一种基于组合测试的软件故障调试方法

徐宝文[1),2),3),4)] 聂长海[1),2)] 史 亮[1),2)] 陈火旺[3)]

[1)](东南大学计算机科学与工程系 南京 210096)

[2)](江苏省软件质量研究所 南京 210096)

[3)](国防科学技术大学计算机学院 长沙 410073)

[4)](武汉大学软件工程重点实验室 武汉 430072)

摘 要 在研究了组合测试基本模型的基础上,提出了一种基于组合测试的故障原因诊断方法.该方法基于组合测试的结果,补充一些附加测试用例进行重新测试,并对其结果作进一步分析和验证,从而迅速将故障原因锁定在很小的范围内,这样可为软件的调试和测试工作提供更方便、更有价值的线索和参考.

关键词 软件测试;组合测试;软件调试;测试用例集;待测软件

中图法分类号 TP311

A Software Failure Debugging Method Based on Combinatorial Design Approach for Testing

XU Bao-Wen[1),2),3),4)] NIE Chang-Hai[1),2)] SHI Liang[1),2)] CHEN Huo-Wang[2)]

[1)](*Department of Computer Science and Engineering, Southeast University, Nanjing* 210096)

[2)](*Jiangsu Province Institute of Software Quality, Nanjing* 210096)

[3)](*School of Computer Science, National University of Defense Technology, Changsha* 410073)

[4)](*Key Laboratory of Software Engineering, Wuhan University, Wuhan* 430072)

Abstract This paper presents a failure diagnosing method based on the combinatorial design for testing. This method analyzes the schemas included by the correct running test cases and failure caused test cases in the combinatorial testing suite, it then can conclude that the errors must be in a very small range through analyzing the test cases and retesting with some complementary test cases. So it can provide the very efficient and valuable guidance for the debugging and testing of software. This paper gives a further study on Combinatorial Design Approach for Testing, which is studied and applied widely for its scientificity and effectiveness in software testing with a quite small test suite, especially for the software under testing whose faults come mainly from the parameters or the interactions of the system parameters.

Keywords software testing; combinatorial design approach for testing; software debugging; test suite; software under testing

1 引 言

组合测试作为一种重要的软件测试方法,因能以较少的测试用例来实现对被测试系统进行科学有效的测试而得到广泛的研究和应用.对于那些故障来源于系统中一些参数和参数之间的相互作用的系统,这种方法的效果尤其明显.

收稿日期:2003-03-08;修改稿收到日期:2005-10-03.本课题得到国家杰出青年科学基金(60425206)、国家自然科学基金(60373066,60403016,90412003)和江苏省自然科学基金(BK2005060)资助.徐宝文,男,1961 年生,博士,教授,博士生导师,主要从事程序设计语言、软件工程、并行与网络软件、知识与信息获取技术等方向的教学与科研工作.E-mail:bwxu@seu.edu.cn.聂长海,男,1971 年生,副教授,博士,主要从事软件工程和软件测试技术、模糊信息处理、神经网络等方面的教学与科研工作.史 亮,男,1979 年生,博士研究生,主要从事软件分析与测试领域的研究.陈火旺,男,1936 年生,中国工程院院士,数理逻辑学家,计算机专家.

应用组合测试方法进行测试，首先要找出待测试系统中确定测试场景空间的各个参数. 这些参数可以是待测软件系统的配置参数、内部事件、用户输入参数以及外部事件参数等. 以下我们不妨设一个待测系统 SUT(Software Under Testing)由 n 个参数 $c_1, c_2, \cdots, c_n$ 组成，这组参数分别取一组离散值 $T_1, T_2, \cdots, T_n$，其中 T_i 表示参数 c_i 可取的有限离散点集. 并假设符号 a_i 表示第 i 个参数 c_i 可取值的个数，即 T_i 中元素的个数：$a_i = |T_i|$.

对于系统 SUT 而言，导致系统运行时出现故障的因素可能是某一个参数或者是某些参数的相互作用. 如果对这 n 个参数的各种组合进行完全测试，需要 $a_1 \times a_2 \times \cdots \times a_n$ 个测试用例，这在一般情况下，因测试用例数量太大而不可行，也没有必要. 如果要对每个参数的各个取值进行覆盖(即在测试用例中，要保证每个参数的各个取值都至少出现一次)，那么需要 $m = \max_{1 \le i \le n} a_i$ 个测试用例；如果对任意两个参数的所有取值组合进行覆盖，保证它们每一个都在测试用例中出现，那么至少需要 $m = \max_{1 \le i \ne j \le n} a_i \times a_j$ 个测试用例；依此类推，随着参数组合覆盖要求的递增，测试用例的数量也呈指数上升. 同时如果某组测试用例能实现对任意 $l(1 \le l \le n)$ 个参数的组合覆盖，则也能实现对任意 $l-1, l-2, \cdots, 2, 1$ 个参数的组合覆盖. 组合测试方法充分考虑了所选择的测试用例应能够以最少的数量最大限度地实现对各种可能性进行覆盖，同时综合考虑测试用例的规模、覆盖能力以及测试成本、时间等因素，来决定测试方案，实现科学有效的测试.

