



Testing Embedded Software

嵌入式软件 测试

[美] Bart Broekman 著

Edwin Notenboom

张君施 张惠宇 周承平 译

许菊芳 审校



电子工业出版社

Publishing House of Electronics Industry

<http://www.phei.com.cn>

嵌入式软件测试

Testing Embedded Software

嵌入式系统在我们的生产和生活中随处可见。现代工具，不管是电视机还是手机，都离不开嵌入式系统。嵌入式软件已经在许多产业中占据了很重要的位置，比如汽车、航空航天、生物医药和军事系统。鉴于许多应用系统对人的生命安全都有潜在的威胁，对严格的软件测试的需求也越来越迫切。本书针对这些需求，为嵌入式系统软件提供了全面、切实而且强调安全的测试策略。

本书是市面上第一本全面而有深度的嵌入式系统测试专著，它将是嵌入式系统产业领域的测试工程师、程序员、项目经理和团队领导的得力助手。

本书主要特点：

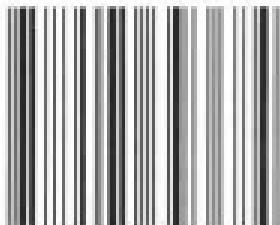
- ↗ 深入剖析了可以直接应用的测试设计技术
- ↗ 详细讲述了测试组织成员角色、任务和职责及其组织结构，指导如何建立专业测试组织
- ↗ 为时间和资源有限的测试提供了切实指导

作者简介：

Bart Broekman：1990年开始从事软件测试，起初他是飞利浦数据系统公司一个操作系统内核测试团队的成员。五年后他加盟了 Sogeti 公司，并在 Sogeti 成功完成了测试自动化、组织和管理大型测试活动等任务。之后他参与了欧洲嵌入式软件研究项目（ITEA），并且与其他人合作撰写了一本测试自动化方面的专著。

Edwin Notenboom：从 1996 年起就是 Sogeti 公司的专业测试员。他曾经担任许多公司的顾问，这些公司包括一些荷兰和德国的嵌入式工业企业，他的服务内容包括结构化测试、测试自动化和测试过程改进等方面。1999 年 1 月他和 Bart Broekman 一起参与了 ITEA 嵌入式系统项目。

ISBN 7-5053-9368-5



9 787505 393684 >



责任编辑：赵红燕
贺瑞君

封面设计：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-5053-9368-5 定价：35.00 元

嵌入式软件测试

Testing Embedded Software

[美] Bart Broekman
Edwin Notenboom 著

张君施 张思宇 周承平 译
许莉芳 审校

电子工业出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

随着软硬件技术的发展，嵌入式系统在生产、生活乃至军上的各个领域应用都日渐广泛，功能也越来越强大，但设备和软件也日趋复杂。本书立足于工业实践，旨在为有效控制复杂的嵌入式软件测试过程提供解决方案。书中全面讲述了嵌入式软件测试的一般过程，内容包括结构化测试和嵌入式系统的原理、测试生命周期、重要的应用技术、基础设施、测试组织形式和测试原则。本书在测试设计技术和测试组织方面的精辟论述，将会对软件测试的规范化和高效化大有帮助。

本书特别适合与嵌入式系统的软件打交道的人士以及嵌入式项目的管理人员，对嵌入式硬件开发和测试人员也很有裨益。

Authorized translation from the English language edition, entitled Testing Embedded Software, ISBN: 0321159861 by Bart Broekman, Edwin Notenboom, published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2003. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Simplified Chinese language edition published by Publishing House of Electronics Industry, Copyright © 2004.

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

本书中文简体专有翻译出版权由 Pearson 教育集团所属的 Addison-Wesley 授予电子工业出版社。其原文版权及中文翻译出版权受法律保护。未经许可，不得以任何形式或手段复制或抄袭本书内容。

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

版权贸易合同登记号 图字：01-2003-3948

图 书 在 版 编 目 (CIP) 数据

嵌入式软件测试 / (美) 布鲁克曼 (Broekman, B.) 著；张君施等译。

-北京：电子工业出版社，2004.1

书名原文：Testing Embedded Software

ISBN 7-5053-9368-5

I. 嵌… II. ①布… ②张… III. 软件 - 测试 IV. TP311.5

中国版本图书馆 CIP 数据核字 (2003) 第 106514 号

责任编辑：赵红燕 贺瑞君

印 刷：北京兴华印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

经 销：各地新华书店

开 本：787 × 980 1/16 印张：20.25 字数：363 千字

印 次：2004 年 1 月第 1 次印刷

印 数：5000 册 定价：35.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。
联系电话：(010) 68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

译者序

嵌入式设备已经在工业市场、自控市场以及国防建设等领域得到了广泛应用，而且随着数字化产品时代的来临，大量硬件结构日趋复杂、功能日益强大的嵌入式系统正不断进入人们的工作和生活空间中，这为嵌入式软件产品创造了巨大的商业机会，同时也对嵌入式软件的开发技术和测试技术提出了新的挑战。面对时间和资源都很有限的情况，嵌入式软件测试的计划、组织和有效实施往往会更加困难。

尽管国内市场上有关软件测试的书籍多得让人眼花缭乱，但涉及嵌入式软件测试的却寥寥无几。本书的作者 Bart Broekman 和 Edwin Notenboom 在嵌入式软件测试方面都具有丰富的实践经验。本书对嵌入式软件测试进行了全面的介绍，并向读者提供了大量的工程实践指导内容。文中不仅给出了一种嵌入式软件结构化测试方法，还详细描述了测试生命周期、嵌入式软件的开发与测试过程，提供了测试生命周期不同阶段所采用的各种经典及现代测试技术，同时也覆盖了进行测试所需的基础设施以及测试组织方面的内容。

本书具有很强的实践性，不仅适用于专门从事嵌入式系统开发和测试的人员，也适用于那些负责嵌入式系统开发和测试的项目经理或团队领导。同时，本书对于其他系统的软件开发和测试人员也非常有参考价值。

本书的第1章~第10章由张君施翻译，第11章~第20章由张思宇翻译，附录A~E及以后部分由周承平翻译，全书由许菊芳审校和统稿。由于译者水平和实际工作经验有限，译文中的不当之处在所难免，恳请读者批评指正。

序

近年来，在工业领域的各个方面，软件的重要性戏剧性地得到提升。在商业管理领域，软件也已经成为不可或缺的核心技术。而且在多数技术领域，越来越多的产品和革新都是基于软件的。一个典型的例子是，在汽车工业中，迅速出现的大量技术创新都是基于电子设备和软件的，这些产品和技术不仅增强了汽车的安全性，而且提高了乘坐者的舒适度，减少了能耗和废气排放。在现代高级豪华轿车的制造中，已经有大约 20% 到 25% 的费用是用在电子设备和软件上，而且在未来十年里，这一比例估计将提高到 40%。

软件对产品的质量有着实实在在的影响，同时也影响着企业的生产力。遗憾的是，实践告诉我们，不可能开发出一种复杂的、一蹴而就的软件系统。因此，需要找到一种全面的分析方法，来检测软件开发不同阶段的结果，以便尽可能早地发现系统中的错误，这样测试就成了最重要的分析技术。但是软件测试是一项非常复杂而耗时的工作，对于嵌入式系统更是如此。测试嵌入式软件的时候，不仅需要考虑软件本身，而且需要考虑和硬件部件的紧密关系，这种关系通常是条件苛刻的时间约束和实时要求，以及其他和性能相关的关系。

呈现在读者面前的这部作品，对嵌入式系统测试领域必将有重要的、实质性的贡献。它全面描述了嵌入式软件测试的整个领域，覆盖了各个重要的方面，比如测试生命周期、测试技术等，同时也提供了测试的基本方法和组织结构的内容。作者将内容浓缩在实践性的工业应用方面，使得该书对任何从事嵌入式软件测试的人员都具有参考价值。由于嵌入式软件的广泛应用，这本书将对许多工业领域都有用处。

在通往高效率嵌入式软件测试的漫漫长路上，对于如何经济地开发出高性能的嵌入式系统而言，该书的全面性和实用性使得它成为嵌入式软件系统开发和测试的里程碑，书中的一些观念已经成为 DaimlerChrysler 集团的基本工作准则。在许多不同工业领域里，可靠的嵌入式系统的开发已经成为核心业务的一部分。我也希望本书中的观念能为这些领域中数目庞大的测试者和软件开发人员所用，对他们有所裨益。

Klaus Grimm 博士
DaimlerChrysler AG 软件技术研究室主任

致 谢

想避免类似于“本书决不如何如何”这样的陈腐信息是件很不容易的事情，但是我们也不想忽略这样一个事实：我们从许多人那里得到了有价值的帮助。提到他们的贡献，这里略举一二：

当 Rob Dekker 和 Martin Pol 以极好的口才邀请我们参加一个欧洲 ITEA 项目时，我们得到了为嵌入式工业开发出一种测试方法的机会，这个项目是为汽车工业的嵌入式系统而设立的。Klaas Brongers 管理和激励着我们，这里充满了自由和创新的气氛。这个项目被证明是发展嵌入式软件测试方法的沃土，在这里，我们可以研究新的思想，测试新思想的实用价值。尤其是项目合作伙伴 Mirko Conrad、Heiko Dörr 和 Eric Sax，他们都是知识和经验的无价源泉，他们的鼎力支持和高度严格的要求极大地提高了我们工作的效率。需要特别感谢 Mirko 和 Eric，他们提供了完整的一章，是专门讲述“混合信号”的。

很显然，ITEA 项目中发展出来的一些思想，对于其他从事嵌入式工业的人来说也是有用的。我们很幸运，有如此多的主管、经理拥有推进这一目标的远见和决心，他们是：Luciëlle de Bakker、Hugo Herman de Groot、Jan van Holten、Ronald Spaans 和 Wim van Uden，正是这些人提供了关键的管理支持。

决定了写一本这方面的书后，我们也从许多同事那里获得了支持，Bart Douven、John Knappers、Peter van Lint、Dre Robben、Coen de Vries 和 Paul Willaert 承担了输入和反馈工作。Peter 和 Coen 尤其必须被提及，是他们努力在嵌入式软件测试环境方面提出了大量有益的想法，这些想法形成了有关嵌入式软件测试环境相关章节的基础。在让文档变成书的过程中，Rob Baarda 也提供了大量的帮助，使得该书得以出版。在客户方面，Jack van de Corput 和 Boy van den Dungen 用他们的高度热情，使得我们的想法能够付诸实践。

当越来越多的章节完成时，“外部”的关注就显得越来越多，我们非常高兴，因为我们能够得到测试领域和嵌入式软件领域众多专家的帮助，他们是：Simon Burton、Rix Groenboom、Klaus Grimm、Norbert Magnusson、Matthias Pillin，特别是 Stuart Reid 和 Otto Vinter，他们两位审阅了最后的手稿。应当感谢的是，他们并没有迁就我们，而是给予我们真实的看法和建设性的批评，他们的许多批评和建议值得尊敬，而且在书中体现了出来。

衷心感谢那些给予这本书无价贡献的人，他们都会和我们一样，为这项工作而感到骄傲。

前　　言

发展中的工业需要结构化的测试

嵌入式系统是一个快速成长的工业领域。在历史上，嵌入式系统由工程师和技术人员控制，这些工程师和技术人员都在各自的领域有技术专长，他们生产产品，同时完成测试任务，因为只有他们才最了解产品的工作原理。这在技术人员只和那些小的、单一的产品打交道的时候情形还好。然而嵌入式领域在快速变化着：系统变得越来越大，也越来越复杂和集成化。软件现在占系统的较大部分，经常替代硬件。以前只工作在孤立状态下的系统现在已经联系在一起，提供集成的功能。这些系统不可能由某个聪明人独自完成，而是需要有组织的集体智慧。同样，测试的过程也变得越来越庞大、复杂而难于控制。于是，能提供一种控制复杂测试过程的方法就成了现实的需求。

本书的范围

嵌入式系统必须依赖于高品质的硬件和高性能的软件，因此对于测试嵌入式系统而言，硬件测试和软件测试都是至关重要的部分。但是，本书更关心的是嵌入式系统的软件测试部分。本书会涉及到许多硬件，但是不会讨论测试单个硬件的技术细节，这是硬件本身的事情。通常在硬件测试时，技术人员能够处理复杂的技术细节。本书的目标读者是和嵌入式系统的软件打交道的人。书中会告诉这些人有关的工作环境，测试软件时遇到的具体问题，以及通常不会在软件教学中学到的技术。

本书目标是为“有效控制复杂测试过程”这个问题提供解决方案。在各种各样的软件和硬件环境工作里，本书将主要关注如何组织全面的测试过程这一较高层次。作者从“Software Testing, a Guide to The Tmap Approach”（《软件测试——TMap 方法指南》）一书中引用了一些概念和内容，但是已经修改成了适合嵌入式软件的形式。

本书的目标不是作为一篇研究性的论文，它具有很强的实践性，其目的是提供大量总结性的、深入的实践指导，而没有细节性的学术论证。

本书的结构

测试远不是试验系统和检验系统的正确性那么简单。测试还包括计划的制定、

设计测试用例、管理测试基础设施、建立测试组织、安排策略等。本书描述了一种称为 TEmb 的测试方法，其目的是对嵌入式软件进行结构化测试。TEmb 方法覆盖了测试过程中的一些关键步骤，回答了“做什么、什么时候做、如何做、用什么方法做和谁去做”的问题。TEmb 应用了在 TMap 中定义的结构化测试的 4 个要素：开发和测试的生命周期（“做什么、什么时候做”），测试过程中的技术（“如何做”）、基础设施（“用什么做”）和组织（“由谁做”），这 4 个要素是本书结构的基础。

本书由六个部分组成：

第一部分讲述结构化测试和嵌入式系统的--般原理，提供 TEmb 方法的综述，指出如何为特定的嵌入式系统组织合适的测试步骤。

第二部分讲述嵌入式系统测试的生命周期，以及开发和测试嵌入式系统的过程。生命周期要素是测试过程的支柱，它提供应该在什么顺序下做什么的路线图。其他 3 个要素中的各种问题，都对应生命周期要素中各个相关点。

第三部分提供对多数嵌入式软件测试项目都有用的几种技术，包括基于风险的测试策略、可测性审查、正式评审和安全性分析。这部分还提供各种不同的技术，用来设计可以适合于不同目标和不同环境的测试用例。

第四部分讲述的是基础设施。测试者需要它们才能完成测试工作。在测试过程的不同阶段，需要不同的测试环境，这一部分提供适用于各种不同测试行为和目标的工具的概述。在协助完成自动化测试任务的工作中，工具的运用已经非常普遍。这部分还讲解这种自动化测试中技术性的和组织性的问题。最后探讨的是各种具体问题，当测试者在一个环境中同时处理模拟和数字信号（即所谓“混合信号”）时，这些问题将会出现。

第五部分描述各种不同的测试组织形式，也就是在测试活动中，人员该如何各司其职、如何相互交流。此外还包括各种测试原则的描述，以及对管理和组织结构的描述。测试者该如何报告测试进度和系统质量的问题也在这部分讲解。

第六部分是一些附录，讲述一些主题的背景信息。比如风险级别、状态表模型等，此外还包括一个自动测试套件的设计方案和一个测试计划的例子。

目标读者

这本书的目标读者是与嵌入式系统开发和测试相关的人员，尤其是那些和嵌入式软件相关的人。书中提供嵌入式系统测试的组织和技术方面的指导原则，总体的和细节的都有。不同类型的读者有可能对不同的章节有所偏好，下面的这些建议将指出哪些章节最适合哪些读者。

对所有的读者，推荐阅读第一部分的介绍性章节，还有第 3 章、第 4 章和第 7 章等，这些内容是 TEmb 方法的精髓。

开发经理、测试项目经理，或者测试协调人和测试团队领导，将会从第二部分、第五部分以及第 7 章中受益最多。

测试人员、开发人员以及其他实际上从事主要的软件测试的人员将从第三部分和第四部分中发现许多具有实践价值的信息。如果读者要提交正式的进度和质量报告，则可以从第五部分中得到许多信息。

对于那些和开发、测试硬件相关的人员，建议阅读第 3 章和第 13 章，这两章阐述了对于嵌入式硬件和软件来讲二者协同作战达到一个共同目标的重要性。对于软件开发人员和测试人员来讲，这两章同样是他们应该感兴趣的。

从第 17 章和第 18 章里，人力资源管理者可以找到特别适合于他们的信息。

目 录

第一部分 介 绍

第1章 基础	2
1.1 测试的目标.....	2
1.2 什么是嵌入式系统.....	3
1.3 走近嵌入式系统测试.....	4

第2章 TEmb 方法	5
2.1 概览	5
2.2 TEmb 通用元素	8
2.3 组合专用测试方法的机制.....	12

第二部分 生 命 周 期

第3章 多 V 模型	21
3.1 介绍	21
3.2 多 V 模型中的测试活动.....	22
3.3 嵌套多 V 模型	25

第4章 制定主测试计划	27
4.1 制定主测试计划的要素	27
4.2 活动	30

第5章 由开发人员执行的测试	39
5.1 介绍	39
5.2 集成方法.....	40
5.3 生命周期.....	44

第6章 独立测试团队的测试	49
6.1 介绍	49

6.2	计划与控制阶段	49
6.3	准备阶段	59
6.4	细化阶段	62
6.5	执行阶段	64
6.6	完成阶段	67

第三部分 技术

第 7 章	基于风险的测试策略	73
7.1	介绍	73
7.2	风险评估	74
7.3	主测试计划中的策略	76
7.4	测试层次中的策略	79
7.5	测试过程中的策略变更	83
7.6	维护测试的策略	84
第 8 章	可测性审查	86
8.1	介绍	86
8.2	规程	86
第 9 章	评审	89
9.1	介绍	89
9.2	规程	90
第 10 章	安全性分析	93
10.1	介绍	93
10.2	安全性分析技术	94
10.3	安全性分析生命周期	98
第 11 章	测试设计技术	102
11.1	概述	102
11.2	状态转换测试	109
11.3	控制流测试	122
11.4	基本比较测试	126
11.5	分类树方法	132

11.6	进化算法	138
11.7	统计使用测试	145
11.8	稀有事件测试	152
11.9	突变分析	153

第 12 章	审查清单	156
12.1	介绍	156
12.2	每个质量特性的审查清单	156
12.3	高层次测试的一般审查清单	162
12.4	低层次测试的一般审查清单	164
12.5	测试设计技术审查清单	165
12.6	测试过程审查清单	167

第四部分 基 础 设 施

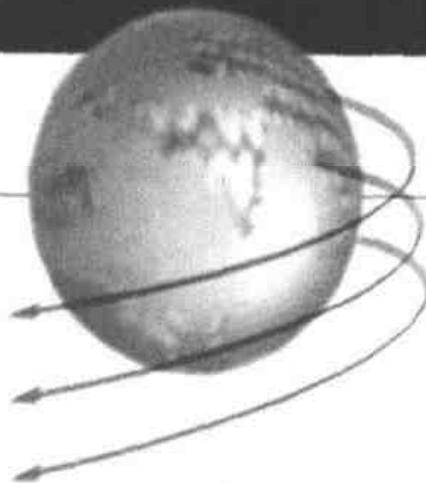
第 13 章	嵌入式软件测试环境	180
13.1	介绍	180
13.2	第一阶段：模拟阶段	182
13.3	第二阶段：原型阶段	185
13.4	第三阶段：临近生产阶段	191
13.5	开发后阶段	193
第 14 章	工具	195
14.1	介绍	195
14.2	测试工具的分类	196
第 15 章	测试自动化	203
15.1	介绍	203
15.2	测试自动化技术	204
15.3	实现测试自动化	208
第 16 章	混合信号	214
16.1	介绍	214
16.2	激励描述技术	218
16.3	测量和分析技术	228

第五部分 组织

第 17 章 测试角色	235
17.1 一般技能	235
17.2 特定的测试角色	236
第 18 章 人力资源管理	246
18.1 人员	246
18.2 培训	247
18.3 职业前景	249
第 19 章 组织结构	253
19.1 测试组织	253
19.2 通信结构	256
第 20 章 测试控制	258
20.1 测试过程的控制	258
20.2 测试基础设施的控制	263
20.3 测试交付物的控制	265

第六部分 附录

附录 A 风险级别	272
附录 B 状态表	274
附录 C 一个自动化测试包的设计方案	279
附录 D 进化算法的伪代码	290
附录 E 测试计划例子	293
词汇表	303
参考文献	309



第一部分 介 绍

这一部分讲述嵌入式系统结构化测试的一般原理，提供 TEmb 方法的概览。

第 1 章介绍关于测试和嵌入式系统的基础知识，解释测试的目标以及结构化测试过程中的要素，提供了一个嵌入式系统的通用图例，用来解释“嵌入式系统”代表的意思。这个通用图例也将用在本书的其他部分，尤其是在第 17 章中。

第 2 章描述用来进行嵌入式软件结构化测试的 TEmb 方法，指出不存在一个能适用于所有嵌入式系统的测试方法。对于特定的嵌入式系统而言，TEmb 就是组成合适的测试步骤集合的方法，它由一些基本测试步骤组成，这些测试步骤提供几种特定的措施，以处理特定测试系统中的具体问题。本章还解释了如何用有限的系统特征来区分不同种类的嵌入式系统，从而能够在测试步骤中制定出相应的具体措施。在这一章里还提供这种“基本测试步骤”的综述，以及把系统特征和具体措施联系起来的一个矩阵。

第1章 基础

1.1 测试的目标

测试是一个过程，它的中心任务就是发现系统中的缺陷。对于每一个测试过程，从系统调试和可接受性的方面来说，发现缺陷是最为关键的部分。尽管所有人都承认预防缺陷总比发现和改正它们要好，但现实是我们现在还无法生产无缺陷的系统。在系统开发过程中，测试是个基本要素，它有助于提高系统的品质。

测试的最终目标是提供经常性的好建议，告诉组织该如何处理系统缺陷。提出建议的前提是发现和系统需求相关的缺陷（不管是明显的缺陷还是隐含的可能性），测试本身并不能直接提高系统的品质，而是通过发现系统缺陷，对组织的相关风险提出深入建议。这使得管理者能做出更好的决定，调配资源来提高系统性能。

为达到这些测试目标，每一个测试过程都包含这些项目：制定计划、列出测试清单和执行测试用例。有一个普遍规律：不可能发现所有的缺陷，也决不会有足够的时间（或者人力和财力）来测试每一件事情。为了最大限度地利用可用资源，必须做出选择。因为所有测试过程都有一些共同的东西，所以能够定义一些基本的、通用的结构化测试步骤，以组织一个可控的测试过程。

关于测试有一个非常简单的例子：一支圆珠笔。

假设某公司计划生产圆珠笔，他们将其中的一支交给我们的测试人员，要求测试。这支圆珠笔就被称为测试对象，测试者可以测试圆珠笔的许多方面，比如：它能用正确的颜色、合理的线宽写字吗？笔上面的标志是否符合公司的标准？咬嚼它是否是安全的（是否含有毒物质）？在 100 000 次按压之后，它的按压装置是否还起作用？在汽车碾过它之后，是否还能写？等等。

为了回答这些问题，测试者需要获得这支笔的预期信息，而“适合写字”这一信息，是决定测试哪些项目和结果是否可以接受的基础，这可以称为测试基础。

测试圆珠笔的墨水或其他部分是否有毒，需要高昂的设备和专家费用，测试 100 000 次的按压需要许多时间。在这些测试中，真的需要测试者投入这么多财力和精力吗？测试者将这些问题和相关人员进行研究，比如主管和圆珠笔的潜在用户，他们将决定测试的最重要部分以及该测试到何种深度，最后的结果就称为测试策略。

当测试者按照测试策略进行测试时，他可能会发现缺陷，这意味着圆珠笔没有按照期望的目标工作，依据缺陷的严重程度以及使用中可能存在的风险，测试者会估算出笔的质量，也会提供如何处理的建议。

为执行测试，测试者除了一支笔之外，还需要测试有毒化学物质的设备。一定的基础设施是必需的。测试还需要其他有特定知识和技能的人，比如化学设备操作员，因此必须有合适的组织。

图 1.1 表明了在测试过程中，上面提到的这些因素是如何与测试过程交互的。测试过程定义了必需的项目和在生命周期中如何组织它们。对复杂的项目来说，需要设计特定的技术，以便帮助完成测试任务。

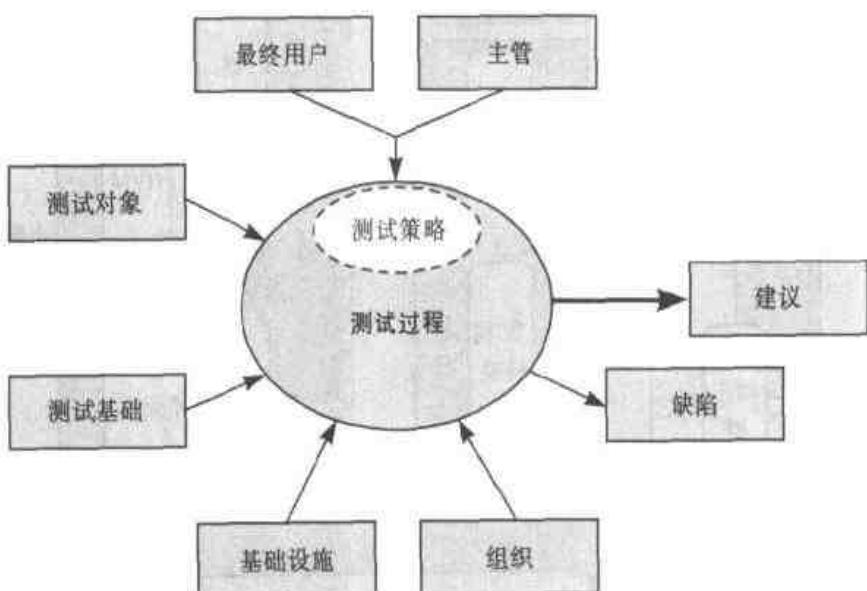


图 1.1 测试过程的通用元素

1.2 什么是嵌入式系统

“嵌入式系统”也是一个实际上并不能真正道出实质内容的广义通用术语，它包含蜂窝电话、铁路信号系统、听觉辅助和导弹跟踪系统等系统，但是所有的嵌入式系统都和现实的物理世界相结合，控制着某些特定的硬件。图 1.2 所示的结构实际上适合于所有的嵌入式系统，在这个图里包含了嵌入式系统的典型组成部分。

嵌入式系统和真实环境相互影响，通过传感器接收信号，给动作器发送输出信号，而动作器控制着环境，嵌入式系统的环境，包括动作器和传感器，经常被称做“物理环境”。

系统的嵌入式软件被存储在非易失性存储器（NVM）中，通常是 ROM，但

这些软件也可以保存在闪存卡、硬盘和 CD-ROM 中，还可以通过网络或人造卫星下载。嵌入式软件被编译成适合特定的目标处理器，“处理单元”通常需要一定数量的 RAM 来操作。因为处理单元只能处理数字信号（暂且忽略模拟计算机），而环境可能需要模拟信号，因此数/模转换和模/数转换就产生了。处理单元通过专门的 I/O 层输入/输出所有的信号。通过特定的接口，嵌入式系统能够和物理环境以及其他可能的（嵌入式）系统交互。嵌入式系统可以从普通电源获得电能供应，还可以有自己的电力供应系统，比如电池。

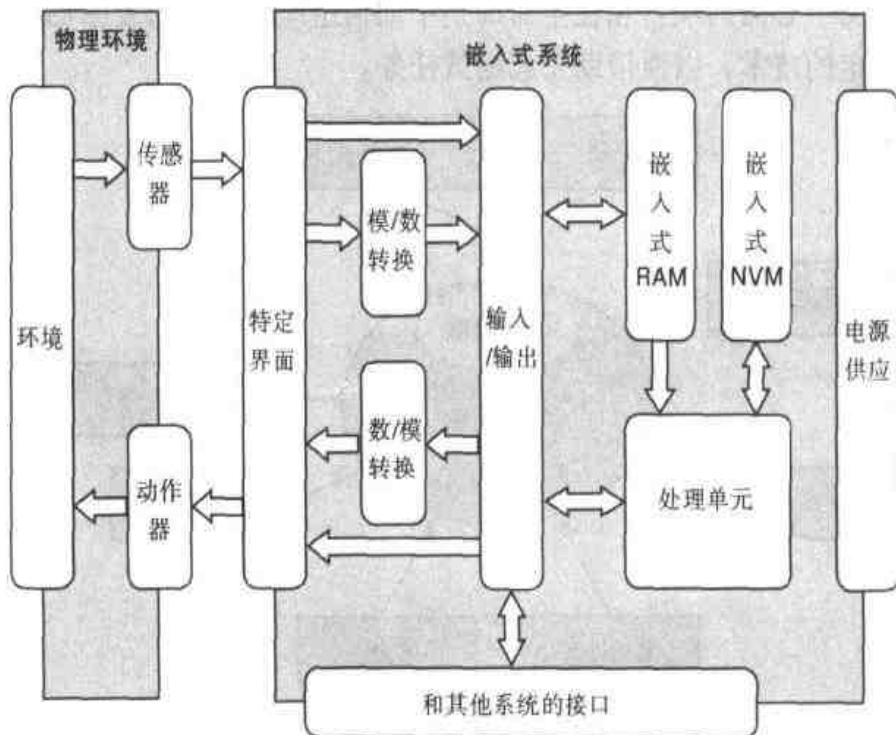


图 1.2 嵌入式系统的一般结构

1.3 走近嵌入式系统测试

很显然，移动电话的测试会和视频机顶盒的测试或者汽车巡航测试完全不同。每一种测试方法都需要特定的标准来覆盖相关问题。因此寻找一个嵌入式系统的通用测试方法是不可能的。

尽管有许多理由说明为什么不同的嵌入式系统必须有不同的测试方法，但是对于任何测试方法来讲，仍然存在许多相似的问题和相似的解决办法。某些“基本测试原理”必定适合于所有的嵌入式测试项目。但这些原理也必须和特定的方法结合起来，才能在特定的系统测试中解决具体的问题。

这就是为什么本书提供了一种方法，它能帮助“组合”恰当的测试方法。在以后的章节里，将有更进一步的解释。

第 2 章 TEmb 方法

TEmb 是一种方法，它能够为特定的嵌入式系统组合恰当的测试方法。它提供一种机制，可以从适用于任何测试项目的通用元素和一组相关的特定方法中组合出恰当的专用测试方法，这组特定方法和所观察到的嵌入式系统的系统特性相关联。本章将首先讲述 TEmb 方法的概览，然后给出“通用元素”、“特定方法”、“机制”和“系统特性”的详细解释。

2.1 概览

图 2.1 描述了 TEmb 方法，其工作的基本原理如下：

对任何嵌入式系统而言，测试方法的基础都是由通用元素组成的，这些通用元素是任何结构化测试的组成部分。比如，根据一定的生命周期来计划测试项目、采用标准化技术、专用的测试环境、组织测试团队和编写正式报告等等。它们都和结构化测试的四个要素相关，即生命周期、基础设施、技术和组织（在图 2.1 中称为“LITO”）。这些基本的测试方法仍然很不具体，还有待更多的细节来丰富，比如需要采用什么设计技术、需要使用或必须首先开发哪些工具和其他基础设施部件等等。在项目初始阶段，首先必须选择出将包含在测试过程中的特定方法。在 TEmb 方法中，这被称做“组合专用测试方法的机制”。

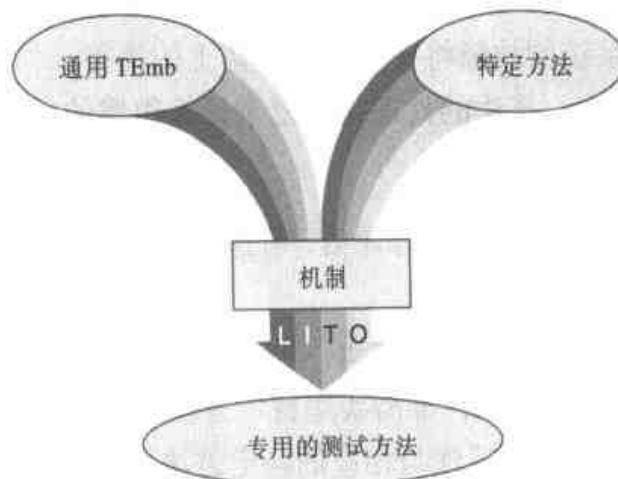


图 2.1 TEmb 方法概览：专用测试方法是由通用元素和特定方法组合而成的，这两者都和结构化测试的 4 个要素（LITO）相关

“机制”基于下面的分析：

- **风险** 需要采取一定的措施来分析由于产品质量低下而导致的商业风险。在第 7 章中将给出更详细的解释。
- **系统特性** 需要采取措施来处理和产品的（技术）特性相关的问题。这些都是高级特性，比如技术-科学算法、混合信号、强调安全等。

在这一段中，将通过下面这些例子来进一步解释和系统特性分析相关的部分。请考虑下面 10 个嵌入式系统：

- 机顶盒。
- 导航控制。
- 天气预报。
- 晶片移位。
- 心脏起搏器。
- 核磁共振扫描仪。
- 红外线温度计。
- 铁路信号设备。
- 电信交换。
- 导弹防御系统。

这些嵌入式系统是完全不同的系统，很自然地就可以设想出，这些系统的测试方法也会大不相同。但它们并不是要采用完全不同的测试方法。那么是什么使得每个系统在测试时各具个性呢？在它们各自的特性中，是否存在共性呢？在本例中，我们来看下面两个特性是否可以适用于每一个系统：

- **强调安全** 系统的故障将导致人类身体上的伤害。
- **技术-科学算法** 算法的处理是由复杂计算组成的，比如求解微分方程或计算导弹的轨迹。

现在对上面 10 个系统分别进行评估，看这两个特性是否都适用。这样，就将 10 个系统分成了 4 组，见表 2.1。

属于同一组的系统具有相近的性质，可以用同样的办法来处理。比如对于铁路信号设备和心脏起搏器来说，都需要用特定方法来保证系统的安全，而不用考虑技术-科学算法。由于系统特性与相应的特定方法之间存在联系，因此组合出恰当的测试方法，可以说是件容易的事情。对于这个例子的两个特性来说，建议采用以下特定方法：

表 2.1 利用系统特性来对系统个性进行分类

技术-科学算法

		强调安全	
		否	是
否		机顶盒 晶片移位 电信交换	铁路信号设备 心脏起搏器
	是	天气预报 红外线温度计	导航控制 核磁共振扫描仪 导弹防御系统

1. 对强调安全的系统来说，可以采用生命周期模型 MOD-00-56（见表 2.2），划分出专门的测试层次来执行安全测试。在测试过程中会出现“安全经理”和“安全工程师”两个新角色。建议采用这样几种技术：故障模式及后果分析（FMEA）、故障树分析（FTA）、模型检查和正式检验。
2. 为了处理技术-科学算法，可以借助于进化算法和隐患检测等多种技术，以确保复杂的处理流程能够被完全覆盖。此外还需要有用于覆盖范围分析和隐患检测的工具。在测试（或开发）过程的早期，需要制定计划来对这些算法进行明确的验证。在这个专门领域，需要得到数学专家的支持。

表 2.2 说明了特定方案是与哪一个测试要素（生命周期、基础设施、技术和组织）相关联的。

表 2.2 对每一个系统特性来说都有一套提议的和四个要素相关的特定方法

系统特性	生命周期	基础设施	技术	组织
强调安全	主测试计划，包括 MOD-00-56	覆盖范围分析程序	FMEA / FTA 模型检查 正式检验 稀有事件测试	安全经理 安全工程师
	安全测试(测试层次)			
	负载/强度测试			
技术-科 学算法	算法确认	覆盖范围分析程序	进化算法	数学专家
		隐患检测程序	隐患检测	

关注系统特性的好处是，只需要有限的几个方面就可以覆盖系统中庞杂的多样性。对于每一个嵌入式系统测试项目，通过分析与哪些系统特性相关，就可以使系统的独特性变得越来越具体和可管理。将可能有用的方法和每个系统特性联系起来，就可以为专用测试方法中需要包含的内容提供指导。

这里所说的机制，可以看做是对基于风险的测试策略的一个扩展：对感知风

险（因为它具有主观性，因而通常称为“主观性风险”）进行分析，其结果是确定在测试方法中哪些该做（以及哪些不该做）。对系统特性（自然更带有主观色彩）进行分析，将得到适用于测试方法的解决方案，以便处理特定嵌入式系统测试项目的具体问题。

2.2 TEmb 通用元素

这一段描述通用元素，它们是所有结构化测试过程的基础。

在开发一个新系统的过程中，会有许多不同的人员或团队来执行各种不同的测试（见第 4.1.2 节）。在新项目开始时，需要制定出一份主测试计划，决定由谁来负责哪项测试，以及各测试之间的关系是什么（见第 4 章）。

对每一个单独的测试层次来说，结构化测试的四个要素同样存在（见图 2.2）。这四个要素也是这样一些主要问题的答案，即“做什么与什么时候做”、“如何做”、“通过什么做”和“由谁做”。

- **生命周期 (Lifecycle, L)** 它定义必须进行哪些活动以及按照什么顺序来执行。它使测试人员和经理都能够把握测试过程。
- **技术 (Technique, T)** 它解决的是如何做的问题，通过制定出标准化方法来执行特定的活动。
- **基础设施 (Infrastructure, I)** 它定义的是在测试环境中需要哪些设施，以便能够执行计划中的活动。
- **组织 (Organization, O)** 它定义的是执行计划中活动的人员的角色和所需的专业技能，以及和其他团队交互的方法。

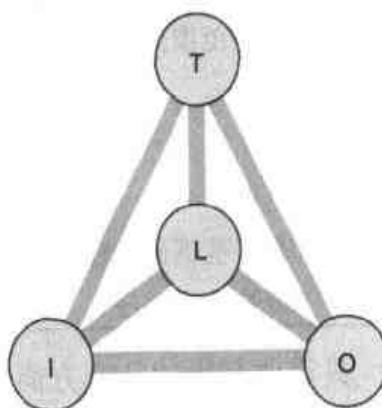


图 2.2 结构化测试过程的四要素

一个基本思想是，四个要素都必须均等地体现在测试过程中。如果其中一个要素被忽略了，那么测试过程将会遇到麻烦。可以将生命周期视为中心要素，它

是各要素之间的“黏合剂”。在生命周期的不同阶段，其他三个要素的应用方式也各不相同。比如详细设计活动和测试执行活动以及分析活动需要有不同的技术和专业技能。

当四个要素都被充分考虑到之后，测试过程就可以被认为是结构化的。在预防遗漏关键因素和使测试更趋可管理和可控制方面，结构化是一个主要的因素。但是结构化并不能确保测试永远不会遇到麻烦，每一个项目早晚都会遇到无法预料的事情，从而使精心计划的活动陷入混乱。但是结构化的测试过程能够尽可能地以最小的代价从这些事件中快速恢复过来。

下面几节将分别简要讨论测试的每一个要素。

2.2.1 生命周期

在生命周期模型中，主要的测试活动被划分成五个阶段（见图 2.3）。除了计划和控制阶段、准备阶段、详细设计阶段以及执行阶段之外，还定义了一个完成阶段，以便完整地结束测试过程，并将测试件正式交付给相关组织，以便进行后续的产品维护和发布。

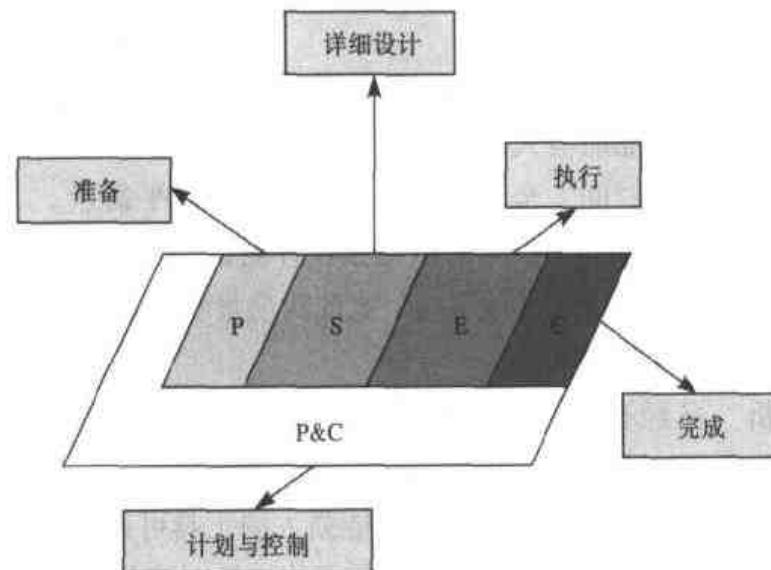


图 2.3 生命周期模型中的五个阶段

生命周期的基本思想就是：除了把握项目关键路径，还要尽可能快、尽可能多地执行测试活动。如果被测系统在执行阶段被交付给测试人员时，测试人员才开始考虑测试用例应该是什么样子，那么将是对宝贵时间的极大浪费。这应当是在前面的阶段，即详细设计阶段就能够而且应当完成的任务。如果测试人员在详细设计阶段还不能开始细化测试用例，同样是件烦人的事情，因为他们仍然必须考虑应当选择哪些设计技术，以及应当如何将设计技术运用到他刚刚得到的系统

详细设计中。而这应当是在前面的阶段（即准备阶段）就该完成的任务。

尽管和测试组织所能够考虑到的测试活动相比，生命周期能够给出明显更多的活动，但也没有必要增加额外的活动。生命周期只不过将必须做的事情描述得更清楚，将它们安排得更有效率而已。如果加以适当应用，生命周期就可以成为一个节省时间的有力机制，尤其是在处理关键路径上的活动时。

2.2.2 技术

“技术”要素通过向测试人员提供详细的、经过证明的和通用的工作方法来支持测试过程，而且也使得管理人员（和审计人员）能够跟踪测试过程的进度并对结果进行评估。对每一种类型的活动来说，原则上都可以运用一项或多项技术。世界上每天都有大量的新技术出现。当有一个可能多次重复的测试活动存在时，为该测试活动提供特定的技术，就可以支持以后的测试过程。

TEmb 模型提供了多种技术。有些是古老但实用的技术，已经在实践中证明了它们的价值，而且并不需要修改。而有些是新技术，还有的是基于现有的技术发展起来的。许多技术都是多种技术的混合体。

这里提供了一些技术的例子：

- **策略开发** 测试策略就是基于风险评估来做出选择和协调。策略开发的技术就是在所有产品所有人之间对应该给测试投入多少达成一致，以便能够在所需的质量与时间、金钱和资源之间找到最佳平衡点。
- **测试设计** 已经有许多技术可以用于设计测试用例。至于选择采用哪些设计技术，则依赖于许多其他事情，比如对质量特性的评估、测试的覆盖范围、可用的系统规范等等。
- **安全分析** 已经有了用来评估系统的安全性以及在应当采取的措施上达成一致的专门技术。
- **数据驱动的测试自动化** 有许多功能强大的工具可以用于自动化测试。它们不仅能够捕捉和回放测试过程，甚至能支持对高级测试脚本的编程。这些工具实际上是建立自动化测试套件的开发环境。数据驱动的测试技术已经被成功地应用，以实现持续而灵活的自动化测试。
- **审查清单** 审查清单是一项简单而有效的技术。从审查清单可以获得测试人员的经验，这可以在以后的测试项目中重用。

2.2.3 基础设施

测试基础设施包括结构化测试所需的所有设备。它可以分为 3 类：执行测试

所需的设备（测试环境）、使测试得以有效执行的设备（工具和测试自动化）、人员的工作场所（办公环境）。

测试环境

在测试环境中，最重要的三个元素是：

- 硬件/软件/网络 被测系统在不同的开发阶段有着不同的物理属性。比如：模型、原型、和模拟器相连的独立单元、成型产品等。对每一阶段每一类型的产品需要有不同的测试环境。
- 测试数据库 多数测试都是可以重复的，这有助于重现测试结果。这意味着必须以适当的方式来保存某些测试数据。
- 模拟和测量设备 当测试在真实环境中需要外部信号才能运行的系统（或部件）时，需要有模拟器来替代外部信号。而且需要有专门的设备来检测和分析系统产生的输出信息。

工具和测试自动化

在执行测试过程时并不是绝对需要有工具，但是工具能够使测试人员更加轻松。测试工具的数量庞大，它们的范围和种类千变万化。测试工具可以按照它们所支持的活动来分类，从而也可以按照使用它们的测试阶段来分类。下面的列表所示的就是在生命周期的每个阶段，可以应用的一些测试工具的例子。

■ 计划和控制阶段

- 计划和进度控制。
- 缺陷管理。
- 配置管理。

■ 准备阶段

- 需求管理。
- 复杂性分析。

■ 细化阶段

- 测试用例生成器。
- 测试数据生成器。

■ 执行阶段

- 捕捉和回放程序。
- 比较程序。
- 监视器。
- 覆盖范围分析程序。

办公环境

很显然，测试质量主要依赖于测试人员的技能和态度。应当避免无谓的分心和糟糕的心情。为测试人员提供舒适的环境是一个很重要的因素，这样他们才能够组织和执行从计划阶段到完成阶段的测试工作。这看似简单，但在实际中，测试人员经常发现他们的桌子和电脑都没有按时准备好，不得不和同事挤在一起。

2.2.4 组织

组织指人以及人与人之间的交流。测试不是一项可以在孤立、不受外部世界干扰的情况下可以进行的简单任务。由于涉及多种不同学科、利益冲突、不可预见性、专业人员短缺和时间约束等因素，测试团队的建立和管理并不是一件容易的事情。

“组织”要素包括以下几个方面：

- 测试组织的结构 测试组织结构定义的是测试组织在整个组织中的地位，以及测试组织的内部结构。它覆盖了不同团队层次、它们的职责及相互之间的关系。
- 角色 角色为每个团队定义了必须执行的任务以及执行好任务所需要的技能和专业知识。
- 人员和培训 这主要考虑如何获得以及如何留住测试所需的人员。这是关于获得人员、培训和职业道路的学问。
- 管理和控制规程 因为测试经常发生在不断有变化的环境中，所以测试过程必须有能够处理变化的规程。需要区分开三种不同的控制规程：
 - 测试过程。
 - 测试基础设施。
 - 测试产品。

2.3 组合专用测试方法的机制

不同嵌入式系统的测试项目都包含着一些相同的基本原理（见前面的论述），但是如果考虑细节的话，还是有许多不同点。每个测试项目都会选择许多具体的具体方法来达到项目的特定目标并处理特定的嵌入式系统的特定问题。在 TEmb 方法中，这被称为“组合专用测试方法的机制”，它基于对风险和系统特性的分析。

基于风险分析来选择测试方法是一个对“该做什么和不该做什么”的取舍，以及分配测试优先级的过程。这在第 7 章有详细描述，这里不再进一步讨论。

对系统特性的分析会客观得多，因为处理过程更多的是客观事实，而不是个人观点。关于这一点在本部分将有更详细的解释。首先解释什么是系统特性，它对测试过程产生何种影响，然后是对所谓的 LITO 矩阵的一个概览，它描述了对每个系统特性而言，可以应用的特定解决方案。

2.3.1 系统特性

2.1 节已经解释了利用系统特性来对不同系统进行分类的原则。有必要指出的是，TEMb 方法的目标不是对嵌入式系统进行科学而完整的分类。它只不过是为了解释实践，完全是从测试人员的立场来分类。它的目标就是协助测试经理回答这样两个问题：这个系统的特别之处是什么？在测试方法中必须包含哪些东西才能够处理这些特别之处？

下面提供一组有用的系统特性。但这里并不是要列出全部的系统特性，测试组织可以根据产品特点来定义其他的系统特性。

- 强调安全系统。
- 技术-科学算法。
- 自治系统。
- 惟一系统——“一次性 (one-shot)” 开发。
- 模拟输入和输出（通常称为混合信号）。
- 硬件限制。
- 基于状态的行为。
- 硬实时行为。
- 控制系统。
- 极端的环境条件。

注意没有为反应系统定义系统特性。当一个系统能够快速地对任何输入事件都做出响应时，该系统就被称为反应系统 (Erpenbach 等, 1999)。反应系统要完全负责与环境的同步，这完全可以通过“基于状态的行为”特性和“硬实时行为”特性一起来描述。

以下简要地描述上面所列的系统特性。

强调安全系统

当嵌入式系统的故障会导致对人体健康的严重伤害（或更为可怕的后果）时，则称该系统是“强调安全”的。管理系统很少是强调安全的，这种系统的故障通常会令人烦恼并导致财务损失，而几乎不会对人体造成伤害。但是大量的嵌入式

系统都和人体有直接的物理接触，故障会直接导致身体的损伤，例如航空电子设备、医疗设备及核反应堆等。对这样的系统进行风险分析是极其重要的，需要采用严格的技术来分析并提高系统的可靠性。

技术-科学算法

从表面看来，嵌入式系统的行为通常简单又明了。比如导航控制系统，提供给操纵者的使用说明往往只有简单的几页纸。但实现这样的系统仍然是非常困难的一件事情，可能需要大量的控制软件来处理复杂的科学运算。对于这样的系统，它的更多更复杂的活动在内部，从外面是无法看见的。这意味着测试的重点将是面向白盒的测试层次，而面向黑盒的可接收性测试则会相对少一些。

自治系统

有些嵌入式系统可以在无限长的时间内自主运行，它们担负某种任务。一旦运行之后，就不再需要有人干预或与人交互。比如交通信号系统和某些武器系统，这些系统一旦启动之后，就完全自动地执行任务。这些系统中的软件被设计为连续运行，不再需要或者甚至不可能有人的干预，就能够对某些事件做出响应。这种特性导致的一个直接结果是：这样的系统不可能进行手工测试。因此就需要特殊的测试环境和测试工具来执行测试用例，并测量和分析结果。

惟一系统——“一次性”开发

某些系统，比如人造卫星，一旦发射之后就无法对其进行维护。这样的系统是“惟一系统”（也称为“专门系统”），意味着其建造是一次性的。它和大批量的市场产品正好相反，市场产品处在一个竞争性的市场里，必须定期地升级到新版本，以保持对用户的吸引力。对大批量的市场产品而言，其后续的维护保养是非常重要的因素，因此在开发和测试阶段就要采用特殊手段，以使维护的费用和时间能够达到最小。比如复用测试件和回归测试自动化都可能是标准的规程。但是，惟一系统的测试基本上没有长期目标，因为在其首次也是惟一一次发布之后，测试就终止了。对这种类型的系统，需要多考虑维护、复用和回归测试等因素。

模拟输入和输出（通常称为混合信号）

在管理领域，能够准确地预知和度量一个事务的处理结果。发票上的数目、姓名或地址都可以被确切地定义，这不同于嵌入式系统中处理模拟信号的情况。系统的输入输出不是一个确定的数值，而是用一定的“偏差”来定义系统可接受的范围。同样，期望输出的结果也不能够被确切地定义。输出值超出了预先定义

的范围，也并不总是表明它就是错误的。存在一个灰色区域，即有时候要依靠直觉来判断测试输出是否可以接受。在这种情况下，人们使用术语“硬边界”和“软边界”来描述。

硬件限制

硬件资源的局限性会给嵌入式软件带来一定的约束，比如内存使用和电力消耗。有时特定的硬件也依赖于软件中计时的解决方法。对软件方面的这些约束并不会影响到需求的系统功能，但它们却是系统运行的首要条件，需要对此进行大量深入而技术性又很强的测试。

基于状态的行为

基于状态的行为可以描述为由于特定事件的触发，使系统从一个状态转换到另一个状态。这种系统对输入的响应依赖于以往事件的历史（它使得系统处于一个特定状态）。当同一个输入并不总是能够被接受，而且接受后可能产生不同输出的时候，这样的系统就表现出基于状态的行为（Binder, 2000）。

硬实时行为

“实时”的本质就是输入或输出在发生的瞬间就能够影响到系统的行为。比如一个工资系统就不是实时的，系统是现在或15分钟之后计算你的工资，结果不会有丝毫差异（当然如果系统在1年之后去计算，结果可能会有所不同，但这并不能够使系统成为“实时”的系统）。为了对实时行为进行测试，测试用例中必须包含输入输出时间的详细信息，而且测试用例结果通常依赖于它们被执行的顺序。这对于测试设计和测试执行都有着重大的影响。

控制系统

控制系统通过连续的反馈机制和环境相互作用：系统的输出会影响环境，而环境反过来会影响到控制系统的 behavior。因此不能单独描述控制系统的 behavior，而需要考虑环境的 behavior。这种系统的测试通常要求能够精确地模拟环境的 behavior，比如工业过程控制系统、飞机控制系统等。

极端的环境条件

有些系统需要在极端的环境条件下进行连续的操作，比如极热或极冷、机械震动、化学物质或放射性等环境条件。测试这些系统需要有特殊的设备来提供极端条件。当真实环境中的测试太危险时，就需要用设备来模拟环境。

2.3.2 特定方法

对特定的嵌入式系统，由“TEmb 通用元素”提供的基本测试方法必须与一些特定方法一起使用来处理测试这种特定系统时遇到的具体问题。通常情况下，每一种特定方法都可能有助于测试某一种嵌入式系统，而测试另一种嵌入式系统时则可能毫不相干（因而这里采用的术语是“特定”而不是“通用”）。建立一个完整的所有特定方法的列表当然是不可能的，下面只是给出了一些典型例子中所用到的特定方法：

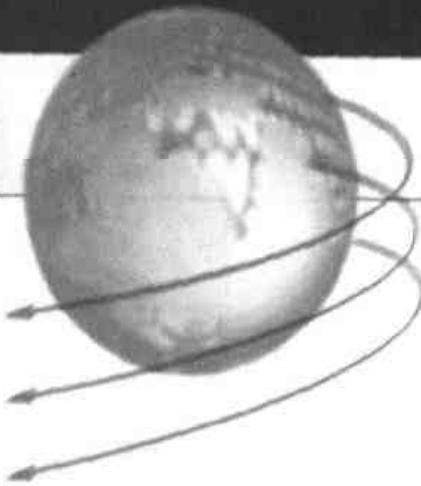
- 特定测试设计技术能够用于对系统基于状态的行为进行测试。
- 在某些开发环境中，系统可能首先被建模，然后动态测试这个模型。这使得在软件编码之前，就可以测试系统的实时行为。
- 有些专门的工具可以用于处理所谓的“隐患检测”，它们不能够发现致命的缺陷，但是能够检查出输入范围内可能导致软件错误的条件。
- “进化算法”（或“遗传算法”）是一种特殊的测试设计技术：测试用例的设计不是从系统文档导出，而是从测试用例本身的行为导出。这是一个优化的过程，能够使测试用例“进化”成更好的测试用例。该算法基于“适者生存”的生物学法则。
- 当测试需要一些昂贵的模拟器时，就有必要成立一个专门的控制部门来进行维护和管理。当许多不同的团队共享这些工具出现冲突的时候，这样的组织尤其有用。
- 英国标准 MOD-00-56 为安全分析活动定义了一个专门的生命周期（见 10.3.1 节）。

2.3.3 LITO 矩阵

系统特性通过引发一些需要由测试方法来解决的问题，从而对测试过程有所影响。特定方法有助于解决与一个或多个系统特性相关的某些问题。特定方法可以按四要素归类：生命周期（L）、基础设施（I）、技术（T）和组织（O）。系统特性和特定方法（在每个要素中）之间的关系可以用一个矩阵来描述，即 LITO 矩阵。该矩阵提供了系统特性和在测试方法中可以采用的特定方法的一个总的关系图。

表 2.3 是 LITO 矩阵的一个例子。它只能被做一个指导方针或起始点，既不是完整的，也不能被认为是惟一正确的描述，有些人也许会提出其他更为重要的系统特性，有些人可能认为表中的某些方法并不实用，而提出更为有效的方法来。只要新方法的提出有助于深入考虑测试是哪种类型的，以及如何用测试方法来进

第二部分 生命周期



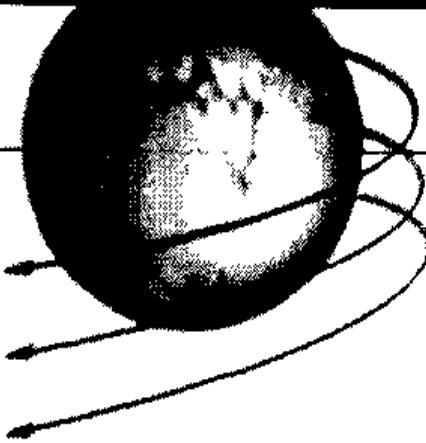
这一部分讲述开发和测试嵌入式软件的过程。生命周期就是将开发和测试过程划分成不同的阶段，描述在不同阶段需要执行哪些活动以及按照哪种顺序来执行。

第3章首先介绍多V模型，描述嵌入式系统的开发生命周期。它基于这样一种思想：在整个开发过程中，不同实物形态的系统被开发出来：首先是一个模拟系统行为的模型，然后是各种各样的原型，在经过一系列迭代之后，最后发展成为“实际”形态。这一章将概述在多V开发生命周期模型中，在什么时候该进行什么样的测试。

在多V开发的复杂环境中，组织测试本身就是一项复杂的任务。在生命周期的不同阶段，会有许多不同领域的专家，从事各种不同的测试。在这种情况下，一个主测试计划（见第4章）将非常有价值。它能够提供所有相关测试活动和测试因素的全貌，并指出它们之间是如何相互关联的。

在开发生命周期的早期阶段（多V模型的左侧），进行的测试是低层次测试，比如单元测试和集成测试。它们通常由开发人员单独完成，或（由测试人员）与开发人员一起协作完成。这些测试通常被组织为开发计划的一部分，并没有单独的测试计划或预算。第5章会讲述如何支持和组织这些低层次的测试。

第二部分 生命周期

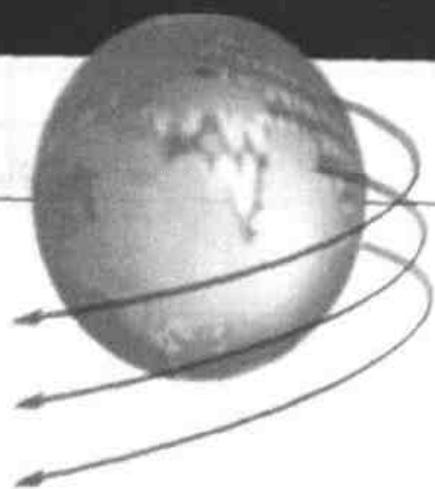


这一部分讲述开发和测试嵌入式软件的过程。生命周期就是将开发和测试过程划分成不同的阶段，描述在不同阶段需要执行哪些活动以及按照哪种顺序来执行。

第3章首先介绍多V模型，描述嵌入式系统的开发生命周期。它基于这样一种思想：在整个开发过程中，不同实物形态的系统被开发出来：首先是一个模拟系统行为的模型，然后是各种各样的原型，在经过一系列迭代之后，最后发展成为“实际”形态。这一章将概述在多V开发生命周期模型中，在什么时候该进行什么样的测试。

在多V开发的复杂环境中，组织测试本身就是一项复杂的任务。在生命周期的不同阶段，会有许多不同领域的专家，从事各种不同的测试。在这种情况下，一个主测试计划（见第4章）将非常有价值。它能够提供所有相关测试活动和测试因素的全貌，并指出它们之间是如何相互关联的。

在开发生命周期的早期阶段（多V模型的左侧），进行的测试是低层次测试，比如单元测试和集成测试。它们通常由开发人员单独完成，或（由测试人员）与开发人员一起协作完成。这些测试通常被组织为开发计划的一部分，并没有单独的测试计划或预算。第5章会讲述如何支持和组织这些低层次的测试。



在开发生命周期后期（多V模型的右侧）的测试是高层次测试，比如系统测试和验收测试。它们通常由一个独立的测试团队来进行，该团队只对单独的测试计划和预算负责。第6章将专门详细地讲述这些测试层次的生命周期。

第3章 多V模型

3.1 介绍

在嵌入式系统中，测试对象不只是可执行代码。通常是按照产品形态逐渐变成成品的顺序来开发一个系统的：首先是在PC上建立一个系统的模型，用于仿真所要求的系统行为。当模型正确无误后，就根据模型生成代码，然后嵌入到原型中。原型的实验性部件逐渐被真实部件所取代，最后形成系统的“最终”形态，可以投入使用并大批量生产。引入这些中间产品形态，当然是因为修改一个原型要比修改最终的产品划算而又快得多，而修改模型会更划算、更快速。

3.1.1 简单的多V模型

多V模型[基于著名的V模型(Spillner, 2000)]是考虑到上述现象而构造的一个开发模型。原则上，每一种产品形态（模型、原型和最终产品）都遵循一个完整的V型开发周期，包括设计、开发和测试活动，因而采用术语“多V模型”（见图3.1）。多V模型的本质是为同一系统开发不同的实物形态，大体上每一种形态都有同样的功能特性。这意味着对模型、原型和最终产品而言，都可以进行完整的功能测试。另一方面，一些详细的技术特性就不能够很好地在模型上进行测试，而必须在原型上进行。又比如，环境条件对系统的影响最好是在最终产品上测试。对不同实物形态进行测试，经常需要特定的技术和特定的测试环境。因此在多V模型和各种各样的测试环境之间，就存在一个清晰的对应关系（参见第13章）。

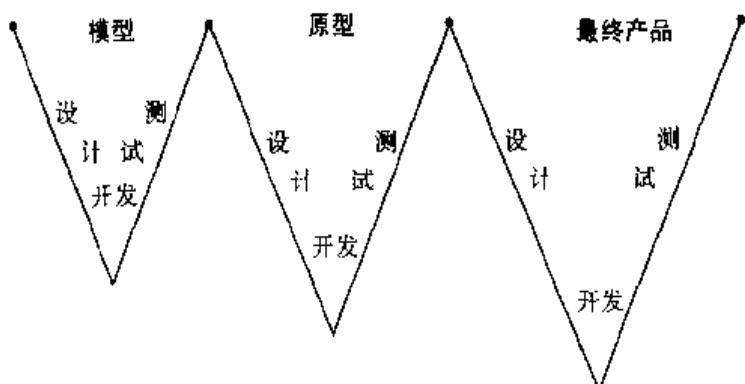


图3.1 多V模型的开发生命周期

3.1.2 迭代与并行开发

多 V 模型展示了 3 个连续的 V 型开发周期（用于模型、原型和最终产品），要能够理解，这对于嵌入式系统的开发过程来说，只不过是一个简化了的描述方式。它不应当被看做是一个简单的连续过程（“瀑布模型”），首先是对原型的全面设计，然后是开发、测试，最后成为产品。而实际上，尤其是开发原型的地方，即中间的“V”处是需要多次迭代开发的，这时可以采用的迭代开发模型有 RUP（Rational 统一过程，Kruchten, 2000）和 XP（极限编程，Beck, 2000）。由于下面的原因，现实中开发一个嵌入式系统是一个复杂的过程：

- 这一般是多团队的项目，包括软件和硬件开发团队。他们通常是独立、并行地工作，这样就会存在着当发现硬件和软件不能很好地协同工作时，已为时太晚的风险。一个好的项目经理需要经常进行交流、集成和测试，而这通常会导致迭代过程：首先实现系统的一小部分功能（包括硬件和软件部分），在集成和测试成功之后，再实现其他方面的功能，依此往复。这些集成活动是项目计划中重要的里程碑，表明项目已经有了坚实的进展。在这一过程中，越来越多的功能得以实现，越来越多的实验部件被真实部件所代替。
- 开发大型复杂系统需要对系统进行（功能）分解，这就导致了部件的并行开发和分阶段集成（它们对多 V 模型的影响将在 3.3 节中进一步解释）。多 V 模型应用于每一个部件的开发。对每一个部件开发一个模型，接下来是硬件部分与软件部分的迭代开发。在开发过程中的任何时刻，都可能会对不同的部件进行集成：很早的时候，例如在原型阶段的实验硬件上进行集成，或者在很晚的时候，例如当所有部件都已经开发完成，对处于成型的“最终产品”进行集成。

上面解释了为什么对于一个特定的嵌入式系统而言其开发必定是一个复杂的过程。在不同时刻会面临许多开发和测试活动，要求有大量的沟通与协调，所涉及的包括开发人员和测试人员在内的每一个人都必须对这一复杂过程有清晰的认识。多 V 模型可能是经过了简化的模型，但仍然可以为认识这一复杂过程打下坚实的基础。下一部分将概述与测试相关的因素，以及它们与多 V 模型的哪一个位置点最为相关。

3.2 多 V 模型中的测试活动

测试过程包含大量的测试活动，其中有许多可以被应用及将被应用的测试设

计技术、必须要执行的测试层次与测试类型（见4.1节），以及需要引起注意的测试相关因素。多V模型能够帮助组织这些活动和因素，通过将它们映射到多V模型上，从而提供对如下问题的深入理解：什么时候是执行测试活动的最佳时间？在开发过程中，在某一个阶段最为相关的测试因素有哪些？

在需要历经多V模型开发的复杂情况下，测试的组织本身就是一项复杂的任务。为了计划和管理好测试过程，测试经理需要对所有相关的测试活动和测试因素，以及它们之间是如何相关的进行全面的了解（见第4章）。当然对于每一个项目来讲，这种全面了解的细节是独有的，但是组织测试过程的通用原则总是可以应用于所有项目。

这一部分将讲述如何将各种各样的测试活动和测试因素映射到多V模型上，讨论了与测试过程相关的广泛的测试活动和测试因素。表3.1列举了这里所考虑的测试活动和测试因素，它们按照字母顺序排列分成三类：测试技术、测试层次与测试类型，还有其他因素。每一类都被放入一个或多个与之相关阶段的“V”中。某些测试因素可能既出现在模型的“V”上，又出现在原型的“V”上，例如“低层次需求”。图3.2到图3.4表现的就是测试活动和测试因素在什么时刻最适合于放在哪个“V”上。

表3.1 开发和测试生命周期中需要被分配的测试相关的活动和因素

技术	测试层次与类型	其他因素
代码覆盖范围分析	体系架构设计确认	体系架构设计
控制流测试	代码审查	认证
Fagan检查	一致性测试	详细设计
故障模型及后果分析(FMEA)	详细设计确认	详细测试计划
故障注入	硬件/软件集成测试	设计&构建工具
故障树分析(FTA)	主机/目标机测试	设计&构建模拟器
正式确认	模型集成测试	设计&构建占位程序(stub)
接口测试	实地检测	设计&构建驱动程序(driver)
模型检查	回归测试	可测性设计
突变(Mutation)测试	需求确认	高层次需求
随机测试	软件验收测试	法律要求
稀有事件测试	软件集成测试	低层次要求
模拟	系统验收测试	主测试计划
状态转换测试	系统集成测试	生产需求
统计使用测试	单元测试	发布标准/建议
		安全计划

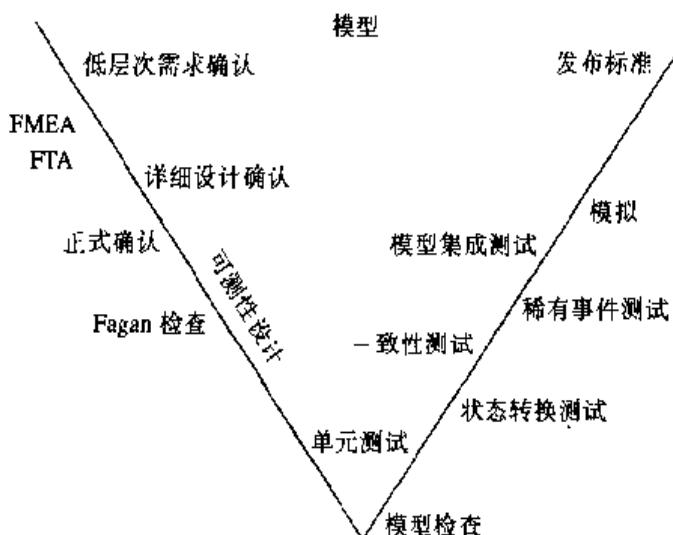


图 3.2 模型开发周期中与测试相关的因素的分配

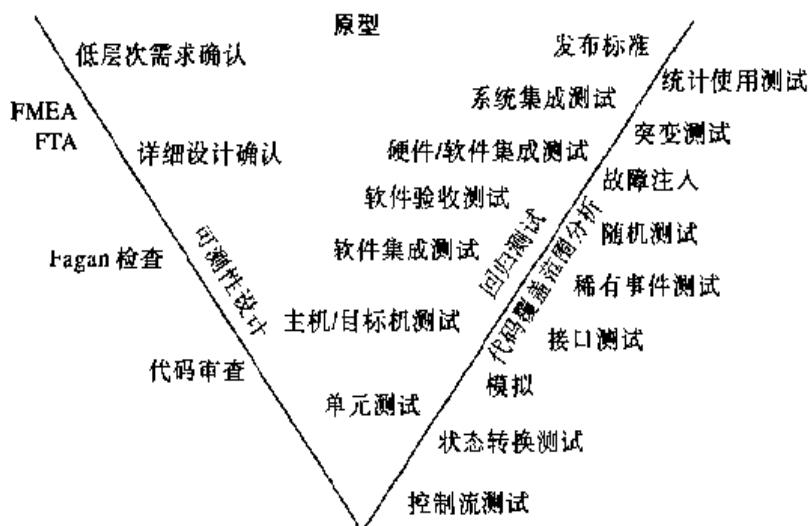


图 3.3 原型开发周期中与测试相关的因素的分配

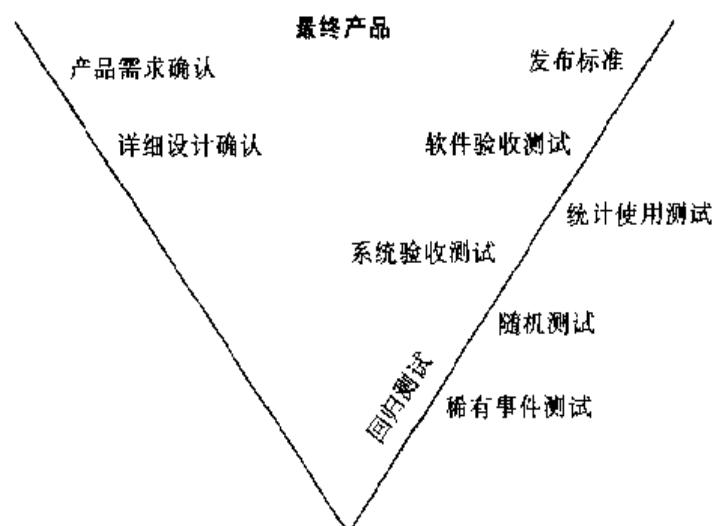


图 3.4 最终产品开发周期中与测试相关的因素的分配

3.3 嵌套多V模型

上面讨论的三个连续V型的多V模型中，没有考虑复杂系统实际的（功能）划分。复杂系统的开发是从一个高层次的需求规范开始，接下来是体系架构设计阶段，决定需要哪些部件（硬件和软件）来实现系统。然后，这些部件被单独开发，最后集成为一个完整的系统。实际上，简单的V模型可以应用于这种开发过程的较高层次上。“V”型的左侧是将系统分解成部件，中间部分由所有部件的并行开发周期组成，右侧是对这些部件的集成，如图3.5所示。

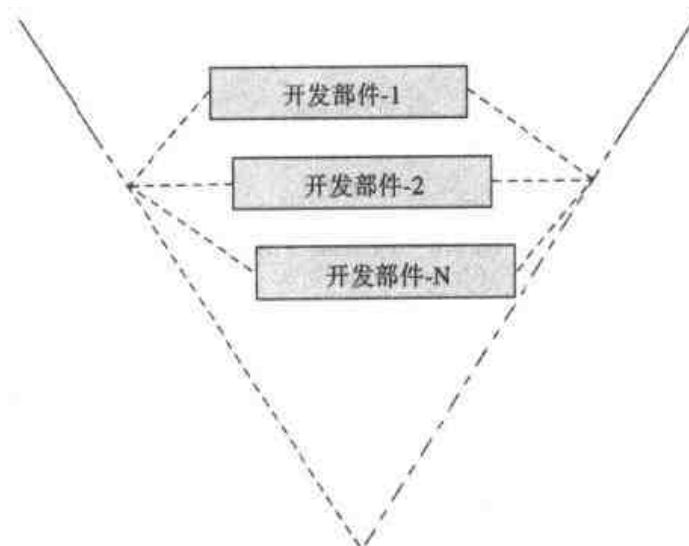


图3.5 一个V模型中的并行开发阶段

这一原则同样适合于那些太大或太复杂的部件，它们不能被当做一个实体来开发。对于这样的部件，需要进行体系架构设计来决定需要哪些子部件。因为这会导致在一个V模型中又包含一个V模型（依此反复……），所以这种开发生命周期模型被称为“嵌套多V模型”。

实际上完全线性的多V模型主要适合于部件级的开发。通常情况下单个部件并不是完整的系统，完整系统首先需要全面建模，然后全面原型化，等等。这是一个逐步构建部件的过程。这可以解释为什么一些开发活动与开发事宜不能够很好地映射到多V模型的3个V上，例如高层次和低层次需求、安全计划、设计与构建特定的工具等，这是因为这些活动在整个开发过程中都需要进行。

将系统级的V模型和部件级的多V模型结合在一起，就构成了我们所说的“嵌套多V模型”（见图3.6）。

利用这一模型，就可以将与测试相关的所有活动和因素分配到模型中合适的位置和级别上。可以将系统级较高层次的测试因素分配到整个开发周期中，如图3.7所示。

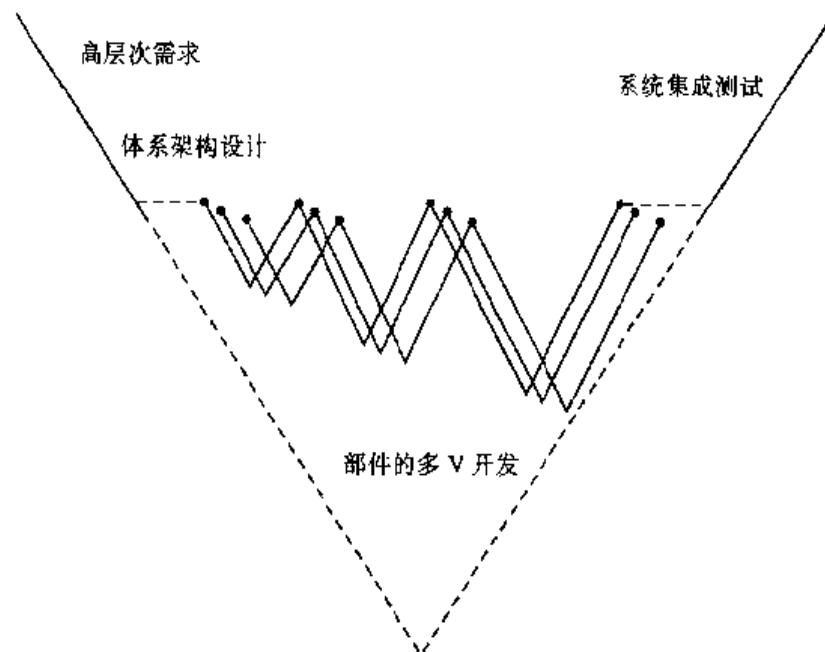


图 3.6 嵌套多 V 模型

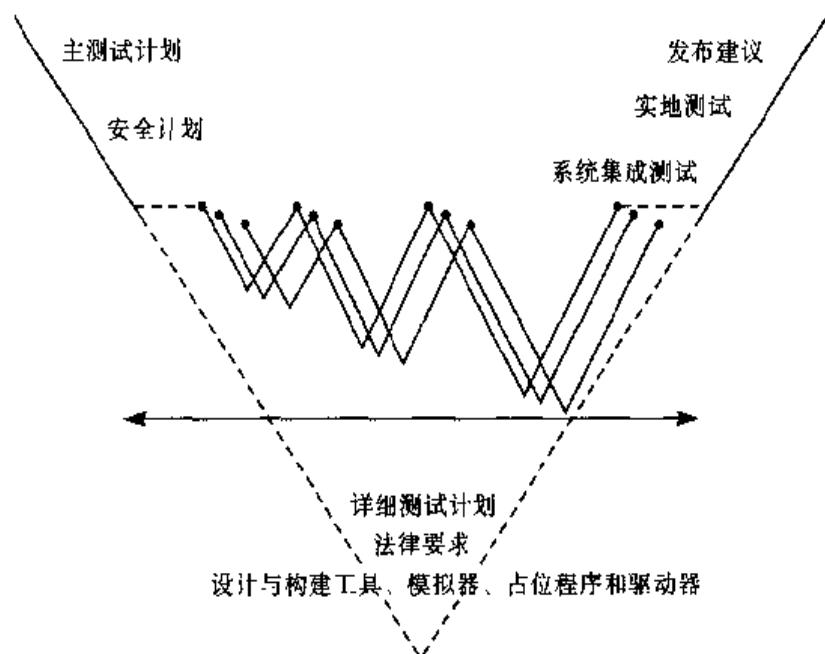


图 3.7 嵌套多 V 模型中较高层次的测试因素

多 V 模型不仅对全新开发的产品测试计划和执行有帮助，而且对于现有产品发布新版本也非常有用。实际上，一个产品的开发项目也就是发布一些新部件。因而多 V 模型以及与整个产品开发相关的完整的（主）测试计划有助于确定与产品开发相关的测试活动。首先应当确定开发活动在多 V 模型中的合适位置，然后用完整的主测试计划来选择集成活动和测试活动，因为一份有效的主测试计划是专用于新产品部件发布的。

第 4 章 制定主测试计划

4.1 制定主测试计划的要素

对一个由许多硬件部件和包含成百上千行代码的软件构成的大型系统进行的测试将是一个复杂的系统工程，它需要许多专业人员在项目的各个时间点执行不同的测试任务。一些人会专门测试某些部件的状态和转换，而一些人会在更高的集成层次上测试状态转换，还有的人会评估用户友好性。一些团队会专门测试性能，另一些团队会专门模拟真实环境。对这些复杂情形，主测试计划提供了控制整个测试过程的机制。

首先应当解释测试类型和测试层次的区别，二者的区别在于一个是测试“哪些方面”，另一个是“由哪些组织实体”来执行测试。主测试计划将二者联系起来。

4.1.1 测试类型

可以对系统的各个方面进行测试，例如功能、性能、用户友好性等等。它们都是描述系统行为不同方面的质量特性。可以采用一些现成的标准，如标准 ISO 9126 就定义了一系列的质量特性。在实际的单个测试中，往往综合了多个质量特性。因而这里引入测试类型的概念。

测试类型是用一组相关的质量特性来评估系统的一组活动。

测试类型是指需要测试什么（和不需要测试什么）。例如进行功能测试的测试人员不会考虑系统的性能，反之亦然。下面的表 4.1 中，描述了一些常见的测试类型。当然这些测试类型并不全面，任何组织都可以在需要的时候随意创建新的测试类型。

表 4.1 测试类型

测试类型	描述	包含的质量特性
功能	测试功能行为（包括处理输入错误）	功能性
接口	测试和其他系统的交互性	连通性
负载和强度	允许大批量数据的处理	连续性，性能

(续表)

测试类型	描述	包含的质量特性
(人工) 支持	在系统运行的环境中提供预期的支持(比如与用户手册规程相一致)	适用性
生产	测试生产规程	可操作性, 连续性
恢复	测试恢复和重启工具	可恢复性
回归	测试在系统改动之后, 是否所有的部件仍然能够正常工作	所有的
安全	测试安全性	安全性
标准	测试是否遵守标准	安全性, 用户友好性
资源	度量所需的资源(内存、数据通信、电力……)	效率

4.1.2 测试层次

测试活动是由许多测试人员和测试团队, 在项目的各个时间段, 在各种各样的环境中所进行的活动。测试活动的组织是引入测试层次概念的原因。

测试层次是一组被当做一个实体来组织和管理的活动。

测试层次是指谁将在什么时候进行测试。不同的测试层次和系统的开发生命周期相关。通过增量测试原则来划分测试过程: 在开发过程的前期, 系统各部分独立进行测试, 检查它们是否符合各自的技术规范。当一些部件达到质量要求后就集成为更大的部件或子系统, 然后测试它们是否符合更高层次的需求。

