上海大学

硕士学位论文

变异测试中测试数据生成及等价变异体的检测

姓名：朱经纷

申请学位级别：硕士

专业：计算机系统结构

指导教师：徐拾义

20090101

摘　要

当前，随着普适计算时代的到来，从小到儿童玩具大到国家安全，计算机系统已经渗透到社会生活的各个角落。人们的日常生活也越来越依赖于计算机系统，如家庭电脑，娱乐设施，交通运输，通信网络，工业控制，金融保险服务，医疗，艺术创作和文化活动，商业活动，农业管理，政府管理等等。然而，我们却也似乎很容易忽视这样一种事实。

在当今的信息化社会时代中，计算机系统的可信性显得尤为重要的。一旦计算机系统发生失效，有可能造成大量无辜生命的丧失，引发一场经济灾难，更严重点甚至爆发一场战争。这就使我们不得不对它们提供服务的可信性提出质疑，如何确保软件的高可靠性是我们面临的一个紧迫任务。

本文首先回顾了可信性的起源，介绍了可信性属性和影响可信性的因素，讨论了提高可信性的主要措施。其次，重点介绍了可信性的主要属性之一，即软件可靠性，对传统软件可靠性模型的不足进行了详细地分析，并提出了一种软件综合可靠性模型。该综合模型不仅可以反映出软件的复杂性，而且也考虑到软件测试的测试有效性。本文采用变异测试原理来评估软件测试的测试有效性。

接着本文对变异测试进行了详细介绍，指出变异测试的主要瓶颈极其解决方法。本文在第四章和第五章中采用基于约束的测试技术来生成测试数据和进行等价变异体的检测，提出一种能够杀死多个同位变异体的方法。最后，对研究工作进行了总结，对未来的研究方向进行了展望。

关键词：变异测试， 变异算子， 等价变异体，测试有效性，软件可靠性

ABSTRACT

Today, with the coming of pervasive computing era, our life increasingly depends on computer systems in innumerable aspects, ranging from children's games to national security. People's daily life cannot exist without computers in home, amusement facilities, transportation, communication networks, industrial activities, financial and insurance services, medical services, art and cultural activities, commercial activities, agricultural management, government administration, and so on. Furthermore, we tend to be unaware of this fact.

Under these circumstances, dependability of a computer is of the most importance. If a failure occurred in computer systems which our advanced information society fully depends on, tremendous human lives could be lost, or an economic disaster could occur. There is even a possibility of a war. Thus, we have to question ourselves about the dependability of the services which our computer systems provide. So, the problem on how to ensure the high reliability of software becomes more and more urgent.

Firstly, this thesis reviews the origin of the dependability of computing systems, presenting the threats to their dependability and their attributes, and discusses the means to achieve their dependability. Secondly, we give the definition of software reliability, which is one of the most important attributes of dependability, analyzing the shortcomings of traditional software reliability models, and propose a new synthetic model for software reliability which incorporates the influence factors of complexity of the software under test and the test effectiveness. We evaluate the test effectiveness the software under test by using mutation testing in this thesis.

Thirdly, we introduce the concept of mutation testing, addressing the cost of mutation testing. In chapter 4 and chapter 5, we employ constraint based testing technique to address the problems of automatic test data generation and equivalent mutants, proposing an new approach to generate test data which can kill multiple same-location mutants. At last, conclusions are given and future researches are directed.

Keywords: mutation testing, mutation operator, equivalent mutants, test effectiveness, software reliability

原 创 性 声 明

本人声明：所呈交的论文是本人在导师指导下进行的研究工作。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已发表或撰写过的研究成果。参与同一工作的其他同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签 名：朱经纶 日 期：2009.03.18

本论文使用授权说明

本人完全了解上海大学有关保留、使用学位论文的规定，即：学校有权保留论文及送交论文复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容。

（保密的论文在解密后应遵守此规定）

签 名：朱经纶 导师签名：[illegible] 日期：2009.03.18

第一章 绪论

1.1 课题来源

本课题来源于国家自然科学基金资助项目《软硬件可测性设计新途径—软硬件交互式测试及可测性设计研究》，项目编号：60473033， 课题主要对变异测试中检测等价变异体的方法进行了研究。

1.2 课题研究的目的和意义

当前，从小到儿童玩具大到国家安全，计算机系统已经渗透到社会生活的各个角落。人们的日常生活也越来越依赖于计算机系统，如家庭电脑，娱乐设施，交通运输，通信网络，工业控制，金融保险服务，医疗，艺术创作和文化活动，商业活动，农业管理，政府管理等等。然而，我们却也似乎很容易忽视这样一种事实。

在当今的信息化社会时代中，计算机系统的可信性显得尤为重要的。一旦计算机系统发生失效，有可能造成大量无辜生命的丧失，引发一场经济灾难，更严重点甚至爆发一场战争。这决不是不可能的事，我们可以经常看到相关媒体报道由于计算机系统的失效而造成了许多事故的发生。例如，由于飞行姿态控制系统的失效而导致客机偏离正常的飞行路线，由于空中交通管理控制系统的失效而致使机场的起降流程陷入瘫痪，由于数字交换系统的失效而导致长途电话服务被迫中断，由于银行在线交易系统的失效而造成 ATM 机被迫暂停，由于座位预订系统的失效而无法进行正常订票等等。

只要计算机系统中存在着故障(包括硬件和软件)，它们迟早都会被暴露出来，这就给计算机系统的可信性埋下了隐患，增加了不确定性因素，必须有相应的技术手段来排除这些故障，从而提高计算机系统的可信性。随着数字系统的迅速发展，作为计算机的灵魂，软件在整个系统中所占的地位和比重越来越重要。因此，软件可靠性作为计算机系统可信性的一个重要属性也就显得更为突出和引人关注[1]。高可靠性软件可以使人们完全免受劣质软件所带来的不良

后果，使人们尽情地享受信息技术、可信计算系统提供的各种前所未有的服务和乐趣。

软件测试是保证获得高可靠性高质量软件的主要关键技术手段[2, 3]，一方面它能够帮助我们揭示软件中的缺陷，另一方面当软件测试达到一定的充分度时它能够帮助我们掌握软件的质量水平，建立对软件的信心。因此，它对提高人们对软件质量的信心有着重要的作用。长期以来，软件测试的一个核心问题则是测试用例的选择，尤其是对黑盒测试而言。测试用例的选择指导着整个测试工作，是软件测试质量稳定的根本保证。如何以最少的测试用例在最短的时间内完成软件测试，发现软件系统中隐藏的缺陷，从而提高软件的可靠性，保证软件产品的优良品质，一直是各大 IT 软件厂商和广大科研人员追求和探索的目标。然而对于生成的测试用例，它的测试质量如何？它究竟能不能检测到软件中的故障？人们有什么理由相信经过这些测试用例测试后的软件，这就涉及到测试用例的测试有效性问题，必需要有相应的方法来评估测试用例的有效性。变异测试是一种基于故障的测试，它不仅可以用来评估测试用例的有效性，而且可以寻找故障检测能力更强的测试用例。

另一方面，由于软件可靠性的研究和发展要迟于硬件领域，因此，软件可靠性中的许多理论和技术是从硬件领域中借用过来的。虽然两者在本质上是相通的，但在形式上还是有很大差别的。例如，软硬件测试中都存在着对故障模型的研究，但软件的故障模型要比硬件复杂得多，对一个待测软件来说，很难估计它可能存在的故障数，更不用说预测存在于该软件中的故障模型。然而传统的软件可靠性模型中仅仅只考虑到时间和失效率这两个因素，使得其并不能确切地反映绝大多数软件的真实可靠性。为此，我们对传统的软件可靠性模型进行了改进，加入了影响软件可靠性的软件复杂性因素和测试有效性，使得改进后的模型更加精确。我们采用变异测试的方法通过向软件中注入人为故障模拟实际软件故障来求测试的有效性。

变异测试是一种计算量超大的单元测试方法。文献[4]中给出了一个例子，对于一个具有 81 行代码的程序产生了 20690 种变异体，如果每个变异体都在同一台计算机上编译并运行一次，其计算量将大的相当惊人。正是由于难以接受

的巨大计算开销使得变异测试在工业界的推广并不很成功，因而一直以来广大科研工作者的工作重心主要是围绕着寻找有效方法来降低变异测试所需的计算开销从而加快其执行过程。由于程序变异后生成了很多变异体，其中很多是等价变异体，而这些等价变异体无助于评价测试用例的有效性，却仍然要测试用例运行于这些等价变异体之上。如果能够尽早检测出这些等价变异体，就可以不需再继续用大量测试用例来测试这些变异体，从而可以节省大量的计算开销加快变异测试过程。基于上述讨论，在变异测试中对检测等价变异体方法进行研究具有重要的理论和现实指导意义，是十分必要和迫切的。

1.3 国内外研究概况

1.3.1 国外研究概况

从 20 世纪 70 年代初期国外就开始对变异测试进行了研究，其发展过程如下。Lipton 于 1971 年在他的学期论文《Fault Diagnosis of Computer Programs》中首次提出变异测试这一基本概念。Demillo、Lipton 和 Sayward 于 1978 年首次发表了关于变异测试方面的论文，详见文献[5]。Acree、Budd、Demillo、Lipton 和 Sayward 于 1980 年在论文《Mutation Analysis》首次实现了一个变异测试工具。Offutt 等人于 1987 年开发了一个著名的支持 Fortran 语言的变异测试工具 Mothra，该工具支持 22 种类型的变异算子。1988 年 Muthar、Krauser 提出在向量处理器上实现变异测试，但后来该思想并没有实现。真正意义上的变异测试在高性能计算机上实现是在 1988 年，Muthar、Krauser 等人在一台具有 128 个处理器的 SIMD 机器上开发了 PMthora，令人遗憾的是 PMothra 运行速度比在单个处理器上的 Mothra 还慢。Offutt、Pargas 等人于 1992 年在一台具有 17 个处理器的 MIMD 机器上开发出了 HyperMothra。

1.3.2 国内研究概况

长期以来由于种种条件的限制，变异测试在国内研究不是很热门，研究检测等价变异体方法的则更是罕见。主要的研究成果有：清华大学郑人杰等人在

文献[6]中讨论了通过弱变异来自动补足测试数据，从而提出了一种提高测试覆盖率的方法。上海大学徐拾义老师在文献[7]中讨论了通过故障控制和等价关系来减少需要注入的故障数，也即减少了等价变异体数，从而节省了计算开销提高了变异测试效率。由上可知，我国在这一领域的研究甚少，亟待广大科研人员投入大量的精力。

1.4 论文的主要研究内容

本论文是以作者攻读硕士学位期间承担课题的工作为基础，在第一章中阐述了课题研究的来源、目的、意义以及国内外研究的现状。第二章阐述了系统可信性的基本概念及其属性，并着重讨论的软件的可靠性及其模型，并进一步引出本课题研究的内容，。第三章阐述了变异测试的基本概念，分析了变异测试的性能瓶颈。第五章为本文的核心，主要分析了三种检测等价变异体的方法。最后第六章总结全文，提出下一步将要进行的工作。

第二章　计算系统的可信性

2.1 可信性的定义及其属性

当前，随着计算机技术的飞速发展，我们的社会正面临着越来越依赖于计算机系统这样一种事实，这就使得我们必须思考这些计算机系统所提供服务的可信性及由此带来的挑战。首先，我们回顾一下近几年来所发生的一些令人困惑的事件。

1) 1994 年，英特尔公司刚刚推出奔腾处理器。一位加拿大的教授发现，奔腾芯片浮点运算有出错的情况。不过，出错概率很小，约在 90 亿次除法运算中可能出现 1 次错误。英特尔公司为此损失了 5 亿美元。

2) 2002 年 9 月，覆盖中国全国的鑫诺卫星多次遭到非法电视信号攻击，一再以非法信号持续攻击正常的卫星通信。

3) 2001 年 4 月，在美国，一个操作错误引起一个网络自治系统(AS)通告从它到 9177 个地址前缀都不可达。这个错误的通告又被通知到其他自治系统。要走这些路由的包就全部被踢掉。

4) 2005 年 7 月，北京部分地区出现上网中断问题，很多 ADSL 以及宽带用户断网超过 30 分钟。通信线路中断故障不时出现。美国 1990 年一次电话网故障引起整个东部地区电话不通。

5) 现在网络受到攻击的事件层出不穷，不胜枚举。电视信号偶然中断或不清晰，用户也常有感知。据美国卡内基梅隆大学统计，在互联网上，2001 年出现 52,655 次入侵事件，而 2002 年前三季度已达 73,359 次，可见增长速度之快。

6) 磁存储设备寿命只有几十年，而且，信息增长的速度远远超过设备增长的速度。美国两院院士、IEEE Fellow、ACM Fellow、麻省理工学院教授 Fernando J. Corbató 说，“几十、几百年后，不可读的数据库和程序语言，丢弃的机器将大量出现。”我们怎么向子孙后代交待？他还惊呼：“用黑盒子系统做选举，可信吗？”现在媒体上的许多统计数据，是经过软件处理以后得到的。经过软件处理以后的统计数据是可信的吗？这里当然也包括选票的统计。

所有这些问题从技术上来说无非是由于硬设备故障、线路故障、软件故障或人为故障。可信计算系统就是要在有故障的情况下，仍然让系统能正常提供服务，或者有紧急预案，自动进行处理。

2.1.1 可信性的起源

1957 年在远程通信领域，开始了一场革命—在电话交换中采用存储程序控制。AT&T 的电话交换系统要求 40 年只有 2 小时停用(即每年 3 分钟)，他们花费了很多年才达到这一目标。1965 年引入的第一个 ESS 处理器采用双机比较的容错技术。1976 年 AT&T 在芝加哥推出了计算机化的电子交换机 4ESS 系统，AT&T 首先在他们的产品中使用了计算机容错技术。1999 使用最后一个 4ESS 系统在电话网上实现包交换，为了达到电话系统的高可用性,在这类系统上实现了几乎所有的容错技术。今天，一个最重要的共识是没有计算机就没有网络。电话网、有线电视网都需要计算机，而且是高可靠、高性能的计算机。计算机网就更不用说了。截止 1995 年，全世界有 5 亿多台电话,多于 10 万个电话交换局，除了大量采用计算机外别无它法，而且一旦计算机出现故障,网络就要失效。这已经被许多事实所证明。

