

软件测试基础

(美) Paul Ammann Jeff Offutt 著 郁莲 等译

乔治·梅森大学

北京大学



Introduction to Software Testing



机械工业出版社
China Machine Press

软件测试基础

“关于测试的书很多，但是大部分书涉及的主题范围都很窄并且讲述不详细。而Ammann和Offutt的这本书中所展示的概念和技术广泛地覆盖了业界和学术界使用的各种语言及平台。是一本全面、实用的测试书。”

——Roger Alexander, 华盛顿州立大学

本书采用创新方法来讲述软件测试，定义测试为将几个通用的测试准则应用于软件结构或软件模型的过程。书中融入了最新的测试技术，包括现代软件方法（如面向对象）、Web应用程序和嵌入式软件。另外，本书包含了大量的实例。

作者简介

Paul Ammann

在美国弗吉尼亚大学获得计算机科学博士学位，现为乔治·梅森大学软件工程副教授。他于2007年获得乔治·梅森大学Volgenau信息技术与工程学院的杰出教学奖。



Jeff Offutt

在乔治亚理工学院获得计算机博士学位，现为乔治·梅森大学软件工程教授。他是《Journal of Software Testing, Verification and Reliability》的主编，是IEEE软件测试、验证和确认国际会议指导委员会主席，还是许多期刊的编委。他于2003年获得乔治·梅森大学Volgenau信息技术与工程学院的优秀教师奖。



影印版
ISBN 978-7-111-29398-9
定价：42.00元

客服热线: (010) 88378991, 88381066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzjsj@hzbook.com

华章网站: <http://www.hzbook.com>

网上购书: www.china-pub.com

CAMBRIDGE
UNIVERSITY PRESS
www.cambridge.org



计算机、软件工程、软件测试

978-7-111-29398-9



9 787111 293989

定价：36.00元

计 算 机 科 学 丛 书

软件测试基础

(美) Paul Ammann Jeff Offutt 著 郁莲 等译

乔治·梅森大学

第1章 绪论
1.1 软件测试的重要性
1.2 软件测试的类型
1.3 软件测试的术语
1.4 软件测试的模型
1.5 软件测试的度量
1.6 软件测试的缺陷
1.7 软件测试的缺陷报告
1.8 软件测试的缺陷跟踪
1.9 软件测试的缺陷预防
1.10 软件测试的缺陷预防模型
1.11 软件测试的缺陷预防模型
1.12 软件测试的缺陷预防模型
1.13 软件测试的缺陷预防模型
1.14 软件测试的缺陷预防模型
1.15 软件测试的缺陷预防模型
1.16 软件测试的缺陷预防模型
1.17 软件测试的缺陷预防模型
1.18 软件测试的缺陷预防模型
1.19 软件测试的缺陷预防模型
1.20 软件测试的缺陷预防模型
1.21 软件测试的缺陷预防模型
1.22 软件测试的缺陷预防模型
1.23 软件测试的缺陷预防模型
1.24 软件测试的缺陷预防模型
1.25 软件测试的缺陷预防模型
1.26 软件测试的缺陷预防模型
1.27 软件测试的缺陷预防模型
1.28 软件测试的缺陷预防模型
1.29 软件测试的缺陷预防模型
1.30 软件测试的缺陷预防模型
1.31 软件测试的缺陷预防模型
1.32 软件测试的缺陷预防模型
1.33 软件测试的缺陷预防模型
1.34 软件测试的缺陷预防模型
1.35 软件测试的缺陷预防模型
1.36 软件测试的缺陷预防模型
1.37 软件测试的缺陷预防模型
1.38 软件测试的缺陷预防模型
1.39 软件测试的缺陷预防模型
1.40 软件测试的缺陷预防模型
1.41 软件测试的缺陷预防模型
1.42 软件测试的缺陷预防模型
1.43 软件测试的缺陷预防模型
1.44 软件测试的缺陷预防模型
1.45 软件测试的缺陷预防模型
1.46 软件测试的缺陷预防模型
1.47 软件测试的缺陷预防模型
1.48 软件测试的缺陷预防模型
1.49 软件测试的缺陷预防模型
1.50 软件测试的缺陷预防模型
1.51 软件测试的缺陷预防模型
1.52 软件测试的缺陷预防模型
1.53 软件测试的缺陷预防模型
1.54 软件测试的缺陷预防模型
1.55 软件测试的缺陷预防模型
1.56 软件测试的缺陷预防模型
1.57 软件测试的缺陷预防模型
1.58 软件测试的缺陷预防模型
1.59 软件测试的缺陷预防模型
1.60 软件测试的缺陷预防模型
1.61 软件测试的缺陷预防模型
1.62 软件测试的缺陷预防模型
1.63 软件测试的缺陷预防模型
1.64 软件测试的缺陷预防模型
1.65 软件测试的缺陷预防模型
1.66 软件测试的缺陷预防模型
1.67 软件测试的缺陷预防模型
1.68 软件测试的缺陷预防模型
1.69 软件测试的缺陷预防模型
1.70 软件测试的缺陷预防模型
1.71 软件测试的缺陷预防模型
1.72 软件测试的缺陷预防模型
1.73 软件测试的缺陷预防模型
1.74 软件测试的缺陷预防模型
1.75 软件测试的缺陷预防模型
1.76 软件测试的缺陷预防模型
1.77 软件测试的缺陷预防模型
1.78 软件测试的缺陷预防模型
1.79 软件测试的缺陷预防模型
1.80 软件测试的缺陷预防模型
1.81 软件测试的缺陷预防模型
1.82 软件测试的缺陷预防模型
1.83 软件测试的缺陷预防模型
1.84 软件测试的缺陷预防模型
1.85 软件测试的缺陷预防模型
1.86 软件测试的缺陷预防模型
1.87 软件测试的缺陷预防模型
1.88 软件测试的缺陷预防模型
1.89 软件测试的缺陷预防模型
1.90 软件测试的缺陷预防模型
1.91 软件测试的缺陷预防模型
1.92 软件测试的缺陷预防模型
1.93 软件测试的缺陷预防模型
1.94 软件测试的缺陷预防模型
1.95 软件测试的缺陷预防模型
1.96 软件测试的缺陷预防模型
1.97 软件测试的缺陷预防模型
1.98 软件测试的缺陷预防模型
1.99 软件测试的缺陷预防模型
2.00 软件测试的缺陷预防模型

Introduction to Software Testing



机械工业出版社
China Machine Press

本书经过了大量的课堂检验，是深受学生 and 行业专业人员欢迎的软件工程指南。本书所展示的软件测试概念和技术广泛地覆盖了各种语言及其平台。与其他软件工程书籍相比，本书内容更加全面，并具有很大的实践价值。

本书适合作为国内高等院校计算机及相关专业本科生的软件工程课程教材，也可供软件工程技术领域的人员参考。

Introduction to Software Testing (ISBN 978-0-521-88038-1) by Paul Ammann and Jeff Offutt first published by Cambridge University Press 2008.

All rights reserved.

This simplified Chinese edition for the People's Republic of China is published by arrangement with the Press Syndicate of the University of Cambridge, Cambridge, United Kingdom.

© Cambridge University Press & China Machine Press in 2010.

This book is in copyright. No reproduction of any part may take place without the written permission of Cambridge University Press and China Machine Press.

This edition is for sale in the People's Republic of China (excluding Hong Kong SAR, Macau SAR and Taiwan Province) only.

此版本仅限在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）销售。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2009-2583

图书在版编目（CIP）数据

软件测试基础 / (美) 阿曼 (Ammann, P.), 奥法特 (Offutt, J.) 著; 郁莲等译. —北京: 机械工业出版社, 2010.9

(计算机科学丛书)

书名原文: Introduction to Software Testing

ISBN 978-7-111-29398-9

I. 软… II. ①阿… ②奥… ③郁… III. 软件—测试 IV. TP311.5

中国版本图书馆CIP数据核字 (2009) 第237911号

机械工业出版社 (北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 李俊竹

三河市明辉印装有限公司印刷

2010年10月第1版第1次印刷

184mm × 260mm · 16.5印张

标准书号: ISBN 978-7-111-29398-9

定价: 36.00元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章教育

译者序

两位给软件工程和计算机科学专业的学生教授了15年软件测试课程的老师，历时7年，融合大量课堂经验，终于磨砺出这本《软件测试基础》。我同样从事了多年软件测试技术教学，看到这本书时，不禁感慨这是一本不可多得的教科书和参考书。

关于测试的书很多，但是大部分书涉及的主题范围都很窄并且讲述不详细，或围绕一个典型的软件开发周期的各个阶段展开，这样的方法使原本的测试主题变得难懂，而本书正是依靠其独特之处成为教科书或参考书的上佳选择。

经过大量的课堂检验，本书对于软件测试采用了可谓独具匠心的理解。它将软件测试定义为把许多定义良好的、通用的测试标准应用到软件结构或模型的过程，同时是生产高质量软件的一个不可或缺的实践工程活动。

本书用一种新颖而简单的结构把测试覆盖标准中复杂、晦涩的观点组织起来。从技术侧面说，软件测试是基于满足覆盖标准的。本书的观点是，真正不同的覆盖标准很少，各种覆盖标准很容易归为以下4类：图形、逻辑表达式、输入空间和语法结构。这不仅简化了测试，而且也易于将每个分类直接地理论化处理。传统的方法将开发过程中各个阶段的测试区别对待，而该方法与之形成鲜明对比。

本书的写作风格直接，从基础讲解概念，把所需的背景知识保持在最低，通篇包含了大量的实例，它把测试当作了客观的、可测量的和重复的量化活动的集合，同时也在必要的地方提出理论概念以支持测试工程师的后续实践活动。

本书采用模块化设计，彼此间相互关联，合理搭配，可以用于多种课程。书中的大部分内容仅需要基础的离散数学和编程知识就可以了。

本书在理论和实践应用之间保持了巧妙的平衡，重点讲解如何管理测试过程和测试者基于基础理论的具体测试技术，并且特别注重设计和创建设计测试用例的基本技术问题，旨在综合软件开发的整个过程，覆盖尽可能多的技术。

本书同时也可以使不同的角色从中受益。对于学生，本书使其可以学到软件测试背后的基本原理，学到如何应用这些原理来更快、更好地生产软件；对于教师，大量的练习、启发式的问题、课堂上的幻灯片和给出的课外活动使得教师很容易教授这些材料；企业的测试者，将发现本书收集了帮助提高他们测试水平的技术。

致谢

译者近几年来一直在北京大学软件与微电子学院从事软件测试技术的研究与授课。在翻译本书的过程中，学院对我的工作给予了极大的支持与重视。我的硕士研究生张瑒、李磊、张坚、伍晓东、赵文博、曹宇奇等同学参与了本书的部分翻译与整理工作，付出了很多努力，在此表示感谢。

郁 莲

20010年6月

前言

本书把软件测试当作生产高质量软件必不可少的一个工程实践活动。可以作为本科生或研究生软件测试课程的主要教材，也可以作为软件工程或数据结构等一般课程的补充材料，或者作为软件测试工程师和开发人员的资源。这本书有许多独特之处：

- 本书用一种新颖而简单的结构把测试覆盖标准中复杂的、晦涩难懂的观点组织起来。从技术层面上来说，软件测试是以满足覆盖标准为基础的。本书的中心观点是，真正不同的覆盖标准很少，各种覆盖标准很易于归为以下四类：图形、逻辑表达式、输入空间和语法结构。这不仅简化了测试，而且也易于将每个分类直接地加以理论化处理。传统的方法将开发过程中各个阶段的测试区别对待，而该方法与之形成鲜明对比。
- 本书是作为一本教科书来构思和撰写的，写作风格直截了当，从基础讲解概念，把所需的背景知识降到最低。本书包括了大量的例子、家庭作业和教辅材料。它在理论和实际应用之间保持了很好的平衡，把测试当作客观的、可测量和重复的量化活动的集合。本书会在必要的地方提出理论概念以支持测试工程师的后续实践活动。
- 本书认为，测试是帮助IT专业人士开发更高质量软件的一种智力训练。测试不是一个反工程化的活动，也不是一个具有内在破坏性的过程。本书不仅仅是针对测试专家的，也面向对编程或数学知之甚少的领域专家。
- 本书采用模块化结构，彼此间相互关联，所以可以用于多种课程。书中的大部分内容仅需要基础的离散数学和编程知识，需要更多背景知识的地方会清晰地标明。正如在前言的稍后部分描述的那样，对讲述内容进行合理搭配，本书可以用于多种课堂。
- 本书假定，读者学习的目标是成为用最低的成本做出最好的软件的工程师。书中的概念在理论上是非常基础的，但也是非常实用的，大多是当前正在使用的。

为什么要用这本书

不久前，软件开发公司可能会雇用不会测试的程序员和不会编程的测试者。对于大多数企业来说，双方没有必要知道对方背后的技术原理。在行业历史上，软件测试一直被当成一个非技术性的活动。软件企业主要从管理和过程的角度来看测试，对测试者的技能培训没抱多大期望。

随着软件工程行业日趋成熟，软件已渗透到人们日常生活的各个方面，对软件可靠性、可维护性和安全性的要求越来越高。软件企业必须用多种方法来应对这些变化，其中就包括改善软件测试方法。这需要提高测试工程师的专业技能，还需要不断强调软件开发者做测试的重要性。值得庆幸的是，经过30多年的研究和实践，已涌现出许多知识和技术。本书把这些知识整理起来，使学生、测试工程师、测试管理者和开发者都能使用。

同时，我们也发现在大学中教授测试的课程相对较少。只对少数本科生安排了测试课，对计算机科学或软件工程专业的硕士生几乎没有安排软件测试课程，少数十几个教学课程中安排一个测试选修课。测试不仅没有成为本科生计算机科学教育的必要部分，而且大部分计

计算机科学的学生根本就没有任何测试知识，或者只是作为一般课程的一部分，在软件工程课上讲几节课。

本书的作者给软件工程和计算机科学专业的学生讲授软件测试已经超过15年了。在此期间，我们得出了一个很不期望看到的结论：没有人写出一本我们想要的书。所以，如果我们想要，就必须自己写。

以前的测试类书籍把软件测试当作一个相对简单的学科，认为这个学科依赖于过程，而不是从技术的角度去理解软件是如何构成的；有的书把测试作为一个需要详细理解大量软件开发技术的复杂的、割裂的学科；还有的书把测试当作一个只有数学家和计算机理论科学家才能掌握的纯理论学科。大多数关于测试的书籍围绕着一个典型的软件开发周期的各个阶段展开，这种方法会使原本普通的测试主题变得难懂。最后，大多数测试类书籍是作为参考书而写的，而不是教科书。所以，只有先前有过专门软件测试知识的教师才能轻松使用。而本书对于那些不是测试专家的教师也是易于使用的。

本书在许多重要方面不同于其他的软件测试类书籍。许多书讲解如何管理测试过程，当然这很重要，但告诉测试者基于基础理论的具体测试技术同样重要。本书在理论和实践应用之间保持了很好的平衡。这是软件公司必须有的重要信息，但是本书特别注重设计和创建测试用例的基本技术问题。目前市面上其他的测试类书籍主要关注技术或活动，比如系统测试或单元测试，而本书旨在综合软件开发的整个过程，涵盖尽可能多的技术。

如前所述，本书的目的是支持多种软件测试课程。我们在乔治·梅森大学软件工程硕士研究生的软件测试课上做了第一个尝试，每学期有30多个计算机科学和软件工程专业的学生选修这门课程。我们还组织了软件测试的博士研讨会，举办了特定方向的短期企业培训，还为许多本科课程进行了讲座。虽然有关软件测试的本科课程不多，但我们相信不久的将来会有很多。许多关于测试的书并不是用在课堂上的，我们特地写了这本书来支持课堂教学，因此在本书网站（www.cs.gmu.edu/~offutt/softwaretest）上本书目录的后面看到我们的测试课程的提纲就不足为奇了。

本书采用了许多精心打造的实例来帮助学生和老师学习略显复杂的概念。教辅资源包括高质量的PPT、演讲提示、习题解答和相关软件。我们的思想是：我们不仅仅是在写一本书，同时也在为社区提供课程。我们的目标之一是，所写的内容作为研究文献有学术性，对于非研究人员也是易于使用的。虽然本书的论述与出自研究论文的资料有些不同，但其本质思想是忠实于文献的。为了使文章更为通顺，我们删除了论述中的引用。对于那些喜欢追根溯源的研究者，每章最后都有一个参考文献注释，它对概念的来源进行了总结。

本书的读者对象

学生通过本书可以学到软件测试背后的基本原理，学到如何应用这些原理来更快、更好地生产软件。他们不仅会成为优秀的程序员，而且他们还可以为未来的雇主实施高质量的软件测试活动。教师即使没有软件测试的实际经验，也可以在课堂上使用本书。即使教师不是软件测试方面的专家，本书附带的大量的练习、启发式的问题、课堂上的幻灯片和给出的课外练习也会使他们很容易教授这些课程。研究生（如一年级的博士生）会发现这本书是介入这一领域的无价之宝。清晰合理地讲述理论，用实际应用说明哪些有用，哪些没用；用高级阅读和参考文献把对这些内容感兴趣的读者引向相关资料。虽然软件测试的研究生相对较少，

但是我们认为他们是关键读者群，因为通俗、低门槛会使研究生较容易加入测试研究者社区。已经熟悉这一领域的研究者会发现“标准-方法”新颖而有趣。可能有人不同意这种教学方法，但是我们发现，把测试当作将有限的几个标准应用到极少数的软件结构中的观点对我们的研究很有帮助。我们希望将来的测试研究能够从寻找更多的标准转向对现有标准的创新使用和评测上。

企业中的测试者将发现本书收集了帮助他们提高测试水平的宝贵技术，无论他们当前的水平如何。这里所提出的标准更倾向于成为发现缺陷的技巧“工具箱”。阅读这本书的开发者将发现大量改善软件的方法。他们的自测活动将变得更快速、更有效，关于测试工程师找出软件缺陷的讨论将帮助开发者避开它们。正如一个很有名的寓言所讲的，如果你想教一个人成为好的渔夫，就要讲解鱼如何以及在哪里游泳。最后，管理者将发现本书很好地解释了聪明的测试工程师如何做好他们的工作，以及测试工具如何运行。这样，他们在雇人、晋升和购买工具时，就可以做出更有效的决定。

如何使用本书

本书结构的一个主要优势是，它能够轻松地用于多种不同的课程。大多数教材依赖大学和高中所教授的知识：数据结构和离散数学的基本概念。本书各个部分的组织使得每章前面的内容对于低年级或者基础不太好的学生也很易于使用，需要更高级知识的材料都会明确标出。

特别地，本书定义了7个单独的章节集合，它们组成了贯穿本书的模块分类：

- 1) 计算机专业大学二年级课程模块。
- 2) 软件测试专业大学二年级水平的课程模块。
- 3) 一般软件工程课程模块。
- 4) 软件测试高级课程模块。
- 5) 理学硕士一年级水平的软件测试课程模块。
- 6) 面向研究的研究生高级软件测试课程模块。
- 7) 从业人员的相关章节模块。

这种模块分类方法可见后面的简略目录。每个章节都标明了它属于哪个模块。当然，有的教师、学生和读者更愿意按照自己的兴趣或目标来使用这些模块。我们的建议是，第1章的前两节和第6章的前两节用于在数据结构（CS II）课程中阅读，它们后面都有一个简单的作业。我们最喜欢的是让学生们找一个他们以前写的评过分的程序，然后，让程序满足某种简单的测试标准，比如分支覆盖。每发现一个缺陷，我们都给出分数，这使人理解了两个概念：一是“A”级并不意味程序总可以运行，二是发现缺陷才是一件好事情。

	模块						
	1	2	3	4	5	6	7
第一部分 概览							
第1章 概述	■	■	■	■	■	■	■
1.1 测试工程师的工作	■	■	■	■	■	■	■
1.2 软件测试的局限性和术语	■	■	■	■	■	■	■
1.3 测试覆盖标准		■	■	■	■	■	■
1.4 以往的软件测试术语					■	■	
1.5 参考文献注释						■	

	模块						
	1	2	3	4	5	6	7
第二部分 覆盖标准							
第2章 图覆盖		☒	☒	☒	☒	☐	☐
2.1 概述		☒	☒	☒	☒	☐	☐
2.2 图覆盖标准		☒	☒	☒	☒	☐	☐
2.3 源代码的图覆盖		☒	☒	☒	☒	☐	☐
2.4 设计元素的图覆盖				☒	☒	☐	☐
2.5 规格说明的图覆盖				☒	☒	☐	☐
2.6 用例的图覆盖				☒	☒	☐	☐
2.7 用代数方法表示图					☒	☐	
2.8 参考文献注释						☐	
第3章 逻辑覆盖		☒	☒	☒	☒	☐	☐
3.1 概览：逻辑谓词和子句		☒	☒	☒	☒	☐	☐
3.2 逻辑表达式覆盖标准		☒	☒	☒	☒	☐	☐
3.3 程序的结构化逻辑覆盖		☒		☒	☒	☐	
3.4 基于规约的逻辑覆盖				☒	☒	☐	
3.5 有限状态机的逻辑覆盖			☒	☒	☒	☐	
3.6 析取范式标准					☒	☐	
3.7 参考文献注释						☐	
第4章 输入空间划分			☒	☒	☒	☐	☐
4.1 输入域建模			☒	☒	☒	☐	☐
4.2 组合策略标准			☒	☒	☒	☐	☐
4.3 划分中的约束					☒	☐	☐
4.4 参考文献注释						☐	
第5章 基于句法的测试		☒	☒	☒	☒	☐	☐
5.1 基于句法的覆盖标准		☒	☒	☒	☒	☐	☐
5.2 基于程序的语法			☒	☒	☒	☐	☐
5.3 集成与面向对象测试				☒	☒	☐	
5.4 基于规范的语法					☒	☐	
5.5 输入空间语法		☒	☒	☒	☒	☐	☐
5.6 参考文献注释						☐	
第三部分 在实践中运用的标准							
第6章 实际的考虑	☒	☒	☒	☒	☒	☐	☐
6.1 回归测试	☒	☒	☒	☒	☒	☐	☐
6.2 集成和测试	☒	☒	☒	☒	☒	☐	☐
6.3 测试过程			☒	☒	☒	☐	☐
6.4 测试计划			☒	☒	☒	☐	☐
6.5 识别正确的输出		☒	☒	☒	☒	☐	
6.6 参考文献注释						☐	
第7章 技术的工程标准				☒	☒	☐	☐
7.1 测试面向对象软件				☒	☒	☐	☐
7.2 测试Web应用和Web服务					☒	☐	☐
7.3 测试图形用户界面					☒	☐	☐
7.4 实时软件和嵌入式软件					☒	☐	☐
7.5 参考文献注释						☐	

(续)

	模块						
	1	2	3	4	5	6	7
第8章 创建测试工具							
8.1 图和逻辑表达式标准的插桩							
8.2 构造变异测试工具							
8.3 参考文献注释							
第9章 软件测试中的挑战							
9.1 测试紧急性属性：安全性和保密性							
9.2 软件的可测试性							
9.3 测试标准和软件测试的未来							
9.4 参考文献注释							

大学二年级水平的软件测试课程（模块2）紧跟着数据结构课程。标出部分的材料只需要数据结构和离散数学的知识。

一般的软件工程课程模块（模块3）可以增加这些课程的文献综述材料。标出部分提供了软件测试的基础知识。

软件测试的高级课程（模块4）是本书的主要部分。它增加的材料就软件开发而言需要比大学二年级模块更多的背景知识。这包括第2章中有关数据流测试的章节，涉及多个模块集成测试的章节和依赖于文法或有限状态机的章节。大多数三年级计算机科学专业的学生已经在其他的课程中看到过这些材料。出现在模块4但不在模块2中的大多数章节，可以加上适当的简介到模块2中。值得注意的是，测试工程师使用数据流测试时不需要懂得所有的语法分析理论，要使用测试状态图也不需要知道所有有限状态机的理论。

研究生水平的测试课程（模块5）增加了许多章节，这些章节需要大量的理论上有待成熟的背景知识。例如，某些章节需要初级形式化方法、多态和UML图的知识。许多高级主题和构建测试工具的整章内容也是为研究生准备的，可以为一个好项目提供一个基础，比如，实现一个简单的覆盖分析器。

研究生的高级软件测试课程（模块6）侧重于研究，就像一个博士的研讨会，包括仍未证实或还在研究的问题。参考文献注释是给那些想在将来进一步深入阅读的学生准备的。

最后，在模块7中标注出来的章节可广泛地用于企业中，特别是那些有商业工具支持的企业。这些章节涉及的理论最少，省略了那些在可用性上仍有问题的标准。

在本书的网站上可以获取大量的补充材料，包括教学大纲、PPT、演讲提示、习题解答、可运行的软件和勘误表。

致谢

在撰写本书的过程中，许多人帮助过我。乔治·梅森大学软件测试课上的学生们不仅能够非常宽容地使用本书的半成品，而且还很热情地提供反馈来改进本书。我们不能把这些学生一一列出（有10个学期的学生都使用了本书），下面仅列出了做出突出贡献的几位学生：Aynur Abdurazik、Muhammad Abdulla、Yuquin Ding、Jyothi Chinman、Blaine Donley、Patrick Emery、Brian Geary、Mark Hinkle、Justin Hollingsworth、John King、Yuelan Li、

Xiaojuan Liu、Chris Magrin、Jyothi Reddy、Raimi Rufai、Jeremy Schneider、Bill Shelton、Frank Shukis、Quansheng Xiao和Linzhen Xue。我们特别感激那些给本书提出无私的评论的人：Guillermo Calderon-Meza、Becky Hartley、Gary Kaminski和Andrew J. Offutt。我们感谢其他教育机构的初期使用者提出的反馈：Roger Alexander、Jane Hayes、Ling Liu、Darko Marinov、Arthur Reyes、Michael Shin和Tao Xie。我们也想感谢许多为本书提供素材的人：Roger Alexander、Mats Grindal、Hong Huang、Gary Kaminski、Robert Nilsson、Greg Williams、Wuzhi Xu。我们很高兴收到他们的建议：Lionel Briand、Renée Bryce、Kim King、Sharon Ritchey、Bo Sanden和Steve Schach。很感谢我们的编辑Heather Bergman提供的坚定支持，他我们的项目推迟了截稿时间，也同样感谢剑桥大学出版社的Kerry Cahill为这个项目提供的强大支持。

我们还要感谢乔治·梅森大学为我们两个人提供公休，还在关键时刻为我们提供助教。我们的系主任Hassan Gomaa也为这个项目提供了热情的支持。

最后，这本书的完成也离不开我们家人的支持。谢谢Becky、Jian、Steffi、Matt、Joyce和Andrew让我们稳定地生活，感谢他们在过去的5年中给我们带来的快乐。

正如所有程序都有缺陷一样，所有的书都有错误。我们的书也是如此。软件缺陷的责任在于开发者，而书中内容错误的责任在于我们作者。值得一提的是，参考文献注释部分反映了我们对于软件测试领域的观点，这项工作庞大而繁琐。我们提前为书中的纰漏致歉，敬请指正。

Paul Ammann

Jeff Offutt

目 录

出版者的话
译者序
前言

第一部分 概 览

第1章 概述	1
1.1 测试工程师的工作	2
1.1.1 基于软件活动的测试级别	3
1.1.2 基于测试过程成熟度的Beizer的 测试级别	5
1.1.3 测试活动的自动化	6
1.2 软件测试的局限性和术语	7
1.3 测试覆盖标准	12
1.3.1 不可行性与包含	14
1.3.2 好的覆盖标准的特征	15
1.4 以往的软件测试术语	16
1.5 参考文献注释	17

第二部分 覆盖标准

第2章 图覆盖	19
2.1 概述	19
2.2 图覆盖标准	23
2.2.1 结构化覆盖标准	24
2.2.2 数据流标准	33
2.2.3 图覆盖标准中的包含关系	38
2.3 源代码的图覆盖	40
2.3.1 源代码的结构化图覆盖	40
2.3.2 源代码的数据流图覆盖	41
2.4 设计元素的图覆盖	50
2.4.1 设计元素的结构化图覆盖	50
2.4.2 设计元素的数据流覆盖	51
2.5 规格说明的图覆盖	57
2.5.1 顺序约束测试	57
2.5.2 软件状态行为测试	60

2.6 用例的图覆盖	68
2.7 用代数方法表示图	71
2.7.1 把图简化成路径表达式	73
2.7.2 路径表达式的应用	75
2.7.3 得到测试输入	75
2.7.4 在流图中计算路径数并确定最大 路径长度	76
2.7.5 到达所有边的路径的最小值	77
2.7.6 互补运算分析	77
2.8 参考文献注释	79
第3章 逻辑覆盖	82
3.1 概览：逻辑谓词和子句	82
3.2 逻辑表达式覆盖标准	83
3.2.1 有效的子句覆盖	84
3.2.2 无效子句覆盖	87
3.2.3 不可行性和包含	88
3.2.4 使子句决定谓词	89
3.2.5 寻找满足的取值	91
3.3 程序的结构化逻辑覆盖	94
3.4 基于规约的逻辑覆盖	104
3.5 有限状态机的逻辑覆盖	106
3.6 析取范式标准	109
3.7 参考文献注释	116
第4章 输入空间划分	119
4.1 输入域建模	120
4.1.1 基于接口的输入域建模	121
4.1.2 基于功能的输入域建模	122
4.1.3 识别特性	122
4.1.4 选择块和值	123
4.1.5 使用一种以上的输入域模型	125
4.1.6 检查输入域模型	125
4.2 组合策略标准	126
4.3 划分中的约束	130

4.4 参考文献注释	131
第5章 基于句法的测试	134
5.1 基于句法的覆盖标准	134
5.1.1 BNF覆盖标准	134
5.1.2 变异测试	136
5.2 基于程序的语法	139
5.2.1 编程语言的BNF语法	139
5.2.2 基于程序的变异	139
5.3 集成与面向对象测试	151
5.3.1 BNF集成测试	151
5.3.2 集成变异	151
5.4 基于规范的语法	155
5.4.1 BNF语法	156
5.4.2 基于规范的变异	156
5.5 输入空间语法	158
5.5.1 BNF语法	158
5.5.2 输入语法的变异	161
5.6 参考文献注释	166

第三部分 在实践中运用的标准

第6章 实际的考虑	169
6.1 回归测试	169
6.2 集成和测试	170
6.2.1 桩和驱动程序	171
6.2.2 类的集成测试顺序	171
6.3 测试过程	172
6.3.1 需求分析和规格说明书	173
6.3.2 系统和软件设计	174
6.3.3 中级设计	174
6.3.4 详细设计	175
6.3.5 实现	175
6.3.6 集成	175
6.3.7 系统部署	176
6.3.8 操作和维护	176
6.3.9 总结	176
6.4 测试计划	177

6.5 识别正确的输出	181
6.5.1 输出的直接验证	181
6.5.2 冗余计算	182
6.5.3 一致性检查	182
6.5.4 数据冗余	183
6.6 参考文献注释	184
第7章 技术的工程标准	185
7.1 测试面向对象软件	185
7.1.1 面向对象软件测试特有的问题	186
7.1.2 面向对象的错误类型	186
7.2 测试Web应用和Web服务	201
7.2.1 测试静态超文本Web站点	202
7.2.2 测试动态Web应用	202
7.2.3 测试Web 服务	204
7.3 测试图形用户界面	205
7.4 实时软件和嵌入式软件	206
7.5 参考文献注释	209
第8章 创建测试工具	211
8.1 图和逻辑表达式标准的插桩	211
8.1.1 节点覆盖和边覆盖	211
8.1.2 数据流覆盖	213
8.1.3 逻辑覆盖	213
8.2 构造变异测试工具	215
8.2.1 解释方法	215
8.2.2 分离编译的方法	216
8.2.3 基于模式的方法	216
8.2.4 使用Java反射机制	217
8.2.5 实现一个现代的变异系统	217
8.3 参考文献注释	217
第9章 软件测试中的挑战	220
9.1 测试紧急性属性: 安全性和保密性	220
9.2 软件的可测试性	222
9.3 测试标准和软件测试的未来	225
9.4 参考文献注释	227
参考文献	229

第一部分 概 览

第1章 概 述

软件测试的思想和技术已经成为所有软件开发人员必备的知识。一个软件开发人员在其职业生涯中必会经常用到本书中所提到的概念。本章介绍了软件测试的一些主题内容,包括描述测试工程师的工作,定义一系列的关键术语,并且解释测试覆盖的核心概念。

软件已经遍布我们整个社会,在众多的设备和系统中它已成为关键的组成部分。软件定义了网络路由器行为、金融网络行为、电话交换网行为、因特网以及现代生活的其他基础设施行为。无论是对于像飞机、宇宙飞船、交通管理系统这样特殊的物件,还是对于像手表、烤箱、汽车、DVD播放器、车库自动门、手机以及远程遥控器这些家常的物品,软件都已经成为这些嵌入式应用的必要的组成部分。现代家庭有至少50个处理器,而有些新车则超过了100个处理器。乐观的消费者认为这些运行着的软件永远不会出错!虽然有很多因素影响着可靠软件的制作,这其中当然包括精心的设计和过程管理,但是,测试仍然是业界用于评估正在开发的软件的最首要的方法。幸运的是,对于多种多样的大量的软件应用来说,只用为数不多的基本的软件测试概念就可以进行测试设计了。本书的一个目标就是介绍这些概念,使得学生或是在职的工程师可以轻松地把它应用于任何的软件测试情况。

本书与其他的软件测试类书是有所区别的,主要体现在如下几个方面。最首要的不同在于它如何看待测试技术。Beizer在他的代表作《软件测试技术》(Software Testing Techniques)中写道,测试很简单,测试人员所要做的只是“找到一个图然后覆盖它”。得益于Beizer的卓见,对于我们来说很明显,现有的文献中大量的测试技术有很多的共同点,这与我们初次阅读这些文献的感受是不一样的。测试技术一般结合特定软件工件(artifact)的情况(例如需求文档或代码)或是软件生命周期的特殊阶段(例如需求分析或实现)来陈述的。不幸的是,这样的表述模糊了技术之间潜在的相似性,本书则阐明了这些相似性。

事实上图并没有完全体现出所有测试技术的特点,所以,我们还需要一些其他的抽象模型。令我们惊喜的是,我们已经发现有少数抽象模型能够满足我们的需要:图(graph)、逻辑表达式(logical expression)、输入域特征(input domain characterization)和句法描述(syntactic description)。本书最大的贡献在于将覆盖标准分为四大类,以此来简化测试,这也是本书第二部分分为四章的原因所在。

本书将理论与实践应用相结合,从而将测试展示为一系列客观的可度量、可复现的量化行为。该理论基于已出版的文献,没有过多地拘泥于形式。最重要的是,当测试工程师的实践活动需要有理论支持的时候,我们会介绍理论概念。也就是说,本书是供软件开发人员使用的。

1.1 测试工程师的工作

在本书中，测试工程师（test engineer）是专业的信息技术（Information Technology, IT）人员，他负责一到多项技术型测试活动，包括设计测试输入，生成测试用例值，执行测试脚本，分析测试结果，以及向开发人员和经理报告测试结果。虽然我们以测试工程师为角色进行描述，但是参与软件开发的每个工程师都应该意识到，在某些时候自己扮演着测试工程师的角色。原因在于在产品开发过程中所产生的软件工件都有（或应该有）相应的测试用例集，而最适合定义这些测试用例的通常是工件的设计者。一个测试经理负责一到多个测试工程师。测试经理制定测试策略和过程，在项目上与其他经理相互合作沟通，另外还会帮助测试工程师们工作。

图1.1说明了测试工程师的一些主要活动。测试工程师必须创建测试需求，以此来设计测试用例。这些需求之后会转化成用于测试执行的实例值和脚本。这些可执行的测试是在软件上运行的，在图中用P表示，而对测试结果的评估决定了这些测试是否揭露出软件的错误。这些活动是由1人或多人完成的，而测试的整个过程由测试经理进行监控。

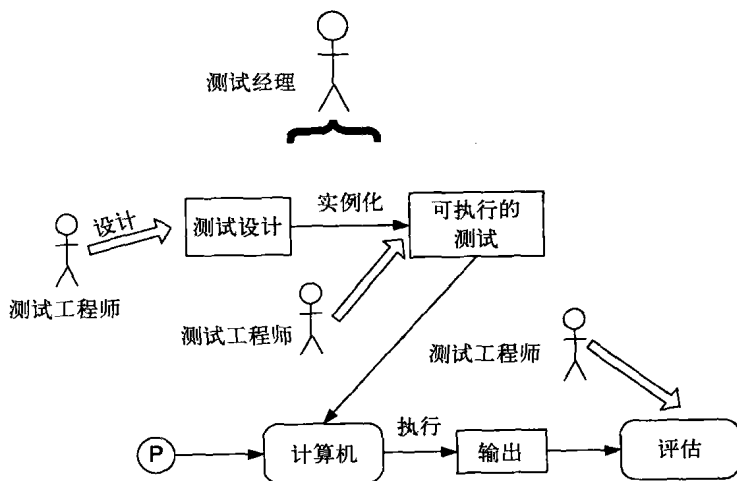


图1.1 测试工程师的活动

测试工程师最有力的工具之一就是正式的覆盖标准。正式的覆盖标准给测试工程师提供方法，来决定在测试中要用哪些测试输入，使测试人员更有可能发现程序中的问题，并且能够对软件的高质量和高可靠性提供更强有力的保障。覆盖标准也为测试工程师提供了测试停止准则。本书的技术核心是介绍当前可用的覆盖标准，描述工具（商用或其他）是如何支持这些覆盖标准的，解释如何最好地应用它们，并且提出如何将它们集成到整个开发流程中。

长久以来，软件测试活动按级别分类，其中有两种级别是经久沿用的。最常用的级别分类是基于传统的软件过程步骤。虽然大多数的测试类型只有在一部分软件实现了以后才能执行，但是在软件开发各步骤都可以进行测试的设计和构造。测试中最花费时间的事实上是测试的设计和构造，因此测试活动可以也应该贯穿开发的整个过程。第二级别分类是基于测试人员的态度和看法的。

1.1.1 基于软件活动的测试级别

测试可以由需求和规约、设计工件或源代码派生。不同的测试级别，伴随着不同的软件开发活动：

- 验收测试：根据需求评估软件；
- 系统测试：根据体系结构设计评估软件；
- 集成测试：根据子系统设计评估软件；
- 模块测试：根据详细设计评估软件；
- 单元测试：根据代码实现评估软件。

图1.2展示了各个测试级别的一个典型场景，以及这些测试级别与分步的软件开发活动是如何分别对应关联的。每个测试级别中的信息都典型地由相应关联的开发活动所派生出来。的确，标准的建议是在每个开发活动的同时就把相应的测试用例设计好，虽然在实现阶段以前软件还是不可执行的。这个建议的理由是仅仅明确、清晰地说明测试的流程可以识别设计决策的缺陷，否则这些设计决策看上去似乎是合理的。迄今为止，早期发现缺陷是减少最终开支的最有效方法。应该注意到，这幅图并不是就意味着它是瀑布式开发过程。综合与分析这些活动可以广泛地应用于任何开发过程。

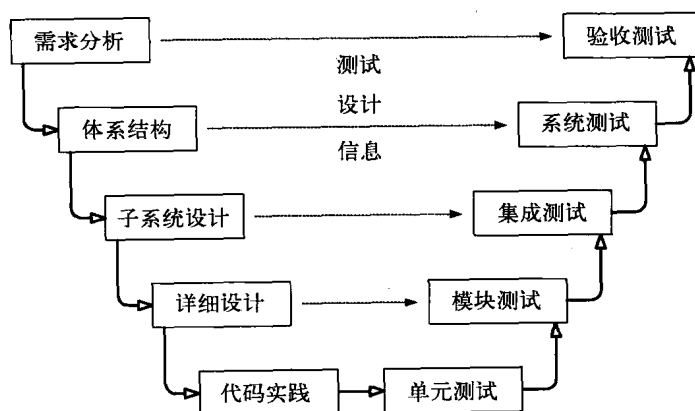


图1.2 软件开发活动与测试级别——“V模型”

软件开发过程的需求分析阶段是为了捕捉用户的需求。设计验收测试是为了确定软件成品事实上是否满足了这些需求。换言之，验收测试是为了查明软件是否是用户所需要的。验收测试必须有用户或那些拥有很好的领域背景知识的其他人员参加。

软件开发过程的体系结构设计阶段，是选择组件（component）和连接器（connector），二者结合来实现一个系统以便符合先前所确定的需求。设计系统测试是为了确定集成后的完整的系统是否与规约一致。系统测试假设各个部分单独工作正常，而且要问系统作为一个整体是否工作正常。这一级别的测试通常是为了寻找设计和规约中存在的问题。在该阶段，寻找更低级别的错误代价是昂贵的，而且通常不是由程序员来做而是由独立的测试组来完成。

软件开发过程的子系统设计阶段详细说明子系统的结构和行为，每个子系统分别实现整个体系结构中的某个功能。通常，子系统采用之前已开发好的软件。设计集成测试是为了评估子系统内模块（在下面定义）之间接口是否一致、通信是否正确。集成测试的前提是各个模块能正确工作。有些测试文献中把术语集成测试与系统测试互换使用，在本书中，集成测

试不涉及测试被集成的系统或者子系统。集成测试通常是由开发组成员负责的。

软件开发过程的详细设计阶段确定各个模块的结构和行为。一个程序单元 (unit) 或过程 (procedure), 是由一个或多个连续的语句组成, 并有一个名字, 软件其他部分使用该名字对程序单元进行调用。在C和C++中, 单元称为函数 (function); 在Ada中称为过程或函数; 在Java中称为方法 (method); 在Fortran中称为子例程 (subroutine)。将一系列相关联的单元组合在一个文件、包或者类中, 就称为一个模块 (module)。相当于C中的文件, Ada中的包, C++和Java中的类。设计模块测试是为了独立地评估各个模块, 包括组件单元间如何相互作用, 以及关联的数据结构。大多数软件开发组织将模块测试的职责交给程序员。

软件开发过程的实现阶段是实际产生代码的阶段。设计单元测试是为了评定实现阶段所产生的单元的正确性, 它是最底层的测试。有些时候, 比如在建一些通用的库模块时, 单元测试无需封装软件应用的知识。跟模块测试一样, 大部分软件开发组织会把单元测试的责任交给程序员。直截了当的做法是用一些像测试Java类的JUnit的工具将单元测试与相关的代码一起打包。

在图1.2中没有描述回归测试 (regression testing), 它是软件开发维护阶段的一个标准部分。回归测试是在对软件进行一些修改之后进行的测试, 目的在于帮助确保更新后的软件仍然具备更新前所拥有的功能。

在需求和高级别设计中存在的错误都会导致最终实现的程序中的缺陷; 测试可以揭露出这些缺陷。不幸的是, 由于需求和设计中的错误而导致的缺陷, 在这些原始的错误之后数月或数年内只有通过测试才能展现出来。这些错误的影响往往会蔓延到多个软件构件中, 因此这样的错误通常很难加以约束并且修正的代价很高。从积极的一面看, 即使不进行测试, 定义测试用例的过程本身就已经可以鉴别出一些重大的需求和设计错误。因此, 在需求分析和设计的同时就并行地推行测试计划是很重要的, 而不要将测试计划工作推迟到项目后期。幸运的是, 在标准的软件实践中, 用一些像用例分析 (use-case analysis) 这样的技术, 测试计划能更好地与需求分析相结合。

虽然很多文献中强调测试所应用的级别, 但事实上更重要的区分是在于我们所要找的缺陷类型。这些缺陷是基于我们所测试的软件工件, 以及我们生成测试用例所用到的软件工件。例如, 单元测试和模块测试是分别基于单元和模块的, 而我们通常试图发现在独立地执行单元和模块测试时所发现的错误。

区分单元测试与系统测试的一个最好的例子是声名狼藉的奔腾错误。在1994年, 英特尔公司推出奔腾微处理器, 而几个月之后, 弗吉尼亚州Lynchburg大学的一名数学家Thomas Nicely就发现对于某些浮点除法运算, 芯片会给出错误的运算结果。

芯片对于某些数字的组合会有微小的不精确性, 英特尔声称 (可能是真的) 只有90亿分之一的除法运算会出现精确度降低的问题。这个错误是由于除法算法所用到的一个含有1 066个数值的表中有五个数据遗漏。这五个输入地址应该包含常数+2, 而事实上这五个输入地址没有初始化而用0代替了。麻省理工学院的数学家Edelman称“奔腾的这个错误是个很容易犯的错误, 但是很难捕捉”, 一个有关遗漏某个基本点的分析。在系统测试中这是很难被发现的, 而事实上, 英特尔声称用这个表进行了数百万次的测试。但是表中这五个输入地址仍被留空, 这是由于一个循环的退出条件没有写对, 即在循环完成之前就停止了存储数据的操作。这在单元测试中其实是很容易发现的, 分析显示几乎任何单元级覆盖标准都能找到这个数百万美

元的错误。这个奔腾错误事件不仅印证了测试级别间的差异，而且也最有力地说明了应该给予单元测试更多的关注。没有捷径可走——软件的方方面面都需要测试。

另一方面，有些错误只能在系统级被发现。一个生动的例子就是第一颗Ariane 5火箭的发射失败，在1996年6月4日，火箭升空37秒后爆炸。深层的原因是一个惯性制导系统功能中一个浮点转换异常没有进行处理。事实上在Ariane 4火箭系统中使用这个制导系统永远不会发生这样的异常。也就是说，在Ariane 4上，这个制导系统功能是正确的。Ariane 5的开发人员理所当然想要复用Ariane 4的这个成功的惯性制导系统，但是没有人根据Ariane 5与Ariane 4实质上不同的飞行轨道进行重新分析。并且应该能发现这个问题的系统测试在技术上很难实现，因而就没有执行。后果是惨烈的，代价是巨大的！

另一个众所周知的失败案例是1999年9月的火星登陆者（Mars lander），由于两个独立软件团队分别开发的两个模块间对度量单位的理解不一致造成了它的坠毁。一个模块是英制单位计算推进器数据的，并将它传给另一个模块，而另一个模块所期望的数据是以公制单位计算的。这是一个很典型的集成错误（但这个案例无论是在资金还是在声誉方面都付出了高昂的代价）。

最后的一点是面向对象（object-oriented，OO）软件改变了测试分级。OO软件的单元和模块间界限模糊，因此OO软件测试方面的文献给出的是略有变化的测试级别。方法内测试（Intramethod testing）是为各个独立的方法构造测试。方法间测试（Intermethod testing）是将同一个类里的成对方法一起进行测试。类内测试（Intraclass testing）是针对一个完整的类构造的测试，通常是顺序调用类里的方法。最后，类间测试（Interclass testing）是同时测试多个类。前三个是属于单元和模块测试的变形，而类间测试是一种集成测试。

1.1.2 基于测试过程成熟度的Beizer的测试级别

另一种级别分类是基于一个组织测试过程成熟度水平。每个级别体现测试工程师的测试目的。下列材料是根据Beizer [29]改写的。

- 0级（Level 0） 没有区分测试与调试。
- 1级（Level 1） 测试的目的是证明软件能用。
- 2级（Level 2） 测试的目的是证明软件不能用。
- 3级（Level 3） 测试的目的不是为了具体证明什么，而是为了降低软件使用的风险。
- 4级（Level 4） 测试是一种智力训练，能够帮助所有的IT专业人员开发出更高质量的软件。

0级是一种将测试等同地视为调试的观点。许多计算机专业的学生都抱有这样的看法。在许多计算机科学的编程课上，学生编译程序后，用随意取的几个输入值或者教授给的值调试一下。这种模型并不区分一个程序的不正确行为和程序中的错误，因而对于软件的可靠性和安全性没有多大帮助。

在1级测试中，目的是说明正确性。从原始的0级到上升到一个重要的步骤，我们面临着一个令人遗憾的问题，事实上，除了非常小的程序外，对于其他的软件，正确性其实是根本无法达到或证明的。我们可以假设一下，在执行了一系列测试之后没有发现任何问题。我们能得到什么结论呢？我们该说我们有个好软件还是有个糟糕的测试呢？正是因为正确性这个目标是不现实的，测试工程师往往就没有严格的目标、真正的停止规则或正规的测试技术。

如果一个开发经理要问还剩多少测试没有完成，测试经理是无法回答这个问题的。事实上，测试经理处在一个被动的位置上，因为他无从量化或评估他们的工作。

在2级测试中，目的是找出错误。虽然找错显然是个合理的目标，但这也是个负面目标 (negative goal)。测试人员可能喜欢找出问题，但是开发人员永远都不希望找出问题——他们希望软件能正常工作（开发人员很自然会采取1级测试的思维方式）。因此，2级测试会将测试人员与开发人员放在对立的位置上，这对于团队士气很不利。换个角度，当我们的主要目标是要找错时，那么如果找不到错我们仍然会存有质疑该做什么。我们的工作真的干完了吗？是我们的软件很好了还是测试很弱呢？对于测试人员来说，在测试结束时能对产品有信心是个很重要的目标。

3级测试的想法是基于这样的认识，即测试能够展示失败的存在性，而非不存在性。这意味着我们必须接受这样的事实，当我们使用软件时，必然会面临一些风险。风险可能是比较小而结果也并不重要的，风险也可能是那种比较大而后果是灾难性的，但无论如何风险是一定存在的。这使我们意识到整个开发团队都有着同样的目标——降低使用软件的风险。在3级测试中，测试人员和开发人员一起协同工作来降低风险。

一旦测试人员与开发人员身处同一个“团队”中，组织就可以向真正的4级测试前进了。4级测试是将测试定义为一种提高质量的智力训练 (a mental discipline that increases quality)。能够提高质量的方式多种多样，创建能够使软件出错的测试用例只是其中之一。有了这样的思想准备，测试工程师就可以成为项目的技术领了（这在很多其他的工程学科中很普遍）。他们对于度量和提高软件质量担负着主要责任，并且应该用他们的专业才能帮助开发人员。Beizer把这比喻为一个拼写检查器。我们常常认为拼写检查器的目的是找出拼错的单词，但事实上，它最终的目的是提高我们拼写的能力。每次用拼写检查器找到一个拼错的单词，我们都获得了一次机会来学习如何正确地拼写该单词。拼写检查器是拼写质量方面的“专家”。同样，4级测试意味着测试的目的在于提高开发者生产高质量软件的能力。测试人员应该训练你们的开发人员。

作为本书的读者，你可能是从0级、1级或2级起步的。大部分的软件开发总会在其职业生涯的某个阶段经历这些级别。如果你是在软件开发领域工作，或许你可以停一下脚步，仔细想一下你所在的公司或团队处于哪个测试级别。本章接下来会帮助你转到2级思考方式，并能帮助你理解3级的重要性。后面的章节将会介绍3级所要用到的知识、技能以及工具。本书的终极目标是提供一个哲学的基础，使我们的读者能够以4级的思维成为其所在公司的“变革者”，而使测试工程师成为软件质量专家。

1.1.3 测试活动的自动化

软件测试代价高且是劳动密集型的。软件测试占用高达50%的软件研发经费，对于安全性应用软件甚至更高。软件测试的目标之一就是尽量使其自动化，从而显著地降低成本，将人工差错最小化，并且使得回归测试更容易。

软件工程师有时将收入任务 (revenue task) 与消费任务 (excise task) 区别开来，前者对于解决问题有直接的贡献，而后者则没有。例如，编译一个Java类是典型的消费任务，因为虽然要使该Java类变得可执行编译是必需的，但是编译本身对于类的任何特定的行为都没有贡献。相反，决定哪些方法适合于定义一个给定的数据抽象作为Java类则是一个收入任务。

消费任务是自动化的目标对象，而收入任务则不是。软件测试中的消费任务或许比软件开发过程中其他任何方面的消费任务都要多。维护测试脚本，重新执行测试用例，将实际结果与期望结果进行比较，这些都是很普遍的消费任务，它们周而复始地耗费着测试工程师的大量时间。将消费任务自动化，事实上是在很多方面都为测试工程师服务。首先，解决了这些消费任务就解决了苦差事，使得测试工程师的工作满意度提高。第二，自动化测试省出了时间，使得测试工程师能够专注于测试中更有趣更有挑战性的部分，就是我们所谓的收入任务的部分。第三，自动化可以帮助消除一些遗漏错误，例如，没有用新的期望结果集去更新所有的相关文档。第四，自动化可以消除因个体能力差距所导致的测试质量差异。

过去有许多测试任务难以实现自动化，而现今随着技术的进步，已经成为实现自动化的候选对象。例如，生成满足给定测试需求的测试用例是个典型的难题，因为它需要测试工程师的介入。尽管如此，现在已有一些正在研发或已经商业化的工具，都不同程度地使生成测试用例任务实现了自动化。

1.1 节练习

1. 哪些因素会帮助开发组织从Beizer的测试等级2（显示错误的测试）跳跃到测试等级4（一种提高质量的智力训练）？
2. 下面的练习能帮助你以一种比过去更严谨的方式理解测试。该练习也提示了规约的清晰度、错误以及测试用例之间密切的关联关系。

(a) 写一个具有以下签名的Java方法

```
public static Vector union (Vector a, Vector b)
```

此方法应该返回一个对象向量，这些对象来自两个参数向量中的对象。

- (b) 经过深思，你可能会发现在给定的作业中有各种缺陷与歧义。换句话说，它为错误的存在提供了充分的机会。所以，尽你所能识别出更多的错误。（提示：向量（Vector）是一个Java的集合（Collection）类，如果你在使用另一种语言，那么向量可以理解为一个列表（list）。）
- (c) 创建一组你认为有合理的机会揭示你在上面识别出来的错误的测试用例集。在你的测试集中，用文档为每一个测试写明其基本原理。如果可能，用某种简洁摘要方式描述所有的基本原理的特征。基于你的程序实现运行你的测试。
- (d) 重写这个方法的签名（signature），让它精确到足以能澄清之前识别的缺陷与歧义。你可能想用一些测试用例来举例说明你的规约。

1.2 软件测试的局限性和术语

如1.1.2节所说，软件测试最重要的限制之一是测试只能说明失败的存在，而不能说明失败不存在，这是一个根本性的、理论的限制。一般来说，在一个程序中找到所有错误的问题是无可判定的。测试者通常称一个成功的（或者有效的）测试是找到一个错误。然而这是一个2级测试思维的例子，也是一个常用的特征，稍后我们将在本书中用到这一点。

本节介绍若干术语，它们在软件测试中具有重要作用，并且本书后面将会使用到这些术语。多数术语摘自标准文档，即使我们自己措辞来表达这些术语，也尽量和标准文档相一致。要了解这些标准的更多细节，请参照IEEE软件工艺术语的标准词汇、美国国防部的DOD-

STD-2167A和MIL-STD-498标准, 以及英国计算机学会的软件组件测试标准。

其中最重要的一个区别是“确认”和“验证”的不同。

定义1.1 确认: 在软件开发结尾时, 评估软件以保证所开发的软件和预期用途相符的过程。

定义1.2 验证: 在软件开发过程的某个阶段, 决定此时的产品是否满足在前一个阶段所确定的需求的过程。

验证通常是一种更为技术的活动, 它使用有关软件工件、需求以及规格说明书等各方面的知识。确认通常依赖于领域知识, 也就是软件所应用的领域。例如, 飞机软件的确认需要来自航空工程师和飞行员的知识。

缩写“IV&V”表示“独立的验证与确认”(independent verification and validation), 其中“独立的”表示评估的过程由“非开发者”来完成。有时候, IV&V小组在同一个项目中, 有时候是在同一个公司里, 有时候完全是一个外部的实体。部分原因是因为IV&V的独立特性, 该过程通常直到软件完成时才开始并且通常由具有领域知识的专家来完成, 而不是由软件开发领域的专家来完成。有时候这意味着确认比验证更加重要。

我们已经使用的两个术语是“故障”和“失败”。理解这个区别是从0级思维转换到1级思维的第一步。我们采用可靠性社区对于软件故障、错误和失败的定义。

定义1.3 软件故障: 软件中的静态缺陷。

定义1.4 软件错误: 不正确的内部状态, 该状态是某个故障的表现。

定义1.5 软件失败: 与需求或其他期望行为的描述有关的、外部的、不正确的行为。

考虑一个医生为病人作出诊断。这个病人带着一些失败(即症状)进入医生办公室, 医生必须发现故障或者症状的根源。为了帮助诊断, 医生指定一些测试来寻找异常的内部条件, 比如高血压、心律不齐、高血糖或者高胆固醇。在我们的术语中, 这些异常的内部条件相当于错误。

虽然这个比喻可以帮助学生澄清他们对于故障、错误和失败的理解, 但是软件测试和医生诊疗在方式上有一个关键不同。具体地说, 软件中的故障是设计错误, 它们不是自发地出现的, 而是由人做出的一些不好的决策的结果。医疗问题(和计算机系统硬件故障一样), 从另一方面讲, 经常是物理退化的结果。这种区分很重要, 因为它说明了任何开发过程希望控制软件故障所能达到的程度极限。具体来说, 因为没有万无一失的方式来捕获由人类造成的任意错误, 所以我们不能从软件中消除所有的故障。在口头上, 我们可以说软件开发万无一失, 但是我们无法做到, 也不该尝试着做到根本不可能的万无一失。

对于故障、错误和失败, 有一个更加专业的例子, 参看下面的Java代码:

```
public static int numZero (int[] x) {  
    // 效果: if x == null 抛出异常 NullPointerException  
    // 否则返回序列x中0出现的次数  
    int count = 0;  
    for (int i = 1; i < x.length; i++)  
    {  
        if (x[i] == 0)  
        {  
            count++;  
        }  
    }  
}
```

```

    }
}
return count;
}

```

该程序中的故障是，它是从索引1而不是索引0开始寻找0，Java中的数组必须从0开始。例如，`numZero([2, 7, 0])`被运算为1，这是正确的，`numZero([0, 7, 2])`被运算为0，这是不正确的。在这两个例子中，故障被执行了。虽然这两个例子都以错误而结束，但只有第二个例子的结果是失败。为了理解错误状态，我们需要识别程序的状态。对`numZero`来说，其状态包括了变量`x`、`count`、`i`和程序计数器（记为PC）的值。对于上面给定的第一个例子，在循环的第一次迭代中`if`表达式的状态是（`x = [2, 7, 0]`, `count = 0`, `i = 1`, `PC = if`）。注意，因为在第一次迭代时`i`的值应当为0，所以该状态是一个错误。但是，因为总数`count`的值恰巧是正确的，错误的状态没有一直传播到输出，所以软件没有失败。换句话说，如果一个状态不是所期望的状态，那么它就是一个错误，即使单独考虑该状态中所有的值时是可以接受的。更一般地说，如果需要的状态序列是 $s_0, s_1, s_2, \dots$ 而实际的状态序列是 $s_0, s_2, s_3, \dots$ 那么在第二个序列中状态 s_2 是一个错误。

在第二个例子中，相应的（错误）状态是（`x = [0, 7, 2]`, `count = 0`, `i = 1`, `PC = if`）。在这个例子中，错误传播到变量`count`并且显示在方法的返回值中，因此这是一个失败的结果。

故障和失败的定义让我们把测试和调试区分开来。

定义1.6 测试：通过观察其执行来评估软件。

定义1.7 测试失败：执行的结果是失败。

定义1.8 调试：对于给定的失败找到故障的过程。

当然，中心问题是对于一个给定的故障，不是所有的输入都会“触发”故障来创建不正确的输出（一个失败）。通常也很难将一个失败与相应的故障关联起来。这些分析引出了故障/失败模型，表明要找出一个失败应当考虑下面3个条件：

1. 程序中包含故障的位置必须找到（可达性，reachability）。
2. 执行该位置后，程序的状态必须是不正确的（影响，infection）。
3. 受到影响的状态必须传播出来，引起该程序的某个输出是不正确的（传播，propagation）。

“RIP”模型对于覆盖标准，比如突变（见第5章）非常重要，对于自动产生测试数据也很重要。有一点需要指出：即使在故障有遗漏的情形下，也可以应用RIP模型。特别是当执行遍历有丢失的代码时，程序计数器（也是内部状态的一部分）有一个错误的值很必要。

下面的一些定义不是标准化的，而且随文献不同变化较大。这些定义虽然是我们自己给出的，但是和通常的用法是一致的。一个测试工程师应当意识到测试不仅仅是输入值，而实际上是多方软件工件。然而通常所指的一个测试用例就是我们所说的测试用例值。

定义1.9 测试用例值：完成被测软件的某个执行所需的输入值。

注意，测试用例值的定义是非常广泛的。在一个传统的批处理环境中，定义很清楚。在一个Web应用中，一个完整的执行小到生成一个简单网页的部分，或者复杂到完成一系列商业交易。在一个实时系统中，比如航空电子设备的应用中，一个完整的执行可以是一个单帧，或者是一次完整的飞行。

测试用例值是在测试阶段测试工程师通常关注的程序的一些输入。它们真正定义了我们完成的是什么样的测试，但是，仅是测试用例值还不够。除了测试用例值，要运行一个测试通常还需要其他输入，这些输入可能依赖于测试的源，可以是命令、用户输入或者一个带有参数值的软件方法的调用。为了评价一个测试的结果，我们应当知道对于该测试程序的正确版本应当产生什么样的输出。

定义1.10 预期的结果：当且仅当程序满足其期望行为时，执行测试时产生的结果。

与软件测试相关的两个常见的实际问题是如何为软件提供正确的值并且观察软件行为的细节，这两点想法用于提炼出测试用例的定义。

定义1.11 软件可观察性：如何简单地通过观察一个程序的输出、对于环境及其他硬件和软件组件的影响，观察一个程序的行为。

定义1.12 软件可控性：如何容易地给程序提供一个带有所需的输入，包括值、操作和行为。

这些概念在嵌入式软件环境中很容易说明。嵌入式软件通常不为人的消费产生输出，但是影响一些硬件的行为。因此，可观察性会非常低。同样，有些软件所有的输入都是从键盘输入的值，这些软件比较容易控制。但是一个从硬件传感器得到输入的嵌入式程序就比较难控制了，并且要提供某些输入可能是困难的、危险的或不可能的（例如，当火车偏离轨道时，自动驾驶员是如何采取行动的）。许多可观察性和可控性问题可以用模拟器来处理，通过额外的软件构建来“绕过”和测试相冲突的硬件和软件组件。可观察性和可控性有时更低的其他应用包含了基于组件的软件、分布式软件和网络应用。

依照软件、测试级别和测试源，测试者可能需要为软件提供其他的输入来影响其可控性或可观察性。例如，如果我们测试一个移动电话软件，测试用例值可以是长途电话号码。我们也可能需要打开电话将其置于合适的状态，然后可能需要按下“talk”按钮和按下“end”按钮来看测试用例值的结果，并且终止测试。这些想法表述如下：

定义1.13 前缀值：将软件置于合适的状态来接受测试用例值的任何必要的输入。

定义1.14 后缀值：测试用例值被发送以后，需要被发送到软件的任何输入。

后缀值可以再细分为两种类型。

定义1.15 验证值：查看测试用例值的结果所必须要用到的值。

定义1.16 结束命令：需要终止程序或者返回到稳定状态所需要用到的值。

一个测试用例是所有这些组件的组合（测试用例值、期望结果、前缀值和后缀值）。但是，当语义环境非常清楚时，我们将遵从传统，并且使用术语“测试用例”来代替“测试用例值”。

定义1.17 测试用例：一个测试用例由测试用例值、期望结果、前缀值、后缀值所组成，以便实现一个完整的执行以及对被测软件的评估。

我们给测试集提供了一个明确的定义，用来强调覆盖是测试用例集的属性，而不是单个

测试用例的属性。

定义1.18 测试集：一个测试集是测试用例的集合。

最后，聪明的测试工程师尽可能多地使测试活动自动化。测试自动化的一个关键方式是准备测试输入，作为软件的可执行测试。这可以像Unix shell脚本、输入文件那样来完成，或者通过使用可以控制被测的软件或软件组件的工具来完成。理想情况下，执行应当从以下方面完成：运行带有测试用例值的软件，获得结果，把结果和期望结果相比较，以及为测试工程师准备一个清晰的报告。

定义1.19 可执行测试脚本：一个测试用例的形式是自动化地执行被测软件并且产生一个报告。

测试工程师不希望自动化的唯一情况是自动化的花费比收益多。例如，如果我们确定测试只使用一次，或者如果自动化需要测试工程师所不具有的知识、技能时，这种情况就可能发生。

1.2节练习

1. 测试者为什么要使用自动化？自动化的限制是什么？
2. 故障和失败是如何关联到测试和调试的？
3. 以下是4种有故障的程序，每个包含了一个以失败为结果的测试用例。回答以下对于每段程序的问题。

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
// else return the index of the last element
// in x that equals y.
// If no such element exists, return -1
for (int i=x.length-1; i > 0; i--)
{
if (x[i] == y)
{
return i;
}
}
return -1;
}
// 测试值: x=[2, 3, 5]; y = 2
// 期望值 = 0
```

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
// else return the index of the LAST 0 in x.
// Return -1 if 0 does not occur in x

for (int i = 0; i < x.length; i++)
{
if (x[i] == 0)
{
return i;
}
}
return -1;
}
//测试值: x=[0, 1, 0]
// 期望值 = 2
```

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
// else return the number of
// positive elements in x.
int count = 0;
for (int i=0; i < x.length; i++)
{
if (x[i] >= 0)
{
count++;
}
}
return count;
}
// 测试值: x=[-4, 2, 0, 2]
// 期望值 = 2
```

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
// are either odd or positive (or both)
int count = 0;
for (int i = 0; i < x.length; i++)
{
if (x[i]%2 == 1 || x[i] > 0)
{
count++;
}
}
return count;
}
// 测试值: x=[-3, -2, 0, 1, 4]
// 期望值 = 3
```

- (a) 确定故障。
- (b) 如果可能, 确定一个不会执行故障的测试用例。
- (c) 如果可能, 确定一个执行故障但不会导致错误状态的测试用例。
- (d) 如果可能, 确定一个导致错误而不是失败的测试用例。提示: 别忘记程序计数器。
- (e) 对于给定的测试用例, 确定第一个错误状态, 确保描述完整的状态。
- (f) 修改故障并且验证给定的测试现在可以产生一个期望的输出。

1.3 测试覆盖标准

在测试中偶尔会出现一些错误定义的术语, 例如“完全测试”、“彻底测试”和“全面覆盖”。这些术语之所以被称为错误定义是因为软件基本理论的限制。特别是, 大多数程序的输入可能会有很多, 甚至趋向无穷大。考虑一个Java编译器, 对于该Java编译器的输入可能不仅仅是所有的Java程序或所有正确的Java程序, 而是所有的字符串。编译器的分析器所读取的文件大小就是唯一的限制, 因此, 输入的数量实际上是趋向于无穷, 不能被一一列举。

以上问题正是规范的覆盖标准要解决的。因为我们不可能去测试所有的输入, 覆盖标准用来决定要用到哪些测试输入。软件测试社区相信这样的原则, 有效地使用覆盖标准可以让测试工程师更可能发现程序中的错误, 而且提供了非规范的保证, 即软件是高质量的, 具有高可靠性。虽然这更是一种信赖而不是一个有科学依据的命题, 但我们认为这是现有的最好的选择。从一个实用的观点来看, 覆盖标准提供了有用的规则去决定什么时候结束测试。

本书是根据测试需求定义覆盖标准。基本思想是在测试用例集中具有不同的属性, 每个属性由一个测试用例^①提供 (或不提供)。

定义1.20 测试需求: 测试需求是软件工件的一个特定元素, 测试用例必须满足或覆盖这个特定元素。

测试需求通常是成套出现, 我们用缩写TR表示一组测试需求。

测试需求可以针对不同的软件工件来描述, 包括源码、设计文档、规格说明模型元素甚至输入空间的描述。稍后, 本书将介绍如何从以上述工件生成测试需求。

让我们以一个非软件的实例开始吧。假设我们有一份令人羡慕的工作——测试袋装的软心豆粒糖。我们必须想办法筛选出不同的袋子。假设这些糖豆有以下6种口味和4种颜色: 柠檬口味 (黄色)、开心果口味 (绿色)、哈密瓜口味 (橙色)、梨子口味 (白色)、橘子口味 (也是橙色) 和杏口味 (也是黄色)。一个简单的测试方法可以是每种口味尝试一个糖豆。那么, 我们就有了6个测试需求, 每一个需求对应一种口味。我们通过选择和尝试袋中柠檬口味的糖豆去满足“柠檬味”测试需求。读者在尝试之前要思考如何去判断给定的一颗黄色糖豆是柠檬味的还是杏味的。以上这个困难的选择说明了一个典型的可控性问题。

用一个软件实例来说明, 如果目标是覆盖程序中的所有判定 (分支覆盖), 那么每个判定将产生两个测试需求, 一个使判定为false, 一个使判定为true。如果每个方法必须要被调用一次 (调用覆盖), 每个方法会产生一个测试需求。

一个覆盖标准可以简单地看做是一个菜单, 它可以系统地产生测试需求:

① 虽然这是一个很好的一般性规则, 但也有例外。对于一些逻辑覆盖标准的测试需求要求成对的相应测试用例而不是单独的测试用例。

定义1.21 覆盖标准：一个覆盖标准是一条规则，或者是将测试需求施加在一个测试集上的一组规则。

也就是说，覆盖标准以一种全面并且准确的方式描述了测试需求。“口味标准”为糖豆的选择提供了一个简单的策略。这样，测试需求集 TR 可以规范地描述如下：

$$TR = \{ \text{flavor} = \text{Lemon}, \text{flavor} = \text{Pistachio}, \text{flavor} = \text{Cantaloupe}, \\ \text{flavor} = \text{Pear}, \text{flavor} = \text{Tangerine}, \text{flavor} = \text{Apricot} \}$$

测试工程师需要知道一个测试集有多好，所以我们用一个覆盖标准来度量测试集。

定义1.22 覆盖：给定一个覆盖标准 C 和相关的测试需求集合 TR ，要使一个测试集合 T 满足 C ，当且仅当对于测试需求集合 TR 中的每一条测试需求 tr ，在 T 中至少存在一个测试 t 可以满足 tr 。

还是回到上一个例子，一个测试集合 T 为12颗软心豆粒糖：3个柠檬口味，1个开心果口味，2个哈密瓜口味，1个梨子口味，1个橘子口味，4个杏口味，满足我们的“口味标准”。注意，用多于一个测试去满足一个给定的测试需求是可以接受的。

覆盖的重要性体现在两个方面。首先，有时要满足一个覆盖标准的代价是很大的，因此我们希望通过达到一定的覆盖程度来解决这个问题。

定义1.23 覆盖程度：给定一个测试需求集合 TR 和一个测试集合 T ，覆盖程度就是 T 满足的测试需求数占 TR 总数的比例。

第二，而且更重要的是，某些测试需求是不能满足的。假设橘子口味的软心豆粒糖很少，某些袋子中可能没几颗或者根本找不到。在这种情况下，“口味标准”就不能100%地满足，那么最大覆盖程度很可能是5/6或83%。因此，从 TR 集合中放弃那些不满足的测试需求，或者用不太严格的测试需求去取代那些不满足的测试需求。

那些不能得到满足的测试需求称为不可行的(infeasible)。规范地说，不存在这样的测试用例值可以满足不可行测试需求。在本书中，会有一些针对特定软件标准的实例，但是这些实例可能对大家来说已经很熟悉了。死码(dead code)将导致不可行测试需求，因为有些语句根本不能被执行到。对于大多数覆盖标准来说，检测不可行测试需求形式上是不可判定的，虽然有很多研究人员试图去找出一一些局部的解决方案，但成效甚微。因此，100%的覆盖率在实际中是不可能存在的。

通常会选择两种方式之一来使用覆盖标准。第一种方式是直接生成测试用例值去满足给定的标准。这通常是研究社区所采取的方法，是使用标准的最显而易见的方法。但在某些情况下，尤其是当我们没有足够自动化的工具来支持生成测试用例值时，这种方法的实现是很困难的。另一种方式是外部地生成测试用例值（例如手动或使用伪随机工具），然后根据覆盖标准来度量测试。业界实践者们通常喜欢使用这种方法，因为生成直接满足覆盖标准的测试难度很大。不幸的是，这种方法有时存在误导。如果我们的测试不能达到100%的覆盖率，将意味着什么呢？我们无从知晓99%的测试覆盖率到底比100%的测试覆盖率差多少，或者90%，甚至是75%。因为使用这种覆盖标准去评价现有的测试集合，覆盖标准通常也会被称为度量(metric)。

这种区别实际上是有坚固的理论基础的。生成器(generator)是一个程序，它可以自动

生成满足标准的值, 识别器 (recognizer) 是一个程序, 它可以判定给定测试用例值集合是否满足标准。理论上来说, 可证明对于大多数标准以上两个问题是不可判定的。然而在实际中, 识别测试用例是否满足标准常常比生成满足某标准的测试用例更有可能。识别的主要问题就在于不可行测试需求, 如果不存在不可行测试需求, 那么问题就容易解决了。

在商用自动化测试工具中, 生成器相当于一种可以自动创建测试用例值的工具, 识别器是一个覆盖分析工具。已经存在很多覆盖分析工具, 既有商用产品, 也有免费软件。

意识到 TR 集合是依赖于特定的被测工件很重要。在上面的那个软心豆粒糖的例子中, 测试需求 $color = Purple$ 是没有意义的, 因为我们假定工厂不会生产紫色的糖豆。在软件环境中, 考虑语句覆盖的情况。测试需求“执行语句42”只有当被测程序真的包含语句42时才有意义。考虑这个问题的一个好的方式是测试工程师有一个给定的软件工件, 然后来选择一个特定的覆盖标准。结合工件和标准来生成特定的测试需求集合 TR , 这个需求集合与测试工程师的任务相关联。

覆盖标准之间常常是相互关联的, 根据包含关系来进行比较。就拿“口味标准”来说, 要求每一种口味都要被尝试一次。我们也可以定义一种“颜色标准”, 要求我们尝试每一种颜色的软心豆粒糖 {黄色, 绿色, 橙色, 白色}。如果满足了口味标准, 那也就满足了颜色标准。这就是包含关系的本质, 即满足一种标准可以保证也满足另一种。

定义1.24 标准包含: 覆盖标准 C_1 包含 C_2 , 当且仅当满足 C_1 的每一个测试集合都满足 C_2 。

注意, 这个定义强调的是每一个测试集合, 而不是某些测试集合。包含与集合和子集的关系具有很强的相似性, 但是也不完全相同。通常来说, 一个标准 C_1 可以通过两种方式去包含另一个标准 C_2 。简单的方式是如果 C_1 的测试需求总能形成一个 C_2 测试需求的超集。例如, 另一个软心豆粒糖标准可能是这样定义的, 要求所有的口味名称以字母“C”开头。这将会产生一个测试需求{哈密瓜口味}, 它是口味标准{柠檬口味, 开心果口味, 哈密瓜口味, 梨子口味, 橘子口味, 杏口味}的一个子集。因此, “口味标准”包含“字母C开头”标准。

“口味标准”和“颜色标准”之间的关系反映了另一种包含方式。因为每一种口味都有一种特定的颜色, 而每一种颜色至少由一种口味所表示, 如果我们满足了“口味标准”, 我们也会满足“颜色标准”。一般来说, 一个多对一的映射存在于“口味标准”的需求和“颜色标准”的需求之间。因此, “口味标准”包含“颜色标准”。(如果一个一对一映射存在于两个标准的需求之间, 那么它们相互包含。)

对于一个软件的实例, 要考虑分支和语句覆盖。(这两种覆盖大家可能已经很熟悉了, 至少直观地说, 在第2章中将正式给出它们的定义。) 如果一个测试集合覆盖了一个程序中的所有分支 (满足分支覆盖), 那么这个测试集合也可以保证覆盖到每一条语句。因此, 分支覆盖标准包含了语句覆盖标准。在随后的章节中我们将用更严格方式、更多的例子来讲述包含。

1.3.1 不可行性与包含

在不可行性与包含之间存在着一种微妙的关系。特别地, 当且仅当所有的测试需求都是可行的, 有时标准 C_1 包含另一个标准 C_2 。如果 C_1 的某些测试需求是不可行的, C_1 将不包含 C_2 。

不可行的测试需求很常见, 而且会很自然地出现。假设我们把糖豆分为水果类型和干果

类型^①。现在,考虑“相互作用标准”,即每一种口味的糖豆与同一类型的其他口味的糖豆一起采样。在我们关注于特征之间的相互作用的情况下,软件领域中这样的标准有一个非常有用的对比物。例如,我们可能混合尝试柠檬口味与梨子口味或柠檬口味与橘子口味,但我们不会尝试单独的柠檬口味,或者混合尝试柠檬口味与开心果口味。我们或许认为“相互作用标准”包含了“口味标准”,因为每一种口味都与其他某种口味一同尝试。然而,在我们的例子中,开心果是干果类型的唯一成员,因此混合尝试开心果和干果类型中的其他口味的测试需求是不可行的。

重新建立包含的一种可能的策略是用相关的“口味标准”的测试需求取代每个不可行的“相互作用标准”的测试需求。在这个例子中,我们仅单独尝试开心果。一般说来,我们期望可以定义覆盖标准,使它们足够强壮,以便能应付出现不可行的测试需求时所涉及的包含的问题。在测试文献中通常不这么做,但是我们会在本书中尝试着去做。

也就是说,这个问题主要是理论方面的,而不应当过度地涉及实际的测试者。理论上讲,有些时候覆盖标准 C_1 包含另一个 C_2 ,那是因为我们假设 C_1 中不存在不可行的测试需求,但是如果 C_1 对于一个程序确实生成了一个不可行的测试需求,跳过那些不可行的测试需求来满足 C_1 的一个测试套件(test suite)也可能会跳过了可以满足的 C_2 的一些测试需求。实际上,对于任何给定程序, C_1 的不可行测试需求只是一小部分,如果某些测试需求是不可行的,那么相关的 C_2 的测试需求也将是不可行的。如果不是这样,丢失少数的测试用例可能不会导致不同的测试结果。

1.3.2 好的覆盖标准的特征

根据上面的讨论,一个有趣的问题是“什么使得一个覆盖标准成为好的覆盖标准”?当然,这个问题没有明确的答案。事实上可以部分地解释为什么有如此多的覆盖标准。然而,下面三个重要问题会影响到覆盖标准的使用:

1. 处理测试需求的困难。
2. 生成测试的困难。
3. 测试揭示故障的能力。

包含最多是一种非常粗糙比较标准的方式。直觉告诉我们,如果一个标准包含另一个,那么它就会揭示更多的错误。然而,不存在理论保证,而实验研究通常还不能令人信服并且远远没有完成。研究社区已对一些标准之间的关系达成广泛共识。处理测试需求的困难将取决于使用的工件及覆盖标准。生成测试的困难将直接影响到测试揭示错误的能力,这个事实并不使人感到惊讶。一个软件测试人员必须努力为被测软件权衡、选择合适的成本效益的软件测试标准。

1.3节练习

1. 假设覆盖标准 C_1 包含覆盖标准 C_2 。进一步,假设在程序 P 上测试集合 T_1 满足 C_1 和在 P 上测试集合 T_2 满足 C_2 。
 - (a) T_1 一定满足 C_2 吗?请解释。

^① 读者可能会问我们是否需要“另一个”类别来保证我们有一个划分。在我们的例子中,是没有问题的,但一般而言,我们需要这样一个类别来处理软心豆粒糖,例如土豆味、菠菜味等。

(b) T_2 一定满足 C_1 吗?请解释。

(c) 如果 P 中有错误, T_2 找出这个错误, T_1 却不一定也能找出这个错误。请解释。[⊖]

2. 除了包含, 我们还能如何用去比较测试标准?

