摘　　要

软件测试是保证软件质量和软件可靠性的重要手段，但随着软件规模的不断扩大，复杂度的不断提高，以及面向对象程序设计方法和工具的使用，软件测试的难度也进一步加大，测试质量更加难以度量。

软件内建自测试正是针对测试难的问题而提出的，其理论基础是软、硬件测试一致性。它将硬件测试中内建自测试的思想移植到软件测试和软件可测性设计中。内建自测试在硬件测试和可测性设计上已是一个成熟的技术，利用这个成熟的技术到软件测试中去可以大大降低软件测试的复杂度，提高可测性。借鉴硬件内建自测试的设计，软件内建自测试提出在软件开发阶段预先埋入测试信息，这些信息通过程序开发人员与模板交互得到并保存在模板中，然后编写程序自动从模板导出测试用例，自动运行测试用例进行测试。系统基本结构包括测试点、模板和自测试部分，模板中建立了该程序测试需要的数据；测试点负责向模板中写入测试需要的数据；自测试部分根据模板信息生成测试用例；最后由测试程序完成测试功能。

作为该项目的一部分，本论文主要讨论：软件内建自测试系统中模板的设计、测试步骤以及整体构建方法。论文首先研究了模板的概念、模板的内容、模板内容的组织、模板与程序员的交互、模板的管理和模板的参数评估等内容。详细介绍了模板内容的获取，首先分析了 C/C++语言中六类常见的故障，包括：变量未初始化故障、空指针、数组越界故障、内存泄漏、内存操作的未定义故障和编译器本身不足的故障，并介绍了根据故障模型静态检测故障和设计模板函数动态检测故障，给出了检测故障的算法。讨论了软件内建自测试系统中单元测试、集成测试和回归测试，引入关键模块，并对关键模块的量化作了详细的介绍；同时详细介绍了以关键模块为核心的单元测试和集成测试。文中还介绍了软件内建自测试系统的代码规范检测，模板和测试点、测试用例之间的接口以及测试用例和测试点的管理方式，测试用例的生成、运行测试用例并最终生成测试报告等内容。

最后，本文介绍软件内建自测试系统在实际中运用的效果，给出几中故障，然后利用软件内建自测试系统进行检测。说明软件内建自测试系统的可行性，验证了软件内建自测试思想的正确性。

关键字：软件内建自测试，模板，软件故障模型，关键模块，静态测试，动态测试

ABSTRACT

Software testing is not only one of the main methods to ensure the software quality, but also the final opportunity to qualify the software product, so it plays an important role in the life cycle of software development. Traditional software test methods are always to implement the software test independently after finishing the software building through manually generating test cases. With the intensively exploding of the size of software scale and the rocketing of software complexity, these traditional software test methods seem to become time-consuming and costly.

The method of Built-In-Self-Test for software was aiming at reducing the test complexity; it was based on the consistency theory between hardware testing and software testing. The thought of Built-In-Self-Test in hardware testing was applied to software testing and the research field of software design-for-testability. Built-In-Self-Test in hardware testing is already one of the mature technologies, if this method was applied to software testing, we can benefit from it. According to the implementation of Built-In-Self-Test for hardware, useful information for testing was embed in the software project to summit to program tester during programming in Built-In-Self-Test for software. Those information were stored in container called Template. The basic structure includes Template, Checkpoint and Self-testing. All data needed for Self-testing was written to the template by Checkpoint information. Having been set up every data needed for self-testing in the template, the part of self-testing turns these data into test case and test program according to the template information, and then finishes a testing for the function.

As a part of this project, the design of Template, the process of testing and construction of system in this thesis was discussed. Firstly, this paper focuses on the content, content organizing, content obtainment, accessing, management, parameters of template and interaction between programmer and Template. Secondly, there are six fault models were analyzed in C/C++ programming language, which is uninitialized variable, null pointer, out of bounds array, memory leak, undefined operation in memory, self-defect of compiler. According to fault model, static testing method and dynamic testing method were proposed. Thirdly, this paper discussed the recognition of kernel module and the unit testing, integrated testing and regression testing by the kernel module in detail. Furthermore, code criterion, interface among Template, Checkpoint and Test Case, management of Checkpoint by Template, testing reports was discussed in this paper.

At last, as an example, a concrete implementation is also given in this paper which demonstrates how to instrument the checkpoint and template function in source code and how to test in

Built-In-Self-Test for software step by step. As a result, those faults could be efficiently detected by the system of Built-In-Self-Test for software. So we draw a conclusion that the method of Built-In-Self-Test for software is feasible in some extent.

Keywords: Built-in-Self-Test for Software, Template, Software Fault Model, Kernel Module, Static Testing, Dynamic Testing

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，论文中不包含其它人已发表或撰写过的研究成果。参与同一工作的其它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：杨成涛 日 期 2006.3.22

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：杨成涛 导师签名：[illegible] 日期：_____

第一章 绪 论

本章主要介绍软件测试的概念、测试方法的分类、测试步骤以及软件测试的发展和研究现状；同时还介绍了硬件测试和软硬件可测性设计等中的基本概念。本章所采用的概念和定义也适用于本文以后的各个章节。

1.1 引言

信息技术的飞速发展，使软件产品应用到社会的各个领域，软件产品的质量自然成为人们共同关注的焦点。不论软件的生产者还是软件的使用者，均生存在竞争的环境中，软件开发商为了占有市场，必须把产品质量作为企业的重要目标之一，以免在激烈的竞争中被淘汰出局。用户为了保证自己业务的顺利完成，当然希望选用优质的软件。质量不佳的软件产品不仅会使开发商的维护费用和用户的使用成本大幅增加，还可能产生其它的责任风险，造成公司信誉下降，继而冲击股票市场。在一些关键应用(如民航订票系统、银行结算系统、证券交易系统、自动飞行控制软件、军事防御和核电站安全控制系统等)中使用质量有问题的软件，还可能造成灾难性的后果。

软件质量问题由来已久，软件危机曾经是软件界甚至整个计算机界最热门的话题。早在 60 年代前后，人们刚开始开发较大型软件时，就从几个大型软件项目的失败中意识到了软件危机的存在。为了克服软件危机，1968 年，北大西洋公约组织(North Atlantic Treaty Organization－NATO)的学术会议提出了“软件工程”的概念。之后的 30 年是人们与软件危机进行不懈斗争的 30 年，也是软件工程不断发展的 30 年[1]。

为了解决这场危机，软件从业人员、专家和学者做出了大量的努力。现在人们已经逐步认识到所谓的软件危机实际上仅是一种状况，那就是软件中有错误，正是这些错误导致了软件开发在成本、进度和质量上的失控。有错是软件的属性，而且是无法改变的，因为软件是由人来完成的，所有由人做的工作都不会是完美无缺的。问题在于我们如何去避免错误的产生和消除已经产生的错误，使程序中的错误密度达到尽可能低的程度。

人们曾经认为更好的程序语言可以使我们摆脱这些困扰，这推动了程序设计语言的发展，更多的语言开始流行，为了使程序更易于理解开发了结构化程序设计语言，如 PL/1，PASCAL 等；为了解决实时多任务需求开发了结构化多任务程序设计语言，如 Modula，Ada 等；为了提高重用性开发了面向对象的程序设计语言，如 Simlasa 等；为了避免产生不正确的需求理解，开发形式化描述语言，如 HAL/S 等，

这使得建立基于自然语言的描述成为可能，人们以形式化语言来描述需求；为了支持大型数据库应用，开发了可视化工具，如 Visual Studio, Power Builder 等。程序语言对提高软件生产效率起到了一定的积极作用，但它对整个软件质量尤其是可靠性的影响，与其它因素相比作用较小。

可能是因为程序语言基于严格的语法和语义规则，人们企图用形式化证明方法来证明程序的正确性。将程序当作数学对象来看待，从数学意义上证明程序是正确的是可能的。数学家对形式化证明方法最有兴趣，在论文上谈起来非常吸引人，但实际价值却非常有限，因为形式化证明方法只有在代码写出来之后才能使用，这显然太迟了，而且对于大的程序证明起来非常困难。

在过去的四分之一世纪里，软件生产并未沿着形式化方法所描绘的方向发展。相反，被形式化方法所抛弃的、以软件测试为中心的软件质量保障技术在软件生产实践中得到了迅速发展，软件测试已成为软件生产中必不可少的质量保障手段。

现在，人们普遍认识到，对软件产品进行测试是保证软件产品质量、提高产品可靠性的重要手段。据统计，国外软件开发机构 40%的工作量是花在软件测试上，软件开发费用的近½用于软件测试。对于一些要求高可靠、高安全的软件，如飞行控制或核反应监控等软件，测试费用可能相当于软件工程所有其它步骤费用总和的 3~5 倍。由此可见，要成功地开发出高质量的软件产品，必须重视并加强软件测试工作[1]。

1.2 软件测试概述

1.2.1 软件测试的概念

软件测试作为保障软件质量关键的技术越来越受到人们的重视。那么究竟什么是软件测试？

1983 年 IEEE 提出的软件工程标准术语中给软件测试下的定义是："使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别"[2]。G. J. Myers 则持另外的观点，他认为："软件测试是为了发现错误而执行程序的过程" [3]。这两个测试定义分别侧重于"检验软件是否满足需求"和"寻找软件中的错误"，我们认为这两点都是软件测试的主要目标。目前，关于软件测试的普遍接受的定义为：软件测试是发现并指出软件(包含软件经过建模、需求、设计等阶段所产生的大量输出工作)中存在缺陷的过程，这个过程指明和标注问题存在的正确位置，详细记录导致问题出现的操作步骤，及时存储当时的错误状态，以上组合在一起便于测试后问题能够准确再现。因而，可以说软件测试在软件开发中的地位仍然是其它质量保证手段所不能代替的。软件测试在软

件生存周期中占有非常突出的重要地位，是保证软件质量的重要手段。

1.2.2 软件测试方法的分类

1. 从是否需要执行被测软件的角度可分为静态测试和动态测试

静态测试(Static Testing)又叫静态分析，其主要特征是不利用计算机运行被测试的程序，而是采用其它手段达到检验的目的。但上述静态方法的特征并不意味着完全不利用计算机作为分析的工具。常用的一些静态分析方法有如下几种：

- 收集一些程序信息，以利于查找程序中的各种欠缺和可疑的程序结构；
- 从程序中提出语义的或结构要点，供进一步分析；
- 以符号代替数值求得程序的结果，便于对程序进行运算规律的检验；
- 对程序进行一些处理，为进一步动态分析做准备；

动态测试(Dynamic Testing)与静态测试相反，在测试时要运行被测程序，然后根据运行结果来检验程序。这种方法的关键在于如何得到测试数据和期望的正确输出，黑盒测试和白盒测试是两种常用的动态测试方法。

2. 从是否针对系统的内部结构和具体实现算法的角度可分为黑盒测试和白盒测试

黑盒测试(Black-box Testing)又称功能测试、数据驱动测试或基于规格说明的测试(Specification-based Testing)。用这种测试方法进行测试时，被测程序被当作打不开的黑盒，因而无法了解其内部构造。在完全不考虑程序内部结构和内部特性的情况下，测试者只知道程序输入和输出之间的关系，或是程序的功能。他必须依靠能够反映这一关系和程序功能的需求规格说明书考虑确定测试用例，和推断测试结果的正确性。即所依据的只能是程序的外部特性。因此，黑盒测试是从用户观点出发的测试。

白盒测试(White-box Testing)又称结构测试、逻辑驱动测试或基于程序的测试(Program-based Testing)。采用这一测试方法，测试者可以看到被测的源程序，他可以分析程序的内部构造，并且根据其内部构造设计测试用例。这时测试者可以完全不顾程序的功能。

这两类测试方法是从完全不同的起点出发，并且是两个完全对立的出发点，可以说反映了事物的两个极端。两类方法各有侧重，在测试的实践中都是有效和实用的。在进行单元测试时大都采用白盒测试，而在确认测试或系统测试中大都采用黑盒测试。

1.2.3 软件测试策略

在一般情况下，软件测试过程和整个软件开发过程是平行进行的。测试计划早在需求分析阶段即应产生。软件测试策略是指测试过程中各阶段的不同特点，制定

测试目标，分步骤地进行测试。每个测试步骤在逻辑上是前一个步骤地继续。一般各种测试方法在实现测试策略上可以分为下列几个步骤[2]：

1. 单元测试(Unit Testing)：单元测试是对最小软件开发单元(程序模块、功能模块)的测试。单元测试的测试重点是程序的内部结构，主要使用白盒测试方法，也要使用黑盒测试。单元测试一般由开发人员负责，单元测试必须是可重复的，在代码修改、升级、产品维护等阶段，所有的测试都必须在整个系统的生命周期中进行维护。

2. 集成测试(Integrated Testing)：集成测试包括子系统测试和系统测试，它是在单元测试的基础上将所有模块按照设计要求组装成系统或子系统，对模块组装过程和组装接口进行正确性检查。它主要是将各个模块以增量的方法集成在一起测试，其测试依据主要是需求规约和设计文档。集成测试主要关注下面问题：把各个模块连接起来的时候，穿越模块接口的数据是否会丢失、一个模块的功能组合起来是否会对另一个模块的功能产生不利的影响、各个子功能模块组合起来，能否达到预期要求的父功能、全局数据结构是否有问题、单个模块的误差积累起来，是否会放大，从而达不到能接收的程度、是否满足需求、功能实现是否正确等。总而言之，单元测试关注的是每个模块的功能，而集成测试关注的是整体流程，模块之间的接口，数据一致性。

3. 回归测试(Regression Testing)：对某些已经测试过的测试集合重新进行测试看是否经过修改的程序产生了新的 bug。

4. 确认测试(Verification Testing)：确认测试也称有效性测试，如果加入用户信息，也称验收测试。它的任务是验证软件的功能和性能，以及其特性是否与用户的要求一致。

5. 系统测试(System Testing)：系统测试是将软件系统与硬件环境、网络环境等集成在一起进行测试。

6. 平行运行：它是指同时运行新开发出来的系统和将被它取代的旧系统，以便比较新旧两个系统的处理结果。它的优点在于：可以让新系统在准生产环境中运行而不冒风险；用户能有一段熟悉新系统的时间；可以验证用户指南和使用手册之类的文档的正确性。

软件测试的过程流程可用图 1-1 表示：

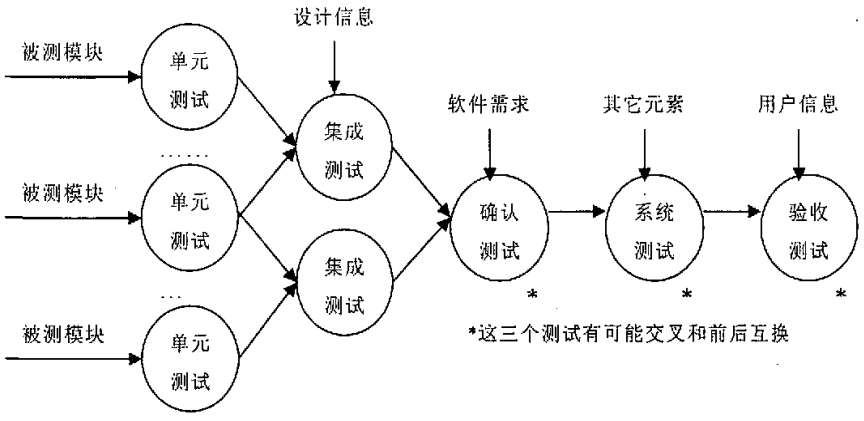


图 1-1 软件测试的过程流程

1.2.4 软件测试的发展

70 年代以来，软件工作者开始加深对测试工作的认识，到了 70 年代中期，软件测试技术的研究达到高潮。J. B. Goodenough 和 S. L. Gerhart 首先提出了软件测试的理论，从而把软件测试这一实践性很强的学科提高到理论的高度，被认为是测试技术发展过程中具有开创性的工作。此后不久，著名测试专家 W. E. Howden 指出了上述理论的缺陷，并进行了新的开创性工作。以后，又有 Weyuker 和 Ostrand，Geller，以及 Gerhart 进一步总结原有的测试理论并进一步加以完善，使软件测试成为有理论指导的实践性学科。

在软件测试的理论迅速发展的同时，各种高级的软件测试方法也将软件测试技术提高到了初期的原始方法无法比较的高度。J. C. Huang 提出了程序插装(Program Instrumentation)的概念，使被测程序在保持原有逻辑完整性的基底上，插入"探针"，以便获取程序的控制流和数据流信息，并可得到测试的覆盖率。W. E. Howden 对测试路径进行了深入的分析，提出了系统功能测试及代数测试等概念。W. E. Howden、L. A. Clarke 和 J. A. Darringer 等人把符号执行的概念引入到软件测试中，提出了符号测试方法，并且建立了 DISSET 等符号测试系统。Woodward 和 Hedley 等人分析了路径测试中的藕联效应假设，并以此为基础发展出基于程序变异(Program Mutation)的测试方法，使传统的测试技术领域增加了新的成员——错误驱动测试。接着 T. A. Budd 和 F. Sayward 等人进一步发展了程序变异的思想，论述了能采用其它测试方法的地方基本上都能使用程序变异方法进行测试，并且开发了变异测试系统的原型。后来，W. E. Howden 把 DeMillo 等人的变异方法称为强变异，依据原始的变异思想提出了弱变异的测试技术。使用弱变异方法无需生成被测程序的变异因子，而是分析程序中易于出错的部分进行变异测试。在此之后，S. J. Zeil 提出了一种新的程序模型，把程序描述为环境的改变，通过扰动减少错误空间的维数，这就

是与程序变异相似的程序扰动测试方法(Permutation of Program Statements)。1977 年 L. Osterweil 和 L. D. Fosdick 等人首先引入了数据流测试方法，该方法通过对数据流的静态测试找出程序中潜藏的错误。Osteerweil 还把这一方面推广到并发程序的数据流分析。1983 年日本学者 Ryoichi Hosoya 等在数据流测试方法中加入了变量值域分析，使数据流方法检测的错误类型更多。

如何确定测试数据，选取测试点，仍然是实施测试，提高测试效率和错误命中率的关键问题。L. White 和 E. Cohen 提出了一种新的计算机程序测试策略，这就是域测试方法(Domain Testing Strategy)。输入域分析是把程序的输入按谓词划分，进一步以此划分为依据，给出各个域的测试点。E. J. Weyuker 和 T. J. Ostrand 接着发展了他们的方法。1985 年 D. J. Richardson 和 L. A. Clarke 在此基础上又提出了划分分析的概念，将形式化规格说明和程序本身的输入变量的取值范围都划分为域，找出共同的域，通过在这些共同域上的测试找出规格说明和程序之间的不一致性。L. A. Clarke 还深入讨论并改进了 White 和 Cohen 提出的域测试方法。此外，S. Redwine Jr 总结了一套工程化测试方法。

近年来，人们针对应用于各种专门场合的各类软件，也研究了一些比较专业化的测试方法，这些测试方法一般是在传统测试方法的基础上根据各类软件的不同特点改进而成的，较常见的有图形用户界面(Graphic User Interface－GUI)测试、实时系统测试、分布式系统测试以及面向对象软件测试等。

软件测试的另一重要发展方向就是软件测试的自动化。自动化测试可以减少软件测试中人力的投入、缩短测试时间、降低测试成本并提高测试效率，因此测试工具的研制和开发一直是人们关注的重点并且已初见成效，如我们在前面介绍过的每一种测试方法几乎都有相应的测试工具。但由于软件测试的复杂性和不确定性，到目前为止，还尚未出现真正高效和实用的系统。

另外，目前较热的的方向之一是人们开始尝试将人工智能技术运用于软件测试中，并取得了初步的成果，常用的方法有[4]：

- 遗传算法：遗传算法(Genetic Algorithm—GA)是一种基于自然选择原理和自然遗传机制的通用搜索算法，与其它搜索算法不同的是，它在整个搜索空间随机采样，按一定的评估策略对每一样本评价，并采用特定的遗传算子进行样本优化，直至产生最优解。遗传算法现在主要被用在路径覆盖测试中，为一定的程序路径产生测试数据。其一般过程是：首先随机地在程序输入空间中产生数量较大的数据(种群)；用这些数据驱动被测程序的运行；根据运行时返回的信息，以及一定的评价策略(如路径覆盖率)对每一个体的适应度进行评估，保留那些适应度较高的个体，并通过遗传算子(选择，交义，突变)的操作改变数据值，形成新一

代更优种群；如此往复，直至达到一定的目标(如达到100%的覆盖率等)。

- 神经网络：人工神经网络模拟人脑的结构和功能，具有自适应、自组织和自学习能力，可通过训练样本，根据周围环境和变化的信息来调整自身结构。在软件测试中，神经网络主要用于测试用例的精化。具体作法是：从测试用例中抽取一种计量器(Metrics)，该计量器用于计量测试用例的长度、函数调用频率以及参数使用频率等；将此计量器(输入模式)和测试用例所检查出的错误等级(输出模式)作为样本对神经网络进行训练；训练成的神经网络便能对新产生的测试用例检查错误的能力进行评估，并据此将测试用例分成不同的等级。这样，在实际测试前就可以剔除那些不能检查出错误而强调那些最能揭露出错误的测试用例，从而提高测试的效能，降低测试成本。
- 规划求解：某些软件的测试用例和人工智能中的规划具有相似性，二者都是为达到某一目的的命令序列，同时也都必须遵循这些命令的句法要求和命令间的语义交互。因此规划求解可被用在这样的测试用例生成中。