低层次测试和高层次测试通常是有差异的: 低层次测试只测试单个部件, 在类似于开发的环境中, 在生命周期的早期进行(多 V 模型的左侧); 高层次测试则对集成系统或子系统进行测试, 是在(模拟的)真实环境中, 在生命周期的后期进行(多 V 模型的右侧)。在第 13 章中, 将更详细地探讨针对不同测试层次的不同测试环境。通常情况下, 低层次测试多为面向白盒的测试, 而高层次测试多为面向黑盒的测试。

下面的表 4.2 中, 描述了嵌入式系统中一些常见的测试层次。这个列表并不是完备的, 任何人都可以定义适合自己的列表。

表 4.2 测试层次

测试层次	高/低	环境	目标
硬件单元测试	低层次	实验室	测试单个硬件部件的行为
硬件集成测试	低层次	实验室	测试硬件的连接和协议

(续表)

测试层次	高/低	环境	目标
模型循环	高/低	仿真模型	概念证明；测试控制率；设计优化
软件单元测试	低层次	实验室、主机+目标机处理器	测试单个软件部件的行为
主机/目标机测试			
软件集成测试	低层次	实验室、主机+目标机处理器	测试软件部件之间的交互
硬件/软件集成测试	高层次	实验室、目标机处理器	测试硬件和软件部件之间的交互
系统测试	高层次	模拟真实情况	测试系统的工作是否符合规范
验收测试	高层次	模拟真实情况	测试系统能否满足用户/客户的需求
实地测试	高层次	真实情况	测试在真实条件下系统是否持续工作

4.1.3 主测试计划

如果每一个测试层次都只定义自己最好做的事情，则会存在很大的风险：有些测试被多次执行，而有些测试则可能被遗漏。实际中的情况也是如此，例如在系统测试和验收测试中，使用同样的测试设计技术和相同的系统详细设计，这是对宝贵资源的一种浪费。另一个风险是由于不同测试层次的计划没有协调一致，导致整个测试过程在关键路径上所花的时间比预期的要长。

当然最好是对不同的测试层次进行协调，以防止不必要的冗余，确保能够执行一致、全面的测试策略。需要决定哪个测试层次最适合测试哪些系统需求和哪些质量特性。可以在明确达成协议的时间内，将一些稀缺资源，比如专门的测试设备和特殊的专业人员，分配到不同的测试层次。为了协调以及管理整个测试过程，需要制定一个全面的测试计划，为每个测试层次分配任务、职责和范围。这样的一个全面的测试计划被称为“主测试计划”（见图 4.1）。通常由项目经理将这一任务委派给一个专门任命的测试经理，由他负责所有的测试活动。



图 4.1 主测试计划，对测试层次进行全面协调

可以将主测试计划看做是需要测试什么（测试类型）和由谁来执行测试活动（测试层次）的结合。一旦完成主测试计划，每一个测试层次就能确切知道需要做什么以及能够获得哪些支持和资源。主测试计划为制定每个测试层次的详细测试计划奠定了基础。它并不需要详细规定每个测试层次该做什么，这是详细测试计划该做的事情。主测试计划处理的是确定在哪些领域，测试层次之间可以相互帮助或相互牵制。主测试计划主要关心的三个领域是：

- 测试策略的选择：该测试什么以及如何测试；
- 稀缺资源的分配；
- 相关团队成员之间的沟通。

这些领域是由主测试计划中的以下活动分别处理的：确定主测试策略、指定基础设施和定义组织（见下一节“活动”）。

4.2 活动

测试经理需要在开发过程中的早期就尽可能地准备整个测试过程。这可以从制定主测试计划开始。首先是明确测试的总体目标和职责，定义主测试计划的范围。接下来是对被测系统和开发执行过程进行全面分析。下一步是根据这些分析信息，在组织内讨论需要采取哪些措施和质量保证来确保系统能够成功发布。这由主测试策略来决定，并涉及到要对“该做什么”和“不该做什么”，以及如何来实现它们做出选择。测试经理的主要职责是实现主测试策略，因而他必须定义测试过程所需要的资源（基础设施和人员）。为了便于管理，他还必须定义所有相关团队人员之间的沟通机制和报告机制，换句话说，即对组织进行定义。

一份好的主测试计划不是关起门来冥思苦想就可以完成的，它很大程度上是一种行政性的任务，需要与组织内部的许多成员进行讨论、协商以及说服等工作。这些结果被记录在文档中并由所有的产品权益人来确定。为了制定主测试计划，需要执行下面的活动：

1. 规划任务分配。
2. 全面调查研究。
3. 确定主测试策略。
4. 指定基础设施。
5. 定义组织。
6. 制定整体进度表。

4.2.1 规划任务分配

这一活动的目的是，确保组织中的其他成员对测试组织能够为他们做什么有正确的期望值。测试组织经常会遇到要求他们完成期望值过高的事情，但又无从入手的困境。一旦明确规定了每个成员的任务，而且在组织内部有很好的沟通渠道，那么这种情况就能大大减少，并且得到更好的处理。

对“规划任务分配”，需要讨论以下几个主题：

委托人

委托人就是分配制定主测试计划任务的发起人，即决定应当进行测试的那个人（或组织实体）。这个人可以被看做是测试团队的客户。通常情况下，委托人就是系统开发过程的总（项目）经理，由他来将测试职责委派给测试经理。通常情况下，用户组织和产品管理组织都扮演着客户角色。对测试经理来说，重要的是需要让这些人能够明白，作为客户不仅仅可以提出需求，而且也要提供完成任务的手段（至少要支付报酬）。

承包人

承包人负责制定主测试计划，这个人通常是测试经理。

测试层次

测试层次在主测试计划中定义。需要考虑的有硬件与软件的单元测试、集成测试、系统测试、功能验收测试及产品验收测试。当有的测试层次被排除在主测试计划以外时（例如不受测试经理控制的测试），应当与这些测试层次多进行协调和沟通。

此外，还应当考虑开发专门的测试工具或基础设施中的其他部件。

范围

范围是指对整个测试过程的约束与限制，例如：

- 被测信息系统的惟一标识。
- 与相邻系统的接口。
- 转换或端口活动。

说明什么不属于测试任务分配的范围同样是很重要的。

目标

测试过程的目标可以用交付物来描述。例如：

■ 服务：

- 提供度量质量及其相关风险的管理建议。
- 保持高优先级的问题和解决方案可以立即被测试的环境。
- 对特殊问题提供第三套方案的信息支持。

■ 每个测试层次交付的产品。

前提条件

前提条件描述由“外部”施加给测试过程的条件，例如：

- 确定了的最后期限——当测试任务被委派的时候，信息系统必须完成测试的最终日期通常就已经确定下来了。
- 计划——当测试任务被委派的时候，交付测试基础、测试对象和基础设施的计划通常就已经确定下来了。
- 可用的资源——客户往往对人员、手段、预算和时间都设定了限制。

测试假定

测试假定描述由第三方施加给测试过程的条件，即完成测试过程所需的条件，例如：

- 必要的支持——测试过程需要有各种不同类型的支持，例如测试基础、测试对象和/或基础设施。
- 测试基础的变更——必须把将要发生的变更通知给测试过程。在大多数情况下，这是指参加系统开发过程中的项目会议。

在更进一步的计划过程中，需要更详细地描述前提条件和测试假定。

4.2.2 全面调查研究

这一活动的目的是深入把握（项目）组织，全面了解系统开发过程的目标、将要开发的信息系统以及系统应当满足的需求。它包含以下活动：

- 研究可用的文档。
- 面对面访谈。

研究可用的文档

对客户提供的可用文档进行研究，这里要考虑的有：

- 系统文档，例如信息分析或定义研究的结果文档；
- 项目文档，例如系统开发过程的计划文档；
- 组织的结构和职责，质量计划和（全面的）工作量估计；
- 对包含标准的系统开发方法的描述；
- 对主机平台和目标机平台的描述；
- 与供应商的合同。

如果是对现有系统的升级性开发，则需要调查已有测试件的状态和复用性。

面对面访谈

与系统开发过程所涉及的各个人员面对面地交谈，这里要考虑的有：

- 通过产品市场代表来了解公司目标和产品的卖点；
- 通过用户代表来了解系统中“最受欢迎的功能”和“最不能容忍的缺陷”；
- 通过“实地代表”来了解系统在客户方的生产环境；
- 为了在不同团队之间很好地协调，测试基础、测试对象和基础设施的供应商应当在早期阶段就参加进来。

而且，建议咨询那些和项目有着间接关系的人员，比如会计、律师、制造经理、将来的维修组织等等。

4.2.3 确定主测试策略

这一活动的目的是根据公司希望达到的质量要求，在所有产品权益人中达成一致意见，决定哪些测试该做（及哪些不该做）。要对所有的因素进行利弊权衡并做出决策。百分之百的测试当然是不可能的，或者至少从经济上来讲不是切实可行的，因此公司必须判定一旦系统达不到质量要求可以承担多大风险。测试经理担负着一个重任——向所有产品权益人解释如果不进行某项测试的话会产生什么影响，将会给公司带来多大风险。

在这一过程中，需要确定系统的哪些质量特性是更为重要的，然后确定哪些测试方法最适合覆盖这些质量特性。质量特性越重要，就越需要采用全面的测试设计技术和成熟的测试工具。同时还需要确定这些测试方法适合哪些测试层次。按照这种方法，主测试策略就为每个测试层次设置了目标，它从总体上描述了每

个测试层次需要执行哪些测试，以及为之分配的时间和资源。

确定测试策略涉及到以下活动：

- 检查现有的质量管理措施；
- 确定测试策略；
- 对测试层次的总体评估。

检查现有的质量管理措施

在系统开发过程中，测试是整个质量管理的一部分。一个质量系统可能包含检查、审计、开发一些体系架构、遵循一定的标准和变更控制规程。当确定需要进行哪些测试活动时，那些和测试本身不相关的质量管理活动及其目标也应当考虑。

确定测试策略

这里只是简单描述获得测试策略的步骤，更详细的讨论见第 7 章。

- 确定质量特性 基于风险来为系统选择一组相关的质量特性。这些特性是各种测试活动的主要评判手段，在测试过程的正常报告中必须体现它们。
- 确定质量特性的相对重要性 利用前面步骤的结果，就可以确定出各种质量特性的相对重要性。这里没有任何可供借鉴的东西，因为“相对重要性”是个主观判断，不同的人对特定系统中一些质量特性的重要性有着不同的判断。选择将稀缺资源用于哪些方面的测试可能会产生“利益冲突”。对测试过程来说，重要的是产品权益人应当已经经历过这种冲突局面，每个人都应当理解“最终的决策可能对某个人来说并不完美，但综合考虑，它就是最好的选择”这个道理。
- 为测试层次分配质量特性 为了使稀缺资源的分配最优化，需要确定必须用哪些测试层次才能覆盖所选的质量特性，以及大致的覆盖方式。其结果是一个简明扼要的概述：为了覆盖哪些质量特性，将在哪些测试层次中执行什么样的测试活动。这可以用一个矩阵来表示，其中测试层次作为行，质量特性作为列（见表 4.3）。每一个交叉点都表示从总体来看测试层次是如何覆盖质量特性的。采用下面的符号：
 - “++” 测试层次将完全覆盖质量特性。它是该测试层次的主要目标；
 - “+” 测试层次将覆盖一部分质量特性；
 - [空] 测试层次与该质量特性无关。

如果需要，可以提供更多的解释信息，例如需要使用的特定工具或技术。

表 4.3 一个策略矩阵例子

	功能性	连接性	可靠性	可恢复性	性能	适用性
相对重要性 (%)	40	10	10	5	15	20
单元测试	++			+		
软件集成测试	+	++				
硬件/软件集成测试	+	++		++		
系统测试	++		+		+	
验收测试	+			++		++
实地测试			++		++	

对测试层次的总体评估

对每个测试层次，从策略矩阵就能简单地推断出必须进行哪些测试活动。接下来对测试工作进行总体评估，更详细的评估可以在各个测试层次的计划和控制阶段进行。

4.2.4 指定基础设施

这一活动的目的是确定在测试过程的早期所需要的基础设施，尤其是那些用于服务多个测试层次或需要相对较长时间才能交付的基础设施。

指定基础设施涉及到以下活动：

- 指定所需的测试环境；
- 指定所需的测试工具；
- 确定基础设施计划。

指定所需的测试环境

测试环境由执行测试所需的辅助设施组成，它依赖于系统开发环境和未来的生产环境（见第 13 章）。一般而言，测试环境越是和真实的生产环境相像，其成本也就越高。有时候需要用专门而又昂贵的设备来测试系统特有的属性。主测试计划必须概括定义不同的测试环境，并且要说明每个测试层次需要分配什么样的测试环境。

声明测试过程的特殊要求是很重要的。例如必须模拟随机的外部触发器，或者必须有办法来产生特定的输入信号。

在实际情况下，测试环境往往是固定不变的，这时就必须采取对策，在主测试计划中，必须清楚地说明测试环境不完备可能导致的风险。

指定所需的测试工具

测试工具能够为与计划和控制相关的测试活动、初始化数据和输入集合的建立、测试执行和分析输出提供支持（见 14.1 节）。测试人员所需的工具通常也是开发人员所需要的，例如能够处理输入信号或能够分析系统特定行为的工具。对于如何共享这些工具及其用法，双方应当讨论并达成一致意见。

当某个必备工具无法得到而必须专门开发时，应当把它视为一个独立的开发项目。主测试计划必须清楚地说明该工具的开发项目和主测试计划中测试活动之间的相关性。

确定基础设施计划

确定对所有必要的基础设施部件进一步细化、选择、采购或开发的职责，制定时间进度表，记录达成的各种协议。而且，还必须大致描述各种辅助设施的可获得性。

4.2.5 定义组织

这一活动的目的是在整个测试过程的层面上，定义测试中的角色、权限、任务和职责。在第 21 章中将详细描述各种形式的组织及其所扮演的角色。建立组织涉及到以下活动：

- 确定所需的角色；
- 建立培训机制；
- 分配角色（或任务）、权限和职责。

确定所需的角色

在各种不同测试层次之间，需要确定哪些角色是必须的，以便能够协调安排好人员。这些角色并不是和单个测试层次相关，而是和整个测试过程相关。这要涉及到协调不同的进度、优化使用稀缺资源以及信息收集和报告的一致性。这里尤其需要考虑的有：

- 全面的测试管理与协调；
- 集中式控制，比如对基础设施或缺陷管理的集中式控制；
- 集中式质量保证。

要达到这些目标，线性组织中可能常聘用以下几方面的专业人员：

- 测试策略管理；
- 测试配置管理；
- 测试方法与技术支持；
- 计划和监督。

在考虑测试层次的计划和控制阶段时，就需要确定各个测试层次内部所需的测试角色。

建立培训机制

如果参与测试的人员对于基本的测试原理、测试中用到的特定技术以及工具不是很精通，那么就应当对他们进行培训（见第18章）。可以利用商业化的课程，或者是组织内开发的课程来进行培训。主测试计划必须为培训预留足够的时间。

分配任务、权限和职责

对所定义的测试角色，要为其分配相应的任务、权限和职责。这尤其适用于在各个测试层次之间相互协调以及要采取决策时，例如：

- 制定各个测试层次所交付产品的规章制度；
- 监督规章制度的执行（内部审查）；
- 协调各个测试层次之间的公共测试活动，比如技术性基础设施的建立和控制；
- 在整个测试过程中的各测试层次之间以及各团队人员之间，制定沟通与报告的指导性文件；
- 建立方法、技术和功能支持；
- 保持各项测试计划的一致性。

4.2.6 制定整体进度表

这一活动的目的是为整个测试过程设计一个整体进度表。这里所说的整个测试过程包括（在主测试计划范围之内的）所有的测试层次，诸如基础设施部件的开发以及人员培训等特定活动。

对每个测试层次而言，确定其开始时间、结束时间及交付物。在各个测试层次的计划和控制阶段，制定出更详细的计划。

整体进度表至少应该包含以下内容：

- 描述（每个测试层次的各个阶段）要执行的高层次行动；
- 这些行动的交付物；
- 为每个测试层次分配的时间（人-时）；
- 要求的交付时间；
- 与其他活动（测试过程内部或外部的活动以及测试层次之间的活动）的关系及依赖性。

各个测试层次之间的相互依赖性尤其重要。毕竟，多数测试层次的执行阶段都是按顺序来执行的：首先是单元测试，然后是集成测试和系统测试，最后是验收测试。这往往也正是整个测试过程的关键路径。

第 5 章 由开发人员执行的测试

5.1 介绍

再也不会出现由一个开发人员单独开发整个系统的情况了。现在的系统都是由一个大团队来开发，有时候细分为每个团队开发一个子系统，有的团队位于不同的地方，甚至位于不同的洲。开发的系统庞大而又复杂，因而对质量的要求越来越高，同时还要考虑市场的激烈竞争或系统的使用强调安全的情形。这些都说明了在系统开发的早期进行测试乃至进行良好测试的必要性。单独测试团队的存在，并不意味着开发阶段的测试就不重要了。在最终产品的准备过程中，为了达到预期的质量要求，开发团队与测试团队都有着各自的重要角色。单独的测试团队主要是根据需求来执行测试，他们的目标是确保系统满足这些需求。相反，开发人员是利用软件的内部结构知识，从单元级开始测试。他们同样利用这些知识来测试不同单元的集成，以便提交一个稳定的系统。按照 Beizer (1990) 的说法，理论上如果没有时间限制的话，基于需求的测试能够找出所有的 bug。由于单元测试和集成测试都有时间限制，因此即使完整地执行两种测试，也不能找出所有的 bug。只有结合开发人员和测试人员两方面的测试，并且采用基于风险的测试策略，才能发现重要的缺陷。这使得对于生产符合质量要求的系统来讲，两种类型的测试都是必要的。下面的理由能够解释为什么开发人员的测试是非常重要的：

- 早期发现的错误容易解决。一般来讲，修复缺陷的费用会随时间的推移而上升 (Boehm, 1981)。
- 高质量的基础元素更容易建立起高质量的系统，相反，低质量的基础元素将导致不可靠的系统，而通过功能测试几乎不可能解决这个问题。
- 在开发后期发现的缺陷，很难追溯到其根源。
- 必须解决开发后期发现的缺陷时，将导致时间浪费在回归测试上。
- 开发阶段的良好测试，将对整个项目时间产生积极影响。
- 异常处理只有在单元级才能被很好地测试。

基本上不能通过产品后期的测试来提高质量，而应当从一开始严把产品质量关。为检查每个开发阶段的质量，测试是很必要的。

许多开发人员并没有意识到测试是开发过程中最有益的工作。为了使开发人

员的测试工作达到验收水平，那么测试就必须是有效的，而有效的测试必须选择集成策略。在 5.2 节中将讲述各种不同的集成策略，在 5.3 节里将讲述计划和控制各种必须的测试活动。

5.2 集成方法

要想获得稳定而功能正确的系统，仅进行单元测试是不够的。许多缺陷与模块的集成有关。如果需求没有被正式描述，那么每个人就要对需求做出自己的解释。只要这些解释与其他模块的交互无关，那就没有什么问题。模块之间的错误交互，通常都是由于各自对需求有不同的解释而引起的。检测这些缺陷的最好手段是集成测试。集成策略就用来确定如何将不同的模块集成到一个完整的系统中。这里的集成包含硬件和软件两方面。由于不同的软件模块之间、不同的硬件之间以及硬件和软件之间都存在依赖性，因而必须决定采用哪种集成策略。在特定时刻，所有这些部分都必须准备好以便集成。这个时刻依赖于集成策略。由于集成策略对于项目活动的时间安排有重大影响，应当尽可能早地确定采用哪种策略。自上向下集成、自下向上集成和混合 (big bang) 集成是三种不同的基本策略。因为这三种策略并不相互排斥，因此基于这三种策略的组合可以派生出多种策略。集成策略依赖于：

- 集成部件的可用性（例如第三方软件或硬件）。
- 系统规模。
- 是新系统还是在现有系统上增加/改变功能。
- 体系架构。

混合集成

这一策略只能在下列条件下才能获得成功：

- 系统的绝大部分是稳定的，只需添加小部分新的模块。
- 系统规模相对较小。
- 各模块之间是紧耦合的，几乎不可能逐步集成不同的模块。

这一策略十分简单，只需将所有的模块集成在一块，将系统当成一个整体进行测试就可以了。

其主要好处是不需要使用占位 (stub) 与驱动程序，而且策略也很简单。不过比较难以发现引起缺陷的原因，而且只有在所有的模块都准备好了的情况下，才能开始集成。

自下向上集成

对任何系统几乎都可以采用这一策略。该策略是从低层次的、相互之间依赖性最少的模块开始的，可以用驱动程序来测试这些模块。这种策略能用来逐步建立系统，或者首先并行地建立起子系统，然后集成为一个完整系统。这种集成可以从开发过程的早期就开始进行。当然，如果项目计划中模块提交也是采用自下而上的方式，那么采用这种方法就能够尽早检测出接口问题，而且这些接口问题也比较容易被隔离，因此解决起来成本就低。其主要缺点是需要使用许多驱动程序来执行这一策略，而且因为测试需要迭代，所以也是一种非常耗时的策略。

自上向下集成

这种策略由系统的控制结构来引导。控制结构按照自上向下的顺序开发，这也提供了从上层控制模块开始，自上而下集成模块的能力。对每一个新的层次，位于同一层次的相关模块被集成起来并得到测试。还不存在的模块角色可以用占位来实现。采用该集成策略的一个缺点是：如果需求发生了变化，变化对底层模块产生影响，从而也将导致上层模块需要更改。这可能导致需要（部分）重新开始集成以及测试过程。另一个缺点是用于测试每个集成步骤所必须用的占位数目很大。如果在早期从上层模块开始集成测试，即使采用占位来替代系统的主要部件，仍然可以观察到整个系统的概貌以及工作方式。

集中式集成

下列情形可以采用集中式集成：

- 当系统的中心部分对其他部分的运行必不可少时（比如操作系统的内核）。
- 必须有中心部分才能进行测试，而且该部分很难由占位来代替。
- 系统的体系架构是这样的：首先开发中心部分作为产品，然后发布新模块或子系统来升级系统或增加新的功能。

首先测试系统的中心部分，下一步是集成控制结构。利用自下而上或自上而下策略来并行测试耦合子系统。这种方法的瓶颈在于系统中心部分的集成。有时唯一可用的集成策略是混合策略，因为中心部分的模块经常联系非常紧密。只有在核心部件经过非常充分的测试之后，这种策略才能获得成功。这种方法很突出的优点是可以采用不同集成策略的组合，而且可以为每个系统部件选择最有效的策略。

分层集成

这种策略可以用于分层式体系架构的系统。在这种架构中，各层之间仅通过接口与其上下层直接相连。每一层可以单独使用自上而下、自下而上或混合策略来进行测试，然后按照自上而下或自下而上策略来集成每一层。其优点和缺点与自上而下以及自下而上策略是一样的。这种接口的集成和分隔比较容易，因此发现缺陷的根源也变得容易了。

客户/服务器集成

这种策略用于客户/服务体系架构。在客户端，可以通过自上而下、自下而上或混合策略来集成，而服务器可以用占位和驱动程序来替代。反过来在服务器端也可以采用这样的方法。最后将服务器和客户端集成到一起。

协作集成

协作是一组对象为达到共同目标而进行的合作，比如实现一个用例。由于系统必须实现多个用例，因而它支持多个协作。许多对象都是多个协作的一部分。这意味着合理选择协作就可以覆盖整个系统。协作集成只适用于面向对象的系统。只有在协作被清楚定义而且覆盖所有的部件和接口之后，才能采用这种策略。协作覆盖所有部件和接口也是一个缺点，因为在不同协作之间细微的依赖关系可能被排除在协作模型之外。因为一个协作的所有部件需要组装在一起，而且只有当该协作完成之后，集成测试才可以开始，因而这种策略某种程度上是混合方式的实现。它只需进行少数的测试，因为测试重点是端对端的功能。由于协作之间相互重叠，因而没有必要测试每一个协作。因为已经测试了一组相关的部件，因此不需要或者只需要很少一些占位和驱动程序就可以进行测试。

5.2.1 应用集成师

虽然单元测试和集成测试是两个不同的测试层次，但二者联系紧密。单元测试的质量对集成过程的效率有很大的影响。主测试计划中描述这两种测试层次的范围和所做的事情。实际中仍然需要有人来协调这两个层次。协调这两个层次的人被称为应用集成师（AI）。AI 对集成过程的进展以及交付的系统质量负责。AI 和客户（开发项目领导或团队领导）共同决定希望达到的质量等级。质量等级以输出标准的形式来度量。为了满足输出标准，必须使集成部件在被用于集成过程之前，具备一定的质量等级。这种期望的质量等级是集成部件的输入标准。AI 是每个集成部件的入口，由他判定该部件是否满足输入标准。这区别于另外一种测

试——集成部件的提交者，即开发人员，必须证明该部件满足输入标准。只有当所有的输出标准都满足之后，系统才能被发布（见图 5.1）。

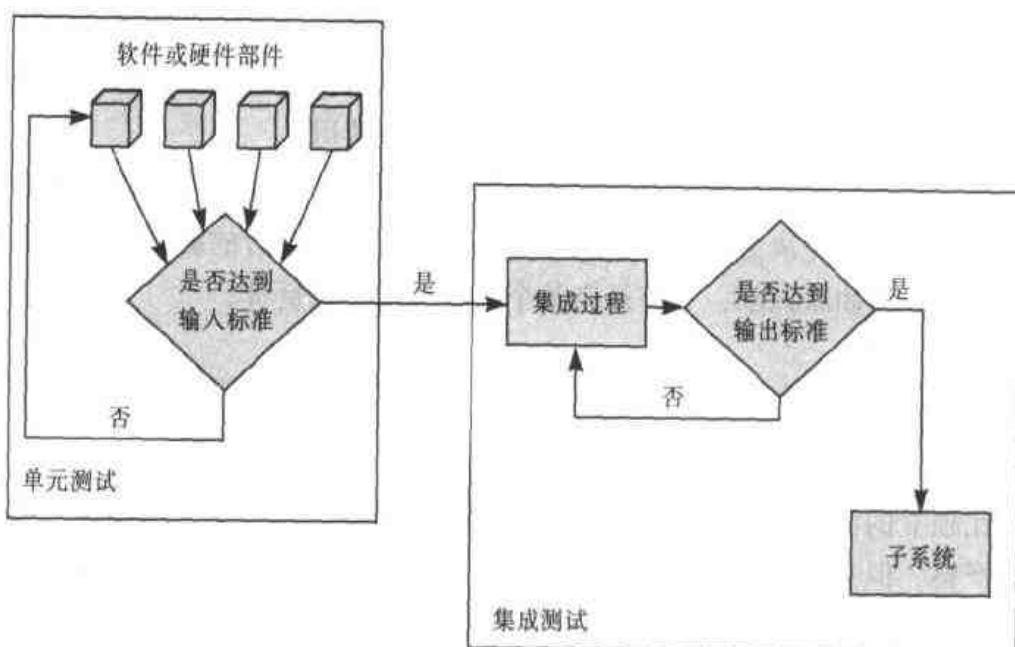


图 5.1 单元测试和集成测试的关系

输入标准和输出标准是成功的集成过程中最为重要的手段。这些标准越具体，越能更好地使用。输入标准清楚地说明了开发者应当提交什么，以及部件何时才能够被接受。输出标准清楚地说明了集成过程在何时成功完成。当以这种方式来使用这些标准时，它们就是非常有力的控制手段。输入标准和输出标准可能包含这些方面：待测试的质量特性、覆盖范围的期望程度、一些测试设计技术的运用以及能够提供的证据。如果标准足够详细，那么它们可以为单元和集成测试的测试策略开发提供有用的信息。可以使集成过程以并行方式进行，比如同时集成两个子系统，这两个子系统会在某个时刻被集成为一个完整的系统。AI 的工作方式主要由所选择的集成策略来决定。

一个 AI 的最大控制范围是十个开发人员。当开发人员超过十个时，就需要增加 AI 人数。AI 越多，就意味着使用更多的集成策略的综合。而且，当硬件和软件集成都是该过程的一部分时，就至少需要两个 AI，这是因为对硬件和软件来说，需要不同的知识。

在实践中已经表明，采用 AI 方法后续测试将产生的严重缺陷非常少。有时在完成集成过程之前，就有可能开始下一个测试层次的测试，其先决条件是对构造中的系统已经有足够的信心。开发人员能够比以前更早地获得反馈，尽管并不是每一个缺陷都与开发人员的出错相关，但通过对错误的研究，能够让他们提高自己。发现错误越早，其解决成本就越低。这种方法能够提供更稳定的模块，从而

对整个系统的稳定性产生积极作用。

AI在项目中只扮演应用集成师这一个角色，他才能够发挥作用。AI被委任对系统质量负责，这会和开发项目领导的职责发生冲突，开发项目领导必须在一定的预算之内按时提交系统。有时候他们相互对立，委托人根据他们的信息来决定下面哪个是更重要的：质量、产品上市的时间以及费用。

这种方法使得我们必须有意识地做出选择，即对于预期的质量以及递交给下一个测试层次之前完成测试，做出选择。由于开发人员进行的测试变得越来越透明，因而在开发团队内部，一定得有某个人对测试真正负责。

5.3 生命周期

相比独立的测试团队，开发人员测试的生命周期并没有那么结构化，要求也没那么严格，但是当项目扩大或风险增加时，生命周期作为控制工具就变得越来越重要了。实际上，这意味着在独立团队测试和开发人员测试的生命周期之间，已经没有什么区别了。

5.3.1 计划与控制

计划与控制阶段由下面的活动组成：

- 明确任务。
- 建立测试基础。
- 定义测试策略。
- 列出测试交付清单。
- 设置组织。
- 定义基础设施。
- 组织控制。
- 建立进度表。
- 合并与维护测试计划。
- 控制测试。
- 报告。

明确任务

指定客户（委托人）和承包人，确定测试范围，描述前提条件。

建立测试基础

明确完成任务必需的所有文档(功能和技术设计、源代码、改动需求、标准)。这些文档的最新版本应当是可用的，而且应当是最新的。这些文档是设计测试用例的基础。编码标准可用来配置静态分析器(见14.2.4节)，以便检测与标准的偏差。当出于安全原因而使用编码语言的一个子集时，毫无疑问这是非常重要的。

定义测试策略

测试策略指测试什么、如何测试以及测试到什么程度。测试策略也是时间和资源分配的基础。单元和集成测试的测试策略基于主测试计划中定义的策略。在单元和集成测试期间，功能、性能和可维护性是测试的主要质量特性。对功能测试而言，运算法则测试和基本比较测试这两个测试设计技术非常有用。对性能测试而言，可采用统计使用测试。而对可维护性测试，可采用审查清单。单元测试主要测试单元的功能，而集成测试则主要测试接口功能、系统或系统部件的性能。

列出测试交付清单和报告

和委托人一起，决定需要交付的测试产品是什么，一旦进行测试并取得进展，如何向委托人通报产品的状况。测试可以交付测试用例、报告、覆盖范围度量、自动测试脚本等。

设置组织

单元测试和集成测试要求有一个结构化的组织。应当制定出主要的测试任务和测试职责。通常是开发项目领导对单元和集成测试负责。不幸的是，开发项目领导也需要对产品的按时提交和预算负责。好的测试需要花费更多时间，因此费用也更高。当项目有时间压力时，最先的动作往往就是砍掉测试。通过引入应用集成师(AI)的角色，由另一个人对测试和所提交系统的质量负责。开发项目领导再也不能自己决定砍掉测试，而必须和AI一起向委托人报告项目进展，委托人基于所报告的信息来决定怎么做。

定义基础设施

单元测试和集成测试可能对基础设施有着特定的需求。比如用于单元测试的覆盖范围工具(见14.2.4节)、应当开发的占位和驱动程序(见14.2.4节)，有时还必须建立模拟环境(见第13章)。对一些独特的项目，有时还需要开发专门的测试工具。所有这些开发活动都应当有计划安排而且按时开始，否则测试需要的

时候就无从得到它们。

组织控制

因为测试活动是开发过程的一部分，所以控制也是日常开发活动的一部分。独立测试团队使用的缺陷管理工具非常庞大，在开发期间无法使用。但是如果单元产品或集成为完整系统的产品带着缺陷被提交，那么这些缺陷需要在一个文档或缺陷管理工具中描述。

建立进度表

通过对时间和资源的分配可以控制测试过程。委托人可以获知测试的进展情况，能够及时意识到时间的延误，并采取措施赶上进度。

合并、维护测试计划

以上提到的所有活动提供了编写测试计划的必要信息。测试计划是一个动态文档，当项目状况需要时可以调整计划。

测试控制

在整个过程中，测试控制是很重要的一环。有三种不同形式的控制：测试过程控制、基础设施控制和测试交付控制。对于按时提交质量过关的合适系统而言，测试过程控制是必需的。

报告

按照要求，委托人需要定期地得到测试进度和被测对象的质量报告，进度和质量报告经常综合在一起。报告应当提供下列信息：

- 测试用例的总数。
- 尚需执行的测试用例的数量。
- 成功完成的测试用例的数量。
- 测试了哪些部分，以及仍然有待解决的问题。
- 与测试策略的偏离情况，以及发生的原因。

5.3.2 准备阶段

准备阶段的两个主要任务是：

- 审查测试基础的可测性。

■ 描述基础设施。

可测性审查

可测性意味着测试基础的全面性、连贯性和可获得性，以及是否可获得导出测试用例的信息。多数情况下，使用审查清单来进行可测性审查。

描述基础设施

集成测试通常并不是在真实环境中进行的，以避免复杂因素和控制问题的出现。真实环境应当被模拟出来。必须开发模拟环境（基础设施）而且必须按时准备好。在模拟环境中应当包含各种类型的措施，以使测试容易进行。有关模拟的更多细节见第13章。

有时候，系统开发的第一阶段是模型，利用这个模型（例如状态转换图）来（自动）生成代码。当模型通过它们被设计的同一抽象级别的检查时，测试和调试就有效得多（Douglass, 1999）。这意味着在生成任何代码之前，测试就已经开始了。有些建模工具支持在工具环境中运行模型，如果模型能够在工具环境内的模拟环境中运行，则会更好。如果这种早期的测试对项目团队有好处，则需要选择一个支持这种测试的建模工具。

5.3.3 细化阶段

在细化阶段，使用测试设计技术导出测试用例。在细化阶段，有两项主要的活动：

- 细化测试。
- 创建工具。

细化测试

每一个单元或集成步骤的测试用例，都是在指定的测试设计技术的基础上创建的。没有指定测试设计技术，并不意味着就没必要测试。开发人员需要基于待测单元的知识来设计和执行测试。有可能在文档详细设计过程中就能够发现测试基础中的缺陷，应当通报并详细阐明这些缺陷。

采用极限编程时，测试以小段测试程序的形式来指定。在预期的软件编码工作实际开始之前，测试程序就指定了。当测试程序发现不了任何缺陷时，测试对象就通过了测试。

创建工具

对测试单元和集成步骤而言，经常需要使用占位和驱动程序。这些占位和驱动程序可以在细化阶段进行开发。如果采用普通的体系架构来搭建测试站（testbed），就有可能使测试站和一个自动测试套件相连（见 14.2.4 节）。现在这组测试套件能够更容易地用来搭建自动测试（细节请见第 15 章）。

如果必须使用模拟器，而且模拟器专门用于该项目，则应当尽早进行开发。模拟器的开发将会花费相当多的时间，而且必须在测试执行之前就完成。

5.3.4 执行阶段

执行阶段执行测试用例并记录结果。将实际结果和预期结果加以比较，如果有差异，则进行分析。如果这种差异是由于测试对象的问题而引起的，那么就应当解决这种问题。对单元测试和集成测试来说，其终止标准稍有些不同。

单元测试的终止标准用集成测试的输入标准来描述。如果待测单元达到了集成测试的输入标准，那么终止单元测试。

当所有集成部件都被集成起来，而且待测系统符合集成测试的输出标准时，则终止集成测试。

单元测试的执行者通常是该单元的开发者。开发人员熟悉系统的内部操作，因此能够快速测试自己的工作。因为开发人员对自己所建造的产品存在盲点或盲目的信心，因而很有可能遗漏某些缺陷。由另外一个开发人员来测试这个单元就可以克服这个问题，尽管测试可能没有开发者本人测试来得快。极限编程（Beck, 2000）中的两个原则可以解决这些问题。第一个原则是：在实际的单元开发之前，首先构造一个测试程序来测试编程。如果已开发的单元通过了测试程序而且没有错误，则认为它是正确的。第二个原则是对等编程，即由两个开发人员共同开发和测试一个单元，并且随时修复单元的错误。

5.3.5 完成阶段

完成阶段的任务就是报告测试对象的状况。如果测试对象的质量满足标准，则可以将它提交给下一个测试层次。在必要时可以对测试工具（testware）进行调整，或者将测试工具保留以备发布后续版本或相似产品时使用。

第 6 章 独立测试团队的测试

6.1 介绍

独立测试团队主要进行高层次的测试。这类测试在开发生命周期的后期进行，这就是为什么这些团队的测试活动是处在开发过程的关键路径上的原因。这也意味着在这个关键路径上，能够缩短测试时间的工具会非常受欢迎。生命周期模型就是这样一个工具，它能清楚地区分支持活动和测试执行活动。支持活动的目标是使得测试执行所需的时间达到最少，通过和开发活动并行执行的方式，支持活动被安排在关键路径之外。如果支持活动计划得好，那么当测试团队完成测试用例描述时，被测对象被同时交付。

因为涉及到许多产品权益人，因而这里描述的生命周期更为正规。测试团队所需的大多数信息，应当是由团队以外的组织来提供。而且并不是每一个产品权益人都和测试团队一样有相同的目标和期望值。有时因为产品的认证，测试过程的成果要受到外部审查。这些就是测试过程正规化的全部理由。

6.2 计划与控制阶段

目标

为了在给定的时间、预算以及分配的人员范围内把握测试对象的质量，而对测试过程进行协调、监督和控制。

规程

在系统的功能细化或设计阶段过程中，计划控制阶段也就开始了。为了进行计划，需要执行各种各样的活动，这些活动将在下而详细描述。所有这些活动的结果就是测试计划的基础。测试计划应当基于主测试计划来制定。如果没有主测试计划，则可以在各个测试层次上单独开发测试计划。

测试计划包含计划、资源分配和任务范围划定。因此测试计划是协调、监督和控制测试过程的基础。附录 E 给出了一个测试计划的例子。

活动

计划与控制阶段包含下列活动：

1. 分配任务。
2. 整体评审与研究。
3. 建立测试基础。
4. 确定测试策略。
5. 设置组织。
6. 列出测试交付清单。
7. 定义基础设施。
8. 组织管理和控制。
9. 制定测试过程进度表。
10. 整理测试计划。

在测试过程的协调、监督和控制框架内，能够确定下列活动：

11. 维护测试计划。
12. 控制测试。
13. 报告。
14. 建立详细进度表。

图 6.1 描述了计划与控制阶段中各活动之间的相关性。

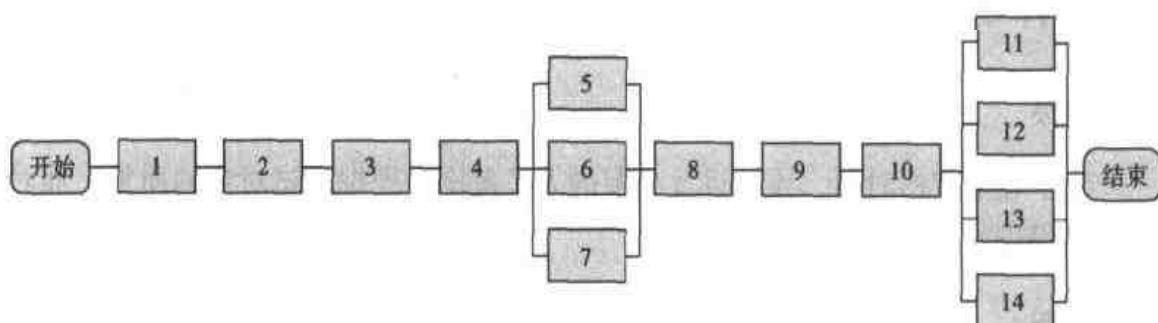


图 6.1 计划与控制活动

6.2.1 分配任务

目标

分配任务的目标是确定谁是委托人和承包人，测试过程的范围和目标是什么，测试过程的前提条件是什么。

规程

需要确定下列事情：

- **委托人** 测试任务的提供者，因此也必须向他报告进度、问题，同时需要他同意对测试进行调整。
- **承包人** 负责执行测试任务的人。
- **范围** 测试对象是什么，以及不对什么进行测试。例如：
 - 被测系统的惟一标识；
 - 和相邻系统的接口；
- **目标** 测试过程的预期结果，例如：
 - 交付的产品；
 - 需测试的质量特性，例如：
 - 功能；
 - 可维护性；
 - 性能；
- **前提条件** 有两种类型的前提条件。第一种是强加给测试过程的前提条件（外部），第二种是在测试过程中强加的条件（内部）。

下面是外部前提条件的例子：

- **最后期限** 完成测试的最后期限是固定的，因为开始生产和/或产品介绍的时间已经安排好了。
- **进度** 如果开发过程已经开始进行，那么通常测试任务也就已经分配了，因此结果、测试基础、测试对象和基础设施被交付的进度也已经安排好了。
- **分配的资源** 委托人通常对人力资源、预算和时间设定了限制。

下面是内部前提条件的例子：

- **对测试基础的支持** 必须有人来回答有关测试基础的不明确的问题。
- **对测试对象的支持** 当缺陷阻碍了测试进程时，必须有开发团队的及时支持。修复这类缺陷的优先级必须很高，因为它严重威胁到测试过程的进展。
- **测试基础的变更** 测试团队必须能被及时通知有关测试基础的变更。多数情况下，这仅仅意味着与系统开发过程中原有的规程连接起来。

6.2.2 整体评审与研究

目标

这一活动的目标是深入把握可用系统和项目文档、功能和质量方面的系统需求、系统开发过程的组织、测试领域可用的知识和经验、验收测试涉及的用户要求。

规程

规程由下面的子活动组成：

- 研究可用的文档。
- 面对面的访谈。

研究可用的文档

对可用的文档进行研究。如果测试过程涉及到维护测试，则还需要对现有测试件的到位情况和有用性进行调查。

面对面的访谈

与委托人、市场经理（作为客户的代表）以及开发领导进行面对面的访谈，以便提供有关测试对象以及测试需求的必要背景信息。

6.2.3 建立测试基础

目标

这一活动的目标是确定测试基础是什么，以及基本的文档有哪些。

规程

选择相关的文档，确定文档的状态和文档的交付进度。

为了能正确地执行测试并成功完成测试任务，需要知道与系统功能和质量相关的要求。所有描述这些需求的文档都包含在测试基础之中。

6.2.4 确定测试策略

目标

这一活动的目标是决定测试什么、如何测试以及测试的范围是什么。

规程

任务被转换为测试过程的具体方法。规程由下列子活动组成：

- 策略开发。
- 草拟预算。

策略开发

测试策略描述了系统的哪些部分必须彻底测试，哪些部分可以少些测试（细节请见第7章）。

草拟预算

基于测试策略并考虑可用的资源，草拟出测试过程的可靠预算。

一个测试过程包含许多不确定因素，因此如果在预算中包含一个“不可预见”项，则对于预算的预期可能会很有用。这一项通常占整个预算的10%到15%。

6.2.5 设置组织

目标

这一活动的目标是确定如何设立测试组织、包括角色、任务、权力、责任、层次、磋商结构以及报告流程等，同时也需要考虑对培训的需求。关于测试团队组织的更多信息，可以在第17章、第18章和第19章中找到。

规程

- 描述测试过程中明确的任务、权力和责任，以及如何把它们分配给测试角色。任务和该生命周期模型中所描述的活动相关。
- 描述各种测试功能之间的关系、测试团队内部关系以及与系统开发过程中涉及的其他团队的关系。
- 当已经确定了测试过程中所需的测试角色时，就需要吸收必要的新成员，这得考虑成员的技能以及每个测试角色所要求的知识和技能（见第17章）。一个人可担当多个角色。
- 当结构化测试或自动化测试的知识不够用时，需要建立培训课程。
- 如果组织内部没有人具备所需的知识而且培训起来特别浪费时间，就需要从外部雇佣。
- 建立报告流程。

6.2.6 列出测试交付清单

目标

这一活动的目标是指定由测试团队交付的产品。

规程

指定测试交付物包含下列活动：

- 建立测试件。
- 建立标准。

建立测试件

测试件被定义为测试过程中产生的所有测试文档，它们可以用于后续产品发布，因此必须是可移植的并且是可维护的。比如在回归测试中，就经常使用现有的测试件。测试件的例子包括测试计划、逻辑和物理测试设计、测试输出等。

建立标准

需要确定测试交付物命名的标准和协议。如果可能的话，需要为各种不同文档创建模板或是利用已有的模板。

6.2.7 指定基础设施

目标

这一活动的目标是在测试过程早期就确定所需的基础设施。

规程

规程包含下列子活动：

- 指定测试环境。
- 指定测试工具。
- 建立基础设施进度表。

指定测试环境

描述所需要的测试环境。测试环境包括执行测试所必需的设施（见第 13 章）。

指定测试工具

描述所需要的测试工具。在计划与控制、测试执行和结果分析方面，测试工具可以为测试活动提供支持（见第 14 章和第 15 章）。

建立基础设施进度表

对基础设施所需的所有部件，确定由谁对它们的细节、挑选和采购负责。达成的协议必须有记录。同时建立一个进度表，指定各项基础设施可以提供给测试团队使用的时间。

6.2.8 组织管理和控制

目标

这一活动的目标是描述如何对测试过程、基础设施和测试交付物进行管理和控制。

规程

规程包含下列子活动：

- 定义测试过程控制。
- 定义基础设施控制。
- 定义测试交付物控制。
- 定义缺陷规程。

这些活动的细节在第 20 章中讲述。

6.2.9 编制测试过程进度表

目标

这一活动的目标是详细描述测试活动所需的时间、资金和人力。

规程

规程包含下列子活动：

- 建立全面的进度表。
- 建立财务进度表。

建立整体进度表

基于已制定的（时间）预算、可用资源和各种系统部件和文档的交付进度表，就可以建立测试过程的整体进度表。在测试过程中，新成员和产品都将分配给要执行的测试活动。

建立财务进度表

根据人员和基础设施的情况，安排财务进度。

6.2.10 整理测试计划

目标

这一活动的目标是记录到目前为止已经执行的活动结果，以及从委托人处获得正式的认可。

规程

规程由下列活动组成：

- 标识威胁、风险和措施。
- 建立测试计划。
- 建立测试计划的变更规程。
- 整理测试计划。

在测试计划中，有必要标识测试过程中潜在的威胁。威胁和下面几种情况相关：

- 可行性，提议的测试计划和各种提供者的进度表，在多大程度上是现实可行的。
- 可测性，为完成测试过程，测试基础的预期质量在多大程度上是足够的。
- 稳定性，在测试过程中，测试基础的变更程度有多大。
- 经验，为顺利执行测试过程，对测试团队的经验或知识水平的要求是什么。

测试计划列出了应对各种风险的措施，包括为避免风险而采取的预防性措施，也包括在早期鉴别风险的探测性措施。

建立测试计划

到目前为止，所执行的活动结果被记录在测试计划中，测试计划包含下列部

分(见附录E示例):

- 分配任务。
- 测试基础。
- 测试策略。
- 制定计划。
- 威胁、风险和措施。
- 基础设施。
- 测试组织。
- 测试交付物。
- 配置管理。
- 附录。
 - 测试计划的变更规程;
 - (时间)预算的理由。

概括策略、进度、预算、威胁、风险和措施等内容的管理概要是可选的。

建立测试计划的变更规程

对于经过批准的测试计划，建立变更规程。这个规程详细描述变更测试计划的标准和所要求的权限。

整理测试计划

测试计划提交给委托人以便获得批准。建议用测试经理和委托人共同签名的方式，来表明测试计划被正式批准。此外，对筹划指导委员会和其他相关参与者的介绍，对获得批准也有帮助，有时获得组织内部的支持更为重要。

6.2.11 维护测试计划

目标

这一活动的目标是使测试计划和所有的进度表都是最新的。

规程

当发生变更，导致测试计划依照既定标准进行调整时，就要维护测试计划。

- 重新调整测试计划和/或测试策略 测试计划的变更实际上会影响测试过程中执行的所有活动，尤其是测试策略经常受到变更的影响。改变测试策

略的一个正当理由是：有些测试在实际中会发现缺陷多于或少于预期。根据这些发现决定是否需要进行额外的测试，或者只需执行测试计划中的一部分，甚至全部取消它们。

这些变更被记录在新版的测试计划中，或者记录在附件中。同时必须将变更提交给委托人，以便获得批准。测试经理的责任是，将变更结果清晰地传达给委托人。

- **维护进度表** 最平常而又能预见到的变更是那些和进度相关的变更。变更的理由可能和测试基础的可用性、测试对象、基础设施和测试人员等有关。测试基础或待测系统的质量也可能成为调整进度的理由。

6.2.12 控制测试

目标

这一活动的目标是控制测试过程、基础设施和测试交付物，以便能够不间断地把握测试过程的进展和测试对象的质量。

规程

与测试计划中建立的规程相一致，控制测试过程、基础设施和测试交付物。

6.2.13 报告

目标

这一活动的目标是向组织提供有关测试过程的进展和待测系统的质量等信息。

规程

定期或按照要求，提交有关测试过程进展和测试对象质量的报告。在第 20 章中，测试计划列出了报告的形式和频率。进展和质量报告包含最近报告周期内的数据和整个测试过程的累计数据。其他的报告按照委托人的要求进行裁减。报告中可以包含下面一些要素：

- 在测试计划中给出的测试已经完成多少。
- 还有哪些需要测试。
- 针对测试对象的质量和发现的缺陷，能不能发现什么趋势。

6.2.14 建立详细进度表

目标

这一活动的目标是为不同阶段建立和维护详细的进度表，包括准备阶段、细化阶段、执行阶段和完成阶段。

规程

详细进度表至少应当包含每个阶段的下列几个方面：

- 执行的活动。
- 与（测试过程内部或外部）其他活动的联系和依赖性。
- 分配给每个活动的时间。
- 整个项目所需的时间和可用的时间。
- 交付的产品。
- 相关人员。

6.3 准备阶段

目标

准备阶段最重要的目标，是确定测试基础是否能够为测试规范以及测试用例的成功执行（可测性）提供足够的质量保证。

前提条件

测试基础应当是可用的并固定下来。

活动

在准备阶段主要进行的一些活动是：

1. 测试基础的可测性审查。
2. 定义测试单元。
3. 分配测试设计技术。
4. 定义基础设施。

图 6.2 描述了准备阶段各个活动之间的相关性。

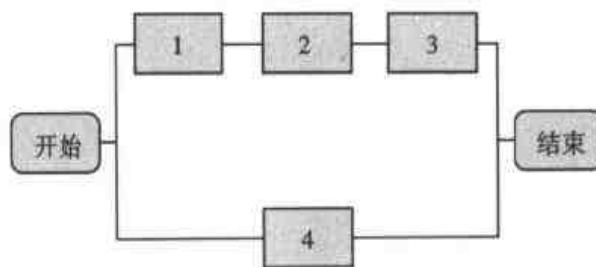


图 6.2 准备阶段的活动

6.3.1 测试基础的可测性审查

目标

可测性审查的目标是保证测试基础的可测性。可测性在这里指的是全面性、一致性、可访问性以及可转换为测试用例的特性。

规程

以下是可测性审查的简要描述，详细的描述可见第 8 章。

- **选择相关的文档。**收集测试基础并将它们列在测试计划中。如果在此期间有任何变更，则测试计划也必须调整。
- **起草审查清单。**和测试策略相关，为各个子系统准备审查清单。这些审查清单的作用是充当评估测试基础的指导。
- **评估文档。**根据审查清单来评估测试基础。如果测试基础不够充分，需要通知委托人和测试基础提供者。
- **报告。**将评估结果和建议报告给委托人和测试基础提供者。

6.3.2 定义测试单元

目标

这一活动的目标是将子系统划分成独立的可测单元。

规程

定义测试单元的规程由下列子活动组成：

- **确定测试单元。**
- **实现测试单元表。**

确定测试单元

在一个子系统中，那些逻辑一致和/或相互高度依赖的活动，被放入一个测试单元中。如果相关的活动很大，则合并后测试单元也会变大。

实现测试单元表

对每个子系统而言，需要指出它包含的测试单元。这种细分可以从测试单元表（见表 6.1）中看到。

表 6.1 测试单元表示例

系统	测试单元
子系统 A	测试单元 A1
	测试单元 A2
	测试单元 A3
	等等
	测试单元 B1
子系统 B	测试单元 B2

6.3.3 分配测试设计技术

目标

这一活动的目标是基于测试策略，为测试单元分配测试设计技术。

规程

前一活动中实现的表可以用所分配测试设计技术加以扩展（见表 6.2）。对一个测试单元，可以分配多种技术。

表 6.2 用所分配测试设计技术扩展的测试表

系统	测试单元	测试设计技术
子系统 A	测试单元 A1	分类树方法
	测试单元 A2	状态转换测试
	测试单元 A3	初步比较测试
	等等	初步比较测试
	测试单元 B1	统计使用测试
子系统 B	测试单元 B2	统计使用测试
		状态转换测试

6.3.4 定义基础设施

目标

这一活动的目标是对所需的基础设施进行详细描述。

规程

在必要的时候，需要详细描述测试计划中的测试基础设施。同时要与供应商达成协议并制定交付进度表。如果某些基础设施必须自行开发，则需要建立开发计划。

6.4 细化阶段

目标

细化阶段的目标是利用分配的测试设计技术，建立测试集。

前提条件

在细化阶段开始之前，必须满足下列条件：

- 测试基础应当是可用且固定的。
- 测试对象和基础设施的交付进度表可以满足建立测试方案的要求。

活动

在细化阶段主要进行的一些活动是：

1. 导出测试用例。
2. 起草测试脚本。
3. 建立测试方案。
4. 定义测试对象和基础设施的人口检查。
5. 安装基础设施。

图 6.3 描述了细化阶段各个活动之间的相关性。

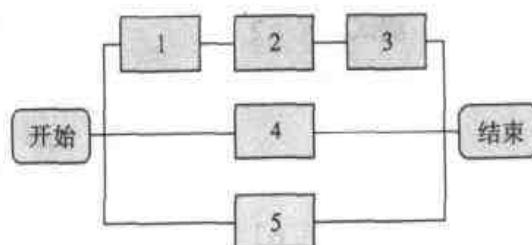


图 6.3 细化阶段的活动

6.4.1 导出测试用例

目标

这一活动的目标是基于所分配的测试设计技术，为每个测试单元导出测试用例。

规程

利用所分配的测试设计技术，导出测试用例。同时，应当确定各种测试用例能否单独执行，或者是否会对相互的结果产生影响。需要按照测试计划中描述的标准来准备测试设计。在测试设计阶段，可以发现测试基础的缺陷，这些缺陷可以通过测试计划中描述的缺陷管理规程来报告。

6.4.2 起草测试脚本

目标

这一活动的目标是将测试设计中描述的测试用例转换成可执行的、具体的测试动作。这包括建立动作顺序和测试脚本中执行的条件。

规程

测试用例被转换成可执行的、可验证的测试动作，这些动作按照正确的顺序排列。理想的情况是这些动作都是互不相干的，以避免一个失败的动作破坏一大段脚本的情况出现。

测试脚本至少应该描述前提条件和执行动作。

6.4.3 建立测试方案

目标

这一活动的目标是在一个测试方案中记录测试脚本的执行顺序。

规程

测试方案就是测试执行的路线图。测试方案描述测试脚本的执行顺序和形式，成为测试执行的控制手段。可以将测试方案扩展，把测试脚本分配给单个测试人员。应当将不同测试脚本之间的相互依赖性控制到最小。

如果可能的话，在测试执行的初始阶段就执行那些记录在测试策略中，和系统最关键部分相关的测试脚本，以检测最重要的缺陷。

测试方案必须是一份灵活的、有效的文档。有许多原因可以引起测试方案的变更，例如阻塞错误、基础设施故障、测试人员缺乏，等等。

6.4.4 定义测试对象和基础设施的人口检查

目标

这一活动的目标是描述如何执行测试对象和基础设施的人口检查。

规程

准备一份说明全部应交付给测试团队的交付物的审查清单。（测试团队）利用该审查清单来执行测试对象的人口检查，以确定测试对象是否完整以及测试能否开始。

可以用基础设施的详细说明来准备审查清单。利用这个审查清单来确定基础设施所有指定的部件是否存在以及测试能否开始。

准备一个测试前的测试脚本。执行这个脚本来判断测试对象是否足够稳定，以便开始执行测试。稳定意味着在执行测试脚本时没有发现任何缺陷。

6.4.5 安装基础设施

目标

这一活动的目标是根据规范来安装基础设施。

规程

基础设施的安装和细化阶段的其他活动同时进行。这一活动的执行通常由下列要素组成：

- 解决瓶颈和问题，记录在新协议中采取的任何措施。
- 基础设施的人口检查。
- 安装检查。
- 试运行。

6.5 执行阶段

目标

执行阶段的目标是执行指定的测试脚本，以了解测试对象的质量。

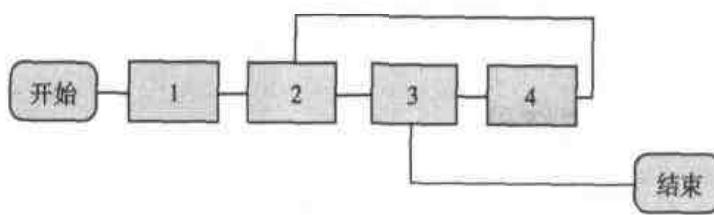
前提条件

基础设施已经安装，且测试对象已经交付给测试团队。

活动

在执行阶段需要区分下列活动（如图 6.4 所总结）：

1. 测试对象/基础设施的入口检查。
2. 执行（再）测试。
3. 比较并分析测试结果。
4. 维护测试方案。



6.5.1 测试对象/基础设施的入口检查

目标

这一活动的目标是判断所交付测试对象的部件和基础设施是否已经为测试做好了准备。

规程

如果在细化阶段没有执行基础设施的入口检查，则需要在这个时候完成它。最好是在细化阶段完成基础设施的入口检查，因为如果这个时候发现有问题，则在执行测试开始之前有充足的时间来解决这些问题。

通过预先准备好的审查清单，来检查所交付的测试对象的完整性。需要报告遗漏的项目，并决定是否继续进行对象的入口检查。

如果测试对象已经成功安装，而且基础设施也已经准备好，那么就可以执行入口检查中准备好的测试用例。在执行过程中发现的缺陷需要立即报告，并且要优先解决它们。入口检查的成功完成是开始执行测试的先决条件。换句话说，如果入口检查没有成功完成，就不能开始测试执行阶段。如果测试团队无法开始测试，开发人员应当承担责任。

6.5.2 执行(再)测试

目标

这一活动的目标是得到测试结果来评估测试对象的质量。

规程

按照测试方案中指定的顺序来执行测试脚本。在整个执行测试的过程中，纪律是非常重要的。测试应当按照方案和测试脚本中所描述的来执行。如果测试人员偏离了测试脚本，就无法保障测试策略的正确执行，其结果是无从得知风险检查是否按照计划覆盖到了。

除了测试脚本之外，审查清单也被用于执行静态测试。

6.5.3 比较与分析测试结果

目标

这一活动的目标是发现测试对象的意外行为，并分析产生这些行为的原因。

规程

将实际结果和预期结果相比较，记录比较的结果。如果实际结果和预期结果之间有差异，就需要分析这种差异。差异可能是由不同的原因引起的：

- 测试执行错误，这意味着相关的测试必须重新执行。
- 测试设计错误。
- 编程错误。
- 测试环境的缺陷。
- 测试基础中的前后矛盾或含糊不清。

只有当差异是由系统故障引起时，才能按照测试计划中描述的缺陷规程来报告这个缺陷。

6.5.4 维护测试方案

目标

这一活动的目标是使测试方案保持最新，以便在各个时候都能清楚地说明哪个测试脚本必须执行，按照什么顺序来执行。

规程

在执行(再)测试的时候,有可能发现对执行测试有影响的问题。首先是可能的测试缺陷,这就需要判断是否需要修改测试件,是否需要再次运行测试。测试的再运行包括在测试方案中,任何与测试件相关的工作都需要重新开始。

其次,缺陷几乎总会使测试方案包含重新测试。然而,重要的是需要确定该如何进行重新测试。测试脚本的全部还是部分重新执行,取决于下面的情况:

- 缺陷的严重程度。
- 缺陷的数量。
- 已经执行的测试脚本由于缺陷而受到破坏的程度。
- 可用的时间。
- 功能的重要性。

维护测试方案是极其重要的。通过它可以深入了解那些必须执行的测试脚本,同时它为需要包含在执行阶段的详细进度表,以及整个测试过程的总体进度表中的变更提供了基础。

6.6 完成阶段

目标

完成阶段的目标包含以下几个部分:

- 存档测试件,以便能复用于以后的测试中。
- 获取经验图表,以便更好地控制以后的测试过程。
- 完成最后的报告,向委托人报告测试过程,并解散测试团队。

前提条件

测试必须完成包括重新测试在内的全部过程。

活动

完成阶段由下列活动组成(见图 6.5):

1. 评估测试对象。
2. 评估测试过程。
3. 存档测试件。
4. 解散测试团队。

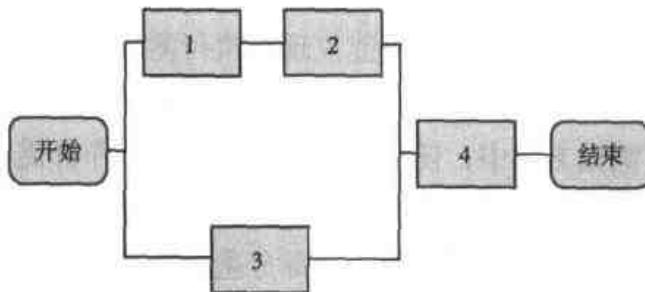


图 6.5 完成阶段的活动

6.6.1 评估测试对象

目标

这一活动的目标是评估测试对象的质量，提供最终的发布建议。

规程

基于已经完成的测试、测试报告和已登记的缺陷的状态，起草最终的发布建议。重点是指出还有哪些缺陷没有解决，与这些缺陷有关的风险是什么。这意味着还应当给出其他可供选择的办法，比如延期发布、提供较低等级的功能，等等。

6.6.2 评估测试过程

目标

这一活动的目标是了解测试过程的运行情况，为将来的测试过程收集经验数据。

规程

在执行阶段之后，需要评估测试过程，以了解执行过程中的优势和劣势。该评估可能是产品改进的基础。评估的结果将被报告，最终的报告至少需要包含下列几个方面：

- 对测试对象的评估。给出测试对象的评估结果，尤为重要的是需要列出那些尚未解决的问题以及相关风险。
- 对测试过程的评估。给出测试过程的评估结果，可以分解成下面几个部分：
 - 对测试策略的评估。
 - 与所选测试策略的偏差程度多大。

- 所选的测试策略是否恰当，系统的哪些部件测试太多，哪些部件测试太少。
- 进度表和执行情况的比较。
 - 进度表的实现情况如何。
 - 发现了结构性的偏差没有。
- 对资源、方法和技术的评估。
 - 所选资源的利用程度如何。
 - 所选方法和技术使用是否正确，这些方法和技术是否存在缺陷。

6.6.3 存档测试件

目标

这一活动的目标是选择和更新已经建好的测试件，用于以后的测试。

规程

如果有必要，需要收集和调整测试件，保存并备份测试件。如果计划发布产品的新版本，就将测试件移交给新版本的项目领导。存档测试件为测试脚本的复用提供了可能，在下次测试时，用较少的付出就可达到相同的效果。

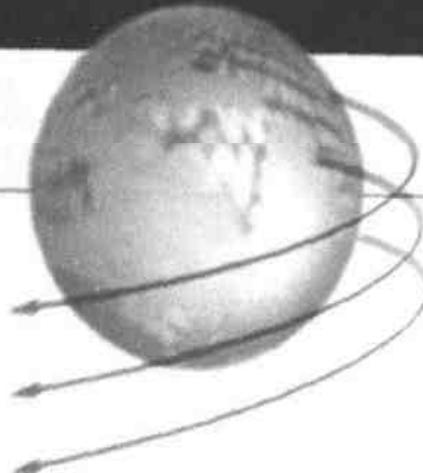
6.6.4 解散测试团队

目标

解散测试团队，表明测试过程的最终完成。

规程

在完成评估报告及移交测试件之后，委托人正式宣布结束测试过程，解散测试团队。接到命令后，测试团结就解散了。



第三部分 技术

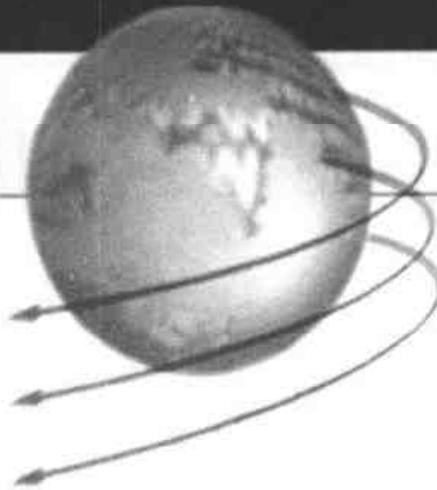
一种技术主要描述某个活动必须如何执行。许多测试者就会自然想到技术就是“测试系统”，也就是使用系统并评估系统的行为。但这仅仅是技术的一个例子。原则上对每个活动而言，都能开发出技术来支持该活动。

这一部分对各种有用的技术提供选择，以用于测试生命周期中的不同阶段。

我们从计划控制阶段运用基于风险的测试策略的技术开始（第7章）。这一技术将促使组织者决定应当测试什么，不应当测试什么。对测试经理而言，这是非常重要的技术。它能确保测试专注于“正确的事情”，以使组织者能对测试所达到的成果有现实的预期。

在准备阶段应用可测性审查技术（第8章）。可测性审查能防止大量的时间和努力被浪费在“流沙型”的测试用例设计上。该技术关注的重点是是否拥有足够的、可靠的信息作为基础来设计测试。

第9章描述的是如何组织和执行正式检查。这是一种可应用于评估和提高文档质量的通用技术。它可以作为一种可测性审查技术，也可以应用于整个开发生命周期中所有类型系统文档的审查。



第10章描述了一些应用于分析安全评价方面的技术。在部件出现故障的情况下，也可以应用这些技术来分析系统的弱点。因为安全性分析通常是单独组织的，平行于其他测试活动，在这一章里还描述了安全性分析的一个单独生命周期，以及它与测试生命周期的关系。

内容最多的第11章是关于测试设计的，也就是如何从一些系统详细设计导出测试用例。第11章提供多种多样的测试设计技术，可以在不同的环境中用于不同的目的。它提供一组有用的基础技术，这对于多数测试项目来说已经足够了。不过仍然有更多的测试设计技术被发明出来，我们鼓励读者去钻研与此相关的文献和出版物。在这一章的介绍里，将解释测试设计技术的一般法则，它们在哪些方面不同，以及它们各自要达到的目的。

最后，第12章提供了可用于整个测试生命周期的多种审查清单。

第7章 基于风险的测试策略

7.1 介绍

开发基于风险的测试策略是一种和产品权益人沟通的方法，可以探讨对于公司来说，该系统最重要的是什么，对该系统的测试而言这意味着什么。测试所有的东西从理论上来讲不可能，从经济上讲也是不可行的，否则是对稀缺资源（时间、资金、人力和基础设施）的浪费。为了最大限度地利用资源，需要决定系统的哪些部分和特性是重要的，哪些不是重要的。基于风险的测试策略是结构化测试方法中一个重要的因素，它有助于更好地管理测试过程。

必须规定测试的优先级，决定哪些是重要的，哪些不是重要的。这看起来似乎简单易懂，但有几个问题需要弄明白：

- 我们所说的“重要”指的是什么。
- 我们所评估的“重要”或“不重要”的事情指的是什么。
- 谁对此做出决定。

我们所说的“重要”指的是什么？这难道不是一个很主观的概念吗？采用基于风险的测试策略，评估重要性就是回答这个问题：“在一定范围内，如果某件事情没有准备好，那么其（商业）风险有多高？”风险的定义是：当问题发生时，预计与其相关的损害发生的几率。如果系统没有足够的质量，则可能意味着对组织的严重损害。比如可能导致市场份额的丢失或者公司可能被迫召回上百万已出售的产品，以更换存在缺陷的软件，因而这种情况会对组织构成风险。通过测试来了解系统在多大程度上满足质量要求，就有助于避免这样的风险。如果系统的某个地方或某些方面隐藏着高风险，则更为彻底的测试显然是一种解决办法。反过来也是如此：没有风险，就没有测试。

我们所评估的“重要”或“不重要”的事情指的是什么？在测试策略中，需要评估两类事情的相对重要性：子系统和质量特性。基本上是采用“信号解码子系统比进度子系统更重要”，“可用性比性能更重要”等类似的语句。子系统可以是系统的任何部分，只须从策略决定的角度看来便于独立对待即可。系统的架构分解通常就用于这个目的。质量特性描述所需的系统行为类型，比如功能、可靠性和可用性，等等。已经存在定义质量特性的标准，比如 ISO 9126（见 www.iso.ch）。

谁对此做出决定？建立基于风险的策略的任务就好像外交使命一样，它涉及到许多不同的人或部门，这些人或部门都代表不同的利益：销售部、维修人员、开发人员、最终用户、等等。对于什么是重要的，他们都有自己的观点。如果不倾听他们的意见，就会使测试项目变得困难，他们会说“我不同意，我会反对它”。在做决定的过程中应该让这些人们参与进来，更多人的参与能够减少项目后期出现麻烦的风险。但这并不意味着完美的测试策略就是满足每一个人愿望的测试策略，这可能意味着每件可能的事情都会有人觉得重要，因而最终每件事情都需要测试。最佳测试策略的一个较好定义是：测试策略不能使每个人都完全满意，而是在所有的事情都考虑到后，每个人都同意这个策略是最佳的折中方案。

建立测试策略确实要涉及到大量的妥协，并做出可接受的选择，但是有些事情是不能妥协的。在开发和测试过程中，公司可能强制实现某些标准。有些行业要求遵守产品认证的工业标准，比如航空电子工业的安全评估软件需要遵守 DO-178B 标准 (RTCA/DO-178B, 1992)。许多标准能用来充当部分测试策略 (Reid, 2001)。比如软件验证与确认标准 IEEE 1012 就和软件完整性等级的测试活动有关，定义完整性等级的过程在 ISO 15026 标准中定义。其他有用的标准有定义多种测试文档的标准 IEEE 829、软件测试词汇表标准 BS7925-1 以及定义和解释一些测试设计技术的标准 BS7925-2。

7.2 风险评估

开发一个测试策略需要了解风险。当验证模块不能正确工作时，其负面后果将是什么？当系统的性能不足时，会有什么损害？不幸的是，风险评估在行业里还不是一门科学。风险不能被计算和表达成一个绝对的数字。不过，这一部分仍将帮助读者获得对风险的必要认识，以开发合理的测试策略。

为了开发测试策略，需要基于质量特征和子系统来评估风险。为此，利用下面的著名方程式 (Reynolds, 1996) 来分析风险的各个方面：

$$\text{风险} = \text{失败几率} \times \text{受到的损失}$$

这里失败几率与系统的使用频率和单个故障几率有关系。如果某个部件在一天里被许多人使用了许多次，则单个故障几率就大大增加。下面所列的项目有助于估计故障几率，它给出了容易发生故障的位置 (Schaefer, 1996)：

- 复杂部件；
- 全新部件；
- 经常改动的部件；

- 首次采用某种工具或技术的部件；
- 在开发过程中从一个开发人员移交给另一个开发人员的部件；
- 在时间极其紧迫的情况下构建的部件；
- 超过优化频率平均值而频繁优化的部件；
- 在早期发现过许多缺陷的部件（比如在前一个版本或在早期的审查中）；
- 有许多接口的部件。

下面所列各项也会导致发生故障的几率比较高：

- 无经验的开发人员；
- 用户代表参与不足；
- 开发过程中缺少质量保障；
- 质量不高的低层次测试；
- 新的开发工具和开发环境；
- 大型开发团队；
- 沟通不畅的开发团队（比如由于距离遥远或个人原因）；
- 在组织内存在尚未解决的冲突的情况下，迫于行政压力而开发的部件。

接下来需要估计可能的损失。当系统出现问题时（没有满足其质量要求），给组织带来的损失是什么？损失形式可能是多种多样的，例如修理费用、由于负面新闻报道而带来的市场份额丢失、用户的合法要求（索赔）、收入减少等。应当尽量将每一种形式的损失都转换成现金，这样更容易将损失用单一的数字来表示，更容易与其他被评估的相关风险进行比较。

进行风险评估所需要的信息通常是从不同的来源获得的。那些了解产品在其环境中如何使用的人，知道产品的使用频率和可能的损失。这些人包括最终用户、支持工程师和产品经理等。项目团队成员，比如架构设计师、程序员、测试人员和质量保证人员，最了解产品开发过程中的难点，他们可以提供评估故障几率的信息。

由于事情的复杂性，不可能完全客观和精确地评估风险。它是一个对可感知风险的相对顺序排列的全面评估。因此重要的是：进行风险评估不仅仅是测试经理的事情，也是其他许多和项目相关的团队和产品权益人的事情。风险评估不仅能提高测试策略的质量，而且使相关的每个人都能对风险有更好的认识，并知道什么测试能够处理这些风险。在“关于测试要树立正确的预期”方面，风险评估尤为重要。必须知道测试只不过是管理风险的方法之一。图 7.1 给出了风险管理的多种方法（Reynolds, 1996）。组织必须决定想要通过采取哪些措施（包括测试）来避免哪些风险，以及他们愿意为此付出的代价。

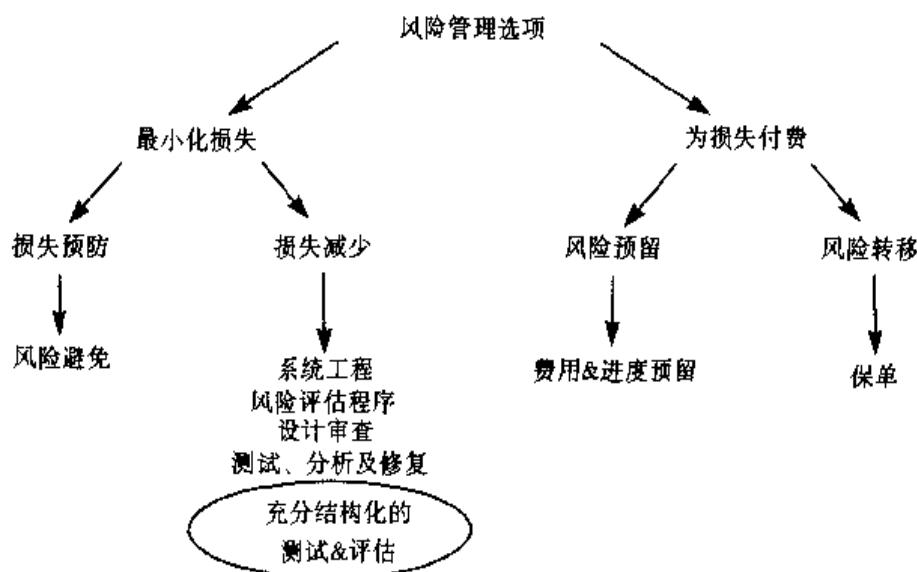


图 7.1 风险的处理

7.3 主测试计划中的策略

在主测试计划中，测试策略的目标是：使组织内的成员对必须避免的风险获得认识，以及约定在开发过程中，在何时何地需要执行多少测试。必须按下列步骤来制定测试策略：

- 选择质量特性；
- 确定质量特性的相对重要性；
- 为测试层次分配质量特性。

7.3.1 选择质量特性

在与各方的协调过程中，选择那些测试必须集中的质量特性。这里主要考虑的是商业风险（见 7.2 节）、组织、国际标准及规则。所选择的质量特性必须通过即将进行的测试得到处理。预计通过测试将报告如下质量特性：系统在多大程度上能够满足质量要求，不符合这些质量要求的风险是什么。

在开始时，必须有一套能够覆盖所有可能的质量要求的质量特性。国际标准，比如 ISO 9126，为组织制定自己的、适合于特定文化和所开发系统种类的质量特性定义提供了有用的参考。因为许多不同的人将讨论特定质量特性的重要性和实际操作方法，因此有必要让每个人都“谈论同一件事情”。对每个质量特性来说，可以通过对下列主题的讨论而提出，并将结果归档来进行选择：

- 详细说明质量特性 比如对“性能”而言，是指“反应有多快”还是“它

能够处理的负载是多少”？如果有两个因素相关，则为每个因素都定义单独的质量特性，以防止概念的混淆。

- **提供典型的例子** 质量特性的正式定义，尤其是从国际标准拷贝来的定义，对于开发测试策略的人来说，具有太浓的学术味道。因此用一个例子来描述在一个特定系统中某个质量特性的含义是什么，将有助于对质量特性的理解；
- **定义衡量质量要求的方法** 在测试过程中，必须分析系统在多大程度上满足质量要求。那么如何来进行分析呢？必须花些时间用于调查如何建立与特定质量特性相匹配的系统行为。如果没有定义衡量质量要求的方法，就不要指望测试组织能够提交任何有意义的、可信赖的报告。

7.3.2 确定质量特性的相对重要性

在选择了一组质量特性之后，必须研究每个质量特性相对于其他特性的重要性。同样，对风险的评估是决定系统中某个质量特性重要性的基础。研究结果可以用一个矩阵描述，每一个质量特性的重要性用百分比表示。表 7.1 是这种矩阵的一个例子，在这个例子里使用了一组组织特定的定义。

表 7.1 质量特性的相对重要性矩阵

质量特性	相对重要性 (%)
连接性	10
效率（内存）	-
功能	40
可维护性	-
性能	15
可恢复性	5
可靠性	10
安全性	-
适用性	20
可用性	-
总计	100

这种矩阵的目的是提供一份总体、快速的概览。那些百分比高的质量特性在测试过程中将得到特别关注，如果时间允许的话，则可以对那些低百分比的质量特性稍加注意。采用这样一个矩阵的好处是，可以迫使组织里的产品权益人放弃那些含糊不清的、泛泛的想法，从而在具体的条款中确定哪些是重要的。

就人类的品性而言，有一种自然倾向，即对质量持有谨慎的看法。多数人不喜欢说某件事情是“不重要的”，这就可能导致矩阵里充斥着许多低百分比的项，从而使矩阵里的主体变得模糊。因此必须规定百分比的最小值不得少于 5%。当某个质量特性的相对重要性是 0% 时，虽然这并不一定意味着它毫不相干，但将稀缺资源花在这种质量特性上不是明智的做法。

7.3.3 为测试层次分配质量特性

前一部分确定了系统中哪些质量特性是最重要的，必须在测试过程中予以重点关注。现在，整个测试过程由许多测试活动组成，这些测试活动由不同的测试者和团队在项目的不同时期、不同环境中执行（见 4.1.2 节）。为了使稀缺资源的分配最优化，需要确定哪些测试层次必须覆盖到所选择的质量特性中，以及大致的覆盖方法。其结果是要简明扼要地指出：为了覆盖哪些质量特性，将在哪些测试层次中执行什么样的测试活动。