1.4 以往的软件测试术语

在过去的20年里, 测试研究社区一直都非常活跃。关于应该测试什么以及如何测试, 我们的一些基本观点都已发生了变化。本节介绍的多年来使用的一些术语, 它们因为各种原因已变得过时了。尽管这些术语已经不再像以往那么相关了, 这些术语仍然被使用, 并且对于测试专业的学生和专业人士来说, 熟知这些术语是很重要的。

从抽象的角度来看, 黑盒和白盒测试非常相似。特别的是在这本书中, 我们把测试表述为来自软件的抽象模型, 例如各种图, 可以容易地从黑盒角度或白盒角度中获取它们。因此, 本书独特的哲学结构的一个最显著的结果就是这两个术语均已变得过时。

定义1.25 黑盒测试: 从软件的外部描述中生成测试, 包括规格说明书、需求和软件设计。

定义1.26 白盒测试: 从软件的内部源代码中生成测试, 特别是包括分支、单个条件和语句。

早在20世纪80年代初, 就有关于测试应该自上而下还是应该自下而上的讨论。这是对以前讨论有关如何开发软件的一个写照。但是, 当我们开始意识到自上而下的测试是不切实际的, 这种区别就消失了, 既而面向对象的设计让这种区别成为过去。下面的定义假设软件可以被看做是一个树状的软件程序, 边表示调用, 树根可以看做是主程序。

定义1.27 自上而下测试: 测试主程序, 然后是主程序向下调用的过程, 依次类推。

定义1.28 自下而上测试: 先测试树叶 (不调用任何过程的过程), 然后向树根推进。

当且仅当一个过程所调用的子过程全部被测试后才去测试这个过程。

面向对象的软件带来了一个更普遍的问题。类之间的关系将被表示为带有循环的一般图形 (general graphs with cycles), 这就要求测试工程师做出以什么顺序去测试类的艰难抉择。我们会在第6章中讨论这个问题。

一些文献中将静态测试与动态测试区分如下:

定义1.29 静态测试: 无须执行程序的测试。包括软件的检查和某些形式的分析。

定义1.30 动态测试: 用实际的输入去执行程序, 进行测试。

大多数的文献中用“测试”表示动态测试, 而“静态测试”被称为“验证活动”。在本书中我们将使用这种叫法, 并应指出本书只涉及动态或以程序执行为基础的测试。

由于缺乏相关定义, 最后一点要被提到的是, 测试策略可以代表很多概念, 包括覆盖标准、测试过程和使用的测试技术。我们将尽量避免使用测试策略这个词。

[⊖] 正确回答这个问题对于理解包含关系的弱点有很大的帮助。

1.5 参考文献注释

对所有软件测试的书籍和研究人员有里程碑意义的书籍分别为1979年Myers [249], 1990年Beizer [29]和2000年Binder [33]所著。一些优秀的概述单元测试标准的观点也已问世, 例如由White [349]和最近由Zhu、Hall和May [367]提出的一些观点。软件测试占软件开发成本的50%的观点由Myers和Sommerville [249, 316]提出。而最近Pezze和Young[289]从一些测试文献中报告了测试相关的过程、原理和技术, 包括许多实用的课堂资料。Pezze和Young的著作中以传统的基于生命周期的方式描述了覆盖标准, 没有把覆盖标准归入本章所提到的四种抽象模型中去。

其他许多软件测试的书籍并不打算作为教科书使用, 或者不用于课上使用。Beizer的《软件系统测试和质量保证》[28]以及Hetzel的《软件测试完全指南》[160]涵盖了软件的测试管理和过程的各个不同方面。有几本涉及测试的某些特定方面[169, 227, 301]。佐治亚理工学院的STEP项目对20世纪80年代由美国国防部承包人的软件测试活动做了全面的调查[100]。

单元的定义是由Stevens、Myers和Constantine [318]提出的。模块的定义则是由Sommerville [316]提出的。集成测试的概念是Beizer[29]提出的。OO测试级别的分类, 例如方法内、方法间和类内测试由Harrold和Rothermel [152]提出, 类间测试由Gallagher、Offutt和Cincotta [132]提出。

有关奔腾错误和火星登陆者的信息来源有若干个, 包括Edelman、Moler、Nuseibeh、Knutson和Peterson [111, 189, 244, 259, 286]。了解Ariane 5型飞行器501错误细节的最好来源是事故报告[209]。

1.1.2节中测试级别的定义最初来自于Beizer [29]。

有关“在一个程序中发现所有错误是不可判定的”基本成果归于Howden [165]。

大部分测试术语来源于标准文档, 包括IEEE软件工程术语的标准词汇 (IEEE Standard Glossary of Software Engineering Terminology) [175], 美国国防部[260, 261], 美国联邦航空管理局FAA-DO178B, 以及英国计算机学会的软件组件测试标准 [317]。可观察性和可控性的定义来源于Freedman[129]。类似的定义也在Binder的书 (《Testing Object-Oriented Systems》) 中出现[33]。

错误故障/失败模型分别由Offutt和Morell独立开发并写入博士论文[101, 246, 247, 262]。Morell使用了术语执行 (execution)、影响 (infection) 和传播 (propagation) [246, 247], 而Offutt使用了可达性 (reachability)、充分性 (sufficiency) 和必要性 (necessity) [101, 262]。为了更具备描述性, 本书融合了这两套术语。

我们使用的测试用例的若干部分是基于对测试用例规格说明的研究基础上的[23, 319]。

最先以非纯粹理论的形式讨论不可行性的两个人是Frankl和Weyuker[128]。这个问题被Goldberg等人[136]、DeMillo以及Offutt [101]证明为不可判定的。部分结论被描述在[132, 136, 177, 273]。

Budd和Angluin[51]从一个测试的观点分析了生成器和识别器的理论区别。他们证明了二者都是不可判定的, 并讨论了接近二者的折衷办法。

包含作为一种分析比较测试技术的方式得到了广泛的应用。我们遵循Weiss[340]、Frankl和Weyuker[128]提出的包含的定义。Frankl和Weyuker实际上使用了术语包括 (include)。这

个术语被Clarke等人这样定义：一个标准 C_1 包含一个标准 C_2 ，当且仅当每一个满足 C_1 的执行路径集合 P 同时满足 C_2 [81]。包含这个术语目前更广泛地被使用，而且两个定义等价。这本书遵循Weiss的建议，使用术语包含(subsume)来表示Frankl和Weyuker的定义。

Cooper[89]描述了消费任务和收入任务。

虽然本书的侧重点不在于软件测试的理论基础，但对研究感兴趣的学生应该对这些课题更深入地学习。有些这方面的文献年代已经比较久远了，通常在现今的文献中不会出现，并且其中所提出的思想也开始消失。但作者鼓励对那些老的文献的研究。例如20世纪70年代由Goodenough和Gerhart [138], Howden [165], Demillo、Lipton、Sayward和Perlis [98, 99]所著的开创性文章。Weyuker和Ostrand [343], Hamlet [147], Budd和Angluin [51], Gourlay [139], Prather [293], Howden [168]以及Cherniavsky和Smith [67]沿用和改进了这些文章。此后一些理论性的文章由Morell [247]、Zhu [366]和Wah [335, 336]继续发表。每一个博士生的导师肯定有他或她自己最喜爱的论文，但这个清单可以提供一个很好的起点。

第二部分 覆盖标准

第2章 图 覆 盖

本章主要介绍现今使用的主流测试覆盖标准。虽然本章以一个非常理论的方式开始，但是对图和图覆盖的理论方面知识的牢固掌握将使后面的内容变得更为容易理解。我们首先强调一个普通的图，而不去考虑图的来源。在这样一个模型建立起来之后，本章其余部分的内容将转向实际应用，讲述怎样从各种各样的软件工件中获取图以及怎样把一些通用的覆盖标准用到这些图上。

2.1 概述

有向图为许多覆盖标准建立了基础。给定一个被测工件，一个想法是获得这个工件的图形式的抽象。例如，源代码的最常见的图形式的抽象是把代码映射到一个控制流图。需要重点理解的是，抽象出来的图和工件本身是不完全相同的。实际上，工件通常有若干有用的但是差别较大的图形式的抽象。从工件中获得的图的抽象还可以将工件的测试用例映射到图中的路径。因此，基于图的覆盖标准评估工件测试集的优劣是看测试用例对应的路径对该工件图的抽象中边的“覆盖”程度。

下面我们给出图的基本概念，在后面的章节里根据需要增加更多的结构。一个图 G 可以形式化地定义为：

- 节点的集合 N
- 初始节点的集合 N_0 ，其中 $N_0 \subseteq N$
- 终止节点的集合 N_f ，其中 $N_f \subseteq N$
- 边的集合 E ，其中 E 是 $N \times N$ 的一个子集

要使一个图对生成测试有意义，那么 N 、 N_0 和 N_f 每个必须至少包含一个节点。有些时候，仅仅考虑图的一部分是有所帮助的。子图也是一个图，以 N 的一个子集和与其相对应的 N_0 、 N_f 和 E 的子集来定义。特别是，如果 N_{sub} 是 N 的一个子集，那么由 N_{sub} 所定义的子图中，初始节点的集合为 $N_{sub} \cap N_0$ ，终止节点的集合为 $N_{sub} \cap N_f$ ，边的集合为 $(N_{sub} \times N_{sub}) \cap E$ 。

需要注意的是，初始节点可以有不止一个，也就是说， N_0 是一个集合。拥有多个初始节点对于一些软件工件是必要的，例如，如果一个类有多个进入点，但是有的时候我们也会限制一个图只能包含一个初始节点。图中的边可以看做是从一个节点到另一个节点的连线，表示为 (n_i, n_j) 。边的初始节点 n_i 有时也称做前驱节点， n_j 有时也称做后继节点。

我们通常需要确定终止节点，并且每个图必须至少有一个终止节点，理由是每一个测试都要开始于某个初始节点并且结束于某个终止节点。终止节点的概念也依赖于图所表示的软件工件的种类。某些测试标准要求测试在某一个特定的终止节点结束。另外一些测试标准允

许任意一个节点作为终止节点,在这种情况下集合 N_f 和集合 N 是相同的。

术语节点有很多同义词。在图论的相关课本中有时称节点为顶点,在测试的相关课本中一般用它表示的结构来确定一个节点,这个结构通常指一个语句或者一个基础块。类似地,图论的相关课本中有时称边为弧,在测试的相关课本中一般用它表示的结构来确定一条边,这个结构通常指一个分支语句。本节使用一般的方法来讨论图标准,所以我们使用一些通用的图的术语。

我们通常使用圆圈和箭头来画图。图2.1表示了三种图例:有入边但没有前驱节点的节点为初始节点,圆圈加粗的节点为终止节点。图2.1a有一个单独的初始节点并且不存在环。图2.1b有三个初始节点,同时存在一个环($\{n_1, n_4, n_8, n_5, n_1\}$)。图2.1c没有初始节点,所以它对生成测试用例是无用的。

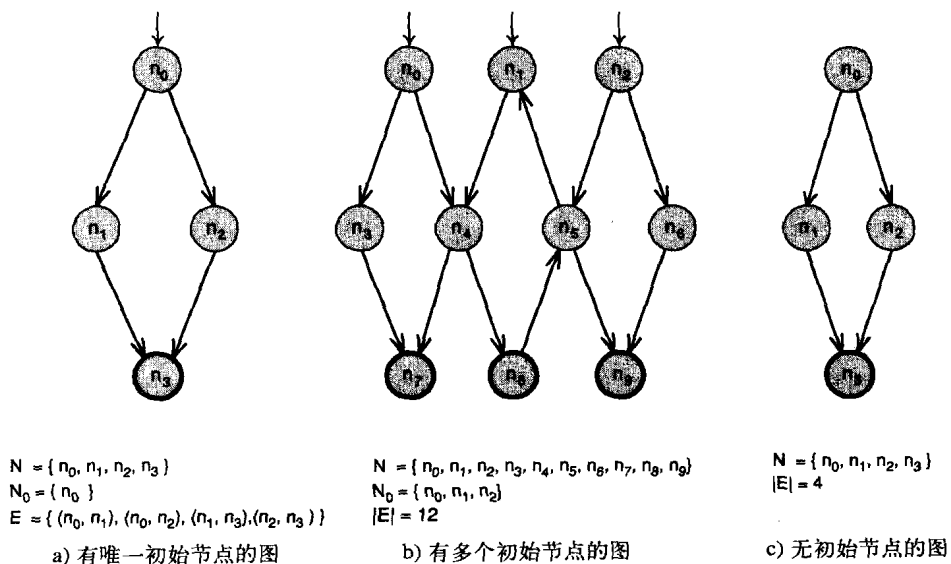


图 2.1 图a有唯一初始节点,图b有多个初始节点,图c无初始节点

路径是一个节点序列 $[n_1, n_2, \dots, n_M]$,其中的每一对相邻节点 (n_i, n_{i+1}) 的连线存在于边的集合 E 中 $1 \leq i < M$ 。路径的长度定义为它包含的边的数目。我们有时会考虑长度为0的路径和子路径。路径 p 的子路径是 p 的一个子序列(也可能是 p 本身)。根据边的概念,我们所说的路径是从这个路径的第一个节点到这个路径的最后一个节点。我们也可以说一条路径从边 e 开始(或者到达边 e),简单地表示 e 是这条路径的第一条(或者最后一条)边。

图2.2给出了一个图的一些路径的例子和一些非路径的例子。例如,序列 $[n_0, n_7]$ 不是一个路径,因为这两个节点没有被某条边连接起来。

许多测试标准都需要从一个节点开始并且在另一个节点结束的输入。然而这个要求只有在这些节点被某一个路径覆盖时才是可行的。当我们在一些特定的图上应用这些标准时,有时会发现我们要求的某些路径由于某种原因是不可能被执行的。例如,在程序运行的实际结果是某个循环会至少被执行一次的情况下,一条路径却可以要求一个循环被执行0次。这种类型的问题是基于一图所表示的软件工件的语义。现在,我们强调一下,我们仅仅按照图的语法进行寻找。

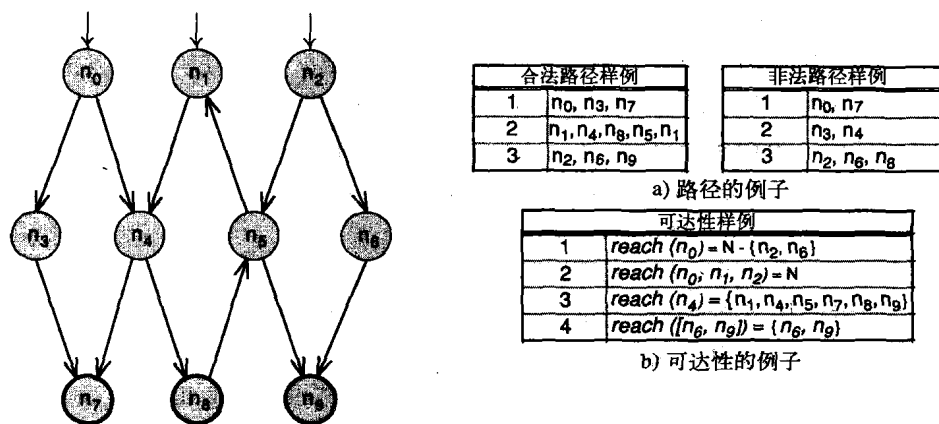


图2.2 路径举例

如果存在从节点 n_i 到节点 n (或者边 e) 的一条路径, 我们就说一个节点 n (或者一条边 e) 是从节点 n_i 在语法上可达的。如果至少有一条带有某个输入的路径是可以执行的, 那么这个节点 n (或者边 e) 也叫做语义上可达的。我们可以定义一个函数 $reach_G(x)$ 表示从参数 x 语法上可达的图中的一部分。 $reach_G()$ 的参数可以为一个节点、一条边或者一个节点或边的集合。这时 $reach_G(n_i)$ 表示从节点 n_i 在语法上可达的 G 的子图; $reach_G(N_0)$ 表示从任一初始节点在语法上可达的 G 的子图; $reach_G(e)$ 表示从边 e 在语法上可达的 G 的子图; 等等。在我们的使用中规定 $reach_G()$ 包含开始节点。例如, $reach_G(n_i)$ 和 $reach_G([n_i, n_j])$ 都包括节点 n_i , $reach_G([n_i, n_j])$ 还包含边 $([n_i, n_j])$ 。有一些图包含某些节点或者起始边在语法上不能被任一初始节点 N_0 所到达, 这样的图无法满足覆盖标准, 所以, 我们一般把注意力限定在 $reach_G(N_0)$ 上。^①

考虑图2.2中的例子, 从节点 n_0 可能到达除 n_2 和 n_6 之外的所有节点。从整个初始节点的集合 $\{n_0, n_1, n_2\}$ 可能到达图中的所有节点。如果我们从节点 n_4 开始, 它可能到达除节点 n_0, n_2, n_3 和 n_6 之外的所有节点。如果我们从边 (n_6, n_9) 开始, 它仅可能到达节点 n_6, n_9 和边 (n_6, n_9) 。另外, 一些图 (例如有有限状态机) 会有从一个节点到其自身的显式的边, 即 (n_i, n_i) 。

通常在标准的数据结构课本中给出的基础图算法是可以用来计算语法上的可达性的。

一条测试路径代表了一个测试用例的执行。测试路径一定要开始于 N_0 是因为测试用例总是开始于一个初始节点。需要注意的是, 一条单独的测试路径可能对应于软件上大量的测试用例。当然, 如果一条测试路径是不可行的, 它也有可能对应于0个测试用例。我们将在后面的2.2.1节中讨论关键而又理论化的不可行性问题。

定义2.31 测试路径: 长度允许为0的一条路径 p , 它起始于 N_0 中的某个节点, 终止于 N_f 中的某个节点。

在某些图中, 所有的测试路径均开始于一个节点并且终止于一个单一节点。我们把这样的图称做单入/单出或者SESE图。对于SESE图, 集合 N_0 有且仅有一个节点 n_0 , 集合 N_f 有且仅有一个节点 n_f , 这里, n_f 和 n_0 可以是同一个节点。我们需要 N 中的每一个节点在语法上可到达 n_f , 同时, n_f 不能在语法上可到达 N 中除了 n_f 本身外的任一节点 (除非 n_0 和 n_f 是同一节点)。换句话说

① 通过例子说明, 典型的控制流图几乎没有语法上无法达到的节点, 但是调用图, 特别是面向对象程序的调用图就经常存在这样的节点。

说,除非 n_0 和 n_f 是同一个节点,否则没有从 n_f 开始的边。

图2.3是SESE图的一个例子。这种特殊的结构有时也称做“双菱形”图,它对应于两个if-then-else语句的连续序列控制流图。初始节点 n_0 用一个进入的箭头来标出(记住我们只有一个初始节点),终止节点 n_6 用一个加粗的圆圈来标出。在这个双菱形图中正好存在四条测试路径: $[n_0, n_1, n_3, n_4, n_6]$, $[n_0, n_1, n_3, n_5, n_6]$, $[n_0, n_2, n_3, n_4, n_6]$ 和 $[n_0, n_2, n_3, n_5, n_6]$ 。

我们需要一些专业术语来表达在测试路径中出现的节点、边和子路径的概念,因此我们从与旅游相关的词汇中选择了一些熟悉的术语。如果节点 n 在一个测试路径 p 中,我们称 p 访问了节点 n 。如果边 e 在一个测试路径 p 中,我们称 p 访问了边 e 。术语“访问”对于单独的节点和边是非常有效的,但是有的时候我们还想把注意力转向子路径。对于子路径,我们使用术语“游历”(tour)。如果路径 q 是路径 p 的一条子路径,我们称测试路径 p 游历了子路径 q 。图2.3的第一条路径 $[n_0, n_1, n_3, n_4, n_6]$ 访问了节点 n_0 和 n_1 ,访问了边 (n_0, n_1) 和边 (n_3, n_4) ,游历了子路径 $[n_1, n_3, n_4]$ (这些列表并不完全,只是其中一部分)。因为子图的关系是自反的,所以游历的关系也是自反的。也就是说,任一给定的路径 p 总是游历其自身。

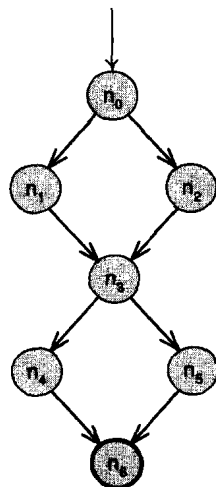


图2.3 单入单出图

我们为测试定义一个映射 $path_G$,所以对于一个测试用例 t , $path_G(t)$ 表示图 G 中被 t 执行的测试路径。因为我们讨论的图通常都是显而易见的,所以可以省略掉标记 G 。我们也可以对某一测试集合游历的所有路径的集合进行定义。对于一个测试集合 T , $path(T)$ 表示被 T 中的测试所执行的测试路径的集合,即 $path_G(T) = \{path_G(t) | t \in T\}$ 。

除了非确定性结构(直到第7章我们才会考虑),每一个测试用例都将游历图 G 中的一条测试路径。图2.4举例说明了关于测试用例/测试路径映射到有确定性结构的软件和有非确定性结构的软件上的区别。

图2.5举例说明了一个测试用例的集合和它在SESE图中相对应的测试路径,图中的终止节点 $n_f = n_2$ 。图中的一些边使用谓词进行了注释,这些谓词描述了一条边在什么样的情况下会被遍历。(这个概念将在本章的后面部分正式定义。)所以在这个例子中,如果 a 小于 b ,唯一的途径就是从 n_0 到 n_1 ,然后再到 n_3 ,最后到 n_2 。本书中所有对于图的覆盖标准的描述都是按照测试路径和我们所考虑的图的关系进行的。需要注意的是,测试是通过测试用例执行的,测试路径仅仅是一个模型,它是由图所捕捉到的测试用例的抽象。

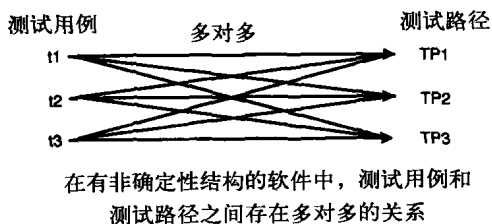
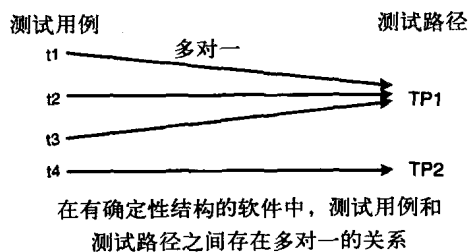
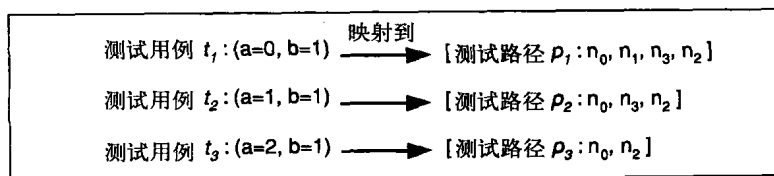
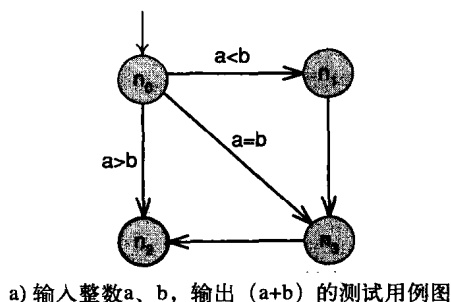


图2.4 测试用例到测试路径的映射



b) 测试用例和测试路径间的映射

图2.5 测试用例集合和对应的测试路径

2.1节练习

1. 给出图2.2中集合 N 、 N_0 、 N_f 、 E 的内容。
2. 给出图2.2中的一条路径，使其不是测试路径。
3. 列出图2.2中的所有测试路径。
4. 在图2.5中找出恰当的测试用例输入，使得该测试用例对应的测试路径访问边 (n_1, n_3) 。

2.2 图覆盖标准

2.1节的叙述适合定义图的覆盖。在测试文献中我们通常将这些标准分为两类，一类是我们常说的控制流覆盖标准，这里将其一般化，称为结构化图覆盖标准。另一类标准是基于图所表示的软件工件中的数据流，称为数据流覆盖标准。根据第1章的讨论，我们首先确定适当的测试需求，然后根据这些测试需求定义每个标准。一般而言，对于任何基于图的覆盖标准，我们的想法是按照图中的不同结构来确定测试需求。

对于图来说，依照图 G 中测试路径的属性，覆盖标准定义了测试需求（ TR ）。一个典型的测试需求是通过访问一个特定的节点或边，或者游历一条特定的路径来满足的。目前为止我们对于访问的定义是充分的，但是对于游历的定义需要进一步延伸。后文我们将进一步讨论游历的含义并将其在数据流标准的背景下细化。下面的定义是对第1章中给出的覆盖定义的细化：

定义2.32 图覆盖：给定一个图 G 上满足覆盖标准 C 的测试需求集合 TR ，当且仅当对于 TR 中的每一个测试需求 tr ，在测试路径集合 $path(T)$ 中都至少存在一条测试路径 p 满足 tr ，我们说测试集合 T 满足图 G 上的覆盖标准 C 。

上述定义非常笼统，对于各个特定情况需要给出更加细化的定义。

2.2.1 结构化覆盖标准

我们通过详细说明一个测试需求的集合 TR 来详细说明图覆盖标准。我们从把标准定义为访问图中的每一个节点开始，然后将标准上升到访问图中的每一条边。第一个标准可能很熟悉，它是基于原先执行程序中的每一条语句的概念。这个概念有多种叫法：“语句覆盖”、“语句块覆盖”、“状态覆盖”和“节点覆盖”。我们使用通用的图术语“节点覆盖”。虽然这个概念很熟悉也很简单，我们还是介绍一些附加符号。符号最初好像使标准复杂化，但最终可以起到使后续标准更加简洁，数学上精确的作用，避免与更复杂情况相混淆。

由图标准生成的需求是技术上的谓词，可以取值为真（需求被满足），也可以取值为假（需求未被满足）。在图2.3的双菱形图中，对于节点覆盖的测试需求是： $TR=\{\text{访问}n_0, \text{访问}n_1, \text{访问}n_2, \text{访问}n_3, \text{访问}n_4, \text{访问}n_5, \text{访问}n_6\}$ ，这样我们必须使每个节点满足谓词，这个谓词是确认该节点是否被访问过。综上所述，节点覆盖的正式定义如下^①：

定义2.33 节点覆盖（形式化定义）：对于每一个节点 $n \in \text{reach}_G(N_0)$ ， TR 包含谓词“访问 n ”。

上述定义虽然在数学上很精确，但是在实际使用中很繁琐。因此我们选择介绍一种更简单的定义，它对测试需求中的判定问题进行了抽象。

标准2.1 节点覆盖（NC）： TR 包含 G 中每个可达到的节点。

在这个定义中，我们必须理解术语“包含”实际指的是“包含访问 n 的判定”。这种简化可以使我们将图2.3中测试需求的写法也简化到只包含节点： $TR=\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$ 。测试路径 $p_1=[n_0, n_1, n_3, n_4, n_6]$ 覆盖了第一、第二、第四、第五和第七个测试需求，而测试路径 $p_2=[n_0, n_2, n_3, n_5, n_6]$ 覆盖了第一、第三、第四、第六和第七个测试需求。因此，如果一个测试集合 T 包含 $\{t_1, t_2\}$ ，且 $\text{path}(t_1)=p_1$ ， $\text{path}(t_2)=p_2$ ，那么 T 满足 G 上的节点覆盖。

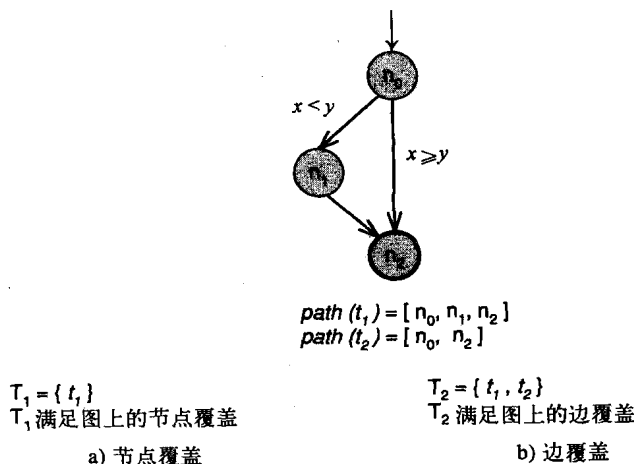


图2.6 展示节点覆盖和边覆盖的图

对节点覆盖通常的定义省略了明确确定测试需求的中间步骤，经常被表示成下面给出的

① 我们的数学家读者可能会注意到这种定义是具有构造性的，因为它定义了什么在 TR 集合中，而不限制这个集合。这正是我们的意图， TR 集合并不包含其他任何元素。

形式。注意上述使用的定义形式相对标准定义而言的节约性。一些练习通过指导学生对其他标准以标准的形式进行重新定义来强化这一点。

定义2.34 节点覆盖 (NC) (标准定义): 测试集合 T 满足图 G 上的节点覆盖, 当且仅当对于 N 中每一个语法上可到达的节点 n , 存在某条 $path(T)$ 中的路径 p , 使得路径 p 访问节点 n 。

本节末尾的练习让读者对其余一些覆盖标准同时使用形式化的方法和标准的方法进行重新构造定义。我们选择中间的定义是因为它更加简洁紧凑, 避免了在标准覆盖定义中额外的空话, 把关注点集中在标准间变化的覆盖定义的部分。

节点覆盖已经被许多商业化的测试工具所实现, 最常见的形式为语句覆盖。对于下面提出的非常流行的边覆盖标准也是如此, 测试工具更多地将其实现为分支覆盖:

标准2.2 边覆盖 (EC): TR 包含图 G 中每一个可到达的长度小于等于1的路径。

读者可能会好奇为什么边覆盖的测试需求同时显式地包含了节点覆盖的测试需求, 也就是为什么“小于等于”会包含在定义中。事实上, 所有的图覆盖标准都是类似定义的。这样做的目的是为了包含那些结构不是很复杂的图。例如, 考虑一个含有一个节点而没有边的图。如果在定义中不包含“小于等于”, 边覆盖标准就不可能覆盖到这个节点。然而直觉上我们更希望边覆盖的测试至少可以做到节点覆盖的程度。这种风格的定义是达到这个特性的最好方法。为了使我们的集合 TR 更加可读, 我们仅仅列出最长的路径。

图2.6说明了节点覆盖和边覆盖的不同之处。如果使用编程的术语描述, 这是一个常见的“if-else”结构的图。

其他一些覆盖标准也仅仅使用了到目前为止介绍的关于图的相关定义。例如, 一个需求要求每一条长度小于等于2的路径都被某一条测试路径游历。在这个上下文环境中, 节点覆盖可以被重新定义为包含每一条长度为零的路径。显然, 这种想法可以推广到任意长度的路径, 尽管可能减少返回值。我们在这里正式定义这些标准中的一个, 其余的留给对此感兴趣的读者作为练习。

标准2.3 边-对覆盖 (EPC): TR 包含图 G 中每一条可到达的长度小于等于2的路径。

一个实用的测试标准为软件从某一个状态开始 (即有穷状态自动机中的某一节点), 经过转换 (也就是边), 最终到达与初始状态相同的最终状态。这种类型的测试被用来验证一个系统没有被特定的测试输入改变。接着, 我们就会正式定义这种概念为往返覆盖。

在定义往返覆盖之前, 我们需要增加一些定义。在一条从 n_i 到 n_j 的路径中, 除了第一个节点和最后一个节点可能是同一节点的情况, 如果没有节点在路径中的出现多于一次, 我们称这条路径是简单的。也就是说, 虽然简单路径本身可以结束为一个环, 但是它没有内部的环。简单路径的一个非常有用的方面是: 任何路径都可以通过组合简单路径来创建。

即使是非常小的程序也可能有大量的简单路径。这些简单路径中的大部分是不值得被显式强调的, 因为它们是某些其他简单路径的子路径。对于一个简单路径的覆盖标准, 我们尽量避免罗列简单路径的整个集合。由于这个原因, 我们仅仅列出那些最长的简单路径。为了澄清这个概念, 我们给最长的简单路径一个正式的定义, 称之为主路径。我们在标准的定义中也使用了“主”这个词。

定义2.35 主路径：一条从 n_i 到 n_j 的路径，如果是一个简单路径且不会作为任何其他简单路径的固有子路径出现，我们称这条路径为主路径。

标准2.4 主路径覆盖 (PPC)： TR 包含图 G 中的每一条主路径。

虽然主路径覆盖的定义在减少测试需求的数量上有实用性的优点，但它有一个问题，即一个给定的不可行的主路径可能包含许多可行的简单路径。对这个问题的解决方法是显而易见的：使用相关的可行的子路径来替代不可行的主路径。根据本书的目标，我们选择不在主路径的正式定义中包含这个方面，但是在后面关于主路径覆盖的理论描述上我们会假设它的存在。

主路径覆盖有两种特殊的情况，由于历史原因，我们在下面会介绍它们。出于实用的想法，简单地使用主路径覆盖通常会比较好。主路径覆盖的两种特殊情况都包含了怎样处理“往返”环。

往返路径是一条起始、终止于同一节点的长度非0的主路径。一种类型的往返测试覆盖要求对于图中的每个节点，至少有一条往返路径覆盖它。另一种要求执行所有可能的往返路径。

标准2.5 简单往返覆盖 (SRTC)： TR 包含经过图 G 中每一个可到达节点的至少一条往返路径。

标准2.6 完全往返覆盖 (CRTC)： TR 包含经过图 G 中每一个可到达节点的所有往返路径。

接下来我们开始介绍路径覆盖，在测试文献中这是惯例部分。

标准2.7 完全路径覆盖 (CPC)： TR 包含图 G 中的所有路径。

不幸的是，完全路径覆盖在图中有环路的情况下是无用的，因为它的结果是一个无穷大数量的路径，从而导致无穷大数量的测试需求。然而这个标准的一个变体是有用的，假设我们不要求所有路径，而是考虑一个特定的路径集合。例如，客户按照使用场合给出的这些路径。

标准2.8 指定路径覆盖 (SPC)： TR 包含一个测试路径集合 S ，其中 S 以参数形式给出。

完全路径覆盖对于有环路的图是不可行的，所以才有了上面列出的其他一些选择。图2.7对比了主路径覆盖和完全路径覆盖。图2.7a显示了一个没有环路的“菱形”图。在这个图上，完全路径覆盖和主路径覆盖均可以通过列出的两条路径满足。然而，图2.7b包含一个从 n_1 到 n_3 到 n_4 再到 n_1 的一个环，这种情况下图中有无穷个可能的测试路径，所以完全路径覆盖是不可能的。然而主路径覆盖的需求可以被两条测试路径遍历，例如 $[n_0, n_1, n_2]$ 和 $[n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]$ 。

游历、侧访、绕路而行

有一个重要而细微的问题需要注意，那就是虽然简单路径没有内部循环，但是我们不要要求游历简单路径的测试路径也有这个性质。也就是说，指定一个测试需求的路径和满足一个需求的测试路径的部分是不同的。区分这两个概念的优势会在遇到不可行的测试需求时体现。在描述这种优势之前，让我们先完善游历的概念。

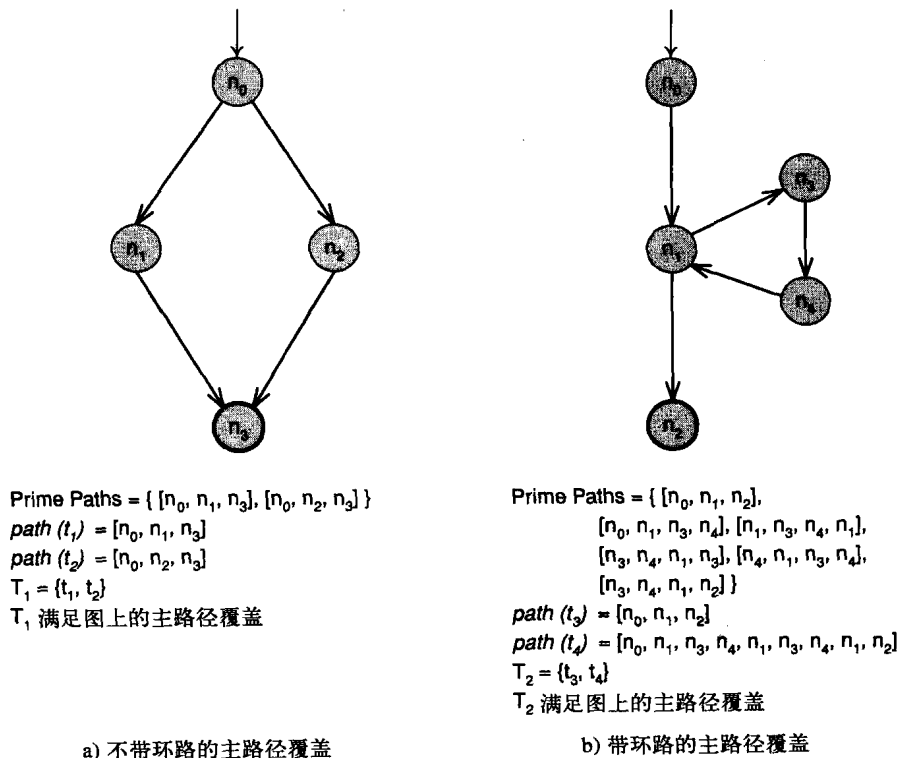


图2.7 显示主路径覆盖的两个图

我们之前定义了“访问”和“游历”的概念，用路径 p 游历子路径 $[n_1, n_2, n_3]$ 意味着这条子路径是 p 的一条子路径。这是一个很严格的定义，因为在这条子路径中的每一个节点和每一条边都必须严格按照它们在 p 的子路径中出现的顺序被访问。我们可以放约束，使得游历中可以包括循环。如图2.8所示， b 到 c 和 c 到 b 中就有一个小循环。

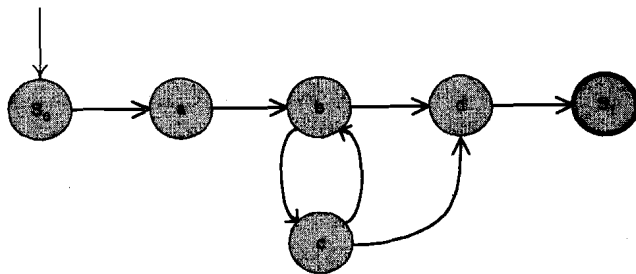


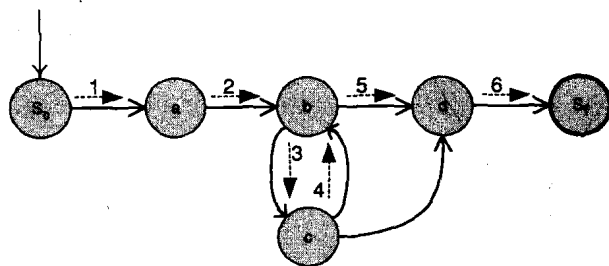
图2.8 有一个循环的图

如果我们需要游历子路径 $q=[a, b, d]$ ，严格的游历定义禁止使用任何包括节点 c 的路径来满足需求，比如 $p=[s_0, a, b, c, b, d, s_1]$ ，因为我们不能按照与子路径 q 中完全相同的顺序来访问 a, b 和 d 。我们用两种方式放宽对游历定义的约束。一个是允许游历包括“侧访”，即我们可以暂时从路径上的一个节点离开，然后再返回到这个节点。第二种方式是允许游历包括更通用的“绕道而行”，即我们可以从路径上的一个节点离开，然后返回到在路径上的下一个节点（跳过一条边）。在下面的定义中， q 是一个必需的假设为简单的子路径。

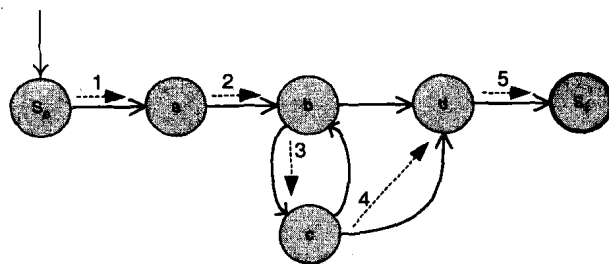
定义2.36 游历：当且仅当 q 是 p 的子路径时，测试路径 p 称为游历子路径 q 。

定义2.37 侧访的游历：当且仅当 q 中的每一条边在 p 中以相同的顺序出现时，测试路径 p 称为侧访的游历子路径 q 。

定义2.38 绕路而行的游历：当且仅当 q 中的每一个节点在 p 中以相同的顺序出现时，测试路径 p 称为绕路而行的游历子路径 q 。



a) 侧访的游历图



b) 绕路而行的游历图

图2.9 在图覆盖中的游历、侧访、绕路而行

图2.9说明了在图2.8基础上的侧访与绕路而行。在图2.9a中，虚线展示了在侧访的游历中边的执行顺序。虚线上的数字说明边是顺序执行的。在图2.9b中，虚线展示了在绕路而行的游历中边的执行顺序。

虽然这些差别很细微，但是由它们得出的结果有很大差别。侧访与绕路而行的不同可以从图2.9中看出。子路径 $[b, c, b]$ 对于 $[a, b, d]$ 是侧访，因为它在节点 b 上离开了子路径，然后在节点 b 回到了子路径。所以，子路径 $[a, b, d]$ 中的每一条边都以相同的顺序在该路径中被执行。子路径 $[b, c, d]$ 对于 $[a, b, d]$ 是绕路而行，因为它在节点 b 上离开了子路径，然后通过边 (b, d) 从子路径上 b 的下一个节点 d 回到了子路径。因此， $[a, b, d]$ 中的每一个节点都以相同的顺序在该路径中被执行，而每条边却不是。绕路而行有彻底改变测试行为的可能性，也就是说，一个经过边 (c, d) 的测试和一个经过边 (b, d) 的测试可能会展示完全不同的行为，而且测试程序的不同方面。

使用侧访和绕路而行的概念，你总可以选择到一种游历来“装饰”每个合适的图覆盖标准。例如，主路径覆盖使用游历的概念可以定义得很严格，允许使用侧访的概念定义则可放松一些，甚至还可以更不严格地允许使用绕路而行的概念来定义。

如后面将提到的，在本书中侧访是处理不可行测试需求的实用方法。因此在我们的标准里明确地包括它们。绕路而行似乎不太实用，所以我们没有深入探讨。

处理不可行的测试需求

如果不允许侧访,就会存在大量不可行的需求。再看看图2.9,在很多程序里,不可能存在不经过节点 c 的从 a 到 d 的路径,例如,已经写好的不能被跳过执行的循环体。如果这样的情况发生,我们需要允许侧访。也就是说,也许不通过侧访就不可能游历路径 $[a, b, d]$ 。

前面的讨论建议放弃严格的游历概念,使用侧访满足测试需求。然而,这并不总是个好主意。具体来说,如果一个测试需求可以不通过侧访满足,那么这样做是明显地优于通过侧访满足需求。再次考虑循环的例子,如果循环可以被执行0次,那么路径 $[a, b, d]$ 应该被游历而不用侧访。

前面的讨论使我们想到一个既有实用性的又有理论性的混合处理方案。这个想法是首先使用严格的游历满足测试需求,然后在不能满足的测试需求中允许侧访。很明显,这种论点可以很容易地扩展到绕路而行中,但是,如同前面提到的,我们选择不这样做。

定义2.39 最大努力的游历:假设 TR_{tour} 是可以被游历的测试需求的子集, $TR_{sidetrip}$ 是可以使用侧访游历的测试需求的子集。注意到 $TR_{tour} \subseteq TR_{sidetrip}$,如果对于 TR_{tour} 中的每一条路径 p ,存在 T 中的某条路径直接游历 p ;对于 $TR_{sidetrip}$ 中的每一条路径 p ,存在 T 中的某条路径直接游历 p 或者通过侧访游历 p ,这时,我们说测试路径的集合 T 达到最大努力的游历。

最大努力的游历有一个实用的优点,即可以尽量多地满足测试需求,且每一个测试需求也尽可能严格地满足。我们将会2.2.3节中讨论包含,最大努力的游历对于包含有所需要的理论属性。

寻找主测试路径

可以看到在图中找到所有主路径相对容易,而且游历主路径的测试路径可以用机械性的方式构造。考虑图2.10中的例子,它有7个节点和9条边,包括一个循环和一个 n_4 节点到 n_4 节点的边(有时称为“自循环”)。

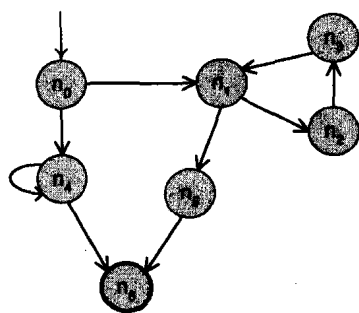


图2.10 一个主测试路径的例子

主路径可以从长度为0的路径开始寻找,然后扩展到长度为1,以此类推。此算法搜集所有简单路径,不论主要与否。通过这个集合可以很容易地筛选出主路径。长度为0的路径集合就是节点集合,长度为1的路径集合就是边的集合。为了简化,我们在这个例子中仅列出节点标号。

长度为0的简单路径(7):

- 1) [0]
- 2) [1]
- 3) [2]
- 4) [3]
- 5) [4]
- 6) [5]
- 7) [6]!

在路径[6]中的惊叹号告诉我们这个路径不能扩展。具体来说,最后的节点6没有向外的边,

所以以6结尾的路径不能继续扩展。

长度为1的简单路径(9):

- 8) [0, 1]
- 9) [0, 4]
- 10) [1, 2]
- 11) [1, 5]
- 12) [2, 3]
- 13) [3, 1]
- 14) [4, 4]*
- 15) [4, 6]!
- 16) [5, 6]!

路径[4, 4]的星号告诉我们这条路径不能继续扩展了, 因为第一个节点和最后一个节点一样(已经构成一个圆)。对于长度为2的路径, 我们首先找出长度为1的路径中没有构成圆的路径(以星号标识)。接下来, 我们扩展这条路径到从这条路径上的最后节点可以到达的每一个节点, 除非该节点已经存在于路径中或者是第一个节点。长度为1的第一条路径[0, 1], 被扩展为[0, 1, 2]和[0, 1, 5]。第二条路径[0, 4]被扩展为[0, 4, 6]而不是[0, 4, 4], 因为节点4已经存在于路径[0, 4]中。([0, 4, 4]不是简单路径, 所以它也不是主路径)。

长度为2的简单路径(8):

- 17) [0, 1, 2]
- 18) [0, 1, 5]
- 19) [0, 4, 6]!
- 20) [1, 2, 3]
- 21) [1, 5, 6]!
- 22) [2, 3, 1]
- 23) [3, 1, 2]
- 24) [3, 1, 5]

长度为3的路径可以用相似的方式计算出来。

长度为3的简单路径(7):

- 25) [0, 1, 2, 3]!
- 26) [0, 1, 5, 6]!
- 27) [1, 2, 3, 1]*
- 28) [2, 3, 1, 2]*
- 29) [2, 3, 1, 5]
- 30) [3, 1, 2, 3] *
- 31) [3, 1, 5, 6]!

最后, 只有一条长度为4的路径存在。3条长度为3的路径不能被扩展, 因为它们存在回路。对于剩下的2个, 以节点3结尾的路径不能被扩展是因为[0, 1, 2, 3, 1]不是简单路径, 因此不是主路径。

长度为4的主路径(1):

32) [2, 3, 1, 5, 6] !

主路径可以通过消去属于某个其他简单路径的子路径的任何路径得到。注意消去每个没有惊叹号或者星号的简单路径，因为它可以被扩展而且是某个其他简单路径的一个合适子路径。这里有8个主路径：

14) [4, 4] *

19) [0, 4, 6] !

25) [0, 1, 2, 3] !

26) [0, 1, 5, 6] !

27) [1, 2, 3, 1] *

28) [2, 3, 1, 2] *

30) [3, 1, 2, 3] *

32) [2, 3, 1, 5, 6] !

这个过程保证能结束，因为最长的可能的主路径的长度就是节点的个数。尽管图中经常有很多简单的路径（在这个例子中有32个，其中8个是主路径），它们可以经常被极少的测试路径游历。很多可行的算法可以找到测试路径来游历主路径。像图2.10一样简单的图，通过观察就足够了。例如，可以看到4个测试路径[0, 1, 5, 6]、[0, 1, 2, 3, 1, 2, 3, 1, 5, 6]、[0, 4, 6]和[0, 4, 4, 6]就足够了。然而，这种方法容易出错。最容易做的事情是只游历循环[1, 2, 3]一次，这忽略了主路径[2, 3, 1, 2]和[3, 1, 2, 3]。

对于更加复杂的图，需要一种机械式的方法。我们建议首先从最长的主路径开始，然后扩展到图中头和尾的节点。对于我们的例子，测试路径的结果是[0, 1, 2, 3, 1, 5, 6]。测试路径[0, 1, 2, 3, 1, 5, 6]游历了3个主路径25、27和32。

下一个测试路径通过扩展剩下的最长的一条主路径来构造得到。我们需要继续返回然后选择30测试路径的结果是[0, 1, 2, 3, 1, 2, 3, 1, 5, 6]，它游历了2个主路径，28和30（它也游历了路径25和27）。

下一个测试路径通过使用主路径26[0, 1, 5, 6]来构造得到。这个测试路径只游历一条最大的主路径26。

按照这种方式继续得到2个测试路径，对于主路径19的[0, 4, 6]和对于主路径14的[0, 4, 4, 6]。

最后完成的测试路径集合为：

1) [0, 1, 2, 3, 1, 5, 6]

2) [0, 1, 2, 3, 1, 2, 3, 1, 5, 6]

3) [0, 1, 5, 6]

4) [0, 4, 6]

5) [0, 4, 4, 6]

该集合可以直接使用，或者通过优化得到测试人员需要一个较小的测试集合。明显地，测试路径2游历了测试路径1所游历的主路径，所以1可以被省略，留下了4个在本章前面非正式地发现的测试路径。简单算法可以使该过程自动化。

2.2.1节练习

1. 以标准形式重新定义边覆盖（参考节点覆盖的讨论）。

2. 以标准形式重新定义完全路径覆盖 (参考节点覆盖的讨论)。
3. 包容有一个很明显的弱点。假设标准 C_{strong} 包容 C_{weak} 标准, 测试集合 T_{strong} 满足 C_{strong} 和测试集合 T_{weak} 满足 C_{weak} , 不一定说 T_{weak} 是 T_{strong} 的子集, 也不一定说如果 T_{weak} 暴露错误则 T_{strong} 也暴露错误。解释其中的原因。
4. 根据如下集合所定义的图回答问题(a)~(d):
 - $N = \{1, 2, 3, 4\}$
 - $N_0 = \{1\}$
 - $N_f = \{4\}$
 - $E = \{(1, 2), (2, 3), (3, 2), (2, 4)\}$
 - (a) 画出图。
 - (b) 列出满足节点覆盖而不是边覆盖的测试路径。
 - (c) 列出满足边覆盖而不是边对覆盖的测试路径。
 - (d) 列出满足边对覆盖的测试路径。
5. 根据如下集合所定义的图回答问题(a)~(g):
 - $N = \{1, 2, 3, 4, 5, 6, 7\}$
 - $N_0 = \{1\}$
 - $N_f = \{7\}$
 - $E = \{(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (4, 6), (6, 1)\}$
 同时考虑以下 (候选) 测试路径:
 - $t_0 = [1, 2, 4, 5, 6, 1, 7]$
 - $t_1 = [1, 2, 3, 2, 4, 6, 1, 7]$
 - (a) 画出图。
 - (b) 列出对于边对覆盖的测试需求。(提示: 对于长度2你应该得到12个需求。)
 - (c) 给出的测试路径满足边对覆盖吗? 如果不满足, 找到缺少了什么。
 - (d) 考虑简单路径 $[3, 2, 4, 5, 6]$ 和测试路径 $[1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]$ 。这个测试路径直接游历了简单路径吗? 侧访游历呢? 如果是侧访游历, 请确定侧访游历。
 - (e) 列出图中对于节点覆盖、边覆盖和主路径覆盖的测试需求。
 - (f) 列出图中满足节点覆盖但不满足边覆盖的测试需求。
 - (g) 列出图中满足边覆盖但不满足主路径覆盖的测试需求。
6. 根据图2.2回答问题(a)~(c):
 - (a) 列举图中满足节点覆盖, 边覆盖和主路径覆盖的测试需求。
 - (b) 列举图中满足节点覆盖但不满足边覆盖测试需求。
 - (c) 列举图中满足边覆盖但不满足主路径覆盖的测试需求。
7. 根据如下集合所定义的图回答问题(a)~(d):
 - $N = \{0, 1, 2\}$
 - $N_0 = \{0\}$
 - $N_f = \{2\}$
 - $E = \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0)\}$
 同时考虑以下 (候选) 路径:

- $p_0 = [0, 1, 2, 0]$
- $p_1 = [0, 2, 0, 1, 2]$
- $p_2 = [0, 1, 2, 0, 1, 0, 2]$
- $p_3 = [1, 2, 0, 2]$
- $p_4 = [0, 1, 2, 1, 2]$

- (a) 列出的路径哪些是测试路径? 解释不属于测试路径的路径存在的问题。
- (b) 列出满足边对覆盖的8条测试需求 (只有长度22的子路径)。
- (c) 测试路径 (a部分) 集合满足边对覆盖吗? 如果不满足, 确认缺少了什么。
- (d) 考虑主路径 $[n_2, n_0, n_2]$ 和路径 p_2 , p_2 直接游历了主路径吗? 侧访呢?

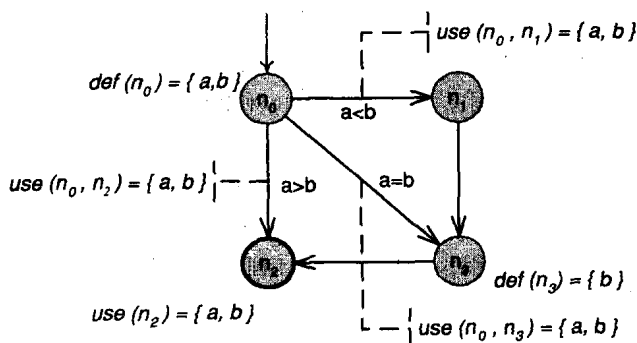


图2.11 显示变量、def集合和use集合的图

8. 设计并实现程序: 计算一个图里的所有的主路径, 然后生成游历这些主路径的测试路径。尽管用户界面可以任意复杂, 最简单的版本应该通过读到节点的序列, 最初节点, 最终节点和边接受一个图作为输入。

2.2.2 数据流标准

以下的测试标准基于这样的假设, 为了充分地测试一个程序, 我们应该关注数据值的流。特别地, 我们应该努力保证在程序中一个点上创建的数值是被正确创建和使用的。这要通过关注数值的定义 (definition) 和使用 (use) 来完成。定义 (def) 是一个在内存中存储变量值的位置 (如赋值、输入等)。使用是一个变量的值被访问的位置。数据流测试标准的依据是数值从defs传递到uses, 我们把这叫做du-对 (du-pairs) (在测试文献中它们也叫做definition-use、def-use和du关联)。数据流标准的思想是以不同方式执行du-pairs。

首先我们必须把数据流集成到已有的图模型上。假设 V 是与程序工件相关联的变量集合, 以图的方式建立程序工件模型。每个节点 n 和边 e 被认为是 V 的一个子集, 叫做 $def(n)$ 或 $def(e)$ 。(虽然从程序图的边上可能没有defs, 但其他软件工件, 比如有穷状态机, 允许defs作为边的权值。) 每个节点 n 和边 e 也被认为要使用 V 的一个子集, 叫做 $use(n)$ 或者 $use(e)$ 。

图2.11给出了一个用defs和uses标注图的例子。假设一个分支决策中包含的所有变量在相关联的边上都会被使用, 所以 a 和 b 在 (n_0, n_1) 、 (n_0, n_2) 和 (n_0, n_3) 三条边的use集合中。

讨论数据流标准时, 一个重要的概念是一个变量的def能否到达一个特定的use。最明显的理由是, 如果从 l_i 到 l_j 不存在路径, 那么变量 v 在 l_i 位置 (这个位置可以是节点或边) 的def不会到达在 l_j 位置的use。另一个更微妙的理由是, 变量的值可能在它到达这个use之前被另一个def

改变。所以如果路径上的每个节点 n_k 和每条边 e_k , $k \neq i$ 并且 $k \neq j$, 变量 v 不在 $def(n_k)$ 或 $def(e_k)$ 中, 这时对于变量 v , 我们说从 l_i 到 l_j 的路径是定义清纯的。也就是说, 在 l_i 到 l_j 之间没有某个位置改变了变量 v 的数值。如果对于变量 v , l_i 到 l_j 的路径是定义清纯的, 我们说 v 在 l_i 的def能够到达 l_j 的use。

为了简化, 我们把du-path的起点和终点作为节点, 虽然definition和use发生在边上。我们会在后面讨论放松这一条件。正常地, 一条对于变量 v 的du-path是一条变量 v 从节点 n_i 到节点 n_j 的定义清纯的简单路径, 其中 n_i 在 $def(n_i)$ 中, n_j 在 $use(n_j)$ 中。我们希望路径都是简单的, 以保证一组合理的少量的路径。注意, du-path总是和特定的变量 v 关联的, 一条du-path总是简单的, 并且在路径上可能存在多次的use。

图2.12给出了一个已经标注了def和use的图。我们选择显示所有与节点和边关联的程序语句, 而不是显示实际的集合。这对人来说是常见的和更有信息量的, 但是对于自动化工具的处理则是实际的集合更为简单。注意, 参数(subject和pattern)是被认为在图的第一个节点显式定义的。也就是, 节点1的def集合是 $def(1)=\{subject, pattern\}$ 。同时注意程序中的决策分支(例如, `if subject[i Sub]==pattern[0]`), 它导致了决策分支的两条边上每个相关变量的use。即 $use(4, 10)=use(4, 5)=\{subject, i Sub, pattern\}$ 。参数subject在节点2(引用它的长度属性)和边(4, 5)、(4, 10)、(7, 8)和(7, 9)使用, 所以从节点1到节点2和从节点1到这四条边的每条du-path都存在。

图2.13显示了同样的图, 但这次图上明确标识了def和use集合。注意, 节点9既定义又使用了变量*iPat*。[⊖]这是因为语句*iPat++*和*iPat=iPat+1*是一样的。在这种情况下, use在def之前发生, 所以从节点5到节点9才是关于*iPat*的定义清纯的路径。

数据流的测试标准将被定义为一套du-path的集合。这使得标准很简单, 但是我们首先需要将各du-path分类到几个组中。

对du-path的第一次分组是按照definition。具体来说, 考虑关于在一个指定节点定义的一个给定的变量的所有du-path。假设从节点 n_i 开始的变量 v 的du-path集合为 $du(n_i, v)$ 。一旦我们明确了数据流覆盖中游历的概念, 我们就可以简单地通过判断是否每个def-path集合中至少存在一条du-path被游历来定义All-Def标准。因为在典型的图上有大量的节点, 并且每个节点上有大量潜在的被定义的变量, 所以def-path集合的数量会非常大。即便如此, 由def-path分组游历的覆盖标准却是会非常弱的。

也许令人惊奇, 但是通过use分组du-path并没有什么帮助, 因此, 我们将不提供和上面def-path集合定义并列的“use-path”集合的定义。

第二, 也是更重要的, 通过每对definition和use对du-path进行分组。我们把这叫做def-pair集合。毕竟, 数据流测试的核心是允许数据从definition流向use。特别地, 考虑关于一个给定变量的所有du-path, 这个变量在一个节点上被定义并且在另一个节点上(可能是相同的节点)被使用。形式化地说, 让def-pair集合 $du(n_i, n_j, v)$ 表示关于变量 v 的du-paths集合, 它开始于节点 n_i , 结束于节点 n_j 。非形式化地说, 一个def-pair集合把从一个指定的definition到一个指定的use的所有(简单的)路径收集在一起。一旦我们明确了数据流覆盖中游历的概念, 我们将简单地通过判定是否游历至少一条来自每个def-pair集合的du-path来定义All-Use标准。因

⊖ 读者可能会奇怪为什么NOTFOUND的错误没有在集合use(2)中出现, 原因如2.3.2节中描述的那样, 是在本地使用。

为典型情况下每个definition能到达多个use，所以def-pair集合经常比def-path集合要多得多。

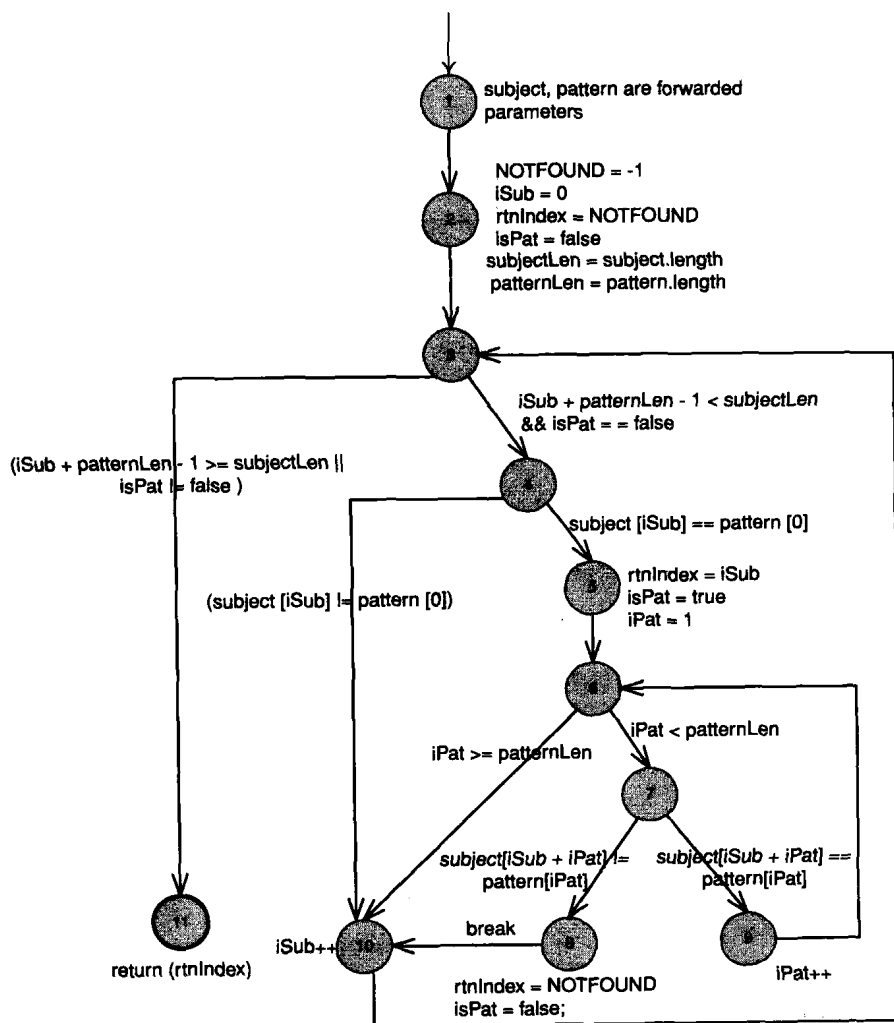


图2.12 用于表示du-path例子的图

事实上，在节点 n_i 上对于一个def的def-pair集合就是这个def的所有def-path集合的并集。更形式化地说： $du(n_i, v) = \bigcup_j du(n_i, n_j, v)$ 。

为了举例说明def-path集合和def-pair集合的概念，考虑对于变量 $iSub$ 的du-path，它在图2.13中的节点10有一个definition。对于 $iSub$ ，从节点10到节点5和节点10，从节点10到边(3, 4)、(3, 11)、(4, 5)、(4, 10)、(7, 8)和(7, 9)都有du-path。

对于 $iSub$ 在节点10的use的def-path是：

$$du(10, iSub) = \{[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], [10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], [10, 3, 11]\}$$

这个def-path集合能够被分成以下的def-pair集合：

$$du(10, 4, iSub) = \{[10, 3, 4]\}$$

$$du(10, 5, iSub) = \{[10, 3, 4, 5]\}$$

$du(10, 8, iSub) = \{[10, 3, 4, 5, 6, 7, 8]\}$
 $du(10, 9, iSub) = \{[10, 3, 4, 5, 6, 7, 9]\}$
 $du(10, 10, iSub) = \{[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10]\}$
 $du(10, 11, iSub) = \{[10, 3, 11]\}$

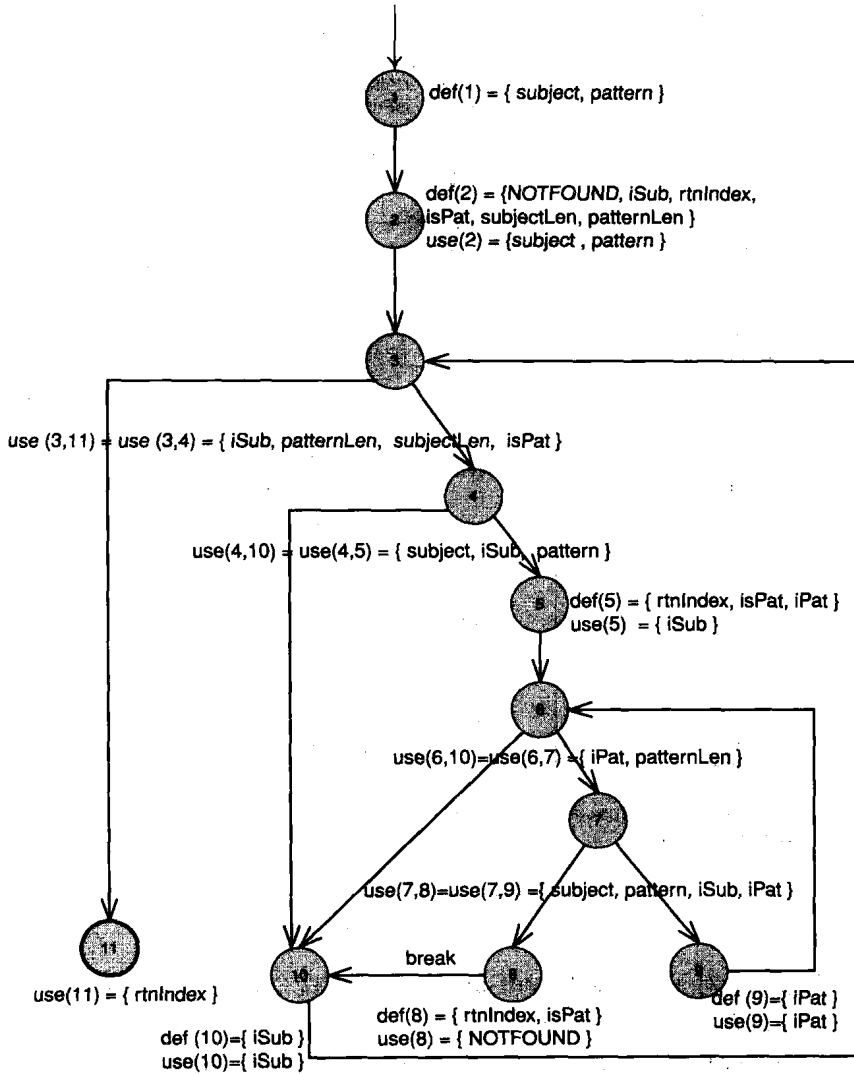


图2.13 说明def和use集合

接下来，我们扩展游历的定义使其应用到 $du\text{-path}$ 上。如果一条测试路径 p 游历了 d ，并且和 d 相对应的 p 的部分对于 v 是定义清纯的，那么我们说 p du 游历了对于 v 的子路径 d 。根据定义覆盖标准的期望，游历一条 $du\text{-path}$ 时，你可以允许或不允许对于 v 的定义清纯的侧访路径。因为定义清纯的侧访路径可能会游历更多的 $du\text{-path}$ ，我们在定义后续的数据流覆盖标准时，如果必要，则允许侧访路径。

现在我们定义基本数据流覆盖标准。三个最常见的定义最好以非形式化的方式来理解，第一个要求每个 def 要到达至少一个 use ，第二个要求每个 def 到达所有可能的 use ，第三个要

求每个def要通过所有可能的du-path到达所有可能的use。就像在构造def-path集合和def-pair集合时提到的一样,标准的形式化定义是简单地从合适的集合中选择合适的选项。对于以下的每个测试标准,我们假设最大努力的游历(见2.2.1节),即侧访路径是对于所关注的变量是定义清纯的。

标准2.9 All-Defs Coverage (ADC): 对于每个def-path集合 $S = du(n, v)$, TR 至少包含一条 S 中的路径 d 。

记得def-path集合 $du(n, v)$ 表示所有从 n 到 v 的所有use的定义清纯的简单路径。所以All-Defs要求我们能游历至少一条路径到至少一个use。

标准2.10 All-Uses Coverage (AUC): 对于每个def-pair集合 $S = du(n_i, n_j, v)$, TR 至少包含一条 S 中的路径 d 。

记得def-pair集合 $du(n_i, n_j, v)$ 表示所有从在 n_i 上对于 v 的一个def到 n_j 上对于 v 的一个use的定义清纯的简单路径。所以All-Uses要求我们对于每个def-use对能够游历至少一条路径。^①

标准2.11 All-du-Paths Coverage (ADUPC): 对于每个def-pair集合 $S = du(n_i, n_j, v)$, TR 包含 S 中的每条路径 d 。

这个定义可以简单地写为“包含每条du-path”。我们选择上面的定义是因为它强调了All-du-Paths与All-Uses之间的关键区别是限定程度的不同。就是说,在All-Uses中“至少一条du-path”变为All-du-Paths中的“每条路径”。基于对“def-use对”的思考,All-Uses需要某条定义清纯的简单路径到每个use,而All-du-Paths需要所有定义清纯的简单路径。

为了简化上面的推导,我们曾假设定义和使用在节点上产生。当然,定义和使用也可以在边上产生。有证据表明上面的推导也可以用于边上的使用,所以在程序流程图上可以容易的定义数据流(在程序流程图的边上的使用,有时称为“p-uses”)。然而,如果在图形边上存在变量的定义,以上推导将不再适用。这是因为,从一条边出发到另一条边的du-path不再简单,这样的—个du-path不仅仅拥有共同的首尾结点,同时可能有共同的首尾边。虽然我们可以修改定义以显式地提及边和节点上的定义和使用,但是这样的定义趋于混乱。参考文献注释包含这种类型推导的指南。

图2.14通过双菱形图举例说明了这三种数据流覆盖标准的差异。图中只有一个def,所以满足all-defs只需要一条路径。这个def有两个use,因此满足all-uses需要两条路径。因为从一个def到每个use都有两条路径,所以满足all-du-paths需要四条路径。注意,这个定义的数据流标准已经开放了游历的选择。文献使用不同的选择,在某些情况下需要直接的游历,在其他情况下,允许定义清纯侧访。我们的建议是选择最大努力的游历,区别于在文献中的处理,以达到理想的包含关系,即使在不可行的测试需求的情况下。从实用的角度来说,最大努力的游历也不无道理——每个测试需求是尽可能被严格满足的。

① 读者要谨慎地注意,尽管标准All-Defs和All-Uses的名字似乎没有关联,考虑到如何处理定义和使用,它们并不是互补的标准。特别地,我们无法通过把All-Def中的def替换成use概念而达到All-Uses。读者如果注意到All-Defs关注于定义,All-Uses关注于def-use对可能有所帮助。然而有人提出这样的命名习惯容易令人误解,相对于All-Uses、All-Pairs可能更为合适,作者选择在数据流文献中坚持这样标准的使用。

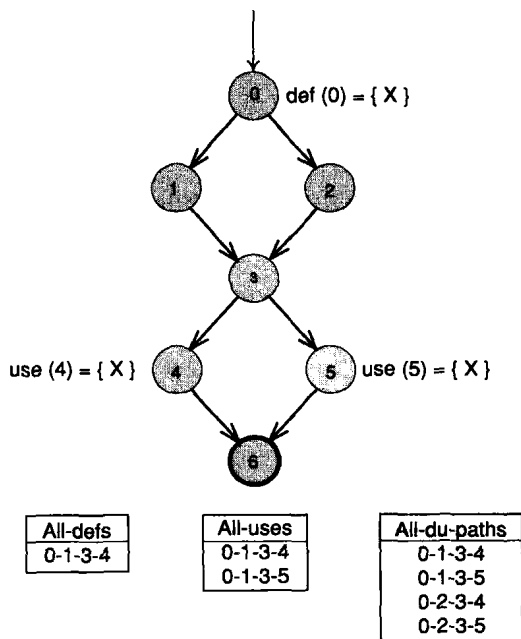


图2.14 三个数据流覆盖标准的差别

2.2.3 图覆盖标准中的包含关系

回顾一下第1章，覆盖标准之间往往通过包含关系相关联。首先值得注意的关系是，边的覆盖包含了节点的覆盖。在大多数情况下，这是因为如果我们遍历图的每一条边，我们将访问每一个节点。然而，如果一个图形有一个节点没有入边或出边，那么遍历每条边也不能达到这个节点。因此，边覆盖定义为包含长度小于等于1的每条路径，就是长度为0的每条路径（所有节点）和长度为1的每条路径（所有边）。这个包含关系在相反方向并不成立。回顾一下图2.6给出的测试集例子，它满足节点覆盖但不满足边覆盖，因而，节点点覆盖不包含边覆盖。

标准之间有各种包含关系。在可适用的地方，结构化的覆盖关系假设了最大努力的游历。由于最大努力的游历已指定，即使当一些测试需求不可行时，包含结果也成立。

数据流标准的包含结果是基于三个假设：（1）每个use之前都有def；（2）每个def到达至少一个use；（3）对每个有多条出边的节点，每条出边上至少用到一个变量，并且在每条出边上用到相同的变量。如果满足All-Uses覆盖，那么我们会隐式地确保每个def都被使用过。因而也满足了All-Defs覆盖，所以，All-Uses包含All-Defs。同样，如果满足All-du-Paths覆盖，那么我们会隐式地确保每个定义达到每一个可能的use。因此也满足了All-Uses覆盖，所以，All-du-Paths包含All-Uses。此外，每条边是基于满足某个断言，所以每条边至少有一个use。因此All-Uses标准保证每一条边至少被执行一次，所以All-Uses覆盖包含了边覆盖。

最后，每个du-path也是一个简单的路径，所以主路径覆盖包含All-du-Paths覆盖[⊖]。这是一个重要的发现，因为计算主路径比分析数据流关系简单得多。图2.15展示了结构化与数据

⊖ 考虑到du-path中关于变量x只能被侧访一次。更进一步讲，假定有两个可能的侧访，其中一个关于x是定义清纯，另一个不是。All-du-Paths测试集中相关的测试路径必然遍历前一个侧访，作为最初路径测试集中相对应的测试路径则遍历后一个侧访。我们的观点是多数情况下测试工程师忽略这个特殊的用例而使用最初的路径覆盖简单地继续下去是完全合理的。

流覆盖标准之间的包含关系。

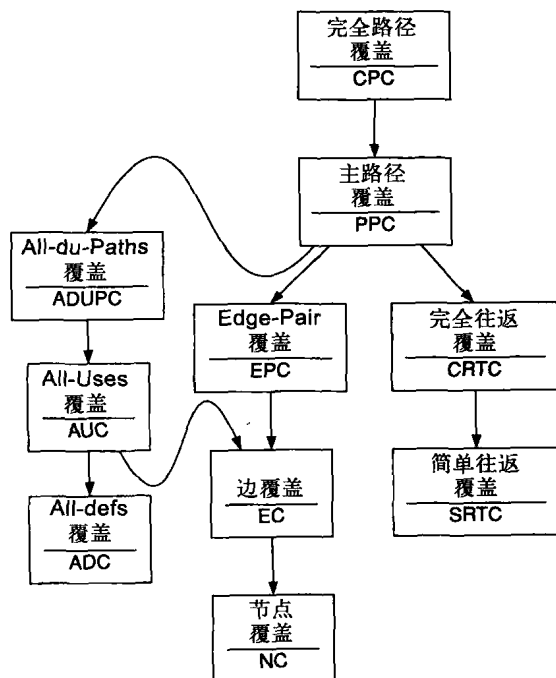


图2.15 覆盖标准的包含关系

2.2.3节练习

下列四个图都由以下元素定义：节点集合、初始节点、终止节点、边、定义（def）以及使用（use）。每个图还包括一系列的测试路径。根据每一幅图回答下列问题。

Graph I.

$N = \{0, 1, 2, 3, 4, 5, 6, 7\}$
 $N_0 = \{0\}$
 $N_f = \{7\}$
 $E = \{(0, 1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$
 $def(0) = def(3) = use(5) = use(7) = \{x\}$

Test Paths:

$t_1 = [0, 1, 7]$
 $t_2 = [0, 1, 2, 4, 6, 1, 7]$
 $t_3 = [0, 1, 2, 4, 5, 6, 1, 7]$
 $t_4 = [0, 1, 2, 3, 2, 4, 6, 1, 7]$
 $t_5 = [0, 1, 2, 3, 2, 3, 2, 4, 5, 6, 1, 7]$
 $t_6 = [0, 1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]$

Graph II.

$N = \{1, 2, 3, 4, 5, 6\}$
 $N_0 = \{1\}$
 $N_f = \{6\}$
 $E = \{(1, 2), (2, 3), (2, 6), (3, 4), (3, 5), (4, 5), (5, 2)\}$
 $def(x) = \{1, 3\}$
 $use(x) = \{3, 6\}$ // 假设语句3中x的使用在定义之前

Test Paths:

$t_1 = [1, 2, 6]$
 $t_2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
 $t_3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$
 $t_4 = [1, 2, 3, 5, 2, 6]$

Graph III.

$N = \{1, 2, 3, 4, 5, 6\}$
 $N_0 = \{1\}$
 $N_f = \{6\}$
 $E = \{(1, 2), (2, 3), (3, 4), (3, 5), (4, 5), (5, 2), (2, 6)\}$
 $def(x) = \{1, 4\}$
 $use(x) = \{3, 5, 6\}$

Test Paths:

$t_1 = [1, 2, 3, 5, 2, 6]$
 $t_2 = [1, 2, 3, 4, 5, 2, 6]$

Graph IV.

$N = \{1, 2, 3, 4, 5, 6\}$
 $N_0 = \{1\}$
 $N_f = \{6\}$
 $E = \{(1, 2), (2, 3), (2, 6), (3, 4), (3, 5), (4, 5), (5, 2)\}$
 $def(x) = \{1, 5\}$
 $use(x) = \{5, 6\}$ // 假设语句5中x的使用在定义之前

Test Paths:

$t_1 = [1, 2, 6]$
 $t_2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
 $t_3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$

- (a) 画出图。
- (b) 列出所有关于x的du-path。(注意：包含所有的du-path以及那些属于某些其他du-path的子路径的du-path。)
- (c) 对于每一个测试路径，确定测试路径经过了哪个du-path。在这部分练习中，要考虑有向游历和侧访两种情况。
- 提示：表结构便于描述关系。
- (d) 列出一个关于x的满足def全覆盖的最小测试集。(仅有向游历)用给出的测试路径。
- (e) 列出一个关于x的满足use全覆盖的最小测试集。(仅有向游历)用给出的测试路径。
- (f) 列出一个关于x的满足all-du-path覆盖的最小测试集。(仅有向游历)用给出的测试路径。

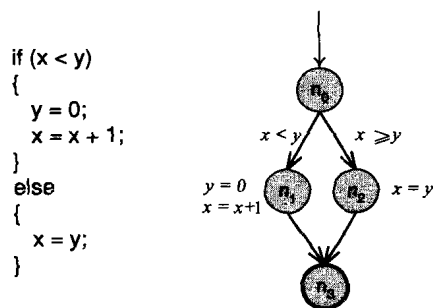


图2.16 if-else结构的CFG片段

2.3 源代码的图覆盖

大多数图覆盖标准是为源代码制定的，这些定义非常符合2.2节的定义。正如在2.2节中所说，我们首先考虑结构化的覆盖标准，然后考虑数据流的覆盖标准。

2.3.1 源代码的结构化图覆盖

使用最广泛的图覆盖标准是在源代码上定义的。虽然细节随编程语言的不同而有所变化，但对于最常见的语言来说，基本模式是相同的。为了应用其中一个图标准，第一步是定义图，对于源代码，最常见的图形是控制流图（CFG）。控制流图把边与程序中每一个可能的分支相对应，把点与程序中每一个语句序列相对应。形式化地讲，基本块（basic block）是一个最长的程序语句序列，在这个基本块中，如果一条语句被执行，那么所有语句都会被执行。一个基本块只有一个入口点和一个出点。第一个例子的语言结构是一个带有else子句的if语句，如图2.16所示Java代码及相应的控制流图。这个if-else结构产生两个基本块。

值得注意的是，if语句中的then部分有两条语句，它们都出现在相同的节点中。节点 n_0 代表了条件测试 $x < y$ ，它有不只一个出边，这个节点称为决策（decision）节点。节点 n_3 有不只一条入边，称为汇合（junction）节点。

接下来我们转向if语句中没有else子句时的退化情形，如图2.17所示。这与图2.6是完全相同的，不过这一回是基于实际的程序语句。

需要注意的是，这种结构的控制流图只有三个节点。读者应注意到，一个 $x < y$ 的测试可以遍历这个控制流图中的所有节点，但不能遍历所有边。

表示循环是有点棘手的，因为我们必须包括一些不是直接来自于程序语句的节点。最简单的一种循环是有初始化语句的while循环，如图2.18所示。（假设y在这一点上是有值的。）

while结构的图包含一个用来做条件测试的决策节点和一个表示while循环体的节点。节点

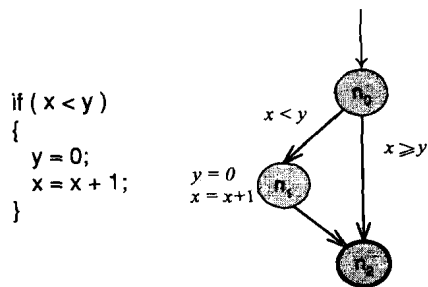


图2.17 没有else的if结构CFG片段

n_1 有时也叫做“虚拟节点”，因为它并不代表任何语句，但是指出了迭代边 (n_2, n_1) 要到哪里去。 n_1 节点也可以看成代表了一个决策。对于初学者来说的一个常见错误是尝试将边引向 n_0 ，这是不正确的，因为这意味着在循环的每一次迭代中都执行了初始化步骤。注意，调用 $f(x, y)$ 的方法不会在这个特定的图上扩展，我们稍后再来讨论这个问题。

现在考虑一个for循环，它相当于前面提到的while循环。如图2.19所示，因为for结构在一个相当高级别上的抽象，所以图变得更加复杂。

虽然初始化、条件测试、循环控制变量 x 的自增都在同一行程序中，但它们必须对应图上不同的节点。这个for循环的控制流图与while循环略有不同。具体地说，我们在与调用 $y = f(x, y)$ 的方法不同的节点上显示 x 的自增。从技术上来说，这违反了基本块的定义，两个节点应当合并，但我们可以对各种可能的程序开发模板结构，然后把控制流有关的节点插入到模板上正确的地方。商业工具典型地使用这种方法使得图的产生更简单。事实上，商业工具通常不按照严格的基本块的定义，并且有时会增加看似随机的节点。它们可能对于通常处理具有一些微不足道的影响（例如，我们可能覆盖 67/73而不是68/75），但对测试来说并不是很重要。

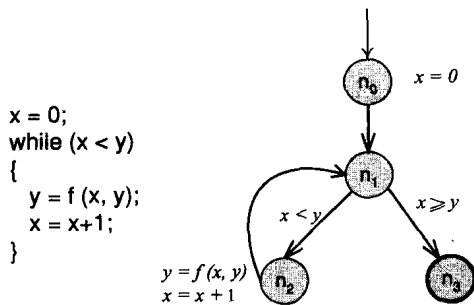


图2.18 while循环结构的CFG片段

```
for (x = 0; x < y; x++)
{
    y = f(x, y);
}
```

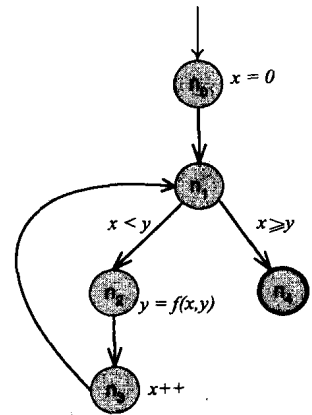


图2.19 循环结构的CFG片段

我们最后讨论的语言结构是Java语言中的case或switch语句。case结构可以图形化为一个具有多个分支的单独节点或者一系列的if-then-else结构。我们选择了具有多个分支的结构，如图2.20所示。

先前的覆盖标准可以应用到源码图。这种应用是直接的，只改动了其中的一些名称。节点覆盖又称为语句覆盖或者基本块覆盖，而边覆盖通常称为分支覆盖。

2.3.2 源代码的数据流图覆盖

本节对2.3.1节的代码例子应用数据流标准。在我们能够这样做之前，我们必须定义出什么构成一个def，什么构成一个use。def在程序中是一个位置，在该位置把一个变量值存放在内存中（赋值、输入等）。use是可以访问该变量值的位置。

```
read (c);
switch (c)
{
    case 'N':
        y = 25;
        break;
    case 'Y':
        y = 50;
        break;
    default:
        y = 0;
        break;
}
print (y);
```

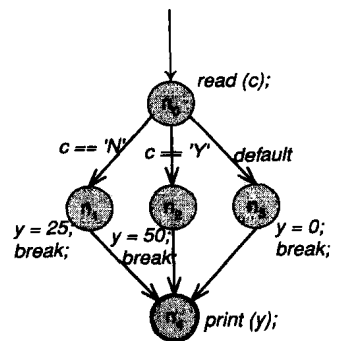


图2.20 实例结构的CFG片段

变量x的一个def可能会在如下一些情况发生：

1. x出现在一个赋值语句的左边。
2. 在调用时x是一个实际参数，并且它的值在该方法中被改变。
3. x是一个方法的形式参数（当方法开始执行时是一个隐式的def）。
4. x是一个程序的输入。

编程语言的一些特性极大地使这个看似简单的定义变得极其复杂。比如，一个数组变量的def是整个数组的，还是仅仅被引用的元素的？对象的情况又如何？def应该考虑整个对象，还是对象的一个特殊实例变量？假如两个变量指向同一地址，即其中一个变量是另外一个的别名，这种情况应该如何分析？最原始的源代码覆盖、优化后的源代码覆盖和机器代码覆盖之间又是什么关系？我们在讲述过程中省略了这些复杂问题，推荐想进一步了解的读者参考文献注释。

假如一个变量在一个基本块中有多重定义，则最后一个定义是与数据流分析相关的唯一定义。

变量x的一个use可能会在如下一些情况发生：

1. x出现在一个赋值语句的右边。
2. x出现在条件测试中（注意此种测试总是关联至少两条边）。
3. x是一个方法的实际参数。
4. x是程序的输出。
5. x在return语句中是一个方法的输出，或者作为一个参数被返回。

不是所有的use都与数据流分析相关。考虑下面一些引用局部变量的语句（忽略并发性）：

```
y = z;
x = y + 2;
```

在第二个语句中变量y的use叫做局部use。另外一个基本块的def不可能到达x=y+2语句的use。原因是y=z; 语句中的y定义总是覆盖其余基本块的y定义。即对于该use没有来自其余的def的定义清纯路径。相反，变量z的use叫做全局use，因为在这个基本块中使用的z的定义一定源于其他基本块。数据流分析仅仅考虑全局use。

在图2.21中实例TestPat阐明了对于一个用Java语言写的简单字符串模式匹配程序TestPat的数据流分析。

TestPat的CFG如图2.12所示，并且在节点和边上用实际的Java语句做了注释。

明确标记了def和use集合的TestPat的CFG如图2.13所示。虽然很多工具能够为程序创建CFG图，但让学生自己动手创建CFG对他们有帮助。这样做时，一个好的习惯是首先根据语句画CFG，然后再重画一遍CFG并标记def和use集合。

表2.1为TestPat列举了CFG每个节点的def和use，以一种更方便的形式重复了图2.13的信息。表2.2对于边包含了同样的信息。我们建议初学者通过验证这两张表的内容是否正确来检查他们对这些定义的理解。

最后，我们在表2.3中列举了TestPat中每个变量的du-path，跟随这些du-path是对每个du-pair的所有du-path。表的第一列是变量名，第二列给出def节点号和变量（如2.2.2节中所定义的，公式的左边列举关于变量的所有du-path），第三列列举了从该def开始的所有du-path。假如一个du-pair有一条以上路径到达相同的use，它们被列举成多行，并且子路径结束于相同的

节点号。第四列“prefix?”是一个方便性符号，将在下面解释。通过手工来得到这个信息特别枯燥，并且测试者容易犯许多错误，因此这分析过程最好自动化进行。

```
// 两个字符串的模式匹配示例程序
class TestPat
{
    public static void main (String[] argv)
    {
        final int MAX = 100;
        char subject[] = new char[MAX];
        char pattern[] = new char[MAX];
        if (argv.length != 2)
        {
            System.out.println
            ("java TestPat String-Subject String-Pattern");
            return;
        }
        subject = argv[0].toCharArray();
        pattern = argv[1].toCharArray();
        TestPat testPat = new TestPat ();
        int n = 0;
        if ((n = testPat.pat (subject, pattern)) == -1)
            System.out.println
            ("Pattern string is not a substring of the subject string");
        else
            System.out.println
            ("Pattern string begins at the character " + n);
    }

    public TestPat ()
    {}

    public int pat (char[] subject, char[] pattern)
    {
        // 提示：假如模式不是主体的子字符串，返
        // 回-1，否则返回模式首先在主体中出现的
        // 位置（从0开始）

        final int NOTFOUND = -1;
        int iSub = 0, rtnIndex = NOTFOUND;
        boolean isPat = false;
        int subjectLen = subject.length;
        int patternLen = pattern.length;

        while (isPat == false && iSub + patternLen - 1 < subjectLen)
        {
            if (subject[iSub] == pattern[0])
            {
                rtnIndex = iSub; // Starting at zero
                isPat = true;
                for (int iPat = 1; iPat < patternLen; iPat++)
                {
                    if (subject[iSub + iPat] != pattern[iPat])
                    {
                        rtnIndex = NOTFOUND;
                        isPat = false;
                        break; // out of for loop
                    }
                }
            }
            iSub++;
        }
        return (rtnIndex);
    }
}
```

图2.21 数据流实例的TestPat

表2.1 TestPat的CFG中每个节点的def和use

节点	def	use
1	{subject, pattern}	
2	{NOTFOUND, isPat, iSub, rtnIndex, subjectLen, patternLen}	{subject, pattern}
3		
4		
5	{rtnIndex, isPat, iPat}	{iSub}
6		
7		
8	{rtnIndex, isPat}	{NOTFOUND}
9	{iPat}	{iPat}
10	{iSub}	{iSub}
11		{rtnIndex}

在TestPat中有若干def/use对有一条以上的du-path。比如，变量iSub在节点2中定义，在节点10中使用。存在三条du-path，分别是[2, 3, 4, 10](iSub)，[2, 3, 4, 5, 6, 10](iSub)和[2, 3, 4, 5, 6, 7, 8, 10](iSub)。

表2.2 TestPat的CFG中每条边的def和use

边	use
(1, 2)	
(2, 3)	
(3, 4)	{iSub, patternLen, subjectLen, isPat}
(3, 11)	{iSub, patternLen, subjectLen, isPat}
(4, 5)	{subject, iSub, pattern}
(4, 10)	{subject, iSub, pattern}
(5, 6)	
(6, 7)	{iPat, patternLen}
(6, 10)	{iPat, patternLen}
(7, 8)	{subject, iSub, iPat, pattern}
(7, 9)	{subject, iSub, iPat, pattern}
(8, 10)	
(9, 6)	
(10, 3)	

表2.3 TestPat中每个变量的du-path集合

变量	du-path set	du-paths	prefix?
NOTFOUND	du (2, NOTFOUND)	[2,3,4,5,6,7,8]	
rtnIndex	du (2, rtnIndex)	[2,3,11]	
	du (5, rtnIndex)	[5,6,10,3,11]	
	du (8, rtnIndex)	[8,10,3,11]	
iSub	du (2, iSub)	[2,3,4]	Yes
		[2,3,4,5]	Yes
		[2,3,4,5,6,7,8]	Yes
		[2,3,4,5,6,7,9]	
		[2,3,4,5,6,10]	
		[2,3,4,5,6,7,8,10]	
		[2,3,4,10]	
		[2,3,11]	
	du (10, iSub)	[10,3,4]	Yes
		[10,3,4,5]	Yes
		[10,3,4,5,6,7,8]	Yes
		[10,3,4,5,6,7,9]	
		[10,3,4,5,6,10]	
		[10,3,4,5,6,7,8,10]	

(续)

变量	du-path set	du-paths	prefix?
iPat	du (5, iPat)	[10,3,4,10]	Yes
		[10,3,11]	
		[5,6,7]	
		[5,6,10]	
		[5,6,7,8]	
	du (9, iPat)	[5,6,7,9]	Yes
		[9,6,7]	
		[9,6,10]	
		[9,6,7,8]	
		[9,6,7,9]	
isPat	du (2, isPat)	[2,3,4]	
		[2,3,11]	
	du (5, isPat)	[5,6,10,3,4]	
		[5,6,10,3,11]	
	du (8, isPat)	[8,10,3,4]	
subject	du (1, subject)	[8,10,3,11]	Yes
		[1,2]	
		[1,2,3,4,5]	
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
pattern	du (1, pattern)	[1,2,3,4,5,6,7,9]	Yes
		[1,2]	
		[1,2,3,4,5]	
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
subjectLen	du (2, subjectLen)	[1,2,3,4,5,6,7,9]	
		[2,3,4]	
		[2,3,11]	
		[2,3,4]	
		[2,3,11]	
patternLen	du (2, patternLen)	[2,3,4,5,6,7]	Yes
		[2,3,4,5,6,10]	

表2.4 TestPat中满足所有du-path覆盖的测试路径

测试用例 (主体、模式、输出)	test path (t)
(a, bc, -1)	[1,2,3,11]
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]
(abc, abd, -1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]
(abc, ac -1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]
(abc, ba, -1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]

一个优化方法利用这样一个事实，即一个du-path必须被任何游历该du-path扩展的测试游历。这些du-path在表的“prefix?”列上面标记了“Yes”。比如，[2, 3, 4](iSub)必然被任何游历du-path[2, 3, 4, 5, 6, 7, 8](iSub)的测试游历，因为[2,3,4]是[2, 3, 4, 5, 6, 7, 8]的前缀。因此，在接下来关联du-path和游历它们的测试路径的表中该路径没有被考虑。我们必须对此优化方法很仔细，因为即使前缀可行，扩展的du-path也许并不可行。

表2.4展示了一个相对小的11个测试用例的集合，该集合满足此例子所有的du-path覆盖。(一个du-path是不可行的。)读者也许希望通过非数据流图覆盖标准来评价此测试集合。

表2.5列举了被每个测试用例游历的du-path。对于第一列里的每个测试用例，被该测试执行的测试路径在第二列展示，被该测试路径游历的du-path在第三列展示。

表2.5 TestPat的测试路径和被覆盖的du-path

测试用例 (主体、模式、输出)	test path(t)	du-path toured
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8](NOTFOUND)
(a, bc, -1)	[1,2,3,11]	[2,3,11](rtIndex)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](rtIndex)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](rtIndex)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[2,3,4,5,6,7,9](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](iSub)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8,10](iSub)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4,10](iSub)
(a, bc, -1)	[1,2,3,11]	[2,3,11](iSub)
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]	[10,3,4,5,6,7,9](iSub)
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]	[10,3,4,5,6,10](iSub)
(abc, ba, -1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]	[10,3,4,5,6,7,8,10](iSub)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[10,3,4,10](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[10,3,11](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10](iPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[5,6,7,8](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[5,6,7,9](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[9,6,10](iPat)
(abc, abd, -1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]	[9,6,7,8](iPat)
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]	[9,6,7,9](iPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4](isPat)
(a, bc, -1)	[1,2,3,11]	[2,3,11](isPat)
No test case	Infeasible	[5,6,10,3,4](isPat)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](isPat)
(abc, ac -1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]	[8,10,3,4](isPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](isPat)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](subject)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](subject)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](subject)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](pattern)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](pattern)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](pattern)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4](subjectLen)
(a, bc, -1)	[1,2,3,11]	[2,3,11](subjectLen)
(a, bc, -1)	[1,2,3,11]	[2,3,11](patternLen)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7](patternLen)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](patternLen)

2.3节练习

1. 对于后面的问题(a)~(e)运用下面的程序段。

```

w = x;      // 节点1
if (m > 0)
{
    w++;     // 节点2
}
else
{
    w = 2*w; // 节点3
}
// 节点4 (没有执行的语句)
if (y <= 10)
{
    x = 5*y; // 节点5
}

```

```

}
else
{
    x = 3*y+5; // 节点6
}
z = w + x;    // 节点7

```

- (a) 针对此程序段画一个控制流图。使用上面给出的节点编号。
 - (b) 对于变量 w ，哪些节点具有def?
 - (c) 对于变量 w ，哪些节点具有use?
 - (d) 从节点1到节点7，有任何关联变量 w 的du-path吗? 假如没有，解释为什么没有。如果存在，请举例子。
 - (e) 列举出关于变量 w 和 x 的所有的du-path。
2. 选择一个商业版覆盖工具。注意，有些工具可以免费试验评价。选择一个工具，下载并且将它运行在某个软件上。你可以使用本书的任何一个例子、你工作环境的软件或者网上可获得的软件。写一篇简短的使用该工具的经验总结报告，确保包含任何安装或者使用该工具的问题。主要的评分标准是你实际在某个程序上为一个合理的测试集合收集了一些覆盖数据。
 3. 考虑图2.21的模式匹配例子。插桩 (instrument) 代码使之能够产生报告中的针对这个例子的执行路径。即针对一个给定的测试执行，你的插桩程序应该能计算并且打印出相应的测试路径。基于2.3节末尾列举的测试用例来运行插桩程序。
 4. 考虑图2.21的模式匹配例子。特别地，考虑2.3节测试的最后一个表。针对变量 $iSub$ ，从该表的 $iSub$ 部分的顶部开始，给（独特的）测试用例从1开始编号。比如 $(ab, c, -1)$ ，在该表的 $iSub$ 部分出现了两次，应该标记为测试t4。
 - (a) 给出一个满足所有def覆盖的最小测试集合。使用给出的测试用例。
 - (b) 给出一个满足所有use覆盖的最小测试集合。
 - (c) 给出一个满足所有du-path覆盖的最小测试集合。
 5. 再次考虑图2.21的模式匹配例子。插桩代码使之能够产生报告中的针对这个例子的执行路径。即针对一个给定的测试执行，你的工具应该计算并且打印出相应的测试路径。运行下面的三个测试用例，并且回答下面(a)~(g)的问题：
 - 主体= “brown owl”，模式= “wl”，期待输出= 7
 - 主体= “brown fox”，模式= “dog”，期待输出=-1
 - 主体= “fox”，模式= “brown”，期待输出=-1
 - (a) 找到每个测试用例所走过的实际路径。
 - (b) 针对每条路径，在2.3节末尾的表中给出该路径遍历的所有du-path。为了减少此练习的范围，仅仅考虑下面的du-path： $du(10, iSub)$ 、 $du(2, isPat)$ 、 $du(5, isPat)$ 以及 $du(8, isPat)$ 。
 - (c) 解释为什么du-path[5, 6, 10, 3, 4]不能被任何测试路径遍历。
 - (d) 从2.3节末尾的表中选择测试来完成（可行的）du-path覆盖，并且这些du-path没有在问题(a)中覆盖到。
 - (e) 根据以上的测试，找出一个最小的测试集合来实现有关变量 $isPat$ 的All-Defs覆盖。
 - (f) 根据以上的测试，找出一个最小的测试集合来实现有关变量 $isPat$ 的All-Uses覆盖。
 - (g) 针对pat方法中的 $isPat$ 变量，All-Uses覆盖和所有的du-path覆盖有任何区别吗?