  为生成规划，一个规划求解系统必须获得：

  (1) 对操作的描述(即参数、前置条件和后置条件)；

  (2) 初始状态；

  (3) 目标状态。当一个软件的行为特性可以用状态机进行建模时(如面向对象的软件、界面类软件等等)，它也就具备了上述这些信息，其规划求解(测试用例生成)的一般步骤为：确定一个目标状态;找到以这个目标状态为后置条件的操作，将其前置条件包含在目标列表中；重复以上步骤直到不存在未决的目标或所有未决的目标都可由初态满足。

1.2.5 软件测试的现状

目前，随着我国信息化系统的广泛应用，以及软件产业的迅猛发展，软件测试技术受到越来越多的关注，企业对软件产品测试的需求愈来愈迫切，要求也越来越高。软件评测对软件产品质量保障，提升产品市场竞争能力的作用得到业界的重视和肯定。一般软件开发的总成本中，用在测试上的开销要占30%到50%。极端情况下，例如在关系到人的生命安全的软件中，测试费用可能相当软件生存周期所有其它阶段费用总和的3～5倍。据美国工业界的统计，对商品化的顺序程序来说，测试在时间和费用两方面的花费都要占整个软件开发周期总开销的50%左右。据统计大约有60%的错误是在设计阶段之前注入的，并且修正一个软件错误所需的费用将随着软件生存期的进展而上升。错误发现得越晚，修复它的费用就越高，而且呈指数增长的趋势。

与此同时，软件测试在全球的发展是不平衡的，在软件产业比较发达的国家和地区，软件测试也已经成为很大的一个产业，但是在中国，可能还算不上一个真正的产业，目前正处于快速发展阶段。尽管软件测试技术有了长足的进步，但总的来说，仍然和软件开发实践提出的要求有相当大的距离。测试手段的进展也远远没有达到令人满意的程度。随着软件开发技术的不断发展，面向对象技术、软件重用技术以及网络和 Internet 的广泛应用等都对软件测试技术提出了新的挑战。软件工程的规模在无限膨胀，软件的复杂度也在迅速增加，直接造成了测试(包括对系统软、硬件的测试和验证等)成本的直线上升。当前国外和国内的测试专家们对软件测试和软件可靠性问题的严重性越来越重视。

1.3 硬件测试概述

随着超大规模集成电路的发展以及"纳米"时代的到来，越来越多的新技术应用到生产中。集成电路中平均每个芯片 I/O 管脚上集成的门数从几个增加到几百个，较大的 VLSI(Very Large Scale Integrated)电路和 ASIC(Application Specific Integrated Circuit)不仅包含随机逻辑，同时也包含 RAM、ROM、PLA、数据通道、多路转换器和像微处理器那样复杂的宏单元，电路变得日益复杂。电路能否正常工作成为一个突出的问题。一方面，电路中一个微小的故障会导致电路不正常工作，甚至发生大灾难；另一方面，检测电路能否正常工作变得越来越困难，测试费用也变得越来越难以接受。

目前最有影响且最有效的组合电路测试生成算法有 PODEM 算法、FAN 算法、SOCRATES 算法、EST 算法、基于传递闭包的 TRANS 算法以及 Kunz 等人采用的递归学习算法。近年来时序电路测试生成较有影响的算法有 BACK、ESSENTIAL、HITEC、FASTEST、GENTEST 以及 CONTEST 等。降低测试代价的方法主要有并行测试及可测性设计。并行测试生成是采用并行计算机系统来处理测试生成问题。最主要的并行测试生成策略有模拟并行、启发式知识并行、搜索并行以及故障并行。可测性设计主要分为特殊的方法与结构设计。结构设计主要是指扫描设计，扫描设计分为完全扫描与部分扫描。

对于集成电路的研究开发人员和制造商来说，总是希望尽可能早地发现产品的故障，因为要检测相同的故障在不同层次所需要的代价不同，层次越高代价越高。通常认为要检测相同的故障在门级(Gate Level)、芯片级(Chip Level)、板级(Board Level)、系统级(System Level)和域级(Field Level)的测试代价依次以 10 倍增长，而且随着电路 I/O 管脚及时钟频率的增加成指数增长。测试生成时间已成为产品设计周期内最长的阶段，测试时间大约占据了整个产品设计与生产总时间的 40%，同时产品投入市场的时间延后半年会导致产品利润降低 33%。

1.4 可测试性设计

1.4.1 硬件可测性设计

随着 VLSI 和 ASIC 电路规模的不断增大，电路结构和功能的日益复杂化，功能越来越强大，使得通过外部测试设备测试芯片和多芯片模块变得十分复杂和困难，测试生成的费用也呈指数增长。单凭改进和研究测试生成方法，或者单纯从测试设备上解决集成电路的测试问题，已不能很好地满足 VLSI 电路，特别是 ASIC 的测试需要。解决 IC 测试问题的根本方法是在做系统设计时就充分考虑到测试的要求，即在设计阶段就开始考虑如何对电路进行测试，并将一些实用的可测性技术引入到芯片设计中，以降低测试生成的复杂性，最有效的方法就是进行可测性设计(Design for Testability—DFT)。

边界扫描设计[5](Boundary Scan Design－BSD)和内建自测试[6](Built-in Self Test－BIST)是目前可测性设计的两种主要技术。边界扫描设计的主要思路是将电路的时序行为转化为组合逻辑来处理。电路中的所有扫描触发器连接成一个扫描链，测试输入通过扫描输入经扫描链移入到所有扫描触发器的输出。这样，所有扫描触发器的输出可看作是原始输入。测试响应可以由扫描链移出到扫描输出。这样，所有扫描触发器的输入可以看作是原始输出。扫描设计可以提高测试覆盖率，但会增加触发器和锁存器的复杂程度，需要增加一个或多个 I/O 管脚，测试时间一般也会延长。

BIST 技术通过电路内部的测试码生成器(Test Pattern Generators—TPG)产生加到被测电路(Circuit Under Test—CUT)的测试码，并由输出响应分析器(Output Response Analyzer—ORA)完成输出响应分析，产生“PASS/FAIL”信号，以确定该电路有无故障。这样，芯片不但能完成自身的逻辑功能，在适当的控制下还能进行自我测试分析，周期性地监测芯片。与扫描技术相比，BIST 技术具有测试速度快，适合于层次化设计等优点。随着电路的复杂性与密度的日益增加、设计层次的提高，BIST 被认为是测试集成电路较有效的方法。BIST 测试电路板如图 1-2 所示:

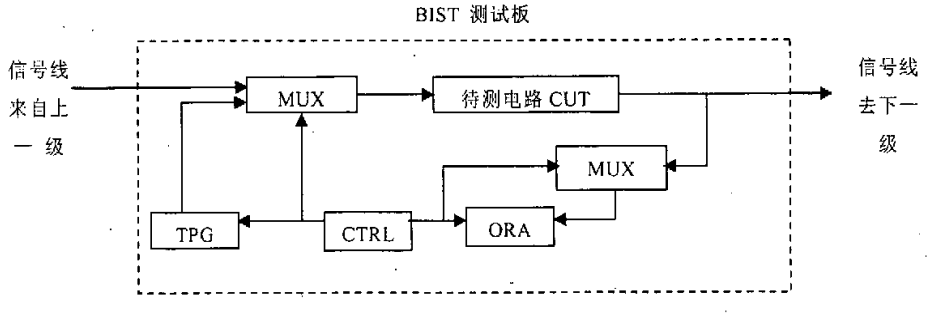


图 1- 2 BIST 测试电路板

其中：

- TPG：测试模式生成器，负责产生伪随机的测试序列；
- ORA：输出响应分析器，负责对测试信号经过待测电路后的输出响应进行分析；
- CUT：待测电路；
- MUX：多路选择器；
- CTRL：控制器。

1.4.2 软件可测试性设计

随着软件工程技术的不断发展，软件的可测性逐渐成为软件开发、测试及可靠性评估过程中需要考虑的一个重要指标。软件可测性是在软件中存在错误的情况下，该错误引起软件失效概率的一种估计，它是针对软件测试工作难易程度进行衡量的一个标准，对于指导测试资源的分配、软件测试程度等起着重要的作用。1990 年 IEEE 颁布软件工程术语集[7]从 2 个方面对软件的可测性进行了定义，这个定义从测试工作达到一定测试标准的难易程度对程序的可测性进行度量。

随着软件工程技术的不断发展，研究人员对软件的可测性赋予了新的含义，软件可测性着重考虑软件系统的语义，关心的是系统中包含错误的情况下系统的行为。J. Voas 在文献[8]中对软件可测性给出了新的解释，即在一个程序中存在着缺陷的情况下，在一定的输入分布下，随机选择测试用例时，这个故障被发现的概率，形式化的表示为：

Testability=P {reject | prob. distribution of inputs, faulty}

按照这个定义，如果程序中存在错误，在测试过程中，这个错误容易被发现，表示这个程序的可测性高，否则，称这个程序的可测性低。

可测试性跟可理解性、可修改性一起成为决定软件可维护性的基本因素，表明了发现程序错误的容易程度，可测性研究目前主要侧重于研究程序本身代码结构和语义，以及当软件包含缺陷时的行为方式，定性/定量的分析它产生故障的可能性，进而用一定的方式在设计软件时提高软件的可测试性。对合理划分资源，提高测试效率，软件项目进度和费用的预期估计都会发生很大的影响。

软件可测性的具体含义主要包括[9]：

- 可操作性。“运行得越好，被测试的效率越高。”
- 可观察性。“你所看见的就是你所测试的。”
- 可控制性。“对软件的控制越好，测试越能够被自动执行与优化。”
- 可分解性。“通过控制测试范围，能够更快地分解问题，执行更灵巧的再测试。”

- 简单性。“需要测试的内容越少，测试的速度越快。”
- 稳定性。“改变越少，对测试的破坏越小。”
- 易理解性。“得到的信息越多，进行的测试越灵巧。”

测试作为目前用来验证软件是否能够完成所期望功能的唯一有效的方法，在其实施过程中，普遍存在着软件测试难的问题。普遍意义上的软件测试都是在程序已经编制完成后再进行实施的。同硬件测试类似，相对于复杂的软件内部结构，用于测试的输入和输出也是十分有限的。因此，软件测试研究者提出了软件可测性设计的概念，即在软件设计时考虑测试的问题。当今国内外测试专家已经充分认识到软件可测性设计的重要性和必要性，而且达到了一致共识，认为“软件的可测性设计肯定是解决软件测试难的重要思想和途径”。它的本质则是通过增加极少量的软件设计复杂性的代码，将易于软件测试的原则融合到软件设计中去，既保证了软件的可靠性，又大大地降低了软件的开发和测试成本同时也加速了软件的开发周期。

1.5 论文的组织

本文由组成八章组成。

本章首先介绍了软件测试、硬件测试、可测性设计等有关概念；

第二章论述了软件内建自测试的来由；

第三和第四章介绍了软件内建自测试系统中模块的概念、作用以及如何生成模板；

第五章介绍了软件内建自测试系统以关键模块为核心的测试步骤；

第六章介绍了软件内建自测试系统的整体运行框架，介绍了各个模块的接口；

第七章给出几个典型的实例在软件内建自测试系统运行的效果，指出软件内建自测试系统检测故障的能力；

第八章总结与展望。

1.6 本章小结

本章首先介绍了软件测试的概念、发展、现状。并介绍了硬件测试和软硬件可测性的概念，指出了软硬件可测性设计是解决目前软硬件测试难的重要思想和途径，阐述了论文将要用到的概念和背景知识。

第二章 软件内建自测试

本章主要介绍了软件内建自测试的来由，软件内建自测试系统的整体框架和本文的研究内容。

2.1 软件内建自测试的提出

软件测试是一个 NP(NonPolynomial)难题，软件测试的难点确实较多。比如，人们可以使用较为简洁的方法计算出一个硬件系统中的可能出现故障(如 Stuck-at 型)的总数，然而我们却难以估计出一个软件(即使是较小的应用软件)中可能存在的故障数。这就对软件测试造成极大的困难。其难点主要有：

- 不彻底性：不论是黑盒测试还是白盒测试，由于测试输入数据数量巨大，都不可能进行彻底的测试。
- 复杂性：无法估计错误总数，难以设计有效的测试用例和预测预期结果。
- 成本高：软件生存期中工作量最大和费用消耗最大的环节。

随着计算机在各个领域的应用越来越广泛。一方面，软件规模和复杂度的迅速膨胀，软件测试的难度也越来越大。另一方面，人们对计算机的依赖程度也越来越强，从而对其可靠性的要求也就越来越高，而软件的可靠性在很大程度上取决于测试的质量，这就给测试提出了更高的要求。

然而，在测试领域中，由于发展不平衡，软件测试技术的研究和开发又明显地落后于硬件测试。因此，当前国外和国内的测试专家们对软件测试和软件可靠性问题的严重性越来越重视。笔者的导师徐拾义教授在担任 IEEE 第八届亚洲测试会议(IEEE The Eighth Asian Test Symposium)程序主席期间，以及近年来在担任多次 IEEE Computer Society, TTTC 主办的国际计算机会议的程序委员时，在审稿和选稿中发现了一些很有新意的论文。这说明软件测试研究的"重点攻关"已经启动。

虽然，软件测试的研究起步较晚，成果还不够成熟，但是有一点对测试工作者倒是一大启发，即，我们完全应该而且可以利用硬件测试中成熟的技术和思想来为软件测试服务。换一句话说，在硬件测试中的许多思想和技术是可以完全移植或者部分移植到软件测试上来的。这样在软件测试的研究中可以少走许多弯路，加速软件测试研究的步伐。

软件内建自测试(BIST for Software)就是在这种情况下由笔者的导师徐拾义教授提出的软件可测性设计研究领域中的一个创新概念。它的基本思想是将软件(主要是针对一般的应用软件而言)可测试性设计与软件测试结合起来以解决软件"测试难"

的问题。BIST 技术在硬件的可测性设计上已是一个比较成熟的技术[10]。但是至今尚未见到把该技术运用到软件可测试设计中来。软件的可测性设计可以大大地降低软件的测试难度和复杂度。事实上，这是一个完全可以"借用"的软件测试技术。

2.2 软件内建自测试思想的基础

软件内建自测试思想之所以能够借鉴硬件内建自测试的思想，本质上讲软件测试与硬件测试具有一致性。虽然软硬件测试由于不同的特性很明显地存在巨大差异，然而我们研究发现这两种测试之间还是存在共性的，即软硬件测试存在一致性。软硬件测试的一致性并不是指行为级硬件的验证和软件测试的相似性，而是指门级硬件测试和软件测试的一致。

2.2.1 软硬件测试模型的一致性

对于软硬件测试，一个好的测试用例是应该能较容易地发现缺陷。它们都是NP难题，即测试规模随着软硬件规模的增长呈指数级增长。一些硬件故障模型可以对应到软件故障中去，而软件故障模型也可以对应到硬件故障模型。比如：

- 硬件测试中的振荡型故障类似于软件测试中的死循环；
- 考虑时间因素的跃迁故障的测试则类似于实时软件系统的测试[11]；
- 两类测试的可靠性增长模型也是相似的。软件可靠性增长模型定义软件可靠性$R(T)$为在$[0,T]$时段内无故障发生的概率$e^{-Z(T)}$，即$R(T)=e^{-Z(T)}$，其中$Z(T)$为失效率函数[12, 13]。而我们知道，硬件的可靠性增长符合负指数分布$e^{-\lambda}$，其中λ是失效率，这和软件测试中的可靠性增长模型是相似的。

2.2.2 软硬件测试方法的一致性

尽管软硬件测试存在一致性，但是几乎没有软硬件测试方法是有意识地通过彼此借鉴得来的。而事实上一些相似的测试方法在软硬件测试中已得到无意识的应用。本节就列举这样几种方法来说明软硬件测试的一致性的存在。

1. 随机测试

由于软硬件测试都是NP完全问题，因而穷举测试是不可行的。但是在软硬件测试中随机测试仍然非常有用，尤其是在测试初期，因为随机测试十分自然并且避免了要考虑测试用例的生成。特别是在黑盒测试中，由于内部执行细节不可知，随机测试有时候会成为最佳选择。但是随机测试的致命弱点是测试效率会随着测试的深入变得越来越低，在测试后期会不可避免地挑选已经用过的输入，并有可能发现前面已经发现了的故障。因而在实际应用中，通常采取反随机、伪随机或相似的测试方法[14]。这些方法一般定义Hamming或Cartesian最大距离从而能够选择尽可能不同

的输入发现不同的故障[15]。

对于软件测试，输入可以是数值、字符和数据结构。它们应编码为二进制并且这些二进制序列能解码为实际的输入，从而能够采用随机测试，这和硬件测试是不同的。

2. 扫描和程序切片

对于VLSI，我们常常需要在设计阶段就考虑测试问题，这一般可通过扫描技术和BIST技术实现。扫描技术是通过可扫描的触发器代替时序电路中的一般触发器，测试时这些触发器连接起来成为一个移位寄存器。此时，时序电路可看作由一个移位寄存器和纯组合电路组成。对于移位寄存器，电路的所有状态可通过初始输入和初始输出控制和观察，因而可以判断哪一个触发器有问题。而组合电路的测试又要比时序电路的测试容易得多。因此，通过扫描技术可以简化时序电路的测试[16, 17]。

抽取相似或相关组件的思想在软件测试中也有应用，这就是Mark Weiser提出的程序切片[18]。程序切片的标准是一个值对(V, n)，V是一个变量，n是一个程序点。当下一个要执行的语句在程序点n时，原程序中所有不会影响变量V的语句和谓词都除去，剩下的语句谓词便构成了源程序的(V, n)切片。

程序切片最初用于调试，它可以追踪引起错误值的语句，减少调试员要考虑的代码量。而在软件测试中，程序切片主要用于回归测试。在进行错误纠正时因为不能保证所作的修改不会引起新的错误，因而需要进行回归测试。回归测试中，程序切片的概念被扩张成了分解切片，它只关注对一给定的变量进行的计算，也就是说和语句行无关[19]。

程序切片只关注会引起问题的部分，而不是无关紧要的部分。扫描技术则将一个复杂而困难的问题分两个相对容易的问题。但它们实现的方式是一样的，即通过抽取相关部件来简化问题。

2.3 软件内建自测试的基本思想及实施框架

软件内建自测试的基本思想是：首先为软件设计人员提供一套预先设计好的模板，这套模板要求软件设计人员在设计软件时必须满足模板中提出的相关条件，并在模板中输入所要求的有关数据。由模板对所编写的程序进行"包装"(包括设置测试点[20]，生成测试用例[21]，结果比较等)。即，模板中建立了测试该程序需要的数据以供自测试部分使用。然后，由自测试部分根据模板中信息生成一定的测试用例，并在测试点上进行比较和测试，以完成基本的测试功能。软件内建自测试所需要的额外负担是十分有限的。对软件设计人员来说，只需在设计软件时和设计完成后按模板的要求将相关数据输入至模板中即可，这样当程序完成及调试结束后其内建测试的功能就已经自动地生成了。

软件 BIST 系统(指：软件内建自测试系统，以下皆同)的主要模块包括：

模板[22]：是为软件自测试项目而生成的一个信息库，其中存有软件自测试所需的软件配置，测试配置，测试工具配置的有关信息，是用户同软件内建自测试系统交互的接口。

测试点[20, 23]：在被测程序中的某些特定位置插便于测试和观测的代码，提取关键信息，读入测试用例，输出动态测试信息，以达到提高程序可测性的目的。测试点根据功能不同又分为控制点和观察点。

自测试部分[24]：根据模板中提供的信息和程序运行时从测试点得到的信息，自动生成测试用例，运行测试用例并比较测试结果。

测试报告生成部分：根据自测试部分运行结果生成测试报告。

整个的软件内建自测试周期分为两个阶段：软件开发阶段和软件测试阶段。在软件开发阶段的工作主要包括以下几点：

- 在开发软件时，引入模板库(以动态连接库的形式提供)；
- 根据程序的种类(如：科学计算程序)在模板库中选择合适的模板；
- 根据需求说明书、系统详细说明书和在系统的详细设计阶段的程序流程图或形式化语言写成的伪代码，通过扫描伪代码或流程图来得到程序的初始逻辑结构，并将其保存在模板中；
- 选择测试点的设置策略(设置条件)，在程序员编程时，软件内建自测试系统会在源程序中满足测试点设置条件的地方提示设置测试点，程序员可以自己决定是否在该位置设置测试点。

在软件测试阶段的工作主要包括下面几点：

- 使用 LEX[25]&YACC[26]来扫描源程序，获得源程序详细的流程图和变量链表，将得到的信息保存到软件 BIST 系统的模板数据库中。并将得到的流程图进行二叉化，以便在生成测试用例时可以使用二叉化基本路径测试法[21]，使得生成的测试用例的路径覆盖率更高，而且生成测试用例过程更容易；变量链表的作用是为了跟踪被测程序中的变量，方便生成测试用例和检测故障而引入的；
- 根据测试目的和测试要求，并结合测试点设置策略，进一步设置测试点；并将测试点设置信息保存在软件 BIST 系统的模板数据库中；
- 根据保存在数据库中的程序流程图的信息、模板中所提供的基于软件故障模型的信息和被测程序的需求分析和详细设计报告的信息来利用软件内建自测试系统的自测试部分来生成测试用例，并将测试用例保存在模板数据库中；