当我们应用组合测试方法对待测系统进行测试时，如果发现了问题，那么如何根据测试结果来确定导致故障发生的原因呢？本文提出了一种基于组合测试的调试方法，该方法根据组合测试的结果来分析运行时发现故障的测试用例以及通过在必要时补充一些附加测试用例进行重新测试的手段来迅速将故障原因锁定在很小的范围内，从而可以为软件的调试和测试工作提供非常方便、有效的线索和参考.

2　相关研究

人们应用组合测试方法对软件进行测试已经有很长时间. 一开始主要是完全将正交实验设计方法应用于软件测试，取得了比较好的效果[1]. 正交实验设计方法是一种比较成熟、有效的测试用例选择方法，它可以实现对各个参数的两两组合的等概率覆盖，而且提供了一整套实验结果分析方法. 然而这种方法依赖于正交表，而正交表的构造还存在很多未解决的难题，特别是对于混合型的正交表，目前还没有比较好的构造方法，所以这给正交实验设计在软件测试中的应用带来较大的局限[1~6]. 通常，在软件测试中，测试用例只要达到对各个参数两两组合的全面覆盖就可以了，并不要求等概率，而且使用正交实验设计产生的测试用例数量在多数情况下要远远多于实现两两组合覆盖所需要的测试用例的数量.

目前，人们关于组合测试的研究主要集中在组合覆盖测试用例的生成方面. Cohen 与 Dalal 等提出了一种基于两两组合覆盖的测试数据启发式生成方法，所产生的测试数据可以根据测试要求实现对系统参数的两两组合覆盖，或者多个参数的组合覆盖[7~17]. 但这种方法无法保证所产生的测试用例最优. Lei 和 Tai 提出一种基于参数顺序的渐进扩充的两两组合覆盖测试数据生成方法，在某些方面具有比较好的特性，同时，这种方法也有很多不足之处[12,13]. 作为启发式方法的一种重要补充，Kobayashi 和 Tsuchiya 等提出了一种代数方法用于生成两两组合覆盖的测试数据，在某些情况下其效果要比启发式方法好[14]. 我们在充分考虑保证生成结果最优的基础上，提出了一种新的关于两两组合覆盖的测试数据生成算法[15].

目前基于组合测试的软件调试方法的研究还未见到，而软件调试是在测试之后不可缺少的关键性工作. 方便、可靠、有效的调试方法能帮助软件测试和调试人员迅速找出故障的原因并消除故障，从而提高软件测试的效率，降低成本. 所以，对于组合测试中调试方法的研究具有相当重要的现实意义.

3　组合测试的调试模型及方法

3.1　基本概念和术语

以下考虑对 SUT 进行组合测试，并假定导致系统运行时出现故障的因素可能是某一个参数或者是某些参数的相互作用. 根据实际要求设计一组测试用例集 St，为了讨论方便，我们先给出测试用例及其模式的有关定义.

定义 1.　称 n 元组 $(v_1, v_2, \cdots, v_n)(v_1 \in T_1, v_2 \in T_2 \cdots v_n \in T_n)$ 为待测系统 SUT 的一个测试用例.

定义 2.　对于待测系统 SUT，取定某 $k(1 \le k \le n)$ 个位置上的参数值后形成的 n 元组 $m = (-, \cdots, v_{l_1}, -, \cdots, v_{l_2}, -, \cdots, v_{l_k}, -, \cdots, -)$，称 m 为待测系统 SUT 的一个 k 值模式，在不引起混淆时，简称为模式，其中“$-$”表示相应位置上参数取值待定.

其中,当 $k=n$ 时对应的模式称为全模式,即为 SUT 的一个测试用例.

定义 3. 设 m_1 和 m_2 是待测系统 SUT 的两个模式,如果模式 m_1 中固定的参数值是 m_2 中固定的参数值的子集,称 m_1 是 m_2 的子模式,m_2 是 m_1 的父模式,记为 $m_1 \prec m_2$.

待测系统 SUT 的一个模式是它自身的子模式,同时也是它自身的父模式.

定义 4. 待测系统 SUT 的某个测试用例 $t=(v_1,v_2,\cdots,v_n)$ 对应的所有的模式组成的集合称为该测试用例 t 的模式集,记为 M_t.