6. 使用下面的方法fmtRerwrap()完成后面的问题(a)~(e)。

```

1. /** *****
2.  * 重新包装字符串 (类似UNIX的fmt)。
3.  * 给定一个字符串S, 去除存在的换行符,
4.  * 在第N列之前的最近空白符增加换行符。同一行中两个换行符
5.  * 被认为是“硬换行符”, 留下这些换行符。
6.  * *****/
7.
8. static final char CR = '\n';
9. static final int inWord      = 0;
10. static final int betweenWord = 1;
11. static final int lineBreak   = 2;
12. static final int crFound     = 3;
13. static private String fmtRerwrap (String S, int N)
14. {
15.     int state = betweenWord;
16.     int lastSpace = -1;
17.     int col = 1;
18.     int i = 0;
19.     char c;
20.
21.     char SArr [] = S.toCharArray();
22.     while (i < SArr.length())
23.     {
24.         c = SArr[i];
25.         col++;
26.         if (col >= N)
27.             state = lineBreak;
28.         else if (c == CR)
29.             state = crFound;
30.         else if (c == '\n')
31.             state = betweenWord;
32.         else
33.             state = inWord;
34.         switch (state)
35.         {
36.             case betweenWord:
37.                 lastSpace = i;
38.                 break;
39.
40.             case lineBreak:
41.                 SArr [lastSpace] = CR;
42.                 col = i - lastSpace;
43.                 break;
44.
45.             case crFound:
46.                 if (i+1 < SArr.length() && SArr[i+1] == CR)
47.                 {
48.                     i++; // Two CRs => hard return
49.                     col = 1;
50.                 }
51.                 else
52.                     SArr[i] = "\n";
53.                 break;
54.
55.             case inWord:
56.                 default:

```

```

57.     break;
58. } //结束switch
59. i++;
60. } //结束while
61. S = new String (Sarr) + CR;
62. return (S);
63. }

```

(a) 为fmtRewrap()方法画控制流程图。

(b) 对于fmtRewrap(), 找到一个测试用例, 使得相应的测试路径访问连接while语句开始到“S=new String(SArr) + CR;”语句的边, 而不用通过while循环体。

(c) 针对fmtRewrap()的图列举每个节点覆盖, 边覆盖和主路径覆盖的测试需求。

(d) 列出图中实现节点覆盖而非边覆盖的测试路径。

(e) 列出图中实现边覆盖而非主路径覆盖的测试路径。

7. 使用下面的方法printPrimes()完成后面的问题(a)~(f)。

```

1. /** *****
2.  * 找出并且打印出n个素数
3.  * Jeff Offutt, 2003年春季
4.  ***** */
5. private static void printPrimes (int n)
6. {
7.     int curPrime;        // 目前考虑的素数的值, 目前找
8.     int numPrimes;        // 到的素数数目, curPrime
9.     boolean isPrime;      // 是素数吗
10.    int [] primes = new int [MAXPRIMES]; // 素数列表
11.
12.    // 将素数列表的第一个元素初始化为2
13.    primes [0] = 2;
14.    numPrimes = 1;
15.    curPrime = 2;
16.    while (numPrimes < n)
17.    {
18.        curPrime++; // 下面一个要考虑的数
19.        isPrime = true;
20.        for (int i = 0; i <= numPrimes-1; i++)
21.        { // 对于之前的每个素数
22.            if (isDivisible (primes[i], curPrime))
23.            { // 找到一个约数, curPrime不是素数
24.                isPrime = false;
25.                break; // 退出素数的循环
26.            }
27.        }
28.        if (isPrime)
29.        { // 保存
30.            primes[numPrimes] = curPrime;
31.            numPrimes++;
32.        }
33.    } // 结束while循环
34.
35.    // 打印出所有的素数
36.    for (int i = 0; i <= numPrimes-1; i++)
37.    {
38.        System.out.println ("Prime: " + primes[i]);
39.    }
40. } //结束printPrimes函数

```

- (a) 为printPrimes()方法画控制流图。
- (b) 考虑测试用例 $t1=(n=3)$ 和 $t2=(n=5)$ 。即使这些测试用例游历printPrimes()方法中相同的主路径，它们不一定找出相同的错误。设计一个简单的错误，使得 $t2$ 比 $t1$ 更容易发现。
- (c) 针对printPrimes()，找到一个测试用例，使得相应的测试路径访问连接while语句开始到for语句的边，而不用通过while循环体。
- (d) 针对printPrimes()的图列举每个节点覆盖、边覆盖和主路径覆盖的测试需求。
- (e) 列出图中实现节点覆盖而非边覆盖的测试路径。
- (f) 列出图中实现边覆盖而非主路径覆盖的测试路径。

2.4 设计元素的图覆盖

在数据抽象和面向对象软件中越来越强调模块化和重用，这也就意味着基于设计的各部分软件测试比以前变得更加重要。这些通常是和集成测试相关联的。模块化的一个好处就是使得程序员在单元测试和模块测试的时候能独立地测试软件的各组件。

2.4.1 设计元素的结构化图覆盖

设计元素的图覆盖通常是从为软件组件间的耦合（coupling）创建图入手。耦合是通过展示两个单元间的相互连接来衡量它们的依赖关系。在其中一个单元中产生错误就有可能影响到与其耦合的另一个单元。耦合为软件的设计和架构提供了总结性的信息。绝大部分设计元素的测试标准都要求程序各组件间是可以相互访问的。

用于结构化设计覆盖的最常见的图叫做调用图。在一个调用图中，节点表示方法（或单元），边表示方法的调用。图2.22就是一个简单的包含六个方法的程序。方法A调用B、C和D，方法C又调用E和F，D也调用F。

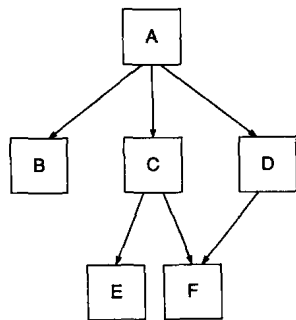


图2.22 简单的调用图

第2.2.1节中提到的覆盖标准也同样适用于调用图。节点覆盖（又叫方法覆盖）要求每一个方法至少被调用一次。边覆盖（又叫调用覆盖）要求每个调用至少被执行一次。在图2.22中，节点覆盖要求每个方法至少被调用一次，而边覆盖要求F至少被调用两次，从C调用一次和从D调用一次。

应用于模块

如第1章中所说，模块就是相关单元的集合。比如说，一个类就是Java版本的模块。相对于整个程序，一个类中的单元也许不完全相互访问，因此，我们可以生成一些不连接的调用图而不是连接的调用图。在一个简单的退化情况中（比如一个简单的栈），单元之间也许就没有调用。在这些情况中，使用这种技术的模块测试并不合适，而是需要基于调用序列的技术。

继承与多态

面向对象语言的继承与多态的特点为设计人员和开发人员引入了新的技术，但对于测试人员来说亦是新的问题。到目前为止，仍然不清楚怎么才能最好地测试这些新特点，什么样的标准比较合适。这里介绍了当前的状况，感兴趣的读者可以关注后续测试面向对象软件成果及技术。参考文献中有一些引用，第7章中也有进一步的讨论。测试这些特点（统一称为

“面向对象语言的特点”)的最有效的图是继承等级关系图 (inheritance hierarchy)。图2.23是四个类的继承等级关系图。类C和D继承自B, B又继承自A。

2.2.1节中的覆盖标准能通过很简单的方法应用在继承等级关系图中, 但仍有一些小问题。在面向对象编程中, 类并不是被直接测试的, 因为它们实际上是不可执行的, 继承等级关系图中的边界线不代表可执行流, 而是继承的依赖关系。为了应用覆盖任何的类型, 我们需要一个模型来表明覆盖的含义。第一步是要求部分或者所有对象被实例化。图2.24就是由图2.23中对每个类实例化一个对象的继承等级图。

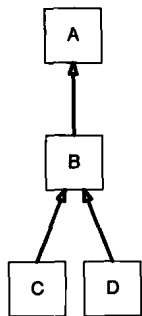


图2.23 简单的继承等级关系图

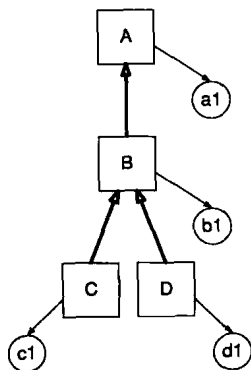


图2.24 对象初始化后的层次继承图

一个图的节点覆盖就是要求为每个类至少实例化一个对象。然而, 这里根本没有谈到执行所以不具说服力。从逻辑上扩展一下就是要求对每个类的每个实例, 根据上面的调用覆盖标准, 调用图都必须被覆盖。所以, 面向对象的调用图标准也可以称为“聚合的标准”, 因为它要求将调用标准应用于每个类的至少一个对象。

它的一个扩展是所有对象调用 (all object call) 标准, 这个标准要求对于每个类的每个实例化对象调用标准都满足。

2.4.2 设计元素的数据流覆盖

在设计元素上进行连接控制简单而直接, 但是基于它的测试对于发现错误通常不是很有效。另一方面, 数据流连接通常很复杂并且很难分析。对于一个测试人员来说, 这应该立刻说明数据流连接是发现软件错误的丰富源泉。主要的问题在于def和use到底在哪里? 当测试程序单元时, def和use在同一单元中。在集成测试中, def和use在不同的单元中。本节将开始讲述标准的编译器/程序的分析术语。

一个单元调用其他单元, 那么前者称为调用方 (caller), 后者称为被调用方 (callee)。调用的语句就称为调用点 (call site)。实参是在调用方中, 它的值赋给被调用方中的形参。这两个单元之间的接口就是将实参和形参关联起来。

设计元素的数据流测试标准的前提是为了使得各集成的程序单元之间的接口能正常通信, 必须保证在调用方单元中定义的变量在被调用方单元中能被恰当地使用。我们可以把这种技术限于单元接口中, 这样只需要注意以下两种情况: 变量在调用和从被调用单元返回前的最后一次定义; 对变量的调用和从被调用单元返回后的第一次使用。

图2.25解释了数据流覆盖标准所测试的关系。这种标准要求从通过调用的实参定义处一直执行到形参的使用。

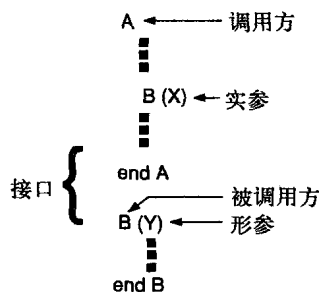


图2.25 参数耦合举例

定义了三种形式的数据流耦合。最显而易见的是参数耦合，这些参数是在调用的时候被传递；共享数据耦合是指当两个单元访问同一个全局或非局部的数据对象；外部设备耦合是当两个数据单元访问同一个外部媒介（如一个文件）。下面所有的例子和讨论都是针对参数的，它的概念同样适用于共享数据耦合和外部设备耦合。我们使用通用术语耦合变量来表示在一个单元定义而在另一个单元中使用的变量。

这种形式的数据流只关注调用和返回前的last-def以及调用或返回后的first-use。也就是说，只关注紧跟方法调用前后的def和use。调用前的last-defs是在调用处到达使用的定义位置，返回前的last-def是在到达返回语句时的定义位置。下面的定义中我们假设变量要么已经在调用方定义，要么已经在被调用方定义，并且在另一方中使用。

定义2.40 last-def: 节点的集合，它定义了一个变量 x ，该变量从调用处的节点到另一个单元的使用处有一条定义清纯的路径。

变量可以作为参数、返回值或者共享变量的引用被传递。如果函数没有返回语句，那么假设在方法的最后一条语句之后将会存在一个隐式的返回语句。

first-use的定义是对last-def定义的补充。它依赖于一些路径，它们不但是定义清纯的，而且还是使用清纯的。从 n_i 到 n_j 的一条路径，对于变量 v 如果对于路径上的每个节点 n_k ($k \neq i$ 且 $k \neq j$)， v 都没有被使用，那么这条路径就是使用清纯的。假设变量 y 在一个单元中定义后，在另一个单元中使用。假设变量 y 接收到从另外一个单元中传递的值，这个值可以通过参数传递，返回语句、共享数据或其他值方式传递。

定义2.41 first-use: 使用 y 的所有节点中存在一条定义清纯并使用清纯的路径，路径起点是入口（如果使用是在被调用方）或者是调用点（如果使用是在调用方）。

图2.26中的函数 $F()$ 是调用方，函数 $G()$ 是被调用方。调用点一共有两个du-pair；函数 $F()$ 中的 x 被传递到函数 $G()$ 中的 a ，函数 $G()$ 中的 b 被返回并在函数 $F()$ 中被赋给 y 。注意，函数 $F()$ 中的 y 被赋值并不是显示使用，而是转移的部分。它是在 $print(y)$ 语句中使用的。

这个定义允许当一个返回值没有被赋值给其他变量的不规则情况，就像 $print(f(x))$ 语句这样。在这种情况下，会有隐式的赋值，first-use也就是在 $print(y)$ 语句中。

图2.27就是在两个部分CFG图中的两个单元之间的last-def和first-use。左边的单元就是调用方，调用 B ，实参 X 被赋给形参 y 。 X 在节点1、2和3中被定义，但是在节点1处的定义不能达到节点4处的调用，所以变量 X 的last-def集是{2, 3}。形参 y 在节点11、12和13处被使用，但是从调用的入口节点10到节点13没有使用清纯的路径，所有 y 的first-use集是{11, 12}。

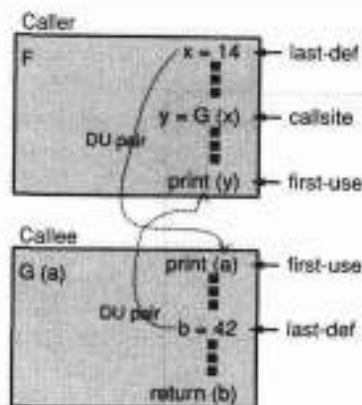


图2.26 耦合du-pairs

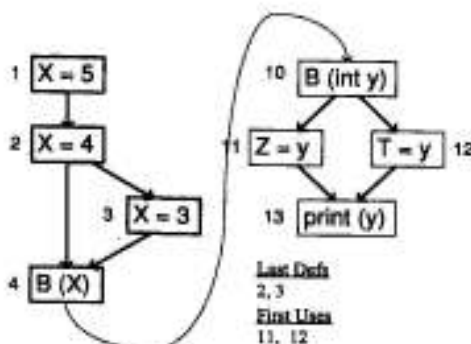


图2.27 last-def和first-use

回忆一下一个du-path就是从def到use出现在同一个图中的路径。这个概念被提炼成关于耦合的变量x的耦合du-path。耦合du-path是指从一条last-def到first-use的路径。

2.2.2节中的覆盖标准同样适用于耦合图。All-Defs覆盖要求一个路径能从每个last-def至少执行到其中一个first-use处，All-Defs又叫做All-Coupling-Def覆盖。All-Uses要求一条路径能从每个last-def执行到每个first-use处，All-Uses又叫做All-Coupling-Use覆盖。

最后，All-du-Paths覆盖要求我们游历每条简单路径，从每个last-def到每个first-use。如前面，All-du-Paths标准可以由包含侧访的游历满足。在这种情况下，又叫做All-Coupling-du-Paths覆盖。

举例

现在我们用例子来解释耦合数据流。图2.28的二次方类就是根据三个整型的系数计算二次根的方程。在主函数第34行中传递三个参数调用Root()。在主调用方，X、Y、Z中的每个变量都有三个last-def (16、17、18行，23、24、25行和30、31、32行)。它们对应于Root()中的三个形参A、B和C。这三个变量在第47行都有first-use。在被调用方定义了类的变量Root1和Root2，在主调用方被使用。它们的last-def分别是在第53、54行，first-use是在第37行。

局部变量Result被返回到主调用方。在第50、55行有可能的两个last-def，在第35行有first-use。

耦合的du-pair可以通过三元组对列出来。每个三元组都有单元名、变量名和一个行号。一个元组对中的第一个三元组是变量定义处，第二个是变量使用处。经典的二次方程的完整耦合du-pair如下：

```
(main(), X, 16) --> (Root(), A, 47)
(main(), Y, 17) --> (Root(), B, 47)
(main(), Z, 18) --> (Root(), C, 47)
(main(), X, 23) --> (Root(), A, 47)
(main(), Y, 24) --> (Root(), B, 47)
(main(), Z, 25) --> (Root(), C, 47)
(main(), X, 30) --> (Root(), A, 47)
(main(), Y, 31) --> (Root(), B, 47)
(main(), Z, 32) --> (Root(), C, 47)
(Root(), Root1, 53) --> (main(), Root1, 37)
(Root(), Root2, 54) --> (main(), Root2, 37)
```

```
(Root(), Result, 50) -- (main(), ok, 35)
(Root(), Result, 55) -- (main(), ok, 35)
```

```

1 //计算两个数的二次方根的程序
2 import java.lang.Math;
3
4 class Quadratic
5 {
6     private static double Root1, Root2;
7
8     public static void main (String[] argv)
9     {
10         int X, Y, Z;
11         boolean ok;
12         if (argv.length == 3)
13         {
14             try
15             {
16                 X = Integer.parseInt (argv[1]);
17                 Y = Integer.parseInt (argv[2]);
18                 Z = Integer.parseInt (argv[3]);
19             }
20             catch (NumberFormatException e)
21             {
22                 System.out.println ("Inputs not integers, using 8, 10, -33.");
23                 X = 8;
24                 Y = 10;
25                 Z = -33;
26             }
27         }
28         else
29         {
30             X = 8;
31             Y = 10;
32             Z = -33;
33         }
34         ok = Root (X, Y, Z);
35         if (ok)
36             System.out.println
37             ("Quadratic: Root 1 = " + Root1 + ", Root 2 = " + Root2);
38         else
39             System.out.println ("No solution.");
40     }
41
42     // 要找到二次方根, A必须非0
43     private static boolean Root (int A, int B, int C)
44     {
45         double D;
46         boolean Result;
47         D = (double)(B*B) - (double)(4.0*A*C);
48         if (D < 0.0)
49         {
50             Result = false;
51             return (Result);
52         }
53         Root1 = (double) ((-B + Math.sqrt(D)) / (2.0*A));
54         Root2 = (double) ((-B - Math.sqrt(D)) / (2.0*A));
55         Result = true;
56         return (Result);
57     } // 结束方法Root
58
59 } // 结束方法Quadratic

```

图2.28 二次方根程序

有几条有关耦合数据流的注意事项，其中有两项很重要需要记住：首先，只考虑在被调用方中定义或使用的变量，也就是说对于测试来说有last-def但没有相应的first-use是没有意义的；其次，我们必须注意全部变量和类的隐式初始化。在一些语言中（比如Java和C），类和

实例的变量为给定的默认值。当这些定义在适合单元的开始处时，能够对这些定义进行建模。比如，类级的初始化可以认为是在main()中或者构造函数中。虽然访问类的变量的其他的方法可能会在第一次调用中使用默认值，但是这些方法也可能使用被其他方法赋予的值，因此我们就需要使用标准耦合数据流分析的方法。同样，这种方法没有专门考虑“转移du-pair”，也就是说，如果单元A调用B，B调用C，在A中的last-def就不会到达C中的first-use。这种类型的分析采用的是当前昂贵的并且价值也值得怀疑的技术。最后，数据流测试用到了传统的访问数组的做法。即使在有限的情况下定义并跟踪数组的引用是非常困难并且代价都是昂贵的，所以大部分的工具都将对数组中一个元素的引用作为整个数组的引用。

继承与多态（高级话题）

前面的讨论主要是关于方法级上的最常用的数据流测试的形式。然而，在主调用方和被调用方之间伴有耦合的数据流，是唯一的一个数据的使用—定义对的复杂的集合的一种类型。图2.29是我们一直都在讨论的du-pair类型。左边是方法A()，它包含了use和def。(在这里的讨论，我们将省略变量并假设所有的du-pair都引用同一个变量。)右边就是两种类型的过程间(inter-procedural)的du-pair。

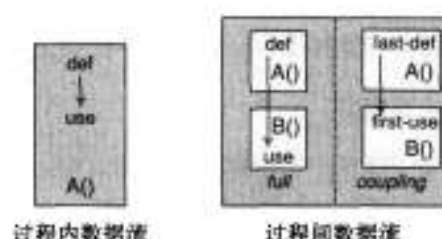


图2.29 在intra-procedural和inter-procedural数据流中的def-use对

完全inter-procedural数据流能识别所有调用方(A())和被调用方(B())之间的du-pair。耦合的inter-procedural数据流在2.4.2节中有描述，只能识别last-use和first-def间的du-pair。

图2.30就是面向对象软件中的du-pair。DU对通常是基于为类所定义的对象或者状态变量。左边的图就是面向对象的du-pair的直接情况。耦合方法F()调用了两个方法，A()和B()。A()定义了一个变量，该变量在B()中被使用。对于要引用相同的变量，A()和B()必须在相同的对象实例的引用下被调用。也就是说，如果调用是o.A()方法和o.B()方法，那么它们是在同一个o的实例上下文下调用的。如果这些调用不是在相同的实例下进行的，那么它们的定义和使用是针对变量的不同实例。

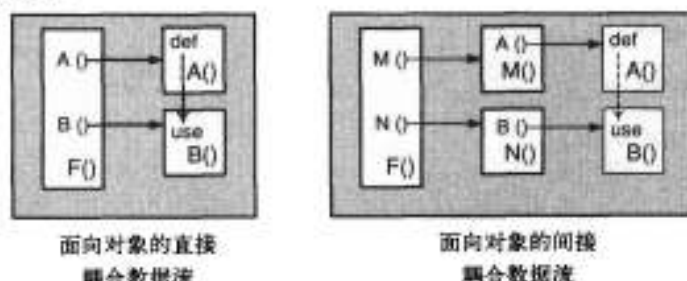


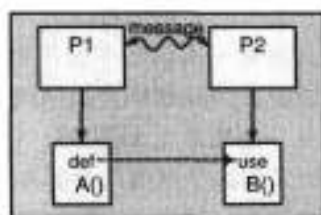
图2.30 面向对象软件中的def-use对

在图2.30的右边是间接的du-pair。在这个场景中，耦合方法F()调用两个方法M()和N()，这两个方法又分别调用另外两个方法A()和B()。定义和使用是在A()和B()中，所以引用是非直接的。相对直接du-pair来说，解释间接du-pair要复杂很多。很明显，在耦合方法与def和use方

法之间的调用很可能不只一个。

在面向对象的数据流测试中，方法A()和B()可能会在同一个类中，它们也可能在不同的类但访问相同的全局变量。

最后，图2.31解释了分布式系统中的du-pair。P1和P2可以是两个进程、线程或者其他分布式系统的组件，它们调用方法A()和B()，而在A()和B()中又定义和使用相同的变量。这种分布式通信可以采取任何一种有效的方法进行，包括HTTP（基于Web的）、远程调用（RMI）或者CORBA。A()和B()可以在同一个类中，也可以是访问如Web会话这样的持久的变量或临时的数据存储。但是由于这种“松耦合”软件的du-pair有可能很少，识别它们并找到定义清纯的路径，然后设计测试用例来覆盖它们就更复杂了。



分布式软件数据流

“消息”可以是HTTP、RMI或其他机制。A()和B()可以在同一个类或存取永久型变量，比如在网络的一次会话中。

图2.31 在Web应用和其他分布式系统中的def-use对

2.4节练习

1. 使用2.5节中的图2.34和图2.35的Stutter类回答如下(a)~(d)问题。

- 画出类Stutter的控制流图。
- 列出所有调用点。
- 列出每个调用点的所有du-pair。
- 设计类Stutter的测试数据以能满足All-Coupling-Use覆盖。

2. 使用下面的程序段回答(a)~(e)问题。

```

public static void f1 (int x, int y)
{
    if (x < y) { f2 (y); } else { f3 (y); }
}
public static void f2 (int a)
{
    if (a % 2 == 0) { f3 (2*a); }
}
public static void f3 (int b)
{
    if (b > 0) { f4(); } else { f5(); }
}
public static void f4() {... f5();...}
public static void f5() {... f6();...}
public static void f6() {...}
  
```

使用如下输入：

- $t1 = f1(0, 0)$
- $t2 = f1(1, 1)$
- $t3 = f1(0, 1)$
- $t4 = f1(3, 2)$
- $t5 = f1(3, 4)$

- 为该段代码画出调用图。
- 求出图中每个测试的测试路径。

- (c) 找出满足节点覆盖的最小测试用例集。
- (d) 找出满足边覆盖的最小测试用例集。
- (e) 给出图的最大主路径，求出不能被上述任何用例覆盖的最大主路径。

使用如下的方法trash()和takeOut()回答问题(a)~(c)。

```

1 public void trash (int x)  15 public int takeOut (int a, int b)
2 {                          16 {
3   int m, n;                17   int d, e;
4                             18
5   m = 0;                   19   d = 42*a;
6   if (x > 0)                20   if (a > 0)
7     m = 4;                  21     e = 2*b+d;
8   if (x > 5)                22   else
9     n = 3*m;                23     e = b+d;
10  else                      24   return (e);
11    n = 4*m;                25 }
12  int o = takeOut (m, n);
13  System.out.println ("o is: " + o);
14 }

```

- (a) 使用已有的行号给出所有调用点。
- (b) 求出所有last-def和first-use对。
- (c) 提供满足all-coupling-uses（注意trash()只有一个输入）的测试输入。

2.5 规格说明的图覆盖

测试工程师也可以利用软件规格说明作为获取图的一种途径。文献列举了多种生成图的技术以及覆盖这些图的标准，但是其中的许多技术事实上是十分相似的。在程序代码中包括很多的类，而一个类中又包括很多种方法。在一个类的众多方法之间存在着一种执行顺序的约束，我们称为顺序约束（sequencing constraint）。我们将从基于顺序约束的图入手，然后再关注表示软件状态行为的图。

2.5.1 顺序约束测试

在2.4.1节中，我们指出类的调用图通常不是连通的。而且在许多情况下，例如带有抽象数据类型（abstract data types, ADTs）时，类中的方法根本不会共享调用。然而，调用顺序几乎总是受到规则的约束。例如对于大多数ADTs需要在使用之前初始化，只有向栈中压入一个栈元素，栈才能弹出元素；又如只有向队列中放置一个队列元素，才能从队列中移除元素。上述这些规则在方法的调用顺序上施加了约束。总的来说，顺序约束就是一条规则，它在某些方法的调用顺序上施加了某种限制。

顺序约束有时会以显式的方式予以表达，有时以隐式的方式予以表达，甚至有时根本就不被表达出来。有时把顺序约束编码成一种前置条件或规格说明，而不是直接地作为一种顺序条件。例如，考虑下面DeQueue()方法的前置条件：

```

public int DeQueue ()
{
    // 前置条件：队列中至少有一个元素

```

```
public EnQueue (int e)
{
    // 后置条件: 元素e处于队列尾
```

尽管没有给出说明, 明智的程序员能够推断: 一个元素置于队列中的唯一方法就是之前是否已经调用了EnQueue()方法。因此, 在EnQueue()和DeQueue()两种方法之间就存在着一种隐式的顺序约束。

当然, 形式化规格说明能够有助于使得这些关系更为精确。在形式化规格说明可用时, 明智的测试工程师理所当然地就会加以利用。但是, 即使在没有显式地给出形式化规格说明时, 负责任的测试工程师也一定会寻找形式化的关系。在这里也需要注意顺序约束无法捕获所有行为, 而只能抽象某些关键方面。在使用DeQueue()之前需要调用EnQueue()这个调用顺序约束无法捕获这样的事实: 如果我们仅调用EnQueue()方法一次向队列中加入一个元素, 接下来尝试两次调用DeQueue()方法从队列中移除两个元素, 这个队列将为空。前置条件可能会捕获到上述情况, 但是通常不是通过自动工具软件中使用的一种规范的方法实现。这种类型的关系已经超过了简单的顺序约束的能力, 但可以通过在下一节中提到的某种状态行为技术加以解决。

在测试过程中有两处需要使用这种关系。我们通过一个小例子来说明一下。类FileADT封装了三个操作文件的方法:

- open(String fName) // 打开文件名为fName的文件
- close(String fName) // 关闭文件, 使其为不可用
- write(String textLine) // 将一行文本写到文件中

这个类中包含有几个顺序约束。这些约束语句以非常明确的方式使用了“必须”和“应该”这两个词。“必须”意味着违反约束就是一种故障。“应该”意味着违反约束可能是一种潜在故障, 但不一定是。

1. 在每次write(t)操作之前都必须执行一次open(F)操作。
2. 在每次close()操作之前都必须执行一次open(F)操作。
3. write(t)操作一定不能在close()操作之后执行, 除非在close()操作和write(t)操作之间出现一次open(F)操作。
4. 一次write(t)操作应该在每次close()操作之前执行。
5. 在close()操作之后一定不能再次执行close()操作, 除非在两次操作之间出现一次open(F)操作。
6. 在open(F)操作之后一定不能再次执行open(F)操作, 除非在两次操作之间出现一次close()操作。

在测试过程中通过两种方式使用上述约束, 基于2.3.1节的控制流图 (CFG) 对使用类的软件 (一个客户端) 进行评估。思考图2.32中的两个 (部分) 控制流图, 它们表示了两个使用了FileADT类的代码单元。我们可以利用这两个图, 通过检测违反执行顺序 (sequence violations) 类型故障的方法对FileADT类的使用进行测试。测试过程既可以是静态的, 也可以是动态的。

静态检查 (被认为不是传统测试) 是通过检查每项约束条件进行的。首先, 考虑图2.32a中的节点2和节点5处的write(t)语句。我们可以检查是否存在从节点1的open(F)语句到节点2和

节点5的路径（约束1）。我们也可以检查是否存在从节点1的open(F)语句到节点6的close()语句（约束2）。对于约束3和约束4，我们可以检查是否存在一条从节点6的close()语句到任意write(t)语句的路径，并且检查是否存在一条从open(F)语句到close()语句，但没有穿过至少一个write(t)的路径。这将会发现一个可能会出现的问题，路径[1, 3, 4, 6]是从一个open(F)语句到一个close()语句，而其间没有出现write(t)调用。

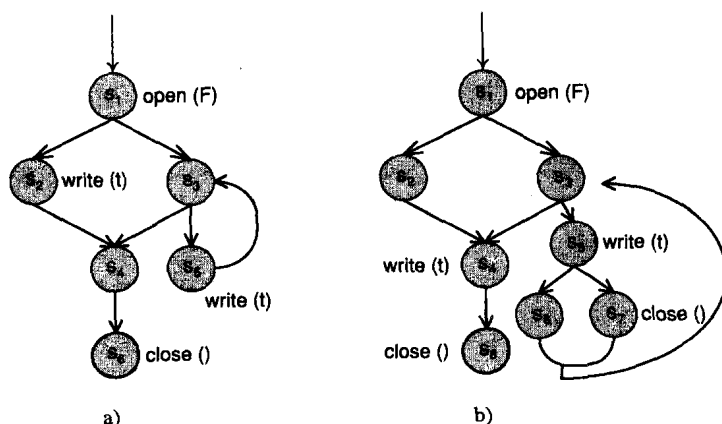


图2.32 使用文件ADT的控制流图

对于约束5，我们可以检查是否存在一条从close()语句到close()语句，但没有穿过open(F)的路径。对于约束6，可以检查是否存在一条从open(F)到open(F)，但没有穿过close()的语句。

这个过程在图2.32b中会发现一个更严重的问题。在节点7的close()语句到节点5和节点4的write(t)语句之间存在一条路径。对于规模较小的图来说，这一点看起来似乎很简单而不需要借助形式化手段。然而，对于一个包含数十个或者上百个节点的大图来说，寻找路径的过程就变得十分困难。

动态测试遵循一个略有不同的方法。思考图2.32a中的问题：在可能的路径[1, 3, 4, 6]上没有出现write()。很有可能程序逻辑控制除非执行循环[3, 5, 3]至少一次，否则不可能执行边(3, 4)。因为是否执行路径[1, 3, 4, 6]是形式上无法判定的，因此这种状态只能通过动态执行的方式加以检测。那么，我们建立测试需求，尝试去违反顺序约束。对于FileADT类，我们建立如下的测试需求集：

1. 覆盖每条从开始节点到包含一个write(t)语句的节点的路径，使得这样的路径不会穿过包含open(F)语句节点。
2. 覆盖每条从开始节点到包含一个close()语句的节点的路径，使得这样的路径不会穿过包含open(F)语句节点。
3. 覆盖每条从包含一个close()语句的节点到包含一个write(t)语句的节点的路径，使得这样的路径不包含open(F)语句。
4. 覆盖每条从包含一个open(F)语句的节点到包含一个close()语句的节点的路径，使得这样的路径不穿过包含write(t)语句的节点。
5. 覆盖每条从包含一个open(F)语句的节点到包含一个open(F)语句的节点的路径。

当然，上述所有测试需求在一个编写完美的程序中是不可行的。然而，如果存在上述任何一种缺陷，那么根据测试需求建立的测试就会将缺陷暴露出来。

2.5.2 软件状态行为测试

使用基于规格说明的图的另外一个主要的方法是通过建立某种形式的有限状态机 (FSM)，对软件的状态行为进行建模。在过去的25年中，对于如何生成有限状态机以及如何测试基于有限状态机软件提出了许多建议。如何生成、绘制、解释一个有限状态机的话题几乎充满全部教材。同时，作者们开展了大量的深入研究工作以定义怎样确切地进入一个状态、如何到达一条边以及如何产生状态转换。我们没有使用某种特定的符号表示法来表示有限状态机，而是为其定义了一个非常通用的模型，它适合于任一种符号表示法。这类有限状态机基本上是图，而且已定义的图测试标准可以用于测试基于有限状态机的软件。

基于有限状态机的测试好处之一就是大量的实际软件应用是基于有限状态机的，或者是可以用有限状态机建模。实际上，所有的嵌入式软件均适合于这一类别，包括远程控制软件、家用电器、手表、汽车、手机、飞行导航、交通信号灯、铁路控制系统、网络路由器和工厂自动化。实际上，有限状态机可以对大多数的软件建模，主要的限制条件是建模软件所需要的状态数量。例如，文字处理软件包括大量的命令和状态，将其建模成有限状态机是不切实际的。

建立有限状态机通常会有很大的价值。如果测试工程师创建一个有限状态机描述当前软件，他几乎可以肯定会在软件中发现故障。有人甚至会提出反面说法，如果设计人员创建了有限状态机，那么测试人员就不会为创建它们而烦恼了，因而测试人员面临的问题将少之又少。

有限状态机可以使用不同类型的活动 (action) 进行标注，包括转换上的活动、进入节点的活动 (entry action)、离开节点的活动 (exit action)。有多种建模语言可以用于描述有限状态机，包括UML状态图、有限自动机、状态表 (SCR) 以及petri网。本书给出了一些带有基本特征的例子，这些特征是多数语言所共有的特征。与UML状态图十分相近，但不完全一样。

一个有限状态机就是一幅图，图中的节点表示软件运行行为中的状态，边表示状态之间的变迁。一个状态表示一种可识别的情况，它将在一段时间内一直存在。一个状态通过一组变量的特定值加以定义，只要该组变量的值未发生变化，那么软件就始终处于该状态之下。（注意，这些变量是在建模设计阶段进行定义，不需要与软件中的变量相互对应。）一个变迁 (transition) 可以看成是用0时间发生，通常代表单个或若干个变量的值的变化。当变量发生变化时，认为软件从前置状态 (pre-state) (前驱状态, predecessor) 变迁到后置状态 (post-state) (后继状态, successor)。(如果一个变迁的前驱状态与后继状态相同，那么状态变量的值将不会发生变化。) 有限状态机通常在变迁上定义前置条件 (precondition) 或监控条件 (guard)，即定义使该转换能发生的特定变量的取值和定义触发事件。触发事件 (triggering event) 是指引发变迁发生的变量值的变化。触发事件“触发”状态变化。例如，SCR建模语言调用WHEN条件和触发事件。在变迁之前，触发事件的值为before-values，而变迁之后的值为after-values。在绘制图的时候，经常使用监控条件和发生变化的值来标注变迁。

图2.33是具有一个简单变迁的模型，这个变迁就是打开电梯门。如果按下电梯按钮（触发事件），只有在电梯不动的情况下（前置条件，elevSpeed = 0），电梯门打开。

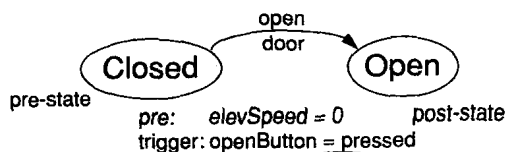


图2.33 电梯开门转换

给定这幅图，上面的一些标准可以直接进行定义。节点覆盖要求有限状态机中的所有节点至少被访问一次，这称为状态覆盖（state coverage）。边覆盖要求有限状态机中的每个变迁至少被访问一次，这称为变迁覆盖（transition coverage）。最初，针对有限状态机定义了边对（edge-pair）覆盖标准，也称为变迁对（transition-pair）覆盖和两路（two-trip）覆盖。

针对有限状态机的数据流覆盖标准有些麻烦。在多数的有限状态机公式中，节点不允许带有变量的定义和使用。也就是说，所有的活动都建立在变迁之上。与基于代码的图不同，来自有限状态机中相同节点的不同边不需要有相同的定义和使用集。此外，触发器的语义暗示相关变量的变化所产生的效果立即被感觉到，并通过变迁到达下一个状态。即触发变量的def立即到达use。

这样的话，All-Def和All-Use标准仅能应用于监控条件中的变量。这样一来也带来了一个更为实际的问题：有限状态机有时无法对所有状态变量的赋值建模。也就是说，在有限状态机中清楚地标记出状态变量的使用，但是不是总能容易找到状态变量定义的地方。由于上述原因，很少人尝试将数据流标准应用到有限状态机当中。

获取有限状态机图

将图技术应用到有限状态机的困难之一首先是得到软件的有限状态机模型。正如我们前面所说，软件的有限状态机模型可能已经存在，也可能不存在。如果不存在，测试人员就可能通过获取有限状态机模型来加深对软件的理解。然而，如何才能获取到有限状态机却不是那么显而易见，因而我们提供一些建议，但这不是构建有限状态机的一个完整的指导方法，实际上，确实有一些关于这个课题的完整的资料，我们推荐有兴趣的读者可以到其他地方进一步学习。

本节提供了一些简单而直截了当的建议以帮助那些不熟悉有限状态机的读者入门，并且避免某些明显的使用错误。我们用一个可运行的例子来解释这些建议，见图2.34和图2.35中的Stutter类。类Stutter检查文档中每对相邻的单词，并且打印出完全相同的一对相邻单词。

类Stutter有一个主方法和三个支持方法。学生们可以根据自己的设计来选择四个策略中的任何一种，从代码生成有限状态机。接下来将对它们进行逐一讨论。

1. 组合控制流图。
2. 利用软件结构。
3. 对状态变量建模。
4. 利用隐式或显式的规格说明。

组合控制流图：对于不太了解有限状态机的程序员来说，这是一种最为自然的获取有限状态机的途径。我们的经验显示如果不加强制，大多数的学生都会使用这种方法来获取有限状态机。图2.36给出了类Stutter的一个基于控制流图的有限状态机。

图2.36中的图根本不是有限状态机，并且这也不是从软件形成图的方法。这个方法存下面几个问题，第一问题就是图中的节点不是状态。方法必须返回到恰当的调用点，这意味着图中包含内在的不确定性。例如，在图2.36中，在checkDupes()的节点2到stut()的节点6之间有一条边，同时在checkDupes()的节点2到stut()的节点8之间也有一条边。哪一条边将被执行取决于从stut()的节点6还是从stut()的节点8的边进入到checkDupes()。第二个问题是在建立图之前，首先要实现类Stutter；回忆一下，在第1章中曾说过我们的目标之一就是尽可能早地准备好测试。然而，更为重要的是这种类型的图不适用大型软件产品。对于如此小的类Stutter

来说，这幅图就已经过于复杂了。对于大型软件产品来说，这样的情况就变得更糟。

```

/** *****
// Stutter检查文本中的重复单词，
// 按行号打印出重复单词
// Stutter接受标准输入或
// 文件名列表
// Jeff Offutt, June 1989 (in C), Java version March 2003
// ***** */
class Stutter
{
    // 在多个方法中被使用的类变量
    private static boolean lastdelimit = true;
    private static String curWord = "", prevWord = "";
    private static char delimits [] =
        {',', '.', ':', ';', '\'', '"', '+', '=', '/', '\\', '?',
         '&', '!', '}', '\n'}; // 列表中的第一个字符是制表符

    /** *****
    // main函数解析参数，确定参数是一个文件
    // 还是标准输入，并且调用Stut()方法
    // *****
    public static void main (String[] args) throws IOException
    {
        String fileName;
        FileReader myFile;
        BufferedReader inFile = null;

        if (args.length == 0)
        { // 不是文件，使用标准输入
            inFile = new BufferedReader (new InputStreamReader (System.in));
        }
        else
        {
            fileName = args [0];
            if (fileName == null)
            { // 不是文件名，使用标准输入
                inFile = new BufferedReader (new InputStreamReader (System.in));
            }
            else
            { // 文件名，打开文件
                myFile = new FileReader (fileName);
                inFile = new BufferedReader (myFile);
            }
        }

        stut (inFile);
    }

    /** *****
    // Stut()读取输入流中的所有行，并且
    // 查找单词。单词由delimits[]数组中
    // 定义的分隔符分隔。每次到达单词尾
    // 部，调用checkDupes()检查当前单词
    // 是否与前一个单词相同
    // *****
    private static void stut (BufferedReader inFile) throws IOException
    {
        String inLine;
        char c;
        int linecnt = 1;

```

图2.34 Stutter-A部分

```

while ((inLine = inFile.readLine()) != null)
{ // 循环每一行

    for (int i=0; i<inLine.length(); i++)
    { // 循环每一个字符
        c = inLine.charAt(i);

        if (isDelimit (c))
        { // 检查是否到达单词尾部
            checkDupes (linecnt);
        }
        else
        {
            lastdelimit = false;
            curWord = curWord + c;
        }
    }
    linecnt++;
    checkDupes (linecnt);
} // Stut方法结束

//*****
// checkDupes()检查全局变量curWord的
// 值是否与变量prevWord的值相同, 如果
// 相同那么打印一条消息
//*****
private static void checkDupes (int line)
{
    if (lastdelimit)
        return; // 如果已经检查, 则跳过
    lastdelimit = true;
    if (curWord.equals(prevWord))
    {
        System.out.println ("Repeated word on line " + line + ": " +
            prevWord+ " " + curWord);
    }
    else
    {
        prevWord = curWord;
    }
    curWord = "";
} // checkDupes结束

//*****
// 检查一个字符是否是一个分隔符
//*****
private static boolean isDelimit (char C)
{
    for (int i = 0; i < delimits.length; i++)
        if (C == delimits [i])
            return (true);
    return (false);
}

} // 类Stutter结束

```

图2.35 Stutter-B部分

利用软件结构：经验更为丰富的程序员可能会考虑软件中整体的操作流。这可能得到的结果如图2.37所示。

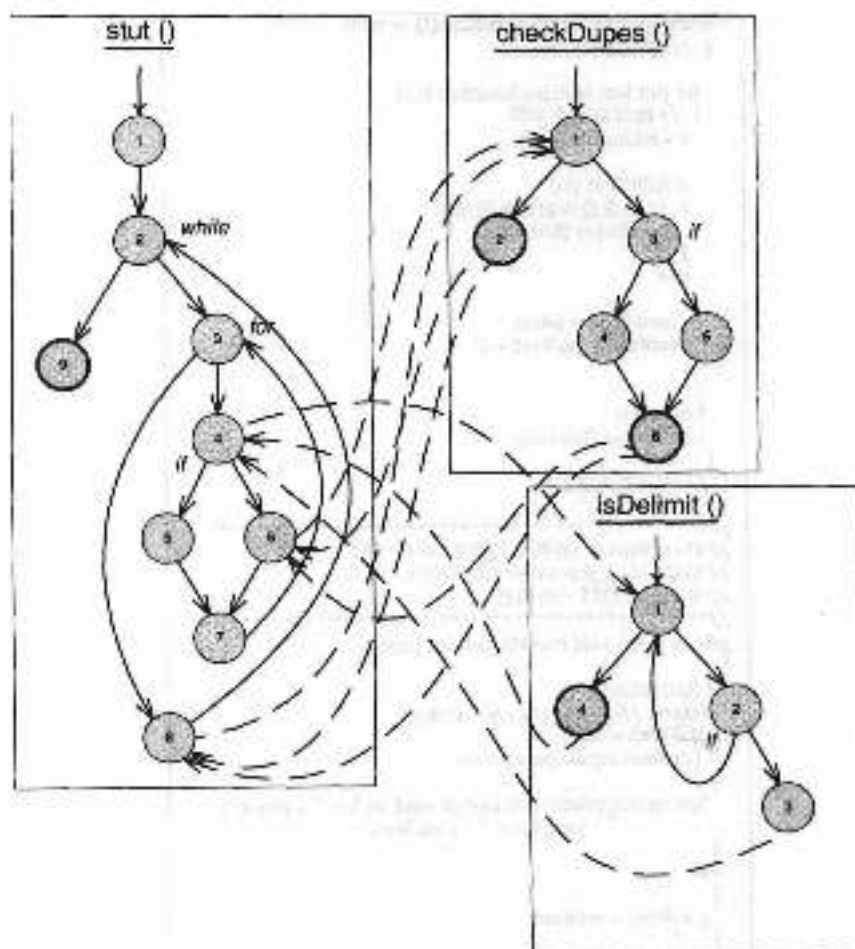


图2.36 类Stutter的基于方法的控制流图生成的有限状态机

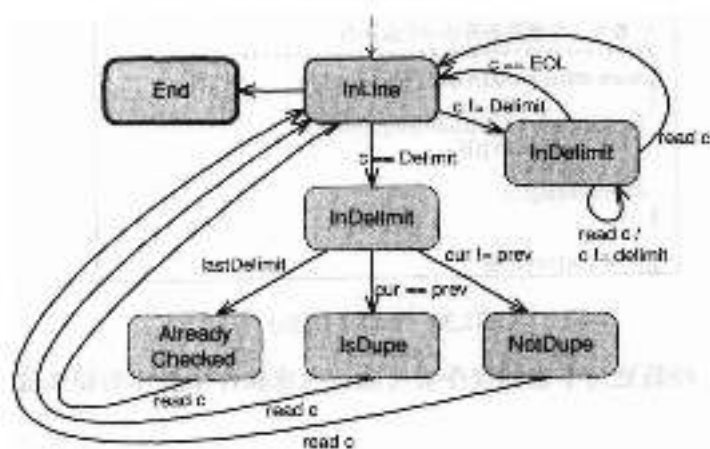


图2.37 类Stutter的基于软件结构与成的有限状态机

尽管对于控制流图有了一些改善，但是图2.37中所展示的这种变换方法太过于主观。不同

的测试人员会绘制出不同的图，这样一来在测试过程中就引入了不一致性。它也要求测试人员对软件有深入的了解，而且必须要等到详细设计完成之后才能开展。同样，这样得到的图也很难适用于大型程序。

状态变量建模：一种获取有限状态机的更机械的方法是考虑程序中状态变量的值。状态变量值早在设计阶段就已经进行了定义。第一步就是要标识状态变量，接下来选择哪些状态变量与有限状态机相关（例如，全局变量或类变量）。类Stutter定义了四个状态变量：lastdelimit、curWord、prevWord和delimits。为了方便，变量delimits是定义在类层次上的，但是不应该被看成是状态的一部分。事实上，在设计阶段没有把该变量包含进去也是有可能的。而lastdelimit、curWord和prevWord确实是真正的状态变量。

从理论上来看，这三个变量的不同取值的组合定义了不同的状态。而实际上，这会产生无限多的状态。例如，curWord和prevWord是字符串，并且有无限多个值。那么，标识出可以作为状态表示的取值或取值范围是很普遍的方法。对于类Stutter，显而易见的方法就是考虑两个变量之间的关系，有如下几种可能的取值：

```
curWord: undefined, word
prevWord: undefined, sameword, differentword
lastdelimit: true, false
```

其中，word是任意一个字符串。这种值的组合将产生12种可能的状态：

1. (undefined, undefined, true)
2. (undefined, undefined, false)
3. (undefined, sameword, true)
4. (undefined, sameword, false)
5. (undefined, differentword, true)
6. (undefined, differentword, false)
7. (word, undefined, true)
8. (word, undefined, false)
9. (word, sameword, true)
10. (word, sameword, false)
11. (word, differentword, true)
12. (word, differentword, false)

不是每种组合都是有可能的。例如，curWord变量给定一个值，之后都不会变为undefined。那么，唯一可能的curWord变量为undefined值的状态是状态1，恰好这个状态也是初始状态。一旦读到一个字符，变量curWord的值就变为word，同时变量lastdelimit设置为false（状态8）。当发现一个分隔符，变量prevWord的值与变量curWord的值或者相同（sameword），或者不同（differentword）（状态10或12）。如果两个词相同，下一个读取的字符就会改变变量curWord，那么软件就转换到状态12。通过确定所有可能的状态转换的方式能够获得到完整的状态机。图2.38给出了类Stutter完整的有限状态机。注意，因为不可能到达状态7，因此在图中将其省略。同时注意，程序在文档的结尾发现分隔符时终止。因此每一个包含lastdelimit=true的状态就是结束状态。

这个策略的较为机械的过程是很吸引人的，因为我们能期待不同的测试人员能够产生相

同或相似的有限状态机。与此同时，想要将该过程完全自动化也是不可能的。因为通过源代码确定变迁存在困难，同时决定哪些变量需要建模也需要做出判定。实现软件不一定想获取这样的图样，但需要软件设计方案。通过状态变量建模手段得到的有限状态机有时不能确切地反映实际的软件。

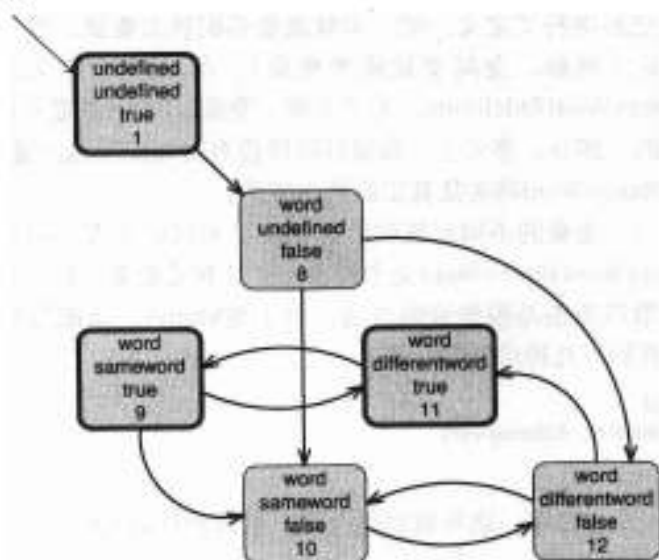


图2.38 类Stutterer的基于状态变量建模生成的有限状态机

利用隐式或显式的规格说明：最后一种获取有限状态机的方法依赖于描述软件行为的显式的需求或形式化规格说明。图2.39展示了类Stutterer的基于这种方法得到的一个有限状态机。

这个有限状态机看起来非常像基于代码的有限状态机，这就是我们想要得到的结果。与此同时，基于规格说明的有限状态机通常更纯正，也更容易理解。如果软件设计得好，这种类型的有限状态机应该包含UML状态图中所包含的信息。

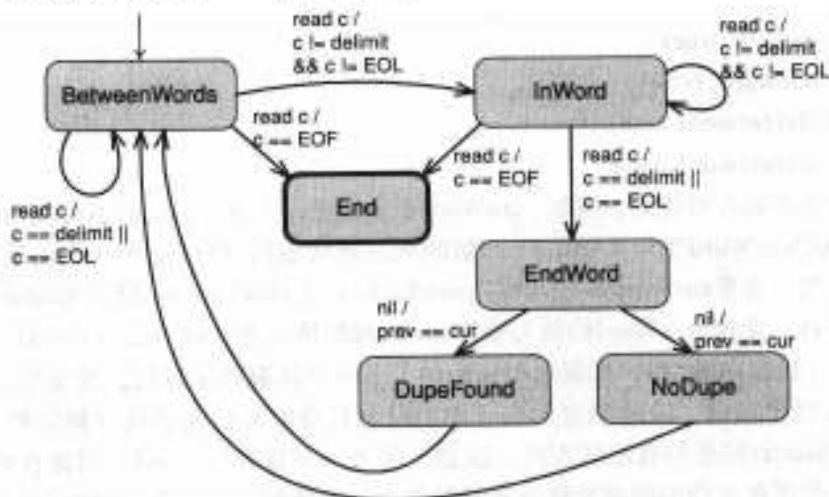


图2.39 类Stutterer的基于规格说明生成的有限状态机

2.5节练习

1. 使用图2.40中的队列 (Queue) 类回答以下(a)~(f)问题。该队列用通用循环模式管理。

```
public class Queue
{ // 概述: 队列是易变的、有边界的先入先出的数据结构,
  // 数据大小是固定的 (在此题中是2), 典型的队列是[1,
  // [o1], [o1, o2], 其中o1和o2都非空, 行入队的元素比
  // 后入队的元素先出队
  private Object[] elements;
  private int size, front, back;
  private static final int capacity = 2;

  public Queue ()
  {
    elements = new Object [capacity];
    size = 0; front = 0; back = 0;
  }

  public void enqueue (Object o)
    throws NullPointerException, IllegalStateException
  { // 修改: this
    // 作用: 如果参数为空, 抛出NullPointerException异常,
    // 否则如果队列满, 抛出IllegalStateException异常,
    // 否则使o成为this指针的最新元素
    if (o == null)
      throw new NullPointerException ("Queue.enqueue");
    else if (size == capacity)
      throw new IllegalStateException ("Queue.enqueue");
    else
    {
      size++;
      elements [back] = o;
      back = (back+1) % capacity;
    }
  }

  public Object dequeue () throws IllegalStateException
  { // 修改: this
    // 作用: 如果队列为空, 抛出IllegalStateException异常,
    // 否则删除和返回最旧的元素

    if (size == 0)
      throw new IllegalStateException ("Queue.dequeue");
    else
    {
      size--;
      Object o = elements [ (front % capacity) ];
      elements [front] = null;
      front = (front+1) % capacity;
      return o;
    }
  }

  public boolean isEmpty() { return (size == 0); }
  public boolean isFull() { return (size == capacity); }

  public String toString()
  {
    String result = "[";
    for (int i = 0; i < size; i++)
    {
      result += elements[ (front + i) % capacity ] . toString();
      if (i < size - 1) {
        result += ", ";
      }
    }
    result += "]";
    return result;
  }
}
```

图2.40 队列类

假设要建立有限状态机，状态由队列类变量表示。也就是说，状态用4元组函数 `[elements, size, front, back]` 的值来表示。例如，初始状态有值 `[[null, null], 0, 0, 0]`，把一个对象由初始状态放入队列中，它的状态转变为 `[[obj, null], 1, 0, 1]`。

(a) 实际上，我们并不关心到底是哪个对象在队列中。因此，对于一个变量 `elements`，只有4个有用的值。列举出这些值。

(b) 有多少个状态？

(c) 这些状态中，有多少是可以达到的？

(d) 将可达到的状态用图表示出来。

(e) 为 `enqueue()` 和 `dequeue()` 增加边。（在这个任务中，忽略异常的返回情况，尽管你能观察到异常发生时，没有任何实例变量被修改。）

(f) 定义一个小的测试集完成边覆盖。实现并执行这个测试集，你会发现它对于写一种在每次调用的时候显示内部变量状态的方法是有帮助的。

2. 对于以下(a)~(c)问题，考虑用有限状态机方法设计一个简单的可编程的自动调温器。假设定义这些状态的变量和状态之间转换的方法如下：

```
partOfDay : {Wake, Sleep}
temp      : {Low, High}
```

```
// 起初, "Wake" 在 "Low" 温度
```

```
// 作用: 进入一天的下一部分
public void advance();
```

```
// 作用: 如果可能, 将当前temp置为高
public void up();
```

```
// 作用: 如果可能, 将当前temp置为低
public void down();
```

(a) 上述问题有多少个状态？

(b) 画图并标出这些状态（用变量值）和转换（用方法名）。注意，图中要包括所有的方法。

(c) 一个测试用例就是仅仅一个方法调用的序列。给出满足你画出的图的边覆盖的测试集。

2.6 用例的图覆盖

UML用例图被广泛用于说明和表达软件需求。它们用来描述软件根据用户输入执行的动作序列。也就是说，它们可以帮助表达计算机应用程序的工作流。因为用例是在软件开发的早期创建的，因此可以帮助测试人员较早期地开始测试。

很多书和文献都可以帮助读者设计用例。像介绍有限状态机一样，本书的目的并不是解释如何设计用例，而是怎样使用它们进行有用的测试。根据用例使用图覆盖标准来开发测试用例的技术可以用一个简单的例子来表达。

图2.41表示ATM机的3个简单的用例。在用例中，角色是使用所建模型软件的人和其他软件系统。它们被画成简单的棍状人形图，在图2.41中，角色是拥有三个潜在用例的ATM客户，三个用例分别是取款（Withdraw Funds）、查询余额（Get Balance）和转账（Transfer Funds）。

图2.41从技术上讲是一张图，但它对于测试而言并不是非常有用。作为测试人员我们所能做的最好是使用节点覆盖，等同于对每一个用例测试一次。然而用例经常被细化或者用一个更详尽的文本描述使其文档化。这个文档描述了操作的细节并且包括了可供选择的方案，用于建立在执行过程中的选择或者条件的模型。图2.41中的取款用例可以用下面文字来描述：

用例名：取款

摘要： 客户使用一张有效的卡从一个有效的账户中取款

用户： ATM客户

前提条件： ATM正在显示空闲时的欢迎信息

描述：

1. 客户向ATM读卡器中插入一张卡。
2. 如果系统能够识别这张卡，读出卡号。
3. 系统提示客户输入密码。
4. 客户输入密码。
5. 系统检测卡的有效期并检测其是否为挂失卡。
6. 如果该卡有效，系统检测客户输入的密码是否和卡的密码匹配。
7. 如果密码匹配，系统找出允许该卡访问的账户。
8. 系统显示客户账号并且提示客户选择一种交易方式。三种类型的交易方式为取款、查询余额和转账。（前八步是三种用例的共同部分，后面的步骤是只属于取款用例。）
9. 客户选择取款项，选择账户号并且输入金额。
10. 系统检查金额是否有效，保证客户的账户中有足够的金额，保证取款金额没有超出日取款额上限，然后检查ATM中是否有足够的钞票。
11. 如果所有的检查通过，系统给出现金。
12. 系统打印带有交易序号、交易类型、取款金额和新的账户余额的收据。
13. 系统弹出卡。
14. 系统显示空闲时的欢迎信息。

可选项：

- 如果系统不能够识别卡片，弹出卡片并显示欢迎信息。
- 如果卡片有效期已过，没收卡片并且显示欢迎信息。
- 如果卡片被报告丢失或者被盗，没收卡片并显示欢迎信息。
- 如果客户输入的密码与卡的密码不匹配，系统提示重新输入的密码。
- 如果客户三次输入不正确的密码，没收卡片并且显示欢迎信息。
- 如果客户输入的账号是无效的，系统报告错误信息、弹出卡片并且显示欢迎信息。
- 如果客户请求的取款金额超过日最大取款金额，系统显示抱歉信息，弹出卡片显示欢迎信息。
- 如果客户请求的取款金额超过ATM机里的金额，系统显示抱歉信息，弹出卡片显示欢迎信息。

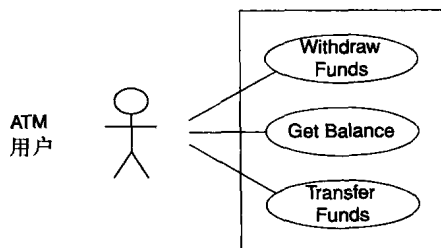


图2.41 ATM用户和用例

- 如果客户输入取消，系统取消交易，弹出卡片显示欢迎信息。
- 如果客户请求的取款金额超过了账户内的金额，系统显示抱歉信息，取消交易，弹出卡片显示欢迎信息。

后置条件：资金已经从客户账户中取出。

在这里，一些测试的学生会奇怪为什么在本章包含图覆盖策略的讨论。这是因为迄今为止几乎和图没有明显的关系。我们想要重申Beizer的警告的第一句话：“测试人员找到一个图，然后覆盖它。”实际上，在用例的文本描述中有一个很好的图结构，这个图结构正好是测试人员想要表达的。它能够像Berzer的第4章的事务流图那样被模型化，或者能够像UML活动图那样被画出来。

一个活动图给出了活动的流动。活动能够用来把多种事情模型化，包括状态变化、返回值和计算。我们提倡通过在用户层次思考这些活动来建立用例的模型图，活动图有两种节点，分别是活动状态和顺序分支。^⑤

我们像下面这样构造活动图，用例描述的数字项用来表达角色行为的步骤，这相当于软件的输入或者输出，活动图中的节点代表行为状态。用例图中的可选项代表软件或参与者做出的决定，用活动图中的节点来表示作为顺序分支。

图2.42是取款场景的活动图。由用例构成的活动图的某些元素是可选的而不是必需的。首先，活动图里通常没有太多的循环，而且其中的大多数循环都是严格限制或者是确定循环次数的。举个例子，图2.42中当密码输入不正确的时候包含了一个三层的循环。这说明完全路径覆盖通常是可行的而且有时是合理的。其次，包含多个子句的复杂的断言是很少见的。因为用例通常是按条目表达，这样方便用户理解。这说明第3章的逻辑覆盖标准通常是无用的。第三，没有明显的数据du-pair，这说明数据流覆盖标准是不适用的。

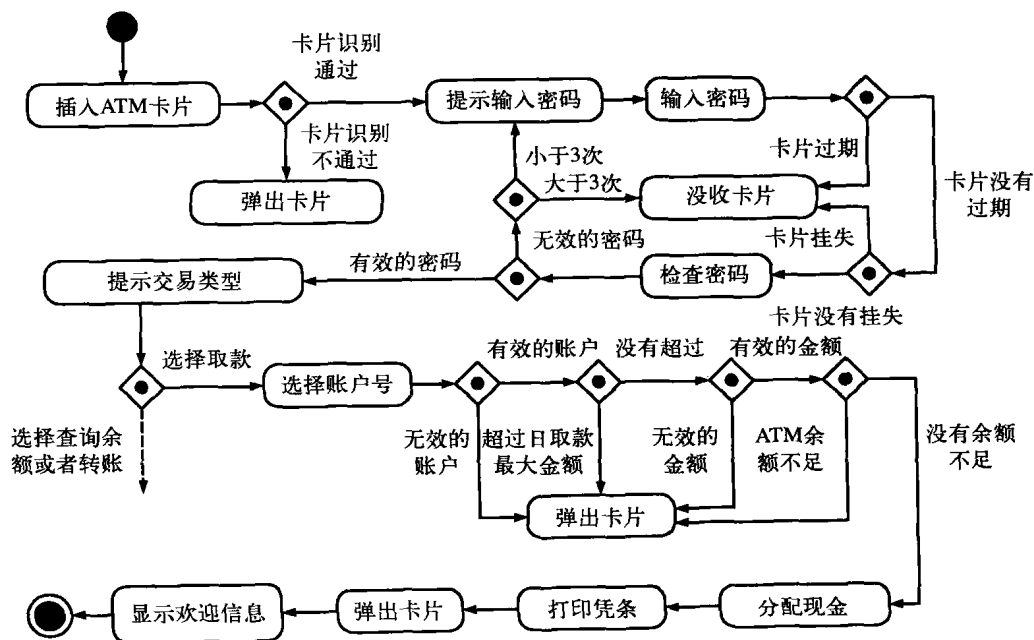


图2.42 ATM取款的活动图

⊖ 在前面的章节中，我们明确地不考虑并发的情况，所以并发的分叉和合并我们不考虑。

对于用例图明显适用的两个标准是节点覆盖和边覆盖。测试用例的值来自对作为软件输入的节点和断言的解释。用例图的另一个标准是基于“场景”的概念。

2.6.1 用例场景

用例场景就是一个用例的实例或者一个完整的路径。场景对于用户来说应该有一些语义并且通常是在用例被构造的时候推导出来的。如果用例图是有限的（通常情况都是这样），那么就有可能列出所有的场景。然而，领域知识能够用来减少场景的数量，这对于建模或者从测试用例的角度看是有帮助和有意义的。注意在本章开篇定义的指定路径覆盖，正是我们所需要的。指定路径覆盖的集合S就是所有场景的集合。

如果测试人员或者制定需求的人选择所有可能的路径作为场景，那么指定路径覆盖等价于完全路径覆盖。场景是由人选择的，人们依赖于领域知识。这样不能保证指定路径覆盖涵盖边覆盖或者节点覆盖。也就是说，有可能选择一个不能包含所有边的场景的集合，然而这很可能是一个错误。因此，在实际情况中，指定路径覆盖能够覆盖所有的边。

2.6节练习

1. 为银行自动取款机交互构造两个单独的用例和用例场景。不要试图在一张图中捕获ATM机的所有功能；考虑两种类型的人使用ATM时分别可能做什么。
2. 为你的场景设计测试用例。

2.7 用代数方法表示图

我们通常把图看成是圆圈和箭头的组合，但它们也能够用多种非图示的方式表示。一种有效的方法是代数表示，能够使用标准代数操作和转换成正则表达式对代数表示进行处理，这些操作作为测试软件的基础并用来回答关于图的各种问题。

首先需要给每一条边唯一的标号或者名称。边的名称来自已经和边关联的标签，或者可以根据代数表达式添加。这本书假定标签用唯一的小写字体表示。图代数中的乘法操作符表示串联，如果边 a 连接边 b 表示成 ab （操作符“ $*$ ”省略掉）。加法操作符表示选择；如果选择边 a 或者边 b ，那么它们的和是 $a + b$ 。一系列相关的边组成一条路径，因此把边的序列叫做路径积。一个路径表达式包括路径积和零或者更多的“ $+$ ”操作符。这样每一条路径积是一个路径表达式。注意，边的标签是一个没有乘法操作的路径积的特殊情况，而且路径积是没有加法操作的路径表达式的特殊情况。路径表达式有时候用大写字母表示，如 $A = ab$ 。

图2.43给出了三个图的例子，一个是双菱形图，一个是循环遍历图，另一个Stutter例子来自前面的小节。图2.43a显示出四条路径；图2.43b和图2.43c包含循环，所以并非所有的路径都被表示出来。在图代数中，循环最好用指数来表示。

如果一条边、路径积或路径表达式被重复使用，那么用指数做标记。因此 $a^2 = aa$ ， $a^3 = aaa$ ， $a^* = aa \cdots a$ ，也就是说，指数代表循环次数。一个特殊情况是，空值或者零长度的路径表示成 $a^0 = \lambda$ 。这使得 λ 成为乘法的单位元，所以 $a\lambda = a$ ，或者更一般地 $A\lambda = A$ 。

用大写字母代表路径或者部分路径使得处理更加简单。举个例子，我们可以在下面的图2.43b中采用部分路径表达式：

$$A = ab$$

$$\begin{aligned} B &= eg \\ C &= cd \\ AB &= abeg \\ C^3 &= cdcdcd \end{aligned}$$

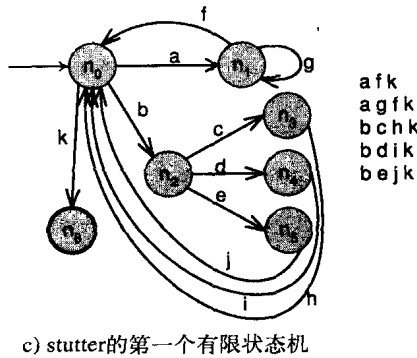
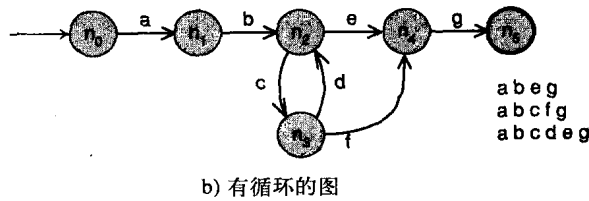
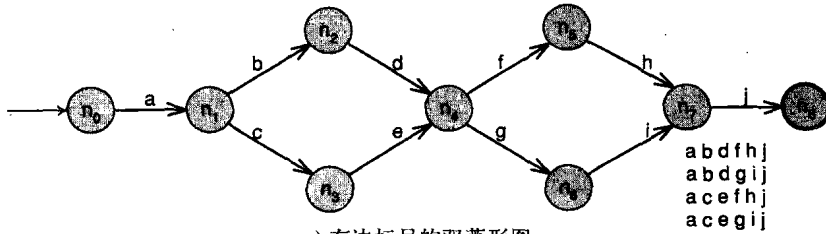


图2.43 路径积的例子

$$\begin{aligned} AC^2B &= ab(cd)^2eg \\ &= abcdcddeg \\ D &= be + bcf \end{aligned}$$

不同于标准代数，路径积并不满足交换律，即 $AB \neq BA$ ，但是满足结合律，即 $A(BC) = (AB)C = ABC$ 。

在上面图2.43a的所有路径能够用表达式 $abdfhj + abdgij + acefhj + acegij$ 表示出来。相加的路径之间可以被认为是独立的或者是并行的路径。因此路径和既满足交换率又满足结合率，也就是说： $A + B = B + A$ ， $(A + B) + C = A + (B + C) = A + B + C$ 。

根据这个原则，我们就可以应用标准代数法则。分配率和吸收率都可以使用。

$$A(B + C) = AB + AC \quad (\text{分配率})$$

$$(B + C)D = BD + CD \quad (\text{分配率})$$

$A + A = A$ (吸收率)

对于重复或循环我们还有两个速记符号。如果一个循环至少执行一次 (如repeat-until 结构), 那么使用指数符号 “+”。即 $AA^* = A^+$ 。如果一个循环有一个确定的边界我们还可以设置循环的边界 (如for循环), 用下划线这样表示: $A^{\underline{1}} = A^0 + A^1 + A^2 + A^3$, 或者更一般地 $A^{\underline{n}} = A^0 + A^1 + \dots + A^n$ 。有时候在符号 “-” 两端设置循环的边界很有用, 也就是循环至少 m 次, 至多 n 次, 为了表示这样的情况, 我们引入 $A^{\underline{m-n}} = A^m + A^{m+1} + \dots + A^n$ 。

吸收符号可以使用一些方法合并指数符号, 它通常用来简化路径表达式, 如下所示:

$$A^n + A^m = A^{\max(n,m)}$$

$$A^n A^m = A^{n+m}$$

$$A^n A^* = A^* A^n = A^*$$

$$A^n A^+ = A^+ A^n = A^+$$

$$A^* A^+ = A^+ A^* = A^+$$

乘法单位元操作符 λ , 也能够用来简化路径表达式。

$$\lambda + \lambda = \lambda$$

$$\lambda A = A \lambda = A$$

$$\lambda^n = \lambda^n = \lambda^* = \lambda^+ = \lambda$$

$$\lambda^+ + \lambda = \lambda^* = \lambda$$

我们同样需要加法单位元。我们使用 ϕ 代表不含任何路径 (甚至没有空路径 λ) 的路径集合。从数学角度看, 任何路径表达式加上 ϕ 还是原来的表达式。加法的 ϕ 可以看做在图中封锁了一些路径, 而形成的空路径。

$$A + \phi = \phi + A = A$$

$$A\phi = \phi A = \phi$$

$$\phi^* = \lambda + \phi + \phi^2 + \dots = \lambda$$

图2.44是一个具有空路径的小图, 如果我们列出从 n_0 到 n_3 的所有路径, 我们得到路径表达式 $bc + a\phi = bc$ 。

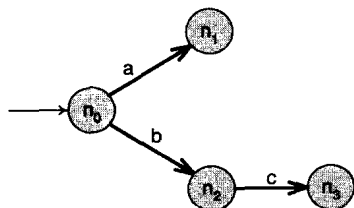


图2.44 导致加法单位元的空路径

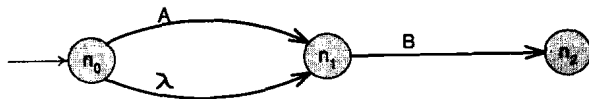


图2.45 A或 λ

2.7.1 把图简化成路径表达式

既然已经有了基本的工具, 我们可以看看怎样着手把任意图简化成路径表达式。这里给出的方法并不是一个严格的算法, 因为它需要一些思考和判定, 但是对于使用它的测试人员来说已经足够好了。据我们所知这种方法还没有被工具自动化和实现, 然而, 它是从确定性的有限状态机构造正则表达式的通用技术的一种特殊情况。这种方法在图2.46中进一步阐述。

第1步：首先合并所有的连续边，相乘边的标签。进一步形式化，对于只有一条入边和一条出边的节点，消除这个节点，合并它的两条边，把它们的路径表达式相乘。对图2.46应用这一步合并边 h 和 i ，得到图2.47。

第2步：接下来合并所有平行的边，添加边的标签。进一步形式化，对于任何具有相同源节点和目标节点的边对，合并成一条边，把它们的路径表达式相加。图2.47包含一个这样的边对 b 和 c ，所以它们被合并成 $b+c$ ，得到图2.48。

第3步：通过创建一个新“哑节点”引入一条有指数操作符“ $*$ ”边的方法去除自循环（从一个节点到它自己），然后用乘法合并这三条边。进一步形式化，对于任何有一条标签为 X 的边指向自己的节点 n_1 ，入边 A 和出边 B ，消除标签为 X 的边，增加一个新节点 n'_1 和标签为 X^* 的边。然后合并这三条边 A 、 X^* 和 B 成为一条边 AX^*B （消去节点 n_1 和 n'_1 ）。图2.48包含一个自循环的节点 n_3 ，边的标签是 e 。这条边首先用节点 n'_3 替换，从 n_3 到 n'_3 的标签是 e^* （如图2.49a所示），然后合并标签 d 、 e^* 和 f 如图2.49b所示。

第4步：现在测试人员开始选择移除节点。选择一个非初始节点也非终节点的点。通过插入它所有的前驱节点到所有的后继节点的边替换掉它，把入边到出边所有的路径表达式相乘。图2.50说明了这种方法，其中有一个节点具有两条入边和两条出边。

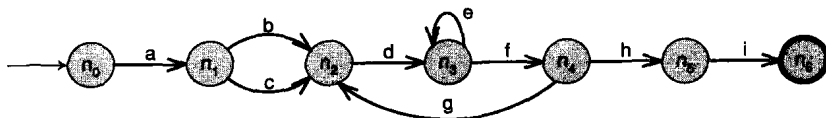


图2.46 约减路径表达式的实例图

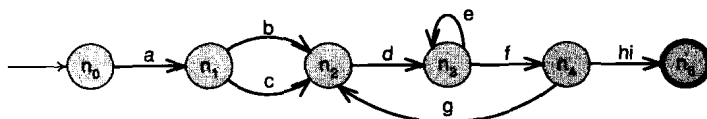


图2.47 第1步之后的路径约减图

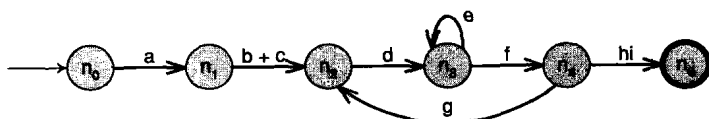
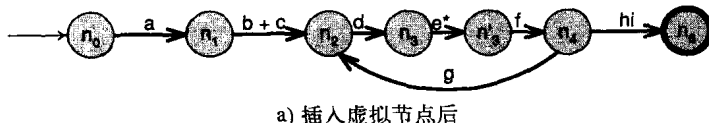
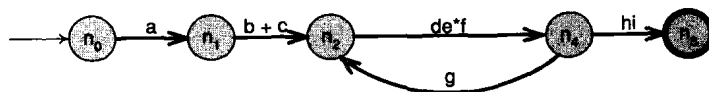


图2.48 第2步之后的路径约减图



a) 插入虚拟节点后



b) 合并边之后

图2.49 在路径约减图的第3步之后

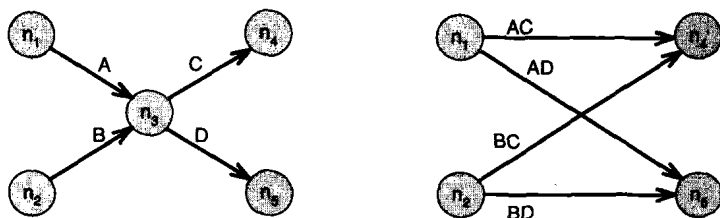


图2.50 移走任意节点

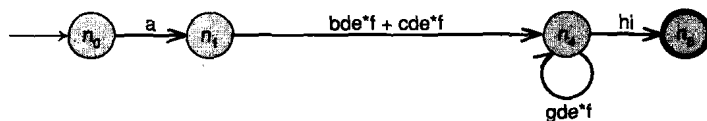
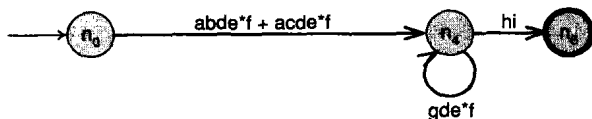
图2.51 排除节点 n_2 

图2.52 移走顺序的边

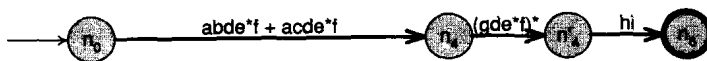


图2.53 移走自循环的边

图2.49b中的节点 n_2 有两条入边和一条出边。边 (n_1, n_2) 和边 (n_2, n_4) 变成了边 (n_1, n_4) ，两个路径表达式相乘，边 (n_4, n_2) 和边 (n_2, n_4) 变成了一个自循环边 (n_4, n_4) ，两个路径表达式相乘。结果如图2.51所示。

第1步到第4步被循环执行直到图中只有一条边。对图2.51应用第1步得到了图2.52。

第2步（合并平行边）被跳过因为图2.52没有平行边。对图2.52应用第3步（移去自循环），移走了一个自循环节点 n_4 ，得到了图2.53。图2.54是我们这个例子中的最后一个图（和正则表达式）。

2.7.2 路径表达式的应用

既然数学方面的准备工作已经就绪，接下来我们就要考虑能用路径表达式来做什么？路径表达式是抽象的、形式化的图的表述。就其本身而言，它们能被用来告诉我们有关所表述图的信息。本节介绍几个路径表达式的应用。

2.7.3 得到测试输入

利用图的路径表达式的最直接的方式是定义覆盖测试用例。每一条路径，也就是路径表达式所定义的每一条路径积，都应该在一个合适的循环限制下被执行。这是特殊路径覆盖（Specified Path Coverage, SPC）的一种形式。如果一个无边界的幂指数（“*”）出现在路径表达式中，它可以被一个或者多个适当的定值所替代，然后将得到一个完整的路径列表。这种技术将确保（即涵盖）图中的节点覆盖和边覆盖。

从例图2.46到图2.54得到的最终的路径表达式是 $abde^5f(gde^5f)^5hi+acde^5f(gde^5f)^5hi$ 。这个表达式有两个单独的路径集，并且幂计算可以被定值5代替。这样就产生了以下两个测试需求： $abde^5f(gde^5f)^5hi$ 和 $acde^5f(gde^5f)^5hi$ 。

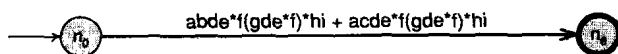


图2.54 一个路径表达式的最终图

2.7.4 在流图中计算路径数并确定最大路径长度

有时候知道一个图中的路径个数是很有用的。它能被用做一个简单的复杂度的度量或者覆盖这个图所需要的测试用例数的一个粗略估计。路径表达式允许通过简单的算术运算来产生一个适当的最大路径数的近似值。

正如前面讨论过的，当一个图有一个循环的时候，理论上这个图有一个无限大的路径条数。但是，一些图没有循环，并且领域内的知识可以被用来对重复次数加以适当的限制。这个限制可能是一个真实的循环的最大值，也可能表达一个测试人员的假设：执行这个循环“N次”就已经足够了。

第一步是用一个权值来标记每一条边。对于大部分的边来说，权值只有一个。如果这条边代表一个昂贵的运算，比如一个方法或者一个外部过程，那么这条边应该被标记为那个运算的近似的权值（例如，方法中路径的数目）。如果一条边代表一个循环，那么就用它可能的最大循环次数（循环权值）来标记它。无限次的循环也是可能的，它意味着这个图中的路径的最大数目是无穷大的。要记住，并不是一个循环中的所有边都被标记为循环权值。一个循环中只有一条边被标记就可以了。有时候，哪条边应该被标记是很明显的，但也有些时候测试人员必须选择一个合适的边来标记，既不能漏掉一个循环，也不能将一个循环标记两次。我们看一下图2.43中的图b和图c，很明显，图b中的循环权值应该被标记在边d上。在图c中循环权值应该被标记在边h、i、j、f和g上。边f经常会出现包含边g的路径中，所以很容易会忘记这些循环权值中的一个。但是，它们却代表了不同的循环。

有时候我们想对一个循环进行注释来标明它能够被执行多少次。标记“0~10”表示这个循环可以被执行0到10次(包含的)。注意这里的注释并不等同于边的权值。

接着，我们要计算这个图的路径表达式并将权值替换成路径表达式。像人们可能期望的那样，要用到操作符。如果路径表达式是 $A+B$ ，那么就替换成 W_A+W_B 。如果路径表达式是 AB ，那么就替换成 W_A*W_B 。如果路径表达式是 A^n ，那么就替换成总和 $\sum_{i=0}^n W_A^i$ 。如果路径表达式是 A^{m-n} ，那么就替换成总和 $\sum_{i=m}^n W_A^i$ 。

图2.55展示了一个有边标记和边权值的简单的图。按照边d上的注释，这个循环将被执行0~2次，并且边d的权值是1。最终的路径表达式是 $a(b+c)(d(b+c))^{0-2}e$ 。

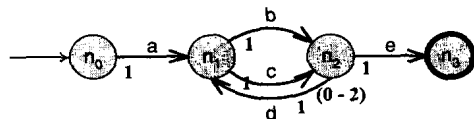


图2.55 计算路径最大值实例图

通过将路径表达式中每条边的标记用适当的值替换，我们就可以计算出路径的最大值。

$$\begin{aligned}
& 1 * (1 + 1) * (1 * (1 + 1))^{0-2} * 1 \\
&= 1 * 2 * 2^{0-2} * 1 \\
&= 2 * \sum_{i=0}^2 2^i * 1 \\
&= 2 * (2^0 + 2^1 + 2^2) * 1 \\
&= 2 * (1 + 2 + 4) * 1 \\
&= 2 * 7 * 1 \\
&= 14
\end{aligned}$$

图中最长的路径的长度也能被发现。如果路径表达式是 $A+B$ ，就用 $\max(W_A, W_B)$ 替代。如果路径表达式是 AB ，就用 W_A+W_B 替代。如果路径表达式是 A^n ，就用 $n*W_A$ 替代。所以图2.55中最长路径的长度是 $1+\max(1, 1)+2*(1+\max(1, 1))+1=7$ 。

我们必须记住，这些分析并不包括可行性分析。一些图可能是不可行的，所以这应该解释为路径条数的上界或者保守的限制。

2.7.5 到达所有边的路径的最小值

一个相关的问题是，想要到达所有的边，有多少路径是必须要遍历的。这个过程和计算路径个数的最大值极为相似，也是利用边的权值，但是计算过程有一点不同。

更具体地说，如果路径表达式是 $A+B$ ，那么就用 W_A+W_B 替代。但是，如果路径表达式是 AB ，就用 $\max(W_A, W_B)$ 替代。如果路径表达式是 A^n ，这时的替代就需要测试人员的判断，到底是1还是 W_A 。如果我们假设所有通过循环的路径都能在一个测试用例中执行，那结果就是1。如果不是这样，那么结果就应该是循环的权值 W_A 。第二种假设更保守，并且会得到一个更大的值。

我们再来考虑一下图2.55，假设边 d 被执行，那么边 d 前面的边肯定会被执行。也就是说，如果边 b 被执行，然后是边 d ，这个图的逻辑规定边 b 必须被再执行一次。这就说明我们必须保守地估计这个循环，产生

$$\begin{aligned}
& 1 * (2) * (1 * (2))^2 * 1 \\
&= 1 * (2) * (1 * 2) * 1 \\
&= \max(1, 2, 1, 2, 1) \\
&= 2
\end{aligned}$$

我们能看出这个图所有的边都被到达需要两次图的遍历。

2.7.6 互补运算分析

路径表达式的最后一个应用不是计数应用，而是寻找可能导致错误的异常的分析。它是基于求补运算的思想的。当它们的行为互为取反，或者其中一个必须在另一个之前被执行，那么两个运算是互补的。例如栈运算中的入栈和出栈，队列运算中的入列和出列，内存的取和存，文件打开和关闭等。

这个过程从一个图的路径表达式开始，除了替代边权值以外，每个边都要被以下的三种标记之一标记：

1. C——构造运算（入栈、入列等）

2. D ——析构运算（出栈、出列等）

3. 1 ——既不是构造运算也不是析构运算

路径表达式的乘法和加法运算符被以下两个表格 $\ominus$ 所替代：

*	C	D	1
C	C^2	1	C
D	DC	D^2	D
1	C	D	1

+	C	D	1
C	C	$C+D$	$C+1$
D	$D+C$	D	$D+1$
1	$1+C$	$1+D$	1

这里要注意和一般的代数定义整数的不同。 $C*D$ 约减为1, $C+C$ 约减为C, $D+D$ 约减为D。

下面我们来考虑图2.56, 所有边都被标记为C、D或者1, 并且它的初始路径表达式是 $C(C+1)C(C+D)1(D(C+D)1)^n1$ 。我们利用代数规则重写这个表达式得到 $(CCCC+CCCD+CCC+CCD)(DC+DD)^n$ 。我们再利用上面的两个表格可以将表达式再约减到 $(CCCC+CC+CCC+C)(DC+DD)^n$ 。

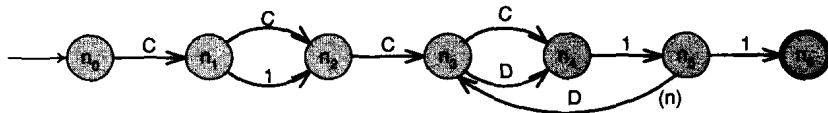


图2.56 互补路径分析实例图

关于这个路径的第一个问题是“有没有可能析构运算比构造运算多呢？”答案是肯定的，一些表达式是：

$$CCCD(DD)^n, n>1$$

$$CCD(DD)^n, n>0$$

$$CCC(DDDCDD)$$

另外一个问题是“有没有可能构造运算比析构运算多呢？”当然，答案也是肯定的，一些表达式是：

$$CCCC$$

$$CCD(DC)^n, n \text{ 取任意数}$$

每一个确定的答案都描述了一个可能导致异常行为的测试的具体说明。

2.7节练习

1. 生成并简化图2.43中的三个图的路径表达式。
2. 生成并简化图2.12中流图的路径表达式。指出适当的循环权值并计算出路径的最大值和到达所有边的最小路径数。
3. 图2.10中用作一个主测试路径的例子。增加适当的边标记，然后生成并简化路径表达式。接着为非循环边加入权值1，为循环边加入权值5。然后计算图中路径的最大值和要到达所有边的最小路径数。这个图有25个主路径。利用路径最大值简单地讨论一下主路径数并且考虑不同的循环权值对路径数最大值的影响。
4. 2.5节给出了Stutter类的4个不同版本的有限状态机。对4个版本分别生成并简化路径表达式，然后计算路径最大值和到达所有边的最小路径数。讨论不同的数量对测试的影响。
5. 在图2.32上进行互补运算分析。假设互补运算为打开和关闭。

$\ominus$ 研究抽象代数的数学家会认为这些图表定义了另一种代数。

6. 对于图2.42中的活动图，生成并简化路径表达式。图中唯一的循环有3次。计算路径最大值和到达所有边的最小路径数。讨论图中的场景和路径表达式中项之间的关系。

7. 根据以下图的定义回答问题(a)~(c)：

- $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- $N_0 = \{1\}$
- $N_f = \{10\}$
- $E = \{(1, 2, a), (2, 3, b), (2, 4, c), (3, 5, d), (4, 5, e), (5, 6, f), (5, 7, g), (6, 6, h(1-4)), (6, 10, i), (7, 8, j), (8, 8, k(0-3)), (8, 9, l), (9, 7, m(2-5)), (9, 10, n)\}$

(a) 画出这个图。

(b) 图中的路径数最大值是多少？

(c) 到达图中所有边的最小路径数是多少？

2.8 参考文献注释

在这本书的研究过程中，非常清楚的一件事是在这个领域有许多相同技术由不同的人独立、并行发现。一些人已经研究了同一技术的多个方面，后来被精化成非常好的测试标准。另外一些人发明了同样的技术，但是它们是基于不同的类型图或者改用了其他的名称。这样的话评功论赏软件测试标准是一件危险的任务。我们会尽最大的努力来解决这个问题，但是我们认为这个参考文献注释为以后更深入的文献研究开一个头。

对于覆盖图的研究往往是从有限状态机(FSM)生成测试用例开始的，有限状态机有相当长而且丰富的历史。最早有关这方面的论文是在20世纪70年代[77, 164, 170, 232, 290]。这些论文的大多数主要焦点在于使用有限状态机为那些由标准的有限状态自动机定义的通信系统生成测试用例，尽管大部分工作与一般图相关。控制流图好像于1975年已经被Legard发明了(或者应该说是被发现)[204]。在1975年已经发表的论文中，Huang[170]提出覆盖有限状态机的每条边，Howden[164]提出覆盖有限状态机的不包括循环的全部路径。1976年，McCabe[232]提出控制流图采用同样的方法作为他的循环复杂性度量的主要应用。1976年Pimont和Rault[290]提出使用术语“开关覆盖”(switch cover)来覆盖边对。1978年Chow[77]提出从有限状态机生成一棵树然后基于这棵树的路径生成测试序列。1991年Fujiwara等人[130]扩展了Pimont和Rault的边对成任意长度，使用“n-switch”来指代边序列。他把“1-switch”或开关覆盖归于Chow并把它称做“W-method”，是一个在论文中重复出现了无数次的错误。这个边对覆盖的思想在20世纪90年代再次被提出，《大英计算机软件构件测试标准》把它称为两往返[317]，Offutt等人[272]称它为“转移对”。

其他基于有限状态机的测试用例生成方法包括遍历[251]、著名的序列方法[137]和独特的输入输出方法[307]。它们的目标是基于输入驱动的状态转换检测输出错误。基于有限状态机的测试用例生成一直用来测试大量的应用程序，包括词法分析器、实时进程控制软件、协议、数据处理和电话。早期的实现，当时本书正在创作之中，有限状态机的覆盖标准和其他图的标准并没有什么不同。

本书介绍了边覆盖需求对节点覆盖需求的明确的包含关系。这种包含关系对于典型的控制流图不是必要的，的确，边覆盖包含节点覆盖经常作为一个基础定理出现，但是对于源自其他工件的图来说是必需的。

晚一些发表的论文关注点在覆盖程序中结构性的元素自动测试数据的生成[39, 41, 80, 101, 117, 166, 190, 191, 267, 295]。大部分工作是基于符号评估[62, 83, 93, 101, 116, 164]和切断[328, 339]的分析技术。这些思想的一部分将在第6章中讨论。

处理循环的问题一开始对于基于图的标准是一个非常棘手的问题。很明显地我们想覆盖路径,但是循环创建了无数多条路径。在Howden 1975年发表的论文中[164],他明确地阐明了通过覆盖没有循环的完全路径来解决循环的问题,Chow 1978年发表的论文提出使用生成树是避免执行循环的一个明确的尝试[77]。Binder的书[33]使用了Chow论文中的技术但是把其中的名字改成了“round trip”。

其他早期的观点建立在测试无循环程序上[66],从理论上讲非常有趣但是并不适用于实际。

White和Wiszniewski[348]提出限制需要基于特殊模式下执行的循环的次数。Weyuker、Weiss和Hamlet尝试选出特殊的循环来做基于数据定义和使用的测试[345]。

Offutt 等人[178, 265]提出了子路径集合的概念来支持类间路径测试,这种方法本质上等价于这里介绍的侧访和游历方法。游历、侧访和绕路而行的思想由Ammann、Offutt和Huang提出[17]。

我们找到的最早的关于数据流测试的参考书目是1974年Osterweil和Fosdick[282]的技术报告。这份技术报告引出了1976年《ACM计算调查》[122]的一篇论文和几乎同时Herman发表在《澳大利亚计算机杂志》[158]上的一篇论文。种子数据流分析程序(没有提及测试)来自Allen和Cocke[13]。

其他基础性的和理论的参考有Laski和Korel于1983年[201]提出的从定义到使用的执行路径的概念,Rapps和Weyuker于1985年[297]定义了标准和诸如All-Def、All-Use的专属名词,还有Frankl和Weyuker于1988年的成果[128]等。这些论文精炼了数据流测试的思想,这个是本文的基础。在本文中讲到对于All-du-Paths覆盖需要直接遍历[128],而对All-Def覆盖和All-Use覆盖只需要侧访。本文允许对所有数据流标准侧访(或非侧访),文中所使用的模式匹配的例子已经在这个领域中被广泛采用了几十年,据我们所知,Frankl和Weyuker[128]是第一个使用这个例子说明数据流覆盖的。

Forman还提出了一种不需要运行该程序就可以检测出数据流异常的方法[121]。

数据流测试中的一些细节问题已经重现。其中包括当定义和使用的路径不能被执行时数据流的应用[127]和处理指针与数组[267,345]。

本书源于这种根据du-path集合定义数据流标准的方法,而且明确建议最佳效果游历。

许多论文对数据流测试的多方面进行了研究,得到了一些经验的结果。其中最早的论文之一的作者有Clarke、Podgurski、Richardson和Zeil,他们比较了一些不同的标准[82]。与变异测试(第5章中介绍)的比较由Mathur在1991年[228]开始研究,其后有Mathur和Wong[230],Wong和Mathur[357],Offutt、Pan、Tewary、Zhang[274]和Frankl、Weiss和Hu[125]。把数据流测试与其他标准的比较已经由Frankl和Weiss[124],Hutchins、Foster、Goradia和Ostrand[172],Frankl和Deng[123]发表论文。

研究者们也开发了一些工具来支持数据流测试。大多数工具的工作方式为以一个程序和数据作为输入,判断一个或者多个数据流标准是否能够满足(识别器)。Frankl、Weiss和Weyuker在20世纪80年代中期[126]建立了ASSET,Girgis和Woodward在20世纪80年代中期建立了一个工具来改进数据流和变异测试[134],Laski在20世纪80年代后期建立了STAD[200]。

Bellcore的研究者们在20世纪90年代早期开发了C语言的ATAC数据流工具[161,162]，第一个包括数据流标准的测试数据生成器由Offutt、Jin和Pan在20世纪90年代晚期开发[267]。