- 在软件内建自测试系统中运行测试用例，并在运行程序的过程中，收集变量信息，并将所得到的信息写入模板数据库中；
- 根据从测试点得到的信息及变量链表的信息和其它测试结果并结合模板的功能来分析测试结果，生成测试报告；
- 也可以根据软件内建自测试提供的静态测试工具来对源程序进行静态分析和测试。

整个系统构成图如下：

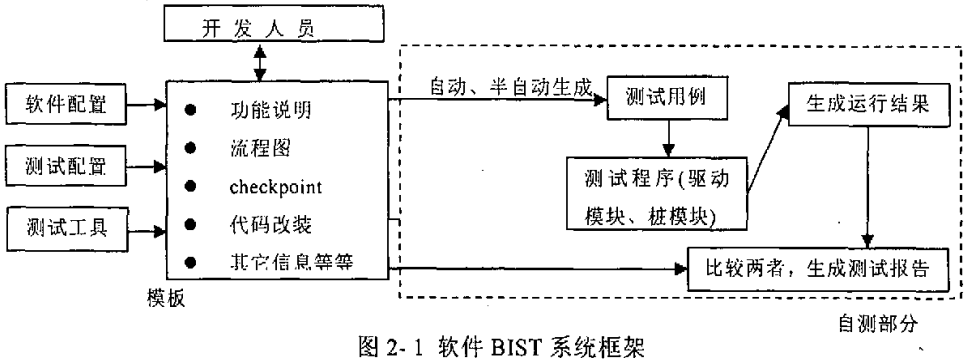


图 2-1 软件 BIST 系统框架

2.4 本文的研究内容

本文所研究的内容属于导师**徐拾义教授**领导的**国家自然基金项目"软件可测性设计新概念—Built-in Self Testing For Software"**中的一部分。主要负责上述模块中的模板以及测试程序生成。

模板是软件 BIST 系统的核心模块之一，它直接影响着生成测试用例的好坏，以及运行结果与预期结果的比较效果等。测试程序的生成是进行自测试部分的关键，同时还要给用户提供一份合理的测试报告。

本文主要包括两个部分：第一部分是针对不同的故障模型如何对模板进行有效的设计和管理；第二部分是软件 BIST 系统中以关键模块为核心的测试。

2.5 本章小结

本章首先介绍了软、硬件测试存在的一致性，在此背景下提出软、硬件一致性思想的一种应用－软件内建自测试。软件内建自测试是一种新的测试思想，它主张在软件开发阶段提前考虑后期的软件测试，期望通过与程序员合理交互，提前得到测试信息存入模板，测试时根据模板信息自动生成测试用例及测试程序，最终实现测试自动化。最后介绍了本文的研究内容。

第三章 软件内建自测试中的模板

本章主要讨论软件 BIST 系统中的核心内容之一：模板。主要讨论了模板的概念、由来、作用、如何获取模板内容、以及模板内容的组织和模板的管理，最后还讨论了模板的评价参数。

3.1 模板的提出及其作用

在软件 BIST 系统中，可测性设计将通过模板来实现。因此，模板既是开发人员与软件 BIST 系统之间交互的桥梁，也是软件 BIST 系统的基础和核心。

3.1.1 与模板相关的概念

软件故障模型(Software Fault Model)[27]：是一些基本故障的组合，这些故障模型是在对以前测试经验的积累、分析的基础上得到的，它可以对以后的编程和测试工作进行指导。建立故障模型的好处是使用模型中给出的信息就可以设计和编制算法，用来产生可以运行的测试用例。软件故障模型的种类有很多种，典型的软件故障模型有空指针、数组越界、变量未初始化和内存泄露等。

模板规则集[28]：是一组与某一类软件测试有关的一组规则集合，这组规则集合可以覆盖该类软件测试的较大部分的故障，并能在程序员编程时提供指导，以尽量减少错误。在程序测试时，可以根据这些规则集来生成测试用例，以完成程序测试。

模板库：在软件 BIST 系统中，是一个为各类程序而设计的模板的集合。

模板库管理系统：是软件 BIST 系统的一部分，主要提供对模板内容访问的控制。主要包括模板的查找、模板的管理、模板的更新、测试用例的存储和复用、程序流程图的存储、测试点的设置策略及其存储、模板版本管理、访问模板库的权限管理等内容。引入模板库管理系统是为了便于复用和管理。

测试的目的是以最少的时间和人力，系统地找出软件中潜在的各种错误和缺陷。如果成功地实施了测试，就能够发现软件中的错误。测试的附带收获是，它能够证明软件的功能和性能是否与需求说明相符合。实施测试收集到的测试结果数据为可靠性分析提供了依据。测试不能表明软件中不存在错误，它只能说明软件中存在错误。软件测试可以看作是对软件错误空间的搜索，即要在数以百万计甚至更多的输入及其状态组合中，尽可能寻找更多的错误的状态及其组合。这种搜索不是漫无目的地乱搜索，而应是系统的、集中的、自动的。

在软件 BIST 项目的模板设计中，我们可以利用故障模型(指：软件故障模型，

以下皆同)作为建立模板规则集的基础。通过对现有的故障模型的分析、整理并结合某一类软件的特点和对程序员经常所犯错误的统计，我们就可以建立针对某一类软件的模板规则集。然后根据模板规则集，可以编制算法、建立模板数据结构并最终建立模板。然后在软件开发人员开发软件的过程中，利用模板所提供的信息指导程序员编程，并在源程序中插入便于测试的代码，如测试点、模板函数等。这样程序员在编写程序时就可以避免很多常犯的错误，并且在测试时也能达到较好的效果。

3.1.2 模板的作用

在第二章中我们讨论了软、硬件测试的一致性并将在硬件测试中成熟的内建自测试思想引入到了软件测试中。但由于软、硬件之间存在的巨大差异，不能完全照搬硬件内建自测试的那一套方法。必须对硬件内建自测试的思想结合软件的特点进行改造。而模板就是实现软件内建自测试思想的一个核心概念。可以从下面几个不同的角度来理解软件 BIST 系统中模板的作用：

- 从软件开发人员角度：模板就是一套规则，指导和规范软件开发人员在编写程序的过程中预先存储有用的测试信息(程序的输入，程序的预期输出和程序流程图等)。并提示软件开发人员在程序中可能出错的地方预先插入测试代码，这在一定程度上也起到了警示作用，使软件开发人员可以注意到在程序中可能出错的地方。
- 从软件测试人员角度：使用模板不仅仅可以检测故障，而且可以生成测试用例。测试人员通过预先设定好的测试用例提取规则从模板中提取信息生成测试用例。
- 从软件工程的角度：模板是联系软件开发过程中编码和测试的桥梁，它以软件开发人员在软件开发过程中需额外的存储一些测试信息为代价，来达到大大减轻软件测试人员测试程序时生成测试用例的负担，并且大大缩减了软件测试周期，提高了测试的故障覆盖率。
- 从测试管理的角度：模板提供了一种测试策略和专业测试人员测试经验的复用。模板库相当于一个知识库，它具有不断学习和更新的能力。模板库提供了软件 BIST 系统的核心。模板库管理系统本身也包含了测试管理系统。

3.2 模板设计的原则

总原则[29]：易于建立，易于检测故障，易于生成测试用例和易于复用，达到高故障覆盖率和较小测试时间开销的目的。

易于建立：模板的设计涉及到故障模型的收集、模板规则集的生成、模板数据结构的定义等一系列工作。而且还要参照软件需求说明书、详细说明书等资料。在

软件开发人员编程时，可能还需要其以交互方式输入一些信息。这些工作将会很繁重，如果再加上每个模板本身设计也很复杂的话，那么设计所花费的代价就会太大，使工作难以开展。

易于检测故障：模板的设计是以故障模型为基础的，在测试时检测出故障应该模板的首要标准之一。

易于生成测试用例：模板里存储的测试信息正是为了自动生成测试用例而预先由程序员埋入的，因此模板的设计应易于导出测试用例。

易于复用：模板的设计需要花费一定的时间和资金，其作为软件开发组织的无形资产应该可以稍加修改或不加修改的用到其它同类软件测试中或生成新的模板。这可以大大的提高模板的利用率和可操作性。

3.3 模板内容的获取

3.3.1 模板内容获取的方式

在本文的前面章节部分，我们简单的讨论了模板的一些基本概念。在这一部分，将详细的讨论作为软件 BIST 系统核心的模板所需要存储的内容，和怎样将这些模板的内容表示为可以具体操作的数据结构和算法。而这些数据结构和算法(具体实现为函数)将可以以测试点的形式被插装到源程序中，从而得到生成测试用例所需要的必要信息。

一个模板往往用来测试一类程序。首先模板必须来源于实践，既可以是经过大量数据测试并且证明有效的测试模式，也可是经过软件项目验证的测试流程；另一方面模板必须通过理论证明，必须是基于故障模型的测试模型。另外，必须认识到，虽然在实际应用中可以将软件按照特定的标准划分为不同的类型，某一种类型的程序可以使用相同或相近的模板来进行测试。但是，即使是同类的软件，也可能由于使用环境、开发人员的素质、应用平台等条件的不同而呈现出不同的故障分布和故障类型。而且，从测试准确性的角度来讲，模板中的信息越详尽越好。但往往由于经费、时间、人员等条件的制约，模板是不可能涵盖所有的软件故障。因此，模板内容的详尽程度是由软件开发中的各种因素的综合考虑而决定的。

在软件 BIST 系统中，将模板内容的获取可以看作是与软件开发过程平行进行的一个子周期。可以简单的以软件开发周期的编码阶段为分界点，将模板的内容获取分为两个阶段，详细信息如图 3-1 所示：

- 在编码阶段以前获得模板的内容；
- 在软件开发周期的编码阶段后，即从已经经过编译、调试的可以运行的源代码中获得模板的内容。

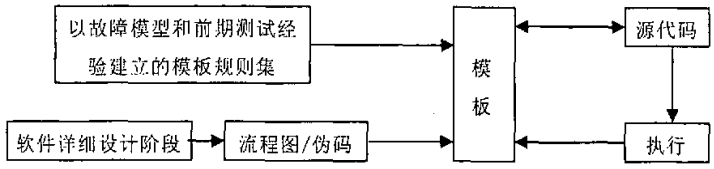


图 3-1 模板内容的获取

下面就详细的讨论关于模板内容获取的这两个方面的内容。

3.3.2 在编码阶段前填充模板

3.3.2.1 根据编程语言的性质得到模板的内容

在编码阶段前可以获取的用来生成模板的信息主要来自两个方面：一是查找资料，并分析、总结出故障模型，由故障模型来生成模板。二是根据待开发软件的需求分析报告和详细设计报告，得到待开发软件内部的大致细节，从而生成模板。这主要是侧重于待开发软件的个性方面。从这两个方面得到的内容而生成的初始模板都可以在编程阶段为程序员提供帮助。

目前，软件 BIST 项目研究的程序类型主要集中在以 C/C++语言编写的科学计算类程序。C/C++语言是目前广泛使用的编程语言，具有比其它编程语言更大的灵活性。适用于 C/C++语言的测试模板也可以很容易的移植到其它语言中。如没有特别指明，本文中的测试模板和代码片断都是用 C/C++语言描述的。

故障模型是模板理论的基础。一个成熟的故障模型必须具备以下的条件[30]：

- 该模型是符合实际的，大多数系统中存在的故障都可以用这种模型来表示；
- 模型下的故障个数是可以容忍的，模型下的故障个数一般和系统的规模成线性关系；
- 模型下的故障是可以测试的，也就是说，存在一个算法，利用该算法可以检测模型中的每一个故障。

从目前所存在的软件测试方法中，其测试理论的基础并不是基于故障模型，而是基于软件中所有可能的故障。其优点是：能够检测软件中的大部分故障、自动化程度较高。缺点是：对小概率故障的检测效果不好，而许多小概率故障往往可能导致系统崩溃；一次性运行不发生异常的故障用这种方法是检测不出来的；难以计算故障的检测效果。

在软件内建自测试的思想中，充分考虑了常规软件测试方法和面向故障模型的测试方法之间的优缺点，并将两者有机的结合起来。在其中用模板来实现软件故障模型的概念，用自测试部分来实现测试的自动化，并用基准程序来度量测试的故障覆盖率。

3.3.2.2 根据需求分析和详细设计得到模板的内容

需求分析报告是经过用户确认的关于软件系统所必须具备的功能和数据方面的基本要求。从需求分析报告中可以得到软件系统所应具备的基本功能，这些基本功能是进行系统测试和集成测试的基础。所以，可以从需求分析报告中得到关于集成测试和系统测试的测试用例。

软件系统的详细设计说明书是软件系统详细设计阶段所产生的文档，其中包括了关于系统由那些模块组成、模块间的调用关系、模块所具备的基本功能、用程序流程图或伪代码表示的程序(模块)内部逻辑结构等信息。通过详细设计说明书，可以使开发人员在总体上把握整个系统的内部逻辑结构和自己所要开发的那个模块所应具备的基本功能要求。通过分析详细设计说明书，可以获得程序员要在编码阶段所开发模块的一些基本信息(如：模块调用关系，用于开发桩模块和驱动模块；基本程序流程图，用于模块的白盒测试；模块基本功能要求，用于进行黑盒单元测试等)，并将这些信息存入模板数据库中(关于模板数据库的详细内容将在下一节进行详细的讨论)。在这里要说明的一点是，这里存入模板数据库的信息是在编码以前根据详细设计说明书得到的一些基本的信息，在程序员编码完成以后，可能会对这些存入在模板数据库中的基本信息要进行一定的修改和细化，这时的修改和细化一般不需要程序员以交互方式进行，而是利用 LEX&YACC 对源程序进行词法和语法分析而得到源程序的更详细的逻辑信息。

3.3.3 在编码阶段后填充模板

在编码阶段后，由于可以得到详细的源代码，这时，就可以通过扫描源代码得到程序的详细的逻辑结构等信息。具体的信息有以下几个方面。

3.3.3.1 程序的控制流程图

通过对源程序的词法分析、语法分析，可以得到程序的控制流图。在过程控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。包括以下 5 个方面：

- 程序的独立路径：指程序中至少引进一个新的处理语句集合或一个新条件的任一路径，采用程序流图的术语，即独立路径必须至少包含一条在定义路径之前不曾用到的边；
- 程序环路复杂性：McCabe 复杂性度量。从程序的环路复杂性可导出程序基本路径集合中的独立路径条数，这是确定程序中每个可执行语句至少执行依次所必须的测试用例数目的上界；
- 导出测试用例；
- 准备测试用例，确保基本路径集中的每一条路径的执行；

- 图形矩阵：是在基本路径测试中起辅助作用的软件工具，利用它可以实现自动地确定一个基本路径集。

3.3.3.2 模块调用关系图

测试一个模块时需要编写一个驱动模块和若干个桩模块，如图 3-2 所示。驱动模块的功能是向被测试模块提供测试数据，驱动被测模块，并从被测模块中接受测试结果。桩模块的功能是模拟被测模块所调用的子模块，它接受被测模块的调用，检验调用参数，模拟被调用的子模块功能，把结果送回给被测模块。

在软件 BIST 系统中，通过使用 LEX&YACC 对源程序进行语法和词法扫描，可以很方便的得到模块调用关系图。可以将得到的函数调用关系图保存在模板数据库中，关于模板数据库的详细结构将在下一节进行讨论。

根据上述保存在模板数据库中的程序流程图和模块调用关系图，我们就可以根据模板数据库提供的内容建立起被测程序的详细的逻辑结构，并根据被测程序的逻辑结构、测试策略和插装策略，自动/半自动的生成测试用例和将检测函数插入到源程序中需要设置测试点的地方，并最终跟踪和检测检测函数的返回值等信息来确定被测程序是否达到预先设置的测试目标或是否实现其应该实现的功能。

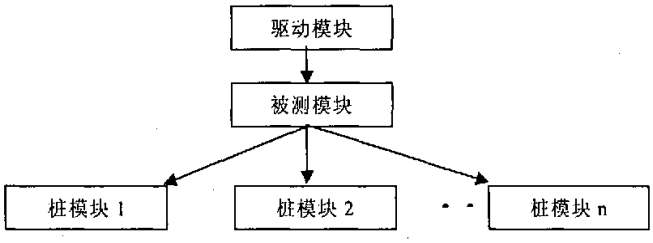


图 3-2 驱动模块和桩模块图

3.3.3.3 变量列表

对于每个程序，变量是程序的关键。在程序运行的时候，出错的原因很大一部分是因为变量的声明、赋值、比较和判断等操作错误导致程序出错，甚至进入死循环。一些致命性的错误可以导致整个系统的崩溃。

因此我们要对程序中各个模块内部的变量进行观测和分析，以及在各个模块之间传递变量的观测和分析。对于被测程序中变量的分析包括两个部分：一是通过 LEX&YACC 静态扫描被测程序，得到变量的个数、类型和在函数调用时实参和形参的对应关系等。特别是实参和形参的对应关系，这对于在动态测试被测程序时的变量跟踪至关重要。二是动态执行被测程序，在程序执行时跟踪变量执行情况，生成变量链表。

3.4 模板内容的组织及其管理

3.4.1 模板内容的组织

在软件 BIST 系统中的需要根据故障模型来生成模板，不同的故障需要不同的故障模型，在软件中不同的故障造成的危害以及出现的概率也不尽相同，这是就是需要合理的组织模板的内容，以便于在模板能够快速的包装待测程序。在软件 BIST 系统中把针对故障检测的称为模板函数(模板函数的内容、设计形式以及分类在下面一章详细介绍)，模板函数主要是按照不同类型(如字符串、输入输出、网络通信等)、不同的使用频率(使用频率高的放在高层，可以让不同的类型继承)等进行组织。这样即有利于模板的管理也减少了运行的代价，在使用的时候可以根据程序员的输入方便的插入函数。借鉴其它集成环境中库函数的组织，我们根据以下规则来组织模板函数：

- 即常见错误(如：未初始化变量、空指针等)的模板函数优先存储；
- 针对编译器的增强、改进的模板函数优先存储；
- 模板函数的组织按照类型(如字符串、输入输出、网络通信等)进行；
- 当规则冲突或插入函数符合多个规则时，以存储方便优先考虑。

3.4.2 模板内容的管理

在 3.3 节中我们讨论了怎样得到模板中的内容，包括在软件开发周期编码阶段前模板内容的获取和编码阶段后模板内容的获取。两者的区别在于是否已经有经过调试后可执行的源代码。在单元测试中，源代码是我们测试的重点，减少源代码中错误的方法主要有两种，一种是在程序员编写程序时提供指导(所谓指导，在软件 BIST 系统中包括两层含义，一是程序员编程时给予提示，提醒程序员注意某类错误。第二层是在可能出现错误的地方插入测试点检测代码，使程序在测试时通过分析检测代码的输出可以确定源代码是否存在某种错误)，使程序员尽量减少犯一些该编程语言或该类程序的一些典型错误(如 C/C++语言编程中的未初始化变量、空指针等)。二是在程序员编写完源代码后，进入到单元测试阶段。这时就可以通过使用 LEX&YACC 对源代码进行静态扫描和动态的执行，就可以得到详细的程序流程图、变量列表、函数调用关系和被测程序运行时的变量链表等信息。通过这些信息，并结合一定的测试策略(如路径覆盖、等价类划分、因果图等)就可以生成测试用例，来测试被测程序。模板所起的作用就是提供一种机制，可以集成这些信息来方便测试，并在编程时向程序员提供必要的指导。本节主要是讨论怎样将 3.3 节中得到的模板的内容存储起来，使其可以在开发人员和测试人员及其它需要知道这些信息的

人员之间共享和方便的使用。

在软件 BIST 系统中，模板是为某一类程序的测试所服务的，而且，现在软件 BIST 项目研究的重点是单元测试和集成测试，但可以很容易的将软件内建自测试的概念扩展到系统测试中。所以，虽然现在与模板交换的主要是开发人员(程序员)，但如果扩展到系统测试中，则与模板交换的就可能还包括测试人员、项目管理人员等。所以必须定义一个良好的结构，可以使模板的内容在多种角色(程序员、测试人员、管理人员等)中共享，并且不同角色具有不同的权限。例如：程序员就没有权力改动测试人员开发的测试用例。综合考虑上面的一些因素和未来软件 BIST 系统的发展，必须定义一个具有较强扩展性的存储机制。

现在，关系数据库理论的发展和应用已经非常成熟，有许多使用非常方便的数据库管理系统，如：微软的 SQL Server2000、DB2 等，而且，有一套通用的 SQL 语言来对数据库进行操作。另外，在数据库管理系统中还内建了权限管理、事务支持等内容。所以使用关系数据库是我们存储模板数据的首选。以下就详细讨论在软件 BIST 系统中怎样将模板的内容存储到 SQL Server2000 关系数据库中。

软件 BIST 系统中模板的内容都保存在名为"BIST"的数据库中，接下来详细讨论一下 BIST 数据库中存储模板内容的一些关键的基本表。在介绍的这些基本表中，每个表都包含一个 ID 字段，ID 字段的数据类型为 int，而且该字段是自动增长的，每次增长步长为 1。这样就可以保证该数据库中的表满足数据库设计中的第三范式要求。