痛苦的努力使无故障系统的幻想被彻底丢弃,转而采用软件容错、软件重用和面向对象的程序设计。同时，新情况、新需求更加刺激计算机系统可信性技术的发展。现在世界上，无论国际还是国内，都面临多家电话公司的竞争,已经不再是一家的天下。但国际通信服务必须畅通无阻，不能几分天下。从技术上来看，从过去老的电话服务到可以移动的个人通信空间、从电视广播到多媒体传播、从模拟交叉点到异步传输方式、从电子到全光子网络，所有这些都说明远程通讯不仅服务和技术在变化，整个网络结构、市场结构和商务都在变化，给可信计算提出了很多新问题，也提供了广阔的应用前景。

由上可知，可信计算技术的发展应当从容错计算说起，容错计算的研究与发展应该以 1971 年召开第一届国际容错计算学术会议（FTCS-1）为起点。“容错”当然不是指“容易错” 而是指“容许错”，更确切些说应该是“容许故障”。从 1975 年开始，商业化的容错机推向市场，到九十年代，软件容错的问题被提

了出来[8]，进而发展到网络容错。1995 年在第二十五届国际容错计算学术会议（FTCS-25）上，IEEE Fellow，A. Avižienis 教授等人提出了可信计算(Dependable Computing)的概念[9]。

2.1.2 可信性的定义

一个计算机系统的可信性是指该系统可以提供可信赖的计算服务的综合能力，而且这种可信赖性是可以验证的[10]。之所以称它为一种综合能力，是因为衡量这个能力的标准并不是，也不可能是单一的，而是一个由多方面综合因素确定的评价体系。它主要涉及到 6 个基本评价属性[11]，即：

- 可靠性(reliability)
- 可用性(availability)
- 可测性(testability)
- 可维护性(maintainability)
- 安全性(safety)
- 保密性(security)

上述诸方面组成了度量可信性的属性。其中，可靠性是对系统在一定条件下能正常完成所规定任务能力的评价，它的评价标准主要在于系统保持正常运行时间的长度；可测性则是对一个系统具有避免故障、防止故障(即，在系统开发阶段，在系统正式投入运行之前和系统发生故障之前就必须考虑到的措施)和检测、诊断故障的能力以及在系统正常运行时，修复故障能力的评价；可用性和可维护性是对系统在具有一定的维护措施以后，随时可提供正常服务能力的评价；安全性则是对系统一旦发生失效(失效的定义将在下一节给出)后限制其发生严重后果能力的评价；而保密性乃是关于对系统保持其内部所存信息的信任度和完整性(系统是否被非法入侵，复制和篡改等)能力的评价。

随着信息和 计算技术的不断发展，可信性计算理论和技术正在深入地应用于各类计算系统中。例如，由于高集成度的微处理器的发展，计算机系统已经从条件优越的计算机机房走向了工业、农业、国防、通信和交通等领域的现场，而这些现场可能压根就没有空调冷却设备，而且温度和湿度随时都有较大幅度

的变化，震动频繁，电源、电网电压变化以及电磁场干扰作用等。这就要求系统必须具有抗恶劣环境的功能，可以通过可靠性、可用性和可维护性技术来确保。又如在网络环境中，由于有线和无线网络的出现，随之而来的是病毒、黑客以及各种非法入侵行为的破坏和干扰，尤其是对软件系统的攻击和破坏使得一些计算系统往往不能提供可信的服务而使正常的用户蒙受巨大的损失和严重的后果。因此，一个可信计算系统必须具有抗干扰的功能，这可以通过保密性和安全性技术来实现。正如上面所述，在实际的应用环境中评估一个系统的可信性时，对上述 6 个属性，实际上并非是一视同仁，而要有所侧重。这在很大程度上取决于系统的应用范围，开发系统的需求和成本以及系统所处的环境等诸方面的因素。

对可信性的评价实际上是一个综合评价的过程，由下式表示：

$$D = w_1R + w_2A + w_3M + w_4T + w_5S + w_6U \tag{1}$$

且

$$\sum_{i=1}^{6} w_i = 1 \tag{2}$$

式中，D 表示一个系统的可信性，R、A、M、T、S 和 U 分别表示该系统的可靠性、可用性、可维护性、可测性、安全性和保密性的属性参数值。而 $w_i(1\le i\le 6)$ 则分别表示该系统对上述各属性重要性的权值。这样，式(1)就表示了可信性的综合评价值，其中各个属性的权重由系统的开发者、设计者、用户等根据具体系统的应用需求、应用环境、开发成本和开发周期等各方面的因素综合决定。因此，对于两个不同的系统，由于不同的复杂因素和应用条件，很难对其可信性方面的评价作定量的比较，但是对于同一个系统来说，如果对系统进行了某些改动或调整，那么在这些改动或调整完成以后，该系统在改动前后的可信性评价作定量比较是可以实现的。因为在这种情况下，式(1)的条件应该是基本相同的。

2.2 影响系统可信性的因素

每一个计算系统在其运行期间都可能受到来自各方面的干扰和影响，从而

降低或损害了系统的可信性。其中，影响可信性最重要的有如下三个相关因素：故障(fault)、错误(error)和失效(failure)。这三个因素是造成系统可信性受损的直接原因，并且三者之间有着密切的因果关系，下面将进一步来分析它们的性质、表现形式、相互关系和对可信性的影响。

2.2.1 失效(failure)

按照事物发展的一般规律，任何一个系统在运行期间都可能出现非正常现象。即，系统在运行到一定的时间，或在一定的条件下偏离它预期设计的要求或规定的功能。这种现象通常称为失效。换句话说，就一般情况而言，系统的失效是不可避免的，是绝对的。而保持系统的正常运行则只是局部的，相对的。同时，由于人们对数字计算系统的依赖程度愈来愈大，因此，一旦一个系统出现失效，给人类带来后果也将愈来愈严重。例如，美国，科罗拉多州首府丹佛的国际机场，由于行李管理系统的失效而延期开张达数月之久。雅丽安娜 V 号火箭由于电子控制系统中错写了一个常数问题而引起的失效不得不使它的首航日期多次改期。而发射到火星的探测器由于设计中使用了不统一长度单位（即，英寸和厘米的差异）而导致多次失效。更有甚者，还有不少由于治疗癌症的仪器中控制系统的失效导致多名病人死亡的例子。这些例子都说明了一个事实：功能强大的计算系统系统在给人类带来福音的同时也存在着失效后产生严重后果的可能性。实际上，这是一对难以回避的矛盾和严峻的事实，我们必须面对现实，正视这对矛盾。而要解决这对矛盾就必须避免或减少由于失效而产生的各种不良后果，必须对失效的起因有一个深入的了解。

2.2.2 故障(fault)

故障，这个词在一般的日常生活中用得很多。在硬件系统中它既有“缺陷”(defect)，“瑕疵” (flaw) 等含义，而在软件技术中也有所谓“隐患”(dormant fault)，“臭虫”(bug)等的意思。事实上，故障可以认为是系统产生失效的似然条件和推理上的原因。之所以说它是“推理上的原因”是因为要确切地诊断或识别系统中的故障是一件十分复杂的过程。它涉及到相关的诊断者自身掌握知

识水准和信息量的多少，对被诊断对象的功能的理解程度，以及对该系统的控制和观察能力等诸多因素。故障和失效存在着密切的因果关系。然而，从失效现象逆向找寻产生失效的根源 — 故障，却并没有直接的明显的方法。因此，人们只能根据失效的表现形式，依靠分析和推理，去发现失效的原因。因此，即使找到了一个故障，也只能是产生失效的一个推理上的原因。另外，即使找到了一个可能的故障，而在系统中确定这个故障的具体位置，也是一个十分复杂的推理过程。例如，对一个硬件系统(产品)中存在的故障，或许只须找到故障所在的模块(如集成电路芯片)即可(称为系统级诊断)。但是，有时还必须作进一步诊断以确定故障在芯片中的哪一个门电路上(称为门级电路诊断)，或者更为详细地需要了解故障是由哪一个MOS电路或晶体管造成的(称为电路级诊断)等等。除此以外，从原因上来分析，引起失效的故障有可能并非是由单个故障造成的，而是由多个故障(包括人为的因素等)共同的影响产生的。这样，推导失效的原因就更为复杂了。类似地，对软件系统的故障诊断同样需要考虑到不同模块(或子程序)之间的关系。然后通过对这些模块之间关系进行分析将故障诊断逐步细化，最终来确定故障可能所在的位置等(可能是在某一个模块中，也可能确定到某一行程序，或者是某个符号或参数等)。另外，由于一个软件系统运行必然同它所使用的操作系统的环境和其它同时运行的软件系统有着密切的关联。而这些因素同时也增加了软件系统故障诊断的复杂性。综上所述，我们的结论是，根据对失效的表现，通过分析研究寻找到失效的根源 — 故障，只能是该失效推理上的起因。

可以从不同方面来对故障进行分类。从故障的基本性质分析，故障有功能性故障和技术性故障之分。前者主要以非故意性人为造成为主，如在输入程序时漏输一行语句或错写一个符号。后者则主要发生在硬件系统中，如信号线的开路或短路，导通的半导体突然截止。从引起故障的故意性分析，有非故意性故障和故意性故障之分。如设计者在设计时对需求分析研究不够，甚至理解错误，从而引起规格说明或设计上的某些故障就属于非故意性故障。而故意性故障主要是通过人为手段故意造成的故障。从故障的持续时间分析，有永久性故障和暂时性故障之分。永久性故障一旦形成就不可能自动消失，如逻辑电路中

的信号线发生 *s-a*-1 或 *s-a*-0 故障。暂时性故障的发生与它的环境、元器件的性能有很大的关系，这类故障可进一步细分为间歇性故障和瞬时性故障。前者以系统的内部原因为主，故障以间歇性状态出现，表现为时好时坏。后者以系统的外部原因为主而引起的，以一次性出现故障为它的表现形式。

2.2.3 错误(error)

故障的出现并不意味着它的影响一定会在系统的(外部)功能上立刻表现出来。相反，一般情况下，一旦出现故障，在系统外部并不是立即暴露， 因而也无法立即发现故障的存在。这是因为在一段运行期间内，故障的出现可能并不立即影响或反映在系统所提供的服务上。如，在一个硬件系统中，某一个逻辑门发生了故障，但是在当前运行的任务中，该逻辑门恰好处在不工作或等待状态。于是，这个故障就不会影响系统的正常工作。也就是说，从出现故障到系统的失效(即系统提供了非正常的服务)之间存在一段延迟时间，称为故障潜伏期。潜伏期可以很短，也可能很长，视具体系统和故障的不同而不同。在潜伏期中的故障称为被动故障或称为非活动故障。当故障一旦被激活产生后果时则称为活动故障。但是，即使是活动故障也未必能立即引起系统的失效。这是因为活动故障所产生的后果需要有一定的条件：即，时间(延迟)和通路(称为故障链)才能传递到系统的输出端，使系统因功能发生改变而失效。因此，故障的后果必定先影响到它所在部分的功能或状态，然后，逐步向外传递或扩张。我们将在系统内部的某一个部分(模块)由于故障而产生了非正常(即违背了设计要求和说明规格)行为或状态的现象称为错误。显然，错误是故障的产物和后果，而故障是错误的起因。换一句话说，错误是系统内部出现故障而导致非正常状态的表现和反映。然而，错误就像传染病一样具有传递性和扩散性，把原始故障的影响传递到系统的输出端(外部)为止。最终，可能造成整个系统的失效。因此，从故障的产生到错误的出现，进而发展到系统的失效形成了一条“故障→错误→失效”的故障传输链，从故障的出现到故障被激活而形成错误之间的时间就是所谓的故障潜伏期。故障、错误、失效三者之间一个周期内的关系可以用如下图来表示。

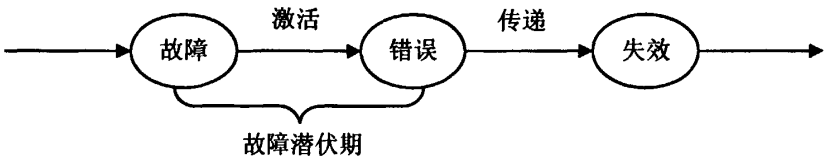

图 1 故障、错误和失效之间的关系

2.3 提高可信性的措施

如上所述，故障及其衍生的错误是影响计算系统可信性最重要的因素，也是使得一个系统发生“崩溃”的根本原因所在。故障可能发生在系统生命周期的各个阶段，即从系统的需求分析、规格说明进入设计阶段，到产品的完成，然后投入运行直到系统结束生命为止，每个阶段都可能引入新的故障，而每引入一个新的故障，就会降低系统的可信性。因此，要提高系统的可信性，就必须采取各种措施来对付故障，以减少故障在系统中的生存空间。在长期的实践中人们积累了以下四种主要技术：

- 避错技术(fault avoidance)
- 防错技术(fault prevention)
- 排除技术(fault removal)
- 容错技术(fault tolerance)

其中，避错技术主要应用在系统设计的初级阶段，从规格说明至设计阶段，应避免人为的将故障引入系统，主要强调验证对设计过程描述的确切性和对已经描述的对象的正确性。防错技术主要应用于生产制造或编程调试阶段，应防止将故障留在系统内。排错技术主要应用于产品完成到出厂之前，必须检测和排除驻留在系统中的故障。而容错技术主要应用于系统投入运行的现场工作阶段，在无法确定故障是否存在的情况下，应具有保证系统继续运行(或降级运行)的能力以及限制由于系统失效而引起的严重后果的能力。