耦合作为设计度量第一次是由Constantine和Yourdon[88]讨论提出的，而把它以隐式方式用于测试是Harrold、Soffa和Rothermel[152,154]，Jin和Offutt[178]以显式方式引入的，他们引入了first-use和last-def的使用。

Kim、Hong、Cho、Bae和Cha用基于图的方法根据UML状态图生成测试用例[186]。

美国联邦航空局意识到通过软件结构化覆盖分析的需求模块化和集成化测试越来越重要，其中结构化覆盖分析应该确认代码和构件之间的数据耦合和控制耦合[305]，第33页，6.4.4.2节。

数据流测试也被Harrold和Soffa[154]、Harrold和Rothermel[152]、Jin和Offutt[178]应用到集成测试中。工作集中在类级的集成问题，但是并不关注继承或多态。数据流测试已经由Alexander和Offutt[10,11,12]，Buy、Orso和Pezze[60,281]在面向对象软件中应用到继承和多态。Gallagher和Offutt把类建模成相互作用的状态机，测试它们之间的并发性和通信问题[132]。

SCR由Henninger[157]首先提出，由Atlee[20]在模型检查和测试中使用。

由UML图表构造测试用例为近期的发展尽管很简单。这种方法首先由Abdurazik和Offutt[2,264]提出，然后由Briand和Labiche深入研究[45]。

把有限数据转换为规则表达式的机制是计算机理论课程中的标准方法。据我们所了解，Beizer[29]是首先关注其在测试上下文中这些转换应用。

第3章 逻辑覆盖

本章介绍基于逻辑表达式的测试标准。尽管逻辑覆盖标准已经出现多年，但它们的使用最近几年才稳步增长，导致这种情况的原因之一是，它们被纳入诸如美国联邦航空管理局商用飞机的安全性至关重要的软件标准。像第2章中那样，我们首先从关于逻辑谓词和子句的完整理论基础开始，力争让后面的测试标准显得更加简单。和前面一样，我们首先大致浏览结构和标准，然后讨论如何从各种软件工件，包括代码、规格书和有限状态机中得到逻辑表达式。

一些熟悉通用标准的读者起初对于认识逻辑表达式可能有些困难。这是因为我们引入了测试标准的一般集合，并且选择了最具有说明性的标准名字。也就是说我们抽象了若干现有的标准，这些标准相互密切关联，但是使用有冲突的术语。

3.1 概览：逻辑谓词和子句

我们以数学方式定义逻辑表达式的格式。谓词是其计算结果是布尔值的表达式，也是最基本的表达式。一个简单的例子是 $((a > b) \vee C) \wedge p(x)$ 。谓词可能包含布尔变量、用操作符 $\{>, <, =, \geq, \leq, \neq\}$ 做比较的非布尔变量和函数调用。由逻辑操作符构成内部结构：

- $\neg$ 取反操作符
- $\wedge$ 和操作符
- $\vee$ 或操作符
- $\rightarrow$ 蕴涵操作符
- $\oplus$ 异或操作符
- $\leftrightarrow$ 等价操作符

对于偏爱代码的读者可能会觉得一些操作符（ $\oplus$ 、 $\rightarrow$ 、 $\leftrightarrow$ ）不习惯，但这些操作符在规格说明语言中很普遍，而且得心应手地表达自己的意思。and 和or操作符的短路（short circuit）情况在一些时候非常有用处，在必要的时候我们将讨论有关问题。我们采用通常的优先级，按照上边所列操作符从高到低匹配顺序。当优先级不明显时，我们就用括号说明。

子句是不含有任何逻辑操作符的谓词。例如，谓词 $(a=b) \vee C \wedge p(x)$ 这个谓词包括3个子句：一个关系表达式 $(a=b)$ ，一个布尔变量 C 和一个函数调用 $p(x)$ 。因为关系表达式可能含有自己的结构，所以需要特殊处理。

一个谓词可由一系列逻辑上等价的方式定义出来。例如， $((a=b) \vee C) \wedge ((a=b) \vee p(x))$ 和 $(a=b) \vee C \wedge p(x)$ 逻辑上是等价的，但是 $((a=b) \wedge p(x)) \vee (C \wedge p(x))$ 就不一样了。我们可以利用布尔代数的一般规则把布尔表达式转换为等价形式。

逻辑表达式来源广泛。其中读者最熟悉的可能是源代码。例如，如下代码：

```
if ((a > b) || C) && (x < y)
    o.m();
else
    o.n();
```

给出表达式 $((a>b) \vee C) \wedge (x<y)$ 。逻辑表达式的其他来源包括有限状态机中的变迁。例如，一个变迁 $button2=true$ （当 $gear = park$ ）可以得到表达式 $gear = park \wedge button2=true$ 。同样地，规格书中的前置条件“前置条件：堆栈不为满而且对象引用参数不为空”会得到表达式 $\neg stackFull() \wedge newObj \neq null$ 。

在3.6节以前我们不是根据语法，而是根据语义处理逻辑表达式。因而，不管逻辑表达式采用哪种方式，对于一个给定的覆盖标准给定的逻辑表达式都可以产生同样的测试需求。

3.1节练习

1. 列出下面谓词的所有子句：

$$((f \leq g) \wedge (X > 0)) \vee (M \wedge (e < d + c))$$

2. 写出谓词（只写出谓词）来表达下面的需求：“列出所有零售价在100美元以上或库存超过20件的无线鼠标。再列出售价在50美元以上的无线鼠标”。

3.2 逻辑表达式覆盖标准

利用谓词和子句来介绍一系列覆盖标准。用 P 作为一个谓词集， C 作为 P 谓词中的一个子句集。对于每个谓词 $p \in P$ 谓词集， C_p 表示 p 的子句，也就是说， $C_p = \{c | c \in p\}$ 。 C 是 P 中每个谓词子句的集合，即 $C = \bigcup_{p \in P} C_p$ 。

标准3.12 谓词覆盖 (PC)：对于每个 $p \in P$ ， TR 包括两个需求： p 赋值为真， p 赋值为假。

作为边的覆盖，谓词覆盖的图的形式我们在第2章中做了介绍，在这里，图覆盖标准和逻辑表达式覆盖标准发生了重叠。在控制流图表中 P 是与分支相关的一个谓词集，谓词覆盖等同于边界覆盖。对于以上所给的谓词， $((a>b) \vee C) \wedge p(x)$ ，可以用两个测试来完成谓词覆盖： $(a=5, b=4, C=true, p(x)=true)$ 和 $(a=5, b=6, C=false, p(x)=false)$ 。

该标准的一个明显的缺陷是不能测试每个子句。上一个子语句的谓词覆盖也可以用 $(a=5, b=4, C=true, p(x)=true)$ 和 $(a=5, b=4, C=true, p(x)=false)$ 这两个保持前两个子句一直为真的测试来完成，为了纠正该问题，我们转到子句一级。

标准3.13 子句覆盖 (CC)：对于每个 $c \in C$ ， TR 包括两个需求： c 赋值为真， c 赋值为假。

谓词 $((a>b) \vee C) \wedge p(x)$ 需要两个不同的值满足子句覆盖。子句覆盖需要 $(a>b)$ 为真一次且为假一次， C 为真一次且为假一次， $p(x)$ 为真一次且为假一次，这些要求可以用两个测试 $((a=5, b=4), (C=true), p(x)=true)$ 和 $((a=5, b=6), (C=false), p(x)=false)$ 来满足。

就像我们在谓词 $p=a \vee b$ 中所展现的那样，子句覆盖不能涵盖谓词覆盖，谓词覆盖也不能涵盖子句覆盖。子句集 C 是 $\{a, b\}$ 。4个输入项列出了子句的逻辑值组合：

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

考虑两个测试组，每组都有一对测试输入项。测试集 $T_{23}=\{2, 3\}$ 满足子句覆盖却不满足谓词覆盖，因为 p 总是为真。而测试集 $T_{24}=\{2, 4\}$ 满足谓词覆盖却不满足子句覆盖，因为 b 总是为假。这两个测试组强调了子句覆盖不能涵盖谓词覆盖，谓词覆盖也不能涵盖子句覆盖。

从测试的角度上，我们可能希望测试标准能够测试每个子句和谓词，要解决这个问题，最直接的办法是测试所有子句组合。

标准3.14 组合覆盖 (CoC): 对于每个 $p \in P$ ，对 C_p 中的子句 TR 有测试需求来评估可能的真值组合。

组合覆盖又称为多条件覆盖。对于谓词 $(a \vee b) \wedge c$ ，完整的真值表包括8种情况：

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

包括 n 个独立子句的谓词 p 有 2^n 个真值组合。因此，组合覆盖对于拥有较多子句的谓词难使用、根本不实用。所以，我们需要一个捕获每个子句影响的标准，并且只需合理数量的测试。经过一些思考^①，这些观察带来了不少有力的测试标准，它们是基于使单个子句“有效”的想法，有关“有效”在后面章节定义。具体地说，我们要检查以确定是否在某种情况下我们改变了子句使得子句影响了谓词，而事实上，子句的确影响了谓词。然后我们转去检查互补问题以确定是否在某种情况下我们改变了子句，但子句没有影响谓词，而事实上，子句的确没有影响谓词。

3.2.1 有效的子句覆盖

子句覆盖和谓词覆盖之间不能相互包含是令人遗憾的，但关于子句覆盖和谓词覆盖还有更深层的问题。具体地说，当我们在子句层次上引入各种测试时，我们也希望能对谓词覆盖有所帮助。最重要的概念是决定，也就是说在这个条件下，一个子句影响谓词的输出。尽管正规定义有些麻烦，基本思想还是很简单的：如果你翻转子句会改变谓词取值，那么就子句决定谓词。采取以下的原则来从余下的子句中找到我们的子句：我们关注的子句 c_i ，是重要子句，余下的都是次子句 c_j ， $j \neq i$ 。典型来说，为了满足给定标准，每个子句都可以依次成为重要子句。

定义3.42 决定：在谓词 p 中给定一个主子句 c_i ，如果次子句 $c_j \in p$ ， $j \neq i$ ，有取值，使得改变 c_i 的真值也改变 p 的真值，我们说 c_i 决定 p 。

注意，这个定义明显没有要求 $c_i = p$ 。在以前的定义中这个问题比较模糊，其中的一些定义要求谓词和子句取相同的值，这种解释并不实际。当用到取反操作符时，比如说，谓词是

① 实际上，这个“思考”显示了许多研究人员的集体努力，他们在几十年间发表了数十篇论文。

$p=\neg a$, 那么重要子句不可能和谓词具有相同取值。

思考一下上面的例子 $p=a \vee b$, 如果 b 取假, 子句 a 就决定了 p , 因为 p 的取值就等于 a 的取值。但是如果 b 的值取真, a 就不能决定 p 。因为不管 a 取什么值, a 都是真的。

从测试的观点来看, 我们希望每个子句的测试都是在子句能够决定谓词的环境下进行的。让我们这样思考一下, 由小组中不同的成员来管理这个团队, 只有他们进行尝试, 我们才知道他们是不是好领导。再想一下, 谓词 $p=a \vee b$, 如果在 b 决定 p 的情况下我们不变化 b 。我们就没有证据说明 b 是被正确使用的。例如, 测试集 $T_{14}=\{TT, FF\}$, 即满足子句覆盖, 又满足谓词覆盖, 对于测试 a, b 都无效。

至于说标准, 我们通行用以下定义的一般方式先给出了一个有效子句覆盖的概念, 然后我们进一步完善概念中可能产生的歧义以便最终获得形式化的覆盖标准。

定义3.43 有效子句覆盖 (ACC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 c_j , $j \neq i$, 使得 c_j 决定 p 。对于每个 c_i , TR 包括两个测试要求: c_i 赋值为真和 c_i 赋值为假。

例如, 对于 $p=a \vee b$, 我们一共用 TR 中4个要求来完成测试, 两个用于子句 a 和两个用于子句 b 。对于子句 a , 当且仅当 b 是假的时候, a 决定 p , 所以我们有二个测试需求 $\{(a=true, b=false), (a=false, b=false)\}$ 。对于子句 b 来说, 当且仅当 a 为假, b 决定 p 。所以我们有二个测试需求 $\{(a=false, b=true), (a=false, b=false)\}$ 。我们在下面的局部的真值表中总结如下:

	a	b
$c_i = a$	T	f
	F	f
$c_i = b$	f	T
	f	F

有两个测试需求相同, 所以对于谓词 $a \vee b$ 我们用3个独立的测试需求来完成谓词的有效子句覆盖, 即 $\{(a=true, b=false), (a=false, b=true), (a=false, b=false)\}$ 。这种重叠总是发生, 对于一个包含 n 个子句的谓词, 我们需要 $n+1$ 个而不是人们可能想的 $2n$ 个不同的测试需求, 就足以满足有效子句覆盖。

ACC与早期一些论文所描述的MCDC的技术很相似, 这种标准有一些不太清楚的问题, 使得很多年里都不能清晰地解释MCDC。最重要的问题是, 次子句 c_j 是否需要与主子句 c_i 取同样的真值或假值。为了解决这个问题, 引出了ACC的3个独特而有趣的特性, 对于一个简单的谓词 $p=a \vee b$, 这3个特性看起来一样。但是对于更复杂的谓词, 这3个特性就不同了。最普通的特性允许次子句有不同的值。

标准3.15 广义有效子句覆盖 (GACC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 c_j , $j \neq i$, 使得 c_j 决定 p 。对于每个 c_i , TR 包括两个测试要求: c_i 赋值为真和 c_i 赋值为假。次子句 c_j 不需要与主子句 c_i 取同样的真值或假值。

不幸的是, GACC不能包含谓词覆盖, 如下面的例子所示。

思考谓词 $p=a \leftrightarrow b$, 不管 b 取什么值, 子句 a 都决定谓词 P ; 当 a 取真, b 也取真, a 为假, b 也为假。我们对子句 b 也这么做。我们只用2个测试输入 $\{TT, FF\}$ 来完成, 而 p 总是得到真值, 所以无法完成谓词覆盖。

许多测试研究人员强烈认为, ACC应该包含PC, 所以ACC的第二种特性要求, 主子句 c_i

一个取值使 p 为真, 而主子句 c_i 另一个取值使 p 为假。注意, 如上边的定义中所说, c_i 和 p 不必非要取值相同。

标准3.16 相关有效子句覆盖 (CACC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 $c_j, j \neq i$, 使得 c_i 决定 p 。对于每个 c_i , TR 包括两个测试要求: c_i 赋值为真和 c_i 赋值为假。次子句 c_j 的取值必须保证对于主子句 c_i 的一个值 p 为真, 而对于主子句 c_i 的另一个值 p 为假。

所以对于上面的谓词 $p = a \leftrightarrow b$, 对于子句 a 测试集 $\{TT, FT\}$ 可以满足CACC测试标准和对于子句 b 测试集 $\{TT, TF\}$ 可以满足CACC测试标准。合并得到CACC测试集 $\{TT, TF, FT\}$ 。

思考例子 $p = a \wedge (b \vee c)$ 。要使 a 决定 p , 表达式 $b \vee c$ 必须取真。可以采用3种方式: b 为真 c 为假, b 为假 c 为真, b 和 c 都为真。所以, 对于子句 a , CACC可以通过两个测试输入 $\{TTF, FFT\}$ 来满足。对于 a 子句也可以有别的选择。下面的真值表列出了这些可能的选择, 真值表的行数取自前边给出的谓词的完整真值表。具体地说, 对于子句 a , 从第1、2、3行选出一个测试需求, 再从第5、6、7行选出一个测试需求可以满足CACC。当然了, 一共存在9种方法。

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

最后一种特性要求次子句 c_j 与主子句 c_i 取同样的真值或假值。

标准3.17 限制性有效子句覆盖 (RACC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 $c_j, j \neq i$, 使得 c_i 决定 p 。对于 c_i , TR 包括两个测试要求: c_i 赋值为真和 c_i 赋值为假。次子句 c_j 的取值必须保证 p 和主子句 c_i 的取值相同。

对于例子 $p = a \wedge (b \vee c)$, 在9组对于子句 a 满足CACC的测试需求中, 只有3个对于子句 a 满足RACC。根据前边给定的完整真值表, 第2行可以和第6行配对, 第3行可以和第7行配对, 或者第1行和第5行配对。因此, 有9种方式满足CACC, 而只有3种方法就可以满足RACC。

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

CACC与RACC相比较

稍后将给出满足3个标准的每一个标准的谓词示例。有一点也许不能够马上看出来的是CACC和RACC在实际应用中的区别。

一些逻辑表达式可在CACC下满足, 但在RACC下, 却找不到可用的测试需求。这些表达

式比较微妙，只有子句间包含独立的关系时才存在。也就是说，子句中的一些值的组合是被禁止的。这些情况在实际程序中经常出现，因为程序变量常常相互依赖，所以考虑这样一个例子会有所帮助。

考虑一个活塞系统，其中的活塞可以开或关，以及若干种模式，其中的两种模式是“运行的”和“待命的”，假设有以下两个限制：

1. 活塞在“运行的”模式下必须打开，其他模式则关闭。
2. “运行的”模式和“待命的”模式不能同时发生。

这引出以下子句定义：

a = “活塞是关闭的”

b = “系统处于运行状态”

c = “系统处于待命状态”

假设仅当活塞关闭并且系统状态是运行状态或待命状态时，才能执行某一个动作。即

$$P = \text{活塞关闭AND (系统状态是运行OR待命)} \\ = a \wedge (b \vee c)$$

这就是前面所分析的谓词。前面的限制可以形式化为：

1. $\neg a \leftrightarrow b$
2. $\neg(b \wedge c)$

这些限制限定了真值表的可行性。作为提醒，该谓词的完整真值表如下。

	a	b	c	$a \wedge (b \vee c)$	
1	真	真	真	真	违反限制1和限制2
2	真	真	假	真	违反限制1
3	真	假	真	真	
4	真	假	假	假	
5	假	真	真	假	违反限制2
6	假	真	假	假	
7	假	假	真	假	违反限制1
8	假	假	假	假	违反限制1

回忆一下，要使a决定p的值，则b、c有一个为真或者都为真，第一条限制规则排除了a、b取值相同的行：1、2、7和8行；第二条限制规则排除了b、c取值相同的行：1和5行。因此，唯一可行的是3、4和6行。前面提到CACC可以通过选择1、2、3中的一行和5、6、7中的一行来满足。但是，RACC需要2和6，3和7，或者1和5中的一对才能满足。所以，在这个谓词中对于a，RACC更不可行。

3.2.2 无效子句覆盖

有效子句覆盖标准注重让主子句影响它们的谓词，ACC的互补标准保证不应该影响谓词的主子句的改变，事实上也不会影响谓词。

定义3.44 无效子句覆盖 (ICC)：对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$ ，选择次子句 c_j ， $j \neq i$ ，使得 c_i 不决定p。在这种情况下，对于 c_i ，TR包括4个测试要求：(1) c_i 为真且p为真，(2) c_i 为假且p为真，(3) c_i 为真且p为假，(4) c_i 为假且p为假。

虽然无效子句覆盖 (ICC) 也像 ACC 那样存在混淆, 但是只定义了两个不同的特性, 叫做广义无效子句覆盖 (GICC) 和限制性的无效子句覆盖 (RICC)。因为 c_i 与 p 不相关, 所以 c_i 不能够决定 p , 相关性概念与无效子句覆盖无关。在可行性方面, 由于定义的结构, 谓词覆盖可在所有的特性中得到满足。

下面的例子强调了无效子句覆盖的取值。假设你测试反应堆关停系统的一个控制软件, 说明书定义特定的阀门状态 (开或关) 与在正常 (Normal) 模式下重启操作有关, 而在重写 (Override) 模式下却不是这样。也就是说, 在重写模式下无论阀门状态开还是关, 重启应该表现一致。持怀疑态度的测试人员要在重写模式下对于阀门的两个位置进行重启测试, 因为可能的实现错误将要考虑各种模式下阀门的设置。

GICC 和 RICC 的形式化定义如下。

标准3.18 广义无效子句覆盖 (GICC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 $c_j, j \neq i$, 使得 c_i 不决定 p 。在这种情况下, 对于 c_i , TR 包括4个测试要求: (1) c_i 为真且 p 为真, (2) c_i 为假且 p 为真, (3) c_i 为真且 p 为假, (4) c_i 为假且 p 为假。次子句 c_j 取值可能随着这4种情况而变化。

标准3.19 限制性无效子句覆盖 (RICC): 对于每个 $p \in P$ 和每个主子句 $c_i \in C_p$, 选择次子句 $c_j, j \neq i$, 使得 c_i 不决定 p ; 在这种情况下, 对于 c_i , TR 包括4个测试要求: (1) c_i 为真且 p 为真, (2) c_i 为假且 p 为真, (3) c_i 为真且 p 为假, (4) c_i 为假且 p 为假。次子句 c_j 取值要与这测试用例 (1) 和 (2) 相同, 还要与测试用例 (3) 和 (4) 相同。

3.2.3 不可行性和包含

一些技术问题把有效子句覆盖变复杂了。像大多数标准一样, 最重要的是可行性问题。有时候因为子句互相关联会产生不可行性的问题, 也就是说, 对一个子句选择真值, 可能影响另一个子句的真值。例如, 思考下面常见的循环结构, 假设它有短路的语义:

```
while (i < n && a[i] != 0) {do something to a[i]}
```

这里的思路是一旦 i 超出范围要避免给 $a[i]$ 赋值, 不但假设有短路, 而且依赖于短路。显然, 生成不满足 $i < n$ 且满足 $a[i] != 0$ 的测试用例是不可能的。

原则上讲, 子句和谓词标准不可行性的问题和图标准不可行性的问题没有区别。在两种情况下, 解决方法都是满足可行的测试需求, 然后决定怎么样处理不可行的测试需求。最简单的方法是忽略不可行的需求, 这通常不会对测试质量造成影响。

然而, 对于某些不可行的测试需求, 更好的办法是用一种所包含的测试标准来考虑相对应的测试需求。例如, 如果对于谓词 p 中的子句 a , RACC 覆盖不可行 (由于子句间有额外约束), 而 CACC 可行, 那就用可行的 CACC 测试需求代替不可行的 RACC 测试需求。这种方法与前面介绍图覆盖中提到的最大努力游历方法类似。

图3.1显示了逻辑表达式覆盖标准的包含关系。注意, ICC 标准不包含任何的 ACC 标准, 反之亦然。该图假设对于不可行性测试需求采用最好的处理方法。在这种方法无法处理不可行性测试需求的情况下, 该图建议我们忽略这些需求。

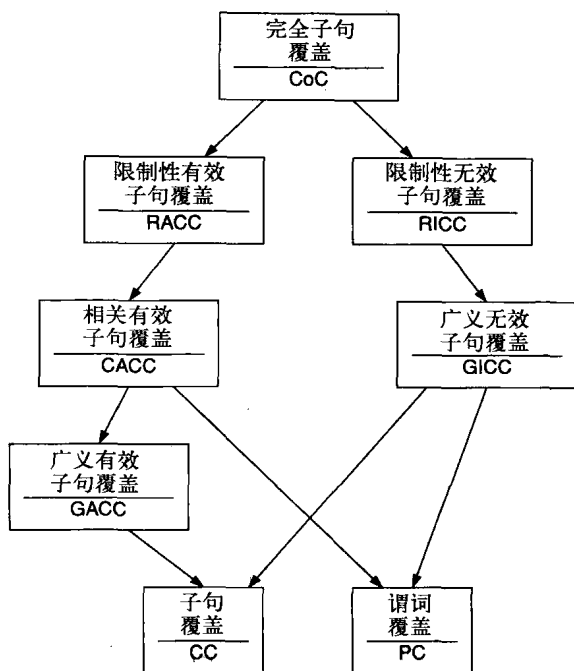


图3.1 逻辑覆盖标准间的包含关系

3.2.4 使子句决定谓词

那么我们该如何去寻找次子句 c_i 的取值使得主子句 c_i 决定 p 的值呢？作者们意识到文献中所陈述的三种不同方法，我们在这里给出一种直接定义的方法，其余两种（其中一种是这种定义方法的算法版本）在参考文献注释里给出。

对于具有子句（或布尔变量） c 的谓词 p ，令 $p_{c=true}$ 代表谓词 p 中 c 的每次出现都由 $true$ 代替， $p_{c=false}$ 代表谓词 p 中 c 的每次出现都由 $false$ 代替。对于剩下的推导，我们假定没有重复（即 p 只包含一个 c ）。注意， $p_{c=true}$ 和 $p_{c=false}$ 都不包含任何子句 c 。现在我们用异或连接这两个表达式：

$$p_c = p_{c=true} \oplus p_{c=false}$$

结果是 p_c 描述了 c 的取值决定 p 值的具体条件。即如果 p_c 中的子句取值使得 p_c 为真，那么 c 的真值决定了 p 的真值；如果 p_c 中的子句取值使得 p_c 为假，那么 p 的真值独立于 c 的真值。这正是我们实现各种形式的有效和无效子句覆盖所需要的。

作为第一个例子，我们举 $p = a \vee b$ 。 p_a 由定义得：

$$\begin{aligned}
 p_a &= p_{a=true} \oplus p_{a=false} \\
 &= (true \vee b) \oplus (false \vee b) \\
 &= true \oplus b \\
 &= \neg b
 \end{aligned}$$

也就是说，为了使主子句 a 决定谓词 p ，仅有的次子句 b 必须为假。这直观上应该很容易理解，因为 a 的取值只在 b 为假时才会影响 p 的值。由对称性可以很容易得到 p_b 为 $\neg a$ 。

如果我们将该谓词改为 $p = a \wedge b$ ，可以得到：

$$\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \wedge b) \oplus (false \wedge b) \\
&= b \oplus false \\
&= b
\end{aligned}$$

也就是说, 我们需要 $b = true$ 使得 a 决定 p 。由相似的分析可以得出 $p_b = a$ 。
等价运算符不是很直白, 并引出了有意思的一点。考虑 $p = a \leftrightarrow b$ 。

$$\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \leftrightarrow b) \oplus (false \leftrightarrow b) \\
&= b \oplus \neg b \\
&= true
\end{aligned}$$

也就是说, 对于 b 的任何取值, a 决定 p 的值而不考虑 b 的取值! 这意味着对于谓词 p , 就像这个, p_c 的值是常量真, ICC 标准对于 c 是不可行的。当表达式中使用等价或异或运算符时, 对其应用无效子句覆盖可能导致不可行的测试需求。

该结论可以得出的更一般的形式也可应用于 ACC 标准。如果谓词 p 包含子句 c 使得 p_c 取常量假, 则 ACC 标准对于 c 是不可行的。其根本原因是该子句是冗余的; 该谓词可以重写成没有它的形式。虽然这听起来好像是理论上的好奇心, 但实际上这对于测试者来说是一个非常有用的结果。如果谓词包含冗余的子句, 那么这便是该谓词有问题的一個强烈的信号!

考虑 $p = a \wedge b \vee a \wedge \neg b$ 。这实际上就是谓词 $p = a$, b 是不相干的。通过计算 p_b , 我们得到:

$$\begin{aligned}
p_b &= p_{b=true} \oplus p_{b=false} \\
&= (a \wedge true \vee a \wedge \neg true) \oplus (a \wedge false \vee a \wedge \neg false) \\
&= (a \vee false) \oplus (false \vee a) \\
&= a \oplus a \\
&= false
\end{aligned}$$

因此 b 不可能决定 p 。

我们需要考虑对于几个更复杂的表达式如何使子句决定谓词。对于表达式 $p = a \wedge (b \vee c)$, 我们得到:

$$\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \wedge (b \vee c)) \oplus (false \wedge (b \vee c)) \\
&= (b \vee c) \oplus false \\
&= b \vee c
\end{aligned}$$

这个例子的结果是不确定的, 其指出了 CACC 和 RACC 之间的关键区别。三种取值使得 $b \vee c$ 为真, $(b = c = true)$ 、 $(b = true, c = false)$ 和 $(b = false, c = true)$ 。对于 CACC, 我们可以在 a 为真时挑选一对取值而在 a 为假时挑选另一对取值。而对于 RACC, 对于 a 的所有取值我们必须选择同一个配对。

对 b 及 c 的推导稍微复杂一些:

$$\begin{aligned}
p_b &= p_{b=true} \oplus p_{b=false} \\
&= (a \wedge (true \vee c)) \oplus (a \wedge (false \vee c)) \\
&= (a \wedge true) \oplus (a \wedge c) \\
&= a \oplus (a \wedge c) \\
&= a \wedge \neg c
\end{aligned}$$

上面所示化简的最后一步可能不是那么显而易见。如果认为不对的话，可以尝试构造 $a \oplus (a \wedge c)$ 的真值表。 p_c 的计算方法是相同的，经推导可得 $a \wedge \neg b$ 。

3.2.5 寻找满足的取值

应用逻辑覆盖标准的最后一步是选择符合该标准的取值。本节展示了如何为一个实例产生值，练习和本章后面的应用章节探讨了更多的情况。该例子取自本章的第1节：

$$p = (a \vee b) \wedge c$$

为谓词覆盖寻找取值是比较容易的，3.2节也已经介绍过了。两个测试需求是：

$$TR_{PC} = \{p = true, p = false\}$$

它们可以由以下子句的取值所满足：

	<i>a</i>	<i>b</i>	<i>c</i>
<i>p</i> = true	t	t	t
<i>p</i> = false	t	t	f

为了运行这些测试用例，我们需要精化这些真值赋值以创建子句 a 、 b 和 c 的取值。假设子句 a 、 b 和 c 由以下Java程序变量定义：

<i>a</i>	$x < y$ ，程序变量 x 和 y 的一个关系表达式
<i>b</i>	done，一个布尔基本类型的值
<i>c</i>	list.contains(str)，List和String的对象

因此，完整展开的谓词实际上是：

$$p = (x < y \vee done) \wedge list.contains(str)$$

那么下面程序变量的取值满足谓词覆盖的测试需求。

	<i>a</i>		<i>b</i>	<i>c</i>
<i>p</i> = true	$x=3$	$y=5$	done = true	list=["Rat," "Cat," "Dog"] str = "Cat"
<i>p</i> = false	$x=0$	$y=7$	done = true	list=["Red," "White"] str = "Blue"

注意，如果目的是为了给一个子句赋以特定的值则程序变量的取值在特定的测试用例中不必一样。例如，子句 a 在两个测试中都是真，即使程序变量 x 和 y 具有不同的值。

满足子句覆盖的取值在3.2节也介绍过。6个测试需求是：

$$TR_{CC} = \{a = true, a = false, b=true, b = false, c = true, c = false\}$$

它们可以由以下子句的取值所满足（空白格子代表“不必关心的”值）：

	<i>a</i>	<i>b</i>	<i>c</i>
<i>a</i> = true	t		
<i>a</i> = false	f		
<i>b</i> = true		t	
<i>b</i> = false		f	
<i>c</i> = true			t
<i>c</i> = false			f

为程序变量 x 、 y 、 $done$ 、 $list$ 和 str 取值以精化真值赋值留给读者作为练习。

在讨论其他标准之前，我们先为次子句选择取值以确保主子句能决定 p 的值。我们先前给出了计算 p_a 、 p_b 和 p_c 的方法。对于这个特定的谓词 p 的计算留作练习。结果如下所示：

p_a	$\neg b \wedge c$
p_b	$\neg a \wedge c$
p_c	$a \vee b$

现在我们可以转去讨论其他子句覆盖标准了。首先是组合覆盖，其要求针对子句的取值的所有组合。在这个例子中我们有8个测试需求，其可由如下取值所满足：

	a	b	c	$(a \vee b) \wedge c$
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

回忆广义有效子句覆盖要求每个主子句取真和假并且次子句的取值要使得主子句决定谓词的值。与子句覆盖相似，可以定义出三对测试需求：

$$TR_{GACC} = \{(a = \text{true} \wedge p_a, a = \text{false} \wedge p_a), (b = \text{true} \wedge p_b, b = \text{false} \wedge p_b), \\ (c = \text{true} \wedge p_c, c = \text{false} \wedge p_c)\}$$

这些测试需求能由如下针对子句的取值所满足。注意，它们可以和子句覆盖一样，除了子句覆盖中的空白格子由确定性分析得出的值代替。在下面的（部分真值）表中，主子句的赋值由粗体的大写字母指示。

	a	b	c	p
$a = \text{true} \wedge p_a$	T	f	t	t
$a = \text{false} \wedge p_a$	F	f	t	f
$b = \text{true} \wedge p_b$	f	T	t	t
$b = \text{false} \wedge p_b$	f	F	t	f
$c = \text{true} \wedge p_c$	t	f	T	t
$c = \text{false} \wedge p_c$	f	t	F	f

注意其中的重复，第1行与第5行是一样的，第2行和第4行也是一样的。也就是说只要4个测试就能满足GACC。

一种不同的看待GACC的方式是考虑针对每对测试需求的所有可能的测试输入的配对。回忆有效子句覆盖标准总是以配对的形式产生测试需求，针对待测谓词的每个子句都产生一个配对。为了识别这些测试输入，我们将使用真值表中的行号。因此，配对（3，7）代表上表中所列的前两个测试。

可以发现（3，7）是仅有的能够满足关于子句 a （其中 a 是主子句）的GACC测试需求的配对，（5，7）是仅有的能够满足关于 b 的GACC测试需求的配对。对于子句 c ，情况变得更加有趣了。有9个配对满足关于子句 c 的GACC测试需求，即

$$\{(1,2), (1,4), (1,6), (3,2), (3,4), (3,6), (5,2), (5,4), (5,6)\}$$

回忆相关有效子句覆盖要求每个主子句取真和假，次子句的取值要使得主子句决定谓词的值，并且谓词必须真假值都有。与GACC类似，可以定义出三对测试需求：对于子句 a ，测

试需求的配对是：

$$\begin{aligned}a &= \text{true} \wedge p_a \wedge p = x \\a &= \text{false} \wedge p_a \wedge p = \neg x\end{aligned}$$

x 可以为真或假，要点是 p 在这两个测试用例中必须有一个不同的真值。我们留给读者写出关于 b 和 c 的相关的CACC测试需求。

对于我们例子中的谓词 p ，通过对GACC的测试用例配对进行一次仔细的检查可以发现 p 在每个配对中取不同的真值。因此，GACC和CACC对谓词 p 来说是相同的，适用相同的测试输入配对。在练习中读者将发现这样的谓词，其中满足关于某个子句 c 的GACC的测试配对不满足关于 c 的CACC。

然而对于例子 p ，RACC的情形却十分不同。回忆受限有效子句覆盖和CACC是相同的，除了它要求对于次子句 c_i 的取值对于主子句 c_i 的两个真值赋值都是相同的。对于子句 a ，RACC产生的测试需求的配对是：

$$\begin{aligned}a &= \text{true} \wedge p_a \wedge b = B \wedge c = C \\a &= \text{false} \wedge p_a \wedge b = B \wedge c = C\end{aligned}$$

对于某布尔常量 B 和 C 。检查上面给出的针对GACC的配对可以发现关于子句 a 和 b 的配对是相同的。因此配对(3, 7)满足关于子句 a 的RACC，配对(5, 7)满足关于 b 的RACC。然而，关于 c 却只有三对满足RACC，即

$$\{(1, 2), (3, 4), (5, 6)\}$$

这个例子的确对ACC标准的不同类别留下了质疑，即它们之间实际上到底有什么不同？也就是问，除了在算术上微妙的不同之外，对于实际的测试人员有什么影响？真正的不同虽不是经常显露出来，但当它们显露时却十分引人注目并且非常恼人。

GACC不要求谓词覆盖满足于每个子句的测试配对，因此用这种方式将意味着我们不能像我们所喜欢的那样对我们的程序做充分的测试。在实际使用中，很容易构造出满足GACC但不满足谓词覆盖的例子，此时谓词非常小（一到两项），但三项或更多项时就很难了，这是因为对于其中的一个子句，很可能选择的满足GACC的测试也是满足CACC的测试。

在另一方面RACC的约束性本质有时会使得这个标准难以满足。当某些子句取值的组合不可行时就更是如此。假设上面例子中用到的谓词中，根据程序的语义实际消除了真值表中的2、3和6行，那么RACC关于子句`list.contains(str)`就无法得到满足（即我们有不可行的测试需求），但是CACC却能。明智的读者（即还清醒着的）将意识到相关有效子句覆盖经常是ACC中最实际的一种类型。

3.2节练习

使用谓词(1)~(10)回答以下问题。

1. $p = a \wedge (\neg b \vee c)$
2. $p = a \vee (b \wedge c)$
3. $p = a \wedge b$
4. $p = a \rightarrow (b \rightarrow c)$
5. $p = a \oplus b$
6. $p = a \leftrightarrow (b \wedge c)$

$$7. p = (a \vee b) \wedge (c \vee d)$$

$$8. p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$$

$$9. p = a \vee b \vee (c \wedge d)$$

$$10. p = (a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$$

(a) 识别谓词 p 的子句。

(b) 计算并化简每个子句决定谓词 p 的条件。

(c) 写出所有子句的完整真值表。以1开始标出行号。使用例子中所用的格式并遵循3.2节中的组合覆盖定义。即第1行应该所有子句都取真。你应该为每个子句决定谓词的条件添加列，并为谓词自己也添加一列。

(d) 识别表中行的关于每个子句的满足广义有效子句覆盖（GACC）的所有配对。

(e) 识别表中行的关于每个子句的满足相关有效子句覆盖（CACC）的所有配对。

(f) 识别表中行的关于每个子句的满足受限有效子句覆盖（RACC）的所有配对。

(g) 识别表中行的关于每个子句的满足广义无效子句覆盖（GICC）的所有四元组。识别任何不可行的GICC测试需求。

(h) 识别表中行的关于每个子句的满足受限无效子句覆盖（RICC）的所有四元组。识别任何不可行的RICC测试需求。

11. 精化GACC、CACC、RACC、GICC和RICC覆盖标准使得次子句上的约束更加形式化。

12. （比较有挑战！）寻找一个谓词和一个额外的约束集使得CACC关于某个子句是不可行的而GACC却是可行的。

3.3 程序的结构化逻辑覆盖

与图覆盖标准一样，逻辑覆盖标准以一种直接的方式应用于程序。谓词直接从程序中的判定点（if、case和loop语句）派生。虽然这些标准在谓词具有很大数量子句时很难应用，但对于程序这常常不是一个问题。程序中的绝大多数谓词仅有一个子句，并且程序员倾向于书写最多两到三个子句的谓词。当谓词仅有一个子句时所有的逻辑覆盖标准合并为一个标准——谓词覆盖，这是显然的。

将逻辑覆盖应用于程序的主要复杂性更多是关于可达性而不是关于标准。即逻辑覆盖标准赋予与程序中特定判定点（语句）相关的测试需求。获得满足这些需求的取值只是问题的一部分；到达那个语句有时更困难。到达那里与两个问题相关，第一个问题就是第1章中的可达性；测试用例必须包含能到达那个语句的取值。在小程序中（即大多数方法）这个问题不难，但当应用于一个任意大的程序中时，满足可达性会非常复杂。满足可达性的取值是测试用例中的前缀取值。

“到达那里”的另一个问题甚至更困难。测试需求是以程序变量的形式表达的，它们可能局部定义到单元里或者甚至是待测的语句块里。另一方面，我们的测试用例能够包含仅用在测程序输入的取值。所以这些内部变量必须以输入变量的形式解决。虽然测试需求中变量的取值应该最终是输入变量的取值的一个函数，但这种关系可能会任意复杂。事实上，内部变量问题形式上是不可判定的。

考虑派生自查找表的一个内部变量 X ，对于该表的索引由一个输入即程序输入的很复杂的

函数所决定。为了给X选一个特定的值，测试人员必须从判定出现的语句处向后规划到X被选出的那张表以及那个函数，最后到一个能使该函数计算出期待取值的输入值。如果该函数包含随机性或是时间敏感的，抑或如果该输入无法由测试者控制，那么满足测试需求将变得不可能。这种控制性问题在自动化测试数据产生文献中已被深入地探讨过，这里就不再赘述。除了注意一下该问题正是为什么程序级逻辑覆盖标准的使用常常受限于单元和模块测试活动的一个主要原因。

图3.2和图3.3中的例子程序用来展示程序^①中的逻辑覆盖。该程序是一个简单的三角形分类程序，叫TriTyp。该程序（或者更精确地说，该算法）在测试文献中用作例子已经好多年了。作为一个例子，它有好几个优势：其目的相对比较好理解，其足够小能适合于课堂练习，并且它有一个非常复杂的逻辑结构能用于阐明大部分概念。这个版本的TriTyp是用Java写的，并经Sun的JDK 1.4.1编译和测试过。添加行号是为了让我们能引用文本中特定的判定语句。

谓词取自程序中的判定点，包括if语句、case/switch语句、for循环、while循环和do-until循环。这可由TriTyp程序中的Triang()方法来展示。Triang()有如下谓词（行号显示在左边，62和78行的else语句没有它们自己的谓词）：

```

42: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
49: (Side1 == Side2)
51: (Side1 == Side3)
53: (Side2 == Side3)
55: (triOut == 0)
59: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
    Side1+Side3 <= Side2)
70: (triOut > 3)
72: (triOut == 1 && Side1+Side2 > Side3)
74: (triOut == 2 && Side1+Side3 > Side2)
76: (triOut == 3 && Side2+Side3 > Side1)

```

TriTyp程序有三个输入，读入到程序主函数中的变量A、B和C，之后传递给Triang()中的形参Side1、Side2和Side3。本节的余下部分介绍如何满足TriTyp中的逻辑覆盖标准。在介绍实际的标准之前，首先有必要分析这些谓词以寻找取值能达到这些谓词（可达性问题），并理解如何给变量triOut赋以特定的值（内部变量问题）。

在表3.1到表3.3中，Side1、Side2和Side3简写成S1、S2和S3以节省空间。首先我们考虑可达性。42行的谓词无论Triang()何时被调用总是可达的，因此它的可达性谓词是真，如表3.1中的第一行所示。49行仅当42行的谓词是假时可达，因此它的可达性谓词是其相反的情况，即 $S1 > 0 \ \&\& \ S2 > 0 \ \&\& \ S3 > 0$ 。该谓词被加上P1的标签并在随后的可达性谓词中引用。通过对前一个边上的谓词取反，可以以相似的方式找到其余部分谓词。

① 测试方面的“老手”可能认同甚至厌倦了三角形的例子。它被用来作为教学工具，在学术文献中也一直起重要作用；它是人们熟知的问题、控制结构有趣，足以说明大多数问题；并且它不使用语言特性，从而使这种分析很难，例如循环和间接引用。这一版本的TriTyp有点过度复杂，但是对概念的认识还是很有帮助的。

```

1 //Jeff Offutt——Java版本 2003年2月
2 //分类三角形
3 import java.io.*;
4
5 class trityp
6 {
7     private static String[] triTypes = { "", // 忽略0
8         "scalene", "isosceles", "equilateral",
9         "not a valid triangle"};
10
11     private static String instructions = "This is the ancient
12     TriTyp program.\nEnter three integers that represent the
13     lengths of the sides of a triangle.\nThe triangle will be
14     categorized as either scalene, isosceles, equilateral\n
15     or invalid.\n";
16
17 public static void main (String[] argv)
18 { // trityp的驱动程序
19     int A, B, C;
20     int T;
21
22     System.out.println (instructions);
23     System.out.println ("Enter side 1: ");
24     A = getN();
25     System.out.println ("Enter side 2: ");
26     B = getN();
27     System.out.println ("Enter side 3: ");
28     C = getN();
29     T = Triang (A, B, C);
30
31     System.out.println ("Result is: " + triTypes[T]);
32 }
33
34 // -----
35 // 三角形分类的主方法
36 private static int Triang (int Side1, int Side2, int Side3)
37 {
38     int triOut;
39
40     // triOut的输出规则:
41     // Triang = 1 如果三角形是不等边的
42     // Triang = 2 如果三角形是等腰的
43     // Triang = 3 如果三角形是等边的
44     // Triang = 4 如果不是三角形
45
46     // 在快速地确认它是一个合法的
47     // 三角形之后, 检测任何等长的边
48     if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
49     {
50         triOut = 4;
51         return (triOut);
52     }
53
54     triOut = 0;
55     if (Side1 == Side2)
56         triOut = triOut + 1;
57     if (Side1 == Side3)
58         triOut = triOut + 2;
59     if (Side2 == Side3)
60         triOut = triOut + 3;
61     if (triOut == 0)
62     { // 在声明它是不等边前确认
63         // 它是合法的三角形
64     }
65 }

```

图3.2 TriTyp-A部分

```

59     if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
60         Side1+Side3 <= Side2)
61         triOut = 4;
62     else
63         triOut = 1;
64     return (triOut);
65 }
66
67 // 在声明它是等腰或等边之前
68 // 确认它是合法的三角形
69
70 if (triOut > 3)
71     triOut = 3;
72 else if (triOut == 1 && Side1+Side2 > Side3)
73     triOut = 2;
74 else if (triOut == 2 && Side1+Side3 > Side2)
75     triOut = 2;
76 else if (triOut == 3 && Side2+Side3 > Side1)
77     triOut = 2;
78 else
79     triOut = 4;
80 return (triOut);
81 } // 结束Triang函数
82
83 // -----
84 // 读入 (或选择) 一个整数
85 private static int getN ()
86 {
87     int inputInt = 1;
88     BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
89     String inStr;
90
91     try
92     {
93         inStr = in.readLine ();
94         inputInt = Integer.parseInt(inStr);
95     }
96     catch (IOException e)
97     {
98         System.out.println ("Could not read input, choosing 1.");
99     }
100    catch (NumberFormatException e)
101    {
102        System.out.println ("Entry must be a number, choosing 1.");
103    }
104
105    return (inputInt);
106 } // 结束getN函数
107
108 } // 结束trityp类

```

图3.3 TriTyp-B部分

表3.1 Triang谓词的可达性

42: True
49: $P1 = (S1 > 0 \ \&\& \ S2 > 0 \ \&\& \ S3 > 0)$
51: P1
53: P1
55: P1
59: $P1 \ \&\& \ (triOut == 0)$
62: $P1 \ \&\& \ (triOut == 0)$ $\ \&\& \ (S1 + S2 > S3) \ \&\& \ (S2 + S3 > S1) \ \&\& \ (S1 + S3 > S2)$
70: $P1 \ \&\& \ (triOut != 0)$

(续)

```

72: P1 && (triOut != 0) && (triOut <= 3)
74: P1 && (triOut != 0) && (triOut <= 3)
    && ((triOut != 1) || (S1 + S2 <= S3))
76: P1 && (triOut != 0) && (triOut <= 3)
    && ((triOut != 1) || (S1 + S2 <= S3))
    && ((triOut != 2) || (S1 + S3 <= S2))
78: P1 && (triOut != 0) && (triOut <= 3)
    && ((triOut != 1) || (S1 + S2 <= S3))
    && ((triOut != 2) || (S1 + S3 <= S2))
    && ((triOut != 3) || (S2 + S3 <= S1))

```

注意到表3.1中的几个谓词引用了变量triOut，这是一个局部（内部）变量，在44、48、50、52、54、61、63、71、73、75、77和79行被赋值。因此产生测试的下一步是发现如何给triOut赋以特定的值。在55行，triOut拥有值域（0~6）范围内的取值，这是由前面语句所赋的。通过应用表3.1中的谓词并跟踪赋值，我们可以确定针对triOut的如下规则：

triOut	确定triOut的规则
0	$S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3$
1	$S1 == S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3$
2	$S1 \neq S2 \ \&\& \ S1 == S3 \ \&\& \ S2 \neq S3$
3	$S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 == S3$
4	$S1 == S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 == S3$
5	$S1 \neq S2 \ \&\& \ S1 == S3 \ \&\& \ S2 == S3$
6	$S1 == S2 \ \&\& \ S1 == S3 \ \&\& \ S2 == S3$

关于triOut值为4和5的谓词是相互矛盾的，因此它在55行后不可能具有那些值。这些值能用于化简表3.1中的谓词，得到的结果如表3.2所示。这些谓词乍一看很复杂，但它们包含了大量冗余。另外两个命名公式被引入了，即 $P2 = (S1 == S2 \ \parallel \ S1 == S3 \ \parallel \ S2 == S3)$ 和 $P3 = (S1 \neq S2 \ \parallel \ S1 \neq S3 \ \parallel \ S2 \neq S3)$ 。表3.2显示了能确保给定语句号可达性的谓词。这些之后会用于寻找满足逻辑标准的取值。

表3.2 Triang谓词的可达性——通过解析triOut化简得到

```

42: True
49: P1 = (S1 > 0 && S2 > 0 && S3 > 0)
51: P1
53: P1
55: P1
59: P1 && (S1 != S2 && S1 != S3 && S2 != S3)           (triOut == 0)
62: P1 && (S1 != S2 && S1 != S3 && S2 != S3)           (triOut == 0)
    && (S1 + S2 > S3) && (S2 + S3 > S1) && (S1 + S3 > S2)
70: P1 && P2 = (S1 == S2 || S1 == S3 || S2 == S3)       (triOut != 0)
72: P1 && P2 && P3 = (S1 != S2 || S1 != S3 || S2 != S3) (triOut <= 3)
74: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
76: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
    && ((S1 != S3) || (S1 + S3 <= S2))
78: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
    && ((S1 != S3) || (S1 + S3 <= S2))
    && ((S2 != S3) || (S2 + S3 <= S1))

```

为TriTyp中42行的谓词寻找满足谓词覆盖的取值是直截了当的。三个变量中的任何一个可以被赋以0或小于0的值以得到真的情况，它们三个必须都为1或大于1的值以得到假的情况。

谓词49、51和53是相似的。它们在三边中的两条具有相同长度时为真，长度不同时为假。谓词55展示了内部变量问题。*triOut*是一个内部变量，在语句48、50、52和54有定义。代数分析可以显示*triOut*仅当49、51和53行的谓词都不为真时为0。这个分析的一个语义泛化是为了使*triOut*为0（谓词55为真），三条边必须具有不同的长度。

通过使用先前的分析对内部变量*triOut*进行取值解析，其他的谓词可以用相似的方式解决。满足谓词覆盖的取值如表3.3所示。对于每个谓词都给出了确保谓词为真和假的输入，以及期待的输出（EO）。推迟选择取值直到可达性确定时通常比较安全一些。

表3.3 Triang的谓词覆盖

谓词	True				False			
	A	B	C	EO	A	B	C	EO
p42: ($S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0$)	0	0	0	4	1	1	1	3
p49: ($S1 == S2$)	1	1	1	3	1	2	2	3
p51: ($S1 == S3$)	1	1	1	3	1	2	2	2
p53: ($S2 == S3$)	1	1	1	3	2	1	2	2
p55: ($triOut == 0$)	1	2	3	4	1	1	1	3
p59: ($S1 + S2 \leq S3 \vee$ $S2 + S3 \leq S1 \vee$ $S1 + S3 \leq S2$)	1	2	3	4	2	3	4	1
p70: ($triOut > 3$)	1	1	1	3	2	2	3	2
p72: ($triOut == 1 \wedge S1 + S2 > S3$)	2	2	3	2	2	2	4	4
p74: ($triOut == 2 \wedge S1 + S3 > S2$)	2	3	2	2	2	4	2	4
p76: ($triOut == 3 \wedge S2 + S3 > S1$)	3	2	2	2	4	2	2	4

在这个例子中程序的谓词覆盖仅仅是表达边覆盖标准的另一种形式。这是显而易见的，没有必要为逻辑覆盖画出图来，但控制流图能有助于为可达性寻找取值。

之前我们谈到为“不必关心的”的输入选择取值应该推迟到可达性被确定的时候，这是由于对可达性的需求和取值的选择之间存在潜在的相互作用。即有些输入可能对于测试需求来说“不必关心的”，但可能需要特定的值以到达判定点。因此，如果我们选择取值过早，可达性就有可能得不到满足。

满足其他标准所需的取值同样适用于仅有一个子句的谓词（p49、p51、p53、p55和p70）。因此，我们只为有多个子句的谓词考虑子句覆盖。

谓词42的子句比较简单，仅仅要求变量之一的一个取值，其他取值可以任意选择。其他的谓词有一点点复杂，但还是能通过简单的代数解决。考虑谓词59的第一个子句， $S1 + S2 \leq S3$ 。如果我们为S1和S2取值为（S1 = 2，S2 = 3），则 $S1 + S2 = 5$ ，因此S3应该至少为5才能得到真的情况。S1和S2同样的取值可以用于假的情况，此时S3可以为4。谓词72的第一个子句有一个额外的复杂性是因为它涉及内部变量*triOut*。*Triang()*的逻辑揭示了*triOut* = 1当且仅当S1 = S2，并且S1和S2都不等于S3。因此真的情况可以由（2，2，3）来满足，而假的情况可通过将变量的取值取成不同值如（2，3，4）来满足。

满足子句覆盖的取值如表3.4所示。

表3.4 Triang的子句覆盖

子句	True				False			
	A	B	C	EO	A	B	C	EO
p42: ($S1 \leq 0$)	0	1	1	4	1	1	1	3
($S2 \leq 0$)	1	0	1	4	1	1	1	3
($S3 \leq 0$)	1	1	0	4	1	1	1	3
p59: ($S1 + S2 \leq S3$)	2	3	6	4	2	3	4	1
($S2 + S3 \leq S1$)	6	2	3	4	2	3	4	1
($S1 + S3 \leq S2$)	2	6	3	4	2	3	4	1
p72: ($triOut = 1$)	2	2	3	2	2	3	2	2
($S1 + S2 > S3$)	2	2	3	2	2	2	5	4
p74: ($triOut = 2$)	2	3	2	2	3	2	2	2
($S1 + S3 > S2$)	2	3	2	2	2	5	2	4
p76: ($triOut = 3$)	3	2	2	2	1	2	1	4
($S2 + S3 > S1$)	3	2	2	2	5	2	2	4

与其讨论完其他所有的标准，我们倒不如把精力集中于相关有效子句覆盖。因为每个谓词只涉及一个运算符（||或&&），确定性分析是直截了当的。对于||，次子句必须为假；对于&&，次子句必须为真。该例子确实有点复杂。在p59中，每个子句都是截然不同的，但各个子句都包含了相同的变量。这有时使得寻找满足CACC的取值变得很困难（甚至不可能）。

例如，考虑主子句 ($S1 + S2 \leq S3$)。为了使它决定谓词，两个次子句 ($S2 + S3 \leq S1$) 和 ($S1 + S3 \leq S2$) 必须都为假。如果测试用例 (0, 0, 0) 被选择使得 ($S1 + S2 \leq S3$) 为真，那么所有三个子句都为真，($S1 + S2 \leq S3$) 不决定谓词。（但是在p59这样特定的情况下，由于代数的要求，(0, 0, 0) 是仅有的有此问题的测试。）满足CACC的取值如表3.5所示。

表3.5 Triang的相关有效子句覆盖

谓词	子句			A	B	C	EO
p42: ($S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0$)	T	f	f	0	1	1	4
	F	f	f	1	1	1	3
	f	T	f	1	0	1	4
	f	f	T	1	1	0	4
p59: ($S1 + S2 \leq S3 \vee$ $S2 + S3 \leq S1 \vee$ $S1 + S3 \leq S2$)	T	f	f	2	3	6	4
	F	f	f	2	3	4	1
	f	T	f	6	2	3	4
	f	f	T	2	6	3	4
p72: ($triOut == 1 \wedge S1 + S2 > S3$)	T	t	—	2	2	3	2
	F	t	—	2	3	3	2
	t	F	—	2	2	5	4
p74: ($triOut == 2 \wedge S1 + S3 > S2$)	T	t	—	2	3	2	2
	F	t	—	2	3	3	2
	t	F	—	2	5	2	4
p76: ($triOut == 3 \wedge S2 + S3 > S1$)	T	t	—	3	2	2	2
	F	t	—	3	6	3	4
	t	F	—	5	2	2	4

3.3.1 谓词变换问题

对于测试人员来说ACC标准被认为是昂贵的测试，很多人致力于减少它的开销。有一种途径是重写程序，消除多子句谓词，减少分支测试带来的问题。一个猜想是，由此产生的测试将等同于ACC。然而，我们明确反对这一做法有两个原因。首先，重写的程序可能比原程序（包含重复语句）包含更复杂的控制逻辑，因而影响可靠性和可维护性。其次，如以下例子表明，转换后的程序可能不需要与原始程序等价的ACC测试用例。

考虑以下程序段，其中 a 和 b 是任意布尔表达式， $S1$ 和 $S2$ 可以是一条语句、语句块或函数调用。

```
if (a && b)
    S1;
else
    S2;
```

对于谓词 $a \wedge b$ ，CACC标准要求满足 (t, t) ， (t, f) 和 (f, t) 。但是，如果程序段转换为以下功能等同的结构：

```
if (a)
{
    if (b)
        S1;
    else
        S2;
}
else
    S2;
```

谓词覆盖标准要求3条测试： (t, t) 到达语句 $S1$ ， (t, f) 到达语句 $S2$ 第一个出现， (f, f) 或 (f, t) 到达语句 $S2$ 第二个出现。选择 (t, t) 、 (t, f) 和 (f, f) 意味着测试不满足CACC标准，因为 a 不能确定谓词的值。而且多年的经验告诉我们，冗余的 $S2$ 是糟糕的编程习惯，因为有冗余代码会有潜在错误。

稍大一点的例子能更清晰地凸显这一缺陷。考虑下面的程序段：

```
if ((a && b) || c)
    S1;
else
    S2;
```

直接重写程序段以消除多子句谓词得到如下复杂的结果：

```
if (a)
    if (b)
        if (c)
            S1;
        else
            S1;
    else
        if (c)
            S1;
        else
            S2;
else
    if (b)
```

```
    if (c)
        S1;
    else
        S2;
else
    if (c)
        S1;
    else
        S2;
```

这一程序段相当繁琐，而且极易出错。将谓词覆盖标准应用于这里相当于将组合覆盖应用于原始谓词。一个明智的程序员（或良好优化的编译器）会将其简化如下：

```
if (a)
    if (b)
        S1;
    else
        if (c)
            S1;
        else
            S2;
else
    if (c)
        S1;
    else
        S2;
```

这段程序仍比原始的更难理解。想象一下，维护程序员如何来试图改变这件事情。

下面的表格展示了真值赋值，可以用于使原始程序段满足CACC，以及修改后的程序满足谓词测试准则。CACC和谓词列里的X表示真值用来满足特定程序段的覆盖标准。显然，等价转换后的程序的谓词覆盖与原始程序的CACC测试不同。修改后的谓词覆盖不包含CACC，CACC也不包含谓词覆盖。

	a	b	c	$((a \wedge b) \vee c)$	CACC	谓词
1	t	t	t	T		X
2	t	t	f	T	X	
3	t	f	t	T	X	X
4	t	f	f	F	X	X
5	f	t	t	T		X
6	f	t	f	F	X	
7	f	f	t	T		
8	f	f	f	F		X

3.3节练习

1. 回答下列关于checkIt()方法的问题：

```
public static void checkIt (boolean a, boolean b, boolean c)
{
    if (a && (b || c))
    {
        System.out.println ("P 为真 ");
    }
    else
    {
```

```

    System.out.println ("P 为假 ");
}
}

```

- 将`checkIt()`改写为`checkItExpand()`，使每个if语句只测试一个布尔变量。插桩`checkItExpand()`以便记录哪些边被遍历（这里可以用打印语句）。
- 为`checkIt()`生成一个GACC测试集 $T1$ 。为`checkItExpand()`生成一个边覆盖测试集 $T2$ 。构造 $T2$ ，使其不满足`checkIt()`中谓词的GACC标准。
- 在`checkIt()`和`checkItExpand()`上分别运行 $T1$ ， $T2$ 。

2. 回答下列关于`twoPred()`方法的问题：

```

public String twoPred (int x, int y)
{
    boolean z;

    if (x < y)
        z = true;
    else
        z = false;

    if (z && x+y == 10)
        return "A";
    else
        return "B";
}

```

- 为`twoPred()`寻找满足约束有效子句覆盖（RACC）的测试输入。
- 为`twoPred()`寻找满足约束无效子句覆盖（RICC）的测试输入。

3. 回答关于下面程序段的问题：

程序段P：

```

if (A || B || C)
{
    m();
}
return;

```

程序段Q：

```

if (A)
{
    m();
    return;
}
if (B)
{
    m();
    return;
}
if (C)
{
    m();
}

```

- 给出程序段P的GACC测试集（注意本例的GACC、CACC和RACC的测试集相同）。
 - 程序段P的GACC测试集是否满足程序段Q的边覆盖？
 - 给出程序段Q的边覆盖测试集，并尽量少包含GACC测试集那些测试。
4. （有些挑战！）对TriTyp程序通过填写“不必关心的”值完成满足下列标准的测试集，保证可达性并产生期望输出。下载该程序，编译并用你得到的测试用例运行程序来验证正确的输出。

- 谓词覆盖 (PC)
- 子句覆盖 (CC)
- 组合覆盖 (CoC)
- 相关有效子句覆盖 CACC

5. 对第2章的TestPat程序重复以上练习。

6. 对第2章的Quadratic程序重复以上练习。

3.4 基于规约的逻辑覆盖

软件规约，无论是形式的还是非形式的，都是以不同的形式和语言展现。这些规约总是包含逻辑表达式，从而可以应用逻辑覆盖标准。我们首先考察方法中简单的先决条件的应用。

程序员常常在方法中包含先决条件。这些先决条件有时作为设计的一部分，有时是作为文档添加的。为了分析在不变量的语境下的先决条件，规约语言经常显式声明先决条件。如果先决条件不存在，则测试人员可能要考虑特别创建先决条件并作为测试过程的一部分。在实践中出于防御性编程及安全等目的，先决条件往往被转换为异常。简单来讲，先决条件是规约里谓词的常见而丰富的来源，因而值得我们关注。当然，其他诸如后决条件和不变量等规约结构也是复杂谓词的丰富来源。

考虑图3.4中的cal方法。该方法用自然语言显式列举了先决条件。这些条件可以转化为以下谓词：

$$\begin{aligned} & month1 \geq 1 \wedge month1 \leq 12 \wedge month2 \geq 1 \wedge month2 \leq 12 \wedge month1 \leq month2 \\ & \wedge day1 \geq 1 \wedge day1 \leq 31 \wedge day2 \geq 1 \wedge day2 \leq 31 \wedge year \geq 1 \wedge year \leq 10000 \end{aligned}$$

对于day1和day2必须在同一年份的注释可以安全地忽略，先决条件由句法强制执行，因为年份只有一个参数出现。而且也可能比较明显这些先决条件不够完整。例如并不是每个月都有31天。这些需求应该在规格说明书或程序里反映出来。

这个谓词的结构十分简单，只有11个子句（看上去很多），但子句间只有与的关系。cal()方法的谓词覆盖很容易——对于真的情况，所有的子句都取真；对于假的情况，至少一个子句取假。所以 (*month1*=4, *month2*=4, *day1*=12, *day2*=30, *year*=1961) 满足真的情况，而假的情况可以通过使子句 *month1* ≤ *month2* 不成立 (*month1*=6, *month2*=4, *day1*=12, *day2*=30, *year*=1961) 而满足。子句覆盖要求所有的子句都取真和假。我们可以用两条测试满足需求，但是有些子句是相关的因而无法同时取假。例如*month1*无法既小于1又大于12。对于谓词的取真测试允许所有的子句都取真，然后我们用下面的测试使每个子句取假：(*month1*=-1, *month2*=-2, *day1*=0, *day2*=0, *year*=0) 和 (*month1*=13, *month2*=14, *day1*=32, *day2*=32, *year*=10500)。

我们必须首先发现如何使每个子句确定谓词来满足ACC标准。这对于析取范式的谓词很简单——仅需令每个次子句为真即可。为了寻找余下的测试，可依次令每个子句取假。因此表3.6列举的测试可以满足CACC（同样适用于RACC和GACC）。这里为了节省空间，变量名称使用了缩写。

```

public static int cal (int month1, int day1, int month2,
                      int day2, int year)
{
    /*******
    // 计算一年中任意两个日期
    // 之间相差的天数
    // 先决条件: day1和day2必须在同一年中
    //      1 <= month1, month2 <= 12
    //      1 <= day1, day2 <= 31
    //      month1 <= month2
    //      year的取值范围: 1~10000
    /*******
    int numDays;

    if (month2 == month1) // 在同一个月中
        numDays = day2 - day1;
    else
    {
        // 跳过第0月
        int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        // 是否闰年
        int m4 = year % 4;
        int m100 = year % 100;
        int m400 = year % 400;
        if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
            daysIn[2] = 28;
        else
            daysIn[2] = 29;

        // 先计算两个月中的天数
        numDays = day2 + (daysIn[month1] - day1);

        // 再把两个月中间的天数相加
        for (int i = month1 + 1; i <= month2 - 1; i++)
            numDays = daysIn[i] + numDays;
    }
    return (numDays);
}

```

图3.4 日历方法

表3.6 对于cal()的先决条件的相关有效子句覆盖

	$m1 \geq 1$	$m1 \leq 12$	$m2 \geq 1$	$m2 \leq 12$	$m1 \leq m2$	$d1 \geq 1$	$d1 \leq 31$	$d2 \geq 1$	$d2 \leq 31$	$y \geq 1$	$y \leq 10000$
1. $m1 \geq 1 = T$	T	t	t	t	t	t	t	t	t	t	t
2. $m1 \geq 1 = F$	F	t	t	t	t	t	t	t	t	t	t
3. $m1 \leq 12 = F$	t	F	t	t	t	t	t	t	t	t	t
4. $m2 \geq 1 = F$	t	t	F	t	t	t	t	t	t	t	t
5. $m2 \leq 12 = F$	t	t	t	F	t	t	t	t	t	t	t
6. $m1 \leq m2 = F$	t	t	t	t	F	t	t	t	t	t	t
7. $d1 \geq 1 = F$	t	t	t	t	t	F	t	t	t	t	t
8. $d1 \leq 31 = F$	t	t	t	t	t	t	F	t	t	t	t
9. $d2 \geq 1 = F$	t	t	t	t	t	t	t	F	t	t	t
10. $d2 \leq 31 = F$	t	t	t	t	t	t	t	t	F	t	t
11. $y \geq 1 = F$	t	t	t	t	t	t	t	t	t	F	t
12. $y \leq 10000 = F$	t	t	t	t	t	t	t	t	t	t	F

3.4节练习

考虑Java Iterator接口的remove()方法。该方法有一个依赖于Iterator的状态的复杂先决条件，

并且程序员可以选择检测对先决条件的破坏，并抛出IllegalStateException异常。

1. 形式化描述先决条件。
2. 找出（或写出）Iterator的实现。Java 的Collection类是寻找的合适的地方。
3. 为该实现设计并执行CACC测试。

3.5 有限状态机的逻辑覆盖

第2章讨论了把图覆盖标准应用到有限状态机（FSM）。回想一下FSM是图，节点表示状态，边表示转移。每个转移包含一个前状态和一个后状态。FSM通常用于对软件的行为建模，根据开发者的需要和爱好，FSM或多或少是形式化的和精确的。本书以一种最基本的方式将FSM看做图，仅当对标准的实施有影响时才考虑标记符的差别。

最常见的FSM逻辑覆盖标准是将状态转移中的逻辑表达式看做谓词。在第2章的电梯例子中，触发器也就是谓词可以写成openButton = pressed。把3.2节的标准应用到这些谓词即可得到测试。

考虑图3.5中的例子，这个FSM对汽车（Lexus 2003 ES300）里的记忆驾驶座建模。记忆座椅为两个不同的驾驶员提供两套配置和对后视镜（sideMirrors）的控制，座椅的垂直高度（seatBottom）、座椅到方向盘的水平距离（seatBack）以及腰部支撑（lumbar）。采用这些配置的意图是记住驾驶员的偏好，只需一个按钮就能够方便地切换。图中的每个状态用一个数字标记。

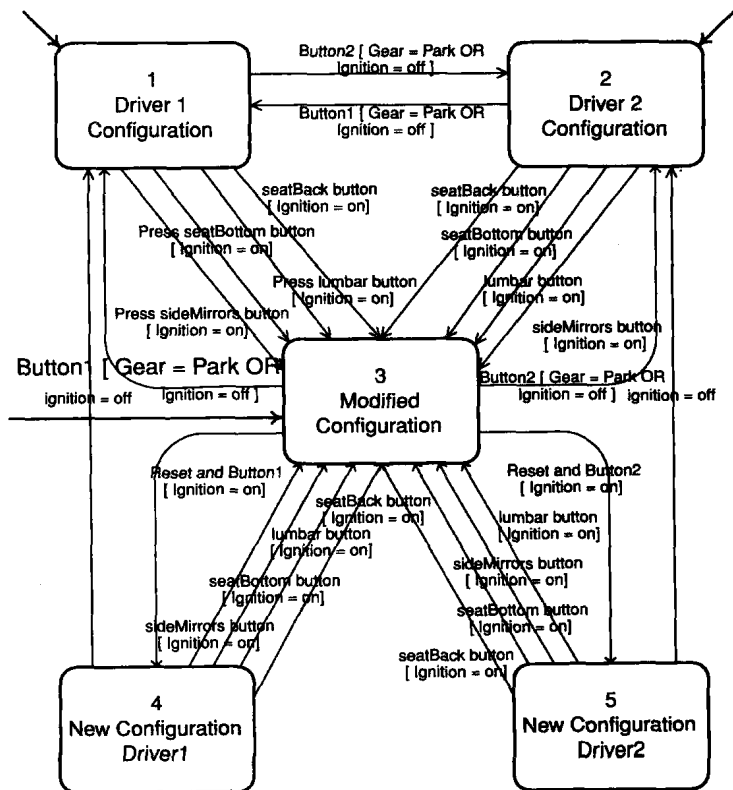


图3.5 汽车座椅的FSM

FSM的初始状态是系统上一次关闭时的状态, Driver1、Driver2或某个修改后的配置。驾驶员可以通过修改上述4种参数来修改配置, 改变后视镜, 前后移动座椅, 升高或降低座椅, 或修正腰部支撑(触发事件)。只有当引擎(ignition)被打开时这些控制才有效。当引擎打开时通过Button1或Button2按钮, 驾驶员可以切换到相应的配置。只有当引擎处于off或变速箱处于Park时保护才允许修改配置。这些是安全约束, 因为在汽车行驶过程中允许驾驶员随便转动座椅是很危险的。

当驾驶员修改了某项设置, 记忆座椅便转入配置改变状态。当引擎打开时可以通过同时按下Reset按钮和Button1、Button2中任意一个按钮来保存新的配置。当引擎关闭时, 新配置就永久保存下来。

尽管在创建谓词和测试值时需要了解并处理一些问题, FSM还是为软件测试提供了一个有效模型。Guards并不总是被显式地列为合取式, 但它们事实上是合取式, 所以应该用“与”运算符将触发器连接起来。在一些规约语言中(如最有名的SCR), 触发器实际上暗示了两个值。在SCR中, 如果一个事件被标记为触发, 则表明表达式的结果必然改变。这暗示了两个值, 触发前的值和触发后的值, SCR用一个新的变量来表示这一情况。例如在记忆座椅的例子中, 当引擎关闭时, 发生从New Configuration Driver1到Driver1 Configuration转移。如果这是SCR中的一个触发转移, 那么相应的谓词包括两个部分: $ignition = on \wedge ignition' = off$ 。 $ignition'$ 是转移后的状态。

从Modified Configuration状态到两个New Configuration状态的转移还有一个问题。两个按钮Reset和Button1(或Button2)必须同时按下。对于该例子的实际条件, 在测试中我们想要知道如果一个按钮比另一个按钮早一点点按下会出现什么情况。不幸的是, 本章用到的数学逻辑表达式无法明确描述这一需求, 因而无法显式地处理。两个按钮在谓词中通过“与”运算符连接。实际上, 这只是时间性一般问题的一个简单例子, 在实时软件里应当加以考虑。

记忆座椅例子的谓词见表3.7(用图3.5的状态数字)。

满足各种覆盖标准的测试很简单, 因而作为读者的练习。当选择测试用例的值时需要考虑几个问题, 首先应当考虑可达性, 测试用例必须包含到达前置状态的前缀值。对于大多数FSM, 这就是要寻找一条从初始状态到达前置状态的一条路径(可采用深度优先搜索策略), 与状态转移相关的谓词经计算可以得到输入。记忆座椅的例子包含3个初始状态, 测试人员无法控制先进入哪一个状态, 因为这取决于系统上次关闭的状态。然而本例有一个很明显的解决方案。我们可以从变速箱在park和按下Button1(部分前缀)开始我们的全部测试。如果系统处于Driver2或Modified Configuration状态, 前面的输入将会使系统进入Driver1状态。如果系统处于Driver1状态, 这些输入没有任何影响。那么对于所有情况, 系统都能在Driver1状态开始。

有些FSM也含有退出状态, 它们必须由后缀值到达。寻找这些值基本上与寻找前缀值一样, 即找到一条从后置状态到达最终状态的路径。记忆座椅不包含退出状态, 因而这一步骤可以跳过。我们还需要查看测试结果(验证值)。我们可以给程序一些输入让程序打印出当前状态, 或引起依赖于该状态其他一些输出。所采取的具体格式和语法依赖于实现, 在没有设计好软件的输入输出格式前无法确定。

这种类型的测试的一个主要优点在于决定期望输出。转移后置状态导致转移为真的测试用例值, 转移前置状态值则使转移为假(系统应停留在当前状态)的测试用例值。有一个例

外就是，有时一个假谓词可能是另一个转移的真谓词。在这种情况下，期望输出应该是另一个转移的后置状态，这种情况可以自动识别。同样，如果一个状态经转移后还是原来的状态，则前置状态和后置状态一样，那么无论转移是真还是假，期望输出都一样。

表3.7 记忆座椅的谓词

前状态	后状态	谓词
1	2	$Button2 \wedge (Gear = Park \vee ignition = off)$
1	3	$sideMirrors \wedge ignition = on$
1	3	$seatBottom \wedge ignition = on$
1	3	$lumbar \wedge ignition = on$
1	3	$seatBack \wedge ignition = on$
2	1	$Button1 \wedge (Gear = Park \vee ignition = off)$
2	3	$sideMirrors \wedge ignition = on$
2	3	$seatBottom \wedge ignition = on$
2	3	$lumbar \wedge ignition = on$
2	3	$seatBack \wedge ignition = on$
3	1	$Button1 \wedge (Gear = Park \vee ignition = off)$
3	2	$Button2 \wedge (Gear = Park \vee ignition = off)$
3	4	$Reset \wedge Button1 \wedge ignition = on$
3	5	$Reset \wedge Button2 \wedge ignition = on$
4	1	$ignition = off$
4	3	$sideMirrors \wedge ignition = on$
4	3	$seatBottom \wedge ignition = on$
4	3	$lumbar \wedge ignition = on$
4	3	$seatBack \wedge ignition = on$
5	2	$ignition = off$
5	3	$sideMirrors \wedge ignition = on$
5	3	$seatBottom \wedge ignition = on$
5	3	$lumbar \wedge ignition = on$
5	3	$seatBack \wedge ignition = on$

最后一个问题是将测试用例（包括前缀值、测试用例值、后缀值和期望输出）转化为可执行的测试脚本。潜在的问题是谓词里变量的赋值必须转变为软件的输入。FSM称之为映射问题，这有些类似于3.3节中的内部变量问题。有时，这一步骤可以简单地将谓词赋值重新写一遍（Button1写成程序的输入button1）。有时，输入值可以直接编码成方法调用并嵌入程序中（例如，Button1变成 `pressButton1()`）。然而另外一些时候，这个问题则比较复杂，涉及将看起来很小的FSM输入转移成冗长的输入以及方法调用序列。具体的情况取决于软件的实现，没有通用的解决方法。

3.5节练习

- 对记忆座椅的FSM，设计测试集满足下列覆盖标准，并满足谓词覆盖，确保可达性，和计算期望输出：
 - 谓词覆盖
 - 相关有效子句覆盖
 - 广义无效子句覆盖
- 重画图3.5以减少转移的数量，但要增加更多的子句。特别是标号为1、2、4和5的节点，每个都有4个到节点3的转移。将转移的数量减少到1个，用“或”将子句连接起来。然后为所得到的谓词生成满足CACCC的测试用例。新测试与由原图所生成的测试比较有什么不同？

3. 考虑下面的确定性的FSM:

当前状态	条件	下一状态
Idle	$a \vee b$	Active
Active	$a \wedge b$	Idle
Active	$\neg b$	WindDown
WindDown	a	Idle

- (a) 画出有限状态机 (FSM)。
- (b) 这个FSM没有说明在什么情况下一个状态经过转移可以到达自身。但是这些条件可由现有的条件得到。找到能使各个状态转移到自身的条件。
- (c) 为从Active状态的各个转移寻找CACC测试。
4. 任选一个家用电器, 手表、计算器、微波炉、录像机、收音机或可编程恒温器。画出描述电器行为的FSM, 并设计满足谓词覆盖、相关有效子句覆盖 (CACC) 和广义无效子句覆盖 (GICC) 标准的测试。
5. 实现记忆座椅的FSM。为你的实现设计一个合适的输入语言, 将第1题的测试转化为测试脚本并执行。

3.6 析取范式标准

在本节中, 我们回顾一下布尔表达式的测试。我们考虑谓词作为析取范式 (DNF) 表述的结构, 而不是关注每个子句本身。

字 (literal) 是一个子句或一个否定子句。项 (term) 是子句的逻辑“与”关系集合。一个析取范式谓词是项的逻辑“或”关系集合。DNF谓词构成中的项也称为蕴涵项, 因为如果一个项为真, 则整个谓词也为真。

例如, 如下谓词就是一个析取范式:

$$(a \wedge \neg c) \vee (b \wedge \neg c)$$

但是这个就不是, 虽然它与上一个是等价的:

$$(a \vee b) \wedge \neg c$$

通常, 谓词的DNF表述并不是唯一的。例如, 上面的谓词可以被重写为下面的DNF形式:

$$(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c)$$

在本节中, 我们遵循DNF表述方式的约定, 使用操作符 $\wedge$ 表示邻接, 使用上横线 ($\neg$) 作为否定操作符。这么做的目的是为了提高某些较长表达式的易读性。因此, 上面式子的最终DNF谓词如下:

$$abc \vee ab\bar{c} \vee a\bar{b}\bar{c}$$

对于DNF表述, 我们感兴趣的是由此产生的测试标准。遵循DNF表述式的一种测试方法是赋值给子句使得DNF表述式的每个蕴涵项至少有一个测试满足。注意, 所有这些测试的结果均为真, 引起这样的问题, 我们从不测试假的情况。我们通过把DNF表达式取反来解决所关注的这一问题, 并且用这个谓词自身所使用的覆盖标准来测试这个谓词的否定形式。依据上述观点, 我们可以定义第一个DNF覆盖标准:

标准3.20 蕴涵项覆盖 (IC): 给定一个DNF表述的谓词表达式 f 以及它的否定形式 $\bar{f}$,

对于 f 与 $\bar{f}$ 中的每个蕴涵项, TR 包含这样的要求: 蕴涵项的取值必须为真。

作为IC的一个示例, 考虑如下的具有3个子句谓词的DNF表达式 f 。

$$f(a,b,c) = ab \vee b\bar{c}$$

它的否定形式如下:

$$\begin{aligned}\bar{f}(a,b,c) &= \overline{ab \vee b\bar{c}} \\ &= \overline{ab} \wedge \overline{b\bar{c}} \\ &= (\bar{a} \vee \bar{b}) \wedge (\bar{b} \vee c) \\ &= \bar{a}\bar{b} \vee \bar{a}c \vee \bar{b}\bar{b} \vee \bar{b}c \\ &= (\bar{a}\bar{b} \vee \bar{b}\bar{b}) \vee \bar{b}c \vee \bar{a}c \\ &= (\bar{b} \vee \bar{b}c) \vee \bar{a}c \\ &= \bar{b} \vee \bar{a}c\end{aligned}$$

由此可以确定 f 和 $\bar{f}$ 的4个蕴涵项:

$$\{ab, b\bar{c}, \bar{b}, \bar{a}c\}$$

一个明显但简单的用来产生测试这4个蕴涵项的方法应该可以为每个蕴涵项选择一个测试用例。但是, 仅仅有很少的测试用例能够满足这个要求。考虑下面的表格, 它显示出这4个蕴涵项需要的真值的赋值。

	a	b	c		
1) ab	T	T		a	b
2) $b\bar{c}$		T	F	b	$\bar{c}$
3) $\bar{b}$		F		$\bar{b}$	
4) $\bar{a}c$	F		T	$\bar{a}$	c

第1行和第2行可以同时被满足, 第3行和第4行也是如此。因此, 对于这个例子, 仅仅需要两个测试用例就可以满足IC标准。

$$T_1 = \{TTF, FFT\}$$

注意, IC包含谓词覆盖, 但不必包含ACC标准的任何一个。

IC的一个问题是通常会选择出单一测试用例可以满足多个蕴涵项的测试用例集。确实, 这就是为什么上面选出的测试用例集 T_1 中仅含有两个元素。虽然这可以使测试人员最小化测试用例集, 但从测试每个蕴涵项可能单独带给一个谓词的贡献的角度而言, 这并不是一件好事。因此, 我们介绍一个保证蕴涵项独立性的方法。

第一步是得到一个DNF式子, 独立地满足其中的每个蕴涵项。幸运的是, 我们可以应用一个已存在的标准方法。蕴涵项的适当子项 (proper subterm) 是一个去除了一个或多个子项的蕴涵项。例如, abc 的适当的子项是 ab 和 b 。基本蕴涵项是这样的, 它使得没有蕴涵项的适当的子项是相同谓词的一个的蕴涵项。也就是说, 在一个基本蕴涵项中, 不可能移除一个项而不改变谓词的值。例如, 下面是在前面示例的基础上重构的一个表达式:

$$f(a,b,c) = abc \vee ab\bar{c} \vee b\bar{c}$$

abc 不是基本蕴涵项, 因为存在一个适合的子项 (即 ab), 它是一个蕴涵项。 $ab\bar{c}$ 也不是基本蕴涵项, 因为存在适合的子项 ab , 它是一个蕴涵项, $b\bar{c}$ 也是同样的情况。

我们需要一个附加的概念。如果一个蕴涵项可以省略而不改变谓词的值, 那么它就是冗

余的。例如，下面的表达式

$$f(a,b,c) = ab \vee ac \vee b\bar{c}$$

包含3个基本蕴涵项，但第一个 ab 是冗余的。一个DNF表述式是这样一种最简形式：所有蕴涵项均为基本蕴涵项并且不存在冗余。最简DNF表述式可以由代数方法计算得出，也可以通过卡诺图手工得出。这两种方法在前面章节已经讨论过了。

依据上面的定义，我们可以认为已经得到了最简DNF表述式。给定表达式 f 的最简DNF表述式，第 i 个蕴涵项的唯真点(unique true point)是这样一个真值点：第 i 个蕴涵项为真并且其他的蕴涵项均为假。值得注意的是，如果“使得其他蕴涵项为假”是不可能的，那么这个蕴涵项就是冗余的，这就违反了我们的假设： f 是最简DNF形式。唯真点的概念引出了一个新的标准：唯真点覆盖(UTPC)：

标准3.21 唯真点覆盖(UTPC)：给定谓词的一个最简形式的DNF表述式 f 和它的否定形式 $\bar{f}$ ，对于 f 与 $\bar{f}$ 中的每个蕴涵项， TR 包含一个唯一的取真点。

回到我们先前的示例：

$$f(a,b,c) = ab \vee b\bar{c}$$

$$\bar{f}(a,b,c) = \bar{b} \vee \bar{a}c$$

f 与 $\bar{f}$ 表述式均为最简DNF表述式。下面的表格给出了子句 a 、 b 、 c 所需的赋值，以满足每个蕴涵项：

	a	b	c
ab	T	T	
$b\bar{c}$		T	F
$\bar{b}$		F	
$\bar{a}c$	F		T

剩余子句的真值赋值必须确保其他蕴涵项不为真。对于第一个蕴涵项， ab 、 c 必须为true或第二个蕴涵项将为true。对于第二个蕴涵项 $b\bar{c}$ ， a 必须为false。对于第三个蕴涵项 $\bar{b}$ ，有3个唯真点的选择 $\{FFF, TFF, TFT\}$ 。上表中我们已列出了第一种选择。最后，对于第四个蕴涵项， $\bar{a}c$ 、 b 必须为true。用下面的表格来总结上述讨论：

	a	b	c
ab	T	T	t
$b\bar{c}$	f	T	F
$\bar{b}$	f	F	f
$\bar{a}c$	F	t	T

因此，下面的测试用例集满足UTPC：

$$T_2 = \{TTT, FTF, FFF, FTT\}$$

尽管IC相对较弱，但UTPC是一个相当强大的覆盖标准。值得注意的是，没有任何有效的或无效的子句覆盖标准是包含UTPC。这个结论可以经过简单的计数得出。一个具有 n 个子句的谓词表达式的最简DNF表述式中可能含有最多 2^{n-1} 个基本蕴涵项。因此UTPC可能需要指数级别的测试用例，这多于子句覆盖标准所需要 $n+1$ 个测试用例(线性的)。实际上，潜在的指

数爆炸不会自动使得UTPC无效。对于许多谓词而言，UTPC产生数量适度的测试用例。看待这一问题的一个方法是UTPC产生如DNF所要求数量的测试用例。换句话说，DNF表述式越复杂，测试用例的数量也就越多。

UTPC不包含有效子句覆盖标准。下面是一个反例。考虑如下谓词：

$$f(a,b,c) = ac \vee b\bar{c}$$

$$\bar{f}(a,b,c) = \bar{a}c \vee \bar{b}\bar{c}$$

可能的UTPC测试用例集为：

$$T_3 = \{TTT, TTF, FFT, FFF\}$$

c 决定 f 的值的条件计为 $a \oplus b$ 。值得注意的是， T_3 中所有的测试用例中 a 和 b 具有相同的值，因此 $a \oplus b$ 的取值总为false。换句话说，在这个测试用例集中， c 从不决定 f 的值。因此，UTPC甚至不包含GACC，更不用说CACC或RACC了。

文献中包含很多其他的DNF覆盖标准。这些覆盖标准的动机是检测某些确定范畴的错误的能力。我们已经有了一个对于UTP (unique true points) 的定义。我们需要一个近假点 (near false points) 的相应定义来清晰地定义这些标准。给定谓词 f 的DNF表述，对于 f 蕴涵项 i 中的子句 c 的近假点是真值的一个赋值使得 f 为false，但如果 c 为否定的并且所有其他的子句保持不变， i (因此，也就是 f) 的取值为true。例如，如果 f 为：

$$f(a,b,c,d) = ab \vee cd$$

于是，蕴涵项 ab 中的子句 a 的近假点为 $FTFF$ 、 $FTFT$ 、 $FTTF$ ，蕴涵项 ab 的子句 b 的近假点为 $TFFF$ 、 $TFFT$ 、 $TFTF$ 。唯真点-近假点对标准 (CUTPNFP) 的定义如下：

标准3.22 唯真点-近假点对覆盖 (CUTPNFP)：给定谓词 f 的最小DNF表述，对于每个蕴涵项 i 的每个子句 c ， TR 包含 i 的UTP和 i 中 c 的NFP，使得这两个点仅在 c 的值不同。

例如：

$$f(a,b,c,d) = ab \vee cd$$

如果我们考虑蕴涵项 ab 的子句 a ，我们选择其3个UTP中的一个，也就是 $TTFF$ 、 $TTFT$ 、 $TTTF$ ，并且与对应的NFP，即 $FTFF$ 、 $FTFT$ 、 $FTTF$ 两两配对。所以，例如我们可以选择第一对 $TTFF$ 、 $FTFF$ 来满足关于蕴涵项 ab 的子句 a 的CUTPNFP。同样地，我们也可以选择 $TTFF$ 、 $TFFF$ 对来满足关于蕴涵项 ab 的子句 b 的CUTPNFP，选择 $FFTT$ 、 $FFFT$ 对来满足关于蕴涵项 cd 的子句 c 的CUTPNFP，选择 $FFTT$ 、 $FFTF$ 对来满足关于蕴涵项 cd 的子句 d 的CUTPNFP。CUTPNFP集合的结果为：

$$(TTFF, FFTT, FTFF, TFFF, FFFT, FFTF)$$

注意前两个为UTP的测试用例，余下4个为响应的NFP的测试用例。

表3.8定义了DNF形式的谓词的语法错误集合[⊖]。图3.6给出了表3.8中错误类型之间的检测关系。具体来说，如果一个测试用例集是为了检测给定类型的错误，那么这个测试用例集也必定能够检测这个错误类型的下游类型错误。注意，ENF是一种异常简单的错误，任何测试用例集都可以检测到它。

⊖ 第5章中的关于变异操作符的概念与这里的错误类别概念有着很紧密的联系。

表3.8 DNF错误类别

错 误	描 述
ENF (表达式否定错误)	表达式错写为它的否定形式, 例如: $f=ab+c$ 写为 $f'=\overline{ab+c}$
TNF (项否定错误)	一个项错写为它的否定形式, 例如: $f=ab+c$ 写为 $f'=\overline{ab}+c$
TOF (项遗漏错误)	一个项丢失, 例如: $f=ab+c$ 写为 $f=ab$
LNF (字面量否定错误)	一个字错写为它的否定形式, 例如: $f=ab+c$ 写为 $f'=\overline{a}b+c$
LRF (字面量引用错误)	一个字错写为其他的字, 例如: $f=ab+bcd$ 写为 $f=ab+bcd$
LOF (字面量遗漏错误)	一个字丢失, 例如: $f=ab+c$ 写为 $f=a+c$
LIF (字面量插入错误)	一个字被错误地添加到了某个项, 例如: $f=ab+c$ 写为 $f'=\overline{a}b+\overline{b}c$
ORF+ (操作符引用错误)	逻辑“或”被错误的逻辑“与”替代, 例如: $f=ab+c$ 写为 $f=abc$
ORF* (操作符引用错误)	逻辑“与”被错误的逻辑“或”替代, 例如: $f=ab+c$ 写为 $f=a+b+c$

作为一个与DNF覆盖标准相关的错误类别的例子, 在TOF错误的上下文中考虑UTPC标准。UTPC可以有效地检测TOF错误。注意, UTPC需要忽略蕴涵项中的一个UTP。因为取真点的唯一性, 所以这个测试用例中的其他蕴涵项将不能碰巧产生这个正确的真值。因此, 实现中将生成测试用例中所需真值的否定形式, 这就可以用来揭示TOF错误。对于图3.6中给出的检测关系我们可以推断出UTPC也可以检测ORF+、LNF、TNF和ENF类型的错误。另一个例子是CUTPNFP可以有效地检测LOF错误。原因是对于项 i 中的每个子句 c , CUTPNFP需要一个UTP和一个NFP。这两个测试仅在子句 c 的取值不同。因此, 如果 c 的字在实现中被错误地删除, 这两个测试将产生相同的真值, 因此就可以揭示出这个错误。给定图3.6中的检测关系, 我们可以推断出CUTPNFP可以检测ORF*、LNF、TNF和ENF类型的错误。另外, 虽然CUTPNFP不包含UTPC的子集, CUTPNFP仍可以检测UTPC能够检测的错误。CUTPNFP不一定检测出LIF错误, 但是它是包含RACC的。对于其他检测方法, 我们给出文献供读者参考, 文献中定义了其他更加有效(同时也非常昂贵)的标准, 其中一些的确能检测LIF。