这可以用一个矩阵来表示，其中用测试层次作为行，质量特性作为列。每一个交叉点的符号（++、+ 或空白）都表示质量特性在测试层次中的覆盖程度。符号意义为：

- “++” 该测试层次将完全覆盖质量特性。它是该测试层次的主要目标；
- “+” 该测试层次将覆盖一部分质量特性；
- [空] 该测试层次与该质量特性无关。

表 7.2 给出了这样一个策略矩阵的例子。在这个例子中，功能是最重要的质量特性（通常都是这样），必须在单元测试和系统测试中进行全面测试。HW/SW（Hardware/Software，硬件/软件）集成测试将依赖于功能，只是进行一些简单的功能测试。HW/SW 集成测试更关注的是测试部件之间、部件与环境之间在技术上是如何交互的（连通性），以及测试当某个部件出现故障时，系统是如何恢复的（可还原性）。

表 7.2 主测试计划中策略矩阵的例子

	功能	连接线	可靠性	可恢复性	性能	适用性
	40	10	10	5	15	20
单元测试	++			+		
SW 集成测试	+	++				
HW/SW 集成测试	+	++		++		
系统测试	++		+		+	
验收测试	+			++		++
实地测试			++		++	

这个矩阵的另一个用处是能提供一个简明的概览。如果有必要，更详细的信息可以用文本形式提供，比如必要的特定工具或技术的使用，可以采用给矩阵加脚注的形式。

7.4 测试层次中的策略

对特定的测试层次，测试策略的目标是对测试什么，如何彻底地测试，采用哪些测试技术做出选择。对所有相关的部分，测试策略必须向各方做出解释，为什么考虑到时间压力和稀缺资源，采用这种方式来测试系统是可能的最佳选择。选择不是随意做出的，而是和组织认为的最重要的（依据商业风险）事情相关。正确的观点是，测试策略的开发不仅仅是一项技术工作，而且也非常受行政因素的影响：它必须向产品权益人保证测试本身并不是目标，而是为了整个组织的利益。同时要让他们知道，他们得到的并不是“最完美的测试”，而是“考虑各种环境因素后的最佳测试”。

测试策略的最终结果是，详细阐述应用于系统特定部分的特定测试技术。每种测试技术可以被看做一个具体的测试活动，可以单独计划和监督。这使得测试策略成为测试经理一个重要的管理工具。

该环节需要进行下面的步骤：

1. 选择质量特性；
2. 确定质量特性的相对重要性；
3. 将系统分解成子系统；
4. 确定子系统的相对重要性；
5. 对每个子系统/质量特性联合体，确定测试的重要性；
6. 确定将被使用的测试技术。

如果已经制定了包含测试策略的主测试计划，则必须基于主测试计划来为某个特定测试层次制定测试策略。这种情况下，步骤 1 和步骤 2 相对比较容易实现，只要将整体上已经研究和决定的内容简单转换就可以了。可以将不同测试层次的测试策略看做是对主测试策略中整体目标更为具体的细化。如果主测试计划中没有包含测试策略，则特定测试层次的测试经理将不得不在组织中发起和协调关于商业风险和质量特性的讨论，就像为主测试计划所做的一样。

7.4.1 选择质量特性

确定一组相关的质量特性，采用的指导原则与主测试计划中策略开发时相应的步骤相同。

7.4.2 确定质量特性的相对重要性

需要确定每个质量特性相对于其他质量特性的重要性。其结果可以用矩阵表示，每个质量特性的重要性用百分比表示。和前面一样，采用的指导原则与主测试计划中策略开发时相应的步骤相同。

表 7.3 是质量特性相对重要性矩阵的一个例子，它基于主测试计划的结果（见表 7.2）。表中的数字和主测试计划中的数字有所不同，因为并不是所有的质量特性都需要在这个测试层次中进行测试，而数字之和仍然必须是 100%。而且，即便两个质量特性在主测试计划中有同样的相对重要性，在一个特定测试层次的相对重要性也没有必要相同。

表 7.3 特定测试层次中质量特性的相对重要性矩阵

质量特性	相对重要性 (%)
功能	40
性能	25
可靠性	10
适用性	25
总计	100

7.4.3 将系统分解成子系统

将系统分解成能单独测试的“子系统”。尽管这里和本章中其他地方采用的是术语“子系统”，但也可以理解成“部件”、“功能单元”或其他术语。只要系统的两个部分质量要求不同，就可以将它们分解成两个子系统。而且，不同的子系统对组织产生的风险也会有所不同。

通常按照体系架构设计将系统分解成子系统，在体系架构设计中已经指定了系统部件及各部件的关系。在测试策略中偏离这个原则应当有明确的动机。其他分解子系统的例子有：基于风险大小或开发人员发布版本的顺序。例如，如果一个数据转换是新产品实现的一部分，则可以将这个转换模块视为一个单独子系统。

有时候将整个系统加入到子系统列表中。这意味着一些质量要求只有通过观察整个系统的行为才能评估。

7.4.4 确定子系统的相对重要性

按照 7.4.2 节中确定质量特性相对重要性的方法，确定出单个子系统的相对重要性。确定每个子系统的风险权值，其结果用一个矩阵来表示（见表 7.4 中的例子）。在该矩阵中并不是精确的百分比，而是对不同子系统重要性的一个大致估计。

在这个矩阵中，不应当表达测试经理个人的观点，而是那些知道产品是如何在其环境中使用的人员（比如最终用户和产品经理）的观点。这一步骤甚至能使这些人就系统中哪些是重要子系统的问题上持相同看法。

表 7.4 子系统的相对重要性矩阵

子系统	相对重要性 (%)
A 部分	30
B 部分	10
C 部分	30
D 部分	5
整个系统	25
总计	100

7.4.5 确定每个子系统/质量特性联合体的测试重要性

这一步骤是通过对质量特性和子系统的组合评估来改进测试策略。例如，性能是一个重要的质量特性（比例为 25%），但这主要是针对子系统 B（比如路由图像数据）而言，对子系统 D（比如记录统计数据的日志）而言则没有任何关系。这种信息有助于更准确地定位测试应当关注的区域。

这一改进步骤的结果可以用一个矩阵来表示，其中质量特性作为行，子系统作为列。每个交叉点能全面体现对某个子系统而言该质量特性的重要性。例如使用下面的符号：

“++” 质量特性对该子系统起主导作用；

“+” 质量特性与该子系统相关；

[空] 质量特性与该子系统无关。

表 7.5 给出了这种矩阵的例子。再次强调：测试策略的开发不是数学练习。它的目的是协助选择应当将主要工作放在哪里，以及应当忽略或少放精力的地方。像表 7.5 这样的矩阵的作用是提供如下简明信息：组织认为哪些是重要的，以及应当将测试重点放在哪里。它反映的是组织在分析风险、权衡结果和分配资源的复杂过程中做出的选择。

表 7.5 每个子系统/质量特性联合体的相对重要性矩阵

相对重要性 (%)	A 部分	B 部分	C 部分	D 部分	整个系统
100	30	10	30	5	25
功能性	40	++	+	+	+

(续表)

相对重要性 (%)	A 部分	B 部分	C 部分	D 部分	整个系统
性能	25	+	++	+	+
可靠性	10		+		++
适用性	25	+		+	++

7.4.6 确定要使用的测试技术

最后一步是选择测试技术，这些技术将用于测试子系统相关的测试特性。从前面步骤得到的矩阵（见表 7.5 中的例子）可以看做是一个“合同”：按照组织已经定义好的重要事情，测试团队要组织和执行测试过程。依据矩阵中所指定的，对某个方面进行更为严格的测试，而对另一个方面进行更全面的测试。

通常，多数系统功能通过用例来进行测试，这些用例专门为目标（子系统或功能）而设计（动态直接测试），有许多测试设计技术可以采用（见第 13 章）。在开发和测试过程中，也可以通过收集数据来进行测试（动态间接测试），或者基于一个审查清单来评估以进行测试（静态测试）。例如在测试功能时，可以用秒表来测量响应时间以测试性能。这里没有设计直接的测试用例来测试性能，而是间接地测试。又比如对于安全性，通过对安全性规章的静态检查来进行测试。

现在测试经理的职责是建立一套能够胜任工作的测试技术。矩阵中表明有高度重要性，就需要采用严格技术或者使用多项技术来实现高百分比覆盖。次重要性就采用实现低百分比覆盖的技术，或者是动态间接测试。至于哪些技术是最佳选择，则依赖于许多因素。这些因素包括：

- 被测试的质量特性。例如某项技术也许能够很好地模拟系统真实情况，但对于功能行为的变化却无能为力。
- 应用的领域。有些技术是用来测试人机交互的，而有些技术则适合测试特定部件内部的所有功能性变化。
- 需要的测试基础。有些技术要求有特定类型的系统文档，比如状态转换图、伪代码或图表。
- 需要的资源。技术的应用需要一定数量的人员和机器能力方面的资源。这一点通常也依赖于“需要的知识和技能”。
- 需要的知识和技能。测试成员的知识和技能影响着技术的选择。技术的有效使用有时要求测试人员具备特定的知识和/或技能。资深的行业专家可能是首先要具备的，有些技术甚至需要有受过训练的（数学）分析天才。

在介绍测试设计技术的第 13 章中，将有这些方面更为详细的信息。在选择所

需要的测试技术时，测试经理对于各种测试设计技术及其优缺点的经验和知识，将是至关重要的。为了适用于特定的测试项目，常常将现成的测试技术调整或组合成“新”的测试技术。技术的选择和调整必须在测试过程的早期就完成，以便对测试团队进行相关领域的培训。

这一步骤的结果是确定测试每个子系统所用到的技术。见表 7.6 中的例子。每个“+”号表示相应的技术将用于系统的特定部分。从管理角度来讲，这是一个非常有用的表，因为可把每个“+”号看成一个单独的实体，独立地进行计划和监督。可以把每个“+”号分配给一个测试人员，他将对所有必要的活动负责，比如设计测试用例、执行测试等。对大型系统而言，可以通过将其分解为单个测试人员能完全控制的更小单元的方法，来将该表进一步细化。一般只有到准备阶段才会将测试策略分解得这么详细，尤其在大型测试项目中。

表 7.6 确定哪些技术用于系统的哪些部分

采用的测试技术	A 部分	B 部分	C 部分	D 部分	整个系统
技术 W	+	+	+		
技术 X		+	+		
技术 Y	+			+	+
技术 Z					+

为了确保尽可能早地执行那些最为重要的测试，在该步骤中还需要确定测试（技术-子系统联合体）的优先级顺序。

7.5 测试过程中的策略变更

测试经理不要心存幻想，认为测试策略一旦确定，就会在整个项目的进程中一成不变。测试过程是持续不断变化的，因此也必须能对这些变化做出适当的反应。

现实中，开发和测试项目经常会面对压力，尤其是在项目的后期。突然之间调整了进度表（通常是可用的时间被缩短），测试经理被要求进行更少或更短时间的测试。哪些测试可以被取消或弱化？利用测试策略作为基础，测试经理就可以和产品权益人以专业方式来讨论这些问题。当变化的环境要求测试应当和以前达成一致的情况有所不同时，对测试应当达到的期望值也必须有所变化。策略问题也必须被重新评估：以前我们认为更重要的方面是否已经发生了变化？当减少某个特定领域的测试时，我们是否愿意接受增加的风险？

当产品发布的内容发生变化时，也会出现同样的情况。例如系统增加了额外功能，或者是系统某些部分的发布被延迟，或者是产品现在被定位于另一个有着

不同要求和预期的市场领域。在这些情况下，必须重新评估测试策略：以前计划的测试面对新情况是否仍然适合？如果质量特性或子系统的相对重要性发生了明显改变，则计划好的测试也必须做相应调整。

改变测试策略的另一个理由是测试本身的结果。当对系统某一部分的测试发现过多的缺陷时，明智的做法是加强对该部分的测试。比如增加额外的测试用例或采用更为彻底的测试技术。反过来也是如此：当发现没有或者只有极少的缺陷时，就要研究是否可以减少测试。在此情况下，需要探讨的话题是：“通过测试，现在我们感知到的风险是什么。这能够证明我们增加或减少测试的决定是正确的吗？”

7.6 维护测试的策略

上面描述的开发测试策略的技术，能够直接用于新开发的系统。一个很自然的问题是：对于一个以前已经开发完毕或测试过的系统，上面所描述的步骤在多大范围内仍然可以用于该系统的版本维护？

从测试策略的观点来看，版本维护主要的不同点在于故障几率。在维护的过程中，存在由于系统发生变化而引入故障的风险。那些对系统行为的有意改变当然需要测试。但也有可能需要测试那些系统中没有变化的功能，它们在以前的版本中表现正常，而在新版本中由于其他变化的副作用而出现问题。这种现象称为回归。在版本维护中，绝大部分的测试工作是测试以前的功能是否仍然工作正常，这被称为回归测试。因为一旦产品进入维护阶段，故障几率会发生改变，所以相关的风险也会发生改变。接下来将改变子系统的相对重要性：例如当开发一个全新的子系统时，由于其商业要求高，因而其重要性就高。在版本维护中，这个子系统没有发生改变，因此相关的风险就低，就可以将该子系统的重要性评定为低。

因此在开发测试策略时，通常将“子系统”的概念用“变更”来代替是有用的。这些变更通常是一组将实现的“变更需求”和一组需要解决的“已知缺陷”，要对每个变更需要对系统的哪些部分做修改，哪些部分会间接受到影响，哪些质量特性与之相关等进行分析。根据风险和预期进行的彻底测试，存在多种可能来测试每个变更：

- 仅仅关注变更本身的有限测试；
- 对变更的功能或部件的全面（再）测试；
- 对变更的部件与其邻近部件的一致性和交互性的测试。

实现每一个变更，都会引起回归风险。回归测试是维护测试项目中的标准元素。通常要维护一组专门的测试用例来进行回归测试。根据风险和可用的测试预

算，需要在执行完整的回归测试，还是测试那些最为相关的测试用例之间做出选择。测试工具可以非常有效地支持回归测试。当回归测试的绝大部分能够自动执行时，选择放弃哪些回归测试用例就不再必要了，因为无需多大努力就能够执行完整的回归测试。

决定按照变更需求而不是子系统来规划测试策略，在很大程度上依赖于变更的数量。当只有相对很少的变更时，通过将这些变更作为风险评估、计划和进度跟踪的基础，就能够更好地管理测试过程。当子系统由于实现许多变更而进行重大修改时，可以采用其最初开发时所用的方法来进行处理。通常人们更喜欢基于子系统来开发测试策略。如果一旦决定按照变更需求来规划测试策略，则下面的步骤就可用来开发测试策略：

1. 确定变更（要实现的变更需求和要解决的问题）；
2. 确定变更和回归的相对重要性；
3. 选择质量特性；
4. 确定质量特性的相对重要性；
5. 确定每个变更（回归）/质量特性联合体的相对重要性；
6. 确定可用的测试技术。

表7.7是从前两个步骤得出的一个矩阵例子。其他步骤和7.4节中描述的步骤类似。

表 7.7 变更和回归的相对重要性矩阵

变更/回归	相对重要性 (%)
变更需求 CR-12	15
变更需求 CR-16	10
变更需求 CR-17	10
缺陷 1226, 1227, 1230	5
缺陷 1242	15
缺陷 1243	5
...	30
回归	10
总计	100

这个例子中回归被赋值 10%，需要指出的是这个 10% 反映的是分配给回归的重要性，但并不意味着测试过程的 10% 就应当是回归测试。实际上，所有回归测试工作所占的比例要比这个数字大得多。原因是通常系统只有相对一小部分功能会发生变化，而绝大部分功能保持不变。在本段的前面就已经指出，对维护测试而言，回归测试自动化可以为释放稀缺时间和资源提供极大帮助。

第8章 可测性审查

8.1 介绍

在制定计划阶段，测试团队将决定测试基础由哪些文档组成。准备阶段的主要工作是测试基础的可测性审查。可测性意味着组成测试基础的文档的完备性、确定性和一致性。在制定测试规范的过程中，高可测性是测试成功的首要条件。可测性审查的目的是确定文档质量是否足以作为测试的基础。

组成测试基础的文档是设计过程中最先的交付物，它第一个被交付，因而测试就早。缺陷发现得越早，解决起来就越容易，成本也就越低。在文档中没有被发现和解决的缺陷，将在以后的开发过程中导致重大的质量问题。其后果是为了解决这些问题，需要花费大量的时间和代价高昂的重复工作，而且使得发布日期受到威胁。

8.2 规程

可测性审查包含下列步骤：

- 选择相关文档；
- 生成审查清单；
- 评估文档；
- 报告结果。

8.2.1 选择相关文档

测试计划应当标识用于导出测试用例的文档。然而测试计划的准备和准备阶段的工作会有一部分重迭。而且，即使测试计划已经完成，也可能有变更发生。因此可测性审查应当从对测试基础的正式标识和文档的真正收集开始。

8.2.2 生成审查清单

应当采用可测性审查的审查清单，这个审查清单依赖于所使用的测试设计技

术。测试计划应当提供关于所使用测试设计技术的信息，以及这些技术将应用于系统哪些部分的信息。

第12章提供了用于各种测试设计技术的可测性审查清单及其他审查清单。这些审查清单可以组合成一个表，以防止测试基础的相同部分被多次审查。可能有必要为每个组织、每个项目和每个测试类型各组合一个新的审查清单。

8.2.3 评估文档

利用组合的审查清单，测试团队评估文档，为每个发现的缺陷生成一个缺陷报告。早期发现的缺陷为提高测试基础质量创造了机会，甚至是在系统开始开发之前。它也能够使人们更透彻地理解系统的特性和规模，从而可以对测试计划进行调整。

8.2.4 报告结果

基于所发现的缺陷，测试团队将提交一份可测性审查报告。这个报告全面总结文档的质量，同时也应当描述质量不高的部分可能导致的后果。可测性审查报告中主要包含下列部分：

- 规划任务分配。标识测试基础，描述委托人和承包人。承包人是指负责执行任务分配的人。测试经理可能是承包人。
- 结论。根据所审查文档的可测性、相关的后果和/或风险做出结论：测试基础是否具有足够的质量确保所设计的测试是可用的。
- 建议。对当前文档提出建议，提出任何能够提高未来文档质量的建设性建议。
- 缺陷。描述发现的缺陷，给出相应缺陷报告的参考资料。
- 附录。用到的审查清单。

8.2.5 深入探讨

可测性审查不应当使得测试团队认为不可能对系统进行测试。对测试基础质量把关不严，其后果是没有足够的信息来选取所要求的测试设计技术。建议依赖于风险，这些风险涉及系统与文档相关的部分。如果风险较低，则建议可以采用不太正式的测试设计技术。高风险就自然意味着必须提高文档的质量，也就有必要重写文档。如果几乎没有任何文档组成测试基础，该怎么办呢？以下部分将给出答案。

8.2.6 不完美的测试基础

有时候需求尚未明朗。缺乏需求的原因之一就是对新市场进行创新性思维，在这种类型的项目中，需求在产品的开发过程中不断得到发展。缺乏需求的另一个原因是，在过去，系统大大小小部分的开发和测试都由同一个人完成。而现在，嵌入式系统的实现是团队工作，对需求的描述是必要的，以便管理项目，并将正确的信息传达给测试团队。

在缺乏需求的情况下，进行可测性审查就是浪费时间。准备阶段只不过是收集正确信息的一段时间。测试策略可以将与子系统和测试设计技术相关的风险告诉测试团队，这种信息可用来作为收集测试基础的正确信息的准则。可以通过与开发人员和产品权益人面对面的访谈来收集这些信息，但也可通过成为设计过程的一员来收集。

第9章 评审

9.1 介绍

评审是一种正式的评估技术，由一个不是作者的人员或组织详细考查软件需求、设计或编码，以便发现缺陷、违反开发标准的情况和其他问题。Fagan 被许多人认为是这种技术的创始人，这方面的书籍已经出版了一些（例如 Fagan, 1986）。评审已经被证明是一种发现缺陷的非常合算的方法。

评审的目的是：

- 验证软件是否符合规范。
- 验证软件是否达到应用标准。
- 对产品质量和过程质量，建立附带的和结构化的改进方法。

评审过程中找到的缺陷，应当和其他缺陷一样，根据其严重性进行修改。

评审早在动态测试之前就可开始。只有在需求明确之后才能进行测试设计。可以对这些需求进行检查，验证它们是否已达到用于开发和设计测试的预期质量。

准备阶段是评审的最重要阶段。在这个阶段，审核员将评估规范。每个审核员被赋予一个或多个角色，因此审核员能够在自己的范围内评估规范。每个审核员要发现其他的审核员不能发现的独特缺陷，因为他们有着不同的角色。

召集原因分析会议可以提升评审的价值。在会议上，应该确定所发现缺陷的原因，还应当防止在未来犯同样的错误。

重要的是，组织和领导检查的那个人必须有某种程度的独立性。为达到高度的有效性和高效率，这个人（主持人）应当具备评审技术方面的渊博知识。

9.1.1 优点

对开发产品进行评审有许多优点：

- 早期发现的缺陷，解决起来成本相对较低。
- 由于评审团队的每个成员都有各自评审的重点，因此发现缺陷的比例会比较高。
- 通过团队对产品进行评估，额外的好处是团队成员之间可以相互交换信息。

- 评审并不只是针对设计文档，还可以是开发过程和测试过程所有交付的文档。
- 评审能够激励对于开发高质量产品的认识和动力。

对提高产品质量而言，评审是一种非常适合的技术。它首先应用于评估产品本身。从评审过程到开发过程获得的反馈，将使过程改进成为可能。

9.2 规程

检查包含下列步骤：

1. 入口检查；
2. 组织评审；
3. 开始；
4. 准备；
5. 缺陷登记会议；
6. 原因分析会议；
7. 修改；
8. 后续工作；
9. 检查输出标准。

9.2.1 入口检查

主持人依据输入标准对产品执行入口检查。输入标准为产品作者提供了明确的指导，告诉他什么时候其产品被接受进行评审。输入标准应当避免对不适合产品开展不必要的工作。输入标准的例子有：

- 产品必须完成；
- 参考文档必须是被认可的文档；
- 参考文档必须是正确的和最新的；
- 由主持人最初进行的快速检查，发现的缺陷应当不多于 x 个；
- 文档必须经过拼写检查；
- 文档是符合标准的文档。

9.2.2 组织评审

如果产品通过了输入标准，主持人就要组织人员进行评审。必须组成一个团队，为每个成员分配角色。成员分配的角色必须是与其兴趣和专业密切相关。角

色的一些例子有 (Gilb & Graham, 1993):

- **用户:** 关注用户或客户的观点;
- **测试人员:** 关注可测性;
- **系统:** 关注广泛的系统问题;
- **质量:** 关注质量特性的各个方面;
- **服务:** 关注服务、维护、供应和安装。

上面给出的这些角色并不全面。而且对每一个评审而言，并不一定都需要这些角色。

所有必要的文档和标准，比如产品文档、参考文档等，都必须分发给所有的成员。必须安排准备和会议的日期。

9.2.3 开始

开始会议并不是必需的。基于下列原因可以组织开始会议：

- 当从事评审的成员没有评审技术经验时，主持人可以简要介绍评审技术，以及评审技术中各成员的角色。
- 对于复杂产品，如果产品的作者对产品进行介绍肯定非常有帮助。
- 一旦评审的规程发生了改变，可以用开始会议来通知各成员。

9.2.4 准备

准备阶段的惟一工作就是发现缺陷。每个成员有各自的角色，从该角色出发来评审产品。尽管很有可能他们只查找与他们的专业和/或角色相关的缺陷，但每个人都应当记录发现的任何缺陷。所有相关的文档都必须服务于评审。

9.2.5 缺陷登记会议

除了记录缺陷之外，缺陷登记会议的目的还有发现新的缺陷和交流知识。

主持人、各成员和产品作者参加缺陷登记会议。主持人负责列出所有的缺陷，产品作者将所有的缺陷记录在一份缺陷报告中。为避免浪费时间和无休止的讨论，不太重要的缺陷和缺陷解决方案不在会议上讨论。缺陷登记会议应当限定在两小时以内，因而只有 10 至 15 页的缺陷能够得到处理。

9.2.6 原因分析会议

在缺陷登记会议之后，紧接着的原因分析会议必须是在一种建设性的气氛中

举行。这个会议是以一种头脑风暴会议的形式来举行的。会议的重点必须是在查找缺陷的根源上，而不是缺陷本身。基于这些缺陷的根源，小小的改进建议就会对过程有实质性的改进。

9.2.7 修改

产品作者必须解决被评估产品中记录的所有缺陷。在相关文档中的所有其他缺陷，当做变更建议被记录。更新后的文档需要提交给主持人。

9.2.8 后续工作

主持人必须检查作者是否已经解决了所有的缺陷。主持人不检查缺陷是否被正确地解决。作为反馈的一种形式，将文档送给能够检查缺陷是否被正确解决的成员。

9.2.9 检查输出标准

评审的最后一步是做出产品是否满足输出标准的正式结论。下面是输出标准的一些例子：

- 修改工作必须完成。
- 新的文档符合配置管理。
- 按照规程来处理相关文档的变更请求。
- 检查报告被移交给质量管理部门。

在评审的整个过程中，将数据收集起来用于存档。这些存档文件可用来说明早期评审的情况。

第 10 章 安全性分析

10.1 介绍

安全就是在一定条件下，系统不会危及到人的生命的期望。

许多嵌入式系统被用于强调安全的情形。系统的故障将导致严重的后果，例如人员死亡、严重伤害或环境受到严重破坏。这类系统的设计和开发过程，必须有一些固有的措施，以避免出现由于系统故障而威胁到安全的情况。

处理系统安全需求的最佳方法是在设计阶段就开始监督。在本章中，将讲述两种安全性分析技术：FMEA（故障模型及后果分析）和 FTA（故障树分析）。

在安全性分析中，需要进行严格的分类（见附录 A），这些类别使得将不同风险分类更加方便。风险分类可用于规划安全策略，对不同类型的风险需要采取不同的措施。

安全性分析是对因果关系（见图 10.1）的探讨，此处的“果”总是和威胁人类生命的事情相关的。

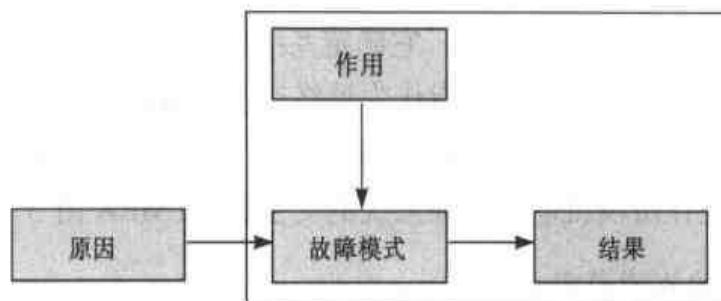


图 10.1 原因、作用、故障模式和结果之间的关系。深色部分通过 FMEA 分析，而整体通过 FTA 分析

故障原因：

- 故障（硬件和软件）——在某些运行条件下，系统的不足或缺陷可能会导致故障。
- 硬件磨损。
- 在电磁干扰、机械和化学干扰（硬件）等环境条件下。

故障模式是描述产品或过程无法执行所期望的功能的一种方法，这些功能是

内部和外部客户对其需求、想法和期望的描述。

故障是指系统或部件不能实现其运行需求。故障可能是由系统或物理上的变化引起的。

后果是由故障模式导致的不利结果。

10.2 安全性分析技术

10.2.1 概述

首先必须定义分析的范围。是应该在模块级、子系统级，还是在系统级进行分析。为了更具体地讨论某件事情应当有“多安全”，需要准备一个风险分类表（见表 A.4）。在这些准备做完之后，就可以应用 FMEA 分析技术（见 10.2.2 节）和 FTA 分析技术（见 10.2.3 节）了。在 10.2.4 节中，概述了各种不同的安全性分析方法。一般由一个小型团队来执行这些分析技术，这个团队包括安全经理、设计人员、专家和测试人员。安全经理的职责是让系统达到适当的安全级别。因为测试人员具备挑剔的态度和中间立场，因而也是这个团队的一部分。基于安全分析的结果，安全经理可能改变项目计划，而设计人员也有可能需要改变产品的设计。

10.2.2 故障模型及后果分析（FMEA）

FMEA 是一种早期的分析方法，它确定系统故障模式（错误数据或意料之外的行为）的后果——这里的系统是指作为最终产品的设备，例如软件和硬件。应该在设计过程的早期阶段就采用 FMEA 技术，因为这时易于采取行动来克服发现的问题，因而在设计阶段就能够增强系统的安全性。FMEA 由 3 个步骤组成：

- 标识潜在的故障模式；
- 确定这些潜在的故障模式对系统功能的影响；
- 制定行动来减少影响和/或故障模式。

正确使用 FMEA 技术，可以有下列结果：

- 大幅度提高系统的安全性；
- 在整个开发生命周期过程中能够跟踪风险；
- 及早确定潜在的安全危险；
- 将风险及为减少风险而采取的行动文档化；
- 将后期系统的改动和相关费用减到最少；
- 测试策略有高度可靠的输入。

通过下列方式来执行一个典型的 FMEA 分析技术：

1. 描述系统及其功能。也可能用到设计及需求文档。如果没有描述系统不同部分之间关系的文档，比如流程图或结构图，就需要编写该文档。在早期阶段，应当让主要设计者参与 FMEA 来填补这个缺陷。
2. 识别潜在的故障模式。Lutz 和 Woodhouse (1999) 已经指出有两种类型的软件故障模式：数据故障模式和事件故障模式。

数据故障模式有：

- 数据丢失（例如丢失消息，由于硬件故障而没有数据）
- 数据不正确（例如不准确的数据、假数据等）
- 数据有时限（例如旧数据、数据来得太快来不及处理等）
- 额外数据（例如数据冗余、数据溢出等）

事件故障模式有：

- 停机/异常终止（例如挂起或死锁）
- 忽略事件（例如当没有事件时继续执行）
- 错误逻辑（例如前提条件不正确，没有按预想的情形触发事件）
- 时间/顺序（例如事件在错误的时刻发生，或事件顺序不正确）

3. 对每一个被研究的功能而言，需要描述潜在的故障模式对其产生的影响。在 FMEA 的这个阶段，不讨论故障模式是否可能存在。第一步描述（故障）对所研究功能的直接影响，下一步描述对系统的影响。依据相关风险，将后果进行分类（见第 21 章）。

对系统后果的预测主要基于假设，这是因为系统还没有被完全理解，或由于设计方案尚不完整或缺乏细节。在这些情况下，讨论后果并没有多大意义。采用 FMEA 进行分析的人必须了解这一点。

4. 对严重的风险（即排在附录 A 风险级别表中前面 3 种风险类别中的风险），需要确定导致后果的原因。对每一个原因，都必须描述将采取的措施。
5. 在开发过程中，对所识别的风险进行监控。为了采取措施处理风险，必须开发能让人有信心的测试。

10.2.3 故障树分析

故障树分析 (FTA) 被用来确定故障的原因。FTA 是一种用来分析设计的安全性和可靠性方面的技术。系统的故障放在故障树的顶端，下一步是考虑系统的哪些不必要的行为是造成故障的原因。这里的系统故障是指错误事件、错误数据

或意外数据或行为。10.2.4 节将提及 FTA 分析技术的几种变体。

图 10.2 给出了用来建立故障树的符号。系统的故障被放置在故障树的顶部，第二步是确定引起故障的原因。接下来的每一步中，上一步的原因都是进一步分析的基础。这种分析将顶部故障的情况进行分解。安全要求是一个基线，它确定哪些是系统意外的或不必要的行为。必须有一个专家组来执行 FTA，图 10.3 是执行 FTA 结果的一个例子。



图 10.2 故障树分析符号

图 10.3 说明的是一个起搏器出现故障的可能原因。在顶部是不安全的情形：起搏速度太慢。这是起搏器中对人的生命构成潜在威胁的一个故障。这种故障可以由许多故障引起。在本例中，时间基准故障或关机故障都可能导致起搏速度太慢。看门狗故障和晶体失效共同导致时间基准故障。通过确定引起每个事件的原因，就生成了一个故障树。如果某个事件只能由两个或多个条件共同引起，那么这些条件就用逻辑“与”门来连接。如果这些条件中的任何一个都能引发事件，那么这些条件就用逻辑“或”门来连接。

如果系统中的危险状况只通过逻辑“或”门来连接，则系统对扰动是非常敏感的。只需一个故障，就能引发一连串的事件，从而导致不安全的情形。只要单个部件出现一个故障，就会导致多个部件失效，从而导致不安全的情形出现，那么这种故障被称为单点失效。

故障树还能用来发现相同模式的故障，这些故障影响到系统的多个部分。在故障树中，这些故障是多种故障情形的原因。

通常没有必要，也不要指望能有一个完整的顶部故障分解结构图。例如图 10.3 中，R1（看门狗故障）可当做另一个故障树中的顶部故障。通过这种方式来使用故障树，这一技术能用于复杂系统以及高层次和低层次的系统分解。

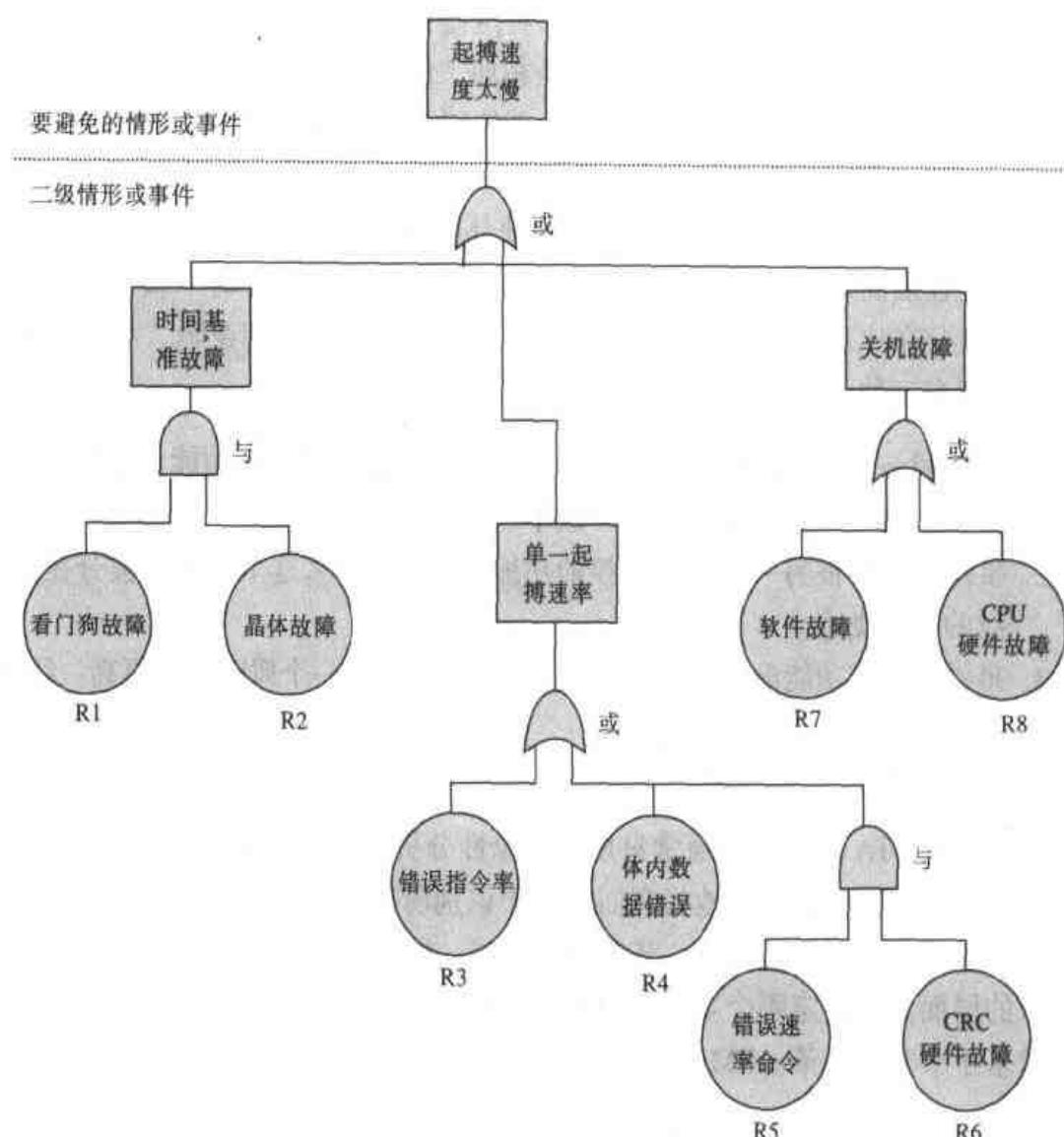


图 10.3 起搏器故障树分析的子集 (Douglass, 1999)

10.2.4 分析技术的变体

FMEA 技术每次只能采用一个故障模式，有一些类似的技术每次可以采用多个故障模式。

通过略微不同的方式，FTA 技术还可以用于确定 FMEA 结果之间的相关性。FMEA 描述的是系统级结果，如果结果被归类为三个最高级别的风险类之一（见前面例子中的说明），则必须采取安全措施。在采取措施之前，明智的做法是判断后果是否可能会发生。首先是确定故障模式是否有可能出现，如果有可能，则采用 FTA 技术来分析该故障模式的后果。如果后果有可能发生，那么就需要采取安全措施。

另一个变体是用 FTA 技术来确定发生非预期行为的概率。确定顶部事件发生

的所有可能原因及每个原因发生的概率。将这些概率组合，就能计算出顶部事件发生的概率。逻辑“或”门的概率计算是相加，而逻辑“与”门的概率计算是相乘。通过从下往上来计算故障树的概率，就能得到顶部事件发生的概率。

FTA 的第三个变体是将每个事件归类到一个风险类别中。风险类定义了开发方法、构建和被监控的方式。图 10.3 中的顶部事件是一种威胁生命的情形，这个事件被分类在最高级别的风险类中。在故障树中，每一个与逻辑“或”门相关的事件，都被分在比事件同一风险类的更高一级中。而每一个和逻辑“与”门相关的事件，则有三种可能性：

1. 事件和一个监控功能有联系，这个监控功能监视另一个功能。这样事件就被分在同一个风险类的更高一级中。
2. 事件和某个被另一个功能监控的功能有联系，那么这个功能就被分在低一级的风险类中。
3. 事件与监督功能没有联系，所以该事件被分在同一个风险类的更高一级中。

10.2.5 分析技术的其他应用

FMEA 和 FTA 分析技术通常只用于安全性分析。不过在确定测试策略的过程中它们也很有用，可以有效地将它们应用于识别弱项和有风险部分（见 7.2 节）。通常确定子系统相对重要性这一步骤不太难，即便没有 FMEA 和 FTA。可是在更细节的层面上，决定哪个功能或单元应该更彻底地测试更像是有根据的推测，而不像基于事实的决策。FMEA 和 FTA 可以有效改进该过程。

10.3 安全性分析生命周期

10.3.1 介绍

建立一个强调安全系统，就是要满足法律和认证。一些与强调安全系统相关的法规和/或要求是非常严格的。为了满足这些要求，需要有结构良好的过程，要有明确的可交付物。对于安全性而言，英国国防部已经制定了一套用于安全管理的标准，称为 MOD-00-56 (MOD, 1996a, 1996b)。该标准的一部分是描述开发和实现一个强调安全系统的结构化过程，该过程与测试过程可以共用一些产品和活动。安全处理还包括一些测试活动。如果这些共用的活动和产品不能很好地协调一致，那么两个过程都可能失败，其结果是产品不能被认证或者产品不能提供正确的功能。

安全性分析生命周期（见图 10.4）是对活动图和数据流的结构化描述。这一过程的目标，是从一些全面需求逐步得到经过认证可以被安全使用的系统。所有的设计、决策和分析结果都集中存储在一个灾害日志中。这个灾害日志是安全案例的基础，该安全案例被专门提交用于认证。

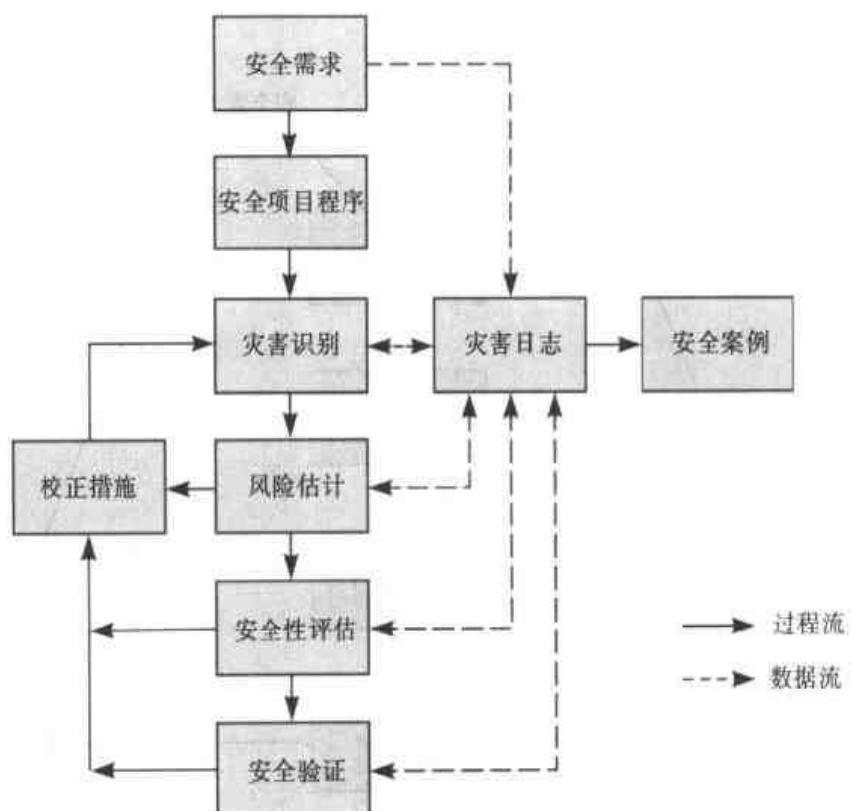


图 10.4 基于 MOD-00-56 的安全性生命周期 (MOD, 1996a, 1996b)

- **安全需求：**这些需求是用于安全验证的测试基础的一部分。制定安全要求是安全过程的第一个活动。这些要求是从法律和其他约束转换而来的，在安全过程的其他活动中不能被改变。
- **安全项目程序：**这是对时间、费用和资源进行预算的项目方案。
- **灾害识别：**灾害识别的目标是确定系统哪些部分出故障，将会导致系统工做出现严重后果。FTA 和 FMEA 是最常用的技术。
- **风险估计：**对所有已识别的灾害，分析它们对系统的影响是什么，其后果是什么。然后基于该分类来估计风险会是什么，是否必须要采取校正措施。
- **安全性评估：**这一评估的目标是确定是否采取了所有必要的措施，而且是以正确的方法来采取措施。
- **安全验证：**根据安全需求，来测试系统是否运行正常。

这一过程及所有活动的详细描述可以在 MOD-00-56 (MOD, 1996a, 1996b, 第一部分和第二部分) 找到。

10.3.2 测试基础

最终设计是“常规”测试和安全测试的测试基础的一部分。最终设计在一个迭代过程中来实现(见图 10.5)。

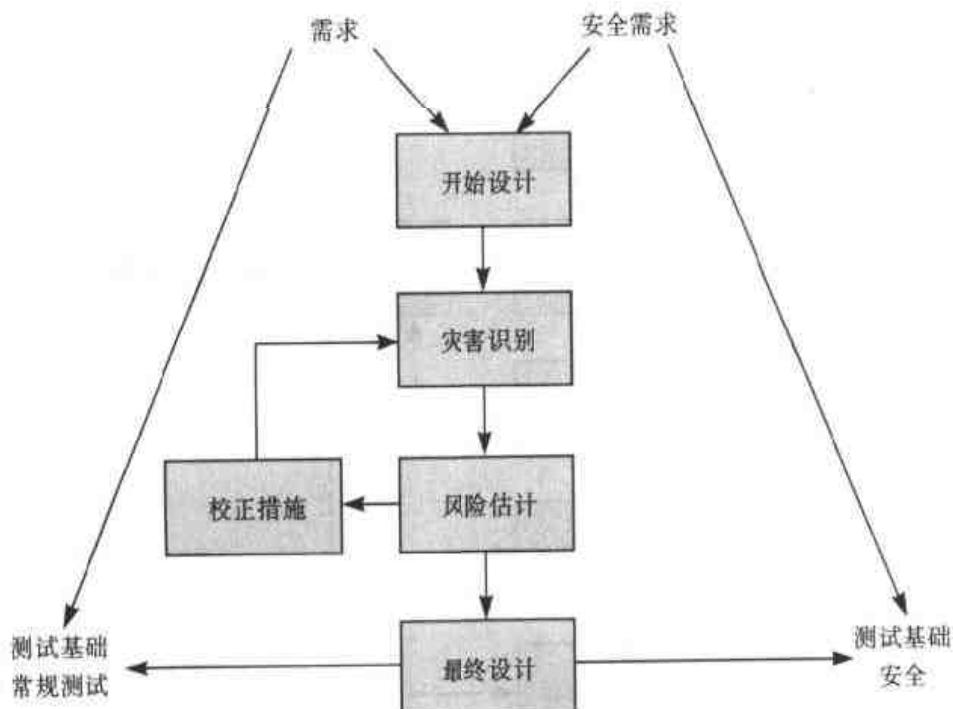


图 10.5 最终设计的实现以及与测试和安全过程的关系

这一过程从整体设计开始，基于该整体设计来执行灾害识别和风险估计。风险估计的结论必然是有必要采取校正措施，否则整体设计就需要更加详细，灾害识别和风险估计需要再次执行。循环往复这个过程，直到再也没有必要采取校正措施为止，而且准备好了详细设计。

最终设计与功能、性能、可用性等要求一起，构成了常规测试的测试基础。最终设计的提交意味着测试组可以开始测试过程的准备和执行阶段。最终设计的可靠性是在准备阶段来判断的。如果将最终设计包含在设计实现过程中，那么这一过程将会很顺利。

最终设计与安全要求构成了安全达标评估和安全估计的测试基础。这些活动可以由常规测试组来准备，或者由一个独立的测试组负责。测试计划有时候要服从审核的需要，而安全经理总是应当接受测试计划。

10.3.3 测试活动

常规测试过程和安全项目中的测试执行会产生突发事件，应当对这些事件进行分析，有时候需要加以解决。而安全缺陷的校正可能对系统功能产生影响，反过来也是如此，功能缺陷的校正可能影响到系统安全。为了克服它们之间相互影响的问题，必须集中进行影响分析并采取校正措施（见图 10.6）。

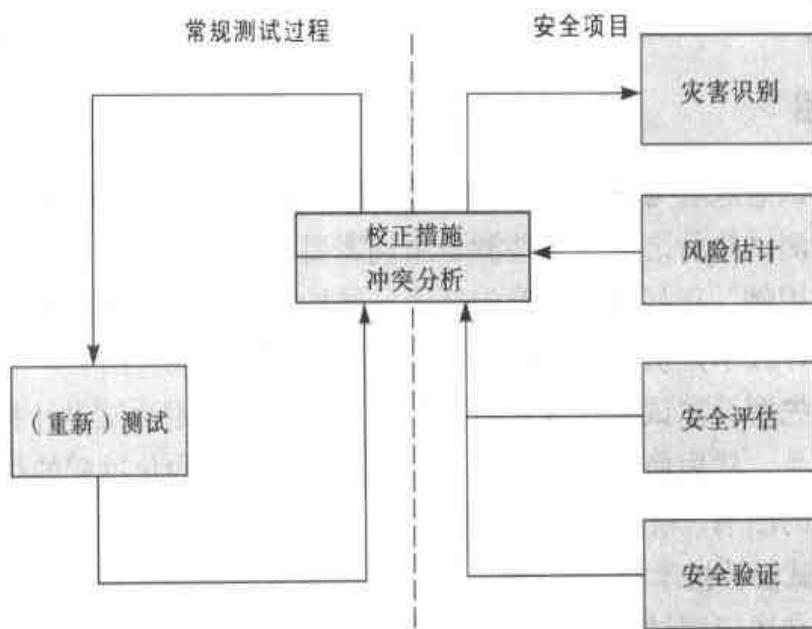


图 10.6 集中进行影响分析并采取校正措施

安全缺陷的校正措施将导致对功能重新进行测试。对与质量属性相关的缺陷进行校正，多数要进行安全验证。

第 11 章 测试设计技术

11.1 概述

11.1.1 介绍

结构化测试意味着为达到所需的测试目标，需要充分考虑采用哪些测试用例，并且在实际的测试执行之前，这些测试用例需要准备好。这和“顺着流程聪明地拼凑一个测试用例”正好相反。换句话说，结构化测试就意味着测试设计技术的应用。测试设计技术是从参考信息中导出测试用例的标准方法。

本章将首先解释测试设计技术的一般原理，描述使用测试设计技术的步骤及最终的测试产品。然后简单讨论这种导出测试用例的结构化过程的优点。

许多出版的图书、杂志，以及会议结论，都提供了大量可供使用的测试技术。本章将探讨测试设计技术的一些特征，以便深入研究一种测试设计技术如何区别于另一种技术。在决定测试策略的时候，这将有助于挑选合适的测试设计技术。

本章最后将详细讲述几种不同的测试设计技术。这几种测试设计技术将为测试人员提供良好的基础。但这里也鼓励读者研究其他著作，学习更多更新的测试设计技术是测试人员提高技巧的方法之一。

11.1.2 测试设计技术步骤的一般描述

对多数测试设计技术而言，结构化测试设计过程，是按照一个不连续步骤组成的计划来进行的。下面说明这些步骤。

确定测试情形

每一种测试设计技术的目的都是找出某种类型的缺陷，并达到一定的覆盖范围。这必须与测试基础中的信息相关，这些信息由功能需求、状态转换图、用户手册或其他规定系统行为的文档组成。测试设计过程的第一步是分析测试基础，明确每一个需要测试的情形。例如，测试设计技术可能陈述为——需要测试的情形包含所有条件，既有 true 也有 false，或者是每一个输入参数都用有效值和无效值测试过。

确定逻辑测试用例

一系列的测试情形被转换成一个逻辑测试用例，该用例从开始到完成整个过程中都贯穿于测试对象之中。逻辑测试用例可能就是测试情形，但是通常测试用例包含多个测试情形。测试用例被称为“逻辑的”，是因为它们描述了“情形的类型”，这些类型不需要为相关参数赋确定的值就可以被覆盖到。逻辑测试用例用于验证预期的测试目的和测试覆盖面是否达到。

确定物理测试用例

下一步是创建物理测试用例。对每个逻辑测试用例，选择能正确表达特定的逻辑测试用例的具体输入值，而且用具体项来定义输出值。具体输入值并不总是可以任意选择的，例如它可能受边界值或特定初始化数据集的限制。通常，物理测试用例提供了执行测试用例所必须的全部信息，包括输入值、执行的测试动作和预期结果等。

建立初始化环境

为执行物理测试用例，必须准备所需的初始化环境。这意味着必须装载一定的数据集，或者系统必须被置于某个状态。通常，多个物理测试用例可以从相同的初始化环境开始。这样就可以有效地准备初始化数据集和程序，将系统置于某个状态之下，从而能被许多测试用例重复使用。

组合测试脚本

最后一步是定义测试脚本。测试脚本能够使测试动作和对每一个测试用例的结果检查以一个最优的顺序来执行。测试脚本能为测试人员提供关于如何执行测试的逐步指导。物理测试用例与准备好的初始化环境一起构成测试脚本的基础。

定义测试方案

这是可选步骤。在某些测试脚本和其他脚本之间存在依赖关系（例如只有在脚本 A 执行完成之后，脚本 B 才能执行）的复杂情况下，建议加入该步骤。测试方案可以被认为是一种“微观测试计划”，它描述测试脚本应当执行的顺序，需要哪些准备动作，以及在“出错”情况下，可选的方案是什么。

11.1.3 优点

对测试过程而言，采用测试设计技术并将结果文档化具有许多优点。概括而言，它提高了测试过程的质量，增强了测试过程的控制。

下面给出许多论据来阐述这个观点：

- 测试策略能够提供正确的测试位置和测试范围，基于测试策略的可靠执行，采用测试设计技术就能够深入把握测试的质量和范围。
- 当测试设计技术的目标是找出某种类型的缺陷（例如在接口、输入有效性或处理过程中）时，那么和随机指定测试用例的情形相比，这些缺陷更能有效地被发现。
- 测试能够很容易地被复现，因为已经详细制定了测试执行的顺序和内容。
- 标准化的工作规程使得测试过程能够不依赖于制定和执行测试用例的人。
- 标准化的工作规程使得测试设计是可移交的，也是可维护的。
- 由于测试的制定和执行过程已经被分解成定义完善的过程块，因而更易于计划和控制测试过程。

11.1.4 特征

为帮助选择和比较测试设计技术，需要探讨测试设计技术的下列基本特征：

- 黑盒或白盒；
- 导出测试用例的原则；
- 正式或非正式；
- 适用范围；
- 被测试的质量特性；
- 必需的测试基础类型

黑盒或白盒

黑盒测试设计技术基于系统的功能性行为，不需要明确的实现细节知识。在黑盒测试中，系统只受输入值的支配，而对于输出结果，分析它是否和预期的系统行为相符合。白盒测试设计技术则是基于系统内部结构的知识，通常是基于代码、程序描述和技术设计。

但是，因为系统级的“白盒”测试可以转换成子系统级的“黑盒”测试，因而黑盒测试与白盒测试之间的界线是模糊的。

导出测试用例的原则

在利用测试设计技术导出测试用例时，有各种各样的原则。下面描述其中的一些原则：

处理逻辑

一个重要的原则是，基于被测试的程序、函数或系统处理逻辑的详细知识，来导出测试用例。这里的处理被看做一组决策点和动作的组合。决策点由一个或多个条件组成，通常表达为“IF 条件为真，THEN 继续动作 1，ELSE 继续动作 2”。可以连续执行的动作和决策的不同组合被称为路径。

下面给出一个例子（见表 11.1）来描述处理逻辑。它描述的是一个电梯在什么时候必须上升、下降或停止。这依赖于电梯的当前位置、电梯中的人选择的楼层以及电梯外面需要电梯的人所在的楼层。

表 11.1 电梯的逻辑控制

详细（伪）代码	解释
IF Floor_Selection = ON	决策 B1 由条件 C1 组成
THEN IF Floor_Intended >	决策 B2 由条件 C2 组成
Floor_Current	
THEN 电梯上升	动作 A1
ELSE 电梯下降	动作 A2
IF Floor_Intended =	决策 B3 由条件 C3
Floor_Current 或	
Floor_LiftRequested =	与 C4 组成
Floor_Current	
THEN 电梯停止	动作 A3

在这个例子中，当电梯停在第一层时，电梯里的某个人按了到第三层的按钮，同时第二层的某个人按下了电梯请求按钮，那么执行的路径为 B1/B2/A1/B3/A3。这使得电梯上升，当到第二层的时候电梯停止。

测试设计技术可以将焦点关注于覆盖路径，或覆盖每个决策点内各种可能的变化。它们由各种不同的覆盖组成。在表 11.2 中列出了与处理逻辑相关的一些很常见的覆盖类型。

表 11.2 与处理逻辑相关的覆盖类型

覆盖类型	相关处理逻辑
语句覆盖	每个动作（=语句）至少执行一次
分支覆盖或决策覆盖	每个动作至少执行一次，决策的每个可能结果（“true”或“false”）至少完成一次。这暗含了语句覆盖

(续表)

覆盖类型	相关处理逻辑
(分支) 条件覆盖	每个动作至少执行一次，条件的每个可能结果至少完成一次。这暗含了语句覆盖
决策/条件覆盖	每个动作至少执行一次，条件和决策的每个可能结果至少完成一次。这暗含了分支条件覆盖与决策覆盖。
修改过的条件的决策覆盖	每个动作至少执行一次，对独立影响决策结果的条件，该条件的每个可能结果至少完成一次。更详细的解释可参见 11.4 节。本覆盖暗含了决策/条件覆盖
分支条件组合覆盖	一个决策中，各种条件结果的所有可能组合至少执行一次。这暗含了修改过的条件的决策覆盖
路径”覆盖， $n = 1, 2, \dots$ 也被称为测试深度 n	<p>前面提到的所有类型的覆盖被称为离散动作与条件。在路径覆盖中，关注的是各种可能路径的数目。“测试深度”概念是指被测试的连续决策之间的相关程度。对测试深度 n 而言，所有的 n 个连续决策的组合都包含在测试路径中</p> <p>选择一个特定的测试深度，一方面将直接影响到测试用例的数目（因而影响到测试工作），另一方面将影响到测试的覆盖程度。使用测试深度参数，我们就可以选择符合所制定测试策略的测试用例数目</p> <p>对于复杂测试深度概念定义的例子，可参见第 11.3 节中的描述</p>

以这种方式导出测试用例的其他术语包括：逻辑测试 (Myers, 1979)、控制流程测试、路径测试以及事务流测试 (Beizer, 1995)。

可以在各种层次上考虑处理逻辑。低层次测试中关注的是程序的内部结构，程序中的语句就构成了决策与动作。高层次测试可以将功能需求视为处理逻辑。

等价类划分

采用这种原则来导出测试用例，输入范围（所有可能的输入值）被划分为“等价类”。这意味着对于一个特定等价类中的所有输入值，系统都表现出同一种行为（执行同样的处理）。通过等价类划分来导出测试用例的另一个术语是范围测试 (Beizer, 1990, 1995)。

有效等价类与无效等价类之间存在区别。无效等价类中的输入值会导致某种异常处理，例如产生一个错误消息。有效等价类中的输入值应当被正确地处理。

这一原则背后的思想是，同一个等价类中的所有输入值发现缺陷的几率均等，从该类中选择更多的输入值几乎不能增加发现缺陷的几率。只需要从每个等价类

中选择一个输入值就足够了，而无需测试每一个可能的输入值。这样可以大大减少测试用例的数目，而仍然能够获得令人满意的覆盖率。

这可以用下面的例子来描述。系统行为受下面的条件限制，该条件是关于输入值“temperature”的：

$$15 \leq \text{temperature} \leq 40$$

temperature 取值可能非常巨大（实际上为无穷）。然而，可以将这个输入范围划分为三个等价类：

- a) $\text{temperature} < 15$
- b) temperature 的取值在 15~40 之间
- c) $\text{temperature} > 40$

三个测试用例就足以覆盖这些等价类。例如可能选择到以下的 temperature 值：10（无效）、35（有效）和 70（无效）。

也可以将等价类划分原则应用于系统的输出范围，导出覆盖所有同等输出类的测试用例。

边界值分析

等价类划分原则一个重要的特殊化是“边界值分析”。独立于等价类的值被称为边界值。

这一原则背后的思想是，缺陷可能是由与边界值使用错误相关的“简单的”编程错误引发。典型例子是，编程人员的编码为“小于”，而实际的编码应当为“小于或等于”。当制定测试用例时，选择边界值周围的值，以便每个边界最少用两个测试用例来进行测试。其中一个用例中的输入值等于边界值，另一个用例中则刚刚好超过边界值。

这里使用上面的例子，并假定温度值有一个偏差 0.1：

$$15 \leq \text{temperature} \leq 40$$

选择的边界值为 14.9（无效）、15（有效）、40（有效）与 40.1（无效）。

边界值分析的另外一个更为彻底的变体，是在每个边界选择三个而不是两个值。选择的这个额外值正好在有边界值定义的等价类中。那么在上面例子中，必须测试两个额外值 15.1（有效）和 39.9（有效）。如果编程人员错误地编码为“ $15 == \text{temperature}$ ”（实际应当为“ $15 \leq \text{temperature}$ ”）。只用两个边界值将无法检测到这一缺陷，而用额外值 15.1 将可以检测出。

边界值分析与等价类划分不仅可以用于输入范围，也同样可以用于输出范围。

假定一个液晶屏幕上的一条消息最多只能为五行，那么可以发送一条包含五行的消息（屏幕上显示所有的行），以及一条包含六行的消息（在第二个屏幕上显示第六行）来进行测试。

应用边界值分析会生成更多的测试用例。但与从等价类中随机选择相比，能够提高发现缺陷的几率。

运行使用

也可以基于系统在实地的预期使用来导出测试用例。这些测试用例被设计为模拟真实情况的使用。例如实际中常用的功能，将制定成比例数目的测试用例，而不考虑功能的复杂度或重要性。基于运行使用的测试通常导致大量的测试用例，这些测试用例都属于同一个等价类。因而，它很少能发现新的功能缺陷，而更适用于性能分析和可靠性分析。

CRUD

有时候，系统行为的一个重要部分集中于数据的生命周期（创建、读取、更新、删除）。数据被提取出来，进行改动，最后又被删除。基于这一原则导出的测试用例，研究的是经由数据实体的功能交互是否正确，以及是否遵循引用关系检查（即数据模型的一致性检查）。这可以深入了解数据或实体生命周期（的完备性）。

因果图

因果图是一种将自然语言规范转变成更结构化、更正式规范的技术。它尤其适用于描述输入环境的组合影响。原因是输入条件，例如一个特定事件或一个输入范围的等价类。结果是描述系统响应的输出条件。现在，一个输出条件本身可以成为另一个结果的原因。通常，因果图展示的是一连串的原因-结果/原因-结果/原因-结果……

采用标准布尔运算符（或、与、非），就可以根据输入条件与输出条件的组合，生成一幅图来描述系统的行为。更为详细的解释可以在 Myers(1979)与 BS7925-2 中找到。

正式或非正式

正式的测试设计技术对于如何导出测试用例有着严格的规定。这样做的优点是，能将测试人员可能遗漏某些最重要用例的风险减到最少。不同的测试人员采用同一种正式的测试设计技术，应当生成相同的逻辑测试用例。缺点是测试用例质量只能像它们所基于的系统规范一样好（一个不好的系统规范将导致测试也很差）。

非正式的测试设计技术给出一般规则，留给测试人员更多的自由空间。这更强调测试人员的创造性，他们可以“搜肠刮肚”地找出系统的可能弱点。这通常要求他们是某一领域的专家。非正式的测试设计技术很少依赖于测试基础的质量，但缺点是不能充分提供对测试基础相关的覆盖度的深入把握。

适用范围

有的测试设计技术尤其适用于测试构件内部的细节处理，而有的技术更适用于测试功能和/或数据之间的集成，而有的技术则是要测试系统与外部世界（用户或其他系统）之间的交互。不同技术的适用性，与在它们的帮助下能够发现的缺陷类型有关，例如错误的输入确认、错误处理或集成缺陷等。

待测试的质量特性

一组足以覆盖待测试功能的测试用例，可能不适于测试系统性能或可靠性。一般来说，选择特定的测试设计技术主要依赖于待测试的质量特性。

要求的测试基础类型

由于测试设计技术被定义为“从测试基础导出测试用例的标准方法”，因而它要求有特定的测试基础类型。例如使用“状态转换测试”技术，要求有基于状态的系统模型。如果没有所要求的测试基础，那么组织可以决定让测试团队自己生成这样的文档。当然，这会带来各种各样明显的风险。

通常，正式的测试设计技术要比非正式的测试设计技术更多地依赖于特定测试基础的可用性。

11.2 状态转换测试

许多嵌入式系统全部或部分表现出基于状态的行为。当设计这些系统时，可使用基于状态的建模。这个过程中创建的模型可作为测试设计的基础。本章描述从基于状态的模型来导出测试用例的技术。

基于状态的测试设计技术目标是验证事件、动作、行为、状态与状态转换之间的关系。通过使用这种技术，人们就可以判定系统基于状态的行为是否满足系统的规范集合。一个系统的 behavior 可以被归为以下三种类型：

- 简单行为：对于特定输入，系统总是确切地以同一种方式做出响应，与系统历史无关。

- **连续行为：**系统的当前状态依赖于历史，而且无法标识一个单独的状态。
- **基于状态的行为：**系统的当前状态依赖于历史，并且能够与其他系统状态清晰地区别开来。

基于状态的行为可以用表、活动图或状态图来表示，但状态图是基于状态的行为最常用的方法，而且通常用 UML（统一建模语言）来描述。

附录 B 通过 UML 标准（OMG, 1997, 1999）来解释在状态图中可以看到哪些元素，每个元素是做什么用的。11.2.1 节描述了一组与状态图相关的故障类别，以及哪些故障类别可以由给出的测试设计技术来覆盖。随后的 11.2.2 节将处理如何使用状态图来作为测试设计的基础。11.2.3 节将描述“覆盖”是什么意思，该部分还将进一步考察这一测试设计技术的缺陷检测能力，并给出一些可能的方式来简化测试用例的实践。

11.2.1 故障类别

基于状态的行为出现错误可能有三种原因。第一个原因是状态图无法表示系统功能规范的正确转换。基于状态的测试设计技术不能够揭示这些故障类型，因为状态图本身被用做测试的基础。

第二个原因是状态图的语法不正确或不一致。可以通过静态测试来发现这些故障（例如通过使用审查清单或工具），如果解决了发现的故障，那么就可以使用基于状态的测试设计技术，使这个状态图成为动态测试的基础。

第三个原因是从状态图到代码的转换。这个转换过程正日益变得自动化。以这种方式生成的代码（应该）是状态图的准确表示。因而，如果采用这种方式，那么将状态图作为测试设计的基础就不再有用，应用基于状态的测试设计技术就显得多余。但是，如果没有使用生成器来自动生成基于状态图的编码，就应当使用基于状态的测试设计技术。

下面是一些状态图和软件中可能发生的故障：

1. 状态

- 1.1 没有进入转换的状态。（规范和/或实现故障）
- 1.2 遗漏初始状态。必须定义状态图中的所有路径。当转换到一个没有给出最终子状态的超状态时，超状态必须包含一个初始状态（参见附录 B.6 节，这些状态用带圆点的箭头符号来表示起点）。如果没有给出初始状态，那么就无法预测转换是在何处终止。（规范和/或实现故障）

- 1.3 额外状态。系统生成比状态图中多的状态。(实现故障)
- 1.4 遗漏状态。系统中没有出现状态图中给出的状态。(实现故障)
- 1.5 破坏性状态。转换到无效状态而导致系统崩溃。(实现故障)

2. 防护

- 2.1 防护必须指向转换而不是状态。(规范故障)
- 2.2 完成事件转换上的防护。如果防护被评估为 false, 那么系统就会陷入死锁。(规范和/或实现故障)
- 2.3 初始转换上的防护。初始转换不能有防护, 如果初始转换上的防护被评估为 false, 那么会发生什么呢? (规范和/或实现故障)
- 2.4 重叠防护。在这种情况下, 无法确定系统将转换到哪一个状态。(规范和/或实现故障)
- 2.5 防护为 false 但仍然有转换发生, 系统将到达一个预期之外的最终状态。(实现故障)
- 2.6 错误的防护实现。在某些条件下, 这将导致预期之外的系统行为。(实现故障)

3. 转换

- 3.1 转换必须有一个接收状态与一个最终状态。(规范和/或实现故障)
- 3.2 相互矛盾的转换。一个事件触发系统从一个子状态改变为另一个子状态, 而同时又触发超状态之外的一个转换, 这实际上将导致不再能够转换到某个子状态。(规范和/或实现故障)
- 3.3 遗漏或错误转换。最终的状态既不正确, 也没被破坏。(规范和/或实现故障)
- 3.4 遗漏或错误动作。执行转换时执行了错误的动作。(规范和/或实现故障)

4. 事件

- 4.1 遗漏事件。事件被忽略。(规范和/或实现故障)
- 4.2 隐含路径。系统对一个定义的事件做出响应, 但状态图中没有定义这个响应(也被称为“潜路径”)。(实现故障)
- 4.3 对一个没有定义的事件做出响应(也被称为“后门”)。(实现故障)

5. 其他

- 5.1 在正交区域使用同步。正交子状态位于一个正交区, 与另一个正交区中的事件没有连接关系。同步伪状态的错误实现将导致系统转换

到无法或很少能达到的状态，没有任何同步，或者无法或很难检测到所要求的行为。（规范和/或实现故障）

大多数建模工具能够进行语法检查，以避免模型内的语法错误。

规范故障可以用审查清单来覆盖（有时候甚至使用工具来自动执行），而实现故障应当通过状态转换来覆盖（不过有的错误类型不能由状态转换技术来覆盖）。见表 11.3。

表 11.3 错误类型的覆盖

故障类别	审查清单	状态转换技术
1.1	✓	✓
1.2	✓	✓
1.3		✓
1.4		✓
1.5		✓
2.1	✓	
2.2	✓	✓
2.3	✓	✓
2.4 ¹	✓	✓
2.5		✓
2.6 ²		
3.1	✓	✓
3.2		✓
3.3		✓
3.4		✓
4.1		✓
4.2		✓
4.3 ²		
5.1	✓	

1 这一故障类别既可以用审查清单，也可以用测试设计技术来进行测试，只要是基于模型来生成代码，那么使用审查清单来测试就足够了。但如果是基于模型进行手工编码，那么也应当使用测试设计技术来测试这一故障类别。

2 类别 4.3 不能够用测试设计技术覆盖，这种情况下认为推测错误是惟一可行的选择。类别 2.6 很难用测试设计技术来覆盖，统计使用测试可能是惟一有机会检测出这类故障的技术。

11.2.2 状态转换测试技术

概述

当测试基于状态的行为时，必须验证系统以对输入事件做出正确的响应动作，

从而使系统到达正确状态。

已经有许多基于状态的测试技术(Beizer, 1990; Beizer, 1995; IPL, 1996; Binder, 1999)。这里要描述的技术被称为 STT, 它是这些技术的综合与细化。该技术描述的是在扁平状态图上进行程度测试。11.2.3 节描述如何来改变覆盖以及如何来处理分层状态图。

状态转换测试技术 (STT) 由以下步骤组成:

1. 编写状态-事件表;
2. 编写转换树;
3. 编写合法测试用例的测试脚本;
4. 编写非法测试用例的测试脚本;
5. 编写测试脚本防护。

编写状态-事件表

状态图是编写状态-事件表的起点。状态-事件表给出状态与事件的关系。如果一个状态-事件的组合是合法的，就将这一组合的最终状态并入到状态-事件表中，而且为这个转换分配一个编号。在状态-事件表中，第一列是初始状态，接下来的列由可以直接从初始状态转换到的状态组成(从初始状态转换一次得到)，然后是从初始状态转换两次得到的状态。依次继续下去，直到所有的状态都出现在状态-事件表中。转换到自身的状态-事件组合(最终状态是接收状态)，也要包含在状态-事件表中。

根据图 11.1 的状态图，可以编写出如表 11.4 所示的状态-事件表。

如果借助防护的方法，则一个组合可能有多个最终状态，所有的最终状态都被包含在这个状态-事件表中。每个转换都分配有各自的编号。

基于状态的测试设计技术的目标，是覆盖所有的状态-事件组合(合法组合与非法组合)。

编写转换树

状态-事件表被用于编写转换树。初始状态为转换树的根。从这个初始状态开始，所有引出的转换与相关状态被加入到树中。还要复制状态-事件表中的编号并标记有防护的转换。从这些状态开始，下一级的转换与状态被加入转换树中。重复上述步骤，直到所有的路径都到达最终状态，或再次到达初始状态。如果在构造转换树的过程中，树中其他地方已经存在一个需要加入的状态，那么这个路径终止，并用一个临时终止点来标记。

如果一个到自身的转换是状态图的一部分，那么最终状态(即接收状态)也

被放在转换树中。图 B.3 也正是按照这种方式构造出来的。这样，测试用例成为一个级联的事件-转换对。在最后生成的事件-转换对中，最终状态不同于接收状态

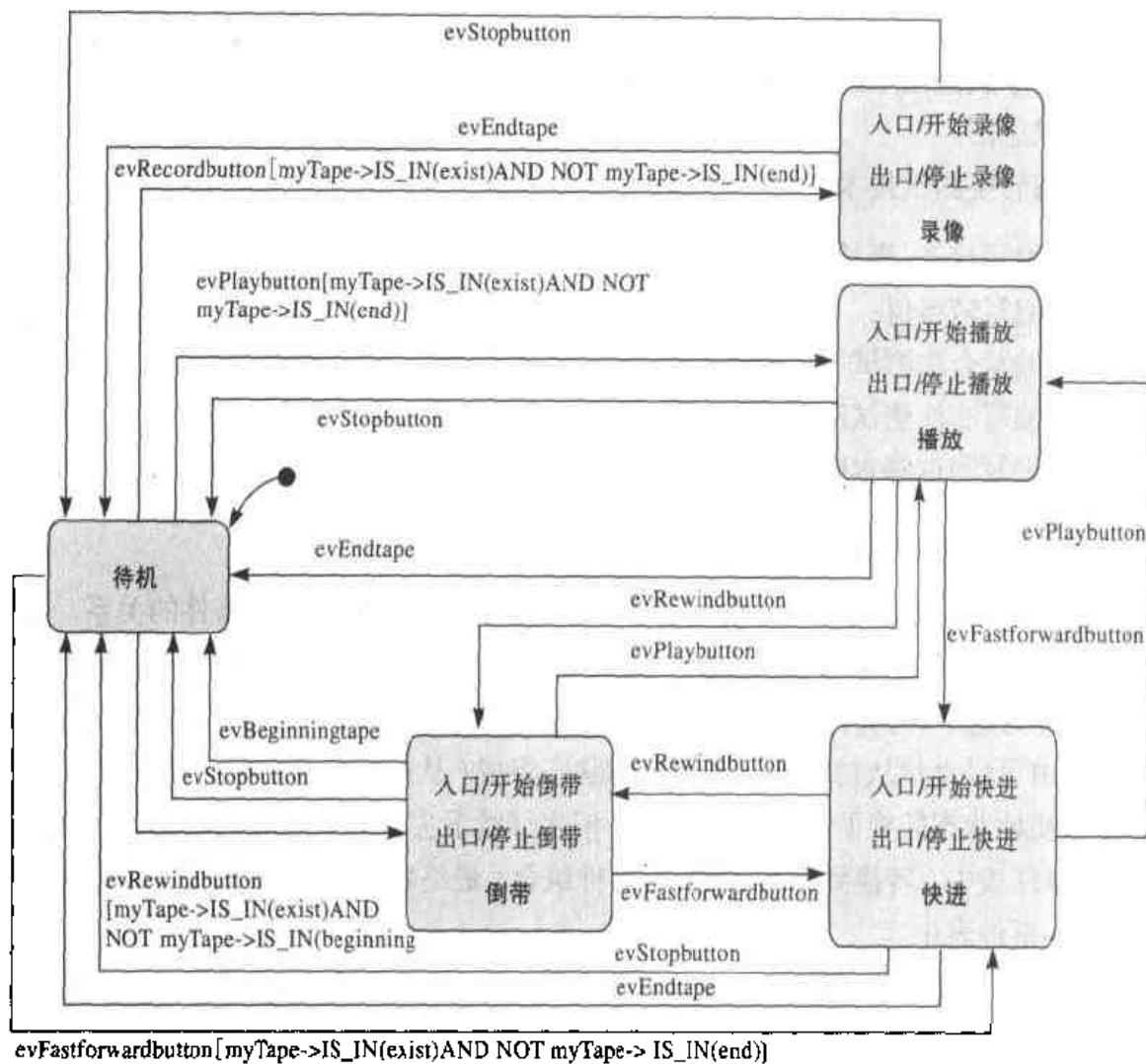


图 11.1 一个简化的录像机 (VCR) 状态图

表 11.4 状态-事件表，标黑点部分为非法的状态-事件组合

	待机	倒带	播放	快进	录音
evRewindbutton	1-倒带	●	9-倒带	13-倒带	●
evPlaybutton	2-播放	5-播放	●	14-播放	●
evFastforwardbutton	3-快进	6-快进	10-快进	●	●
evRecord	4-录音	●	●	●	●
evStopbutton	●	7-待机	11-待机	15-待机	17-待机
evEndtape	●	●	12-待机	16-待机	18-待机
evBeginningtape	●	8-待机	●	●	●
● 非法组合 (潜路径)					

在从转换树导出的测试脚本中，每个路径都沿着最短路线来完成。防护整合在转换树描述中。VCR 状态图该步骤的结果在图 11.2 中给出。

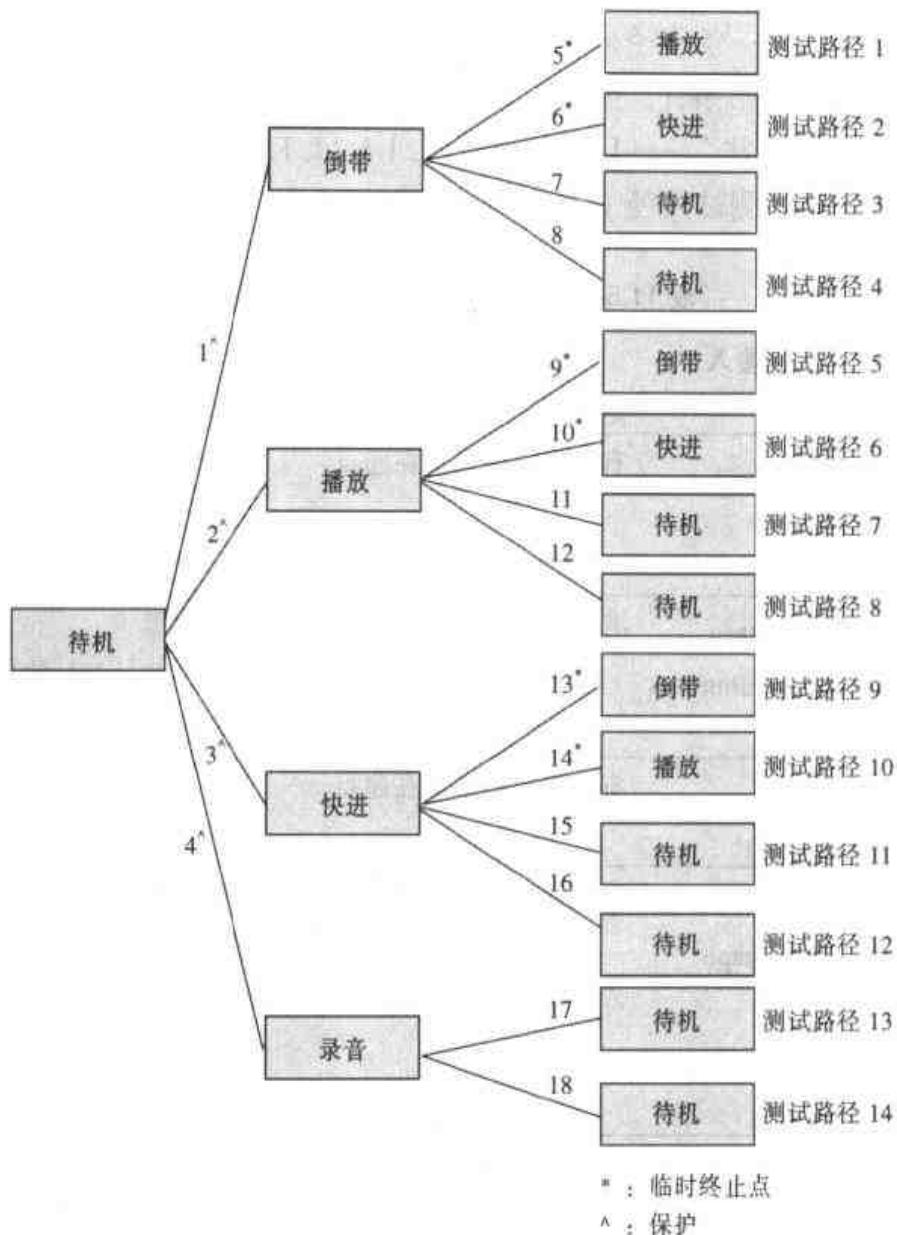


图 11.2 VCR 的转换树

转换树中的防护；

- ```
1 [myTape->IS_IN(exist)]AND NOT myTape->IS_IN(beginning)
2 [myTape->IS_IN(exist)]AND NOT myTape->IS_IN(end)
3 [myTape->IS_IN(exist)]AND NOT myTape->IS_IN(end)
4 [myTape->IS_IN(exist)]AND NOT myTape->IS_IN(end)
```