2.4 软件可靠性及其模型

当前，软件在整个系统中所占比重是越来越大，因此，软件的可靠性作为

系统可信性的一个重要属性，也备受人们的关注和重视。软件是人的大脑逻辑活动的产物，因为人的教育程度、工作经验、思维习惯、认识能力和工作敬业精神等多方面的差异，致使软件出现故障在所难免。对软件建立起可靠性模型、进行可靠性预测和评估是实现高质量、高可靠性软件的最重要手段之一，不仅可以预测软件的可靠性，确定软件质量何时达到预期要求，避免因过度测试而来不及开拓市场的悲剧，而且可以平衡软件开发和软件测试之间成本，将发行软件存在严重问题的风险降至最低。

2.4.1 软件可靠性定义

可靠性是最早为人们所公认的评估可信性的标准之一，最初是用于衡量一个物理系统随着时间的推移而保持连续正常服务工作的能力。近些年来人们也从另一个角度来定义可靠性：一个系统在给定的时间和条件下完成其规定任务的能力。用数学语言来描述的话，一、可靠性是时间的函数；二、假设当时间 t 为 0 时，系统是正常的，可靠性则是在所规定的条件下，当时间为 t 时系统能正常工作的条件概率。

软件可靠性可以定义为一个软件系统在给定的时间段内能正常运行，并完成其在规格说明中规定的所有功能而不发生故障的概率。软件可靠性技术就是一种致力于预测、评估和提高这个概率的技术。

一般说来，在硬件系统中，系统的可靠性是随着时间的推移而下降的。这是因为在硬件系统中，随着时间的推移总会由于系统老化现象使得电子元器件质量下降，从而导致系统发生故障而失效。但是，在软件系统中如果不考虑软件维护问题，则由于系统中不存在因元件老化而质量下降的问题，从理论上讲，软件系统的可靠性往往并不随时间的变化而变化。但是由于软件中存在着固有的故障和软件维护时产生的各种新故障，其可靠性同样可以随着时间的变化而下降。

2.4.2 软件综合可靠性模型

从可靠性理论的基础来看，传统的软件可靠性模型是从硬件可靠性模型借

用过来的，它被定义为：

$$R(t) = e^{-\int_0^t \lambda(\tau)d\tau} \tag{3}$$

其中 $\lambda(\tau)$为软件的失效函数，$R(t)$表示在时刻 $t=0$ 时系统正常工作，而在 t 时刻时系统不发生失效的概率[12]。然而式(3)仅仅只考虑了时间和失效率这两个因素，用它来评估软件的可靠性是不够精确的，甚至还可能对用户或测试者产生误导，引起严重后果。

在软件可靠性模型中必须要考虑引入能够体现影响到软件可靠性的软件复杂性和测试用例的有效性等重要因素。这是因为一般来说，软件越复杂就越有可能存在更多的故障，而软件的可靠性和潜伏在软件中的残留故障数是密切相关的[13, 14, 15]，因此应尽可能降低软件的复杂度。另一方面，目前大多数软件可靠性模型是基于测试和软件排错率来计算的，说明测试用例的有效性对被测软件的可靠性有密切的关系。鉴于这两种因素对软件可靠性的影响，本课题组对传统的软件可靠性进行了改进，提出了一个软件综合可靠性模型，详细的描述可以参见文献[16]。

设 $\lambda(c)$表示软件的复杂性，则它被定义为

$$\lambda(c) = 1 - e^{-(\ln Hv + \ln Ml + \ln If)/3\ln LOC} \tag{4}$$

其中 Hv、Ml、If、LOC 分别表示被测程序的 Halstead 度量、循环分支数度量、信息流度量和代码行度量。设 $\lambda(e)$表示测试用例的有效性，它被定义为被测软件在给定测试时间 t 内由测试用例所发现的注入故障数与注入故障总数之比，即：

$$\lambda(e) = f_d(t)/f_T \tag{5}$$

其中，$f_d(t)$为测试期间所发现的注入故障数，f_T为注入故障的总数。从式(5)可以看出，测试用例的有效性的值应该在[0, 1]之间，$\lambda(e)$的值越大测试用例就越有效，即测试用例检测到的故障数越多。一般总是认为，一个给定测试集在测试期间检测到的注入故障数越多，那么同样的测试集也能检测到更多的非注入故障，从而就可以得到更好的测试有效性。此外，我们还作了一些约定如下：

1)修改后的失效率必须是 $\lambda(t)$、$\lambda(c)$、$\lambda(e)$的函数。

2)如果 $\lambda(e) = 1$，说明该测试集的有效性达到最大值，那么原始失效率 $\lambda(t)$ 已经足够精确了而无须修改。

3)如果 $\lambda(e) = 0$，说明该测试集未检测到任何注入故障，那么这个测试集需要重新修改，且复杂性因素要完全加入到可靠性模型中。

4)如果 $0 < \lambda(e) < 1$，那么原始失效率 $\lambda(t)$ 和复杂性 $\lambda(c)$ 都需要添加到软件可靠性模型中去。

在正常的情况下，测试的有效性 $\lambda(e)$ 不可能去 0 和 1 这两个极端值，根据上述的约定，我们提出的软件综合可靠性模型为

$$R^*(t) = e^{-\{k_1\lambda(t) + k_2\lambda(c)[1-\lambda(e)]\}} \tag{6}$$

其中，k_1 和 k_2 的值依赖于具体的用户需求，通常 $k_1 + k_2 = 1$。

本课题组成员在文献[17, 18]对软件复杂性度量极其对软件可靠性的影响进行了详细的研究，本文不再重复对复杂性因素的探讨，将主要精力集中在测试有效性上。然而我们不可能从一个商业软件来直接计算测试的有效性，即便可以也需花费大量的时间和精力来查找程序中的所有缺陷才能计算出测试的有效性。为了解决这个问题，本文采取变异测试模拟人为故障这种切实可行的故障注入方法进行研究。

第三章　变异测试

3.1 变异测试的基本概念

变异测试是一种功能强大的面向故障的单元级软件测试技术[19]。其主要思想是通过变异算子(mutation operator)对被测原始程序(original program)中的语句作一些语法上合法的微小修改，即人为地引入故障，从而生成大量的新程序，每个新程序称为原始程序的一个变异体(mutant)，且每个变异体中仅包含一个人为故障，然后输入测试用例，分别在原始程序及其变异上执行，通过观察比较他们的输出结果来判断是否检测到语句中的变更。如果两者的输出不一样，则检测到了该变更，就称测试用例将该变异体杀死(killed)了。将该变异体从整个变异体集中删除，也就是接下来的测试用例不必在该变异体上运行，即死亡的变异体无需再保留在测试过程中，因为由该变异体所代表的故障已经被检测到了。如果两者的输出一样，则无法将变异体杀死。导致变异体无法被杀死的原因主要有两个：

1) 测试数据集还不够充分，通过扩充测试数据集便能将该变异体杀死。
2) 该变异体在功能上等价于原始程序，称这类变异体为等价变异体(equivalent mutant)。

杀死变异体的过程一直持续到所以的变异体都被杀死或变异分数达到预期要求。变异分数是已杀死的变异体数目与所以产生的非等价变异体数目之间的比值，测试者的。

以下通过一个例子来更形象地说明上述基本概念。图 2 为一个求两个整型数中的最小值函数 Min 和它的 4 个变异体。图的左边为原始程序，图的右边为相应的变异体。值的注意的是，为了便于比较和讨论，这里将 4 个变异体合写在一个程序中了，每一通过变异得到的语句由 Δ 标明。每一行变异的代码和余下来的非变异代码可以被看作是一个独立的程序错误版本，即变异体。变异算子是由原始程序生成变异体的规则，在本例中也就是由左边的函数映射到右边的函数的规则，总共有两大类变异算子。其中 Δ1、Δ3、Δ4 属于变量间替换

变异，也就是用另外一个语法上合法的变量来替换当前变量，而Δ2(用“>”替换“<”)则属于运算符间替换变异，即，用另外一个语法上合法的运算符来替换当前的运算符。

```
     Original Program                    Mutants
int Min( int i, int j )          int Min( int i, int j )
{                                {
    int MinVal;                      int MinVal;
    MinVal = i;                      MinVal = i;
    if(j < i){                Δ1     MinVal = j;
        MinVal = j;                  if(j < i){
    }                         Δ2     if(j > i){
    return MinVal;            Δ3     if(j < MinVal){
}                                        MinVal = j;
                              Δ4         MinVal = i;
                                     }
                                     return MinVal;
                                 }
```

图 2 函数 Min 及其 4 个变异体

在整个变异测试过程中，首先由变异系统生成被测程序的变异体，然后向系统中输入测试用例，用户检查每一测试用例在被查程序上的执行结果是否正确。如果不正确则说明被测程序中有故障，必须对被测程序进行修改然后重新开始测试。如果正确，就让测试用例在每一活着的变异体上执行。如果变异体的执行结果不同于被测原始程序，就认为该变异体被杀死了，该测试用例也就被认为是一个有效的测试用例而保留下来。在图 2 中，变异体Δ3 用变量 MinVal 来替换变量 *i*，而在其前的一条语句中变量 *i* 的值被赋值给了变量 MinVal，此时变量 *i* 和变量 MinVal 有相同的值。因此，变异体Δ3 是一个等价变异体。

接下来我们用数学语言来对变异测试进行正式的定义。设有一测试集 T 对程序 P 进行测试，且测试集 T 中的任一测试用例都通过了程序 P，即经过这些测试用例测试后未发现程序 P 中有故障。现在假设将程序 P 作一细小修改而产生新程序 P′，显然 P′为 P 的一个变异体，同样用测试集 T 对 P′进行测试。此时

会出现什么样的结果呢？在测试集中也许可能存在一个测试用例 $t \in T$，使得 $P(t) \neq P'(t)$成立，这时我们称测试用例 t 将 P′和 P 鉴别开来，即 t 将变异体 P′杀死。在测试集中也可能不存在任何测试用例 $t \in T$，使得 $P(t) \neq P'(t)$成立，在这种情况下，我们称测试集 T 无法将 P′和 P 鉴别开来，P′称为活着的变异体。如果在 P 的整个输入域中都不存在测试用例 t 将 P′和 P 鉴别开来，就称 P′和 P 是等价的。如果 P′和 P 不是等价的，而测试集 T 中又不存在测试用例将 P′和 P 鉴别开来，这就说明测试集 T 是不充分的，需要对 T 进行改进。

变异测试可以系统地模拟软件中的各种缺陷,并且寻找能够发现这些缺陷(即杀死相应的变异体) 的测试用例集。因而，它不但可以用于对测试用例的有效性进行评估，而且通过与用户的交互，逐步地加强测试用例的完备性，最终可以帮助用户生成完备的测试用例集。此外，它还具备故障检测的功能。

3.1.1 变异测试的两个前提

变异测试是建立在以下两个基本假设之上，即程序员的能力假设(the competent programmer)和藕联效应(coupling effect)[4]。程序员的能力假设指出熟练的程序员所编写的程序是足够接近正确的。虽然熟练的程序员所编写的程序也可能是不正确的，但和正确的程序版本相比，它仅仅是多包含了一小部分的故障而已，可以认为它是从正确版本的经过几次变异而来的。藕联效应指出复杂故障是由简单故障通过各种关系藕联而成的，因此能够检测到程序中所有的简单故障的测试数据集一般也能够检测到程序中大部分的复杂故障。Offutt 在 1992 年对藕联效应准确性进行了实验研究[20]，而 Wah 则从理论上对藕联效应的准确性进行了研究[21, 22]。正因为基于这两个假设，所以在进行变异测试的时候每个变异体只需对原始程序变异一次，每次仅引入一个故障，而无需模拟那些复杂的故障，这样可以大大减少生成的变异体数量，从而降低变异测试的计算开销。同时，研究结果也表明变异体与实际的软件缺陷很相似，而且从某种程度上来说，变异体所注入的故障往往要比真实故障更难检测[23]。

如果对原始程序进行一次变异，即一次只引入一个故障，则称为一阶变异。如果变异体是通过两次变异得到的，则称为二阶变异。类似的也可以定义更高

阶的变异。例如，$x = z + y$ 是 $z = x + y$ 的二阶变异体，因为它应用了两次变量替换运算符。在实际的变异测试中，基于藕联效应和降低变异测试的开销问题考虑，一般只需生成一阶变异体。

3.1.2 变异算子(mutation operator)

经验研究表明，不管是编程老练的高手还是刚刚接触编程的新手在编程过程中都容易犯一些简单的非技术性错误，如本来应该在程序中写 x < y + 1 却因粗心漏掉后面的常数而写成了 x < y。变异算子就是用来模拟那些典型的用户输入错误(如运算符或变量名使用错误等)以及强制使某些表达式满足一定的条件(如使表达式的值为 0)。

实际上，变异算子是一个将原始程序(original program)转换成变异体(mutant)的规则，因此也称为变异规则或变异转换。也可以将一个具体的变异算子 O 看成是一个函数，它将被测程序 P 映射为 k 个变异体。常用的变异算子有变量替换、运算符替换、常数替换和删除或插入整条语句等。如表 1 所示，为一些变异算子的例子。

表 1 变异算子示例

Mutant Operator	In P	In Mutant
变量替换	*z = x * y + 1;*	*x = x * y + 1;* *z = x * y + 1;*
关系运算符替换	*If(x < y)*	*If(x > y)* *If(x >= y)*
算术运算符替换	*z = x * y + 1;*	*z = x * y – 1;* *z = x + y -1;*
加 1	*z = x * y + 1;*	*z = x * (y +1) + 1;* *z = (x +1) * y +1;*
用 0 替换	*z = x * y + 1;*	*z = 0 * y + 1;* *z = 0;*

设计变异算子是变异测试中一项长期而重要的工作，变异算子设计得合理

有利于提高变异测试的效率。不同的程序设计语言由于语法上的差异使得出错的形式也有所不同，这就导致不同语言间设计的变异算子也有所不同。例如，由于面向对象程序设计语言引入了类、对象、继承、多态性和类属机制等新思想，因此，为面向过程式语言 C 和面向对象语言 Java 所设计的变异算子肯定是不同的。对同一种程序设计语言而言，由不同的人所设计的变异算子也可能不同，那么如何对所设计的变异算子的合理性进行评价呢？设 S_1、S_2 是为同一种语言设计的两个不同的变异算子集，一般来说，如果由 S_1 生成的变异体比 S_2 生成的变异体能够检测到更多故障，则认为 S_1 要比 S_2 更加合理。

