 3.5do-until 型循环语句转换图所示。

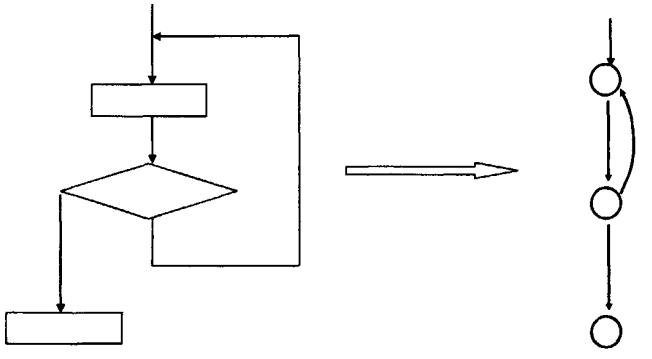

图 3.5 do-until 型循环语句转换图

3.4.2 程序流图向流程树的转换规则

目前关于程序结构图的研究仍然限于含回路的程序流图阶段，对于回路较多的程序流图，不利于便捷地选取测试路径。因此，作者对程序流图的研

究进行了扩展，提出了由程序流图向流程树的转换规则，流程树能够使测试路径更清晰地呈现。

由程序流图向流程树的转换规则如下。

规则 1. 如果是顺序执行的两条或者两条以上语句的结点，则合并为一个结点。

规则 2. 如果程序流图中某一结点的出度 N≥2 时，则将该结点复制为 N 个结点，作为每个前趋结点的孩子结点。表示如图 3.6 原子谓词分支语句转换图所示。

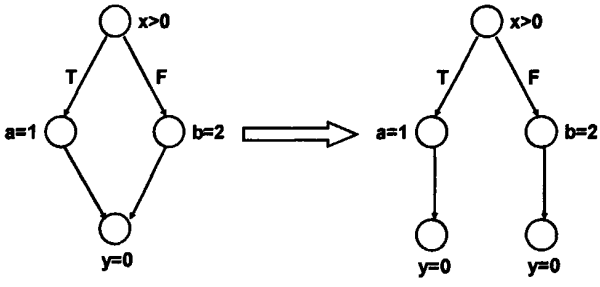

图 3.6 原子谓词分支语句转换图

一个分支的条件是由谓词组成的。单个谓词称为原子谓词，例如原子谓词 a！=0、mid＞0 等都是原子谓词。而原子谓词通过逻辑运算符可以构成复合谓词。当条件语句中用到一个或多个逻辑运算符(“与”、“或”、“非”等)时，就出现了复合条件。此时，要将每一个谓词拆分为一个结点。

规则 3. 如果是带逻辑运算符“与”的复合条件语句的节点，那么第 n+1 个原子谓词节点放在第 n(n≥1)个原子谓词节点为真的分支上。例如，当出现语句 if(x＜0 && y＜0)时，将其拆分为两个判断结点，第二个原子谓词节点在第一个原子谓词节点为真的分支上。如图 3.7 “与”谓词分支语句转换图所示。

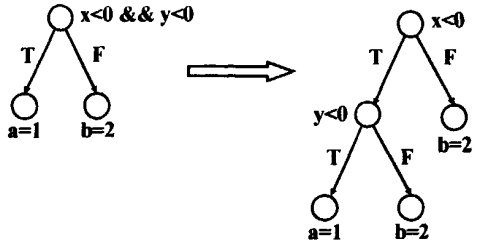

图 3.7 “与”谓词分支语句转换图

规则 4. 如果是带逻辑运算符“或”的复合条件语句的节点，那么第 n+1 个原子谓词节点放在第 n(n≥1)个原子谓词节点为假的分支上。例如，当出现语句 if(x＜0 || y＜0)时，将其拆分为两个判断结点，第二个原子谓词节点在第一个原子谓词节点为假的分支上。如图 3.8 “或” 谓词分支语句转换图所示。

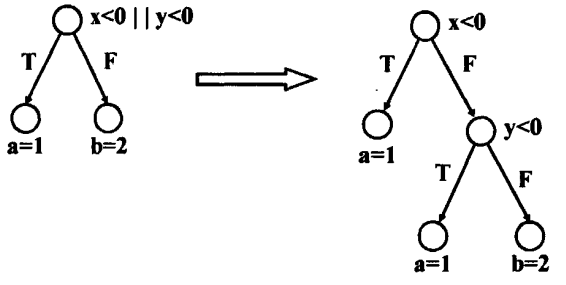

图 3.8 “或”谓词分支语句转换图

规则 5. 当出现 case 条件判定语句的时候，需要判定多个条件，每一个 case 条件相当于一个仅含有原子谓词的 if 条件判定语句。如果 case 语句包含的分支(default 也算其中一个)不少于 2，那么将 case 语句外部的下一个结点复制为 N 个，作为每个 case 分支结点的孩子结点。表示如图 3.9case 分支语句转换图所示。

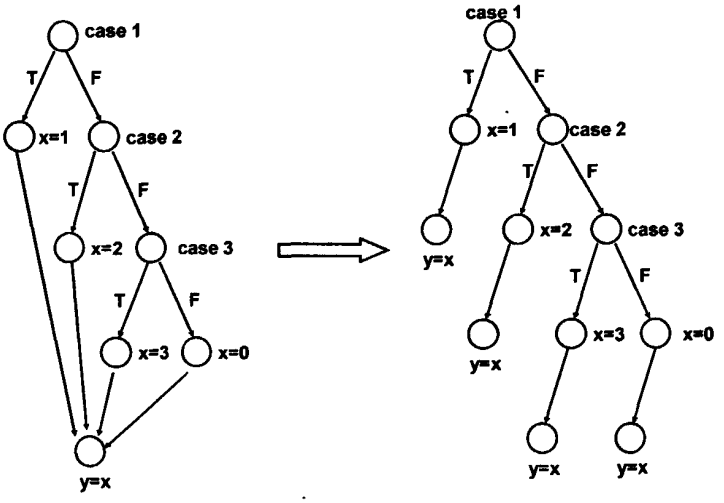

图 3.9 case 分支语句转换图

规则 6. 当出现 while 循环语句的时候，简化的路径覆盖测试方法仅考虑循环执行零次与循环执行一次的情况。此时循环结点被拆分为两条分支路径，一条分支路径是不满足 while 循环条件须执行循环体外部代码的情况，另一条分支路径是满足 while 循环条件须执行循环体内部代码的情况。表示如图 3.10while 循环语句转换图所示。