1. 项目表

所谓一个项目就是指实现需求分析报告中用户所需功能的一个软件单元。一个项目包含很多的模块，每个模块实现项目中的一种或几种功能。项目表包括的字段见下表。ID 为项目编号，是一个自增长的整数。项目名称为该软件项目的名称，项目描述为该软件项目的简短描述，都为 varchar 类型，以字符串形式保存在数据库中。

表 3-1 项目表

ID	项目名称	项目描述
int	varchar(50)	varchar(255)

2. 模块表

软件中一个模块实现某些功能。模块与功能之间的关系是一对多的关系。在 BIST 数据库中体现为模块表和功能表间的关系。模块表包括三个字段，ID 字段为模块编号，模块名称在 C 语言源程序中体现为函数名，功能描述是指对该模块所完成功能的简单描述，可以为空。具体情况如下表所示：

表 3-2 模块表

ID	模块名称(函数名)	功能描述
int	varchar(50)	varchar(255)

3. 功能表

功能就是指某一组输入数据经过某模块处理后得到一组输出数据。这时我们就认为该模块实现了某种功能。比如说给定两个整数 10 和 5 经过模块(函数)sum(10，5)后，得到输出 15。我们就说模块 sum 实现了加法功能。从这个角度上来讲，一个功能可以由一组输入和一组对应的输出来表示。在软件 BIST 系统中功能表的内容可以从需求分析报告和详细设计说明书中得到。满足功能表中的所有功能要求，是该项目软件设计的最低要求。不能满足功能表要求的软件是不能提供给用户使用的。功能表中的这些功能在软件测试时是必须要进行测试的。

功能表的字段有以下几个：ID 字段是功能编号，功能名称字段是给某一模块功能人为起的一个名称。功能描述字段是该功能的简单描述，可以为空。输入字段和输出字段是指与该功能对应的一组输入数据和输出数据。关于输入和输出字段的格式很难进行统一。比如说，如果一个输入是用户单击界面上的一个按钮，这就很难描述为具体输入变量的形式(如变量 i=1，j=3 等)。所以为了便于存储，我们将输入和输出都表示为字符串的形式。如果是纯变量的形式并且没有任何的与用户交互形式的输入，就可以将输入和输出标识为“变量名=值；“这种形式，在生成测试用例时可以通过分析输入和输出字符串来生成测试用例。关联模块 ID 字段是指实现该功能的模块的 ID 号。关于功能表的具体情况见表 3-3 所示：

表 3-3 功能表

ID	功能名称	功能描述	输入	输出	关联模块 ID
int	varchar	varchar	varchar	varchar	int

4. 模块调用关系表

模块调用关系表中保存的是项目中模块之间的调用关系，这些调用关系对于在单元测试时生成桩模块和驱动模块是至关重要的。该表包含的字段有：ID 字段是模块调用关系的唯一编号。关系名称字段是自己命名的调用关系名称，可为空。被调用模块 ID 和调用模块 ID 是该模块调用关系中的被调用模块的 ID 编号和调用模块的 ID 编号。通过被调用模块和调用模块的 ID 编号，就可以查找功能表(根据功能表的关联模块 ID 字段等于模块调用关系表的被调用模块 ID 字段或调用模块 ID 字段)得到输入和输出字段的信息。根据这些信息就可以动态的生成桩模块和驱动模块。模块调用关系表的具体字段及其数据类型见表 3-4 所示：

表 3-4 模块调用关系表

ID	关系名称	被调用模块 ID	调用模块 ID
int	varchar	int	int

5. 程序流程表

程序流程表中主要存储源代码中的有效语句。关于程序流程表的详细内容已经在本章的上一节中进行了详细的讨论，具体内容见表 3-5 所示。在表 3-5 中“模块 ID”字段，记录程序流程表中该行属于那个模块。在软件 BIST 中引入程序流程表的主要目的是为了方便生成测试用例。

表 3-5 程序流程表

ID	行号	函数名称	语句类型	后继左节点	后继右节点	是否调用函数
int	int	varchar	int	int	int	bool

6. 测试用例表

软件测试的本质就是生成一组测试用例来检测被测软件是否实现所要求的功能，并且这组测试用例可以达到较大的故障覆盖率。为了便于生成和复用测试用例，在软件 BIST 项目中将测试用例存储在数据库测试用例表中。该表包括的字段有：ID 字段是测试用例的唯一编号。模块名是指可执行该测试用例模块(函数)名称。具体内容字段是指该测试用例的输入部分(因为我们现在主要研究的是科学计算类程序(基本上没有用户界面形式的输入)的测试模板，所以输入部分同功能表的输入字段一样，也表示为“变量名=值”；这种字符串格式)。预期结果字段保存该测试用例被执行后的期望结果(基于与输入部分同样的原因，也以字符串形式表示)。所需环境字段主要描述执行该测试用例所需的软、硬件环境。测试策略 ID 字段保存生成该测试用例所用测试策略(如：边界值测试、等价类测试、路径测试等)在测试策略表中的 ID 号。实际结果字段保存该测试用例执行后所得到的实际结果，也是以字符串形式保存。见表 3-6 所示：

表 3-6 测试用例表

ID	模块名	具体内容	预期结果	所需环境	测试策略 ID	实际结果
int	varchar	varchar	varchar	varchar	int	varchar

7. 测试点表

测试点是在软件 BIST 系统中提出的一个新概念，在测试点处可以预埋代码，收集执行信息，并对这些预埋代码进行包装，填入预先设计好的模板中，从而赋予程序一种内建自测试的功能。在软件 BIST 中，插入测试点处的预埋代码是以动态连接库(称为插装库)的形式存在的。每个插装库中包含为检测某一种软件故障而所

需的一组插装函数。可以根据一定的插装策略将这些函数插入到被测程序中。在运行被测程序后，可以收集从测试点处得到的信息，从而确定被测程序是否含有某种故障(或没有某种故障)。在软件 BIST 中，测试点的有关信息都存储在测试点表中，该表主要有下面这些字段：测试点字段名称中存储人为定义的测试点的名称，该字段可以为空。行号字段存储该测试点在源程序中的位置。模块 ID 字段存储该测试点所在模块的 ID 号(对应模块表中的 ID 字段)。关联插装库 ID 字段存储该测试点所插装代码(一个插装函数)所在的 DLL 库的 ID 号(即在插装库表中的 ID 编号)。插装函数名字段存储插入该测试点的插装函数的函数名。具体信息见表 3-7：

表 3- 7 测试点表

ID	测试点名称	行号	模块 ID	关联插装库 ID	插装函数名
int	varchar	int	int	int	varchar

8. 测试策略表

目前，关于软件测试已经形成了很多的成熟的测试方法(策略)。概括起来主要可以分为两种，一种是黑盒测试，另一种是白盒测试。黑盒测试是将程序看作是将输入映射到输出的数学函数。典型的黑盒测试策略有边界值分析、等价类测试、基于决策表的测试等。白盒测试的突出特征是其基于被测程序的源代码，而不是基于定义。由于这种绝对化的基础，因此白盒测试支持严格定义、数学分析和精确度量。在软件 BIST 系统中，我们采用了基本路径测试的白盒测试方法。基本路径测试是 Tom McCabe 首先提出来的，基本路径测试方法允许测试用例设计者导出一个过程设计的逻辑复杂性测度，并使用该测度为指南来定义路径的基本集。从该测试集导出的测试用例保证对程序中的每一条语句至少执行一次。软件 BIST 中生成测试用例的方法是基于基本路径测试的一种新的白盒分析方法，即二叉化基本路径测试法[23]。它是在基本路径测试的基础上，提出了二叉化的思想，使得生成的测试用例的路径覆盖率更高，而且生成测试用例过程更容易。

另外，程序员人为因素对一个程序的测试也是很重要的。如 C/C++语言编写的程序，空指针、数组越界、内存泄漏、指针的悬空引用等测试是非常重要的。有些程序员可能由于对编程语言中的某一概念了解不是很清楚，就可能在他所编写的程序中总是犯同一类错误，这时，如果要测试该程序员编写的程序就必须考虑到程序员本身的因素。在进行软件测试时必须要考虑到被测程序编程语言的特性、被测程序的类型和程序员人为因素的影响，而对这些影响在软件 BIST 系统中是归结为故障模型，通过对故障模型的测试就可以消除这些影响。从这一意义上来讲，也可以将故障模型看成是一种测试策略，只是这种测试策略的使用有一定的限制，只能使用在特定的环境下。

综合前面所分析的内容，可以认为软件测试策略包括典型的黑盒测试和白盒测试，还应包括特定的故障模型。这样一来，BIST 数据库中的软件策略表主要包括以下几个字段：字段 ID 是测试策略的唯一编号。策略名称字段存储测试策略的名称(如：边界值分析等)。策略描述字段存储关于该测试策略的描述，可以为空。插装库 ID 字段只在是否故障模型字段为 true 时有效，存储检测该故障模型的插装库的 ID 号。检测函数名也是在是否故障模型字段为 true 时有效，存储检测该故障模型的插装函数的函数名称。详细结构见表 3-8：

表 3-8 测试策略

ID	策略名称	策略描述	插装库 ID	是否故障模型	检测函数名
int	varchar	varchar	int	bool	varchar

9. 变量列表

变量列表中主要储存被测程序中定义的所有变量，它提供的内容主要用于在被测程序运行时进行变量跟踪，生成变量链表，见表 3-9 所示。ID 数据类型是整型，而且该数据是自动增长的(加 1，由数据库系统实现)。变量名是在被测程序中定义的变量名称。数据类型可以是 C/C++语言内置的数据类型也可以是用户自定义的数据类型。模块名表示该变量所在的模块的名称。占用字节数目的主要是为了检测指向该变量的指针是否会越界。对应实参 ID：如果该变量所在的模块是被调用的模块且该变量为形参，则在该字段中记录其对应的实参的 ID 号。否则该字段为空。对应形参 ID：如果该变量所在的模块是调用的模块且该变量为实参，则在该字段中记录其对应的形参的 ID 号。

表 3-9 变量列表

ID	变量名	数据类型	模块名	行号	占用字节数	对应实参 ID	对应形参 ID
int	varchar	varchar	varchar	int	int	int	int

10. 插装库表

插装库中存储的是软件 BIST 系统中可以插装到被测程序源代码中的插装函数。这些插装函数可以插入到被测程序的测试点位置。测试点位置是根据测试目的来选择。如：路径覆盖就是在每条独立路径上插入计数器函数。插装函数根据不同的测试类别而被组织成不同的插装库，如果要在被测程序中插入某一类插装函数，只需在被测程序编译前引入该插装库(以动态连接库的形式存在)就可以了。插装库表中存储的就是分类好的插装库，但如何将插装库保存在数据库的字段中呢？这里就存在两个问题：一是如何将二进制格式文件存储在数据库的字段中，因为编译好的动态连接库是以二进制文件格式存储的。二是插装库的大小可能很难确定，因为检测不同故障的插装库所需的插装函数多少、大小都不同，所以很难定义一个插装库占

用存储空间的上限。在软件 BIST 系统中，我们是这样来解决这个问题的，首先，将插装库表存储插装库内容的插装库内容字段的数据类型设为 image，并且不指定该字段的大小。其次，在将编译好的插装库(动态连接库)存储到数据库的插装库内容字段中，最后，在使用插装库时就可以以字节流方式读取插装库内容字段中的插装库(动态连接库)，并将其存储为本地的动态连接库文件，在将本地的动态连接库文件添加到被测程序中，这样就可以在设置测试点时插入该插装库中的插装函数。

在 BIST 数据库中，插装库表主要包括以下几个字段：ID 字段用来存储插装库的唯一编号，并作为插装库表的主键。插装库名称是自己给插装库命名。插装库版本号字段存储该插装库的当前版本号，该字段主要是为了插装库的版本控制而设的。插装库内容字段存储插装库的具体内容(动态连接库文件)。见表 3-10：

表 3-10 插装库表

ID	插装库名称	插装库版本号	插装库内容
int	varchar	varchar	image

到现在为止，我们已经讨论了模板的内容如何在数据库中存储，并详细的介绍了 BIST 数据库中的各个表结构，数据库中表之间的关系如图 3-3(UML 描述)所示。

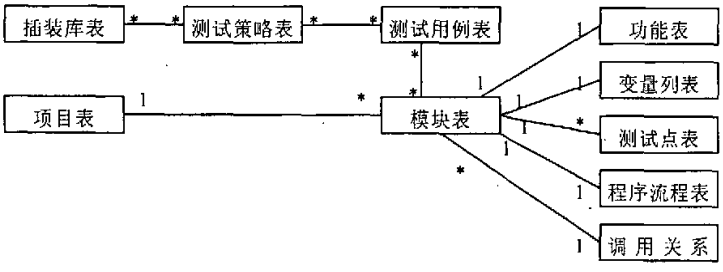


图 3-3 数据库中表之间的关系图

对于模板来说不仅要负责与软件开发人员交互，而且还要与软件测试人员进行交互，这就需要一个模板管理系统 TMS(Template Manage System)对模板中的内容进行管理。TMS 的主要功能应该包括：

- 根据测试人员或开发人员的要求，查找模板库的内容，并将结果返回给开发人员或测试人员；
- 对访问模板库的人员设置权限(读、写等)，并在其访问模板库时检查其权限，不允许进行越权访问；
- 提供统一接口，可以对模板中的内容进行分类和整理；
- 对模板中的插装库进行版本控制。

3.5 模板中测试结果的度量

模板作为软件 BIST 系统的重要组成部分，在开发和测试过程都有极其重要的

作用，因此需要对它量化描述，来评价模板的优劣以及模板给出的一些测试后的信息，下面讨论一下模板的评价参数。

- 模板的故障覆盖率 FC(Fault Coverage)[31]：是指模板生成的测试用例所能检测的软件错误数占程序中所有错误的百分比。模板生成测试用例不是最终目的，测试用例测试程序，发现定位程序错误才是我们的最终目的。因而生成测试用例的质量我们更为关注，测试用例的质量是指每个测试用例所能发现的程序错误的个数。那么所有测试用例检测到的错误的总和除以软件的总错误数就得到了模板的故障覆盖率。FC 主要表明模板所能达到软件测试中黑盒测试的能力，FC 值越大表明模板发现软件功能错误的能力越强。黑盒测试主要是验证软件是否达到软件规格说明中的软件功能，有没有正确达到功能，有没有遗漏某些应该实现的功能。对程序的输入是否能做出合理输出。黑盒测试没有考虑软件的内部结构。
- 测试点的覆盖率 CC(Checkpoint Coverage)：是指模板生成的测试用例所能遍历的测试点数占程序中所插测试点总数的百分比。在软件 BIST 系统中有 3 类测试点(分别是统计各类覆盖率设置的测试点，为了方便观察程序而设置的测试点和针对故障设置的测试点)，所以测试点的覆盖率在一定程度上能够综合反映被测程序在运行测试用例后的故障覆盖率、路径覆盖率、分支覆盖率等。但测试点覆盖率又和这些覆盖率不同，它是根据程序结构，着重于这些覆盖率的综合。
- 语句覆盖率 SC(Statement Coverage)：指模板生成的测试用例所能遍历的程序语句占原程序语句总数的百分比。
- 分支覆盖率 BC(Branch Coverage)：指模板生成的测试用例所能遍历的程序分支数占原程序分支总数的百分比。
- 路径覆盖率 PC(Path Coverage)：指模板生成的测试用例所能遍历的程序独立路径数占原程序独立路径的总数的百分比。
- 模板的有效覆盖率 EC[32](efficiency coverage)：是指模板生成的测试用例对被测程序的作用。显然，EC 与 FC，CC，SC，BC 和 PC 有关。

为了能够给出这几种覆盖率的量化公式，首先引入几个记号：

- NBIF(The Number of Faults Before Inserting Faults)指的是在往程序中植入故障之前，模板所能检测出故障的数目。
- NBAF(The Number of Faults After Inserting Faults)指的是在往程序中植入故障之后，模板所能检测出故障的数目。
- NIF(The Number of Inserting Faults)指的是在程序中植入故障的数目。
- T^C 是用测试点、语句、分支或路径表示的已执行的项目数。

- TNIC(Total Number of Items in the Code)：代码中项目总数。

1972 年，Mill 发明了错误植入法[33]。其基本思想是：一个团队在被测程序中预先植入 n 个已知的错误，然后另一个团队则尽最大努力地去发现被测程序的错误(包括植入的错误和被测程序中原有的错误)，那么发现的植入已知错误数和总的植入已知错误数的比值将等于发现的非植入错误数和总的非植入错误数的比值。也就是说，可以把发现的植入的已知错误数和总共的植入错误数的比值当成此次测试的故障覆盖率(FC)。在软件 BIST 系统中，对 Mill 的错误植入法进行修改：对某个独立的程序，在其中植入故障[34]，用模板来进行测试信息的收集，得到的故障覆盖率就是模板的故障覆盖率[28]。

$$FC=(NBAF-NBIF)/NIF\times 100\%$$

测试点、语句、分支和路径覆盖率都可以下面的用公式计算：

$$测试点、语句、分支或路径测试覆盖率=T^C/TNIC\times 100\%$$

目前对模板的有效覆盖率定义为：

$$EC=(a\times FC+b\times CC+c\times SC+d\times BC+e\times PC)\times 100\%$$

当然对于模板有效覆盖率的定义还是比较粗糙的，在应用中取 $a=b=c=d=e=1/5$。实际上对于这个定义还可以更进一步的研究，EC 虽然与 FC，CC，SC，BC 和 PC 有关，但不一定是线性的关系，而且它们对 EC 的作用也不完全相同，这里仅仅是一种粗略的评价。

至此，我们得到了 6 种覆盖率的计算公式，其中测试点、语句、分支和路径覆盖率主要用于测试报告中；而模板的故障覆盖率和有效覆盖率主要用于评价模板，可以判断设计的模板是不是可信性模板[29]。可信性模板是达到预期测试目标的模板，这里的预期测试目标就是指模板的预期设定的故障覆盖率 FC 和模板的有效覆盖率 EC。只要比较计算出来的模板 FC 和 EC 达到预期模板的设定目标，就可以认为所设计的模板是可信性模板。

3.6 本章小结

模板是软件 BIST 项目的关键，它是联系软件开发工程师和软件测试工程师的纽带。本章讨论了模板的概念、模板内容的获取、模板的分类、模板的使用、故障模型等模板设计中的各个方面的内容。并进一步研究了模板的评估、模板的管理等内容。

本章还讨论了模板的内容－测试点、桩模块和驱动模块、程序块的结构和测试用例，以及它们在软件 BIST 系统中的作用。然后详细设计了这些内容的组织和存储方式及其与程序员交互方式。最后给出模板的 6 个主要评价参数，包括模板的故障覆盖率、测试点的覆盖率、语句覆盖率、分支覆盖率、路径覆盖率和模板的有效

覆盖率，其中测试点覆盖率是软件 BIST 系统独有的，语句、分支和路径覆盖率和一般测试工具中的覆盖率相同，模板的故障覆盖率和模板的有效覆盖率是评价模板是否可信的参数。

第四章 模板的设计与实现

本章主要详细介绍了模板内容的获取。首先分析了 C/C++语言中六类常见的故障，包括：变量未初始化故障、空指针、数组越界故障、内存泄漏、内存操作的未定义故障和编译器本身不足的故障，并介绍了根据故障模型静态检测故障和设计模板函数动态检测故障，给出了检测故障的算法。

4.1 针对故障模型设计模板函数

故障模型是将测试员的经验和直觉尽量归纳和固化，使得可以重复使用；通过理解软件在做什么，来猜测可能出错的地方，并有目的地使它暴露错误。它一般有两个方面的用途：(1)它能够提供测试所必须的信息。(2)它可以用于开发预测。软件故障模型应该是软件物理错误的抽象，并能反映其本质的一定程度的组合。建立故障模型的好处是使用模型中给出的信息就可以设计和编制算法，用来产生可以运行的测试用例[30]。目前在软件 BIST 项目中，主要围绕故障模型为主进行设计研究。

4.1.1 典型故障分析

1. 未初始化变量

变量的初始化是指在创建变量期间对变量进行左值操作，保存一个已知值的过程。这意味着变量一旦被创建就得到了一个值。进一步而言，变量在初始化期间并不删除变量的任何现存值，而是在创建它的时候立即保存一个已知值。变量仅仅声明而没有初始化，则该变量在被赋值之前是未知的。变量的赋值是指删除变量所包含的任何值(甚至是未知的)。我们讨论的变量未初始化不仅仅指变量未初始化，而且包括变量仅仅声明没有被赋值的情况，这两种情况都可能造成程序的错误计算。我们通常所说的变量未初始化实际上包括这两种情况。

在 C/C++程序中未初始化变量仅对于全局非静态变量和局部变量来说，在 C/C++中有全局、局部(自动)变量和静态变量。事实上还有寄存器变量，寄存器变量也是自动变量的一种，不过在实现的时候和一般的变量有所区别(与编译器和计算机系统结构有关)，我们也将其归入自动变量并不影响我们的讨论。在编译程序过程中，编译器会给静态变量和全局变量赋值为 0(这是广义上的说法，对于不同的类型所赋的初值是不同的，如整型、长整型和枚举型一般赋值为 0，float 和 double 一般赋值为 0.0，对于结构体和联合体中的每一项可以递归处理赋值)，这是编译器给变量的一个默认值。局部变量放在函数的函数体内，另外，局部变量是在堆栈中实现的。