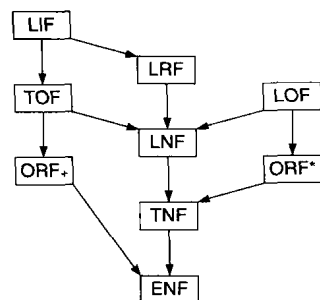


图3.6 错误检测关系

卡诺图

在本节, 我们回顾一下卡诺图, 这对于生成谓词的少量子句的DNF表述是非常有用的。

卡诺图是用特殊属性的表格形式表述一个谓词, 相邻的表项组合对应于简单DNF表述。卡诺图对于谓词的子句数量小于等于5个时是非常好用的, 如果超过了, 那么就变得非常繁琐。一个具有4个子句的谓词的卡诺图如下所示:

		ab			
		00	01	11	10
cd	00			1	
	01			1	
	11	1	1	1	1
	10			1	

谓词“ $ab \vee cd$ ”的卡诺图表

暂且假设表中的条目限制为真值。在具有 2^n 个单元的表格中， n 个子句有 2^{2^n} 种分配方案。于是上面表格中具有 $2^4=16$ 个单元，4个子句共有 $2^{16}=65\,536$ 种可能的分配方案。读者可以放心，我们不会在本文中罗列出所有的这些可能。值得注意的是，在行和列的边上真值的标签。特别是，注意任意的相邻单元的邻接对的真值正好只差一个子句。把卡诺图的边也想象成相互连接的是有帮助的，即最上端与最下端是相邻的，最左端和最右端是相邻的（也就是从2维空间到3维空间的邻接映射）。

上面的卡诺图的特殊功能可以表述为下面的完整形式：

$$ab\bar{c}\bar{d} \vee ab\bar{c}d \vee abcd \vee abc\bar{d} \vee \bar{a}\bar{b}cd \vee \bar{a}bcd \vee a\bar{b}cd$$

这个表述可以简化为：

$$ab \vee cd$$

这个简化形式可以由卡诺图得到，通过把邻接单元分组到一个大小为 2^k 的矩形中（ $k>0$ ），并且对于那些没有邻接单元的单元形成大小为1的矩形，允许分组间的重叠。我们给出了一个有3个子句的例子。考虑下面的卡诺图：

		a, b			
		00	01	11	10
c	0		1	1	
	1	1		1	1

可以从图中抽取出4个大小为2的矩形。它们分别对应于函数 $b\bar{c}$ 、 ab 、 ac 和 $\bar{b}c$ ，并且由如下的卡诺图表述：

		a, b			
		00	01	11	10
c	0		1	1	
	1				

		a, b			
		00	01	11	10
c	0			1	
	1			1	

		a, b			
		00	01	11	10
c	0				
	1			1	1

		a, b			
		00	01	11	10
c	0				
	1	1			1

首先, 这些图的最后一个很难被看做一个矩形, 但是请记住卡诺图边界从左到右和从上到下是联系在一起的。我们可以将原始功能函数写成4个卡诺图的析取, 每个给出了一个基本蕴涵项, 但是请注意第2个卡诺图描述的 ab , 实际上相对于其他3个是冗余的, 因为它的条目中的元素均可以被其他的卡诺图覆盖。最终的最简DNF表达式为:

$$f = b\bar{c} \vee ac \vee \bar{b}c$$

还应该注意 ac 的所有条目可以被其他的卡诺图覆盖, 因此对于其他3个蕴涵项而言 ac 是冗余的。所以不同的最简DNF表述为:

$$f = b\bar{c} \vee ab \vee \bar{b}c$$

DNF的否定形式也可以很容易地从卡诺图中引出。重新考虑上面的表达式 f , 其否定域的卡诺图为:

		a, b			
		00	01	11	10
c	0	1			1
	1		1		

这里, 卡诺图中的三个单元可以用两个矩形覆盖, 一个大小为2, 另一个大小为1。结果是无冗余的, 基本蕴涵项为:

$$\bar{f} = \bar{b}\bar{c} \vee \bar{a}bc$$

卡诺图是一种非常方便的符号标记方法, 可以为许多逻辑覆盖标准生成测试集。例如, 重新考虑谓词 $ab \vee cd$ 。一个简单的矩形就可以覆盖UTP。因此, $ab \vee cd$ 中除了TTTT以外所有的真值点都是UTP。对于任意给定真值点的NFP为那些卡诺图中直接相邻的假值点。为了确定UTPC测试集, 只需简单地取出 f 与 $\bar{f}$ 的卡诺图, 对于每个蕴涵项选择只被该蕴涵项覆盖的单元。对于CUTPNFP而言, 要把UTP与NFP配对, 应该小心为 f 的每个子句获得的配对。把真值点与NFP配对是生成RACC测试用例的一种简易方法。注意, 对于RACC测试用例来说, 它不关心真值点是否是唯一的。

3.6节练习

根据下列表达式回答下面的问题。

1. $f = ab\bar{c} + \bar{a}b\bar{c}$
2. $f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$
3. $f = ab + \bar{a}\bar{b}c + \bar{a}\bar{b}\bar{c}$
4. $f = \bar{a}\bar{c}\bar{d} + \bar{c}d + bcd$

- (a) 画出 f 与 $\bar{f}$ 的卡诺图。
- (b) 找出 f 与 $\bar{f}$ 非冗余的基本蕴涵项表述。
- (c) 给出 f 的满足蕴涵项覆盖(IC)的测试用例集。

- (d) 给出 f 的满足UTPC的测试用例集。
- (e) 给出 f 的满足CUTPNFP的测试用例集。
5. 根据下列谓词回答问题（最简意味着最少的变量引用数目）。
- $W = (B \wedge \neg C \wedge \neg D)$
 - $X = (B \wedge D) \vee (\neg B \wedge \neg D)$
 - $Y = (A \wedge B)$
 - $Z = (\neg B \wedge D)$
- (a) 画出谓词的卡诺图。其中 AB 在上面， CD 在侧面。用 W 、 X 、 Y 和 Z 标示每个单元。
- (b) 用最简表述描述具有多个定义所有的单元。
- (c) 用最简表述描述没有定义的所有单元。
- (d) 用最简表述描述 $X \vee Z$ 。
- (e) 给出表达式 X 的一个测试用例集，每个基本蕴涵项仅使用一次。
6. 给出一个反例说明CUTPNFP不是UTPC的子集。提示：可以使用上面给出的表达式 $f = ab + cd$ 。

3.7 参考文献注释

有效子句标准最初似乎是Myers在1979年的书中[249]提出的。Zhu发表了更进一步的论文[367]。他定义了判断和条件覆盖，并被Chilenski和Miller作为MCDC [73, 305]的概念基础。这本书中最初的定义符合GACC并且没有说明次子句是否必须具有与主子句相同的值。Chilenski也强调，其缩略方式应该为“MCDC”，而不是“MC/DC”，并且他从来没有使用过中间的“/” [72]。许多航空协会的成员把MCDC解释为次子句的值必须相同，并把这解释为“unique-cause MCDC” [72]。Unique-cause MCDC对应于我们提出的RACC。最近，FAA接受了次子句的值可以不同的观点，这被称做“masking MCDC” [74]。Masking MCDC对应于我们的CACC。在我们以前的论文[17]中明确定义了在这本书中所使用的形式，并且引入术语“CACC”。

无效子句标准适应由Vilkomir和Bowen [332]提出的RC/DC方法。

内部变量问题是不可判定的，这个结果来自于Offutt的博士论文[101, 262]。这个问题在自动生成测试数据的著作中[36, 39, 101, 102, 150, 176, 179, 190, 191, 243, 295, 267]很重要。

Jasper等人提出了一种生成满足MCDC[177]测试用例的技术。他们采用Chilenski和Miller在论文中定义的MCDC，采用默认的解释：次子句必须具有和主子句相同的值。他们做了进一步的修改，使得如果两个子句是耦合的，也就是说一定不满足定义，那么这两个子句允许次子句具有不同的值。实际上，仅当子句耦合时允许不同的值，使得他们关于MCDC的解释介于在这本书中的RACC和CACC之间。

Weyuker、Goradia和Singh提出的关于生成软件规范测试数据的技术局限于布尔变量[342]。从杀死变异（第5章将引入该概念）的测试用例能力来比较这些技术[99, 101]。结果是，他们的技术非常接近MCDC，性能优于任何其他的技术。Weyuker等人把语法和语义结合到了他们的标准中。他们提出了一个概念，被称做意味深长的影响，这个概念与决定概念相关，但具有语法基础而无语义基础。

Kuhn 研究了满足各种基于判断标准生成测试用例的方法, 包括MCDC测试用例[194]。他使用了Chilenski和Miller[73, 305]的定义, 并且提出了满足MCDC的布尔衍生形式。实际上, 这是以一种满足CACC的方法来解释MCDC。

Dupuy和Leveson在2000年的论文中采用实验的方法评估了MCDC[108]。他们采用实验方法展示了结果, 比较纯功能测试与经过MCDC方法增强的功能测试方法。实验数据来源于科学卫星的高度控制软件HETE-2 (High Energy Transient Explorer)的测试过程。他们关于MCDC的定义来源于FAA报告中的传统定义和Chilenski和Miller的论文: “程序中的每个入口和出口至少被执行一次, 判定中的每个条件的可能结果至少出现一次, 并且每个条件对判定结果的影响是独立的。当判定中的其他可能的条件保持不变时, 一个条件能够独立地影响判定的输出结果。”

注意下面所述的错误: “仅仅改变那个判定” 应该是 “仅仅改变那个条件”。这不是说当条件的值改变时判定具有不同的值。“保持不变” 可以理解为次子句不能随着主子句变化为不同的值 (这是RACC而不是CACC)。

Offutt、Liu、Abdurazik 和 Ammann [272]提出的完全谓词方法明确地放松了主子句必须与谓词具有相同的值的要求。这与CACC是等价的, 几乎与masking MCDC相同。

Jones和Harrold已经提出了方法来减少为满足MCDC进行的回归测试[180]。他们对MCDC进行了如下定义: “MC/DC是判定 (分支) 覆盖的严格的形式……MC/DC要求判定中的每个条件必须独立影响判定的最终结果。” 这直接来源于Chilenski 和Miller的原始论文, 并且他们论述的定义与CACC是相同的。

SCR由Henninger [157]第一次提出, 并且被应用于由Atlee [20, 21]进行模型检测和测试。

本书中判定给定的 pc 的方法使用了由Akers [6]提出的布尔引申。Chilenski和Richey [74]和 Kuhn [194]都应用了Akers的引申来严格定义本章中给出的问题。另一种方法是Chilenski 和Miller提出的配对表方法和树方法, Chilenski和Richey[74]和Offutt等人 [272]也独立发现了树方法。树方法用程序的形式实现了布尔引申方法。

有序二项决策图 (OBDD) 提供了另外一种决定 pc 的方法。特别是, 考虑任意OBDD, 子句 c 排序最后。实际上, 经过OBDD中被标示为 c 的节点 (可能有0个、1个或2个这样的节点) 的任何路径都是其他变量的赋值使得 c 决定 p 。继续延伸这个路径到常数 T 和 F , 生成对于 c 满足RACC的测试对。选择两条不同的路径到达 c 标示的节点, 并且延伸每一条路径, 使得其中一条到达 T 另一条到达 F , 生成对于 c 满足CACC但不满足RACC的测试用例对。最后, 如果两个节点都标示为 c , 那么可能相对于 c 满足GACC但不满足CACC。选择到达标示为 c 的两个节点的两条路径, 选择 c 为真延伸一条路径, 并且选择 c 为假延伸另一条路径。这两条路径必然会终止于相同的节点 T 或 F 。对于 c 的ICC测试通过考虑在OBDD中到 T 和 F 的路径来生成, OBDD中路径不包括变量 c 。使用OBDD来生成ACC或ICC测试的最吸引人的方面是大量的现有工具可以处理相对大量的子句。最不吸引人的的是对于具有 N 个子句的谓词, 对于给定的功能需要 N 个不同的OBDD, 因为被关注的子句需要是有序序列中的最后一个。根据本书作者的了解, 使用OBDD来生成ACC或ICC测试没有在文献中出现。

Beizer的书[29]中有关于DNF测试的一个章节, 包含了变异了的IC覆盖, 仅包含 f 而不包含 $\bar{f}$, 并且对卡诺图进行了扩展研究。我们不考虑以合取式 (CNF) 表示的谓词中定义的标准。原因是每一个DNF覆盖标准在CNF中都有一个对偶存在。Kuhn [194]第一个展示了缺陷检测

关系。Yu、Lau和Chen在很大程度上扩充了这一工作，发展了针对错误检测能力DNF覆盖标准的关键元素。Chen和Lau [63]发表了两篇很好的论文，对这个主题进行了一定的研究，提出了一系列包括CUTPNFP的覆盖标准，Lau和Yu [202]给出了图3.6中所示错误等级关系。在个人通信中，Greg Williams和Gary Kaminski提供给作者很有价值的帮助用以组织和扩展DNF错误检测材料。Greg Williams也提出了反例用以说明UTPC和ACC标准之间不共享包含关系。

第4章 输入空间划分

在一个非常基础的方法里，所有的测试都是指从待测试的软件的输入空间中选择元素。前面提出的标准可以看做是通过测试需求来定义划分输入空间的方式。这个设想就是，满足相同测试需求的值集合都将是“恰好的”。输入空间划分用更直接的方式采取了那种看法。根据输入参数可能有的值定义了输入域。输入参数可能是方法的参数或者全局变量，或者表现当前状态的对象，或者是用户级给一个程序的输入，它们依赖于什么样的软件产品将要被分析。然后输入域被划分成若干个区域，这些区域从测试的观点来看都被假定包含了等同的有用的值，并且从每一个区域中选出这些值来。

这种测试方法有一些优势。因为它不需要自动化操作就可以使用，并且只需要很少培训，所以很容易实施。测试人员没有必要理解具体实现，所有的事情都是基于对输入的描述。它也可以简单地作为调整技术来获得更多或更少的测试。

考虑某区域 D 上的一个抽象的划分 q 。这个划分定义了一组同价类，我们也可以简单地把它们叫做块， $B_q^\ominus$ 。这些块是两两分离的，即

$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

并且这些块覆盖了区域 D ，即

$$\bigcup_{b \in B_q} b = D$$

在图4.1中，输入域 D 被分成了3个块， b_1 、 b_2 和 b_3 。这种划分定义了在每个块里包含的值，并且根据有关软件将要做的事情的知识来设计这种划分。

对测试过程来说，划分覆盖里的思想就是，在一个块里的任何测试都是和其他测试一样好的。有时候，几个划分被放在一起考虑，在这样做的时候，如果不注意就会导致测试用例的组合产生爆炸，会出现非常多的测试用例。

执行输入空间划分的一般方法是从考虑每个参数的域开始，把每个域的可能值划分进块里，然后为每个参数组合这些变量。有时参数被完全孤立地考虑，有时被关联在一起考虑，通常考虑的都是程序的语义。这个过程称为输入域建模，下一节将给出详细内容。

每一个划分通常都基于相应程序的某种特性 C 、程序的输入、程序的环境等。下面是一些可能的特性的例子：

- 输入 X 是空的。
- 文件 F 的次序（顺序、反序、任意）。
- 两个航天器之间最小的距离。

每一个特性 C 允许测试员去定义一个划分。形式上，一个划分必须满足两个属性：

$\ominus$ 为简单起见，我们选择使用块。在有关文献中，这些块也称为“划分”。



图4.1 输入域 D 划分成了3个块

1. 这个划分必须覆盖整个域（完整性）。

2. 块不能有重叠（分离性）。

作为一个例子，考虑上面提到的这个特性“文件 F 的次序”。这可以用来创建下面的（不完整的）的划分：

- 文件 F 的排序

b_1 = 按升序排序

b_2 = 按降序排序

b_3 = 任意顺序

可是，这不是一个有效的划分。更具体地说，如果这个文件的长度是0或者1，那么这个文件就会属于所有这3个块。那就是说，这些块不满足分离性。解决这个问题的最简单的策略就是保证每一个特性仅仅对应一个属性。上面的问题就出在按升序排序的概念和按降序排序的概念混在相同的特性里。分裂成两个特性，即升序排序和降序排序，这样就解决了问题。结果就是下面（有效的）两个特性的划分。

- 文件 F 升序排序

b_1 = 真

b_2 = 假

- 文件 F 降序排序

b_1 = 真

b_2 = 假

用这些块，长度为0或1的文件对这两个特性来说都属于这个真块。

由于非常实际的原因，完整性和分离性被形式化，这并不仅仅是因为数学表现形式的时髦。不完整或者没有分离性的划分反映了该划分在基本原理中不够清楚。尤其是，如果一个划分实际上译成了两种或者3种基本原理，这个划分就很可能相当凌乱，并且它也可能违反完整性和分离性之一（甚至是二者都违反）。识别和修改完整性或分离性的错误，显而易见，它将会产生更令人愉快的划分。更进一步来说，形式上令人讨厌的划分会在生成测试的时候引起不必要的问题，就像下面将讨论的那样。本章的大部分地方都假定这些划分既满足完整性又满足分离性。

4.1 输入域建模

在输入域建模的第一步是识别可以测试的函数。我们来看看第3章里的TriTyp程序。这个函数仅有一个可以测试的函数，该函数有3个参数。对Java 类API来说，情况会更复杂。每个公共的方法明显是一个应该被独立测试的可测试函数。然而，对几个方法来说，这些特性往往都相同，所以就可以为整个类开发一组共同的特性，然后再为每一个方法开发针对本身的特别的测试。最终，大的系统适于输入空间划分方法的检验，同时这种系统一般都提供复杂的功能。像UML（统一建模语言）用例那样的工具能用来识别可测试的函数。每个用例与系统将要实现的一个具体的功能相关联，因此，用例设计者非常有可能在他们的脑子里就有对开发测试用例来说有用的特性。例如，一个ATM（自动取款机）的取款用例识别“提取现金”作为一个可以测试的函数。更进一步说，它建议了有用的分类，例如“卡是否有效”和“提取策略和提取请求之间的关系”。

第二步就是识别所有的参数，这些参数能影响给定的可以测试的函数的行为。这个阶段不需要特别的创造性，但是对完整的执行测试来说是很重要的。在测试一个无状态方法的简单情况下，参数是简单的形参。如果这个方法有状态，那么这个状态必须作为一个参数被包含，在面向对象的类里，这种情况是相当普遍的。例如，插入方法insert (Comparable obj) 对一个二叉树类来说表现不同，这依赖于obj是否已经存在于树中。因此，树的当前状态需要被明确地识别，把它作为一个参数给插入方法insert()。在一个更加复杂一些的例子中，查找方法find(String str)从文件中查找str的位置，它显然要依赖于将要被搜索的文件。因此，测试工程师明确地识别这个文件，把它作为一个参数给查找方法find()。总的来说，被测函数的所有参数构成了输入域。

第三步，也是关键的创造性的工程步骤，是将前面相关的步骤建模。一个输入域模型(IDM)用一种抽象的方式表现了被测系统的输入空间。一个测试工程师根据输入特性来描述输入域的结构。测试工程师为每个特性创建一个划分。划分是多个块的组合，其中每个块都包含了一组值。从那种特殊性的观点来看，每一个块中的值都被看做是相等的。

一个测试输入是一个元组的值，每个值对应一个参数。根据定义，测试输入严格的属于来自每个特性的一个块。这样，即使我们有中等数目的特性，那么可能出现的组合的数量很可能就不可实现。尤其是，添加另外一个有 n 个块的特性会增加 n 倍组合个数的增加。因此，针对输入域测试来说，控制组合的总的个数是任何实际的方法的一个关键的特征。在我们看来这是覆盖标准的工作，我们将在4.2节介绍它。

不同的测试会得出不同的模型，这依赖于创造性和经验。这些不同点在结果测试的质量里会创造一些潜在的差异。本章中所提出的支持输入域建模的结构化方法能够减少这种变异并且增加IDM整体的质量。

一旦IDM被建立，并且那些值被识别，那么那些值的组合中就有一些是无效的。IDM必须包含能帮助测试人员识别、避免或者移除无效的子组合的信息。这个模型需要一种方式来表现这些约束。这些约束将在4.3节里进一步讨论。

下一节提供了两个不同的方法来进行输入域建模。基于接口的方法是直接从被测程序的输入参数来开发特性。基于功能性的方法是从被测程序的一个功能上或者行为上的观点来开发特性。测试人员必须选择使用哪一种方法。一旦这个输入域模型被开发出来，几个覆盖标准就可以用来决定使用哪些组合值来测试软件。这些将在4.2节里讨论。

4.1.1 基于接口的输入域建模

基于接口的方法孤立地看待每个特殊的参数。这种方法几乎是机械式的遵循，但是结果产生的测试确实相当好。

使用基于接口的方法的一个明显的优势就是它很容易识别特性。事实就是每个特性把它自身限于一个单独的参数，这也使得把抽象的测试转换为可执行的测试用例变得容易。

这种方法的一个缺点是，对测试工程师来说，并不是所有可用的信息都能被反映在接口域模型里。这就意味着IDM可能是不完整的，因此也就需要额外的特性。

另外一个缺点就是，功能性上的某些部分可能依赖几个接口参数的特定组合值。在基于接口的方法里，每个参数都进行孤立地分析，结果导致重要的子组合可能被遗漏。

看看第3章里的TriTyp程序。它有3个整型参数，分别代表了一个三角形的3条边。在一个

基于接口的IDM里,边1会有大量的特性,边2和边3也是如此。由于这三个变量都是相同类型的,所以针对每一条边的基于接口的特性都是相似的。例如,因为边1是一个整数,并且对整数来说,0是一个特殊的值,边1对0的关系就是一个合理的基于接口的特性。

4.1.2 基于功能的输入域建模

基于功能的方法,这个想法是识别被测系统的相应指定的功能,而不是使用实际的接口。这就允许测试人员把一些语义或者领域知识合并到IDM。

测试领域的一些成员相信一个基于功能性的方法能比基于接口的方法产生更好的测试用例,因为基于功能的IDM包含了更多的语义信息。从需求规格说明书转换更多的语义信息到IDM,这使得它更可能为测试用例产生期望的结果,这是一个重要的目标。

基于功能性的方法的另外一个重要的优势是,在软件实现之前,软件的需求是可以获取的。这就意味着输入域建模和测试用例生成都能在软件开发的早期着手进行。

在基于功能的方法里,识别特性和取值可能会非常难。如果系统庞大且复杂,或者需求规格说明书是不规范和不完全的,那么设计合理的特性就可能非常难。下一节对设计特性给了几个实际的建议。

基于功能性的方法也使生成测试变得更难。IDM的特性经常不能够匹配软件接口的单个参数。把这些值转换进可执行的测试用例也比较难,因为一个单独的IDM特性的约束可能会影响接口中的多个参数。

回头看看第3章的TriTyp程序,基于功能性的方法会认为,这个方法的输入是一个三角形,而不是简单地输入3个整数。这就导致了一个三角形的特性,它能够被划分成不同类型的三角形(就像下面所讨论的)。

4.1.3 识别特性

在基于接口的方法里识别特性是简单的,就是一个机械的从参数到特性的转换。开发一个基于功能性的IDM就要难一些。

前置条件对识别基于功能性的特性来说是极好的来源。它们在软件里作为异常行为可能是明确的或者是编码的。前置条件明确地使得已定义的(或正常的)行为和未定义(或异常的)行为相分离。例如,如果假设方法choose()选取一个值,它需要一个前置条件,即一个值必须可以用来选择。一个特性就可能是这个值是否可以取得。

对特性来说,后置条件也是一个好的资源。在TriTyp的例子中,不同种类的三角形都基于该方法的后置条件。

测试工程师还应该查找变量之间的其他关系。这些可能是明确的,也可能是不明确的。例如,给一个好奇的测试工程师一个有两个对象参数x和y的方法m(),他可能想知道如果x和y指向相同的对象(别名),或者指向逻辑上相等的对象,会得到什么样的结果。

另外一个可能的想法就是为可能遗漏的因素做检查,即那些可能会影响执行,但是却没有一个有关的IDM参数的因素。

通常情况下,有许多特性的块很少,这比相反情况要好。有少量块的特性更可能满足分离性和完整性的属性,这也是正确的。

通常,对测试工程师来说,使用需求规格说明书或者其他文档来开发特性要比使用代码

来开发特性更好一些。这个想法是测试工程师应该通过使用关于问题的领域知识来实行输入空间划分，而不是软件的实现。然而，在实际中，代码可能都是可用的。总之，测试工程师能体现特性里的语义消息越多，最后的测试结果集合就越好。

这两种方法一般都会导致不同的IDM特性。下面的方法说明了这种不同：

```
Public boolean findElement(List list, Object element)
// 效果:如果 list或 element 是 null 抛出异常NullPointerException
// 否则如果element 在list 中返回true, 其他情况返回false
```

如果基于接口的方法被使用，那么IDM就会有对list的特性和对element的特性。例如，这里是两个对list的基于接口的特性，包含了一些块和一些值，这些将在下一节里更加详细地讨论：

- list是无效的

$b_1 = \text{True}$

$b_2 = \text{False}$

- list是空的

$b_1 = \text{True}$

$b_2 = \text{False}$

基于功能性的方法导致了更加复杂的IDM特性。像之前提到的那样，基于功能性的方法，在测试工程师方面来说需要更多的思考，但是能产生更好的测试。这个例子的两种可能性都列在下面，再一次包含了块和值。

- 在列表里元素出现的次数

$b_1 = 0$

$b_2 = 1$

$b_3 = \text{超过一次}$

- 在列表里第一次出现的元素

$b_1 = \text{True}$

$b_2 = \text{False}$

4.1.4 选择块和值

在选择了特性之后，测试工程师把这些特性的域划分到叫做块的值的集合中。在任何一种划分方法中的一个关键的问题是，如何能够识别划分，以及怎样使有代表性的值能够从块中被选择出来。这是另外一个创造性的设计步骤，它允许测试工程师调节测试过程。更多的块能产生更多的测试，也需要更多的资源，但有可能找到更多的错误。较少的块会产生较少的测试，也节约资源，但是有可能降低测试的有效性。下面是几个一般的识别值的策略：

- **有效值**：包含至少一组有效的值。
- **子划分**：一系列有效值通常能被划分成子划分，这样每一个子划分执行功能性上有稍微不同的部分。
- **边界**：接近边界的值经常会出问题。
- **正常使用**：如果操作的侧面主要集中在“正常使用”上，失败率就依赖于不是边界条件的那些值。

- **无效值**：至少包含一组无效的值。
- **平衡**：从成本的角度来看，添加更多块给具有极少块的特性，它可能更便宜，甚至是免费的。在4.2节里，我们会看到测试的数量有时候依赖于特性所具有的块的最大数量。
- **遗漏的划分**：检查一个特性的所有块的联合，它完全覆盖了那个特性的输入空间。
- **重叠的划分**：检查没有属于超过一个块的值。

特殊的值能经常被使用。考虑一个Java引用变量，null是典型的一个特殊的例子，它需要与非无效值不同的处理。如果这些引用是对一个容器结构，例如一个组或一个列表，那么这种容器是否为空通常都是一个有用的特性。

再看看第3章的TriTyp程序。它有3个整型的参数代表了一个三角形的三条边的长度。对一个整型变量来说，一个一般的划分考虑在可测试的函数域里是把变量值与一些特殊的值相关联，比如0。

表4.1显示了TriTyp程序的基于接口测试的IDM的一个划分。它有 q_1 、 q_2 和 q_3 三个特性。

表里的第一行应该被读做“块 $q_1.b_1$ 是说边1要比0大”，“块 $q_1.b_2$ 是说边1是等于0”，并且“块 $q_1.b_3$ 是说边1要小于0”。

表4.1 TriTyp的输入的第一次划分（基于接口）

划分	b_1 (块1)	b_2 (块2)	b_3 (块3)
q_1 = “边1与0的关系”	大于0	等于0	小于0
q_2 = “边2与0的关系”	大于0	等于0	小于0
q_3 = “边3与0的关系”	大于0	等于0	小于0

考虑对边1的划分 q_1 。如果从一个块里选一个值出来，结果就是3个测试。例如，我们可能在测试1里选择边1等于7，在测试2里选择边1等于0，在测试3里选择边1等于-3。当然，我们也需要这个三角形的边2和边3的值来完成测试用例值。注意，某些块表达的是有效的三角形，某些块表达的是无效的三角形。例如，一个无效三角形，它的边可能会是一个负值。

表4.2 TriTyp的输入的第二次划分（基于接口）

划分	b_1 (块1)	b_2 (块2)	b_3 (块3)	b_4 (块4)
q_1 = “边1的长度”	大于1	等于1	等于0	小于0
q_2 = “边2的长度”	大于1	等于1	等于0	小于0
q_3 = “边3的长度”	大于1	等于1	等于0	小于0

如果预算允许，很容易细化这个分类来得到更细粒度的测试。例如，能够通过用值1分离输入来产生更多的块。这个结果就会产生一个有4个块的划分，如表4.2所示。

注意，如果边1的值是浮点数而不是整数，第二种分类将不会产生有效的划分。没有一个块会包含从0到1的值（不包含），因此这些块不会覆盖域（不完整）。然而，域 D 包含整数，因此划分就是有效的。

当划分的时候，对测试人员来说为测试中的每个块识别候选值通常都是有用的。现在识别值的原因就是选择具体的值能够帮助测试工程师更具体地思考描述每个块的谓词。虽然这些值不能够充分地证明把测试需求细致到测试用例，但是它们构成了一个好的起点。表4.3显示了能够满足第二种划分的值。

上面的划分是基于接口的，而且仅仅使用了有关那个程序的语法信息（它有3个整数输入）。基于功能性的方法能够使用三角形传统的几何分类的语义信息，如表4.4所示。

表4.3 在表4.2里第二次划分对块的可能的取值

参数	b_1 (块1)	b_2 (块2)	b_3 (块3)	b_4 (块4)
边1	2	1	0	-1
边2	2	1	0	-1
边3	2	1	0	-1

表4.4 TriTyp的输入的几何划分 (基于功能性的)

划分	b_1 (块1)	b_2 (块2)	b_3 (块3)	b_4 (块4)
q_1 = “几何分类”	不等边三角形	等腰三角形	等边三角形	无效的

当然, 测试人员必须知道什么使得一个三角形是不等边三角形、等边三角形、等腰三角形, 以及选择哪些可能的值是无效的 (这可能是简单的中学几何, 但是可能许多人都忘记了)。等边三角形所有的边都是相等的。等腰三角形至少有两条边是相等的。其他任何有效的三角形都是不等边三角形。这就带来了一个很小的问题, 表4.4不能构成一个有效的划分。等边三角形也是等腰三角形, 所以我们必须首先修改这个划分, 如表4.5所示。

表4.5 改正TriTyp的输入的几何划分 (基于功能性的)

划分	b_1 (块1)	b_2 (块2)	b_3 (块3)	b_4 (块4)
q_1 = “几何分类”	不等边三角形	等腰三角形、非等边三角形	等边三角形	无效值

现在可以选择表4.5的值如表4.6所示。三元组表示三角形的三个边。

与上述等边/等腰问题的方法不同, 该方法将特性几何划分分解为4个不同特性, 即不等边、等腰、等边和无效。每个特性的划分是布尔形, 而选择“等边=true”也意味着“等腰=true”则只是一个约束。非常推荐该方法, 它总是满足不相交且完整的属性。

4.1.5 使用一种以上的输入域模型

对一个复杂的程序来说, 建几个小的输入域模型要比只有一个大的模型好一些。当建模特性和块的时候, 这种方法允许使用分治的策略。对相同的软件采用多个输入域模型的另外一个好处是它允许各种级别的覆盖。

例如, 一个输入域模型可能仅仅包含一个有效的值; 另外一个输入域模型包含无效的值来集中处理错误。有效值的输入域模型可能使用一个较高级别的覆盖。无效值的输入域模型可能使用一个较低级别的覆盖。

只要产生的测试用例有意义, 多个输入域模型可以是重叠的。然而, 重叠的输入域可能有多重约束。

4.1.6 检查输入域模型

检查输入域模型是很重要的。通过这些特性, 测试工程师应该看看是否有与函数行为有关的信息没有考虑进一些特性。这必然是一个不正式的过程。

测试人员也应该清楚地检查每个特色的完整性和分离性。这个检查的目的是确保, 对每个特性来说, 不仅仅这些块覆盖了整个输入空间, 而且选择一个特殊的块意味着在那个特性里排除了其他的块。

如果使用了多个IDM，完整性应该相对于输入域的一部分，该部分输入域在每一个输入域模型中被建模。当测试人员满足这些特性和它们的块的时，就到了在块中选择哪一种组合值来测试和识别约束的时间了。

表4.6 在表4.5里用几何划分的块可能的取值

参数	b_1 (块1)	b_2 (块2)	b_3 (块3)	b_4 (块4)
三角形	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

4.1节练习

1. 针对下面的search()方法回答以下问题：

```
public static int search (List list, Object element)
// 效果：如果list或者element是空的就抛出NullPointerException异常
// 否则如果element是在list中，返回element 在list中的一个索引，其他情况下返回-1
// 例如，search([3,3,1],3) = 0 或者1
// search([1,7,5],2) = -1
```

你的答案是基于下面的特性划分：

特性：element在list中的位置

块1：element 是list中的第一条记录

块2：element 是list中的最后一条记录

块3：element 在list中既不是第一也不是最后一条记录

(a) “element 在list中的位置”不满足分离属性。举例说明这个问题。

(b) “element 在list中的位置”不满足完整性。举例说明这个问题。

(c) 提供一个或多个新的划分来捕获“element 在list中的位置”这个特性的意图，并不满足完整性和分离性。

2. 用下面的方法记号来为GenericStack类得出输入空间划分的测试：

- public GenericStack();
- public void Push(Object X);
- public Object Pop();
- public boolean IsEmt();

假定这个栈是通常意义上的栈。尽力让你的划分简单，选择少量的划分和块。

(a) 定义输入的特性。

(b) 把这些特性划分成具体的块。

(c) 定义块中的值。

4.2 组合策略标准

以上描述忽略了一个很重要的问题：“我们应该考虑如何同时进行多重划分？”这和问“我们应该从哪些块的组合中选值”是一样的。例如，我们可能希望要求一个测试用例从 q_2 满足块1和从 q_3 满足块3。最明显的选择是选择所有的组合。然而，就像前面章节介绍的组合覆盖，当定义了大于2或3个划分，用所有的组合是不实际的。

标准4.23 完全组合覆盖 (ACoC)：来自所有特征的所有块的组合必须被用到。

例如, 如果我们有3个划分blocks[A, B]、[1, 2, 3]和[x, y], 那么ACoC需要下面12个测试:

(A, 1, x) (B, 1, x)

(A, 1, y) (B, 1, y)

(A, 2, x) (B, 2, x)

(A, 2, y) (B, 2, y)

(A, 3, x) (B, 3, x)

(A, 3, y) (B, 3, y)

一个满足ACoC的测试集对于每个划分每个块的组合有一个唯一的测试。测试的个数是每个划分中块的个数的乘积: $\prod_{i=1}^Q (B_i)$ 。

如果我们使用4块划分, 类似于 q_2 对三角形3条边中的每条边, ACoC要求 $4 \times 4 \times 4 = 64$ 个测试。

几乎可以确定的是要进行比必要的测试更多的测试, 并且这种测试在经济方面通常也是不实际的。因此, 正如前面的路径与真值表, 我们必须用某种覆盖标准来选择从哪些块中取值。

首先, 基本的假设是从测试的角度看, 从同一个块中选不同的值是等价的。也就是说, 我们需要从每个块中只取一个值。存在一些组合策略, 其结果是一组有用的标准。这些组合策略通过TriTyp例子进行了说明, 该例子使用了表4.2中的第二个分类和表4.3中的值。

第一个组合策略标准相当简单易懂, 并且简单地要求我们尝试每个选择至少一次。

标准4.24 每个选择覆盖 (ECC): 对于每个特征, 选自每个块的一个值必须至少在一个测试用例中使用。

考虑上面3个划分的块的例子[A, B]、[1, 2, 3]和[x, y], ECC可以通过多种方式实现, 包括3个测试 (A, 1, x)、(B, 2, y) 和 (A, 3, x)。

假定被测程序有Q个特征, $q_1, q_2, \dots, q_Q$, 并且每个特征 q_i 有 B_i 个块。那么一个满足ECC的测试集有至少 $\max_{i=1}^Q B_i$ 个值。对于TriTyp划分的块的最大个数是4, 因此ECC要求至少4个测试。

这个标准可以通过从表4.3中选择测试{(2, 2, 2), (1, 1, 1), (0, 0, 0), (-1, -1, -1)}来满足。并不需要太多思考我们就可以得出这些测试对于这个程序并不是十分有效的。ECC给测试人员在怎样组合测试值方面留有很大灵活性, 因此可以把它称做一个相对“弱”的标准。

ECC的弱点可以表述为不要求值与其他值组合。一个顺其自然的下一步是要求清楚值的组合, 称做成对组合 (pair-wise)。

标准4.25 成对覆盖 (PWC): 来自对于每个特征的每个块的一个值, 必须与来自针对其他特征的每个块的一个值相结合。

还是考虑前面3个划分的块的例子[A, B]、[1, 2, 3]和[x, y], 那么PWC需要16个测试来覆盖下面的组合:

(A, 1) (B, 1) (1, x)

(A, 2) (B, 2) (1, y)

(A, 3) (B, 3) (2, x)

(A, x) (B, x) (2, y)

(A, y) (B, y) (3, x)

(3, y)

PWC允许同样的测试用例覆盖多组唯一对值。所以上面的组合可以以以下几种方式组合：

(A, 1, x) (B, 1, y)

(A, 2, x) (B, 2, y)

(A, 3, x) (B, 3, y)

(A, -, y) (B, -, x)

带有“-”的测试表示可以使用任何块。

满足PWC的测试集将把每个值和其他值配对或至少有 $(\text{Max}_{i=1}^Q B_i)^2$ 个值。在TriTyp（见表4.3）中每个特征有4个块，所以需要至少16个测试。

满足PWC的若干算法已经发表，在本章的参考文献部分有适当提及。

对PWC的一个自然延伸是 t 值而不是对值。

标准4.26 T-Wise覆盖 (TWC)：来自对于 t 个特征的每个组的每个块的一个值必须被组合。

如果 T 的值表示划分的个数 Q ，那么TWC等于所有的组合。一个满足TWC的测试集至少有 $(\text{Max}_{i=1}^Q B_i)^t$ 个值。根据测试用例的个数来说，TWC的代价是比较高的，并且经验表明超出pair-wise（即 $t=2$ ）没有什么帮助。

PWC和TWC都“盲目地”不考虑和哪个值结合。下一个标准以一种不同的方式加强了ECC，这种方式是引入一个很小但十分重要的程序的领域知识片段，并查问对于每个划分“最重要的”块是哪个。这个块称为基本选择。

标准4.27 基本选择覆盖 (BCC)：一个基本选择块是针对每个特征选择的，并且一个基本测试是通过对每个特征使用基本选择形成的。非基本测试是通过如下方法选择：除了基本选择保持其他为常数并且在每个其他特征上使用每个非基本选择。

考虑前面3个划分的块的例子[A, B]、[1, 2, 3]和[x, y]，假设基本选择块是“A”、“1”和“x”。那么基本选择测试是(A, 1, x)，后面的要用到的附加测试有：

(B, 1, x)

(A, 2, x)

(A, 3, x)

(A, 1, y)

满足BCC的测试集将有一个基本测试，为每个划分的每个剩余模块加一个测试，因此，总数有 $1 + \sum_{i=1}^Q (B_i - 1)$ 。对于TriTyp每个特征有4个块，因此BCC要求1+3+3+3个测试。

基本选择可以是最简单的、最小的、某种顺序的第一个，或者从终端用户的角度看最可能的一个。组合多于一个无效值经常是没用的，因为软件经常识别一个值而其他值的副作用被掩盖。哪些块被选做基本选择成为测试设计中关键的一步，测试设计对测试结果有很大的影响。所以，测试人员记录所使用的策略是十分重要的，这样将来的测试可以重新评估那些决策。

对TriTyp根据选择最可能的块的策略，我们从表4.2中选择“大于1”作为基本选择块。使用选自表4.3的值给基本测试 (2, 2, 2)。剩余的测试通过依次改变其中的每个得到：{(2, 2, 1), (2, 2, 0), (2, 2, -1), (2, 1, 2), (2, 0, 2), (2, -1, 2), (1, 2, 2), (0, 2, 2), (-1, 2, 2)}。

有时测试人员会在选择单一的基本选择时遇到问题，并决定需要多个基本选择。这就有了如下标准：

标准4.28 多个基本选择标准 (MBCC)：至少一个和可能多个，基本选择块针对每个特征被选择，并且通过至少使用一次针对每个特征的基本选择来形成基本测试。随后的测试是通过如下方法选择：除了基本选择保持其他为常数并且在每个其他特征中使用每个非基本选择。

假定针对每个特征有 m_i 个基本选择，一共有 M 个基本测试，MBCC要求 $M + \sum_{i=1}^Q (M \times (B_i - m_i))$

个测试。

例如，我们可能为TriTyp中的边1选择包括两个基本选择，“大于1”和“等于1”。这会得到两个基本测试 (2, 2, 2) 和 (1, 2, 2)。上面的公式因此通过 $M=2$ 、 $m_1=2$ 和 $m_i=1 \forall i (1 < i \leq 3)$ 求值，即 $2 + (2 \times (4-2)) + (2 \times (4-1)) + (2 \times (4-1)) = 18$ 。余下的测试通过依次改变其中的每一个得到。MBCC标准有时会导致重复测试。例如，(0, 2, 2) 和 (-1, 2, 2) 对于TriTyp都出现了两次。当然，应该消除重复的测试（这也使得测试数目的计算式给出的是上界）。

图4.2展示了输入空间划分结合策略标准的包含关系。

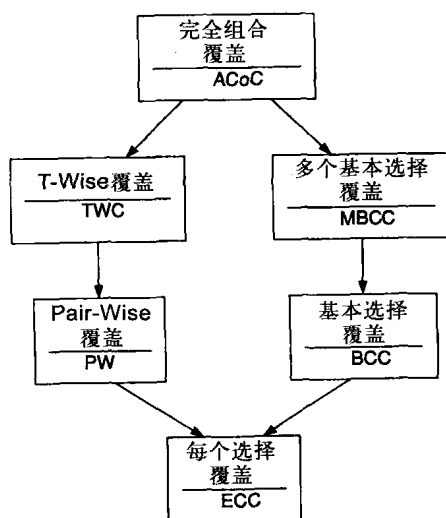


图4.2 输入空间划分标准的包含关系

4.2节练习

1. 为表4.2中的TriTyp输入的第二种分类列举出64种测试，所有这些测试要满足完全组合覆盖 (ACoC)。使用表4.3中的值。
2. 为表4.2中的TriTyp输入的第二种分类列举出16种测试，所有这些测试要满足成对覆盖 (PWC)。使用表4.3中的值。
3. 为表4.2中的TriTyp输入的第二种分类列举出16种测试，所有这些测试要满足多个基本选择覆盖 (MBCC)。使用表4.3中的值。
4. 对于下面的intersection()方法回答下列问题：

```
public Set intersection(Set s1, Set s2)
//效果：如果s1或者s2是空的，则抛出NullPointerException异常，
//否则返回一个等于集合s1和s2的交集的非空集合
```

特性：s1的类型

-s1 = null

-s1 = {}

-s1 至少有一个元素

特性：s1和s2之间的关系

-s1和s2代表了相同的集合

-s1是s2的子集

-s2是s1的子集

-s1和s2没有任何相同的元素

(a) “s1的类型”这个划分满足了完全性吗？如果没有，给s1赋一个值，这个值不属于任何块。

(b) “s1的类型”这个划分满足了分离性吗？如果没有，给s1赋一个值，这个值不只属于一个块。

(c) “s1和s2的关系”这个划分满足了完全性吗？如果没有，给s1和s2一对值，这一对值不适合在任何一个块里。

(d) “s1和s2的关系”这个划分满足了分离性吗？如果没有，给s1和s2一对值，这一对值不适合在一个块里。

(e) 如果“基本选择”标准被应用到两个划分里（正像所写的那样），会产生多少测试需求？

5. 用下面的签字为类BoundedQueue产生输入空间划分：

- public BoundedQueue(int capacity);
- public void Enqueue(Object X);
- public Object Dequeue();
- public boolean IsEmpty();
- public boolean IsFull();

假定一个队列的通常语义有一个确定的、最大的容量。尽力让你的划分简单，选择较少的分区和块。

(a) 确定所有的变量。不要忘记状态变量。

(b) 确定几个表明划分的特性。

(c) 为每个特性确定划分中的块。指定每个划分中的一个块作为“基本”块。

(d) 为这些块定义值。

(e) 定义一个满足基本选择覆盖（BCC）的测试集合。

6. 开发一组特性，同时为检查程序的模式附随一些划分（这个程序是第2章图2.21的方法pat()）。

(a) 开发测试来满足基本选择标准。你的测试应该既有输入又有期望的输出。

(b) 用在第2章开发的有关数据流测试集合分析你的测试。输入空间划分怎样做才会好？

4.3 划分中的约束

输入空间划分中一个微妙的地方是一些块的组合是不可行的，这必须在IDM中被记录。例如，表4.7展示了一个例子，该例子是基于之前描述的boolean findElement(list, element)方法。设计一个IDM有两个参数A（有4个划分）和B（有3个划分）。这两个划分组合并没有意义因此是无效的。在这个例子中，这些被表示为一列无效的参数划分对。一般情况下其他表示被使用，例如一组不等式。

约束是来自不同特征的块与块之间的关系。这里介绍两类约束。一类是来自一个特征的一个块不能和来自其他特征的一个块组合。“小于0”和“不等边三角形”就是这类约束的一个例子。另一类约束正好相反，来自一个特征的一个块必须和来自其他特征的特定块组合。尽管听上去比较简单，当选择值时识别和满足这些约束是困难的。

表4.7 无效块组合例子

特性	块			
	1	2	3	4
A: 长度与内容	一个元素	多于一个元素且无序	多于一个元素且有序	多于一个元素且元素都相同
B: 匹配	没发现元素	发现元素一次	发现元素多次	

注：无效组合(A1, B3), (A4, B2)。

当值的选择时如何处理约束依赖于选择的覆盖标准，并且这个决定通常是在选择值的时候做出。对于ACoC、PWC和TWC标准，唯一合理的选择是在考虑中去掉不可行的对。例如，如果PWC要求一个特定的对，而这个对是不可行的，那么在测试工程师的部分怎么做也不能使要求可行。然而，对于BCC标准情况就完全不一样了。如果一个特殊的变化（例如，“小于0”对于“边1对0的关系”）与基本情况（例如，“不等边三角形”对于“几何的分类”）冲突，那么显而易见需要做的是改变选择基本情况使得变化可行。在这个例子中，“几何的分类”很明显需要改成“无效”。

对于另外一个例子，考虑对一个数组排序。输入将是一个长度变化的任意类型数组。输出将有三部分：（1）输入数组的一个排序，以升序排列；（2）最大值（max）；（3）最小值（min）。我们可能考虑如下特性：

- 数组长度
- 元素类型
- 最大值
- 最小值
- 最大值的位置
- 最小值的位置

这些特征可以依次合理地产生划分，总结如下：

Length {0, 1, 2 ... 100, 101 ... MAXINT}

Type {int, char, string, other}

Max {≤0, 1, >1, 'a', 'Z', 'b', ..., 'Y', blank, nonblank}

Min {...}

Max pos {1, 2 ... Length-1, Length}

Min pos {1, 2 ... Length-1, Length}

有辨识力的读者一定注意到了并不是所有的组合都可行。例如，如果Length=0，那么其他的都没有意义了。还有，一些最大和最小值只在Type=int时才能得到，其他的在Type=char时有意义。

4.4 参考文献注释

在研究文献中，提到了一些测试方法，这些方法是基于这样一个模型的，该模型的特点是测试对象的输入空间应该被划分成子集，且假定同一子集中所有的输入引起相同的行为。这些全部称为划分测试（partition testing）并且包含等价划分[249]、边界值分析[249]、类别划分[283]和领域测试[29]。Grindal等人[143]发表了一个带有例子的广泛调查。

划分和值的出处来自Balcer、Hasling和Ostrand的1988年提出的类别划分方法[23, 283]。一个可视化的成果是在1993年由Grochtman、Grimm和Wegener提出的分类树[145, 146]。分类树把输入空间信息组织成一个树结构。第一层节点是参数和环境变量(特征), 它们可能被递归地分成子类别。块在树中以叶节点出现, 并且通过选择叶节点来选择组合。

Chen等人以经验为主地识别一般的错误, 这些错误一般是测试人员在输入参数建模[64]过程中出现的。本章中关于输入域建模中的很多概念来自于Grindal的博士论文[140, 142, 144]。Cohen等人[84]和Yin等人[363]都指出了针对输入参数建模的面向功能性的方法。面向功能性的输入参数建模也被Grindal等人[141]含蓄地使用过。其他两个输入域建模方法是分类树[145]和基于UML活动图的方法[65]。Beizer[29]、Malaiya[222]和Chen等人[64]也提出了特征选择的问题。

Grindal发表了一个不同约束处理机制的分析/经验比较[144]。

Stocks和Carrington[320]提供了基于规范的测试的正式概念, 这种测试包含了针对输入空间划分测试的大部分方法。特别地, 他们提出了对测试用例精化测试框架(本书中简单和非正式地称为测试需求)的问题。

每个选择和基本选择标准由Ammann和Offutt在1994[16]年提出。Cohen等人[84]指出有效的和无效的参数值应该根据覆盖区别对待。有效值属于测试对象的正常操作范围内, 无效值在正常操作范围外。无效值经常引起错误信息和执行中止。为了避免一个无效值掩盖另一个, Cohen等人建议每个测试用例中只含有一个有效值。

Burroughs等人[58]和Cohen等人[84, 85, 86]提出启发式的成对覆盖作为自动有效测试生成器(AETG)。AETG也包括变化基本选择组合标准。在AETG版本中, 称做“默认测试”, 测试人员一次改变一个特征的值而其他特征保持某个默认值。短语“特征测试”也被Burr和Young[57]使用, 这两个人介绍过另外一个基本选择的变化。在他们的版本中, 除一个特征外, 其他的特征都包含一个默认值, 并且剩下的特征包含最大或最小值。这个变量不是必须满足“每个选择覆盖”。

为产生测试用例的受约束数组测试系统(CATS)工具由Sherwood[313]介绍满足成对覆盖。对于有两个或多个特征的项目, in-parameter-order (IPO) [205, 206, 322]组合策略产生一个测试集, 该测试集对最前的两个参数(用我们的术语是特征)满足成对覆盖。接下来测试集被扩展为对最前面的3个参数满足成对覆盖, 并继续对附加每个参数直到所有的参数都被包括。

Williams和Probert发明了T-wise覆盖[354]。T-wise覆盖的一个特殊例子称做变量强度(variable strength), 变量强度是由Cohen、Gibbons、Mugridge和Colburn[87]提出的。这个策略要求特性的一个子集间高覆盖和跨特性间低覆盖。例如假设一个测试问题有4个参数A、B、C、D。变量强度要求对于参数B、C、D的3-wise覆盖和对于参数A的2-wise覆盖。Cohen、Gibbons、Mugridge和Colburn[87]提出使用模拟退火算法(SA)来对T-wise覆盖产生测试集。Shiba、Tsuchiya和Kikuno[314]提出使用一个基因算法(GA)来满足pair-wise覆盖。同一篇文章还提出了使用蚁群算法(ACA)。

Mandl提出使用正交数组为T-wise覆盖生成值[224]。这个思想被Williams和Probert进一步发展了[353]。覆盖数组[352]是正交数组的一个延伸。正交数组的一个特性是它们是平衡的, 意思就是每个特征值在测试集中出现同样的次数。如果只需要T-wise(例如pair-wise)覆盖,

那么平衡特性不是必需的,并且将使算法效率下降。在满足T-wise覆盖的一个覆盖数组中,每个T-tuple至少出现一次但并不必须是一样的次数。另一个与正交数组相关的问题是对于某些问题大小我们没有足够的正交数组来表示整个问题。这个问题也要通过使用覆盖数组来避免。

一些论文已经提供了使用输入空间划分的根据经验的和实验的结果。Heller[156]用一个实际的例子来展示测试所有特征值的组合在实际中是可行的。Heller得出的结论是我们需要确定大小可管理的组合的一个子集。

Kuhn和Reilly[195]研究了来自两个大型实际项目的365份错误报告并发现pair-wise覆盖在找错方面与测试所有组合一样有效。更多的支持数据由Kuhn和Wallace[196]给出。

Piwowski、Ohba和Caruso[291]描述了怎样成功地应用代码覆盖作为功能测试中的一个结束标准。作者把功能测试表达为从输入参数的所有值的组合中选择测试用例的问题。Burr和Young[57]展示了持续监控代码覆盖能帮助改善输入域建模。最初的实验展示了特殊测试导致了50%左右的覆盖,但通过持续使用代码覆盖和精化输入域模型,决策覆盖增加到了84%。

实际中大量应用输入空间划分的例子已经发表。Dalal、Jain、Karunanithi、Leaton、Lott、Patton和Horowitz[91, 92]通过使用AETG工具报告结果。它被用来为Bellcore的智能服务控制点(Intelligent Service Control Point)产生测试用例,智能服务控制点是一个用来给技术员分配工作要求的基于规则系统,是一个在大型应用中的一个GUI窗口。先前,Cohen、Dalal、Kajla和Patton[85]证明了AETG用于屏测试,这是通过穿过大量的screen为一致性和有效性测试输入域。

Burr和Young[57]还用AETG工具测试了一个Nortel的应用,该应用从一种格式到另一种传递email信息。Huller[171]使用一个与IPO有关的算法为卫星通信测试地面系统。

Williams和Probert[353]展示了输入空间划分怎样用于组织配置测试。Yilmaz、Cohen和Porter[362]使用覆盖数组作为复杂配置空间中缺陷定位的起始点。

Huller[171]展示了pair-wise配置测试,与quasi-exhaustive测试相比可以节省60%以上的成本和时间。Brownlie、Prowse和Phadke[50]比较了两个结果,一个结果是使用正交数组(OA)在一个PMX/StarMAIL发布的一个版本,另一结果是来自对一个较早的版本的传统测试。作者估计如果OA被用于第一个版本,将有22%以上的错误被发现。

一些研究比较了产生的测试数。当使用不确定算法时测试的数目是不同的。一些论文比较了满足2-wise或3-wise的输入空间划分策略:IPO和AETG[205],OA和AETG[141],覆盖数组(CA)和IPO[352]和AETG、IPO、SA、GA和ACA[87, 314]。其中的大部分只有很少的差别。

另一个比较算法的方法是考虑执行时间。Lei和Tai[206]展示了IPO的时间复杂度优于AETG的时间复杂度。Williams[352]报告说在他的研究中,对于最大型的测试问题,CA优于IPO近3个数量级。

Grindal等人[141]通过所找到的错误数量比较算法,他们发现尽管BCC用较少的测试用例,但其表现与AETG和OA一样。

输入空间划分策略也可以在基于代码覆盖的基础上被比较。Cohen等人[86]发现由AETG产生的测试集对2-wise覆盖达到超过90%的块覆盖。Burr和Young[57]对于AETG得到近似的结果,使用47个测试用例达到93%的块覆盖,与之相对比的是,对于BCC的一个受限制的版本使用72个测试用例达到85%的块覆盖。

第5章 基于句法的测试

在前面的章节里，我们学习了如何从图、逻辑表达式以及输入空间划分来生成测试。测试覆盖标准的第4种主要来源是软件工件的句法描述。与图以及逻辑表达式一样，基于句法的测试也有几种工件可以利用，包括源代码和输入需求。）

基于句法的测试的最基本的特征是需要用到一种句法的描述，比如一个语法或者BNF。第2章和第3章讨论了如何从软件工件（比如程序、设计描述和规约）来建立图的模型和逻辑模型。第4章讨论了如何从一个输入空间的描述来构建一个输入的模型，然后把测试标准应用于模型。在基于句法的测试中，软件工件的句法用来作为模型，测试数据也是根据这些句法来生成的。

5.1 基于句法的覆盖标准

句法结构能够以不同的方式在软件测试中运用。我们能够用句法来生成有效（正确的句法）的或者无效（错误的句法）的产品。有时候我们生成的结构就是测试用例自身，另外一些时候生成的结构能够帮助我们发现测试用例。我们会在后面的部分来探讨它们之间的不同。和前面一样，我们先定义句法结构的一般标准，然后再把它们应用于特定的工件。

5.1.1 BNF覆盖标准

在软件工程领域中，借用自动机理论中的结构来描述软件产品的句法的现象非常普遍。编程语言就是表达成BNF语法格式，程序的行为用有限状态机描述，程序的允许输入也是用文法来定义的。正则表达式和上下文无关的文法在这里特别有用。考虑如下的正则表达式：

$$(Gsn|Bt n)^*$$

在上面的表达式中，星号是一个包含符，它表示它所修饰的表达式能够出现0次或者多次；竖线是一个选择符，它表示可以选择两个表达式中的任意一个。因此，这个正则表达式描述任何的“G s n”和“B t n”的序列。G和B可以是一个程序中的命令，s、t和n可以是一些参数，或者带参数的方法调用，还可能是带值的消息。参数s、t和n可以是字母或者表示值的一个大集合（比如数字或字符串）。

一个测试用例可以是一个满足这个正则表达式的字符串的序列。比如，假定参数是表示数字，那么下面的数据可能表示一个测试的4个部分，两个、3个或者4个独立的测试：

```
G 17 08.01.90
B 13 06.27.94
G 13 11.21.94
B 04 01.09.03
```

尽管正则表达式在某些情况下已经足够，但是我们常用的是一个更加复杂的文法。上面那个例子可以简化成如下表示的文法：

```

stream ::= action*
action ::= actG | actB
actG   ::= "G" s n
actB   ::= "B" t n
s      ::= digit1-3
t      ::= digit1-3
n      ::= digit2 "." digit2 "." digit2
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

```

一个文法有一个特殊的符号叫做开始符号，在这里，这个开始符号是stream。文法中的符号分为两种：终结符和非终结符。非终结符表示它后来必须被替换，终结符表示它不需要被替换。在这个例子中，在 ::= 号左边的符号都是非终结符，而在引号中的符号都是终结符。对于一个给定的非终结符的任何可能的重写叫做一个产生式或规则。在这个语法中，标记在右上角的星号表示重复0次或者多次，加号表示重复一次或者多次，数字的表示需要重复指定的次数，数字的范围 ($a \sim b$) 表示它重复至少 a 次，不超过 b 次。

语法系统有两个用途。首先，它可以用做识别器（在第1章中有介绍），它用来决定一个给定的字符串（或测试用例）是否符合一个语法。这是经典的自动机理论问题中的解析，自动化工具（比如资深的lex和yacc）使得识别器的构建非常容易。识别器在测试中十分有用，因为他们使得能够判定一个给定测试用例是否符合一个特定的语法。语法系统的另外一个用处就是用来构建生成器，在第1章中我们也介绍过了。生成器可以从语法的起始符来产生一个终结符的字符串。在这种情况下，这个字符串是测试的输入，比如，下面的推导生成了上面的一个测试用例。

```

stream → action*
       → action action*
       → actG action*
       → G s n action*
       → G digit(1-3) digit2 . digit2 . digit2 action*
       → G digitdigit digitdigit.digitdigit.digitdigit action*
       → G 14 08.01.90 action*
       :

```

这个推导的过程就是系统地把下一个非终结符（在这里是“action^{*}”）换成它的一个产生式。推导继续进行直到所有非终结符都被替换，并且只剩下了终结符。测试的关键就是应该选用哪一个推导，这也是怎么定义有关语法标准的关键。

尽管可以定义许多测试标准，最普通和直接的是终结符覆盖以及产生式覆盖。

标准5.29 终结符覆盖 (TSC): TR 包含了语法 G 中的任何符号 t 。

标准5.30 产生式覆盖 (PDC): TR 包含了语法 G 中的每个产生式 p 。

到目前为止，我们可以很容易地意识到PDC包含TSC（如果我们覆盖了每个产生式，那么我们就覆盖了任何一个终结符）。有些读者可能注意到语法和图有一些自然的联系。所以，TSC和PDC也能够被等效地重写成图上的节点覆盖和边覆盖。当然，这意味着其他的基于图的覆盖标准也能够被定义成语法。据我们所知，既没有研究人员，也没有实践人员进行了这方面的研究与探索。

这里定义了唯一的与其他相关的标准有点不切实际，它就是在在一个图中推导出所有可能的字符串。

标准5.31 推导覆盖 (DC): *TR* 包含了任何可能的从语法 *G* 中能够推导出来的字符串。

由TSC生成的测试的数目会受终结符数目的限制。上面的stream BNF包括13个终结符: *G*、*B*、*.*、0、1、2、3、4、5、6、7、8、9。它有18个产生式(注意“*l*”符号增加了产生式,所以“*action*”有两个产生式,“*digit*”有10个产生式)。从DC衍生的测试用例的数目决定于语法的细节,但是总体来说可以是无限的。如果我们忽略stream BNF的第一个产生式,我们能够有一个有限数目的推导字符串。两个可能的action是(*G*和*B*), *s*和*t*两者都有一个不超过3位的数字,每个数字都有10种选择,所以有1000种可能。非终结符*n*有3个两位数的集合,每位数中的数字都有10种选择,所以有 10^6 种可能。所有stream 语法能够产生出 $2 \times 1000 \times 10^6 = 2\,000\,000\,000$ 个字符串。DC是一种基于理论但不切实际的语法覆盖。(如果有工具推销人员或者应聘人员会宣称他们做了所有的字符串覆盖或者路径覆盖,我们应该保持警惕)。

TSC、PDC和DC生成的测试用例是语法定义的字符串集合的成员。有时候,生成不在语法中的测试用例也是很有用的,这一点将在后面的小节的标准中介绍。

5.1.1 节练习

1. 思考一下节点和边的覆盖为什么经常出现在软件测试领域中。写一篇短文来解释。
2. 对于一张图,存在一个语法能够用来从这张图生成无限多的测试用例。这是如何做到的,是什么使得这个语法存在?

5.1.2 变异测试

语法中一个有趣的事情就是用语法来描述一个输入是不合格。我们说一个输入有效仅当这个输入包含在语法所描述的语言中,否则我们就说输入无效。比如,我们经常见到的是需求一个程序拒绝一个非法的输入,并且这个特性应该被清晰地测试,因为程序员很容易忘记或者把它搞错。

所以,语法被用来产生无效的字符串通常很有用。这对于在测试中用有效的字符串连接一个从一个存在的字符串推导出来的变体字符串很有用。这两种字符串都叫做变体^①。这能够通过变异语法来产生,然后用变异的语法来生成字符串,或者在一个产生的过程中变异字符串的值。

从下面的小节中我们会看到变异能够被应用于很多工件中。当然,它主要还是被用于基于程序的测试方法中,大部分理论和大部分详细的概念都是特定的基于程序的变异。所以,在5.2.2节中将会有很多细节出现。

变异通常是基于一个“变异操作符”的集合,它们都与一个“基础”字符串相联系。

定义5.45 基础字符串: 语法中的一个字符串。

定义5.46 变异操作符: 从语法生成的字符串规定语法变异的规则。

定义5.47 变体: 运用一次变异操作符的结果。

变异操作符一般都要用于基础字符串,但是它们同样能够被应用于语法,或者动态地被

① 变异的使用和遗传算法之间没有什么联系,除了两者都是用生物变异做比喻。用来进行测试分析的变异在遗传算法诞生之前就已经存在好几年了。

应用于一个推导过程。变异操作符这个术语十分宽泛，所以设计合适的变异操作符是对任何工件进行变异的关键部分。一个好的设计的操作符集能够产生十分强大的测试，但是一个设计得差的操作符集就会导致无效的测试。比如，一个商业的工具“实现变异”，但是它也只把判定转换成真和假，那么这就只是一个非常昂贵的实现分支覆盖的方法。

有时我们脑海中有一个特定的基础字符串，但是有时这个字符串只是一个不应用任何操作符的隐式结果。比如，在把变异应用于程序语句中，我们关心的是那个基础字符串。基础字符串是被测程序中的语句的序列，而变体就是对于那个程序的一些简单的句法的变异。我们在非法输入的测试中不关心基础字符串，因为那时候我们只是去看程序是否在一个非法输入的时候运行正常。基础字符串是有效输入，而变体是无效输入。比如，一个有效的输入可能是一个正确登录的用户的交易请求，无效的版本可能是来自一个没有正确登录的用户的相同的交易请求。

考虑一下在5.1.1节中的语法。如果第一个字符串 (G 17 08.01.90) 是一个基础字符串，那么两个有效的输入可能是：

```
B 17 08.01.90
G 43 08.01.90
```

两个无效的输入可能是：

```
12 17 08.01.90
G 23 08.01
```

当基础字符串无关紧要时，用前面介绍的生成器方法，变体能够直接通过在推导的过程中改变产生式来生成。也就是说，如果不需要基础字符串的话，它们就不需要显式地生成出来。

当应用变异操作符的时候，通常会遇到两个问题。首先，创建变体的时候是否应该使用一个以上的变异操作符同时应用于同一个基础字符串来生成变体？也就是说，一个变体的字符串应该包含一个还是多个变异的元素？常识给出的答案是不，并且基于程序的变异、可靠的实验和理论的证据已发现每次只变异一个元素。另外一个问题是否要考虑对于一个基础字符串应用一个变异操作符的所有可能的情况？通常在基于程序的变异中是这么做的。一个理论的原因是基于程序的变异包含一些其他的测试标准，所以如果操作符没有被应用完全，那么那个测试标准就不会被覆盖到。尽管如此，当与基础字符串无关的时候我们通常不会这样做，比如在无效输入的测试中。这个问题在后面的章节中有更加详细的探讨。

变异操作符在许多编程语言中都有实现，比如正式规约语言、BNF语法和至少一种数据定义语言 (XML)。对于一个给定的工件，变体的集合是 M 和每个变体 $m \in M$ 都会生成一个测试需求。

当一个推导变异出有效的字符串，那么测试的目标就是通过造成变体来生成不一样的输出来杀死这些变体。更正式地，对于一个推导 D 和一个测试 t ，给定变体 $m \in M$ ，我们说当且仅当 t 在 D 上面的输出与 t 在 m 上面的输出不同时， t 能够“杀死” m 。推导 D 可能由完全生成式表示，或者它可能简单地由最终字符串表示。比如，在5.2.2节中，字符串是程序或者程序的组件。覆盖用杀死的变体来定义。

标准5.32 变异覆盖 (MC)：对于每个变体 $m \in M$ ， TR 只包含一个需求能够杀死变体 m 。

所以, 变异的覆盖等同于杀死变体。覆盖的数量一般表示成杀死的变体的百分比, 叫做变异得分。

当一个语法用来变异生成非法字符串的时候, 测试的目的就是为了运行变体来看程序的行为是否正确。这个覆盖策略更加简单, 因为变异操作符是测试的需求。

标准5.33 变异操作符覆盖 (MOC): 对于每个变异操作符, *TR*要只包含一个需求, 就是创建一个变异的字符串 m , 它是从变异操作符产生的。

标准5.34 变异产生式覆盖 (MPC): 对于每个变异操作符和这个变异操作符能够应用的每个产生式, *TR*包含一个从产生式生成变异字符串的需求。

对于变异的测试需求的数目有时很难来定量, 因为它取决于工件以及变异操作符的语法。在大多数情况下, 变异比任何其他测试标准产生出更多的测试需求。接下来的章节有更多的数据来定量特定的一些变异操作符, 同时也会有更多的与参考文献注释有关的细节。

变异测试同样也不容易手动来应用, 它的自动化也比大多数的测试标准更加复杂。所以, 变异被普遍认为是一个高端的测试标准, 它更加有效但同时也更加昂贵。变异测试一个比较常见的应用是在实验学习中以所谓的“金牌标准”的地位来与其他测试标准进行比较的。

5.1.2节练习

1. 定义变异得分。
2. 变异得分与第1章中的覆盖是如何联系起来的?
3. 考虑一下5.1.1节中的stream BNF, 基础字符串“B 10 06.27.94”, 分别给出3个有效的和3个无效的字符串变体。

本章接下来的部分将会探讨各种形式的BNF和变异测试。下面的表总结了不同种类的语法测试的章节以及它们的特点。在BNF和变异测试中, 使用语法测试是用来生成有效还是无效的测试都被关注。对于变异测试, 我们同样还要关注是否使用了一个基础字符串, 变体是否被测试, 变体是否被杀死。

	基于程序的测试	集成测试	基于规约的测试	输入空间测试
BNF	5.2.1节	5.3.1节	5.4.1节	5.5.1节
语法	编程语言	没有已知的应用	代数规约	输入语言, 包括XML
总结	编译器测试			输入空间测试
变异	5.2.2节	5.3.2节	5.4.2节	5.5.2节
语法	编程语言	编程语言	FSM	输入语言, 包括XML
总结	变体程序	测试集成	使用模型检测	错误检测
基础字符串?	是	是	是	否
有效?	是, 必须编译	是, 必须编译	是	否
测试?	变体不是测试	变体不是测试	变体不是测试	变体是测试
杀死?	是	是	是	没有杀死变体的意图
注意	变体有强弱之分, 包含许多其他的技术	包括面向对象的测试	相等变体的自动检测	有时候先变异语法, 然后再产生字符串

5.2 基于程序的语法

与大部分标准一样，基于语法的测试标准应用于程序比应用于其他工件多。BNF覆盖标准已经被用来生成程序来测试编译器。变异测试也被应用于方法（单元测试）和类（集成测试）。对于类的应用会在下一节介绍。

5.2.1 编程语言的BNF语法

BNF测试语言的主要目的是为编译器生成测试套件。因为这是十分特殊的应用，所以在本书中我们就不详细介绍了。在参考文献注释部分，我们指出了相关的参考文献，有些甚至比较陈旧。

5.2.2 基于程序的变异

变异原来是为程序而开发的，本节的介绍将明显比本章的其他节深入。基于程序的变异运用操作符，它是定义一个特定编程语言中的语法。我们将从一个基础字符串开始，它是一个被测试的程序。然后我们应用变异操作符来创建变体。这些变体应该是可编译的，所以基于程序的变异创建有效的字符串。变体不是测试，而是用来帮助我们发现测试。

给定一个基础字符串程序或者方法，一个适于变异的测试集合把程序与那个程序的语法变种的集合或变体区别开来。对于程序的变异操作符的一个简单的例子是算术操作符变异操作符，它把一个像“ $x=a+b$ ”的赋值语句变成许多可能的变体，包括“ $x=a-b$ ”、“ $x=a*b$ ”和“ $x=a/b$ ”。除非赋值语句出现在一个非常奇怪的程序中，它可能关系到用到哪个算术操作符，一个有分寸的测试集应该能够区分不同的可能性。事实证明通过谨慎地选择变异操作符，测试人员能够开发非常有效的测试集。

变异测试用来帮助用户来反复地增强测试数据的质量。测试数据被用来评价基础程序，目的是为了使得每个变体都能够展现出不同的行为。在这种情况下，这个变体就被认为是死掉了，并且不需要把它留在测试过程中，因为这个变体代表的错误会被杀死它的相同的测试探测到。更重要的是，这个变体满足了识别一个有用的测试用例的需求。

成功地运用变异测试的一个关键就是变异操作符，它是为每个程序、规约或设计语言而设计的。在基于程序的变异测试中，无效的字符串在语法上是非法的，并且会被编译器捕获。这些叫做死产的变异，它们不应该被生成。一个极其简单的变体会被所有的测试用例杀死。一些变体在功能上等同于原始的程序，也就是说，它们总是与原始程序产生相同的输出，所以没有测试用例能够杀死它们。等效变体代表不可行的测试需求，我们在前面的章节中讨论过。

我们提炼一下基于程序的变异中的术语杀死和覆盖。它们的定义与前一节中的是一致的。

定义5.48 杀死变体：对于基础字符串程序 P 和测试 t ，给定一个变体 $m \in M$ ，当且仅当 t 在程序 P 上的输出与 t 在 m 面的输出不同时，我们就说 t 杀死了 m 。

在5.1.2节中我们说过，很难去定量变异的测试需求的数量。其实，这个数量取决于所使用的特定的操作符的集合，以及应用这些操作符的语言。一个广泛使用的变异系统是Mothra，它为图5.1中的Fortran版本的Min()方法生成了44个变体。对于大多数操作符的集合，基于程序的变体的数目大体上与变量引用的数目与声明的变量数目的乘积成正比（ $O(Refs * Vars)$ ）。在下面的“设计变异操作符”部分提到可选择的变异方式，减少数据对象的数目使得变体的

数目与变量引用的数目成正比 ($O(\text{Refs})$)。更多的细节见参考文献注释。

原始的方法		内置变量的方法
<pre>int Min (int A, int B) { int minVal; minVal = A; if (B < A) { minVal = B; } return (minVal); } //结束 Min</pre>		<pre>int Min (int A, int B) { int minVal; minVal = A; minVal = B; if (B < A) if (B > A) if (B < minVal) { minVal = B; Bomb(); minVal = A; minVal = failOnZero (B); } return (minVal); } //结束 Min</pre>
	Δ1 Δ2 Δ3 Δ4 Δ5 Δ6	

图5.1 Min方法和6个变体

在单元级别的测试中，基于程序的变异通常被应用于单个语句。图5.1包含了一个小的含有6个变体行（每个变体行被 Δ 标记）的Java方法。注意，每个被变异的语句代表一个单独的变异操作符。变异操作符被定义来满足两个目标中的一个。其中一个目标是模仿典型的程序员错误，所以试图保证测试能够发现这些错误，另外一个目标就是迫使测试者来生成测试，这些测试已经被发现能够有效地测试程序。在图5.1中，变体1、3和5用一个变量引用来替代另外一个引用，变体2改变了一个关系操作符，变体4是一个特殊的变异操作符，当运行到该语句时，它能够引起一个运行时的错误。这使得每个语句都被执行到，这样就满足了语句或者节点覆盖。

变体6看起来很不平常，因为这个变体操作符迫使测试者新建一个更加有效的测试。方法 *failOnZero()* 是一个特殊的变异操作符，当参数是0的时候它能够使得程序失败并且如果参数非0（它返回参数的值）将不干任何事情。所以，仅当 *B* 是0的时候变异6能够被杀死，这就迫使测试人员遵循久经考验的启发式的方法来使得每个变量和表达式设置成0。

有时候会使人对变异感到迷惑，那就是测试是如何生成的。当应用基于程序的变异的时候，测试人员最直接的目标就是杀死变体，非直接的目标就是创建好的测试。更加间接一点，测试者想要找到缺陷。杀死变体的测试能够通过直觉找到，或者更严格地说，通过分析变体会被杀死的条件。

RIP错误/缺陷模型在1.2节中讨论过。基于程序的变异用一个变体来代表程序的失败，而可达性、感染性和传播性指到达这个变体，变体使得程序的状态不正确，而且程序的最终输出也不正确。

弱变异放松了杀死变体的定义，使得它只包括可达性与感染性，而不包括传播性。也就是说，弱变异在变异了的部件执行后（也就是说，在表达式，语句或者基本块后）立即检查程序的内部状态。如果状态不正确，那么这个变体就被杀死了。这比标准（强）变异弱因为一个非正确的状态不是经常传播到输出。也就是说，强变异可能比弱变异要求更多的测试来满足覆盖。实验数据表明，大多数情况下它们俩之间的差别非常小。

它们之间的差别能够通过提炼前面给定的杀死变体的定义来形式化地表示出来。

定义5.49 强杀死变体：对于程序 *P* 和测试 *t* 给定一个变体 $m \in M$ ，当且仅当 *t* 运行在 *P* 上与 *t* 运行在 *m* 上的输出结果不同，我们说 *t* 强杀死 *m*。

标准5.35 强变异覆盖 (SMC): 对于每个 $m \in M$, TR 仅包含一个需求, 就是强杀死 m 。

定义5.50 弱杀死变体: 对于程序 P 和测试 t 给定一个变体 $m \in M$, m 修改了程序 P 中的区域 l , 当且仅当用 t 运行 P 状态和在 l 后运行 m 不一样, 我们说 t 弱杀死 m 。

标准5.36 弱变异覆盖 (WMC): 对于每个 $m \in M$, TR 仅包含一个需求, 就是弱杀死 m 。

考虑在图5.1中的变体1。这个变体是第一个语句, 所以可达性的条件就总是满足的 ($true$)。为了满足影响, B 的值必须与 A 的值不同, 形式化表示为 $(A \neq B)$ 。为了满足传播, Min 方法的变异了的版本必须返回一个不正确的值。在这种情况下, Min 必须返回赋给第一个语句的值, 它表示 if 块中的语句必须没有被运行过。也就是说 $(B < A) = false$ 。完整的能够杀死变体1的测试规范如下:

可达性: $true$

影响: $A \neq B$

传播: $(B < A) = false$

完全测试规范: $true \wedge (A \neq B) \wedge ((B < A) = false)$

$$\equiv (A \neq B) \wedge (B \geq A)$$

$$\equiv (B > A)$$

所以, 测试用例的值 ($A=5, B=7$) 会导致变体1失败。原始的方法会返回值5 (A), 但是被变异了的版本返回7。

变体3是相等性变体的一个例子。直觉上的 $minVal$ 和 A 在程序中的那个点会有相同的值。所以如果用其中的一个代替另外一个都没有什么效果。像在变体1中, 可达性的条件为 $true$ 。影响的条件是 $(B < A) \neq (B < minVal)$ 。但是, 忠实的分析能够揭露断言 ($minVal=A$), 从而得到组合的条件 $((B < A) \neq (B < minVal)) \wedge (minVal=A)$ 。通过消除不等号 ($\neq$) 我们可以简化成

$$(((B < A) \wedge (B \geq minVal) \vee ((B \geq A) \wedge (B < minVal))) \wedge (minVal=A))$$

重新整理一下得到

$$(((A > B) \wedge (B \geq minVal) \vee ((A \leq B) \wedge (B < minVal))) \wedge (minVal=A))$$

如果 $(A > B)$ 并且 $(B \geq minVal)$, 通过传递性 ($A > minVal$)。应用传递性到前两个析取式得到

$$((A > minVal) \vee (A < minVal) \wedge (minVal=A))$$

最后, 第一个非连接式能够被化简成一个不等式, 得到如下的矛盾:

$$(A \neq minVal) \wedge (minVal=A)$$

这个矛盾表示不存在能够满足这个条件的值, 所以这个变体就被证明是等价的。通常来说, 发现相等的变体, 就像发现不可能的路径一样, 是一个不可判断的问题。当然, 像代数操作和程序分片一样的策略能够发现一些相等的变体。

作为最后一个例子, 思考一下如下的方法, 其中有一个变体嵌入在语句4中:

```

1  boolean isEven (int X)
2  {
3      if (X < 0)
4          X = 0 - X;
```

```

Δ4      X = 0;
5      if (float) (X/2) == ((float) X) / 2.0
6          return (true);
7      else
8          return (false);
9  }

```

变体Δ4的可达性条件是 $(X < 0)$ ，影响条件是 $(X \neq 0)$ 。如果给出测试用例 $X = -6$ ，那么在执行完语句4之后 X 的值是6并且在变异后的版本的语句执行完语句4之后 X 的值是0。所以，这个测试满足了可达性与影响，这个变异在弱变异标准下会被杀死。然而，6和0都是偶数，所以从语句5开始决策对于变异和没变异的两个版本都会返回真。也就是说，传播不满足，所以测试用例里 $x = -6$ 在强变异标准下不会杀死变异。这个变体的传播条件是这个数得是奇数。所以，为了满足强变异标准，我们要求 $(X < 0) \wedge ((X \neq 0) \wedge (odd(X)))$ ，它能够简化成 X 是一个奇的负整数。

使用变异测试程序

一个测试过程有一系列的步骤来产生测试用例。一个测试标准可能用到很多测试过程，而一个测试过程里甚至可能不包含任何一种测试标准。为变异选择测试过程相当困难，这是因为变异分析实际上是度量测试用例质量的一种方法，而实际的软件测试是副产品。从实际意义上来讲，要么软件被测试，并且测试的效果很好，要么测试用例不能杀死那些变体。这一点可以通过一个典型的变异分析过程上来很好地理解。

图5.2显示的是如何应用变异测试。测试人员提交被测的程序到一个自动化的系统，这个自动化测试的系统首先产生一些变体。这些变体然后随机地被启发式系统进行分析以检测和排除掉尽可能多的等价的变体^①。然后自动生成一组测试用例，并且这组测试用例在原始的程序上执行，随后又在这些变体上执行。如果一个变体程序的输出不同于原始程序（正确的）的输出，那么这个变体就被标记为已经死掉的，并且认为能被该测试用例强杀死。已经死掉的变体不会在剩下的一系列的测试用例上执行。那些没有强杀死至少一个变体的测试用例被认为是无效的并且被排除掉，即使这些测试用例可能弱杀死一个或者更多的变体。这是因为上面所表述的要求需要程序的输出（不是内部状态）不同。

一旦所有的测试用例被执行，覆盖以变异得分的方式被计算。变异得分是已经死掉的变体的数量占总的非等价的变体数量的比例。如果变异得分是1.00，那就意味着所有的变体都被检测出来。杀死所有变体的测试集就可以认为相对于这些变体是充分的。

通常1.00的变异得分是不太实际的，所以测试人员定义了一个阈值，它的大小为可接受的最小变异得分。如果这个阈值没有达到，那么测试过程被重新执行，每次的执行过程中产生针对还存在的变体的测试用例。这个过程一直持续到达到变异得分的阈值。一直到这个环节，整个测试过程都是完全自动化的。为了完成测试，测试人员会检查有效测试用例的期望输出，并且会改正程序中出现的错误。从这些论述上就导出了变异测试的基础前提：在实际中，如果软件包含错误，那么就会有一组变体，这组变体仅仅能被检测出该错误的一个测试用例所杀死。

设计变异操作符

我们必须为每种语言选择出相应的变异操作符，尽管这些语言的操作符有很多是重复的。

① 当然，因为变体检测是不可判定的，一个启发式的算法或许是最佳的选择。

但是有一些不同点是特定于语言的，经常是和语言的一些特性相关。研究人员设计了一些不同语言的操作符，包括Fortran IV、COBOL、Fortran 77、C、C集成测试、Lisp、Ada、Java和Java类关系。研究人员还设计了正则规范语言SMV（将在5.4.2节讨论）和XML消息的变异操作符（将在5.5.2节讨论）。

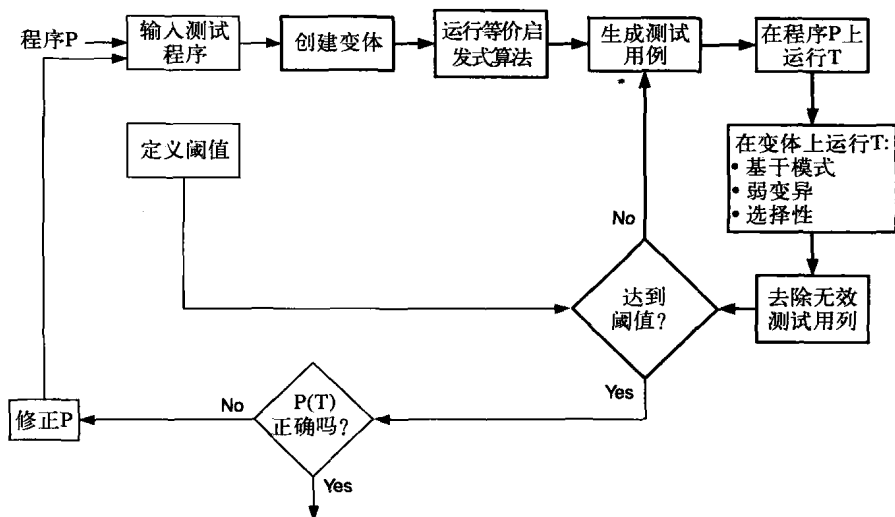


图5.2 变异测试过程

注：加粗的矩形框代表自动化的步骤；其他的矩形框代表手动的步骤。

作为一个研究领域，这些年我们已经学习了很多有关设计变异操作符的知识。一些文献中给出不同语言的变异操作符的详细列表，这些文献在这章的参考文献注释里被引用。变异操作符用来模仿一些程序员所犯的典型错误，或者鼓励测试人员遵循普遍的启发式测试。那些改变关系或者变量引用的操作符就是模仿程序员所犯典型的错误的操作符的例子。在图5.1中使用的failOnZero()操作符是后者的一个例子，测试人员被鼓励去遵循普遍的启发式测试“让每个表达式的值变为0”。

当首先为一种新的语言设计变异操作符的时候，做到“包含性的”是很合理的，也就是说要包含尽可能多的操作符。然而，这通常会导致产生大量的变异操作符，并且有更多数量的变体。研究人员付出了很多努力去寻找一些使用更少的变体和变异操作符的方式。使用更少变体的最普遍的两种方式是：（1）从总的变体集合中随机取出一个样本，（2）使用特别有效的变异操作符。

选择性变异是用来描述使用特别有效的变异操作符的策略。有效性是通过下面的方式进行度量的：如果特定地生成的测试用例能杀死由变异操作符 o_i 产生的变体，同样也能杀死由变异操作符 o_j 所产生的变体，那么变异操作符 o_i 就比变异操作符 o_j 更有效。

这个概念可以扩展成一组有效变异操作符集合：

定义5.51 有效变异操作符：如果特定地生成的测试能杀死由变异操作符 $O=\{o_1, o_2, o_3, \dots\}$ 产生的变体，同样也能以很高的概率杀死其余的变异操作符所产生的变体，那么变异操作符集 O 就定义是有效的。

研究人员总结出：那些插入一元操作符或者修改一元和二元操作符的变异操作符是有效

的。实际的实验在Fortran 77上做过 (Mothra系统), 但是实验的结果也同样适用于本章的Java。相应的一些操作符也能够定义成其他语言的操作符。下面定义的一些变异操作符作为在程序级别上的变异操作符集将在本章的剩下部分使用。

1. ABS——绝对值插入:

每个数学表达式 (和子表达式) 被下面的这些函数修改abs()、negAbs()和failOnZero()。

abs() 返回的是表达式的绝对值, negAbs()返回的是绝对值的相反值。failOnZero()检测表达式的值是否是0。如果是, 那么变体就被杀死, 否则程序继续执行并且表达式的值被返回。这个变异操作符强迫测试人员让每个数字表达式的值变为0、负值和正值。例如, 语句“x=3*a;”被变异后产生了下列3个语句:

```
x = 3 * abs (a);
x = 3 * - abs (a);
x = 3 * failOnZero (a);
```

2. AOR——数学操作符替换:

每一个数学操作符+、-、*、/、**和%的出现都被其他每一个操作符替换。除此之外, 每一个都被替换成特殊的变异操作符leftOp和rightOp。

leftOp返回的是左操作数 (右操作数被忽略), rightOp返回的是右操作数, %计算左操作数除以右操作数得出的余数。例如, 语句“x=a+b;”变异后产生了下面7个语句:

```
x = a - b;
x = a * b;
x = a / b;
x = a ** b;
x = a;
x = b;
x = a % b;
```

3. ROR——关系操作符替换:

每一个关系操作符 (<、≤、>、≥、==、≠) 的出现被其他每一个操作符和falseOp和trueOp替换。

falseOp总是返回false, trueOp总是返回true。例如, 语句“if(m>n)”变异后产生了下面的7个语句:

```
if (m >= n)
if (m < n)
if (m <= n)
if (m == n)
if (m != n)
if (false)
if (true)
```

4. COR——条件操作符替换:

每一个逻辑操作符 (和 (&&)、或 (||)、不带条件判断的和 (&)、不带条件判断的或 (!)、不等 (^)) 的出现被其他每一个操作符替换。除此之外, 每一个操作符都被falseOp、trueOp、leftOp和rightOp替换。

leftOp返回的是左操作数（右操作数被忽略），rightOp返回的是右操作数。falseOp总是返回false，trueOp总是返回true。例如，语句“if(a&&b)”变异后产生了下面的8个语句：

```
if (a || b)
if (a & b)
if (a | b)
if (a ^ b)
if (false)
if (true)
if (a)
if (b)
```

5. SOR——移位替换：

移位操作符（<<、>>和>>>）的每一个出现被其他每一个操作符替换。除此之外，每一个特殊操作符都被leftOp替换。

leftOp返回的是没有移位的左操作数。例如，语句“x=m<<a;”变异后产生了下面的3个语句：

```
x = m >> a;
x = m >>> a;
x = m;
```

6. LOR——逻辑操作符替换：

每个位逻辑操作符（按位与(&)、按位或(|)和按位异或(^)）的出现被其他每一个操作符替换。除此之外，每一个操作符都被leftOp和rightOp替换。

leftOp返回的是左操作数（右操作数被忽略），rightOp返回的是右操作数。例如，语句“x=m&n”变异后产生了下面的4个语句：

```
x = m | n;
x = m ^ n;
x = m;
x = n;
```

7. ASR——赋值操作符替换：

每个赋值操作符（+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=、>>>=）的出现被其他每一个操作符替换。

例如，语句“x+=3”变异后产生了下面的10个语句：

```
x -= 3;
x *= 3;
x /= 3;
x %= 3;
x &= 3;
x |= 3;
x ^= 3;
x <<= 3;
x >>= 3;
x >>>= 3;
```

8. UOI——一元操作符插入：

每个一元操作符（算术操作符(+、-)、条件操作符(!)、逻辑操作符(~)）在每一个正确类型的表达式前被插入。

例如, 语句 “ $x=3*a$ ” 变异后产生了下面的4个语句:

```
x = 3 * +a;
x = 3 * -a;
x = +3 * a;
x = -3 * a;
```

9. UOD——一元操作符删除:

每个一元操作符 (算术操作符(+、-)、条件操作符(!)、逻辑操作符(~)) 被删除。

例如, 语句 “ $if!(a>-b)$ ” 变异后产生了下面的两个语句:

```
if (a > -b)
if !(a > b)
```

在例子中的其他两个有用的操作符是标量变量的替换和 “bomb” 操作符。标量变量的替换产生了很多变体 (如果 V 是一个数字变量, 则是 V^2), 结果证明给定以上的这些操作符它就不是必要的。它包含在这里仅仅是为了方便举例子。“Bomb” 操作符对每个语句只产生一个变体, 但是给定以上的这些变异操作符它同样不是必要的。

10. SVR——标量变量替换:

每个变量的引用都被每个其他的在相同作用域范围内的合适类型的变量所替换。

例如, 语句 “ $x=a*b;$ ” 变异后产生了下面的6个语句:

```
x = a * a;
a = a * b;
x = x * b;
x = a * x;
x = b * b;
b = a * b;
```

11. BSR——Bomb语句替换:

每个语句都被一个特殊的Bomb()函数替换。

只要Bomb()函数被执行就产生一个失败的信号, 因此这要求测试能够到达每一个语句。

例如, 语句 “ $x=a*b;$ ” 变异后产生了下面的语句:

```
Bomb();
```

其他测试标准的包含 (高级话题)

变异被广泛认为是找到最多错误的最强的测试标准。它的代价也是最高的。本节说明变异包含了其他一些覆盖标准。这一点可以通过某个特定变异操作符所要求的条件和另外一个特定的覆盖标准完全相同得到证明。对于一个标准所定义的每个具体的要求, 一个变体就产生了, 只有满足条件的测试用例才可以杀死此变体。因此, 当且仅当与标准要求相关的变体被杀死的条件下才满足覆盖标准。在这种情况下, 我们说保证覆盖标准的变异操作符产生了这个标准。如果一个标准是由一个或者多个变异操作符所产生, 那么变异测试就包含了这个标准。虽然变异操作符随语言和分析工具的不同而不同, 但是本节使用在多数实现中使用的常见操作符。还可以设计变异操作符, 迫使变异包含其他的测试标准。在参考文献注释部分对此有详细解释。