表 2 是著名的变异测试系统 Mothra 针对 Fortran 77 程序语言而设计的 22 个变异算子[24]，它们都是经过长期的修改和精简的。其中每个变异算子都由三个字母组成的助记符，如 ASR 是 Array reference for scalar variable replacement 的缩写，表示在程序中每一处的数组引用被替换为一个标量变量。按照语法规则进行分类，这 22 个变异算子可以划分为三大类。AAR、ACR、ASR、CAR、CNR、CRP、CSR、SAR、SCR、SRC 和 SVR 属于操作数替换算子类，它们均表现为将一个操作数替换为另外一个合法的操作数，主要用于模拟程序员在编程过程中对变量名和数组引用名的误用这种场合，如变量的定义与引用、数组下标越界和非法的算术运算等计算型故障。ABS、AOR、LCR、ROR 和 UOI 属于表达式修改算子类，这一类算子表现为用插入新的运算符或替换原有运算符来修改表达式，主要用于模拟程序员在编程时对表达式误用这种场合，如表达式内部用错算术运算符或关系运算符而导致分支型和循环型故障。DER、DSA、GLR、RSR、SAN 和 SDL 属于语句修改算子类，它们用于修改整条语句，如删除动态内存释放语句可以模拟内存泄漏故障。

Mathur、DeMillo 等人对 C 语言设计了一组变异算子[25]，但由于 C 语言的语法要比 Fortran 语言复杂得多，因而设计出来的变异算子要多得多，一共包含 77 个变异算子。可以将这些变异算子分成四大类：

1) 语句变异(statement mutations)
2) 运算符变异(operator mutations)
3) 变量变异(variable mutations)

4) 常量变异(constant mutations)

每大类下又细分为多个小类，如变量变异又可细分为标量变量引用替换、数组元素替换、指针引用替换、结构引用替换、数组下标变异和结构成员替换等，其它类的变异算子详细情况可以参考文献[25]。

表 2 Mothra 系统中的变异算子

变异算子	全称	描述
AAR	Array reference for array reference replacement	数组引用替换数组引用
ABS	Absolute value insertion	绝对值插入
ACR	Array reference for constant replacement	数组引用替换常量
AOR	Arithmetic operator replacement	算术运算符替换
ASR	Array reference for scalar variable replacement	数组引用替换标量变量
CAR	Constant for array reference replacement	常量替换数组引用
CNR	Comparable array name replacement	同类数组名替换
CRP	Constant replacement	常量替换
CSR	Constant for scalar replacement	常量替换标量变量
DER	DO statement END replacement	DO 语句替换
DSA	DATA statement alterations	DATA 语言变化
GLR	GOTO label replacement	GOTO 标号替换
LCR	Logical connector replacement	逻辑连接符替换
ROR	Relational operator replacement	关系运算符替换
RSR	Return statement replacement	RETURN 语句替换
SAN	Statement analysis	语句分析
SAR	Scalar variable for array reference replacement	标量变量替换数组引用
SCR	Scalar for constant replacement	标量变量替换常量
SDL	Statement deletion	语句删除
SRC	Source constant replacement	源常量替换
SVR	Scalar variable replacement	标量变量替换
UOI	Unary operator insertion	一元运算符插入

3.1.3 等价变异体(equivalent mutant)

如前所述，等价变异体是指该变异体在功能上等价于原始程序，也即等价变异体的输出结果总是一直和原始程序的输出结果相同，因此，根本就不可能

存在测试用例将原始程序和等价变异体鉴别开来。如图 2 中的Δ3 就是原始程序的一个等价变异体，也就是说在函数 Min 的整个输入域中找不到任何的 i、j 值使得Δ3 的输出值和 Min 的输出值不一致。

在每一变异体上都执行完所有的测试用例后，剩余的活变异体可以划分为两大类，一类变异体为可以杀死的变异体，但由于目前的测试数据不够完备而不能将其杀死，可以通过生成专门的测试用例来将其杀死。另一类是等价变异体。等价变异体无助于提高变异分数，如果一直驻留在测试过程中将耗费巨大计算开销，因而应当尽早将其检测出来，一旦识别为等价变异体剩余测试用例就不必再对其进行测试了，从而可以避免不必要的计算资源的浪费。因此，等价变异体的检测就成为变异测试中的一个重要部分，其具体的检测方法将在第四章中详细讨论。

手工检测等价变异体是一个非常费时的过程，它有赖于测试者的经验和所掌握的技能。因此，它造成变异测试的成本极其高昂，降低了变异测试的效率，是阻碍变异测试广泛应用的重要因素之一，必须寻找出相应的自动或半自动等价变异体检测方法。显然，自动检测等价变异体的一个好处是它可以节省测试人员大量的时间和精力。另外一个好处是它可以防止人们在手工检测等价变异体时出错[26]。在对等价变异体进行标记的时候，一般容易出现两种类型的错误：

1) 将非等价变异体标记为等价变异体。

2) 将等价变异体标记为非等价变异体。

其中，第 2 种错误可以在后期的测试过程中得到纠正，因此，实际上只有第 1 种错误才需要值得注意。

Budd 和 Angluin 在文献[27]中对程序间的等价性和测试数据生成之间的关系进行了详细的研究，并且证明了如果存在一个能在有限时间内完成的确定性算法来判断两个程序之间是否等价，那么同样就存在一个能在有限时间内完成的确定性算法来为程序生成充分的测试数据，反之亦然。事实上，一般来说，上面两种算法根本就不存在。因此，等价性问题不可能存在一个完全的解决方法，也就是说判断任意两个程序或两个变异体之间是否等价是一个不可判定性问题。幸运的是，变异体之间的等价性问题要比一般的等价性问题有一个明显

的优势，即我们无需判断任意两个不相关程序的等价性问题，根据变异算子的定义，变异体几乎和原始程序相同，仅在一处不相同。虽然 Budd 和 Angluin 证明了等价性问题是不可判定的，但是对大部分实际情况来说，两个相似的程序之间的等价性问题是可以判定的。我们可以利用这一点来开发相应的自动等价变异体检测方法。

3.2 测试充分性

为了便于说明，我们先做如下假设。设 P 是按照功能需求 R 而编写的一个程序，我们将它记为(P, R)，其中功能需求 R 细分为 n 条具体的需求，即 R_1、R_2、...、R_n。设 T 为测试程序 P 是否满足功能需求 R 中的所有功能而构造的包含 k 个测试用例的测试集，而且 T 中的所有测试用例在 P 上执行都能得到正确的结果。现在我们可能会问： 测试集 T 的测试效率如何？或者换一种形式问，程序 P 是否得到了完全的测试？或测试集 T 够充分吗？

如前所述，变异测试的主要用途之一是评估测试用例的充分性。对程序 P 的一个给定测试集 T，评估其测试充分性的过程如下：

1) 对程序 P 应用变异算子生成变异体集 M，其中 M 中包含 k 个变异体，即 $M = \{M_1, M_2, ..., M_k\}$。
2) 对每一个变异体 M_i，查找在测试集 T 中是否存在一个测试用例 t，使得 $M_i(t) \neq P(t)$成立。
3) 在第二步执行完后，假设有 $k_1(k_1 \leq k)$个变异体被杀死，还有 $k-k_1$ 个变异体活着，则：
   - 如果 $k-k_1=0$，表明测试用例集 T 对该变异而言是充分的。
   - 如果 $k-k_1>0$，则我们可以按如下公式来计算变异分数(MS)：

$$MS = \frac{k_1}{k-e} \qquad (e \leq k-k_1) \tag{7}$$

其中，e 为等价变异体数目。

如果变异分数未达到 1，则说明可能还存在与原始程序 P 不等价的变异体活着。每一活着的变异体都需要和原始程序鉴别开来，因此，需要设计一个新的

测试用例 *t*，使得 *t* 至少能够将一个活着的变异体杀死。最后，将 t 加入到测试集 T 中，再重新计算变异分数 MS。如果 MS 未达到 1，就要重复上述添加过程直到 MS 的值为 1 或者满足测试者设定的阈值。

变异测试不仅仅是用来衡量测试数据的揭错能力及评估测试的充分性，而且可以用来揭示软件中的故障。下面通过一个例子来说明如何用变异测试来改进测试用例集及发现程序中潜伏的故障。

Original Program P

```
int foo(int x, int y){
    return (x – y);        ← 本来应该为 return (x + y);
}
```

M_1

```
int foo( int x, int y){
    return (x + y);
}
```

M_2

```
int foo( int x, int y){
    return (x – 0);
}
```

M_3

```
int foo( int x, int y){
    return ( 0 + y);
}
```

图 3 有故障的函数 foo 及其 3 个变异体

图 3 为返回两个整型数 *x* 和 *y* 之和的函数 *foo* 及其 3 个变异体 M_1、M_2 和 M_3，其中 M_1 是将“–”运算符替换为“+”运算符而得到的，M_2 是由常量 0 替换为变量 *y* 得到的，M_3 是由常量 0 替换变量 *x* 而得到的。显然，*foo* 是不正确的。现在用测试集 T = { t_1: <*x* =1, *y* =0>, t_2: <*x* = -1, *y* =0> }对函数 *foo* 进行测试，不管是输入测试用例 t_1 还是 t_2，*foo* 都能得到正确的结果。接下来我们将测试集 T 中的测试用例输入到每一个变异体中执行直到变异体都被鉴别出来了或所有的测试用例都用完了为止，执行结果如表 3 所示。在这三个变异体执行完后，变异体 M_3 被杀死了，另外两个变异体还仍然活着。变异分数 MS 仅为 1/3，未

达到 1，因此，需要继续添加测试用例以便将活着的变异体杀死或将他们标记为等价变异体。仔细观察 M_1 发现，要将 M_1 和 P 鉴别开来测试用例必须满足如下条件：

$$x-y \neq x+y \quad 即 \quad y \neq 0 \tag{8}$$

于是，我们增加一个新的测试用例 t_3: <x =1, y = 1>到测试集 T 中，t_3 在程序 P 上的执行结果为 $P(t_3)=0$，但根据函数 foo 的功能需求是要求两者之和，应该得到的是 $P(t_3)=2$。因此，原始程序 P 中肯定包含故障。同时，由于 $M_1(t_3)=2$，$M_2(t_3)=1$，测试用例 t_3 将这两个变异体都杀死了，变异分数 MS 已经达到 1 了。那么是否所有能杀死变异体的测试用例同时也可以揭露原始程序中的缺陷呢？

表 3 测试用例执行结果

T	P(t)	$M_1(t)$	$M_2(t)$	$M_3(t)$
t_1	1	1	1	0
t_2	-1	-1	-1	0
		Live	Live	Killed
		Live	Live	Killed

有时存在这样一种变异体，只要测试用例能将该变异体和原始程序鉴别开来，那么该测试用例同样能检测出原始程序中隐含的故障。这是变异体的一个很重要的性质，下面对它进行详细的定义。设 P′是原始程序 P 的一个变异体，测试用例 t 是程序 P 输入域的一个值。如果对任何 t 下列两个条件同时成立：

$$P'(t) \neq P(t)$$

$$P(t) \neq R(t)$$

则称 P′是一个揭错变异体。其中，R(t)是程序 P 根据功能规格说明书的期望输出结果。

3.3 变异测试的成本

变异测试是一个功能强大的软件测试策略，具有排错能力强、灵活性好的特点。在变异测试的过程中，可能会发现原始程序在一个测试用例上的输出是错误的，而一个变异体却能给出正确的结果，变异体与原始程序的差别可以为

排错提供有用的信息。由变异测试生成的测试用例集在理论上是完备的，但同时变异测试系统也相当耗费时间和资源，这极大程度的阻碍了变异测试在软件产业界的广泛使用。

前面陆陆续续对变异测试的过程做了粗略的描述，但还不够完整。为了找出变异测试中的主要性能瓶颈，从而探索有效的解决办法，达到提高变异测试效率的目的，我们首先系统地描述一个典型的变异测试系统是如何工作的，如图 4 所示。首先，测试人员将原始程序提交给变异系统，由系统根据变异算子生成原始程序的多个变异版本，即变异体。接着，将测试数据输入到系统中作为原始程序和变异体的输入，每个测试用例都要先在原始程序上执行一遍，再由测试人员检验其输出结果是否正确。如果输出结果不正确，则说明在原始程序中检测到一个 bug。因此，在继续使用该测试用例进行未完成的测试之前，需要将该 bug 予以纠正，再重新启动系统。如果原始程序输出结果正确，则将该测试用例在每一个变异体上执行一次。如果变异体的输出结果和原始程序的正确输出结果不一样，则称该测试用例将变异体杀死，并将相应的变异体标记为死亡，同时将他们从活变异体集中删除，后续测试过程中的其它测试用例无需在死亡后的变异体上执行。

在执行完测试集中的所有测试用例后，可以计算出变异分数。如前所述，变异分数是死亡的变异体数与非等价变异体总数之比。测试人员的目标就是要不段增加这个分数，使其接近 1，表明生成的所有变异体都被检测到了。如果一个测试集能将所有的变异体杀死，则该测试集相对这些变异体而言为完备测试集。

如果还有变异体活着，测试人员可以通过增加新的测试用例来改进测试集从而将变异体杀死。有些变异体在功能上和原始程序是等价的，由于这些等价变异体产生的输出结果与原始程序产生的输出结果总是相同的，因此无法将它们杀死。在计算变异分数的时候，注入的变异体总数必须减去等价变异体数。值的一提的是，即使测试人员所使用的测试用例集在变异测试过程中并未发现原始程序中任何故障，但变异分数给我们提供了一个很重要的信息，它揭示了测试的程度。此外，活变异体也揭露出测试用例不够完备。通常情况下，测试

