摘要

动态随机测试(DRT)策略利用软件的控制理论改进了传统的随机测试策略。DRT 策略的主要思想是在测试期间利用历史的测试信息动态的改变测试剖面，使得具有较高故障检测能力的分区更有可能被选到。但是 DRT 策略没有考虑每次根据测试结果调整概率的幅度应该是不同的，并且每一个分区被选择的概率容易受其它分区测试结果的影响，这将导致具有较高故障检测能力的分区不容易突显出来。该策略的测试效率还受到分区数目以及初始测试剖面的影响。这篇文章提出两种测试策略命名为 MDRT、RDRT 策略。MDRT 策略利用 Markov 状态转移矩阵，将分区当成状态，使得某一个分区被选择的概率只跟该分区内的测试用例的执行情况有关并且概率的调整幅度由当前分区的被选择概率决定。RDRT 策略利用奖励惩罚的思想改善 DRT 策略在输入域失效率低时测试效率不高的情况。通过实验探究初始剖面以及分区数目对 DRT 策略的影响。数据表明当分区数目较多，初始概率分布为均等分布时，DRT、MDRT、RDRT 策略具有较高的故障检测效率，并且 MDRT、RDRT 策略的测试效率比随机测试、随机分区测试以及动态随机测试的效率高。

1 背景介绍

随机测试[1]以及分区测试[2,3,4]是两个非常著名的测试策略。在传统的随机测试中，按照一致或者不一致的概率分布从软件输入域中选择测试用例并执行。分区测试涉及到一簇的测试技术：状态测试、数据流测试、分枝测试、变异测试等，任何的一个输入域的子域，都需要从中挑选至少一个测试用例。

Cai[5,6]等人将随机测试与分区测试结合，提出了随机分区测试策略。该策略假设待测软件的输入域被分为m个子分区。随机分区测试策略首先根据测试剖面$\{p_1, p_2, \ldots, p_m\}$选择一个分区$c_i$。然后在$c_i$中随机地选择一个测试用例执行。在整个的测试过程中测试剖面的大小不变。

在随机分区测试策略中，一个分区对应的选择概率在整个的测试过程中是不变的，这一点可能不总是好的。因为引起故障的输入在输入域中趋向于聚簇在连续的区域[7-9]，也就是说存在一些分区更可能揭示软件中的故障。Cai 等人依据这一想法，利用软件的控制理论[10]提出了动态随机测试策略（DRT）[11]以改进传统的随机测试与随机分区测试策略。软件的控制理论探索软件工程理论与控制理论

相互作用的关系，被用来解决软件工程中的问题。DRT 策略的主要特点是在测试的过程中根据每一次测试用例的执行结果动态改变测试剖面：假设存在一个分区c_i，若该分区中的一个测试用例揭示了软件中的故障，那么认为该分区具有较高的故障检测能力，因此增大该分区被选择的概率，即$p_i + \varepsilon$。如果这个测试用例没有检测出故障，减小该分区被选择的概率，即$p_i - \delta$。

但是该策略仍然存在一些不足：

1. **找到高故障检测能力的分区的速度慢。**由于在测试过程中参数的取值很小，引起故障的输入所在的分区增加的概率幅度不明显，因此具有更高检测能力的分区很难在短时间内突显出来。特别是软件输入域的失效率很小的情况下，大多数的输入不能引起软件中的故障，因此能够造成故障的输入所在的分区难以保持较高的被选择概率。
2. **不同分区的测试结果相互影响。**某个分区被选择的概率受其它分区测试结果的影响,使得该分区被选择的概率无法准确地反映该分区真正的故障检测能力。将软件的输入域按照一定的方式划分为若干分区之后，每个分区的故障检测能力是独立的。但是在传统的 DRT 算法中每个分区的故障检测能力，随着其它分区的测试用例检测结果发生改变。
3. **每次调整概率的幅度相同。**直觉上，每次调整分区的概率幅度应当根据当前分区的概率大小，而不应该是一致的：如果当前分区被选择的概率比较大说明该分区在理论上具有更高的故障检测能力，但是并不能保证该分区的每一个测试用例都能揭示软件的故障，因此该分区的测试用例没有揭示故障时的调整幅度应当比被选择的概率较小分区没有检测出故障时的调整幅度要小。
4. **分区数目对测试策略有影响。**在黑盒测试中，同一个项目的不同分区策略以及同一个项目的相同分区策略，分区的数目也可能不同，不同数目的分区导致算法的检测效率发生改变。
5. **初始剖面对测试策略有影响。**一般情况下，初始测试剖面$\{p_1, p_2, \ldots, p_m\}$中$p_1 = p_2 = \cdots = p_m$，即初始条件下每一个分区被选择的概率是均等的。当软件的输入域失效率高时，即存在很多测试用例能够造成软件故障，此时每一个子分区的故障检测能力可能相差不大。但是当软件的输入域失效率很低时，即只有很少的测试用例能够揭示故障中的故障，此时子分区之间的故障检测能力很有可能相差很大，甚至一些分区不具备故障检测能力。因此本文提出一种根据子分区测试用例数目占全部测试用例的百分比作为初始剖面的概率分布，比较两种情况下，哪一种初始概率