显然,$(M_t,\prec)$ 是一个偏序集,且对于 SUT 的测试用例 $t=(v_1,v_2,\cdots,v_n)$,对应的 M_t 中所有的 1 值模式有 n 个,它们是偏序集 $(M_t,\prec)$ 中的极小元,所有的 2 值模式有 c_n^2 个,…,所有的 k 值模式有 c_n^k 个,全模式只有一个,即为该测试用例,它是偏序集 $(M_t,\prec)$ 中唯一的极大元,也是最大元. 所以,集合 M_t 中一共有 2^n-1 个模式. 在 M_t 中,对于其中某个 k 值模式,它的子模式有 2^k-1 个,父模式有 2^{n-k} 个.

定义 5. 对于系统 SUT 的一组测试用例,如果它们对应的模式集的交集非空,称这组测试用例具有共同模式.

如果某个 k 值模式是一组测试用例的共同模式,说明这组测试用例在某 k 个参数上取值相同.

定义 6. 设 m 是待测系统 SUT 的一个 k 值模式,它的所有子模式组成的集合称为模式 m 的子模式集,记为 M_m. 它的所有父模式组成的集合称为模式 m 的父模式集,记为 P_m.

根据定义 6,我们可以知道待测系统 SUT 的一个 k 值模式 m 可以有 2^k-1 个子模式,即集合 M_m 中有 2^k-1 个元素;由于 $m\in P_m$,所以模式 m 的父模式集 P_m 中的模式个数为

$$1+\sum_{i\in I/I(k)}a_i+\sum_{i\neq j\in I/I(k)}a_ia_j+\cdots+\sum_{i\neq j\neq\cdots\neq l\in I/I(k)}a_ia_j\cdots a_l \tag{1}$$

在式(1)中一共有 $n-k+1$ 项,其中用 I 表示待测系统 SUT 的所有参数在测试用例中对应的编号集合,用 $I(k)$ 表示 k 值模式 m 中 k 个固定的参数在测试用例中对应的编号集合,$I/I(k)$ 表示 k 值模式中未固定值的参数对应的编号集合.

例如,设有一个待测系统,它有 $n=3$ 个参数,每个参数有 3 个取值,不妨都设为 $T_1=T_2=T_3=\{1,2,3\}$. 三元组 $t_1=(2,2,1)$,$t_2=(1,2,1)$ 都是该系统的测试用例,它们对应的模式集分别为

$M_{t_1}=\{(2,-,-),(-,2,-),(-,-,1),(2,2,-),(2,-,1),(-,2,1),(2,2,1)\}$,

$M_{t_2}=\{(1,-,-),(-,2,-),(-,-,1),(-,2,1),(1,-,1),(1,2,-),(1,2,1)\}$,

$M_{t_1}\cap M_{t_2}=\{(-,-,1),(-,2,1),(-,2,-)\}$.

它们具有公共 2 值模式 $m=(-,2,1)$,其中模式 $(-,-,1),(-,2,1),(-,2,-)$ 都是模式 $m=(-,2,1)$ 的子模式,即 $M_m=\{(-,-,1),(-,2,1),(-,2,-)\}$. 模式 $m=(-,2,1)$ 在测试用例 $t_1=(2,2,1,)$ 的模式集 M_{t_1} 中具有父模式 $(-,2,1)$ 和 $(2,2,1)$,在测试用例 $t_2=(1,2,1)$ 的模式集中具有父模式 $(-,2,1)$ 和 $(1,2,1)$,其在该系统中所有的父模式组成的集合 $P_m=\{(1,2,1),(2,2,1),(3,2,1),(-,2,1)\}$.

3.2 基本模型和方法

导致系统发生这样和那样故障的原因有很多,但往往由于某个参数或某些参数的相互作用,在测试时则集中表现在测试用例中的某些取值模式中,因而可以假设系统 SUT 的故障只由某些取值模式引起. 设 SUT 在应用测试用例集 Ts 进行测试时,有 l 个测试用例(记为集合 $Ts1$)在执行中发现某个故障,而其它 $m-l$ 个测试用例(记为集合 $Ts2$)不发生该故障. 假如当某个模式是引发该故障的原因时,其任意父模式也将引发这个故障. 为了分析产生这个故障的原因,我们先引入如下定理.

定理 1. 待测系统 SUT 应用测试用例集 Ts 进行测试时,在执行中发现某个故障的测试用例组成集合 $Ts1$,未发现故障的测试用例组成集合 $Ts2$,如果假设该故障由某个模式 $m=(-,\cdots,v_{l_1},-,\cdots,v_{l_2},-,\cdots,v_{l_k},-,\cdots,-)$ 引起的,且任意模式 $m'\in P_m$ 也将导致系统出现该故障,那么一定有 $P_m\cap M_t\neq\varnothing(\forall t\in Ts1)$,$P_m\cap M_t=\varnothing(\forall t\in Ts2)$.