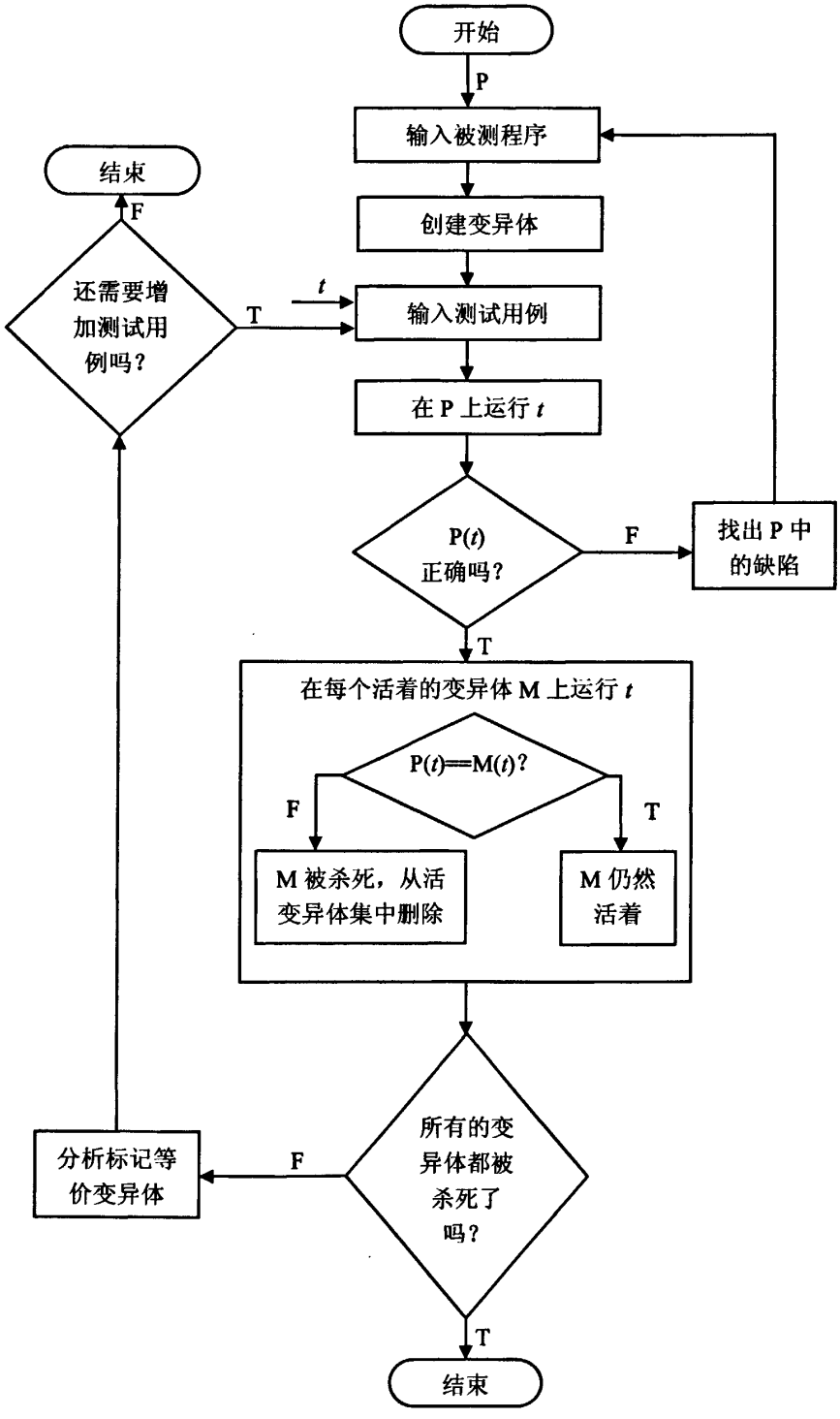
开始
P
输入被测程序
创建变异体
t
输入测试用例
在 P 上运行 t
P(t)
正确吗?
F
找出 P 中
的缺陷
T
在每个活着的变异体 M 上运行 t
P(t)==M(t)?
F
T
M 被杀死，从活
变异体集中删除
M 仍然
活着
所有的变
异体都被
杀死了
吗?
F
分析标记等
价变异体
T
结束
还需要增
加测试用
例吗?
T
F
结束

图 4 变异测试流程图

人员需要针对特定的活变异体设计出相应的测试用例以便将该变异体杀死。在整个变异测试过程中，添加新的测试用例、检查原始程序的输出结果和杀死变异体这几个步骤要反复进行下去，直到测试人员对变异分数感到满意为止或所有的变异体都被杀死了。从图 4 中也可以看出，变异测试过程有两个退出点：

1) 所有变异体都被杀死了，此时，测试人员无需增加新的测试用例，目前的测试数据已经为完备测试集。
2) 虽然有些变异体仍然活着，但变异分数已经达到测试人员所设定的阈值，亦无需增加新的测试用例。

一般来说，前一个退出条件要比后一个退出条件苛刻得多。

Budd 的研究表明，一个软件模块所能产生的变异体数正比于程序中数据对象的数量与数据对象的引用次数之乘积[28]。Offutt 等人做了大量的实验研究，证明了这一比例关系的存在[29]。从上面的描述中也可以看出，变异测试需要运行大量的变异体，每一变异体至少需要执行一个测试用例以便将其杀死，这就需要大量的计算能力。因此，即使是对一小规模的软件模块，也将产生大量的变异体。如图 2 中仅有 4 行的 Min 函数，经过 Mothra 变异系统可以生成 44 个变异体。文献[4]中的一三角形分类程序 Trityp 的代码规模为 30 行，变异后能产生 951 个变异体。由于每个变异体都至少要执行一个测试用例，一般需要运行多个(最多的时候需要运行完测试用例集中的所有测试用例)，变异测试在这个步骤需要巨大的计算开销。正是由于难以接受的巨大计算开销阻碍了变异测试在软件工业界的广泛应用，长期以来，广大研究人员和专家一直围绕降低变异测试的计算开销而展开相关的研究，主要形成了三种策略：

1) 更少，在不损失变异测试中一些有用信息的情况下，寻找有效方法尽量运行更少的变异体程序。
2) 更智能化，将变异测试的整个计算开销分散到多台机器上，或者通过保存变异体运行期间的状态信息来将计算开销分解成几趟执行，或者探索有效方法来避免完全执行变异体。
3) 更快，寻找方法来更快地生成和运行每一个变异体。

3.3.1 选择变异(selective mutation)

第一种策略主要思想是减少生成的变异体数量，有选择变异和取样变异两种方法。不同的程序设计语言有不同的变异算子，不同类别的变异算子所产生的变异体数目是不尽相同的，如 Mothra 系统针对 Fortran 77 程序语言所使用的 22 类变异算子中有 6 类变异算子所产生的变异体数目占到了所有变异体总数的 40%- 60%。从某种意义上说，这 6 类变异体是冗余的，因为杀死由其它变异算子生成的变异体的测试数据在大多数情况下同样可以杀死由这 6 类变异算子生成的变异体。因此，在进行变异测试生成变异体时可以不应用这 6 类变异算子，只应用一些比较关键的变异算子，即通过限制变异算子的方法来达到减少变异体数量的目的[30]。在对程序进行变异时，不应用那些生成变异体数目最多的变异算子，而只是有选择性地应用一些关键性的确实能够生成不同变异体的变异算子，这种近似的方法称为选择变异[31]。

选择性变异可以大幅度提高变异测试的效率，其复杂性与原始程序的规模成正比，而不像传统的变异测试那样与变量引用数的平方成正比。Offutt 等人做了一系列的实验来评估选择变异的效率[31]，方法是通过选择变异测试获得相对选择变异而言完备的测试用例集，然后将该测试用例集输入到传统的变异测试中，再计算出变异分数。表 4 和表 5 是实验的结果[31]，其中表 4 描述了选择变异生成的测试用例集在传统变异测试中获得的变异分数，可以发现选择变异测试基本上达到了传统变异测试的完备性。表 5 描述了选择变异相对与传统变异测试的效率，可以发现由于减少了变异算子，生成的变异体数量大大减少了，随之而来的则是测试效率的明显提升。实验结果还发现，在 Mothra 系统的 22 个变异算子中仅仅只有 5 个变异算子被认为是比较关键的，它们分别为 ABS、AOR、LCR、ROR 和 UOI，应用这 5 个变异算子基本上能达到传统变异测试的完备性，对于小一点的程序至少可以提高 4 倍的运行速度，大一点的程序可以高达 50 倍。因此，选择变异在不损失测试完备性的前提下，节省了计算开销，从而提高了测试效率，是提高变异测试实用性的有效方法之一。

取样变异是指从所生成的变异体中随机选取一定比例的变异体来运行测试用例。Wong 以 5%的递增量随机选取 10%到 40%的变异体进行实验，发现取样

变异在故障检测能力方面比传统变异测试仅仅低 16%左右。在某些要求不高情况下，取样变异也是一种提高变异测试实用性的有效方法。

表 4 选择变异的变异分数

Program	Test Case	Number of Live Mutant	Mutation Score
Banker	57.4	0.0	100.00
Bub	7.2	0.0	100.00
Cal	50.0	0.0	100.00
Euclid	3.8	0.0	100.00
Find	14.6	0.6	99.94
Insert	3.8	0.0	100.00
Mid	25.2	0.0	100.00
Quad	12.2	0.0	100.00
Trityp	44.6	0.2	99.98
Warshall	7.2	0.0	100.00
Average	22.6	0.1	99.99

表 5 选择变异的测试效率

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	1944	29.69
Bub	338	277	18.05
Cal	3010	2472	17.87
Euclid	196	142	27.56
Find	1022	622	39.15
Insert	460	352	23.48
Mid	183	133	27.32
Quad	395	279	22.28
Trityp	951	810	14.83
Warshall	305	259	15.08
Average	9589	7290	23.98

3.3.2 弱变异(weak mutation)

第二种策略主要有弱变异和高性能变异两种方法。弱变异最初是由 Howden 提出的一种近似方法[32]，它的主要思想是判断杀死变异体时比较的是原始程序和变异体的内部状态，即比较点紧接着程序的变异部分之后，只需运行部分语句，不需要执行完整个程序。弱变异有四个比较点：

1) 表达式级比较点(expression weak)，在对变异部分之后最内层表达式第一次求值时进行比较。
2) 语句级比较点(statement weak)，在第一次执行完变异语句之后进行比较。
3) 基本块级比较点 1(basic block weak/1)，在第一次执行完包含变异语句的基本块之后进行比较。
4) 基本块级比较点 2(basic block weak/n)，这种情况主要针对于带有循环的变异，在第一次循环时还无法将变异体杀死。因此，需要在每次执行完包含变异语句的基本块后都进行比较，直到将变异体杀死。

传统的变异测试系统如 Mothra 比较的是原始程序和变异体的最终输出结果，需要完全执行完整个程序，因此，被称为强变异(strong mutation)。弱变异由于避免了完全执行整个程序，从而节省了计算开销，提高了变异测试的效率，是一种改进变异测试实用性的好方法。

弱变异当然也有缺点，有的变异体虽然在程序中某个语句的输出发生错误，但是在程序最后输出时的结果却是正确的。此时变异体在强变异中被认为还活着，而在弱变异中则认为已经被杀死。因此，弱变异最后生成的测试用例集合在测试完备性上不如强变异，弱变异仅仅是满足了必要性条件，而未满足充分性条件。

高性能变异是指利用体系结构先进的高速大型计算机，如向量机、SIMD 机、MIMD 机等，来分担变异测试的整个计算开销[33]。由于每一个变异体是相互独立的，因此，彼此间的通信开销是十分有限的，这使得在多处理器型计算机上进行变异测试成为可能，有利于发挥它们高速的特点。研究成果表明随着处理器数量和程序规模的增加，所获得的加速比会明显得到提高[34, 35]，尤其是随着高性能集群式计算机的飞速发展，为提高变异测试的效率提供了有利的保障。

第四章　测试数据生成

任何软件测试方法中最为关键的一步，也是技术难度最大的一步，就是如何生成能够满足测试标准所需要的测试数据。在变异测试中，手工构造能够杀死变异体的测试用例不仅费时效率低下，而且容易出错，是阻碍变异测试广泛应用的重要因素之一。如果能够自动生成所需的测试用例，无疑将大大提高变异测试的效率。另一方面，如果生成的测试用例能够高效的杀死变异体，则可以节省继续迭代生成测试用例的次数，从而节省了变异测试的计算开销。Offutt等人开发了一种基于约束的测试数据生成方法(constraint-based test data generation, CBT)[36]，可以用来达到上述两个目标。

4.1 杀死变异体的三个条件

CBT 方法的主要思想是采用控制流分析、符号执行和变异体信息来建立约束系统，然后采用域削减、选择域最小的变量随机赋值和回代相结合的方法来求解约束系统生成所需的测试数据。CBT 方法利用了变异测试的概念来自动生成测试用例，这些测试用例是针对变异体而设计的，目的就是有效地杀死变异体。设 M 是对程序 P 的语句 s 变异后的一个变异体，一个测试用例 t 要将变异体 M 杀死，必须满足以下三个条件：

1) 可达性条件 C_r (reachability condition)，以 t 运行 M 时必须能够执行到变异语句 s。变异体是对原始程序做一处词法修改而得到的，变异体其余语句和原始程序是一致的。因而，M 和 P 只在变异语句 s 处不同，如果程序执行不能到达 s，则 t 在 M 和 P 上的运行结果必然相同，t 也就不可能杀死变异体 M。
2) 必要性条件 C_n(necessity condition)，t 必须在到达 s 后使 M 产生一个不同于 P 的状态。若以 t 运行 P 和 M 达到 s 后的状态相同，则 P 和 M 的最终状态必然相同，也就无法将变异体 M 杀死。这是因为 P 和 M 仅在 s 处不同，在 s 之后的语句都相同。
3) 充分性条件 C_s(sufficiency condition)，P 和 M 的最终状态不同。也就是说

P 和 M 在 s 处的不一致状态能够传递到程序的结束处，使得 P 和 M 的最终输出结果不同。

在实际应用中，寻找满足充分性条件的测试数据是十分困难的。因为这需要事先知道程序的完整执行路径。因而，一般都是寻找那些满足必要性条件和可达性条件的测试数据。虽然这样的测试数据不能保证一定将变异体杀死，但实验表明，CBT 方法生成的测试用例集具有较高的变异测试充分度，对大多数程序而言，能杀死近 90%的变异体[37]。