转换树与状态-事件表一起，构成了对状态图的完整描述。

## 编写合法测试用例的测试脚本

借助于转换树与状态-事件表，可以创建一个只覆盖合法测试用例的测试脚本。转换树中的每一条路径是一个测试用例，这个测试用例覆盖整个路径，在表 11.5 中，每一行包括事件、防护、预期动作与最终状态。（译者注：ID 列中的字母 L 表示英文“合法”——Legal。比如 L.1.1、L.1.2 与 L.1.3 构成一个完整的合法测试用例，覆盖测试路径 1。）

表 11.5 从 VCR 状态图导出的测试脚本

| 输入    |                     | 预期结果       |           |    |
|-------|---------------------|------------|-----------|----|
| ID    | 事件                  | 防护         | 动作        | 状态 |
| L1.1  | evRewindbutton      | 有磁带；磁带不在起点 | 开始倒带      | 倒带 |
| L1.2  | evPlaybutton        |            | 停止倒带；开始播放 | 播放 |
| L1.3  | evStopbutton        |            | 停止播放      | 待机 |
| L2.1  | EvRewindbutton      | 有磁带；磁带不在起点 | 开始倒带      | 倒带 |
| L2.2  | evFastforwardbutton |            | 停止倒带；开始快进 | 快进 |
| L2.3  | evStopbutton        |            | 停止快进      | 待机 |
| L3.1  | EvRewind            | 有磁带；磁带不在起点 | 开始倒带      | 倒带 |
| L3.2  | EvStopbutton        |            | 停止倒带      | 待机 |
| L4.1  | EvRewind            | 有磁带；磁带不在起点 | 开始倒带      | 倒带 |
| L4.2  | EvBeginningtape     |            | 停止倒带      | 待机 |
| L5.1  | EvPlay              | 有磁带；磁带不在末端 | 开始播放      | 播放 |
| L5.2  | evRewind            |            | 停止播放；开始倒带 | 倒带 |
| L5.3  | evStopbutton        |            | 停止倒带      | 待机 |
| L6.1  | evPlaybutton        | 有磁带；磁带不在末端 | 开始播放      | 播放 |
| L6.2  | evFastforwardbutton |            | 停止播放；开始快进 | 快进 |
| L6.3  | evStopbutton        |            | 停止快进      | 待机 |
| L7.1  | evPlaybutton        | 有磁带；磁带不在末端 | 开始播放      | 播放 |
| L7.2  | evStopbutton        |            | 停止播放      | 待机 |
| L8.1  | evPlaybutton        | 有磁带；磁带不在末端 | 开始播放      | 播放 |
| L8.2  | evEndtape           |            | 停止播放      | 待机 |
| L9.1  | evFastforwardbutton | 有磁带；磁带不在末端 | 开始快进      | 快进 |
| L9.2  | evRewind            |            | 停止快进；开始倒带 | 倒带 |
| L9.3  | evStopbutton        |            | 停止倒带      | 待机 |
| L10.1 | evFastforwardbutton | 有磁带；磁带不在末端 | 开始快进      | 快进 |

(续表)

| ID    | 事件                  | 输入         | 防护 | 动作        | 预期结果 | 状态 |
|-------|---------------------|------------|----|-----------|------|----|
| L10.2 | evPlaybutton        |            |    | 停止快进；开始播放 |      | 播放 |
| L10.3 | evStopbutton        |            |    | 停止播放      |      | 待机 |
| L11.1 | evFastforwardbutton | 有磁带；磁带不在末端 |    | 开始快进      |      | 快进 |
| L11.2 | evStopbutton        |            |    | 停止快进      |      | 待机 |
| L12.1 | evFastforwardbutton | 有磁带；磁带不在末端 |    | 开始快进      |      | 快进 |
| L12.2 | evEndtape           |            |    | 停止快进      |      | 待机 |
| L13.1 | evRecord            | 有磁带；磁带不在末端 |    | 开始录音      |      | 录音 |
| L13.2 | evStopbutton        |            |    | 停止录音      |      | 待机 |
| L14.1 | evRecordbutton      | 有磁带；磁带不在末端 |    | 开始录音      |      | 录音 |
| L14.2 | evEndtape           |            |    | 停止录音      |      | 待机 |

### 编写非法测试用例的测试脚本

可以从状态-事件表中得到非法的状态-事件组合。非法的状态-事件组合是指当在该特定状态时，系统没有指定要对该事件做出响应。因而，所有非法测试用例的预期结果，应当是系统对此不做出响应。

注意：不应当将非法测试用例与系统中明确指出不对某个事件做出响应，而只是输出一个错误消息的测试用例相混淆。这些测试用例中的错误消息实际上就是系统的预期响应。例如，如果对于状态-事件组合 Record 与 evPlayButton，系统指定生成一个错误消息但同时继续录音，那么这个测试用例就是合法的。如果没有指定错误消息（非法组合）而出现了错误消息，那么这被认为是一个缺陷。

在测试脚本中要描述起始情况（从非法状态-事件组合而来的接收状态）。如果这个状态与初始状态不一致，那么就需要创建一个到该状态的路径，这可以用合法测试脚本中所描述的测试步骤来创建。上述步骤的结果在表 11.6 中给出。（译者注：ID 列中的字母 I 表示英文“非法”——Illegal。）为了到达第三列给出的状态，就必须先执行第二列中给出的测试步骤（见表 11.5 ID 列）。不需要预期结果列，而是用空列记录系统响应。

表 11.6 非法测试用例的测试脚本

| ID | 准备步骤 | 状态 | 事件           | 结果 |
|----|------|----|--------------|----|
| I1 |      | 待机 | evStopbutton |    |
| I2 |      | 待机 | evEndtape    |    |

(续表)

| ID  | 准备步骤  | 状态 | 事件                  | 结果 |
|-----|-------|----|---------------------|----|
| I3  |       | 待机 | evBeginningtape     |    |
| I4  | L1.1  | 倒带 | evRewind            |    |
| I5  | L1.1  | 倒带 | evRecord            |    |
| I6  | L1.1  | 倒带 | evEndtape           |    |
| I7  | L5.1  | 播放 | evPlay              |    |
| I8  | L5.1  | 播放 | evPlay              |    |
| I9  | L5.1  | 播放 | evBeginningtape     |    |
| I10 | L9.1  | 快进 | evFastforwardbutton |    |
| I11 | L9.1  | 快进 | evRecordbutton      |    |
| I12 | L9.1  | 快进 | evBeginningtape     |    |
| I13 | L13.1 | 录音 | evRewindbutton      |    |
| I14 | L13.1 | 录音 | evFastforwardbutton |    |
| I15 | L13.1 | 录音 | evRecordbutton      |    |
| I16 | L13.1 | 录音 | evBeginningtape     |    |

### 编写测试脚本防护

如果防护由一个带有边界值的条件组成，那么需要对该防护进行边界值分析。对每一个防护，要从边界左侧和右侧两个方向来应用边界条件与测试用例。

如果防护由一个复杂的条件组成，那么可以通过修改过的条件/决策的覆盖原则来覆盖测试（见 11.4 节）。

例子：对事件 X，只有当  $timer > 100 \text{ s}$  时才有转换发生。这意味着要编写三个测试用例，其中两个为非法用例。一个测试用例中  $timer = 100 + \delta$ ，这里  $\delta$  是指该数据类型可以表示的最小数（例如整型时  $\delta$  为 1）；另一个测试用例中  $timer = 100 \text{ s}$ ；第三个测试用例中  $timer = 100 - \delta$ 。

这三个测试用例的预期结果如下：

- 防护 = true，与防护相连的转换发生；
- 防护 = false；
  - 另一个防护为 true 且与该防护相关的转换发生；
  - 无响应（→忽略）。

就像在非法测试用例表中一样，防护测试用例表中也给出指向起始情况的路径。这个路径是从合法测试用例的测试脚本复制而来的。在这个例子中不需要复

制, 因为对于所有的测试用例, 初始状态就是起始情况 (因此准备步骤列为空)。如果有的测试用例已经在合法测试用例的测试脚本中给出, 那么就在准备步骤列中标出 (见表 11.7)。防护测试用例的 ID 是指在转换树中有该防护的用例 ID。(译者注: ID 列中的字母 G 表示英文 “防护” ——Guard。)

表 11.7 为防护而编写的测试用例的测试脚本

| ID   | 准备步骤           | 状态 | 事件                  | 条件           | 预期结果           |
|------|----------------|----|---------------------|--------------|----------------|
| G1.1 | 已覆盖<br>(L1.1)  | 待机 | evFastforwardbutton | 磁带存在; 磁带不在起点 | 开始倒带;<br>状态为倒带 |
| G1.2 |                | 待机 | evRewindbutton      | 磁带存在; 磁带不在起点 | 忽略             |
| G1.3 |                | 待机 | evRewindbutton      | 磁带不存在        | 忽略             |
| G2.1 | 已覆盖<br>(L5.1)  | 待机 | evPlaybutton        | 磁带存在; 磁带不在末端 | 开始播放;<br>状态为播放 |
| G2.2 |                | 待机 | evPlaybutton        | 磁带存在; 磁带在末端  | 忽略             |
| G2.3 |                | 待机 | evPlaybutton        | 磁带不存在        | 忽略             |
| G3.1 | 已覆盖<br>(L9.1)  | 待机 | evFastforwardbutton | 磁带存在; 磁带不在末端 | 开始快进;<br>状态为快进 |
| G3.2 |                | 待机 | evFastforwardbutton | 磁带存在; 磁带在末端  | 忽略             |
| G3.3 |                | 待机 | evFastforwardbutton | 磁带不存在        | 忽略             |
| G4.1 | 已覆盖<br>(L13.1) | 待机 | evRecord            | 磁带存在; 磁带不在末端 | 开始录音;<br>状态为录音 |
| G4.2 |                | 待机 | evRecord            | 磁带存在; 磁带在末端  | 忽略             |
| G4.3 |                | 待机 | evRecord            | 磁带不存在        | 忽略             |

### 11.2.3 广泛性和实用性

#### 测试分层状态表

到目前为止, 我们都假定一个状态图仅描述一个层级 (实际上是一个扁平的状态图), 它与其他状态图之间没有任何关系。而实际上, 状态图可以由不同的层级组成, 因而最上层的状态图被分解为多个底层的状态图。这些状态图可以按照自底向上与自顶向下的方式进行测试。

如果底层的状态图细节不是很清楚, 实践中多采用自顶向下的方法。选择这种方法的另一个原因, 是需要透彻了解几个子系统是否已正确连接。我们只在超状态层来检查底层状态图, 只对进入与流出超状态的转换进行检查。

在自底向上的方法中, 首先测试最底层的状态图。然后连续测试更高层的集

成，直到整个系统最终被测试。当测试某一层功能的时候，用“黑盒”方式来测试其低层的功能，即只对进入与流出超状态的转换进行测试。

在这两种测试方法中，按照相同的方式来应用技术。在自顶向下的测试中，不用考虑测试下一层，因为将会在随后对下一层进行测试。然而，在自底向上的测试中，不用考虑测试下一层，是因为已经在前面对下一层进行了测试。

### 广泛性

通过确定测试连续转换之间的依赖程度，可以改进基于状态的测试方法的有效性。可以像基于状态的测试中所描述的那样，对一个转换进行单独测试。但也可能测试一些转换的组合。在一个组合中被测试的转换数被称为测试深度。

测试深度被用于计算测试覆盖率。在本文中，使用术语  $n$  次转换覆盖率，或  $n-1$  次切换覆盖率，这里的  $n$  表示转换次数（因此也用于测试深度）。

$$1 \text{ 次转换覆盖率}/0 \text{ 次切换覆盖率} = \frac{\text{执行的转换数}}{\text{状态模型中的转换总数}}$$

$$2 \text{ 次转换覆盖率}/1 \text{ 次切换覆盖率} = \frac{\text{执行的两次转换的序列数}}{\text{状态模型中两次转换的序列总数}}$$

测试深度  $n$  ( $n$  次转换覆盖率或  $n-1$  次切换覆盖率) 意味着覆盖所有  $n$  次连续转换。

在上面的覆盖率描述中，我们只是从“正面测试”（测试正确路径）的观点来评判覆盖。但也可以考虑被覆盖的状态-事件组合数，从而将覆盖率的度量扩展到包含负面测试。

$$\text{状态-事件覆盖程度} = \frac{\text{状态数}-\text{执行的事件对}}{\text{状态数}\times\text{事件数}}$$

在这里，不能依据覆盖程度来推断出任何有关测试质量的评判。覆盖程度只是指出在规范期间是否忽略了某些活动。

### 故障检测

这里基于测试深度来描述基于状态的测试设计技术。这个测试深度实际上就是指测试基于状态的行为所需的最少工作。Binder (2000) 认为这一点还不够，他更喜欢用的是来回测试 (round trip test)，即应用测试深度  $n$  ( $n$  等于模型中的

转换数)。这需要进行大量的工作,只应用于系统的关键部分。

11.2.1节提到了用基于状态的测试技术覆盖的一些故障类别。表11.8给出了基于状态的测试设计技术中有助于故障检测的那些部分。这个表揭示出了测试深度的增加是如何来影响故障检测的。

表11.8 测试效果与故障检测的关系

| 故障类别              | 合法测试用例 | 非法测试用例 | 边界值分析 | 更深程度的测试 |
|-------------------|--------|--------|-------|---------|
| 1.3 额外状态          | +      | +      | +     | +       |
| 1.4 遗漏状态          | +      | -      | -     | -       |
| 1.5 破坏性状态*        | -      | +/-    | +/-   | +       |
| 2.4 重复防护          | -      | -      | +/-   | -       |
| 2.6 防护为 false 的转换 | -      | -      | +     | -       |
| 3.2 相互矛盾的转换       | +      | -      | -     | -       |
| 3.3 遗漏或错误转换       | +      | +      | +     | +/-     |
| 3.4 遗漏或错误动作       | +      | -      | -     | +/-     |
| 4.1 遗漏或错误事件*      | +      | -      | -     | +/-     |
| 4.2 隐含路径          | -      | +      | +     | +/-     |

"-" 表示检测出故障的希望很渺茫。

"+/-" 表示检测出故障的概率很小。

"+" 表示检测出故障的概率很大。

\* 进行更深程度的测试可能不足以检测出破坏性状态(故障类别1.5),或隐含路径(错误类别4.2)。通常这些缺陷只能在不把系统设为初始状态的情况下,重复测试动作许多次才会显现,当然问题就是我们无法预测进行多少次重复测试才能保证没有这种缺陷。

基于这个表,可以对基于状态的测试设计技术的不同部分,以及测试深度的使用做出选择。

### 实用性和可行性

在实践中,对系统基于状态的行为进行测试并不总是简单的。由于从外部观察来看,系统的行为被大量隐藏,因而难以确定系统的行为。例如,在面向对象中,用数据封装来隐藏系统的数据状态。在系统到达能被观测的状态之前,一个事件可能导致一系列的转换。因此,很难甚至不可能将单个转换与状态区分开来。

确实有许多因素可以影响实用性和系统响应的确定:

1. **步进方式:** 这可以使系统每次执行一个转换。
2. **重置选项:** 这可以返回到初始状态。
3. **状态设置:** 这可以将系统设置为一个特定的状态。

4. 对状态、转换、输入、输出与事件进行惟一编码，这样在任意时间都可以请求该信息；同时在任意时间都能够察看事件状态。

5. 转换跟踪：可以查看出转换的顺序。

可以采用模拟器、UML-编译器（比较模拟器与编译器之间的相对优点与缺点可参见 Douglass, 1999）或最终产品来测试软件。模拟器通常包含步进模式、重置和状态设置功能。但编译器就不一样了，通常使用的多是定制编译器。为了使它们对测试人员实用，将前三个选项作为编译器中的功能是很有用的。最终产品只是提供了用户所需的功能。因此，上面提到的选项通常就不会出现。

在编码或建模中必需实现惟一编码。不仅仅每个状态、转换、输入、输出与事件必须有各自惟一的名称，而且与这些项相关联的系统状态也必须是“可请求的”。例如，这可以通过为每一个状态连接一个布尔值来实现。当进入该状态时，设置这个相连的布尔值为 true，而一旦脱离该状态，则设置这个相连的布尔值为 false。如果执行的代码与输入、输出或事件相关联，则记录在跟踪文件中。这样就建立了转换跟踪，按照时间顺序可以读出动作、转换、状态、输入、输出与事件。当测试环境无法提供步进模式来跟踪模型或代码时，转换跟踪将是非常有用的。

## 11.3 控制流测试

### 11.3.1 介绍

控制流测试的目标是测试程序结构。从算法和/或程序的结构来导出测试用例。每个测试用例由一组动作组成，这一组动作覆盖该算法的一个特定路径。控制流测试是一个正式的测试设计技术，主要用于单元测试与集成测试。

### 11.3.2 规程

控制流测试由下面的步骤组成：

1. 编制决策点的清单；
2. 确定测试路径；
3. 定义测试用例；
4. 建立初始数据集；
5. 组合测试脚本；
6. 执行测试

## 决策点

程序设计或技术设计被用做测试基础。这个测试基础应当包含待测算法的结构描述，例如流程图、决策表、活动图等。如果测试没有提供结构的描述，那么必须基于可用的信息来准备关于程序结构的文档。如果无法提供算法结构的描述，那么不能使用控制流测试。

必须确定出程序结构中的所有决策点并惟一标识它们。而且将两个连续决策点之间的动作也视为一个大动作，并惟一标识。

## 测试路径

将动作组合到测试路径中依赖于预期的测试深度。测试深度被用于确定被测试的连续决策点之间的相关程度。测试深度  $n$  意味着在 1 个决策点之前， $n-1$  个决策点之后的动作的所有相关性都被检验。 $n$  个连续动作的所有组合都被用到。

测试深度对测试用例数目及测试的覆盖程度有直接的影响，对测试用例数目有直接影响也意味着对测试工作有直接的影响。

为了说明测试深度如何影响测试用例，下面给出两个测试深度的例子。对每一个标识出的决策点，确定其动作组合。对于测试深度 2，编制一个清单，列出两个连续动作所有可能的组合。在测试深度 2 中，每个动作组合是决策点之前的动作与之后动作的组合。列出所有可能的组合。如图 11.3 中使用的流程图例子。

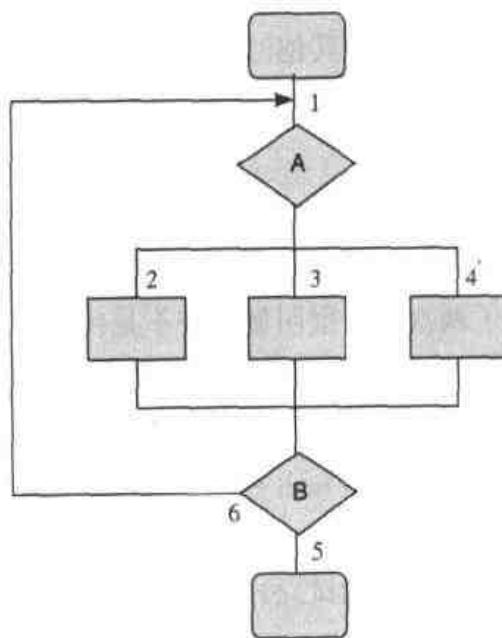


图 11.3 一段程序的结构流程图

决策点的动作组合为：

A : (1,2); (1,3); (1,4); (6,2); (6,3); (6,4)

B : (2,5); (3,5); (4,5); (2,6); (3,6); (4,6)

1. 将动作组合升序排列：(1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4)。
2. 必须链接动作组合以创建从算法起点到算法终点的运行路径。在这个例子中，这意味着每个路径必须从动作 1 开始，到动作 5 结束。
3. 从还没有被包含在路径中的第一个动作组合开始。在这个例子中是(1,2)。接下来，从还没有被包含在路径中，以 2 开始的第一个动作组合开始。将它链接在第一个被选择的组合之后。在这个例子中是(2,5)。这样就创建了路径(1,2,5)，可以开始选择其他的路径了。
4. 剩余的动作组合为：(1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4)。
5. 继续处理剩余的动作组合。还没有被包含在路径中的第一个动作组合是(1,3)。以 3 开始还没有被包含在路径中的第一个动作组合是(3,5)。这样就创建了路径(1,3,5)，可以开始选择其他的路径了。
6. 剩余的动作组合为：(1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4)。
7. 继续剩余的动作组合。还没有被包含在路径中的第一个动作组合是(1,4)。以 4 开始还没有被包含在路径中的第一个动作组合是(4,5)。这样就创建了路径(1,4,5)，可以开始选择其他的路径了。
8. 剩余的动作组合为：(1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4)。
9. 还没有被包含在路径中的第一个动作组合是(2,6)。因为这个动作组合不是从算法起点开始的，因而必须确定其前面的动作组合，可选择(1,2)。这个组合已经被使用了两次，但很明显这并不是什么问题。接下来，继续选择以 6 开始还没有被包含在路径中的第一个动作组合，可选择 (6,2)。选择组合(2,5)形成一个完整的路径。这样就创建了路径(1,2,6,2,5)。
10. 剩余的动作组合为：(1,2); (1,3); (1,4); (2,5); (2,6); (3,5); (3,6); (4,5); (4,6); (6,2); (6,3); (6,4)。
11. 剩余的动作组合被包含在测试路径(1,3,6,4,6,3,5)中。现在在下面的路径中已经包含了所有的动作组合：

路径 1：(1,2,5)

路径 2：(1,3,5)

路径 3: (1,4,5)

路径 4: (1,2,6,2,5)

路径 5: (1,3,6,4,6,3,5)

测试深度二是基于以下想法：执行一个动作可以在决策点之后立即有结果。对一个动作来说，不仅是这个决策点重要，而且前面的动作也重要。相反，测试深度一假定一个动作只受到决策点的影响。

对于测试深度一，会形成下面的动作组合：

- : (1)

A : (2); (3); (4)

B : (5); (6)

这将生成下面的路径组合：

路径 1: (1,2,5)

路径 2: (1,3,6,4,5)

测试深度越低，测试工作就越少，其代价是覆盖程度越低。

测试深度越高，也就意味着执行一个动作会影响到其后面两个或多个动作。只有在安全性相关的部分或程序很复杂的部分，才使用高测试深度。

### 测试用例

导出的逻辑测试路径被转化为物理测试用例。对每一个测试路径，必须确定其测试输入。确定一个测试路径的正确测试输入是一件很难的事情。所选择的测试输入必须要使得每一个决策点都沿着正确的路径。如果在测试路径执行算法的期间输入发生了改变，那么就必须返回到起点来重新计算该值。但有时候算法十分复杂，因而这实际上是不可能的，例如当使用科学算法的时候。对每一个测试输入，必须确定其预期的输出。对于那些不会影响测试路径的变量和参数，应当为其设定默认值。

### 初始化数据集

有时候，测试执行需要有一个特定的初始化数据集。简要描述这个初始化数据集，并将这个描述加入到测试路径和测试用例的描述中。

### 测试脚本

前面所有的步骤都是测试脚本的基础。测试脚本描述测试动作，并保证按照

正确的顺序来执行测试。测试脚本也列出测试脚本要执行的前提条件。前提条件通常有存在一个初始化数据集、系统应当处在某个特定的状态或一些特定的占位与驱动程序的可用性。

### 测试执行

依照测试脚本来执行测试，将结果与预期的输出进行比较。对于单元测试，测试者必须像开发者一样进行多次测试。下一步是查明输出结果与预期输出之间存在差异的原因。

---

## 11.4 基本比较测试

### 11.4.1 介绍

在基本比较测试（ECT）中，需要详细测试过程。测试要验证一个函数的所有功能路径，必须标识出所有的功能条件并转化为伪代码。要从伪代码导出测试用例，这些测试用例要覆盖标识出的功能路径。

ECT 测试要充分覆盖条件并保证相当程度的完全性。由于这种正式的技术需要进行大量的工作，因而它主要用于很重要的功能和/或很复杂的计算。

### 11.4.2 规程

ECT 技术由以下步骤组成：

1. 分析功能描述；
2. 确定测试情况；
3. 确定逻辑测试用例；
4. 确定物理测试用例；
5. 确定测试动作；
6. 检查；
7. 确定起始情况；
8. 组织测试脚本。

为了阐述上述这些步骤，使用下面的功能描述来举例说明：

```
If |TempSensor1 - TempSensor2| ≤ 40
Then Actorheater_1 = OFF
Else
```

```

If TempSensor2 ≥ 70 AND TempSensor1 < 45 AND Actorvalve_1 = ON
 Actorvalve_2 = OFF
ENDIF

If VolumeSensor < 1000 AND Actorvalve_2 = ON AND Actorheater_1
 = ON AND (Actorheater_2 = OFF OR TempSensor2 ≥ 50)
 Actorvalve_1 = OFF
ENDIF

ENDIF

```

### 分析功能描述

功能描述应当明确地描述出决策路径和相关特性。功能已经通过伪代码或某种技术，例如决策表得到描述。如果明确地描述出决策路径，即使是按照自由格式来描述功能，那么也可以应用 ECT 技术。如果功能描述不是伪代码，那么将描述转化为伪代码将很有帮助。

第一步是标识出条件。通常可通过如“DO”、“IF”或“REPEAT”等项来识别出条件。将条件一个接一个地挑选出来，并为每个条件分配一个惟一的标识，只需考虑那些由数据输出驱动的条件。类似于“DO as long as counter < 10”的条件是内部驱动，而不是由外部的数据输入来驱动，因而将被忽略。

```

C1 If |TempSensor1 - TempSensor2| ≤ 40
 Then Actorheater1 = OFF
 Else
 C2 If TempSensor1 ≥ 70 AND TempSensor2 < 45 AND Actorvalve1 = ON
 Actorvalve2 = OFF
 ENDIF
 C3 If VolumeSensor < 1000 AND Actorvalve2 = ON AND Actorheater1
 = ON AND (Actorheater2 = OFF OR TempSensor1 ≥ 50)
 Actorvalve1 = OFF
 ENDIF
 ENDIF

```

### 确定测试情况

在已经标识出条件之后，必须要确定每个条件被测试的环境。区分简单和复杂条件之间的差异是很重要的。简单条件只由一个比较，即基本比较组成，条件 C1 就是基本比较的例子。复杂条件是包含多个比较的条件，通过 AND 或 OR 等

关系来连接，条件 C2 就是复杂条件的例子。

简单条件导致两种情况，即条件为 false 或 true。在上面的例子中，将导致两个测试环境，在表 11.9 中给出。

表 11.9 上例中条件 C1 的测试环境

| 测试环境                                              | 1    | 2     |
|---------------------------------------------------|------|-------|
| C1                                                | true | false |
| TempSensor <sub>1</sub> = TempSensor <sub>2</sub> | ≤ 40 | > 40  |

复杂条件是简单条件的组合。复杂条件或者为 true 或者为 false，但这个值依赖于所组成的每个简单条件是 true 还是 false。条件是通过连接词 AND 还是 OR 来连接很重要。连接词对于复杂条件的取值影响在表 11.10 中给出。

表 11.10 复杂条件中连接词 AND 和 OR 的作用

| 复杂条件 |   | 结果  |    |
|------|---|-----|----|
| A    | B | AND | OR |
| 0    | 0 | 0   | 0  |
| 0    | 1 | 0   | 1  |
| 1    | 0 | 0   | 1  |
| 1    | 1 | 1   | 1  |

如果通过连接词 AND 连接的复杂条件取值为 true，则只有当两个简单条件都为 true 时才成立。而通过连接词 OR 连接的复杂条件取值为 true，只要其中有一个简单条件的取值为 true 就成立。

复杂条件的可能情况与简单条件的数目成指数关系。由两个简单条件组成的复杂条件有  $2^2 = 4$  种可能情况，由三个简单条件组成的复杂条件有  $2^3 = 8$  种可能情况，对于七个简单条件的组合，则有 128 种可能情况。ECT 技术就是基于这样的事实，即确定出所有的可能情况并进行测试是没有用处的，应当选择那些只要其中任何一个基本比较的值发生了改变，就会改变复杂条件结果的测试情况。例如在 OR 关系时 true-true 的情况，或在 AND 关系时 false-false 的情况都将被忽略。在 true-false 或 false-true 情况，发现错误的可能性是最小的。

在 AND 关系情况下，只需要测试以下情况：每个基本比较取真值 1，以及有且仅有 1 个基本比较取真值 0。对 OR 关系，需要测试的情况正相反：每个基本比较取真值 0，以及有且仅有 1 个基本比较取真值 1。

ECT 将测试情况数目减少为基本比较数目加 1。

对例子中的复杂条件 C2 应用 ECT 技术，其结果在表 11.11 中给出。

表 11.11 例子中复杂条件 C2 的测试情况

| 测试情况                                 | C2.1   | C2.2   | C2.3   | C2.4   |
|--------------------------------------|--------|--------|--------|--------|
| C2(=C2a & C2b & C2c)                 | 1(111) | 0(011) | 0(101) | 0(110) |
| C2a TempSensor <sub>1</sub> ≥ 70 AND | ≥70    | <70    | ≥70    | ≥70    |
| C2b TempSensor <sub>2</sub> < 30 AND | <30    | <30    | ≥30    | <30    |
| C2c Actor <sub>valve1</sub> = ON     | ON     | ON     | ON     | OFF    |

对于只有 AND 或 OR 关系的复杂条件，很容易找到测试情况。而当复杂条件由一个或多个 AND 和 OR 关系组成时，要找到测试情况就变得更加困难，例如对例子中的条件 C3。这时候要使用一个矩阵（见表 11.12）。基本比较被排在第一列，第二列给出了复杂条件的真值如何才能为 true，并使得只要有一个基本比较的取值（该值有下划线）发生改变，复杂条件的取值将变为 false。下一列给出了复杂条件的取值为 false 的情况。很明显，在第二列与第三列之间的惟一区别就是带下划线的值。

表 11.12 例子中条件 C3 的测试情况

| C3 (=C3a & C3b & C3c & (C3d C3e))  | 1 (Actor <sub>valve2</sub> = OFF) | 0                  |
|------------------------------------|-----------------------------------|--------------------|
| C3a VolumeSensor < 1000            | <u>1</u> .1.1.01                  | 0 <u>1</u> .1.1.01 |
| C3b Actor <sub>valve2</sub> = ON   | 1 <u>1</u> .1.1.01                | 1 <u>0</u> .1.1.01 |
| C3c Actor <sub>heater1</sub> = ON  | 1.1. <u>1</u> .01                 | 1.1.0 <u>0</u> 1   |
| C3d Actor <sub>heater2</sub> = OFF | 1.1.1. <u>1</u> 0                 | 1.1.1. <u>0</u> 0  |
| C3e TempSensor ≥ 50                | 1.1.1.0 <u>1</u>                  | 1.1.1.0 <u>0</u>   |

在去掉重复测试情况后（见 11.13），剩下的就是逻辑测试情况。

表 11.13 例子中条件 C3 的剩余测试情况

| C3 (=C3a & C3b & C3c & (C3d C3e))  | 1 (Actor <sub>valve2</sub> = OFF) | 0                  |
|------------------------------------|-----------------------------------|--------------------|
| C3a VolumeSensor < 1000            | <u>1</u> .1.1.01                  | 0 <u>1</u> .1.1.01 |
| C3b Actor <sub>valve2</sub> = ON   | <u>1</u> <u>1</u> .1.01           | 1 <u>0</u> .1.1.01 |
| C3c Actor <sub>heater1</sub> = ON  | 1 <u>1</u> . <u>1</u> .01         | 1.1. <u>0</u> 01   |
| C3d Actor <sub>heater2</sub> = OFF | 1.1.1. <u>1</u> 0                 | 1.1.1. <u>0</u> 0  |
| C3e TempSensor <sub>1</sub> ≥ 50   | 1.1.1.0 <u>1</u>                  | 1.1.1.0 <u>0</u>   |

在第五行，复杂条件 (C3d|C3e) 为 10 而不是 01，是因为当为 01 时如果基本条件 C3d 发生改变，它就会成为可以忽略的情况 (11)。

接下来将剩余的测试情况转化为逻辑测试情况（见表 11.14）。

表 11.14 例子中条件 C3 的逻辑测试情况

| 测试情况                     | C3.1     | C3.2     | C3.3     | C3.4     | C3.5     | C3.6     |
|--------------------------|----------|----------|----------|----------|----------|----------|
| C3                       | 1(11101) | 1(11110) | 0(01101) | 0(10101) | 0(11001) | 0(11100) |
| VolumeSensor             | < 1000   | < 1000   | ≥ 1000   | < 1000   | < 1000   | < 1000   |
| Actor <sub>valve2</sub>  | ON       | ON       | ON       | OFF      | ON       | ON       |
| Actor <sub>heater1</sub> | ON       | ON       | ON       | ON       | OFF      | ON       |
| Actor <sub>heater2</sub> | ON       | OFF      | ON       | ON       | ON       | ON       |
| TempSensor <sub>1</sub>  | ≥ 50     | < 50     | ≥ 50     | ≥ 50     | ≥ 50     | < 50     |

### 确定逻辑测试用例

确定逻辑测试用例是指找出功能路径，在这些功能路径中，每个条件的每一种情况都至少应当完成一次。这种方法可能导致同一个比较会在多个条件中发生。当选择路径组合时，也应当确保预期的最终结果就像选择路径组合一样尽可能唯一。要注意的是后面的条件可能会相互影响。在这个例子中，条件 C2 为 true 将使 Actor<sub>valve2</sub> 变为 OFF，这将自动意味着条件 C3 将为 false。

为了找出测试用例，并且检查在测试用例中是否所有的情况都已经被记录下来，可以采用下面的矩阵（见表 11.15）。所有定义的测试情况在第一列中给出，第二列给出了在该测试情况下的条件值，第三列为下一个要处理的条件。接着列出的是测试用例。如果测试用例已经被正确地定义，那么被测试的所有情况将用记号来标记至少一次。为了覆盖上面例子中所有的测试环境，确定了七个测试用例。

表 11.15 帮助构造逻辑测试用例的矩阵

| 测试情况 | 值 | 转向  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 控制 |
|------|---|-----|---|---|---|---|---|---|---|----|
|      |   |     |   |   |   |   |   |   |   |    |
| C1.1 | 1 | End | x |   |   |   |   |   |   | 1  |
| C1.2 | 0 | C2  |   | x | x | x | x | x | x | 6  |
| C2.1 | 1 | C3  |   | x |   |   |   |   |   | 1  |
| C2.2 | 0 | C3  |   |   | x |   |   | x |   | 2  |
| C2.3 | 0 | C3  |   |   |   | x |   |   | x | 2  |
| C2.4 | 0 | C3  |   |   |   |   | x |   |   | 1  |
| C3.1 | 1 | End |   |   |   |   | x |   |   | 1  |

(续表)

| 测试情况 | 值 | 转向  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 控制 |
|------|---|-----|---|---|---|---|---|---|---|----|
| C3.2 | 1 | End |   |   | x |   |   |   |   | 1  |
| C3.3 | 0 | End |   |   |   | x |   |   |   | 1  |
| C3.4 | 0 | End |   | x |   |   |   |   |   | 1  |
| C3.5 | 0 | End |   |   | x |   |   |   |   | 1  |
| C3.6 | 0 | End |   |   |   |   | x |   |   | 1  |

有的情况会被测试多次。这通常无法避免，而且不应当被看做是浪费时间，而应当被看做是一次机会：通过改变测试情况的输入，可得到更为全面的测试。

### 确定物理测试用例

下一步是将逻辑测试用例转化为物理测试用例（见表 11.16）。

表 11.16 例子中的物理测试用例

| 测试用例                     | 1                                                                            | 2    | 3    | 4    | 5    | 6    | 7   |
|--------------------------|------------------------------------------------------------------------------|------|------|------|------|------|-----|
| 路径                       | C1.1 C1.2, C2.1, C1.2, C2.2, C1.2, C2.3, C1.2, C2.4, C1.2, C2.2, C1.2, C2.3, |      |      |      |      |      |     |
|                          | C3.4                                                                         | C3.2 | C3.5 | C3.3 | C3.1 | C3.6 |     |
| TempSensor <sub>1</sub>  | 50                                                                           | 20   | 30   | 145  | 20   | 10   | 40  |
| TempSensor <sub>2</sub>  | 40                                                                           | 90   | -20  | 95   | 90   | 60   | -15 |
| Actor <sub>valve1</sub>  | OFF                                                                          | ON   | ON   | ON   | OFF  | ON   | ON  |
| Actor <sub>valve2</sub>  | OFF                                                                          | OFF  | ON   | ON   | ON   | ON   | ON  |
| Actor <sub>heater1</sub> | OFF                                                                          | ON   | ON   | OFF  | ON   | ON   | ON  |
| Actor <sub>heater2</sub> | ON                                                                           | ON   | OFF  | ON   | ON   | ON   | ON  |
| VolumeSensor             | 500                                                                          | 700  | 800  | 500  | 1200 | 600  | 200 |

### 确定测试动作

测试动作是预先确定的动作，可能成功也可能失败。为了这个目标，首先列出所有相关的子任务。这个例子只包含一个测试动作，即系统开始运行。

### 检查

有必要用检查来确定处理是否已经成功。检查是指在测试用例或在测试动作已经执行后对预期情况的描述。表 11.17 给出了本章例子中的检查列表。

表 11.17 例子中的所有检查

| 检查  | 测试用例 | 预期情况                                                      |
|-----|------|-----------------------------------------------------------|
| C01 | 1    | Actor <sub>heater1</sub> = OFF and other actors unchanged |
| C02 | 2    | Actor <sub>valve2</sub> = OFF and other actors unchanged  |
| C03 | 3    | Actor <sub>valve1</sub> = OFF and other actors unchanged  |
| C04 | 4    | All actors unchanged                                      |
| C05 | 5    | All actors unchanged                                      |
| C06 | 6    | Actor <sub>valve1</sub> = OFF and other actors unchanged  |
| C07 | 7    | All actors unchanged                                      |

### 确定起始情况

有时候，在测试用例开始执行之前，必须要满足某些特定的前提条件。这些前提条件可能是必须装填一个特定的初始化数据集，或者是系统必须被设定为一个特定的状态。对每一个测试用例来说，这个特定状态和初始数据集可能都不相同。在本章的例子中，不需要有专门的措施来建立起始情况。

### 组织测试脚本

测试脚本应当提供执行测试用例所需的全部信息。因此，一个测试脚本应当包含测试用例、与用例相关的动作以及按正确顺序排列的检查。此外，还要列出所有的前提条件与后置条件，并描述起始情况。

## 11.5 分类树方法

### 11.5.1 介绍

分类树方法（CTM）支持黑盒测试用例的系统设计。它是一个非正式的测试设计技术，虽然有很清晰的规程描述来导出测试用例。CTM 确定被测试系统的有关特性，即那些可能影响到诸如功能行为或系统安全性的特性。依据确定的特性，采用等价划分法将测试对象的输入域划分到不相交的类中。输入域的划分采用树的形式来图形化表示。通过将不同特性类组合来形成测试用例，其实现方法是通过将分类树当成组合表的前部，测试用例在该组合表中标记。

最理想的方式是，将 CTM 与功能需求一起用做测试基础。但不幸的是，这些需求并不总是可以得到，或者是不完整或者不是最新的。CTM 一个独特的优点在于，它即使在这种情况下也可以应用，由此测试人员可以填写遗漏的信息。测

试人员应当至少对被测系统有一个基本的了解。如果是遗漏了功能性需求，那么要求测试人员必须对系统所期望的功能性行为有深入的了解。

分类树方法是由 Grochtmann 与 Grimm (Grochtmann & Grimm, 1993) 开发出来的。16.2.2 节有专门开发用于混合信号描述的嵌入式系统的扩展内容。

## 11.5.2 规程

CTM 技术由以下步骤组成：

1. 确定测试对象的特性；
2. 依据特性来划分输入域；
3. 确定逻辑测试用例；
4. 组织测试脚本。

可以用分类树编辑器 (Grochtmann, 1994) 等工具来支持该规程。

下面用一个导航控制软件作为例子来描述上述步骤。这个例子已经被简化而且有许多特性没有被考虑，只考虑了汽车的实际速度、实际速度和预期速度之间的差值，还有一个复杂因子，即导航控制系统还能对前面的汽车做出反应等几个特性。前面的汽车与这个装备有导航控制系统的汽车之间的速度差值，对于导航控制系统的行为是很重要的。除此之外，前面汽车与这个装备有导航控制系统的汽车之间的距离，也会影响到导航控制系统的行为。事实上，这里提到的许多特性应当被描述为时间、空间和速度的一个函数。

### 确定测试对象的特性

第一步是确定出影响被测系统处理的特性。对这个例子来说，这些特性包括汽车的实际速度、实际速度与预期速度之间的差值，以及前面是否有汽车。这些特性可以用一个树来图形化表示（见图 11.4）。



图 11.4 智能导航控制的三个特性

### 依据特性来划分输入域

依据确定出的特性，将测试对象的输入域划分为类。对一个类中的所有成员，

系统都表现出同一种行为。这种划分方法被称做等价划分法（见 11.1.4 节）。这些类是正交类，即每一个输入值是且仅是一个类中的成员。图 11.5 给出了划分到类中的结果。

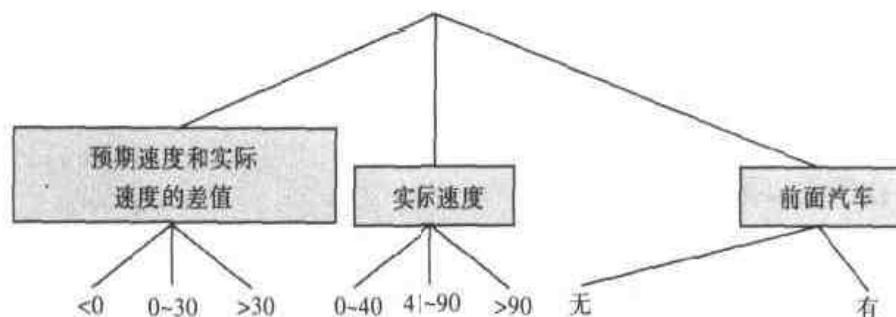


图 11.5 输入域的划分

构造分类树可以是一个递归的活动。在本例中，如果这个汽车前面有别的汽车，而且两个车之间的距离及速度差值将对系统功能产生影响，那么特性“前面有汽车”的“Yes”类构成了另一级分类树的根（见图 11.6）。接下来再一次确定特性，并依据这些特性来划分输入域，重复进行这些活动，直到确定出所有有关的特性，输入域的划分完成。

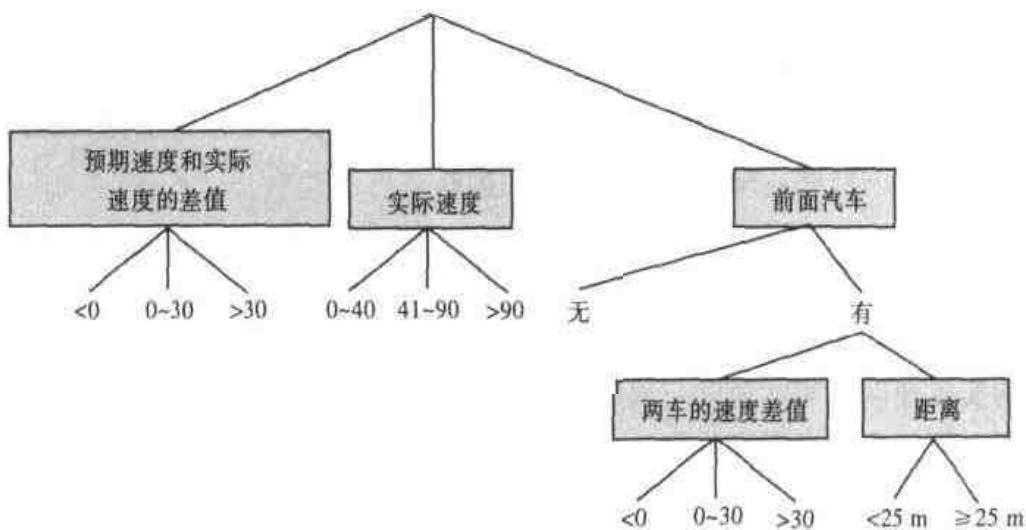


图 11.6 在迭代应用步骤 1 与步骤 2 之后，得出的完整分类树

### 指定逻辑测试用例

完成分类树以后，就有了准备测试用例所有必要的信息。测试用例通过以下步骤来构造：

- 首先为每一个特性选择一个类；
- 接下来组合这些类。

当每个类被至少一个测试用例所覆盖时，就得到了最小数量的测试用例。对本例来说，采用这种方法将得到最少四个测试用例（见图 11.7）。当需要更为彻底的测试时，类之间的组合就必须被测试用例所覆盖。理论上的最大值要求所有特性类的每一种可能的组合，都必须被测试用例所覆盖。但由于这是一件不可能的事情，比如会有逻辑上不一致的测试用例，因而在实际中，大多数情况下可执行的测试用例数量是很小的。在本例中，这意味着  $3 \times 3 \times (1 + (3 \times 2)) = 63$  个测试用例。（特性“预期速度与实际速度之间的差值”与“实际速度”的每一个类组合，都必须与特性“前面汽车”的所有类进行组合。特性“前面汽车”包含两个类，其中“有”类被分为两个特性，分别有三个类和两个类。）

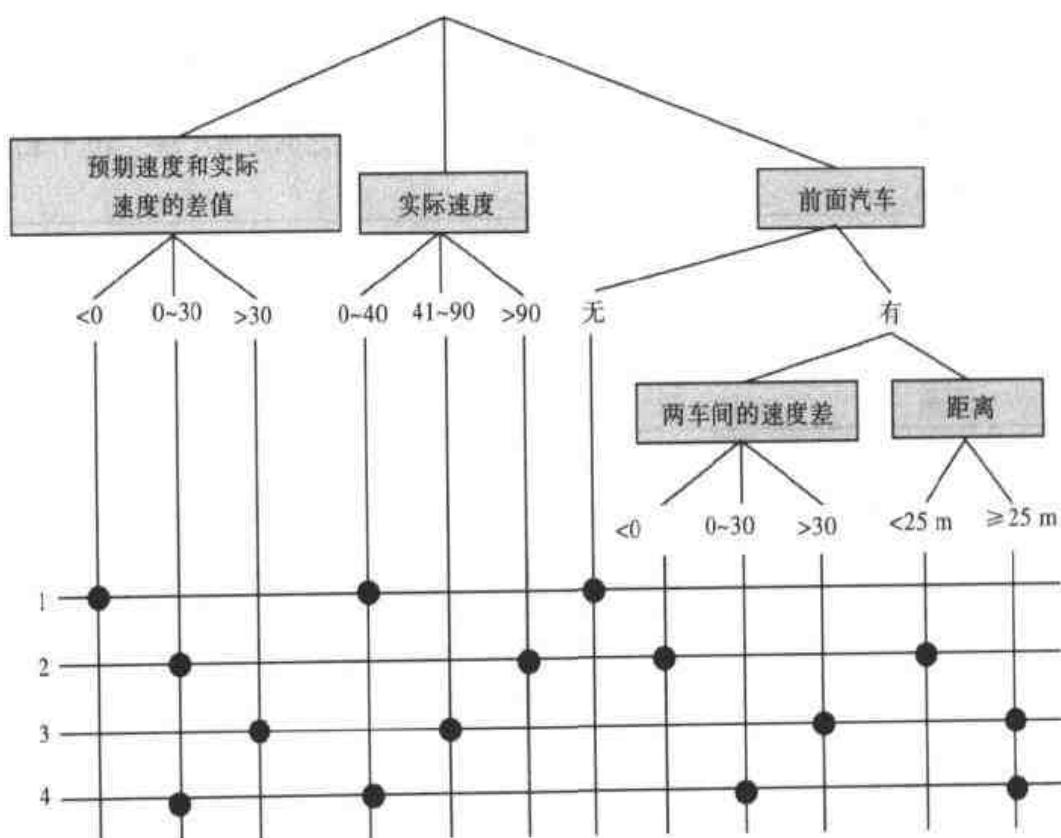


图 11.7 每一行是一个逻辑测试用例。用最少四个逻辑测试用例，可至少覆盖每个类一次

### 组织测试脚本

测试脚本的准备由以下活动组成：

1. 构造物理测试用例；
2. 定义测试动作；
3. 定义检查点；
4. 确定起始情况；
5. 建立测试脚本。

## 构造物理测试用例

类的组合构成了逻辑测试用例。为了执行这些测试用例，就必须为每个对应类选择相应的值。对本例来说，这将生成表 11.18 所示的物理测试用例：

表 11.18 物理测试用例

| 测试用例 | 特性     | 值         | 注释                      |
|------|--------|-----------|-------------------------|
| 1    | 汽车速度   | 39 千米/小时  |                         |
|      | 预期速度   | 30 千米/小时  | 预期速度与实际速度之间的差：-9 千米/小时  |
|      | 前面有无汽车 | 无         |                         |
| 2    | 汽车速度   | 100 千米/小时 |                         |
|      | 预期速度   | 120 千米/小时 | 预期速度与实际速度之间的差：20 千米/小时  |
|      | 前面汽车速度 | 90 千米/小时  | 前面汽车与本车之间的速度差：-10 千米/小时 |
|      | 距离     | 10 米      |                         |
| 3    | 汽车速度   | 65 千米/小时  |                         |
|      | 预期速度   | 135 千米/小时 | 预期速度与实际速度之间的差：70 千米/小时  |
|      | 前面汽车速度 | 105 千米/小时 | 前面汽车与本车之间的速度差：40 千米/小时  |
|      | 距离     | 100 米     |                         |
| 4    | 汽车速度   | 25 千米/小时  |                         |
|      | 预期速度   | 50 千米/小时  | 预期速度与实际速度之间的差：25 千米/小时  |
|      | 前面汽车速度 | 55 千米/小时  | 前面汽车与本车之间的速度差：30 千米/小时  |
|      | 距离     | 30 米      |                         |

表中列出的注释给出了与逻辑测试用例的关系。

## 定义测试动作

测试动作就是预先确定的动作，可能成功也可能失败。对自动化系统来说，惟一的动作就是接通系统。对于与用户有大量交互的系统来说，有许多的测试动作需要完成。在后一种情况下，必须把测试动作描述得十分清楚，因为用户经常会在不同的动作之间做选择。

对本章中给出的例子，仅有的动作就是接通导航控制系统并设定预期的速度。

## 定义检查点

检查点是指对什么时候以及如何来检查系统行为的描述。对每一个测试用例，都要描述预期的系统行为。

对本章中给出的例子，检查点是指汽车速度保持为常量的时刻。要测出汽车速度以及本车与前面汽车的距离。

C01 汽车速度

C02 与前面汽车的距离

例子中，测试用例的预期行为如表 11.19 所示：

表 11.19 测试用例的预期行为

| 检查  | 测试用例 1   | 测试用例 2   | 测试用例 3    | 测试用例 4       |
|-----|----------|----------|-----------|--------------|
| C01 | 30 千米/小时 | 90 千米/小时 | 105 千米/小时 | 50 千米/小时     |
| C02 |          | 40 米     | 45 米      | > 30 米（不断增加） |

与前面汽车的距离是速度的一个函数。只有在前面没有汽车，或者前面车的速度与预期速度相同或更高的情况下，才能够达到预期速度。

#### 确定起始情况

有时候，在测试用例开始执行之前，必须要满足某些前提条件。这些前提条件可能必须有一个特定的初始化数据集，或者系统必须被设定在一个特定的状态。对每一个测试用例来说，这个特定状态和初始数据集可能都不相同。在本章的例子中，不需要有专门的措施来建立起始情况。

#### 建立测试脚本

测试脚本应当提供执行测试用例所需的全部信息。

前面活动的所有交付物构成了组织测试脚本所需的信息。测试脚本首先是描述开始情况，然后按照正确的顺序来描述测试动作与检查点。如果还需要更多的测试设备，那么就要增加相应的信息，或是给出所需设备列表的参考资料。

### 11.5.3 复杂系统

对于复杂系统，有许多特性会影响到系统行为。一个特性的某个类可以派生出另外的特性。在这些情况下，分类树就变得让人难以理解。这时，可以用细化符号来表示分类树的下一级将会在其他地方继续。图 11.8 中给出了这样一个例子。

细化技术可以在不同的级别处理拥有大量特性的测试对象。

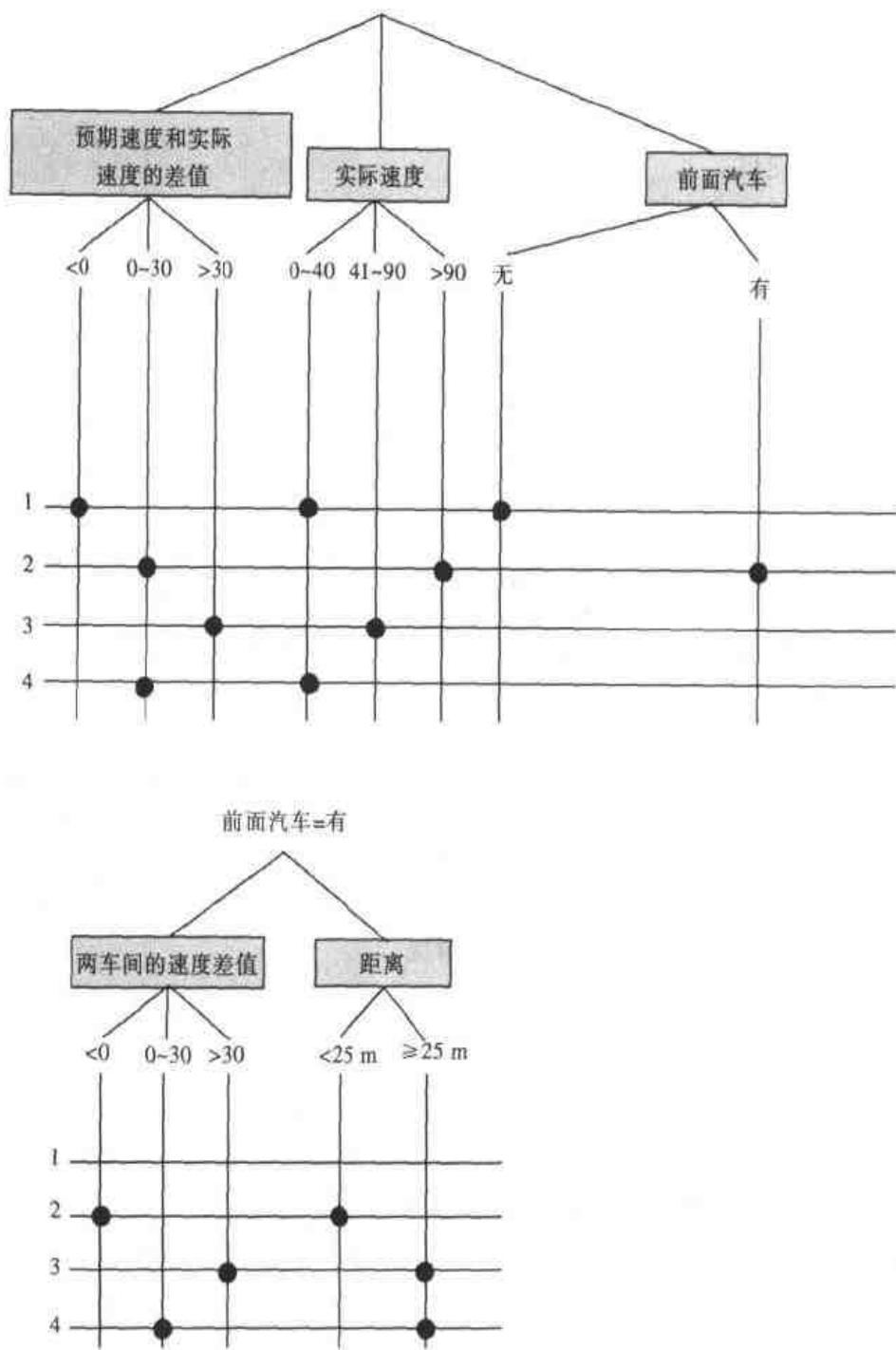


图 11.8 分类树的下一级在另一个图中继续的例子

## 11.6 进化算法

### 11.6.1 介绍

进化算法（也称为遗传算法）是基于达尔文的进化论发展起来的搜索技术和程序。适者生存是这些算法背后的驱动力。进化算法被用于处理最优化问题或可

以转化为最优化问题的问题。在进化算法中，测试设计技术集中于单个测试用例，进化算法处理单个测试用例的“种群”与“适应度”。进化算法可以用于如下应用：查找系统违反其计时约束的测试用例，或是创建一组可以覆盖部分代码的测试用例。进化算法的一些适用范围可见 Wegener and Groditmann (1998), Wegener and Mueller (2001)：

- 安全测试；
- 结构测试；
- 突变测试；
- 健壮性测试；
- 短暂行为测试。

在这些关于进化算法的段落中，只解释了其基本原理。欲了解更多的科学背景知识，建议进一步阅读 Pohlheim (2001) 与 Stamer (1995) 的读物。

### 11.6.2 进化过程

图 11.9 给出了单个种群在一个进化过程中的步骤。进化过程从生成开始种群开始，然后评估每个测试用例的适应度。被测系统被用于确定每个测试用例的适应度。将系统行为与定义的系统最优化行为进行比较，系统行为越接近定义的最优化行为，那么测试用例的适应度就越高。这一步采用适应函数来执行。定义恰当的适应函数并不容易，但非常重要。没有恰当的适应函数，就不可能找出正确的测试用例。

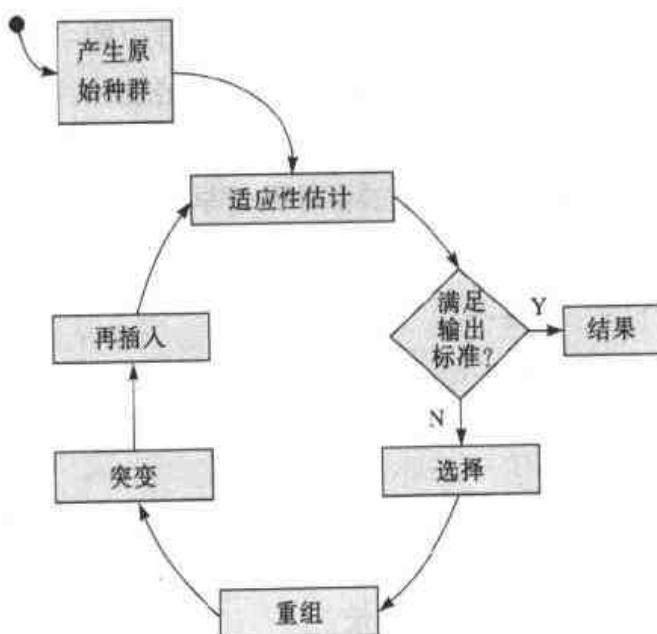


图 11.9 进化过程

如果一个测试用例满足输出标准，那么该过程停止并得到最终结果，可以通过重组与突变步骤来选择其他的测试用例，形成新的测试用例（后代）。在重组步骤中，测试用例的一部分与另一个测试用例相互交换，形成两个新的测试用例。在这步之后，测试用例发生突变，在测试用例中一个随机选择的位置处有变化发生。

新形成的有少许变化的测试用例成为这组测试用例的一部分，这一步被称为插入，原有种群的测试用例被新的测试用例所替代。插入步骤形成的种群被称为新一代。从这时候起又开始新一轮的进化过程，继续适应度评估，直到满足输出标准。

### 11.6.3 过程元素

这一部分详细解释进化过程的不同元素。为了更清晰地描述各个步骤，使用下面的例子来说明。在这个例子中，必须找出四个两位数字（85, 31, 45, 93）的编码。每个测试用例也有四个两位数字。将测试用例加入系统中，如果它符合这个编码，系统将做出反应。系统将在屏幕上用黑圈来表示每个在正确位置的数字。对编码中不在正确位置的每个数字，在屏幕上用白圈来表示。系统并没有给出有关哪些数字是在编码中，或哪些数字在正确的位臵等信息。用适应度函数对系统的反应做出解释。每个黑圈被认为是十个点，每个白圈被认为是三个点，不在编码中（系统没有做出反应）的每个数字被认为是一个点。适应度函数的最大值是 40，这也是输出标准。

#### 开始种群

一个种群是一组测试用例。每个测试用例是种群中的一个个体。随机生成开始种群，开始种群的大小为 50~100 个成员（H. H. Sthamer, 1995）。小型种群的优点在于需要更少的计算量，通常将很快达到最优化，缺点在于可能像自然界中一样存在近亲繁殖。使用小型种群有时候会导致过早收敛，结果就是种群将保持为局部优化。

#### 与“种群”相关的一些定义：

- 一个种群被称为  $P$ ；
- 第  $x$  代为  $P_x$ ；
- 种群大小为  $P_{sz}$ ；
- 开始种群（第一代）为  $P_1$ 。

开始种群的一个例子如表 11.20 所示：

表 11.20 开始种群

| 个体 | $P_1$       |
|----|-------------|
| 1  | 41 03 90 93 |
| 2  | 04 72 95 31 |
| 3  | 44 31 45 93 |
| 4  | 93 45 32 91 |

### 适应度评估

适应度是指测试用例与全局最优化的相关程度。每个测试用例都要计算其适应度。适应度被用于比较测试用例。一个测试用例越接近最优化，那么适应度就越高。适应度通过使用适应度函数来计算。适应度必须能够尽可能确切地来说明问题。要想找出一个恰当的适应度函数，并不总是一件容易的事情。

■ 一个个体的适应度为： $f_i$ ；

■ 种群的适应度为： $f_{total} = \sum_i f_i$ 。

本例中种群的适应度如表 11.21 所示：

表 11.21 种群适应度

| 个体 | $P_1$       | $f_i$ |
|----|-------------|-------|
| 1  | 41 03 90 93 | 13    |
| 2  | 04 72 95 31 | 6     |
| 3  | 44 31 45 93 | 31    |
| 4  | 93 45 32 91 | 8     |
|    | 总适应度        | 58    |

### 选择

在选择阶段过程中，需要选择测试用例来形成新的测试用例。可以随机选择或根据测试用例的适应度来进行选择。如果使用适应度，那么就必须对测试用例进行排序。适应度最高的个体被选择的几率也就最高。在本例中，使用适应度来选择测试用例。

第一步是将测试用例的适应度归一化，即每个测试用例的适应度被种群的总适应度所除：

$$f_{i, \text{norm}} = \frac{f_i}{\sum_{i=1}^{p_n} f_i}$$

这里  $f_{i, \text{norm}}$  是第  $i$  个测试用例的归一化适应度。

下一步是合计测试用例的归一化适应度：

$$f_{i, \text{accu}} = \sum_{k=1}^i f_{k, \text{norm}}$$

这里  $f_{i, \text{accu}}$  是第  $i$  个测试用例的累积归一化适应度。

例子中的归一化适应度与累积归一化适应度如表 11.22 所示：

表 11.22 归一化适应度与累积归一化适应度

| 个体    | $P_1$       | $f_i$ | $f_{i, \text{norm}}$ | $f_{i, \text{accu}}$ |
|-------|-------------|-------|----------------------|----------------------|
| 1     | 41 03 90 93 | 13    | 0.2241               | 0.2241               |
| 2     | 04 72 95 31 | 6     | 0.1034               | 0.3275               |
| 3     | 44 31 45 93 | 31    | 0.5345               | 0.8620               |
| 4     | 93 45 32 91 | 8     | 0.1379               | 1.0                  |
| 总的适应度 |             | 58    | 1.0                  |                      |

接下来用随机生成器来选择测试用例。选择下一个有最高  $f_{i, \text{accu}}$  值的测试用例。有着最高适应度的测试用例被选择的几率也最高，因为其累积归一化适应度的范围最广，为  $(f_{i, \text{accu}} - f_{i-1, \text{accu}})$ 。

例子：如果生成的数字为 0.7301、0.1536、0.9005 与 0.5176，那么测试用例 3、1 和 4 被选择，且用例 3 被再次选择。

### 重组

两个测试用例（双亲）被重组以生成两个新的测试用例（后代）。在这个过程中发生交叉。交叉是指测试用例的一部分被相互交换，依据交叉概率  $P_c$  来进行交叉。交叉有三种实现：

- **单一交叉**，交叉发生在测试用例的一个位置。随机生成器决定双亲分裂的位置。在产生后代的过程中，双亲的第二部分发生变化。后代现在成为双亲的一个组合，概率由交叉概率  $P_c$  来确定。
- **双交叉**，交叉发生在两个位置，这两个位置将双亲分为三部分。

均衡交叉，发生变化位置的数目和位置本身是由随机生成器决定的，所以最大交叉率等于测试用例的元素数目。

本例中，重组结果如表 11.23 所示。

表 11.23 重组结果

|    |    |    |    | <i>f</i> |    |    |    |           |
|----|----|----|----|----------|----|----|----|-----------|
| 44 | 31 | 45 | 93 | 44       | 31 | 90 | 93 | 22        |
| 41 | 03 | 90 | 93 | 41       | 03 | 45 | 93 | 22        |
| 31 | 85 | 32 | 91 | 31       | 85 | 32 | 91 | 17        |
| 44 | 31 | 45 | 93 | 44       | 31 | 45 | 93 | <u>22</u> |
|    |    |    |    | 83       |    |    |    |           |

### 突变

突变是指测试用例中一个元素的随机变化。变化可能引起下一个值增大或减小，或被随机值替代。突变可能为测试用例的一个元素引入在开始种群中不存在的值。最优突变率 ( $P_m$ ) 是测试用例元素数量的倒数 (Sthamer, 1995)。

本例中突变率为  $P_m=0.25$ ，即只有四分之一元素突变。用随机生成器来确定哪一个元素将发生突变，还有突变是增加还是减少。结果如表 11.24 所示。发生突变的元素在表中用黑体字来表示。

表 11.24 突变结果

|             |         |             |
|-------------|---------|-------------|
| 44 31 90 93 | 0.637 - | 44 31 89 93 |
| 41 03 45 93 | 0.352 + | 41 04 45 93 |
| 31 85 32 93 | 0.712 + | 31 85 33 93 |
| 44 31 45 91 | 0.998 - | 44 31 45 90 |

### 重新插入

并不是所有新形成的测试用例都将在下一个周期成为种群的一部分。新形成的测试用例存活与否，依赖于它们的适应度与重新插入率。重新插入率 ( $P_i$ ) 是指后代生存的概率。

本例中，新形成的测试用例的适应度如表 11.25 所示。

表 11.25 新形成的测试用例的适应度

| 个体 | $O_1$       | $f_i$ | $f_{i,\text{norm}}$ | $f_{i,\text{accu}}$ |
|----|-------------|-------|---------------------|---------------------|
| 1  | 44 31 89 93 | 22    | 0.2651              | 0.2651              |
| 2  | 41 04 45 93 | 22    | 0.2651              | 0.5302              |
| 3  | 31 85 33 93 | 17    | 0.2048              | 0.7350              |
| 4  | 44 31 45 90 | 22    | 0.2651              | 1.0                 |
|    | 总的适应度       | 83    | 1.0                 |                     |

在这个例子中，生存率 ( $P_s$ ) 为 0.25，即在新种群中只存在一个测试用例。

例子：如果生成的随机数为 0.8193，那么测试用例 4 将被插入到种群中。

在父种群中，测试用例的生存与否也依赖于它们的适应度。适应度最低的测试用例，其退出种群的概率也就最高。为了确定生存与否，需要计算适应度的倒数。

本例中  $P_1$  的适应度倒数函数  $f_{i,\text{rec}}$  定义为  $1/f_{i,\text{norm}}$ 。初始数据集（父种群  $P_1$ ）的适应度倒数 ( $f_{i,\text{rec}}$ ) 计算值如表 11.26 所示。它们是由来自选择步骤（见表 11.22）的适应值  $f_{i,\text{norm}}$  算出来的。接下来这些值被归一化 ( $f_{i,\text{recnorm}}$ ) 并累积 ( $f_{i,\text{recaccu}}$ )。最后一列用来随机决定哪个测试用例应该从初始集中删除。

表 11.26 父种群  $P_1$  的适应度倒数值（归一化和累积）

| 个体 | $P_1$       | $f_{i,\text{rec}}$ | $f_{i,\text{recnorm}}$ | $f_{i,\text{recaccu}}$ |
|----|-------------|--------------------|------------------------|------------------------|
| 1  | 41 03 90 93 | 4.4623             | 0.1919                 | 0.1919                 |
| 2  | 04 72 95 31 | 9.6712             | 0.4159                 | 0.7221                 |
| 3  | 44 31 45 93 | 1.8709             | 0.0804                 | 0.8025                 |
| 4  | 93 45 32 91 | 7.2516             | 0.3118                 | 1.0                    |
|    | 总的适应度       | 23.2560            | 1.0                    |                        |

例子：如果生成的随机数为 0.4764，那么测试用例 2 退出种群。后代中的测试用例 4 替代了父种群中的测试用例 2。新种群的适应度  $P_2$  如表 11.27 所示：

表 11.27 新种群适应度

| 个体 | $P_2$       | $f_i$ |
|----|-------------|-------|
| 1  | 41 03 90 93 | 13    |
| 2  | 44 31 45 90 | 22    |
| 3  | 44 31 45 93 | 31    |
| 4  | 93 45 32 91 | 8     |
|    | 总的适应度       | 74    |

$P_1$  的总适应度为 58。所以，进化增加了种群的适应度。

在重新插入后，计算测试用例的适应度。如果有一个测试用例满足输出标准，那么就找到了最优化的测试用例，进化过程终止。否则继续下一个周期的进化过程。

#### 11.6.4 测试基础设施

只有与自动化测试相结合时，进化算法才有用。测试用例的数量，以及需要得到优化结果的周期次数，都使得不可能手工来执行测试用例。这意味着进化算法必须与自动化测试包相连。自动化测试包执行被测系统的所有测试用例，并且将被测系统的输出结果提供给进化算法。图 11.10 给出了进化算法与测试包相连接的位置。

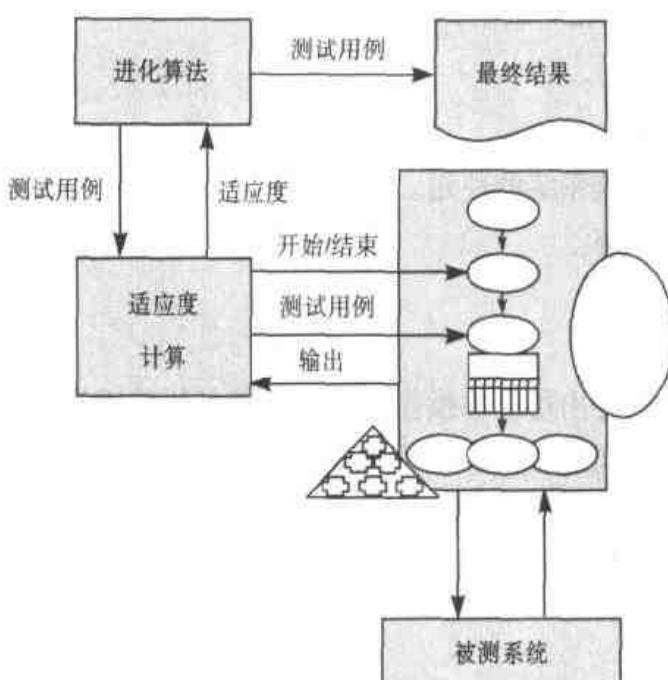


图 11.10 进化算法与测试包

自动化测试包的细节和测试包与被测系统之间的通信将在 14.1 节中讲述。进化算法与测试包之间通过适应度计算来相互通信：适应度计算功能块按照正确的格式将测试用例发送给测试包，并通过测试包将被测系统的输出返回。这个输出被用于计算测试用例的适应度，适应度又被送回给进化算法。

---

## 11.7 统计使用测试

### 11.7.1 介绍

现实中测试的目的是检测系统运行使用时的缺陷，因而测试应当尽可能地模

拟真实情况。一种描述系统实际使用情况的方法称为操作分布图，它是系统如何被使用的定量描述。Musa (1998) 引入了操作分布图概念，作为软件可靠性工程的一部分。在 Musa 研究的基础上，Cretu (1997) 和 Woit (1994) 又推进了该技术的发展。统计使用测试利用操作分布图来生成一组与统计相关的测试用例。

统计使用测试关注的是最常使用的操作，它所关注的重点是系统通常使用的情况。操作分布图用于可描述为状态机的系统，或有着基于状态行为的系统，它描述与系统所处状态相关的输入概率，然后就可以基于这些概率分布来准备测试用例。

11.7.2 节描述了如何来准备操作分布图，以及如何使用它们来导出测试用例。

11.7.3 节讲述了自动化测试用例的生成。

## 11.7.2 操作分布图

操作分布图处理的是系统的使用。系统的用户可以是一个人，也可以是触发被测系统的另外一个系统。

### 操作分布图分析

为了用结构化方式导出可靠的操作分布图，就必须采用自顶向下的方法。这种方法首先是确定出不同的客户群，然后进一步细分这些客户群，直至可以描述出操作分布图。因此这将导致一个树形结构，其根部是系统，底部是功能分布图。

可以按照下面五个步骤来开发操作分布图：

1. 区分客户；
2. 区分用户；
3. 区分系统模式；
4. 区分功能分布图；
5. 区分操作分布图。

### 客户分布图

客户是拥有系统的人。这一步骤的目标是将客户划分为客户群，每个客户群或多或少有着相同的行为和相同的理由来购买系统。下一步是计算每个客户群占总客户群的百分比。另一种方式是确定不同客户群的相对重要性。通过这种做法，可以基于相对重要性，而不是基于相对客户群的大小来区分测试工作。客户群的大小与重要性并不一定相关。

### 用户分布图

用户是使用系统的人或（子）系统。用户分布图描述系统的使用和客户群使用的百分比，也可以用大致估计出的这个分布图的相对重要性来代替百分比。

### 系统模式分布图

系统模式是一组相互有关联的功能。

### 功能分布图

每个系统模式被分为功能，需要准备出功能列表并确定每个功能发生的概率。在准备功能列表时应当使功能的定义能够表示为一个个实质性的不同任务。

### 操作分布图

对每个功能分布图，需要创建一个操作分布图，操作分布图描述的是功能分布图的使用。

### 操作分布图描述

操作分布图从数量上描述系统如何被一定数量的用户所使用。它通过问如下问题完成这个任务：什么时候系统处于这种状态？用户触发下一个事件的可能性有多大？具体来说，操作分布图说明了每个“状态”和“下一事件”组合的概率。Woit (1994) 建议用“系统历史”代替“状态”，因为用户行为不仅依赖于系统的并发状态，也依赖于之前所发生的事件。这样描述系统的实际使用状况可以更灵活，而且仍然可以用来描述“初始操作分布图”。本章采用具体的方法。

必须执行以下步骤，来为每个功能分布图确定操作分布图：

1. 确定系统的可能状态。如果某种原因使得系统历史很重要，那么就必须确定状态顺序。这将生成状态与历史的一个列表，新定义历史的区别只在历史中为事件定义了不同概率时有用。在这个列表中不考虑不相关事件的状态。
2. 确定可能事件的列表。
3. 确定每个事件组合的概率。

可以用一个矩阵来表示组合信息（见表 11.28）。

表 11.28 用于操作分布图的表示技术

|         | 事件 1      | 事件 2      | ..... | 事件 $m$    |
|---------|-----------|-----------|-------|-----------|
| 历史类 1   | $P_{1,1}$ | $P_{1,2}$ | ..... | $P_{1,m}$ |
| 历史类 2   | $P_{2,1}$ | $P_{2,2}$ | ..... | $P_{2,m}$ |
| .....   | .....     | .....     | ..... | .....     |
| 历史类 $n$ | $P_{n,1}$ | $P_{n,2}$ | ..... | $P_{n,m}$ |

### 操作分布图规范

为了说明如何将已有的信息转化为操作分布图规范，这里使用了一个简单的 VCR 操作分布图的例子。VCR 有六个历史类，最后一个历史类是一个关于系统历史如何影响操作分布图的例子。如果磁带的位置超过其长度的 75%，那么事件概率就会发生显著的变化。VCR 用一个计数器来指出磁带的位置。对于这个磁带来说，最大位置为 20 000。所以，如果磁带位置超过 15 000，那么事件概率就会发生变化（可见表 11.29）。除 evStop 之外的所有事件都有一个变量，该变量是集合  $X$  中的一个成员， $X$  所包含的值为 50, 100, ..., 20000。

表 11.29 VCR 操作分布图的例子。每一列是所有可能的事件，  
每一行是系统历史类。对每一行来说，概率总和为 1

|                  | evStop | evRewind                 | evPlay                   | evFastforward            | evRecord                 |
|------------------|--------|--------------------------|--------------------------|--------------------------|--------------------------|
|                  |        | $\langle a \rangle :: X$ | $\langle b \rangle :: X$ | $\langle c \rangle :: X$ | $\langle d \rangle :: X$ |
| 待机               | 0      | 0.2                      | 0.5                      | 0.2                      | 0.1                      |
| 倒带               | 0.3    | 0                        | 0.6                      | 0.1                      | 0                        |
| 播放计数器 < 15 000   | 0.5    | 0.3                      | 0                        | 0.2                      | 0                        |
| 快进               | 0.5    | 0.1                      | 0.4                      | 0                        | 0                        |
| 录音               | 1      | 0                        | 0                        | 0                        | 0                        |
| 播放 计数器 => 15 000 | 0.8    | 0.15                     | 0                        | 0.05                     | 0                        |

必须将操作分布图转化为规范。可以使用诸如 C、C++ 或 Java 等高级编程语言来生成测试用例。

可以用转换函数来描述历史类。对每一个概率不为零的历史与事件的组合，都要定义一个转换函数。所以，转换函数的总量小于或等于历史与可能事件的乘积。转换函数是对事件与最终状态的描述。

每个历史类都有一个标识，该标识的范围为 1 到  $n$ 。每个事件也有一个标识，该标识的范围为 1 到  $m$ 。

每个事件可以有属于一个特定集合的一个或多个变量。这个集合是操作分布

图规范的一部分。如果变量不受系统历史的影响，那么可以为该变量随机选择一个值。在这个例子中，如果磁带已经处于位置 10 000，那么不可能将磁带再快进 15 000 个位置。这表示变量的取值要受到某些条件的约束。这些条件被转化为条件函数（见图 11.11）。

```
init: Rewind till tape at start position
 counter = 0
 max = 20000

F1,2: if counter = 0
 return 1
 else
 counter = RewindCondition(counter) + counter
 return 2

F1,3: if counter = max
 return 1
 else
 counter = PlayCondition(counter) + counter
 if counter >= 15000
 return 6
 else
 return 3

F1,4: if counter = max
 return 1
 else
 counter = FastforwardCondition(counter) + counter
 return 4

F1,5: if counter = max
 return 1
 else
 counter = RecordCondition(counter) + counter
 return 5

F2,1: return 1

etc.

Set: X={50, 100, ..., 20000}

RewindCondition (in: counter)
```

```
initialize random generator
rand = random generator
rank = (1 + rand div 0,0025)* 50
while rank > counter
 rand = random generator
 rank = (1+ rand div 0,0025)*50
end while
return -rank

PlayCondition (in: counter)
 return Cond(counter)

FastforwardCondition (in: counter)
 return Cond(counter)

RecordCondition (in: counter)
 return Cond(counter)

Cond (in: counter)
 initialize random generator
 rand = random generator
 rank = (1 + rand div 0.0025) * 50
 while rank + counter => max
 rand = random generator
 rank = (1+ rand div 0.0025) * 50
 end while
 return rank
```

X 的步长为 50，这表示共有  $20\ 000/50 = 400$  个不同的值。  
 $1/400 = 0.0025$ 。所以，对随机生成器来说，每一个 0.0025 步长都表示 X 中的下一个数。