证明. 首先证明 $P_m\cap M_t\neq\varnothing(\forall t\in Ts1)$. 由于假定发现的故障是由模式 m 引起的,并且该模式的任意父模式也将引发这个故障. 所以当一个测试用例 t 在执行时如果发现该故障,t 中一定包含模式 m;否则,与假设矛盾. 所以有 $P_m\cap M_t\neq\varnothing(\forall t\in Ts1)$.

同理,如果模式 $P_m\cap M_t\neq\varnothing$,即存在 $m'\in P_m$ 导致系统故障,而对于某个测试用例 $t\in Ts2$,同时有 $m'\in M_t$,那么这个测试用例 t 一定会在测试中发生该故障,这与条件矛盾. 所以有 $P_m\cap M_t=\varnothing(\forall t\in Ts2)$. 证毕.

定理 1 表明,导致系统出现某个故障的取值模式 $m=(-,\cdots,a_1,\cdots,a_2,\cdots,a_k,\cdots,-)$ 及其父模式一定在发生该故障的 l 个测试用例中都出现,而在另外 $m-l$ 个没有发生该故障的测试用例中不会出现. 这个结果是基于假设:如果模式 m 引起某个故障,那么它的任意父模式也引起这个故障. 这个假设在有些情况下不一定成立,但为了说明问题,我们以

下讨论都基于这个假设.

根据定理 1,对于待测系统 SUT 我们可以得出如下推论.

推论 1. SUT 的模式 $m=(-,\cdots,a_1,\cdots,a_2,\cdots,a_k,\cdots,-)$ 引发某个故障,对于任意测试用例 t,如果有 $m\in M_t$,那么测试用例 t 也将导致系统出现这个故障.

推论 2. 待测系统 SUT 应用测试用例集 Ts 进行测试时,在执行中发现某个故障的测试用例组成集合 $Ts1$,未发现故障的测试用例组成集合 $Ts2$,如果待测试系统 SUT 的某个故障是由多个取值模式引起的,即一个模式集 $M=\{m_1,m_2,\cdots\}$,那么 $M\cap\bigcup_{t\in Ts1}M_t\neq\varnothing$,且 $M\cap\bigcup_{t\in Ts2}M_t=\varnothing$.

推论 2 说明引起待测系统 SUT 某个故障的这些取值模式一定只会出现在发生该故障的 l 个测试用例中,而不会出现在另外 $m-l$ 个没有发生该故障的测试用例中.

推论 3. 待测系统 SUT 应用测试用例集 Ts 进行测试时,在执行中发现某个故障的测试用例组成集合 $Ts1$,未发现故障的测试用例组成集合 $Ts2$,如果设 $M=\bigcap_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t$,那么对于任意 $m\in M$,m 最有可能就是引起系统 SUT 故障的模式.

推论 3 中的集合 M 是从发生某个故障的 l 个测试用例中找到某些共同的取值模式组成的集合,这些共同的取值模式只在这 l 个测试用例中都出现,而在另外 $m-l$ 个没有该故障的测试用例中不出现,M 中的取值模式都是有可能引起系统 SUT 出现故障的因素.

推论 4. 待测系统 SUT 应用测试用例集 Ts 进行测试时,在执行中发现某个故障的测试用例组成集合 $Ts1$,未发现故障的测试用例组成集合 $Ts2$,若 $m\in\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t$,则模式 m 可能是引起故障的因素.而若 $m\in\bigcup_{t\in Ts2}M_t$,则模式 m 不可能是引起故障的因素.

基于以上定理和推论,要确定故障因素,首先,要找出 $M_1=\bigcap_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t$,该集合包含了最有可能引发这个故障的因素.其次,找出 $M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t$,这里包含了引发 l 个测试用例出现该故障的所有可能的因素.显然 $M_1\subseteq M_2$,在一般情况下,这两个集合中的元素比较多,需要进行进一步精简和确认.