这就是说某函数所申请的局部变量可以传给它所调用的函数，但是不能把函数中的变量返回给调用者用，否则很容易导致程序出现问题。局部变量是自动的分配和清除存储空间，并且初始化的值是随机的。所以如果程序中没有对自动变量进行赋值就加以使用，那么就可能会出现一些意想不到的结果。

在全局非静态变量和自动变量的声明点和该变量的使用点之间没有对该变量进行左值操作的语句就会产生该故障。

2. 数组越界

数组是 C/C++等程序设计语言中一种常用的数据类型，它是同一种数据类型(包括基本数据类型和用户自定义数据类型)在内存中的连续存放。数组变量所拥用的内存空间可以在程序运行前确定，也可以在程序运行时动态确定，空间大小在变量定义时就已经确定，因此程序在运行中数组变量的索引超过了定义时的大小，就可能导致程序运行时出现异常，从而降低程序的可靠性。

数组越界主要是由于程序员的理解偏差和粗心所引起的，虽然通过一些方法可以降低这种错误的出现的概率，但是毕竟不能完全避免。

在软件 BIST 系统中，对数组越界主要是分为两类：非字符型数组越界和字符型数组越界，然后分别加以检测。

3. 空指针

在 C/C++语言中，指针是一种重要的数据类型，也是其成为最流行的编程语言的主要原因。指针(pointer)：实际上是一个变量的地址。在利用 C/C++语言编写程序是存在大量的指针的应用。而空指针是在利用指针进行编程时常见的一种错误。

在程序不指向任何对象，这种指针被称为空指针，空指针的值为 NULL，NULL 是在<stdio.h>中定义的一个宏，它的值和任何有效指针的值都不同。NULL 是一个纯粹的零，它可能会被强制转换成 void*或 char*类型，也就是说 NULL 可能是 0，0L 或(void *)0 等。

指针的值不能是整型值，但空指针是一个例外，空指针可以是一个纯粹的零(空指针的值未必是一个纯粹的零，但这个值是唯一有用的值。在编译时产生的任意一个表达式，只要它是零，就可以作为空指针的值。在程序运行时，最好不要出现一个为零的整型变量)。如果在程序中直接/间接引用一个空指针，则程序可能会得到毫无意义的整型变量，或者得到一个全部是零的值，或者会突然停止运行。

在 C/C++语言中，空指针的应用通常在以下两种情况是合法的：

- 空指针做函数调用失败的返回值或者做警戒值；
- 用空指针终止对递归数据结构的间接引用。如：对一个链表的操作时，一般将最后一个节点的后向指针设为 NULL，以便在遇到 NULL 时中止对链表的

操作。

在 C/C++语言中，产生空指针故障的可能情况：

- 在使用全局指针变量前，没有将其初始化(编译器在编译时自动将该指针变量置为 0，这时就会产生空指针故障)；
- 在调用一个返回值为指针的函数时，没有对其返回值进行判断而直接使用，如果返回值是一个空指针将会产生空指针故障；
- 在定义一个指针变量时，人为将其值置为 0 或者 NULL，并且在没有对其重新赋值的情况下使用了该指针变量，将会产生空指针故障。

4. 内存泄漏

内存的大小是有限的，过量的往存储器存储数据会导致存储空间不够而使系统崩溃。过量的存储数据有时候并不是程序员有意的行为。C/C++标准库提供了一些内存分配的函数，程序员使用这些函数在程序中动态分配内存，同时在使用完之后程序员必须自己释放这些内存，否则就会造成内存泄漏。内存泄漏的错误往往容易忽略，因为它短时间内并不会暴露，直至存储器的过量存储导致系统运行性能下降甚至因为内存的耗尽而崩溃。简单的来说，申请了内存而没有释放或者不完全释放，导致了那块内存不可用这种故障就是内存泄漏。

5. 对内存操作的未定义故障

操作的未定义是指：可能在开发阶段良好，在测试阶段良好，但在最后的应用中出现问题；也能在开发阶段和测试阶段就出现问题，总之是不可预测的。

在 C/C++中，还存在一些对内存错误的操作，这些操作 C/C++标准明确表示，出现这些故障后系统的行为是未定义的。在本文讨论的未定义行为的故障包括两类。第一，多次释放同一块内存空间，当出现这种情况时，系统的行为是未定义的。第二，内存的分配和释放所用的标准函数不匹配，在 C/C++中用 malloc/calloc/realloc 申请的内存必须用 free 释放，用 new 申请的内存必须用 delete 来释放，使用 new[] 申请的内存要用 delete[]来释放。反之就会造成了系统行为的未定义。

6. 编译器本身不足的故障

C/C++中还存在这样的问题，编译器能够检测的这类故障，但编译器并不处理，如果用户在一定情况下激活故障同样会引起程序的崩溃。以 scanf[35]函数为例，对于这样的一个程序段如下所示：

```
int x,y;
scanf("%d, %d",&x,&y);
```

该代码段是用户输入一串数字，数字以逗号隔开，但是假设用户键入"10 20"(没有逗号),x 被赋值为 10，而 y 被赋值为某个不可见的数。因此在键盘输入时，最好

能够接受字符串执行的语法检查以获得正确的输入。

4.1.2 根据故障模型实现模板函数

对于故障的检测可以分为静态检测和动态检测。有些故障如未初始化变量、部分空指针、内存泄漏和内存操作的未定义可以在运行前检测出来，另外一些故障如数组越界和编译器本身的不足必须在程序动态运行的时候才能检测出来。

对于必须动态检测的故障，为了实现软件的内建自测试，采用的方法是针对故障模型插入检测函数，这些函数称为模板函数。不同类型的故障对应的模板函数也是不同的。

4.1.2.1 针对故障模型的静态检测

1. 检测未初始化变量和空指针

在软件 BIST 系统为了检测未初始化故障，首先要收集的变量的信息，定义数据结构如下所示：

```
struct var_init {
  char [20 ]   namespace;    //变量作用范围
  int          line;         // 行号
  char [ 20 ]  type;         // 类型
  char [ 20 ]  name;         // 变量名
  bool         lvalue;       // 左值标志 #表示有操作 -表示无操作
  bool         rvalue;       // 右值标志 #表示有操作 -表示无操作
  char         state;        // 变量状态
  struct var_init *next;     // 变量链表
  }
```

根据程序的书写格式，基本上可以将变量的操作方式归结为以下几种情况，而变量未初始化就是下面的 D 标记状态。如表 4-1 所示：

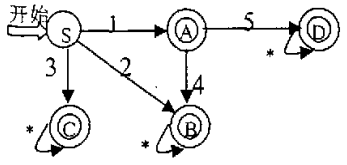
表 4- 1 预编译处理器扫描源程序对数据结构填充的例子

变量作用范围	行号	类型	变量名	左值	右值	范例	标记	下一个变量
		int	x	-	-	int x ;	A	
		float	x	#	-	float x = 0 ;	B	
			x	-	#	F (x = 3 , y = x * 4) ;*	D	
			x	#	#	x = x + 1 ;	C/D	
			x	#	-	x = 1 ;	B	

表 4-1 中*的位置主要是为了说明逗号运算符的一种实例。C/D 表示状态是 C 或

者 D，具体取决于前一个状态。

根据以上情况的总结，可以根据自动机的理论来实现变量未初始化的检测，如图 4-1 所示。



注释：1－声明；　　5－有右值操作；
2－定义；　　*－任意操作。
3－未定义；
4－只有左值操作；

图 4- 1 变量状态转移图

软件 BIST 系统对每一行代码进行扫描，为每一个变量都生成上述数据结构，同时根据扫描的代码为对变量的数据结构进行填充，当变量状态转移 B 状态则不需要再对变量进行跟踪说明此变量已经初始化,如果变量状态转移到状态 D 则说明变量未初始化，如果变量状态转移到状态 C 则说明使用了未定义的变量，如果变量状态转移到状态 A 则说明变量定义未使用(在软件 BIST 系统不处理后两类故障，编译器可以检测这两类故障)，算法如下：

```
//变量状态扫描分析
while(源代码未扫描结束)
  {
   var = getscanvar(); // var 是接受扫描到的变量
   if ( var 不在变量状态链表中)
          产生新节点，插入到变量状态链表中；
   else   if (var.state == B ) var.state = B ;
       else if ( getVarPorperity() == lvalue ) //变量只具有左值操作或赋值操作等
          var.state = B;
        else   if   (getVarPorperity() == rvalue ) //变量具有右值操作
                  switch ( var.state )
                    {
                      S : { var.state = C; 报错；// 此错误编译器可发觉可以不
处理}
                          break;
                      A : { var.state = D; 报错；// 变量未初始化}
                          break;
                     B :   var.state = B;
                          break;
                     C :   var.state = C;
```

```
                    break;
                default :
                    break;
                }
            else var.state = A;
    }
  //处理变量状态报错信息，并回收资源
  pnext = varList ->next;
  while (pnext != NULL )
    {
      if  (pnext.state == B )  销毁结点;
      else (pnext.state == C )  变量未初始化，根据变量的数据结构显示出错位置;
      //其它不用处理，编译器会报警告错误：变量未使用或使用未定义的变量;
      pnext++;
    }
```

软件 BIST 系统的预编译器主要完成词法、简单的语法和语义分析，我们不考虑编译器能够检测的常规错误，主要是针对编译器不能处理的故障实现预编译器。

对于空指针的部分故障，可以用同样的方法来检测，只不过收集的变量信息不同，算法和检测过程都是完全类似的，在此不再赘述。

2. 针对内存泄漏和内存操作的未定义故障的静态检测

内存操作的未定义故障相对内存泄漏来说是比较简单的，在静态检测的过程中只需要判断同一指针是否被释放多次以及分配和释放操作是否匹配。当然有可能出现程序员对内存的分配和释放操作是匹配的，但有意/无意出现不完全释放造成内存泄漏，这一点必须动态检测，限于时间关系我们仅仅实现了静态检测。

对于内存泄漏故障，我们是从以下几个方面来分析的[36]：

- 首先记录内存的分配操作，在 C/C++中也就是内存分配函数，如 strdup，malloc，calloc，realloc，new 等。在这些函数调用的节点上意味着内存泄漏故障的开始；
- 记录内存释放操作和程序退出操作，在 C/C++中内存释放函数为 free、delete；程序的退出会导致程序申请的内存返还给操作系统，C/C++中常用 abort、exit 等，对于 C++中因为发生异常引起内存的泄漏是非常复杂的事情，有许多专业书籍特别对此有专门的阐述[37, 38, 39]，在此不再讨论；
- 和内存相关的指针变量的赋值操作，当两个或多个不同的指针指向同一块内

存空间，他们都可以对内存进行访问和释放操作，但精确的别名是一个 NP 问题[40]。

- 作用域的变化。作用域的变化可能造成某些指针变量的失效，如指针变量从局部到全局作用域则变量失效。

针对内存泄漏故障的检查的关键就是如何记住各个指针和它们所对应的内存空间之间的关系。只用一个 Check_pointer 类来记录每个指针的所有情况是不够的，因为必须将指向相同内存空间的所有指针的情况都统一起来。所以，需要引入了一个 Mallocspace 的类，这个类用来记录内存空间的情况，而每个指针都有一个指向该类的一个指针域，如果要表示，几个指针指向了同一个内存空间，只需将这几个指针的 mallocspace 指针都指向同一个 mallocspace 的对象就可以了，下面先给出这两个类的定义：

```
class Check_pointer
  {
  public:
     bool initiation;
     CString malloc_length;
     char namespaces[10];
     Mallocspace * mallocspace;
     CString name;
     CString type;
     Check_pointer * next;
     Check_pointer(Cstring namea,CString typea)
     {
        name=namea;
        type=typea;
        next=NULL;
     };
     ~Check_pointer()
     {};
  };
class Mallocspace//记录内存块的信息
  {
  public :
```

```
    CString var[50]; //共享的变量名
    int num; //共享次数
    bool free; //释放标志
    Mallocspace()
    {
        num=0;
        free=false;
    };
    ~Mallocspace()
    {};
    int remove(CString name); //删除共享变量名
};
```

由此可以实现内存故障的检测，步骤如下：

有了 Check_pointer 和 Mallocspace 这两个类就可以设计出指针检查的算法，在扫描源程序时，将所有的指针变量全部加入到 Check_pointer 类的链表结构中，以便将来对每个指针变量进行检查。

当扫描到有指针类型的变量有左值操作时，表示该指针将分配一个新的空间，而分配给这个指针的空间可以分为：

- 分配的空间是另一个指针所指的空间如：int *q=(int*)malloc(3*sizeof(int)); *p=q;这个时候先在 Checkpointer 链表中查找指针 p，找到后，在软件 BIST 系统中就可得到关于 p 这个指针信息的 Check_pointer 对象的空间，设指向该空间的指针为 pcurrent，通过 pcurrent->mallocspace->remove(pcurrent->name)的调用将原来 mallocspace 对象中的 var 数组中的"p"删除，并且执行 pcurrent->mallocspac->num--(注：自减 1 操作)，然后执行 pcurrent->mallocsapce=q->mallocspace，这样*p 就和原来的 mallocspace 没有任何关系了，接着继续将新指向的 mallocspace－>num++，并且将"p"加入到 q->mallocspace->var[]数组中(别名处理的一种简单方式)，这样 p 指针的 mallospace 对象就得到了更新。
- 分配的空间是新分配出的内存空间：操作方式和第一种方式类似，只是在更新 pcurrent->mallocspace 时为指向一个新的 mallocspace，如 pcurrent->mallocspace=new Mallocspace()；并初始化该新 Mallocspace 对象 num++，并且将指针名加入 mallocspace->var[]中，同时 free 域设为 false，表示空间未被释放。

- 分配为空，这个时候就应该将 pcurren->mallocspace 置为空。

检查危险操作并给出警告：

- free 操作：当扫描到 free(p)这样的操作时，先检查 p->mallocspace->free 的值，如果该值为 true 说明该空间已经被释放，此时就给出警告信息，说明该指针所指空间已经被释放；如果该值为 false，说明还未被释放，此时将 p->mallocspace->free 置为 true。这时检查操作并没有结束，如果有其它指针指向了该指针所指的内存空间，应该给出警告，有指针悬空。
- 检查悬空操作步骤：检查 p->mallocspace->num 的值，如果大于 0，说明还有其它指针指向该空间。这个时候将 p->mallocspace->var[]数组中除 p->name 外的其它指针名全都列出，给出警告信息，还有这些指针被悬空。
- 置空操作：当扫描到类似 p=0 的操作时，先检查 p->mallocspace->free 的值，如果为 true，则不用给出警告；如果为 false 要给出警告，说明有内存泄漏。
- 资源未被释放：当程序结束时检查 Check_pointer 链表，查看每个指针 p->mallocspace->free 的值是否为 true，如果为 false，说明该指针所指向的空间并没有被释放，发生内存泄漏故障，要给出警告。

4.1.2.2 针对故障模型的动态检测

1. 检测数组越界模板函数

非字符型数组和字符型数组在程序中的应用都很广泛，字符数组除了能够象非字符数组一样根据下标索引进行操作，而且可以对整个字符串进行操作，这种情况一般是字符数组应用在标准的 C/C++的库函数中，因此需要把数组分为两类分别处理。在此我们对非字符型数组越界和字符型数组越界这两类故障分别插入不同的模板函数进行检测。

(1) 针对字符数组的处理

C/C++的库函数 memcpy，strcpy，strncpy，strcat，strncat，memmove，memset 等进行字符数组操作的时，编译器并不检测是否越界的问题。当从源到目标的拷贝或者移动设置等操作，源参数所占的空间大小大于目标空间大小就会发生故障。对此我们的处理是：首先分析字符串所需要的空间大小，这些作为模板函数的参数，并据此来判断是否有越界故障。

字符数组的声明在 C/C++中一般可以总结为以下四种情况：

(1) string 类型的常量，如：char *str="×××";

(2) char* str2=(char*)malloc(10*sizeof(char));

(3) char string1[3];

(4) char ch[]={'a','b','c'}

对于这四种情况，我们总能够在编译期间得到字符数组的空间大小。在软件BIST系统中可以通过调用字符库函数strlen()以及编译常量sizeof()分别获得其空间大小。例如在程序中出现了strcpy()这样的库函数，我们将屏蔽它并插入模板函数BIST_strcpy()，BIST_strcpy()的函数定义如下：

```
inline char * BIST_strcpy(char *des, char *src, int deslen, int srclen, int line)
  {
    if (deslen >= srclen)   //  源的空间大小大于目的的空间大小
      {
        strcpy(des, src);
      }
    else   //  否则截断处理并给出警告
      {
        while(*des) *des++=*src++;
        std::cout <<"警告：源串没有完全拷贝到目的串中，发生了截断，位于第
        "<<line<<"行"<<endl;
        return des; // 保证和strcpy()的返回值一致，便于嵌套处理；
      }
  }
```

其中deslen和srclen分别是软件BIST系统通过对源代码的扫描得到的常量。这些常量求法就是上面所说的关于字符数组空间大小的求法，它们作为参数在模板函数被插入到待测程序的时候已经是一个已知量。line对应的参数是一个宏常量__LINE__，用于报告出错代码的位置。

比较BIST_strcpy()和strcpy()会发现，新的模板函数BIST_strcpy()不仅仅保持了strcpy()的功能，即BIST_strcpy()可以完成字符数组的拷贝并支持函数的嵌套处理，同时还有检测故障的能力。另外BIST_strcpy()的函数被声明为inline函数，也就是说BIST_strcpy()在执行期间基本上不会比strcpy()占用额外的时间，保证了效率。对于其它的库函数memcpy，strncpy，strcat，strncat，memmove，memset等处理方式和strcpy类似。

(2) 针对非字符数组的处理

C/C++中用户声明基本类型(除字符型)或者用户自定义类型的数组，一般是通过索引下标来访问数组元素的，所以我们的插装函数是要确保没有非法访问。处理方法是：首先找到数组元素被使用的地方，然后分解出该元素的索引下标，并判断是否发生越界故障。假设用户使用了这样的语句：int a=array[2]，其中array[]是数组，

这样的语句将被插装成：int a = intcheckarray(array,2,sizeof(array)/sizeof(int), __LINE__)；其中 intcheckarray()是针对 int 类型数组的越界检测，并传入数组元素的下标、数组空间的大小以及代码所处的位置(用于报错)。在软件 BIST 系统中对于每一种数据类型都应该有这样的函数，可以用 C++中的模板来实现，减少代码的书写量，编译时 C++的模板根据需要的实际类型特例化。

```
Template <class Type>
inline T check_array(T array[], int i, int len, int line)
  {
     if(i>=0 && i<len) return array[i];
     else
       {
         std::cout<<"警告：数组越界，位于第"<<line<<"行"<<endl;
         exit(0); //如果有数组越界则程序退出，也可以给出错误信息，继续执行
       }
  }
```

函数也是被声明为 inline 函数，也就是说插装函数在执行期间基本上不会比原来直接引用数组元素占用额外的时间，保证了效率。

2. 编译器本身不足的故障

对于此类故障软件 BIST 系统的处理是插入合适的代码弥补编译器的不足，检测故障并提示用户。对于上一小节中所提到的代码是

```
int x,y;
scanf("%d, %d",&x,&y);
```

软件 BIST 系统的处理方式是：首先分析用户输入的可能方式，然后插入代有预见性的代码，如下：

```
#include   <string.h> // declares strchr()
char str[128];
fgets(str,128,stdin);
if   ( ( char * s = strchr ( str , ',' ) ) = =NULL )
  //strchr(str,',')找到第一个出现在串中的',', 否则返回 NULL。参见 MSDN
     scanf ( str,"%d %d ", &x,&y);
else
     scanf ( str,"%d , %d ", &x,&y);
printf ("x=%d,y=%d",x,y);//最后的输出验证
```

首先判断输入序列是否存在字符',', 然后选择不同的 scanf 函数(本示例仅仅是

对非字符输入的改装),在最后输出初试输入的结果供测试人员验证。插装后的程序避免了输入时由于疏忽而产生的错误,具有一定预见性并增加了整个程序安全系数。

4.2 针对需求分析设计模板

在上一章，已经探讨过模板内容还要与需求分析和详细设计有关，这些内容对于不同的软件来说并不相同，设计这一模块的目的就是让软件 BIST 系统能够自动来分析测试结果。

因此需要程序员提交软件的功能、模块、调用关系图等关于软件开发前期的内容。在软件 BIST 系统中会把用户提交的内容和系统自动分析的内容进行比较，例如比较模块的调用关系图是否正确，同时软件 BIST 系统也会根据用户提交的功能为生成测试用例提供指导。

4.3 针对代码设计模板

为了能够验证代码的正确性，必须扫描源代码得到程序的详细的逻辑结构等信息，主要有程序的控制流图、调用关系图等。

目前软件 BIST 系统中已经完成的工作是：生成程序的控制流图和独立路径数。在此不再赘述，可参见[28]中有详细的说明。

4.4 本章小结

本章讨论了模板的概念、模板内容的设计，分析了 C/C++语言中六类常见的故障，包括：变量未初始化故障、空指针、数组越界故障、内存泄漏、内存操作的未定义故障和编译器本身不足的故障，并详细介绍了根据故障模型静态检测故障和设计模板函数动态检测故障，给出了检测故障的算法。