这种证明有一个小小的问题。覆盖标准提出的要求仅仅是局部要求, 比如, 边界覆盖要求程序中的每个分支都要被执行。而变异不仅有局部要求, 还有全局要求。即变异还需要变

体程序产生出错误的结果。对于边界覆盖，有一些特定的变体只有在每个分支被执行而且此变体的最终结果是错误的情况下才会被杀死。一方面，这意味着变异比条件覆盖标准施加了更强的要求。另一方面，有点反常的是，这也说明了满足覆盖标准的测试集有时候并不能杀死所有的相关变体。因此，正如之前定义中所述，变异并不严格包含条件测试标准。

在弱变异基础上进行包含就可以解决这个问题。关于包含其他的覆盖标准，弱变异仅施加局部要求。在弱变异中，在影响阶段不等同但是在传播阶段等同的变体（即一个不正确的状态被掩盖或修复了）被留在了测试用例集中，于是这个边覆盖被包含了。正是由于测试用例被撤走，强变异不包含边覆盖。

因此，本节说明弱变异包含覆盖标准，而不是强变异。

第2章中的图覆盖标准和第3章的逻辑覆盖标准都涉及包含。有一些变异操作符只适用于源程序语句，而其他的变异操作符能适用于任意性结构，如逻辑表达式。例如，一个常见的变异操作符是把那些语句替换成“bomb”，这些“bomb”立即导致程序终止执行或者抛出异常。这种变异只能对程序语句来定义。另外一个常见的变异操作符是将关系操作符（<、>等）替换成其他的关系操作符（ROR操作符）。这种关系操作符替换适用于任何逻辑表达式，包括FSM中的哨条件（guard）。

		(T T)	(T F)	(F T)	(F F)
	$a \wedge b$	T	F	F	F
1	$true \wedge b$	T	F	T	F
2	$false \wedge b$	F	F	F	F
3	$a \wedge true$	T	T	F	F
4	$a \wedge false$	F	F	F	F

图5.3 $a \wedge b$ 不完全真值表

节点覆盖要求执行每一条语句或者基本的程序块。将相关语句替换成“bomb”的变异操作符产生了节点覆盖。要杀死这些变体，我们需要找到能达到各个基本程序块的测试用例。由于这正是节点覆盖的要求，所以这个操作符产生了节点覆盖，而变异包含了节点覆盖。

边覆盖要求执行控制流程表中的每条边。一个常见的变异操作符是将每个断言取代成真或假（ROR操作符）。要杀死真变体，测试用例必须使用假分支；要杀死假变体，测试用例必须使用真分支。这种操作符迫使程序中的每个分支都被执行，于是就产生了边覆盖，变异包含了边覆盖。

子句覆盖要求每个子句成为真语句，或者假语句。ROR、COR和LOR变异操作符一起将每个断言中的每个子句替换成真或假。为了杀死真变体，测试用例必须将这个子句（以及整个谓词）变成假；为了杀死假变体，测试用例必须将这个子句（以及整个谓词）变成真。这正是子句覆盖的要求。简单地阐述这一点的方法是使用一个真值表的修改表。

考虑一个谓词有两个由“and”连接起来的子句，假设谓词是 $(a \wedge b)$ ， a 和 b 是任意布尔子句。图5.3中 $a \wedge b$ 不完全真值表第一行显示了 a 和 b 在4种不同组合情况下的结果值。图5.3中的4种变异，将 a 和 b 分别用 $true$ 和 $false$ 来替换。要杀死变体，测试员必须选择一种导致结果与原谓词不同的输入（图5.3中上方4个真值赋值中的一种）。考虑变体1， $true \wedge b$ 。变体1与原子句中的4种赋值里的3种结果相同。因此，要杀死这个变体，测试员的必须使用结果是真值赋值（FT）的测试用例输入值，如表中方框所示。类似地，变体3， $a \wedge true$ ，只有在使用真值赋值（TF）的情况下才能被杀死。因此，当且仅当满足子句覆盖时，变体1和变体3才能被杀死，而且变异操作符在这种情况下产生子句覆盖。注意，变体2和变体4不需要包含子句覆盖。

用逻辑操作符来举例证明变异操作符产生子句覆盖非常简单直接、明白易懂，但是有些

笨拙。更广泛地,假设谓词 p 、子句 a 和测试 $p(a)$ 的子句覆盖要求,而且 a 的值必须都能够取得真和假。考虑到变异 $\Delta p(a \rightarrow \text{真})$ (即 a 被取代成真的谓词。这个变体能满足影响条件(从而进一步杀死这个变体)的唯一方式是找到一个测试用例,这个测试用例使得 a 呈现出假值。类似地,变异 $\Delta p(a \rightarrow \text{假})$ 只能被使得 a 呈现出真值的测试用例杀死。因此,在一般情况下,将子句取代成真和假的变异操作符产生子句覆盖并被变异所包含。

组合覆盖要求谓词中的子句能够取得各种可能的真值组合。一般情况下,组合覆盖对于一个有 N 个子句的谓词有 2^N 个要求。由于没有一个单一的或者变异操作符组合产生 2^N 个变体,所以显然变异不能包含COC。

主动子句覆盖要求谓词 p 中的每个子句 c 的值必须都能够取得真和假,而且决定谓词 p 的值。第3章中的第一个版本,一般主动子句覆盖允许 p 中的其他语句在 c 的值为真或者假的时候有其他不同值。证明变异包含一般主动子句覆盖比较简单,我们已经证明了这一点。

要杀死变体 $\Delta p(a \rightarrow \text{真})$,我们必须满足影响条件,即使得 $p(a \rightarrow \text{真})$ 与 $p(a)$ 的值不同。也就是说, a 必须决定 p 。类似地,要杀死 $\Delta p(a \rightarrow \text{假})$, $p(a \rightarrow \text{假})$ 必须与 $p(a)$ 的值不同,即 a 必须决定 p 。因为这正是GACC的要求,这个操作符产生了节点覆盖并且变异包含了一般主动子句覆盖。注意,只有当变异程序中的错误值一直传播到表达式的结尾,这才是成立的,这是弱变异的一种解释。

变异操作符既不包含相互关联的主动子句覆盖,也不包括限制性主动子句覆盖,因为CACC和RACC需要若干对测试才能显示出某些特征。对于CACC,其特征是关于某一特定子句的两次测试谓词结果不相同。对于RACC,其特征是minor子句关于某一特定子句的两次测试结果完全相同。由于每个变体被(或是不被)测试用例杀死,(与一对测试用例刚好相反),至少如传统意义上所定义的一样,变异分析不能包含那些在不同对的测试用例之间施加关系的标准。

研究人员还没有弄清楚变异是否包括非活动的子句覆盖标准。

All-defs数据流覆盖标准要求对于一个变量的每个定义都至少到达一次使用。也就是说,对每个在节点 n 上的变量 X 的每个定义,必须存在一个相对于 X 的从 n 节点到某个节点或者某条边的定义清纯(definition-clear)的子路径,同时在这条子路径上有 X 的一个使用。对于All-defs的包含论述稍微比这个要复杂一些,并且不像其他的论述,All-defs要求必须使用强变异。

一个常用的变异操作符是删除一些语句,以强制要求每个程序语句都能对程序的输出产生影响^①。为了说明All-defs的包含,我们的注意力仅限于那些含有变量定义的语句。假设语句 s_i 包含有变量 x 的一个定义,变体 m_i 是删除 s_i 的一个变体($\Delta s_i \rightarrow \text{null}$)。为了在强变异条件下杀死 m_i ,一个测试用例必须(1)让被变异语句能够被达到(可达性), (2)使程序在执行 s_i 之后的运行状态不正确(影响), (3)使最终的程序的输出不正确(传播)。任意一个到达 s_i 的测试用例将会产生一个不正确的运行时状态,因为被变异的 s_i 不会赋值给 x 。为了让变体的最终输出不正确,这样将产生两种情况。首先,如果 x 是一个输出变量, t 必须在一条从删除了 x 的定义处到最终的输出的子路径执行,并且在这条子路径上没有任何 x 的定义(def-clear)。由于输出可以看做一个use,这就满足了标准。其次,如果 x 不是输出变量,那么在 s_i 处不定义 x 就必须产生不正确的输出状态。只要在执行过程中变量 x 在后面某点没有被重定义的使用,这种情

① 这个目标和强迫谓词中的每个子句各自有所不同,但在某种意义上来说是相同的。

况是可能的。因此， t 相对于 x 在 s 处的定义就满足了all-defs标准，并且变异操作符产生出all-defs，所以就能确保变异包含all-defs。

特定地设计一种包含all-uses的变异操作符是可能的，但是这样一个操作符没有发布出来，也没有在任何工具中使用。

5.2节练习

1. 提供一些可达性条件、影响条件和传播条件以及一些测试用例的数据杀死在图5.1中的变体2、4、5和6。

2. 对两个方法findVal()和sum()中的变体，回答(a)~(d)几个问题。

- 如果有可能，找出一个测试输入不能到达变体。
- 如果有可能，找出一个测试输入满足可达性但相对于变体为不满足影响。
- 如果有可能，找出一个测试输入满足影响，但是相对于变体为不满足传播。
- 如果有可能，找出一个测试输入杀死变体m。

```
// 作用：如果numbers为null，就抛出NullPointerException
// 的异常，否则返回val在numbers[]中最后一次出现的
// 下标，如果val不在numbers[]中就返回-1
1. public static int findVal(int numbers[], int val)
2. {
3.     int findVal = -1;
4.
5.     for (int i=0; i<numbers.length; i++)
5'.// for (int i=(0+1); i<numbers.length; i++)
6.         if (numbers[i] == val)
7.             findVal = i;
8.     return (findVal);
9. }
```

```
// 作用：如果x为null，则抛出NullPointerException
// 的异常，否则返回x数组中所有元素的和
1. public static int sum(int[] x)
2. {
3.     int s = 0;
4.     for (int i=0; i < x.length; i++) {
5.         {
6.             s = s + x[i];
6'. // s = s - x[i]; //AOR
7.         }
8.     return s;
9. }
```

3. 参考在第2章中的TestPat程序。考虑下面给出的变体A和变体B：

- 如果有可能，找出一个测试用例不能到达这个变体。
- 如果有可能，找出一个测试输入满足可达性但相对于变体为不满足影响。
- 如果有可能，找出一个测试输入满足影响，但是相对于变体为不满足传播。
- 如果有可能，找出一个测试输入杀死变体m。

```
• while(isPat==false &&isub+patternLen-1<subjectLen)
  While(isPat==false &&isub+patternLen-0<subjectLen) //变体 A
• isPat=false;
  isPat=true; //变体 B
```

4. 为什么移除无效的测试用例很有必要？

5. 使用前面给出的有效的变异操作符为下面的方法cal()定义出12个变体。尽量至少使用每个变异操作符一次。估算一下如果所有cal()方法的变体都被产生，大概有多少变体？

```
public static int cal (int month1, int day1, int month2, int day2, int year)
{
//*****
// 计算出在同一年内在两个
// 给定日期之间的天数
// 前置条件：day1和day2必须在同一年里
//          1 <= month1, month2 <= 12
//          1 <= day1, day2 <= 31
//          month1 <= month2
```

```
//          年数的范围: 1~10000
//*****
int numDays;

if (month2 == month1) // 在同一个月中
    numDays = day2 - day1;
else
{
    // 跳过月份0
    int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    // 是在闰年里吗?
    int m4 = year % 4;
    int m100 = year % 100;
    int m400 = year % 400;
    if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
    else
        daysIn[2] = 29;

    // 从这两个月里的天数开始计算
    numDays = day2 + (daysIn[month1] - day1);

    // 加上这两个月中间的月份的天数
    for (int i = month1 + 1; i <= month2 - 1; i++)
        numDays = daysIn[i] + numDays;
}
return (numDays);
}
```

6. 使用前面给出的有效的变异操作符为下面的方法power()定义出12个变体。尽量至少使用每个变异操作符一次。估算一下如果所有power()方法的变体都被产生, 大概有多少变体?

```
public static int power (int left, int right)
{
    //*****
    // 求出Left的Right次幂
    // 前置条件: Right>=0
    // 后置条件: 返回Left**Right
    //*****
    int rslt;
    rslt = Left;
    if (Right == 0)
    {
        rslt = 1;
    }
    else
    {
        for (int i = 2; i <= Right; i++)
            rslt = rslt * Left;
    }
    return (rslt);
}
```

7. 一个基本假设被陈述为: “在实际中, 如果软件包含一个错误, 那么就会有一组变体, 这组变体能被检测出该错误的一个测试用例杀死。”

(a) 给出一个简明的支持该变异假设的描述。

- (b) 给出一个简要的反对该变异假设的描述。
8. 试着设计出一组包含组合测试覆盖标准的变异操作符。为什么我们不需要这样的操作符？
 9. 在网上查找Jester[⊖]这个工具，它是基于JUnit的。根据你所掌握的知识，给出Jester作为一个变异测试工具的评价。
 10. 下载安装Java的变异工具muJava:<http://ise.gmu.edu/~offutt/mujava/>。将前面的问题中的cal()方法放入一个类中，并且使用muJava测试cal()。注意，一个测试用例是调用cal()函数的一个方法。
 - (a) 产生了多少变体？
 - (b) 需要多少测试用例来杀死这些变体？
 - (c) 不分析等价的变体，你能达到多少变异得分？
 - (d) 有多少等价变体？

5.3 集成与面向对象测试

本书使用术语“集成测试”，用于描述测试各个单独的程序单元间的连接。在Java中，集成测试包含测试连接类、包以及组件间的方法。本章节使用通用的术语“组件”，其中包含面向对象语言所独有的一些特性，如继承、多态和动态绑定。

5.3.1 BNF集成测试

据我们所知，BNF测试还没有在集成层次上被应用。

5.3.2 集成变异

本节首先讨论的是在不考虑面向对象的关系时，变异是如何应用在集成层次上的测试，然后是变异如何测试出涉及继承、多态和动态绑定方面的问题。

在两个组件间集成时发生的错误往往是由错误的假设所造成的。例如，在第1章我们讨论了1999年9月的火星登陆车，是因为其中一个组件的值按英国计算单元（英里）来设的，而接收的组件认为它是按千米来设的，因此造成登陆车被撞毁。关于究竟是通过改调用者、被调者或是两者都改变来修复这类缺陷，取决于系统的设计以及哪种改变能更加有效并且容易。

集成变异（又称为“接口变异”）通过在组件间的连接上产生变异来完成。大多数变异是基于方法的调用，调用（调用方）和被调（被调方）的方法都必须被考虑在内。接口变异操作符执行下列操作：

- 通过修改调用方法传给被调方法的值改变调用方法。
- 通过修改调用改变调用方法。
- 通过修改进入和离开方法的数据改变被调用方法。这个方法包含更高层次的参数及变量（类级别、包、公有的等）。
- 通过修改方法的返回语句改变被调用方法。

1. IPVR——集成参数变量替换：

在方法调用中，方法中每个参数被替换成相兼容类型的其他变量。

⊖ Jester的主页是<http://jester.sourceforge.net/>。

IPVR不能使用不兼容类型的变量，因为它们在语义上不合法（编译器会捕获它们）。在面向对语言中，这个操作除了替换简单类型的变量之外，也可以替换一些对象。

2. IUOI——集成一元运算符插入：

修改一个方法调用中的每个表达式，通过在它之前和之后加入所有可能的一元运算符。

对每种语言的类型，一元运算符可能会不同。在Java中，++和--都可以用作数字类型的前缀和后缀操作。

3. IPEX——集成参数交换：

在方法调用中，交换两个类型相同的参数值。

例如，如果max(a, b)是一个方法调用，那么变异后的方法调用为max(b, a)。

4. IMCD——集成方法调用删除：

删除每个方法调用。如果一个方法有返回值并且该值被用在一个表达式中，那么这个方法调用被一个合适的常量值所代替。

在Java中，对于返回简单类型值的方法应该使用默认值。如果方法返回一个对象，那么调用的方法应该用new()产生一个合适的类的调用来替代。

5. IREM——集成返回表达式修改：

应用5.2.2节的UOI和AOR修改在方法中的每个返回语句中的每个表达式。

面向对象的变异操作符

第1章我们定义了intra-method、inter-method、intra-class和inter-class测试。那些变异操作符都是用于inter-method层次（同一个类的方法之间）和inter-class层次（不同类的方法之间）的。在进行inter-class层测试时，测试人员同样需要考虑在继承和多态方面的错误。这些都是较强的语言特性，并可以用来解决复杂的程序问题，但同样会引入复杂的测试问题。

包含继承和多态特性的语言同样也包含信息隐藏和重载的特性。这样，用于测试这些特性的变异操作符通常包含面向对象的操作，尽管这些并不是将一种语言定义为“面向对象”所至关重要的特性。

为了理解变异测试是如何应用于这些特性的，我们需要在深层次上对语言进行考查。这些在Java上已完成了，其他面向对象的语言趋于相近但有些微小的不同。

封装是一种用于信息隐藏的抽象机制，是使客户端不需要依赖于抽象的具体实现的一种设计技术。封装允许对象限制其他对象对其变量和方法的访问。Java对成员变量和方法有不同层次的访问控制：private（私有的）、protected（受保护的）、public（公有的）和default（默认的）（也称为package）。这些访问权限对许多程序员来说非常难于理解，并且在程序设计中经常不被认真地考虑，因此它们会是许多错误的根源。表5.1总结了这些访问权限。一个私有成员只能由定义它的类来进行访问，如果没有特殊指明访问权限，那默认会是package，它只允许在同一个包内的其他类对其进行访问，但不是其他包的子类。受保护的成员只由定义它的类、子类和相同包下的类所访问。公有成员可由继承结构或包的任何类所访问。

Java并不支持多重类继承，因此每个类只有一个直接的父类。一个子类从它的父类还有它的祖先那儿继承变量和方法，并可以按它们定义的那样使用，或者重写那些方法或隐藏变量。通过使用关键字“super”（super.methodname();），子类同样可以使用它们的父类的变量和方法。Java的继承允许方法的重写、变量隐藏以及类的构造。

表5.1 Java的访问权限

标识符	同一类中	不同的类/相同的包	不同的包子类	不同的包非子类
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

方法重写允许子类的方法拥有和父类一样的方法名、参数的返回值类型。重写允许子类重新定义继承的方法。子类的方法拥有相同的签名，但是不一样的实现。

变量隐藏的实现，是靠在子类中定义的变量拥有相同的名字和类型的继承变量来完成的。这样有效地在子类中隐藏继承的变量。这是很强大的特性，但同样也是潜在错误的发源地。

类构造器不是像其他方法那样继承的。为了使用父类的构造器，我们必须明确地使用super关键字调用，而且还必须是在子类构造器中第一个语句并且参数列表符合父类构造器的参数列表。

Java支持两种类型的多态，即属性和方法，它们都用了动态绑定。每个对象都有一个声明的类型（在声明语句中的类型，即“Parent P;”）和实际的类型（在实例化语句中的类型，即“P=new Child();”或赋值语句，“P=Pold;”）。实际类型可以是声明类型或从声明类型继承过来的其他类型。

多态属性是一个可以是多种类型的对象引用。在程序的任何地方，在不同的执行过程中对象引用的类型可能会不一样。多态方法可以通过声明Object类型的参数，来接受不同类型的参数输入。多态方法常被用来实现类型抽象（C++中的模板和Ada中的泛型）。

重载是在同一个类中对于不同的构造器或方法使用相同的名字。它们必须有不同的签名或参数列表。重载很容易与重写相混淆，因为这两种机制拥有相似的名字和语义。重载发生在同一个类中有两个方法，而覆盖发生在一个类还有它的后代中。

在Java中，成员变量和方法能与类相关联，而不是单个的对象。与类相关联的成员称为类或静态变量和方法。在Java运行时系统中，在类的变量第一次被定义的时候会创建一个单独的静态变量的拷贝。所有该类的实例共享这些静态变量。静态方法只能操作静态变量，它们不能访问类中定义的实例变量。不幸的是这些术语可能会变化，我们说实例变量在类中被定义并且对对象是可访问的，类变量以static声明而局部变量在方法内声明。

变异操作能够为所有的这些语言特性所定义。变异的目的是为了保证程序员能够正确地使用它们。我们尤其需要关心面向对象语言特性的使用的一个原因，是因为现在许多程序员学习它们只是为了完成工作，并没有机会学习怎样有效地使用这些特性的理论规则。

以下是针对信息隐藏、继承、多态、动态绑定、方法重载和类这些语言特性的变异操作。

1. AMC——访问修饰词改变：

改变每个实例变量和方法的访问权限。

AMC操作帮助测试者生成测试，以保证访问权限是正确的。仅当新的访问权限拒绝其他类的访问或允许访问但是会引发命名冲突时，可以杀死这类变异。

2. HVD——隐藏变量删除：

删除每个声明的重写或隐藏变量。

为会造成对其父类（或祖先）中定义的变量的访问，这也是常见的程序错误。

3. HVI——隐藏变量插入：

在子类中增加声明，以隐藏在祖先中声明的每个变量。

当测试用例显示引用重写的变量不正确时，可以杀死这类变异。

4. OMD——重写方法删除：

删除整个重写方法的声明。

该方法的引用使用了其父类方法。这保证了调用的方法就是我们需要的的方法。

5. OMM——重写方法移动：

对重写方法的每个调用都移至方法的第一个和最后一个语句，并向上和向下移动一个语句。

子类中的重写方法有时会调用父类中的原始方法，例如，修改一个对父类是私有的变量。一个常见的错误是在错误的时候调用了父类的方法，从而造成不正确的状态。

6. OMR——重写方法重命名：

重命名父类中被子类重写的方法，如此重写不会影响到父类的方法。

OMR操作被设计用来检查是否一个重写的方法影响了其他方法。考虑在一个类List中一个方法 $m()$ 调用另一个方法 $f()$ 。更进一步，假设 $m()$ 是子类Stack直接继承没有修改的方法，但是Stack中的 $f()$ 被重写。当Stack类型的对象的 $m()$ 被调用时，它调用Stack的 $f()$ 而不是List的。在这种情况下，Stack的 $f()$ 可能会与基父类的方法有交互，从而产生意料之外的结果。

7. SKD——关键字super删除：

删除每个关键字super。

删除后，会引用到本地变量而不是祖先的。SKD操作被设计成用来确保隐藏/被隐藏变量和重写/被重写方法被正确地使用。

8. PCD——父类构造函数删除：

删除每个对父类构造函数的调用。

父类（或其祖先）默认的构造函数会被使用。为杀死这类变异，有必要设计一个测试用例，使父类默认的构造函数创建了的初始状态是不正确的。

9. ATC——实际类型改变：

在new语句中新建对象的实际类型被改变。

这造成对象引用到与原始实际类型不一样的对象的类型上。新的实际类型必须是原始实际类型的同样类型家族（一个后代）。

10. DTC——声明类型改变：

在声明中改变每个新建的对象的声明类型。

新声明的类型必须是原始类型的祖先。实例化仍是合法的（它仍然是新声明类型的后代）。为杀死这类变异，测试用例必须造成对象的特征与新声明的类型不一致。

11. PTC——参数类型改变：

在声明中改变每个参数对象的声明类型。

除了是在参数上，与DTC相似。

12. RTC——引用类型改变：

在赋值语句的右边的对象都改成与之相兼容的对象类型。

例如，如果一个Integer被赋给一个Object类型的引用，可以将指派改为一个String。因为Integer和String都是从Object继承过来的，它们都能够被互相替换。

13. OMC——重载方法改变：

对方法名相同的每对方法，交换它们的内容。

这样保证重载的方法被正确地调用。

14. OMD——重载方法删除：

每次删除一个重载方法的声明。

OMD操作保证了对重载方法的覆盖，即所有的重载方法都必须被调用一次。如果删除方法后变体仍然工作正常，可能是在调用重载方法的过程中出现错误，可能调用了错误的方法，或发生不正确的参数类型转换。

15. AOC——参数顺序改变：

如果存在另一个重载的方法，改变方法调用的参数顺序为那个重载方法的参数顺序。

这样会引发对另一个方法的调用，从而检查在使用重载时的一些常见错误。

16. ANC——参数数量改变：

如果存在另一个重载的方法，改变方法调用的参数的数量为那个重载方法的参数数量。

这样帮助确保程序员没有调用错误的方法。当加入新的值时，它们是简单类型的常数默认值或是由对象默认构造函数产生的。

17. TKD——关键字this删除：

删除出现关键字this的出现。

在方法体中，如果成员被有相同名称的变量局部变量或方法参数所隐藏，使用关键字this来引用当前对象。TKD操作通过替换用“X”“this.X”，来检查成员变量是否被正确使用。

18. SMC——static修饰词改变：

移除每个static修饰词实例，并给每个实例变量加上static修饰词。

这个操作验证了实例和类变量的使用。

19. VID——变量初始化删除：

移除每个成员变量的初始化。

实例变量能够在类的变量声明和构造函数中被初始化。VID操作移除这些初始化，以使得所有的成员变量初始化为默认的值。

20. DCD——默认构造函数删除：

删除每个默认构造函数（不带参数）的声明。

这样保证用户自定义的构造函数被正确地执行。

5.4 基于规范的语法

通用术语“基于规范”应用在抽象层次上描述软件的语言上。它包括形式化规范的语言，如Z、SMV、OCL和非形式化规范语言和设计标识，如状态图、FSM和其他UML图标识。这

些语言越来越被广泛应用，部分是因为对强调软件质量的意识的增加，部分是因为UML的广泛使用。

5.4.1 BNF语法

据我们所知，终端符号覆盖 (terminal symbol coverage) 和产出覆盖 (production coverage) 仅被应用在一种类型的规范语言上：代数规范。它的思想是把代数规范中的等号看成是语法的产出规则，并导出方法调用的字符串用来覆盖等式。由于代数规范还没被广泛应用，所以本书不讨论这个主题。

5.4.2 基于规范的变异

在规范层次上，变异测试同样也是有价值的方法。事实上，对某些特定的规范，变异分析更加简单些。在本节中我们用FSM来描述规范。

像在第2章中定义的那样，FSM基本上是一个图G。它包括的状态（节点）集、初始状态集（初始节点）和状态转移关系（边的集合）。当使用FSM时，有时边和节点需要明确定义，就像传统的带圆框和箭头的图一样。但是，有时FSM可以通过如下的方式简洁地描述。

1. 状态暗含的用声明的变量定义，这些变量的范围是有限的。此时的状态空间是这些变量区间的笛卡尔图。

2. 初始状态是由一些或所有的区间有限的变量所定义的。

3. 状态转移根据规则来制定，这些规则能用来描述每个转换的源状态和目标状态。

以下的例子澄清了这些思想。我们描述了一个语法简单的机器，并显示了有详细列举状态和转移的同样的机器。尽管这个例子很小不足以说明全部，但在SMV中语法形式一般比图形的形式更简短。事实上，因为状态空间的增长是组合式的，所以我们很容易定义出状态空间太大不能穷尽描述的FSM，尽管状态机能被有效地分析。下面是用SMV语言描述的一个例子。

```
MODULE main
#define false 0
#define true 1

VAR
    x, y : boolean;

ASSIGN
    init (x) := false;
    init (y) := false;

    next (x) := case
        !x & y : true;
        !y   : true;
        x    : false;
        true  : x;
    esac;

    next (y) := case
        x & !y : false;
        x & y  : y;
        !x & y : false;
```

```

true : true;
esac;

```

这里出现了两个变量，每个变量只有两个值（boolean），因此状态空间的大小为 $2*2=4$ 。在ASSIGN下的两个初始语句定义了一个初始状态。状态转移图如图5.4中所示。SMV的转移图能机械地按照规范生成出来。给定一个状态并决定下一个状态这些变量的值是多少。例如，假设上面描述的状态是（true, true），x的下一个值是由语句“x:false”指定。x为true，因此下一个值是false。同样地，x&y是true，因此y的下一个值是当前的值，或者为true。这样，状态（true, true）的下一个状态是（false, true）。如果在一个case语句中有多个条件为true，那么选择第一个是为true。SMV没有像C或Java语言中的“fall-through”语义。

我们的上下文结构中有两个尤其需要注意的方面。

1. 有限状态描述能在一个较高的层次上捕获系统的状态——适合于与终端用户交流。FSM对硬件部分的测试、命名系统的测试是非常有用的。

2. 一些研究验证的社区为FSM开发了强大的分析工具。这些工具有很高的自动化能力。更进一步，这些工具以证据和反例的形式产生出详细的证明在FSM中不存在的属性。这些反例能用来产生测试用例，这样，从FSM自动产生测试用例能比从源程序产生的更加容易些。

变异和测试用例

变异状态机描述的语法与变异源程序相类似。变异操作必须被定义，并把它们应用到描述上。一个例子是常量替代操作，它把每个常量替换为另一个常量。对y的下一个语句，给定式子!x&y:false，替换为!x&y:true。这个变体的FSM如图5.5所示。新的转换用额外的粗箭头表示，替换的语句用叉箭头表示。

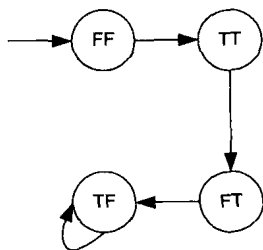


图5.4 SMV规范的FSM

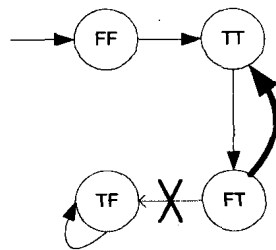


图5.5 SMV规范变异的FSM

生成测试用例来杀死这类变体，与基于程序的变异稍有不同。我们需要根据原始状态机的转移关系得到状态序列，而不是根据变异的状态机得到的。这样的序列能产生测试用例来准确杀死这类变异。

使用模型检查器能自动地找到测试用例杀死基于SMV的FSM的变体。模型检查器有两个输入，一个是用像SMV这样的形式语言定义的FSM，另一个是以时序逻辑语句描述的一些属性的声明。我们这里不会详细说明时序逻辑，这些逻辑能用来表达“当前”或未来可能为true的属性。下面是一个简单的时序逻辑语句：

原始表达式!x&y : false与变异的表达式xly : true 永远是相同的。

对于给定的例子，当且仅当这个状态序列被变异的状态机拒绝时，对于原状态机所允许的状态序列这个语句是错的。换句话说，这样的序列无疑可以杀死变体。我们在以上的状态机中加入如下的SMV语句：

SPEC AG (!x&y) $\rightarrow$ Ax(y = true)

模型检查器将会显式地产生如下可供考虑的测试序列：

```
/* state 1 */ { x = 0, y = 0 }
/* state 2 */ { x = 1, y = 1 }
/* state 3 */ { x = 0, y = 1 }
/* state 4 */ { x = 1, y = 0 }
```

有时一些变异的状态机与原始的状态机相等。模型检查器能有效地应对这种情况。关键的理论原因是模型检查器工作在一个有限域，相关于突变的问题是可判定的（不像程序代码）。换句话说，如果模型检查器不能产生反例，那么我们就知道变体是等效的。

5.4节练习

（有挑战的！）找到或写一个简单的SMV规范和一个相应的Java实现。用SPEC断言来重述程序的逻辑。系统地变异断言，并从（非等价的）变体中收集追踪信息。用这些追踪信息来测试实现。

5.5 输入空间语法

正式地定义一个程序、方法或者软件组件的输入句法是一种常见的语法的使用。本节主要内容就是解释如何将本章的准则应用到定义程序输入空间的语法中。

5.5.1 BNF语法

5.1.1节的内容主要是讲BNF语法的标准。语法的常见的作用是定义程序或者方法输入时的精确句法。

假设有一个程序用于处理存贷款流程，每一笔存款是deposit account amount的形式，而每一笔贷款是debit account amount的形式。可以通过下面的正则表达式来表达这个程序的输入结构：

(deposit account amount|debit account amount)*

这个正则表达式描述了存贷款过程中的任何一种顺序。（5.1.1节中的例子实际上是这个例子的抽象版本。）

这个正则表达式的输入描述依然很抽象，它并没有讨论有关账户或数目究竟是什么，长成什么样。在后面的章节中，我们将继续精化这些细节。从这个定义的语法中得到的一种输入如下：

```
deposit 739 $ 12.35
deposit 644 $ 12.35
debit 739 $ 19.22
```

画图来表达这个正则式的效果并不难。形式上，这些图都是有限自动机（确定的或非确定的）。无论在何种情况下，每个都可以直接应用第2章的覆盖率标准。

图5.6给出了以上结构的一个可能的图。它包含了一个状态（就绪）和代表两种可能输入的两个转移。以上给出的这个输入测试例子

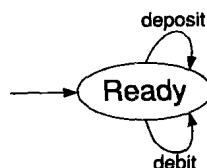


图5.6 银行FSM的例子

满足了图5.6中所有节点和所有边的标准。

虽然正则表达式能够满足部分程序，其他的需要通过语法来满足。因为语法比正则表达式更加具有表达力，所以我们不需要两种方法都加以使用。之前的例子用基于语法的形式来表达，所有关于账号和金额都包括在内：

```
bank ::= action*
action ::= dep | deb
dep ::= "deposit" account amount
deb ::= "debit" account amount
account ::= digit3
amount ::= "$" digit+ "." digit2
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

如图5.7所示，即使像这样的简单例子，如果所有包含的细节越多，图就会变得越大。

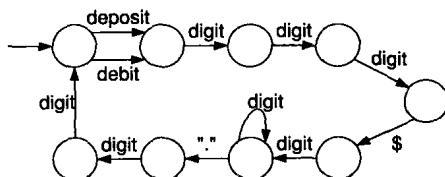


图5.7 表示银行例子语法的FSM

对于上述的测试用例的完整推导可以由以下的形式开始：

```
stream → action*
      → action action*
      → dep action*
      → deposit account amount action*
      → deposit digit3 amount action*
      → deposit digit digit2 amount action*
      → deposit 7 digit2 amount action*
      → deposit 7 digit digit amount action*
      → deposit 73 digit amount action*
      → deposit 739 amount action*
      → deposit 739 $ digit+ . digit2 action*
      → deposit 739 $ digit2 . digit2 action*
      → deposit 739 $ digit digit . digit2 action*
      → deposit 739 $1 digit . digit2 action*
      → deposit 739 $12. digit2 action*
      → deposit 739 $12. digit digit action*
      → deposit 739 $12.3 digit action*
      → deposit 739 $12.35 action*
      :
```

从这个语法推导测试用例的执行，会利用它的一个产生式系统地替代下一个非终端符号（动作）。下面的练习需要实现完整测试来满足终端符号覆盖以及产出式覆盖。

当然，通常输入语法的一个非正式的描述是存在的，但是并不一定是正式的语法。这意味着，测试工程师将要负责对输入语法正式地描述的工程任务。这个过程很有价值，而且它将把需求和软件中的二义性和缺失性暴露出来。因此，这一步需要在软件开发的初期完成，

肯定要在软件实现之前，最好在设计开始前。一旦定义了，会对在程序中直接利用语法实现运行期间输入的验证有帮助。

XML举例

一种很快流行起来的用于描述输入的语言叫可扩展标记语言（eXtensible Markup Language, XML）。XML语言一个最常用的用途是网络应用程序以及网络服务，但是XML语言的结构设计得很一般化，它可以适用于很多地方。XML语言是一种用于描述、编码以及传输数据的语言。所有的XML消息（有时也叫做文档）都是纯文本的形式，并且用类似于HTML的语法。XML语言是和以语法（模式）形式描述输入消息的内置式语言一起产生的。

与HTML类似，XML也使用标签，标签是对于数据的纯文本的描述，用尖括号（“<”和“>”）括住。所有的XML消息都必须按照严格的格式编写，即都有一个单独的文档元素，而其他的元素正确地嵌套在其中。而且每个开始标签必须有一个对应的结束标签。图5.8是XML消息的一个简单的描述书的例子。这个例子用于阐明软件中利用XML消息的BNF测试的使用。这个用例列出了两本书，标签名字（“书籍”、“书”、“ISBN”等）必须能自我描述，XML消息形成了一个总体的层次结构。

XML文档可以被用XML模式写的语法定义约束。图5.9描述了书籍的模式，这个模式表示一个“books”XML消息可以包含无限多的book标签，这个book标签包含6项信息。“title”、“author”和“publisher”这3项是简单字符串类型。“price”是十进制（数值）类型，在十进制位数后有两数字，而且最低位是0。“ISBN”和“year”这两个数据项是稍后在模式中将定义的类型。“yearType”是一个4位整数表示的类型，“isbnType”是一个最多有10位数字字符的类型。每本书必须有一个书名、作者、出版社、单价和年份，ISBN值是可选的。

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file for books-->
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
  <book>
    <ISBN>0471043281</ISBN>
    <title>The Art of Software Testing</title>
    <author>Glen Myers</author>
    <publisher>Wiley</publisher>
    <price>50.00</price>
    <year>1979</year>
  </book>
  <book>
    <ISBN>0442206720</ISBN>
    <title>Software Testing Techniques</title>
    <author>Boris Beizer</author>
    <publisher>Van Nostrand Reinhold, Inc</publisher>
    <price>75.00</price>
    <year>1990</year>
  </book>
</books>
```

图5.8 书籍描述的简单XML消息

给定一个XML模式，在5.1.1节中定义的标准，可以用来生成用做测试输入的XML消息。根据产出式测试的覆盖标准，这个简单的模式共需要两条XML消息，一条是包含ISBN的消息，而另外一条是不包含ISBN的消息。

5.5.2 输入语法的变异

通常程序会拒绝一些不规范的输入，这一属性肯定是需要测试的。而这点却往往很容易被程序的实现者所忽略，因为他们往往把全部精力都放在程序的功能实现上面。

无效的输入真的会产生影响吗？从程序的正确性来看，无效输入是指那些超出特定功能的前置条件的输入。这些行为包括程序失败中断、运行时异常和“总线出错，存储器清除”。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:element name="books">
    <xs:annotation>
      <xs:documentation>XML Schema for Books</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="publisher" type="xs:string"/>
              <xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
              <xs:element name="year" type="yearType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="yearType">
    <xs:restriction base="xs:int">
      <xs:totalDigits value="4"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="isbnType">
    <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{10}" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

图5.9 书籍的XML模式

然而，程序预计功能的正确性仅仅是一小部分内容。从实用的角度来说，无效的输入有时会造成很大的影响，因为这些无效的输入可能造成始料未及的后果。例如，一个没有处理过的无效输入通常会有安全性方面的弱点，会让恶意攻击者破坏软件。无效的输入通常会使软件按照很奇怪的方式运行，这就会被恶意攻击者利用。这就是典型的“缓冲区溢出攻击”的工作流程。缓冲区溢出攻击中的关键步骤是提供一个极长的输入，以至于无法放入空闲的缓冲区。类似地，网络浏览器攻击中一个关键的步骤就是传送一个含有恶意HTML、JavaScript或者是SQL代码的字符串输入。软件需要对无效的输入做出“理性”的回应。“理性”的响应可能不总是被定义，但是测试工程师如论如何需要对此响应负责。

为了能够既评价软件的行为,又能提供安全性,通常需要设计一些包括非法输入的测试用例。一种实现此测试用例的常用方法是变异一个语法。当语法发生变异的时候,变体是测试,我们负责创造有效的和无效的字符串。不使用基础的字符串,所以消灭变体就不能被应用到变异语法中。一些变异操作符有如下的定义。

1. 非终端替换:

产生式中的每个非终端符号被其他非终端符号替换。

这是一个非常广的变异操作符,这种变异可能产生很多字符串,这些字符串不仅无效,而且使它们离有效的字符串很远,所以这种变异往往对测试来说没有太大意义。如果语法提供特殊的规则或者句法约束,一些非终端的替换可以避免。这和在基于程序的变异中避免编译错误是同一个道理。例如,一些字符串表示的类型结构,只有相同的类型或者兼容的类型可以被替换。

产生式`dep::="deposit" account amount`可以变异成以下产生式:

```
dep ::= "deposit" amount amount
dep ::= "deposit" account digit
```

这可以导致以下对应的测试:

```
deposit $19.22 $12.35
deposit 739 1
```

2. 终端替换:

产生式中的每个终端符号被其他终端符号所替代。

与非终端替换一样,一些终端替换可能会不合适。识别它们依赖于被变异的特殊语法。例如,产生式`amount::="$"digit+"."digit2`可以被变异成以下3种产生式:

```
amount ::= "." digit+ "." digit2
amount ::= "$" digit+ "$" digit2
amount ::= "$" digit+ "1" digit2
```

会有下面3种对应的测试:

```
deposit 739 .12.35
deposit 739 $12$35
deposit 739 $12135
```

3. 终端和非终端的删除:

产生式中的每个终端和非终端符号被删除。

例如,产生式`dep::="deposit" account amount`可能会被变异成下面3种产生式:

```
dep ::= account amount
dep ::= "deposit" amount
dep ::= "deposit" account
```

会有以下3种对应的测试:

```
739 $12.35
deposit $12.35
deposit 739
```

4. 终端和非终端的重复:

产生式中的每个终端和非终端符号被重复。

此类情形通常叫做“结巴”操作符。例如，产生式`dep::="deposit" account amount`可能会被变异成下面的3种产生式：

```
dep ::= "deposit" "deposit" account amount
dep ::= "deposit" account account amount
dep ::= "deposit" account amount amount
```

会产生下面3种对应的测试：

```
deposit deposit 739 $12.35
deposit 739 739 $12.35
deposit 739 $12.35 $12.35
```

我们显然对于处理基于程序的变异操作符，比处理基于语法的变异操作符要更加有经验，所以这个列表也并不是完全的权威。

可以用两种方式应用变异操作符。一种是变异语法并且生成相应的输入。另外一种是使用正确的语法，但是在每次推导过程中，对于使用的产生式应用一次变异操作符。操作符通常是在生成过程中被应用，因为这样生成输入会比通过毁坏整个语法而得到的测试用例更接近于有效输入。这种方法在之前的例子中也使用过。

与基于程序的变异一样，来自语法规则变异后的一些输入仍然可能是正确的输入。上面的例子讲述了把规则

```
dep ::= "deposit" account amount
```

改变成

```
dep ::= "debit" account amount
```

带来了“相等的”变体。变异后的输入结果`debit 739 $12.35`，仍然是一个有效的输入，然而对客户来说会是截然不同的两种效果。如果专门是为了生成无效的输入的话，必须找到一些方法来排除经过变异之后仍然是正确的变异输入。虽然这听起来很像程序中等价的问题，区别很小但是却很重要。这个问题是可以解决的，并且可以通过从语法创建一个标识器来解决，同时检查每个生成的字符串。

许多程序都应该会从较大的语言中接受一些输入而不是全部。例如，网络应用程序可能会约束其输入必须是HTML的子集。在这种情况下，我们有两种类型的语法：一种是整个语法，另外一种语法的某个子集。在这种情况下，生成的最有用的无效测试往往适用于第一类语法，而不适用于第二类语法。

XML举例

5.5.1节列举了如何从模式语法定义中以XML消息格式生成测试。对XML模式应用变异来产生无效消息也很方便。一些程序会使用XML解析器对照语法验证消息正确性。如果这样做，那么很多无效消息很有可能会有正确的结果，但是测试人员仍然需要证实这一点。如果一个有效的解析器没有被使用，这很可能造成许多编程错误。也有很多程序会使用没有明确模式定义的XML消息。在这种情况下，测试工程师在开发测试的时候第一步先建立一个模式是很有用的。

XML模式有很多内置的数据类型，这些类型往往有很多约束方面。在XML中，约束方面往往用来进一步约束数据值的变化范围。在图5.9的例子中，使用了几个约束方面，包括小数点位数（`fractionDigits`）、包括最小值（`minInclusive`）以及最小出现次数（`minOccurs`）。这就

要求进一步修改约束方面值的XML模式的变异操作符。这样常常能导致很多软件的测试以XML语言描述作为输入。

下面给出的4行是图5.9的书籍模式：

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
<xs:totalDigits value="4"/>
<xs:pattern value="[0-9]{10}"/>
```

我们可以这样构造变体：

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="1"/>

<xs:element name="price" type="xs:decimal" fractionDigits="1" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="3" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="1"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" maxInclusive="0"/>

<xs:totalDigits value="5"/>
<xs:totalDigits value="0"/>

<xs:pattern value="[0-8]{10}"/>
<xs:pattern value="[1-9]{10}"/>
<xs:pattern value="[0-9]{9}"/>
```

5.5节练习

1. 以银行系统为例，基于5.5.1节中提到的BNF生成满足TSC的测试用例。尽量使其不满足PDC。
2. 以银行系统为例，生成满足PDC的测试用例。
3. 考虑下面的以符号A开头的BNF：

```
A ::= B "@" C "." B
B ::= BL | L
C ::= B | B "." B
L ::= "a" | "b" | "c" | ... | "y" | "z"
```

并且有以下6种可能的测试用例：

```
t1 = a@a.a
t2 = aa.bb@cc.dd
t3 = mm@pp
t4 = aaa@bb.cc.dd
t5 = bill
t6 = @x.y
```

对于6个测试用例中的每一个用例，(1) 识别满足BNF的测试序列，并给出一个推导。或者(2) 识别不满足BNF的测试序列，并且给出一个能影响测试结果的变异的推导过程。(每个测试仅仅用一个变异，而且每次测试只用一个变异)。

4. 为5.2.2节的家庭作业中的cal()方法设计一个满足BNF描述的输入。并简洁地描述很难通过BNF建模的输入的需求和约束。
5. 根据以下语法，回答问题(a)~(c)：

```
val    ::= number | val pair
number ::= digit+
pair   ::= number op | number pair op
```

```
op      ::= "+" | "-" | "*" | "/"
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

同样考虑下面的变异，针对此语法增加了一个附加的规则：

```
pair ::= number op | number pair op | op number
```

(a) 下面的哪一个字符串可以由（未变异的）语法生成？

```
42
4 2
4 + 2
4 2 +
4 2 7 - *
4 2 - 7 *
4 2 - 7 * +
```

(b) 找出一个是由变异语法得到的字符串，并且此字符串不能由原语法直接生成。

(c)（有挑战性！）找出一个字符串，它是由新的变异语法规则生成的，并且也可以由初始的语法生成。用两种相关的推导方法来描述你的结论。

6. 利用下面的语法，回答问题(a)和(b)：

```
phoneNumber ::= exchangePart dash numberPart
exchangePart ::= special zeroOrSpecial other
numberPart   ::= ordinary4
ordinary     ::= zero | special | other
zeroOrSpecial ::= zero | special
zero         ::= "0"
special      ::= "1" | "2"
other        ::= "3" | "4" | "5" | "6" | "7" | "8" | "9"
dash         ::= "-"
```

(a) 对以下的号码按是否是电话号码进行分类。对于那些不是电话号码的，说出理由。

- 123-4567
- 012-3456
- 109-1212
- 346-9900
- 113-1111

(b) 考虑以下的语法变异：

```
exchangePart ::= special ordinary other
```

如果可能，确认一个字符串在变异语法中会出现，而在原语法中不存在。再找一个字符串是在原语法中，但不在变异语法中。再找一个字符串，既在原语法中，又在变异语法中。

7. Java提供了一个包（java.util.regex），用于操作正则表达式。为URL写一个正则表达式，并且用你定义的正则表达式来评价这类URL地址。这项作业包括编程，因为没有自动化的输入结构测试是没有意义的。

(a) 为一个URL地址写（或者找）一个正则表达式。你写的正则表达式可以不用对每个URL地址都普遍适用，但是给出你最大的努力（例如，“*”号就不能被认为是一个很好的努力）。强烈地鼓励你去网上搜索相关正则表达式的一些写法。建议可以访问正则表达式库（Regular Expression Library）。

(b) 从一个很小的Web站点（例如一组课程网站页面）搜集一组URL地址。必须至少有20个

不同的URL地址。使用java.util.regex包和所定义的正则表达式，来验证每个URL地址。

- (c) 构造一个有效的URL地址，但根据你所定义的正则表达式来看，却是无效的（并且通过对java.util.regex合适的调用来展示）。如果你第一部分做得特别好，那么请解释一下为什么你的正则表达式没有包含此类的URL地址。

8. 为什么等价的变异问题对于BNF语法来说是可以解决的，而对于基于程序的变异却是不能解决的呢？（提示：问题的答案是基于一些相当微妙的理论。）

5.6 参考文献注释

利用语法来编译测试的做法我们可以追溯到Hanford[150]，他驱动了后序的相关工作的进行[26, 107, 176, 285, 294]。Maurer的数据生成语言（DGL）工具[231]表明了基于语法的生成在很多类型的软件中都有相应的应用，同样的主题在Beizer[29]的书书中也有提到。

历史上第一次提到有关变异分析的思想是在1971年Richard Lipton的课程学期论文中提到的。最初的学术研究论文是Budd和Sayward[52]，Hamlet[148]，DeMillo、Lipton以及Sayward[99]在20世纪70年代末发表的。DeMillo、Lipton以及Sayward的论文被选为具有重大影响的论文。变异最初是通过创建变异版本的源文件这种方式被应用到软件中去的，但也被应用到正式的软件规格说明书中。

对于变体数量的最初分析是由Budd[53]完成的，他分析了程序生成的变体的数量，并且发现变体的数量与变量引用次数和数据对象的乘积大体上是成比例的（ $O(Refs * Vars)$ ）。后来的一个分析证实了变体数量是 $O(Lines * Refs)$ ——假设程序中数据对象的数目是和行数成正比的。对于大部分程序，复杂度降为 $O(Lines * Lines)$ ，这个数字在文献中经常出现。

Offutt等人[269]对实际程序做了一个统计的回归分析表明，程序的行数和程序的变体数并没有太大的关系，但是Budd的数据是精确的。这个变异选择的方法在“设计变异操作符”中提到，减少了数据对象的数目，使得变体的个数和变量的引用数成正比（ $O(Refs)$ ）。

已经被广泛地讨论的变异的变种是弱变异[134, 167, 358, 271]。然而，实验证明差异很小[163, 226, 271]。很多语言都有自己量身定做的变异操作符，包括Fortran IV[19, 56]、COBOL[151]、Fortran 77[101, 187]、C[95]，C集成测试[94]、Lisp[55]、Ada[40, 276]、Java[185]以及Java类关系[219, 220]。

为Fortran IV和77、COBOL、C、Java以及Java类关系等研究概念证明的工具已经创建。到目前为止，使用最广泛的工具是Mothra，一个专门针对Fortran 77的变异系统，于20世纪80年代中期在Georgia Tech创建。Mothra是在Rich DeMillo的领导下进行开发的，其中的主要设计是由DeMillo和Offutt完成，而大部分实现则是由Offutt和King来完成的，Krauser和Spafford给予了一定的帮助。在20世纪90年代初期是全盛时期，Mothra在100多个站点上安装，并且对其进行了开发研究，以及把Mothra用作实验室，使得数十篇博士论文和上百篇论文得以发表。根据我们现在所知的，现在唯一一个持变异的商业工具是Certess公司设计的，在集成电路设计领域。

耦合效应表明，复杂的错误往往是和简单错误联系在一起的。所以如果检测到了所有的简单错误，那么也会检测到多数的复杂错误[99]。这一耦合效应在1992[263]年针对若干程序被经验地证明了正确性，并且在1995年证明耦合效应对于大型软件程序盖然地成立[335]。

Budd[51]讨论了程序邻接性的概念。这一邻接性理论被用来表明有为程序员的假设[99]。变异测试的基础前提,正如Geist等人[133]所提到的那样:在实践中,如果软件产品包含一个错误,那么总会有一个组变体能够被一组测试用例杀死,而这组测试用例也能发现程序中的那个错误。

用“bomb”替换每个语句的操作在Mothra[187]中叫做语句分析(SAN)。Mothra的关系操作符替换(ROR)操作把每一个关系操作符($<$ 、 $>$ 、 $\leq$ 、 $\geq$ 、 $=$ 、 $\neq$)之间的互相替换以及表达式的真假值变换。以上的证明仅仅只包含了后者的操作符。Mothra的逻辑连接替换(LCR)操作替换每个逻辑操作符的表达式($\wedge$ 、 $\vee$ 、 $\equiv$ 、 $\neq$)之间的互相替换,以及整个表达式用真假值, true、false、leftop以及rightop替换。leftop和rightop是特别的变异操作符,这两个操作符分别返回一个关系表达式的左侧和右侧。变异操作符表达式可以删除每个程序表达式,在Mothra[187]中叫做语句删除(SDL)。

许多作者[14, 15, 37, 298, 350]已经用了模型检查器来生成测试用例,包括基于变异的测试用例。Huth和Ryan[173]提供了一个简单的访问模型检查的介绍,并且讨论了SMV系统的使用。

在网页方面被应用到异构软件组件的数据传输中的一项核心的技术是可扩展标识语言(XML) [1, 43]。基于数据的变异定义了有关变异操作的一般类别。这些变异操作类将会和不同的语法规则一起协同工作。目前的文献引用了修改字符串值的长度的操作类,并且决定一个值是否是已定义的值。

第三部分 在实践中运用的标准

第6章 实际的考虑

本书的前5章介绍了一些测试标准，它们帮助测试人员丰富了“工具箱”。尽管单独学习这些测试标准是有必要的，但是一个软件组织在走向第3级或第4级测试时的一个最大障碍，就是将有效的测试集成到其软件开发过程中。本章将讨论在软件开发过程中应用这些标准时会遇到的各种问题。最重要的一条就是思考：如果测试人员能够将它视为一个技术问题，并寻求技术上的方案，他将发现障碍并不像刚见到时那么可怕。

6.1 回归测试

回归测试是软件被修改以后进行的再次测试过程。回归测试构成了商业软件开发过程中最重要的部分，并且它是任何有生机的软件开发过程的必不可少的一部分。大的组件或系统往往有更大的回归测试用例集。尽管如此，很多开发者不愿意相信（即使面对无可争论的证据），系统某个部分的小小变化常常会引起离其很远的部分的问题。回归测试就是用来解决这种问题的。

有必要强调的是，回归测试必须是自动化的。事实上，可以这么说，没有自动化的回归测试相当于没有进行回归测试。有大量的商用自动化回归测试工具可以使用。录制/回放工具能够对图形化用户界面程序进行自动化测试。已经用于管理系统不同版本的版本控制软件，能够有效管理与每个版本关联的测试用例集。脚本软件能够管理获取测试输入、执行被测软件、整理输出、对比实际和期望结果和生成测试报告的流程。

本节的目标就是说明回归测试集应该包含哪些类型的测试、应该执行哪些测试用例以及如何对回归中的失败用例做出反应。我们会逐一讨论这些问题，并引导读者对参考文献注释的详细话题感兴趣。

测试工程师在决定哪些测试应该回归时面临着金发带（Goldilocks）问题。如果包含每个测试集可能会导致回归测试集太大而不可管理，这样回归测试执行的频率就赶不上被测软件发生变化的频率。对于很多软件开发组织，这个频率大约是一天，回归测试在夜晚对软件变化进行评估，开发人员在第二天早上去查看测试结果。如果回归测试没能够执行完成，那么开发的过程就会被打乱。当然你可以多投入一些计算机硬件来增加计算资源，使得测试能够更快地运行，但是这个投入远大于你运行多余的测试用例所获得的好处。另一方面，如果一个测试集太小以至于不能包含软件所要测试的功能，那么通过回归测试的软件也可能包含很多缺陷。

上一段实际上并没有提到如何挑选测试用例到回归测试中以使得回归测试用例集的大小适中。一些组织的策略是这样的，对于每个报告的缺陷，这个缺陷所属的那一部分都要有测

试用例能够检测到这个缺陷。这里面的道理就是，相对于同一个问题反复地在不同版本中出现，用户更倾向于容忍新版本中出现新的缺陷。上述的方法有一定的合理性，因为每个测试的选择都有具体的理由。

本书的核心部分关于覆盖标准的介绍为评估回归测试用例集提供了良好的基础。例如，如果方法调用覆盖说明某些方法从未被调用，那么说明这个方法是死代码或者在需要增加一个测试用例使这个方法能够被执行。

如果一个或多个回归测试失败，第一步就是判断软件的这次更新是不是有缺陷，或者回归测试集本身是不是有问题。对于任何一种情况，我们都需要做额外的工作。如果不是回归测试失败，我们仍然有事情要做。原因是某个版本的回归测试集不一定适用于后续版本的软件。软件发生的改变可以分为这样几类：修正、完善、适应、预防。所有这些改变都需要有回归测试。即使软件的外部功能没有发生改变，我们仍要检查它对于新版本的软件是否适用。例如，预防性的维护可能导致大规模的内部代码重构，如果原来选择的回归测试是根据内部实现的结构来生成的，那么我们肯定要修改测试集来适应新的软件实现。

随着软件发生的改变对回归测试集进行演化是很有挑战性的工作。软件外部接口的变化尤其令人感到头疼，因为这样的改变可能会导致所有的测试都失败。例如，某个特定的输入从一个下拉菜单改到另一个地方了，这会导致所有的录制/回放测试用例必须都要更新。或者假设新版本的软件会产生比原来更多的输出结果，那么测试用例中所有期望结果都要随之更新并且相应增加。很明显，测试集的自动化维护和测试的自动化执行一样重要。

添加少量的测试到回归测试中往往比较简单，每个附加的测试的边际成本一般很小。但是累计下来，测试集也会变得很难处理。从回归测试集中移除一个测试集的风险很大。通常，被移除那个测试用例本来可以发现在运营现场将会出现的错误。幸运的是，我们可以使用测试用例构建时的方法来指导回归测试用例集的更新。

限制回归测试执行时间的另一个方法是，从回归测试中选择一个子集来运行，这在文献研究中引起了很大的关注。例如，如果一个测试用例的执行没有访问到任何修改的部分，那么这个测试用例在软件修改后执行的结果应该和修改前的结果是一样的，因此它可以被安全地忽略掉。这里面的选择技巧包括线性方程、标记执行、路径分析、数据流分析、程序依赖图、系统依赖图、修改分析、防火墙定义、丛聚识别、切片、图走查和修改项分析。例如，看过第2章的读者可能会猜测，用数据流选择技术只选择接触到新的、修改了的或删除了的DU对，而忽略了其他的测试。

一个选择技术是有包含性的是指其包含能揭露修改的测试用例的程序。不安全选择技术的包含性小于100%。一个技术是精确的是指其舍去不能揭露修改的回归测试用例的程度。一个技术是有效率的是指其所选择合适的回归测试用例子集比起简单执行省去的测试用例所需的计算时间少的程度。最后，一个选择技术是有通用性的是指其对于多种实际情况应该都能够适用的程度。继续上面的例子，用数据流的方法来选择回归测试用例并不能保证安全性，也不是精确的，一些程序属性具有多项式复杂度，显然需要数据流图这一级别的数据流信息和程序插桩信息。本章参考文献注释部分包含了有关这项工作进一步的研究参考文献，包括实际评估等。

6.2 集成和测试

软件是由很多大小不同的片段组成的，独立的程序员通常只负责测试最底层的组件（类、

模块和方法)。接着,随着软件的集成,程序员必须协作进行测试。软件可以有多种集成方式,本节讨论在测试过程中必须使用的技术策略。我们不会将它们按照流程的角度进行归类。

集成测试是用来检查各个模块之间的兼容性和接口的正确性,也就是说,它是将小的子组件组合成大的工作组件所需的测试。这和对已经集成好的组件进行的测试是不同的。

本章使用“组件”这个软件术语来表达很宽泛的语义:组件是程序的一个单元,它能够被独立地测试。所以,类、模块、方法包甚至代码片段都可以看成组件。

集成测试常常在尚未完成的系统中采用。测试人员可能会评估系统的许多成分中的两个在一起运行会怎么样,可能在整个系统完成之前测试集成的方面,或者可能将系统一点一点地集中到一起,并且评估每个新加进来的成分是否与之前集成的成分相适合。

6.2.1 桩和驱动程序

当测试软件不完整的一部分时,开发人员和测试人员常常需要额外的软件组件,有时叫做脚手架(scaffolding)。两个最普遍脚手架的类型就是测试桩和测试驱动。测试桩是软件模块的一种概括或者具有特殊目的的实现,用来开发或测试一个调用或者依赖桩的组件。桩替换掉了一个被调用的组件。对于OO程序,XP组织已经开发出一种桩的版本,叫做mock。mock就是一种具有特殊目的的替换类,包含行为验证以检查被测类对mock作出的正确的调用。测试驱动程序是一种软件组件或者测试工具,它替换的组件管理一个软件组件的控制和/或调用。

测试桩的责任之一是将参数返回调用的组件。这些值很少和被插桩的那些真实组件返回的一样,否则我们就不需要桩了。但是有时候它们必须满足特定的约束。

桩的最简单的作用是为输出指派一个常数。更复杂的方法可能是返回随机值,这个随机值来自表格查找(table lookup)或者让用户在执行的过程中进入返回值。从20世纪80年代开始就有具有自动建桩能力的测试工具了。更复杂的工具能够发现需要插桩的方法,并且询问测试人员桩应当具有哪种行为。一些工具收集了具有正确返回类型的对象的例子,并使得这些返回类型可以作为潜在的桩的返回值被利用。这是一种强大的方法,测试工程师只需要在必要的时候重载就可以了。从正式的软件组件的规格说明书中自动生成测试桩是可能的,但是在一些可用的工具中我们没有发现这个功能。程序员在执行他们自己的单元或者模块测试时也会生成他们自己的测试桩。

驱动程序的最简单形式就是类里有一个main()方法。出色的程序员常常在每个类中包含一个main()方法,包含能够执行类的简单测试的语句。如果类是一个ADT,main()测试驱动程序将会创建类的一些对象,添加值,读取值,以及使用类的其他操作。前面章节中提到的技术,比如顺序约束和基于状态的测试,可以在驱动程序中实现。

测试驱动程序可以包括硬编码值,或者从外部来源比如测试人员或者文件中找到这些值。已经有现存的工具来自动生成测试驱动程序。测试驱动程序和测试桩的生成器都包含在其他测试工具中。

6.2.2 类的集成测试顺序

当集成多个组件时,决定用什么顺序来集成和测试类或子系统是很重要的。类以各种方式相互依赖。一个类可能使用另一个类中定义的方法或变量,一个类可能继承另一个类,或

者一个类可能将另一个类的对象包含到它的数据对象中。如果类A调用了类B中定义的方法，而B不是可用的，那么我们需要为那些方法准备测试桩来测试A。因此，先测试B是有意义的，当测试A的时候我们可以用B的实际对象，而不是测试桩。

这称做类的集成测试顺序问题（CITO），而一般的目标是以需要最少的测试桩的顺序来集成和测试类的。创建测试桩被认为是集成测试的一个主要代价。如果类之间的依赖没有循环，那它们的集成就相当简单了。首先测试不依赖任何其他类的类。然后它们与只依赖于它们的类进行集成，并且对新加进来的类进行测试。如果类由用边来表示依赖的“依赖图”的节点来表示，这时候的方法就是对图进行拓扑排序（topological sorting）。

当依赖图中有了循环时，问题就变得更加复杂，因为最终我们会得到一个依赖于另外一个尚未集成和测试的类。这就是需要一些种类的测试桩的时候。比如，假设类A使用类B的方法，B使用类C的方法，而C包含了类A的一个对象。当这种现象发生时，集成测试人员必须通过选择循环中的一个类来首先进行测试，以“打破循环”。期望是选择一个导致额外工作（主要是创建桩）最少的类。

软件设计者可能会观察到类图中常常有很少或者没有循环，实际上，许多程序设计教科书强烈推荐不要在设计中包含循环。然而，在设计过程中加入类和关系是很普遍的，比如，为了提高性能或者可维护性。结果，在低层设计的结束或者实现的时候，类图经常包含循环。实际上测试人员需要解决CITO问题。

研究文献提出了针对CITO问题的大量解决方法。这仍然是一个活跃的研究领域，而这些解决方法还没有应用到商业工具中。

6.3 测试过程

许多组织把所有的软件测试活动推迟到开发的最后，在实现开始之后，甚至在实现结束之后。等到这么晚，测试以简练的方式草草结束，剩余的资源（时间和预算）已经不够了，解决以前的阶段中的问题需要时间和金钱，而测试人员没有时间来为测试做计划。开发人员只有时间来运行测试，常常是以一种形式化的态度，而没有计划和设计测试。关键的一点是创建高质量的软件，而那句古老的谚语“质量不是靠测试创造的”依然非常中肯。一个测试人员不可能在最后一分钟才出现而且还能把一个差劲的产品变好。高的质量从一开始就是过程的一部分。

本节讨论了如何将测试与开发结合起来，开发活动一开始测试活动也要开始，并且与开发阶段平行地进行。特定的活动，包括计划、动态测试以及受开发影响的活动，与每个传统的生命周期阶段相联系。这些活动可以由开发者或者独立的测试人员来执行，并且可以与开发过程中任何特定的开发阶段相联系。这些测试活动允许测试人员在软件开发过程中检测和阻止错误的发生。

在实现完成之后才开始测试活动的项目常常会产生出非常不可靠的软件。聪明的测试人员（以及测试等级4的组织）在软件开发的第一步就将一系列的测试计划和步骤加入进来，并且在接下来的所有步骤中继续进行。通过将软件测试活动与软件开发生命周期的所有阶段结合起来，我们可以使测试的有效性和高效性得到很大提高，并且通过这种方式来影响软件开发过程，这样更有可能开发出高质量的软件。

其他的教科书和研究文献包含了许多软件过程（瀑布模型、螺旋模型、原型进化模型、

极限编程等)。本节用到以下各个不同的阶段,但是没有设定它们的顺序,也没有把它们放置于某个特定的过程中。这样,本节中的建议对于用到的任何过程都是适用的。

1. 需求分析和规格说明书
2. 系统和软件设计
3. 中级设计
4. 详细设计
5. 实现
6. 集成
7. 系统部署
8. 操作和维护

任何开发过程都包含在各阶段之间信息交流、理解和传送。错误在任何阶段都可能产生,可能在信息处理的过程中产生,也可能在从一个阶段到另一个阶段的传送中产生。集成测试是尝试发现每个阶段中的错误,以及阻止这些错误波及其他阶段。同时,贯穿生命周期的集成测试提供了一种在各阶段间验证和追踪一致性的方法。测试不应被隔离成独立的阶段,而应在一个平行轨迹上,对所有阶段都有影响。

测试在每个阶段都有不同的目标,这些目标以不同的方式实现。然后每个阶段的这些测试子目标将达到确保高质量软件的总体目标。对于大多数阶段,测试行为可以分为3个宽泛的类别:测试活动——测试那个阶段产生的产品和工件;测试设计——利用那个阶段的开发工件或者测试前面阶段得来的工件来为测试最终的软件做准备;还有测试影响——利用开发或者测试工件来影响后面的开发阶段。

6.3.1 需求分析和规格说明书

软件需求和规格说明书包含了对软件系统的表面行为的完整描述。它为与开发过程的其他阶段进行交流提供了一种方式,并且定义了软件系统的内容和边界。表6.1总结了需求分析和规格说明书的主要测试目标和活动。

表6.1 需求分析和规格说明书的测试目标和活动

目 标	活 动
确保需求是可测试的	建立测试需求
确保需求是正确的	• 测试标准
确保需求是完整的	• 需要的支持软件
影响软件架构	• 每一级的测试计划
	• 构建测试原型
	阐明需求条目和测试标准
	开发项目测试计划

主要的测试活动目标是评估需求本身。应当对每个需求做出评价,来确保它是正确的、可测试的,并且需求同时还是完整的。针对这些,已经有很多种得到普遍验证和原型化了的方法描述出来。这些主题在别的地方已经得到了很好的描述,本书中就不明确地涵盖了。关键是应当在设计开始之前对需求进行评估。

主要的测试设计目标是为系统测试和验收活动做准备。应当写出测试需求,来陈述软件

系统的测试标准，而且要开发高级的测试计划，来概括描述测试策略。测试计划应当也包括每个阶段的测试范围和目标。这个高级的测试计划将会后面的详细测试计划中作为参考。测试需求应当描述在每个测试阶段需要支持软件。后面的测试应当满足测试需求。

主要的测试影响目标是影响软件架构设计。应当建立项目测试计划和代表的系统测试场景来证明系统满足了需求。开发测试场景的过程常常有助于检查出歧义和不一致的需求规格说明书。测试场景也能给软件架构设计者提供反馈，并且帮助他们开发出一种容易测试的设计。

6.3.2 系统和软件设计

系统和软件设计将需求区分为硬件或软件系统，并且构建了总体的系统架构。软件设计应当描述软件系统功能，所以它可以转化成可执行程序。表6.2总结了系统和软件设计过程中主要的测试目标和活动。

主要的测试活动目标是验证需求规格说明书和系统设计之间的映射。任何需求规格说明书的变化都应当反映在对应的设计变化上。这个阶段的测试应当有助于验证设计和界面的正确性。

表6.2 系统和软件设计过程中的测试目标和活动

目 标	活 动
验证需求规格说明书和系统设计之间的映射	证明设计和界面的正确性
确保可追踪性和可测试性	设计系统测试
影响界面设计	开发覆盖标准
	设计验收测试计划
	设计可用性测试（如果有必要）

主要的测试设计目标是准备验收和可用性测试。验收测试计划包括验收测试需求、测试标准和测试方法。并且，需求规格说明书和系统设计规格说明书应当保持可追溯和可测试，用于在以后的阶段里参考和了解变化。在系统和软件设计阶段进行测试也通过从以前的章节中选择测试标准来为单元测试和集成测试做准备。

主要的测试设计目标是影响用户界面的设计。可用性测试或者界面原型应当设计出来，弄清楚用户对界面的要求。当用户界面是系统不可缺少的一部分时，就要执行可用性测试了。

6.3.3 中级设计

在中级设计中，软件系统被划分为组件以及与每个组件相联系的类。设计规格说明书是为每个组件和类而写的。在大型软件系统中许多问题的出现是由于成分接口的错误连接。主要的测试活动目标是避免接口的错误匹配。表6.3总结了在中级设计阶段中主要的目的和活动。

主要的测试设计目标通过写测试计划来为单元测试、集成测试和系统测试做准备。单元和集成测试计划在这个级别上由接口和设计决策的信息来进行完善。为了给后面阶段中的测试做准备，应当获取或构建测试支持工具，像测试驱动程序、测试桩以及测试度量工具。

主要的测试影响目标是影响详细设计。6.2.2节中提到的类的集成和测试顺序（CITO）应当得到确定，以便于对详细设计产生合适的影响。

表6.3 中级设计过程中的测试目标和活动

目 标	活 动
避免接口的错误匹配	具体说明系统测试用例
准备单元测试	开发集成和单元测试计划
	构建或收集测试支持工具
	建议类的集成顺序

6.3.4 详细设计

在详细设计阶段，测试人员为模块写子系统规格说明书和伪代码。表6.4总结了详细设计阶段主要的测试目标和活动。在详细设计阶段的主要测试活动目标是保证当写好模块时，所有的测试材料都准备好进行测试了。测试人员应当同时为单元和集成测试做准备。测试人员应当提炼出详细的测试计划，为单元测试生成测试用例，以及为集成测试写出详细的测试规格说明书。主要的测试影响目标是影响实现以及单元和集成测试。

表6.4 详细设计阶段的测试目标和活动

目 标	活 动
当模块准备好的时候测试也准备完毕	创建测试用例（如果是单元）
	构建测试规格说明书（如果是集成）

6.3.5 实现

在软件开发过程中的某个时间，“新车开始上路”（rubber hits the road），程序员开始写下并编译类和方法。表6.5总结了实现过程中主要的目标和活动。

主要的测试活动目标是执行有效和高效的单元测试。单元测试的有效性很大程度上基于使用的测试标准和生成的测试数据。这个阶段执行的单元测试就像被在早期阶段中制定的单元测试计划、测试标准、测试用例和测试支持工具所指定一样。单元测试结果和问题应当恰当地保存和报告，用于进一步的处理。在这一点上设计者和开发者的责任变轻，以便他们可以帮助测试人员。

主要的测试设计目标是准备集成和系统测试。主要的测试影响目标是有效的单元测试可以帮助于保证早期的集成和系统测试。在单元测试中发现和修正bug代价要小得多也容易得多！

表6.5 实现过程中的测试目标和活动

目 标	活 动
有效的单元测试	创建测试用例值
自动生成测试数据	执行单元测试
	恰当地报告问题

6.3.6 集成

在前面讨论过的CITO问题是软件集成过程中的主要问题。表6.6总结了集成过程中主要的目标和活动。

主要的测试活动目标是执行集成测试。集成子系统需要的组件一通过单元测试，集成和集成测试就开始。在实际中，CITO问题可以用一种实用的方法来解决，那就是类一从单元测试中递交过来，就对它们进行集成。集成测试关注的是发现那些由于组件之间没有预期到的相互影响而导致的错误。

表6.6 集成过程中的测试目标和测试活动

目 标	活 动
有效的集成测试	执行集成测试

6.3.7 系统部署

表6.7总结了系统部署过程中主要的目标和活动。主要的测试活动目标是执行系统测试、验收测试和可用性测试。系统测试的特殊目的是比较软件系统和原始的对象，特别是，验证软件是否达到了功能需求和非功能需求。根据前几章介绍的标准，我们知道系统测试用例是从系统和项目测试计划中产生的，这些测试计划来自需求规格说明书和软件设计阶段。验收测试确保已完成的系统满足了客户的需要，并且需要客户参与完成。测试用例来自以前建立的验收测试计划和测试数据。可用性测试评估软件的用户界面，这也同样需要用户参与来完成。

表6.7 系统部署过程中的测试目标和活动

目 标	活 动
有效的系统测试	执行系统测试
有效的验收测试	执行验收测试
有效的可用性测试	执行可用性测试

6.3.8 操作和维护

软件发布（投递、部署或者其他）以后，用户有时会发现新的问题或者想要新的功能。如果软件发生了变化，需要对它进行回归测试。回归测试有助于确保更新了的软件产品实现它在更新前就已有的那些功能，以及新的和修改了的功能。表6.8总结了操作和维护过程中主要的目标和活动。

表6.8 操作和维护过程中的测试目标和测试活动

目 标	活 动
有效的回归测试	捕获用户问题 执行回归测试

6.3.9 总结

一个关键因素是，向一个开发过程灌输质量是基于个人的职业道德的思想。开发人员和测试人员都可以把质量放在首位。如果在某个过程中，测试人员不知道如何去测试，那么就不要构建这个过程。这有时会导致与时间驱动管理相冲突，但是即使你在争论中落败，你也会赢得尊敬。开发人员应当尽早开始测试活动，这一点很重要。这也有助于避免走捷径现象

的出现。几乎所有的项目最终都会面对走捷径的问题，而那样最终将降低软件的质量。战斗吧！如果你在争论中落败，你也能获得尊敬：把你的目标文档化，坚定你的立场，不要畏惧成为正确的一方！

测试工件的管理也是至关重要的。组织化的缺乏是失败的真正原因。对测试工件进行版本控制，使得它们易于使用并且定期更新它们。这些测试工件包括测试设计文档、测试、测试结果和自动化支持。对测试中基于标准的源代码进行追踪很重要，从而使得当源代码发生变化时，追踪到哪个测试需要变化成为可能。

6.4 测试计划

对于许多组织来说，一个主要的关注点是文档化，包括测试计划和测试计划报告。不幸的是，过多地关注文档可能导致这样一种状况：除了一堆无意义的报告，什么有用的东西也没有得到。那就是本书关注内容而不是形式的原因。测试计划内容的本质是测试如何创建，为什么要创建测试，以及它们如何执行。

然而，产生测试计划对于许多组织来说是一个基本要求。公司和客户经常会利用一些模板和大纲。我们不必去研究种类繁多的测试计划，而是看一下IEEE标准定义。不幸的是，这标准实在是太旧了（1983年），但是这依然是最广为人知的。在互联网上搜索一下，将提供你未曾使用过的测试计划和计划大纲。ANSI/IEEE标准829-1983这样描述测试计划：

“一个文档，它描述了有意识的测试活动的范围、方法、资源和时间安排。它识别了测试项目、需要被测试的功能、测试任务、谁来执行每个测试以及需要临时计划的各种风险事件。”

测试计划的几种不同的一般类型：

1. 说明“为什么”的任务计划。通常一个组织或组只有一个任务计划。任务计划描述了组织存在的原因，一般都很短（5~10页），是所有计划的最精炼的细节。

2. 说明“何事”和“何时”的策略计划。同样，一个组织通常只有一个策略计划，尽管一些组织为项目的各个类别都开发了策略计划。策略计划可以这样描述：“我们在进行单元测试的时候总是进行边界覆盖”和“集成测试总是由耦合驱动的”。策略计划比任务计划更详细也更长，有时达到20~50页或者更多。它们很少详细到可以直接用于开发人员或者测试人员的实践。

3. 说明“如何”和“谁”的战略计划。大部分组织对于一个项目采取总体的战略计划。战略计划是所有计划的最详尽的表示，并且通常是活跃的文档。也就是说，战略计划可能开始于一个内容的表，然后在产品或者产品开发的生命过程中不断地得到补充。比如，一个战略测试计划应当详细说明如何对每个单元进行测试。

下面是两个测试计划模板的大纲，只是作为例子给出。这些计划来自已经在网上发布了的无数的样本，所以并没有准确地代表某个组织。第一个是针对系统测试的，第二个是针对单元测试的战略计划。这两个都是基于IEEE829-1983标准的。

1. 目的

测试计划的目的是定义策略、测试范围、基本原理、测试退出和进入标准以及将要用到的测试工具。计划也应当包括管理信息，比如资源配置、人员分配以及时间安排。

2. 目标受众和应用

- (a) 测试人员和质量保证人员必须能够理解和实现测试计划。
- (b) 质量保证人员必须能够分析结果，并且对被测软件的质量写出推荐信，用于管理。
- (c) 开发人员必须能够了解是哪些功能要进行测试，测试的执行处于什么样的条件之下。
- (d) 市场人员必须能够了解产品是在什么样的配置（软件和硬件）条件下被测试。
- (e) 管理人员必须能够对进度掌握到这样的程度：知道什么时候执行测试，什么时候测试完成。

3. 可交付的东西

测试的结果就是得到以下可交付的东西：

- (a) 测试用例，包括输入值和期望结果。
- (b) 已满足了的测试标准。
- (c) 问题报告（产生的测试结果）。
- (d) 测试覆盖分析。

4. 包含的信息

每个测试计划应当包含以下信息。注意，这可以作为实际测试计划的大纲（经常如此），并可以成功地应用于大多数环境。

- (a) 介绍
- (b) 测试条目
- (c) 需要测试的功能
- (d) 不需要测试的功能（每周期）
- (e) 测试标准、策略和方法
 - 语法
 - 功能描述
 - 测试的参数值
 - 期望输出
 - 特定的例外
 - 依赖
 - 测试用例成功标准
- (f) 通过/失败标准
- (g) 开始测试的标准
- (h) 暂停测试的标准和重新开始的需求
- (i) 可提交的东西/状态交流文档
- (j) 硬件和软件需求
- (k) 确定问题严重性和改正问题的责任
- (l) 人员和培训的需求
- (m) 测试时间安排
- (n) 风险和偶然事件
- (o) 批准认可

上面的计划是一个非常普遍和高级的样式。下面这个例子是一种更为详细的样式，并且

更为适用于针对工程师的战略测试计划。

1. 目的

测试计划的目的在于描述所有测试活动的范围、方法、资源和时间安排。这个计划应当识别需要测试的条目，需要测试的功能，需要执行的测试任务，每个任务的人员责任，以及与这个计划相关的风险。

测试计划应当是一个可以被测试人员、管理人员和开发人员使用的动态文档。测试计划应当随着项目的发展而更新。在项目结束的时候，测试计划应当把活动文档化，并且能够作为一个媒介物，根据它各方人士对最终产品是否接受作出一个表示。

2. 大纲

测试计划具有以下结构：

- (a) 测试计划标识
- (b) 介绍
- (c) 测试参考条目
- (d) 需要测试的功能
- (e) 不需要测试的功能
- (f) 测试方法
- (g) 通过/失败标准
- (h) 暂停测试的标准和重新开始的需求
- (i) 测试交付件
- (j) 测试任务
- (k) 所需环境
- (l) 责任
- (m) 人员和培训需求
- (n) 时间安排
- (o) 风险和偶然事件
- (p) 批准认可

各节都是按照以上顺序排列的。如果需要的话，还会有附加的部分。如果某个部分的一些或者所有内容都在另外一个文档中，那个文档可以作为参考被列出。参考材料必须能够方便地取用。下面各节给出了各个部分的细节。

3. 测试计划标识

对这个测试计划给出一个特定的标识（名字）。

4. 介绍

给出这个软件的描述或者目标，所以测试人员和客户都能够清楚软件的目标以及测试的方法。

5. 测试参考条目

指出在测试中参考的条目，包括它们的版本/修订本和日期。如果可能，提供以下文档作为参考：

- (a) 需求规格说明书

- (b) 设计说明书
- (c) 用户指导手册
- (d) 操作指导手册
- (e) 安装指导手册
- (f) 分析图表, 包括数据流图等
- (g) UML或者其他模型化文档

6. 需要测试的功能

指出所有需要测试的功能和功能组合。指出与每个功能和功能组合相关联的测试设计。

7. 不测试的功能

指出所有不测试的功能和功能组合。最重要的是, 说明为什么。

8. 测试方法

对于每个主要的功能或功能组合的组, 指明为了确保这些功能组能够恰当地被测试而采用的方法。指明主要的活动、标准和使用的工具。

每个方法都要描述得足够详细来识别每个测试任务, 以及估计每个任务需要多长时间。

9. 通过/失败标准

用来确定每个测试条目是通过还是没有通过测试的量度标准。它是基于一个准则吗? 已知缺陷的数目是多少?

10. 暂停测试标准和测试重启需求

在特定的情况下, 测试必须停止并且软件被返回到开发人员那儿。指明暂停所有或者部分测试的标准。指明为了重新开始测试活动必须重复的活动。

11. 测试交付件

指出在报告中应当包括的文档。以下是备选的文档。

- (a) 测试计划
- (b) 测试设计说明书
- (c) 测试用例说明书
- (d) 测试过程
- (e) 测试日志
- (f) 测试故障报告
- (g) 测试总结报告
- (h) 测试输入数据和测试输出数据 (或者它们在什么地方)

12. 测试任务

指出准备和执行测试所必需的任务。指出各任务之间的所有依赖。

13. 所需环境

具体指出测试环境的必要的和希望的属性。这个说明应当包括:

- (a) 设备的物理特性, 包括硬件
- (b) 一切通信和系统软件
- (c) 使用模式 (单机、易变的基于网络的等)

- (d) 运行测试所需的任何其他软件或者供应品
- (e) 需要的测试工具
- (f) 任何其他的测试所需（例如出版物）以及如何得到他们

14. 责任

指明各组对测试的各个方面和修正缺陷的责任。另外，具体指明各组对提供测试参考项目和提供上面提到的那些测试所需环境的责任。

15. 人员和培训需求

具体说明在知识和技能方面对测试人员的要求。在适当和必需的时候指出培训的选择。

16. 时间安排

包括所有在软件项目时间表中确认了的测试里程碑。定义任何需要附加的测试里程碑。估计完成每个测试任务需要的时间，指明针对每个测试任务和里程碑的时间表。

17. 风险和偶然事件

指出测试计划的任何风险假设。例如，需要但不可用的专业知识。针对每一个风险制定备选计划。

18. 批准认可

指明所有必须批准这个计划的人的名字和头衔，并为他们留下签名字和日期的空白处。

6.5 识别正确的输出

本书的主要贡献是测试覆盖标准集。但是不管采用什么覆盖标准，迟早会有人想知道对于一个给定的程序，在给定输入的前提下，是不是执行正确。这是软件测试中的预言（oracle）问题。

oracle问题令人惊异地难以解决，而且它划定了一个可用方法的范围。本节描述了针对oracle问题的几个通用的方法。

6.5.1 输出的直接验证

如果你很幸运，你的程序会带有一个规格说明，并且这个规格说明很清晰，指出了对给定输入的输出是什么。比如，一个排序程序把它的输入按一个特定的顺序进行重排列。

对于一个给定输出的正确性进行人工评估经常是有效的，但也是代价高昂的。把这个过程自动化很自然会降低成本。如果可能的话，自动地直接验证输出是检查程序行为的最好的方法之一。下面是一个检查排序的概述。注意，检查算法不是另外一个排序算法。它不仅不同，而且不是特别简单。即写一个输出检查器可能会很难。