图 11.11 VCR 操作分布图的伪代码

这个规范包含了足够的信息来生成自动化的测试用例。

测试用例生成器一开始为历史类 1(待机)。用一个随机生成器来选择一个列，以及与该组合相关的转换函数，现在，测试用例生成器知道了最终状态，并再次选择一个列。一直持续该过程直至满足停止标准。每一个结果就是一个测试用例。所有的测试用例被存储在一个文件中。在过程期间，记录所选中的组合，并计算选择的频率。

### 11.7.3 测试用例的生成

操作分布图规范采用高级编程语言来编写，以便生成测试用例。测试用例生成器产生由初始状态开始的一系列的状态与事件。生成的顺序基于操作分布图的概率。图 11.12 给出了一个用伪代码编写的测试用例生成器的例子。F 只有在进行过可靠性估计之后才有必要。

```
P[1...n, 1...m] = probability matrix operational profile cumulated per row
CaseLength = length of test case
AmountTestCase = number of test case needed
F[i...n, 1...m] = frequency of combination in test cases
TestCase(1...AmountTestCase, 1...CaseLength)
rand = random number

Initialize random generator

amount = 0

while amount < AmountTestCase
 Init ()
 amount ++
 length = 0
 hisstat = 1
 While length < CaseLength
 rand = random generator
 Event = SelectEvent(hisstat, rand)
 F(hisstat, event) ++
 hisstat = ExecuteFunction(hisstat, Event)
 Testcase(amount, length) = Event & ", " & hisstat
 length ++
 end while
end while
Testcase to file
F to file
```

图 11.12 测试用例生成器的伪代码

现在已经得到了测试用例，那么可以开始执行测试了。对大型系统而言，这意味着将生成大量的测试用例，那么是否可以让这些测试用例自动执行呢？

对此进行可靠性估计是有可能的 (Woit, 1994)。为了进行可靠性估计，有必要知道测试用例中转换的频率。Woit 已经采用 C 语言描述了一个包含可靠性估计

的测试用例生成器。

## 11.8 稀有事件测试

### 11.8.1 介绍

统计使用测试技术“忽略”了很少发生的事件。但往往这些事件需要由被测系统正确地处理，这是非常重要的，这些事件被称为稀有事件。稀有事件通常与系统的安全特性或系统的主要功能相关。稀有事件不处理系统的故障。

稀有事件测试就像统计使用测试一样，是在接近发布日期的时候执行。如果系统的主要功能是处理稀有事件，例如对于在路面光滑的条件防止汽车轮胎侧滑的 ABS 系统来说，这种稀有事件的处理必须在过程的前期就进行。

### 11.8.2 确定稀有事件测试用例

确定稀有事件测试用例的前面一些步骤，与统计使用测试中的做法类似。

#### 客户分布图

在这里，客户是指购买系统的人。这一步骤的目标是确定客户群，每个客户群是出于不同的理由或不同的使用目的来购买系统。只有在影响到稀有事件或稀有事件的频率时，确定不同的客户群才有意义。

#### 用户分布图

用户是使用系统的人或（子）系统。用户本身可以引入稀有事件，或通过使用系统而遇到稀有事件。

#### 系统模式分布图

系统模式是一组相互有关联的功能。

#### 功能分布图

每个系统模式被分为功能。如果已经知道不再有与这个分布图相关的稀有事件，那么将系统模式分布图再进一步分解就没有什么用处。

#### 操作分布图

只有在稀有事件发生的情况下，才对于每个功能分布图相应生成一个操作分

布图。一个操作分布图是一个系统历史和事件相关的矩阵。每一个历史与事件的组合都有一个概率，这里真正重要的是那些发生概率非常小的组合。操作分布图是用于稀有事件测试用例规范的基础。

如果在统计使用测试中已经确定了操作分布图，那么就有可能复用这些分布图。要注意的是，由于事件在统计使用测试中极少发生，因而可能被忽略（概率值为0）。对概率值为0的事件，最好重新评估。

需要选择所有的发生概率很低的历史/事件组合，每个组合是一个测试用例，这样一个测试用例的测试脚本包含下面的内容：

- **前提条件：**如何将系统置于正确的状态。历史状态揭示出系统必须处于哪个状态。如何将系统置于正确状态的信息也可以从操作分布图中得到。前提条件有时候是一个状态/事件组合列表，按照顺序到达该测试用例所需的状态。有时候，在系统环境中有必要采取其他的措施，来引发稀有事件。
- **稀有事件的描述：**必须有一个如何引发稀有事件的描述。
- **最终状态：**描述系统的最终状态。有时候，有必要描述如何来检查该状态。

### 11.8.3 彻底性

每个测试用例表示一个历史与一个稀有事件的组合。为了增强测试的彻底性，可以测试一组级联的稀有事件。这只有当下述情况发生时才有可能：一个稀有事件导致系统的一个历史状态，在此状态中另一个稀有事件才有可能发生。

---

## 11.9 突变分析

### 11.9.1 介绍

采用突变分析的目的是检测出产品制造时的故障。故障播种是指在被测系统中引入故障，系统测试人员并不知道这些故障的存在。在测试集合被执行以后，可以计算出故障检测率，其值是检测出的故障数除以引入的故障数。而在模块或类层次上，是以一种完全不同的方式来进行突变分析。被称为突变的故障在一行代码中被引入，每一个突变在语法上都是正确的，每个模块或类只引入一个突变。一旦引入一个突变，那么含有突变的模块或类的程序就要被编译——该程序被称为一个变体。用测试集合来测试这个变体，在一组变体被测试完成之后，计算出故障检测率，这个故障检测率是被检测出的变体数除以总变体数。

测试的一个目的是得出被测系统的置信度。通过使用测试用例来执行系统并

检查输出结果，就可以完成这一目的。测试就是做出选择，不仅仅是选择什么是很重要的，而且要选择应当使用哪种测试设计技术。如果选择的是用非正式的测试设计技术，那么就必须做出决策，应当准备哪些测试用例。所有这些决策都不是按照正式的方式来做出的。基于这种方法导出的测试用例的置信度有可能会蒙骗人，具有很大的危险性。

可以用突变分析方法来获得测试集合质量中的置信度。采用这种方法，可以度量出测试集合的故障检测率。可以将这种方法在系统层次上应用，也称为故障播种，它也可以在模块或类层次上应用。

该方法的基本思想是测试集合的故障检测率是测试质量的一个指数。故障检测率高的测试集合可以给出被测系统质量的可信度量，或者可以很好地给出系统中没有被检测出的故障数的一个指数。因此，它可以表示出被测系统的质量或系统的置信度。

这种方法的好处与有效性人们还在争论。这种方法并不会对被测系统的质量有任何直接的贡献，而只是给出了测试集合的一个质量指数。如果用的是正式的测试设计技术，那么采用突变分析是否有意义就值得怀疑了。

### 11.9.2 故障播种

故障播种被用于系统层次的测试。采用故障播种，要执行下面的步骤：

1. 引入故障：在被测系统中引入一些故障，这些故障在系统中的分布与测试策略的风险分析相关。必须是由某个对被测系统有深刻了解的人来引入故障，但必须避免由于引入故障而掩盖其他故障的可能性。
2. 测试：系统进行测试时要包含引入的故障。测试小组必须根据其测试设计来测试系统，他们并不知道所引入的故障，而是执行完整的测试。
3. 分析：从缺陷列表中去掉引入的故障并统计。将检测出的故障数除以引入的故障数，得出故障检测率。
4. 删除故障：在产品发布之前，要从源代码中删除引入的故障。

### 11.9.3 突变分析

在模块或类层次上执行突变分析。当依照测试集合得出模块或类的功能正确时，就可以做出测试已经充分的结论。按照以下步骤来进行突变分析：

1. 产生变体：将模块或类修改一行代码，形成原有模块或类的一个变体。采用称为突变运算符的规则来生成变体。这些突变运算符依赖于所用的编程语言，因为每一种编程语言都有各自的故障种类。例如，对 C 语言来说，

就有算术运算符移位、二进位运算符移位、语句删除等 (Untch, 1995)。对 Java 语言来说，有兼容引用类型运算符、构造函数运算符和改写方法运算符等 (Kim 等, 1999)。所有的变体都必须语法正确。引入突变可以迫使出现被零除、死锁或死循环等情况。被零除可以自己消除，限定执行时间可以消除其他两种情况。如果变体的执行时间超过某个限定的时间，那么这个变体就是一个错误的程序。

2. 测试变体：用原有的测试集合来测试这些变体。如果检测到突变，或是已经执行完全部的测试，则停止对变体的测试。
3. 分析：一个变体与原有的模块或类之间存在相同的行为，总是有可能的。这些变体被称为等价体，它们是无法检测的。可以通过被检测出的变体数除以总变体数与等价变体数之间的差，来计算故障检测率。

这种方法的精确应用将导致出现大量的变体，都要用测试用例来进行测试。Untch 指出 1 个等价体可以等于 71 个突变 (Untch, 1995)。处理这样的大量变体，惟一方法是采用自动化测试工具，以及可以自动引入突变的工具。MOTHRA 软件测试环境就是为此目的而开发出来的工具 (Hsu 等, 1992)。另外一种办法是采用由 Untch 描述的生成和测试变体的方法 (Untch, 1995)。他建议采用一种方法，每一行代码都被重写为一行包含变体运算符的代码。他使用的一种工具可以读出这些代码，并将它们转化为带有突变的程序代码。

# 第 12 章 审查清单

## 12.1 介绍

审查清单被用做一种技术，它按照正式方式给出与测试过程相关的各个特性的状态信息。这一章包含下列审查清单类别：

- 每个质量特性的审查清单；
- 高层次测试的一般审查清单；
- 低层次测试的一般审查清单；
- 测试设计技术审查清单；
- 测试过程审查清单。

每个质量特性的审查清单被用于评估系统是否满足相关质量特性的需求。如果审查清单中有一个问题的回答为否，那么就表示存在一个缺陷。对于测试设计技术审查清单，检测出来遗漏项就需要采取行动来弥补遗漏的信息，或是决定略过该设计技术而采用其他的设计技术。

其他审查清单被用于支持测试经理。如果审查清单中有一个问题的回答为否，那么测试经理就要采取行动来解决该问题，或采取措施来弥补遗漏项。这些审查清单也是测试经理需要记住的要点列表，同时也被用做一种与产品权益人沟通的手段。

## 12.2 每个质量特性的审查清单

这一章描述质量特性的审查清单，它主要用于静态测试。该清单中除质量特性以外，还包括用户友好性。可以用动态测试和调查问卷方式来测试质量特性。每个清单包含描述成例子的特性。这些清单被用做准备定制清单的基础，以用于组织、项目和/或产品的特殊需求。审查清单涉及到下列质量特性：

- 连通性；
- 可靠性；
- 灵活性；

- 可维护性；
- 可移植性；
- 可复用性；
- 安全性；
- 可测性；
- 可用性。

### 12.2.1 连通性

连通性是指一个系统与其他系统，或在分布式系统中建立连接的能力。

#### 审查清单

- 用到一般参考模型了吗？
- 是否描述了要采取哪些措施来防止错误的输入与动作？
- 是否描述了如何实现故障恢复？
- 系统之间的交互描述清楚吗？
- 是依据国家（国际）标准或工业标准来设计连通性的吗？
- 用到了标准机械接口吗？
- 可以对其他关键功能或基本功能加以保护吗？
- 是否描述了要连接的系统？
- 是否已制定并使用了通用的接口？
- 数据交互制定标准了吗？
- 硬件与基础设施部件之间的连接制定标准了吗？

### 12.2.2 可靠性

可靠性是指在指定条件下，系统的使用维持在一个指定性能级别的能力。质量特性的可靠性被分为三个子属性：完备性、容错性与可恢复性。

#### 完备性

完备性是指系统中出现故障而系统不会失效的能力。

#### 审查清单

- 采用国家（国际）或工业标准了吗？
- 输入、输出与处理是分别实现的吗？

- 用到检查点与重启工具了吗?
- 实现双处理了吗?
- 数据处理被分为子事务吗?
- 实现看门狗 ( watchdog ) 了吗?
- 它是一个分布式系统吗?
- 程序模块被复用了吗?
- 软件参数化了吗?
- 算法优化了吗?

### 容错性

容错性是指系统中出现故障或不兼容指定的接口时，系统维持在一个指定的性能级别的能力。

#### 审查清单

- 实现看门狗了吗?
- 实现双处理了吗?
- 实现一致性检查了吗?
- 实现重启工具了吗?
- 实现自动防故障条件了吗?
- 在系统执行期间是否明确检查了前提条件常量?
- 实现故障标识了吗?
- 核心功能是存储在不同的模块中吗?
- 它是一个分布式系统吗?

### 可恢复性

可恢复性是指在故障情况下，系统重新恢复到一个指定的性能级别，并恢复受故障直接影响的数据的能力。

#### 审查清单

- 用到国家 ( 国际 ) 或工业标准了吗?
- 是否实现了恢复系统?
- 是否实现了内置的自动测试?
- 是否实现了周期性的快速重启工具?
- 实现看门狗了吗?
- 实现双处理了吗?

- 实现重启工具了吗?
- 实现一致性检查了吗?
- 它是一个分布式系统吗?
- 实现异常处理了吗?

### 12.2.3 灵活性

灵活性是指用户可以在无须改变系统本身的情况下，对系统加以扩展或修改的能力。

#### 审查清单

- 软件参数化了吗?
- 使用的是否是逻辑而不是固定编码值?
- 数据处理与数据检索是分开的吗?
- 可以为了改变系统功能修改输入功能吗?
- 可以为了改变系统功能修改控制功能吗?
- 可以为了改变系统功能修改处理功能吗?
- 可以为了改变系统功能修改输出功能吗?
- 可以改变用户对话框吗?
- 实现扩展槽了吗?

### 11.2.4 可维护性

可维护性是指系统可被修改的能力。这里的修改是指根据环境、要求和功能需求的变化，而对系统进行修正、改进或改造。

#### 审查清单

- 采用国家(国际)或行业标准了吗?
- 制定开发标准了吗?
- 使用看门狗了吗?
- 数据处理划分为子事务了吗?
- 输入、输出与处理是分开实现的吗?
- 基本功能是存储在不同的模块中吗?
- 描述系统分解结构了吗?
- 有开放、一致而且最新的技术文档吗?
- 软件参数化了吗?

- 它是分布式系统吗?
- 它是实时系统吗?
- 算法被优化了吗?

### 12.2.5 可移植性

可移植性是指软件从一个环境转移到另一个环境的能力。

#### 审查清单

- 软件参数化了吗?
- 采用国家(国际)或行业标准了吗?
- 采用标准机械接口了吗?
- 机器相关性是在不同模块中实现的吗?
- 算法被优化了吗?
- 子系统是分布式的吗?

### 12.2.6 可复用性

可复用性是指软件可以复用部分系统或设计，以开发其他系统的能力。

#### 审查清单

- 采用国家(国际)或行业标准了吗?
- 软件参数化了吗?
- 数据处理划分为子事务了吗?
- 输入、输出与处理是分开实现的吗?
- 机器相关性是在不同模块中实现的吗?
- 采用了标准机械接口吗?
- 程序模块被复用了吗?
- 算法被优化了吗?

### 12.2.7 安全性

安全性是指系统确保只有经授权的用户(或系统)才可以访问系统功能的能力。

#### 审查清单

- 是否实现了身份识别、身份验证、授权、记录日志及报告等功能?
- 安全功能与系统的其他部分物理上是分开的吗?

### 12.2.8 可测性

可测性是指系统能够被验证的能力。

#### 审查清单

- 是否完成了测试件，并保留下以用于以后的测试？
- 是否使用了计划与缺陷管理工具？
- 有开放、一致而且最新的功能文档吗？
- 采用开发标准了吗？
- 使用看门狗了吗？
- 输入、输出与处理是分开实现的吗？
- 机器相关性是在不同模块中实现的吗？
- 基本功能是存储在不同的模块中吗？
- 程序模块被复用了吗？
- 描述系统分解结构了吗？
- 子系统是分布式的吗？
- 实现多处理了吗？
- 算法被优化了吗？

### 12.2.9 可用性

可用性是指在指定的条件下，系统能够被用户理解、学习、使用并吸引用户的能力。

可用性是一个相当主观的质量特性，系统有什么样的可用性依赖于用户的个人喜好。然而，通常可以根据需求不同来确定出不同的用户群。另一个经常被忽略的因素是文化，它也会影响到用户的接受程度。要描绘清楚系统的可用性，惟一的办法就是让一个用户在日常实践中使用系统。如果认为文化是一个重要因素，那么用户论坛就应当包含不同的文化背景或使用更多的论坛。

可以采用让用户论坛中的成员来填写调查问卷的形式，来了解系统的可用性。调查问卷的可能项包含：

- 对用户界面有什么意见？
- 有没有负面评论？
- 清楚用户界面中使用的所有的项和信息吗？
- 有没有帮助并使它成为可执行的任务？

- 整个用户界面中采用的是同一个标准吗？
- 所有的任务都易于操纵吗？
- 利用用户控制工具，通过用户界面导航容易吗？

除了调查问卷方式以外，也可以在系统使用期间采用其他的衡量标准。如果该措施与定义的某些需求相关，那么可以给出有关可用性更具体的想法。可以采取下列衡量标准：

- 新用户执行一个特定任务需要花费多长时间？
- 新用户在执行一个特定任务期间会发生多少个错误？
- 新用户执行系统任务需要使用多少次文档？
- 有经验的用户在一个特定时间段内可以执行多少任务？
- 失败与成功的任务比例是多少？
- 执行每个任务平均需要使用多少个命令？
- 命令数是否超过了给定的最大值？
- 对新用户来说，失败与成功的比例是如何随时间变化而变化的（成为有经验的用户需要花费多长时间）？
- 用户失去对系统的控制有多少次？
- 某一任务的执行时间超过给定的最大值了吗？

## 12.3 高层次测试的一般审查清单

### 12.3.1 概要

- 有没有充分、清楚地描述各种文档之间的关系？
- 所需文档都有吗？
- 系统划分为子系统已经描述了吗？
- 所有的接口都描述了吗？
- 所有的功能与占位功能都描述了吗？
- 描述对话框的设计了吗？
- 描述错误处理了吗？
- 每个功能输入与输出都描述了吗？

### 12.3.2 划分为子系统与接口

- 在所选子系统的描述中，包含需要执行的功能、所需的过程及数据的概要

描述了吗？

- 实现顺序已经说明了吗？
- 接口的位置已经说明了吗？
- 接口有描述吗？
- 物理构造有描述吗？

### 12.3.3 功能结构

- 所有的功能已经用诸如数据流程图的方式来显示了吗？
- 功能的简短描述已经给出了吗？
- 执行所需的条件及频率已经列出了吗？
- 安全需求已经描述了吗？
- 在数据的正确性与完备性框架内所采取的措施已经描述了吗？例如：
  - 输入确认；
  - 冗余输入；
  - 重复输入；
  - 程序对数据处理结果的检查。
- 性能需求已经描述了吗？
- 该功能与其他功能相比的相对重要性已经描述了吗？
- 有数据流的简要描述吗？
- 在功能和输入输出流之间有交叉引用吗？
- 是否已经列出并描述了所有的输入来源？
- 是否已经列出并描述了所有的输出目的地？
- 数据流程图中是否包含了所有的基本功能？
- 处理已经描述了吗？

### 12.3.4 用户对话框

- 有功能与用户对话框之间关系的概述吗？
- 用户对话框的结构已经描述了吗？
- 用户对话框遵循应用指导准则吗？
- 用户对话框已经描述了吗？
- 用户对话框的输出已经描述了吗？
- 用户输入设备的使用已经描述了吗？
- 用户对话框的布局描述了吗？

### 12.3.5 质量需求

- 质量需求指定了吗?
- 质量需求是量化、可度量的吗?
- 质量需求可以被用做验收标准吗?

---

## 12.4 低层次测试的一般审查清单

### 12.4.1 概要

- 各文档之间的关系已经充分并清楚地描述了吗?
- 所有需要的文档都是可用的吗?
- 系统结构已经描述了吗?
- 有程序描述吗?

### 12.4.2 系统结构

- 是否显示了子系统的系统结构? 这是程序划分的基础。

### 12.4.3 程序划分

- 所有的程序已经按照系统流的形式来显示了吗?
- 对系统流中所有的程序, 下列各项已经描述了吗:
  - 程序名称
  - 输入与输出
  - 处理

### 12.4.4 程序描述

下列各项已经描述了吗:

- 程序的标识和目的;
- 功能描述;
- 输入与输出流的描述;
- 链接到其他程序和/或(子)系统的描述;
- 缓冲区描述;
- 关键数据描述;

- 参数（包含检查）描述；
- 日志描述；
- 程序的示意图表示；
- 属于图表的程序的描述，是否包含了所调用段和节部分？是否使用了标准例程？

#### 12.4.5 性能需求

- 性能需求已经指定了吗？
- 性能需求是可度量的吗？

---

### 12.5 测试设计技术审查清单

#### 12.5.1 控制流测试

预期的测试基础应当是流程图，或者是有描述补充的决策，下列审查清单用于检查该文档：

- 算法已经描述了吗？
- 触发事件的说明清楚吗？
- 是否清楚地指出了哪些数据被使用，以及数据源是什么？
- 可以对输出进行预测吗？
- 各个决策点，包括相随的条件，已经被描述了吗？

#### 12.5.2 分类树方法

可以用下列审查清单来检查规范：

- 处理功能的方法，包括输入、输出与决策路径等已经被描述了吗？
- 处理的触发事件已经说明清楚了吗？
- 输入域描述了吗？
- 可以对输出进行预测吗？

#### 12.5.3 基本比较测试

可以用下列审查清单来检查规范：

- 处理方法已经描述了吗、可以辨认出不同的处理路径吗？

- 『 哪些条件下来决定各种处理路径标识清楚了吗?
- 『 包含输入与输出的处理已经描述清楚了吗?
- 『 每一个输入项的处理已经描述了吗?
- 『 可以对输出进行预测吗?

#### 12.5.4 统计使用测试和稀有事件测试

可以用下列审查清单来检查规范:

- 『 是否已经在功能层次上描述了预期的使用频率?
- 『 每种类型的用户可能执行的功能已经描述了吗?
- 『 确定出不同的客户群了吗?
- 『 每种功能何时被使用已经描述了吗?

#### 12.5.5 基于状态的测试技术

可以用下列审查清单来检查规范:

- 『 给出状态转换图了吗?
- 『 所有的事务/功能被区分开了吗?
- 『 描述与状态转换图一致吗?
- 『 对状态唯一标识了吗?
- 『 事件是如何触发的描述了吗?
- 『 功能的前提条件与后置条件描述了吗?

#### 12.5.6 进化算法

可以用下列审查清单来检查规范:

- 『 计时需求描述了吗?
- 『 可以准备出一个适应度函数吗?

#### 12.5.7 安全性分析

安全性分析是一个迭代过程，要用到大量的特定知识。当第一次执行安全性分析时，至少应当清楚系统结构全面的分解结构。除了设计规范以外，法律与认证机构的需求也是很重要的，这些都是审查清单的主要输入。

## 12.6 测试过程审查清单

这一章的审查清单为测试过程中各种有关活动提供了指导原则。这里并不是要穷举出所有的项，而是作为构造依赖于组织、依赖于项目的审查清单的基础。主要包含以下审查清单：

- 测试项目评估。该审查清单包含项目中间评估与最终评估的属性。
- 系统的全面调查研究。该审查清单可用于支持从计划和控制阶段开始的全面评审和研究活动。
- 前提条件与假定。该审查清单包含在测试计划中包括的前提条件与假定例子。
- 测试项目风险。与测试计划有关的风险必须清晰明了。该审查清单描述了许多可能的风险。
- 测试过程结构化。该审查清单被用于评估测试的当前状况。
- 测试设备。该审查清单可用于在计划和控制阶段，组织和建立测试基础设施、测试组织与测试控制。
- 产品发布。该审查清单可用于确定产品、测试活动、生产要素与生产操作的完备性。
- 编码规则。该审查清单用于编码分析相关的低层次测试活动。

### 12.6.1 测试项目评估

该审查清单有助于防止在测试项目期间出现问题。

- 完成产品发布的审查清单了吗？
- 测试项目是在时间压力之下开展的吗？这是否导致了测试策略的调整？
- 测试项目早期涉及客户了吗？
- 在测试项目期间，可以缩减计划的测试工作量与测试强度吗？
- 有无有经验的（外面的）顾问和测试专家来支持测试团队呢？
- 在测试团队中，学科知识是否足够用？
- 缺陷处理被合理组织了吗？版本控制功能够吗？
- 在出现任何问题时，指定的测试方法（该测试方法在测试手册中有）能够提供足够的信息和支持吗？
- 有对指导原则是否被遵循的检查吗？
- 有偏离开这些指导原则的规程吗？
- 依据规章制度来制定测试计划了吗？

- 有时测试必须完成的一个严格截止日期吗？
- 在测试执行期间，测试计划是否制定得足够详细而且/或得到了维护？
- 能够按时、足够地得到所请求的支持工具（工具、用具等）吗？
- 可以在设定的期限内，完成预期的测试计划吗？
- 测试计划清晰吗？可以由测试团队的成员来执行吗？
- 采用了时间进度安排的技术和工具吗？
- 测试活动和测试进展得到充分监控了吗？
- 进度调整的原因被明确地归档了吗？
- 可以明确指出测试活动是否按进度进行吗？
- 测试计划和整个项目计划之间每时每刻都配合得很好吗？
- 标准化的技术和工具够吗？
- 在测试团队中，有足够的下列成员吗：
  - 系统设计员
  - 系统管理员
  - 系统管理职员
  - 领域专家
  - 客户
  - 审计部门成员
- 测试过程中的管理介入程度够吗？
- 任务分配是恰当而明确的吗？
- 实现了系统部件的版本控制吗？
- 需要对硬件、软件和/或测试进行专门的培训吗？
- 测试团队成员是统一调配吗？
- 是按照指定方案来应用测试生命周期吗？
- 在测试期间，所有的系统功能都可以完成吗？
- 有指定以外的测试用例被执行了吗？
- 客户是否评估了由开发团队使用的测试方法的质量？
- 开发团队向验收测试人员提供了复杂的测试用例吗？
- 错误猜测使用了吗？
- 实用程序被测试了吗？
- 在测试结束时，规定要进行回归测试吗？
- 使用了自动化工具来生成测试用例吗？
- 所有预定的测试脚本都被执行了吗？
- 所有可用的审查清单都被使用了吗？

- 测试团队和开发团队中的顾问结构成功吗？
- 有足够的文件规范吗？
- 测试件系统地完成了吗？
- 测试方法中的缺陷原因、进度安排和技术都被完整地归档了吗？
- 预算够吗？
- 有足够的时间来进行测试活动和咨询活动吗？
- 办公环境与所需的一样吗？
- 有足够的测试工具、终端和打印机等可用吗？
- 有足够的人员吗？
- 按时给出测试结果了吗？
- 是在测试执行时执行检查（随机检查）的吗？
- 每份测试文档的内容与约定的一致吗？
- 每份测试文档的内容完整吗？
- 用户可以得到测试文档吗？

## 12.6.2 系统的全面调查研究

在计划阶段的总体审查和研究过程中，要收集有关测试项目和待测系统的信  
息。这有助于明晰测试团队的预期目标，还有他们对组织的预期。

### 项目信息

- 委托人；
- 承包人；
- 目标（分配任务、策略的重要性）；
- 协定（合同）；
- 项目组织，包含人员；
- 报告流程；
- 活动和时间进度；
- 财务计划；
- 将被执行的测试的前提条件集合；
- 风险和措施。

### 系统信息

- 标准与指导方针（开发方法等）；
- 需求研究（包括现有情况的描述）；

- ※ 基本设计阶段给出的文档；
- ※ 开发基础设施（硬件、工具与系统软件）；
- ※ 测试件已经给出；
- ※ 测试设计；
- ※ 测试脚本；
- ※ 测试方案；
- ※ 测试基础设施；
- ※ 测试工具；
- ※ 度量；
- ※ 以前测试的评估报告。

### 12.6.3 前提条件和假定

前提条件由测试项目以外的组织定义，但也可以由测试项目内的测试团队来定义。外部组织定义的前提条件或多或少就是对测试团队必须完成的任务提出的限定条件。一般来说，这些是与所需的资源、人员、预算和时间有关的限定和条件。测试团队为了完成其任务，也可以定义由其他组织必须完成的前提条件。

- ※ **最后期限。**应当在固定的最后期限之前完成测试。
- ※ **项目计划。**当前的项目计划引导所有的测试活动。
- ※ **测试单元的交付。**由开发团队交付的软件必须在功能上可用且是可测试的单元。而且用户手册必须是可用的。每个功能单元都必须递交来进行单元测试和系统测试。
- ※ **开发计划的把握和变化。**测试团队应当清楚开发团队的软件交付进度，该进度的变化必须通知测试团队。
- ※ **参与开发进度。**测试团队必须有机会参与软件交付顺序的决策。交付顺序会对测试工作的强度和速度产生影响。
- ※ **系统测试的质量。**开发团队根据指定的质量特性和目标来进行系统测试。
- ※ **深入把握系统测试。**系统测试将下列交付物交付给验收测试团队：
  - 进度；
  - 测试策略；
  - 测试用例；
  - 测试结果。
- ※ **测试基础的范围。**测试团队有权使用所有的系统文档。
- ※ **测试基础的变化。**测试基础发生变化时，必须立即通知测试团队。

- 测试基础的质量。如果测试基础的质量不够或遗漏了基本信息，必须立即报告以便采取相应的措施。
- 测试团队的可用性。必须能依据测试进度来安排测试团队。团队的成员应当具备应有的知识和经验。
- 开发团队的支持。为了对阻碍测试进程的缺陷加以修正，开发团队的成员必须提供支持。
- 测试环境的定义和维护。指定的测试环境必须按时可用，而且在测试执行期间也要有相应的维护支持。
- 测试环境的可用性。在测试执行期间，测试团队是测试环境的拥有者。没有测试管理人员的允许，不允许修改或使用测试环境。

#### 12.6.4 测试项目风险

测试计划中定义并描述项目风险。对每一个风险，需要描述出风险的后果，以及使风险降到最低程度的解决方法和/或采取的措施。

- 对多个子系统，开发团队有没有详细的进度表。
- 与测试计划中所提到的一样，前面确定的最后期限会对整个测试活动的执行产生影响。
- 测试基础能不能按时准备好。
- 测试基础的质量够不够。
- 测试人员的可用性（在测试经验和专长方面的能力与适应性）。
- 期望的测试环境的可用性和可控性。

#### 12.6.5 测试过程结构化

如果在测试组织中引入结构化测试，那么就应当首先确定测试的当前状况，这需要一个称为测试目录的简单调查。

##### 测试生命周期

- 有哪些测试层次，每个测试层次的范围是什么？
- 在这些测试层次之间有协作吗？
- 每个测试层次有各自的生命周期吗？

##### 计划和控制阶段

- 每个测试层次都制定了测试计划吗？
- 是基于主测试计划来制定详细测试计划吗？

- 测试计划的内容包含了对分配任务、定义职责、范围、假定和前提条件的描述吗？
- 测试计划中描述测试基础了吗？
- 测试策略确定了吗？
- 测试计划中已经定义测试基础设施了吗？对测试基础设施的控制达成协议了吗？
- 测试计划中描述报告缺陷的规程了吗？
- 测试计划中描述测试交付物是什么了吗？
- 测试计划中描述测试件的各部分了吗？已经安排好如何来控制测试件了吗？
- 测试计划中描述了哪些（测试）功能需要被区分开吗？这里指的（测试）功能包含任务划分、职责和权限。
- 测试计划中描述测试团队的各种报告流程了吗？
- 测试计划中包含测试活动进度、财务和人员计划了吗？
- 已经为测试建立了充足的预算吗？

#### 准备阶段

- 对每个测试层次，都需要执行测试基础的可测性审查吗？
- 仔细考虑并对将使用的测试技术做出选择了吗？

#### 细化阶段

- 测试设计准备好了吗？
- 测试设计技术被用于测试设计的准备了吗？

#### 执行阶段

- 缺陷管理规范化了吗？
- 有再测试吗？

#### 完成阶段

- 有没有为维护测试而保存测试件？
- 有正式的移交/发布吗？

#### 测试技术

- 使用技术来确定测试策略了吗？
- 使用预先安排的技术了吗？
- 使用测试设计技术了吗？

### 项目组织

- 测试项目是如何组织的?
- 权力和职责是如何分配的?
- 谁来执行测试?
- 关于报告有协议达成吗?
- 有规程来进行缺陷控制吗?
- 如何进行审查和移交?
- 谁有权发布已测试完成的系统?

### 流程组织

- 测试中使用标准了吗?
- 标准的使用经过检查了吗?
- 收集与测试有关的任何经验数据了吗?
- 有专门领域的专家可用于测试吗?
- 是通观整个测试项目来协调测试的吗?
- 在测试方面进行培训了吗?
- 成员在测试方面具备的知识水平如何?

### 基础设施和测试工具

- 有单独的测试环境来用于测试吗?谁是该测试环境的拥有者?
- 测试环境与产品环境进行比较了吗?
- 有规程来考虑测试环境的版本和配置控制吗?
- 测试环境的组织和使用权限是与开发分开的吗?
- 有可用于下列各项的测试工具吗:
  - 测试管理(制定计划、时间登记和进度监控);
  - 测试设计;
  - 缺陷管理;
  - 测试预算;
  - 测试执行;
  - 文件管理。

### 12.6.6 测试设备

这部分包含了在计划和控制阶段期间,对组织测试基础设施、测试组织与测试控制等方面来说很重要的一些属性。

### 工作空间

- 房间；
- 会议室；
- 办公用具（椅子、书架、办公桌和橱柜）；
- 复印设备；
- 办公补给；
- 表格；
- 磁盘。

### 硬件

- PC 机、便携式电脑；
- 输入和输出媒体，如打印机、显示器和 PIN/读卡器；
- 通信线路；
- 存储空间（磁带、磁盘空间）；
- 通讯服务器、网络管理硬件；
- 服务器、换向器；
- 接口；交换机、调制解调器；转换器、适配器；
- 与公共网的连接；
- 存储媒体，如磁带、盒式录音带、磁盘等；
- 电源、冷却、电缆；
- 模拟器。

### 软件

- 操作系统；
- 通信软件；
- 字处理软件包；
- 计划和进展监控软件包；
- 测试工具；
- 日志软件；
- 授权与安全；
- 会计/统计；
- 汇编程序、编译器；
- 测试软件；
- 仿真软件。

### 培训

- 测试的介绍课程；
- 测试技术课程；
- 功能设计的基本理解；
- 技术设计的基本理解。

### 后勤

- 咖啡、茶叶；
- 午餐；
- 费用规章制度；
- 加班制度；
- 办公以外时间的进入安全。

### 职员

- 测试团队；
- 系统控制；
- 用户；
- 产品监督；
- 管理。

### 规程

- 产品控制；
- 从开发团队向测试团队的交付；
- 测试对象、测试环境与测试文档的版本控制；
- 缺陷规程；
- 测试说明书；
- 进度监控；
- 测试咨询；
- 配置管理；
- 缺陷报告和处理，重新开始测试；
- 授权与安全。

### 文档

- 测试规章制度或一份测试手册；
- 用户文档。

### 12.6.7 产品发布

在做出产品发布的建议之前，应当完成下列产品发布的审查清单。该清单仅适用于所有的测试都已经执行完成的情况。

#### 交付物的完成

- 回归测试可用于维护了吗？
- 软件有可用的必要帮助、出现紧急情况时的校正措施吗？
- 所需的系统文档、用户文档和产品文档都有吗？是否已完成、易得而且一致？

#### 测试活动的完成

- 各个软件模块之间的接口都已经测试并得到认可了吗？
- 各个子系统之间的接口都已经测试并得到认可了吗？
- 与其他系统的接口已经测试并得到认可了吗？
- 与分布应用程序的接口已经测试并得到认可了吗？
- 与分布设备的接口已经测试并得到认可了吗？
- 与操作系统的接口已经测试并得到认可了吗？
- 与手工处理程序的接口已经测试并得到认可了吗？
- 测试件完成了吗？
- 测试件已经通过所需的质量检查了吗？
- 满足所需的性能需求吗？
- 有检查点恢复机制吗？它有效吗？
- 所有的重新启动选项都已经成功测试了吗？
- 在合适（安全）的请求下，可以访问文件吗？
- 输入文件为空，程序也可以运行吗？
- 系统对无效输入能够正确响应吗？
- 各个表项已经至少被测试了一次吗？

#### 生产准备

- 在将要投入生产时，仍然存在缺陷吗？
- 采取措施来处理遗留下来的缺陷了吗？

#### 生产实现

- 系统的生产量确定了吗？
- 用户和控制功能是完全分开的吗？

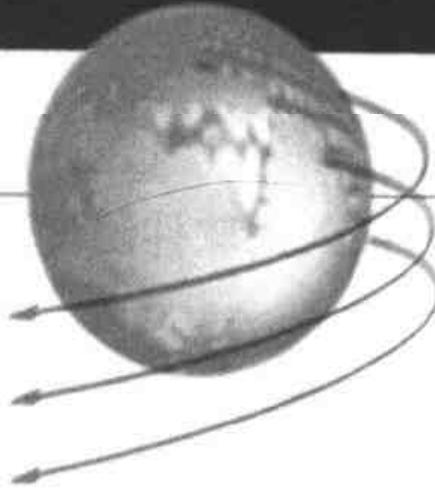
- 发现并排除故障的规程清晰吗？

#### 12.6.8 开发阶段的审查清单：编码规则

系统如何开发和归档，将影响到可维护性、可复用性和可测性等质量特性。在开发时，需要使用一些标准来增强编码的统一性。这些编码标准可以由类似于代码分析程序的静态分析程序来强制执行并检查。这些工具多数有自己内建的编码测试，可以调整来满足组织的需求或标准。编码标准也依赖于所使用的编程语言。下列审查清单给出了一些例子：

- 程序段的复杂度超过给定的最大值了吗？
- 语句数目超过给定的最大值了吗？
- 在函数体中描述前提条件与后置条件了吗？
- 使用注释行来解释代码了吗？
- 嵌套层数目超过给定的最大值了吗？
- 有用于函数名称和变量名称的标准吗？





## 第四部分 基础设施

基础设施是指结构化测试所需的全部设备。它包括执行测试所需的设备（测试环境），和支持有效而高效率地执行测试活动的设备（工具和测试自动化）。

通常，在开发（和测试）过程的前期，还没有大量的硬件可供测试人员使用。此时他们需要硬件部件的模拟器或原型。在测试过程的后期，模拟器和原型必须被更新的原型甚至是实际部件所替代。第 13 章描述了测试过程的不同阶段所需的不同测试环境。

工具可以使测试人员更能够自如工作。测试工具的范围是非常广的，它可以为许多种测试活动提供支持。第 14 章简要介绍了测试过程各个阶段中可以应用的各种工具。

测试执行的成功自动化需要的不仅是功能强大的工具。“工具将为你自动测试”的梦想很久以前就已经破灭。第 15 章解释了如何来获得成功的测试自动化，描述了技术与组织等方面的情况。

第 16 章讨论了当测试人员在处理混合信号（模拟与数字信号）环境中的工作所面临的特定情况。它描述了测试环境的特定需求、当前各种可用的工具及技术。

# 第 13 章 嵌入式软件测试环境

## 13.1 介绍

包含嵌入式软件的系统在开发过程的不同阶段，涉及大量测试活动，每个测试活动都需要有特定的测试设备。本章将讨论这些测试活动需要什么样的测试环境，它们要达到什么测试目标。我们将用嵌入式系统中的一般布局来描述所需测试环境之间的差异（见图 1.2）。

在开始开发嵌入式系统到产品发布之间，通常将开发阶段划分为下列几个阶段：

- 模拟阶段；
- 原型阶段；
- 临近生产阶段。

对于复杂的系统，这些阶段还可以由更多的子阶段组成；对于不太复杂的系统，可以省略其中一个或多个阶段。在开发阶段，要对产品进行测试和提高质量，直到确定已经具备足够的质量可以投入生产。在开发阶段之后，要有一个最终阶段，对制造过程进行测试和监督：

- 开发后阶段。

这一章可应用于所有分阶段来开发的嵌入式系统项目：首先是被模拟的部分（模拟阶段），然后用真实部件一个一个来替代模拟部件（原型阶段），直到最终真正的系统在其真实环境下工作（临近生产阶段）。开发完成后，有一个测试和监控生产流程的阶段（开发后阶段）。可以很容易地将这些阶段与多 V 生命周期（见第 3 章）相关联起来，如图 13.1 所示。本章依据多 V 模型的结构来详细描述测试环境，而且可以明确指出，这些不会受到应用该模型的项目的限制。

一些组织将测试过程划分为诸如“MT/MiL”、“RP”、“SiL”、“HiL”与“ST”等测试（见图 13.2）。可以按照下面的方式，将这些测试映射到上面提到的开发阶段：在“模型测试”（MT）和“模型循环”测试（MiL）中，对早期开发的用于模拟系统的模型进行测试，这对应于模拟阶段。在真实情况环境下，采用“快速原型法”（RP）来测试有着充足的性能和资源（例如 32 位浮点数处理）的试验模型，验证它是否可以实现系统的目标，这也是模拟阶段的一部分。在模拟环境或

试验硬件上，采用“软件循环”测试（SiL）来测试真正的软件（考虑所有的资源约束，例如16位或8位整数处理）。采用“硬件循环”测试（HiL），是在模拟环境中使用和测试真正的硬件。“软件循环”测试和“硬件循环”测试都可以被原型阶段所覆盖。“系统测试”（ST）是在真实环境中对真正的系统进行测试。这对应于临近生产阶段。

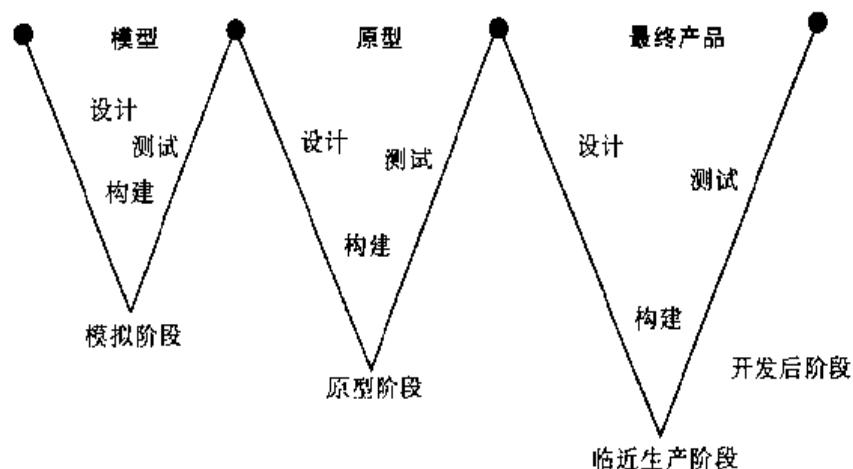


图 13.1 （在本章中用到的）开发阶段与多 V 模型之间的关系

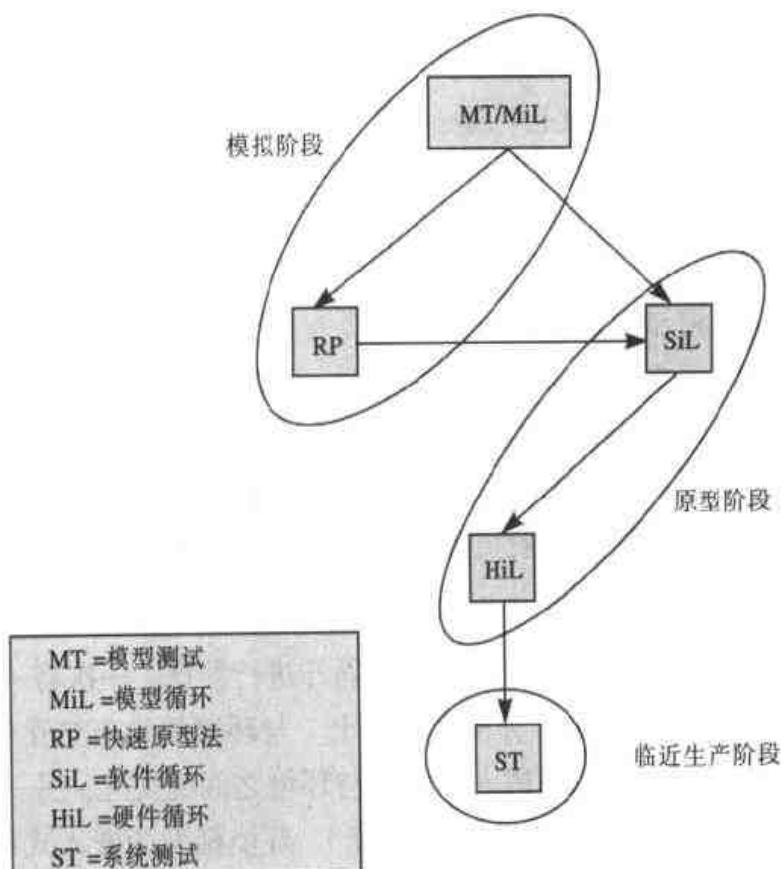


图 13.2 一个从“模拟”系统到“真实”系统逐渐转变的测试过程

本章关注的是嵌入式系统执行测试需要的测试设备。除此之外，测试设备也可以用于支持测试过程、审查和分析早期的测试结果，并与当前结果相比较是有必要的。有时候，客户或政府官员可能要求递交有关如何、何时、何地以及用什么样的配置来获得测试结果的详细信息。因而，很重要的一点是保存准确的测试日志，并对原型单元维护严格的 H/W 与 S/W 的配置控制。如果开发（和测试）过程是长期而又复杂的，建议维护一个数据库来存储所有的测试数据。该数据库应当包含所有的相关数据，例如测试日期/时间、测试地点、被测原型的序列号码、硬件与软件配置标识、数据记录设备的校正数据、测试运行次数、数据记录号码、测试计划标识和测试报告标识，等等。在第 14 章中，将讨论更多这样的测试设备。

测试的并行执行通常需要有多个测试环境。而且由于原型可能会在环境等测试中被毁坏，因而同一个原型可能需要有多个样品。

## 13.2 第一阶段：模拟阶段

基于模型的开发中，在需求确认和概念设计之后，开始创建一个仿真模型。嵌入式系统的开发人员使用这个可执行的仿真模型来支持初始设计、验证概念并开发和校验下一开发步骤的详细需求。这一开发阶段也被称为“模型测试”和“模型循环”。在这一阶段的测试目标是：

- 验证概念；
- 优化设计。

执行一个仿真模型并测试其行为，需要有一个特定的测试环境或测试床。需要有专用的软件工具来建造和运行仿真模型，生成模型中的信号并分析响应。这里应当提到的是，Petri nets ( Ghezzi 等, 1991 ) 的使用作为一门技术，尤其适用于异步系统。这一部分将解释测试仿真模型的不同方法及所需的测试环境。因而，我们这里简化嵌入式系统的一般描述（图 1.2），只区分“嵌入式系统”与其环境。

在模拟阶段的测试一般由以下步骤组成：

1. 单向模拟。嵌入式系统的模型被分隔开进行测试。一次将一个输入信号注入模拟的嵌入式系统并分析最终输出。与环境的动态交互被忽略。
2. 反馈模拟。测试模拟的嵌入式系统与环境之间的交互。另一个术语是校验“控制率”（通常用于处理控制系统）。环境模型为嵌入式系统生成输入，模拟的嵌入式系统的最终输出被反馈回环境模型，从而为嵌入式系统生成新的输入。

3. 快速原型法。与真实环境相连来测试仿真的嵌入式系统，见图 13.3。它是评估仿真模型正确性的根本方法。



图 13.3 用于汽车嵌入式系统的快速原型法（来源：ETAS）

“反馈模拟”需要首先开发环境的有效模型，该模型与所预期的环境模型的动态行为尽可能接近。通过与环境的实际行为相比较来校验该模型。常常是首先开发一个详细模型，接下来进行简化，详细模型与简化模型都需要得到校验。可以通过“单向模拟”来校验环境模型。

在基于模型的开发中，一个有效的工程实践是，在开发嵌入式系统模型之前开发环境的模型。然后用环境的（简化）模型来导出在嵌入式系统中必须实行的控制规则。接下来，用单向模拟、反馈模拟和快速原型法来验证控制规则。

### 13.2.1 单向模拟

嵌入式系统用一个可执行的模型来模拟，而环境行为被忽略。生成输入信号，注入到嵌入式系统的模型中，模型的输出信号被监控、记录并加以分析。这种模拟是“单向”的，因为在模型中不包含“环境”的动态行为。图 13.4 用图形方式来描述这种模拟。

需要有工具来为嵌入式系统的模型生成信号，同时记录模型的输出信号。可以手工来比较记录信号与预期结果，但也可以通过工具来完成。信号生成工具甚至可以基于实际结果生成新的输入信号，以减少设计中的可能缺陷。



图 13.4 单向模拟

根据模拟环境的不同，可以采用不同的方法，来为模型生成输入信号并记录模型的输出信号：

- 在计算机平台上模拟嵌入式系统。可以依靠在模拟平台外围总线上的硬件，来为系统注入输入信号并记录输出信号。可以通过计算机平台的操作终端来手工控制并读出模型中的变量。
- 模拟 CASE 环境中的嵌入式系统，生成输入信号的激励并捕获该环境中的输出响应。通过类似 UML 等建模语言来创建可执行的模型。根据对应的用例和顺序图，可以自动地生成测试用例。

单路模拟也被用于环境模型的模拟。此时，模型中不包含（有控制规则的）嵌入式系统的动态行为。

### 13.2.2 反馈模拟

在一个可执行的动态模型中，嵌入式系统及其周围环境被模拟。如果能够对嵌入式系统及其环境的仿真模型的复杂度加以约束，而且不会降低模型的逼真度，那么这一选项就是可行的。

例如，如果设计的嵌入式系统将用于汽车的导航控制，系统环境就是汽车本身，加上道路、风、外部温度、驾驶员等。从可行性和有用的观点来看，可以确定该模型（初始）只限于导航控制、节流阀的位置和汽车速度。

图 13.5 给出了反馈模拟的图形化表示。

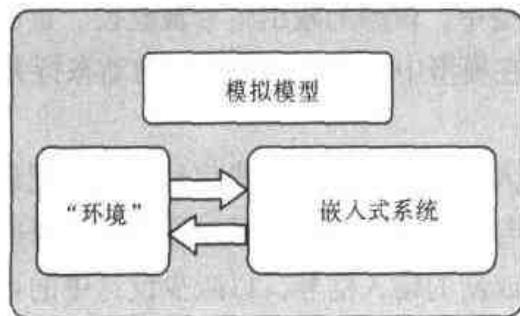


图 13.5 反馈模拟

在对嵌入式系统本身或“环境”单向模拟之后，反馈模拟可能是复杂设计过程中的下一步。

可以通过执行许多测试用例来确认设计，用例可能偏离了模型特征并且更改了模型运行的状态。模型的响应被监控和记录，随后进行分析。

模型的测试床需要能够更改嵌入式系统模型或环境模型的特征，而且也必须能够读取并捕获其他特征，从而导出模型的行为。再者，就像单向模拟中一样，测试床也可以被安排在一个专用的计算机平台以及 CASE 环境中。

在这一阶段，使用的工具有：

- 信号生成设备和信号监控设备；
- 模拟计算机；
- CASE 工具。

### 13.2.3 快速原型法

一个可选的步骤是，通过采用实际或足以等价的环境来代替对环境的详细模拟，从而可以对控制规则有更多的确认（见图 13.6）。例如，在一台模拟计算机上运行对导航控制控制规则的模拟，可以通过将计算机放在汽车的乘客座位上，而且与汽车的机械和电子器件附近的传感器和制动器相连（见图 13.3）来得到校验。这里一般使用高性能的计算机，而忽略系统的资源约束。例如，快速原型法软件可以是 32 位浮点数处理，而最终产品被限制为 8 位整数处理。

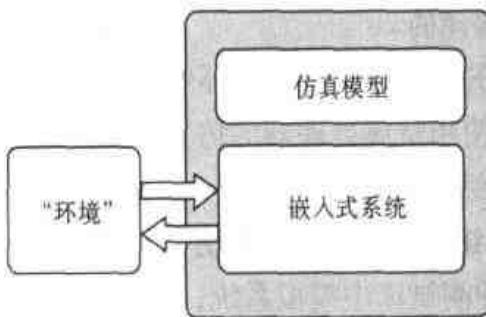


图 13.6 快速原型法

## 13.3 第二阶段：原型阶段

此时不是利用模型来开发嵌入式系统。以上（第一阶段）描述的基于模型的开发中，在达到模拟阶段的目标以后，进入原型阶段。

这一阶段的目标是：

- 证明（来自前一阶段）仿真模型的有效性；
- 确认系统满足需求；
- 临近生产的单元发布。

在原型阶段，实际的系统硬件、计算机硬件、软件的试验与实际版本将逐渐替代模拟部件。这时，在模拟模型与硬件之间需要有接口。可以使用一个处理器仿真器。在模拟模型中可容易得到的信号，实际硬件可能不会很容易地得到。因而，必须安装专门的信号传感器与转换器，以及信号记录与分析设备。可能有必要校准传感器、转换器与记录设备，并且保存校准日志来修正所记录数据。开发一个或多个原型通常是一个需要迭代好多次的过程。软件和硬件是并行开发的，常常要集成到一起来检验是否仍然可以工作。每一次的原型开发，都意味着与最终产品的差距越来越小。随着每一步骤的进行，在前期阶段得出的有效性结论的不确定性也进一步减少。

为了阐明这一模拟部件逐渐为真实部件所替代的过程，在嵌入式系统的一般结构中，标识出四个主要的区域（见图 13.7）。它们在布局中可以分别从属于原型阶段的模拟、仿真、试验或初步版本部分：

1. 用嵌入式软件来实现系统行为。可以通过下面方式来模拟：
  - 在主机上运行面向该主机编译的嵌入式软件；
  - 在目标处理器的仿真器上运行嵌入式软件，该目标处理器运行在主机上。嵌入式软件的运行可以被认为是“真实的”，因为该软件是面向目标处理器而编译的。
2. 处理器。在早期开发阶段，在开发环境中可以使用高性能的处理器。可以在开发环境中，使用最终处理器的仿真器来测试实际软件，实际软件是面向目标处理器而编译的。
3. 其他的嵌入式系统。可以通过下面方式来模拟：
  - 在主计算机的测试床中模拟系统；
  - 构造一个试验硬件配置；
  - 构造一个初步的（原型）PCB 及其所有的部件。
4. 嵌入式系统的环境。可以通过信号生成方法来静态地模拟，或通过模拟计算机来动态模拟。

在原型阶段，可以应用下列测试层次（见 4.1.2 节）：

- 软件单元测试；
- 软件集成测试；

■ 硬件/软件集成测试；

■ 系统集成测试；

■ 环境测试。

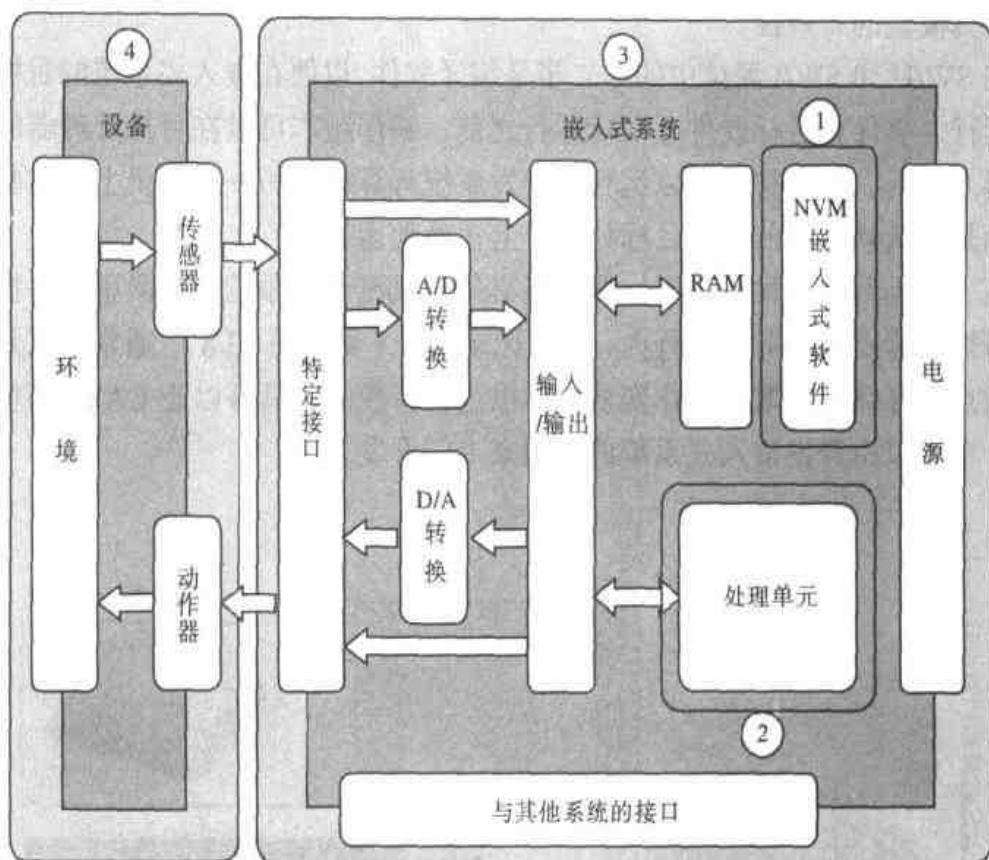


图 13.7 一个嵌入式系统中的模拟区域

每一个测试层次都需要有专门的测试环境，这将在下面部分予以描述。在下面部分，将反复有一个表出现，该表引用图 13.7 中的模拟区域。从这些表中，就可以很容易地看出每个测试层次的进展是如何朝着“最终实际产品”方向而去的。

在原型阶段，有时候使用“软件循环”与“硬件循环”两个术语。它们与这里所描述的软件集成测试、硬件/软件集成测试与系统集成测试等测试类型相对应。

### 13.3.1 软件单元测试与软件集成测试

对于软件单元 (SW/U) 测试和软件集成 (SW/I) 测试，创建一个可以与仿真模型测试环境相比较的测试床。两者的区别在于执行的对象。在原型阶段，测试对象是一个软件单元或一组集成软件单元的可执行版本，它是在设计基础上开发的，或是从模拟模型生成的。

第一步是编译软件以便在主机上执行。这个环境（主机）对资源、性能或功

能强大的商品工具没有限制，这使得开发和测试要比在目标环境中容易很多。这种测试也被称为主机/目标机测试。对这些在“主机上编译”的软件单元和集成软件单元的测试目标是：根据技术设计来校验它们的行为，以及确认在前一阶段使用的仿真模型的有效性。

在 SW/U 和 SW/I 测试中的第二步是编译软件，以便在嵌入式系统的目标处理器上运行。软件在目标硬件上实际运行之前，编译版本可以在目标处理器的一个模拟器上运行。该模拟器可以运行在开发系统计算机或另一个主机上。这些测试的目标是确认软件将能够在目标处理器上正确地运行。

在上面提到的两种情况下，测试床必须提供测试对象的输入激励，并提供其他的特性来监控测试对象的行为，同时记录信号（见例图 13.8）。通常方式是提供进入断点，存储、读取和操作变量。提供这些特性的工具可以是 CASE 环境的一部分，也可以由开发嵌入式系统的组织来专门开发。

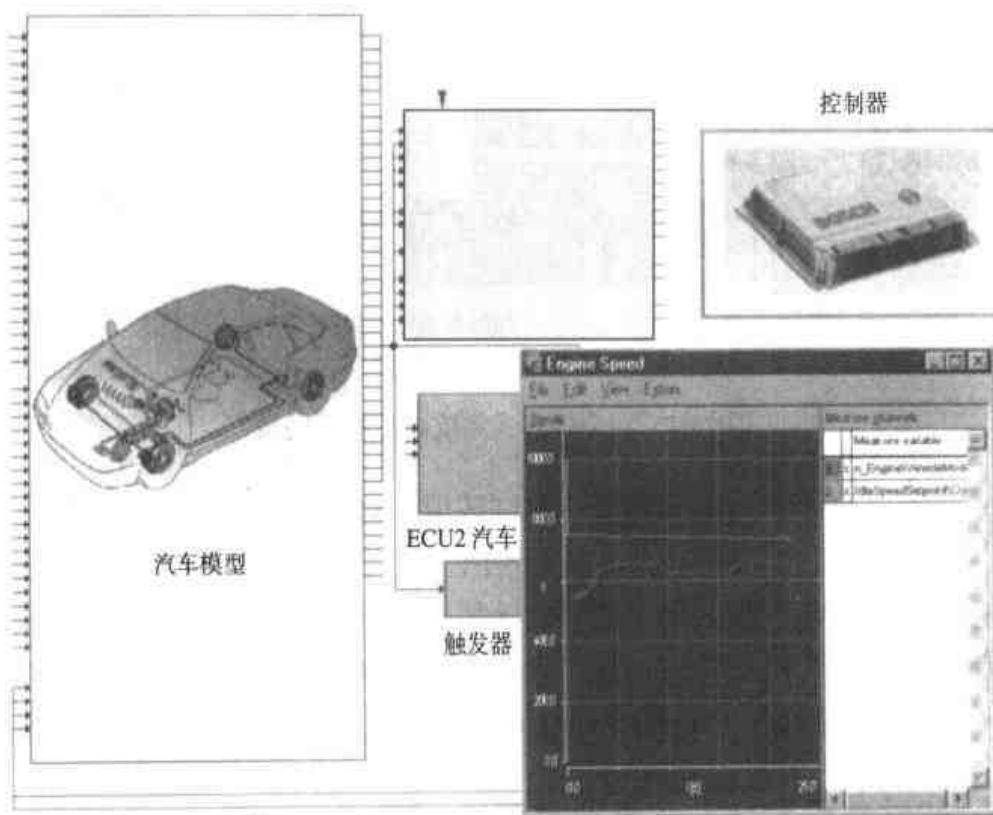


图 13.8 “软件循环”测试（来源：ETAS）

表 13.1 概要给出了在 SW/U 和 SW/I 测试步骤中的模拟层次。表中的列是在嵌入式系统一般结构中的模拟区域（见图 13.7）。

表 13.1 用于 SW 单元测试和 SW 集成测试的模拟层次

| 嵌入式软件          | 处理器     | 嵌入式系统其他部分 | 环境  |
|----------------|---------|-----------|-----|
| SW/U, SW/I (1) | 试验（主机）  | 主机        | 模拟的 |
| SW/U, SW/I (2) | 实际（目标机） | 仿真器       | 模拟的 |

### 13.3.2 硬件/软件集成测试

在硬件/软件集成 (HW/SWI) 测试中，测试对象是要加载集成软件的硬件部分。软件被加载到硬件的存储器中，通常是指 (E) EPROM。这部分硬件可能有一个试验配置，例如是一个硬布线电路板，该电路板包含几个部件，存储器是其中的一部分。术语“试验的”表示所使用的硬件将不会再进一步开发（与一个“原型”相对比，原型通常还要继续被开发，而试验硬件是一个要被“丢弃的原型”）。HW/SWI 测试的目标，是校验嵌入式软件在目标处理器上能否正确与周围的硬件协同运行。由于硬件行为是该测试的一个基本部分，因而通常也被称为“硬件循环”（见图 13.9）。



图 13.9 “硬件循环”测试（来源：ETAS）

用于硬件/软件集成测试的测试环境必须与硬件有接口。依赖于测试对象及其完备程度，存在下列可能性：

- 用信号生成器提供输入激励；
- 用示波器或逻辑分析仪，以及数据存储设备一起来监控输出；
- 用电路内嵌的测试设备来监控非输出的其他点的系统行为；
- 在实时模拟器中模拟测试对象（“环境”）的环境。

表 13.2 概要给出了在 HW/SW 集成测试中的模拟层次。表中的列指在嵌入式系统一般结构中的模拟区域（见图 13.7）。

表 13.2 用于 HW / SW / I 集成测试的模拟层次

|                | 嵌入式软件   | 处理器     | 嵌入式系统其他部分 | 环境  |
|----------------|---------|---------|-----------|-----|
| SW/U, SW/I (1) | 试验（主机）  | 主机      | 模拟的       | 模拟的 |
| SW/U, SW/I (2) | 实际（目标机） | 仿真器     | 模拟的       | 模拟的 |
| HW/SW/I        | 实际（目标机） | 实际（目标机） | 试验的       | 模拟的 |

### 13.3.3 系统集成测试

在系统集成测试中，嵌入式系统包含的所有硬件部分被整合起来，一般是在一个原型 PCB 上。很明显，这要求前面所有的测试：单元测试、软件集成测试、硬件/软件集成测试都必须成功执行。系统的所有软件部分都必须被加载。系统集成测试的目标是校验整个嵌入式系统能否正确运行。

系统集成测试的测试环境与硬件/软件集成测试的环境很类似。毕竟，整个嵌入式系统也是包含软件的一个硬件部分。两者的差异实际上在于，整个系统的原型 PCB 提供最终的 I/O 和电源连接器。可以通过这些连接器来提供激励输入和监控输出信号，以及与动态模拟结合。在 PCB 预先定义的位置，可以监控连接器的信号并执行电路内嵌的测试。

表 13.3 概要给出了系统集成测试中的模拟层次。表中的列指在嵌入式系统一般结构中的模拟区域（见图 13.7）。

表 13.3 用于系统集成测试的模拟层次

|                | 嵌入式软件   | 处理器     | 嵌入式系统其他部分 | 环境  |
|----------------|---------|---------|-----------|-----|
| SW/U, SW/I (1) | 试验（主机）  | 主机      | 模拟的       | 模拟的 |
| SW/U, SW/I (2) | 实际（目标机） | 仿真器     | 模拟的       | 模拟的 |
| HW/SW/I        | 实际（目标机） | 实际（目标机） | 试验的       | 模拟的 |
| 系统集成           | 实际（目标机） | 实际（目标机） | 原型        | 模拟的 |

### 13.3.4 环境测试

在这一阶段，环境测试的目标是检测和校正前面一些阶段中环境中可能出现的问题。这可以与临近生产阶段的环境测试相对比，那是为了论证环境的符合度。环境测试更适宜在已经有足够成熟度的原型上进行。这些原型建立在 PCB 上，按照正确的规范与硬件相连。如果要求在嵌入式系统中有防电磁干扰（EMI）措施，

那么也必须在测试的原型上有防EMI措施。

环境测试要求有特殊的测试环境。在大多数情况下，测试环境可以通过购买或利用购买的部件组装来满足要求。如果某些特定类型的环境测试只是偶尔执行，那么建议租用必要的设备或将测试承包给专业公司。

在环境测试期间，可能需要测试运行中的嵌入式系统。这时，就必须像前面两个测试层次一样来创建模拟的“环境”。

环境测试由下面两类测试组成：

1. 强调测量。这类测试是要确定嵌入式系统对环境影响的程度，例如系统的电磁兼容性等。因而，这类测试所需的工具是测量设备。
2. 强调生成。这类测试是要确定嵌入式系统对周围条件的敏感性，例如温度和湿度测试、电磁干扰的冲击测试和振动测试等。所需的测试设施就是将这些条件施加在系统上的设备，诸如空调室、冲击仪表以及为EMC/EMI测试所装备的区域。同样，这里也需要有测量设备。

表13.4概要给出了环境测试中的模拟层次。表中的列是在嵌入式系统一般结构中的模拟区域（见图13.7）。

表13.4 用于环境测试的模拟层次

|                 | 嵌入式软件   | 处理器     | 嵌入式系统其他部分 | 环境  |
|-----------------|---------|---------|-----------|-----|
| SW/U, SW/I(1)   | 试验（主机）  | 主机      | 模拟的       | 模拟的 |
| SW/U, SW/I(2)   | 实际（目标机） | 仿真器     | 模拟的       | 模拟的 |
| HW/SW/I<br>系统集成 | 实际（目标机） | 实际（目标机） | 试验的<br>原型 | 模拟的 |
| 环境测试            | 实际（目标机） | 实际（目标机） | 实际        | 模拟的 |

## 13.4 第三阶段：临近生产阶段

临近生产阶段需要建造一个生产前的单元，它用于最后一次校验是否所有的需求，包括环境需求和政府需求，都已经得到满足，以及用于发布最终设计投入生产。

这一阶段的测试目标是：

- 最后一次校验所有的需求是否得到满足。
- 论证是否符合环境标准、行业标准、ISO标准、军方标准及政府标准。临近生产单元是代表最终产品的一个单元。在早期的原型上可能已经执行

了初步测试并修正了一些缺陷，但这些测试的结果可能仍然不能被接受为最终认可。

- 论证可否在预定时间、预定工作之内的生产环境下建造出系统。
- 论证系统可否在实际运行环境中得到维护，满足 MTTR 要求。
- 向（潜在的）客户演示产品。

临近生产单元是一个在实际运行环境中被测试的实际系统。它是前面所有阶段工作的终点，如表 13.5 所示。临近生产单元等于，或至少是代表最终生产单元。与生产单元的区别在于可能对临近生产单元仍然有测试规定，像信号拦截，等等。然而，这些是生产质量规定，而不是实验质量规定。也可能需要有不只一个样品，因为测试可能并行开展或者在测试中单元可能被毁坏。有时候，临近生产单元也被称为“红色标签单元”，而实际生产单元被称为“黑色标签单元”。

表 13.5 临近生产阶段是逐步用实际部件替代模拟部件的最后一个阶段

|                 | 嵌入式软件   | 处理器     | 嵌入式系统其他部分 | 环境  |
|-----------------|---------|---------|-----------|-----|
| SW/U, SW/I (1)  | 试验（主机）  | 主机      | 模拟的       | 模拟的 |
| SW/U, SW/I (2)  | 实际（目标机） | 仿真器     | 模拟的       | 模拟的 |
| HW/SW/I<br>系统集成 | 实际（目标机） | 实际（目标机） | 试验的       | 模拟的 |
| 环境测试            | 实际（目标机） | 实际（目标机） | 原型        | 模拟的 |
| 生产前期            | 实际（目标机） | 实际（目标机） | 实际        | 模拟的 |
|                 |         |         |           | 实际  |

依据一个或多个预先确定的测试方案，对临近生产单元进行实际情况测试。在导航控制的例子中，实际测试可能由多个测试组成，测试是由安装在一个原型汽车上的导航控制临近生产单元来驱动的。对其他临近生产单元可能需要进行环境限制测试。在前面一些阶段中，可能已经从模拟模型中生成了测试输入信号，而且可能对输出数据进行在线监控，或存储在硬盘中以便日后进行分析。在临近生产阶段，需要有检测仪器、数据显示与数据记录设备，可能还需要遥感勘测，例如，如果测试涉及到用仪表设备或靠人只能观察到有限空间的运载工具，诸如导弹、飞行器、赛车和其他运载工具。也应当关注测试仪器、遥感勘测、数据显示设备与记录设备的质量与校准。如果在测试中收集到的数据不可信，那么测试就毫无价值。

在这一阶段，可应用的测试类型有：

- 系统验收测试；
- 质量鉴定测试；

- 安全执行测试；
- 生产和维护测试设备的测试；
- 由政府官员进行的检查和/或测试。

可应用的测试技术有：

- 实际情况测试；
- 随机测试；
- 故障引入。

在这一阶段，使用的典型工具有：

- 环境测试设备，如空调室、振动表等；
- 数据采集和记录设备；
- 遥感勘测；
- 数据分析工具；
- 故障引入工具。

表 13.6 对所有上面讨论的测试层次（使用 4.1.2 节和图 13.2 中的术语），以及逐步用实际部件替代模拟部件过程，给出了一个总结。

表 13.6 所有测试层次及逐步用实际部件替代模拟部件过程的总结

| 测试层次              |        | 嵌入式软件   | 处理器     | 嵌入式系统其他部分 | 环境  |
|-------------------|--------|---------|---------|-----------|-----|
| 单向模拟              | MT     | 模拟的     | -       | -         | -   |
| 反馈模拟              | MiL    | 模拟的     | -       | -         | 模拟的 |
| 快速原型法             | RP     | 试验的     | 试验的     | 试验的       | 实际  |
| SW/U, SW/I(1) SiL |        | 试验（主机）  | 主机      | 模拟的       | 模拟的 |
| SW/U, SW/I(2) SiL |        | 实际（目标机） | 仿真器     | 模拟的       | 模拟的 |
| HW/SW/I           | HiL    | 实际（目标机） | 实际（目标机） | 试验的       | 模拟的 |
| 系统集成              | HiL    | 实际（目标机） | 实际（目标机） | 原型        | 模拟的 |
| 环境测试              | HiL/ST | 实际（目标机） | 实际（目标机） | 实际        | 模拟的 |
| 生产前期              | ST     | 实际（目标机） | 实际（目标机） | 实际        | 实际  |

## 13.5 开发后阶段

原型与临近生产阶段最终使系统可以发布并投入生产。这就意味着被测系统有着合格的质量，可以销售给客户。但是，组织如何能够确保所有生产的产品都

有着同样的质量等级呢？换句话说，即使发布成功能投入生产，但如果生产过程的质量不合格，那么生产的产品质量将不合格。因此，在大规模生产之前，组织可能需要采取更多的措施，使得生产过程是可控的，并保证所制造产品的质量。

应当考虑采取下面的措施：

- ※ **生产设备的开发和测试** 在嵌入式系统的开发中，对已发布的生产设备进行开发和测试可能同样很重要。生产线的质量对嵌入式系统质量有着较大的影响。因而很重要的是要承认这一措施的必要性。生产线的开发和随后的测试被定义为开发后活动，但也可以在嵌入式系统开发期间的任何时间进行，不过不能在建模阶段进行，因为这时还缺乏系统的最终特性。另一方面，生产线必须在实际的生产开始之前可用。
- ※ **首件产品检查** 有时候，需要对第一套生产单元进行首件产品检查。依据最终规范、变更请求、生产图纸、质量标准来检查单元，以确保生产单元符合所有的规范和标准。在多V模型中（见第3章），这可以被视为最后的“V”，其目的是开发和测试生产过程。毕竟，如果最终产品的质量不过关，那么所有的开发和测试工作都将失去意义。
- ※ **生产和维护测试** 出于质量控制目的，可能需要对生产单元执行测试。这可能是对每一个单元进行测试，或是对生产样品进行更为详细的测试。可能需要在现场有内置的测试设备来发现并修理故障以及维护。用于生产和维护测试的设备必须是在单元（测试设计）阶段就设计出来，而且在开发过程中得到测试。此外，开发测试和开发测试结果可以被用做生产和维护测试的基础。

# 第 14 章 工具

## 14.1 介绍

软件已经成为嵌入式系统的最大部分，而且软件的重要性仍然在增加。嵌入式软件变得越来越复杂，越来越多地在关键系统中出现。这对测试产生了重大的影响，有更多的东西需要测试，而且这变得更为复杂。除了嵌入式系统本身发生变化以外，市场也在发生变化。产品推向市场的时间和产品质量成为主要的竞争因素。这些意味着越来越复杂的软件，必须在更短时间内以更高质量开发出来。因而，对测试的时间压力和质量要求增加了。充分结构化的测试过程再也无法独自在这样短的时间内满足质量要求。这些以及系统的复杂性使得今天的测试工作几乎不可能不使用测试工具。正确地使用测试工具，就可能加快测试的进度，而同时又能保证质量。

测试工具是可以向一个或多个测试活动提供支持的自动化资源，例如计划和控制、详细设计、构造初始化测试文件、执行测试以及评估（Pol 等, 2002）。

测试工具的使用本身不是目的，它们必须对测试过程提供支持。只有在时间、金钱或质量方面得到益处，使用测试工具才有意义。在非结构化测试过程中引入测试工具很少能够取得成功。结构化和自动化对高质量测试都是必要的。要想成功地引入测试自动化，就必须有生命周期和关于所有测试活动的知识。测试过程必须是可重复的。

可以通过以下途径找到关于测试工具的最新信息：

- 会议和展览
- 提供商展示工具的免费试验版
- 研究报告，如 CAST（Graham 等, 1996）
- Internet
  - [www.ovum.com](http://www.ovum.com) (ovum 报告)
  - [www.testing.com](http://www.testing.com) (Brian Marick 测试基金会)
  - [www.soft.com/institute/HotList](http://www.soft.com/institute/HotList) (SRI)
  - 大多工具提供商都有详细介绍其（商业）产品的网站

### 14.1.1 优点

只有在时间、金钱或质量方面有益处，那么才应当使用测试工具。在测试执行期间，如果使用测试工具有充分的准备，那么通常能节省时间。充分的准备会花费大量的时间和金钱，这应当在关键路径以外来完成。由于需要长期的准备，因而需要用时间方面的投资才能够在金钱方面获利。

因为可以再三反复而又准确地按照同一种方式来执行测试，所以测试质量将会保持在同一等级。由于某些类型的测试只能够使用测试工具来进行，因此可以提高测试的质量。

测试过程的自动化提供了下面的优点：

- 开发好的测试自动化可以使测试不需要有人看管，无需使用人力就可以在夜间和周末进行测试，从而可以增加可用的时间。
- 令人乏味的日常测试活动可以由测试自动化来代替。由于这些测试活动（由人执行）易于出错，因而（自动化）将可以提高测试的质量。
- 建立回归测试集，可以使测试团队将时间集中于测试新功能或有改动的功能。
- 测试自动化确保同一个测试在每个时刻，都可以准确地按照同样的方式得到执行。
- 工具可以更好地检测实际输出和预期输出之间的差异，例如对视频图像的详细检查只可能通过工具来进行。
- 在测试工具的帮助下，可以生成大量的测试数据。

---

## 14.2 测试工具的分类

测试生命周期中的每个阶段，都有工具可供使用。图 14.1 给出了如何根据工具在测试生命周期中的使用位置来对工具进行分类（见第 6 章）。下面段落中将更详细地解释这些工具。这里的列表并不是穷举出所有的工具。