最后指出了根据需求分析和详细设计，以及源代码设计模板，由于这一部分已经在软件 BIST 系统中有一定研究，所以本文仅仅简单的指出并没有详细的讨论。

第五章 软件 BIST 系统的测试步骤

本章主要讨论软件 BIST 系统中单元测试、集成测试和回归测试。首先引入关键模块，并对关键模块的量化作了详细的介绍；同时详细介绍了以关键模块为核心的单元测试和集成测试。

在第一章中已经提到软件测试过程可分为 5 个步骤：单元测试，集成测试，确认测试，系统测试和验收测试。同样在软件 BIST 系统中仍然分成这几个步骤来进行测试。

5.1 驱动模块/桩模块的相关概念

在对一个或者多个模块(方法)进行测试时，往往发现它在很多时候并不是一个独立的程序，无法对其进行单独的测试。所以在对它进行测试时必须建立一些辅助模块，通过使用这些辅助模块来模拟与被测模块所联系的其它模块。而这些辅助模块分为两种情况。

- 驱动模块(Driver Module)：在大多数场合称为“主程序”，相当于调用被测模块的主程序。它接收测试数据并将这些数据传递到被测试模块，被测试模块被调用后，最后输出实际测试结果。
- 桩模块(Stub Module)：即被测模块运行时所需要调用的子模块。用于代替被测模块调用的子模块。桩模块无需实现子模块所有具体的功能，可以进行少量的数据操作，但不允许什么都不做。

所测模块与它相关的驱动模块及桩模块共同构成了一个“测试环境”，如下图所示：

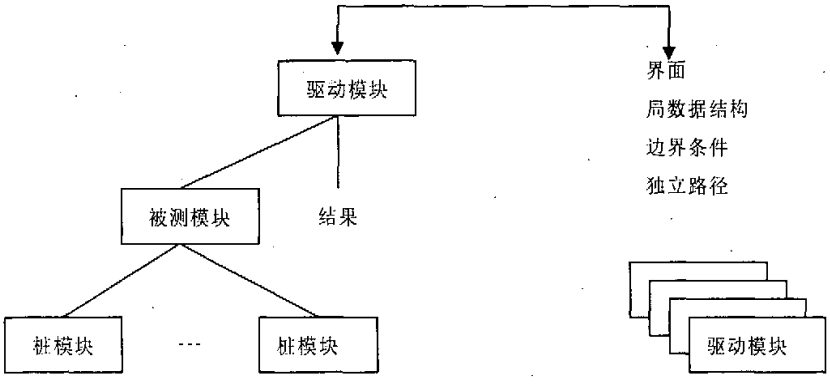


图 5-1 测试环境

目前软件 BIST 系统中已经确立了驱动模块的建立方法和七种建立桩模块的指导原则。在此不再赘述，可参见[41]中有详细的介绍。

5.2 关键模块的识别

虽然现有的工作可以完成软件的测试，但是效率并不高。原因是在进行单元测试和集成测试时仅仅按照模块关系调用图进行，我们知道在软件中存在一些调用关系复杂、本身极易出错的模块，在测试的时候应该优先对这些模块进行测试。本文的工作的之一就是找出关键模块，根据关键模块进行单元测试和集成测试。

所谓关键模块一般都具有下述一个或多个特征[9]：

- 和几个软件需求有关，即模块所实现的功能被多个软件需求所要使用到，该模块被调用的频率较高；
- 具有高层控制功能，即该模块类似于程序中的主程序模块，调用其它的模块，处于调用关系的上层；
- 本身复杂或者容易出错，即该模块本身的结构复杂，有递归调用，循环嵌套等结构，或者该模块自身的结构功能容易出错，存在不稳定性的情况；
- 含有确定性的性能要求，即该模块实现了确定的有着特殊性的功能。

由于关键模块和几个软件需求有关的这一特征涉及到软件需求，而每个软件的软件需求都是不一样的，没有共性，所以这个特征也是有着不确定性，在之后的实现过程中暂不考虑。同样，关键模块的含有确定性的性能要求这个特征也有其特定性，由于每个被测程序的模块性能、功能要求是不一致的，根据这一特征可以让用户指定关键模块。

另外，从以上几点可以看出关键模块的测试非常重要。在软件测试中有 20-80 原则，所谓 20-80 原则是指 80%的错误发生在 20%的代码中。我们可以看到关键模块正是这 20%中的一部分。所以优先测试关键模块是非常重要的。

在软件 BIST 系统中，我们需要根据源代码来识别关键模块。而根据程序的源代码仅仅能找出程序的调用关系图，但在程序调用关系图不能够反映出关键模块来，所以我们需要改进程序调用关系图，以便能够标示出关键模块。根据关键模块的特征中对于含有高层控制的模块以及本身复杂、容易出错的特征，我们在找到程序调用关系图的同时引入模块的被调用次数和调用其它模块的次数，分别称为模块的入度和出度。我们称带有入度和出度的程序调用关系图为强调用关系图。分析源代码，我们找出强调用关系图之后，对于每一个模块的入度和出度之和进行递减排序，然后依次进行单元测试。即入度和出度之和比较大的模块优先测试。

在软件 BIST 系统中为了生成强调用关系图，首先要识别用户自定义的函数，这需要词法分析和语法分析来识别。一般地，C 语言函数的文法形式化的可描述为：*类型修饰符 函数名 (参数)复合函数体*，具体可参见文献[42]的附录 A 详细列出了 C 语言的 BNF 语法。

为了识别用户自定义的函数，以及存储强调用关系图，我们分别为其设计了数据结构。识别用户自定义函数的数据结构如下：

```
struct Func_tab{
    char *storType;          //存储类型
    char *returnType;        //返回类型
    char *funcName;          //函数名
    int parameterNum;        //参数个数
    Para_tab *parameter;     //嵌套 Para_tab 结构，参数列表
    int firstrownum;         //起始行号
    int lastrownum;          //终止行号
    Func_tab *nextFunc;      //指向下一个函数
}
struct Para_tab{
    char *parameterType;     //参数类型
    char *parameterName;//参数名
    Para_tab *nextpara;      //指向下一个参数
};
```

在识别用户自定义的函数之后，我们需要根据源代码分析找出带调用次数的调用关系图，并且存储起来。为此我们定义其数据结构如下所示。

```
//用邻接表表示函数关系图
struct Func_relation
{
char *funcName;     //函数名
int weight;         //调用次数
int testflag;       //测试标志位，说明该单元是否已经进行了测试。
Func_relation *BrotherFunc;   //指向兄弟的指针
Func_relation *SonFunc;       //指向儿子的指针
};
```

由于高级语言的复杂性，模块之间的调用关系并非是线性的。例如在源代码中出现了递归调用、在循环中调用了其它的模块等等。对于这些模块一般来说是关键模块，如递归调用的模块一般来说是本身比较复杂且容易出错。这些模块在程序中仅仅的显式的被调用一次，而根据对源代码的扫描是不能直接定义为关键模块，这需要制定一些规则来生成强调用关系图。

由此我们根据关键模块的特征，并结合源代码的复杂性，具体制定规则如下：

- 对于第一个入口函数(在 C/C++即 main 函数)，定义调用次数为 1；
- 对于一般的模块，如果函数被调用，定义调用次数增加 1，如果调用其它模块。每调用一次其它模块调用次数增加 1；
- 对于调用递归模块的模块，每调用一次调用次数增加 1，递归模块本身调用次数为∝(表示重点，优先测试，以下皆同)；
- 对于构成间接递归的模块，调用次数都定义为∞；
- 对于出现在循环中被调用的模块，定义其调用次数为∞；
- 对于出现在被调用次数为∞模块调用的模块，定义其调用次数为∝(这种模块一般是公共部分，优先测试)；
- 对于在生成强调用关系图时，如果构成环路，环路上的节点调用次数为∞；
- 最后允许用户指定关键模块。

在软件 BIST 系统中，对源代码进行扫描，并按照上述规则生成强调用关系图。强调用关系图是一个有向图，每一个节点表示一个函数，边表示调用关系。对强调用关系图进行遍历，根据每个节点的度(入度和出度之和)依次进行测试。其中将关键模块定义为：出度和入度之和大于 2，值越大测试优先级越高。

5.3 以关键模块为核心的测试

为了提高和改进测试的效率，我们根据关键模块进行单元测试和集成测试，下面分别进行讨论并给出算法。

5.3.1 以关键模块为核心的单元测试

在软件 BIST 系统中，我们对单元测试的方法和传统的单元测试方法并没有太大的区别，仅仅按照关键模块为核心进行单元测试，其算法如下：

```
void unittesting()
  {
  while(测试不结束)
   {
    对强调用关系图进行遍历，根据每个节点的度，依次选出度最大的模块来；
    为该模块建立驱动模块/桩模块；
    运行测试模块，得到测试报告；
    进行下一个模块的测试。
    }
  }
```

5.3.2 以关键模块为核心的集成测试

传统的集成测试过程中组合模块的方式，有两种不同的集成方式：一次性集成方式和增殖式集成方式[9]。

一次性集成方式是一种非增殖集成方式，也叫整体拼装。按这种集成方式，首先对每个模块分别进行模块测试，然后再把所有模块集成在一起进行测试，最终得到符合要求的软件。

增殖式集成方式：增殖式集成方式也称为递增集成法，即逐次将未曾测试的模块和已测试的模块(或子模块)结合成程序包，然后将这些模块集成为较大系统，在集成的过程中边连接边测试，以发现连接过程中产生的问题。最后增殖逐步集成为符合要求的软件系统。根据集成的过程又可以分为自顶向下集成、自底向上集成、"三明治"集成法、定向冒险集成法、功能定向集成法等。其中常用的方法是：自顶向下集成和自底向上集成。

- 自顶向下集成是构造程序结构的一种增量式方式，它从主控模块开始，按照软件的控制层次结构，以深度优先或广度优先的策略，逐步把各个模块集成在一起。深度优先策略首先是把主控制路径上的模块集成在一起，至于选择哪一条路径作为主控制路径，这多少带有随意性，一般根据问题的特性确定。
- 自底向上测试是从"原子"模块(即软件结构最低层的模块)开始组装测试，因测试到较高层模块时，所需的下层模块功能均已具备，所以不再需要桩模块。

自顶向下集成的优点在于能尽早地对程序的主要控制和决策机制进行检验，因此较早地发现错误。缺点是在测试较高层模块时，低层处理采用桩模块替代，不能反映真实情况，重要数据不能及时回送到上层模块，因此测试并不充分。

自底向上集成方法不用桩模块，测试用例的设计亦相对简单，但缺点是程序最后一个模块加入时才具有整体形象。它与自顶向下综合测试方法优缺点正好相反。当然有时混和使用两种策略更为有效。

在软件 BIST 系统中，我们不仅仅优先测试关键模块，而且以关键模块为核心进行集成测试。

对每个模块进行单元测试，通过之后，将以关键模块为核心进行集成测试。根据集成测试的方法，我们在模块集成的过程中采用自顶向下集成和自底向上集成的混合模式，其中遵守的原则是：在每次增量集成尽可能的把相对最关键的模块集成进来，因此在集成测试时，同时可能需要建立驱动模块和桩模块。其中相对最关键的模块是指：与已经测试的模块集合中发生调用关系的模块所构成的集合，并将集合中度最大的模块加入到待测模块集合中进行测试，这个模块也称为与已测模块集合关系最密切的模块。以关键模块为核心的集成测试算法如下：

```
void intergratedtesting()
  {
      U 是所有函数模块的集合
      struct Func_rlation maxnode , nextnode;
      maxnode = Find_max(∅, Func_realation);
      maxnode.testflag = false; //标志此模块已测试
      Test = { maxnode }; //Test 表示目前的已测试的模块
      while ( Test ≠U ) //还有模块没有集成继续
      {
          nextnode = Find_max(Test , Func_realation);
          nextnode.testflag = false;
          Test = Test ∪ {nextnode}
          根据调用关系图对 Test 集合中的模块进行包装测试
      }
  }//集成测试的伪代码
//该函数返回的是与 Test 集合中关系最密切的一个模块
struct Func_rlation Find_max(Test , Func_realation)
  {
      //对 Test 集合的每一个元素都进行
      //深度优先搜索调用关系图，查找与这个元素调用关系中度最大的模块
      //返回与 Test 集合元素调用度最大的模块
  }
```

5.4 软件 BIST 系统中的回归测试

软件的改变可能是源于发现了错误并做了修改，也有可能是因为在集成或维护阶段加入了新的模块。当软件中所含错误被发现时，如果错误跟踪与管理系统不够完善，就可能会遗漏对这些错误的修改；而开发者对错误理解的不够透彻，也可能导致所做的修改只修正了错误的外在表现，而没有修复错误本身，从而造成修改失败；修改还有可能产生副作用从而导致软件未被修改的部分产生新的问题，使本来工作正常的功能产生错误。同样，在有新代码加入软件的时候，除了新加入的代码中有可能含有错误外，新代码还有可能对原有的代码带来影响。因此，每当软件发生变化时，我们就必须重新测试现有的功能，以便确定修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。同时，还需要补充新的测试用例来测试新的或被修改了的功能。为了验证修改的正确性及其影响就需要进行回归测试。

回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的工作量比重，软件开发的各个阶段都会进行多次回归测试。在渐进和快速迭代开发中，新版本的连续发布使回归测试进行的更加频繁，而在极端编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的效率和有效性是非常有意义的。在软件 BIST 系统中我们也考虑了回归测试。

回归测试的目标是利用软件现有的某些测试用例，对软件的新特性或修改的部分进行测试，以减少测试的工作量。

回归测试的目标之一是在程序修改之后，只对进行修改的部分进行重新测试，从而达到与完全测试相同的测试覆盖。

回归测试中主要有四个主题[43]：

- 区分通过测试而受影响的成分；
- 重新测试受影响的成分；
- 为重新测试对选择的测试用例确定标准；
- 为选择测试，要重用且修改现有的测试用例，并产生新的测试用例。

在软件 BIST 系统中，我们以语句为单位来区分受影响的模块，限于时间的关系，我们没有考虑因回归测试而进行的测试用例的选择、添加、修改和删除过时的测试用例。我们仅仅是找到受影响的模块，然后对模块进行单元测试和集成测试。

5.5 本章小结

本章主要讨论软件 BIST 系统中单元测试、集成测试和回归测试。首先引入关键模块，并且在本系统中对关键模块的量化作了详细的介绍；同时详细介绍了以关键模块为核心的单元测试和集成测试，给出了测试算法。

最后简单的讨论了回归测试，回归测试也是在测试过程中非常重要的一步，在软件 BIST 中也引入了回归测试，但限于时间的关系有关回归测试的很多内容没有进一步讨论，这也将是我们下一步研究的内容之一。

第六章　软件 BIST 系统的整体运行框架

本章介绍了软件 BIST 系统的代码规范检测，随后重点介绍了模板和测试点、测试用例之间的接口，对测试用例和测试点的管理方式进行了讨论。同时介绍了软件 BIST 系统中测试用例的生成、运行测试用例并最终生成测试报告等内容。

6.1 软件 BIST 的系统流程

设计软件 BIST 系统的目的是为了测试的方便，由于时间和资源的关系，我们还要借助原有的编译器实现编译。软件 BIST 系统的工作的流程可见下图所示：

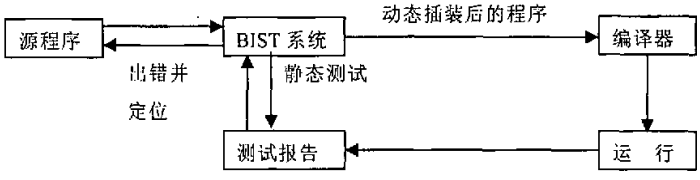


图 6-1 BIST 系统的工作流程

在第二章已经介绍了软件 BIST 系统的框架，主要介绍了软件 BIST 系统的两大模块，以及两个模块的主要功能和他们的协同工作。为了能够清楚的说明软件 BIST 系统的运行框架还有必要简单的介绍一下软件 BIST 系统的设计框架。首先在 BIST 系统中编写程序或者打开源程序文件，当执行 BIST 系统后，BIST 系统的预编译器对代码进行相应的词法分析、语法分析和简单的语义分析；然后 BIST 系统的内建部分根据扫描到的信息结合模板中的信息插入相应的自测函数，同时生成测试用例；最后运行测试程序验证程序并生成测试报告。其系统框架如下图所示：

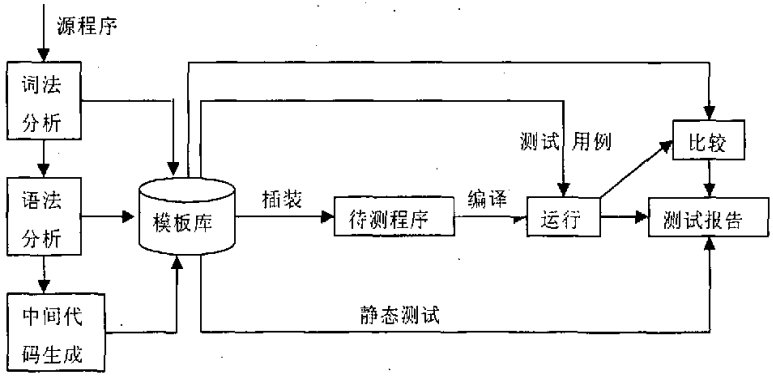


图 6-2 BIST 系统的设计框架

软件 BIST 系统运行过程如下：首先在软件 BIST 系统中编写程序或者打开源程

序文件，当执行软件 BIST 系统后，软件 BIST 系统的预编译器对代码进行相应的词法分析、语法分析和简单的语义分析；然后软件 BIST 系统的内建部分根据扫描到的信息结合模板中的信息插入相应的自测函数，同时生成测试用例；最后运行测试程序验证程序并生成测试报告。

6.2 软件 BIST 系统代码规范检测

为了提高程序代码的可读性，进一步使其规范化，增强其可靠性，不同的编程语言都有各自的编码规范。实践证明，严格地遵守编码规范可以大大地提高代码的质量。故在软件 BIST 中，首先将进行被测模块的代码规范测试。这其中即包括标准代码规范，也参考了一些工业用代码规范，软件 BIST 系统还允许程序员根据不同的程序类型随时添加自己的规范。前面说过，不同的编程语言有各自不同的编码规范，目前在软件 BIST 系统中我们将首先实现 C/C++语言的编码规范测试，在以后的研究中，BIST 系统完全可以扩充以实现多种编程语言的静态测试。在代码规范测试方面，前两届学长们已经进行了一定的研究。在此基础上，我对软件 BIST 系统中的 C/C++语言代码规范测试的内容进行了进一步的扩充和完善，并提出了初步的实现方法。代码规范的主要内容包括基本要求、结构化要求、正确性与容错性要求三个方面。例如：

- 每行只包含一个语句；
- 常量和宏定义必须全部以大写字母来撰写；
- 常量和宏定义的右侧必须有一简单的注释,说明其作用；
- 所有标识符不超过 31 字符 ；
- 不同名空间中的变量不得同名；
- 所有数字常数应当加上合适的后缀表示类型，例如 51L,42U,34.12F 等；
- if, else if, else, while, do..while, for 等复合语句块必须使用{}括起；
- 一个函数只能有一个出口。

等规则，在以前的工作已经有详细的介绍，具体可参见[41]。

在软件 BIST 系统中，代码规范测试借助 LEX&YACC 来完成。首先把上述规则加入到 YACC 语法文件中，然后在编译程序的同时，一边扫描，一边检查扫描到的语法规则是否和自定义规则相符，如果符合那么记下它的行号和警告类型。YACC 语法分析中所解析的标识符是否有效，通过 LEX 来进行词法分析，在分析文件的同时我们可以附加 C 语言程序，实现特定的功能。如计算每一行字数、初始化所有标示变量、根据标示变量设置警告信息等。

目前市场上流通的测试软件都有代码规范的检测，在软件 BIST 系统中的代码规范检测不仅仅是系统预先定义好的，而且用户可以添加。软件 BIST 系统拥有自

己的代码检测库，同时还有一个用户自定义的代码检测库，用户可以根据自己的需求以及对代码规范的要求程度进行添加规则。软件 BIST 系统在进行代码检测的时候首先匹配用户的代码检测规范，同时标出已经进行的规范化代码，然后匹配软件 BIST 系统的代码检测库，对于同一代码当两者的规则冲突时，以用户规则为主(主要是考虑到实用方面的原因，这也是给软件 BIST 系统的用户提出一种消减系统规则的途径，即当用户感到规则太严格不利于自己的代码时可以添加相应的用户规则，这样系统规则就无效了，为用户提供了方便)。

6.3 软件 BIST 系统各模块的工作

在第二章中，我们提到软件 BIST 系统由四部分组成，即测试点、测试用例、模板和测试程序模块。关于模板和生成测试程序已经介绍过了，下面将介绍测试点和测试用例以及四个模块的协同工作。

测试点(Checkpoint)：在被测程序中的某些特定位置插便于测试和观测的代码，提取关键信息，读入测试用例，输出动态测试信息，以达到提高程序可测性的目的。

测试用例是用于测试的输入的实际值和期望输出。由于测试用例使用明确，所以一个测试用例能够识别出对测试过程的约束。测试用例独立于测试设计是考虑到测试用例可以用于多个设计和在其它的情况下再次使用。