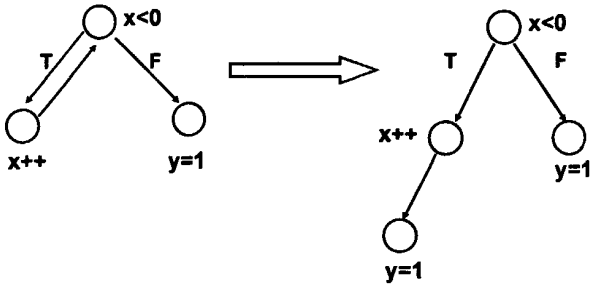

图 3.10 while 循环语句转换图

规则 7. 当出现 for 循环语句的的时候，情况与 while 语句类似。表示如图 3.11for 循环语句转换图所示。

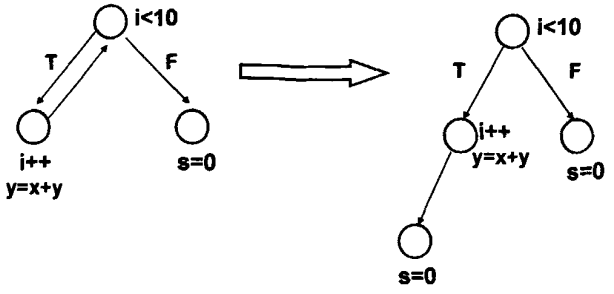

图 3.11 for 循环语句转换图

规则 8. 当出现 do-until 循环语句的时候，简化的路径覆盖测试方法仅考虑循环执行一次的情况。因为执行 do-until 循环语句时，循环体中的内容至少要被执行一次，所以执行一次便可以实现路径覆盖。表示如图 3.12do-until 循环语句转换图所示。

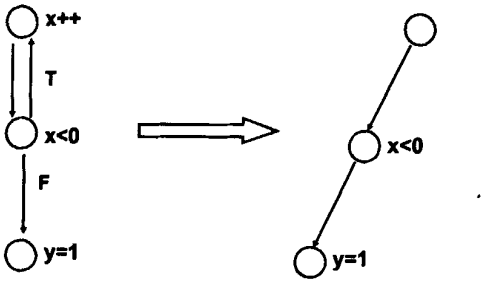

图 3.12 do-until 循环语句转换图

通过以上 8 个规则，可以实现任何程序流图向流程树的转换。研究程序流程图转换为流程树的意义在于，当遍历流程树叶子结点到流程树根结点的一条路径时，即生成与该路径相对应的一组测试用例。可以使程序流程更加清晰，也使测试用例选取更加便捷。

3.5 本章小结

作者在程序流程图与程序流图的基础上，提出了继续由程序流图向流程树转换的思想与规则。流程树形式的图不存在回路，每条从根节点到叶子节

点的路径都是一条完整的测试路径，不但表达更加清晰，而且算法方面实现起来也比较容易。构造流程树是基于流程树的白盒测试方法中关键的步骤。在本章阐述了 DD 图无约束边的相关分析，目的在于结合无约束边集合，精简流程树中的测试路径集合，使用最少个数的测试路径，达到最充分的覆盖。在本章还提到了关于模块复杂性的度量与代码依赖关系的分析，这是出于对基于流程树的白盒测试方法的全面考虑，在集成测试或程序段非常复杂的情况下，先根据模块复杂性的度量选择测试模块或根据代码依赖关系的测算切取程序片段，然后再进行基于流程树的白盒测试。

第 4 章 基于流程树的白盒测试方法

程序流图回路较多，往往导致测试路径选取繁琐的状况。本文提出的基于流程树的白盒测试方法，通过将程序流图改造为流程树，使程序流程呈现树的形式，测试路径清晰可见，而且算法实现也简单易行。再结合对 DD 图无约束边理论的补充，精简测试路径的选取，提高了白盒测试的效率。

4.1 基于流程树的白盒测试方法的基本思路与方法描述

为了达到提高白盒测试效率的目的，作者在第 3 章对白盒测试关于模块重要度、DD 图无约束边、程序流程图、路径覆盖等内容进行了分析与研究，在本章提出了一个基于流程树的白盒测试方法，并对这种方法进行了实验。

4.1.1 基本思路

基于流程树的白盒测试方法测试全过程如图 4.1 测试过程图所示。

整个测试过程可以分为 5 个步骤。第 1 个步骤是重要度评价与程序切分。如果是在集成测试阶段，需要分析模块复杂度，选取当前最重要的模块进行测试。如果是在单元测试阶段，可认为当前的惟一模块就是最重要模块。然后，将复杂程序代码利用程序切片技术，截取为若干有逻辑意义的代码片段。第 2 个步骤是程序流图构造。按照本文第 3 章 3.4.1 程序流程图向程序流图的转换规则，构造对应的程序流图。第 3 个步骤是 DD 图生成。在生成的 DD 图中提取无约束边集合。第 4 个步骤是流程树获取，按照本文第 3 章 3.4.2 程序流图向流程树的转换规则，获取对应的流程树。每一条从树根节点出发到叶子节点为止的路径都是一条可测试的独立路径。第 5 个步骤是最佳路径筛选。在流程树所有可测试的独立路径中，遵循尽量充分地覆盖 DD 图无约束边的原则，筛选最佳测试路径，用最少的路径达到对程序段最充分的测试。

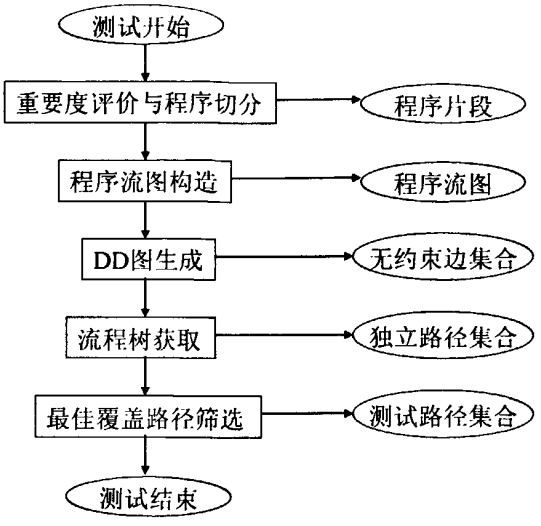

图 4.1 测试过程图

4.1.2 方法描述

步骤 1．重要度评价与程序切分

对于复杂的程序，尤其是代码行很多的程序，首先要以模块为单位进行重要度评价，然后再在模块内以“兴趣”为单位提取程序片段，这是整个测试的准备工作，如图 4.2 提取程序片段流程图所示。

(1)模块重要度

通过模块之间的关系绘制模块结构图，可以清晰看到模块的层次结构，便于计算模块复杂度。对于模块与模块之间的关系的研究与测试多用于集成测试阶段。而在单元测试阶段，不需进行此项步骤。

模块的重要度定义，见式(2-9)：

$$im(X)=\frac{C(X)}{\sum_{X}C(X)} \tag{2-9}$$

式中：$C(X)$—— 从顶结点开始到达和经过 X 的路径数目[40]