大多数工具是由不同的公司开发并推向市场的，然而，也可以在组织内部为项目开发专用的工具。

### 14.2.1 计划和控制

用于计划和控制阶段的大多数工具是基于项目管理的。这些工具有除了测试和缺陷管理工具以外，都不是专门为测试而开发的。下列工具可以为计划和控制提供支持：

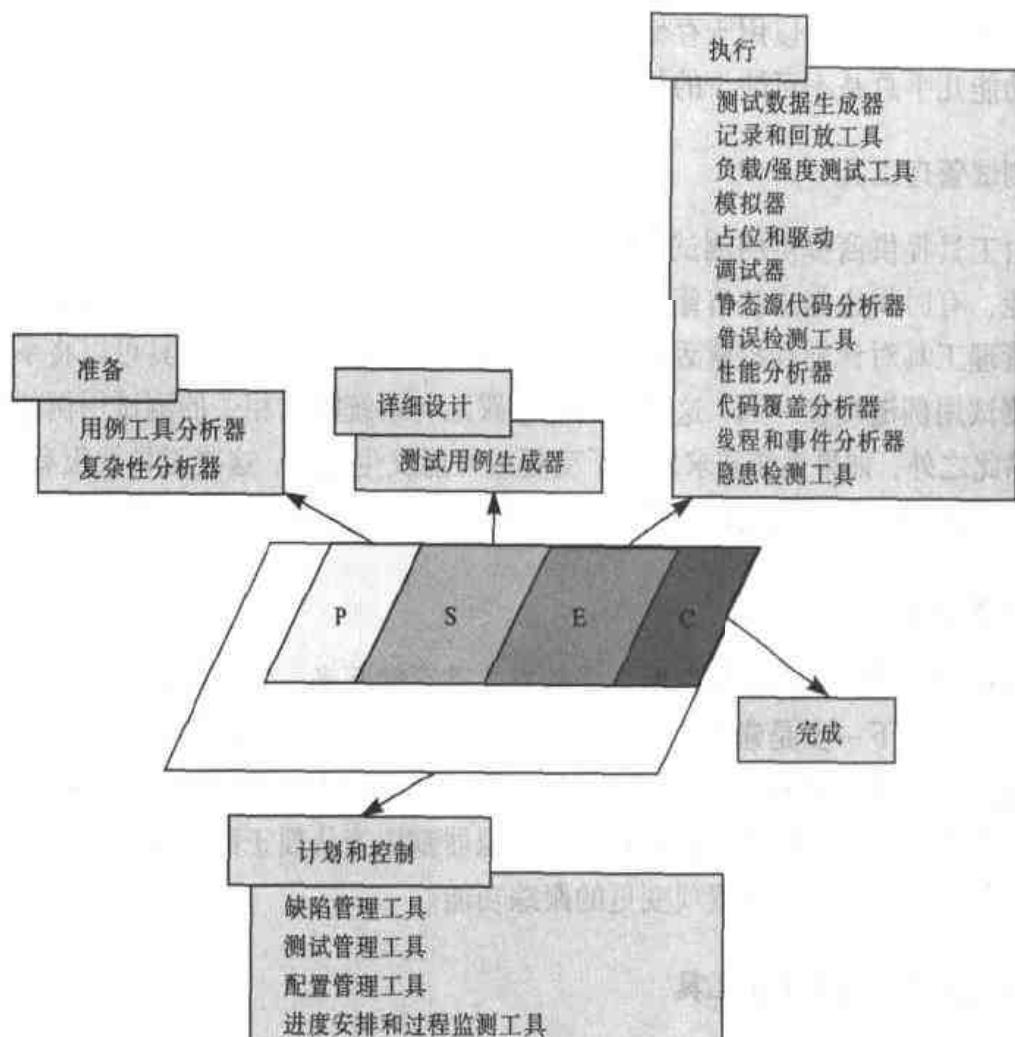


图 14.1 与测试生命周期相关的测试工具

- 缺陷管理工具；
- 测试管理工具；
- 配置管理工具；
- 进度安排和进度监测工具。

### 缺陷管理工具

测试过程期间发现的缺陷必须按顺序整理。对于小型项目，日常的文件系统和一些控制规程就足够了。比较复杂的项目至少需要一个数据库，要能够生成进展报告，给出诸如所有缺陷的状态，或已解决的缺陷与出现的缺陷的比例等信息。有许多工具提供商已经开发了缺陷管理系统。所有这些工具都基于数据库系统，而且有跟踪和报告功能。这些工具可以实现具有安全与访问级别的工作流。多数工具还具有 e-mail 功能，有的可以通过 web 来使用。

缺陷管理系统可以用于存储缺陷、跟踪缺陷并生成进展和状态报告。跟踪和报告功能几乎总是不可缺少的控制手段。

### 测试管理工具

由工具提供商提供的测试管理工具只提供存储测试用例、测试脚本和测试方案功能，有时候也集成缺陷管理，但没有工具来存储测试计划和项目进度。这类测试管理工具对计划和控制活动没有什么用处。现在，已经有工具可以将系统需求与测试用例链接在一起。这些工具能够跟踪与系统需求相关的测试用例的覆盖率。除此之外，如果系统需求发生了变更或可能发生变更，这些工具就很有用处，测试经理能够用它们说明展示这些变更的影响，从而可能阻止这些变更发生。

### 配置管理工具

所有的测试文档都必须按照正确的方式被存储起来。测试文档应当在质量检查之后冻结，下一步是建立测试件的配置项并存储在配置管理工具中。除了通常的测试件（测试用例和测试文档）以外，测试环境描述、使用的工具、校准报告以及专门开发的占位程序、驱动程序和模拟器都应当从属于配置管理。配置管理工具提供了对配置项和配置项变更的跟踪功能。

### 进度安排和过程监测工具

对于进度安排和过程监测，有各种各样的工具可以专门用于项目管理。在大多数工具中，进度安排和过程监测合并在一起。对测试经理来说，这类工具是很常用的。将它们与缺陷和测试管理工具得出的信息综合在一起，测试经理就能够跟踪进度，在必要情况下可以得到十分详细的信息。有时候，这些工具十分复杂，不容易使用。所以，这些工具会花费大量的时间，单个测试人员无法使用。测试人员必须用很简单的工具来跟踪进度。多数情况下，一个简单的电子报表就足够了。测试经理可以使用由这些电子报表提供的信息来更新进度安排。

## 14.2.2 准备阶段

### CASE 工具分析器

CASE 工具，如 UML 工具等可以进行一致性检查。一致性检查可以用于检查设计是否有遗漏和不一致。当 CASE 工具被用于设计时，可用来支持测试基础的可测性审查活动。

### 14.2.2.2 复杂度分析器

这种工具能够给出软件复杂度的指标。复杂度可以指示错误几率，也是需要彻底进行测试的测试用例数的指示器。度量复杂度的一个例子是 McCabe 循环复杂度度量 (Ghezzi 等, 1991)。度量  $C$  被定义为：

$$C = e - n + 2p$$

这里的  $e$  是指边数， $n$  是指节点数， $p$  是指连接的部件数 (通常为 1)。 $C$  定义了一个程序中线性独立路径的数目。对没有 GOTO 语句的结构化程序来说，循环数等于程序中的条件数目加 1。开发人员应当尽可能使复杂度降低到可接受的程度。McCabe 并不是惟一提出方法来描述软件复杂度的人，另一个有名的方法是 Halstead 理论 (Ghezzi 等, 1991)。

### 14.2.3 细化阶段

#### 测试用例生成器

测试设计需要进行大量的工作，使用工具来进行测试设计将是非常好的。不幸的是，没有多少可用的工具。为了使用测试设计工具，系统需求规范就必须使用一种形式语言 (大多数时间都是基于数学符号)。这种形式语言的一个例子是 Z 语言。

模型检查可以基于无穷状态机来生成测试用例。模型检查是一种技术，目前仍然没有商业工具可用。

采用统计使用测试时，也可能使用测试用例生成器 (见 11.7 节)。

### 14.2.4 执行阶段

在执行阶段，有许多类型的工具可供使用。许多工具提供商既提供一般工具，也提供高度专业的工具。这一章讲述下列类型的工具：

- 测试数据生成器；
- 记录和回放工具；
- 负载/强度测试工具；
- 模拟器；
- 占位程序和驱动程序；
- 调试器；
- 静态源代码分析器；

- 错误检测工具；
- 性能分析器；
- 代码覆盖分析器；
- 线程和事件分析器；
- 隐患检测工具。

### 测试数据生成器

这类工具可以在一个预先定义的输入范围内，生成大量的不同输入。输入被用于测试系统是否能够处理输入域内的数据。当不能预先确定系统是否能够处理在某个特定输入域内的所有数据时，使用这种类型的测试。进化算法就是基于自动化测试数据生成的（见 11.6 节）。

### 记录和回放工具

这些工具对于建立回归测试很有帮助。测试执行的大部分可以使用这类工具完成，测试人员可以将精力集中在新功能或发生变更的功能。建立这些测试需要花费大量的工作，而且总是存在维护问题。为了使这类工具能够在时间、金钱或质量方面有收益，就必须有一些反应快捷的组织和技术决策。13.5 节讲了如何来组织一个测试自动化项目，附录 C 讲述了如何从技术上实现测试自动化。

### 负载/强度测试工具

必须能够在不同的负载条件下正常运行的系统，必须测试其性能。负载和强度测试工具能够模拟不同的负载条件。有时候，人们可能只对负载下的性能感兴趣。而有的时候，人们可能更感兴趣的是在负载或强度条件下系统的功能行为。后者的测试必须和功能测试执行工具结合起来进行。

使用这类工具的另一个原因是加速系统的可靠性测试。一般的可靠性测试将可能花费几天左右的时间。采用负载和强度工具，可以在 12 或 24 小时之内，90% 的时间都是最大负载的情况下完成系统的测试。这应当能够模拟出几天时间内的 一般使用情况。

### 模拟器

模拟器是在受控条件下测试系统。模拟器被用于模拟系统的环境，或模拟与被测系统相连的其他系统。模拟器也被用于测试那些在非常危险的条件下，无法进行实际情况测试的系统，例如核电厂的控制系统。

有时候，可以在市场上购买到模拟器，但大多数情况下必须开发和建造专用的模拟环境，对系统进行测试。参见第 13 章关于模拟的详细信息。

## 占位程序和驱动程序

两个系统部分之间的接口测试，只有当两个系统部分都准备就绪后才能进行。这将给测试时间带来严重的影响。为了避免这个问题，尽可能早地对一个系统部分进行测试，可以使用占位程序和驱动程序。占位程序被被测系统调用，提供另一个系统部分应当给出的信息。驱动程序调用系统部分。为了更好地利用占位程序和驱动程序，可以使用标准化和测试床架构。构造的测试床几乎可以对每一个单元进行测试（见图 14.2）。测试床可以在集成测试期间被复用。

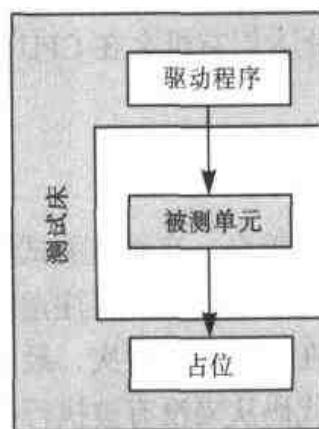


图 14.2 利用占位程序和驱动程序来对一个单元进行测试的测试床

如果是按照标准接口来使用占位程序和驱动程序，那么可以开发可用于每个被测单元的测试床。如果只使用一个测试床，那么使单元测试自动化就会变得非常容易。

## 调试器

调试器可以让开发人员有机会在受控条件下，在编译之前来运行程序。调试器能够检测出语法失效。调试器可以一步步来运行程序，并设置、监视和操控变量。调试器是开发工具中的一个标准部分。

## 静态源代码分析器

通常，在项目期间要使用编码标准。这些标准定义了诸如代码布局、条件之前和之后应当带有描述，应当用注释来解释复杂的结构等编码信息。可以使用更多的标准来预先强制使代码有一定的稳定性，禁止采用那些易于犯错、令人混淆的或稀奇古怪的结构，避免高度复杂的编码风格。这些标准可以是强制性的，也可以通过静态分析器来检查。这些分析器已经包含了一组编码标准，但这个集合可以由用户自己定制。分析器对软件进行分析，指出违反标准的代码。要注意的是，编码标准依赖于所使用的编程语言。

### 错误检测工具

这类工具被用于检测运行时错误。工具检测缺陷并对其诊断。它生成的错误信息不是模糊的、隐晦的，而是基于诊断给出一个确切的问题描述。这有助于定位源代码中的故障并加以解决。

### 性能分析器

当负载和强度测试工具被用于系统级的性能检测时，可以使用这类工具给出算法级的资源使用情况。这些工具能够给出算法的执行时间，以及内存与 CPU 的使用情况。所有这些数字可以让人们有机会在 CPU 使用、内存使用或执行时间方面来对算法进行优化。

### 代码覆盖分析器

这些工具给出测试用例的覆盖率。它对由测试用例执行触发的代码行进行监视。代码行是测试用例覆盖率的一个度量。要注意的是，代码覆盖率并不适合于度量测试集合的质量。例如，在单元测试层次，最小要求应当是百分之百的语句覆盖（否则将意味着有一部分代码从来没有被执行）。

### 线程和事件分析器

这类工具可以对运行时多线程中的问题检测提供支持，这些问题会降低 Java 程序的性能和可靠性。这些工具有助于检测出线程负载过轻、过载、死锁、没有同步访问数据及线程溢出等问题的原因。

### 隐患检测工具

这些工具能够对可能引发故障的隐患进行检测。隐患类型包括内存泄露、空指针调用、错误的存储单元分配、越界访问数组、变量没有初始化或者可能被零除。工具检测的是可能的隐患，这意味着需要进一步分析来确定它是否是真正的隐患。

## 14.2.5 完成阶段

完成阶段使用的工具基本上与计划和控制阶段使用的工具相同。虽然在完成阶段，关注的焦点是归档和提取结论，而不是计划和控制。配置管理工具用于存档测试件、基础设施、工具描述和校正报告等；时间安排和进度监测工具用于得出项目度量以用于评估报告；缺陷管理工具用于得出产品度量，有时候可以提供项目度量信息。

# 第 15 章 测试自动化

## 15.1 介绍

对产品进行测试是将产品推向市场的关键步骤之一，而使用测试自动化则可以减少测试执行所需的时间，测试自动化可以赢得时间、金钱和/或质量，它也可以替代重复性的令人乏味的测试。而且如果没有测试自动化，诸如统计使用测试和进化算法等测试将不可能进行。

从理论上讲，几乎每一个测试都可以自动化。而在实际中，只有很小一部分测试能够自动化。测试自动化经常用于以下情况：

- 测试必须要多次重复；
- 有大量输入变量的基本测试；每次执行的步骤相同，但数据不同（例如进化算法）；
- 十分复杂或易于出错的测试；
- 测试需要专用设备来生成合适的输入、激活系统和/或捕捉并分析输出。

系统、设计和需求经常会发生变更，这也对自动化测试的可用性造成了威胁。这些变更造成的结果可能是：

- 额外的测试用例；
- 测试用例发生变更；
- 不同的结果检查；
- 系统接口发生变更；
- 功能发生变更；
- 信号不同；
- 目标平台不同；
- 内部的技术实现不同。

如果测试集合能够被多次使用，那么自动化测试就能够节省金钱和时间（如果完整的测试集合执行 3 至 6 次，就认为有收益了）。这意味着自动化测试应当被用于产品的连续发布或用于开发的不同阶段。这也就意味着自动化测试必须被设

计为能够处理由于系统、设计和需求的变更引起的不确定性。

本章首先讲述对于一般商业化工具和专有工具，如何从技术上实现自动化测试。然后讲述引入自动化测试是如何类似于开发过程，也有着需求、设计、部署和维护阶段。

## 15.2 测试自动化技术

### 15.2.1 可维护性设计

为了得到最大程度的可维护性、灵活性和易使用性，必须做出一些设计决策。为了防止测试包被束之高阁，其设计应当尽可能不依赖于变更，包括测试对象的变更、测试包和被测系统之间的技术连接，以及使用的测试方法和数据。这是通过以下减少依赖性的措施来实现的：

1. 测试数据存放在测试包的外部。
2. 测试动作的描述与这些技术实现是分离开的。
3. 处理自动化测试的过程的实现不依赖于测试环境和被测系统。这可以解释为什么任何测试包都需要有一些基本部件。
4. 被测系统与测试包之间通信的实现不依赖于测试包的核心过程。

下面部分将会更详细地来讨论这些要点。

本章中将用到以下术语：

- **自动化测试**：用自动化测试包来执行测试。
- **测试包**：执行测试所需要的只是按一个按钮。
- **数据驱动**：实际测试动作与测试数据相分离的一种方法，测试数据是测试的推动力。
- **框架**：可复用的模块和设计库。

#### 测试用例

测试用例被存放在测试包的外部（见图 15.1），可以存储在数据库或电子报表中。增加额外的测试用例意味着只需要将该测试用例加入到测试用例的存储区中，而无需改变测试包。

#### 测试动作

特定测试动作的技术实现可能相当复杂。对于测试包的一般用户，技术实现

应当被隐藏起来。一般用户必须知道的惟一一件事情就是有哪些测试动作可用(见图 15.2)。通过选择正确的测试动作，用户就可以组合出正确的测试脚本。如果被测系统的功能保持相同，但技术实现发生了变化(例如使用另一个接口)，那么只需要更改技术实现。用户仍然可以使用同样的脚本，就好像什么都没有发生一样。表 15.1 给出了一个将测试数据和测试动作组合成测试脚本的例子。这种组合方式是很常见的。

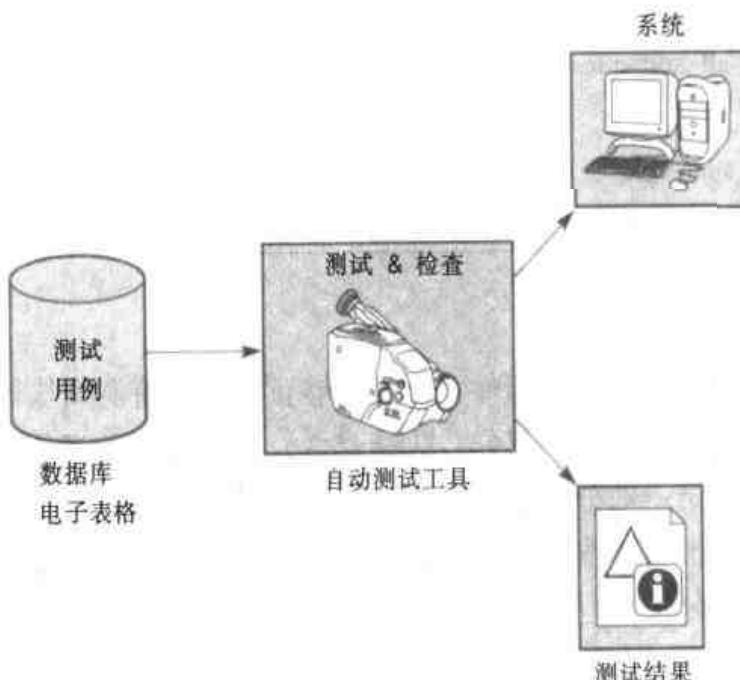


图 15.1 测试数据被存放在测试自动化环境的外部

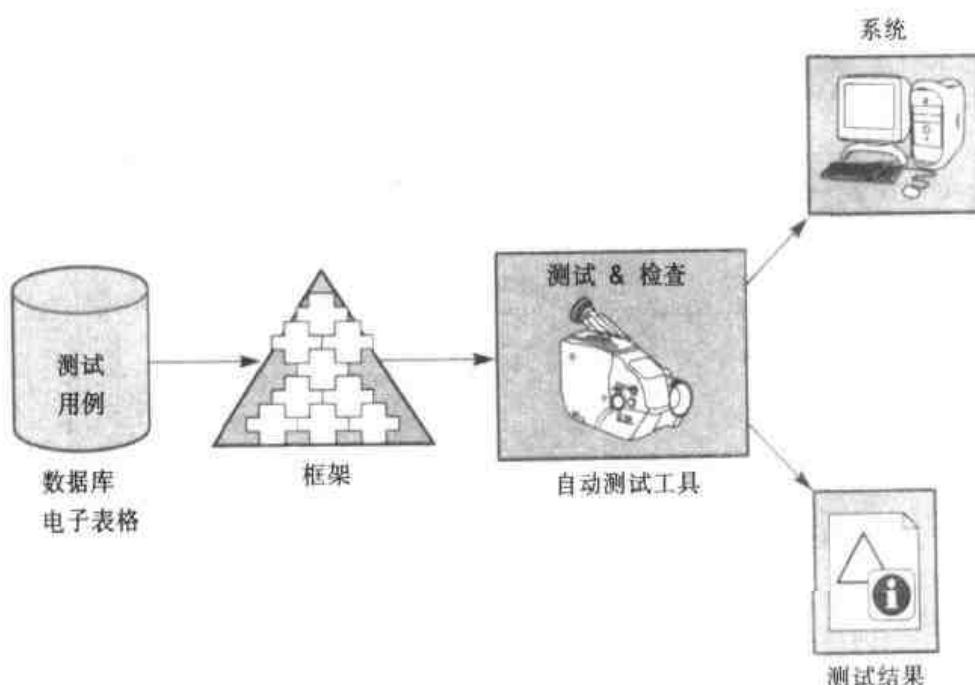


图 15.2 测试动作的技术实现被存储在一个框架中，该框架对测试包的一般用户是透明的

表 15.1 一个测试脚本方案。当第一列为空时，其他行表示注释

| 标识   | 测试动作                 |         |      |            |      |
|------|----------------------|---------|------|------------|------|
| TC01 | Open_address_book    |         |      |            |      |
|      |                      | 名字      | 姓氏   | 街道         | 门牌号码 |
| TC02 | Add_address          | Stewart | Mark | Oak street | 15   |
| TC03 | View_address         | Stewart | Mark | Oak street | 15   |
| TC04 | Delete_all_addresses |         |      | 校验         |      |
| TC05 | Count_addresses      | 0       |      |            |      |
| TC06 | Close_address_book   |         |      |            |      |

### 基本部件

测试包必须包含一些基本部件，以便能够正确地支持测试执行过程。这些基本部件是测试包通用设计的一部分（见图 15.3），我们称之为蓝图。设计方案由三个主要部分组成，第一部分是带有测试方案和测试脚本的输入方，第三部分是带有状态报告与进度报告的输出方，在输入与输出之间是处理单元或测试包的核心部分。这个核心部分使得只需要按一个按钮，测试就可以开始执行。测试包在没有人工干预的情况下一直工作，直到执行完成整个测试方案（见 6.4.3 节）。附录 C 给出了更为详细的设计方案的描述。

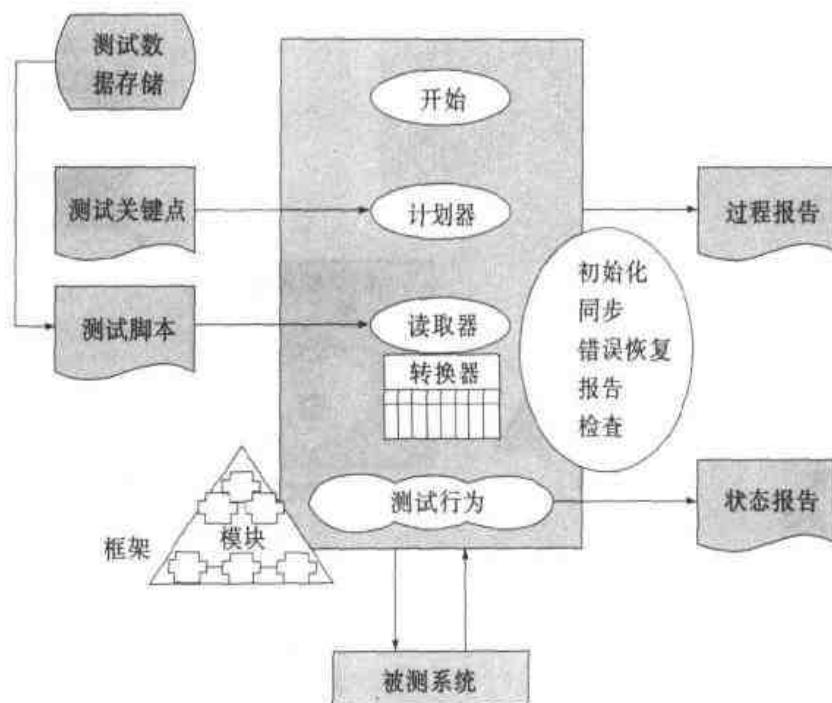


图 15.3 一个测试包的蓝图

## 通信

被测系统可以有以下存在形式：

- 在与测试包安装的同一台机器上，以可执行代码形式存在；
- 在测试目标机上，以可执行代码形式存在（与测试包环境分隔开）；
- 以模拟环境中的被测系统形式存在；
- 以作为临近生产的被测系统形式存在。

这里提到的形式是与开发过程中的不同阶段相关的。所以，在测试过程中，被测系统的形式在不断地发生变化。在开发过程的所有阶段中，会用到许多测试用例。这意味着通信层应当不依赖于所定义的测试用例。当通信层的通信方式或技术实现发生变化时，测试用例将保持不变。要能够满足所有形式的被测系统，就必须有一个通信层（见图 15.3）。

### 15.2.2 维护

可能会由于发生某些变更，无法正常、正确地运行自动化测试，因而有必要进行维护。蓝图就是为了使维护降到最低而开发的。变更可能使下面部分受到影响：

- 测试数据；
- 同步、错误恢复、检查和初始化；
- 应用程序特定的模块；
- 通信层。

在本章的介绍中，给出了一个可能的变更列表。下面列出这些变更对测试包的影响：

- 增加测试用例：只有数据存储必须改变。
- 测试用例改变：数据存储必须改变。有时候，在框架中有必要增加新的测试动作。
- 不同结果检查：如果只有输出预测发生了变化，那么只有数据存储必须改变。否则，必须改变或增加检查和/或测试动作。
- 系统接口改变：通信层必须改变。如果信号定义发生了改变，而且如果测试数据包含信号信息，那么测试数据也应当改变。
- 功能改变：功能改变可能对测试数据产生影响，因为必须增加新的测试用例。同步、数据恢复、检查和初始化可能也要改变，因为系统会话和功能顺序可能都发生了改变。新功能意味着新的测试动作。新功能也可能影响

到通信层。

- **不同的信号：**通信层中必须有改变。有时候也有必要改变测试数据。
- **不同的目标平台：**这可能影响到通信层。
- **不同的内部技术实现：**如果外部接口和功能没有改变，那么这只会对非插件式的软件通信实现造成影响。

### 15.2.3 可测性

为了使测试自动化更为容易，在系统的设计和实现过程中，就必须认识到可测性是一个质量特性。建造一个专用的通信层特别耗时而且代价高昂，因为无法对系统的进行监控。一些测试可以由自身内嵌的测试来代替。有时候，只有在测试过程中才能够访问一个特定管脚的输出。而在生产中，软件中的一个参数或硬件中的一个移位变化，就可能关闭某个选项。

---

## 15.3 实现测试自动化

测试工具被用于提高质量、降低测试执行费用和/或节省时间。在实际中，对功能测试来说，这并不总是那么容易实现。发布同样一个产品，自动化测试执行一次或两次都不成问题。问题是发布多个产品，采用大致相同的测试就不那么容易了。问题在于系统或测试要求发生了变化，而自动化测试不再能够执行或覆盖测试要求，无法再实现质量、费用和时间等目标。

对这些类型的工具实现进行的投资是巨大的，至少应当对完整的测试集合执行 3 至 6 次才能得到回报。这也将意味着，在开始使用测试自动化时，人们就应当知道这是一项长期的项目。可以通过使用生命周期模型，以及正确的分析活动和决策点来克服大多数问题（见图 15.4）。这里的生命周期有三个连续阶段：

- 初始；
- 实现；
- 应用（exploitation）。

初始化阶段只是进行分析和调查的一段时间。这一阶段将确定出测试自动化的目标，以及实现这些目标的可行性。如果不了解合适的测试工具的可用性，那么可以开始对工具粗略搜索。有时候，详细的测试选择也包含在这一阶段。专用工具的开发就可能是这一搜索的结果，这将对实现目标的可行性有着重要的影响。在这一阶段，将正式决定是否开始实现测试自动化。

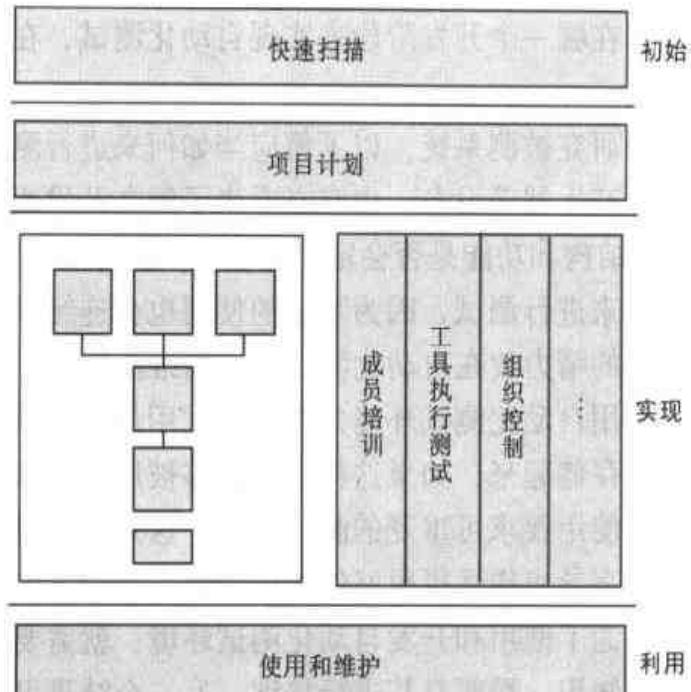


图 15.4 测试工具实现项目的生命周期

在实现阶段开始时,如果测试工具的详细选择活动不是初始化阶段的一部分,那么就在这一阶段来实施。这一阶段将量化测试自动化的目标,详细描述测试目标,分析被测系统所涉及的技术。该过程的输出更倾向于设计过程。根据设计来开发自动化测试环境。这一阶段的最终产品将是一个能够实现的测试自动化环境,这将能够实现测试自动化的目标。

在应用阶段,将测试工具环境用于测试执行。由于不断有变更以及新增加的功能,所以应当对测试环境加以维护。

### 15.3.1 初始话

测试自动化项目是从可行性研究开始的。该研究的主要目标是获取足够的信息,来决定是否应该进行测试自动化。为了获得正确的信息,需要对以下各项加以分析:

- 测试自动化目标;
- 被测系统;
- 测试组织。

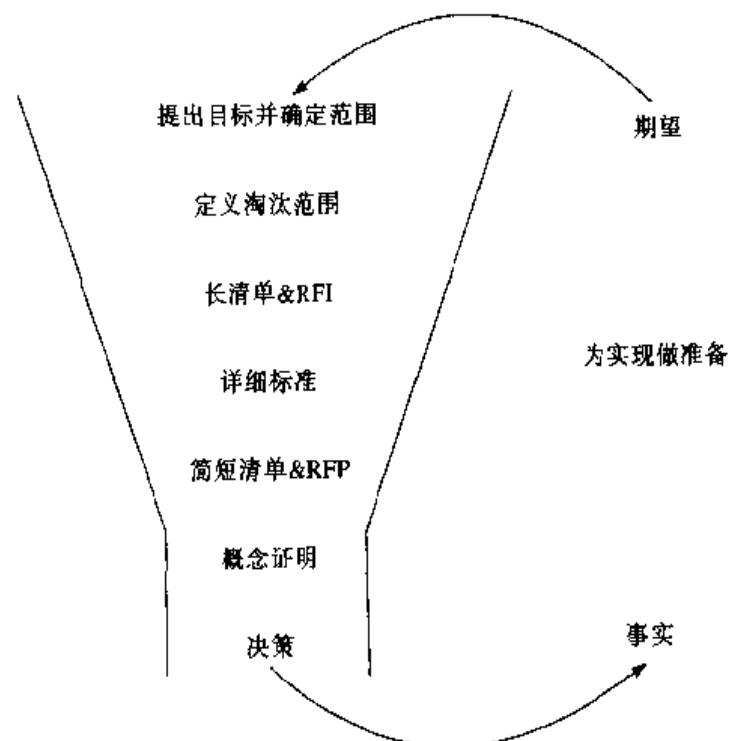
测试自动化目标就是组织对测试自动化的预期。可以用质量、费用和时间三个要素来描述目标。在这三个要素之间应当有一个平衡点。高质量通常也就意味着节省费用和时间。测试自动化目标也是判断测试自动化项目是否成功的标准。

其他目标还有：应当在哪一个开发阶段来实现自动化测试，在自动化测试环境中要对什么来进行测试。

要从技术层面来研究被测系统，以了解应当如何来进行测试，是否需要有更多类似模拟器和信号产生器等设备。也有必要来了解在开发或连续发布期间，系统的接口、内部技术结构和功能是否会定期地变化。系统在这些点经常发生变化，就很难使用测试工具来进行测试，因为系统的使用也会连续发生变化。这将使费用增高，需要将更多的精力放在自动化测试环境的维护性上。

测试组织应当使用自动化测试环境并提供测试用例。测试用例可能已经按照某个特定标准格式被存储起来，如果这种格式能够被用于测试自动化环境就更好了。测试工具的正确使用要求可重复的测试过程。这意味着可以通过使用测试设计技术和测试策略，来导出描述得很好的测试用例，它们决定着系统部件的相对重要性和质量特性。为了使用和开发自动化测试环境，就需要具备专门的知识。如果雇员不具备这些知识，需要对其进行培训。另一个结果可能是，通过开发一个直观的用户界面，使自动化测试环境的使用尽可能简单。该用户界面将隐藏自动化测试环境的复杂性。

除了所有这些主要条件之外，也应当了解是否有工具可以用于被测系统。这将依赖于被测系统、目标、被测系统的实例和编程语言。图 15.5 给出了测试工具选择过程中的不同阶段。



测试工具选择首先是从定义期望开始的。基于期望值和不同测试工具能力，最终做出决策。决策可以是没有合适的工具可用。最终结果是开发专用工具或不执行自动化测试。

下列活动将被执行：

1. 确定目标和范围。目标可以是改进测试过程，缩短测试执行时间，减少测试执行工作量或提高测试执行质量。要知道，类似于解决组织问题，或不需要给测试更多的关注等目标是不能由工具来实现的。范围是指自动化测试环境，是否应当只用于一个特殊系统、一组系统或公司内所有的系统。
2. 定义淘汰标准。这些标准主要是基于被测系统所涉及的技术。不能够满足这些标准的工具是没有用的，应该被立即从列表中删除。
3. 长清单& RFI。采用淘汰标准，准备出一个长列表。所有的工具提供商得到一个信息请求（RFI）。RFI是提供商应当回答的一个问题列表。答案的准确性也是对工具提供商服务水平的一个度量。没有回答的工具提供商将被淘汰。
4. 详细标准。准备出一份选择标准的详细清单，共有三类标准：
  - 技术（兼容性、稳定性和脚本语言等）
  - 功能（对象识别、同步、报告功能、操作系统和目标平台等）
  - 工具提供商的要求（连续性、培训、支持和安装基础等）这些标准被分组为淘汰标准、要求和希望三组。
5. 简短清单& RFP。根据详细标准，从长清单中筛选出满足标准的工具。将筛选出的工具放入一个简短清单中，将详细标准和建议请求（RFP）发送给工具提供商。RFP给出了关于费用、支持与培训可能性等指标。基于这个信息，邀请两到三家工具提供商，在一个演示测试环境中演示各自工具的能力。为演示准备一份能力清单，工具提供商用他们的工具来演示这些能力。
6. 概念证明。在实际使用工具和将大量工作花费在执行测试工具之前，可以开始一个试点。如果这个试点成功，那么就可以实现该工具。工具提供商大多都允许试用工具一段时间。
7. 决策。概念证明的结果已经可以给出足够信息，来判断该工具的实现是否能够满足期望值。结果可能是只有开发专用的工具才能够满足目标。

下一步将分析成本和收益，两者都有客观和主观属性。可以根据费用和使用的人力来度量成本，这里甚至应当做出某些估计。主观成本是指引入某些新东西和工作方式的改变。从通过花费更少的时间和人力用于测试方面来讲，好处也可

以被客观化。测试过程的质量提高是好处的主观部分。

要知道，工具的直接成本很高，但是在工具的实现、使用和维护期间，人力成本往往更加高昂。

整个过程还应当能够增强认识和积极支持实现。

### 15.3.2 实现

实现的主要目标是准备一个能够满足测试自动化目标的自动化测试环境。为了实现这一目标，要定义一些活动。第一个活动是准备一份行动计划，其目的是给出活动、涉及的人员和时间估计。

接下来将开始设计和开发。在设计和开发过程中，确定出六个活动（见图 15.6）。

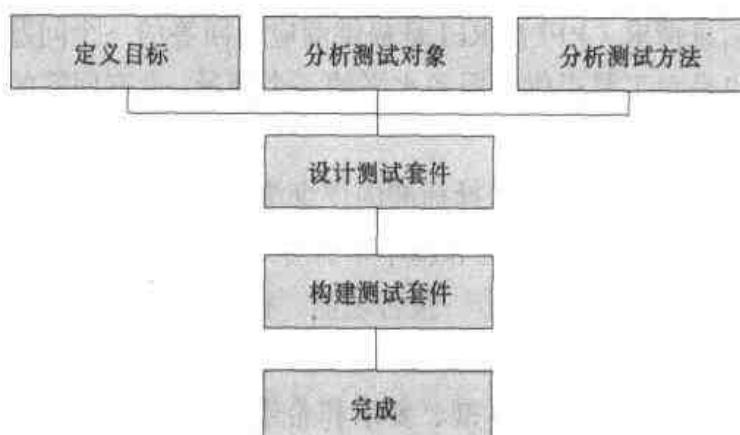


图 15.6 设计和开发阶段的活动

制定测试自动化目标、详细分析测试对象和测试方法对于设计决策是很有必要的。如果主要的着重点是测试过程，而不是测试细节，那么这就意味着可以在一个很高的层次上来制定测试行动。如果测试对象不是很稳定（意味着在开发过程和连续发布期间将有许多变更），那么测试自动化环境的设计就应当便于维护。如果主要的着重点是节省时间，那么只须将劳动和时间密集的测试自动化。

架构师要考虑到上述因素，基于设计方案来设计自动化测试环境，见附录 C。根据这个设计，工程师开发自动化测试环境，并负责实现必要的功能，以执行交付来的测试用例。基于测试设计技术，功能测试人员开发出测试用例。作为最后一步，对自动化测试环境执行最终测试。

现在，有关自动化测试环境的设计和所有其他文档都是最新的。这些文档与自动化测试环境一起，将被用于维护阶段。

除了这些主要活动以外，必要时可以执行一些并行活动。这些活动可能有培训人员使用测试工具、实现这些工具、维护这些工具和组织控制。

### 15.3.3 应用

现在就可以准备使用自动化测试环境了。自动化测试环境的可用性对测试过程的影响在图 15.7 中给出。自动化测试环境影响到测试过程的所有阶段。在引入测试基础时，头脑中要一直清楚自动化测试环境的影响，以及使用该环境的可测性。在细化阶段，应当用自动化测试环境可读的格式来描述测试用例，而且应当根据测试对象的变更来对该环境进行调整。测试执行的一部分被自动化测试环境覆盖，在完成阶段，所有的变更被冻结并保存起来。配置管理有一份完整的自动化测试环境拷贝是很有必要的。

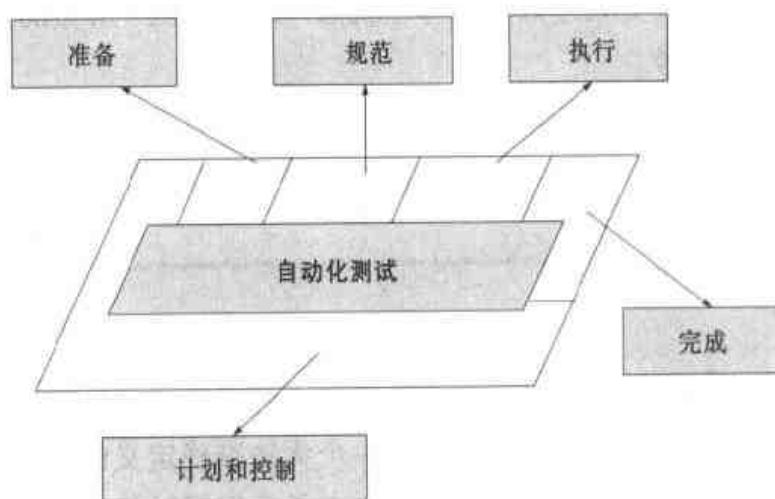


图 15.7 自动化测试的测试生命周期

应用阶段的主要目标是维护和使用自动化测试环境。在该阶段，组织会因自动化测试环境而受益。为了满足测试自动化的目标，并且能够从该环境获益，到下一个版本发布时，关注的焦点就必须是对该环境的维护。如果忽视了这部分工作，那么环境将在一次或两次发布之后，不再有用处，而这时投资可能还没得到回报。

# 第 16 章 混合信号

此部分由 Mirko Conrad 和 Eric Sax 撰写。

典型的嵌入式系统与现实世界相交互，通过传感器接收信号，并将输出发送给动作器来操纵环境（见图 1.2）。因为物理环境（本性）不是离散的，因而这样一个嵌入式系统就不仅要处理数字信号（就像“纯计算机”所做的那样），而且要处理连续（模拟）信号。

本章首先介绍典型的数字信号和连续信号的混合（“混合信号”），及其对测试的影响。然后讨论在输入端和输出端如何来处理混合信号，分别见 16.2 节和 16.3 节。

---

## 16.1 介绍

在连续环境中，没有什么可以与离散系统中的状态进行比较。不会由于事件而有状态转换和状态突变，整个系统是在一个无法准确定义的“状态”中。为了阐明连续系统和离散系统之间的这个差异，本段将根据时间和值来对信号分类。

### 16.1.1 信号的分类

信号通常有一个随着时间而变化的值。时间和值的变化都可以是离散的或连续的。利用时间/值和离散/连续的不同组合，可以将信号分为 4 类。这些在图 16.1 中描述。

这些信号在实际使用时有一些差异：

- 模拟信号在时间和值上都是连续的。这是信号的物理类型，例如在电气领域，可以表达为电压随时间而变化。对于激励描述来说，可以用一个函数来表达（例如电压 $=f(t)$ ）。
- 时间量化信号是只在具体时间点（采样点）才知道信号值。典型例子是捕获的诸如系统响应信号，它是由时间和值成对来表示的。
- （值）量化信号是指信号值为离散而时间为连续的信号。一个正通过数/模转换器，以进一步在处理单元中被分析的模拟信号，是这类信号的典型代表。

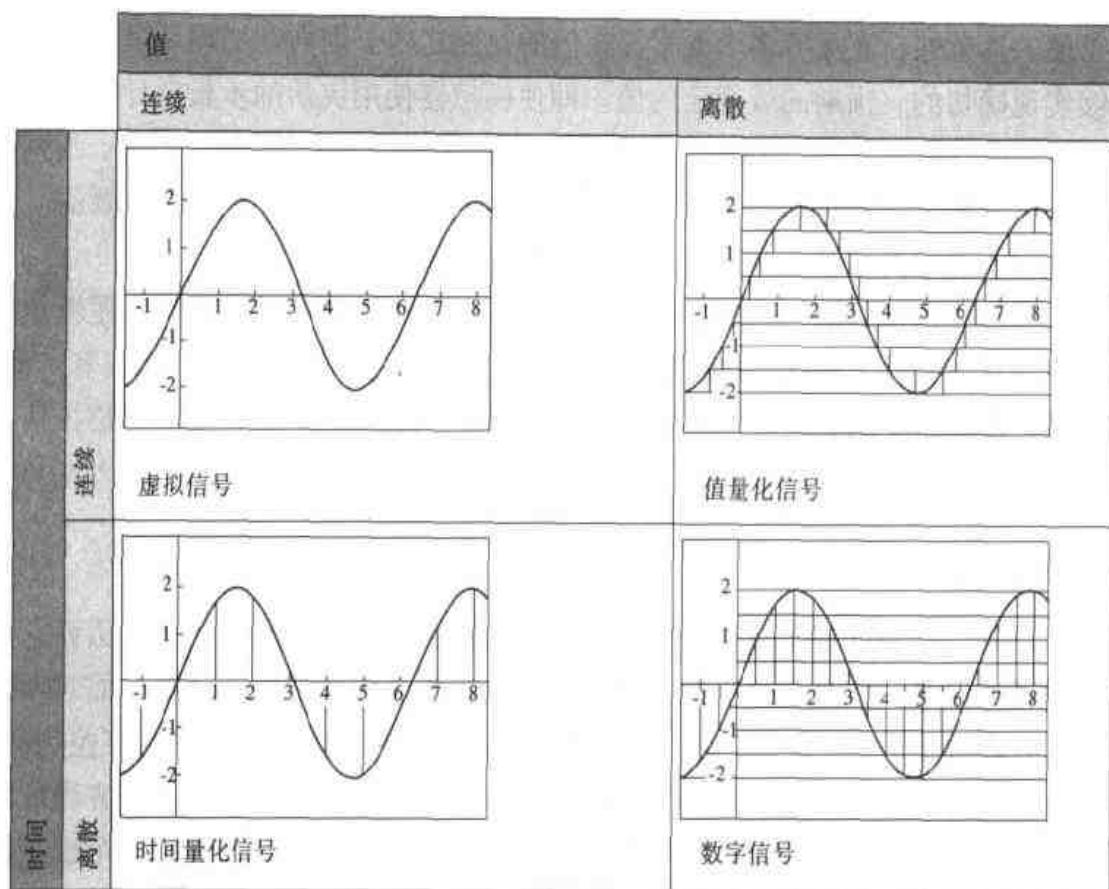


图 16.1 信号类别/信号变体

■ 数字信号的值只是一个有限集合中的元素。如果该集合只包含两个元素，那么这种特殊情况被称为二进制信号。由于半导体领域数据处理的方式，在嵌入式系统中，二进制信号对于内部信号处理和控制有着非常重要的意义。

此处上下文中，术语“混合信号系统”被用来指那些包含数字部分和模拟部分的信号系统。因而混合信号系统就是必须处理离散信号，也必须处理连续信号的系统。

为了描述在处理连续信号时遇到的典型问题，现在将进一步讨论纯模拟（连续）信号和混合信号的情况。

### 纯模拟信号

输入信号（激励）的定义和预期响应，以及捕获结果之间存在着根本性的差异。信号可以用数学方程来描述（例如  $f(x)=\sin(x)$ ），这可以很好地用于表示连续信号（见 16.2 节）。然而，一个被捕获的信号总是经过了采样，因而在时间上是离散的。此外，将同样是连续信号的噪声与捕获信号分离开，要远比数字信号更为复杂。因而，不可能准确地捕获一个模拟信号。在采样点之间总是有偏差存在。

由于值的连续性，有无穷多个元素都在值的范围之内。因而在物理环境中，很少能够实现确切的、预期的或规定的值。即使模拟器使用灵活的步长来计算复杂算法（例如微分数学方程，DAE），但通常也无法预测出准确的结果。因此，当判断是否可以将两个信号视为“相等”时，信号的容差和范围定义是很有必要的。例如，这对于确定测试结果是通过还是不合格来说就是很重要的。

当测试模拟部分时，一个主要的困难在于，模拟故障并不会导致在逻辑值中有简单状态变化。因此，判断是否有故障发生并不是一件简单的“高”和“低”的事情，而是需要用到限度（时间容差）和频带（值的公差）。因而，对模拟信号的测量和分析并不是一件简单任务，这将在 16.3 节详细讲述。

### 混合信号

二进制信号的处理是具体的：一个信号或者是“高”或者是“低”。因而它是可以明确地定义，并且有大小的，可以确切地检测出故障并执行计算。而对模拟信号的处理就不同了。但是，混合信号系统的显著特性就在于，它是数字控制器、计算机、与控制器耦合的被建模为有限时序机的子系统，还有用偏微分方程或常微分方程及微分方程来建模的环境的组合。因此，混合处理，尤其是两个域（数字与模拟）的同步就是一个很重要的事宜（见图 16.2）。

对内部信号处理来说，模拟信号和数字信号的组合是极具挑战性的。因而，在设计中，对时间的处理往往不同于模拟和数字信号中的时间。例如，在模拟环境下用两个不同的时间步长算法来解方程。基于时钟的数字部分是用一个步长来运行的，该步长定义了值和状态有具体变化的时间点。为了对描述模拟部分的 DAE 方程求解，就需要有连续的时间基准（见图 16.1 和图 16.3）。而如果在数字部分有事件发生并影响到模拟信号，那么就必须进行校准，需要同步两个不同的时间。通过倒退领先时间或等待后面的时间就可以进行同步。而且，当超过阈值或有触发脉冲发生时，就需要进行同步，并初始化模拟信号对数字信号的影响。

混合信号域的所有这些特定的方面，都需要有经过调整的和专门的测试度量。

#### 16.1.2 各种测试层次中的混合信号处理

在所有的开发阶段，都有测试发生（可见第 3 章和第 13 章）。在早期设计阶段，用于开发混合信号系统的基于模型的方法就提供了一个可执行的系统模型（即方块图和 VHDL）。该系统模型也可能已经由功能开发人员进行了动态测试（模型测试、模型循环或“MiL”，见图 16.4）。如果将模型看做是系统规格，那么就可以根据模型来生成软件。作为嵌入式系统软件开发的第一步，这一软件还没有在产品目标机上运行，但它已经是完整的控制软件。在这一层次上的动态测试也基

于执行（软件循环或“SiL”）。最后，软件被集成到目标控制器上以用于生产，测试方法在一个原型环境中（硬件循环或“HiL”）执行。也可以见第13章中与不同测试层次相关的不同测试环境。

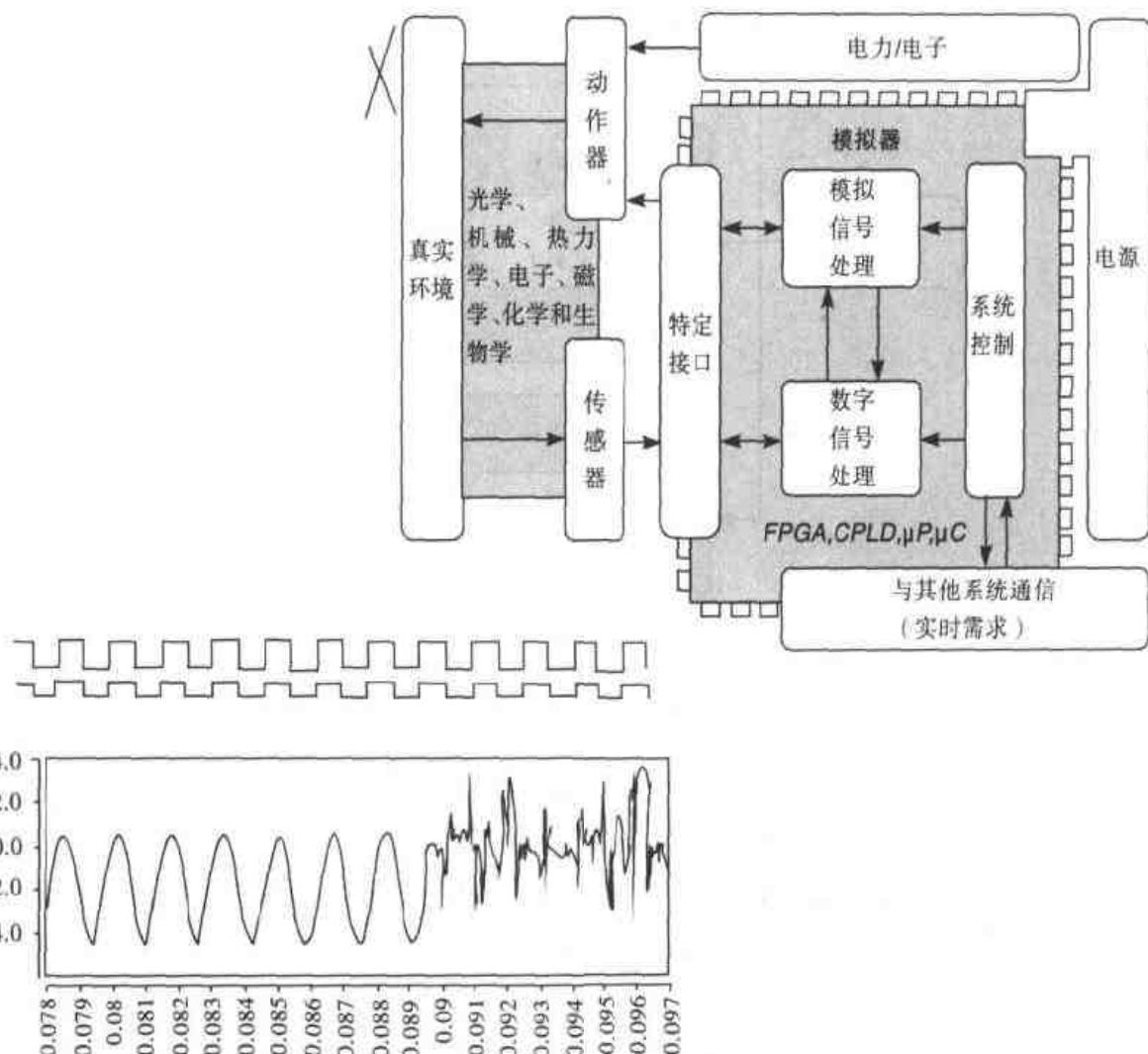


图 16.2 混合信号作为嵌入式系统的输入/输出

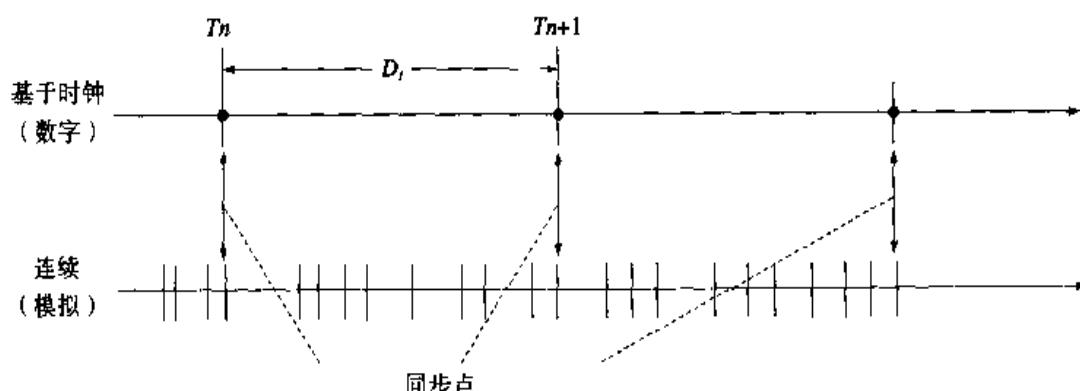


图 16.3 数字和模拟信号在时间上的同步

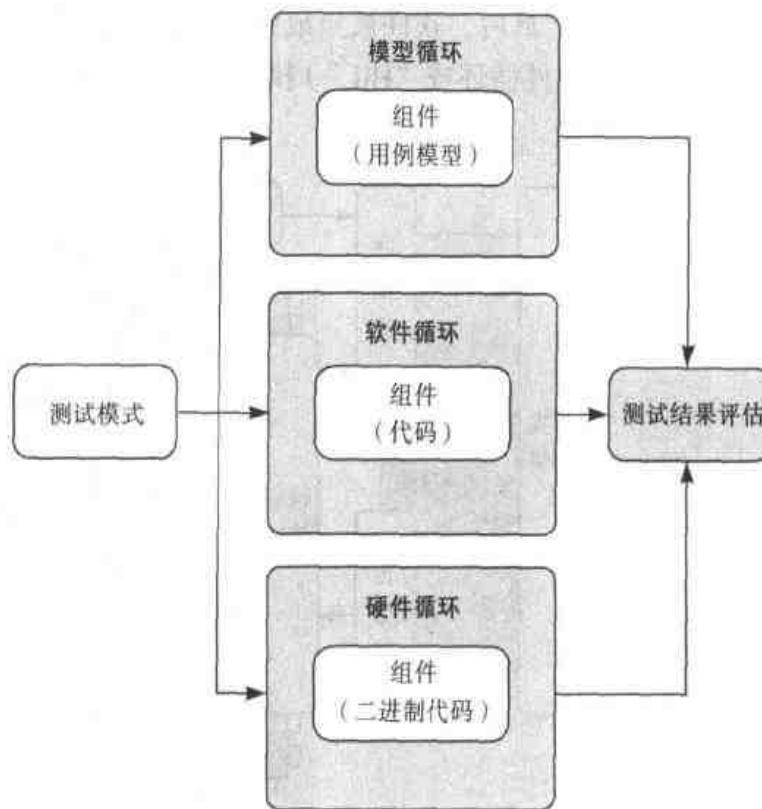


图 16.4 在不同测试层次中，测试输入和测试输出的复用

对所有这些测试层次使用测试信号（通常被称为范式）。信号的抽象描述（例如，逻辑值的改变或连续信号的数学描述）对所有的测试层次都是相同的。因而，在随后的开发阶段和测试阶段期间（例如 SiL、HiL），这些测试范式就可以被复用。而且，不同测试层次的测试结果可以被比较，并被复用为随后阶段的“预期结果”。

所以，对输入和输出逻辑描述的复用是可能的。但是对于不同的测试层次，信号的物理表示是不同的。当测试模型或采用 SiL 时，抽象的数据流就足够了。但对于 HiL，电压和电流就必须用具体的项来表示。

下一章，将更详细地来描述对工具和技术的分类，以便于描述输入信号（16.2 节）以及对测试输出的测量和分析（16.3 节）。

## 16.2 激励描述技术

这一章中，讨论用于混合信号选择的激励描述技术，提供标准来支持给定项目情况下对能解决问题的激励描述技术和工具的选择。

对嵌入式系统来说，激励描述就是一组由多个范围组成的模拟或数字信号

(见图 16.5)。数字信号是基于时钟的。与时钟相比，可以对上升时间和下降时间给出确切的计时延迟的定义。而且，信号值具体变化的格式可以被确定下来，例如依据以前的时钟来确定。对一个纯模拟信号来说，可以定义出基于符号表达式、偏差、相移和振幅的描述，以及功能表达式。最后，测试复杂系统的实际激励，需要连续描述多个独立定义的信号范围。

### 16.2.1 评估和选择标准

事实上，现有的测试方法和工具用到了一些完全不同的激励描述技术。作为描述技术多样性的例子，我们可以指出的就有（测试）脚本语言、信号波形、树形和表格综合标记 (ISO/IEC 9646, 1996)、消息顺序图 (ITU, 1996; OMG, 1997)、时序图和分类树 (Grochtmann 和 Grimm, 1993)。

尽管功能测试对于验证系统是非常重要的，并在业界被广泛使用，但是只有很少一些技术和工具可用于系统地基于方法来生成相应的激励描述。因此，在许多情况下，只有有限的工具和标记被使用。

为了支持对特定问题的激励描述技术，或对于一个具体项目要用的工具进行选择，需要有合适的标准。在本章对工具和技术的描述中，使用以下标准（并非完整列出），也可参见 Conrad (2001)：

1. 支持的信号类别：该技术支持 4 个信号类别（模拟、时间量化、值量化和数字信号）。
2. 标记类型：激励信号可以用文字（例如脚本或编程语言）、图形（例如消息顺序图或流程图）或两者的组合来描述。尽管文字描述通常提供更好的灵活性，但可视化往往能够增强对激励描述的理解。
3. 抽象程度：激励是能够采用低层次的抽象（即用离散化的时间-值对）来描述，还是在较高抽象层次上（即按步长来定义的函数）来描述，或是支持在不同抽象层次上的多层次描述？多层次描述的优点在于，测试人员能够首先将精力集中于要点（例如最有可能覆盖的值域）并对与本阶段不相关的细节进行抽象化。
4. 复杂性：它只能够描述信号的基本形式，还是有可能通过按顺序排列和编程（例如 if-then-else 循环），来将这些基本信号合并为复杂的信号波形？见图 16.6 中的例子。

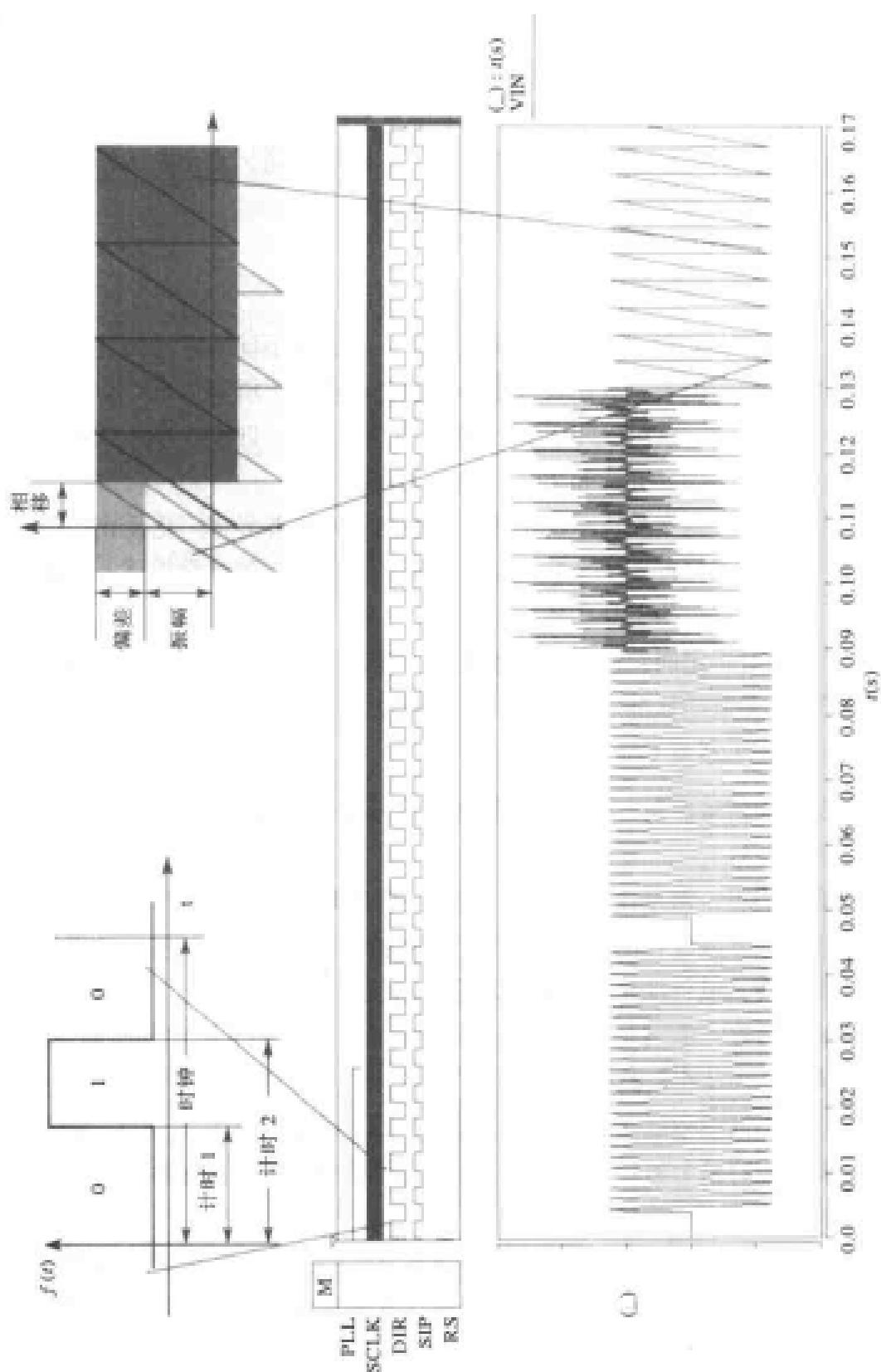


图 16.5 一个复杂信号定义的例子

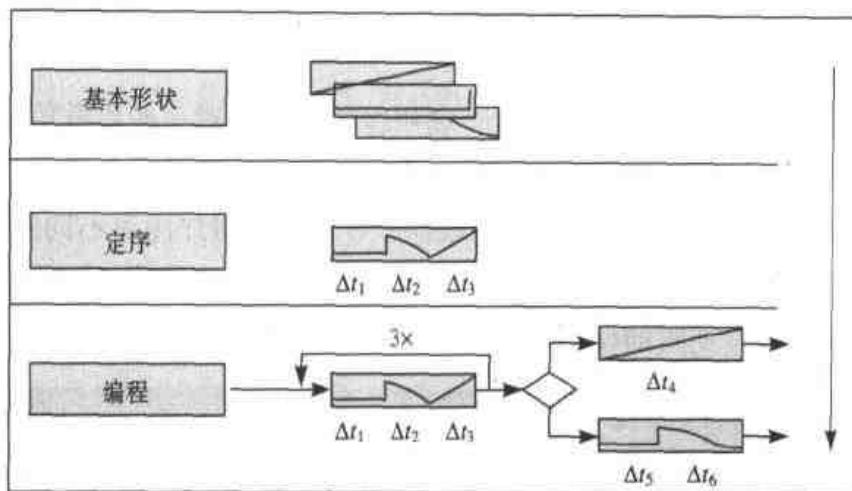


图 16.6 激励描述的复杂性等级

5. **输出的考虑:** 在测试运行期间, 可以访问被测系统当前的输出值, 并由此来改变激励生成吗? 换句话说, 激励的技术规范是作为开环来发生的(例如一个 *priori*, 而且没有反馈), 还是作为闭环来发生的(即激励信号的波形依赖于被测系统的状态并且只能在运行期间被指定)?
6. **覆盖范围标准:** 提供了确定测试覆盖范围, 尤其是用于更为抽象的描述层次的度量/测量吗?
7. **方法支持:** 有没有方法来支持激励信号的确定? 换句话说, 有专门的测试设计技术(就像在分类树方法中那样)可供使用吗? 这个问题的回答是很关键的, 因为许多常见的测试设计技术没有处理混合信号系统中激励描述的计时特性, 因而往往只有有限的适用性。
8. **工具支持:** 有什么工具来支持描述技术吗? 有商业可用的工具、原型吗? 或者只是一纸方法呢?
9. **数据存储格式:** 激励描述是按照可以被公开访问的、标准化的或至少归档化的格式被存储起来的吗? 还是按照专有格式来存储的呢? 标准化的或可以被公开访问的格式便于将测试输入规范与测试执行分隔开, 便于不依赖于工具的激励描述在不同测试层次过程中(例如, MiL → SiL → HiL)的复用。

### 16.2.2 当前可用的技术和工具例子

为了说明现有激励描述技术和工具的范围, 以及演示上面提到的一组标准的应用, 在以下部分中将描述和评估对工具和技术的选择。下面描述的激励描述技术中, 有些是与一种专门工具紧密相关, 其他技术要由许多工具来支持, 可以用许多独立的工具来描述。

## 时序图

时序图可以在一个全局时间轴上，利用符号波形来对一些离散信号，以及这些信号之间的关系（例如：延迟、调定、保持）进行抽象图形化表示。可以指定激励信号和预期输出，并将它们相互关联（见图 16.7）。时序图有不同的习惯用语。例如，时序图被用于图形化描述 VHDL 电路模型的测试工作台（Mitchel, 1996），或用于描述模型检查期间的模型属性（Bienmüller 等, 1999）。

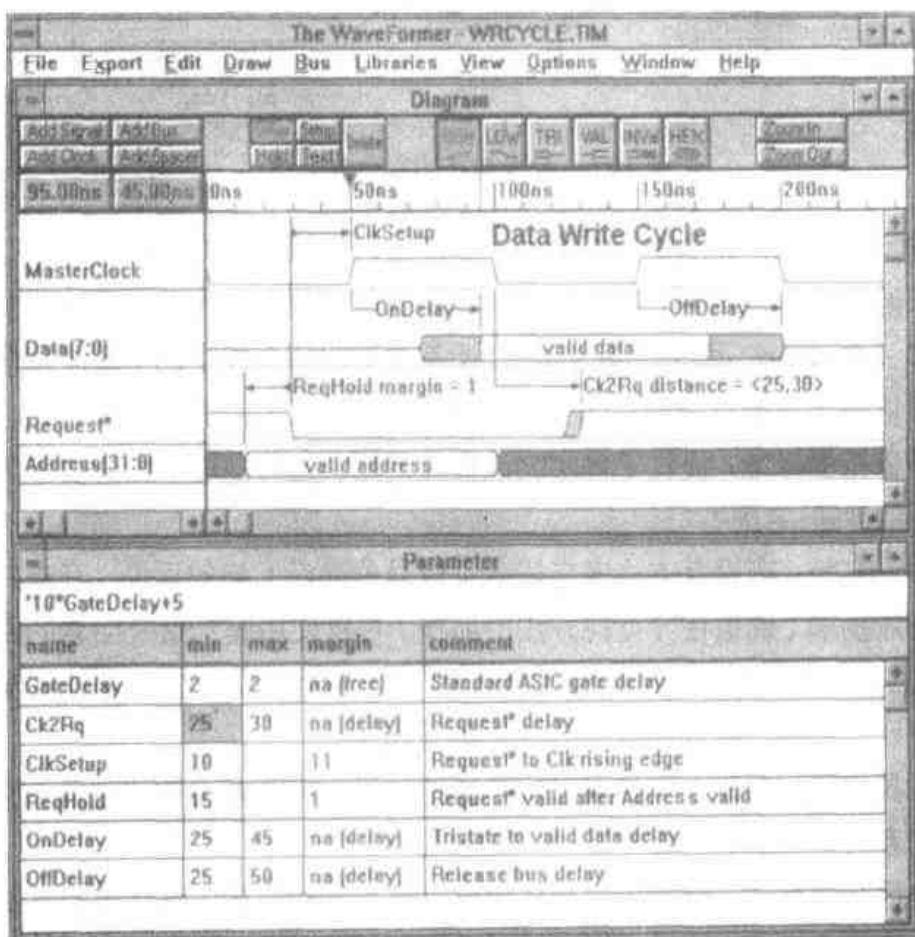


图 16.7 时序图的典型例子

有许多商业工具，例如由 SynaptiCAD 开发的 WaveFormer 与 Testbencher Pro，可以对时序图提供支持。

时序图标记语言（Timing Diagram Markup Language, TDML）提供的交换格式，是不同时序图和波形格式的一个超集。基于 XML 的 TDML 语言能够将信息存储为结构化形式，能够被其他工具方便地处理。对 TDML 激励描述技术的评估如表 16.1 所示。

由 dSPACE (dSPACE, 1999) 开发的 ControlDesk Test Automation 工具提供了一个图形化的激励编辑器，它可以很轻松地创建任何可能的信号序列（见图 16.8）。

表 16.1 TDML 激励描述技术评估

| 评估项        | 评估结论                      |
|------------|---------------------------|
| 1. 支持的信号类别 | - 主要集中于数字信号               |
| 2. 标记类型    | + 文字 (TDML) 与图形方式         |
| 3. 抽象程度    | + 抽象                      |
| 4. 复杂性     | o 定序, 不用编程, 可以描述信号之间的依赖关系 |
| 5. 输出的考虑   | +                         |
| 6. 覆盖范围标准  | -                         |
| 7. 方法支持    | -                         |
| 8. 工具支持    | + 有商业工具                   |
| 9. 数据存储格式  | o 已发布 TDML 格式, 但不是标准化格式   |

### ControlDesk Test Automation 工具的图形化激励编辑器

激励编辑器上面的部分通过矩阵中逐步定义的函数, 方便了激励信号的抽象描述(见图 16.8 上半部分)。每一行(信号槽)描述一个基本波形, 比如正弦、平方、常量、脉冲、倾斜、三角或指数等。生成的信号包含按照一定时间间隔(时间标签)同步的基本波形序列。如果必要, 还可以对这些基本波形进行参数化。除了信号槽以外, 还可以在所谓的控制槽中定义控制结果(循环、条件分支或无条件跳转), 这样就可以描述出复杂而又有周期性的信号波形。

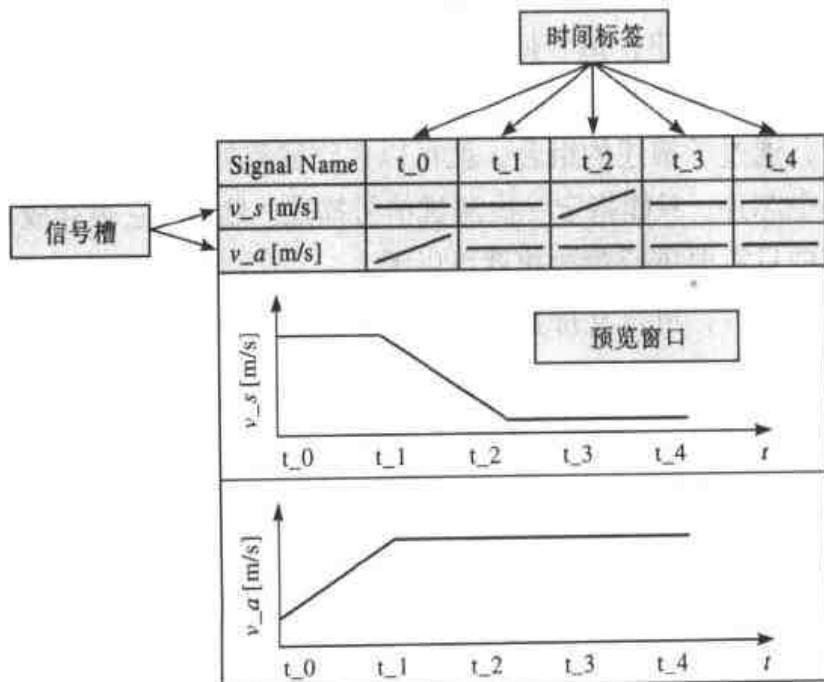


图 16.8 ControlDesk 工具的激励编辑器

激励编辑器的下面部分给出了一个低层次的图形描述（见图 16.8 下面的部分）。矩阵中描述的信号波形在一个预览窗口中给出。

该工具将激励描述转换为脚本语言 Python。

ControlDesk Test Automation 工具的适应性评估见表 16.2：

表 16.2 ControlDesk 激励描述技术评估

| 评估项        | 评估结论                     |
|------------|--------------------------|
| 1. 支持的信号类别 | + 所有的信号类别                |
| 2. 标记类型    | + 图形与文字（Python）方式        |
| 3. 抽象程度    | + 两个不同的抽象层次              |
| 4. 复杂性     | + 定序和编程                  |
| 5. 输出的考虑   | +                        |
| 6. 覆盖范围标准  | -                        |
| 7. 方法支持    | -                        |
| 8. 工具支持    | + 有商业工具                  |
| 9. 数据存储格式  | 0 已发布 Python 格式，但不是标准化格式 |

### TESSI-DAVES

由 FZI Karlsruhe (Sax, 2000) 开发的 TESSI-DAVES 是一个完整的测试工作平台生成环境。它采用通用的信号描述方式，独立于目标系统来描述激励信号。因此，它支持对数字和数字/模拟混合信号的描述。

在 TESSI-DAVES 中，信号描述的基础是将信号严格划分为子描述，从而表示出所期望的测试信号的特定需求。这一面向对象的方法就能够按步长来创建信号描述。最后，通过子描述的组合，就可以创建完整的测试描述规范。因此，对一个模型或一个芯片，只能指定一次测试信号描述。然后，它被存放在一个库中，在以后的调用中只需要进行参数化就可以了。

通过一组编辑器，可以支持直观的激励描述，例如：

- 模式编辑器，为模型输入定义数字测试模式，并为其他信号或活动定义触发命令（见图 16.9 中编号为 3 的子图）；
- 计时编辑器，为在一个时钟周期内的信号变化，定义确切的计时和格式（见图 16.9 中编号为 1 的子图）；
- 模拟波形编辑器，在一个标准区间的不同范围内，分段或按照功能来指定一个模拟信号，以及定义诸如周期、振幅、偏差等参数（见图 16.9 中编号为 4 的子图）；

■ 信号控制编辑器，指定不同信号的顺序，例如通过使用循环和迭代来缩短数字模式（见图 16.9 中编号为 2 的子图）。

TESSI-DAVES 中还带有一个预览窗口，可以在一个窗口中查看所有的数字信号和模拟信号（见图 16.9 中编号为 5 的子图）。

此外还有一个数据管理器可以使用模板和已有的测试描述或对象。这样用户就可以在原有测试的基础上建立和修改出新的测试，而不需要总是从头开始。

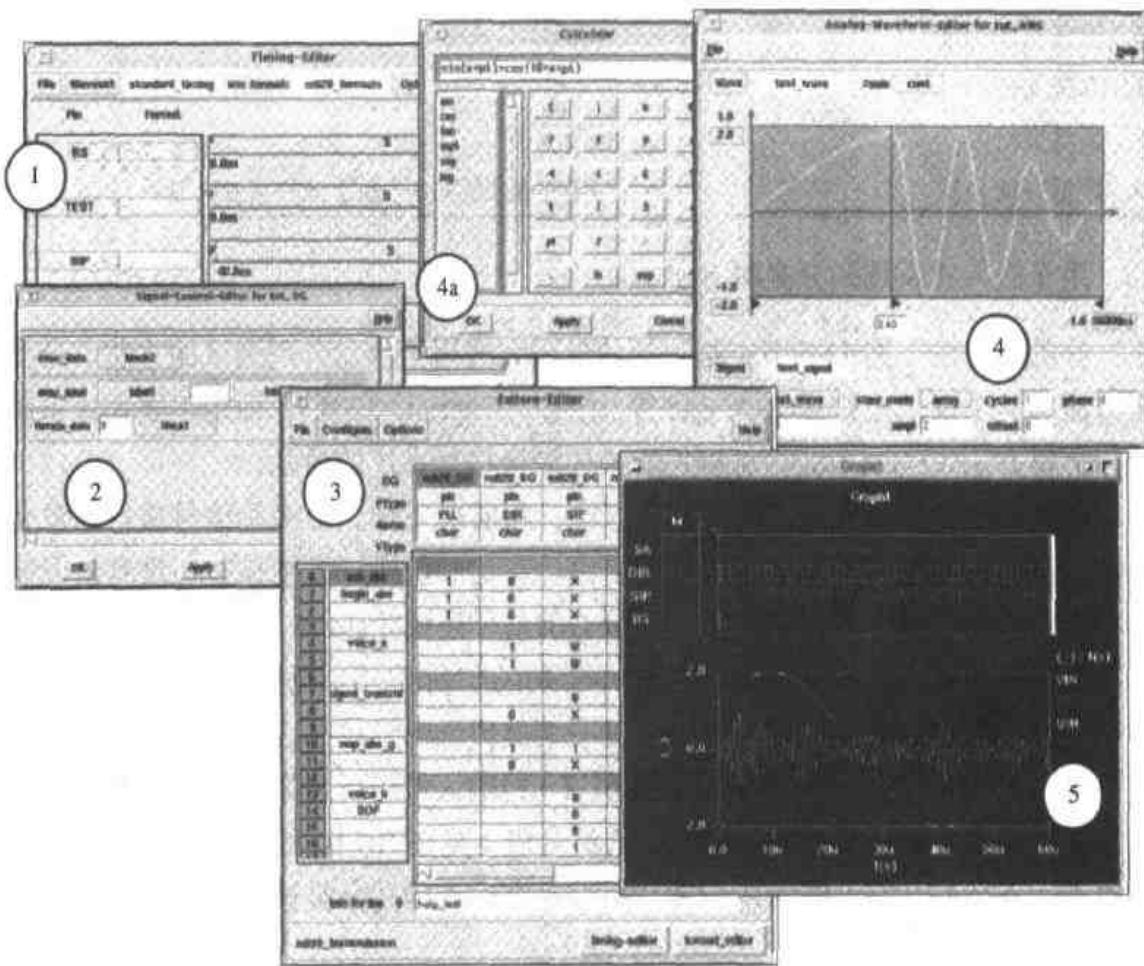


图 16.9 一些 TESSI 编辑器

为了将 TESSI-DAVES 用做各种各样设计过程的集成框架，就要求能够从抽象的、独立于工具的测试设计规范转换为工具专用的信号描述，以便运行完整的系统。很自然地，这就暗示了如果目标系统是一个模拟器，那么生成器将被自动包含到模拟器的模型中；如果目标系统是一个仿真器或物理信号生成器，那么就会生成测试信号文件；如果是为一个测试系统进行转换，那么测试信号文件就会被编程。

TESSI-DAVES 激励描述技术的评估如表 16.3 所示：

表 16.3 TESSI-DAVES 激励描述技术评估

| 评估项        | 评估结论                            |
|------------|---------------------------------|
| 1. 支持的信号类别 | + 所有的信号类别                       |
| 2. 标记类型    | + 图形与底层为文字的表示                   |
| 3. 抽象程度    | + 编辑器支持逐步细化                     |
| 4. 复杂性     | + 按顺序排列和编程                      |
| 5. 输出的考虑   | 0 if-then-else 结构可能依赖于中间的测试执行结果 |
| 6. 覆盖范围标准  | -                               |
| 7. 方法支持    | -                               |
| 8. 工具支持    | + 原型实现                          |
| 9. 数据存储格式  | 0 专有的数据格式：基于文件（ASCII）           |

### 用于嵌入式系统的分类树方法

分类树方法（见 11.5 节）是一个功能强大的用于黑盒分割测试的方法。分类树方法的一个标记化扩展（Conrad 等, 1999; Conrad, 2001），即用于嵌入式系统的分类树方法（CTM/ES），可以用于描述混合信号系统的激励。