```

Input: Structure S
  Make copy T of S
  Sort S
  // Verify S is a permutation of T
  Check S and T are of the same size
  For each object in S
    Check if object appears in S and T same number of times
  // Verify S is ordered
  For each index i but last in S
    Check if S[i] <= S[i+1]
```

不幸的是，直接验证不总是可行的。考虑一个分析Petri网的程序，Petri网对有状态的过程建模是很有用的。这种分析的一个输出是处于任何一个给定状态的概率。观察给定的概率并且断言那是正确的是很困难的。毕竟那只是一个数。你如何知道是否所有的数字实际上是正确的？对于Petri网来说，最终的概率不容易联系回到输入Petri网。

6.5.2 冗余计算

当直接验证不可用时，后备计算可以被使用。例如，自动估算一个求最小值的程序的正确性，可以使用另外一个取最小值的实现，一个可靠的或者“黄金的”版本。这最终看起来像是一个循环，为什么应当相信一个实现更甚于另外一个呢？

让我们使这个过程形式化。假设被测程序被标记为 P ，而 $P(t)$ 是在测试 t 过程中 P 的输出。对于 P 的规格说明 S 也详细指出了输出 $S(t)$ ，我们经常要求 $S(t) = P(t)^\ominus$ 。假定 S 本身是可执行的，因此，允许我们将验证输出的过程自动化。如果 S 本身含有一个或者多个缺陷，一个普通的事件 $S(t)$ 可能就发生错误了。如果 $P(t)$ 恰好以同样的方式发生错误， P 的错误就探测不到了。如果 P 在一些测试 t 中以某种不同于 S 的方式发生了错误，那么不一致的地方就会被调查，至少有可能 S 和 P 中的错误都被发现。

潜在的问题是当 P 和 S 都有错误的时候，会导致不正确但同样的输出（因此不容易被注意到）。一些作者已经建议 S （oracle）应当独立于 P 来开发，以降低这种可能。从一个现实的角度来看，这样的独立开发是很难做到的。

而且，独立开发非常可能导致独立的错误。实验证据和理论论据显示共有错误的发生率实质上高于采取独立之后所期望的值。基本原因是一些输入比其他的“更难”正确，而恰恰是这些输入更可能触发跨各种实现的共有错误。

对于测试来说，以一种实现为标准来测试另外一种实现，仍然是有效、适用的技术。在工业中，该技术最常用在回归测试中，在这里规格说明 S 的可执行版本仅仅只是软件的早期版本。回归测试对于确认软件中的问题非常有效，而且应当成为一切重大的商业软件开发活动中的一个标准部分。

有时一个问题可能有不同的算法来解决它，对不同算法的实现是针对彼此进行检查的非常好的备选项，尽管共有错误问题仍然存在。比如，考虑一下检索算法。二分检索程序可以通过对比线性检索的结果很容易地被测试。

6.5.3 一致性检查

除了直接分析或者后备计算，还有一个选择就是一致性分析。一致性分析通常不完备，再次考虑Petri网的例子。给定一个假定的概率，当然可以说如果它低于或者高于单位值，那它就是错的。一致性分析同样可以是主观的。回忆一下第1章中针对错误的RIP（reachability、infection、propagation）模型。外部检查只能检查输出，所以影响必须波及将要发现的错误。

内部检查提高了识别只具有前两个属性（RI）的有缺陷的行为的可能性。对于程序员来说，需要特定的关系来维持内部结构，这太普遍了。比如，一个对象表示可能需要一个绝不会拥有重复对象的特定容器。检查这些“稳定的”关系是发现错误的非常有效的方法。在开

⊖ 如果 S 的说明不十分确定，那么 $S(t)=P(t)$ 的需求是不正确的。相反， S 应当看做是可能输出的一个集合，而且正确性约束是 P 产生了这些输出中的一个，即 $P(t) \in S(t)$ 。

发合同模型软件中受过训练的程序员能够在一般的开发中写出用于这种检查的代码。对于面向对象软件，这种对象通常围绕不变量（包括对象的抽象和它的表示）以及对象方法的前置条件和后置条件来组织。比如像断言这样的工具，如果为了性能的需要，在操作过程中可以有效地在测试过程中开启或者关闭这种检查。

6.5.4 数据冗余

一个给定输入的有效性的非常强大的方法是考虑程序如何处理它的输入。考虑正弦函数的计算，给定一个针对某个输入 x 的 $\sin(x)$ 计算，确定输入是否准确是很难的。与另外一种求正弦的实现进行比较是可行的办法，但是也可以采用其他技术。如果求正弦是可用的，那么余弦也是一样的。有 $\sin(x)^2 + \cos(x)^2 = 1$ 恒等式，对于一切 x 都成立。

这种最终检查在很多情况下都适用，但是在这种情况下不适用：即 $\sin(x)$ 和 $\cos(x)$ 恰巧以一种互补的方式出现了错误。比如，如果余弦的实现恰巧调用了正弦，那么我们就不会有什么收获。

一致性仍然是一个非常有用的方法。而且，它们经常在类中运用。比如，给容器增加一个元素，然后将这个元素从容器中移除，这经常对容器产生一个明确的影响。对于某些容器，比如包，结果完全没有变化。对于另外一些容器，比如集合，结果可能没有变化，也可能改变了一个元素，这取决于这一项是否原来就处于这个容器中。

更加有效的一致性是使用相同的程序，但是有不同的输入。再次考虑 $\sin(x)$ ，另一个恒等式是 $\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$ 。^①我们知道了输入的一个关系（即 $a+b=x$ ），输出的一个关系（ $\sin(x)$ 是一个用于求 a 和 b 的正弦的一个简单表达式）。这样的检查可以根据 a 的不同随机选择进行多次重复。那么即使是对正弦函数的最恶意的实现方法也只有几乎为0的机会来骗过这样的检查。这确实是强大的输出检查方法！

这种方法只有应用于运行良好的数学函数才能展现它的最为强大的形式，比如正弦函数。然而，这种方法也应用于许多其他种类的软件。考虑一个TCAS的实现，即部署在商用飞机上的交通碰撞和回避系统。这个系统的功能是给予飞行员指导，来更有效地避免潜在的碰撞。在“垂直解决办法”模式下，TCAS的输出，或者叫做“解决办法建议”，就可能是停在当前高度、攀升或者是下降。

TCAS是一个复杂的系统，它考虑了很多的因素，包括不同的飞机最近的各种位置，其他飞机上存在的互补TCAS程序，离地面的距离，等等。

为了运用数据冗余技术，假定我们将一些或者所有飞机上的TCAS软件改变一个很小的位置后将它放回。我们期望，在大多数情况下，解决办法建议应当没有变化。如果解决办法建议对某些紧密相关的输入表现不稳定，我们就得到有力指示，即飞行员可能不会对解决办法建议有很大的信心。返回到实验室，TCAS工程师可能希望对这样的输入投入格外的关注——可能甚至会将它们看做是错误。

这个被TCAS软件已经证明了的技术可以应用于任何输入空间具有某种连续概念的软件。在任何这样的系统中，说“临近的”输入是有意义的，并且在许多这类系统中，输出也可以期望以某些分段连续的方式变化。

① 如果我们愿意，我们可以将方法调用 $\cos(x)$ 改为 $\sin(\pi/2-x)$ ，但这不是十分必要。

6.6 参考文献注释

Binder[33]对回归测试做出了非常出色并且详尽的实用描述,其中他主张非自动化的回归测试等同于没有回归测试。Rothermel和Harrold公开发表了具有包容性、精确性、有效性和普遍性的回归测试框架[303],并且凭借经验评价了一种安全技术[153]。稍后由Li、Harman和Hierons[208],以及Xie和Notkin[361]完成的论文是回归测试文献的很好的起点。

桩和驱动程序的概念已经普及几十年了。它们早在20世纪70年代就在书中被讨论过了[80,104,165,249]。Beizer指出,桩的发明是有易错倾向和代价高昂的[29]。当前的工具,比如JUnit,使用“mock”这个词来作为桩的一种特殊形式[27]。

第一篇定义了CITO问题的论文是由Kung等人完成的[197]。他们展示了当类之间没有依赖的循环时,导出一个总体的顺序相当于基于类的依赖图对它们执行一个拓扑排序——一个广为人知的图论问题。对于存在依赖的循环的情况,他们提出了一种策略,识别强连通组件(SCC)并且移除联系,直到没有循环存在。如果有不止一种方法来终止循环,Kung等人随机选择其中一种。

大多数研究者[197, 48, 321, 329]通过测试桩的数目来评估CITO代价,而测试桩需要在集成测试中产生。这种方法假定写所有的桩是相同难度的。Briand等人指出桩的代价是不稳定的,并且开发了一种进一步的算法来解决CITO问题[44]。Malloy等人首次尝试在估计测试工作量的时候考虑测试桩的复杂度[223]。

Briand等人展示了对大多数继承成员函数的测试桩构建可以降低对父类测试桩构建的复杂度[47]。Abdurazik和Offutt基于耦合分析开发了一种新的算法,其中运用了更多的关于被插桩的类如果与其他类耦合来发现代价更小的排序的信息[3]。

关于测试过程和术语已被接受的定义的详细信息来源于IEEE标准[175]、BCS标准[317]、Hetzel[160]、DeMillo等人[100]、Kaner、Falk和Nguyen[182]、Dustin、Rashka和Paul[109]以及Copeland[90]的书。

Weyuker[341]写了一篇早期的论文,其中指出了预言(oracle)问题以及对付这种问题的各种方法。Meyer[241]和Liskov[355]都在关于契约模式的文章中谈到了如何表述可检查的断言。一些商业工具支持断言检查。

一些作者在关于容错的文章中主张构建多个版本,其中最出名的就是Avizienis[22]。多版本软件的可靠性限制首次由Knight和Leveson[188]进行了实验研究,然后由Eckhardt和Lee[110]以及Littlewood和Miller[214]进行了理论研究,并且在另外的文章中Geist等人[133]也进行了研究。多版本软件实际上在测试中比在容错中更能起到作用。如果一个程序的两个版本对于同样的输入采取了不同的行为,那么我们已经发现了一个好的测试,至少一个版本是错的。在实际中,把回归测试看做是多版本测试的一种,这是很有帮助的。

Blum和Kannan[38]以及Lipton[210]对于特定的良好定义的数学问题给予了数据冗余的理论处理方法,Ammann和Knight[18]提供了一种没有那么强大,但却更广泛地应用的方法。

第7章 技术的工程标准

本章讨论了怎样将第2章至第5章讲述的标准工程应用于几种不同类型的技术。经过软件测试领域许多文献的研究，这几种技术已很突出，现在已非常常见，并且很大部分的新应用都是基于这些技术建立的。在这里，有时候我们修改标准，有时候只是简单讨论怎样建立已有标准适用的模型。有些应用，例如Web应用和嵌入式软件，往往要求有极高的可靠性。因此，测试是这些应用成功的关键。本章从测试的角度解释了这些技术的区别，总结了已存在的使用这些技术用于软件测试的方法。

面向对象技术在20世纪90年代中期已经发展成为一种优秀的软件技术，研究人员花了相当多的时间来研究面向对象所特有的问题。在前几章里已经讨论了面向对象软件相关的许多问题，包括在第2章所讲的应用图表标准（graph criteria）的若干方面，第5章的集成变异和第6章的CITO问题。本章探讨了类的使用是如何影响测试的，重点在于研究人员最近面临的挑战。大部分解决方案并没有能通过自动化测试工具来实现。所遇到的最大的挑战是测试那些由于使用继承、多态和动态绑定所带来的问题。

一个开发技术和测试都非常活跃的领域是Web应用和Web服务。许多Web软件在本质上是面向对象的，但是Web允许使用很多“interesting”[⊖]结构，而这些结构要求测试人员调整他们的技术和标准。Web应用和服务一个很有趣的方面是它们必须工作得非常好——运行环境很具挑战性并且用户不能容忍出错。Web应用还有严格的安全方面的需求，这将在第9章讨论。Web应用使用一种特殊的GUI（在浏览器中运行HTML），测试通用的GUI会带来额外的复杂性。本章中有的想法仍处于研究和开发阶段，所以可能不能用于实际应用。

最后，本章将讨论测试实时软件和嵌入式软件所遇到的问题。之所以把它们放在一起是因为很多系统同时遇到这两类问题。随着嵌入式软件出现在电子设备中，嵌入式软件的数量增长非常迅速。很多这类系统也有着安全标准方面的需求，该主题将在第9章讨论。

7.1 测试面向对象软件

面向对象语言着重于定义软件中的抽象。抽象（例如抽象数据类型）为应用领域中的概念建立模型。这些抽象通过类来实现，类表现了用户定义的包括状态和行为的类型。这种抽象方法具有很多优点，但是也改变了测试的实施。最重要的因素是面向对象软件把软件的复杂性从单元和方法的算法转到怎样连接软件组件上。因此，我们需要更着重于集成测试而不是单元测试。

另一个因素是面向对象软件中各组件之间的关系变得很复杂。继承和集合（特别是跟多态结合使用的时候）的组合关系产生了新的错误类型，这就要求有新的测试方法。这是因为类和组件集成的方法在面向对象语言中是不同的。

⊖ 不同的读者可能会对文中的“interesting”一词作不同的理解，对于研究者可以理解为这里面有很多有趣的问题需要解决，作为开发者，特别是管理人员，应该将“interesting”理解为对软件质量的时效性挑战和威胁。

面向对象语言使用类（数据抽象）、继承、多态和动态绑定来支持抽象。通过继承创建的新类型是已有类型的后代。如果一个子类拥有新的方法并且没有覆盖父类中的任何方法，则说该类扩展（extend）了它的父类。这种新方法称为扩展方法。如果一个子类提供了被覆盖方法里所没有的行为，不再调用被覆盖方法，并且它的行为与被覆盖方法的行为在语义上保持一致，则说该子类精化（refine）了父类。

程序员使用两种继承：子类型（subtype）和子类（subclass）。如果类B使用子类型继承类A，那么可以使用B的实例代替A的实例并且依然满足类A的任何客户，这就是替换原则（substitution principle）。换句话说，B和A有“是一个”（is-a）的关系。例如，椅子“是一个”家具的特例。子类继承允许后代类使用祖先类的方法和变量而并不需保证后代类的实例满足祖先类型的规范。虽然对于测试人员来说哪一种继承是合适的存在激烈的争议，但是专业的程序员两者皆使用。在子类型继承的情况下，测试人员应该着重于验证替换原则是否依然有效。由于缺乏坚定有力的子类型继承指导原则，使得测试人员很容易找到错误。

这些抽象主要影响了组件的集成。如果类B继承了类A，并且A和B都定义了方法m()，那么m()被称为多态方法。如果对象x被声明为类型A（在Java里用“A x;”表示），那么执行x可以有两种实际类型：类型A（“x = new A();”）或类型B（“x = new B();”）。当调用一个多态方法（例如“x.m();”），实际执行的版本取决于对象的当前实际类型。方法的可执行的版本集合称做多态调用集合（polymorphic call set, PCS）。在这个例子中，x.m()的PCS是{A::m(), B::m()}。

7.1.1 面向对象软件测试特有的问题

有些测试问题是面向对象软件所特有的。一些研究人员声称传统的测试技术对面向对象软件不起作用，有时候甚至出错。方法变得更小也更简单，所以基于路径的测试技术发挥的作用更小了。如前所述，继承、多态和动态绑定引入了特殊的问题。执行路径不再基于静态声明的类型而是动态类型，并且动态类型直到执行的时候才可知的。

测试面向对象软件的时候，通常把类看做是基本的测试单元。针对类有以下4种测试级别。

1. 方法内测试：为单独的方法构建测试（即传统的单元测试）
2. 方法间测试：同时对类中的多个方法进行测试（即传统的模块测试）
3. 类内部测试：针对单个类构建测试，通常顺序调用类里边的方法。
4. 类间测试：同时测试一个以上的类，通常着重于观察它们之间的交互（即继承测试）

早期的面向对象测试研究关注方法间测试和类内部测试。后来的研究重点在测试单个类与其用户的交互和面向对象软件系统级别的测试。在方法间测试和类内部测试级别并不能处理与继承、多态和动态绑定相关的问题。这些问题需要对发生继承和多态的多个类进行类间测试。

大部分面向对象的测试关注于以下两个问题之一。一个是类应该按怎样的顺序进行集成和测试，第6章已讨论了CITO问题。另一个是选择测试的技术和覆盖标准的发展。这些覆盖标准就是対在前几章所讨论的一个或多个标准的细化。

7.1.2 面向对象的错误类型

面向对象软件工程师的一个很艰巨的任务之一就是要把在继承、多态和动态绑定中发生

的交互可视化，这些交互通常都很复杂。这种可视化假定类把状态信息封装在状态变量集合里，并且类拥有一个由使用这些状态变量的方法实现的行为的集合。

例如，让我们看看如图7.1所示的UML类图和代码块。如图所示，V和X扩展了W，V覆盖了方法m()，并且X覆盖了方法m()和n()。减号(“-”)指明该属性是私有的，加号(“+”)指明该属性是非私有的。o的声明类型为W，但是在第10行，实际类型可能是V或W。由于V覆盖了方法m()，所以m()的执行版本依赖于方法f()的输入参数。

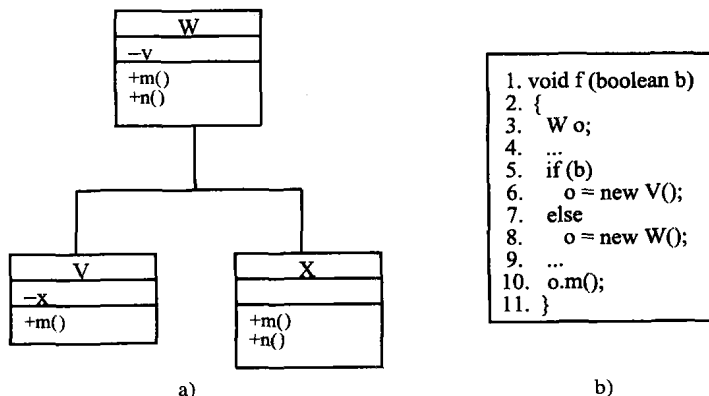


图7.1 用UML描述类层次结构图

为了说明方法覆盖和多态带来的问题，我们再看看图7.2a。根类A包含4个状态变量和6个方法。这些状态变量都是被保护的，对于A的后代(B和C)是可用的。B声明了一个状态变量和3个方法。C只声明了3个方法。图中的箭头指出了方法覆盖：B::h()覆盖A::h()、B::i()覆盖A::i()、C::i()覆盖B::i()、C::j()覆盖A::j()、C::l()覆盖A::l()。图7.2b的表在方法中状态变量的定义和使用，问题由调用A::d()开始。这个小例子有相当复杂的交互，有可能产生非常难以建模、理解、测试和调试的问题。

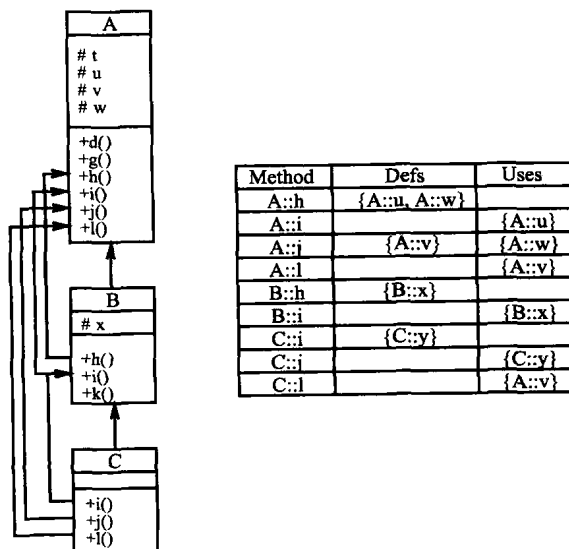


图7.2 与多态相关的数据流异常

假使对象o是A的一个实例 ($o = \text{new } A();$), 一个调用通过对象o调用了 $A::d()$ ($o.d()$), $A::d()$ 调用了 $A::g()$, $A::g()$ 调用 $A::h()$, $A::h()$ 调用 $A::i()$, $A::i()$ 最终调用了 $A::j()$ 。变量 $A::u$ 和 $A::w$ 定义后由 $A::i()$ 和 $A::j()$ 使用, 没出现什么问题。

现在假设对象o是B的一个实例 ($o = \text{new } B();$), 并对 $d()$ 进行调用 ($o.d()$)。这时候, 类B的 $h()$ 和 $i()$ 被调用, $A::u$ 和 $A::w$ 未赋值, 从而导致对 $A::j()$ 的调用产生数据流异常。

使用Yo-Yo图可视化多态

要确定方法的哪个版本将被执行和方法有哪些版本可以执行, 这对开发人员和测试人员来说都很复杂。执行会在继承层次间上下“跳动”, 这种现象叫做Yo-Yo效应。Yo-Yo图是根据拥有一个根和多个后代的继承层次定义的, 它展示了每个后代的所有新的、继承的、被覆盖的方法。方法的调用用从调用方指向被调用方的箭头表示。在Yo-Yo图里每个类都被指定一个层次, 表示了如果一个对象具有该层次的实际类型所做的实际调用。粗箭头表示实际调用, 细箭头表示该调用因为覆盖而不能实行。

再看看如图7.2所示的继承层次结构。假设实现类A的时候, $d()$ 调用 $g()$, $g()$ 调用 $h()$, $h()$ 调用 $i()$, 然后 $i()$ 调用 $j()$ 。接下来, 即使实现类B的时候, $h()$ 调用 $i()$, $i()$ 调用其父类 (也就是类A) 的 $i()$, $k()$ 调用 $l()$ 。最后, 再假设实现类C的时候, $i()$ 调用其父类 (也就是类B) 的 $i()$, $j()$ 调用 $k()$ 。

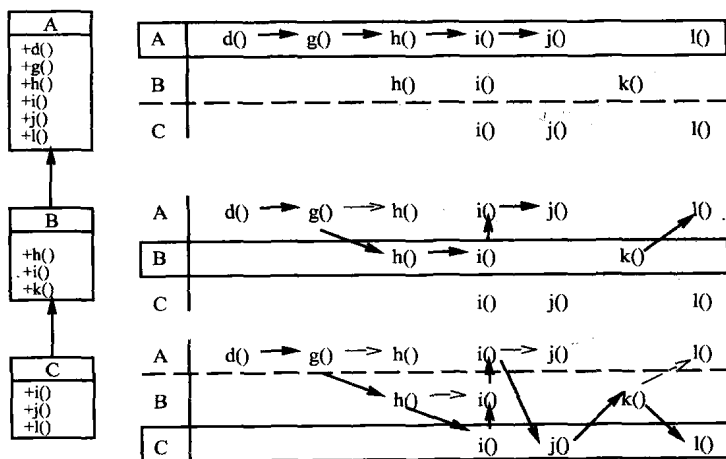


图7.3 当对象具有多种实际类型时对 $d()$ 的调用

图7.3是上述情况的Yo-Yo图, 说明了实际类型分别为类A、B和C的实例调用方法 $d()$ 时产生的调用顺序。图的最顶层的假使实际类型为A的对象调用 $d()$, 这时调用序列是简单的。第二层显示了实际类型为B的对象的情况, 此时变得复杂一些了。当 $g()$ 调用 $h()$ 时, 在B中定义的 $h()$ 被执行 (由 $A::g()$ 指向 $A::h()$ 的细箭头强调了 $A::h()$ 未执行), 然后执行 $B::i()$ 、 $A::i()$ 和 $A::j()$ 。

当对象的实际类型为C的时候, 我们就可以知道“Yo-Yo”这种称呼是怎么得来的了 (译者注: Yo-Yo有溜溜球之意)。执行路径从 $A::g()$ 、 $B::h()$ 到 $C::i()$, 然后从 $B::i()$ 后退至 $A::i()$, 再到 $C::j()$ 、 $B::k()$, 最终到达 $C::l()$ 。

这个例子说明了在面向对象软件中由于方法覆盖和多态所引入的复杂性, 正因为这种复杂性使测试变得更困难。

与继承相关的错误和异常的分类

继承使得开发人员更富创造性，更有效率，并且帮助开发人员重用各种已存在的软件组件。不幸的是，继承也带来了大量的异常和潜在错误，很多证据显示这些异常和错误很难检测、诊断和修正。表7.1总结了继承和多态导致的错误类型，这基本适用于所有编程语言，虽然有时因为语言得不同而使得错误的细节不太一样。

表7.1 跟继承与多态相关的错误和异常

缩写	故障/异常
ITU	类型不一致使用（上下文互换）
SDA	状态定义异常（可能违反后置条件）
SDIH	状态定义不一致（由于状态变量隐藏）
SDI	状态定义错误（可能违反后置条件）
IISD	状态定义间接不一致
ACB1	异常构造行为(1)
ACB2	异常构造行为(2)
IC	不完全构造
SVA	状态清晰度异常

在前面已经指出，面向对象错误不同于非面向对象软件中的错误。接下来的讨论假设异常或者错误是在使用祖先类的一个实例上下文的多态中产生的。因此，我们假设后代类的实例可替换祖先类的实例。

在**类型不一致错误**（ITU）中，后代类没有覆盖任何继承的方法，因此不存在多态行为。在期望祖先类 T 的实例的时候使用后代类 C 的任何一个实例，该实例的行为只能跟 T 的实例的行为极其相似。也就是说，只有 T 的方法可以被使用。由于 C 的实例被当做 T 的实例使用，其他在 C 中声明的方法被隐藏。然而，异常行为依然有可能出现。如果 C 的实例在多重上下文中使用（即通过强迫的方式，比如首先作为 T ，接着作为 C ，然后又作为 T ），如果 C 有扩展方法，则异常行为可能发生。在这种情况下，一个或多个扩展方法调用 T 的一个方法或者直接定义从 T 中继承的状态变量。如果导致继承状态不一致，则将会发生异常行为。

图7.4展示了一个类层次结构图。 $Vector$ 类是一个序列数据结构，支持对其元素的直接获取。 $Stack$ 类使用从 $Vector$ 继承的方法来实现栈。顶部的表总结了每个方法的调用，底部的表总结了 $Vector$ 的状态空间的定义和使用（分别用“d”和“u”表示）。

方法 $Stack::pop()$ 调用 $Vector::removeElementAt()$ ， $Stack::push()$ 调用 $Vector::insertElementAt()$ 。很明显，这两个类有着不同的语义。既然 $Stack$ 的实例只作为一个 $Stack$ 使用，这样不存在行为方面的问题。同样，如果 $Stack$ 的实例仅作为一个 $Vector$ 使用，这样也不存在行为方面的问题。然而，如果同样一个对象有时候被当作 $Stack$ 有时候被当做 $Vector$ 来使用，异常行为就可能发生。

表7.2中的代码段说明了这个问题。3个元素压入一个 $Stack$ s 中，然后调用 $Vector$ 的方法 $g()$ 。不幸的是， $g()$ 从栈的中部移走了一个元素，这会破坏它的语义。更糟的是，调用 $g()$ 后的3个弹出操作不再生效。错误出现在第14行，也就是第一次调用 $Stack::pop()$ 时。在这里，从栈中移走的元素不是最后添加的元素，因此栈的完整性约束被破坏了。第3次调用 $Stack::pop()$ 时，程序运行就会失败，因为此时栈是空的。

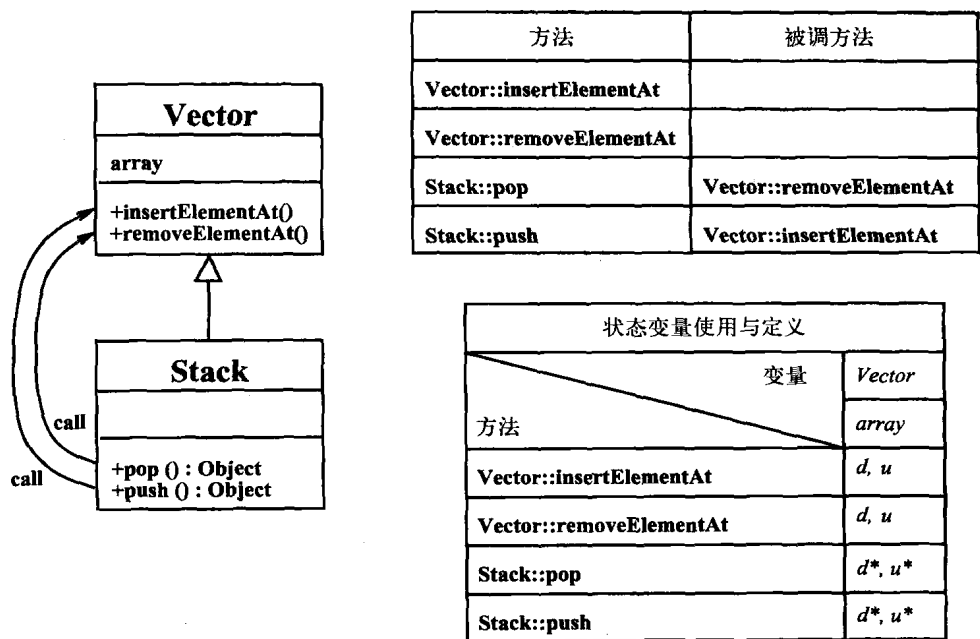


图7.4 ITU：后代类没有覆盖方法

表7.2 IUT：不一致类型使用的代码示例

1 public void f (Stack s)	13 s.pop();
2 {	14 s.pop();
3 String s1 = "s1";	15 // 哎呀！堆栈是空
4 String s2 = "s2";	16 s.pop();
5 String s3 = "s3";	17 ...
6 ...	18 }
7 s.push (s1);	19 public void g (Vector v)
8 s.push (s2);	20 {
9 s.push (s3);	21 // 移去最后一个元素
10	22 v.removeElementAt (v.size()-1);
11 g (s);	23 }
12	

在状态定义异常错误（SDA）中，后代类的状态交互与其祖先类不一致。后代类实现的泛化方法应该使得祖先类的状态与祖先类的被覆盖的方法保持状态一致。为了实现这一点，后代类的细化方法的状态交互必须与其覆盖的公共方法一致。从数据流的视角来看，这意味着泛化方法必须对继承的状态变量做与覆盖方法同样的定义。如果不这样做，则可能存在潜在的数据流异常。至于是否实际存在异常则取决于对祖先类有效的方法调用顺序。

图7.5展示了类的层次结构图和定义与使用表。W是父类，X和Y是其后代。如表所示，W定义了方法m()和n()。假设有效的方法调用顺序为W::m()、W::n()。如表所示，W::m()定义了状态变量W::v，W::n()使用了该变量。现在来看类X及其泛化方法X::n()，它同样使用了变量W::v，与被覆盖方法和上面的方法顺序一致，因此X与W的状态交互时不存在不一致现象。

现在来看类Y及其方法Y::m()，它通过泛化覆盖了W::n()。注意Y::m()并没有像W::m()一样

定义 $W::v$, 但是定义了 $Y::w$ 。现在, 对于状态变量 $W::v$, 方法序列 $m(); n()$ 存在数据流异常。当这个方法顺序由 Y 的实例调用的时候, $Y::w$ 首先被定义 (因为执行 $Y::m()$), 但是接下来 $W::v$ 被方法 $X::n()$ 使用。因此, 在实现 $X::n()$ 的时候所做的假设不成立 (该假设为 $m()$ 先于 $n()$ 调用, 从而定义了 $W::v$), 出现了数据流异常。在这个特殊的例子里, 因为类型为 Y 的实例使用之前没定义 $W::v$ 而导致了错误的出现。一般来说, 程序运行有可能并不失败, 而只是创建了一个错误的状态。

在由隐藏状态变量而引发的状态定义不一致错误 (SDIH) 中, 引入一个本地状态变量会导致数据流异常。如果类定义的本地变量 v 与继承变量 v 同名, 则继承变量在后代类的作用域中被隐藏了 (除非以 *super.v* 等方式特意声明)。对 v 的引用将引用后代类中的 v 。如果所有继承方法都被覆盖并且没有其他方法隐式引用继承变量 v , 那将没什么问题。然而, 这只是继承中的特例而不是规则。一般情况下有些方法未被覆盖, 如果继承的状态变量 v 被本地定义隐藏, 而定义该变量的方法在后代类中被覆盖, 那么则可能存在数据流异常。

再来看看如图 7.5 所示的类层次结构图。假使类 Y 中定义的本地状态变量隐藏了继承变量 $W::v$, $W::m()$ 定义了 $W::v$, $Y::m()$ 定义了 v 。对于方法调用序列 $m(); n()$, W 与 Y 之间存在关于 $W::v$ 的数据流异常。

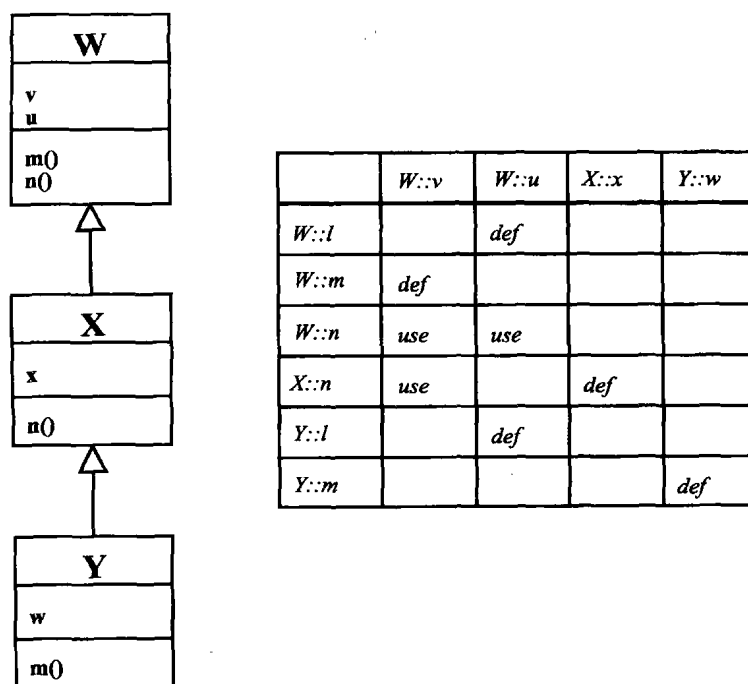


图 7.5 SDA、SDIH: 状态定义异常

在状态定义错误 (SDI) 中, 覆盖方法定义了与被覆盖方法相同的状态变量 v 。如果对状态变量 v 的计算在覆盖方法中在语义上不等于被覆盖方法, 那么祖先类中的基于状态的行为子序列将被影响, 后代类的行为将与祖先类不同。虽然这不一定是一种数据流异常, 但潜在行为异常的可能。

在状态定义间接不一致错误 (IISD) 中, 后代类增加了一个扩展方法, 该方法定义了一个继承状态变量。例如, 让我们看看如图7.6a所示的类层次结构图, 类Y声明了状态变量x和方法m(), 后代类D声明了方法e()。e()是一个扩展方法, 因为其对继承方法不可见, 所以不能被继承方法 (T::m()) 直接调用。然而, 如果某个继承方法被覆盖, 那么覆盖方法 (如图7.6b所示的D::m()) 就能调用e(), 从而影响祖先类中与被覆盖方法语义上不相等的状态 (如例中的变量T::y), 引入数据流异常。错误的发生取决于e()所定义的状态变量和祖先类在e()定义的变量上有哪些基于状态的行为。

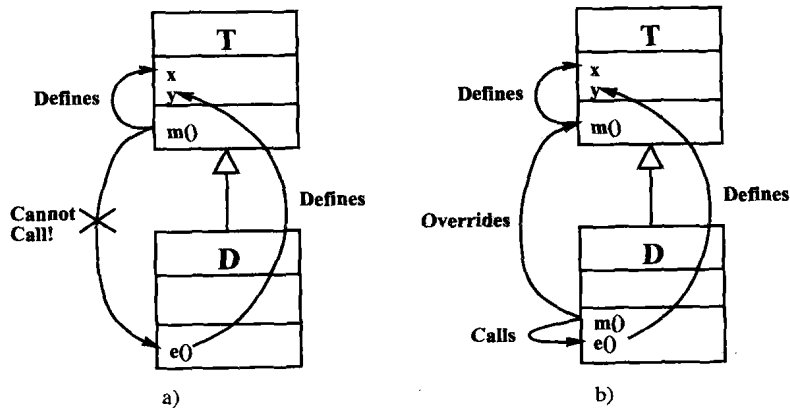


图7.6 IISD: 状态定义间接不一致定义

在构造异常行为错误版本1, (Anomalous construction behavior fault, ACB1) 中, 祖先类C的构造方法调用了本地定义的多态方法f()。因为f()是多态的, 所以后代类D可以覆盖它。如果D覆盖了该方法, 则C的构造函数调用f()时将执行D中的f(), 而不是C中定义的版本, 如图7.7左半部分的类层次结构图所示。类C的构造方法调用C::f()。类D包含定义本地状态变量D::x的覆盖方法D::f()。因为D::f()与C的状态变量没有交互, 所以类D和类C没有明显交互。然而C类通过构造函数调用C::f()从而与D的状态变量交互。在大多数面向对象语言 (包括Java和C#) 中, 调用多态方法的构造方法执行最接近正在创建的实例的方法。对于图7.7中的类C, 最接近C的方法f()的版本就是类C自己声明的, 当C的实例被创建的时候就执行它。对于类D, 最接近的版本是D::f(), 当创建D的实例时, 在C的构造方法对f()的调用实际上执行了D::f()。图7.7右半部分的Yo-Yo图说明了这一点。

如果D::f()使用D状态空间里定义的变量, 在图7.7中显示的行为的结果可能是数据流异常。根据构造方法的顺序, 此时D的状态空间还未构造。是否存在异常取决于声明f()使用的变量的默认初始化值。如果D::f()的假设或者前提条件没有在构造方法前得到满足, 那么很可能引发错误。

在异常构造行为错误版本2 (Anomalous construction behavior fault, ACB2) 中, 祖先类C的构造方法调用了本地定义的多态方法f()。如果f()在后代类D中被覆盖并且其覆盖方法使用从C中继承的状态变量, 那么有可能发生数据流异常。如果D::f()使用的状态变量还没有在C::f()中正确构建, 异常就会发生。这取决于D::f()使用的变量集, C中状态的变量的构建顺序, 以及C的构造函数对f()的调用顺序。注意, 编写类C的程序员并不是一直事先知道f()的哪

个版本将会被实际执行，或者执行的版本是基于哪些状态变量。因此，在构造方法内调用多态方法是不安全的，并且引入非确定性到构造过程中。对ACB1和ACB2皆是如此。

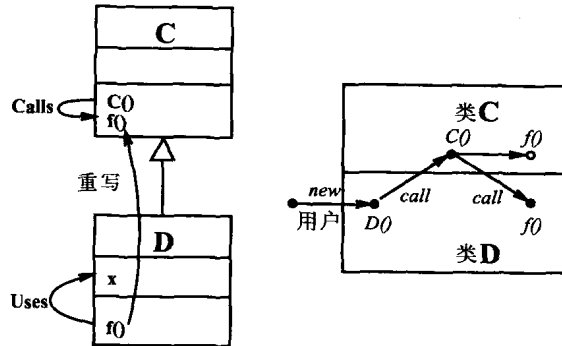


图7.7 ACB1: 构造器行为异常

在不完全构造错误 (Incomplete (failed) construction fault, IC) 中，对象的初始状态并没定义。在某些编程语言中，构造之前的类的状态空间的变量值未定义。在C++中是这样的，而Java不是。构造器设置了初始状态条件并且状态不会因新实例的产生而改变。构造器通常有语句来定义每个状态变量。在某些情况下（取决于编程语言），默认的或者隐含的初始化可能就足够了。在其他情况下，当构造器结束时，实例的状态应该被很好地定义。错误的产生有两种方式。第一种，构造过程可能对特定状态变量赋予错误的初始值；第二种，特定状态变量的初始化被忽略。在构造器和构造过程结束后第一次使用该变量的方法间将存在数据流异常。

表7.3中的代码段显示了一个不完整构造器。类AbstractFile包含了未被构造器初始化的状态变量fd。类AbstractFile的设计意图是后代类在变量fd被使用前对其定义，该意图由后代类中SocketFile的方法open()完成。如果所有后代类实例化时定义了fd，并且没有方法在定义fd之前使用它，则不会出错。然而，如果任一条件未满足，程序将会出错。

表7.3 IC: 状态变量fd的不完全构造

1	Class abstract AbstractFile	14	Class SocketFile extends AbstractFile
2	{	15	{
3	FileHandle fd;	16	public open()
4		17	{
5	abstract public open();	18	fd = new Socket (...);
6		19	}
7	public read() {fd.read (...); }	20	
8		21	public close()
9	public write() {fd.write (...); }	22	{
10		23	fd.flush();
11	abstract public close();	24	fd.close();
12	}	25	}
		26	}

设计者的意图是后代类提供必要的定义，而在类AbstractFile的方法read()和write()中存在关于fd的数据流异常。这两个方法都使用fd，如果其中任意一个在构造后立即被调用，那么将会出错。因为在设计的时候不知道fd将被绑定的实例的类型，也不知道fd是否将被绑定，从而

*AbstractFile*中引入了非确定性。假使*AbstractFile*的设计者还设计和实现了类*SocketFile*（见表7.3）。设计者希望通过类*SocketFile*的设计消除在类*AbstractFile*中存在的数流异常。但非确定性并不能被消除，并且当新的后代类没提供必要的定义时，错误依然可能产生。

在状态清晰度异常错误（State visibility anomaly fault, SVA）中，祖先类的状态变量被声明为私有，多态方法 $A::m()$ 定义了 $A::v$ 。如图7.8a所示，*B*是*A*的后代，*C*是*B*的后代。*C*覆盖了 $A::m()$ ，而*B*没有。因为 $A::v$ 是私有的， $C::m()$ 不能通过直接定义 $A::v$ 来与*A*的状态交互， $C::m()$ 必须调用 $B::m()$ ，然后 $B::m()$ 调用 $A::m()$ 。因此， $C::m()$ 并不直接导致数据流异常。一般都会存在私有状态变量，唯一避免数据流异常的方法就是后代类的每个覆盖方法调用祖先类的被覆盖方法。如果上述过程出错，则肯定会导致类*A*的状态和行为错误。

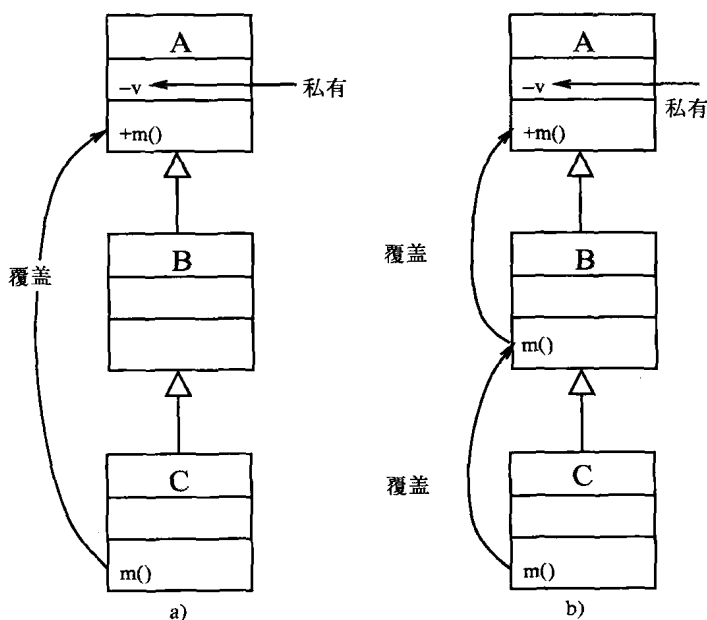


图7.8 SVA: 状态清晰度异常

测试继承、多态及动态绑定

通过扩展在2.4.2节中提出的耦合（coupling）的概念，数据流测试可以用于面向对象软件的测试。我们重新回顾概念：耦合定义（coupling-def）即在一个方法中某一变量的最近一次定义，而对这个变量的访问可能发生在另一个方法中，这种使用称为耦合使用（coupling-use）。一条耦合路径指的是在两个程序单元中存在一条从耦合定义到耦合使用的路径。该路径必须是定义清纯的（def-clear）。

在使用了继承、多态和动态绑定的程序中来识别定义、使用和耦合是非常复杂的，因此应该认真考虑面向对象程序的语义。

在接下来的定义中，*o*是一个变量，它是一个实例对象的引用，指向包含了一个类型的实例的内存区域。引用*o*指向的实例只能通过*o*的基类或者*o*的类型的子类实例化得到。因此，在如下的Java源代码语句“*A o* = new *B*();”中，*o*的基类或声明类型是*A*，它的实例化类型或者实际类型是*B*。*B*必须是*A*的子类。

图7.9表示一个以W为根节点类图。类图中的所有成员都共享一些在每个类的祖先类中定义公共的属性。一个类的每个类型定义是定义了一个类型族。族的每个成员包含层次结构的基类和该基类后代的所有类型。图7.6b显示了图7.9a中层次结构所定义的类型族。

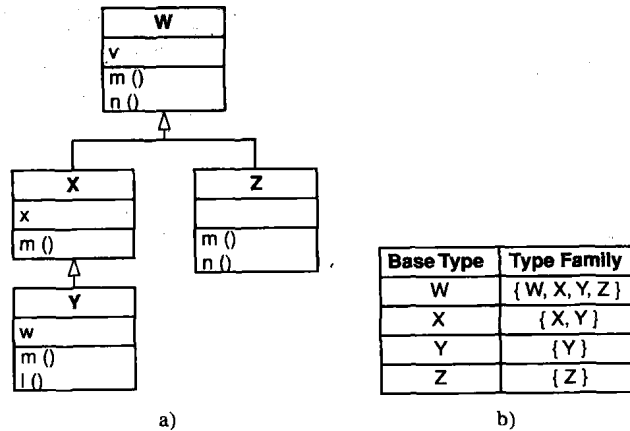


图7.9 样本类层次结构a)和关联类型族b)

面向对象的数据流分析中最困难的部分是面向对象程序的静态非确定性，即多态和动态绑定。多态允许一个函数调用指向多个方法，具体指向取决于对象引用的实际类型，动态绑定意思是指在程序执行之前我们不知道具体调用了哪个方法。

当对象的状态变量v被定义，那么称该对象实例是已定义的。方法m()定义v，称为间接定义(indirect definition或i-def)。类似地，间接使用(indirect use或i-use)是指方法m()引用了v的值。

当我们去查找对象引用的调用点的间接定义与使用的时候，不仅要考虑方法调用的方法，而且要考虑可能要执行的方法集合，也就是之前定义的多态调用集合(PCS)。幸运的是，潜在调用的方法是有限的并且可以被静态地决定。这种分析用到了满足集合(satisfying set)的概念。

定义7.52 满足集合：对用对象引用中的多态方法m()，重载了方法m()的方法集合加上m()自身构成了满足集合。

图7.10是基于图7.9。假设W包含一个返回W实例的方法FactoryForW()。图7.10a显示了一个将W的实例绑定为对象o的控制流图片段。这是一个局部定义，通过调用方法FactoryForW()便得到了一个对象的引用o。图7.10b的表中显示了W.m()定义v，从这里我们知道在节点2通过o.m()产生了一个间接定义。因此，我们可以知道，任何W的实例o，通过调用m()方法都会导致实例o的一个间接定义的出现。在方法m()中没有间接使用。在图7.10b的表中显示了所有W的类型集合的任何实例所包含的间接定义和使用。节点3不包含任何定义，但是有一个v的间接使用。

对于节点2的方法m()的满足集合是{W.m(), X.m(), Y.m(), Z.m()}和间接定义集合包含如下的有序对：

$$i_def(2, o, m()) = \{(W.m(), \{W.v\}), (X.m(), \{W.v, X.x\}), \\ (Y.m(), \{W.v, Y.w\}), (Z.m(), \{W.v\})\}$$

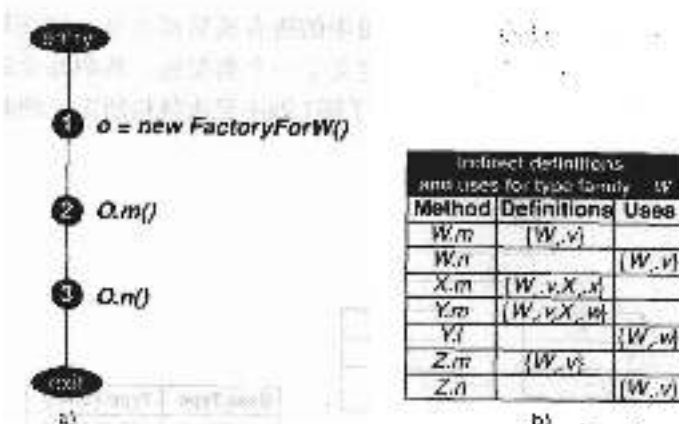


图7.10 控制流图片段a)和关联的定义和使用b)

在图7.10中的每个定义和使用对指示了一个方法的满足方法以及方法定义的状态变量的集合。在这个例子中 $X.m()$ 类 W 中的状态变量 v 以及 X 中的 x 。

从图7.10b中我们可以看到，节点2的间接使用集合为空，因为 $m()$ 方法的满足方法集合没有引用任何状态变量。但是，在节点3有两个拥有非空的间接使用集合的方法，这两个方法可以满足 $o.n()$ 的调用（这两个方法的间接定义集合为空），它们能够产生如下的间接使用集合：

$$i_use(3, o, n()) = \{(W.n()), \{W.v\}\}, \{(Z.n()), \{W.v\}\}$$

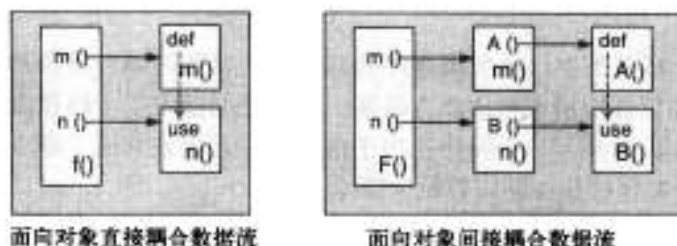
对多态路径的分析

定义使用对、耦合路径对于面向对象的程序来说是非常复杂的。我们先给出如下定义： $m()$ 是一个方法， V_m 是方法 $m()$ 引用的变量集合，以及 N_m 是方法 $m()$ 的控制流图中的节点集合。另外， $defs(i)$ 表示在节点 i 中定义的变量集合， $uses(i)$ 表示的是在节点 i 中所使用的变量的集合。 $entry(m)$ 是方法 $m()$ 的入口节点， $exit(m)$ 是出口节点， $first(p)$ 是路径 p 的第一个节点， $last(p)$ 是最后一个节点。

接下来的定义用来处理继承和多态。属于同一个类型集合 c 的类的集合表示为 $family(c)$ ，这里的 c 表示祖先基类（the base ancestor class）。 $type(m)$ 是定义了方法 $m()$ 的类， $type(n)$ 是类 c ，它是变量 n 的定义类型。 n 必须指向 c 的类型集合中某个类的实例。 $state(c)$ 是类 c 的状态变量的集合，这些状态变量要么在 c 中定义，要么从祖先继承而来。 $i_defs(m)$ 表示在方法 $m()$ 中间接定义的变量集合， $i_uses(m)$ 是指方法 $m()$ 使用的变量的集合。

耦合序列是指在程序执行路径中依次调用两个方法的节点对，第一个方法定义了变量，第二个方法使用该变量。这两个方法都必须通过相同的对象实例来调用。这种调用方法称为耦合方法 $l()$ ，它调用方法 $m()$ ，前驱方法（antecedent method）定义 x 和方法 $n()$ ，后继（consequent method）使用 x 。这在图7.11里面有说明。如果前驱方法和后继方法是和耦合方法一样，那么这是一种很难处理的特殊情况。如果前驱方法或者后继被另外一个叫做 $f()$ 的方法调用，那么这是一个间接耦合流图（在图7.11的右边），这种情况我们不去讨论。

图7.12是一个控制流原理图，它表示了一个耦合方法调用了前驱方法和后继方法。这个原理图将控制流图的细节部分都抽象出来，只留下与耦合分析相关的节点。细线段表示了控制流，粗线表示了耦合路径中的控制流。线能够表示多重子路径。一个路径可以表示为传递集合，例如 $[o, o.v]$ ，它包含了变量的路径是定义清晰的（definition-clear）。



方法`m()`和方法`n()`可能在同一个类,或存取一个全局的或其他非局部的变量。

图7.11 面向对象软件的Def-use对

假设对于状态变量 $o.v$ 中间的路径是定义清纯的,在图7.12中从 h 到 i 到 j 到 k 再到 l 的路径对于 $o.v$ 形成了一个传递路径。对象 o 定义为上下文变量。

每个耦合序列 $s_{j,k}$ 有一些在前驱方法中定义的状态变量,这些变量在后继方法中也使用。这个变量的集合称为 $s_{j,k}$ 的耦合变量集合 $\Theta'_{j,k}$,被定义为在通过一个 i 类型实例的上下文变量 o 方法 $m()$ 中定义的变量和在方法 $n()$ 中使用的变量的交集。耦合集合中的元素称为耦合变量。

耦合序列要求在序列中的每两个节点之间必须存在一条定义清纯的路径。通过检查这些路径是否为节点的完整路径的一部分来确定耦合路径集合。一个耦合路径被看成是一个变量从定义到使用的路径。

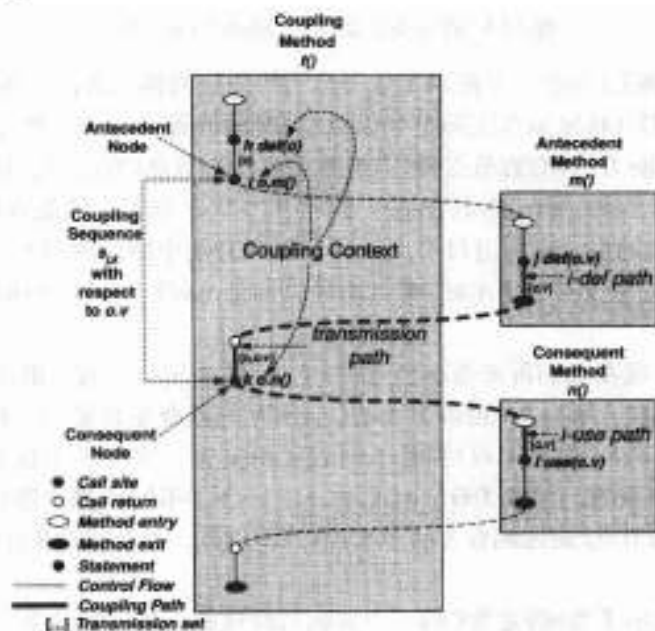


图7.12 原型耦合序列控制流原理图

每条路径包含三条子路径。间接定义子路径是耦合路径的一部分,该路径出现在前驱方法 $m()$ 中,从耦合变量的最后一次被定义到方法 $m()$ 的出口节点。间接使用子路径是后继方法 $n()$ 的一部分,从方法 $n()$ 的入口节点到耦合变量的第一次使用。传输子路径是耦合方法中的耦合路径的一部分,开始于前驱节点,结束于后继节点,在该条路径中,耦合变量和上下文变

量都不能被修改。

对于每个耦合子路径类型，每个耦合序列都有一个单独的耦合路径集合。通过匹配每个集合中的元素，这些集合被用来构造耦合路径。通过把间接定义子路径集合元素和传输子路径中的元素进行组合，再加入一个间接使用子路径中的元素就构成了耦合路径的集合。一个完整的耦合路径通过对这些集合进行乘积运算得到的。

为了观察继承和多态在路径上的效果，考虑图7.13a中的类图。类型集合包含了类A、B和C。类A定义了方法m()和n()以及状态变量u和v。类B定义了方法l()，重载了A的方法n()。另外，C重载了类A的方法m()。在图7.13b中显示了这些方法的定义和使用。

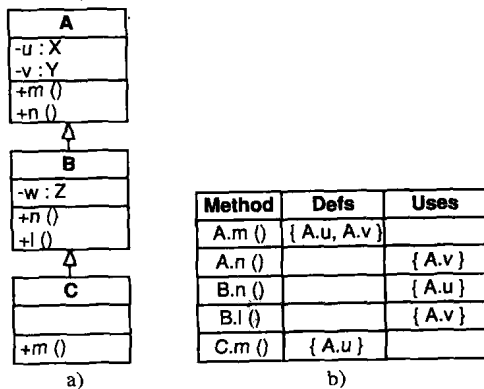


图7.13 样本类层次结构a)和def-use表b)

图7.14显示了在图7.13a中一个使用了层次结构的方法的耦合路径。图7.14a表明耦合变量o的声明类型是A，图7.14b显示当实际类型仍然是A时的前驱方法和后继方法。耦合序列 $s_{j,k}$ 指示的是一条从调用前驱方法m()的节点j到调用随后节点的节点k的路径。从图中可以看到，当o被绑定为A的实例时，相应的 $s_{j,k}$ 的耦合集合是 $\Theta_{s_{j,k}}^A = \{A.v\}$ 。因此，该集合包含 $s_{j,k}$ 的耦合集合，该集合从A.m()的节点e到A.m()的出口节点，再到耦合方法中的后继节点k，并且通过A.n()的入口节点到节点g。但是没有关于A.u的耦合路径，因为A.u没有在A.m()和A.n()的耦合集合中出现。

如图7.14c所示，现在我们来考虑当o被绑定为B的实例时，对于组成耦合路径的元素的效应。该种情况下的耦合集合与o被绑定为A的实例时的耦合集合是不一样的。这是因为B提供了一个重写方法B.n()，它和A.n()有着不同的使用集合。因此关于前驱方法A.m()和后继B.n()，耦合集合是不同的，结果为 $\Theta_{s_{j,k}}^B = \{A.u\}$ ，这导致了不同的耦合路径集的形成。此时，耦合路径集从A.m()的节点f通过耦合方法中节点k的调用点，并且通过B.n()的入口节点到B.n()的节点g。

最后，图7.14c显示了当o绑定为C的一个实例的时候耦合序列的结果。首先，我们可以观察到执行耦合方法中的节点j，将会调用前驱方法C.m()。另外，执行节点k将会调用后继方法n()。由于C没有重写方法m()以及C是B的后代，调用n()实际上调用的是B.n()。因此， $s_{j,k}$ 的耦合集合与前驱方法C.m()以及后继方法B.n()有关，使得 $\Theta_{s_{j,k}}^C = \{A.u\}$ 。相应的耦合路径集包含了从C.m()中的节点e开始到C.m()的出口节点，再到耦合方法的节点j，然后通过B.n()的入口节点到节点g的路径。

表7.4总结了图7.14中的例子的耦合路径。这些路径都以节点序列的形式展现。每个节点

的形式表示为 $method(node)$, $method$ 表示包含节点的方法名, $node$ 表示为方法内节点的标识符。前缀“call”和“return”代表了相应的方法调用或返回。

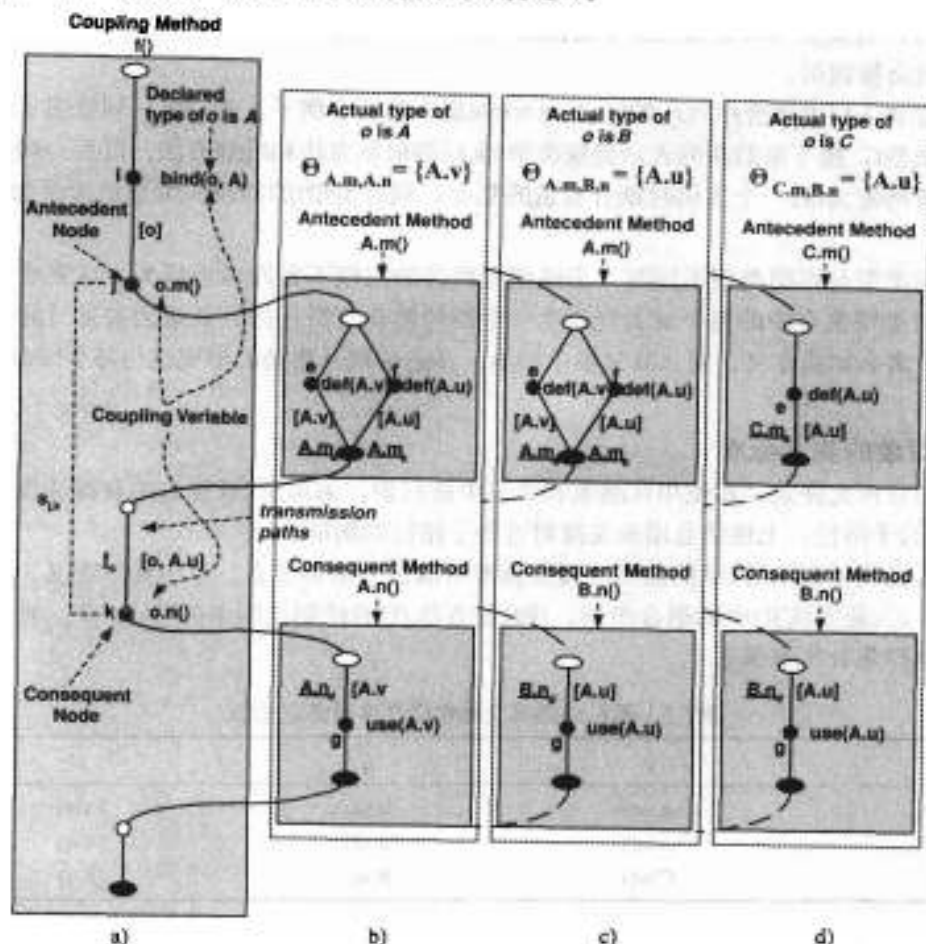


图7.14 耦合序列: a) 类型A的o, b) 绑定到A的实例; c) 绑定到B的实例; d) 绑定到C的实例

为了解释在一个调用点的多态行为的可能性, 必须修正耦合序列的定义来处理所有可能执行的方法。一个耦合序列的binding triple (绑定元组) 包含了前驱方法 $m()$ 、后继方法 $n()$ 以及绑定一个上下文变量到一个特殊类型实例所产生耦合变量的集合。这个triple匹配了一个 $p()$ 和 $q()$ 方法对。这个方法对的执行就犹如执行了前驱节点 j 和后继节点 k 。每个方法可能来自不同的类, 这些类都是定义为 c 的类型集合的成员。 $p()$ 是方法 $m()$ 的重写方法, 或者 $q()$ 是方法 $n()$ 的重写方法。对于类 $d \in family(c)$, 必有一个binding triple定义了方法 $m()$ 或 $n()$ 的重写方法。

表7.4 样本耦合路径概要

类 型	耦合路径
A	$(A.m(e), A.m(exit), f.j.return), f.k.call, A.n(entry), A.n(g))$
B	$(A.m(e), A.m(exit), f.j.return), f.k.call, B.n(entry), B.n(t))$
C	$(C.m(s), C.m(exit), f.j.return), f.k.call, B.n(entry), B.n(t))$

耦合序列包含了一个binding triple集合。即使有没有方法重写,对于相应的前驱方法和后继方法,这个集合总是包含binding triple。在这种情况下, binding triple集合的唯一成员为上下文变量的声明类型(假设该类型不是抽象的)。如果该类型是抽象的,那么最近的具体的后代的实例就会被调用。

表7.5是图7.14中耦合序列 $s_{j,k}$ 的binding triple集合的一个例子。表的第一列给出了 $s_{j,k}$ 的上下文的变量类型 t ,接下来的两列表示特殊类型 t 执行的前驱方法和后继方法,最后一列指的是当上下文变量绑定为 t 的一个实例时耦合变量的集合。耦合类型 t 的相应的类型层次图如图7.13所示。

在实际类型与声明类型不同时,上述耦合路径的实例不允许多态行为。这需要一个耦合路径实例对类型集合中的每个成员能产生一个路径集合。路径的数量是由被重写的方法数量来决定的,要么间接定义,要么从父类中继承。多态的耦合路径的形成也与每个binding triple有关。

面向对象的测试标准

上面的分析允许定义和使用在继承和多态中被标识。采用第2章提出的数据流覆盖标准定义OO程序的子路径,上述信息用来支持对这些子路径的测试。

接下来说明了利用第2章的数据流覆盖标准来测试继承和多态。给出如下定义, $f()$ 表示了被测方法, $s_{j,k}$ 是方法 $f()$ 中的耦合序列, j 和 k 是方法 $f()$ 的控制流图中的节点, $T_{s_{j,k}}$ 表示测试用例集合,这些集合用来满足 $s_{j,k}$ 。

表7.5 图7.13类层次结构耦合序列绑定元组

t	p	q	s
A	A.m()	A.n()	{A.v}
B	A.m()	B.n()	{A.u}
C	C.m()	B.n()	{A.u}

第一个标准基于这样的假设:每个耦合序列在集成测试的时候将被覆盖。相应地,所有耦合序列(All-Coupling-Sequences)要求 $f()$ 中的每个至少被一个测试用例覆盖。

定义7.53 所有耦合序列 (ACS):对于方法 $f()$ 中的每个耦合序列 s_j ,至少存在一个测试用例 $t \in T_{s_{j,k}}$,使得存在一个由 $s_{j,k}$ 导出的耦合路径为执行路径 $f(t)$ 的子路径。

ACS没有考虑到继承或多态,因此下一个覆盖标准将包含了调用的上下文。通过确保对每个类有一个测试用例,该测试用例对于每个耦合序列能够提供一个上下文实例,达到覆盖继承和多态的目标。方法如下,在出现给定的耦合上下文中的耦合序列用每个可能的类型替换来进行测试。

定义7.54 所有多态类 (APC):对于方法 $f()$ 中的每个耦合序列 $s_{j,k}$,以及对于上下文 $s_{j,k}$ 中的类型集合中每一个类,总至少存在一个测试用例 t ,当用 t 执行 $f()$ 的时候,存在着一条 $s_{j,k}$ 的耦合路径集合的路径 p 是 $f(t)$ 的执行路径的子路径。

联合 $(s_{j,k}, c)$ 是可能的,当且仅当 c 的类型和 $s_{j,k}$ 中上下文变量所声明的变量一样,或者说 c 是声明类型的子类型并且对前驱方法和后继方法定义了重写方法。也就是说,只有重写了前

驱方法或后继的方法将会被考虑。

APC要求覆盖到耦合序列,但是该标准没有考虑可能被包含的多重耦合变量的相互操作。因此在测试过程中一些定义或者使用的耦合变量可能没有被覆盖到。

下一个标准通过要求在 $s_{j,k}$ 的前驱方法中的每一个最后定义的耦合变量到达后继的 v 的最先使用来说明这些限制。因此,至少存在一个测试用例,来通过变量 v 可能的耦合路径 p 。

定义7.55 所有耦合定义使用 (ACDU): 对于耦合序列 $s_{j,k}$ 的每个变量 v 至少存在一个测试用例 $t \in T_{s_{j,k}}$, 覆盖由 $s_{j,k}$ 产生的耦合路径 p , p 是执行路径 $f(t)$ 的一条子路径。

APC要求用到多个上下文实例,ACDU要求定义之后才能使用。最终标准将这两个需求合并起来。除了继承和多态,ACDU要求对于每个类型集合的成员,所有执行的耦合路径必须被耦合序列的上下文所定义。

定义7.56 所有多态耦合定义使用 (APCDU): 对方法 $f()$ 中的每个耦合序列 $s_{j,k}$, 对于在 $s_{j,k}$ 的上下文中定义的类型集合中的每个类, 对 $s_{j,k}$ 中的每个耦合变量 v , 对于每个含有最后 v 的定义的节点 m 以及最先使用 v 的节点 n , 至少存在一个测试用例, 当 $f()$ 执行该测试用例 t 的时候, 在 $s_{j,k}$ 的耦合路径中存在一条路径, 该路径是执行路径的子路径。

7.2 测试Web应用和Web 服务

使用万维网来部署软件,对软件测试人员提出了很多有趣的、待解决的问题。首先,以这种方式发布的软件与常规桌面软件的一个本质上的差别就是,Web应用是部署在Web服务器上的,这种部署方式使得任意一个客户端都可以通过Internet发送HTTP请求的方式从服务器端获取信息。HTTP协议的无状态特性和客户机/服务器的分布式架构构成了这种应用的特殊环境。

Web应用是一个虚拟的世界,它允许人们从世界的任意角落接入、访问。但这将带来一系列的问题,这使得应用将面对在世界范围内的各种各样的用户,他们的地理位置、人口特征、时区和语言等很多方面都存在着显著差异。互联网行业竞争激烈,通常Web应用有很高的可靠性要求。用户期望Web应用每次都能正确地工作,一旦Web应用发生了错误,那么用户可能会转移到其他相类似的网站。这使得对测试提出了严格的要求。

Web应用通常还会以一种新颖的方式来构建。它们通常是由一系列相对简单的组件组合而成,这些组件分布在不同的机器上,它们并行地运行并以一种特定的方式共享存储。HTTP是无状态协议,这就意味着每个从用户机到服务器的请求/响应交互都是相互独立的。因此,任何状态信息都需要由应用软件显式地进行管理,比较常见的技术有cookies、session 对象和数据库的离线存储等。

Web应用通常会由很多新技术共同构建,常用的技术有JSP、ASP、C#、Java、JavaBeans、XML、JavaScript、Ajax、PHP等。对于它们中的单独组件的测试与传统的软件测试并没有多大区别,但我们所不知道如何测试多种技术的相互作用。而且,Web应用通常是由大量的小型组件组成,这些组件是由新的方式集成在一起。

可以将对Web 应用的测试划分为三大类:

1. 测试静态的超文本Web站点。
2. 测试动态Web应用。
3. 测试Web服务。

对于本书来说, 我们所讨论的Web页面是指可以在一个单独浏览器窗口中查看的包含HTML内容的实体。Web页面可以是一个静态的HTML文件, 或者它也可以是由如JSP、Servlet或ASP这样的技术动态产生出来的页面。Web站点是指由一组在与以上相关的或通过超链接相关联的Web页面以及其他一些相关的软件所组成的实体。静态页面是指对于所有的用户来说都是不变的页面, 它们是通常存储在服务器上的HTML文件。动态页面是指一段程序根据用户的需求动态产生的页面, 它的内容和结构都可能根据用户的输入的不同而改变, 比如用户的位置、用户所使用的浏览器、操作系统甚至是用户的当地时间的不同。Web应用是一个部署在网络上的完整的软件程序。用户访问Web应用程序通常是使用用户计算机上的浏览器来发送HTTP请求。Web应用的测试用例可以描述为客户机和服务器之间的一个交互序列, 即Web应用之间转换路径。

7.2.1 测试静态超文本Web站点

测试静态站点相对来说比较容易, 主要关注各种链接在客户端的验证和静态的服务器端的验证。在Hower管理的Web站点上列出了一组支持辅助Web测试的工具^①。包括了商用的和免费的超链接验证工具、HTML验证工具、捕获/回放工具、安全测试工具以及下载测试工具和性能压力测试工具。

这些都是静态验证和评测工具, 这种测试寻找失效链接, 即链接的网址不再有效, 评估导航结构寻找页面间无效的路径, 或是用户可能希望要有的便捷导航链接。

一个对静态网站建模的常用的方法是将其描述成一个图, 将网页抽象成节点而将链接抽象成边。该图可以从一个首页开始构建, 然后递归执行广度优先搜索所有链接。由此产生网站图, 然后通过对该图的每一条边的遍历进行测试(边覆盖标准)。

7.2.2 测试动态Web应用

用户界面(客户端)和软件(服务器端)的分离是测试Web应用程序的众多挑战中的一个。测试者通常并不具有在服务器上访问和存取数据、状态或源代码的权限。本节首先讨论客户端测试的策略及其局限性, 然后讨论服务器端的测试策略, 当测试者可以获得Web应用的代码实现时, 服务器端的测试策略可以采用。

Web应用的客户端测试

当所有的网页和链接用HTML静态地编码时, 对静态超文本链接的测试工作得很好, 但如果部分内容是由服务器根据用户的输入动态创建的时候, 这种测试就不是很有效了。我们需要通过某种方式生成表单输入域。如果某些网页或链接只有对特定的输入才有效那么生成一个网站的图模型也将变得不可预测。

一个解决的办法就是从给定的URL开始非确定性地探测“响应序列”。对表单输入域的测试数据也是从事先给定的一组数据中抽取。

另一个生成输入数据的方法就是从Web应用的先前用户中抽取数据。这称为用户会话数

① 本书的支持网址: <http://www.softwareqatest.com/qatweb1.html>。

据。大多数的服务器或者是捕获用户提交给服务器端Web应用的数据，或者是改变设置来收集数据。

还有一个获取输入数据的方法叫做省略测试。许多Web应用都对HTML表单的输入域有一定的约束条件。这些约束通常有两种形式，客户端的脚本验证通常是用JavaScript编写的，这种验证是运行在客户端的，它可以对用户的输入在发送到服务器端之前进行验证。这通常用来确保必须填写的域被填写了，对输入限制为数字的域的输入只包含数字等。另一种方式是使用与HTML表单域相关的显性的属性，比如只能输入固定数目的文本框或是一个下拉列表来提供预设的值。默认值测试通过以一种可以确保不会违反约束规则的方式产生测试数据，这样就可以不进行页面验证直接将数据发送到服务器端了。

这两个方法有一个共同的不足，就是寻找一个Web应用中的所有页面时的不确定性。它们通常依赖于某种特定的启发式搜索算法期望能识别出Web应用的所有页面，但事实上可能有的页面只有当用户输入特定的值的时候才可能出现，这种页面是很难找出来的。而基于服务器端的测试方法就不存在这些不足，因为它们可以查看程序的源码和许多潜在的在客户端不可见的页面。

Web应用的服务器端测试

Web应用软件在运行时允许有执行控制权的转移，这是在传统应用软件中所没有的特点。在传统的程序中，程序的运行控制流完全由程序自身控制，这就保证了测试人员只有通过不同的测试输入影响程序的运行。而Web应用没有这种特点。在Web应用程序运行的时候，用户可以在不改变“程序控制者”的情况下中断一个正常的控制流。在程序设计、操作系统等课堂上讨论的程序控制模型不能完全应用于Web应用，因为程序的控制流是在客户端和服务器端相互转移的。用户可以在浏览器上通过按刷新、后退等按钮来改变浏览器上的URL，这些交互使得原有的程序执行流程引入了一些不可预测的变化，这引入了一些原有的程序流图等传统的建模方法所不能表示的控制路径的出现。用户还可能通过修改隐藏表单值的方式直接改变程序预期的有效数据，甚至是用户浏览器的不同设置都有可能影响到Web应用程序的正常执行，比如，用户可能关闭了浏览器的cookies功能，这就会造成Web应用的很多操作将变成不可用的。

基于如上分析，我们将众多的新特征归为以下几类：

- 传统的静态的HTML链接，用<A>标签标识。
- 动态的<A>链接可以从一个静态的页面向服务器发出请求，以要求服务器执行某些处理程序。这种情况下，请求中不包含任何表单数据，而且这样的HTTP请求总是可得的。
- 动态的表单链接，通过使用<FORM>标签向服务器的特定软件组件发送请求以处理某些数据。这样的HTTP请求可以通过get发送的，也可以是通过post方式发送的，这取决于<FORM>标签中<method>属性的值。通过表单发送的数据影响着后台处理程序的处理过程，这种情况是我们测试的重点。
- 由Web软件组件动态生成HTML，这是典型的根据用户请求产生响应的情形。这种情况下的页面内容通常取决于用户的输入，这对测试分析造成了非常大的困难。
- 基于状态的动态创建的GUI，这种情况下HTML页面不仅取决于用户的输入，还与其他的一些因素有关，比如服务器的状态、现在的时间或日期、用户、数据库中的数据或session信息。HTML文档还可以包含JavaScript脚本，这是一段可以在客户端运行的程序。

它们也可以包含超链接,这些也可能会对程序的执行造成影响。这些JavaScript程序产生和链接也会随着用户的不同操作产生不同的结果。

- 操作性状态转变,这是在HTML或其他软件控制范围之外,由用户引入的。操作性状态转移主要包括使用后退按钮、前进按钮和URL重写等方式。这种操作是在浏览器这种软件中新出现的特有的操作方式,这种操作的主观性使得它们难以预测,这也为测试人员带来了巨大的麻烦。
- 本地软件连接,是这样的一组运行在服务器端的后台软件组件,比如方法调用。
- 远程软件连接,这是指当Web应用访问那些存在于外部站点的软件组件的情形。这可以通过HTTP或其他协议向其他服务器端运行的软件发送请求消息。这种外部连接功能强大,但对于测试者来说这却增加了测试的困难,因为测试者可能对外部软件是一无所知的。
- 动态连接是指在软件运行时新安装的软件组件。J2SE和.NET平台都允许动态地部署新的软件组件。这种连接的评测是异常困难的,因为这种连接在新的软件组件部署以后才是可用的。

这些状态转变所造成的结果就是传统的分析方法如控制流图、调用图、数据流图和数据依赖图都很难准确地对Web应用进行建模。即程序可能执行的控制流路径是不能够静态地观测的。对这些结构的分析需要我们引入新的分析技术。

一个较早的对Web应用测试的尝试是试图将数据流图分析应用到Web软件组件分析中来。可以将def-use对在客户端页面和多个服务器端软件组件之间进行划分。

这里将HTML文档的原子部件(其中可能包含了像JavaScript这样的脚本语言)定义为拥有如下一些特性的HTML文档的片段,在发送给客户端的过程中,只要其中的一部分元素发送了,那么整个部分就保证会被全部发送。原子部件的定义使我们可以将Web应用建模成一个个基本的块状结构,从而可以用控制流图对非Web应用软件进行建模。这些图可以用来构建我们将在第2章中提出的图形标准中使用。

到目前为止,这些想法都还只出现在研究型的文献中,还没有被运用到实际的应用中。

7.2.3 测试Web服务

Web服务给测试人员带来的麻烦不是特别多。但不幸的是,“Web 服务”这个术语本身就不是标准化的,它存在着很多不同版本的定义,其中一些还是相互冲突的。其中最为常见的标准是XML技术和简单对象访问协议(SOAP)。本书将在一个更广泛的层面上提出一个更通用的方法。我们将Web服务定义为一种分布式的模块化的应用,它的各个模块通过结构化的消息格式进行通信。

对Web服务的测试的困难在于它们是由一组有着不同的运行时行为特征的分布式的软件组件构成的。它们的设计和实现的详细信息是不可获得的,测试者只能通过客户端进行测试。一个单一的商务处理流程经常涉及多个Web服务,而且多个Web服务可能是运行于不同公司的不同服务器上。这些Web服务通过结构化的消息格式传递信息进行交互。虽然已经有了关于这种通信的语法方面的验证技术,但真正的问题在于任意的两个Web 服务之间的交互行为是否对于任意的消息都能保证其行为的正确性。

Web服务的通信技术的目标就是要使得不同的Web服务可以通过Internet被描述、发布、

发现和调用。这有好几项关键的技术使得Web服务能够很好地一起工作，主要的技术包括：XML被用做包装传递的消息的通用数据格式；UDDI用于维护一组Web服务的目录的指针，这些指针保存了Web服务的各种描述信息，并且它们使用的是一种符合规范的格式，以使得其他的Web服务程序能够读取其中的内容；Web服务描述语言（WSDL）用于描述如何访问一个Web服务以及该Web服务提供了哪些操作；SOAP协议则帮助软件在Internet上传输和接收XML消息。

对于Web服务的测试的研究还处于开始阶段，目前的测试集中在对消息的验证上。Web服务的输入都是一些符合特定规范的XML消息，这些需求都以XML Schemas的形式进行了描述。研究人员开始应用在本书的第5章中给出的语法测试标准对Web服务进行测试。

7.3 测试图形用户界面

图形用户界面（GUI）通常要占到现代软件系统源码的一半以上。但这对于测试这种大型的软件系统来说并没有带来什么益处。对于GUI的测试可以分为两大类，可用性测试是指这些接口的易用性，虽然易用性测试是极为重要的一环，但这个话题的讨论已经超出了本书的范围。功能性测试是指测试这些接口是不是像期望的那样工作的。

功能性测试又可以划分成4种类型。GUI系统测试是指通过GUI进行软件系统测试，GUI系统测试与其他类型的测试的唯一区别就是如何实现测试的自动化。回归测试是指当工程发生了改变之后对UI进行重新测试。最常用的回归测试是各种录制/回放工具。一般来说，就是记录下用户的输入，在UI发生了改变之后再通过回放的方式进行测试，并报告不同之处。市场上有很多这样的工具，它们的思想也比较简单，因此在本章中就不做深入讨论。输入的有效性测试是指测试软件识别非法输入并作出相应的响应的能力。这种测试与用不用图形用户界面的关系不大，因此也不做深入讨论。最后，GUI测试还包括GUI工作的有效性。测试者可能会问“是不是所有的UI组件都如预期的一样工作的”，“软件能保证用户可以自由的导航到所有的他们希望的页面上去吗”或是“非法的导航是不是被禁止的”。

7.3.1 测试GUI

一种显而易见的GUI测试方法就是使用某种有限状态机来对GUI系统建模，这可以采用第2章中我们提出的基于图形的标准。将GUI系统建模成一个状态机是相当直观的，因为它们本身就是基于事件的系统。每一个用户发出的事件（按下按钮、输入文本、切换到另一个屏幕）将引起GUI的状态变化。一条路径是穿过这样的状态转移边的序列，代表着一个测试用例。这个方法的优点就是期望的输出是测试用例输入所达到的最终状态。而这个方法的缺点就是可能引起状态的爆炸，即使是一个小型的GUI系统，它都可能有着数以千计的状态及转换。

状态的数目可以通过很多方式进行消减。可变的有限状态机通过增加模型中的变量来减少抽象状态的数目，这些模型都必须手工创建。并且如果这些模型被用于自动化的测试语言，就需要将状态机的各个状态和GUI的具体状态有效地对应起来，这个过程也是需要测试人员手工来完成。

一种针对GUI测试的状态机变种，将状态空间根据用户的任务的不同划分为不同的状态机。测试者首先识别出用户的任务（称为职责），这个过程可以用GUI方式完成。每一个职责被转换成一个完全交互序列（CIS）。这些都和第2章中的用例的描述非常相似但并不完全相同。

每个CIS是一个图，图的覆盖标准可以用来覆盖CIS。虽然定义指责的工作比较简单，但将它们转变成有限状态机模型的工作还是必须得手工完成，这是一项非常耗时的工作。

另一个测试GUI的方法依赖于对用户行为的建模，特别是对初学者行为的模拟。这是因为熟练的用户通常会采取便捷、直接的方式来操作，这对于测试GUI系统来说并没有什么特殊的帮助。而新手的操作经常以一种间接的方式进行，并且对GUI的操作会有很多不同的方式，这对于GUI测试是很有用的。通过这种方法，我们可以这样进行GUI测试：首先让专家完成一个有很少操作步骤的输入序列，这个序列将被用于产生测试用例，产生测试用例的算法的大致思想是对这个序列进行修改以使它们看上去像是由新手们产生的操作序列。

一个更为折中的方法是基于GUI系统的事件流模型的方法。事件流模型通过两个步骤进行处理。第一步，将每一个事件按先决条件进行编码，预定义标准包括执行过程中可能通过的状态，以及事件影响即由事件引发的各种结果。第二步，测试者将时间流的GUI中所有可能执行的序列用有向图的方式表示出来。第三步，在一种目标导向的方法中用先决条件和各种事件结果来产生测试用例。由于预期的测试输出就是这些事件的目标状态，这就使得测试语言可以自动产生和自动检查。有向图模型可以用来产生满足第2章中提出的测试标准的测试用例。

关于如何使用这些方法的详细情况可以参考本书的参考文献。

7.4 实时软件和嵌入式软件

实时软件系统必须在输入的同时或是一个有限的时间内完成响应。实时系统通常是嵌入在一个很大的工程系统环境中的系统，有些时候是专门为特定的领域和特定的应用平台设计的。

实时系统通常要与子系统进行交互，并且通常处于一个特定的物理环境中，这些构成了实时系统的运行环境。例如，有这样一个控制机械手臂的实时系统，其运行环境包含了传输带传过来的工件和生产线上其他的机械控制系统发送过来的消息。实时系统通常有着严格的响应时间限制。例如一个飞行系监控系统的响应时间被限定在30秒以内，这种时间限制称为截止期限。时间限制通常是由环境中动态产生的各种参数或系统设计者在进行系统设计时出于安全和设计的需要所决定的。

时间线描述的是一个软件满足时间限制条件的能力。例如，一个飞行监控系统的响应时间限制为30秒。软件中的错误可能导致软件违反时间限制，这将会导致事故的发生。因此测试者还得测试软件是否违反时间限制条件。

实时系统有时也称为反应型系统，因为它们的行为通常表现为对环境中的变化产生响应。由于实时系统控制着硬件与现实世界的实体、人之间的交互，因此通常要求它们是可靠的。

实时应用是由一系列实现了特定功能的实时系统的任务组成的。实时应用程序的运行环境是由硬件和软件所共同组成的，如实时操作系统和I/O设备。

有两种实时任务被广泛使用。一种是定时任务，定时任务是被以固定频率激活执行的任务，所有激活任务的时间点在事先都已经确定了。例如，一个每4个时间单位执行一次的任务将会在0、4、8等时间点上被激活。另一种是非周期性任务，非周期性任务可能在任何时间点上被激活。为了达到实时系统的时间响应限制，非周期性的任务通常要对其活动特征有一个明确的限制。当有限制时，任务被称做不定时发生的。一个常见约束是连续的任务激发的最小到达间隔，任务通常还有一个属性来描述任何实例被激活的偏移时间。

测试人员通常想了解程序的最长执行时间（通常称为“最坏情况”）。不幸的是，这通常

很难评测,为了对此做出评估一般的做法就是让软件执行尽可能长的时间。

实时任务的响应时间是指任务的开始执行到执行完毕的时间。响应时间是一套依赖于调度计划的并发任务,这称为任务的执行顺序。

与实时系统测试相关的一些问题

在Schütz写的一篇划时代的论文中,他提出了测试一个实时系统所需要考虑的各个方面。在实时软件的环境下,系统的可观察性是一种监控和记录实时系统行为的能力。可观察性的提高是通过插入探针的方式来实现的,探针揭示当前状态及系统内部状况变化。但这又带来了新的问题,增加的这些探针会影响系统的响应速度,如果将它们去除的话就会引起测试结果的变化。这个问题通常被人们描述成“探针效应”。解决“探针效应问题”的一般做法就是将探针留在软件中,并将这些探针的消耗集中到一个与探针花费的资源相当的但在真正的执行过程中不可用到的通道中去。

这种特定的探针是一种软件或硬件中内建的模块,它可以监控系统的活动。在一个资源有限的环境下,指针效应使人们需要将探针的日志信息输出的花费降到最小。这在硬件资源极为有限的嵌入式的实时系统中是非常常见的。

与此相关的两个概念是可再现性和可控制性。可再现性是指当对一个系统提供相同输入的时候,系统的行为能很好地重复之前的测试结果。可再现性在软件测试和软件调试中都是一个非常重要的特性。

在事件触发和动态规划执行的实时系统中,可再现性是很难达到的。这是由于系统的行为依赖于系统当时的状况,系统的输入只是其中的一小部分。比如一个任务的响应时间取决于系统的加载时间和持续变化的硬件加速效率等,这样的系统是非确定的。一个能有效地测试不确定性软件的典型的需求就是软件的高可控性。

如果被测试的软件是一个不确定型的低可控制性软件,那么测试人员就必须使用统计的方法来确保测试结果的有效性。通常的做法就是反复多次执行测试直到测试结果具备了统计显著性要求。最小的可控制性的需求就是多次对一个导致操作被拒绝的相同的输入序列能够以相同的方式被拒绝。

时效性故障、错误和失败

术语时效性故障是指由于实时系统错误地实现或配置可能导致不正确的暂时性行为。例如,时效性错误可能是由于一个条件的分支声明错误从而导致一个任务迭代了不正确的次数。另一个例子是两个任务相互影响(比如共享了未加保护的硬件或软件资源)。这些错误都可能导致任务的某一部分的执行时间超出原来的预期。另一种时效性错误发生在当环境的行为与预期的差别超出了原有的预期的时候。例如,一个中断处理机制受到了未曾预期的延迟影响,使得内部到达时间比期望的要短。

时效性错误发生在系统内部的行为偏离了原来预想的行为轨道的时候。当一个非实时程序有内部状态错误时,情况相似。缺乏系统内部行为的详尽日志和准确知识,检测时效性错误很困难。而且可能只有一种特定执行顺序,时效性错误才会被检测到,并导致系统级别的时效性错误。

时效性失败是指外部可见的违反时间约束的情况。在一个严格的实时系统中,这可能会对整个系统的继续执行都会带来严重的后果。这是因为时间限制通常是通过与外部的系统行为

进行比较来表示的,因此时效性失败是很容易检测的。

对时效性的测试

测试标准必须做出适当的调整以便可以处理时效性约束,这是因为对那些不考虑其他任务的影响和实时性协议的一个严格的输入序列的描述是非常困难的。在本书的前几个章节提出的测试标准都没有关于实时系统测试用例的信息,而且它们也不能预测现实执行的顺序以揭露那些违背我们预期的错误。

时效性通常使用调度分析的方法来进行分析和维护,或者通过进入控制及应变模式来在线式地规范时效性。然而这些几乎都假设所有的任务和活动特征都必须保证维护的时效性约束的正确性。更进一步,一个对于非平凡的系统模型的可调度性分析是非常复杂的,而且需要满足运行系统的一些特殊规定。对于时效性的测试更为普遍,这被运用到了所有的系统架构中,用来提高可能导致误时执行顺序系统性抽样的假设检验的置信度。如此,这个工作的挑战就在于找出这样的可以引发时效性错误的执行顺序。

实时系统的测试通常依赖于对软件的形式化建模。一个描述软件的方法是增加了时间标记的Petri网。这样就能将图标准中的边覆盖标准或是路径覆盖标准用来帮助我们找出这样的违反时效性约束的测试输入。

另一个对时效性约束建模的方法是将事件约束表示成约束图,并且用代数方式表示待测系统。这个方法仅考虑系统输入的约束条件。

用时钟区域图来表示时效性约束是另一种建模的方法。一个基于时间的对系统的自动化规范将被“展开”成易于表示的输入和输出形式,这些将用来导出每一个时钟区域实现的一致性检测。

还有一种称为时序逻辑的方法可以用来对实时系统进行建模。测试用例的各个元素由增加了时间戳的输入和输出对来表示。这些输入和输出对可以以时间为标准进行合并和转换,进而创建出更多的测试用例。这些输入和输出对的数目将会随着系统中存在的约束的数目的增加而迅速增加。

时间自动机也被用来验证附有时间标记的响应序列。这个方法使用了状态转移的可达性分析,这种分析方法是一种基于图的方法。这个方法存在着“状态空间爆炸”的风险。减轻这种问题的一个方法是通过一种基于网格自动机的抽样和非确定性自动机方法来减少测试成本。

一种非形式化建模的方法使用了遗传算法来处理这个问题。这种方法通过在真实系统的执行过程中收集数据,并将这些数据用于以后的分析中。测试用例的适应度是根据测试用例的唯一性及由系统测试装置执行所生成的。

还有一种产生测试用例的方法,是在将一个测试用例真实运行之前通过静态推导执行的方式模拟系统的执行。每一个执行顺序被当做一个独立的程序序列,这一过程中可以使用其他一些传统的方法。这种方法只有在所有的任务响应的确定时间的时候才能使用。

以上所说的这些测试方法都以某种形式使用了第2章中阐述的图标准。另一种不同的方法是基于程序变异操作的(见第5章)。在这种基于程序变异的测试方法中,潜在的错误是通过变异操作符来揭露的。可能违背系统时效性要求的程序变体被识别出来,然后通过产生构建各种测试用例来“杀死”这些程序变体。

这种方法定义了8种不同类型的程序变体。**任务集变异操作符**改变某一项资源被占用的时间点。**执行时间变异操作符**以一个时间常数 Δ 为单位增加或减少一项任务的执行时间。占有时

间变异操作符改变对一项资源的加锁时间间隔。加锁时间变异操作符增加或减少对一项资源的加锁时间。未加锁时间变异操作符改变一项资源的未被加锁时间。内部到达时间变异操作符以一个时间常量 Δ 为单位改变执行某项任务所需的各项请求被满足的时间。特征偏移变异操作符以一个时间常量 Δ 为单位改变执行某项任务时间特征。然后通过模式检查和遗传算法产生测试用例“杀死”这些程序变体。

7.5 参考文献注释

参考文献注释的记录顺序是按照本书章节安排的顺序进行的：面相对象软件、Web应用软件、GUIs以及实时和嵌入式系统软件。

Meyer[241]、Liskov、Wing和Guttag [212, 213]和Firesmith [119]等人的这些论文是关于使用面向对象语言构建的软件抽象的很好的资料。Liskov、Wing和Guttag提出了可替代性原则，在Lalonde、Pugh[199]以及Taivala [323]这些论文找出了可替代性原则是否总是应该被遵守的结论。

Binder在他的论文[35]中指出了软件系统中的OO关系是如何变得日趋复杂的。Berard在他的论文[31]中首先提出了整合中的差异性。Barbey在他的论文[24]中指出了继承、多态、动态绑定和非确定性之间的关系。

Doong和Frankl在论文[105]中提出了一种非常有创意的基于状态的面向对象的测试方法。Harrold和Rothermel在论文[152]中提出了这种方法中的三种测试级别：方法内测试，方法之间测试和类内部测试。Gallagher和Offutt[132]对这一方法进行了补充，增加了类之间的测试。

有多篇关于类内部测试的论文[118, 152, 281, 315]，论文[284]讨论了测试类和它们的使用者之间的关系，论文[181]讨论了系统级别的测试。

Alexander和Offutt在论文[266]中以Binder的“程序的执行在某些时候可能会在对象的继承层次上下变动”理论上提出了Yo-Yo图的方法，Alexander和Offutt在论文[9, 266]中列举了OO错误及异常的分类。大多数关于OO计算和数据流覆盖标准是由Alexander在他的系列论文[7, 10, 11]中提出来的。

Ricca和Tonella在论文[300]中首次提出了一种用图来对静态网站进行建模的方法。Kung等人在论文[198, 215]中也开发出了一种以图来表示Web站点的模型，并提出了一些用以测试网页间转换概念的初步定义。他们的模型引入了静态链接转换并聚焦在客户端并少量使用了服务器端软件。他们定义了一种对象内部测试，测试路径使用的对象内部的数据def-use路径。对象间测试使用的是筛选的对象间变量def-use路径和客户端内部测试，测试的是从客户端进行交互的连通图中推导出来的路径。

Benedikt、Freire和Godefroid在他们的论文[30]中原创性地在一个名为VeriWeb的工具中提出了一种称为“响应序列”的测试方法。VeriWeb的测试方法是基于图形模型的，这种图形模型以Web页面作为图的节点，页面之间的链接作为图的边，图的大小由一个剪枝程序控制。

Elbaum、Karre和Rothermel在论文[112, 113]中提出了“用户会话数据”来对Web应用程序产生测试用例的想法。

Liu、Kung、Hsia和Hsu在论文[216]中首次将数据流分析应用到了Web软件测试中来。他们的关注点是数据交互和他们的模型而不是动态生成的Web页面或网页间的转移的不匹配性。

Offutt、Wu、Du和Huang在论文[278]中提出了旁路测试方法。本书7.2.2节中关于Web应

用程序有关关联的分析引用的是Offutt和Wu[359, 360]的论文。他们还提出了一种原子软件测试结构的概念。

Di Lucca和Di Penta[217]提出了一种Web 应用的测试序列, 这种测试序列和一些操作状态转移相关联, 特别是浏览器上的前进和后退操作。从发表时间来看, 这篇论文是业界第一次对于处理操作转移的公开研究。尽管晚于Offutt和Wu[359]的技术报告。

Offutt 和Wu在论文[297,280]中提出了创建XML信息来测试Web服务组件的语义测试技术。

Nielson在论文[253]中给出了一个非常好的关于Web可用性的概论, 包括对可用性测试的讨论。

早期的关于使用状态机来对GUI进行测试的方法由Clarke在论文[79]、Chow在论文[77]、Esmiloglu在论文[115]以及Bernhard在论文[32]中提出来的。Shehady等人论文[312]中提出了一种有限状态机的变体模型。

将GUI状态空间按照它们的责任划分成不同部分的思想是由White等人论文[346,347]中提出来的。对初学者的行为进行建模用以测试的想法由Kasik在论文[183]中提出。Memon等人论文[235, 236, 237, 238, 239, 240, 250]中对事件流模型进行了详细深入的讨论。

关于测试实时软件和嵌入式软件的参考文献相对于其他方面的参考文献来说要少很多。在本书中提出的这些相关概念只是对这个主题的一个介绍。如下的参考资料希望对感兴趣的读者有所帮助。

关于实时软件系统和时效性相关的一半概念可以参考Young的论文[364]、Ramamrithaam的论文[296]以及Schütz的论文[309]。测试实时系统的很多相关事项是由Schütz在他的论文[310]中提出来的。探针效应是由Gait在论文[131]中提出的。Verissimo和Kopetz在他们的论文[331]中对时效性进行了讨论。

使用Petri网进行测试的方法是由Braberman等人论文[42]中提出来的。Cheung等人论文[68]中提出了一个多媒体软件的测试框架, 其中包括对有着模糊的最后期限限制的多个任务临时性关系的讨论。

Clarke 和Lee在论文[78]中提出了使用约束图和进程代数来测试时间约束。

Petitjean和Fochal[287]提出时钟域图的方法。Krichen和Tripakis[193]处理之前客户端应用的实用性的限制, 并建议用不确定部分观察模型进行一致性测试的方法。Hessel等人[159]受到测试标准的启迪。

Mandrioli等人论文[225]中对时序逻辑方法进行了讨论, 这种方法是基于SanPietro等人的论文[308]。Oliver和Glover在论文[61]中提出了时间自动机测试方法。另一种自动化测试方法由En-Nouaary等人论文[114]中提出, 这篇论文还提出了使用网格自动机的测试方法。类似地, Nielsen和Skou等人论文[252]中提出了一种子类化的时间自动机测试方法。Raymond等人论文[299]中提出了一种产生事件序列的响应系统。

Watkins等人的论文[337]对遗传算法进行了讨论。Morasca和Pezze在论文[245]中提出了一种使用高阶Petri网作为规范和实现的测试并行实时系统的测试方法。Thane的论文[326]和Pettersson、Thane的论文[288]讨论了静态推导执行顺序的测试方法。Wegener等人开发了一种具备测试具有时效性的实时系统的遗传算法[338], 这种算法的目标是自动产生能导致出现最好的执行时间和最坏的执行时间的测试输入。Nilsson的系列论文[254,255,257,256]将程序变异测试方法引入到测试软件的时效性故障上。

第8章 创建测试工具

测试标准有很多使用方式，但是最普遍的方式是将测试标准用于测试评估。也就是说，评价测试用例集覆盖一个测试标准的程度。然而以这样的方式应用测试标准成本过高，所以需要自动覆盖分析工具来帮助测试人员。覆盖分析工具接受一个测试标准、一个被测程序以及一组测试用例，然后计算出被测程序所达到的测试覆盖。本章所要讨论的是这类工具的设计技术。我们不讨论孤立的各个工具，尽管这样的工具已经很多了。我们也不讨论用户界面的问题，而是关注度量覆盖的核心内部算法。

8.1 图和逻辑表达式标准的插桩

用于度量覆盖的主要机制是插桩。插桩是一段附加程序代码，它不改变程序功能性行为，而是搜集一些附加信息。在实时系统中，插桩语句可能影响适时性 (timing)，并且有可能影响系统的并发处理，因此应用插桩技术时需要格外小心。精良的设计可以使插桩技术更加准确有效。

对测试标准的覆盖来说，附加信息指的是各个测试要求是否被满足。图8.1给出了使用插桩技术的第一个例子。它表示增加一个语句，用来记录程序执行是否到达“if块”的语句体。

插桩前的函数	插桩后的函数
<pre>public int min (A, B) { int m = A; if (A > B) { m = B; } return (m); }</pre>	<pre>public int min (A, B) { int m = A; if (A > B) { Mark: "if body has been reached" m = B; } return (m); }</pre>

图8.1 插桩的初始例子

8.1.1 节点覆盖和边覆盖

需要进行插桩的最简单的标准之一是节点覆盖。显然，这个方法对于源程序的节点覆盖很有效，但是该方法的基本思路适用于任意图形。给图中的每一个节点赋一个唯一的标识符。创建一个序列并以节点号作为索引（称做nodeCover[]）。接下来，在每一个节点i，插入语句“nodeCover [i]++;”。

把序列nodeCover[] “持久化”是非常重要的，也就是说，每一个测试用例完成后必须将其保存到磁盘里，这就可以将多个测试用例的运行结果进行累积。执行完一些测试后，nodeCover[i]为0的每一个节点i均表示没有覆盖到。如果nodeCover[i]不为0，这个值就表示节点i被到达的次数。

这一过程参见图8.2。测试执行开始前必须读入节点序列nodeCover[], 如图8.2中节点1所示。每一个节点都有相应的序列表达式 (“nc” 是 “nodeCover” 的缩写)。一个自动覆盖分析工具可以将这样的序列插入基本模块的开头或者单个语句的前面。后者不如前者精确, 但易于执行, 所以多见于商业工具中。插桩表达式也可以插在源代码中, 然后编译成单独可执行文件或者类似Java ByteCode的中间形式。Java 反射 (Reflection) 技术也可以用来插入表达式, 尽管我们还不知道有没有工具使用了这项技术。

边覆盖的插桩技术比节点覆盖要稍微复杂一点。图中每条边都被赋一个唯一的标识符。创建一个序列并以边号作为索引 (称做edgeCover[]), 接下来, 在每一个边*i*插入语句 “edgeCover [i]++;”。

详细过程见图8.3的表述。和节点覆盖一样, 边覆盖序列edgeCover[]必须在测试执行前被读入。图8.3中的每条边用 “edgeCover” 的缩写 “ec” 表示。如果测试覆盖已由其他插桩表达式满足, 那么有时候会省略一些插桩表达式。比如在图8.3中, 任何使边3执行的测试也将使边5得到执行。

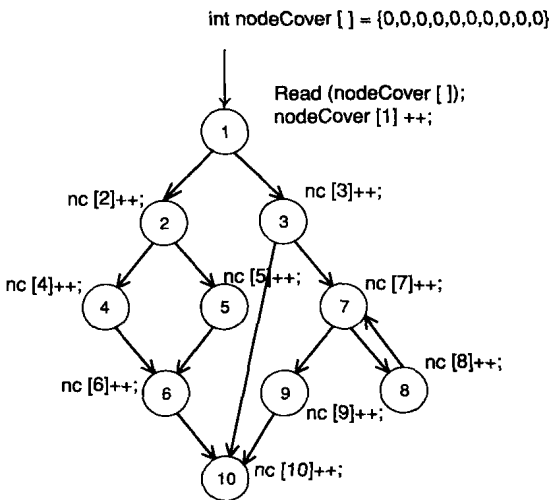


图8.2 节点覆盖插桩

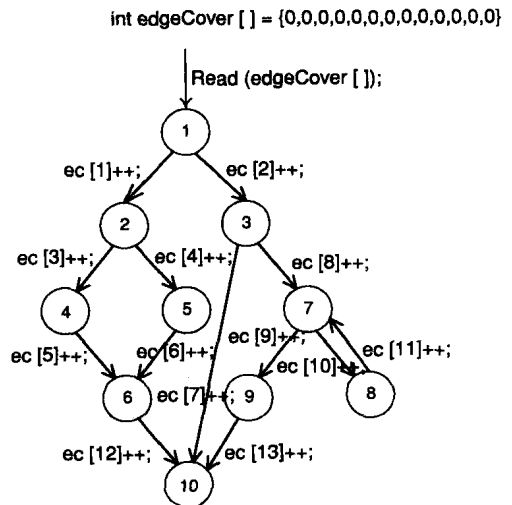


图8.3 边覆盖插桩

一些结构没有明确地标识所有的边。比如, 下列程序的源代码在else边中没有加入插桩序列的位置:

```
if (isPrime)
{ // save it!
  primes[numPrimes] = curPrime;
  numPrimes++;
}
```

因此else子句中必须明确地加进插桩语句, 以满足分支覆盖:

```
if (isPrime)
{ // save it!
  primes[numPrimes] = curPrime;
  numPrimes++;
}
else
  edgeCover[5]++;
```

8.1.2 数据流覆盖

数据流覆盖在某种程度上要比节点覆盖和边覆盖复杂得多。最主要的区别是基于程序中的两个位置：一个是定义，一个是使用。对于节点覆盖，每个节点需要赋一个唯一标号，但要使用两个序列。因为边也可能包含使用，每条边也必须赋一个唯一标号。这个方法是用一个序列追踪定义，而用另一个序列追踪使用。

在变量x的每一个定义位置，加入语句“defCover[x]=i;”，这里i是代表节点号。这意味着defCover[x]将存放变量最后一个定义的位置。第二个序列追踪使用，对一个变量x来说，useCover[]存放有关节点或边的使用。对变量x被使用的每个节点或每条边i，加入语句“useCover[i,x,defCover[x]]++;”。序列所在的位置useCover[i,x,defCover[x]]表示变量x在节点（或边）i被使用，从节点defCover[x]的定义到达位置i的使用。

图8.4展示了“All-uses”标准的插桩语句。在节点1处定义了变量x和y（通过参数传递或赋值），而y在节点2处又被重复定义。节点4和5处使用了变量x，节点6处使用了y。如果经过1、2、6这个路径，在节点6到达语句“useCover[6,y,defCover[y]]++;”，那么defCover[y]应该等于2，并将记录DU对[2, 6]被覆盖。然而，如果经过的是1、3、4、6的路径，那么到达节点6时defCover[y]应该等于1，所以将记录DU对[1, 6]被覆盖。

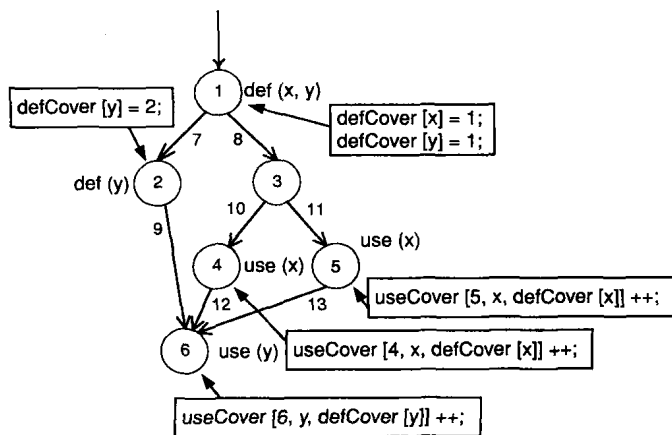


图8.4 All uses覆盖插桩

开始时，“所有使用”标准插桩比较难掌握。如果还有疑问，读者应该通过遍历一些简单的例子来验证useCover[]序列存放了所需的信息。执行完毕后，useCover[]序列记录的DU对为0则表示没有覆盖到，DU对非0的话表示覆盖到了。本书没有讲到的其他分析是需要决定哪些定义与哪些使用相匹配，以及哪些DU对有定义清纯（def-clear）路径。如果定义和使用对之间没有定义清纯路径，则会导致useCover[]序列的值为0。

8.1.3 逻辑覆盖

我们只展示如何为在第3章里讲述的一种逻辑覆盖标准进行插桩。其他覆盖标准都与此类似。

为逻辑覆盖标准进行插桩需要的插桩语句比结构或数据流覆盖中使用的插桩语句更多。大致的观点是每当遇到一个谓词，谓词的每一个子句都必须单独求值以决定谓词的哪个测试

需求被满足。求值的过程以独立的方法执行，这些独立的方法表示了特殊的序列来记录哪些测试需求被满足。

考虑一下图8.5a中的程序图。第一个谓词判定是 $(A \&\& B)$ ，它产生的测试需求是 (F, T) 、 (T, T) 和 (T, F) 。图8.5b给出在节点1处被插桩语句调用的方法。它在边 $(1, 3)$ 上执行谓词判定，并标记了序列 $CACCCover[]$ 来表示哪一个测试需求被满足。

第二个谓词判定是 $C \&\& (D \parallel E)$ ，其中有三个子语句，并且因为是“或”条件，一些选择可能在测试需求里，参见图8.5c。另一方面，求判定条件值，每次求值一个子语句，并记录下适当的测试需求覆盖。注意， $CACCCover[5]$ 在3个不同的地方都有记录。这表示了一个事实，那就是一个测试需求可以被三条为真的语句的任意一条所满足。

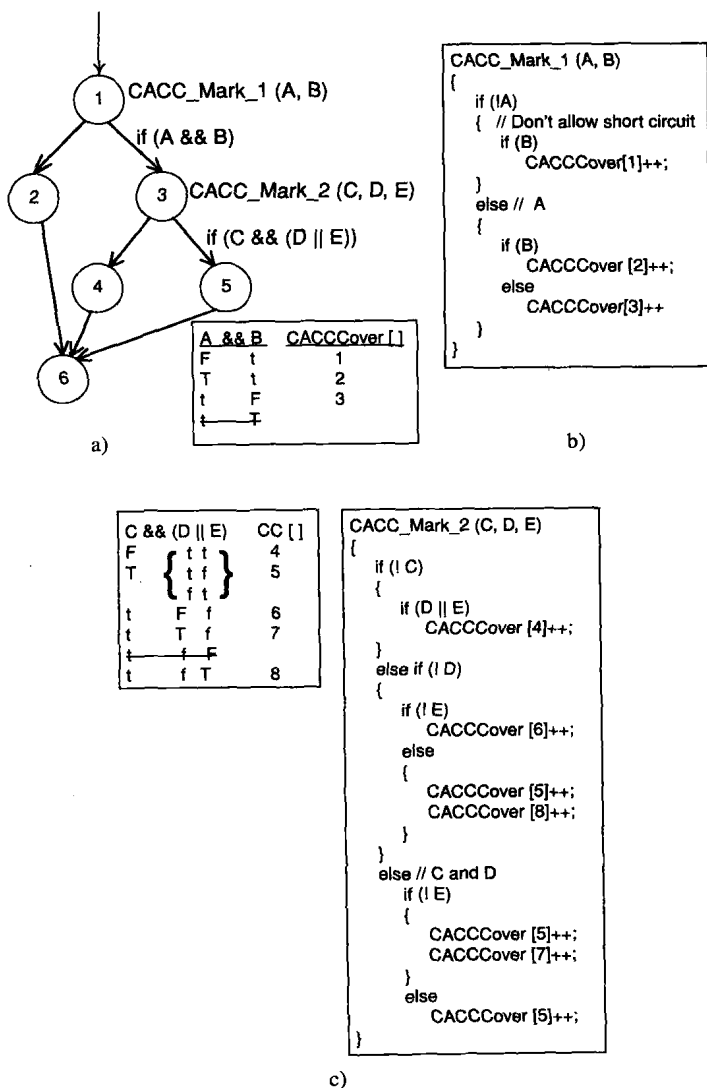


图8.5 相关活子句覆盖插桩

为ACC标准进行插桩的一个不同的方法就是，对每一个谓词判定，只是记录该谓词判定中每个子语句真值的组合。如果子语句有负面效应，或者如果谓词只依赖于短路求值，比如

在取用序列元素之前测试该序列的检索，那么这个方法并不适用。不过，这个方法的好处就在于ACC标准的满意度分析是从被插桩的源代码中独立出来的。因此标准分析引擎可以应用于从任何工件中搜集到的ACC覆盖数据。

8.2 构造变异测试工具

第5章描述了变异测试。变异测试广泛地被认定为最难满足的测试标准，而且以往实验研究已经持续地证实了从变异测试能发现错误的数量来看，它比其他的测试标准更强。但它不可能完全通过手工完成，因此自动化是必需的。当然，变异测试的自动化比其他自动化测试要复杂得多，仅仅把插桩语句添加到程序中是无法工作的。有关构造变异测试系统的文献很庞大，并且我们已经有经验，至少在研究领域是这样。幸运的是，语言设计的改变和工具水平的提高使变异测试系统的创建变得容易多了。本节探讨创建变异测试系统中的问题，并解释如何以适度、有效的方式构造一个变异测试系统。

对于变异测试系统，我们应该首先意识到的是它通常是一个语言系统。其中的程序必须被解析、修改并且执行，请参见图5.2。要创建变体，程序首先必须被解析，并且变体创建工具必须了解这门语言。同样地，等价检测器也必须建立在语言的语义基础之上。当程序运行时（“在P上运行T”），系统必须能够识别两种可能的情况，它们通常是异常行为，然而在变异测试中这些非常规的情况都是正常的行为。如果一个变体崩溃了，这对于变异系统来说是一件好事，而且变异系统应该标志这个变体已经死了。类似地，如果这个变体进入了一个无限的循环，这也标志着这个变体失败了。运行时系统必须能处理这些情况。

最初的变异系统是基于解释中间形式的。本节将展现该解释架构，然后指出了这种方法的问题。另外一个解决方案是编译架构，但这却带来了其他的问题。一个折衷的方法，即基于模式的变异，是目前如何构造变异工具的方式。

8.2.1 解释方法

在解释架构中，被测试程序首先被解析成一个中间形式。这个格式通常并不是编译器使用的标准的中间形式，而是一种专门用来支持变异的特殊用途的语言。创建变体组件直接修改这个中间形式来添加变体，并且这个变异系统包含了一个特殊用途的解释器。这个解释器可以轻松地处理变体被杀死时所产生的记录，并且可以对程序产生的错误作出回应。处理无限循环的常用方法是首先在原始的程序上运行一个测试脚本，计算中间指令执行的次数，然后在变体的程序上再运行这个测试脚本。如果变体程序运行了大于X倍数的中间指令数（X一般设定为10），那么这个变体就被认定为是一个无限循环，并且会被标志为死亡。

这种解释的方法有几个优点。对于整个执行环境都能控制是非常有益的，可以有效地解析程序，然后可以创建出足够的变体。通过对中间形式做很小的修改来创建变体是非常简单和有效的。单个的变体不需要被存储到硬盘上，只有那些需要改变中间形式的规则需要存储。

这个方法的难点是这个变异系统必须是一个完整的语言系统：解析器、解释器和运行时的执行引擎。它类似于创建一个编译器，但是从很多角度来说它更加复杂。因此，创建这样的系统是一项很重大的投资项目。另一个缺点是它运行起来比较慢，一个编译的程序的运行速度是一个被解释的程序的10倍左右。研究人员发现为运行完30行代码程序，他需要花费30分钟来运行完所有的变体。

8.2.2 分离编译的方法

分离编译的方法试图用更多前台的消费来节省后台运行的时间。每一个变体都由改变原始的被测的源代码并被创建为一个完整的程序来实现。然后每一个变体都被单独地编译、链接并运行。

这样的优势是比利用解释的方法来运行变体要快很多。但是，这样很难追踪哪个变体已经被杀死，而且很难处理运行时的失败和无限循环。这种分离编译的方法也会有编译瓶颈的问题，尤其是在处理大程序的时候。这种情况也会发生在快速运行小程序时，因为用来编译和链接的时候要比执行的时间多得多。而且，用分离编译的方法也很难应用于弱变异的情形。

8.2.3 基于模式的方法

“基于模式的方法”就是用来解决上述问题的，与变异产生一个中间形态不同，MSG（变异模式生成器）方法把所有的变异都编码成一个源代码级别的程序，称做元变异（metamutant），然后，该元变异程序被一个在开发过程中使用的相同的编译器编译（一次）并按照编译器的速度在相同的运行环境中执行。因为基于变异模式的变异系统不需要提供整个运行时的语义和环境，这些系统就比解释系统更加简单和更加容易创建了，并且更加轻便，容易迁移。因为这些额外的计算，MSG系统运行起来要比编译系统慢一些，但是比翻译系统要快很多了。

让我们来更细致地看看MSG是如何工作的。程序模式是一个模板，部分解释的程序模式在语法上类似于一个程序，但是它包含了自由标识符，称做抽象实体。抽象实体用来代替一些程序变量、数据类型变量、常量以及程序语句。一个模式是由一系列的抽象创建的。一个模式可以通过提供抽象实体的适当的代换被实例化为一个完整的程序。

为了变异，一个变体模式是通过使用抽象实体来代表程序中被变异修改的元素来创建的。一个变体模式含有两个组件，元变异和元方法集。两者都由语法有效的（即可编译的）程序构造表示。

作为一个例子，考虑算数运算符的置换（AOR）变异操作符。如果如下的语句出现在被测程序中：

```
delta = newGuess - sqrt;
```

它将会被变异并产生如下7个变体：

```
delta = newGuess + sqrt;
delta = newGuess * sqrt;
delta = newGuess / sqrt;
delta = newGuess ** sqrt;
delta = newGuess % sqrt;
delta = newGuess;
delta = sqrt;
```

这些变异可以被“一般”地表示为：

```
delta = newGuess arithOp sqrt;
```

其中，arithOp是一个元操作符的抽象实体。这个抽象可以由如下的元方法来实现：

```
delta = arithOp (newGuess, sqrt, 44);
```

arithOp()方法执行了一个算数操作，第三个参数，“44”表示了程序中这个元方法被调用的位置。这个元程序接收一个告诉它执行哪个变体的参数。元方法检查这个全局参数，如果这

个变体是一个AOR变体，位置在第44行，元方法就会执行适当的算数操作，否则它会返回原始的表达式（`newGuess-sqrt`）。

8.2.4 使用Java反射机制

一种方法是使用反射机制，它利用现代语言融合基于解释和编译的方法。反射机制允许程序存取它内部的结构和行为，并且操纵这个结构，从而可以基于外部程序提供一定的规则来修改它的行为。反射机制只能在支持它的语言中使用，如Java和C#语言。它们都支持反射机制并且允许存取中间形式，如Java字节码（Java Bytecode）。反射机制有三种不同的方式。编译时反射机制允许程序在编译时被修改。加载时反射机制允许程序在加载到运行系统（Java的JVM）时被修改。运行时反射机制允许程序在执行时被修改。

反射机制在实现变异分析方面是一个很好的方法，其原因主要有以下几个方面：

1. 它允许程序员通过提供一个代表类定义的逻辑结构的对象来提取出有关类的信息。这意味着变异系统不需要解析程序。
2. 它提供了API以便在程序运行时来修改程序，可以用来创建程序变异的版本。
3. 它允许动态地实例化对象和调用方法。
4. 一些面向对象的运算符不能用MSG来实现。例如，第5章的隐藏变量删除（HVD）要求删掉变量声明，这些变量隐藏了一个祖先变量。这样会影响对该变量每一个引用，因此不能用变体模式实现。

Java提供了一个内嵌的专用的API的反射机制。允许Java程序来执行功能，比如询问给定对象的类名、查找该类中的方法以及调用这些方法。但是，Java不能支持完全的反射功能。特别是，Java仅仅支持自省功能，提供自省数据结构的能力，但是不能直接支持对程序行为的修改。有些反射系统已经创建并且可以支持编译时和加载时的反射机制。

Java字节码翻译和反射类似，但是用的是一种不同的方法。字节码翻译审查并修改Java程序的中间代码、字节码。因为字节码翻译是直接操纵字节码，所以它比分离编译和MSG方法有优势。首先，它可以处理现成的程序或者库函数，而不需要它们的源代码。第二，它可以在程序加载时运行，即当JVM（Java Virtual Machine）加载一个类文件的时候。Java字节码翻译无法像运行时反射那样高效，但是这种方法目前更加稳定。

8.2.5 实现一个现代的变异系统

现代的变异系统将会使用MSG来实现不影响程序架构的变体，并利用反射变体机制来实现影响程序架构的变体。虽然现代的变异系统比插桩语句覆盖或分支覆盖更为复杂，但是编程量要比解释系统少很多。这是在第5章提到的muJava系统使用的方法。

8.3 参考文献注释

我们不能找到任何有关如何进行插桩的出版发行的参考资料。然而研究人员和工具开发者很多年前已经开始利用我们所讲述的插桩技术。

PIMS[5, 52, 54, 211]，一个早期的变异测试工具，率先将其一般流程用到变异测试中，来创建变体（在Fortran IV程序中），接受用户测试用例，并且在变体上执行测试用例来确定多少变体被杀死了。

在1987年,相同的流程(添加测试用例、运行变体、检查测试结果,然后重复)被采用并延伸到Mothra变异测试工具包[96, 101, 262, 268],这个工具包提供了一系列的工具,每一个工具都可以完成一个单独的、分离的任务来支持变异的分析和测试。虽然,继Mothra[95, 97, 220, 330]工具之后,也有其他的变异测试工具被开发,但是Mothra仍然是目前最著名的变异测试工具。

变异测试的许多推动者都是致力于解决性能成本问题。他们经常都追求三条策略之一:做得少、做得巧、做得快。

做得少的方法是试图运行尽量少的变体程序,而且不会导致不可接受的信息丢失。变体样本[4, 53, 356]利用了变体的随机样本,并且是最简单的做得少的方法。举例来说,变体程序中10%的样本,只比在一个全集中确认错误检测效率低16%的精确度。后来,经Wong[356]调查,以5%为步长从10%~40%递增改变样本比例的影响效果。由Sahinoğlu和Spafford[106]提出了另一种采样方法,它不需要某种先验的固定大小的样本,而是基于一种贝叶斯序列概率比例测试,选择变体程序直到收集足够的证据以便决定已经获得了统计意义上合适的样本大小。

Wong和Mathur建议选择性变异的思想,仅将变异测试应用到最关键的、被使用的变异操作上[230, 357]。这个想法后来被Offutt等人[269]进一步发展,他们用Mothra为Fortran-77定义了一套选择算子。结果表明选择性变异测试与非选择性的测试几乎达到了相同的测试覆盖。本章中讲述的这些算子是基于选择性方法的。

使用非标准计算机架构已经被看做是做得巧的方法。这种方法将计算代价分布到若干机器上。有许多工作已经使变异分析系统适应于向量处理器[229]、SIMD机器[192]、超立方体(MIMD)机器[75, 275]和网络(MIMD)机器[365]。因为每一个变异程序都是独立于其他变异程序的,所以通信消耗是很低的。至少一个工具对于中等规模的程序函数几乎可以达到线性提速[275]。

弱变异[167]是另一种做得巧的方法。弱变异是一种近似技术,在执行完程序的变异部分后立即比较变体和原始程序的内部状态。实验表明弱变异可以产生接近强变异的测试用例,并且可以节省至少50%甚至更多的执行时间。用来作为Mothra的部分实现的Leonardo系统[270, 271],主要做两件事情。它实现了一个运转的弱变异系统,可以很容易地同强变异系统进行比较,并且通过允许在变异组件后的四个不同地方的比较对于范围/强度概念进行评估:(1)在第一次给包围变异符号的最内部表达式赋值之后;(2)在第一次执行变异语句之后;(3)在第一次执行含有变异语句基本块之后;(4)在每次执行含有变异语句基本块之后。(一旦检测出非法的状态,执行立即终止。)

在另一种“做得巧”的方法中,Fleyshgakker和Weiss描述了一些算法,这些算法以增加空间复杂度为代价,来改善通常变异分析系统的运行时复杂度[120]。通过智能地存储状态信息,它们的技术是将执行一个变体的开销分到多个相关的变体执行上,这样就降低了整体的计算消耗。

在分离的编译方法中,每一个变体都是单独地被创建、编译、链接并且执行。Proteum系统[95]是分离编译方法的一个案例。当变体的运行次数明显大于单个编译/链接次数时,基于该策略的系统会比解释系统执行快15~20倍。然而,当这种条件不满足时,可能会导致编译瓶颈[75]。

为了避免编译瓶颈，DeMillo、Krauser和Mathur开发了一种编译器-整合程序变异模式用来避免过多的编译瓶颈的开销，并且可以执行编译代码[97]。利用这种方法，被测程序由一个特殊的编译器编译，在编译过程中，变异的影响被记录下来，代表这些变异的代码补丁也会相应地准备好。这样，运行一个特定的变异仅仅需要相应的代码补丁在执行前被调用。打补丁是便宜的，而且变体的执行是按照编译的速度进行的。然而，制作这样一个特殊的编译器是非常昂贵也很困难的。

Untch为变异开发了一种新型的执行模型，即变体模式生成（MSG）方法[330]。程序模式是一个完整的、可以编译的程序，该程序把所有的变体集成编码到一个元程序中。这是目前的变异的研究状态[277]。

面向对象变异操作符被设计用来测试主要的面向对象语言功能，如继承、多态及动态绑定[11, 219, 281]。在muJava（变异Java测试）中使用的变异操作符是通过若干篇研究论文发展出来的，论文包括Kim、Clark和McDermid[184, 185]，Chevalley和Thévenod-Fosse[69, 71]，Alexander等人[8, 266]和Ma、Kwon和Offutt[219, 220, 221]。MuJava采用OpenJava[324, 325]（开放Java）的编译时反射机制，因为该功能为生成变体提供了足够的信息，并且使用起来很简单。

第9章 软件测试中的挑战

我们用对软件测试中的三个挑战领域的讨论来结束本书。虽然研究人员多年来都对紧急性属性很有兴趣，但是这一领域还远没有被解决，其对行业的重要性也在持续提升。同样地，因为某些较新的软件技术的特征，可测性正在重新引起人们的注意。最后，我们会在实践和研究这两个领域为软件测试提出一些建议。

9.1 测试紧急性属性：安全性和保密性

测试紧急性属性（emergent properties）带来额外的挑战。本节为测试强调安全性（safety）和/或保密性的系统的工程师提供高层的指导。

紧急性属性作为将组件集中到单个系统的结果出现。它们不在任何特定的组件中独立存在。安全性和保密性是系统设计中典型的紧急性属性。例如，一个飞机的整体安全性不是由控制软件单独决定的，也不是由引擎或者其他任何组件单独决定的。虽然一个特定组件的个体行为对全局的安全性非常重要，但当装配成一架完整的飞机时，所有这些组件的交互决定了飞机的整体安全性。换句话说，仅考虑一个飞机引擎，既不能说它安全，也不能说它不安全，因为一个飞机引擎自己并不能飞。只有完整的飞机才能飞，因此只有完整的飞机才能在飞行方面被认为是安全的或者不安全的。同样地，一个网络应用的保密性既不是由后台数据库服务器的保密性单独决定的，也不是由一个代理服务器或者使用的加密系统单独决定的，而是由所有这些组件之间的交互决定的。

系统安全性和保密性的重要性已远比它们初次出现时普遍，而且因为各种技术和社会原因它们正变得更加普遍。例如，考虑一个空调控制系统的保密性。一个传统的空调控制系统只能通过物理方法控制，而不能远程处理。显然，如果可以从一个中央位置远程控制几个这样的系统更便宜，并且使用网络建立这些空调控制系统到中央控制器的连接也容易实现。然而，这样的方式使系统面临来自世界各地的恶意攻击的风险。此外，如果中央控制器的开发者在系统的初始设计期间没有考虑保密性，即使后来在提高保密性方面付出大量的努力，最终产品也未必足够安全。

更一般地说，三个基本的主题使安全性和保密性的问题变得普遍。首先，是在上面例子中用到的连通性（connectivity）。网络是无处不在的，这对许多想要利用这种“免费”连通性的应用有很强的吸引力。其次是复杂性（complexity）。网络的、分布式的软件难以很好地构建，更难以访问。第三是扩展性（extensibility）。应用的某些部分总是在改变，它们就像是“飘浮在空中”，这意味着当一个测试工程师要做评估时，它们甚至还不能被很好地定义。

分清测试安全性和保密性的功能和测试安全性或保密性是很重要的。前者基本上和测试其他功能没有两样。后者则将注意力放在可能出现的恶意行为。例如，如果一个控制系统有一个紧急关闭（emergency shutdown）的功能，想要用于操作员确认为危险的情况，那么这就是一个安全功能，测试工程师会以和其他系统功能相同的方式来评估它的功能。又例如，在安全环境中，考虑一个使用用户名和口令机制来认证用户的功能。测试工程师以和其他系统

功能相同的方式访问认证功能时,测试工程师需要考虑以下情况:存在紧急关闭功能应该被调用而没有被调用的情况,或者有人能不通过调用认证功能就成为一个认证用户的情况。到这里,读者应该认识到处理后两个问题要难得多,因为它们迫使分析师拿出某些坏事不会发生的反面证据,而不是某些好事会发生的正面证据。这后两个例子分别描述了测试安全性和保密性的问题,本节的剩余部分会主要关注这类问题。

粗略地说,如果一个系统可以很好地免于无法接受的危险,那么它是可靠的;如果一个系统面对恶意威胁时相当强健,那么它就是安全的。对于这些粗略定义,文献提供了有价值的精细描述,但它们并不适用于本节所使用的材料的等级。我们无法在安全性和保密性方面做到完美,理解这一点是很重要的。更确切地说,这是一个持续进行的过程:鉴别评定危险和威胁,选择以适当方式描述这些危险和威胁的需求,然后选择承诺满足这些需求的设计和实现。有时这一过程也会失败,在这样的情况下,只有在安全性和保密性方面做出巨大让步才能完成系统。或者说,最好不要构建这样的系统。

众所周知,上面描述的过程是理想化的,在实际中常常不是这样。不幸的是,当测试工程师面对这样一个系统的测试任务时,将发现无论如何都有必要执行这一过程,否则将不知道他在测试什么或者为什么要测试。更糟糕的是,由于属性的涌现本质(emergent nature),测试系统的安全性和保密性基本无望。换句话说,这些属性没有在组件中定义,如果设计中也并没有提供,那么它们只会在系统中偶然出现。不幸的是,靠瞎碰运气是不能解决问题的。

使事情变复杂的是许多安全性和保密性需求中的技术难点。安全性的情况中,定量需求可能会很严格,以至于它们超出了工程实践的边界。考虑经常被人们提起的关于商用飞机的紧急安全软件的需求,即每10小时的飞行时间 10^{-9} 的失败率。Butler和Finelli写了一篇很好的论文来解释为什么满足这种“极端的”需求是不可能满足的,不管使用什么样的开发和测试技术,不管是现在还是将来。与其满足不可行的定量需求,不如采用一般的方法,即使用定性的“安全用例”。不要吃惊,这样的定性方法主要依靠合理的开发流程。确保安全性是困难的,确保保密性更加困难,因为它被人类的聪明复杂化了。恶意的人类行为很难预测,更不用说阻止了。

那么,我们该如何测试这样的系统呢?

从上面的讨论中学到的第一课是,测试工程师必须对要测试什么有一个清晰的认识。描述安全性和保密性需求的清晰文档,以及关于危害和危险的链接,对做出决定非常重要。一些安全性和保密性需求是不可测试的。例如,考虑开发中插入的“后门”代码问题。系统测试根本没有机会发现这段代码,因为关于怎样打开后门,恶意的设计者有很多的选择。换句话说,将测试资源投入到这种威胁上是对资源的浪费。相反,这种威胁必须通过流程、个人实践或者检验来处理。

一个更值得投入测试资源的地方是在安全性和保密性模型中探寻潜在的假定问题。每个设计都依赖于这样的假定,因为假定可以完全一致地回溯到对危害和危险的评估。例如,Arianne 4火箭的设计者在设计时会火箭在飞行早期可能经历的位置、速度以及加速度做出假定,而这些假定会被用在控制代码的异常处理中。这些假定对Arianne 4是有效的,但遗憾的是,对重用了大部分控制软件的Arianne 5无效。从测试的角度,掌握这样的假定信息是非常有价值的,这意味着文档化这些假定是一个关键的过程。

对于测试保密性,假定也是很丰富的资源。考虑从网络下载软件到设备上这个常见的问

题。一个策略是通知用户即将执行下载操作（及其固有的风险），允许用户接受或拒绝。这一策略至少混合了三个假定：首先，用户可以做出一个明智的决定；其次，用户愿意做出一个明智的决定；第三，用户可以说“不”。许多手机依赖于第三个假定，但在一个恶意的对手能以很高的频率呼叫用户的环境中，它很容易被破坏。特别是，如果一个手机带有病毒，它可以请求附近的手机自己下载该病毒，如果附近的手机用户回应“不”，则立即再次请求，那么用户就遭到了经典的拒绝服务（DOS）攻击，手机用户在无奈下可能最后不得不点“是”，此时手机就会影响病毒。注意，这里的安全问题在手机的组件层是不固定的，甚至不能很好地定义，但却在手机、手机网络以及用户（包括恶意用户）中涌现出来。此外，处理这样的问题需要对手机之间交互的基本规则重新评估。

软件的重用是一个先前的假定可能被破坏的经典领域，因此测试是必要的。以网络软件为例，在一个典型的网络应用中，网络服务器中嵌入的业务逻辑会保护后台应用（如数据库），防止恶意用户输入。但是如果这个网络应用被重新部署为一个网络服务，网络服务器将可能被移除，这时后台的应用又再次容易受到攻击。

9.1.1 紧急性属性的测试用例分类

之前讨论的目的是描述一些测试紧急性属性的基本问题。这里我们提供一些选择测试用例的常用策略。

1. 开发误用用例。用例是列举软件的期望用法的通用建模工具，主要应用于需求分析阶段。误用用例（misuse case）也是这样，只是它们探索的是计划外的用法。需求阶段是考虑这样的场景的理想时间，因为它是去除或缓和它们的正确地方。

2. 识别假定，然后设计破坏它们的测试用例。即使做了误用用例分析，变化的假定还是会有剩余。正如上面所讨论的，测试这些是非常有价值的。

3. 识别配置信息，然后设计检查它们的测试。在使用版本不一致的组件来实现系统时，配置信息是一个会出现问题的常见领域。除非产品组件明确知道它们的配置数据，开发过程往往要比产品自身需要更多的配置测试。

4. 开发无效输入测试。无效输入是安全测试中非常重要的领域。简单的无效输入攻击如缓冲区溢出(buffer overflow)和查询注入(query injection)攻击造成了很多安全漏洞。每次当系统从环境中搜集数据时，这些数据都可能违反对它的大小或内容的假定。幸运的是，生成测试数据来探寻输入数据的限制很容易自动化，也容易用第5章中的技术来理解。

评估安全性和保密性中的另一个复杂因素是开发者和用户之间的差异。自然，系统由开发者构建，因此开发者为系统的安全性和保密性提供准备。但是，系统由用户使用，如操作专家和网络管理员。开发者和用户通常在应用领域有着非常不同的技能集合、培训等级和思考模型。开发者关注软件工件，特别是代码。相反，用户往往只模糊地知道他们的系统中有软件，他们只关注适用于他们工作的部分。我们需要系统可靠和安全，不仅要在开发者所构想的理论上，而且还要在用户使用的实践中。因此，当考虑安全性和保密性的测试时，测试工程师应该将他们的观点向使用环境偏移，而不是开发环境。

9.2 软件的可测试性

软件的可测试性是一个区别于软件测试的重要概念。一般而言，软件可测试性是条件概

率的评估或者度量,即假定一个给定的软件工件中包含错误,通过测试发现这个错误的可能性有多大。我们都熟悉这样的软件开发项目,即使做了丰富的测试,后期使用中还是会出现错误。可测试性的核心是:在使用了很好的测试包的情况下,错误仍然可以不被发现的难易程度。

学习可测试性有各种原因。为一组软件工件给出可测试性评估,测试工程师有如下选择:

1. 对低可测试性的工件,测试工程师知道仅仅测试可能不能很好地检验工件是否满足需求,所以测试工程师需要采用替代的验证方法。对于关键的软件,这可能意味着形式化的分析。在更多普通的软件中,常常采用复查设计和代码的方法。

2. 低可测试性的工件能以各种方式被替换,以改进可测试性。虽然我们将在本节后面才谈到技术细节,但是任何能改进可测试性的方法都值得研究。

3. 测试工程师能着手测试高可测试性的工件,确信测试结果准确地指出了工件的质量。

4. 万一可测试性低且难以改进,也没有可行的替代的验证方法,测试工程师会有切实的证据提供给管理部门,表明该工件存在过多的风险。

不同的作者提出了不同的处理可测试性的方法。一些作者关注于软件的可观察性和可控制性,将它们作为可测试性的关键组件。我们开始在第1章的缺陷、错误和失败模型上构建一个可测试性模型。

我们用第1章的RIP模型的三个属性来定义可测试性:可达性、影响和传播。我们假定一个工件包含缺陷,而那个缺陷一定存在于工件中某个位置。指定位置的缺陷导致了一个失败,一定会发生三件事。首先,执行必须达到缺陷的位置。其次,缺陷的执行必须在指定位置导致程序状态受到影响,即错误。最后,错误必须从该位置传播到工件的输出。指定位置的灵敏度是即是可达性、影响和传播发生在该位置的可能性。

注意,我们没有指定一个缺陷在指定位置看起来应该是什么样。读者可以想象所有该位置的错误平均情况或者最坏情况的行为。一个工件的可测试性被定义为工件中所有位置的灵敏度的最小值。

要阐明灵敏度是可能性的观点,有必要将输入建模为一个或多个使用分类(usage distribution)。换句话说,我们假定测试用例是从某种概要(profile)中抽取的样本。实际上,定义使用分布非常容易,这样的测试方法得到了很多人的支持,可以在目录部分找到引用。

为所选的测试输入给定一个分布,衡量任何特定位置的可达性概率是一种直接的事务。此外,如果认为特定位置的可达性概率太低,很容易改变输入分布,将测试输入的更大部分投入到该位置。更直接地说,可测试性的可达性属性由测试工程师直接控制。另一种思考方式是将这种属性认为是可控制性。低可控制性的软件无论怎样改变输入分布,都可能具有低的可达性概率。

评估影响概率有点麻烦,因为它需要采用一个缺陷模型和一个影响模型,这两个模型都必须适合问题中的位置。一种选择是缺陷模型利用变异分析,通过观察候选的程序状态上的变异结果来衡量影响。

确定传播概率也涉及一定数量的建模。一种选择是利用扰动模型(perturbation model)来改变问题位置的程序状态,然后观察是否有影响持续到输出。另一种思考方式是将这种属性认为是可观察性。低可观察性的软件可能具有低传播。但是,可以通过重新设计代码来得到更高的传播。

首先,需要考虑输出空间的维数。例如,考虑只有一个布尔输出的方法。这种方法往往具有低可测试性,因为如果没有其他原因,任何影响的传播都被整个内部状态逐渐限制到单一的布尔输出。测试工程师可以和系统设计师交互,看看是否可以增加输出,从而更好地访问内部状态,并因此更好地传播。

其次,有断言检查的概念。断言机制将是内部状态的不一致性转化为可观察事件的理想机制,通常是通过一个异常机制。设计者可以提供广泛的内部状态检查来协助这一过程。这样的检查无须在运行部署期间启用。从可测试性的角度看,只在测试期间启用它们是足够的。

9.2.1 常见技术的可测试性

面向对象的软件为可测试性提出了特别的挑战。主要原因是对象将状态信息封装在实例变量中,而这些实例变量通常不能直接访问。考虑一个栈的简单例子。`push(Object item)`方法改变了栈的状态,可能是将`item`保存在一个实例变量`Object[] elements`中。`top()`方法给出访问栈中最新元素的方法,因此紧随`push()`的`top()`使测试工程师能验证`top()`返回的元素确实是最新的元素。但是,对更早的元素的访问是隐藏的。典型的栈接口不允许,也不应该允许对这些元素的直接访问。从可观察性的角度看,栈的接口对测试工程师是个问题。许多类远比栈类复杂,所以我们对面向对象的软件往往具有低可测试性就不用感到太惊讶了。

继承也导致了面向对象软件中可测试性的缺乏。继续栈的例子,假设一个子类记录了`push()`调用的数目。为了实现子类,程序员通常会覆盖`push()`方法。在新的`push()`方法中,先增加记录,然后调用超类的`push()`方法改变状态。这种类的改变是面向对象的开发中的惯例。从可观察性的角度看,子类的可测试性遭遇的事实是,某些被更新的实例变量甚至不在被测试的子类中。实际上,子类的开发者很可能无法访问超类的源代码。

有两种基本的方法来改进可观察性。第一个方法需要开发者提供额外的`get`方法,以便测试工程师能访问全部状态。在并行或提前给类生成大量测试用例的情况下,虽然这可能是一种合理的方法,但是很可能测试工程师最后仍然要测试这样的代码,无论任何原因,开发出来的软件不符合可测试性指南。

第二种方法是使用工具,利用某些像Java的反射机制那样的东西去访问内部变量,从而不依赖于是否可以得到源代码。另一方面,这种方法遭遇的事实是,对测试工程师而言,解释捕获的数据值不是一项简单的任务。

测试的维护也是一个问题,因为每次开发者改变了类的实现,访问那些类的内部状态的测试就未必能执行,更不用说正确地执行了。

网络应用对可测试性也提出了一组不同的挑战。同样地,对一个典型的网络应用,其可控制性和可观察性都可能是非常低的。为了说明这种情况,考虑一个典型网络服务器的架构。其中,单个代理和单个的客户端交互,一个会话管理器监视整个客户端池,应用逻辑决定如何处理客户端请求,数据在后台数据库中流入和流出。从客户端的角度看,几乎所有这些架构都是不可见的,因此从客户端访问大部分状态是不可能的。服务器很可能是分布的,不仅跨越了多个硬件平台,还跨越了多个公司组织。为这样一个环境引入高可测试性仍然是一个研究课题。

9.3 测试标准和软件测试的未来

当作者在20世纪80年代中期开始他们的事业时，软件工程领域和今天完全不一样。很大的差异体现在经济学方面。在20世纪80年代，软件工程的经济学是这样的，应用尖端的测试标准的成本通常超过了经济利益。这意味着软件开发组织很少有动力去测试他们的软件。

这种情况和软件如何销售和部署有很大关系。大多数软件是捆绑在电脑上——软件的成本包含在电脑里，用户只有很少的选择。其他软件则通过“收缩包装”（shrink-wrapped）购买，即软件被密封包装，用户购买和支付时没有机会试用。另一个销售软件的主要机制是通过“合同”，用户和供应商签约，要求提供某种特定软件。迄今为止这个时代最大的买家是以美国国防部为首的美国政府。虽然始终试图让软件供应商负起质量责任，但系统仍然有许多问题。从一个广泛的角度看，当发现问题时，供应商常常可以得到额外的资金来修复软件。事实上几乎没有测试标准和质量保证，需求在开发期间也频繁地改变，这使得很难让开发商为质量问题和费用超支负责。

当然，这些问题并没有涵盖整个领域。我们一直需要高安全性的软件，如保密性至关重要的软件。但是，这部分市场在20世纪80年代时很小，而且也没有足够多的支持工具供应商或者广泛的软件测试教育。

在我们职业生涯中的大多数时间，软件市场始终是供应商和买主占主导地位。这导致了一个缺少竞争的市场氛围，相应地也几乎没有质量保证和测试的动力。也就是说，软件测试成为了需要花钱购买却基本使用不到的技术。

从更积极的方面看，该领域已经有显著的改变。软件市场更大了，更有竞争性，有了更多的用户，我们正在更多的应用中使用软件。这一变化的一个主要推动力是网络的发展。万维网为软件部署提供了不同的方式。不是购买绑定了软件的电脑，不是从商店购买压缩包装的CD，也不是雇用程序员定制软件，现在用户可以通过网络运行部署在服务器上的软件。这使市场更有竞争性——如果用户对一个网站上的软件不满意，他们可以很容易地“用鼠标投票”，转到一个不同的站点。网络带来的另一个好处是更多的用户可以获得更多的软件。这样一个增长的市场也带来了相应增长的期望。受过高等教育、精通技术的用户可以容忍很多问题。蓝屏死机对一个工程师来说是很熟悉的情况：系统进入了错误状态，我们需要重启电脑。但是，重启任务和其他补偿行为并不能被更广泛的用户群体所接受。当亿万并不确定要购买软件的用户可以使用软件时，他们难以容忍错误。如果Amazon或Netflix送错了产品或将产品送到错误的地点或错收了用户的钱或在使用时很容易被冻结，他们会成功吗？在网络和电子商务应用中，效率和产品投入到市场的时间不是非常重要。Google相对于早期的搜索引擎的成功就是关于这一事实的一个众所周知的例子，但对于很多小型企业这也是真理。正如在我们的课程中所说的，在网络上“迟到但做得好比早到但做得坏要好”（磨刀不误砍柴工）。

软件质量和安全性的需求会持续增长还有其他原因。在20世纪80年代我们只有少量的嵌入式软件应用。大多数是高端的，有特定目的而且很贵。设备和软件都由专门的人构建，通常是在军事或航天工业中。而我们现在被嵌入式软件所包围。制造手机的公司不仅把自己看做是手机提供商，还是将他们的软件放到特定设备上的软件公司。许多其他的电子设备上实际也包含软件：PDA（掌上电脑）、移动音乐播放器、相机、手表、计算器、电视盒、家用无线路由器、家用遥控器、车库开门器、冰箱和微波炉。甚至连烤面包机也有了软件控制的传

感器。新的轿车中到处都是传感器和软件——自动打开的门，安全气囊的传感器，自动移动的座椅，探查轮胎疲劳度的传感器，以及自动停车。所有这些软件都需要工作得很好。当我们将软件嵌入到一个器具中时，用户期望软件比器具更可靠。

有一段时间，安全都是关于加密数据的精巧算法。后来安全变成一个数据库问题，然后是一个网络问题。今天大多数软件安全弱点是基于软件缺陷。软件必须高度可靠和安全，任何使用网络的软件都是容易受到攻击的。

我们该怎样为这些应用开发软件，让它们按我们的需要可靠地工作？答案的一部分是软件开发者需要更多和更有效的测试。本书中的大多数标准和技术都已流传多年。在计算机科学和软件工程课程中传授这些概念的时机已经成熟，结合更多和更好的软件测试工具，最重要的是用于工业。

9.3.1 继续进行测试研究

本书陈述了看待软件的一种不同方式。不是将软件测试看做贯穿开发过程的使用不同的标准的指导（即单元测试、组件测试、集成测试和系统测试都各不相同），本书陈述的观点是在软件的4个模型（图表、逻辑表达式、输入空间和语法）上定义测试标准，而模型可以开发自任何软件工件。即一幅图表可以创建自单元（方法）、模块、集成信息或者整个系统。一旦创建，相同的标准就被用于图表，无论这个图表来自哪里。

本书在4种结构之上陈述了总共36种标准。许多出现在文献中的其他紧密相关的标准并没有包含在本书中，因为我们觉得它们不太可能在实践中用到。过去30年的测试研究大多关注发明新的测试标准。我们的观点是该领域已经不需要更多的标准，但它确实需要几样其他东西。

始终需要为新技术和新情况研究已存在的工程标准。第7章通过展示第2章到第5章中的标准是怎样被工程师应用到面向对象的软件、网络应用、图形用户界面以及实时嵌入式软件中描述了这点。计算技术将继续发展，向测试研究者呈现新的有意思的挑战。我们希望本书中表述的标准能为这类研究提供基础。

开发管理者的一个主要问题是决定使用哪个测试标准。事实上，这个问题拖延了我们对改进的测试技术的采纳，特别是当采纳的成本很高时。从第2章到第5章都有一个包含图，指出标准间的包含关系。例如，如果我们满足了一幅图表上的边覆盖，那么也一定满足了节点覆盖。此信息只针对部分问题和一小部分实际的软件开发者。假如从可以找到更多缺陷的意义上“subsumes”意味着“更好”，则一个开发经理想知道“好了多少”和“贵了多少”。虽然发布了几十甚至上百的论文来陈述测试标准的实验比较，我们仍然远没有足够的知识来告诉一个开发经理用哪个标准和什么时候用。

这个问题的一个扩展是怎样比较那些不能在subsumption中比较的测试标准的问题。这种比较一定是自然的实验，而且几乎可以肯定的是，它需要被复制。例如，数据流标准（见第2章）和变异标准（mutation criteria）（见第4章）在subsumption方面是不可比较的。在20世纪90年代发表的几篇论文将程序转化和一个或多个数据流标准作了比较。虽然单独地看任何一篇论文都不具有充足的说服力，但是将它们合在一起却可以向研究者们证明变异测试可以找到更多的缺陷。但是，这些研究中没有一个是看到了大规模的工业软件，也没有多少关于成本（超出观察的变异通常需要更多测试）的有用信息。

另一个主要问题是怎样使这些标准完全地自动化。大多数早期的关于单个标准的出版物以及很多后来的论文包含了算法、工具开发和效率改进。第8章介绍了工具构建，但需要注意这些概念还只限于实验室研究中。商业的测试工具必须比研究工具更强健、更有效率以及有更好的用户界面。虽然数十年来公司都在尝试销售基于标准的测试工具，但也就只是最近的一些公司如Agitar和Certess取得了商业成功。认为测试对软件公司的经济成功很关键的观点表明软件测试工具的市场处于增长模式。

要在实践中采用任何新技术，都面临一个重要的问题，即它怎样影响当前的开发过程。让开发者能轻松地将测试技术整合到他们已有的过程中，这是成功的关键因素。如果测试工具能被整合进编译器或者集成开发环境（IDE），如Eclipse（就像Agitar所做的），那么工具将更容易被开发者接受。

自动化方面的一个突出问题是测试数据的生成。自动的测试数据生成从很早开始就一直是一个研究课题，但离有用的商业工具仍然很遥远。不是对这一问题缺乏研究的兴趣，而是问题实在太复杂。一些最早的软件测试的研究论文就关注了测试数据的自动生成，事实证明要取得进展很难，以至于大多数研究者转而研究更容易管理的问题。一些早期的研究工作关注随机生成测试输入，虽然在实现上比其他方法简单，但随机测试数据生成有两个问题。第一个如Dick Lipton所说，“缺陷不是鱼”，也就是说，缺陷并没有像池中的鱼那样不一致地（随机地）在程序中分布。任何程序员都知道，缺陷往往集中在程序中的复杂部分或者输入空间中难以定义的区域。随机测试数据生成的第二个问题是结构化数据。随机生成数字和字符很容易，但当数据存在某些结构（地址、产品记录等）时，随机生成就变得更难了。

最复杂的想法是分析程序源码，使用程序的详细信息来自动生成满足测试标准的测试数据。最一般的分析方法是符号赋值（symbolic evaluation），通常与限制表达（constraint representation）和不时切片（sometimes slicing）结合在一起。一些分析技术是静态的，而另一些是动态的。通过确切的解决方案如约束解决（线性编程）和不确切的特定目的启发，已经解决对实际输入值的平衡和约束。基于分析技术的重大问题是所谓的“内部变量问题”、循环和不确定的内存引用（如指针、数组和别名）。内部变量问题指这样的事实，我们可能对测试输入有特定的需求，如“在第5306行X必须比Y大”，而X和Y不是输入。查找间接控制X和Y值的输入是很麻烦的。动态分析技术是最有希望解决这些问题的方法，但是这些问题正是为什么基于分析的技术主要用在细小的结构上（如程序单元）的原因。

最近的自动测试数据生成方法使用了搜索技术，如遗传算法。基于搜索的方法有几个优点，包括易于实现和非常灵活，它们不像基于分析的方法那样需要很多信息。例如，如果利用基于搜索的技术在一个方法级别的控制流图中来达到边覆盖，一个像 $x+y=z$ 这样的谓词很难用基于搜索的技术表示。如果要达到方法的一个部分必须满足几个这样的谓词，问题会更复杂。因为这个原因，基于搜索的技术在系统级别比在单元级别获得更多的成功。

9.4 参考文献注释

要了解安全关键系统的软件，一本稍微有点过时但非常棒的入门书籍是Leveson的著作[207]。Butler和Finelli写了一篇很好的文章来阐述不管是以什么方法在软件中确保高置信级别是无价值的，包括测试[59]。Heimdahl[155]很好地描述了关于软件加强系统中安全性挑战的发展前景。要考虑构建安全软件的很多问题，McGraw[233]的著作是当前的一个不错的来源。

要了解更早的处理方法,读者可以尝试阅读Rubin[306]的著作。Denning[103]的书是一本经典但是已经绝版的书,很难得到。

将可测试性问题转换为可控制性和可观察性问题的想法最初产生于硬件测试;Friedman[129]和Binder[34]在软件测试中采用了它。Voas[333]开发了本章中描述的灵敏度模型,将这个可测试性模型应用在随机测试,引出了一系列有趣的验证辩论[334]。

Pezze和Young的最近一本书中列出了被我们省略的其他标准[289]。可以在前面章节引用的研究论文中找到其他参考资料。

要了解网络是怎样改变人们看待软件的方式,一本不错的入门书籍是Powell的著作[292]。一些作者已经评论了subsumption在实际应用中的不足[149, 340, 344]。我们已知的有4篇论文比较了数据流和变异测试[125, 230, 267, 274]。而大多数其他论文则以经验为主比较了其他测试标准[25, 49, 81, 123, 124, 135, 258, 302, 304, 311, 327]。

随机自动生成测试数据的方法要追溯到20世纪70年代中叶[36, 176, 218, 243]。较早的基于规格的生成测试数据的方法使用正式的规格说明书[23, 76]。更近期的方法使用基于模型的语言,如UML[2, 46, 70]。基于语法的生成测试数据的方法早在35年前就出现了,现在基于XML的软件(如网络服务),又使其重新复苏了[150, 231, 248, 280]。一些研究人员调研了怎样使用程序分析技术来自动生成测试数据[39, 80, 101, 102, 174, 190, 191, 267, 295, 342]。使用基于搜索的技术,主要是遗传算法,最初只是简略地获取测试相关的知识[351],到现在已经得到了精化[179, 234, 242]。

参考文献

- [1] W3C #28. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation, October 2000. <http://www.w3.org/XML/#9802xml10>.
- [2] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, England, October 2000.
- [3] Aynur Abdurazik and Jeff Offutt. Coupling-based class integration and test order. In *Workshop on Automation of Software Test (AST 2006)*, pages 50–56, Shanghai, China, May 2006.
- [4] Alan T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.
- [5] Alan T. Acree, Tim A. Budd, Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, September 1979.
- [6] S. B. Akers. On a theory of boolean functions. *Journal Society Industrial Applied Mathematics*, 7(4):487–498, December 1959.
- [7] Roger T. Alexander. *Testing the Polymorphic Relationships of Object-oriented Programs*. PhD thesis, George Mason University, Fairfax, VA, 2001. Technical report ISE-TR-01-04, <http://www.ise.gmu.edu/techrep/>.
- [8] Roger T. Alexander, James M. Bieman, Sudipto Chosh, and Bixia Ji. Mutation of Java objects. In *13th International Symposium on Software Reliability Engineering*, pages 341–351, Annapolis, MD, November 2002. IEEE Computer Society Press.
- [9] Roger T. Alexander, James M. Bieman, and John Viega. Coping with Java programming stress. *IEEE Computer*, 33(4):30–38, 2000.
- [10] Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara, CA, August 1999.
- [11] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose, CA, October 2000. IEEE Computer Society Press.
- [12] Roger T. Alexander and Jeff Offutt. Coupling-based testing of O-O programs. *Journal of Universal Computer Science*, 10(4):391–427, April 2004. http://www.jucs.org/jucs.10.4/coupling_based_testing_of.
- [13] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.
- [14] Paul Ammann and Paul Black. A specification-based coverage metric to evaluate tests. In *4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, November 1999.
- [15] Paul Ammann, Paul Black, and W. Majurski. Using model checking to generate

- tests from specifications. In *2nd International Conference on Formal Engineering Methods*, pages 46–54, Brisbane, Australia, December 1998.
- [16] Paul Ammann and Jeff Offutt. Using formal methods to derive test frames in category-partition testing. In *Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.
- [17] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering*, pages 99–107, Denver, CO, November 2003. IEEE Computer Society Press.
- [18] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, April 1988.
- [19] D. M. St. Andre. Pilot mutation system (PIMS) user's manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, April 1979.
- [20] Jo M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.
- [21] Jo M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
- [22] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.
- [23] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.
- [24] Stephane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In *SQM'94 Second Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK, 1994.
- [25] Vic R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.
- [26] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Fourth International Conference on Software Engineering*, pages 425–432, Munich, September 1979.
- [27] Gamma and Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, May 1999.
- [28] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand, New York, 1984.
- [29] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990. ISBN 0-442-20672-0.
- [30] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic Web sites. In *11th International World Wide Web Conference (WWW'2002) – Alternate Paper Tracks (WE-3 Web Testing and Maintenance)*, pages 654–668, Honolulu, HI, May 2002.
- [31] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, New York, 1993.
- [32] Philip J. Bernhard. A reduced test suite for protocol conformance testing. *ACM*

- Transactions on Software Engineering and Methodology*, 3(3):201–220, July 1994.
- [33] Robert Binder. *Testing Object-oriented Systems*. Addison-Wesley, New York, 2000.
 - [34] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.
 - [35] Robert V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, 6(3/4):125–252, 1996.
 - [36] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–345, 1983.
 - [37] Paul Black, Vladim Okun, and Y. Yesha. Mutation operators for specifications. In *Fifteenth IEEE International Conference on Automated Software Engineering*, pages 81–88, September 2000.
 - [38] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Twenty-First ACM Symposium on the Theory of Computing*, pages 86–97, 1989.
 - [39] Juris Borzovs, Audris Kalniņš, and Inga Medvedis. Automatic construction of test sets: Practical approach. In *Lecture Notes in Computer Science, Vol. 502*, pages 360–432. Springer Verlag, 1991.
 - [40] John H. Bowser. Reference manual for Ada mutant operators. Technical report GIT-SERC-88/02, Georgia Institute of Technology, February 1988.
 - [41] R. S. Boyer, B. Elpas, and K. N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, June 1975. SIGPLAN Notices, vol. 10, no. 6.
 - [42] V. Braberman, M. Felder, and M. Marré. Testing timing behavior of real-time software. In *International Software Quality Week*, 1997.
 - [43] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, February 1998. <http://www.w3.org/TR/REC-xml/>.
 - [44] Lionel Briand, J. Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, Ischia, Italy, 2002. IEEE Computer Society Press.
 - [45] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. In *Fourth International Conference on the Unified Modeling Language (UML '01)*, pages 194–208, Toronto, Canada, October 2001.
 - [46] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, 2002.
 - [47] Lionel Briand, Yvan Labiche, and Yihong Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. Technical report sce-01-02, Carleton University, 2001.
 - [48] Lionel Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.
 - [49] Lionel Briand, Massimiliano Di Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering*, 30(11):770–793, November 2004.
 - [50] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/

- StarMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, May/June 1992.
- [51] Tim Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.
- [52] Tim Budd and Fred Sayward. Users guide to the Pilot mutation system. Technical report 114, Department of Computer Science, Yale University, 1977.
- [53] Tim A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.
- [54] Tim A. Budd, Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. The design of a prototype mutation system for program testing. In *NCC, AFIPS Conference Record*, pages 623–627, 1978.
- [55] Tim A. Budd and Richard J. Lipton. Proving Lisp programs using test data. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 374–403, Ft. Lauderdale, FL, December 1978. IEEE Computer Society Press.
- [56] Tim A. Budd, Richard J. Lipton, Richard A. DeMillo, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.
- [57] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *International Conference on Software Testing, Analysis, and Review (STAR'98)*, San Diego, CA, October 1998.
- [58] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *IEEE International Conference on Communications (Supercomm/ICC'94)*, pages 745–752, New Orleans, LA, May 1994. IEEE Computer Society Press.
- [59] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *Software Engineering*, 19(1):3–12, 1993.
- [60] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 39–48, Portland, OR, August 2000. IEEE Computer Society Press.
- [61] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. *Lecture Notes in Computer Science*, 1486:251–261, 1998.
- [62] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4), July 1979.
- [63] T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Software Testing, Verification, and Reliability*, 11(3):165–180, June 2001.
- [64] T. Y. Chen, P. L. Poon, S. F. Tang, and T. H. Tse. On the identification of categories and choices for specification-based test case generation. *Information and Software Technology*, 46(13):887–898, 2004.
- [65] T. Y. Chen, S. F. Tang, P. L. Poon, and T. H. Tse. Identification of categories and choices in activity diagrams. In *Fifth International Conference on Quality Software (QSIC 2005)*, pages 55–63, Melbourne, Australia, September 2005. IEEE Computer Society Press.
- [66] J. C. Cherniavsky. On finding test data sets for loop free programs. *Information Processing Letters*, 8(2):106–107, February 1979.
- [67] J. C. Cherniavsky and C. H. Smith. A theory of program testing with applications. *Workshop on Software Testing*, pages 110–121, July 1986.

- [68] Shing Chi Cheung, Samuel T. Chanson, and Zhendong Xu. Toward generic timing tests for distributed multimedia software systems. In *12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 210, Washington, DC, 2001. IEEE Computer Society Press.
- [69] Philippe Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macau SAR, China, December 2001.
- [70] Philippe Chevalley and Pascale Thévenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *Proc. of IEEE 25th Annual International Computer Software and Applications Conference (COMPSAC2001)*, Chicago, IL, October 2001.
- [71] Philippe Chevalley and Pascale Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, September 2002.
- [72] John J. Chilenski. Personal communication, March 2003.
- [73] John J. Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
- [74] John J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, Seattle, WA, 1997.
- [75] Byoung-Ju Choi and Aditya Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, February 1993.
- [76] N. Choquet. Test data generation using a prolog with constraints. In *Workshop on Software Testing*, pages 51–60, Banff, Alberta, July 1986. IEEE Computer Society Press.
- [77] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.
- [78] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, CA, February 1997.
- [79] James M. Clarke. Automated test generation from a behavioral model. In *Software Quality Week Conference*, Brussels, Belgium, May 1998.
- [80] Lori Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.
- [81] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A comparison of data flow path selection criteria. In *Eighth International Conference on Software Engineering*, pages 244–251, London, UK, August 1985. IEEE Computer Society Press.
- [82] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15:1318–1332, November 1989.
- [83] Lori Clarke and Debra Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, January 1985.
- [84] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

- [85] David M. Cohen, Siddhartha R. Dalal, A. Kajla, and Gardner C. Patton. The automatic efficient test generator (AETG) system. In *Fifth International Symposium on Software Reliability Engineering (ISSRE'94)*, pages 303–309, Los Alamitos, CA, November 1994. IEEE Computer Society Press.
- [86] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, pages 83–88, September 1996.
- [87] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colburn. Constructing test cases for interaction testing. In *25th International Conference on Software Engineering, (ICSE'03)*, pages 38–48. IEEE Computer Society, May 2003.
- [88] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [89] Alan Cooper. *About Face: The Essentials of User Interface Design*. Hungry Minds, New York, 1995.
- [90] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, Norwood, MA, 2003.
- [91] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. In *9th International Symposium in Software Engineering (ISSRE'98)*, pages 174–178, Paderborn, Germany, November 1998. IEEE Computer Society Press.
- [92] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *21st International Conference on Software Engineering (ICSE'99)*, pages 285–294, Los Angeles, CA, May 1999. ACM Press.
- [93] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4), April 1978.
- [94] Márcio Delamaro, José C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.
- [95] Márcio E. Delamaro and José C. Maldonado. Proteum – A tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.
- [96] Richard A. DeMillo, Dana S. Guindi, Kim N. King, W. Michael McCracken, and Jeff Offutt. An extended overview of the Mothra software testing environment. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, July 1988. IEEE Computer Society Press.
- [97] Richard A. DeMillo, Edward Krauser, and Aditya P. Mathur. Compiler-integrated program mutation. In *Fifteenth Annual Computer Software and Applications Conference (COMPSAC '92)*, Tokyo, Japan, September 1991. Kogakuin University, IEEE Computer Society Press.
- [98] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), May 1979.
- [99] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

- [100] Richard A. DeMillo, W. Michael McCracken, Rhonda J. Martin, and John F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings, Menlo Park, CA, 1987.
- [101] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [102] Richard A. DeMillo and Jeff Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.
- [103] Dorothy Denning. *Cryptography and Data Security*. Addison Wesley, New York, 1982.
- [104] M. S. Deutsch. *Software Verification and Validation Realistic Project Approaches*. Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [105] R. K. Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [106] Mehmet Şahinoğlu and Eugene H. Spafford. A Bayes sequential statistical procedure for approving software products. In Wolfgang Ehrenberger, editor, *The IFIP Conference on Approving Software Products (ASP-90)*, pages 43–56, Garmisch-Partenkirchen, Germany, September 1990. Elsevier/North Holland, New York.
- [107] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Fifth International Conference on Software Engineering*, pages 170–178, San Diego, CA, March 1981.
- [108] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Digital Aviations Systems Conference (DASC)*, October 2000.
- [109] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, New York, 1999.
- [110] Dave E. Eckhardt Jr. and Larry D. Lee. Fundamental differences in the reliability of n-modular redundancy and n-version programming. *Journal of Systems and Software*, 8(4):313–318, September 1988.
- [111] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39:54–67, March 1997. <http://www.siam.org/journals/sirev/39-1/29395.html>.
- [112] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web application testing with user session data. In *25th International Conference on Software Engineering*, pages 49–59, Portland, OR, May 2003. IEEE Computer Society Press.
- [113] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.
- [114] R. En-Nouaary, Khendek F. Dssouli, and A. Elqortobi. Timed test case generation based on a state characterization technique. In *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, December 1998.
- [115] Sadik Esmelioglu and Larry Apfelbaum. Automated test generation, execution, and reporting. In *Pacific Northwest Software Quality Conference*. IEEE Press,

- October 1997.
- [116] R. E. Fairley. An experimental program testing facility. *IEEE Transactions on Software Engineering*, SE-1:350–3571, December 1975.
 - [117] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, January 1996.
 - [118] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, April 1989.
 - [119] Donald G. Firesmith. Testing object-oriented software. In *Testing Object-Oriented Languages and Systems (TOOLS)*, March 1993.
 - [120] Vladimir N. Fleyshgakker and Stewart N. Weiss. Efficient mutation analysis: A new approach. In *International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 185–195, Seattle, WA, August 17–19 1994. ACM SIGSOFT, ACM Press.
 - [121] I. R. Forman. An algebra for data flow anomaly detection. In *Seventh International Conference on Software Engineering*, pages 278–286. Orlando, FL, March 1984. IEEE Computer Society Press.
 - [122] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.
 - [123] Phyllis G. Frankl and Yuetang Deng. Comparison of delivered reliability of branch, data flow and operational testing: A case study. In *2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 124–134, Portland, OR, August 2000. IEEE Computer Society Press.
 - [124] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.
 - [125] Phyllis G. Frankl, Stewart N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.
 - [126] Phyllis G. Frankl, Stewart N. Weiss, and Elaine J. Weyuker. ASSET: A system to select and evaluate tests. In *Conference on Software Tools*, New York, April 1985. IEEE Computer Society Press.
 - [127] Phyllis G. Frankl and Elaine J. Weyuker. Data flow testing in the presence of un-executable paths. In *Workshop on Software Testing*, pages 4–13, Banff, Alberta, July 1986. IEEE Computer Society Press.
 - [128] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
 - [129] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.
 - [130] S. Fujiwara, G. Bochman, F. Khendek, M. Amalou, and A. Ghedasmi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
 - [131] J. Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.
 - [132] Leonard Gallagher, Jeff Offutt, and Tony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, 17(1):215–266, January 2007.

- [133] Robert Geist, Jeff Offutt, and Fred Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992. Special issue on Fault-Tolerant Computing.
- [134] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Eighth International Conference on Software Engineering*, pages 313–319, London, UK, August 1985. IEEE Computer Society Press.
- [135] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Workshop on Software Testing*, pages 64–73. Banff, Canada, July 1986. IEEE Computer Society Press.
- [136] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle, WA, August 1994.
- [137] G. Gonenc. A method for the design of fault-detection experiments. *IEEE Transactions on Computers*, C-19:155–558, June 1970.
- [138] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2), June 1975.
- [139] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, November 1983.
- [140] Mats Grindal. *Evaluation of Combination Strategies for Practical Testing*. PhD thesis, Skövde University / Linköping University, Skövde Sweden, 2007.
- [141] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. An evaluation of combination testing strategies. *Empirical Software Engineering*, 11(4):583–611, December 2006.
- [142] Mats Grindal and Jeff Offutt. Input parameter modeling for combination strategies. In *IASTED International Conference on Software Engineering (SE 2007)*, Innsbruck, Austria, February 2007. ACTA Press.
- [143] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15(2):97–133, September 2005.
- [144] Mats Grindal, Jeff Offutt, and Jonas Mellin. Conflict management when using combination strategies for software testing. In *Australian Software Engineering Conference (ASWEC 2007)*, pages 255–264, Melbourne, Australia, April 2007.
- [145] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification, and Reliability*, 3(2):63–82, 1993.
- [146] Matthias Grochtmann, Klaus Grimm, and J. Wegener. Tool-supported test case design for black-box testing by means of the classification-tree editor. In *1st European International Conference on Software Testing Analysis & Review (EuroSTAR 1993)*, pages 169–176, London, UK, October 1993.
- [147] Richard Hamlet. Reliability theory of program testing. *Acta Informatica*, pages 31–43, 1981.
- [148] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
- [149] Richard G. Hamlet and Richard Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

- [150] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 4:242–257, 1970.
- [151] Jim M. Hanks. Testing cobol programs by mutation: Volume I – Introduction to the CMS.1 system, Volume II – CMS.1 system documentation. Technical report GIT-ICS-80/04, Georgia Institute of Technology, February 1980.
- [152] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.
- [153] Mary Jean Harrold and Gregg Rothermel. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.
- [154] Mary Jean Harrold and Mary Lou Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.
- [155] Mats E. Heimdahl. Safety and software intensive systems: Challenges old and new. In *International Conference on Software Engineering: Future of Software Engineering*, pages 137–152, May 2007.
- [156] E. Heller. Using design of experiment structures to generate software test cases. In *12th International Conference on Testing Computer Software*, pages 33–41, New York, 1995. ACM.
- [157] K. Henninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.
- [158] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.
- [159] A. Hessel, K. Larsen, B. Nielsen, and A Skou. Time optimal real-time test case generation using UPPAAL. In *Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, October 2003.
- [160] Bill Hetzel. *The Complete Guide to Software Testing*. Wiley-QED, second edition, 1988.
- [161] J. Robert Horgan and Saul London. Data flow coverage and the C language. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [162] J. Robert Horgan and Saul London. ATAC: A data flow coverage testing tool for C. In *Symposium of Quality Software Development Tools*, pages 2–10, New Orleans, LA, May 1992. IEEE Computer Society Press.
- [163] J. Robert Horgan and Aditya P. Mathur. Weak mutation is probably strong mutation. Technical report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, December 1990.
- [164] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, SE-24, May 1975.
- [165] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.
- [166] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4), July 1977.
- [167] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE*

- Transactions on Software Engineering*, 8(4):371–379, July 1982.
- [168] William E. Howden. The theory and practice of function testing. *IEEE Software*, 2(5), September 1985.
 - [169] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.
 - [170] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.
 - [171] J. Huller. Reducing time to market with combinatorial design method testing. In *10th Annual International Council on Systems Engineering (INCOSE'00)*, Minneapolis, MN, July 2000.
 - [172] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Sixteenth International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.
 - [173] Michael Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, Cambridge, UK, 2000.
 - [174] G. Hwang, K. Tai, and T. Hunag. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.
 - [175] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.
 - [176] D. C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, 1987.
 - [177] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle, WA, August 1994.
 - [178] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.
 - [179] B. F. Jones, D. E. Eyres, and H. H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Computer Journal*, 41(2):98–107, 1998.
 - [180] J. A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.
 - [181] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, 1994.
 - [182] Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. John Wiley and Sons, New York NY, second edition, 1999.
 - [183] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *Conference on Human Factors in Computing Systems: Common Ground*, pages 244–251, New York, April 1996.
 - [184] Sun-Woo Kim, John Clark, and John McDermid. Assessing test set adequacy for object-oriented programs using class mutation. In *Symposium on Software Technology (SoST'99)*, pages 72–83, September 1999.
 - [185] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented strategies with the mutation method. In *Mutation 2000*:

- Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 4–100, San Jose, CA, October 2000. Special issue of the *Journal of Software Testing, Verification, and Reliability*, December 2001.
- [186] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings – Software*, 146(4):187–192, August 1999.
- [187] Kim N. King and Jeff Offutt. A Fortran language system for mutation-based software testing. *Software – Practice and Experience*, 21(7):685–718, July 1991.
- [188] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.
- [189] Charles Knutson and Sam Carmichael. Safety first: Avoiding software mishaps, November 2000. <http://www.embedded.com/2000/0011/0011feat1.htm>.
- [190] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.
- [191] Bogdan Korel. Dynamic method for software test data generation. *Software Testing, Verification, and Reliability*, 2(4):203–213, 1992.
- [192] Edward W. Krauser, Aditya P. Mathur, and Vernon Rego. High performance testing on SIMD machines. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, Banff, Alberta, July 1988. IEEE Computer Society Press.
- [193] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *The SPIN'04 Workshop on Model-Checking Software*, Barcelona, Spain, 2004.
- [194] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.
- [195] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *27th NASA/IEEE Software Engineering Workshop*, NASA Goodard Space Flight Center, MD, December 2002. NASA/IEEE.
- [196] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.
- [197] D. Kung, J. Gao, Pei Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC '95)*, pages 239–244, Dallas, TX, August 1995. IEEE Computer Society Press.
- [198] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *IEEE 24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, pages 537–542, Taipei, Taiwan, October 2000.
- [199] W. LaLonde and J. Pugh. Subclassing != subtyping != is-a. *Journal of Object Oriented Programming*, 3(5): 57–62, January 1991.
- [200] Janusz Laski. Data flow testing in STAD. *Journal of Systems and Software*, 12:3–14, 1990.
- [201] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy.

- IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.
- [202] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology*, 14(3):247–276, July 2005.
- [203] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong, China, November 2001. IEEE Computer Society Press.
- [204] H. Ledgard and M. Marcotty. A genealogy of control structures. *Communications of the ACM*, 18:629–639, November 1975.
- [205] Yu Lei and K. C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Third IEEE High Assurance Systems Engineering Symposium*, pages 254–261, November 1998. IEEE Computer Society Press.
- [206] Yu Lei and K. C. Tai. A test generation strategy for pairwise testing. Technical Report TR-2001-03, Department of Computer Science, North Carolina State University, Raleigh, 2001.
- [207] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.
- [208] Zhang Li, Mark Harman, and Rob M. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.
- [209] J. L. Lions. Ariane 5 flight 501 failure: Report by the inquiry board, July 1996. <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>.
- [210] Richard Lipton. New directions in testing. In *Distributed Computing and Cryptography, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 2, pages 191–202, Providence, RI, 1991.
- [211] Richard J. Lipton and Fred G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.
- [212] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison Wesley, New York, 2000.
- [213] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(1):1811–1841, November 1994.
- [214] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.
- [215] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structural testing of Web applications. In *11th International Symposium on Software Reliability Engineering*, pages 84–96, San Jose, CA, October 2000. IEEE Computer Society Press.
- [216] C. H. Liu, D. C. Kung, P. Hsia, and C. T. Hsu. An object-based data flow testing approach for Web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, 2001.
- [217] Giuseppe A. Di Lucca and Massimiliano Di Penta. Considering browser interaction in web application testing. In *5th International Workshop on Web Site Evolution (WSE 2003)*, pages 74–84, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.
- [218] Stephen F. Lundstrom. Adaptive random data generation for computer software

- testing. In *National Computer Conference*, pages 505–511, 1978.
- [219] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis, MD, November 2002. IEEE Computer Society Press.
- [220] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.
- [221] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. mujava home page. online, 2005. <http://ise.gmu.edu/offutt/mujava/>, <http://salmosa.kaist.ac.kr/LAB/MuJava/>, last access April 2006.
- [222] Yashwant K. Malaiya. Antirandom testing: Getting the most out of black-box testing. In *International Symposium on Software Reliability Engineering (IS-SRE'95)*, pages 86–95, Toulouse, France, October 1995.
- [223] Brian A. Malloy, Peter J. Clarke, and Errol L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *14th International Symposium on Software Reliability Engineering*, Denver, CO, 2003. IEEE Computer Society Press.
- [224] Robert Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, October 1985.
- [225] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 4(13):365–398, Nov. 1995.
- [226] Brian Marick. The weak mutation hypothesis. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 190–199, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [227] Brian Marick. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.
- [228] Aditya P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. In *Sixth Annual Pacific Northwest Software Quality Conference*, Portland, OR, Lawrence and Craig, 1991.
- [229] Aditya P. Mathur and Edward W. Krauser. Mutant unification for improved vectorization. Technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 1988.
- [230] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.
- [231] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.
- [232] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
- [233] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, New York, 2006.
- [234] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification, and Reliability*, 13(2):105–156, June 2004.
- [235] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven

- approach to generate test cases for GUIs. In *21st International Conference on Software Engineering*, pages 257–266, May 1999.
- [236] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, New York, November 2000.
- [237] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.
- [238] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, pages 118–127, September 2003.
- [239] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267, September 2001.
- [240] Atif M. Memon and Qing Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. In *The International Conference on Software Maintenance 2004 (ICSM'04)*, pages 8–17, Washington, DC, September 2004.
- [241] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.
- [242] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.
- [243] E. F. Miller and R. A. Melton. Automated generation of testcase datasets. In *International Conference on Reliable Software*, pages 51–58, April 1975.
- [244] Cleve Moler. A tale of two numbers. *SIAM News*, 28(1), January 1995.
- [245] S. Morasca and Mauro Pezze. Using high level Petri-nets for testing concurrent and real-time systems. *Real-Time Systems: Theory and Applications*, pages 119–131. Amsterdam, North-Holland, 1990.
- [246] Larry J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park, MD, 1984. Technical Report TR-1395.
- [247] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.
- [248] Carlos Urias Munoz. An approach to software product testing. *IEEE Transactions on Software Engineering*, 14(11):1589–1595, November 1988.
- [249] Glenford Myers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.
- [250] Adithya Nagarajan and Atif M. Memon. Refactoring using event-based profiling. In *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, Victoria, British Columbia, November 2003.
- [251] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Fault Tolerant Computing Systems*, pages 238–243. IEEE Computer Society Press, 1981.
- [252] B. Nielsen and A. Skou. Automated test generation from timed automata. In *21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, FL, 2000. IEEE Computer Society Press.

- [253] Jakob Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, 2000.
- [254] Robert Nilsson. *Automatic Timeliness Testing of Dynamic Real-Time Systems*. PhD thesis, Skövde University/Linköping University, Skövde Sweden, 2006.
- [255] Robert Nilsson and Jeff Offutt. Automated testing of timeliness : A case study. In *Second Workshop on Automation of Software Test (AST 2007)*, Minneapolis, MN, May 2007.
- [256] Robert Nilsson, Jeff Offutt, and Sten F. Andler. Mutation-based testing criteria for timeliness. In *28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 306–312, Hong Kong, September 2004.
- [257] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. In *2nd International Workshop on Model Based Testing*, pages 102–121, Vienna, Austria, March 2006.
- [258] S. C. Ntafos. An evaluation of required element testing strategies. In *Seventh International Conference on Software Engineering*, pages 250–256, Orlando FL, March 1984. IEEE Computer Society Press.
- [259] Bashar Nuseibeh. Who dunnit? *IEEE Software*, 14:15–16, May/June 1997.
- [260] Department of Defense. *DOD-STD-2167A: Defense System Software Development*. Department of Defense, February 1988.
- [261] Department of Defense. *MIL-STD-498: Software Development and Documentation*. Department of Defense, December 1994.
- [262] Jeff Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1988. Technical report GIT-ICS 88/28.
- [263] Jeff Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [264] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag Lecture Notes in Computer Science Volume 1723.
- [265] Jeff Offutt, Aynur Abdurazik, and Roger T. Alexander. An analysis tool for coupling-based integration testing. In *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 172–178, Tokyo, Japan, September 2000. IEEE Computer Society Press.
- [266] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong, China, November 2001. IEEE Computer Society Press.
- [267] Jeff Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software – Practice and Experience*, 29(2):167–193, January 1999.
- [268] Jeff Offutt and Kim N. King. A Fortran 77 interpreter for mutation analysis. In *1987 Symposium on Interpreters and Interpretive Techniques*, pages 177–188, St. Paul MN, June 1987. ACM SIGPLAN.
- [269] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

- [270] Jeff Offutt and Stephen D. Lee. How strong is weak mutation? In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 200–213, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [271] Jeff Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.
- [272] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.
- [273] Jeff Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.
- [274] Jeff Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software – Practice and Experience*, 26(2):165–176, February 1996.
- [275] Jeff Offutt, Roy Pargas, Scott V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. In *1992 International Conference on Parallel Processing*, pages II257–266, Chicago, August 1992.
- [276] Jeff Offutt, Jeffrey Payne, and Jeffrey M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA, March 1996. <http://www.ise.gmu.edu/techrep/>.
- [277] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.
- [278] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *15th International Symposium on Software Reliability Engineering*, pages 187–197, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society Press.
- [279] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. In *Workshop on Testing, Analysis and Verification of Web Services*, Boston, MA, July 2004. ACM SIGSoft.
- [280] Jeff Offutt and Wuzhi Xu. Testing web services by XML perturbation. In *16th International Symposium on Software Reliability Engineering*, Chicago, IL, November 2005. IEEE Computer Society Press.
- [281] Alex Orso and Mauro Pezze. Integration testing of procedural object oriented programs with polymorphism. In *Sixteenth International Conference on Testing Computer Software*, pages 103–114, Washington, DC, June 1999. ACM SIGSOFT.
- [282] L. J. Osterweil and L. D. Fosdick. Data flow analysis as an aid in documentation, assertion generation, validation, and error detection. Technical report CU-CS-055-74, Department of Computer Science, University of Colorado, Boulder, CO, September 1974.
- [283] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.
- [284] Jan Overbeck. *Integration Testing for Object-Oriented Software*. PhD dissertation, Vienna University of Technology, 1994.
- [285] A. J. Payne. A formalised technique for expressing compiler exercisers.

- SIGLPAN Notices*, 13(1):59–69, January 1978.
- [286] Ivars Peterson. Pentium bug revisited, May 1997. http://www.maa.org/mathland/mathland.5_12.html.
- [287] E. Petitjean and H. Fochal. A realistic architecture for timed testing. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems*, Las Vegas, October 1999.
- [288] A. Pettersson and H. Thane. Testing of multi-tasking real-time systems with critical sections. In *Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications (RTCSA'03)*, Tainan City, Taiwan, February 2003.
- [289] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2008.
- [290] S. Pimont and J. C. Rault. A software reliability assessment based on a structural behavioral analysis of programs. In *Second International Conference on Software Engineering*, San Francisco, CA, October 1976.
- [291] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measure experience during function test. In *14th International Conference on Software Engineering (ICSE'93)*, pages 287–301, Los Alamitos, CA, May 1993. ACM.
- [292] T. A. Powell. *Web Site Engineering: Beyond Web Page Design*. Prentice-Hall, Englewood Cliffs, NJ, 2000.
- [293] R. E. Prather. Theory of program testing – an overview. *Bell System Technical Journal*, 62(10):3073–3105, December 1983.
- [294] Paul Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, 1972.
- [295] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.
- [296] K. Ramamritham. The origin of time constraints. In *First International Workshop on Active and Real-Time Database Systems (ARTDB 1995)*, pages 50–62, Skövde, Sweden, June 1995. Springer, New York, 1995.
- [297] S. Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.
- [298] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91, April 2001.
- [299] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.
- [300] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34, Toronto, CA, May 2001.
- [301] Marc Roper. *Software Testing*. International Software Quality Assurance Series. McGraw-Hill, New York, 1994.
- [302] Dave Rosenblum and Gregg Rothermel. A comparative study of regression test selection techniques. In *IEEE Computer Society 2nd International Workshop on*

- Empirical Studies of Software Maintenance*, pages 89–94, Bari, Italy, October 1997. IEEE Computer Society Press.
- [303] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.
 - [304] J. Rowland and Y. Zuyuan. Experimental comparison of three system test strategies preliminary report. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 141–149, Key West, FL, December 1989. ACM SIGSOFT 89.
 - [305] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.
 - [306] Avi D. Rubin. *White-Hat Security Arsenal: Tackling the Threats*. Addison-Wesley, New York, 2001.
 - [307] K. Sabnani and A. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 14(4):285–297, 1988.
 - [308] P. SanPietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE Transactions on Software Engineering*, 26(2):128–149, February 2000.
 - [309] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, Norwell, MA, 1993.
 - [310] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.
 - [311] Richard W. Selby. Combining software testing strategies: An empirical evaluation. In *Workshop on Software Testing*, pages 82–90, Banff, Alberta, July 1986. IEEE Computer Society Press.
 - [312] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *27th International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington, Brussels, Tokyo, June 1997.
 - [313] G. Sherwood. Effective testing of factor combinations. In *Third International Conference on Software Testing, Analysis, and Review (STAR94)*, Washington, DC, May 1994. Software Quality Engineering.
 - [314] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 72–77, Hong Kong, China, September 2004. IEEE Computer Society Press.
 - [315] M. D. Smith and D. J. Robson. Object-oriented programming: The problems of validation. In *6th International Conference on Software Maintenance*, pages 272–282, Los Alamitos, CA, 1990. IEEE Computer Society Press.
 - [316] Ian Sommerville. *Software Engineering*. Addison-Wesley, New York, 6th edition, 2001.
 - [317] British Computer Society Specialist Interest Group in Software Testing. *Standard for Software Component Testing, Working Draft 3.3*. British Computer Society, 1997. http://www.rmcs.cranfield.ac.uk/~cised/sreid/BCS_SIG/.
 - [318] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
 - [319] Phil Stocks and Dave Carrington. Test templates: A specification-based testing framework. In *Fifteenth International Conference on Software Engineering*, pages

- 405–414, Baltimore, MD, May 1993.
- [320] Phil Stocks and Dave Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
 - [321] K. C. Tai and F. J. Daniels. Test order for inter-class integration testing of object-oriented software. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 602–607, Santa Barbara, CA, 1997. IEEE Computer Society.
 - [322] K. C. Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, January 2002.
 - [323] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.
 - [324] M. Tatsubori. OpenJava WWW page. Tokyo Institute of Technology, Chiba Shigeru Group, 2002. <http://www.csg.is.titech.ac.jp/~mich/openjava/> (accessed May 2004).
 - [325] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, LNCS 1826:117–133, June 2000.
 - [326] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, KTH, Stockholm, Sweden, 2000.
 - [327] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 410–417, Montreal, Canada, June 1991. IEEE Computer Society Press.
 - [328] F. Tip. A survey of program slicing techniques. Technical report CS-R-9438, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica, 1994.
 - [329] Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Pierre Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, March 2000.
 - [330] Roland H. Untch, Jeff Offutt, and Mary Jean Harrold. Mutation analysis using program schemata. In *1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge, MA, June 1993.
 - [331] P. Verissimo and H. Kopetz. Design of distributed real-time systems. In S. Mullender, editor, *Distributed Systems*, pages 511–530. Addison-Wesley, New York, 1993.
 - [332] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In *ZB2002: 2nd International Conference of Z and B Users*, pages 295–313, Grenoble, France, January 2002. Springer-Verlag, LNCS 2272.
 - [333] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.
 - [334] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):553–563, May 1995.
 - [335] K. S. How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.
 - [336] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification, and Reliability*, 10(1):3–46, March 2000.

- [337] A. Watkins, D. Berndt, K. Aebischer, J. Fisher, and L. Johnson. Breeding software test cases for complex systems. In *37th Annual Hawaii International Conference on System Sciences (HICSS'04) – Track 9*, page 90303.3, Washington, DC, USA, 2004. IEEE Computer Society Press.
- [338] J. Wegener, H. H. Sthammer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.
- [339] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.
- [340] Steward N. Weiss. What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Notes*, 14(6):42–49, October 1989.
- [341] Elaine Weyuker. The oracle assumption of program testing. In *Thirteenth International Conference on System Sciences*, pages 44–49, Honolulu, HI, January 1980.
- [342] Elaine Weyuker, Thomas Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.
- [343] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.
- [344] Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. Comparison of program testing strategies. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [345] Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. Data flow-based adequacy analysis for languages with pointers. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 74–86, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [346] Lee White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *9th International Symposium on Software Reliability Engineering*, pages 110–121, October 2000.
- [347] Lee White, Husain Almezen, and Nasser Alzeidi. User-based testing of GUI sequences and their interaction. In *10th International Symposium on Software Reliability Engineering*, pages 54–63, November 2001.
- [348] Lee White and Bogdan Wiszniewski. Path testing of computer programs with loops using a tool for simple loop patterns. *Software – Practice and Experience*, 21(10):1075–1102, October 1991.
- [349] Lee J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, New York, 1987.
- [350] Duminda Wijesekera, Lingya Sun, Paul Ammann, and Gordon Fraser. Relating counterexamples to test cases in CTL model checking specifications. In *A-MOST '07: Third ACM Workshop on the Advances in Model-Based Testing, co-located with ISSSTA 2007*, London, UK, July 2007.
- [351] Christian Wild, Steven Zeil, and Gao Feng. Employing accumulated knowledge to refine test descriptions. *Software Testing, Verification, and Reliability*, July 2(2): 53–68, 1992.
- [352] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *13th International Conference on the Testing of Communicating Sys-*

- tems (TestCom 2000), pages 59–74, Ottawa, Canada, August 2000.
- [353] Alan W. Williams and Robert L. Probert. A practical strategy for testing pairwise coverage of network interfaces. In *7th International Symposium on Software Reliability Engineering (ISSRE96)*, White Plains, New York, November 1996.
 - [354] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, pages 304–311, Beirut, Lebanon, June 2001.
 - [355] Barbara Liskov with John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, New York, 2001.
 - [356] W. Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).
 - [357] W. Eric Wong and Aditya P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.
 - [358] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, Banff Alberta, July 1988. IEEE Computer Society Press.
 - [359] Ye Wu and Jeff Offutt. Modeling and testing Web-based applications. Technical report ISE-TR-02-08, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2002. <http://www.ise.gmu.edu/techrep/>.
 - [360] Ye Wu, Jeff Offutt, and Xiaochen Du. Modeling and testing of dynamic aspects of Web applications. Technical report ISE-TR-04-01, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004. <http://www.ise.gmu.edu/techrep/>.
 - [361] Tao Xie and Dave Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.
 - [362] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 45–54, Boston, MA, July 2004. ACM Software Engineering Notes.
 - [363] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic test generation using checkpoint encoding and antirandom testing. Technical Report CS-97-116, Colorado State University, 1997.
 - [364] S. J. Young. *Real-Time Languages: Design and Development*. Ellis Horwood, Chichester, UK, 1982.
 - [365] Christian N. Zapf. Medusamothra – A distributed interpreter for the mothra mutation testing system. M.S. thesis, Clemson University, Clemson, SC, August 1993.
 - [366] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.
 - [367] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.