为了进一步确定出是哪些取值模式导致了软件故障,我们需要为发生某个故障的每个测试用例设计 n 个附加测试用例:$t_1=(*,v_{2i_2},\cdots,v_{ni_n})$,$t_2=(v_{1i_1},*,\cdots,v_{ni_n})$,$\cdots$,$t_n=(v_{1i_1},v_{2i_2},\cdots,v_{(n-1)i_{n-1}},*)$,其中"$*$"号表示我们可以用任意不同于原来测试用例$(v_{1i_1},v_{2i_2},\cdots,v_{ni_n})$相应位置的参数的取值来代入.这样需要产生 $n\times l$ 个测试用例,我们称这组测试用例为附加测试用例集,记为 CTs.运行这组测试用例,记仍然发生该故障的测试用例组成的集合为 $CTs1$,不发生该故障的测试用例组成的集合记为 $CTs2$,显然 $CTs1\cup CTs2=CTs$.根据这个结果,我们就可以确定出很小范围的导致故障的因素,即

$$M_1=\bigcap_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t-\bigcup_{t\in CTs2}M_t,$$
$$M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t-\bigcup_{t\in CTs2}M_t,$$

此时亦有 $M_1\subseteq M_2$.

根据定理 1,当系统故障由一个原来发生某个故障的测试用例中某两个参数的一个取值组合引起的时候,上面设计的相应的 n 个测试用例中一定至少有 $n-2$ 个包含这个取值模式,所以,当用这 n 个测试用例对软件系统进行测试时,至少有 $n-2$ 个测试用例会使系统发生该故障.依此类推,可以得到如下定理.

定理 2. 如果系统中某个故障是只由测试用例$(v_{1i_1},v_{2i_2},\cdots,v_{ni_n})$中某 k 个值$(2\leq k\leq n)$的相互作用导致的,即某一个 k 值模式引起的,且相应构造的 n 个测试用例为:$(*,v_{2i_2},\cdots,v_{ni_n})$,$(v_{1i_1},*,\cdots,v_{ni_n})$,$\cdots$,$(v_{1i_1},v_{2i_2},\cdots,v_{n-1i_{n-1}},*)$,其中"$*$"号我们用不同于原来测试用例$(v_{1i_1},v_{2i_2},\cdots,v_{ni_n})$相应位置的参数的任意值代入,那么,在用这 n 个测试用例进行测试时,至少有 $n-k$ 个会使系统出现该故障.

证明. 当测试用例$(v_{1i_1},v_{2i_2},\cdots,v_{ni_n})$中某个 k 值模式$(2\leq k\leq n)$的引发系统故障时,依据测试用例$(v_{1i_1},v_{2i_2},\cdots,v_{ni_n})$生成的附加测试用例中一定有一个会包含这个 k 值模式,根据假设,所有包含这个 k 值模式的测试用例都会引发同样的故障,因此在用这 n 个测试用例进行测试时,至少有 $n-k$ 个会使系统出现该故障. 证毕.

运行了附加测试用例之后,根据其中无该故障产生的测试用例,我们可以将集合 M_1 和 M_2 中的元素大大减少,从而将错误的原因最终锁定在极少的几种取值组合上.如果在组合测试中发现多个故障,以上原理和方法对其中每个故障都可适用,故在文中我们只对一个故障的情况进行讨论.

将 M_1 和 M_2 中的模式按照其中固定的参数值个数进行排列,即把 1 值模式,2 值模式,$\cdots$,k 值模式分别放在一起.

定理 3. 若 k 值模式 m 是 M_1 中的最大元,则 $M_m=M_1$,且 m 在 M_2 中含有 $2^{n-k}-1$ 个父模式;若模式 m 是 M_2 中的一个 k 值模式,则在 M_2 中含有该模式 m 的 2^{n-k} 个父模式.

定理 4. 对于以$\prec$为序的偏序集M_1和M_2，M_1中的所有极小元和M_2中的所有极小元是最有可能导致系统故障的因素.

证明. 用反证法. 假设导致系统发生某个故障的k值模式m既不是M_1中的极小元，又不是M_2中的极小元. 设k值模式m不是$M_1(M_2)$中的极小元，那么它在$M_1(M_2)$中一定存在子模式，该子模式不导致系统出现故障，根据定理 2，当我们使用附加测试用例CTs对系统进行测试时，一定有k个测试用例不包含模式m，但这k个测试用例将包含模式m的任意子模式(m自身除外)，即m的任意子模式不可能在$M_1(M_2)$中，因此假设不成立. 所以$M_1(M_2)$中的极小元是导致系统故障的因素. 证毕.

根据定理 4，要确定出导致系统故障的因素，只要找出M_1和M_2中的极小元就可以了. 由于$M_1 \subseteq M_2$，M_1中的极小元也一定是M_2中的极小元，但反之不一定成立. 所以最后问题就可以归结为：M_2中的极小元是最有可能导致系统故障的因素.