激励描述出现在两个不同的抽象层次：在层次 1 中，对激励信号执行抽象化描述。然而，这也留下了一定程度的自由。在这一阶段，激励描述还没有被限制为单个的激励信号，而是指定了一组可能的信号路线（signal course）。自由度在层次 2 中被删除，即从许多可能的输入顺序中选择出一个激励信号。

对抽象逻辑的激励描述来说，通过一个类似于树形状的表示，即分类树（见图 16.10 的上面部分）来对被测系统（SUT）的输入域进行分类和可视化。从 SUT 接口得出的分类树用底层矩阵即所谓的组合表（见图 16.10 的下面部分）来定义描述测试方案的“语言指令”。这里，以抽象方式来描述分类输入信号的所有时间变化。

通过正交节点来表示激励信号，被称为分类树中的分类。每一个信号的允许值被确定，并被划分到等价类中（按照区间或单个值）。

抽象激励描述的构建模块是测试步骤，它们依据时间顺序来组织组合表的多个行。测试步骤的这样一个序列被称为一个测试序列。每个测试步骤定义在一段时间间隔内 SUT 的输入。时间间隔被列在组合表右侧的另一个列中。这些时间间隔的起始点和终点被称为同步点，因为它们是在每个时间步长的起始点和终点来同步激励信号。

通过标记单个激励信号在分类树中定义的类，可以描述该信号在每一时间步长的信号值。这在组合表的中间部分给出。这样，就将在各自测试步长内的激励

信号化为标记类的部分间隔或单个值。测试步长被标记的输入类的组合，确定了在各自支撑点的 SUT 输入。

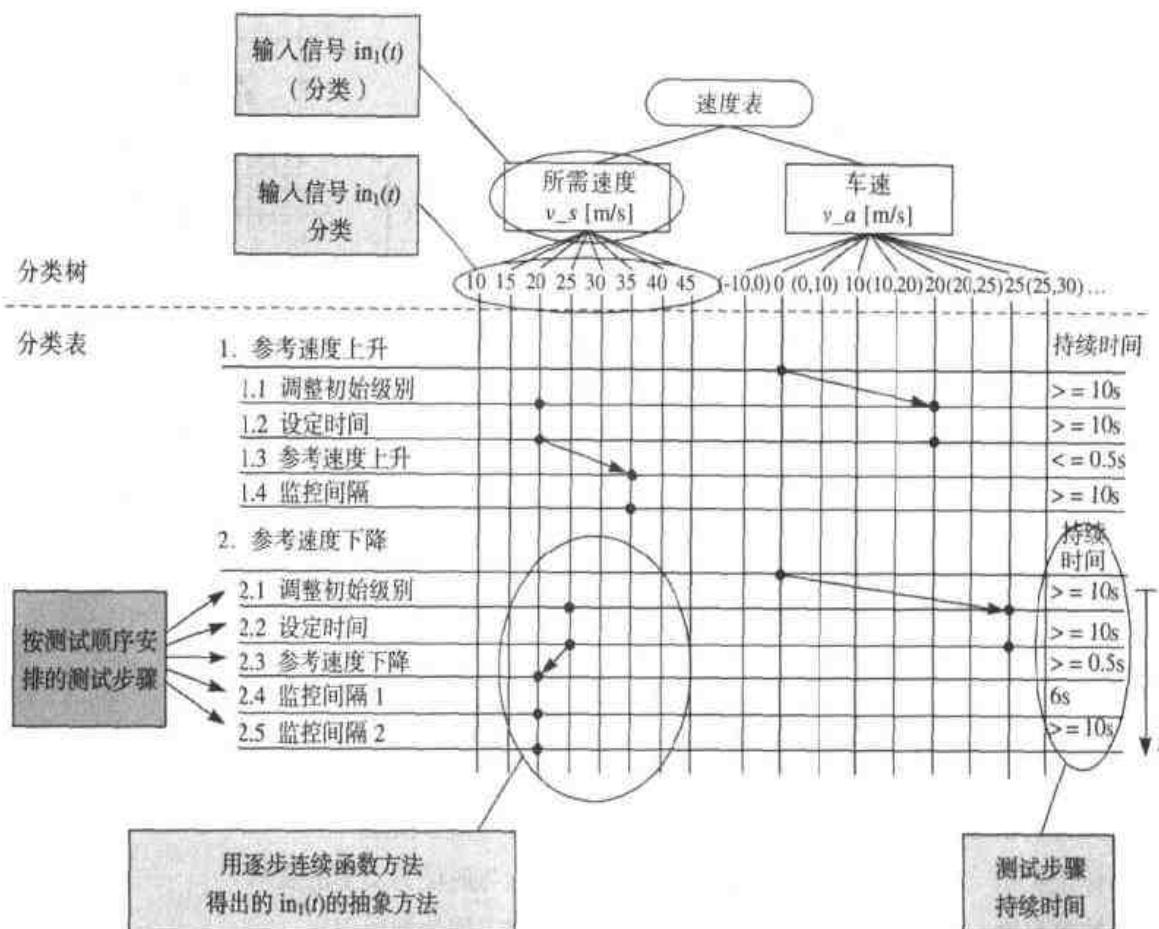


图 16.10 使用 CTM/ES 来抽象描述测试方案

单个激励信号在同步点之间的值是通过基本信号形状来描述的。通过在组合表中用不同类型的箭头连接两个连续标记，来表示不同信号形状（例如有斜度的、阶梯函数、正弦等，见图 16.11 左侧）。通过这种方法，并使用参数化的、按步长定义的函数，就可以用抽象方式来描述模拟和值量化激励信号。

其他的自由度是通过选择具体的测试数据被去除的。接下来，将信号值离散化，这样在最后得到量化或数字信号（见图 16.11）。

在分类树中，输入信号的抽象描述也可以被理解为实际信号线路必须符合的信号管道，或信号通道（见图 16.11 的中间部分）。因为测试对象的激励在时间上是离散发生的，因而必须指定一个采样率来离散化信号线路（见图 16.11 右侧）。离散化的结果是一个具体的由时间-值对组成的激励描述。

值得一提的是，这里描述的符号可以被嵌入到一个方法框架中（Grochtmann 和 Crimn, 1993; Conrad, 2001）。在分类树上可以定义抽象的覆盖标准（Grochtmann 和 Crimn, 1993; Simmes, 1997）。

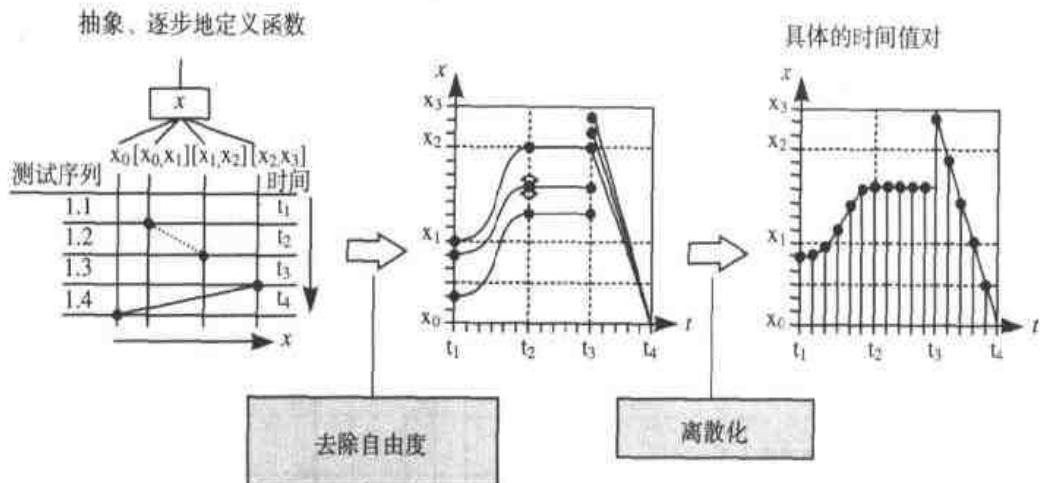


图 16.11 抽象激励描述（测试序列）被逐步细化为具体的信号线路

由 Razorcat 开发的嵌入式系统分类树编辑器（CTE/ES），是一个可用于支持嵌入式系统中使用分类树方法的商业工具。

表 16.4 给出了 CTE/ES 工具的评估：

表 16.4 CTE/ES 工具的评估

| 评估项           | 评估结论               |
|---------------|--------------------|
| 1. 支持的信号类别    | + 所有的信号类别          |
| 2. 标记类型       | + 图形与底层为文字的表示      |
| 3. 抽象程度       | + 两个不同的抽象层次        |
| 4. 复杂性        | 0 按顺序排列            |
| 5. 考虑了激励规范的输出 | 0                  |
| 6. 覆盖范围标准     | + 分类数覆盖范围，数据范围覆盖范围 |
| 7. 方法支持       | +                  |
| 8. 工具支持       | + 有商业工具            |
| 9. 数据存储格式     | 0 已发布基于 ASCII 的格式  |

通常也可以将上述激励描述技术应用于确定预期结果。

### 16.3 测量和分析技术

这一章讨论混合信号系统所用的测量和分析技术。

将捕获的系统输出信号（这里称为“ $f'(t)$ ”）与预期结果（ $f(t)$ ）相比较是一个最为常用的技术（见 16.3.1 节），另一种技术是计算特征值（见 16.3.2 节）。这些分析方法适用于检测特定的故障类别，故障的分类如下：

■ 功能故障：设计没有满足需求。

■ 统计故障：振幅噪声，时域噪声。

■ 系统故障：

- 变换： $f'(t) = f(t+t_0)$ ；
- 被捕获的信号相对预期信号平移了一个特定值；
- 伸长和压缩： $f'(t') = f(at)$ ；
- 在时间轴上采样点增加或遗漏了；
- 放大： $f'(t) = b \times f(t)$ ；
- 被捕获的信号被一个从 0 到无穷的因子放大了；
- 干扰信号的调制： $f'(t) = f(t) + s(t)$ 。例如：一个斜率被增加到预期的信号上，最终的捕获信号为  $f'(t) = f(t) + ct$ 。

多个故障类别的组合是可能的。而且，可能在所有的时间上（全局地）或只有一部分时间槽上（部分地）出现各种故障。

### 16.3.1 预期信号与捕获信号之间的比较（容差）

为了用一个参考信号来验证捕获信号，可以使用许多技术。这里，将讨论下面的技术：

■ 静态技术：

- 相关性；
- 参数计算。

■ 非静态技术：距离测量。

- 不带计时容差进行比较；
- 带计时容差进行比较。

一般来说，当将捕获信号与一个预期信号比较时，要计算某个已定义类型的偏差。由于没有任何一种捕获方法能够保证可以准确地重现系统响应，因而就需要定义容差。通过这种方式，就可以根据容差来区分绝对误差和相对误差。

$$|f(x) - f'(x)| \leq tol_y \quad (\text{绝对误差}) \quad |f(x) - f'(x)| \leq \Delta_y \quad (\text{相对误差})$$

这个简单的容差定义仅仅是相对于值来定义，暗含着一些严重的问题（见图 16.12）。当信号值在短时间内发生急剧变化时，如果没有很好地考虑时间容差，仅仅由一个特定时间点的值来定义容差管道，将导致出现大量的观测误差。换句话说，当在时间上有小的偏移时，如果信号的斜率高，那么就很容易超出容差。

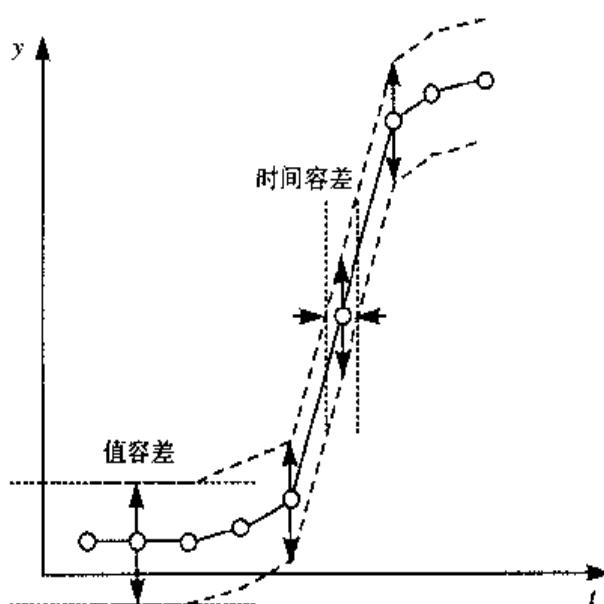


图 16.12 容差

由于这个原因，信号比较就必须考虑时间以及值的  $y$  偏差和  $t$  偏差。可以采用下面的一些方法：

- 建议：容差的定义与信号正交；  
 问题：定义的“正交”只能与轴刻度相关  
     需要将轴标准化  
     容差带可能产生螺旋。
- 建议：容差根据梯度来变化；  
 问题：在快速上升的起始和结束点，梯度很高，但是偏差管道最好不要很高。
- 建议：容差的定义绕着一个采样点旋转；  
 问题：圆周可能太大而无法形成完整的管道。

还可以考虑其他一些建议。总之，对容差定义并没有什么绝佳方法，必须考虑的是时间上的容差和值的容差。

### 16.3.2 分析特征值

比较两个信号时，并不是对所有的采样点进行比较，而是对一些特殊的、能够表达问题的采样点加以比较。这可以极大地减少评估的工作量。这些表达点的门限值可以是只在一个点来比较信号的指示器。图 16.13 中，一些特殊点就很可能与结果评估相关：

- 全局和局部最大、最小值；

- 振幅值的一半 (50%);
- 信号上升和下降的起始点与结束点 (10%和90%)。

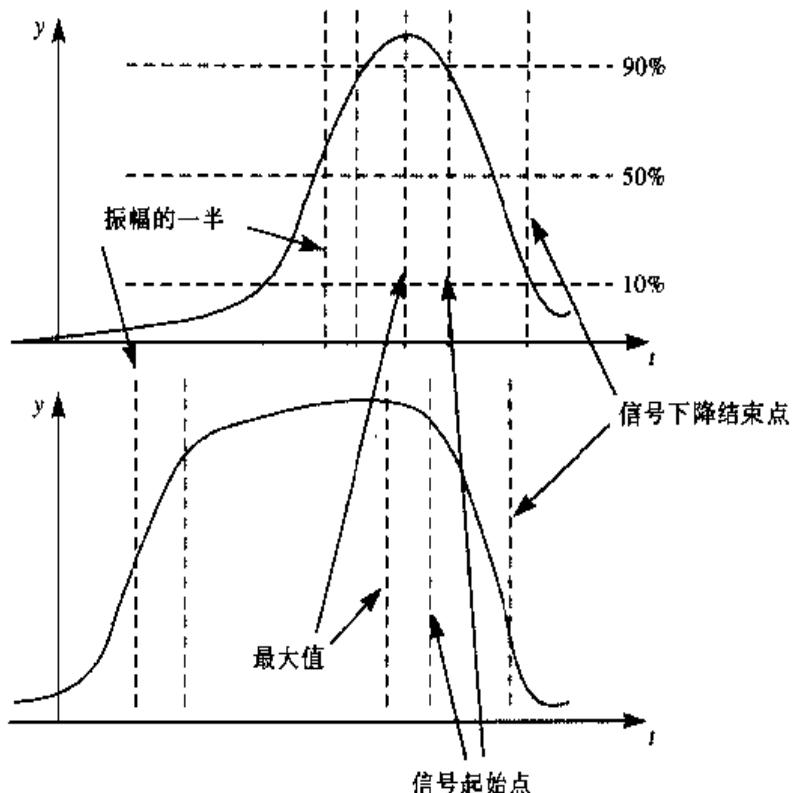


图 16.13 信号比较中的特征点

### 16.3.3 使用统计方法

根据捕获信号与预期结果的比较来进行功能评估，往往是非常昂贵的。必须捕获大量的采样点并加以比较，偏差精度常常要依靠直觉得出。可以不采用这种方法。除了这种方法以外，统计方法也是很有帮助的。当信号不是与时间上的某一点，而是与平均值相关的情况下，统计方法尤其适用。将信号作为一个整体来进行计算是很有用的，例如可用于计算语音或视频信号的质量清晰度。下面简要介绍一些常用的方法。

#### 相关性

在比较记录信号与预期信号时，相关性有助于检测相对于原有信号的类似性和偏移。尤其是交叉相关，它是最能够表达出该结果的计算：

$$z(k) = \sum_{i=0}^{N-1} x(i)h(k+i)$$

交叉相关

参考信号是  $x(i)$ , 捕获信号是  $h(i)$ ,  $z(k)$  是结果。 $k$  是两个信号之间的偏移。这一计算的问题在于, 它是针对整个信号来进行计算。这隐藏了局部误差, 只是将信号作为一个整体来进行验证。

由此可得出结论, 就是要将被评估的信号分到不同的区域中。应当将这些区域限制到所需的短区间内, 以得到有表达意义的结果。

也可以省略最终的总和, 而且, 成对的采样点不使用乘法操作, 只需要执行减法操作。因而, 结果就是由  $k$  和  $i$  展开的一个矩阵。在这个矩阵中, 第三维由差分值给出。从下面公式中可以得出差分的数字表示法 (-1, 0, 1):

$$temp = abs [refsignal(i) - simsignal(i + \tau)]$$

$$z(\tau, i) = \begin{cases} -1 & \text{如果样本不相关} \\ 0 & temp > \text{容差} \\ 1 & temp \leq \text{容差} \end{cases} \quad 1 \leq \tau \leq 2 \times N, 1 \leq i \leq 3 \times N$$

### 信噪比

信噪比由右式给出:

$$S = 20 \text{ dB} \times \log \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

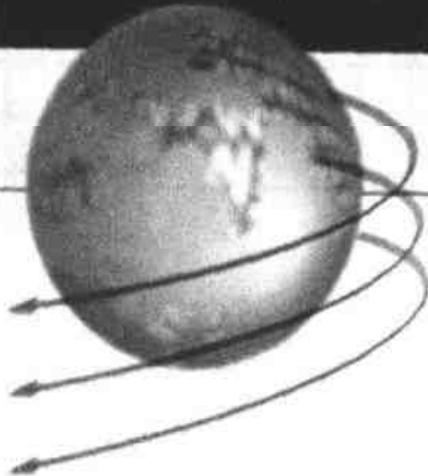
当可以捕获有噪声的信号并且知道所需的信号时, 使用上面的公式来计算信噪比 ( signal-to-noise ratio, SNR )。信噪比的一个应用领域是数/模转换, 它指示经过量化造成的信息损失。

### 总谐波失真

总谐波失真 ( Total Harmonic Distortion, THD )  $k$  是包含  $t$  的表面效应值 ( surface effective worth ) 与总效应值的比例。

$$k = 100\% \times \sqrt{\frac{U_2^2 + U_3^2 + \dots}{U_1^2 + U_2^2 + U_3^2 + \dots}}$$

THD 是表示信号非线性失真的一个特征数。为了捕获这些误差, 要定义一个正弦信号来激励系统并对结果进行评估。额外的谐波数量也是失真的一个指示。THD 主要用于高频和音频等应用。



## 第五部分 组织

本书的这一部分描述测试中的各种组织。这里讨论的不是技术上的事情，而是关于人的事情。即执行所有测试活动的人员，人员之间的相互关系及与其他人员之间的交互。“组织”部分涉及以下主题：

- 测试角色。为了在一个测试项目中执行各种测试活动，测试组织内部必须具备特定的知识和专业技能。第 17 章定义了与测试活动相关的各种角色及其所需的专业技能。
- 人力资源管理。第 18 章讲的是测试项目中的人员。如何获得具备合适资格的合适人员来充当所需的测试角色。这涉及到获得临时人员以及对专业测试人员的培训，并为他们提供值得去奉献的职业生涯。
- 组织结构。测试要达到一定的目标来支撑业务。在大型业务组织的分层结构内，测试组织所处的地位必须与测试的期望值相匹配。第 19 章描述了实现测试组织的一些可能方法，包括垂直型组织和项目设置中的测试组织。
- 测试控制。第 20 章讲述测试过程的控制和产品。测试团队必须能够提供与测试状态、进度趋势和质量趋势有关的信息，这就要求人员都按照一定的规程来办事。



这些章节描述了各种测试角色的组织、管理和组织结构的组成模块。通常并不需要按照这里描述的来完整地建立所有这些模块。而是根据诸如测试层次、测试对象大小、组织文化等因素来选择测试角色、人员和管理的最优组合。

在高层次测试时（系统测试或验收测试），这里描述的大部分组成模块在许多项目中都需要。对低层次测试，通常这些构件的某个部分的一个分支就足够了。组成模块种类很多，测试经理可以仔细选择用于高层次测试和低层次测试时的最优测试组织实现。通常总是面临着选择得太多或太少的危险，选择太多时将导致没有生产效能的官僚机构，选择太少时将导致缺少控制，造成混乱。主要的目标应当总是：在时间和预算限制之内得到可能的最佳测试。

# 第 17 章 测试角色

为了在适当的时刻有适当的测试人员，就有必要对测试中的职责和任务，以及所需的知识和技能有很好的了解。这一章讲述测试中的各种职责。

首先描述一般技能，然后描述不同的测试角色及他们的特定技能。

---

## 17.1 一般技能

测试团队的每个成员至少都应当具备结构化测试、信息技术和一些常识等基础知识。

### 测试知识

- 测试生命周期的一般知识；
- 评审和使用测试设计技术的能力；
- 测试工具知识。

### 信息技术技能

- IT 领域的一般知识和经验；
- 系统开发方法的广泛知识；
- 分析和解释功能需求的能力；
- 一般计算机技能；
- 对硬件、软件和数据通信设施有一般的了解；
- 架构和系统开发工具方面的知识。

### 常识

- 领域专业知识和垂直型组织知识；
- 有在一个项目团队工作的项目组织知识和经验；
- 创造性和准确性。

## 17.2 特定的测试角色

要描述的测试角色有：

- 测试工程师；
- 团队领导；
- 测试经理；
- 测试策略经理；
- 方法支持；
- 技术支持；
- 领域专家；
- 协调人；
- 测试配置经理；
- 应用集成师；
- 测试自动化架构师；
- 测试自动化工程师；
- 安全经理；
- 安全工程师。

### 17.2.1 测试工程师

测试工程师的任务是最有创造性的，但也是最为费力的测试角色。从准备阶段到完成阶段，都需要有测试工程师角色。

#### 典型任务

- 评审测试基础（规范）；
- 制定逻辑和物理测试用例及开始情形的规范；
- 执行测试用例（动态测试）；
- 静态测试；
- 记载缺陷；
- 存档测试件。

### 17.2.2 团队领导

一个团队领导管理一个小型的测试人员团队（2~4个测试人员）。从准备阶段到完成阶段，都要有团队领导角色。

### 典型任务

- 制定详细计划；
- 将工作量分配给团队成员并监控进度；
- 安全措施；
- 保持进度的记录并分配（时间）预算；
- 报告测试进度和测试质量；
- 提出瓶颈的解决方案并实现这些方案；
- 指导和支持测试人员；
- 对团队成员进行评估。

### 其他技能

- 管理项目的能力；
- 激励成员士气的能力；
- 良好的沟通技能；
- 清晰的报告风格。

## 17.2.3 测试经理

测试经理的职责是在时间、预算和相应的质量要求之内，计划、管理和执行测试过程。测试经理保持测试过程进度和测试对象质量的记录。从测试过程的开始直到完成，测试经理角色都存在。

### 典型任务

- 创建测试计划，使之得到批准并进行维护；
- 依据时间进度和预算来执行测试计划；
- 报告测试过程的进度和测试对象的质量。

### 其他技能

- 有领导团队、协调人和支持任务的经验；
- 大量的管理项目经验；
- 使用计划和进度管理工具的能力；
- 优秀的沟通技能；
- 极好的报告风格；
- 激励成员士气的能力；
- 有挑剔而又开放的态度；
- 机智、有抵抗恐慌和承受批评的能力；

- 精通谈判技术，有解决冲突的能力；
- 有注重实效和解决问题的态度。

#### 17.2.4 测试策略经理

当测试完全或部分嵌入到垂直型组织结构中时，测试策略经理开始发挥作用。“部分嵌入到垂直型组织结构中”是指在测试策略经理角色实现为一个专职功能，和测试规章制度角色包含在除测试角色以外的所有角色中的所有事情。

在整个测试生命周期过程中，测试规章制度的角色可以一直发挥作用。

##### 典型任务

- 建立和维护测试规章制度；
- 传达测试规章制度；
- 监督测试规章制度的遵守；
- 监督测试规章制度的适用性；
- 报告结果；
- 开发新的测试技术和规程。

##### 其他技能

- 以创造性的、准时的和严格的方法学来工作；
- 激励成员士气的能力；
- 良好的沟通技能；
- 清晰的报告风格；
- 有挑剔而又开放的态度；
- 能够让组织成员看清楚质量管理和测试的重要性，必要时能够保证提高各自的测试活动；
- 有谈判和解决冲突的能力，熟悉幻灯演示、报告和会议技术；
- 有注重实效和解决问题的态度。

#### 17.2.5 方法支持

方法支持角色是从最广泛的意义上，提供与方法学有关的帮助。它是指所有的测试技术和测试团队中的所有成员，这中间不包括测试管理人员开发的策略和测试人员开发的测试用例规范。

在整个测试生命周期过程中，方法支持角色可以一直发挥作用。

### 典型任务

- 建立测试技术；
- 开发新的测试技术；
- 制定测试规章制度；
- 开发相关的培训课程；
- 教学和辅导；
- 对所有可能的测试技术的执行给出建议和帮助；
- 向管理人员提出建议。

### 其他技能

- 以创造性的、准时的和严格的方法学来工作；
- 激励成员士气的能力；
- 良好的沟通技能；
- 清晰的报告风格；
- 有挑剔而又开放的态度；
- 能够让组织成员看清楚质量管理和测试的重要性；必要时能够保证改进各自的测试活动；
- 额外的解决冲突和谈判的能力，以及熟悉幻灯演示、报告和会议技术；
- 有注重实效和解决问题的态度。

## 17.2.6 技术支持

技术支持角色是从最广泛的意义上，提供与技术有关的帮助。该角色负责使高质量的测试基础设施可以连续用于测试过程。这也保证了测试环境、测试工具和办公空间的监督管理与可操作性。

在整个测试生命周期过程中，技术支持角色可以一直发挥作用。

### 典型任务

- 建立测试基础设施；
- 监督测试基础设施；
- 物理配置管理；
- 技术问题解答；
- 确保测试的重现；
- 给出帮助和建议。

### 其他技能

- 广泛的测试工具知识；
- 监督工具的高级知识；
- 熟悉开发和构造测试环境的方法；
- 模拟环境的知识；
- 注重实效和解决问题的态度。

## 17.2.7 领域专家

领域专家是从最广泛的意义上，提供与测试过程有关的功能帮助。该角色提供在重要功能方面执行活动的支持。这些活动可能包含由测试管理人员执行的测试策略开发或风险报告，以及由测试人员执行的可测性审查或测试用例设计。

在整个测试生命周期过程中，领域专家角色可以一直发挥作用。

### 典型任务

在必要时，提供有关测试对象功能的帮助和建议。

### 其他技能

- 激励成员士气的能力；
- 良好的沟通技能；
- 清晰的报告风格；
- 被测系统和垂直型组织的专门知识；
- 将功能解决方法和实际应用相结合的能力。

## 17.2.8 协调人

协调人是测试团队和其他方之间关于缺陷的联系人。所有的缺陷及修复都要通过协调人，协调人专门关注于缺陷和解决方法。在产品发布给其他方之前，协调人要分析所有的缺陷，他可以要求改正，但是也可以拒绝他的修改要求。协调人角色是风险报告中的一个重要部分。

从开始可测性评审时的准备阶段到完成阶段，协调人的角色都是必须的。

### 典型任务

- 为缺陷管理规程和工具制定需求；
- 汇总缺陷和各自的背景信息；
- 检查缺陷的整体性和修正（方案），分析、分类和评估缺陷的紧急程度；
- 使用可用的规程，向相应的实体报告缺陷；

- (有必要时) 参与与分析和决策相关的沟通;
- 为测试人员收集来自其他方的与缺陷和问题有关的相关信息;
- 检查解决方法的完整性和处理解决方法的受控实现, 报告解决方法用于再测试的可用性;
- 保存与实际测试对象有关的缺陷状态和解决方法的记录;
- 帮助测试管理人员(如果需要) 创建风险报告。

#### 其他技能

- 优秀的沟通技能;
- 优秀的报告风格;
- 爱挑剔的态度;
- 有解决冲突的能力并熟悉谈判技术。

### 17.2.9 测试配置经理

测试配置经理是一个管理和后勤角色。该角色负责将测试过程内需要控制的所有对象登记、存储并使之可用。在一些情况下, 控制人员自己执行这些职责。而在其他情况下, 他们建立和管理一个控制体。该角色与日常配置经理的角色相同, 只是专心于测试。

操作控制职责的分配主要依赖于测试过程的环境和类型(范围、项目或垂直型组织、新开发或维护)。如何将深入的控制任务在组织中推广当然是很重要的。

#### 典型任务

- 进度控制;
- 测试文档控制;
- 保存缺陷记录并汇总统计;
- 让测试基础和测试对象的逻辑控制对测试过程可见;
- 测试文档的逻辑控制;
- 前提条件和后勤;
- 代理监管职责的控制。

#### 其他技能

- 广泛的控制工具知识;
- 垂直型管理知识;
- 良好的沟通技能;
- 优秀的管理技能。

### 17.2.10 应用集成师

应用集成师(AI人员)负责将各个独立的部分(程序、对象、模块、构件……)集成为一个能够正确工作的系统。AI人员依据集成测试计划，向项目领导报告测试对象的质量和集成过程的进度。

为了避免利益冲突，AI人员应当不担任项目开发领导的角色。这样，有可能在AI人员和项目开发领导之间存在利益冲突，AI人员负责质量，主要是在提供的功能、时间和预算基础上来评估项目开发领导的工作。

AI人员最好具有开发背景。与高级测试人员相比，AI人员应该更清楚系统的内部操作，能够更好地装备来查找出引起缺陷的根源。与其他开发人员相比，AI人员在测试和系统方面具备更多的知识。

与开发人员进行充分接触，是AI人员很好工作的基础。除此之外，在后期的测试层次中，AI人员是主要的联系人。

#### 典型任务

- 编写集成测试计划，使之得到批准并进行维护。
- 编写输入标准。独立的程序、对象、模块、构件……将遵循该标准，以便获得许可进入集成过程。
- 编写输出标准。系统的集成部件将遵循该标准，以便得到批准而进入下一阶段。
- 帮助和支持单元测试执行中的程序员。
- 将独立的程序、对象、模块、构件……集成到用户功能或子系统中。
- 依据计划和预算来管理集成测试计划。
- (加强)配置和发布管理。
- (加强)内部缺陷管理。
- 在后面的测试层次期间，其角色是作为与客户组织之间的协调人。
- 报告集成过程的进展和测试对象的质量。
- 对产品发布进行授权。
- 评估集成过程。

#### 其他技能

- 有团队领导、协调人和支持任务方面的经验；
- 熟悉系统设计方法；
- 有广泛的架构和系统开发工具方面的知识；
- 清晰的报告风格；

- 良好的沟通技能；
- 激励成员士气的能力；
- 有挑剔态度；
- 机智、抵抗恐慌和承受批评的能力。

### 17.2.11 测试自动化架构师

测试集是由测试工具、测试用例、测试脚本和观测的结果组成的。测试自动化架构师负责作为一个整体的自动化测试集的质量，为每个测试集制定需要达到的自动化目标。架构师将目标、选择的方法和测试对象特性转化为测试集架构。他们将专门把注意力放在设计和构造测试套件上。

#### 典型任务

- 管理测试自动化分配任务的快速审查。
- 为构造测试集编写行动计划，使该计划得到批准并维护该计划。
- 设计测试集。
- 负责设计和构造测试集。
- 报告构造测试集的进度和质量。
- 加强或管理测试工具的选择。
- 对用户进行自动化测试集应用的培训。

#### 其他技能

- 作为团队领导的经验；
- 有丰富的（使用）测试工具方面的知识；
- 有测试过程自动化经验；
- 熟悉结构化编程；
- 良好的概括和分析技能；
- 快速学习新技术的能力；
- 清晰的报告风格；
- 良好的沟通技能；
- 激励成员士气的能力；
- 有挑剔态度。

### 17.2.12 测试自动化工程师

测试自动化工程师负责自动化测试集的技术。该工程师将测试集的设计转化

为测试工具内要构造的模块，他或她通过新的或已有的测试工具来构造测试集，在测试自动化架构师给出的框架内，测试自动化工程师进行详细设计、构造和实现测试集。

#### 典型任务

- 构造、测试、指导和维护测试集；
- 对测试集的设计提供支持。

#### 其他任务

- 熟悉测试过程自动化；
- 有结构化编程经验（最好是工具所需的编程语言）；
- 良好的分析技能；
- 有创造性并能够按时完成任务；
- 快速学习新测试工具的能力；
- 有挑剔态度。

### 17.2.13 安全经理

安全经理的职责是建立安全项目和招聘合适的项目人员。安全经理有了解安全和启动安全程序的职责。他必须使项目成员相信安全的重要性，并为该项目分配合适的项目成员。

#### 任务

- 准备安全计划；
- 建立项目组织；
- 项目管理；
- 咨询。
  - 认证机构
  - 政府（如果有必要）
  - 委托人
  - 承包人

#### 知识和技能

至少拥有与安全程序应用领域相关的工程或理科学位，最好有安全工程的正式培训。

### 17.2.14 安全工程师

实际中的大多数安全活动是安全工程师工作的一部分。除此之外，安全工程师还组织和领导日常的项目安全委员会会议。

#### 任务

- 确定风险等级；
- 确定风险并对风险划分等级；
- 指导分析；
- 维护安全日志；
- 对与安全有关的变更请求的影响提出意见；
- 评审测试计划；
- 评审缺陷。

#### 知识和技能

至少拥有与安全程序应用领域相关的工程或理科学位，最好有安全工程的正式培训。

# 第 18 章 人力资源管理

测试要求有大量的专业人员。第 17 章列出了各种角色和任务，以及所需的知识和技能。这一章讨论人力资源以及与人力资源有重要关联的培训。测试管理人员负责将合适的人员分配到合适的位置，当然最好事先经过咨询人力资源管理部门（HRM）。需要周密地安排招聘和测试人员的培训，合适的有经验的测试人员十分缺乏。提供一个有前景的职位给专业测试人员可能是一个解决方法。

## 18.1 人员

在一个测试组织内部，合理地组合专业人员是很重要的。测试需要有以下领域的专业人员：

- 开发系统；
- 基础设施（测试环境，开发平台，测试工具）；
- 测试；
- 测试管理。

低层次的测试人员通常是开发人员，他们主要关心已开发的系统和开发基础设施。高层次的测试涉及到开发人员、用户、经理和/或测试人员，所需的专业人员更多地转向领域专家和测试知识方面的人员。

除了与人员相关的一般问题以外，所需测试人员的某些特质，在测试人员的选择、引入和培训时必须要加以考虑：

- 测试人员可能有不正确的测试思想。通常测试活动与正常活动分开进行，设计人员或用户只是临时性地承担测试任务。实际上，测试团队中专业测试人员的数量是很不够的。只有很少的组织将测试功能的价值与开发功能的价值看得同等重要。
- 测试需要有测试态度和专门的社交技能。当同事们将最佳的可能产品提交给测试人员时，测试人员接受同事产品的方式，有可能对被测产品质量的提高有所帮助。一个表达测试人员“同意接受”的恰当方式，也许就可以提高产品的质量。因此测试也需要有高超的社交技能。

- 测试人员必须能够厚着脸皮，因为他们要承受来自个方面的批评，面临不少困难，比如：
  - 他们遗漏了缺陷（因为不可能进行 100% 的测试）。
  - 测试人员爱挑剔！
  - 测试人员在关键阶段停留得时间太长，测试花费得时间太长。
  - 测试过程太昂贵了。难道就不能更便宜，（尤其）是用更少的人员吗？
  - 测试人员太强调现在就要产品。我们不能为了测试而构造！
  - 当系统出现缺陷时，往往是测试人员受到谴责。当系统功能良好时，得到褒奖的是系统开发人员。
  - 并没有完成所有的测试，测试建议常常被忽略。测试人员必须要处理这一情况。
  - 测试人员通常缺乏经验，因为（结构化）测试对他们来说是一个新服务。
  - 测试人员必须有即席创作和创新才能。前面过程的滞后常常阻碍测试计划。
- 制定测试人员招聘和淘汰计划的困难。测试过程的开始、进展和完成难以预测。测试过程的特点是工作时间没有被充分利用、有额外工作和无法预料的工作量高峰。人力资源提供商想要能够提早知道哪些人员可用以及在什么时间可用，这将会导致冲突、失望、产品质量下降和计划落空。
- 测试培训被认为是多余的。测试通常被看做是任何人都可以干的工作。

测试经理应该精通职业人事管理。经验已经表明人事官员很乐意提供帮助，当然是涉及本书中描述的角色描述和培训要求。要确保对需求有深入了解，这包括工作需求和（人员引入）的计划需求。如果在测试过程中需要外部专业人员，那么就必须深入了解可用的预算。

## 18.2 培训

（测试活动）对测试人员有大量的要求。他们被期望拥有各方面的能力（一般的 IT 知识和社交技能），和对知识的透彻了解（测试方法和技术）。培训可以在那里起到作用。应当在需求和被选择的测试人员拥有的知识和技能之间建立一座桥梁。对测试人员的培训程序要包含大量的主题：

### 测试特定主题

- 测试方法和技术；
  - （结构化）测试的重要性

- 模型的测试生命周期和计划
- 测试设计技术
- 评审和检查技术
- 测试环境和测试工具；
  - 测试过程的自动化
  - 测试工具
- 测试管理和控制；
  - 人员和培训事项
  - 测试计划、测试预算
  - 风险评估和策略开发技术
  - 控制规程
  - 改进测试过程
- 报告和文档技术；
- 质量管理；
  - 质量术语（ISO9000）
  - 标准
  - 审计
- 测试态度，即社交技能。

#### 一般主题

- IT 的一般知识；
  - 策略，计划
  - 设计策略
- 系统开发的方法和工具；
  - 生命周期模型
  - 建模技术
  - CASE，面向对象（OO），第四代编程语言（4GL），工作平台和基于构件的技术
- 计划和进度监控技术；
  - 计划工具
- 良好的沟通和社交技能，以及个性；
  - 报告和幻灯演示
  - 社交技能
- 技术基础设施；

- 硬件：大型机、PC 机等
  - 软件：操作系统、中间件
  - 通信：服务器、局域网及广域网
  - 图形用户界面、客户/服务器及因特网等
- 领域专家和组织知识。
- 功能
  - 组织结构
  - 业务目标

结构化地开展测试培训课程并采用模块化方式来进行是很合理的。根据角色和雇员的专业，培训时针对每个（子）主题可多可少。培训结构在图 18.1 中给出。

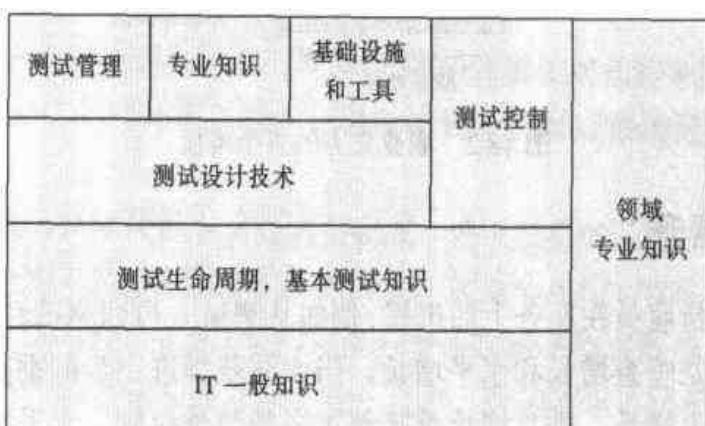


图 18.1 测试培训课程结构

培训课程通常必须要依赖于第三方。因此，这里给出的模块化和培训策略等相关要求，实际上就是对培训机构的要求。

英国计算机协会下属的信息系统考试部（ISEB）对测试人员进行独立认证。ISEB 向测试人员提供三种难度逐次增加的证书，一旦测试人员通过考试就可以获得证书。

## 18.3 职业前景

### 18.3.1 介绍

为了在组织内部使测试专业化，就必须提供相应的职业前景。这一节提供了用于说明测试职业生涯的方法。基础部件包括培训、增加工作经验和一个监管程序。这些可以用所谓的职业立方来给出每个功能和每个等级（见图 18.2）。这个立方是人力资源经理进行职业指导的工具，目的是能够依据组织内部的需求来匹配

可用的知识和技能，以及专业测试人员的抱负。职业管道的三维定义如下：

- 高：职务晋升；
- 宽：职务差异；
- 程：知识和技能。

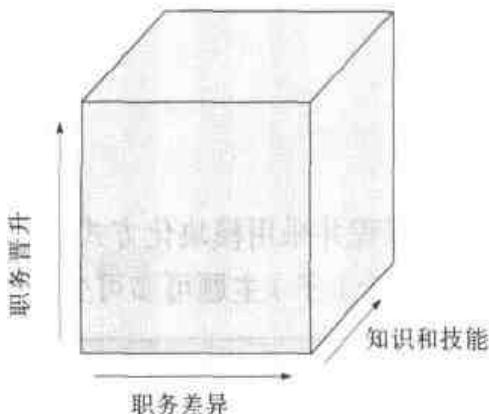


图 18.2 职业立方的三个维度

### 18.3.2 职务晋升

职务晋升是指雇员在职务上的进展，例如从测试人员到测试经理（见图 18.3）。这里需要区分的是垂直增长和水平增长。当无法获得进一步的垂直增长时，还有可能在水平方向上增长。垂直增长意味着更高的职务级别，水平扩展是指在同一职务级别内更高的执行级别。在这个结构中，雇员的提升条件可以通过到达一个更高职务级别或更高执行级别来获得。



图 18.3 职务晋升

### 18.3.3 职务差异

要区分开三个职务差异（见图 18.4），它们是：

- 团队和项目领导。有才能以及对领导测试项目感兴趣的雇员可以选择这个方向。

- **方法建议。**这个方向是针对那些想要提出测试建议的雇员，例如当制定一个测试策略时对测试设计技术或组织控制的选择。
- **技术建议。**对测试技术感兴趣的个人可以选择技术（测试）建议。这包括组织和利用测试工具，或者是搭建和管理测试基础设施。

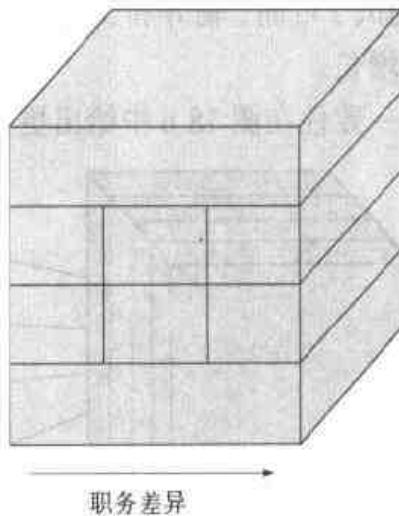


图 18.4 职务差异

图 18.5 给出了职务级别的例子，每个方向上表示的都是职务名称。在级别 A，对于列来说没有清晰地划分。在这个级别，人们可以在各个方面获得测试执行的广泛知识。在职务级别 B 和 C，就有了差异。在职务级别 D，又没有了差异。在这一级别的雇员是要能够成功地实现和/或管理每一个职务差异。

|   |        |        |        |
|---|--------|--------|--------|
| D | 一般测试管理 |        |        |
| C | 测试项目领导 | 方法测试建议 | 技术测试建议 |
| B | 测试团队领导 | 方法测试特化 | 技术测试特化 |
| A | 测试执行   |        |        |

图 18.5 职务级别例子

#### 18.3.4 知识和技能

增加知识和技能是由职业立方的第三维来表示的。需要区分开以下构件：

- (测试)培训；
- 社交技能；
- 经验；
- 指导和支持。

指导和支持是成功职业道路的基本条件，尤其是在（开始）较低职务级别的情况下。每个雇员都能够得到各种方式的支持。这是由测试经理、人事官员和/或有经验的雇员来执行的。刚开始起步的测试人员会得到一个辅导员的专门注意，由他来监管测试人员的各个方面和今后的发展。

当职务级别提升时，（测试）培训、辅导和支持相关的实践经验和社交技能培训的重要性，也要有相应的增长。

所需知识和技能之间的一致性在图 18.6 中给出更详细的描述。

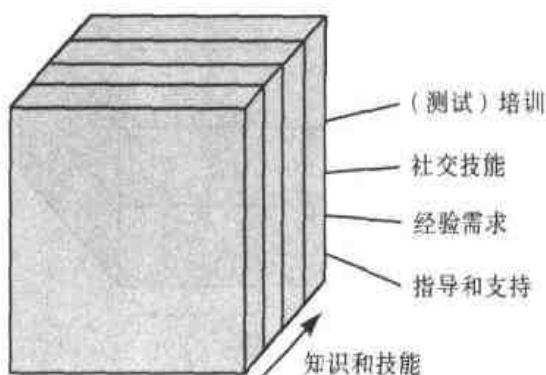


图 18.6 知识和技能

最终，在以上讨论的基础之上，就可以得出完整的“职业立方”（见图 18.7）。它给出了每个职务（差异）所需的知识和技能。

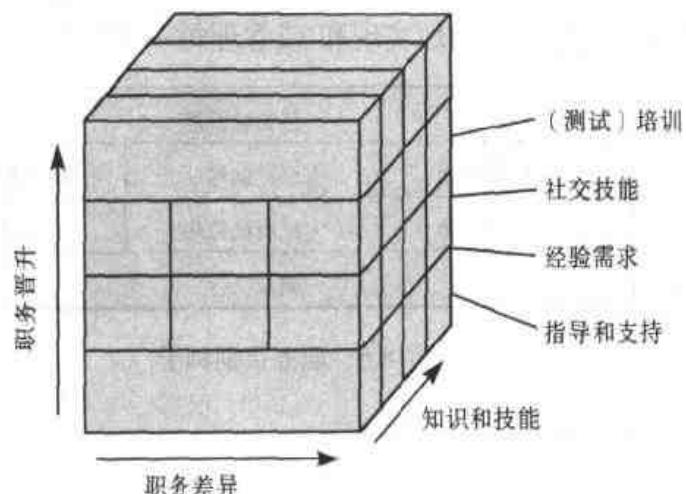


图 18.7 完整的“职业立方”

# 第 19 章 组织结构

在一个组织中，职责、任务、层次关系和通信结构应当是清晰的。通过标识所需的测试角色，以及将这些角色赋予所选择的合适人员，就应当能够实现。通过和测试组织以外的产品权益人一起，在测试组织内部建立咨询和报告结构，就应当能够建立清晰的通信结构。

## 19.1 测试组织

低层次的测试大多数时间内都是由开发人员来执行的。因此，组织独立于开发的低层次测试并没有什么用处。

独立的测试组织大多数时间用来执行高层次测试，可以按照项目或线性活动来组织。如果产品要定期发布，那么将测试建立为线性组织就非常有用。这样有利于获得与测试有关的专业技能和知识，能够得到更多的经验，而且可以为测试团队成员提供良好的职业前景。它也提供了机会来建立必要的基础设施和维护该基础设施的专业人员。这一点对于系统特定而又十分昂贵的基础设施来说是很重要的。如果产品是惟一产品（即一次性系统），那么项目形式就是最好的组织测试方法，应该为该项目建立专门的组织结构。

### 19.1.1 项目组织

大多数系统都是按项目来开发的。通常都是为了开发一个特定的产品而启动一个项目，即便开发和测试都是由线性部门来执行。项目组织具有以下优点：

- 目标清晰（实现产品）；
- 组织专门致力于设定的目标；
- 人员是通过为实现产品所需的特定技能来进行招聘的；
- 因为焦点只有一个目标，因而管理和控制就更为容易。

缺点是：

- 大量的事情需要从零开始起步，例如建立基础设施、引入标准和规程等；
- 如果人们在项目结束时离开（当产品正在进行验收测试时），就很难在必要的时候将他们召回。在这个阶段，每个人都还没有足够的工作，但是让人们

离开会使知识遗漏掉。这可能会对解决缺陷造成严重的后果。

通过在一个线性组织中启动项目，就可以避免上述许多缺点（见 19.1.2 节），虽然并不总是能够避免。

图 19.1 用一个独立的高层次测试团队来形象化地表现项目组织，开发团队执行的是低层次测试。

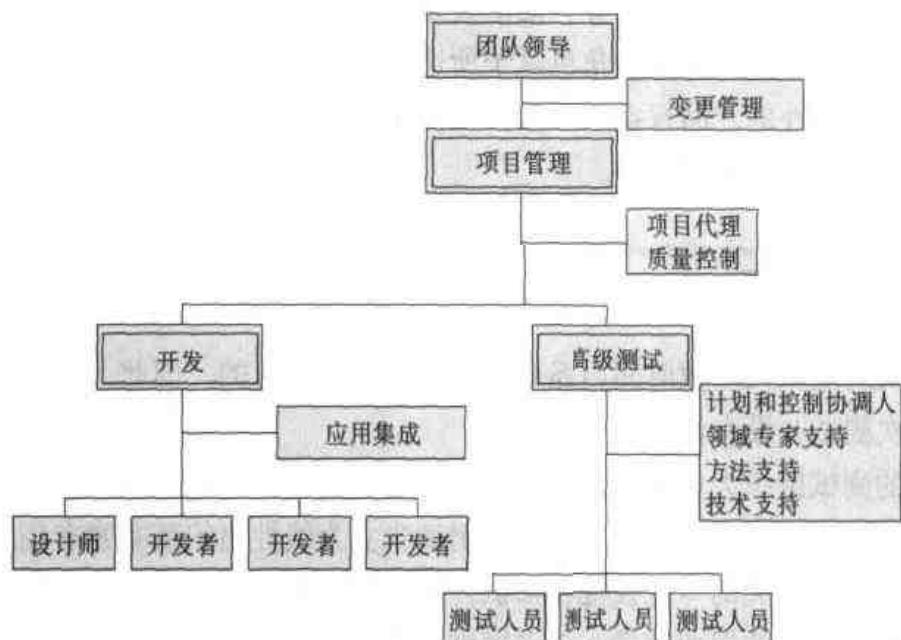


图 19.1 项目组织例子

开发部门执行在应用集成师职责之内的单元测试和集成测试，应用集成师负责向开发部门领导报告产品的质量状况。为集成测试制定的输出标准，必须与从事高层次测试的测试团队所制定的输入标准相同。考虑到高层次测试，他们必须正式验收产品。所有（低层次和高层次）测试的状态报告都要受到质量控制。

开发领导面临着在时间和预算范围之内，需要交付一定质量产品的压力。实际上，很难一起实现所有这三个目标。开发领导必须定期地向项目管理人员报告这三个要素的情况。如果存在严重的问题，那么就要报告给指导委员会。指导委员会由委托人、项目领导、开发领导、测试领导和其他产品权益人组成。他们基于应用集成人员所写的报告，最终确定产品是否“足够好”，可以开始进行高层次测试。重要的是他们不能太快批准，因为在低层次测试期间，缺陷解决起来更容易，也更廉价。除此之外，启动高层次测试也常常就是缩减开发人员的开始。在高层次测试期间，虽然仍然需要一些开发人员来解决缺陷，但不再需要所有的开发人员。

当高层次测试结束时，指导委员会基于测试领导的报告，决定是否投入生产，并随后解散项目组织。除此之外，指导委员会成员要定期会面，以监督项目进展，必要时要采取相应的措施来保证项目进度。

### 19.1.2 线性组织

许多公司被组织为生产新产品或定期变更产品。这样，产品开发就成为组织的一个部门。在产品开发期间，有一个开发部和一个独立的测试部。测试部从事高层次的测试，而开发部也进行了低层次测试活动。虽然这听起来像是一个线性组织，但实际上，产品的开发和测试是按照项目来进行的。线性组织通过给出合适的人员和规程等来支持项目。在这种结构中，大量的支持性任务可以被集中起来并有效组织。例如，基础设施可以被用于测试团队和开发团队的不同项目，而且可以进行中心式维护。这对于工具、变更管理和测试自动化架构都是同样的。在项目中，由于团队以外的人员会提供和维护正确的设备，因而人员的焦点就可以放在开发和测试新产品。

在图 19.2 中，可以通过系统开发部和测试部内部的不同团队来形象地表现项目。与各个组织的文化相关，这些团队或多或少在预算、人员和目标方面有各自的自主权，或是它们被置于部门领导的严格控制之下。

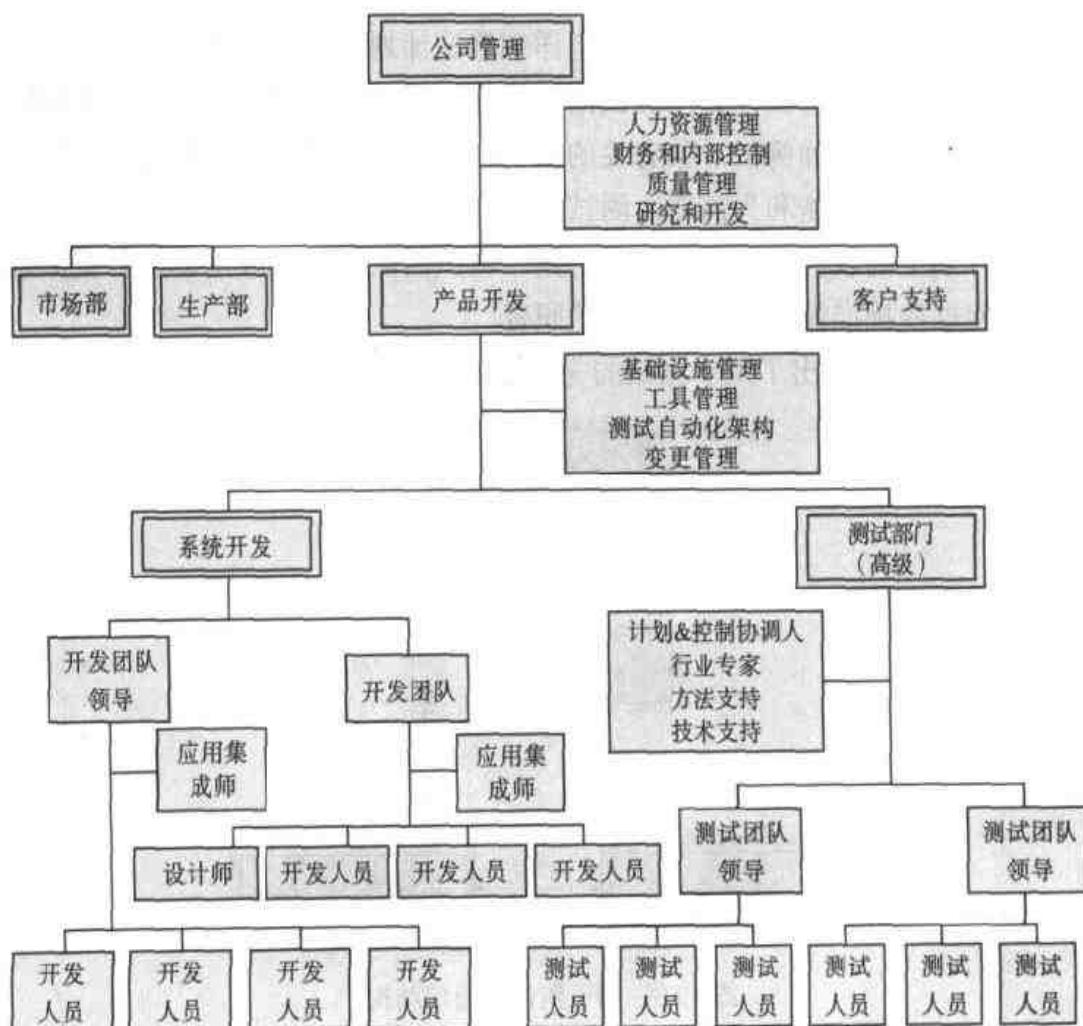


图 19.2 一个线性组织的例子，其中开发活动和测试活动是作为一个项目来执行的

## 19.2 通信结构

公司管理人员批准开发特定的产品。在此决策之前，应当提供大量与开发、生产、市场、维护费用、开发所需的时间和产品盈利能力等有关的信息。这一节只是介绍从想法到产品实现过程中测试的角色。

需要大致估算出包括测试在内的开发费用，但多数情况下无须咨询测试部。按照经验，需要为包括低层次测试在内的总开发时间预留一定百分比的时间，以用于高层次测试。

在项目产品开发开始时，需要成立一个指导委员会，由市场代表、生产代表、客户支持代表、委托人和其他产品权益人组成，委员会小组会议是关于所涉及各种项目进展话题的决策讨论会。这个小组基本上定期以及在任何必要的时候会面。当测试策略制定出来，项目已经开始时，委员会小组成员就已经要面对测试了。测试策略和预算及时间的分配，都是主测试计划的基础。在此过程中，开发团队领导或开发部领导需要参与，因为要决定在哪些测试层次上应当测试什么。

应用集成师可以为低层次测试制定详细测试计划，同时测试团队领导需要为高层次测试准备详细测试计划。测试团队领导应当和开发团队接触，因为他们必须提供测试基础以及和测试对象有关的知识。应用集成师必须为集成测试制定输出标准，这些标准不能和为高层次测试制定的输入标准相抵触。这些标准也可以在主测试计划中的较高层次来定义。否则，测试团队领导和应用集成师就必须对此讨论，使这些标准保持一致是他们的职责。

在图 19.3 中，给出了用于测试的完整的通信结构。

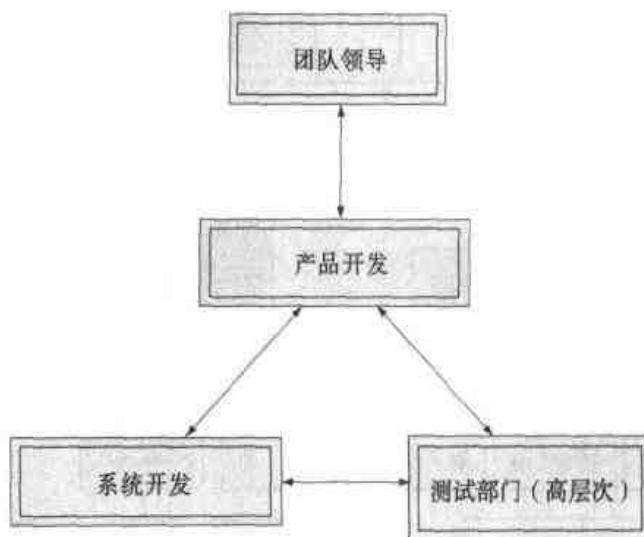


图 19.3 用于测试的通信结构

低层次测试在开发团队内执行。在这个阶段，只要求有与产品开发进展有关

的交流，进展报告要受到中心质量控制的支配。如果开发完成，即集成测试满足其输出标准，那么指导委员会小组就要基于应用集成师的报告，来决定该测试对象是否可以移交给测试团队来进行高层次测试。这还不是高层次测试的开始，因为在测试环境中检查该测试对象是否能够正常工作是很明智的。检查测试对象看主要的功能是否已经具备，评估是否足够可以进行测试（也可见 6.5.1 节）。然后，指导委员会小组才正式批准开始高层次测试。

指导委员会小组也是讨论突发事件报告，做出相应决策的组织。有时候，可以有一个更小的组织来筛选那些困难的事件：测试团队领导和开发领导讨论对每个事件应当如何来处理，只有当他们无法达成一致意见时，事件才被“上报”给委员会小组（也可见 20.3.4 节）。

指导委员会小组的最后一件任务是基于测试报告来决定产品的发布。

虽然必须有一些正式的交流和报告机制存在，但非正式的交流同样重要。非正式接触可以使过程更加容易，并且有助于建立尊重和理解的氛围。测试实际上就是一个有大量交流的过程。

## 第 20 章 测试控制

在一个测试项目中，要执行许多活动来生成测试产品（缺陷、测试设计、测试数据库等）。在有限时间之内，所有东西都只有有限的资源。为了避免造成混乱，就必须对过程和产品建立控制。

在越来越多的组织中，必须提供清晰的报告来证明已经执行了测试，更进一步的要求是必须提供已经处理了缺陷的证明。涉及政府规章制度时，预计这些要求将会更加严格。建立测试控制的方式应该满足严格的可跟踪性要求，和提供证据的要求。这意味着：

- 测试设计要清晰地给出它们是从测试基础的哪些部分导出的。
- 在测试执行过程中，需要提供哪些测试用例已经被执行的证明。
- 指明是哪些测试用例导致哪些缺陷的。
- 在重新测试过程中，记录已经证明哪些缺陷被解决了

测试控制可以进行如下分组：

- 测试过程的控制；
- 测试基础设施的控制；
- 测试交付物的控制。

在接下来的章节中，将更详细地讨论测试控制。

### 20.1 测试过程的控制

过程控制的基础是收集数据、得出结论和初始化校正动作。对于测试过程的控制，需要收集测试项目中时间和预算的使用和测试对象质量的信息。对这两个主题，必须给出以下信息：

- 当前状况，提供判断准则来决定产品是否足够好（或一般来说，业务是否仍然能够满足目标）。
- 趋势，给出开发和测试过程中的问题，提供判断准则来决定它们是否需要被更改。

测试管理人员定期或在请求时，报告时间、预算和质量的状况和趋势。报告频率和形式应当达成一致并在测试计划中描述。除了用图形来表示以外，定期报告也应当提供与活动、交付物、趋势、可能的瓶颈等相关的文本信息。

从测试过程开始，即测试计划被提交的时候开始，定期向客户报告是很重要的。在准备和细化阶段过程中，管理倾向于与测试无关。只有当测试是项目的关键路径时，才对测试执行期间的进度感兴趣。

要找出在测试过程所有阶段中的趋势，重要的是要在尽可能早的阶段找出并报告趋势。这样就可以及时采取措施，这往往是更好、更便宜和更为快捷的办法。

在接下来的章节中，将更详细地讨论测试项目的时间和预算，以及测试对象的质量。

### 20.1.1 时间与预算的使用和进度

在测试生命周期的所有阶段中，都必须在整体上，并在详细级别上来监控测试项目的进展。如果有必要，可以采取及时的措施来校正负面趋势。

根据从测试过程收集的数据来监控进度。到测试过程结束时，测试计划中的活动和交付物与时间和资源相关。在这个框架内，可以收集信息来深入了解测试计划已经实现的程度。

对一个测试过程来说，表 20.1 中的活动和交付物（从生命周期模型导出）可以被用做进度监控的基础：

表 20.1 用于监控进度的活动和交付物

| 活动      | 交付物            |
|---------|----------------|
| 评审测试基础  | 测试基础缺陷         |
| 测试设计    | 测试用例/测试脚本/测试方案 |
| 建立测试数据库 | 测试数据库          |
| 测试执行    | 测试结果           |
| 比较和分析结果 | 缺陷             |
| 保存      | 测试件            |

进度是由测试交付物的完成程度来表示的。每个活动必须登记的基本数据是：

- 进度状况（未开始、忙、暂停、完成的百分比和已完成）；
- 时间和资源（计划的、已花费的和剩余的）。

从这些基本数据可以得到许多指示器，它们有助于监控项目是否仍然“处于正确的轨道上”。一般来讲，这属于项目管理技术，并不是测试特定的。这里鼓励

读者参考项目管理书籍来学习与此有关的更多知识。一些常见的指示器例子是：

- 每个活动类型的时间（计划的、已花费的和剩余的）。例如可测性审查、设计测试用例、第一次测试执行、再测试和经常性活动。当“已花费的”时间加上“剩余的”时间与“计划的”时间不相符时，就要采取诸如修正时间进度、改变活动或改变人员等校正活动。
- 每个测试单元的再测试花费。当这个数目高时，测试项目就会处于时间和预算被用光的危险中，因为测试对象没有足够的质量。测试不应当由于需要更多的时间而受到谴责。测试经理应当问一些简单明了的问题：“你想要让我们执行这些计划外再测试吗？那么提供更多的测试预算。或者你想要坚持原计划的话，那么我们就必须终止进一步的再测试。”
- 每个测试角色的时间或预算。例如测试管理、测试、技术支持和支持方法。这意味着测试组织所必需的某些装备，对于计划未来的测试项目很有用。
- 每个测试阶段的时间或预算。这与所用的生命周期有关，例如计划和控制、准备、详细说明书、执行和完成。该信息对于估计未来的测试项目很有用。

当登记进度时，可以区分开主要活动、管理活动和支持活动。由于效率原因，建议不要很明确地登记所有的小活动。应当定期地评审它们的使用情况（通过与客户磋商）。例如，如果信息量巨大（每周多于1天），那么将管理活动和支持活动分开登记是有用的。

必须向客户或项目经理提供结构化信息，以便深入了解测试过程的进度。为此，能够在测试过程的交付物级别上来报告进度是很重要的。建议在普通数字上加入详细的文字注释。

应当充分利用工具来登记时间和预算。测试经理当然可以开发自己的电子报表，但是已经有大量的商业化工具可供使用。在大多数情况下，测试经理可以与组织中其他部门使用相同的时间登记系统，而且通常这是由组织所要求的。

### 20.1.2 质量指示器

在整个测试过程期间，测试团队都要提供测试对象的质量信息。该信息基于测试活动执行过程中所发现的缺陷，它经常被用于其他报告，以及提供编写正式发布的建议。如果是关于质量的报告，则总是存在基于个人喜好来判断什么是“可接受质量”的危险，这将不会得出任何结论。为了便于进一步给出建设性的质量报告，从一个逻辑思考链来讲，质量报告必须包含以下部分：

- 目标事实。这通常是由与缺陷和时间有关的度量数据组成。这些度量值不

应当受讨论约束。

- **主观结论。**测试经理根据以前报告的目标事实，得出与事件状态有关的结论。其他人可以不同意这个结论。因此，测试经理必须仔细地表达其结论，并用事实来佐证。
- **建议。**当（大多数）管理人员都认可该结论时，那么就可能必须启动校正动作，来解决观测到的问题和风险。测试经理可以通过建议采用一些特定的解决方案来帮助组织，这些方案通常是下面一些类型：
  - 改变时间进度。发布时间将被推迟，以便有更多的时间来消除缺陷。有时侯，当测试只发现很少几个缺陷或只是次要缺陷时，就可以给出相反的建议。
  - 改变发布内容。产品可以在计划的日期被发布，但功能可能被减少。
  - 改变测试策略。对系统脆弱部分执行更多的测试，或者在只表现出很少的缺陷或只是次要缺陷时，减少测试工作量。

可以在所使用的测试基础、测试对象和基础设施中以及测试规程中发现缺陷。20.3.4 节更为详细地描述了记录和管理缺陷的规程，以及记录哪些数据。在测试计划之前，对收集的数据集合和从这些数据得出的质量指示器，达成一致意见会更好。

在测试项目过程中，下面的质量指示器将会对监控将要做的事情和开始校正动作很有用：

- 在某个特定的时间，每个严重级别仍然未解决缺陷的数目。这可以被称为“产品质量快照”，很容易度量并立即提供当前的产品质量信息。
- 产品发布的稳定性。这可以用于指示是否不再对产品进行变更。当仍然有许多变更需要被映射到产品时，产品质量的置信度就低。可以用于该质量指示器的一个简单度量是：
$$\frac{\text{已发现的缺陷数目} + \text{已解决的缺陷数目}}{\text{按照一定的时间周期和每个严重级别来进行度量。}}$$
- 每人每小时测试所发现的缺陷数目。这可以用于指示找出缺陷所花费的时间量。例如“每人每小时可以找出 1.4 个严重级别高的缺陷”可以被转化为：“由三个测试人员组成的团队平均每周工作 30 小时，到下周将可以找出  $3 \times 30 \times 1.4 = 126$  个严重的缺陷”。为了得出这些结论，更好的方式是，只测量在测试用例的实际执行上所花费的时间。
- 每个产品构件（测试单元）再测试的次数。这可以用于指示交付一个足够质量的构件，需要测试的次数，也可以用于指示产品的可维护性或开发过

程的质量。

- 每个缺陷的转变时间。这是用缺陷记录和缺陷消除被交付之间的时间来度量的。就像前一个指示器一样，它也可以用于指示产品的可维护性或开发过程的质量。
- 每个缺陷原因导致的缺陷数目。缺陷原因可以被分类为诸如测试基础、测试对象、基础设施、测试设计和测试规程。这个指示器说明了在哪里采取校正动作最为有效。

前三个指示器尤其适用于验收标准。如果这三个指示器中有一个仍然明显比零大，那么终止测试是不明智的。图形化表示这几个指示器与时间之间的关系相对容易一些，这可以提供清晰而又简单的图形来用于管理：如果一切顺利，所有三个指示器的图形都应当朝零基线运动。在图 20.1 和图 20.2 中给出了这样的图形化报表例子。

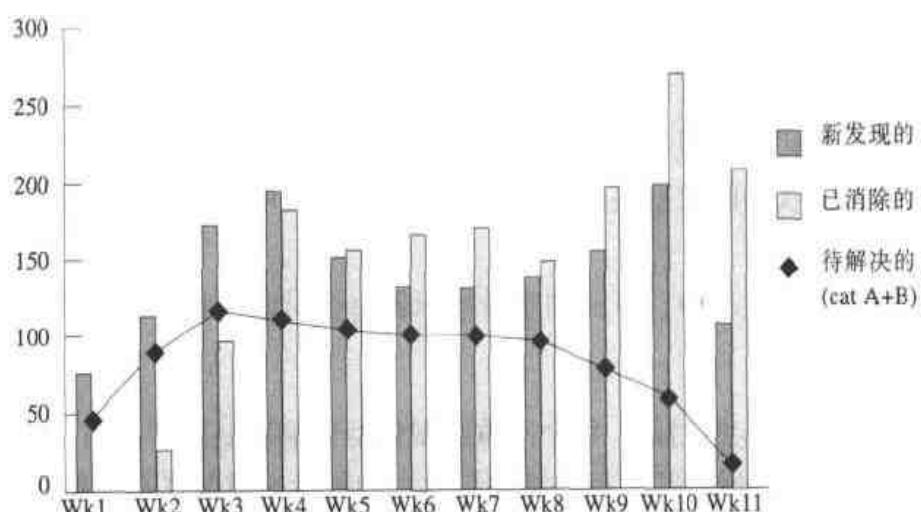


图 20.1 图形化表示未解决缺陷的数目（线型）和产品的稳定性（柱状总和）

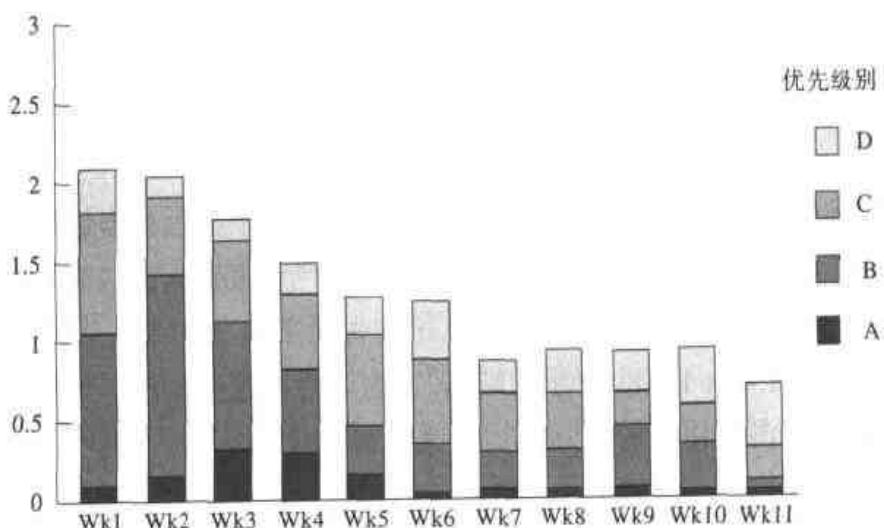


图 20.2 测试效率的图形化表示

在测试项目终止时，下面的质量指示器对于分析测试对象、开发过程和测试过程的质量尤其有用：

- 测试单元（或构件）上缺陷的分布状况。这可以用于指出产品的薄弱部分。
- 每个类型的测试缺陷。例如可以区分以下的类型：详细设计错误、执行错误、结果评估错误和环境错误。这可以用于指示需要在团队中提高哪些测试技能或规程。
- 产品每一单元大小的缺陷数目。这个指示器将产品（发布）质量与其他产品（发布）质量相比较。为了对此进行度量，组织就必须定义如何来度量产品的大小。度量的例子有：代码行数、功能点和类数目。

除了这些指示器以外，在测试项目过程中也可以使用前面提到的质量指示器。它们作为历史数据，可以用于估计和计划未来的测试项目，并评估未来的测试项目和产品。

## 20.2 测试基础设施的控制

在测试项目的早期阶段，定义并订购测试基础设施。在安装和验收之后，测试管理人员负责测试基础设施的控制。

测试基础设施可以被分到下面的部分中：

- 测试环境；
- 测试工具；

为分割控制任务，测试基础设施的各个方面可以被分为两组：

- 技术控制：
  - 测试环境（硬件、软件和控制规程）
  - 测试文件
  - 测试环境和办公环境的网络
  - 技术办公组织
  - 测试工具
- 后勤控制。

办公环境的非技术构件，诸如就餐设施、信息传递、通行证等。

在后勤控制框架中执行的任务，是控制和/或协助服务支持角色的一部分。本书中不讨论这些任务。

执行的技术控制任务是技术支持角色的一部分。当这些任务被执行时，供应商或系统管理人员可以提供帮助。

技术支持角色的最重要的控制任务是：

- 组织；
- 供给；
- 维护；
- 版本管理；
- 解决问题。

在 17.2 节，详细描述了技术支持角色的任务。

### 20.2.1 测试环境和测试工具的变更

在项目过程中，会有许多内部和外部因素引起基础设施的变更，这些因素包括：

- 基础设施的分阶段交付和变更；
- （部分）测试对象的交付或重新交付；
- 新规程或变更的规程；
- 模拟软件和系统软件的变更；
- 硬件、协议、参数等的变更；
- 新的或有变更的测试工具；
- 测试文件、表格等的变更：
  - 将测试输入文件转换为新格式
  - 测试文件的重新组织
  - 命名约定的变更

只有在测试管理人员（可能委托给技术支持人员）的允许下，才能够在技术基础设施中引入变更。测试管理人员在测试控制规程的辅助下，登记变更。一般情况下是根据变更类型和变更数量，来将变更通报给测试团队。

### 20.2.2 测试环境和测试工具的可用性

技术支持人员负责为基础设施定期地创建备份，而且必须能够根据恢复规程来修复基础设施。

测试人员必须将测试环境和测试工具中的故障报告给技术支持人员，然后技术支持人员分析故障，根据故障的严重程度来立即修复。对于不能够立即修复的故障，测试人员必须编写一份正式的缺陷报告。

技术支持人员定期或根据请求，提供关于测试环境和测试工具的可用性，以及已发生故障的（可能）进度和持续时间的报告数据。

## 20.3 测试交付物的控制

在这个过程中，要区分外部交付物和内部交付物。外部交付物包括测试基础和测试对象。内部交付物包括测试件、测试基础（内部的）、测试对象（内部的）、测试文档和测试规章制度。

### 20.3.1 外部交付物

外部交付物的控制是外部职责，虽然大多数测试项目往往缺乏测试基础和测试对象的外部控制。因而，需要为测试团队设置与这些规程有关的要求。

### 20.3.2 内部交付物

#### 测试件

测试件被定义为：在测试过程期间产生的所有测试交付物。需求之一是这些产品必须被用于维护目的，因而必须是可移交、可维护的。测试件包含以下内容：

- 测试计划；
- 逻辑测试设计；
- 测试脚本；
- 测试方案；
- 可跟踪矩阵（给出哪些测试用例覆盖哪些需求）；
- 测试输出；
- （包含缺陷的）测试结果和报告。

#### 测试基础（内部的）

属于测试基础的文档都按照其原有形式来保存。控制为测试活动提供备份并登记分配的版本号等。

#### 测试对象（内部的）

在组织内部，通过规程和其他可用的方法来控制软件和硬件。文档的管理方式与测试基础中的管理方式相同。

### 测试文档

在测试过程中，收到的或编写的各种文档只是对这个特定的项目有意义。在测试项目完成之后，除了成为项目的“历史记录”以外，它们基本上就不再具有任何价值。文档的种类包括：

- 项目计划（不是测试计划）；
- 会议报告；
- 信件；
- 备忘录；
- 标准和指导手册；
- 测试、审查和审计报告；
- 进度和质量报告。

控制支持角色可以帮助正确地归档和提取测试文档。

### 20.3.3 测试交付物的控制规程

这一规程的目的是管理内部交付物。在这个框架内执行的任务，是控制支持角色的一部分。对变更测试规章制度的特定控制，通常是方法支持人员的任务。在这一规程中，术语“交付物”是指在测试过程期间交付的所有产品。

测试交付物的控制规程由四个步骤组成：

- **交付。**测试人员将必须受控的交付物提供给控制人员。交付物必须是作为一个整体来提供，例如应当包含一个版本日期和一个版本号。控制人员检查交付物的完整性。
- **登记。**控制人员基于一些数据来将交付物登记到管理程序中，例如提供者的名称、交付物的名称或日期和版本号。当登记发生变更的交付物时，控制人员应当确保各交付物之间的一致性。
- **归档。**要对新文档和变更交付物加以区别。一般来说，这是指新的交付物被增加到档案库中，变更交付物取代原有的版本。
- **发布。**向项目组成员或其他人员提供交付物，这是通过向他们提供所需交付物的一份拷贝来进行的。控制人员登记所提供的交付物的版本、交付给谁以及何时交付。

### 20.3.4 缺陷管理

测试经理可以利用缺陷管理来跟踪缺陷的状况。这可以向他提供迄今为止产

品测试的进度和质量报告。缺陷管理帮助开发团队的领导将缺陷分配给具体的开发人员，并查看开发团队的进度和工作量。需要对内部缺陷和外部缺陷加以区分。内部缺陷是指测试团队引发的错误，外部缺陷是指测试基础、测试对象、测试环境和测试工具中的错误。

图 20.3 给出了交付物和缺陷之间的关系。当一个交付物被交付给测试人员进行正式测试时，需要执行入口检查，确定交付物是否具备了足够的质量以供测试。如果交付物的质量不够，那么交付物就不能被接受进行测试，而是应当返还给交付物的提供者。如果交付物被接受，那么开始执行测试，执行结果将与预期结果相同或不同。如果是后一种情况，就意味着必须进行分析来确定预期结果和实际结果之间出现差异的原因。如果原因是测试错误，那么测试团队就可以解决这个问题。但这并不总是意味着测试必须被再次执行，例如在预期结果错误而实际结果为正确的情况下就无需再进行测试。