这两个模块有另外的同学来完成，这四个模块的构成了软件 BIST 系统。其中以模板为核心进行交互，下面将详细的他们之间的接口。

6.3.1 模板与测试点的接口

测试点和模板的接口是通过约定数据库、表、字段而实现的。在软件 BIST 系统中存在 3 类测试点(分别是统计各类覆盖率设置的测试点，为了方便观察程序而设置的测试点和针对故障设置的测试点)，每类测试点的数据结构不同，所以用于存储信息的字段也不同。因此在模板中，需要 3 个表来存储测试点的信息。这些表依次是 T_checkpoint1、T_checkpoint2、T_checkpoint3。其中 T_checkpoint1 用来存储第一类测试点的信息，在该表中定义了七个字段：

- CP_id 表示测试点 ID；
- CP_type 表示测试点类型 1 为路径覆盖统计 2 分支覆盖统计 3 语句覆盖统计 4 条件覆盖统计；
- CP_file 表示记录当前测试的文件名；
- CP_line 表示记录当前测试到的行号；
- CP_value 表示记录路径、分支、语句、条件执行的次数；
- CP_number 表示已插入的测试点个数；

- CP_next 表示下一个测试点的 ID;

其中 CP_id 为 auto_increment 类型，即增加一个测试点该字段的值就自动增加 1。这样当插入完测试点后，提取到的所有信息就写入到了模板中。

6.3.2 模板与测试用例的接口

测试用例和模板的接口也是通过约定数据库、表、字段而实现的。不过在生成测试用例的过程，很多信息来自测试点对程序插装后的提取。所以测试点与测试用例之间要存在一定的"共识"。这种"共识"也是通过定义数据库、表、字段实现的。提供生成测试用例的表为 T_testcase，该表中设置了六个字段：

- Lang_ID 表示语句的 ID；该字段为 auto_increment 类型，即每向数据库中增加一条语句该字段的值就自动增加 1;
- section_No 表示程序块的编号;
- line_No 表示该程序块中所记录的程序代码的行号;
- data_String 表示该行的程序代码，为了减轻模板开发的负担，这里用字符串的形式来表示程序代码，并且如果语句中有条件判断，在字符串中只表示了其中的条件部分。例如"if (a>b)"语句，表示成字符串"a>b";
- data_Type 表示此行程序代码的类别，如"if 语句"，"while 语句"，"一般语句"等等;
- left_Child 表示此行程序代码的左分支，如果是顺序类的语句，没有分支，则左分支表示的是顺序执行的下一语句的行号。如果是包含条件判断的语句，则左分支表示条件为真时所执行的下一语句的行号;
- right_Child 表示此行程序的右分支，如果是顺序类的语句，没有分支，则此右分支数值设置为 0。如果是包含条件判断的语句，右分支表示的是条件为假时所执行的下一语句的行号。

这样当插入完测试点后，提取到的所有信息就写入到了模板中。例如对于以下待测程序：

```
START:(代码前数字表示行号)
3: i=1;
4: while(c>0)
5: {
6:   printf("path testing");
7:   c--;
8: }
9: if(a>5 && b>10)
```

10: x=10;

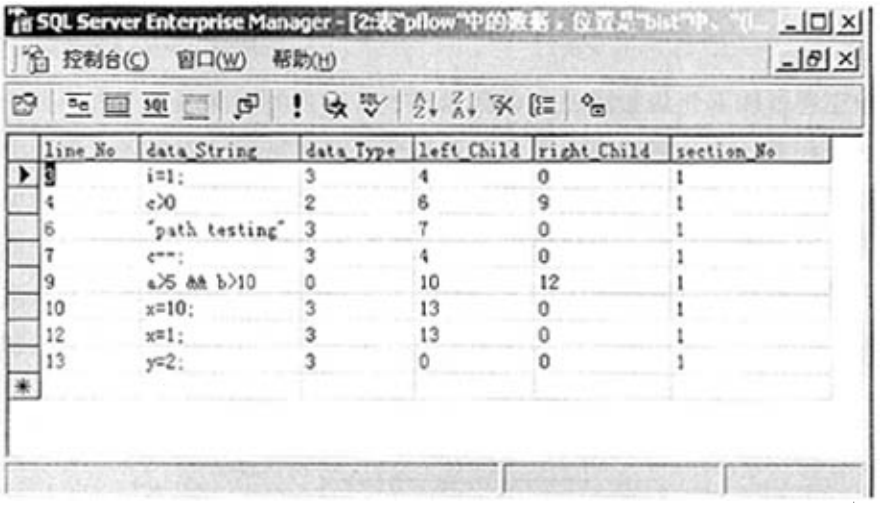
11：else

12: x=1;

13：y=2;

END.

对程序块进行分析后，相关信息存储在表 T_testcase 中。依靠这些表中提供的信息，才能完成测试用例的自动生成工作，测试用例生成程序并不直接与原始的待测程序块打交道。对以上的程序段提取信息如下图所示：



line_No	data_String	data_Type	left_Child	right_Child	section_No
3	i=1;	3	4	0	1
4	c>0	2	6	9	1
6	"path testing"	3	7	0	1
7	c--;	3	4	0	1
9	a>5 && b>10	0	10	12	1
10	x=10;	3	13	0	1
12	x=1;	3	13	0	1
13	y=2;	3	0	0	1

图 6- 3 T_testcase 表中的内容

测试点和测试用例的接口除了通过约定数据库、表、字段以外，还有另外一种形式。对于复杂的信息我们选取数据库作为存取信息的容器，而对于一些简单的信息，我们选取的存储容器是文件。这是因为读取数据库需要一定的时间，直接把数据存入文件中，既节省了时间，又减少了操作。在插入函数调用检查探针函数时，我们选取的容器形式就是文件，因为相比较整个程序而言，函数调用的语句个数是很少的。这时，测试点和测试用例之间的接口就是对文件格式的“约定”。

该文件的格式为：

函数 1 (参数类型 1，参数类型 2，.....参数类型 n) 函数开始行号 函数结束始行号

函数 2 (参数类型 1，参数类型 2，.....参数类型 m) 函数开始行号 函数结束始行号

.....

在此文件中，每个函数调用语句占用一行。函数名称后为空格，然后是各个参数的数据类型，在最后为函数体的开始和结束行号。

举例如下：

Halt (int, int, float) 1　　23

Skip (char, int, long, struct) 24　　52

Face (char, int) 52　　100

……

根据“约定”，在生成测试用例的时候，每读入一行语句，就相当于被测文件中有一句函数调用。再根据一定的测试用例生成算法，就可以自动地生成需要的测试用例。

6.3.3 测试程序的生成

前面讲述了静态测试的内容和方法以及为动态测试做准备的一些工作，如驱动模块和桩模块的生成、测试用例和测试点的生成与存储等，接下来讨论如何进一步完成动态测试。在上述准备工作完成之后，接下来就是要自动/半自动生成可编译运行的测试程序。生成测试程序的主要工作是和模板进行交互，在被测模块中插入测试信息，将被测模块组装成一个可在 BIST 系统中独立运行的程序，并将测试结果信息写入模板中的测试报告内。生成测试程序的流程图如下：

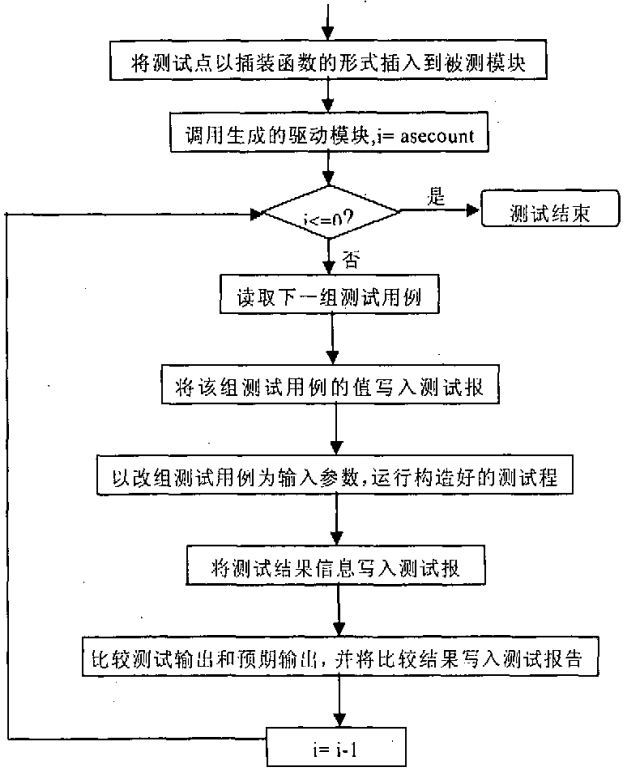


图 6-4 建立测试程序流程图

生成的测试程序以“test_”+modulename 命名，和被测程序存在同一个文件夹下。

这样在测试完成之后，源程序就是一个通过软件 BIST 系统测试的代码模块，可以保证一定的可靠性，并可以进入下一级的测试。以下面的代测模块为例：

```
#include "stdio.h"
string f(int i,int j)
  {
   int k;
   while(i<5)
   {
          k=0;
          i++;
   }
   k=j%i;
   if(k>3)
     return k;
   else
     return(k+10);
   }
```

测试该函数，生成的测试程序代码如下：

```
//include the head files
#include "stdio.h"
//the function with checkpoints under test
string f(int i, int j)
  {
   int k;
   while(i<5)
   {
     k=0;
     i++;
     //checkpoint
     printf("i=%d\n",i);
     //end checkpoint
   }
   k=j%i;
```

```
    if(k>3)
        return k;
    else
    {
      //checkpoint
      printf("k=%d\n",k);
      //end checkpoint
      return(k+10);
    }
  }
// the main function
main( )
    {
        int x, y;
        int iCasenumber=0;
        int iPassornot;
        string StrResult, StrExpectResult;
        LinktoTemplate( );
        icasenumber=casemanagement.casescount;
        if(iCasecount>0)
         {
            StrExpectResult=GetValue(tc_f, "result");
            x=GetValue(tc_f, 1);
            y=GetValue(tc_f, 2);
            WriteCasetoTestReport(x, y);
            StrResult=f(x, y);
            WriteResulttoTestReport(StrResult);
            iPassornot=Passornot(StrResult, StrExpectResult);
            WritePassornot(iPassornot);
            iCasecount=iCasecount-1;
         }
    }
```

6.4 软件 BIST 系统的测试报告

测试程序生成后，将下来的工作就是编译运行测试程序，得到测试结果。运行测试程序有两种方式可以进行。一种是在软件 BIST 系统中调用特定的编译器和链接器进行。比如对 C/C++程序来讲调用 Visual c++ 6.0 自带的 CL.EXE 编译器和 LINK.EXE 连接器对测试程序进行编译连接。在编译后运用 Windows 管道技术截获编译器的输出结果写入测试报告。这样做不仅对开发软件 BIST 系统来讲是方便一些，但却大大地限制了软件 BIST 系统的功能，还给程序员带来了很多的不方便。基于上述情况，我们拟将软件 BIST 系统作为插件插入到程序员熟悉的开发环境中，如 Visual C++6.0 等。

当测试程序生成完毕，待测程序块进行测试后，要生成各种类型的测试结果报告，帮助软件开发人员和测试人员分析错误结果和原因，为进一步完善软件提供技术支持。测试报告应该分两部分，一部分是静态的分析程序块后的错误报告；另一部分是经过驱动模块、待测程序块、测试用例和桩模块所组装成的测试程序运行后，实际结果与预期结果比较后的错误报告。在软件 BIST 系统中，测试报告将以 Word 文档的形式存在，名称组成为"report_"+ modulename 组成，并与被测模块放置在同一文件夹下。软件 BIST 系统有两种测试报告：

- 静态测试报告

软件 BIST 系统中的静态测试结果为违反了被测模块违反了 BIST 系统中定义的代码规则，或其中存在某一种故障，或全部通过。因此，静态测试报告内容包括：

（1） 规则号，规则内容，违反规则的代码行号；

（2） 存在的故障类项号和故障描述及出错代码行号；

（3） 违反规则数统计和故障统计；

（4） 是否通过。

其中，程序块的编号由模板数据库中的表所提供；出现问题的程序行号则是由静态分析器在进行静态错误分析的时候在程序中动态扫描生成的；出现问题时所对应的规则号是我们所命名的系列规则的编号之一，也就是在扫描时所用到的特定规则的编号。

- 动态测试报告

软件 BIST 系统中的动态测试结果是指测试用例和测试点的运行结果。包括如下内容：

（1） 测试用例组名称；

（2） 每对测试用例的值，该对测试用例对应的测试结果以及预期结果；

（3） 测试用例对数，通过的对数、百分比以及未通过的对数、百分比；

（4） 不同类型的测试点以插装函数的形式显示出的测试结果，比如特定位置的变量值等；

（5） 不同测试策略所具有的特定的测试结果，比如路径覆盖率、分支覆盖率等。

（6） 动态测试时检测出来的特定的故障类型比如内存泄漏等并指出出错代码的位置。

（7） 程序员对此次测试的评价或对有关问题的解释等，以备以后查看。

另外，如果需要的话，亦可通过 Windows 的管道技术结果编译器的输出信息，一并写入测试报告中。测试用例对的值和测试结果是在每次运行程序的同时一步一步地写入测试报告的。而测试点的值则是通过插装函数直接写入测试报告。最后，把测试的统计结果信息写入测试报告。把静态测试结果和动态测试结果写入同一个文档中，提供给程序员查看。

6.5 本章小结

软件 BIST 系统由多个模块组成，每个模块不是孤立存在的。各个模块之间的关系非常紧密。要使整个系统正确无误地运行，必须定义每个模块的接口。本章首先介绍了软件 BIST 系统的代码规范检测，随后重点介绍了模板和测试点、测试用例之间的接口，对测试用例和测试点的管理方式进行了讨论。并在此基础上生成测试用例、运行测试用例并最终生成测试报告。最后讨论了测试的最终运行的问题，并给出了软件 BIST 系统的测试报告，静态测试报告和动态测试报告，分别对其应该包含的内容进行了分析。

第七章　典型实例分析

本章主要介绍软件 BIST 系统在实际中运用的效果，给出几种故障，然后利用软件 BIST 系统进行检测。说明软件 BIST 系统的可行性，验证了软件 BIST 思想的正确性。

7.1 数组越界故障的检测

根据第六章介绍，经软件 BIST 系统插装和静态检测后程序经编译器编译然后运行，为了说明软件 BIST 系统的效果，比较源程序和经过 BIST 系统插装后程序在三个不同编译器(VC6.0、TurboC 2.0 以及 Linux 系统中 Gcc 和 G++)中运行结果。下面是一个关于字符串数组的例子：

```
void main( )
  {
     char *string1= "ab";
     char *string2="123456789";
     strcpy(string1,string2);
     printf("string1=%s\n",string1);
     printf("string2=%s\n",string2);
  }
```

（1） 在 VC 中的动态测试运行结果：

string1=123456789

string1=123456789 (之后程序给出出错报告：遇到问题需要关闭)

（2） TurboC 2.0 动态测试运行结果：

string1=123456789

string1=456789

（3） Gcc/G++ 动态测试运行结果：

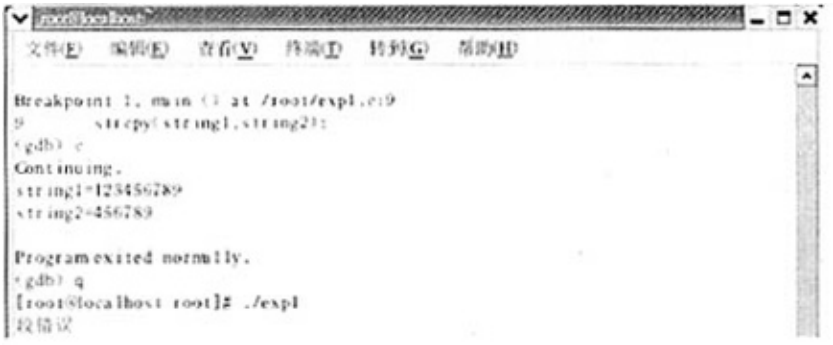


图 7-1 字符数组越界的源程序在 Linux 下运行结果

不管在以上任何编译器下，用户得到的结果都是错误且不可理解。在 TC 下 string2 所指的串被修改了，在 VC6.0 和 Linux 下程序运行无结果，且无提示或者提示段错误。在 Linux 下使用 GDB 来调试程序，可以跟踪发现 string1 和 string2 的值和 TC 中相同(见上图所示)，由此说明在程序中存在错误，仔细阅读和理解程序之后，发觉这一错误正是由于字符串数组的越界引起。经过 BIST 系统插装后的代码如下：

```
#include <iostream>
inline char * BIST_strcpy(char *des, char *src, int deslen, int srclen, int line)
  {
    if (deslen >= srclen)   //   源的空间大小大于目的的空间大小
      {
      strcpy(des,src);
      }
    else   //   否则截断处理并给出警告
      {
      while(*des) *des++=*src++;
      std::cout <<"警告：源串没有完全拷贝到目的串中，发生了截断，位于第
      "<<line<<"行"<<endl;
      return des; // 保证和 strcpy()的返回值一致，便于嵌套处理；
      }
  }
  void main()
  {
      char *string1="ab";
      char *string2="123456789";
      BIST_strcpy(string1,string2,strlen(string1),strlen(string2),__LINE__);
```

```
    printf("string1=%s\n",string1);
    printf("string2=%s\n",string2);
}
```

（1） 在 VC++6.0 中的动态测试运行结果(还有关于运行时的一些内容，如命名空间等信息略去)：

警告：源串没有完全拷贝到目的串中，发生了截断，位于第 18 行

string1=12

string2=123456789

*注：第 18 行就是上面程序中的调用字符拷贝函数的位置

可以看出因为在执行 strcpy 的时候，string1 的长度小于 string2 的长度，程序给出了错误信息，没有导致程序崩溃，说明插装函数起到了检查字符型数组越界的功能并给出了错误信息和错误的准确定位。

（2） 在 TurboC 2.0 测试运行结果(插装后的程序要符合 TC 的标准，如不支持 inline，命名空间等)：

警告：源串没有完全拷贝到目的串中，发生了截断，位于第 18 行

string1=12

string2=123456789

*注：第 18 行就是上面程序中的调用字符拷贝函数的位置

（3） 在 Gcc/G++测试运行结果：

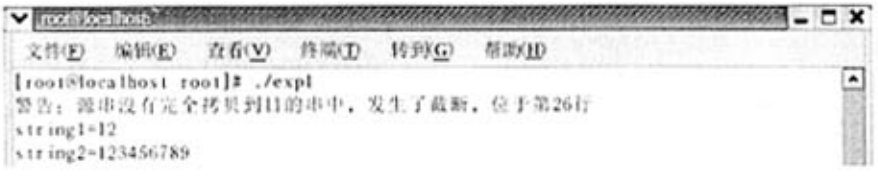


图 7-2 插装后的字符数组越界的程序在 Linux 下运行结果

上图中给的警告信息是第 26 行，这是因为源代码在不同的操作系统和不懈哦难过得编译环境中编译运行需要一些库函数和宏命令，如 stdio.h，stdlib.h，namespace std 等，在文中为了节省篇幅，将这些省略，但并不影响结果，实际上结果是一致的。

从这个例子可以看出，使用软件 BIST 系统不仅可以保证程序的正确性，找出故障，而且有更好的可移植性、消除不同编译器的细微差别。

下面是一个关于非字符串数组的例子：

```
void main()
  {
     int a[3]={1,2,3};
     for(int n=1;n<4;n++)
```

```
    printf("%d\t",a[n]);
}
```

（1）在 VC++6.0 中的测试运行结果：

2　3　1245120

说明：可以看出源程序中数组大小为 3，但是经过 for 循环输出后，数组下标达到了 3，但是在 C/C++中数组下标是从 0 开始，到 2 结束，所以数组越界，运行结果出错。

（2）在 TurboC 2.0 中测试运行结果：

2　3　-34

（3）在 Gcc/G++测试运行结果：

图 7- 3 非字符数组越界的源程序在 Linux 下运行结果

可以发现在三种编译器下程序的运行结果都不相同，对这个小的程序可以根据经验知道最后一个输出是随机数，由此找出错误。但是当程序规模较大时，很难根据经验找到这类错误以及其位置，但是在软件 BIST 系统却比较容易。

首先经过 BIST 系统插装后源程序：

```
inline int intcheckarray(int array[],int i,int len,int line)
  {
    if(i>=0 && i<len) return array[i];
    else
    {
        std::cout<<endl<<"警告：数组越界，位于第"<<line<<"行"<<endl;
        exit(0);
    }
  }
void main()
  {
     int a[3]={1,2,3};
     for(int n=1;n<4;n++)
     printf("%d\t",intcheckarray(a,n,sizeof(a)/sizeof(int),__LINE__));
  }
```