4 实例分析

假设我们要测试一个交换机的打通电话功能，而且以 4 个参数作为这个测试模型的输入，分别为呼叫种类(Call Type)，资费方式(Billing)，接入方式(Access)和状态(Status). 对于这 4 个参数变量的每一个参数均可以有 3 种不同的状态，如表 1 所示. 如果要把表 1 中的所有情况都测试到，那么将需要$3^4=81$个测试用例来遍历所有的测试情况. 我们认为 81 次测试的工作量太大，而且没有必要进行完全测试. 使用组合设计方法可以得到所示的这样一组测试用例. 表 2 中每一行都给出了一个测试用例，实现了对每个两参数间所有变量组合的完全覆盖，而且在这个例子中组合出现的频率都是相同的，都只出现一次.

表 1 待测系统 SUT 的测试需求

Call Type	Billing	Access	Status
C1 Local	B1 Caller	A1 Loop	S1 Success
C2 Long distance	B2 Collect	A2 Isdn	S2 Busy
C3 International	B3 Free call	A3 Pbx	S3 Blocked

表 2 组合测试的测试用例

Call Type	Billing	Access	Status
Local	Collect	Pbx	Busy
Long distance	Free call	Loop	Busy
International	Caller	Isdn	Busy
Local	Free call	Isdn	Blocked
Long distance	Caller	Pbx	Blocked
International	Collect	Loop	Blocked
Local	Caller	Loop	Success
Long distance	Collect	Isdn	Success
International	Free call	Pbx	Success

当我们用这组测试用例对交换机进行测试后，如果没有发现问题，那么就说明交换机功能在这个水平上的测试是合格的. 当然不能因此就说交换机肯定没有问题，因为任何测试只能证明被测对象的不完美性，而无法证明它的完美性. 如果这组测试用例中有一个或几个运行时发生了故障，那么作为测试应该说是比较成功的，因为经过测试我们发现了系统故障. 接下来的重要工作就是分析故障的原因，对它进行定位，并排除故障. 我们不能撇开前面所做的测试工作，泛泛地去重新分析、检查系统，而应该以前面所做的测试工作为基础，进行全面分析. 如果能迅速找出错误原因，将大大提高软件测试的效率，缩短时间，并降低成本.

不失一般性，下面我们仅考虑当用表 2 中这组测试用例对交换机进行测试时，系统运行时发生某个故障的测试用例组成的集合$Ts1$(见表 3)，即表 2 中第 2 个和第 6 个测试用例. 不发生该故障的测试用例集合$Ts2$是表 2 中剩下的 6 个测试用例.

表 3 发生某个故障的测试用例集 $Ts1$

Call Type	Billing	Access	Status
International	Collect	Loop	Blocked
Long distance	Free call	Loop	Busy

根据上一节的讨论，首先找出集合M_1和M_2：

$$M_1=\bigcap_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t=\varnothing,$$

$$M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t=\{(\text{International},\text{Collect},-,-),(\text{International},-,\text{Loop},-),(\text{International},-,-,\text{Blocked}),(-,\text{Collect},\text{Loop},-),(-,\text{Collect},-,\text{Blocked}),(-,-,\text{Loop},\text{Blocked}),(\text{International},\text{Collect},-,\text{Blocked}),(\text{International},\text{Collect},\text{Loop},-),(-,\text{Collect},\text{Loop},\text{Blocked}),(\text{International},-,\text{Loop},\text{Blocked}),(\text{International},\text{Collect},\text{Loop},\text{Blocked}),(\text{Long distance},\text{Free call},\text{Loop},\text{Busy}),(-,\text{Free call},\text{Loop},\text{Busy}),(\text{Long distance},-,\text{Loop},\text{Busy}),(\text{Long distance},\text{Free call},-,\text{Busy}),(\text{Long distance},\text{Free call},\text{Loop},-),(-,-,\text{Loop},\text{Busy}),(\text{Long distance},-,-,\text{Busy}),(\text{Long distance},\text{Free call},-,-),(-,\text{Free call},-,\text{Busy}),(\text{Long distance},-,\text{Loop},-),(-,\text{Free call},\text{Loop},-)\}.$$

M_1是空集，M_2中的元素是所有可能导致系统出现这个故障的因素. 在这种情况下，单个参数不是导致系统出错的原因，因为单个参数的某个取值多次出现在其它测试用例中，例如这个出错的测试用例中参数 Call Type 的取值 International 就在其它两个无故障的测试用例中出现过，因此参数 Call

Type 的取值 International 不会成为系统出错的原因.

为了进一步找出导致系统故障的因素,需要设计一个附加测试用例集 CTs,如表 4,其中 * 号表示相应位置的参数值取任意一个不同于原来出错测试用例在相应位置的值(在本例中取括号内的值).