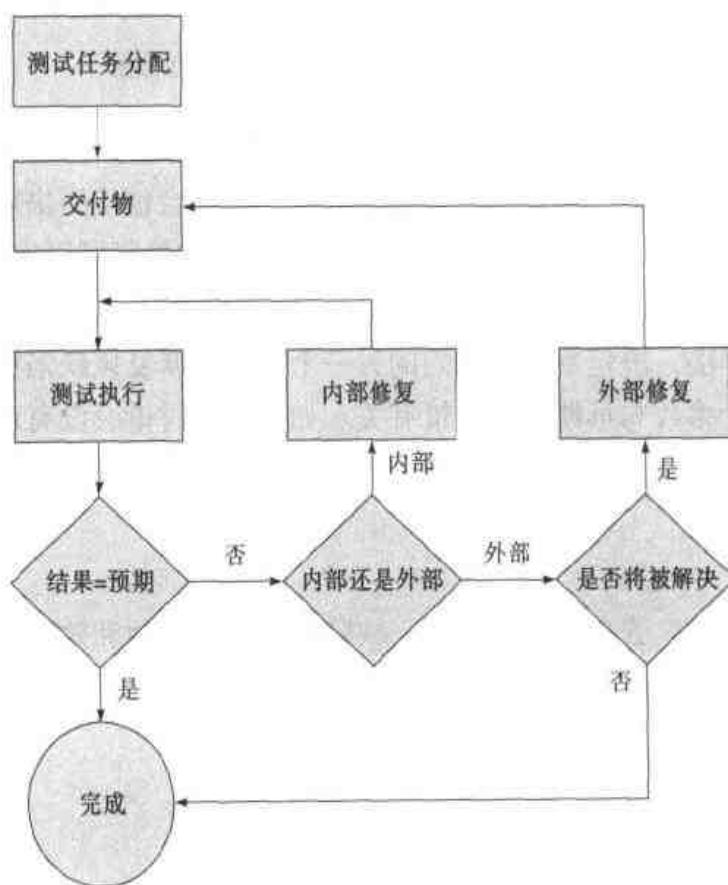


图 20.3 交付物和缺陷

如果预期结果和实际结果之间出现差异的原因是在测试团队之外，那么就要决定是解决该问题还是接受这一缺陷（作为一个“已知错误”）。图 20.4 中给出了如何来做出决定的详细规程。

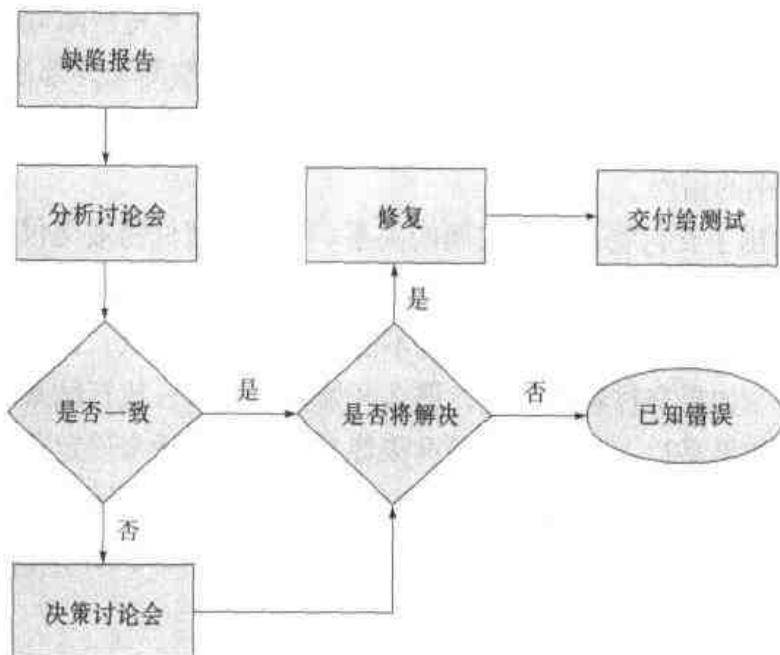


图 20.4 决定缺陷报告和变更请求的规程

在分析讨论会上，讨论涉及外部缺陷的所有缺陷报告。分析讨论会的任务是接受或拒绝出现的缺陷，以及将这些缺陷分类。讨论会也应当决定是否必须要修复缺陷，还是接受缺陷作为“已知错误”。这些缺陷的修复可以推迟到下一版本，而且要通过分类决定是否要解决缺陷。由于缺少时间和人力，不严重的缺陷大多数时间可以不解决。决定不解决缺陷的另一个原因是修复该缺陷将造成重大影响，这类缺陷大多数与早期设计决策有关。如果分析讨论会没有对某些缺陷达成一致意见，那么必须由决策讨论会来对这些缺陷做出正式决策。

### 缺陷报告

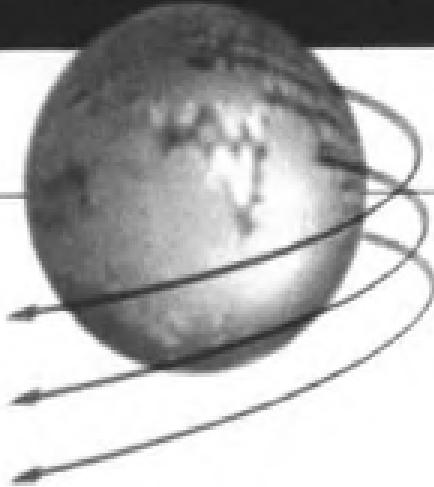
缺陷报告是缺陷管理的中心项目。缺陷管理的有效性和效率与缺陷报告质量密切相关。每个缺陷报告至少应当包含以下项目：

- 惟一标识；
- 缺陷类型；
- 测试对象；
- 测试脚本；
- 使用的测试基础版本号；
- 描述缺陷和引发缺陷的基本步骤，或测试脚本步骤参考；
- 发现缺陷的测试人员名字；
- 日期；

■ 参考使用的测试环境。

缺陷报告应当提供足够的信息来快速解决缺陷。这是指在单元测试或集成测试过程中，如果是由开发人员来执行测试，那么开发人员必须自己解决缺陷；当缺陷是由独立测试团队发现的，那么就需要较少的信息。

有大量免费、共享或商业化的工具可用于缺陷管理和编写缺陷报告。大多数商业化工具可以定义安全级别和访问级别，而且能够提供工作流管理。



## 第六部分 附录

## 附录 A 风险级别

风险级别表被用于确定系统故障的风险。根据对人类生命的危及程度来确定风险。风险级别表给出了每一种可能的严重性和可能性的组合。

如果一个公司要设计和/或生产强调安全系统，那么就必须有用于故障概率、毁伤级别和风险级别的标准。公司也可以使用或开发自己的标准，或使用工业标准。有时候，法律强制要使用工业标准。工业标准的一个例子是 RTCA DO-178B (1992)，该标准被航空当局用于验证航空软件。在本章中风险分类的例子大体上基于 RTCA DO-178B 标准。风险级别、故障概率和毁伤的定义是建立安全管理的一部分。

风险的定义为：

$$\text{风险} = \text{故障概率} \times \text{毁伤}$$

为了更容易区分风险级别，就要制定出风险的级别（见表 A.1）。大多数情况下，不可能准确地确定故障的可能性。由于这个原因，需要使用概率范围（见表 A.2）。毁伤也被分了类，它是对人类伤害情况的一种转换形式（见表 A.3）。

现在将故障概率和毁伤放到一个新表中（见表 A.4）。对每一个概率和严重性的组合，确定它属于哪一级风险。确定方式依赖于项目、组织、系统的使用，等等。这个表也说明，并不需要详细讨论故障概率或毁伤属于哪一个类别，因为二者的组合才有意义。在 FMEA 故障模型及后果分析中只用到了表 A.1 中的风险级别。

表 A.1 风险级别表，A 级是最高的风险级别

| 风险级别 | 描述                            |
|------|-------------------------------|
| A    | 无法容忍的                         |
| B    | 不期望有的风险，而且只有当风险降低到不可接受时才可以被接受 |
| C    | 项目安全审查委员会认可就可以容忍              |
| D    | 一般项目审查认可就可以容忍                 |
| E    | 所有的条件都可以容忍                    |

表 A.2 概率范围

| 意外事件频率 | 运行生命期间发生                                   |
|--------|--------------------------------------------|
| 频繁     | $10\ 000 \times 10^{-6}$ ; 可能是连续发生         |
| 可能     | $100 \times 10^{-6}$ ; 可能是经常发生             |
| 偶尔     | $1 \times 10^{-6}$ ; 可能发生几次                |
| 极少     | $0.01 \times 10^{-6}$ ; 可能有时候发生            |
| 不可能    | $0.0001 \times 10^{-6}$ ; 不可能, 但在意外情况下可能发生 |
| 难以置信   | $0.000\ 001 \times 10^{-6}$ ; 事件根本不可能发生    |

表 A.3 意外事件严重性级别

| 级别   | 定义                              |
|------|---------------------------------|
| 灾难性的 | 多人死亡                            |
| 危险的  | 一人死亡, 和/或多人严重受伤或严重的职业疾病         |
| 重大的  | 一人严重受伤或严重的职业疾病, 和/或多人轻伤或轻微的职业疾病 |
| 次要的  | 最多一人轻伤或轻微的职业疾病                  |
| 可忽略的 | 没有影响                            |

表 A.4 风险分类表

|      | 灾难性的 | 危险的 | 重大的 | 次要的 | 可忽略的 |
|------|------|-----|-----|-----|------|
| 频繁   | A    | A   | B   | D   | E    |
| 可能   | A    | A   | B   | E   | E    |
| 偶尔   | A    | B   | C   | E   | E    |
| 极少   | A    | B   | C   | E   | E    |
| 不可能  | B    | C   | D   | E   | E    |
| 难以置信 | C    | D   | D   | E   | E    |

## 附录 B 状态表

### B.1 状态

在基于状态的系统中，系统的不同条件被描述为状态。可以将状态定义为系统的一个清晰的、可以区分开的独立条件，并持续一段有效时间。术语“独立”是指在任何一个时刻，只可能有且只有一个状态存在。

初始状态或默认状态是第一个事件被接受时的状态（在图 B.1 中为关闭状态）。在某一特定时刻，系统所处的状态被定义为当前状态。

一个系统有多个状态。例如，在图 B.1 中的状态有关闭、开启和录音。系统对事件做出响应而改变状态（例如，在图 B.1 中的 evPowerOn 事件）。状态的改变被称为转换（例如，在图 B.1 中从关闭到开启的转换）。如果一个事件导致一个转换，那么旧状态被称为接受状态，新状态被称为结果状态。如果系统处于可以接受事件的状态，那么该状态也被称为活动状态。反之被称为最终状态，即系统不再接受事件的状态。在最终状态时，系统本身处于其终止状态。

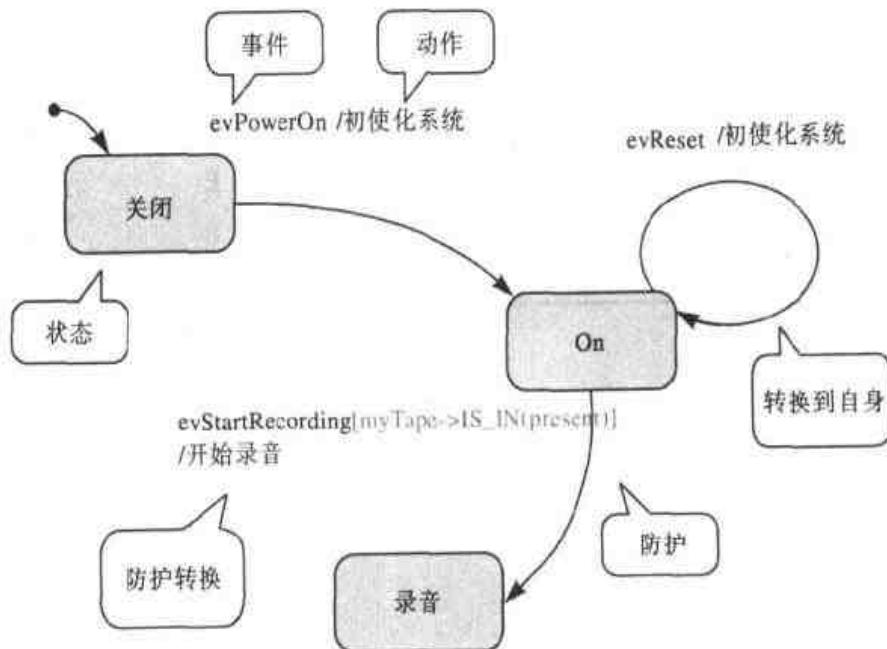


图 B.1 一个录音机的部分状态图

## B.2 事件

系统内部或外部发生的事件能够导致转换的发生。总共有四类事件：

1. **信号事件**：在状态机范围之外发生的异步事件。
2. **调用事件**：一个对象显式地同步通知另一个对象。
3. **改变事件**：基于属性改变的事件。
4. **时间事件**：计时器到期或到达某个绝对时间。

当测试时，要认识到系统可以按照三种不同的方式对一个事件做出响应：

1. 系统通过进行转换来对事件做出响应。
2. 当系统不是处于正确的状态，或者进行转换条件不满足时，事件被忽略。  
因为并没有指定对于这些事件，系统应当如何做出响应，因而这些事件信息被系统抛弃。
3. 当一个同步伪状态是状态图的一部分时，系统的响应就更为复杂。在一个同步伪状态中，存储有某些事件的信息。如果系统处于状态 1a 且事件 3 发生（见图 B.2），那么事件 3 的信息被存储在同步伪状态中（中间为数字 1 的圆），发生转换进入状态 3a。在正交区域 A 中的行为并不受同步伪状态的影响。在正交区域 B 中的行为就不太那么简单明了。如果系统处于状态 1b 且事件 5 发生，那么只有当同步伪状态被填充时，才转换到状态 2b。这个转换的一个结果就是在同步伪状态中，事件 3 的信息被清除。如果事件 5 发生，而同步伪状态中没有信息，那么系统仍然处于状态 1b，而且事件 5 的信息被系统抛弃。

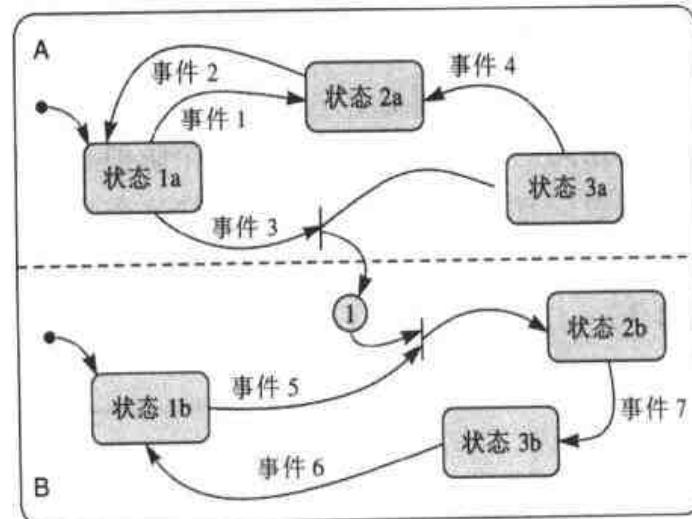


图 B.2 存储有事件信息的正交区域之间的同步

## B.3 转换

转换就是从一个状态改变为另一个状态。如果结果状态和接受状态相同，那么转换被称为到自身的转换。在图 B.1 中，事件 evReset/初始化系统将系统从状态开启改变为状态开启。如果是在状态中发生转换，那么该转换被称为内部转换。

条件可能被附加到状态上，这被称为防护转换。这类转换的一个例子是图 B.1 中，由事件 evStartRecording 导致的转换。只有当磁带存在时，该事件才触发一个转换，这是由防护 myTape-IS\_IN(present) 来限定的。如果在同一个状态上，一个事件能够导致多个转换，那么防护可以将这些转换相互分隔开。这些防护不应当有重叠，因为在重叠区域，无法确定系统的行为。为了更清晰地描述这样一个图形表示，可以用 UML 中的条件连接器。在这种情况下，一个标记有事件的转换离开这个状态，终止于条件连接器，然后防护转换转向不同的结果状态。

## B.4 动作和活动

动作（图 B.1：初始化系统和开始录音）被定义为在状态机的某个特殊点，执行不可中断的行为，该行为被认为需要花费一定量的时间。一个动作也可以是一个整体不能被中断的一串动作。一个动作可以被分配给状态的人口或出口（用人口和出口状态标记），或执行转换（例如图 B.1 中，从开启状态到录音状态的转换上的开始录音）。活动不同于动作，因为活动需要花费一定量的时间，因而可以被中断。因此，只能在状态中来定义活动。例如，开始倒带是一个动作，而倒带是一个活动。如果当状态仍然处于激活时完成了活动，那么将生成一个完成事件。在离开完成转换的情况下，状态将被离开。如果磁带已经倒完，那么活动“倒带”终止，系统给出一个不能再倒带的信号。该信号确保倒带活动将停止，系统从状态倒带转向状态待机（见图 B.1）。

动作和防护可以引用同一个变量。例如，在一个防护中，评估某个变量必须要小于一个给定的门限值。与该转换相关的动作改变这个变量。虽然有的人认为这些结构不正确（不好的编程习惯），但在用于给计数器建模的同步时间模型中，它们很常见（见图 B.3）。

## B.5 执行顺序

动作是按照一定的顺序来执行的。首先执行的是接受状态的出口标记中定义的动作，接下来是指向转换的动作，最后是结果状态的人口标记中定义的动作。

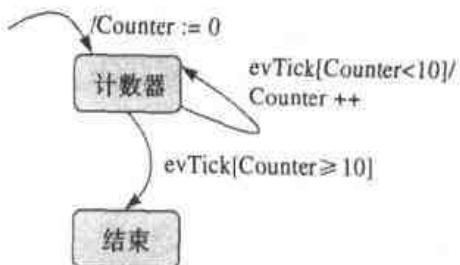


图 B.3 在到自身的转换中，防护和动作引用同一个变量

这同样可以应用于嵌套状态，但这里涉及的情况有些复杂。图 B.4 给出了一些由嵌套状态组合而成的转换。入口动作是按照与嵌套相同的顺序来执行的。

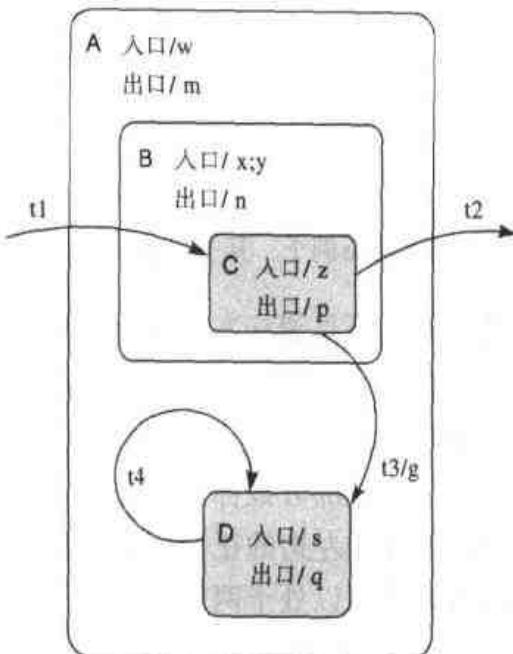


图 B.4 动作和嵌套状态

如果转换  $t_1$  被执行，首先执行的动作是  $w$ ，然后是  $x$  和  $y$ ，最后是  $z$ 。对于转换  $t_2$ ，动作的执行顺序是由里到外。所以第一个执行的动作是  $p$ ，然后是  $n$ ，最后是  $m$ 。如果转换  $t_3$  被执行，首先执行的动作是  $p$  和  $n$ ，然后是  $g$ ，最后是  $s$ 。如果转换  $t_4$  被执行，首先执行的动作是  $q$ ，接下来又到达状态  $D$ ，执行动作  $s$ 。如果一个内部转换被执行，那么与该状态相连的入口和出口动作就不被执行，仅执行与转换直接相连的那些动作（即在转换标记  $u/u''$  后面部分出现的动作）。

## B.6 嵌套状态

状态图鼓励人们使用抽象。细节部分可以用超状态来隐藏。超状态本身由许

多被称为子状态的状态所组成。这些子状态或者用同一个图来描述，如图 B.5 一样，或者在另一个状态图中详细描述超状态。采用这种方式，就可以用多级抽象来形成一个结构。

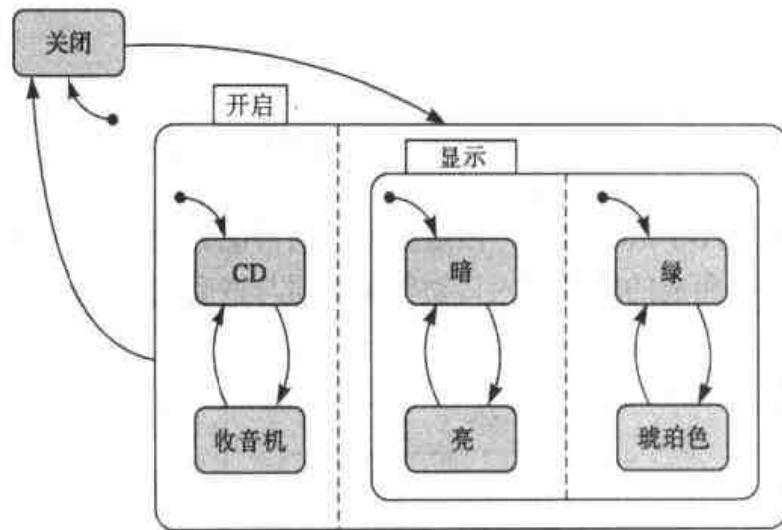


图 B.5 一个汽车收音机的状态图例子，其中超状态开启由两个正交区域组成。

在图 B.5 中，状态开启和显示是超状态，左边区域包含两个子状态，右边区域包含子状态显示，这个子状态又是一个超状态，可以分解成多个子状态。该图中对这些超状态进行了详细描述。但是也可以在另一个图中给出详细描述。状态开启和显示都由两部分组成，这些部分被称为正交区域。当汽车收音机处于超状态开启时，那么汽车收音机既处于左边的正交区域，也处于右边的正交区域。正交区域被称为与状态，因为系统可以处于两个区域。但在超状态开启左边的正交区域，存在着或状态。汽车收音机处于 CD 状态或收音机状态，但在同一时刻不能既处于 CD 状态又处于收音机状态。如果汽车收音机离开状态开启，那么将离开超状态开启的所有正交区域。

## 附录 C 一个自动化测试包的设计方案

### C.1 测试数据

测试数据被存储在数据库或文件系统中。按照测试内核可以理解的格式来输出数据，可以是一个文本文件格式，或者测试内核通过 ODBC 链路与数据库相连。测试数据有两种类型。测试方案中包含有必须执行什么样的测试脚本，以及何时必须执行等信息（见表 C.1），由测试内核来进行调度。另一种类型是测试脚本。测试脚本中包含了必须对什么进行测试以及如何来进行测试（见表 15.1）。

表 C.1 测试方案设计，当第一列为空时，这一行其他的列都被解释为注释

| 测试方案 1：测试电子组织者的地址簿。 |         |                  |       |
|---------------------|---------|------------------|-------|
| 测试脚本标识              | 测试脚本名称  | 执行时间             | 注释    |
| 脚本_1                | 脚本_地址_1 | 12:01 2001-03-03 | 只添加地址 |
| 脚本_2                | 脚本_地址_2 | 12:03 2001-03-03 | 只删除地址 |

可以有多种方法来检查一个特定动作的结果。

1. 检查是动作的一部分。动作从测试脚本中的输入开始，在测试脚本的输入参数给出之后定义的是预期输出。动作的最后一个步骤是检查结果是否与预期输出相一致。
2. 动作还可以被用于检查。例如，可以用动作“查看\_地址”来查找一个地址，或用于检查地址簿中的地址是否按正确的方式被保存。
3. 检查被定义为另一个动作。动作就是检查被测系统（SUT）的某个状态或某个输出。

基本上，都是用逗号分开的文件来输出测试方案和测试脚本。

### C.2 启动

测试包在启动的过程中，初始化测试包环境。初始化所有的参数，如果有必要，加载正确的模块，建立与 SUT 的连接。下一步是初始化 SUT。这个初始化或

者是启动 SUT，要不然就只是建立 SUT 的初始状态。启动模块也要启动和关闭进度报告，并很平滑地终止测试循环。

### C.3 Planner

planner（计划器）读取测试方案。测试方案交付与应当执行的测试脚本有关的信息，但并不强制要求何时来执行这些脚本。如果测试方案没有包含执行时间信息，那么按照方案中提到的同样顺序来执行测试脚本。

planner 与测试方案是测试包的驱动力。如果测试方案没有提供新的信息（到达列表结尾或所有调度的任务都执行完毕），那么测试包生成进度报告和状态报告，然后终止。

planner 基本上只是一小段读取测试方案的循环（见图 C.1）。可以在其中增加一些报告功能和调度机制。当 planner 完成时，将控制权返还给起始模块。

```
Planner(test_scenario){
 open_file (test_scenario);
 while NOT_END_OF_FILE
 {
 read_line (test_scenario, line);
 split (line, scr, ","); // 用分隔符来分割一行，并将各部分存储到
 // 一个指针
 if scr[1] != "" // 如果第一列为空，那么这是一个注释行
 Reader (scr[2]); // 想定的第二列存储要执行的脚本名称；
 // 将在 C.4 节解释 reader.
 }
 close_file (test_scenario);
 return;
}
```

图 C.1 用伪代码写的一个 planner 的例子

### C.4 Reader

reader（读取器）提供了需要打开的测试脚本的名称，reader 从测试脚本提取信息。当最后一个测试用例执行完之后，reader 将控制权返还给 planner。

在测试执行过程中如果发生意外，则 reader 将触发错误恢复（见图 C.2）。

```
Reader(test_script) {
 last testcase = "";
 error = NULL;
 open_file (test_script);
 while NOT_END_OF_FILE{
 if error == 1{//如果发生致命错误，跳至下一个测试用例。
 while read_line (test_script, line) != NOT_END_OF_FILE {
 split (line, case, ",");
 // 如果发生致命错误，则转向下一个测试用例
 if case[1] != last testcase{
 last testcase = case[1];
 error = NULL;
 break;
 }
 }
 } else {
 read_line (test_script, line);
 split (line, case, ",");
 last testcase = case[1];
 }
 if *case != "*{
 if Translator (case) == ERROR { // C.5 节解释 translator
 Write_to_status_report ();
 Error_recovery ();
 error = 1;
 }
 }
 }
 close_file (test_script);
 return;
}
```

图 C.2 用伪代码写的 reader 的例子

## C.5 Translator

translator (转换器) 将测试脚本给出的动作链接成测试包库中的相应函数。在一个函数执行之后，translator 将控制权及执行函数的返回值返还给 reader (见图 C.3)。

```
Translator (action){
 switch (tolower((action[2]))
 {
 case "action1":
 return (Action1(action));
 break;
 case "action2":
 return (Action2(action));
 break;
 default: Write_to_status_report ("Test action does
 not exist!");
 return -1;
 break;
 }
}
```

图 C.3 一个 translator 例子。默认返回值为错误

## C.6 测试动作

动作总是一个系统特定的函数。系统特定的函数可以只覆盖系统中一个很小的动作，也可以覆盖系统中的一部分过程或整个过程。覆盖整个过程的高层次动作可以引用低层次动作。低层次动作使测试包更为重要灵活，但高层次动作可以使测试包更容易使用。每个实现动作的函数都是建立在一些标准元素之上 (见图 C.4)。

```
system_specific_function(){
 synchr_func() 至少要有一个同步函数，而且大多数为第一个函数
 <synchr_func> 同步函数和检查函数必须有返回值；应当用 return 语句将这个值
 返回给 translator

 <check_func>
 <common_func> 通常利用自建功能，比如数据处理

 <tool_func>
 return return_value}
```

图 C.4 这些是系统特定函数的元素。在<>之间的函数不是强制性的

## C.7 初始话

有四种类型的初始化：

1. 测试包的初始化。
2. 启动系统并设定初始状态。
3. 其他测试设备的初始化。
4. 在出现故障或执行完测试用例之后，恢复到初始状态。

### 测试包的初始化

在测试包启动时执行初始化。需要设置所有的环境变量，终止所有隐含的和可能有妨碍的进程。设置测试方案变量和测试脚本目录，并设置进展和缺陷报告目录。最后加载正确的模块。

### 启动系统

启动被测系统。测试包可以以手动或自动方式来启动系统。在这步之后，就必须建立初始状态。在基于状态的系统中，这基本上与系统的初始状态一致。

### 其他测试设备的初始化

其他测试设备必须被启动，而且也必须被设置为正确状态。这可以通过手工完成，但如果这些设备完全由测试包控制，那么也可以自动完成。其他测试设备的例子可能有信号生成器或监视器。

### 恢复到初始状态

在测试过程中，系统并不总是像预期的那样来执行。如果系统出现意外行为，那么就必须初始化系统来执行下一个测试用例。有时候，这意味着必须全部重新启动系统。这个功能被称为错误恢复。而有时候需要重新设置系统上必须执行的测试用例。

---

## C.8 同步

同步是指测试包与被测系统之间的协调，使得被测系统准备好时，测试包只产生输入并对输出进行检查。同步错误通常是由测试包的故障引起的。同步的可能方法有：

- **GUI 指示器：**如果使用 GUI，那么大部分时间里可以将指示器用做同步点。例如，窗口名称发生变化，或某个特定对象的外形发生变化。
- **信号：**在一个端口上一个特定信号出现或消失，或是信号改变。
- **超时：**应当尽可能避免使用这种方法。超时是指在一定的时间之后，系统必须准备好。只有当超时对系统很明确有必要时，才可以使用超时方法，要确保测试组确实能和最大可能的超时同步。这通常会降低测试包的性能。

## C.9 错误恢复

如果在测试执行过程中发生意外，测试包必须能够重新设置被测系统，并继续下一个测试用例。每个新测试用例必须从系统的初始状态开始。错误恢复处理的是意外情况。错误恢复有以下几个任务：

- 将合适的信息加到状态报告与进度报告中。
- 必要时通过调用一个初始化函数将系统置于其初始状态。
- 将控制权返回给 reader。

错误可能有多种类型。对不同类型的错误，采用的错误恢复机制也不相同。有以下类型的错误：

- **不是致命的错误条件**，可以继续执行测试用例。将该错误记入日志，然后继续执行测试。
- **实际输出与预期输出不相符**。将该错误记入日志并将系统置于其初始状态。
- **意外行为**。由于未知的原因使测试包的运行失去同步。将该错误记入日志并将系统置于其初始状态。可能惟一的方法就是重新启动系统。
- **致命错误，系统不再有任何响应**。将该行为记入日志并重新启动系统。如果系统无法重新启动，则终止测试。

## C.10 报告

可以完全根据组织需求来定制测试报告的格式。进展报告与测试环节相关，它给出被执行的测试脚本的概要信息及执行结果。

在进展报告中的一些典型主题有：

- 日期、时间、测试环境描述及测试包描述（必要时）。
- 在这一测试周期执行的测试脚本。

- ※ 总的执行时间。
- ※ 哪些测试脚本发现了缺陷。
- ※ 被执行的测试脚本与测试用例的总数。
- ※ 检测出的缺陷数目。
- \* 如果有一个缺陷分类系统，那么也可能要按照类别将缺陷进行排序，并与所涉及的测试脚本建立关联。

可以用多种方法来生成状态报告。一种方法是每发现一个缺陷，就生成一个状态报告，另一种方法是对每个测试脚本生成一个状态报告。后者多少相当于测试脚本级的进展报告。缺陷报告中的典型主题有：

- ※ 日期、时间、测试环境描述及测试包描述（必要时）。
- ※ 产生缺陷的测试用例，或如果测试通过则为 OK。
- ※ 预期情况。
- ※ 检测情况。
- ※ 预期情况与检测结果的差异。
- ※ 缺陷的严重性类别。
- ※ 发现缺陷的系统部分。

生成这些报告所需的功能主要是依赖于报告格式。常见的格式主要有：

- ※ ASCII 码文件：报告以平面文件格式生成，这可能是最容易实现的方式。
- ※ RTF 文档：测试包使用 RTF-解析器，按照组织定义的标准格式来生成报告。
- ※ HTML 文档：可以将 HTML 文档放在服务器上，有权限的人可以读取。另一种可能是将文档直接放在 Web 服务器上，通过内部网来访问。
- ※ 电子邮件：通过电子邮件直接将进展报告送给测试经理，直接将缺陷报告送给测试经理和开发部领导。也可以使用含有文档跟踪和任务调度系统的工作组应用程序。
- ※ XML 文档：XML 文档可以以不同格式来展示报告。例如，可以向管理人员展示详细报告，给测试员的是侧重信息而不带修饰的小型报告。
- ※ 与缺陷管理系统相连：直接将缺陷加入到缺陷管理系统中，然后使用缺陷管理系统的功能来进行处理。
- ※ 放入数据库中：报告被存储在数据库中。必要时可以通过硬拷贝来生成报告。

## C.11 检查

检查对测试来说是绝对必要的。自动化检查可以大大减轻工作量，而且使用自动化测试，可以对更多的项进行检查，因为这种方法更快且不容易出错。这也是一个缺陷，因为可能使人们倾向于对太多项进行测试。

检查是指将实际输出与预期输出进行比较。实际输出可能是一个信号、一个状态、图像或数据。常规检查由三个动作组成：

1. 提取输出
2. 将实际输出与预期输出进行比较
3. 将比较结果写入状态报告，并给出正确的返回值

预期输出可以作为测试动作的参数出现或存储在文件中，可以在线自动比较，或在测试之后使用专门的工具或手工进行比较。有时候会由于输出太大而无法在运行时处理。在这种情况下，必须将输出存储，以便在测试执行完成之后进行分析。

一些信号分析仪能够将输出流与预期流进行比较，这是在测试包之外对数据进行比较。分析仪必须按照测试包可以读取的格式来提供比较结果。测试包使用该信息将比较结果写入报告，必要情况下进行错误恢复。

## C.12 框架

大量的维护性问题是由于冗余代码引起的。间或要对冗余代码进行搜索。用函数来替代所有的冗余代码，这些函数被存储在支持模块中。通过将支持函数存储在另一些模块中，就形成了一个金字塔形的框架，上层都是建立在底层功能之上的。

框架不仅存储代码，而且也存储测试包的设计。在测试包的设计中，可以将相同的层次看成模块。

## C.13 通信

可以通过一个硬件层或一个软件层，来建立被测系统与测试包之间的通信（见图 C.5）。有时候，为了建立稳定的通信机制，需要将软件加到被测系统中。这在其他情况下可能并不合适，例如对被测系统的真实行为进行测试。如果通信不占用被测系统的任何资源，而且在最终产品中不引入其他的代码，那么这种通信方式被称为非插入式。

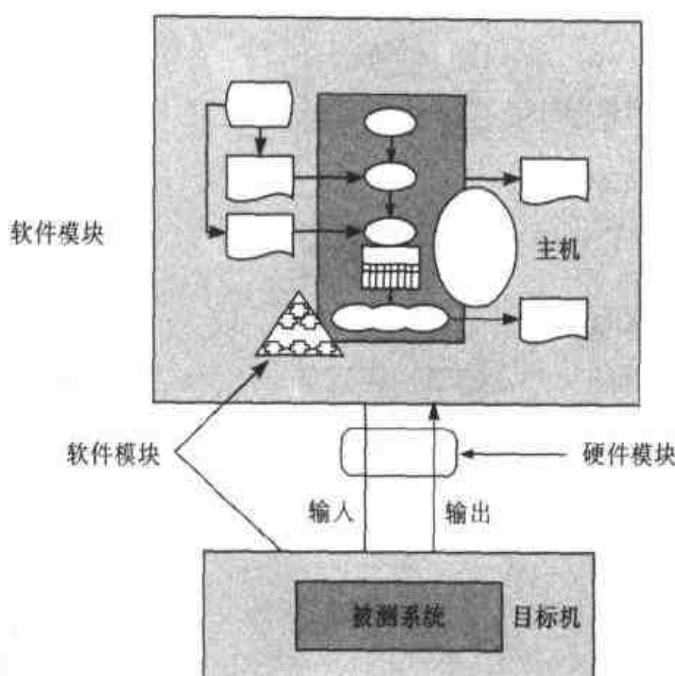


图 C.5 测试包与被测系统之间可能的通信方式

可以用三种方式来建立通信：

- 使用硬件，非插入式；
- 使用软件，插入式；
- 使用软件，绕过外部接口。

#### 使用硬件，非插入式

在这种情况下，只可能使用一个专门的硬件层来与被测系统进行通信。可以采用商业成品设备、采用自己专门建造的构件或结合二者来建立这个硬件层。也可以用一些工具来建立通信，不过这个硬件层应该已经成为该工具的一部分。

与 SUT 的通信在硬件层建立（见图 C.6）。硬件层生成适用于 SUT 处理的所需输入，SUT 生成的输出由硬件层监视。或者 SUT 从一个状态转换到另一个状态，生成的状态信息被硬件层监视。

测试包控制硬件层（见图 C.6）。测试包基于测试脚本告诉硬件层该做什么。有时侯，如果硬件层能够比较实际输出与预期输出，那么也就可以给出预期输出的信息，或将预期输出存储在某个位置。在这种情况下，硬件层应当提供比较结果是通过还是失败的信息。

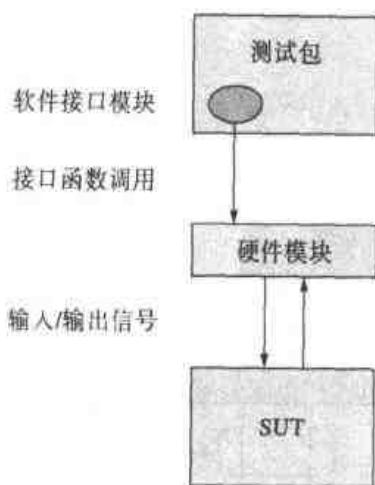


图 C.6 示意性描述一个使用非插入式硬件层的测试包

硬件层的例子有：

- 信号产生器/分析仪；
- 模拟环境；
- 用 PCB 插入目标的特殊设备。

#### 使用软件，插入式

当不可能或很难使用外部接口来测试系统时，采用这种方法。为了控制 SUT，一个软件模块被置于 SUT 的内部（见图 C.7），这样就可以与测试包通信。它使用 SUT 的资源，这意味着这种方法并不适用于每一个 SUT，因为有时系统资源十分有限，不能保证软件模块和 SUT 的正常运行。这种方法完全不适用于进行性能测试。软件模块使用 SUT 的资源可能会对 SUT 的性能产生负面影响。

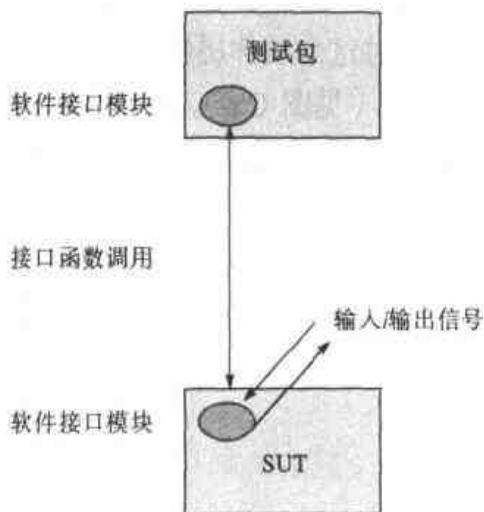


图 C.7 通过在被测系统中植入软件模块，建立测试包与被测系统之间的通信

SUT 内部的软件模块应当控制 SUT，它还要向测试包提供必要的信息。另一方面，测试包必须控制这个软件模块。

实现 SUT 与测试包之间通信最容易的办法是使用 SUT 的一个现有端口。由测试包中的通信模块来处理测试包与 SUT 之间的通信（见图 C.7）。

### 使用软件，绕过外部接口

在以下情况中，这种方法可以给出一种解决方案：

- ※ 外部接口十分复杂，不能够用于自动化测试；
- ※ 没有外部接口或很难访问；
- ※ 外部接口不能提供足够的信息来判断 SUT 的运行是否正确。

绕过外部接口（如果有），使用一个内部接口来控制 SUT（见图 C.8）。这种方法的一个例子是从外面直接访问类，不是用接口来访问，而是由测试包内部的通信模块提供通信。在 SUT 与测试包之间，也应当有一个连接。如果没有外部接口，那么就必须构造一个连接器来访问 SUT。这个连接器最好不会影响到 SUT 的功能。

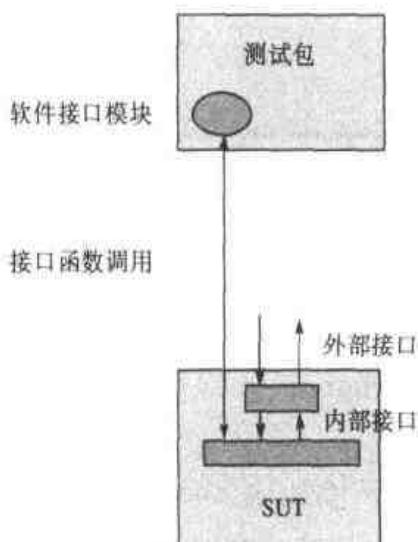


图 C.8 绕过外部接口，建立测试包与被测系统之间的通信

## 附录 D 进化算法的伪代码

---

### D.1 主过程

```
Initialise P(1)
t = 1
while not exit criterion
 evaluate P(t)
 selection
 recombination
 mutation
 survive
 t = t + 1
end
```

---

### D.2 选择

```
Calculate for each individual the accumulated fitness;
for Pez loop
 rand = random number;
 for i loop
 if rand <= fi, accu then
 select individual;
 exit loop;
 end if;
 end loop;
end loop;
```

### D.3 重组

```
For all parents loop
 rand = random number;
 if rand <= Pc
 rand = random number;
 parents interchange at random position and create
 offspring;
 end if;
end loop;
```

$P_c$  是交叉概率，对单交叉来说它等于 1 或者测试用例的元素数目。

---

### D.4 突变

```
Pm = 1/(number of elements in test case)
For offspring size loop
 rand = random number;
 position = rand div Pm + 1;
 mutate individual at position;
end loop;
```

---

### D.5 插入

```
Calculate the normalized accumulated fitness for offspring
population;
Calculate the reciprocal normalised accumulated fitness for parent
population;
for all parents loop
 rand = random number;
 if rand < Ps then
 rand = random number
 for all offspring loop
 if rand <= fi, accu offspring then
 select individual;
```

```
 exit loop;
 end if;
end loop;
rand = random number
loop for parent population
 if rand <= $f_i, record$ then
 select individual;
 replace with selected
 offspring individual;
 exit loop;
 end if;
end loop;
end if;
end loop;
```

## **附录 E 测试计划例子**

这是一个已填好的测试计划例子，它可以被用做其他测试计划的模板。虽然这是一个纯粹虚构的测试计划，但它包含了所有必要的信息。它有助于向读者提供更为具体的要点，必须将什么样的信息记录在测试集合中。细节层次要依赖于具体情况具体分析，而且必须由测试经理来决定。在这个测试计划例子中，所有的人员名称和产品名称都是作者想像出来的。如果与现实生活中其他人员或产品有任何雷同，纯属偶然。

该测试计划的内容包括：

1. 分配任务；
2. 测试基础；
3. 测试策略；
4. 制定计划；
5. 隐患、风险和措施；
6. 基础设施；
7. 测试组织；
8. 测试交付物；
9. 配置管理。

---

### **E.1 分配任务**

#### **E.1.1 委托人**

分配任务的委托人为 J. Davis，他是 Solidly 嵌入式股份有限公司的产品经理。

#### **E.1.2 承包人**

执行分配任务是 T. Testware 的职责，他是 Solidly 嵌入式股份有限公司的测试经理。

### E.1.3 范围

验收测试的范围是：

- “James” 电子管家版本 1.0；
- “Mother” 基站版本 1.0。

### E.1.4 目标

目标是：

- 确定系统是否满足需求；
- 报告观测行为与预期行为之间的差异；
- 交付测试件，以便可以复用于“James”以后的版本中。

### E.1.5 前提条件（外部）

前提条件是：

- 在 8 月 14 日，系统文档应当交付；
- 在 9 月 10 日，开发团队交付系统以进行验收测试；
- 在 9 月 26 日，测试应当完成。

### E.1.6 前提条件（内部）

测试团队为完成所分配的任务，下面的前提条件是必须的：

- 当测试对象已经成功地通过了项目检查时，就可以执行测试了；
- （部分）测试基础或测试对象的延迟提交，必须立即报告给测试经理；
- 在测试执行时，所有必要的工具和测试基础设施都必须准备好；
- 在测试执行过程中，开发部门必须随时支持解决任何阻碍性的问题，否则会严重影响测试执行的进度；
- 测试基础的改变必须立即和测试经理沟通。

---

## E.2 测试基础

测试基础包含：

产品规范

1. 一般功能设计 1.1。

2. 详细功能设计 1.0，应当在 8 月 14 日前交付。
3. 用户手册 “James” 1.0，应当在 8 月 14 日前交付。
4. 用户手册 “Mother” 1.0，应当在 8 月 14 日前交付。

#### 标准

5. 测试产品内部标准 2.0.
6. 手册《测试嵌入式软件》

#### 用户手册

7. 测试环境用户手册，必须在 8 月 14 前交付。
8. 测试工具用户手册，必须在 8 月 14 日前交付。

#### 项目计划

9. 项目计划 “James”，版本 1.0。
10. 项目计划 “Mother”，版本 1.0。

#### 制定计划

11. 计划 “开发团队 ‘James’”，版本 1.0。
12. 计划 “开发团队 ‘Mother’”，版本 1.0。

---

### E.3 测试策略

可接受性测试的测试策略前提条件是：假设模块测试和集成测试已经由开发团队完成。

测试策略是和委托人、项目领导、测试经理商定的结果。

#### E.3.1 质量特性

表 E.1 说明的是哪些质量特性必须测试，以及它们的相对重要性。

表 E.1 质量特性的相对重要性

| 质量特性 | 相对重要性 (%) |
|------|-----------|
| 功能   | 50        |
| 可用性  | 30        |
| 可靠性  | 20        |
| 总重要性 | 100       |

### E.3.2 策略矩阵

基于功能设计，系统被分为 5 个部分：

- ※ A 部分：命令输入/处理包含下列过程：
  - 作为命令输入机制的语音识别
  - 处理，对命令的处理
- ※ B 部分：动作控制，“James”的所有动作以及这些动作之间的协调，以及和环境之间的关系；
- ※ C 部分：预期行为，学会理解主人（客户）的行为，以及对主人行为的预期；
- ※ D 部分：基本情况，包括服务/监视软件和服务提供者的 Internet 链接；
- ※ 整个系统：“James”和“Mother”的组合。

表 E.2 所示为每个子系统-质量特性组合的测试重要性。

表 E.2 可接受性测试的策略矩阵

| 相对重要性 | A 部分 | B 部分 | C 部分 | D 部分 | 整个系统 |
|-------|------|------|------|------|------|
|       | 100% | 40   | 20   | 10   | 15   |
| 功能    | 50   | ++   | +    | +    | +    |
| 可用性   | 30   | ++   |      |      | +    |
| 可靠性   | 20   | +    | ++   |      | +    |

### E.3.3 每个系统部分的测试设计技术

表 E.3 所示为系统不同部分所用的测试设计技术。测试设计技术的选择以系统部分的特征、所选策略和待测系统特性为依据。

表 E.3 分配给子系统的测试技术

| 分配的测试设计技术 | A 部分 | B 部分 | C 部分 | D 部分 | 整个系统 |
|-----------|------|------|------|------|------|
| STT       | +    | +    |      | +    | +    |
| ECT       | +    |      |      |      |      |
| CTM       | +    |      |      |      |      |
| 统计使用测试    |      | +    | +    | +    | +    |
| 稀有事件      | +    | +    |      |      |      |

STT = 状态转换测试

ECT = 基本比较测试

CTM = 分类树方法

### E.3.4 估计的工作量

表 E.4 所示为对项目过程中不同阶段所需时间的估计。估计的依据是开始和结束的日期以及所选策略。

表 E.4 测试过程的估计工作量

| 任务     | 时间(小时) |
|--------|--------|
| 测试计划   | 32     |
| 计划控制阶段 | 120    |
| 测试管理   | 48     |
| 测试配置管理 | 28     |
| 方法支持   | 44     |
| 准备阶段   | 32     |
| 细化阶段   | 186    |
| 执行阶段   | 250    |
| 完成阶段   | 16     |
| 总计     | 636    |

### E.4 制定计划

下面的表 E.5 用来为测试项目制定计划，确定了任务的开始和结束日期，其中包括任务的超时预算。

表 E.5 为“James”和“Mother”制定计划。TM = 测试管理, MS = 方法支持,  
 TS = 技术支持, DE = 领域专家, TCM = 测试配置经理, TST = 测试人员

| Activity | Start | End   | TM | MS | TS | DE | TCM | TST |
|----------|-------|-------|----|----|----|----|-----|-----|
| 测试计划     | 07 08 | 11 8  | 32 |    |    |    |     |     |
| 计划和控制    | 14 08 | 04 10 |    |    |    |    |     |     |
| 测试管理     |       |       | 48 |    |    |    |     |     |
| 测试配置管理   |       |       |    |    | 28 |    |     |     |
| 方法支持     |       |       |    | 44 |    |    |     |     |
| 准备       | 14 08 | 15 08 |    |    |    |    | 32  |     |
| 规范       | 16 08 | 08 09 |    |    |    |    | 186 |     |
| 执行       | 10 09 | 26 09 |    |    | 30 | 50 |     | 170 |
| 完成       | 27 09 | 28 09 |    |    |    |    | 16  |     |

## E.5 隐患、风险和措施

项目可能存在着以下隐患：

- ※ 测试的交付日期可能被延迟。如果固定下了结束日期，那就意味着必须使用测试策略来计算跳过哪些测试。
- ※ 当产品被交付进行测试时，开发团队被解散。这意味着缺陷的解决将更为困难，而且测试过程中不可能得到开发人员的立即支持。在测试执行过程中，至少要有一个有经验的开发人员的支持
- ※ 只有在已计划好的测试执行时间内，才有领域专家。测试对象的延迟交付将意味着不能再用到领域专家，而必须雇佣其他的人力。这样不仅得不到领域专家的支持，而且将直接导致某些测试可能需要花费更多的时间，而且更容易出错。
- ※ 在这个时候，设计人员仍然在进行功能设计，编写用户指南。它们必须按时交付，如果在测试执行开始时，还没有指定要测试的部分，那么测试就很难重现。如果发生了这种事情，需要使用捕获工具来记录测试动作，以便使测试可以重现。

## E.6 基础设施

### E.6.1 测试环境

测试团队由五个人组成。在测试项目期间，必须有一台标准配置的 PC 机，安装有 Solidly 嵌入式有限公司的标准软件包（一些特殊软件，见 E.6.2）。

产品管理目录至少要有 20 GB 的磁盘空间。

### E.6.2 测试工具

需要有以下测试工具：

- ※ 缺陷管理工具“DefectTracker”；
- ※ 变更管理工具“ChangeMaster”；
- ※ 计划软件；
- ※ 要有更多的硬件来跟踪“James”的内部行为。

### E.6.3 环境

使用标准测试环境。在测试项目期间，测试人员使用各自的工作办公室。

---

## E.7 测试组织

### E.7.1 测试角色

在本测试项目中，需要有以下测试角色：

- 测试工程师；
- 测试经理；
- 方法支持；
- 技术支持；
- 领域专家；
- 测试配置管理员。

#### 测试工程师

- 审查测试基础（规范）；
- 细化逻辑测试用例、实物测试用例及开始情况；
- 执行测试用例（动态测试）；
- 登记缺陷；
- 存档测试件。

#### 测试经理

测试经理负责在一定的时间和预算范围之内，依据质量要求来对测试过程进行计划、管理与执行。测试经理要记录测试过程的进度和测试对象的质量。

- 制定测试计划，使之得到批准并维护测试计划；
- 依据时间进度和预算来执行测试计划；
- 报告测试过程的进度和测试对象的质量。

#### 方法支持

- 确定测试技术；
- 制定测试规章制度；
- 给出建议并协助实现所有可能的测试技术。

#### 技术支持

- 建立测试基础设施；

- 监管测试基础设施；
- 物理配置管理；
- 解决技术问题；
- 确保测试的重现；
- 给出建议并提供协助。

#### 领域专家

- 当必要时，对测试对象的功能提供建议及帮助。

#### 测试配置管理员

- 进度控制；
- 测试文档控制；
- 记录缺陷并进行统计；
- 更改测试基础和测试对象的控制；
- 更改测试件的控制。

### E.7.2 组织结构

图 E.1 所示为测试项目的组织。

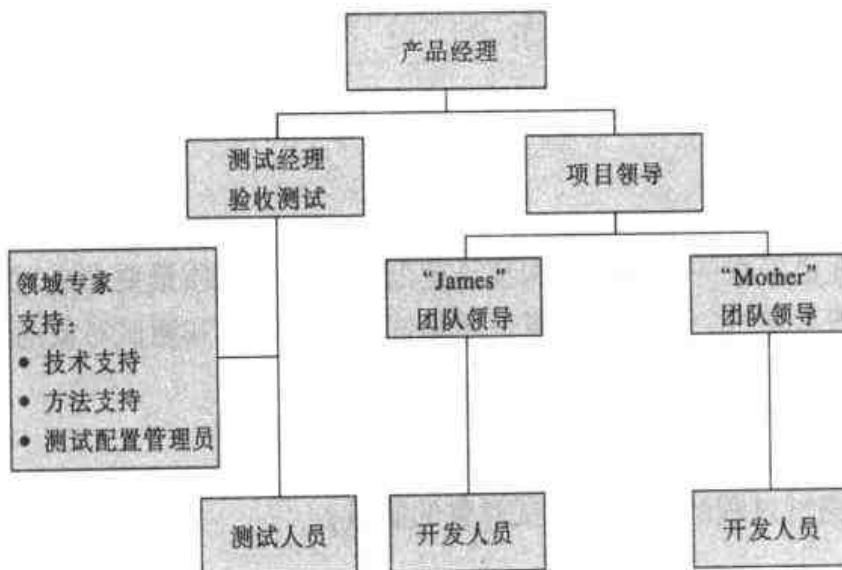


图 E.1 组织结构

测试经理直接向产品经理报告。

在测试项目期间，计划每周召开一次进度会议。由产品经理、项目领导和测试经理参加，必要时召集“James”和“Mother”的团队领导。其他会议在必要时召开。

### E.7.3 测试成员

测试成员情况见表 E.6。

表 E.6 测试成员职务及全职时间

| 职务   | 名字            | 全职时间           |
|------|---------------|----------------|
| 测试经理 | T. Testware   | 0.20           |
| 测试人员 | P. Testcase   | 1.00           |
| 测试人员 | F. Testscript | 1.00           |
| 支持   | G. Allround   | 1.00           |
| 领域专家 | A. Expert     | 0.60 (在测试执行期间) |

## E.8 测试交付物

### E.8.1 项目文档

在测试项目期间，将生成以下文档：

- 测试计划：该文档的当前版本，以前及以后的版本；
- 缺陷报告：报告发现的所有缺陷；
- 每周报告：进度报告由测试经理编写，在每周进度会议的前一天将报告发送给会议的所有成员；
- 发布建议：这标志着测试执行阶段的正式完工；
- 审查报告：该报告描述在测试项目期间，对测试过程的评估。目标是改进测试过程，以用于下一个版本。

### E.8.2 测试件

- 测试脚本：对如何进行测试的描述。它包含有与测试用例相关的、指示执行顺序的测试动作和检查；
- 测试方案：一个协调执行几个测试脚本的“微观测试计划”；
- 初始化数据集：开始测试所需的文件和数据集。

### E.8.3 存储

这里给出的是在 Solidly 嵌入式有限公司中央服务器上实现的目录结构。该目录被存储在产品管理目录下（如表 E.7 所示）：\PROD\_MANAG。

表 E.7 目录结构

| ACC_TEST_JAMES_MOTHER |             |
|-----------------------|-------------|
| PROCDOC               | 项目文档        |
| WORK_TESTWARE         | 测试件工作目录     |
| FINAL_TESTWARE        | 测试件存档目录     |
| DEFECTS               | 缺陷存储数据库     |
| OTHER                 | 其他有用的或生成的文档 |

## E.9 配置管理

### E.9.1 测试过程控制

在测试项目期间，监控和报告测试进度及所花的预算和时间，每周进行一次，并在每周进度会议上报告这些结果。

#### 缺陷管理

使用缺陷管理工具“DefectTraker”，遵循标准缺陷管理规程（SolEmb 76.1 “Defect procedure”）。

#### 度量

测试经理保持跟踪以下的度量：

- 在一段时间内每个严重级别下未解决的缺陷数目；
- 在一段时间内每个严重级别下已解决的缺陷数目；
- 出现的缺陷总数；
- 每个缺陷的再测试次数；
- 总的再测试次数。

### E.9.2 配置管理项

一旦测试计划完成第一个版本，就要受到配置管理的支配。其他测试件是在完成阶段之后受到配置管理的支配。

测试基础设施的变更要受到 Solidly 嵌入式有限公司技术支持部配置管理的支配。

# 词汇表

由 BCS SIGIST ( BS7925-1 ) 制定的“软件测试词汇表”标准是下面词汇表的基础。

|                                      |                                                                 |
|--------------------------------------|-----------------------------------------------------------------|
| 验收测试 ( Acceptance testing )          | 使用户、客户或被授权的实体决定是否接受系统或构件而进行的正式测试。                               |
| 动作器 ( Actor )                        | 一个由嵌入式系统输出信号所控制、专用于对环境进行操纵的模块。                                  |
| 实际结果 ( Actual result )               | 处理测试输入而观察到的系统行为。                                                |
| 行为 ( Behavior )                      | 由输入值、前提条件、系统功能所需的响应组成的组合。一个功能的完整规范一般由一个或多个行为组成。                 |
| 黑盒测试 ( Black-box testing )           | 测试用例的选择是基于对构件规范的分析，而不涉及构件的内部工作原理。                               |
| 边界值 ( Boundary value )               | 在等价类之间边界处的输入值或输出值，或是边界任意一侧的一个增量。                                |
| 边界值分析<br>( Boundary value analysis ) | 用于构件的一种测试设计技术，所设计的测试用例包含有代表性的边界值。                               |
| Bug                                  | 见故障。                                                            |
| 认证 ( Certification )                 | 确认系统或构件遵循其指定的需求，可以接收以用于实际使用的过程。                                 |
| 审查清单 ( Checklist )                   | 审查清单是一个答案只能为是或否的问题清单。                                           |
| 代码覆盖范围 ( Code coverage )             | 一种用于确定软件的哪些部分已经执行了测试用例 ( 被覆盖 )，哪些部分还没有执行测试用例，因而可能需要给以更多关注的分析方法。 |
| 构件 ( Component )                     | 具有单独规范的软件或硬件的最小项。                                               |
| 覆盖 ( Coverage )                      | 测试集已测试的与能够被测试的比例关系，用百分比来表示。                                     |

|                                       |                                           |
|---------------------------------------|-------------------------------------------|
| 调试 ( Debugging )                      | 查找缺陷以及消除缺陷的过程。                            |
| 缺陷 ( Defect )                         | 见故障。                                      |
| 驱动程序 ( Driver )                       | 开发或测试一个构件的通用或专门实现的软件模块。                   |
| 动态测试 ( Dynamic testing )              | 依据系统或构件在测试执行期间的行为来对其进行评估的过程。              |
| 嵌入式软件 ( Embedded software )           | 在嵌入式系统内的软件，专门用于对系统特定硬件进行控制。               |
| 嵌入式系统 ( Embedded system )             | 通过动作器和传感器，与现实世界交互的系统。                     |
| 仿真器 ( 程序, Emulator )                  | 能够像一个给定系统一样，只要输入相同就能得出相同输出结果的设备、计算机程序或系统。 |
| 入口检查 ( Entry check )                  | 对测试对象是否被完整交付，以及是否具备足够的质量来开始下一阶段而进行的检查。    |
| 等价类 ( Equivalence class )             | 构件输入域或输出域的一部分。从构件规范来看，对其中的每一个值，构件行为都是相同的。 |
| 错误 ( Error )                          | 由于人为动作而产生的一个不正确结果                         |
| 评估 ( Evaluation )                     | 对系统开发周期中各种中间产品和/或过程的评审与检查。                |
| 预期结果 ( Expected result )              | 在指定条件下，根据对象规范预测的对象行为。                     |
| 失效 ( Failure )                        | 与系统预期的交付或服务的偏离                            |
| 故障 ( Fault )                          | 软件中错误的表现形式。发生故障可能会导致失效。                   |
| 功能需求 ( Functional requirement )       | 系统所需的功能行为。                                |
| 功能规范 ( Functional specification )     | 对与产品预期能力相关的特性详细描述的文档。                     |
| 高层次测试 ( High-level testing )          | 对完整产品进行测试的过程。                             |
| 硬件循环<br>( HiL: hardware-in-the-loop ) | 在一个模拟环境中，使用实际硬件来进行测试的测试层次。                |
| 初始情况 ( Initial situation )            | 测试用例开始时一定处于该状态，通常用相关的输入数据和内部变量的取值描述。      |

|                                   |                                                      |
|-----------------------------------|------------------------------------------------------|
| 初始状态 (Initial state)              | 系统接受第一个事件时所处的状态。                                     |
| 输入 (Input)                        | 由构件处理的信号或变量 (无论是存储在构件内部还是外部)。                        |
| 输入域 (Input domain)                | 所有可能输入的集合。                                           |
| 输入值 (Input value)                 | 一个输入实例。                                              |
| 检查 (Inspection)                   | 一组对书面材料评审的质量改进过程。                                    |
| 集成 (Integration)                  | 将构件组合为更大组合体的过程。                                      |
| 集成策略 (Integration strategy)       | 如何将不同构件集成为一个完整系统的决策。                                 |
| 集成测试 (Integration testing)        | 为暴露集成构件之间接口与交互中的故障而执行的测试。                            |
| 迭代开发 (Iterative development)      | 通过多次由分析、设计、实现与测试阶段组成的开发周期，不断对系统扩展与细化的迭代式生命周期。        |
| 已知错误 (Known errors)               | 已经发现但还没有被解决的缺陷。(注意：这里使用的单词“错误”是不恰当的，更为准确的用法是“已知缺陷”。) |
| 生命周期 (Lifecycle)                  | 生命周期由所划分阶段、描述每一阶段需要执行的活动以及按照何种顺序来执行这样一个过程等部分组成。      |
| LITO                              | 结构化测试的四个要素：生命周期、基础设施、技术与组织。                          |
| LITO 矩阵                           | 系统特性与特定方法之间的关系 (按照四个要素划分)。                           |
| 逻辑测试用例 (Logical test case)        | 从头到尾对测试对象 (例如：一个功能) 进行测试的一系列情况。                      |
| 低层次测试 (Low-level tests)           | 每次测试一个或一组构件的测试过程。                                    |
| 主测试计划 (Master test plan)          | 协调开发项目中不同测试层次的总测试计划。                                 |
| 模型循环<br>(MiL: model-in-the-loop)  | 一种在模拟环境中，对系统的模拟模型动态测试的测试层次。                          |
| 混合信号 (Mixed signals)              | 由数字信号与连续信号混合而成的信号。                                   |
| 基于模型的开发 (Model-based development) | 系统首先被描述为模型，然后依据模型自动生成代码的一种开发方法。                      |

|                                        |                                                                         |
|----------------------------------------|-------------------------------------------------------------------------|
| 输出 (Output)                            | 由构件生成的信号或变量（无论是存储在构件内部还是外部）。                                            |
| 质量 (Quality, ISO 8402)                 | 产品或服务能够满足规定或潜在需求的所有特性与特征 (ISO 8402, 1994)。                              |
| 质量保证<br>( Quality assurance, ISO 8402) | 为提供足够的置信度来表明实体能够满足质量要求，而在质量系统内实现的全部有计划有组织的活动 (ISO 8402, 1994)。          |
| 质量特性 (Quality characteristic)          | 系统本身的特性。                                                                |
| 实物测试用例 (Physical test case)            | 对一个测试情况或逻辑测试用例的详细描述，包含初始情况、要执行的动作以及给出预期结果的具体值。详细程度要使得在后期阶段能够尽可能有效地执行测试。 |
| Plant                                  |                                                                         |
| 前提条件 (Precondition)                    | 与嵌入式系统交互的环境。                                                            |
| 快速原型法 (Rapid prototyping)              | 针对一个特定输入值来执行构件之前，必须满足的环境和状态条件。                                          |
| 实时系统 (Real-time system)                | 对一个模拟的、与实际环境相连的嵌入式系统进行测试的测试层次。                                          |
| 回归测试 (Regression testing)              | 系统行为的正确性依赖于输出生成的确切时刻。                                                   |
| 结果 (Result)                            | 对以前已测试的测试对象在修改后重新测试，确保变更发生没有引入故障或没有暴露出故障。                               |
| 风险 (Risk)                              | 见实际结果或预期结果。                                                             |
| 风险报告 (Risk reporting)                  | 风险 = 失效几率 × 损害                                                          |
| 传感器 (Sensor)                           | 描述系统在多大程度上能够满足指定的质量要求，以及将一个特定版本引入生产将带来的相关风险，包括所有可用的备选项。                 |
| 模拟 (Simulation)                        | 嵌入式系统的一个模块，专用于从环境接收激励，并将它们转换为系统的输入信号。                                   |
| 模拟器 (Simulator)                        | 由另一个系统来表示一个物理系统或抽象系统所选择的行为特征。                                           |
|                                        | 在软件确认过程中所使用的一个设备、计算机程序或系统。当提供一组受控输入时，其行为或运行类似于一个给定系统。                   |

|                                       |                                                                                             |
|---------------------------------------|---------------------------------------------------------------------------------------------|
| 软件循环<br>( SiL: software-in-the-loop ) | 在一个模拟环境或试验硬件环境中，使用实际软件来进行测试的测试层次。                                                           |
| 开始情况 ( Start situation )              | 见初始情况。                                                                                      |
| 状态机 ( State machine )                 | 由当前输入和历史输入决定输出的系统。                                                                          |
| 状态转换 ( State transition )             | 系统从一个状态改变为另一个状态。                                                                            |
| 静态测试 ( Static testing )               | 不执行测试对象而对系统或构件进行评估的过程。                                                                      |
| 占位 ( Stub )                           | 开发或测试一个构件需要调用或依赖的通用或专门实现的软件模块。                                                              |
| 系统测试 ( System testing )               | 测试一个集成系统是否满足其指定需求的过程。                                                                       |
| TEmb                                  | 一种测试嵌入式软件的方法。                                                                               |
| 测试动作 ( Test action )                  | 测试用例定义在测试对象上必须执行的动作。测试动作是测试用例的一部分。                                                          |
| 测试自动化 ( Test automation )             | 使用软件来控制测试执行、比较实际结果与预期结果、建立测试的前提条件，以及其他 的测试控制与测试报告功能。                                        |
| 测试基础 ( Test basis )                   | 所有被用于设计测试用例的系统文档。                                                                           |
| 测试床 ( Test bed )                      | 向测试对象提供激励信号并记录测试对象输出的软/硬件。                                                                  |
| 测试用例 ( Test case )                    | 为一个特定目标而设计的一组输入、前提条件和预期结果，例如为了执行一个特定程序路径，或为了检验是否与一个特定需求相符合。                                 |
| 测试基础设施 ( Test infrastructure )        | 由硬件、系统软件、测试工具、规程等组成的执行测试的环境。                                                                |
| 测试层次 ( Test level )                   | 一组被组织和管理到一起的测试活动。可以划分为高层次测试和低层次测试。                                                          |
| 测试深度 ( Test depth level )             | 表示被测连续决策点之间的依赖程度。当测试深度为 $n$ 时，通过将 $n$ 个动作的所有可能组合放入测试路径，来检验 1 个决策点之前和 $n-1$ 个决策点之后动作的所有依赖关系。 |
| 测试对象 ( Test object )                  | 被测试的系统（或系统部分）。                                                                              |
| 测试组织 ( Test organization )            | 测试角色、辅助工具、规程、活动，以及它们之间关系的总和。                                                                |

|                                     |                                                               |
|-------------------------------------|---------------------------------------------------------------|
| 测试计划 ( Test plan )                  | 包含有管理一个测试项目所需的全部信息的项目计划。                                      |
| 测试过程 ( Test process )               | 用于执行测试的工具、技术和工作方法的集合。                                         |
| 测试角色 ( Test role )                  | 测试角色是一个职责，可以将一项或多项任务的执行分配给一个或多个人员。                            |
| 测试方案 ( Test scenario )              | 一个协调执行几个测试脚本的“微观测试计划”。                                        |
| 测试脚本 ( Test script )                | 对如何进行测试的描述。它包含有与测试用例相关的、指示执行顺序的测试动作和检查。                       |
| 测试集合 ( Test set )                   | 测试用例的集合。                                                      |
| 测试设计技术<br>( Test design technique ) | 从测试基础导出测试用例的标准方法。                                             |
| 测试策略 ( Test strategy )              | 描述系统部分与质量特性的相对重要性，以此来决定所期望的覆盖率、将采用的技术及其分配的资源。                 |
| 测试团队 ( Test team )                  | 负责执行测试计划中所有活动的一组人员。                                           |
| 测试技术 ( Test technique )             | 对如何执行一个特定测试活动的标准描述。                                           |
| 测试工具 ( Test tool )                  | 支持一项或多项测试活动的自动化辅助工具，所支持测试活动有计划与控制、细化、建立初始化文件和数据、测试执行与测试分析等活动。 |
| 测试类型 ( Test type )                  | 用一些相关的质量特性来检查系统的一组测试活动。                                       |
| 测试单元 ( Test unit )                  | 组合起来被测试的一组过程、事务与/或功能。                                         |
| 可测性审查 ( Testability review )        | 对测试基础的可测性进行的详细评估。                                             |
| 测试 ( Testing )                      | 为建立系统的质量等级而进行的计划、准备、执行和分析过程。                                  |
| 测试件 ( Testware )                    | 一个测试项目最终生成的所有产品。                                              |
| 单元测试 ( Unit test )                  | 对单个软件构件进行的测试。                                                 |
| 白盒测试 ( White-box testing )          | 使用对象的内部结构知识，从对象的内部属性导出测试用例的测试设计技术。                            |

## 参考文献

- Beck, K. (2000) *Extreme Programming Explained*, Addison-Wesley.
- Beizer, B. (1990) *Software Testing Techniques*, International Thomson Computer Press.
- Beizer, B. (1995) *Black-box Testing. Techniques for Functional Testing of Software and Systems*, John Wiley and Sons.
- Biennmller, T., Bohn, J., Brinkmann, H., Brockmeyer, U., Damm, W., Hungar, H. and Jansen, P. (1999) 'Verification of automotive control units.' In: Olderog, E.-R. and Steffen, B. (Eds.): *Correct System Design*. LNCS Vol. 1710, pp. 319–341, Springer-Verlag.
- Binder, R.V. (2000) *Testing Object-oriented Systems: Models, Patterns, and Tools*, Addison-Wesley.
- Boehm, B.W. (1981) *Software Engineering Economics*, Prentice Hall.
- BS7925-1 – *Glossary of terms used in software testing*, British Computer Society – Specialist Interest Group in Software Testing.
- BS7925-2 – *Standard for software component testing*, British Computer Society Specialist Interest Group in Software Testing.
- Cretu, A. (1997) *Use Case Software Development and Testing Using Operational Profiles*, Concordia University.
- Conrad, M. (2001) 'Beschreibung von Testszenarien für Steuergeräte-Software – Vergleichskriterien und deren Anwendung' (Description of test scenarios for ECU software – comparison criteria and their application, in German). In: Proc. of 10th Int. VDI-Congress: *Electronic Systems for Vehicles*, Baden-Baden.
- Conrad, M., Dörr, H., Fey, I. and Yap, A. (1999) 'Model-based generation and structured representation of test scenarios.' In: Proc. of *Workshop on Software-Embedded Systems Testing*, Gaithersburg, Maryland.
- Douglass, B.P. (1999) *Doing Hard Time – developing real-time systems with UML, objects, frameworks, and patterns*, Addison-Wesley.
- dSPACE (1999) *ControlDesk Test Automation Guide For ControlDesk – version 1.2*, dSPACE GmbH.
- Erpenbach, E., Stappert, F. and Stroop, J. (1999) 'Compilation and timing of statechart models for embedded systems', *Cases99 Conference*.
- Fagan, M.E. (1986) 'Advances in software inspections,' *IEEE Transactions on Software Engineering*, SE-12.
- Ghezzi, C., Jazayeri, M. and Mandrioli, D. (1991) *Fundamentals of Software Engineering*, Prentice Hall.
- Gilb, T. and Graham, D. (1993) *Software Inspection*, Addison-Wesley.
- Graham, D., Herzlich, P. and Morelli, C. (1996) *Computer Aided Software Testing: the CAST Report*, Cambridge Market Intelligence Ltd.

- Grochtmann, M. (1994) 'Test case design using classification-trees,' *Star '94*, Washington D.C.
- Grochtmann, M. and Grimm, K. (1993) 'Classification-trees for partition testing.' *Software Testing, Verification, and Reliability*, 3 (2), pp. 63-82.
- Hsu, W., Shinoglu, M. and Spafford, E.H. (1992) *An Experimental Approach to Statistical Mutation-based Testing*, SERC TR-63-P, [www.cerias.purdue.edu/homes/spaf/wwwpub/node10.html](http://www.cerias.purdue.edu/homes/spaf/wwwpub/node10.html)
- Information Processing Ltd. (IPL) (1996) 'Testing state machines with AdaTest and Cantate', published on the internet, [www.iplbath.com/products/library/p1001.shtml](http://www.iplbath.com/products/library/p1001.shtml)
- ISO/IEC 9646 (1996) – Part 3: *Tree and Tabular Combined Notation*, 2nd edition.
- ITU Recommendation Z.120 (1996) *Message Sequence Charts*, International Telecommunication Union – Telecommunication Standardization Sector (ITU-T).
- Kim, S-W., Clark J.A. and McDermid, J.A. (1999) *Assessing Test Set Adequacy for Object-oriented Programs Using Class Mutation*, Department of Computer Science, The University of York.
- Kruchten, P. (2000) *The Rational Unified Process: an introduction*, Addison-Wesley.
- Lutz, R.R. and Woodhouse, R.M. (1999) 'Bi-directional analysis for certification of safety-critical software,' *1st International Software Assurance Certification Conference proceedings*, Feb. 28 – Mar. 2, Washington D.C.
- Mitchel, D. (1996) 'Test bench generation from timing diagrams'. In: Pellerin, D. and Taylor, D (Eds.): *VHDL Made Easy*, Appendix D, Prentice Hall.
- MOD (1996a) *Safety Management Requirements for Defence Systems. Part 1: Requirements*, Def Stan 00-56, Ministry of Defence.
- MOD (1996b) *Safety Management Requirements for Defence Systems. Part 2: Guidelines*. Def Stan 00-56, Ministry of Defence.
- Musa, J.D. (1998) *Software Reliability Engineering: more reliable software, faster development and testing*, McGraw-Hill.
- Myers, G.J. (1979) *The Art of Software Testing*, John Wiley and Sons.
- OMG (1997) *UML Notation Guide Version 1.1*, OMG document ad/97-08-05, Object Management Group.
- OMG (1999) *OMG Unified Modeling Language Specification Version 1.3*, published on the internet [www.omg.org/uml](http://www.omg.org/uml)
- Pohlheim, H. (2001) *Genetic and Evolutionary Algorithm Toolbox*. Chapters 1-7, <http://www.geatbx.com/docu/algindex.html>
- Pol, M., Teunissen, R. and van Veenendaal, E. (2002) *Software Testing: a guide to the TMap® Approach*, Addison-Wesley.
- Reid, S. (2001) 'Standards for software testing', *The Tester* (BCS SIGIST Journal) June, 2-3.
- Reynolds, M.T. (1996) *Test and Evaluation of Complex Systems*, John Wiley and Sons.
- RTCA/DO-178B (1992) *Software Considerations in Airborn Systems and Equipment Certification*, RTCA.
- Sax, E. (2000) *Beitrag zur entwurfsbegleitenden Validierung und Verifikation elektronischer Mixed-Signal-Systeme*. PhD Thesis, FZI-Publikation, Forschungszentrum Informatik an der Universität Karlsruhe.

- Schaefer, H. (1996) 'Surviving under time and budget pressure,' *EuroSTAR 1996 Conference Proceedings*.
- Simmes, D. (1997) *Entwicklungsbegleitender Systemtest für elektronische Fahrzeugsteuergeräte* (in German), Herbert Utz Verlag Wissenschaft.
- Spillner, A. (2000) 'From V-model to W-model – establishing the whole test process', *Conquest 2000 Conference Proceedings*, ASQE.
- Shamer, H.H. (1995) *The Automatic Generation of Software Test Data using Genetic Algorithms*, Thesis at the University of Glamorgan.
- Untch, R.H. (1995) *Schema-based Mutation Analysis: a new test data adequacy assessment method*, Technical Report 95-115, Department of Computer Science, Clemson University.
- Wegener, J. and Grochtmann, M. (1998) 'Verifying timing constraints of real-time systems by means of evolutionary testing,' *Real-Time Systems*, Vol. 15, No. 3, pp. 275–298.
- Wegener, J. and Mueller, F. (2001) 'A Comparison of static analysis and evolutionary testing for the verification of timing constraints,' *Real-Time Systems*, Vol. 21, No. 3, pp. 241–268.
- Woit, D.M. (1994) *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules*, Queen's University, Kingston, Ontario.

## 推荐读物

- Burton, S. (1999) 'Towards automated unit testing of statechart implementations.' University of York.
- Dornseiff, M., Stahl, M., Sieger, M. and Sax, E. (2001) 'Durchgängige Testmethoden für komplexe Steuerungssysteme – Optimierung der Prüftiefe durch effiziente Testprozesse' (Systematically consistent test methods for complex control systems: enhancement of testing depth through test automation Control Systems for the powertrain of motor vehicles) in German. In: Proc. of 10th Int. VDI-Congress: *Electronic Systems for Vehicles*, Baden-Baden.
- Grochtmann, M., Wegener, J. and Grimm, K. (1995) 'Test case design using classification-trees and the classification-tree editor CTE'. In: Proc. of 8th International Software Quality Week, San Francisco.
- Isakswi, U., Jonathan, P.B. and Nissanke, N. (1996) 'System and software safety in critical systems,' The University of Reading.
- Lehmann, E. and Wegener, J. (2000) 'Test case design by means of the CTE XL.' In: Proc. of EuroStar 2000, Copenhagen.
- Liggesmeyer, P. (1990) *Modultest und Modulverifikation: state of the art*. BI-Wiss.-Verlag Mannheim.
- Low, G. and Leenanuraksa, V. (1988) 'Software quality and CASE tools.' In: IEEE Proc.: *Software Technology and Engineering Practice*.
- Ostrand, T. and Balcer, M. (1988) 'The category-partition method for specifying and generating functional tests,' *Communications of the ACM*, 31(6), pp. 676–686.
- Sax, E. and Müller-Glaser, K.-D. (2002) 'Seamless testing of embedded control

- systems.' *3rd IEEE Latin-American Test Workshop*, Montevideo.
- Shankar, N. (1993) 'Verification of real-time systems using PVS,' *Lecture Notes in Computer Science*, Vol. 697, pp. 280-291.
- Spafford, E.H. (1990) 'Extending mutation testing to find environmental bugs', *Software Practice and Experience*, Vol. 20, No. 2, pp. 181-189.
- Tracey, N., Clark, J., Mander, K. and McDermid, J. (1998) 'An automated framework for structural test data generation. In Proc. 13th IEEE Conference in Automated Software Engineering, Hawaii.
- Tsai, B-Y., Stobart, S. and Parrington, N. (1997) 'A method for automatic class testing object-oriented programs using a state-based testing method'. In: Proc. of EuroStar 1997, Edinburgh.
- Wegener, J., Baresel, A. and Sthamer, H. (2001) 'Evolutionary test environment for automatic structural testing'. *Information and Software Technology*, Vol. 43 pp. 841-854.
- Wegener, J. and Pitschinetz, R. (1995) 'Tessy - an overall unit testing tool.' Proc. of 8th International Software Quality Week, San Francisco.

[General Information]

书名=嵌入式软件测试

作者=

页数=312

SS号=0

出版日期=



Powered by XiaoGuo's publishing Studio

QQ:8204136

Website: [www.mcuzone.com](http://www.mcuzone.com)

2005