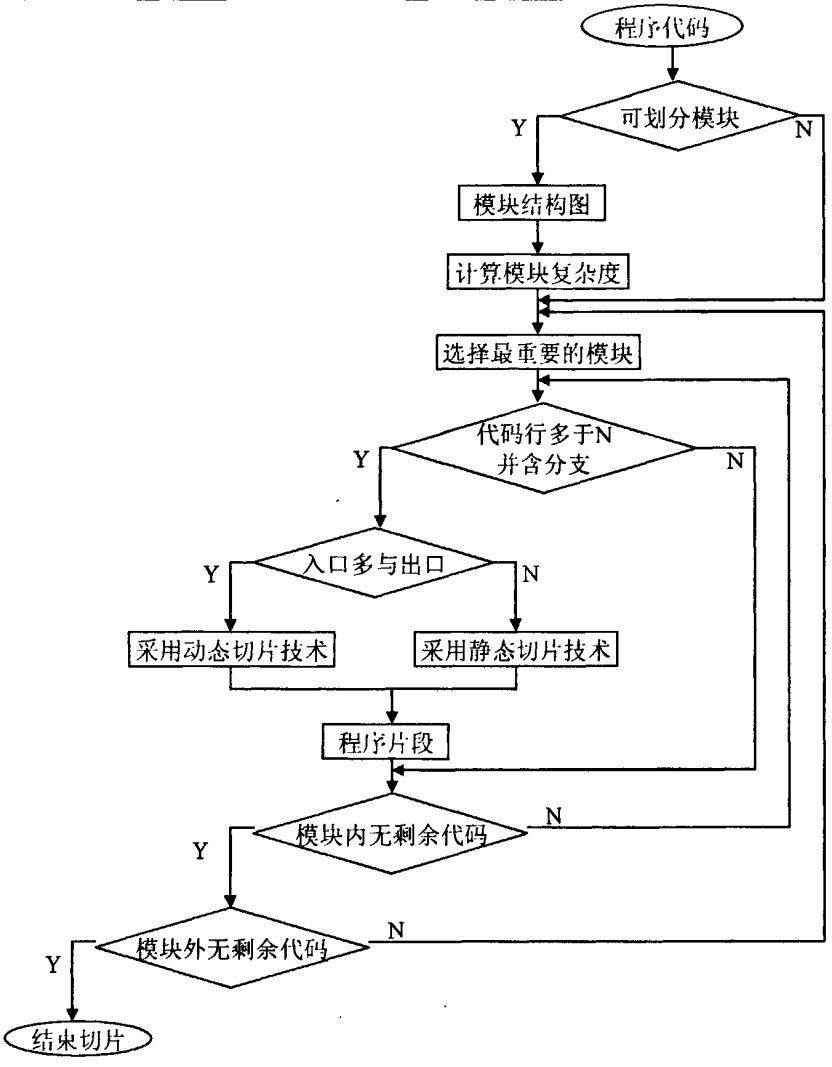

图 4.2 提取程序片段流程图

在图 4.3 模块图中，左图是模块结构图，右图模块关系有向图。

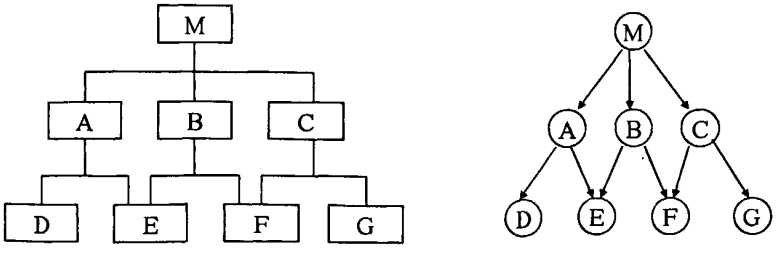

图 4.3 模块图

对图 4.3 模块图中各模块，根据式(2-9)模块重要度的计算结果，按重要度由大到小排列测试顺序，如表 4.1 由重要度产生的测试排队表所示。

表 4.1 由重要度产生的测试排队表

模块号	M	A	B	C	D	E	F	G
C(X)	9	3	3	3	1	2	2	1
im(X)	3/8	1/8	1/8	1/8	1/24	1/12	1/12	1/24
测试顺序	1	2	3	4	7	5	6	8

由以上结果可见，这种测试方法如果在集成测试中应用，将与传统集成测试中的自顶向下、自底向上方法都不同，因为测试顺序是按照重要度而定的，在顺序与意义方面都有不同。需要说明的是，本文为了便于重要度对比，当遇到模块重要度相同的情况时，是按模块图从左到右的顺序排列的，而在实际操作时遇到这种情况，可以根据软件测试人员的经验决定同等重要度的模块的测试顺序。

(2)程序切片

在当前最重要模块被确定以后，可以对该模块进行切片操作。

静态程序切片可以用S(V，N)的形式表示。其中V表示程序中的某一个变量或是变量的集合，N表示在程序中的某一个位置(变量V所在的语句)。S(V，N)的含义是“一个程序切片是由程序中的一些语句所组成的集合，这些语句可能会影响到在程序的某个位置N处所定义或引用的变量或变量的集合V的状态”。

动态切片可以定义为：S(V，N，X，I)。它表示程序在输入为X时，会影响到变量V在第I次执行语句N后的状态的所有语句的集合。

静态切片技术强调的是在可以遍历到的所有轨迹中，对程序中某一点的变量状态造成影响的所有语句；而动态切片技术则是强调程序在一次特定的执行中，会影响变量在程序中某一点的状态的所有语句。可以看出，动态程序切片是相应的静态程序切片的一个子集，因此，在做白盒路径覆盖测试的

时候，提倡使用静态切片技术。

例如，对以下程序段静态切分。

```
1.  int ch，sign，a1，a2，a3;
2.  sign=0;
3.  a1=0;
4.  a2=0;
5.  a3=0;
6.  ch=getchar( );
7.  while(ch!=EOF) {
8.      a3+=1;
9.      if(c= ='\n')
10          a1+=1;
11.     if(c= =' '||c= ='\t')
12.         sign=0;
13.     else if(sign= =0){
14.         sign=1;
15.         a2+=1;
16.         }
17.     c=getchar( );
18.  }
19. printf("%d\n"，a1);
20. printf("%d\n"，a2);
21. printf("%d\n"，a3);
```

S(a1，19)，是可能会影响到在程序的第19行语句printf("%d\n"，a1)所用的变量a1的状态。

对应于 S(a1，19)的程序切片为：

```
1. int ch，sign，a1，a2，a3;
3. a1=0;
6. ch=getchar( );
7. while(ch!=EOF) {
```

```
9.      if(c= ='\n')
10.         a1+=1;
17.     c=getchar( );
18.  }
19. printf("%d\n", a1);
```

由此可见，程序切片技术有针对性地提取了测试者感兴趣的部分，使较复杂的程序更加简明清晰。

步骤 2. 转换为程序流图

先将程序代码先转换为程序流程图，再根据程序流程图向程序流图的转换规则，构造对应的程序流图，规则详见本文第 3 章第 3.4.1 节。

步骤 3. 提取 DD 图无约束边

(1)主宰树

设 G=(V，E)是一个 DD 图，e_0 和 e_k 分别是 G 的惟一进入的边和惟一离开的边。边 e_i 主宰边 e_j，当且仅当从 e_0 到 e_j 的任何一条路径都通过 e_i。通过应用主宰关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_0，称为主宰树 DT(G)。树 T=(V，E)是一个图，拥有一个根节点，该节点的入度为 0。除根节点外，T 中的其他节点的入度都为 1，并且存在从根节点到其他节点惟一一条路径，T 中出度为 0 的节点称为叶子节点。e=(m，n)是 T 的一条边，则 n 是 m 的父节点，m 是 n 的子节点。在 DT(G)中的一个主宰路径 $P_{DT}=e_1e_2...e_q$，e_i=Parent(e_i+1)，$1\le i\le q$。主宰路径一般不是 G 中的路径。