表 4　附加测试用例集 *CTs*

	Call type	Billing	Access	Status
t_1	*(Long distance)	Collect	Loop	Blocked
t_2	International	*(Free call)	Loop	Blocked
t_3	International	Collect	*(Pbx)	Blocked
t_4	International	Collect	Loop	*(Busy)
t_5	*(Local)	Free call	Loop	Busy
t_6	Long distance	*(Caller)	Loop	Busy
t_7	Long distance	Free call	*(Isdn)	Busy
t_8	Long distance	Free call	Loop	*(Blocked)

用这组测试用例再对交换机进行测试,根据测试结果我们就可以确定出导致故障的原因.我们分成以下几种情况讨论.

情形 1.　这 8 个测试用例的运行都没有发生原先的故障.我们可以计算 $M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t-\bigcup_{t\in CTs}M_t=\{$(International, Collect, Loop, Blocked), (Long distance, Free call, Loop, Busy)$\}$,此时模式(Long distance, Free call, Loop, Busy)和(International, Collect, Loop, Blocked)都是极小元,因此是导致故障的因素.

情形 2.　只有某一个测试用例运行时发生原来的故障,例如是表 4 中第二个测试用例.可以计算 $M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t-\bigcup_{t\in CTs2}M_t=\{$(International, −, Loop, Blocked), (International, Collect, Loop, Blocked), (Long distance, Free call, Loop, Busy)$\}$(其中 $CTs2$ 是表 4 中所有未发生该故障的测试用例组成的集合),则此时导致故障的因素是极小元(International, −, Loop, Blocked), (Long distance, Free call, Loop, Busy).

情形 3.　有某 5 个测试用例测试时发生故障,例如 $Ts1=\{t_3,t_4,t_5,t_6,t_7\}$, $Ts2=\{t_1,t_2,t_8\}$.此时可以计算 $M_2=\bigcup_{t\in Ts1}M_t-\bigcup_{t\in Ts2}M_t-\bigcup_{t\in CTs2}M_t=\{$(International, Collect, −, −), (International, Collect, −, Blocked), (International, Collect, Loop, −), (International, Collect, Loop, Blocked), (Long distance, Free call, Loop, Busy), (−, Free call, Loop, Busy), (Long distance, −, Loop, Busy), (Long distance, Free call, −, Busy), (−, −, Loop, Busy), (Long distance, −, −, Busy), (−, Free call, −, Busy)$\}$ 的极小元(International, Collect, −, −), (−, −, Loop, Busy), (Long distance, −, −, Busy)与(−, Free call, −, Busy)是导致故障的因素.

类似地,我们可以讨论其它各种情形.由此我们看出,在组合测试的基础上,并辅之以一些必要的相关测试用例,我们可以对导致系统故障的因素进行有效的定位;在产生故障的测试用例比较多的情况下,分析过程比较复杂.但由于多数情况下,导致系统出错的测试用例数量很少(这种测试用例对软件测试来说是高质量的测试用例),如果这种导致系统出现错误的测试用例数量多,说明系统质量太低,需要进行广泛的重新分析和检查.所以,不论什么情况,基于组合测试的基础,并辅之以一些必要的新的相关测试用例,对故障原因进行分析是非常重要的,也是非常有效的.

我们通过故障植入的仿真实验方法,将该方法应用于 Linnux 开放源代码和一些经典的例子,如三角形判定、日期判定等程序的故障诊断,取得了较好效果.例如日期判定程序的输入有年月日三个参数,这三个参数中有些组合是非法组合,如 2 月 29 日在非闰年份是非法输入.在这些实验中,利用本文方法可以准确地将引发的软件故障的原因诊断出来.

5　结束语

组合设计方法是一种重要的实验设计方法,在很多领域有着广泛的应用.在软件测试领域应用最为广泛的有正交实验设计方法和两两组合覆盖测试方法.组合设计方法具有以下一些优点:(1)可以根据实际需要用尽可能少的实验次数尽可能多地考察一些影响系统的因素;(2)少数次实验的结果能够反映全面实验的内在规律,具有代表性;(3)能够通过少数次实验结果的分析和处理探求可能更优的实验方案.因而它是一种基于数理统计基础上的科学的方法,近年来在软件测试领域受到了非常广泛的重视、应用和研究.

在应用组合设计产生的测试用例集对软件系统进行测试之后,如果发现故障,那么一方面说明测试工作是成功的,另一方面还需要通过对实验结果的分析和处理,初步确定故障原因,并探求可能更优实验方案,来对故障原因进行进一步的分析、处理和验证.目前人们关于这方面的研究还不多见,虽然在正交实验设计方法中有关于实验结果分析的方法,如直观分析方法、方差分析方法等,但是这些方法并不适合于对软件测试结果的分析.