下面简要介绍一下 CBT 方法是如何表示和获得可达性条件和必要性条件。CBT 方法使用路径表达式(path expression)来表示可达性条件 C_r，路径表达式以析取范式的形式来表示到达 s 的多条路径，可以通过分析程序控制流图 CFG 上的谓词表达式来获得路径表达式。图 5 是一个 MinMax 程序和它的一个变异体 M_1，该程序用于计算一个整数数组中的最大元素和最小元素。“Δ”标记的语句为变异后的语句，它是在原程序的语句上使用关系运算符替换(ROR)变异算子将关系运算符“>”用“<”替换得到的。

```
void MinMax ( int a[ ], int n)
{
    int min, max, i;
    min = a[0];
    max = a[0];
    i = 1;
    while ( i < n){
        if ( a[i] < min){
            min = a[i];
        }
        if ( a[i] > max){
Δ       if ( a[i] < max){
            max = a[i];
        }
        i++;
    }
    printf("min is %d, max is %d\n", min, max);
}
```

图 5 程序 MinMax 及其变异体 M_1

在对MinMax程序的程序控制流图进行分析后，可以得到两条从程序的起始语句到达语句*if (a[i] > max)*的路径：*“(i < n) && (a[i] < min)”*和*“(i < size) && (!(a[i] < mi n))”*。故图5中变异体M_1的可达性条件可以用如下的路径表达式来表示：

$$C_r: \quad ((i < n) \&\& (a[i] < min)) \| ((i < n) \&\& (!a[i] < min)))$$

CBT 方法将必要性条表示为以下两种形式：“element1 != element2”和“expression1 != expression2”，并且为各种变异算子建立了必要性条件模板。通过查询必要性条件模板得到变异体 M_1 的必要性条件为：

$$C_n: \quad (a[i] > max) != (a[i] < max)$$

最后，采用域削减、选择域最小的变量随机赋值和回代相结合的技术来求解由可达性条件 C_r 和必要性条件 C_n 构成的数学约束系统以获得所需要的测试数据。

4.2 同位变异体测试数据生成

CBT 方法只考虑了生成杀死单个变异体的测试用例集。事实上，那些在同一位置进行变异得到的变异体(称为同位变异体)具有很多共同的特点，如它们的可达性条件 C_r 相同以及必要性条件 C_n 相似等，可以对杀死这些变异体的条件进行组合，然后生成同时杀死该位置多个变异体的测试数据。这种方法提高了生成测试用例的效率，所生成的测试数据可以同时杀死多个变异体从而节省了测试用例的迭代次数，也就提高了变异测试的效率。在图 5 中的程序语句*“if (a[i] > max)”*上使用 ROR 将“>”用“>=”替换得到另一个变异体 M_2，显然，M_1 和 M_2 为同位变异体。M_1 和 M_2 的可达性条件相同，根据生成变异体时的变异算子查询必要性条件模板得到 M_2 的必要性条件为：

$$C_n': \quad (a[i] > max) != (a[i] >= max)$$

与 M_1 的必要性条件 C_n 相似。只要多个同位变异体的必要性条件不矛盾，就可以将它们组合成一个必要性条件，然后生成满足组合后的必要性条件和共同的可达性条件的测试数据，这些测试数据就有可能同时杀死多个同位变异体。

生成杀死多个同位变异体的测试用例集主要有 3 个步骤：(1) 在生成变异体

的同时，获取它们的可达性条件和必要性条件。(2) 组合同位变异体的必要性条件。(3) 求解由组合后的必要性条件和可达性条件组成的数学约束系统，从而得到所需要的测试数据。以下详细说明如何组合同位变异体的必要条件。

一个变异点能否产生同位变异体是与该点的词汇类型密切相关的，词汇的类型决定了可用的变异算子种类。只有极少数的变异点只能产生唯一一个变异体，程序中的大部分位置，如算术运算符、关系运算符、逻辑运算符、变量、常量、数组名、数组元素下标等，均可以产生同位变异体。能够产生同位变异体的变异点主要有两种类型：

1) 该点允许作用多种变异算子，每种变异算子至少能为之生成一个变异体。例如变量可以作用变量替换(SVR)、用数组元素替换变量(ASR)、用常量替换变量(CSR)、插入绝对值符号(ABS)、插入一元运算符(UOI) 等多种变异算子。表 6 列出了这类变异点。
2) 该点只能作用一种变异算子，但能产生多个变异体。例如二元算术运算符只能作用算术运算符替换(AOR)变异算子，但是由于二元算术运算符种类较多，因而还是可以产生多个变异体。表 7 列出了该类变异点。

表 6 第一类变异点

变异点	变异算子	例子(*int a , c , d[ ]*)
变量	SVR, ASR, CSR, ABS, UOI	*"a + 5"*中的*"a"*可替换为*"c"*, *"d[1]"*, *"6"*, *"abs(a)"*, *"- a "*
常量	SCR, ACR, CRP	*"a + 5"*中的*"5"*可替换为*"a"*, *"d[1]"*, *"20"*
数组元素	AAR, SAR, CAR	*"a + d[2]"*中的*"d[2]"*可替换为*"d[1]"*, *"a"*, *"5"*

表 7 第二类变异点

变异点	变异算子	例子(*int a , b*)
二元算术运算符	AOR	*"a + b"* 中的 "+" 可替换为 "–", "*", "/", "%"
关系运算符	ROR	*"a > b"*中的">"可替换为"<", "<=", ">=", "==", "! ="

在组合同位变异体的必要性条件时，需要注意的是两个同位变异体的必要

性条件之间有可能存在矛盾，因而无法求解出杀死它们的测试用例。这就要求在组合同位变异体的必要性条件时必须检测组合后的必要条件是否矛盾，尽可能地将没有矛盾的必要性条件组合起来。对于第一类变异点和第二类变异点中的二元算术运算符，其多个同位变异体的必要性条件的组合是通过逻辑条件的合取完成的。虽然这种合取不能化简必要性条件，但是可以将原来多个同位变异体的必要性条件组合为一个条件，从而只需要求解一组可达性条件和组合后的必要性条件，而不必分别求解多组可达性条件和必要性条件，从而减少了需要求解的次数。

例如，设 a，c 均为整型变量，d 为整型数组。表达式“$a+5$”中的“a”使用 SVR、ASR、CSR、ABS、UOI 算子分别作用后可变异为“$c+5$”、“$d[1]+5$”、“$6+5$”、“$abs(a)+5$”、“$-a+5$”。对应的必要性条件分别为“$(a+5)!=(c+5)$”，“$(a+5)!=(d[1]+5)$”，“$(a+5)!=(6+5)$”，“$(a+5)!=(abs(a)+5)$”，“$(a+5)!=(-a+5)$”。组合后的必要性条件为“$(((a+5)!=(c+5))\&\&((a+5)!=(d[1]+5))\&\&((a+5)!=(6+5))\&\&((a+5)!=(abs(a)+5))\&\&((a+5)!=(-a+5)))$”，该条件没有矛盾，求解出该条件得到的解将同时满足上述 5 个条件。

对于第二类变异点的 6 个关系运算符，则不能简单地通过逻辑条件的合取来将多个同位变异体的必要性条件进行组合，因为组合后的条件是矛盾的。例如，表达式“$a>b$”作用 ROR 算子后可变异为“$a>=b$”，“$a==b$”，“$a<b$”，“$a<=b$”，和“$a!=b$”，对应的必要性条件分别为：(1)$(a>b)!=(a>=b)$；(2)$(a>b)!=(a==b)$；(3)$(a>b)!=(a<b)$；(4)$(a>b)!=(a<=b)$；(5)$(a>b)!=(a!=b)$。对这 5 个条件进行化简后得到：(1)$a==b$；(2)$a>=b$；(3)$a!=b$；(4)TRUE；(5)$a<b$。这 5 个简化后的条件的合取是矛盾的，但可以对其中不矛盾的部分分别进行组合，并再次进行化简，然后求解可达性条件和组合化简后的必要性条件，仍然可以生成杀死多个同位变异体的测试数据，减少求解次数。例如，对于上述 5 个简化后的条件，由于任意测试数据都满足第 4 个条件 TRUE，所以在进行条件组合时可以不考虑它。在剩下的 4 个条件(1，2，3，5)中，条件 1 与 3 矛盾，条件 2 和 5 矛盾，条件 1 和 5 也矛盾，所以任意 3 个条件的组合都存在

矛盾，因而只能考虑两个条件的组合。由两个条件形成的不矛盾的组合有(1，2)，(2，3)，(3，5)。选择组合(1，2)与(3，5)，因为这两个组合覆盖了剩下的4个条件。对组合(1，2) 与(3，5) 分别再次进行化简得到：(1) $a==b$ 和(2) $a<b$。

采用类似的分析方法，可以得到表达式 $a>=b$、$a==b$、$a!=b$、$a<b$ 和 $a<=b$ 作用 ROR 算子后所产生的变异体的必要性条件组合、化简情况，如表 8 所示。对图 5 中程序的 $a[i]>max$ 作用 ROR 算子得到 5 个同位变异体 $a[i]>=max$、$a[i]==max$、$a[i]!=max$、$a[i]<max$ 和 $a[i]<=max$。这 5 个变异体的必要性条件可以组合为：(1)$a[i]==max$(由 $a[i]==max$ 和 $a[i]>=max$ 组合得到的)；(2)$a[i]<max$(由 $a[i]!=max$ 和 $a[i]<max$ 组合得到的)。它们的可达性条件均为 $i<n$，因此可以得到两组可达性条件和必要性条件约束系统。第一组约束系统的可达性条件为 $i<n$，必要性条件为 $a[i]==max$，求解出这组约束系统得到的测试数据可以同时杀死同位变异体 $a[i]==max$ 和 $a[i]>=max$。第二组约束系统的可达性条件为 $i<n$，必要性条件为 $a[i]<max$，求解出这组约束系统得到的测试数据可以同时杀死同位变异体 $a[i]!=max$ 和 $a[i]<max$。

表 8　ROR 变异算子产生的同位变异体的组合、化简后的必要性条件

表达式	组合、化简后的必要性条件
$a>=b$	$a>b$ 和 $a<b$
$a==b$	$a>b$ 和 $a<b$
$a!=b$	$a>b$ 和 $a<b$
$a<b$	$a==b$ 和 $a>b$
$a<=b$	$a>b$ 和 $a<b$

4.3 实验结果分析

我们在变异测试工具 Proteum 的基础上实现了生成杀死同位变异体测试数据的方法，同时，为了验证上述方法确实能够减小变异测试的测试成本，还专门设计了一组实验，表 9 描述了参与实验的 3 个被测程序。其中有效行数不包括输入输出、{、}和函数声明等语句所在的代码行。该实验是在如下的运行环境中进行的：CPU 为 Pentium®4 2.80 GHz，内存大小为 1.00GB，OS 为 fedora 9。

表 9 三个实验程序

程序名	功能描述	有效行数	变异体数
Bub	对一个整型数组进行冒泡排序	5	36
Mid	返回三个整型数中的中间值	13	180
TriTyp	三角形分类	21	111

图 6 是为了将这些非等价变异体杀死，而由这两种方法生成的测试用例的数目。可以看到，生成杀死同位变异体的测试数据的方法所生成的测试用例数仅仅是原来方法的 20% ~ 40%。其原因在于生成杀死单个变异体的测试数据的方法每次根据一个变异体生成一个测试数据，通常仅能杀死这个变异体。而生成杀死同位变异体的测试数据的方法每次根据相同位置的多个变异体的条件组合得到一个测试数据，有可能同时杀死多个变异体。因而，所需要的测试数据减少了，更少的测试数据意味着执行变异测试时将花费更小的开销。

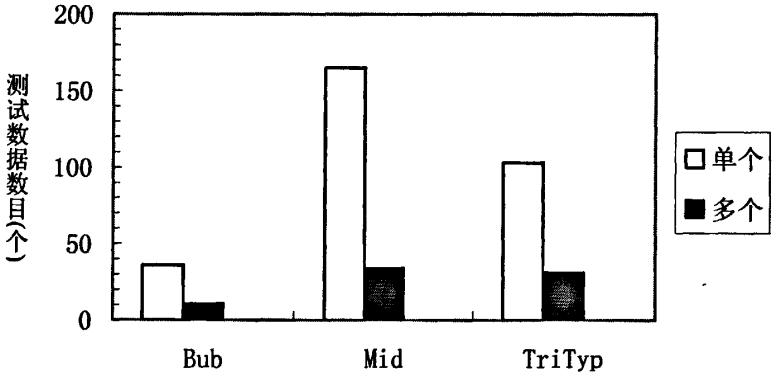

图 6 生成的测试数据的数目比较

第五章　等价变异体的检测

如前所述，变异测试是一种功能强大的软件测试技术，对于确保高可信软件及增加人们对软件的信心有重要的作用，但其运行开销代价太大使得其很难应用到实际的软件测试中。阻碍变异测试广泛应用的主要障碍之一就是等价变异体的检测问题。等价变异体不仅无助于提高变异分数，而且长期滞留在测试过程中将耗费极大的计算开销。因此，应当尽早地将它们检测出来，以提高变异测的效率。此外，手工检测等价变异体不仅费时，而且容易出错，必须有相应的机制来自动检测等价变异体。本文不仅将基于约束测试技术应用于变异测试的测试用例生成，而且将该技术用于等价变异体的检测。

5.1 基于约束测试技术

任何实际测试数据生成技术的最终目的都是根据相应的测试标准从程序的输入域中选择一个它的子集来作为测试数据。这个输入域子集不仅可以帮助测试者建立对被测软件的信心，而且可以发现程序中的许多故障。在变异测试中，测试人员的目标就是找出能够将每一个变异体都杀死的那些测试数据。对于每一个变异体来说，如果一个测试用例能够将该变异体杀死，则该测试用例为一个有效的测试用例，否则就是一个无效的测试用例。一种能够自动生成杀死变异体的测试数据的方法就是通过滤器将那些无效的测试数据过滤掉，从而仅保留有效的测试数据。这样一种过滤器可以由数学约束系统来进行描述。