(2)蕴涵树

设 G=(V，E)是一个 DD 图，e_0 和 e_k 分别是 G 的惟一进入的边和惟一离开的边。边 e_i 蕴涵边 e_j，当且仅当从 e_j 到 e_k 的任何一条路径都通过 e_i。通过应用蕴涵关系，可以把 DD 图 G 转换为树，该树的结点是 DD 图的边，树根是 e_k，称为蕴涵树 IT(G)。在 IT(G)中的一个蕴涵路径 $P_{IT}=e_1e_2...e_q$，e_i=Parent(e_i+1)，$1\le i\le q$。蕴涵路径一般不是 G 中的路径。

(3)无约束边

对于DD图路径覆盖的研究，可以引入无约束边的概念与一个重要的定理。

回顾定义3.1，无约束边：如果一个边e_u不主宰其他的节点也不被其他的

节点所蕴涵，则e_u称为无约束边。

根据定义3.1，可以推断，在主宰树DT(G)中不主宰其他节点的边是DT(G)中的叶子节点，而在蕴涵树IT(G)中不蕴涵其他节点的边是IT(G)中的叶子节点。根据这个定义，DD图无约束边的集合为：

UEdge(G)=DTLeaves(G)∩ITLeaves(G)

其中，其中UEdge(G)的无约束边的集合，DTLeaves(G)是主宰树DT(G)中叶子节点的集合，ITLeaves(G)是蕴涵树IT(G)中叶子节点的集合。

举一个 DD 图的例子，如图 4.4DD 图所示。

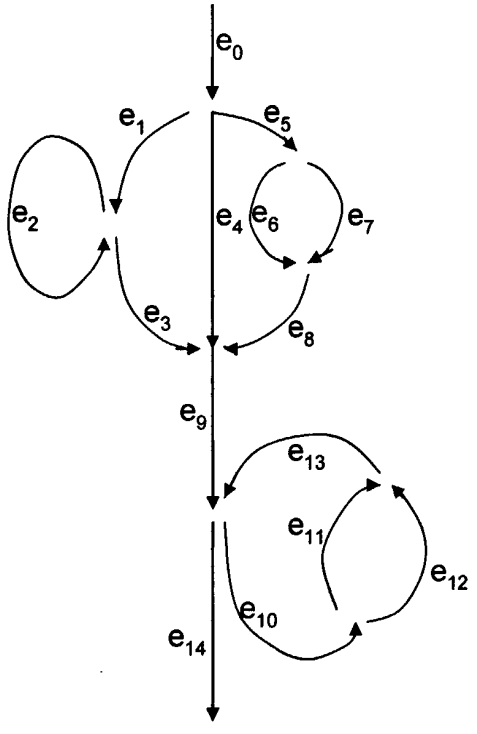

图 4.4 DD 图

通过运用主宰关系，可以把图 4-4DD 图 G 转换为 DT 树，该树的节点是 DD 图的边，树根是 e_0，如图 4.5DT 图所示。

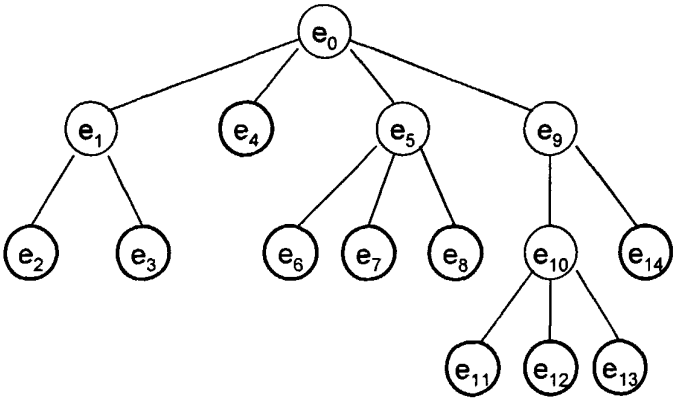

图 4.5 DT 图

可以得出，主宰树的叶子节点集合

DTLeaves(G)={e_2，e_3，e_4，e_6，e_7，e_8，e_{11}，e_{12}，e_{13}，e_{14}}

通过运用蕴涵关系，可以把 DD 图 G 转换为 IT 树，该树的节点是 DD 图的边，树根是 e_k，如图 4.6IT 图所示。

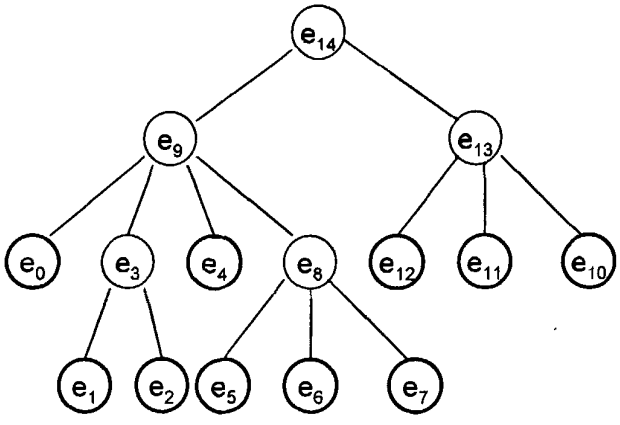

图 4.6 IT 图

可以得出，蕴涵树的叶子节点集合

ITLeaves(G)={e_0，e_1，e_2，e_4，e_5，e_6，e_7，e_{10}，e_{11}，e_{12}}

由 UEdge(G)=DTLeaves(G)∩ITLeaves(G)，从而提取无约束边集合

UEdge(G)= {e_2，e_4，e_6，e_7，e_{11}，e_{12}}

步骤 4. 获取流程树

根据程序流图向流程树的转换规则，获取与程序流图相对应的流程树，规则详见本文第 3 章第 3.4.2 节。图 4.4DD 图对应的流程树如图 4.7 流程树图所示。