（1）在 VC 中的测试运行结果：

2　3

警告：数组越界，位于第 14 行

*注：第 14 行就是上面程序中使用数组元素的位置

说明：从运行结果看出，当程序访问数组，发现下标越界后给出警告，同时终止数组的访问，所以插装函数起到了防止数组越界访问的作用。

（2）在 TurboC 2.0 测试运行结果：

2　3

警告：数组越界，位于第 14 行

*注：第 14 行就是上面程序中使用数组元素的位置

（3）在 Gcc/G++测试运行结果：

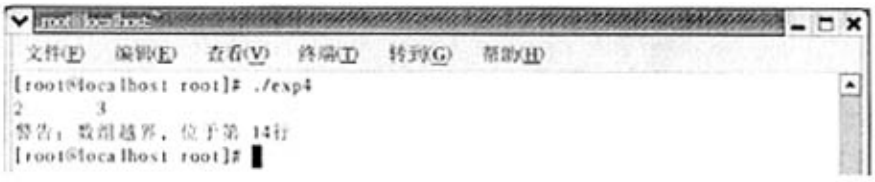


图 7-4 插装后的非字符数组越界的程序在 Linux 下运行结果

结果再次表明使用软件 BIST 系统不仅可以保证程序的正确性，找出故障，准确地进行故障定位，而且有更好的可移植性、消除不同编译器的细微差别。

7.2 内存故障的检测

针对内存的故障采用的是静态检测，下面给出两个关于内存的典型故障，使用软件 BIST 系统进行静态检测，并根据第四章的检测操作给出相应的警告和出错信息。

1、检查指针是否已被悬空警告：

```
#include <stdio.h>
#include <stdlib.h>
void main()
  {
     int *p=(int*)malloc(sizeof(int)*3);
     int *q=p;
     free(q);
     free(p);
  }
```

检测结果：

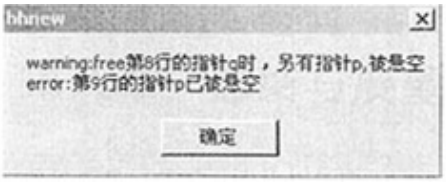


图 7-5 软件 BIST 系统检测指针悬空

2、检查指针空间未释放警告：

```
#include <stdio.h>
#include <stdlib.h>
void main()
  {
      int *p=(int*)malloc(sizeof(int)*3);
      int *q=p;
  }
```

检测结果：

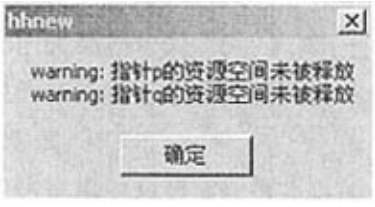


图 7-6 软件 BIST 系统检测内存泄漏

7.3 本章小结

本章主要介绍软件 BIST 系统在实际中运用的效果，给出了有数组越界和内存典型故障的示例程序，然后利用软件 BIST 系统进行检测，结果表明软件 BIST 系统不仅仅可以找出故障，而且可以精确定位，说明了软件 BIST 系统的可行性，验证了软件 BIST 思想的正确性。同时演示了，软件 BIST 系统不仅仅可以和 Windows 下的编译器整合，而且可以和 Linux 下的编译器进行整合。

第八章 总结与展望

本章主要回顾了所做的工作以及下一步的工作重点。

软件规模越来越大，功能也越来越复杂，传统的软件测试方法越来越不能满足软件的发展。本论文就是围绕软件内建自测试这一新的测试方法来展开研究的。在导师徐拾义教授的指导下，我们的项目已经取得了很大的进展，并取得了初步成果。

8.1 研究工作总结

我的研究内容包括以下几个方面：

首先讨论了模板的基本概念，其中详细的讨论了软件 BIST 系统内容的获取，主要根据软件故障模型，包括六类：未初始化变量、数组越界、空指针、内存泄漏、对内存操作的未定义故障和编译器本身不足的故障，并针对故障的性质设计静态检测故障或者插入模板函数动态检测故障。

本文也进行了测试过程中关键模块的研究，借助已有的建立驱动模块和桩模块的方法，提出了以关键模块为核心进行单元测试和集成测试，同时简单的探讨了在软件 BIST 系统中进行回归测试。

讨论了模板、测试点、测试用例和测试程序生成的关系，以及它们的接口，并讨论了为被测程序进行静态和动态测试建立不同的测试报告。

在本文最后给出几个实例，来验证软件 BIST 系统的可行性。

8.2 今后的工作

软件内建自测试思想是一种非常先进的理念，是一种新型的测试框架，以往成熟的设计思想(比如优秀的测试用例生成方法)都可以融入进来，集成进来，使得测试更加全面和完善。随着软件内建自测试项目的深入研究，我们的系统也会越来越完善，功能越来越强大。由于时间和科研条件等方面的限制，本课题还存在许多改进之处，我们将来的工作主要在以下几个方面：

- 设计更加友好程序交互界面，使程序开发人员更方便的使用我们的系统；另外应该使我们的系统与软件开发人员所使用的开发环境联系更加密切，使其对软件开发人员的透明度进一步加强。
- 完善这六类故障的处理，目前对这六类故障的处理还不够完善，例如内存故障还应该进行动态检测。
- 故障模型库的扩充。应该在多分析编程语言特点和前期的测试经验的基础上，

建立起更多的故障模型，丰富模板的内容和测试的故障覆盖率。一个好的故障模型库可以使系统更科学、更充分地对被测程序进行测试。下一步准备对死代码检测、非法的计算和非法的类型转化等故障进行检测。

- 软件 BIST 软件测试框架中各个要素细节的整合，这一点对于测试的效果是非常重要的。比如测试用例和测试点的整合，即如何使测试用例的数据和控制点的数据在一次测试过程中结合起来，使得观察点的数据更能反映问题。
- 在 BIST 系统中进行回归测试，以及测试比较器的研究也是下一步工作之一。
- 结合自然科学基金项目"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"进行深入的研究，将可测性设计应用到软件中，解决测试难的问题。

我们坚信，通过不懈地努力，这个国家自然基金项目会取得不断的进步，以硬件 BIST 思想带动软件测试的发展将用事实证明是可行的，同时，将会保证高质量的软件。

参考文献

[1] 刘杰,叶东升.软件测试技术及其在工程中的应用.计算机世界专题[J-OL].URL http://www2.ccw.com.cn/99/9947/9947c03.asp.

[2] 郑人杰.计算机软件测试技术[M].北京:清华大学出版社, 1992:25－28.

[3] Rex Black 着,天宏工作室译.测试流程管理[M].北京:北京大学出版社, 2001:35-47.

[4] 左咏露.面向对象软件测试及其方法研究[D].西安:西安理工大学硕士论文,2003:9-12.

[5] Breuer M.A., A.D. Friendman. TEST/80—A Proposal for an Advanced Automatic Test Generation System[C]. In: Proc. IEEE AUTOTESTCON, 1979:305-312.

[6] Savaria Y., B. Laguë, B. Kaminska. A Pragmatic Approach to the Design of Self-Testing Circuits [C].In: Proc. Int Test Conf. 1989: 745-754.

[7] IEEE Std 610.12-1990, IEEE Stand Glossary of Software Engineering Terminology [S].

[8] 蔡开元.软件可靠性工程基础[M].北京:清华大学出版社, 1995: 53-140.

[9] Pressman R.S.著,梅宏译.软件工程:实践者的研究方法(原书第5版)[M].北京:机械工业出版社, 2002:318-385.

[10] 曾平英,李兆麟,毛志刚.ASIC 可测试性设计技术[J].微电子学, 1999, 29(3): 149-153.

[11] Waicukauski J.A., E. Lindbloom, B. Rosen, et al. Transition Fault Simulation [J]. IEEE Design & Test of Computers, 1987, (4):32-38.

[12] Ramamoorthy C V, Bastani F B., Modeling of the Software Reliability Growth Process[C].In: Proc. of COMPSAC. Chicago: COMPSAC, 1980:161-169.

[13] Ramamoorthy C V, Bastani F B., Software Reliability — Status and Perspectives [J]. IEEE, Trans. Soft. Eng, 1982, (4): 354 - 371.

[14] Wagner K. D., C. K. Chin, McCluskey E. J. Pseudorandom Testing [J].IEEE Trans on Computers, 1987, C-36(3):332-343.

[15] Malaiya Y. K. Antirandom Testing: Getting the Most out of Black-box Testing[C].In: Sixth Int Symp. on Soft Reliability Engineering. 1995:86-95.

[16] Cheng K. T., V. D. Agrawal. A Partial Scan Method for Sequential Circuits with Feedback [J]. IEEE Trans on Comp., 1990, 39(4):538-544.

[17] Kim K S, C R. Kime. Partial Scan Flip-Flop Selection by Use of Empirical Testability [J]. JETTA, 1995, 7(1-2):47-60.

[18] Weiser M. Program Slices: Formal, Psychological, and Practical Investigation of an Automatic Program Abstraction Method [D]. Ann Arbor: University of Michigan, 1979.

[19] Gallagher K. B., J. R. Lyle. Using Program Slicing in Software Maintenance [J]. IEEE Transaction on Software Engineering, 1991, 17(8):751-761.

[20] 徐拾义,李文.软件内建自测试中测试点的设计及实现[J].同济大学学报(自然科学版), 2004, 32(8): 1057-1060.

[21] 李文锋,徐拾义.软件内建自测试中的测试数据生成方法[C].见:第十届全国容错计算学术会议论文集, 2003: 263-266.

[22] 凌良合,徐拾义.软件内建自测试的模板和基准程序的设计[J].同济大学学报(自然科学版), 2002, 30(10): 1159-1163.

[23] 钟治平,徐拾义.程序插装技术在软件内建自测试中应用[C].见:第十届全国容错计算学术会议论文集, 2003: 74-78.

[24] 李文峰.软件内建自测试中测试用例的生成[D].上海:上海大学硕士论文, 2004: 9-17.

[25] Lesk M. E., E. Schmidt. Lex-A Lexical Analyzer Generator[R]. AT&T Bell Laboratories, UNIX Programmer's Supplementary Documents, Volume1 (PSI) Edition, 1986.

[26] Johnson S. C. YACC-Yet Another Compiler-Compiler[R]. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[27] 朱荣,软件测试中故障模型的建立[C].见:第二界中国测试学术会议论文集, 2002: 247-255.

[28] 罗顶林.软件内建自测试中模板的研究和设计[D].上海:上海大学硕士论文, 2005: 12-43.

[29] 凌良合.软件内建自测试模板和基准程序设计[D].上海:上海大学硕士论文, 2003: 15-37.

[30] 宫云战.软件测试的故障模型[J].装甲兵工程学院学报, 2004, 18(2): 1-5.

[31] Beizer B. Software Testing Techniques [M].New York: Van Nostrand Reinhold Company, 1983.

[32] 徐红,钱红兵.Ada 软件的动态测试技术研究[J].北京航空航天大学学报, 1997, 23(1): 18-24.

[33] Ibarra P.H., S.K. Sahni. Polynomially Complete Fault Detection Problems [J]. IEEE Transaction on Computers, 1975:242-248.

[34] 王宗青.软件内建自测试模板的研究及其在科学计算中的应用[D].上海:上海大学硕士论文,2004:10-27.

[35] 邓劲生,张晓明译.Visual C++ 程序员实用大全[M].北京:中国水利水电出版社, 2002, 85-85.

[36] 肖庆,张威,宫云战等.内存泄漏的一种静态分析方法[J].装甲兵工程学院学报, 2004, 18(2): 23-26.

[37] Scott Meyers 著,侯捷译.Effective C++ 中文版 2nd Edition[M].武汉:华中科技大学出版社,2001:22-48.

[38] Scott Meyers 著,侯捷译.More Effective C++ 中文版 2nd Edition[M].北京:中国电力出版社 2003:44-184.

[39] Hurb Sutter 著,卓小涛译.Exceptional C++中文版[M].北京:中国电力出版社,2003:41-238.

[40] Landi W., B. G. Ryder. Pointer-induced aliasing: A Problem Classification[C]. In: Conf. Rec. 18thAnn. ACM Symp. Principles of Programming Language, 1991:93-103.

[41] 杨艳芳.软件内建自测试中测试程序的生成[D].上海:上海大学硕士论文, 2005: 12-34.

[42] 冯博琴,冯岚.编译原理及实践[M].北京:机械工业出版社, 2000:373-380.

[43] 曹军,李华.面向对象软件的回归测试策略[J].内蒙古大学学报(自然科学版),1998, 29(6): 768-771.

作者攻读学位期间公开发表的论文

[1] 侯惠芳,彭成寒.面向对象软件度量工具的设计实现[J].计算机工程与设计. 2005, 26(6): 1447-1449.

[2] 彭成寒,徐拾义.软件内建自测试中模板内容的研究与实现[J].计算机应用研究. 2006, (已录用, 待发表).

[3] 彭成寒,徐拾义.软件内建自测试中模板的研究和实现[J].上海师范大学学报(增刊). 2005, 34(9): 124-129.

致 谢

谨以此文献给所有关心、教育和帮助过我的人。

感谢我的导师徐拾义教授。他传授给我的故障诊断、容错和测试等方面的知识，为我课题的研究与论文的写作奠定了基础，他所负责的国家自然科学基金项目给我提供了一个非常好的研究平台。徐老师严谨的治学态度、广博的学术知识以及平易近人的品质给我留下了深刻的印象。更重要的是，他数十年如一日，始终站在科研前线，扎扎实实做事，和时间赛跑的工作品质，对我今后的生活和工作将产生巨大的影响，是我学习的好榜样。在此，谨向他表示我崇高的敬意和感谢。

感谢本课题组的肖全亮、徐睿红和朱伟同学，以及已毕业的罗顶林、唐培、李文和杨艳芳等师兄师姐在学习上给予我的帮助。通过与他们的探讨与交流，使我受益匪浅，加深了我对本题的认识，启发了我的思维。他们的支持和鼓励使我能够顺利完成毕业课题。

感谢何鸣、宋炯等每一位和我朝夕相处的同学，感谢他们在生活中给我的帮助，使我度过了愉快的研究生生活。同时感谢班上其它同学和计算机学院的老师对我生活和学习上的帮助。

感谢河南工业大学信息科学与工程学院的侯惠芳老师，不仅仅在本科的学习中给予了我巨大的帮助，在研究生的学习中她一如既往的给予关心、支持和帮助。

感谢我的哥哥彭成晓，在中国科技大学读博期间不仅仅要完成自己的学业，还要关注我的学习、生活。他宛如我的灯塔一般，是我学习和前进的动力。

深深地感谢多年来一直给予我巨大关怀和鼓励的父母，他们的抚育教养伴我走过了近二十年的学业生涯，每当我遇到困难挫折的时候，他们总是在后方尽全力的支持我，在他们爱的眼光中，我一步步走向独立和成熟。

同时还要感谢我的朋友皮静丽、李军、赵爽、李丁、殷钊民等，感谢他们时时给我鼓励。

衷心感谢在我求学期间关心过我的每一个人。

最后，感谢国家自然科学基金对本课题的资助。

附录 A 相关公式和定理

【定理 1】未初始化变量可以通过数据流分析得到。

证明：假定数据流分析建立在中间代码之上，且在分析中不考虑过程调用、指针变量和别名等复杂的情况。

建立的数据流方程，定义 in[B]是基本块 B 开始点的活动变量集合，定义 out[B]是基本块 B 结束点的变量活动集合，定义 def[B]是基本块 B 中定值且在该定值前没有引用的变量集合，定义 use[B]是基本块 B 中引用且在该引用前没有定值的变量集合。

那么，每个基本块的 def 和 use 可以从自身的基本块得到，in 和 out 需要通过方程组求解。方程组可以建立为：

$in[B] = use[B] \cup (out[B] - def[B])$

$out[B] = \bigcup_{S} in[S]$ 其中 S 是 B 的后继的集合

程序起始块的 in 集合就是为初始化变量的变量集合。

使用这种方法，如果程序中的某个变量没有置初值，但是也没有对该变量进行引用，则它不在上述定义的未初始化变量集合中，另外值得注意的是，如程序中的某个变量没有置初值，对它的引用出现在某个分支上，但这个分支属于死代码，即该分支根本不会运行，但这样的变量是在未初始化变量集合中，因为这种方法采用的是稳妥的策略。(证毕)

注：在软件 BIST 系统中为了简单起见，将整个程序视为一个基本块进行处理，对整个程序进行数据流分析，这样做的结果会与实际有一些误差。举例简单说明，如下：

```
int a;
……//这些代码没有使用和定义变量 a 的值
if (condition) a = 5;
else a = a+1;    //变量 a 仅在分支语句中的一个分支初始化
……//其他语句
```

对于这样的语句，如果根据程序的基本块来分析，当采用上面定理中稳妥的策略时变量 a 会被认为是未初始化变量。在软件 BIST 系统中因为把程序视为一个基本块，所以也必须采用稳妥的策略，也就是对于 if-then-else 这样的语句变量必须在 if 和 else 语句中分别被初始化，否则就认为是未初始化的变量。与上面的 if-then-else 语句类似，对于 for 循环、while 循环、do-while 循环还有程序中使用的 goto 语句和

switch 语句中使用 break 都有类似的问题。都需要做一些特殊的处理。

【定理 2】未初始化故障的等价定理：未初始化故障满足等价关系。

证明：等价关系满足三个特性，即自反性、对称性和传递性。

设 R 表示程序中未初始化故障的等价关系，程序设计语言中未初始化故障的等价也就是程序中的赋值语句。再设 a, b, c 是未初始化故障集合中的任意三个元素。等价关系 R 对未初始化故障集合划分成等价类。

（I） a 和 a 在同一个分块中，故有 a R a，即 R 是自反的。

（II） 若 a 和 b 在同一分块中，则 a R b 表示 a = b；即用 b 为 a 赋初值，得到的 a 也是未初始化，那么用 a 为 b 赋初值，得到的 b 也是未初始化，即 b R a 也成立，故 R 满足对称性。

（III） 若 a 和 b 在同一分块中，b 和 c 在同一分块中，则分别有 a R b 和 b R c，即 a = b 用 b 为 a 赋初值，得到的 a 也是未初始化，b = c 用 c 为 b 赋初值，得到的 b 也是未初始化，那么用 c 为 a 赋初值，得到的 a 也是未初始化，即 a R c 也成立，故 R 满足传递性。

所以，未初始化故障满足等价关系。(证毕)

注：在第四章中提供的算法是基于这样的事实，即出现未初始化故障，然后用这个未初始化变量给其他变量进行赋值，在软件 BIST 系统中只指出第一次未初始化的变量，当这个变量被修改完成初始化，则由它组成的等价类也都完成了初始化。如果想进一步得到准确的未初始化集合，按照上面的定理，只要在 4.1.2.1 中的算法加入求等价类的过程即可。

说明：定理 1 给出这样的事实，即对程序的数据流进行分析可以得到未初始化变量，定理 2 说明未初始化变量之间的关系，便于准确找出未初始化变量。

附录 B 图表索引

附录C 术语表

内建测试(Built-in-Test)：在芯片内部设计了"测试设备"来检测芯片的功能，避免了数据需要串行传输到外部设备的问题，将于控制或观察系统状态和相关信息的一些标准化部件和接口在设计时直接融合到系统中去，作为系统的一部分，因而，增强了系统的可控性和客观察性。

自测试(Self-Test)：是指利用设备内部具有自检测能力的硬件和软件来完成对设备的检测。

测试用例(Test Case)：测试内容的一系列情景和每个情景中必须依靠输入和输出，而对软件的正确性进行判断的测试文档，称为测试用例。测试用例是将软件测试的行为活动，做一个科学化的组织归纳。一般包括输入、动作、或者时间和一个期望的结果。

内建自测试(Built-in-Self-Test)：在设计一个电路或系统时,就设计一些附加的自动测试电路,同功能电路集成在同一芯片上,完成芯片加工后,就可以用附加的内建自测试电路对芯片本身进行测试。BIST 是将 BIT 技术与自测试技术的结合。

可测性设计(Design for Testability)：指在电路或者系统的设计阶段就考虑以后测试的需要，将可测试设计作为逻辑设计的一部分加以设计和优化，为今后能够高效率地测试提供方便。一般包括描述输入、动作、或者时间和一个期望的结果等。

回归测试(Regression Testing)：检查系统变更之后是否引入新的错误或者旧的错误重新出现，必须重新测试现有的功能，以便确定修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。

输出比较器(Output Comparators)：通过对几组输出的比较找出它们之间的差别。常用于回归测试中测试用例的筛选。

故障模型(Fault Model)：将测试员的经验和直觉尽量归纳和固化，使得可以重复使用；通过理解软件在做什么，来猜测可能出错的地方，并有目的地使它暴露错误。

静态检测(Static Testing)：不实际运行软件，主要是对软件的编程格式、结构等方面进行评估，在软件 BIST 系统中还包括找出故障。

动态检测(Dynamic Testing)：在 Host 环境或 Target 环境中实际运行软件，并使用设计的测试用例去探测软件漏洞。包括功能确认与接口测试、覆盖率分析、性能分析、内存分析等。

软件质量(Software Quaility)：满足或超出认定的一组需求，并使用经过认可的评测方法和标准来评估，还使用认定的流程来生产。

模块(Module)：程序的模块一般是指能完成一定的功能要求，具备明确的接口关系的独立程序单元。

程序插装(Program Instrument)：简单地说是借助往被测程序中插入操作来实现测试目的的方法。使被测程序在保持原有逻辑完整性的基础上，插入"探测器"，以便获得程序的控制流及数据流信息，并可得到测试的覆盖率。

调试(Debugging)：过程开始于一个测试用例的执行，若测试结果与期望结果有出入，即出现了错误征兆，调试过程首先要找出错误原因，然后对错误进行修正。