在阐明如何用约束系统来检测等价变异体之前，先介绍在 CBT 方法中如何表示这种数学约束系统。一个约束的基本组成部分代数表达式(algebraic expression)，而一个代数表达式是由变量、括号和编程语言的运算符组成的。这些表达既可以直接从被测程序中获得，也可以是来自赋值语句的右边部分和判断语句的谓词部分等。一对代数表达式通过条件连接词$\{<, \leq, >, \geq, =, \neq\}$进行连接便构成一个约束(constraint)。而一系列约束通过逻辑连接词 AND($\wedge$)和 OR($\vee$)进行连接则构成一个子句(clause)。如果在进行连接时仅使用逻辑连接词 AND，则称为合取子句，而如果在进行连接时仅使用逻辑连接词 OR，则称为析

取子句。一系列合取子句由逻辑连接词 OR 进行连接构成一个析取范式(DNF)。在 CBT 方法中，我们把所有约束都表示成析取范式形式。例如，(x > 0)表示一个约束，(x > 0) ∧ (y < 0)表示一个合取子句，((x > 0) ∧ (y < 0)) ∨ (z = 0)为一个析取范式。

基于约束测试技术正是利用变异测试原理来自动生成测试用例的。设 M 是对程序 P 的语句 s 变异后的一个变异体，一个测试用例 t 要将变异体 M 杀死，必须满足以下三个条件：可达性条件 C_r；必要性条件 C_n；充分性条件 C_s。上一章对这三个条件做了详细的分析，在此不再重复。设 D 是程序 P 的所有测试用例 t 组成的域，对于每一个变异体来说，D 可以根据上述三个条件进行划分：

1) $D = D_r \cup D_{\bar{r}}$，D_r 是 D 中满足可达性条件的部分，$D_{\bar{r}}$ 是 D 中不满足可达性条件的部分。
2) $D = D_n \cup D_{\bar{n}}$，D_n 是 D 中满足必要性条件的部分，$D_{\bar{n}}$ 是 D 中不满足必要性条件的部分。
3) $D = D_s \cup D_{\bar{s}}$，D_r 是 D 中满足充分性条件的部分，$D_{\bar{r}}$ 是 D 中不满足充分性条件的部分。

根据上面三个条件和子域的相关定义，我们可以得到以下一些结论：

1) 对变异体 M 而言，t 是一个有效的测试用例 $\Leftrightarrow t \in D_s$ 。
2) 对变异体 M 而言，t 是一个有效的测试用例 $\Leftrightarrow t \in D_r \cap D_n$ 。
3) $D_s \subseteq D_r \cap D_n$。

图 6 形象地描绘了这些子域及它们之间的关系。

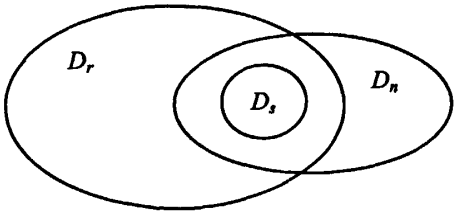

图 7 D_r、D_n 和 D_s 之间的关系

CBT 方法使用路径表达式来表示对应于变异语句的可达性条件 C_r。这种路径表达式是一个代数表达式，如果测试用例能够执行到该变异语句，对应的代数表达式就为真。变异测试中使用变异算子来表示变异体的微小语法改变，满足必要性条件的测试用例必须确保执行完这一微小语法改变后的语句能产生一个不同与原始程序的状态。CBT 方法使用必要性约束来表示必要性条件 C_n，这种必要性约束同样是一个代数表达式。

如图 7 是一个函数 Mid 及其一个变异体，Mid 函数主要用于求三个整型数中大小为中的那个数，其变异体是作用 ROR 算子将关系运算符“<”用“<=”替换得到的。因此，对应于变异语句的路径表达式为$(y < z) \wedge (x >= y)$，必要性约束则为$((x < z) != (x <= z))$。在 CBT 方法中，可以通过满足约束系统来求得到测试用例 t。这些约束系统既可以是可达性约束，必要性约束，也可以是两者的合取。显然，所生成的测试用例 t 满足： $t \in D_r \cup D_n$。

```
void Mid ( int x, int y, int z)
{
    int mid;
    min = z;
    if ( y < z){
        if ( x < y){
            mid = y;
        }else if ( x < z){
Δ       }else if ( x <= z){
            mid = x;
        }
    }else{
        if ( x > y){
            mid = y;
        }else if ( x >= z){
            mid = x;
        }
    }
    return mid;
}
```

图 8 函数 Mid 及其一个变异体

5.2 等价变异体的判别原理

在 3.1.3 节中曾讨论过等价变异体，并且知道等价变异体与原始程序有着相同的功能行为，但并未对“相同的功能行为”给出正式的定义。现在，我们根据输入与输出之间的关系来定义等价变异体。

定义 1：设 P 为原始程序，M 为 P 的一个变异体，则 M 是 P 的一个等价变异体，当且仅当对$\forall t, t \in D$，$P(t) = M(t)$成立。

上述定义表明，如果一个变异体在功能上等价于原始程序，那么我们就找不到任何能够将变异体杀死的测试数据，也即：

$$\neg(\exists t|_{t\in D} \bullet P(t) \neq M(t)) \Leftrightarrow \forall t|_{t\in D} \bullet P(t) = M(t)$$

根据上述等价变异体的定义及前面关于子域的讨论，我们得出判断一个变异体为原始程序的等价变异体的三个非常重要的定理[38]。

定理 1：设 D_r 是对程序 P 的一个变异体 M 而言那些满足可达性条件 C_r 的测试用例组成的域，如果可达性条件 C_r 不可行的，即 D_r 为空集，那么 M 是 P 一个等价变异体。即，$D_r = \phi \Rightarrow$ M 是 P 一个等价变异体。

定理 2：设 D_n 是对程序 P 的一个变异体 M 而言那些满足必要性条件 C_n 的测试用例组成的域，如果必要条件 C_n 不可行的，即 D_n 为空集，那么 M 是 P 一个等价变异体。即，$D_n = \phi \Rightarrow$ M 是 P 一个等价变异体。

定理 3：设 D_r 是对程序 P 的一个变异体 M 而言那些满足充分性条件 C_r 的测试用例组成的域，D_n 是那些满足必要性条件 C_n 的测试用例组成的域，如果 $C_r \wedge C_n$ 不可行的，即 $D_r \cap D_n$ 为空集，那么 M 是 P 一个等价变异体。即，$D_r \cap D_n = \phi \Rightarrow$ M 是 P 一个等价变异体。

由于 CBT 方法使用路径表达是约束系统来表示可达性条件，使用必要性约束系统来表示必要性条件，因此，由上述三个定理可以推出以下这些结论：

1) 如果程序 P 的一个变异体 M 中变异语句处对应的路径表达式约束系统是不可行的，那么能够杀死变异体 M 的测试用例集 D_r 是空的，也即 M

是 P 的一个等价变异体。

2) 如果变异体 M 的必要性约束系统是不可行的，那么能够杀死变异体 M 的测试用例集 D_n 是空的，也即 M 是 P 的一个等价变异体。

3) 如果变异体 M 中由路径表达式约束系统和必要性约束系统合取而组成的约束系统是不可行的，那么能够杀死变异体 M 的测试用例集 $D_r \cap D_n$ 是空的，也即 M 是 P 的一个等价变异体。

上述几个结论均涉及到“不可行”这一概念，如何判断一个约束系统是不可行的呢？如果在一个约束系统中相互之间存在矛盾，那么这个约束系统被认为是不可行的。例如，$(x<0) \wedge (x>0)$这一约束系统中就存在矛盾，因为 x 的取值不可能同时即小于 0 又大于 0。如果变异体 M 以上述约束系统作为路径表达式约束系统，那么变异体 M 就是一个等价变异体。

现在，等价变异体的检测问题已经被转换为在一个数学约束系统中识别是否存在矛盾的问题。对于测试数生成问题而言，是使约束系统得到满足从而生成测试用例。而等价变异体的检测问题，则是通过判别该约束系统是否可行来检测的。

5.3 不可行约束系统识别策略

如上节所述，我们是通过在约束系统中寻找矛盾来检测等价变异体的。令人遗憾的是，判断任意一个约束系统是否不可行是一个不可判定的问题[39]。因此，只能采取特例分析结合启发式方法来识别不可行约束系统。本文主要采用否定、约束分解和常量比较这三种方法来识别不可行约束系统。

5.3.1 否定(negation)

定义 2：约束 C_1 是约束 C_2 的否定，当且仅当它们所描述的域是不重叠的，并且覆盖了 C_1 和 C_2 中约束变量的整个域。

为了能够识别出不可行约束系统，我们更为关注的是前一个条件。因此，我们引入部分否定来将覆盖整个域这一条件进行弱化。

定义 3：约束 C_1 是约束 C_2 的部分否定，当且仅当它们所描述的域是不重叠的，并且没有覆盖了 C_1 和 C_2 中约束变量的整个域。

例如，设 A 为约束($x<1$)，B 为约束($x\geq 1$)，则 A 是 B 的否定，B 也是 A 的否定。这两个约束不可能同时被满足，但约束变量 x 覆盖了 x 的整个域空间。设 A 为约束($x<1$)，B 为约束($x>1$)，则 A 是 B 的部分否定，B 也是 A 的部分否定。因为虽然这两个约束不可能同时被满足，但约束变量 x 没有覆盖到 x 的整个域空间。

定义 4：如果两个约束所描述的域相同，则称这两个约束是语义相同的。

定义 5：如果两个约束所描述的域相同并且包含相同的符号串，则称这两个约束是语法相同的。

显然，如果两个约束是语法相同，那么它们同样也是语义相同的。例如，设 A 为约束($x<0$)，B 为约束($x<1$)，则 A 和 B 是语法相同的，同时也是语义相同的。设 A 为约束($x<0$)，B 为约束($x\leq 1$)，其中 x 为整形约束变量，则 A 和 B 是语义相同，而不是语法相同。

否定是识别不可行约束系统最基本的策略。给定两个约束，首先用否定或部分否定重写其中一个约束，然后比较这两个约束。如果它们是语法相同，那么原先的两个约束之间存在着矛盾，由它们组成的约束系统就是不可行的。从而对应与该不可行约束系统的变异体为一个等价变异体。例如，设 A 为约束($x+y>z$)，B 为约束($x+y\leq z$)，则 A 的否定为($x+y\leq z$)，记为 A'。因为 A' 和 B 是语法相同的，所以 A 和 B 相互矛盾，有 A 和 B 组成的约束系统是不可行的。

5.3.2 约束分解(constraint splitting)

使用 CBT 方法来检测等价变异体时，一个比较常见的情况是：必要性约束形如($x+y>0$)，而路径表达式约束为($x<0$)$\wedge$($y<0$)，使用上一节介绍的否定或部分否定策略无法判定这两个约束是否存在矛盾。

为了解决这一类问题，现采取一种新的策略，即约束分解策略。给定两个约束 C 和 D，将约束 C 分解为两个新的约束 A 和 B，使得 $C\Rightarrow A\vee B$。如果能够证明 A 和 B 分别与 D 存在矛盾，那么 C 和 D 之间也存在矛盾。

由命题逻辑的知识，可以知道：

$$C \Rightarrow A \vee B \quad \Leftrightarrow \quad \neg(A \vee B) \Rightarrow \neg C \quad \Leftrightarrow \quad \neg A \wedge \neg B \Rightarrow \neg C$$

如果能够由 $\neg A \wedge \neg B \Rightarrow \neg C$，$\neg A \wedge \neg B \wedge D$ 推出 $\neg C \wedge D$，则就证明的约束分解的正确性。

证明： $\neg A \wedge \neg B \Rightarrow \neg C$，$\neg A \wedge \neg B \wedge D$ 推出 $\neg C \wedge D$

(1) $\neg A \wedge \neg B \wedge D$　　前提

(2) $\neg A \wedge \neg B$　　I(1)

(3) $\neg A \wedge \neg B \Rightarrow \neg C$　　前提

(4) $\neg C$　　I(2), (3)

(5) D　　I(1)

(6) $\neg C \wedge D$　　I(4), (5) 得证

利用约束分解策略，对一些特例进行了分析，如表 9 所示，分解后得到的新约束 A 和 B 都比原来的约束 C 简单，因而很容易判断 C 和 D 之间是否存在矛盾。

表 10 约束分解特例分析

约束 C		约束 A		约束 B
$(x+y)>0$	$\Rightarrow$	$x>0$	$\vee$	$y>0$
$(x+y)\geq 0$	$\Rightarrow$	$x\geq 0$	$\vee$	$y\geq 0$
$(x+y)<0$	$\Rightarrow$	$x<0$	$\vee$	$y<0$
$(x+y)\leq 0$	$\Rightarrow$	$x\leq 0$	$\vee$	$y\leq 0$
$(x+y)=0$	$\Rightarrow$	$x\leq 0$	$\vee$	$y\leq 0$
$(x+y)\neq 0$	$\Rightarrow$	$x\neq -y$		
$(x-y)>0$	$\Rightarrow$	$x>0$	$\vee$	$y<0$
$(x-y)\geq 0$	$\Rightarrow$	$x\geq 0$	$\vee$	$y\leq 0$
$(x-y)<0$	$\Rightarrow$	$x<0$	$\vee$	$y>0$
$(x-y)\leq 0$	$\Rightarrow$	$x\leq 0$	$\vee$	$y\geq 0$
$(x-y)=0$	$\Rightarrow$	$x\leq 0$	$\vee$	$y\geq 0$
$(x+y)\neq 0$	$\Rightarrow$	$x\neq y$		

5.3.3 常量比较(constant comparison)

第三个用来判断两个约束是否矛盾的策略是常量比较，它主要用来解决那些形如(V rop K)的约束之间的判断，其中 V 为约束变量，rop 为关系运算符，K 为常量。前面介绍的否定或部分否定策略是基于语法相同来判断矛盾的，它无法判断出上述形式的约束之间是否矛盾。例如， 给定两个约束为$(x > 1)$和$(x < 0)$，对第一个约束使用否定或部分否定得到$x \leq 1$ 或$x < 1$，但是两者与$x < 0$之间均不是语法相同，因而就无法判断约束为$(x > 1)$和$(x < 0)$是否矛盾。因此，必须采用常量比较策略。