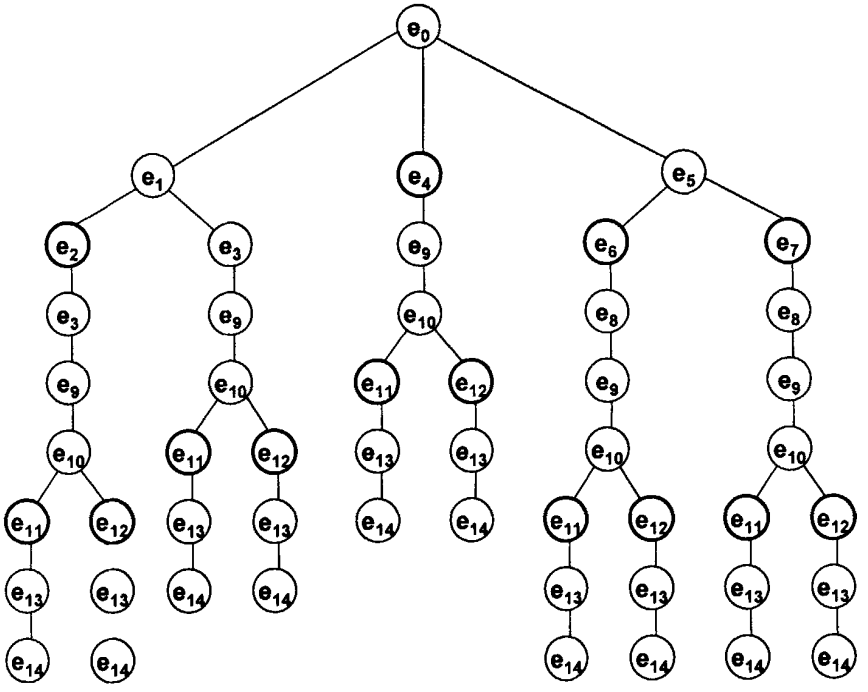

图 4.7 流程树图

在流程树中每一条从根节点到叶子节点的路径，都是一条独立路径，即一组测试用例。该树中所有这样的路径覆盖了需测试的程序段。

步骤 5．筛选最佳覆盖路径

通过图 4.7 流程树图，可以列出流程树中的所有不同的独立路径。如表 4.2 流程树的独立路径表所示。

回顾定理 3.1 ，G=(V，E)是一个 DD 图，e_0 和 e_k 分别是 G 中的惟一进入的边和惟一离开的边，有①覆盖所有无约束边的路径集合也覆盖了所有的路径。②在所有满足①的边的集合中，无约束边是最少的。由此定理可以看出，无约束边对于精简路径的重要性。

根据定理 3.1，可以推断，在选取覆盖路径时，要使每一条从 e_0 到 e_k 的路径尽可能充分地覆盖 UEdge(G)集合中的所有节点，由此可以构造出 G 的

最少路径覆盖集合 ε。

表 4.2 流程树的独立路径表

编号	路径	含无约束边数
1	e_0->e_1->e_2->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}	2
2	e_0->e_1->e_2->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2
3	e_0->e_1->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
4	e_0->e_1->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1
5	e_0->e_4->e_9->e_{10}->e_{11}->e_{13}->e_{14}	2
6	e_0->e_4->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2
7	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	2
8	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2
9	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	2
10	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2

基于以上推断，作者提出了筛选最佳覆盖路径的方案。通过循环计算当前未被选中路径中包含当前未被覆盖的无约束边的个数，选择最佳测试路径，直到无约束边全部被覆盖。实现流程，如图 4.8 实现筛选最佳覆盖路径流程图所示。

用图 4.4 的 DD 图举例，在初始状态时，未被覆盖的无约束边集合为 UEdge(G)= {e_2，e_4，e_6，e_7，e_{11}，e_{12}}，待选路径集合如表 4.2 流程树的独立路径表所示。

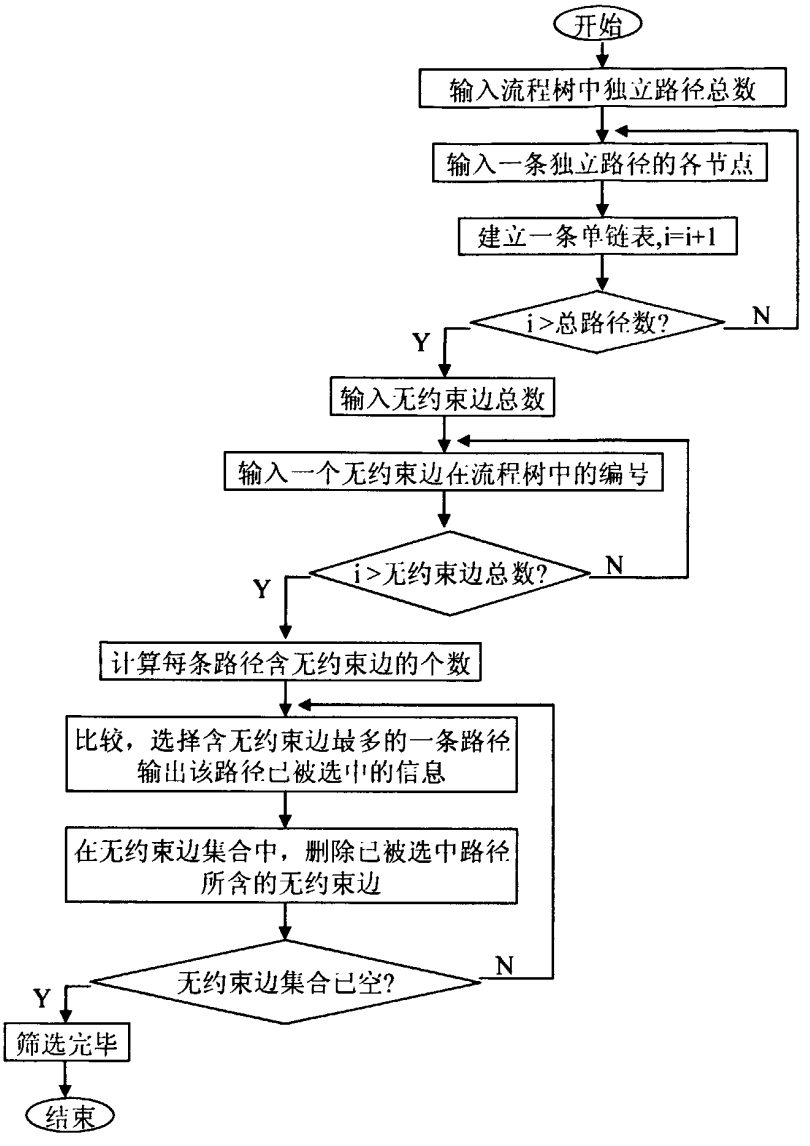

图 4.8 实现筛选最佳覆盖路径流程图

当执行第 1 次筛选时，由于路径 1 的边集为{e_0，e_1，e_2，e_3，e_9，e_{10}，e_{11}，e_{13}，e_{14}}，其中包含 2 个无约束边 e_2，e_{11}，不少于其他任何未被选中路径所含的无约束边个数，所以路径 1 被选中。未被覆盖的无约束边集合更新为 UEdge(G)= {e_4，e_6，e_7，e_{12}}。第 1 次筛选后的待选路径集合，如表 4.3 第 1 次筛选后的待选路径表所示。

表 4.3 第 1 次筛选后的待选路径表

编号	路径	含无约束边数
2	e_0->e_1->e_2->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1
3	e_0->e_1->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}	0
4	e_0->e_1->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1
5	e_0->e_4->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
6	e_0->e_4->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2
7	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
8	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2
9	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
10	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	2

当执行第 2 次筛选时，由于路径 6 的边集为{e_0，e_4，e_9，e_{10}，e_{12}，e_{13}，e_{14}}，其中包含 2 个无约束边 e_4，e_{12}，不少于其他任何未被选中路径所含的无约束边个数，所以路径 6 被选中。未被覆盖的无约束边集合更新为 UEdge(G)= {e_6，e_7}。第 2 次筛选后的待选路径集合，如表 4.4 第 2 次筛选后的待选路径表所示。