本文在假设软件系统故障仅由系统的某些取值

模式引起,并且这些模式的任意父模式也引发同样故障的前提下,提出了一种根据测试结果和附加测试用例的测试结果进行故障原因定位的方法.基于本文的模型,该方法可以解决故障原因的诊断问题.但事实上,在实际系统中的情况是非常复杂的,完全有可能出现与本文模型假设相反的情况,即某个模式引发一个系统故障,但它的某些父模式不会引发这个故障,或者引发其它故障.所以,作为组合测试方法的一种辅助调试方法,这种方法具有一定的适用范围,提供的分析结果只能作为一个有益的参考.我们将在以后的工作中进一步研究各种复杂的情况,从而更有效地提高软件测试的效率、降低软件测试的成本.

致 谢 陆建江、周毓明、陈振强、查日军等博士参与了有关讨论,华为公司为本研究提供了很好的应用背景,在此一并致谢.同时对审稿人提出的有益建议表示感谢!

参 考 文 献

1 Heller E.. Using design of experiment structures to generate test cases. In: Proceedings of the 12th International Conference on Testing Computer Software, New York, 1995, 33～41

2 Mandl R.. Orthogonal Latin squares: An application of experimental design to compiler testing. Communications of the ACM, 1985, 28(10): 1054～1058

3 Brownlie R., Prowse J., Phadke M.. Robust testing of AT&T PMX/StarMail using OATS. AT&T Technical Journal, 1992, (71)3: 41～47

4 Phadke M. S.. Quality Engineering Using Robust Design. Englewood Cliffs, NJ: Prentice Hall, 1989

5 Taguchi G.. System of Experimental Design, Quality Resources, 1976. Translation of Jikken Keikakuho, Maurzen Co., Tokyo, 1987

6 West C. H.. Protocol validation—Principles and applications. Computer Networks and ISDN Systems, 1992, 24(3): 219～242

7 Cohen D. M. *et al*.. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 1997, 23(7): 437～444

8 Cohen D. M. *et al*.. The combinatorial design approach to automatic test generation. IEEE Software, 1996, 13(5): 83～87

9 Cohen D. M., Fredman M. L.. New techniques for designing qualitatively independent systems. Journal of Combinatorial Designs, 1998, 6(6): 411～416

10 Williams A. W., Probert R. L.. A practical strategy for testing pair-wise coverage of network interfaces. In: Proceedings of the 7th International Symposium on Software Reliability Engineer, Antwerpen, Belgium, 1997, 246～254

11 Cohen D. M., Dalal S. R., Kajla A., Patton G. C.. The automatic efficient tests generator. In: Proceedings of the 5th International Symposium on Software Reliability Engineering, IEEE, Los Alamitos, California, 1994, 303～309

12 Lei Y., Tai K. C.. In_Parameter_Oder: A test generation strategy for pairwise testing. Department of Computer Science, North Carolina State University, Raleigh, North Carolina: Technical Report TR-2001-03, 2001

13 Tai K. C., Lei Y.. A test generation strategy for pairwise testing. IEEE Transactions on Software Engineering, 2002, 28(1): 109～111

14 Kobayashi N., Tsuchiya T., Kikuno T.. A new method for constructing pair-wise covering designs for software testing. Information Processing Letters, 2002, 81(2): 85～91

15 Nie Chang-Hai, Xu Bao-Wen. An automatic Black-box test case generation algorithm based on interface parameters. Chinese Journal of Computers, 2004, 27(3): 382～388(in Chinese)
(聂长海,徐宝文.基于接口参数的黑箱测试用例自动生成算法.计算机学报, 2004, 27(3): 382～388)

XU Bao-Wen, born in 1961, Ph.D., professor, Ph. D. supervisor. His research interests include in programming language, software engineering, parallel and network software, acquisition technique on knowledge and information etc.

NIE Chang-Hai, born in 1971, Ph. D., associate professor. His research interests include software engineering and software testing, fussy information processing and neutral network etc.

SHI Liang, born in 1979, Ph. D. candidate. His research interests include in software analysis and testing.

CHEN Huo-Wang, born in 1936, member of Chinese Academyof Engineering, professor of Computer Science and logician.

Background

This paper is supported by the National Natural Science Foundation of China project (No. 60373066), which title is the Research on Software Testing Technology Based on Combinatorial Cover. The project mainly research on these fields: combinatorial cover method and its scientific effectiveness for software testing, the existence and the generation algorithm for several minimal combinatorial cover table, some technique for software fault diagnosis and debugging based on combinatorial testing, the application of the technique for software testing based on combinatorial coverage in web testing and software configuration. Authors have made some work on the test case generation algorithm and the applications. This paper focuses on the research of software fault diagnosis and debugging based on combinatorial testing.