设 A 为约束(X rop1 K_1)，B 为约束(X rop2 K_2)，通过比较这两个常量和两个关系运算符，可以判断出 A 和 B 是否矛盾。我们把这种策略称为常量比较。表 10 对一些特例进行了分析，第 1、2 列为两个给定的约束，第 3 列一为关于常量 k_1、k_2 的谓词，该谓词用于判断这两个约束是否矛盾，值得注意的是这样的谓词不一定总是存在。最后一列是关于两个约束是否矛盾的结论，其中 pred 表示谓词为真，T 表示这两个约束矛盾，F 表示这两个约束不矛盾。

表 11 常量比较特例分析

约束 A	约束 B	谓词	结论
$x > k_1$	$x > k_2$	—	F
$x > k_1$	$x \geq k_2$	—	F
$x > k_1$	$x < k_2$	$k_1 \geq k_2 - 1$	If pred T, else F
$x > k_1$	$x \leq k_2$	$k_1 \geq k_2$	If pred T, else F
$x > k_1$	$x = k_2$	$k_1 \geq k_2$	If pred T, else F
$x > k_1$	$x \neq k_2$	—	F
$x \geq k_1$	$x > k_2$	—	F
$x \geq k_1$	$x \geq k_2$	—	F
$x \geq k_1$	$x < k_2$	$k_1 \geq k_2$	If pred T, else F
$x \geq k_1$	$x \leq k_2$	$k_1 > k_2$	If pred T, else F
$x \geq k_1$	$x = k_2$	$k_1 > k_2$	If pred T, else F
$x \geq k_1$	$x \neq k_2$	—	F
$x < k_1$	$x > k_2$	$k_1 \leq k_2 + 1$	If pred T, else F
$x < k_1$	$x \geq k_2$	$k_1 \leq k_2$	If pred T, else F
$x < k_1$	$x < k_2$	—	F
$x < k_1$	$x \leq k_2$	—	F
$x < k_1$	$x = k_2$	$k_1 \leq k_2$	If pred T, else F

第六章　结论与展望

6.1 结论

当前，随着普适计算的到来，从小到儿童玩具大到国家安全，计算机系统已经渗透到社会生活的各个角落。因而，人们的日常生活也就越来越依赖于计算机系统，这就使人们不得不对它们提供服务的可信性提出质疑，如何确保软件的高可靠性是我们面临的一个紧迫任务。

本文首先回顾了可信性的起源，介绍了可信性属性和影响可信性的因素，讨论了提高可信性的主要措施。其次，重点介绍了可信性的主要属性之一，即软件可靠性，对传统软件可靠性模型的不足进行了详细地分析，并提出了一种软件综合可靠性模型。该综合模型不仅可以反映出软件的复杂性，而且也考虑到软件测试的测试有效性。本文采用变异测试原理来评估软件测试的测试有效性。

软件测试是保证获得高可靠性高质量软件的主要关键技术手段。任何软件测试方法中最为关键的一步，也是技术难度最大的一步，就是如何生成能够满足测试标准所需要的测试数据。再次，本文详细阐述了变异测试基本原理，仔细分析了变异测试的主要性能瓶颈，探讨了相关的解决方法。变异测试中独有的现象之一是等价变异体的检测问题，它是阻碍变异测试广泛应用的重要原因之一。

本文在最后两章讨论了变异测试中测试数据的生及等价变异体的检测问题，采用基于约束的测试技术(CBT)来解决这两个问题。一般来说，直接使用CBT方法只生成杀死单个变异体的测试数据，但是通过对同位变异体的必要性条件进行适当的组合，便可以生成杀死多个变异体的测试数据，从而提高了变异测试的效率。

6.2 展望

变异测试是一种功能强大的面向故障的单元测试方法，它不仅可以用于检测软件中存在的缺陷，而且可以用评估测试集的有效性。但变异测试巨大的计

算开销降低了它的实用性，因而，长期以来广大研究人员主要围绕如何降低变异测试的计算开销进行研究。本文虽然在从测试数据生成及等价变异体的检测来减小其计算开销方面做了一点研究，但还有许多不足，也是下一步所要研究的，如：

1) 对于位于同一语句块内的程序语句，其变异体的可达性条件相同，不同位置的变异体的必要性条件可以分别进行组合，这些组合后的条件构成一种“与”的关系，满足可达性条件和这些组合后的条件的测试数据将可能杀死更多的变异体。

2) 在进行等价变异体检测时，CBT 方法很难处理含有数组元素的约束系统。

目前，变异测试还远未广泛应用于日常的软件开发中，只是在一些重要的关键场合下才有这种需求，但随着人们对高可靠性软件需求的与日俱增和研究人员对变异测试的深入研究，我们有理由相信在不久的将来，人们将享受到这一技术带来的成果。

参考文献

【1】. 徐拾义. 可信计算系统设计和分析[M]. 北京：清华大学出版社, 2006.07.

【2】. 朱少民. 软件质量保证和管理[M]. 北京：清华大学出版社, 2007.01.

【3】. 陈绍英, 张河涛等译. 软件测试与持续质量改进[M]. 北京：人民邮电出版社, 2008.02.

【4】. A. J. Offutt and S. D. Lee. An Empirical Evaluation of Weak Mutation [J]. IEEE Transactions on Software Engineering, 1994, 20(5): 337 ~ 344.

【5】. R.. A. Demillo, R. J. Lipton and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer [J]. IEEE Transactions on Computer, 1978, 11(4): 34 ~41.

【6】. 姜凡, 郑人杰. 软件测试中弱变异方法与关系测试数据[J]. 计算机学报, 1990, 13(8).

【7】. 徐拾义. 降低软件变异测试复杂性新方法[J]. 上海大学学报(自然科学版), 2007, 13(5): 524 ~ 531.

【8】. 徐仁佐. 软件可靠性工程[M]. 北京：清华大学出版社, 2007.05.

【9】. J. C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology [C]. Proceedings of 25th IEEE International Symposium on Fault-Tolerant Computing, June 27 – 30, 1995: 2 ~ 11.

【10】. A. Avižienis, J. C. Laprie, B. Randell and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing [J]. IEEE Transactions on Dependable and Secure Computing, 2004, 1(1): 11~33.

【11】. S. Y. Xu. On Dependability of Computing Systems [J]. Journal of Computer Science and Technology, 1999, 14(2): 116 ~ 128.

【12】. J. A. Whittaker and J. Voas. Toward a More Reliable Theory of Software Reliability [J]. IEEE Transactions on Computer, 2000, 33(12): 36 ~ 42.

【13】. M. L. Shooman. A Micro Software Reliability Model for Prediction and Test Apportionment [C]. Proceedings of International Symposium on Software Reliability

Engineering, May 17 – 18, 1991: 52 ~ 59.

【14】. M. L. Shooman. Software Reliability Models for Use during Proposal and Early Design Stages [C]. Proceedings of International Symposium on Software Reliability Engineering, June 17 – 18, 1999: 55 ~ 64.

【15】. M. L. Shooman. Reliability of Computer Systems and Network: Fault Tolerance, Analysis and Design [M]. New York: John Wiley & Sons Inc., 2002.

【16】. 朱经纷, 徐拾义. 软件可靠性综合模型的分析和研究[J]. 计算机科学, 2009, 36(4).

【17】. 褚彦明. 软件可靠性中的复杂度评估[D]. 上海：上海大学硕士学位论文, 2008.

【18】. 施银盾. 软件可靠性模型的研究[D]. 上海：上海大学硕士学位论文, 2008.

【19】. A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the Orthogonal [C]. Proceedings of Mutation Testing in the Twentieth and the Twenty First Centuries, October, 2000: 44 ~55.

【20】. A. J. Offutt. Investigation of the Software Testing Coupling Effect [J]. ACM Transactions on Software Engineering Methodology, 1992, 1(1): 3 ~ 18.

【21】. K. S. Wah. Fault Coupling in Finite Bijective Functions [J]. The Journal of Software Testing, Verification and Reliability, 1995, 5(3): 3 ~ 47.

【22】. K. S. Wah. A Theoretical Study of Fault Coupling [J]. The Journal of Software Testing, Verification and Reliability, 2000, 10(3): 3 ~ 46.

【23】. J. H. Andrews, L. C. Brand and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? [C]. Proceedings of the 27th International Conference on Software Engineering, May 15 – 21, 2005: 402 ~ 411.

【24】. R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment [C]. Proceedings of the Second Workshop on Software Testing, Verification and Analysis, July 19 – 21, 1988: 142 ~ 151.

【25】. H. Agrawal, R. A. DeMillo, B. Hathaway, et al. Design of Mutant Operators for the C Programming Language [R]. SERC – TR – 41 – P. West Lafayette: Department of Computer Science, Purdue University, 2006.

【26】. A. T. Acree. On Mutation [D]. Atlanta: Georgia Institute of Technology, 1980.

【27】. T. A. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing [J]. Acta Information, 1982, 18(1): 31 ~ 45.

【28】. T. A. Budd. Mutation Analysis of Program Test Data [D]. New Haven: Yale University, 1980.

【29】. A. J. Offutt, A. Lee, G. Rothermel, et al. An Experimental Determination of Sufficient Mutation Operators [J]. ACM Transactions on Software Engineering Methodology, 1996, 5(4): 99 ~ 118.

【30】. W. E. Wong, M. E. Delamaro, J. C. Maldonado and A. P. Mathur. Constrained Mutation in C Programs [C]. Proceedings of the 8th Brazilian Symposium on Software Engineering, October, 1994: 439 ~ 152.

【31】. A. J. Offutt, G. Rothermel and C. Zapf. An Experimental Evaluation of Selective Mutation [C]. Proceedings of the 5th International Conference on Software Engineering, May, 1993: 100 ~ 107.

【32】. W. E. Howden. Weak Mutation Testing and Completeness of Test Sets [J]. IEEE Transactions on Software Engineering, 1982, 8(7): 371 ~ 379.

【33】. B. Choi and A. P. Mathur. High Performance Mutation Testing [J]. The Journal of Systems and Software, 1993, 20(2): 135 ~152.

【34】. E. W. Krauser, A. P. Mathur and V. Rego. High Performance Testing on SIMD Machines [C]. Proceedings of the 2nc Workshop on Software Testing, Verification and Analysis, July, 1988: 171 ~ 177.

【35】. A. J. Offutt, R. P. Pargas, S. V. Fichter and P. K. Khambekar. Mutation Testing of Software Using a MIMD Computer [C]. Proceedings of the 1992 International Conference on Parallel Processing, August, 1992: 257 ~ 266.

【36】. R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation [J]. IEEE Transaction on Software Engineering, 1991, 17(9): 900 ~ 910.

【37】. R. A. DeMillo and A. J. Offutt. Experimental Results from an Automatic Test Case Generator [J]. ACM Transactions on Software Engineering Methodology, 1993, 2(4):

109 ~ 127.

【38】. A. J. Offutt and J. Pan. Detecting Equivalent Mutants and the Feasible Path Problem [C]. Proceedings of the 1996 Annual Conference on Computer Assurance, June, 1996: 224 ~ 236.

【39】. A. J. Offutt. An Integrated Automatic Test Data Generation System [J]. Journal of Systems Integration, 1991, 1(3): 391 ~ 409.

作者在攻读硕士学位期间公开发表的论文

【1】. 朱经纷, 徐拾义. 软件可靠性综合模型的分析与研究[J]. 计算机科学, 2009, 36(4).

【2】. 朱经纷, 徐拾义. 软硬件测试中预确定距离测试[J]. 计算机科学, 2009, 36(5).

作者在攻读硕士学位期间所作的项目

【1】. 国家自然科学基金资助项目“软硬件可测性设计新途径—软硬件交互式测试及可测性设计研究”(60473033)。

【2】. IEEE 第 15 届环太平洋国际可信计算学术会议策划及投稿系统和会议网站建设。

致　谢

谨以此文献给所有关心、教育、支持和帮助过我的人！

首先，由衷地感谢我的导师徐拾义教授。他传授给我容错计算与测试方面的丰富知识为我的课题研究和论文写作奠定了坚实的基础，他所负责的国家自然科学基金项目为我提供了一个非常好的研究平台。徐老师严谨的治学态度、渊博的学术知识、深厚的理论基础、平易近人的品质和乐观豁达的生活态度给我留下了极其深刻的印象。但更为重要的是，他数十年如一日，始终站在科研最前线，勤勤恳恳做事，扎扎实实做学问，一丝不苟，和时间赛跑的工作品质，时刻都在感染着我，必将对我今后的学习和工作产生巨大的影响，永远是我学习的楷模。在此，我谨向他致以最崇高的敬意和最衷心的感谢。同时也感谢郁松年教授耐心细致地对本论文进行了评阅并提出了许多非常宝贵的修改意见。

其次，感谢同课题组的施银盾和褚彦明等业已毕业的师兄师姐及胡笑颖和徐鹏等师弟师妹在学习上给予我的帮助。通过与他们的探讨和交流，使我受益匪浅，加深了我对本课题的认识，启发了我的思维。

再次，感谢曹林、蔡玉华和薛涛等每一位和我朝夕相处的同学在日常生活和学习中给予的关心和照顾，与他们的每一次热烈讨论都使我的眼界开阔，他们的鼓舞给了我前进的勇气。

还要感谢我的父母和亲人，她们在物资和精神上都给予我很大的支持和鼓励，使我能够顺利完成学业，是她们给予我学习的机会，让我有幸度过了整整二十载的学生生涯，是她们给予我努力学习的信心和力量，是她们让我看到了光明和希望，她们永远是我最坚强的后盾。

最后，再次感谢所有那些关心我、支持我和帮助过我的老师、同学、朋友和亲人。谢谢你们！

变异测试中测试数据生成及等价变异体的检测

作者：朱经纷
学位授予单位：上海大学