表 4.4 第 2 次筛选后的待选路径表

编号	路径	含无约束边数
2	e_0->e_1->e_2->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	0
3	e_0->e_1->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}	0
4	e_0->e_1->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	0
5	e_0->e_4->e_9->e_{10}->e_{11}->e_{13}->e_{14}	0
7	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
8	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1
9	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
10	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1

当执行第 3 次筛选时，由于路径 7 的边集为{ e_0，e_5，e_6，e_8，e_9，e_{10}，e_{11}，e_{13}，e_{14}}，其中包含 1 个无约束边 e_6，不少于其他任何未被选中路径所含的无约束边个数，所以路径 7 被选中。未被覆盖的无约束边集合更新为 UEdge(G)= {e_7}。第 3 次筛选后的待选路径集合，如表 4.5 第 3 次筛选后的待选路径表所示。

表 4.5 第 3 次筛选后的待选路径表

编号	路径	含无约束边数
2	e_0->e_1->e_2->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	0
3	e_0->e_1->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}	0
4	e_0->e_1->e_3->e_9->e_{10}->e_{12}->e_{13}->e_{14}	0
5	e_0->e_4->e_9->e_{10}->e_{11}->e_{13}->e_{14}	0
8	e_0->e_5->e_6->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	0
9	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}	1
10	e_0->e_5->e_7->e_8->e_9->e_{10}->e_{12}->e_{13}->e_{14}	1

当执行第 4 次筛选时，由于路径 9 的边集为{e_0，e_5，e_7，e_8，e_9，e_{10}，e_{11}，e_{13}，e_{14}}，其中包含 1 个无约束边 e_7，不少于其他任何未被选中路径所含的无约束边个数，所以路径 9 被选中。未被覆盖的无约束边集合更新为 UEdge(G)= {}。因为未被覆盖的无约束边的集合已为空，所以筛选完毕。

经过上述筛选，有 4 条测试路径被选中。

路径 1：e_0->e_1->e_2->e_3->e_9->e_{10}->e_{11}->e_{13}->e_{14}

路径 6：e_0->e_4->e_9->e_{10}->e_{12}->e_{13}->e_{14}

路径 7：e_0->e_5->e_6->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}

路径 9：e_0->e_5->e_7->e_8->e_9->e_{10}->e_{11}->e_{13}->e_{14}

当进行路径覆盖测试的时候，只需测试路径 1，路径 6，路径 7 与路径 9 这 4 条路径，就足以对图 4.4 的 DD 图进行完整的路径覆盖测试。这是对表 4.2 流程树的独立路径表中的 10 条路径的精简。

4.2 基于流程树的白盒测试方法的实验

基于流程树的白盒测试方法可用各种程序开发语言实现，如 C，Visual C++，Java，Visual Basic 等语言都可实现这种方法的编程。本文使用 C 语言进行实验。

4.2.1 数据结构

对于基于流程树的白盒测试方法的数据结构定义如下：

(1)每一个独立路径用单链表存储，单链表的节点定义为

```
typedef struct linknode
{
  int data;
  struct linknode *next;
}node;
```

data 用于存储路径节点编号，next 用于指向下一个节点。

(2) 所有独立路径的表头用一个指针数组存储，指针数组定义为

```
node *headarry[PSIZE];
```

headarry[1]，headarry[2]，……headarry[PSIZE-1]都是指针，分别指向流程树中的第 1 条路径，第 2 条路径，……第 PSIZE-1 条路径。

(3)无约束边集合用一个全局数组变量存储，定义为

```
int freeedges[SIZE];
```

4.2.2 部分代码

基于流程树的白盒测试方法的部分程序代码如下。

```
#include<stdio.h>
#include<stdlib.h>
typedef struct linknode
{
```

```
    int data;
    struct linknode *next;
}node;                  /*存储独立路径的节点*/
#define SIZE 20     /*DD 图的无约束边的个数*/
#define PSIZE 30    /*树的所有独立路径的个数*/
int freeedges[SIZE];   /*无约束边集合*/
node *createlink()      /*以链表形式创建树的一条路径*/
{
    node *head，*p，*s;
    int x，len，i;
    head=(node *)malloc(sizeof(node));
    p=head;
    printf("独立路径节点的总个数:");
    scanf("%d"，&len);
    printf("\n 请输入路径节点:");
    for(i=1;i<len+1;i++)
    {
      scanf("%d"，&x);
      if(x!=0)
      {
        s=(node*)malloc(sizeof(node));
        s->data=x;
        p->next=s;
        p=s;
       }
      else break;
    }
    head=head->next;
    p->next=NULL;
    return (head);
```

```
}
void printlink(node *head)      /*输出树的一条路径*/
{
 node *p;
 p=head;
 while(p!=NULL)
 {
   printf("->%d", p->data);
   p=p->next;
 }
}
void inputfreeedges()           /*存储无约束边集合*/
{
 int i, j, sum;
 for(i=0;i<SIZE;i++)
 {
   freeedges[i]=0;
 }
 printf("\n 请输入无约束边的个数:");
 scanf("%d", &sum);
 printf("\n 请输入无约束边:");
 for(i=1;i<sum+1;i++)
 {
   scanf("%d", &freeedges[i]);
 }
 freeedges[0]=0;
 for(j=i;j<SIZE;j++)
  {
    freeedges[j]=0;
   }
```

```
}
void printfreeedges()          /*打印无约束边集合*/
{ int i;
  printf("DD 图的无约束节点是:");
  for(i=1;i<SIZE+1;i++)
  {
    if(freeedges[i]==0)
        break;
    else
        printf("freeedges[%d]=%d  ", i, freeedges[i]);
  }
}
int pathedgestotal(node* head)  /*计算一个路径中含有的无约束边个数*/
{
  node *p;
  int i, total;
  p=head;
  total=0;
  printf("\n");
  while(p!=NULL)
    {
      i=1;
      while(freeedges[i]!=0)
        {
          if(p->data==freeedges[i])
            {
              total++;
            }
          i++;
        }
```

```
        p=p->next;
    }
    return total;
}

void pathedgesarry(node *head)   /*更新无约束边集合*/
{
    node *p;
    int i，j;
    p=head;
    j=1;
    printf("\n");
    while(p!=NULL)
      {
          i=1;
          while(freeedges[i]!=0)
              {
                  if(p->data==freeedges[i])
                      {
                          j=i;
                          while(freeedges[j]!=0)
                            {
                              freeedges[j]=freeedges[j+1];
                              j++;
                            }
                       }
                  i++;
              }
          p=p->next;
      }
```

```
}

int Best_testpath()
{
 node *headarry[PSIZE];
 node *sk;
 int pathnumber， i， j， k， n， maxtotal;
 int total[PSIZE];
 for(i=0;i<PSIZE;i++)      /*初始化路径指针与路径无约束边个数数组*/
  {
    headarry[i]=NULL;
    total[i]=0;
  }
 printf("\n 请输入独立路径的总条数:");
 scanf("%d"， &pathnumber);
 for(i=1;i<pathnumber+1;i++)   /*指针数组用于存放树的各条独立路径*/
  {
    printf("\n 请输入第%d 条"， i);
    headarry[i]=createlink();
  }
 inputfreeedges();
 printf("\n\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~");

 printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~");
 printf("\n 您已经输入的信息是:\n");
 for(i=1;i<pathnumber+1;i++)
  {
    printf("\n 第%d 条路径是:"， i);
```

```
    printlink(headarry[i]);
  }
 printf("\n");
 printfreeedges();
 printf("\n\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~");

 printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~");
 printf("\n 最佳测试路径的提取结果是:\n");
 printf("~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
~");
 k=0;
 while(freeedges[1]!=0)    /*如果无约束边集合没有被全部包含，就不断
 筛选*/
   {
     k++;
     printf("\n 现有");
     printfreeedges();
     printf("\n\n 执行无约束边第%d 次筛选的结果:"，k);
     maxtotal=0;
     for(i=0;i<SIZE;i++)
      {
         total[i]=0;
      }
     for(i=1;i<pathnumber+1;i++)   /*通过比较，选择一条包含无约束边
个数最多的独立路径*/
      {
        total[i]=pathedgestotal(headarry[i]);
        printf("\n 第%d 条路径含%d 个无约束边"，i，total[i]);
```

```
            if(total[i]>maxtotal)
                    maxtotal=total[i];
        }
      for(i=1;i<pathnumber+1;i++)
       {
         if(total[i]==maxtotal)
           {
             pathedgesarry(headarry[i]);
             printf("\n 路径%d 已被选中!\n", i);
           break;
            }
       }
printf("\n~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
~");
   }
printf("\n 最佳测试路径筛选完毕!\n");

printf("\n\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~");

printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~");
printf("\n");
return 0;
}
```

4.3 基于流程树的白盒测试方法特点与应用

基于流程树的白盒测试方法通过构造清晰的测试路径，提高了路径覆盖的效率，缩短了软件测试时间，在实际应用中可减少人力、物力消耗，节约

测试成本，提高经济效益。

4.3.1 方法特点

基于流程树的白盒测试方法节省了软件测试的时间。基于流程树的白盒测试方法在构造程序流程图与程序流图的基础上，遵循对程序代码逻辑意义不变的原则，消除程序流图回路，使其转换为流程树的形式。在流程树中，每一条从根节点到不同的叶子节点的路径，就是一条独立、完整的可测试路径。所有这些路径的集合覆盖了整个程序段。这种方法解决了程序流图回路较多时难以高效选取测试路径的问题，并且算法实现简单，节省了软件测试的时间。

基于流程树的白盒测试方法实现了充分的路径覆盖。基于流程树的白盒测试方法结合对 DD 无约束边的研究，提出了在流程树中用尽量覆盖无约束边的原则，通过循环计算未被选中路径中包含未被覆盖的无约束边的个数，选择最佳测试路径，直到无约束边全部被覆盖，精简可测试路径数目，利用最少的测试路径达到最充分的覆盖。

4.3.2 应用

基于流程树的白盒测试方法对模块内部与模块间的情况都进行了考虑，从软件测试阶段的角度来看，不仅可用于软件测试的单元测试阶段，而且可用于软件测试的集成测试阶段。

基于流程树的白盒测试方法节省了软件测试的时间，从软件开发周期的角度来看，不仅可用于普通软件的开发，而且可用语开发周期短的软件的开发。

基于流程树的白盒测试方法实现了充分的程序代码结构测试，从软件测试领域的角度来看，不仅可应用于普通商业软件，而且还可应用于精度要求较高的工业软件或军用软件。

4.4 本章小结

本章主要是利用流程树转换与DD图无约束边覆盖定理相结合，提出了基于流程树的白盒测试模型，通过循环计算流程树未被选中路径中包含未被覆盖的无约束边的个数，选择最佳测试路径，直到无约束边全部被覆盖，实现白盒测试最佳覆盖路径的选取。如果是含多个模块的复杂程序代码，建议使用重要度分析与程序切片技术对代码进行截取与简化，再进行最佳覆盖路径的选取。

结　论

白盒测试是针对软件内部代码的测试，是软件测试的重要部分。一种高效率的白盒测试方法能够及时发现软件代码中的缺陷，缩短软件的开发周期，减少软件的开发成本。而在目前的白盒测试方法中，对程序图处理只构造到程序流图为止。复杂程序的程序流图中回路较多，难以准确规划出每一条独立路径，而且需要用到深度遍历和广度遍历等复杂的算法，实现起来比较浪费时间，这样降低了白盒测试的效率。本文针对白盒测试中常见的测试用例繁多，测试不充分，效率低下等问题，通过研究，提出一种基于流程树的白盒测试方法，是一种较为完善的白盒测试方法。本文主要围绕以下几个方面开展工作并取得了成果。

(1)研究了路径覆盖，程序复杂度，程序切片，DD 图，程序流图等白盒测试的相关概念，找出了白盒测试方法需要改进的方面，为方法研究找准了方向。

(2)在构造程序流程图与程序流图的基础上，根据对程序代码逻辑意义不变的原则，提出了一种消除程序流图回路，使其转换为流程树形式的方法。在流程树中，每一条从根节点到不同的叶子节点的路径，就是一条独立、完整的可测试路径。所有这些路径的集合覆盖了整个程序段。这种方法解决了程序流图回路较多时难以高效选取测试路径的问题，并且算法实现简单，节省了测试的时间。

(3)通过对 DD 无约束边的研究，提出了在流程树中用充分覆盖无约束边的原则，通过循环计算当前未被选中路径中包含当前未被覆盖的无约束边的个数，选择最佳测试路径，直到无约束边全部被覆盖，精简可测试路径数目，利用最少的测试路径达到最充分的覆盖。

(4)通过对模块复杂性的度量，提出在集成测试阶段，先分析模块复杂度，选取当前最重要的模块进行基于流程树的白盒测试，做好测试前的准备工作。

(5)通过综合运用上述的测试方法，提出了一个基于流程树的白盒测试方法，该方法解决了白盒测试中常见的测试用例繁多，测试不充分，效率低下

等问题。

本文提出的基于流程树的白盒测试方法具有较高的应用价值和经济效益：

(1)基于流程树的白盒测试方法使所有独立路径清晰地呈现，能够达到充分的路径覆盖，及时发现软件中存在的隐患，提高软件的质量。

(2)基于流程树的白盒测试方法在达到充分覆盖的前提下，精简可测试路径数目，利用最少的测试路径达到了最充分的覆盖，缩短了白盒测试时间，减少了人力、物力消耗，提高了软件测试效率。

(3)基于流程树的白盒测试方法实现了充分的程序代码结构测试，从软件测试领域的角度来看，不仅可应用于普通商业软件，而且还可应用于精度要求较高的工业软件或军用软件。

本文提出的方法还存在着一些有待解决和改善的地方，例如对集成测试阶段的研究甚少。这些问题将会在本项目的后续工作中完成。

参考文献

[1] 韩永生，章雪梅. 基于程序切片的软件测试技术初探. 专题技术与工程应用. 2007，14(2)：1-2页

[2] 李文豪. 程序切片技术浅析. 小型微型计算机系统. 2006，27(1)：2-3页

[3] 董威，王戟，齐治昌. 并发程序的切片模型检验方法. 计算机学报. 2005，26(3)：1-2页

[4] 王伟，陈平. 程序切片技术综述. 微电子学与计算机. 2006，26(6)：1-4页

[5] Moataz A，Ahmed and Irman Hermadi. GA-based multiple paths test data generator. Computers & Operations Research. 2007，122(6)：89-94P

[6] Wen-Li Wang，Dai Pan and Mei-Hwa Chen. Architecture-based software reliability modeling. Journal of Systems and Software. 2006，79(1)：132-146P

[7] 李必信，王云峰，张勇翔，郑国良. 基于简化系统依赖图的静态粗粒度切片方法. 软件学报. 2006，27(11)：3-4页

[8] 伦立军，赵辰光，丁雪梅，李英梅. 路径覆盖构造方法. 计算机工程. 2006，29(8)：1-2页

[9] 樊庆林，吴建国. 提高软件测试效率的方法研究. 计算机技术与发展. 2006，16(10)：2-3页

[10] 宫云战. 软件测试. 国防工业出版社. 2006：56-60页

[11] 朱少民. 软件测试方法和技术. 清华大学出版社. 2005：120-126页

[12] 徐仁佐. 软件工程. 华中科技大学出版社. 2006：99-102页

[13] 李英梅，伦立军，丁雪梅. 动态程序切片研究及其应用. 软件学报. 2006，36(3)：2-4页

[14] Yoshinobu Tamura，Shigeru Yamada and Mitsuhiro Kimura. A reliability assessment tool for distributed software development environment based on Java and J/Link. European Journal of Operational Research. 2006，175(1)：435-445P

[15] Allen Haley and Stuart Zweben. Development and application of a white box approach to integration testing. Journal of Systems and Software. 2004，4(4)：309-315P

[16] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. Science of Computer Programming. 2007，64(1)：3-28P

[17] Durga P. Mohapatra，Rajeev Kumar，Rajib Mall，D. S. Kumar and Mayank Bhasin. Distributed dynamic slicing of Java programs. Science of Computer Programming. 2006，79(12)：1661-1678P

[18] 任凌燕，刘坚. 一种C++程序的分层切片方法. 计算机研究与发展. 2006，22(9)：2-3页

[19] 戚晓芳，徐宝文. 一种粗粒度并发程序切片方法. 微电子学与计算机. 2006，29(3)：3-4页

[20] 肖健宇，张德运，陈海诠，董皓. 一种改进的并发程序静态切片方法. 计算机工程. 2006，32(14)：1-2页

[21] 肖健宇，张德运，陈海诠，董皓. 一种改进的用于并发程序静态切片的程序依赖图. 微电子学与计算机. 2006，6(22)：2-4页

[22] 周婕，慕晓冬，王杰. 一种C++程序切片系统的设计与实现. 计算机技术与发展. 2006，16(7)：1-4页

[23] 李言平，晏海华，柳永坡. 一种C++单元测试支持工具的研究与设计. 计算机与数字工程. 2007，34(1)：2-3页

[24] 李必信，方祥圣，袁海，郑国良. 一种基于程序切片技术的软件测试方法. 计算机科学. 2006，16(6)：2-3页

[25] Dave Binkley，Sebastian Danicic，Tibor Gyimóthy，Mark Harman，Ákos Kiss and Bogdan Korel. A formalisation of the relationship between forms of program slicing. Science of Computer Programming. 2006，62(3)：228-252P

[26] Moataz A. Ahmed and Irman Hermadi. GA-based multiple paths test data generator. Computers & Operations Research. 2007，36(6)：128-139P

[27] J.Jenny Li，David Weiss and Howell Yee. Code-coverage guided prioritized test generation. Information and Software Technology. 2006，48(12)：

1187-1198P

[28] 周婕，慕晓冬，王杰. 一种基于程序切片算法的软件故障诊断策略. 微电子学与计算机. 2006，23(8)：2页

[29] 杨玲萍，周中元. 伴随测试的有效途径. 电子产品可靠性与环境试验. 2006，26(6)：3页

[30] 李艳华. 谈软件测试中的Bug. 电脑知识与技术. 2006，9(6)：1-4页

[31] 朱平，谭毅，李必信，郑国良. 一种基于分层切片模型思想的源程序信息分析方案. 计算机工程. 2006，8(8)：2-4页

[32] G.H.Walton and J. H. Poore. Measuring complexity and coverage of software specifications. Information and Software Technology. 2005，42(12)：859-872P

[33] N Malevris. A path generation method for testing LCSAJs that restrains infeasible paths. Information and Software Technology. 2005，37(8)：435-441P

[34] D. F. Yates and N. Malevris. An objective comparison of the cost effectiveness of three testing methods. Information and Software Technology. 2006，6(7)：45-47P

[35] Borislav Nikolik. Test diversity. Information and Software Technology. 2006，48(11)：1083-1094P

[36] Bernhard K. Aichernig and Chris George. When Model-based Testing Fails. Electronic Notes in Theoretical Computer Science. 2006，164(4)：115-128P

[37] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Information and Software Technology. 2006，48(10)：960-970P

[38] H. S. Wang，S. R. Hsu and J. C. Lin. A generalized optimal path-selection model for structural program testing. Journal of Systems and Software. 2006，10(1)：55-63P

[39] 易丹，吴方君. 一种基于覆盖测试的动态切片的计算方法. 应用科学学报. 2006，3(3)：2-3页

[40] 赵亮，王建民. 软件测试准则的有效性度量研究. 计算机研究与发展. 2006，8(1)：2-3页

攻读硕士学位期间发表的论文和取得的科研成果

发表论文：

孙长嵩，綦晶．一种优化的白盒路径测试方法(待发表).

科研项目：

2006.7-2007.9　某系统集成一体化设计平台——软件测试，软件使用说明书撰写

致　谢

首先向我的导师孙长嵩教授表达我最诚挚的谢意，感谢孙长嵩教授两年多来给予我的无私指导和帮助。在论文研究过程中，得到了指导教师孙长嵩教授的细心指导。在孙长嵩教授的具体指导下，从论文题目的确定到各阶段的任务都顺利完成。在整个研究生学习阶段，孙长嵩教授严谨踏实的作风、精益求精的治学态度、孜孜以求的进取精神为我培养良好的学风和提高工作能力树立了学习的榜样。同时，孙长嵩教授勤奋坚韧的性格，开拓创新的思想，谦恭礼让的作风，良好的修养，无时无刻不影响着我，激励着我。从孙长嵩教授身上学到的，是我一生宝贵的财富。

特别感谢刘大昕教授。在研究生学习期间，刘大昕教授给了我许多无私的帮助，为我的论文提出了许多宝贵的意见和建议。

感谢实验室各位老师、同学以及已经工作的同学，特别是湛浩旻老师、王卓、王红滨、张万松等。他们端正的学习态度，积极的进取精神一直激励着我不断学习。他们给我提出了许多的宝贵意见和参考资料，和他们的交流总是使我受益匪浅。

最后，要感谢我的家人，是他们一直对我的关爱和支持，使我得以顺利完成学业。