分布具有较高的故障检测率。

因此，加速找到具有高故障检测率的分区是一个很自然的想法弥补 DRT 策略在 1 方面存在的不足。本文通过为每一个分区绑定奖励因子与惩罚因子，并根据测试对象本身信息设定惩罚上限，如果某一个分区中的测试用例揭示了软件中的故障，那么该分区的奖励因子自增，惩罚因子设置为 0 并且下次依然在该分区中随机选择测试用例直到没有揭示软件中的故障，然后根据奖励因子确定该分区被选择概率的增长幅度。当某一个分区中的测试用例没有揭示软件中的故障时，该分区绑定的惩罚因子自增，然后调整测试剖面选择下一个分区。如果某一个分区的惩罚因子大于或者等于惩罚上限，意味着该分区具有很小的故障检测率，甚至不具备故障检测能力，因此将该分区被选择的概率记为 0。将这种测试策略命名为 RDRT。

为了弥补 2、3 方面的不足，本文用 Markov 链的状态转移矩阵思想，将分区作为状态，选择测试用例并执行当作在该状态下的行动，那么根据在某一个状态下测试用例执行情况调整转移到其它状态的概率。由此某一个分区被选择的概率只受该分区内测试用例执行结果的影响，不受其它分区测试用例执行结果的影响。并且在设计根据测试用例的执行结果调整分区被选择的概率时，本文对具体算法进行改进使得被选择概率大的分区没有揭示软件中的故障时概率调整幅度小，揭示故障时概率调整幅度大；被选择概率小的分区没有揭示软件中的故障时概率调整的幅度大，揭示故障时概率调整的幅度小。将这种测试策略命名为 MDRT 策略。

针对问题 4 本文根据五个程序的规格说明运用等价类划分法得到一种分区方式，然后将某一个分区再进行更细粒度的划分，得到分区数目更多的另一种分区方式。根据实验结果对比不同分区数目对各个测试策略的影响。

针对问题 5 本文采取均等的概率分布和不均等的概率分布作为初始剖面。不均等的初始概率分布的设置方式是根据每一个分区内的测试用例数目占输入域所有测试用例总数的百分比作为初始条件下某一分区被选择的概率。

本文通过实验发现在较多分区数目以及初始剖面为均等的概率分布时，MDRT、RDRT 策略的测试效率比 DRT、RPT、RT 策略的效率高。

文章接下里的组织方式是：第二部分展示了相关工作。第三部分介绍了 MDRT 策略、RDRT 策略。第四部分展示了实验设置以及实验结果。数据分析和讨论在第五部分展示。第六部分展示了实验结论以及将来的工作。

2 相关工作

很多的工作对随机测试与分区测试做了研究。Myers[12]认为：“随机测试是所有测试策略中效率最低的”。然而 Duran[13]认为：“随机测试在很多程序中表现良好并且有时候可以用较小的代价揭示相对难发现的缺陷”。Weyuker[14]经过研究之后发现：分区测试可能是一个卓越的测试策略也可能是一个低效率的测试策略，分区测试的效率很大程度上取决于如何将产生错误输出的输入集中在某个或者某些分区中。Hamlet[3]认为成功的分区测试不能激发测试人员对软件质量的信心。Chen[4]认为当存在较高失效率的分区时，分区测试具有较高的检测能力。Gutjahr[2]认为在不确定条件下，分区测试比随机测试效率更高。

Cai 结合了随机测试和分区测试的特点提出了分区随机测试（RPT）策略。RPT 策略首先根据测试剖面选择分区c_i，然后在c_i中随机选择测试用例。Cai[10]利用软件控制理论提出了适应性测试策略(AT)，该策略的测试效率相对于随机测试、分区测试有很高的改进[5, 10]。但是 AT 策略在实际中需要消耗大量的时间。为了解决这一问题 Cai 提出了动态随机测试策略(DRT)。在 DRT 策略中，测试剖面根据测试的反馈信息动态地改变。

Lv 在[15]中假设软件输入域的失效率已知、各个分区的失效率已知、测试过程中各个分区失效率保持不变以及测试用例执行之后放回原来的分区之中，通过理论分析的方式得到了ε/δ的最佳取值范围。然而实际中很难知道输入域的失效率以及各个分区的失效率大小。Yang 在[16]中通过在实验的过程中统计每一个分区的成功检测率，然后调整ε/δ的取值，在软件的输入域失效率比较大的情况下，效果比较好。

3 基于 Markov 链的动态随机测试和基于奖惩机制的动态随机测试

这个章节，先介绍动态随机测试(DRT)策略，然后介绍 Markov 基本理论，接着介绍基于 Markov 链的动态随机测试(MDRT)策略，最后介绍基于奖惩机制的动态随机测试(RDRT)策略。

3.1 DRT 策略

Cai 最先在[8]中提出了动态随机测试，这里将完整的算法策略展示如下。

将待测软件的输入域划分为m个不相交的分区：$C_1, C_2, \ldots, C_m$，每一个分区中有$k_1, k_2, \ldots, k_m$个测试用例。

初始化参数ε, δ，并且$\varepsilon > 0, \delta > 0$。

根据每一个分区所对应的概率p_i随机选取一个分区C_i, 在这里$p_1 + p_2 + \cdots + p_m = 1$。

随机地从C_i中挑选一个测试用例TC。

如果测试用例TC揭示了软件中的故障，就增大测试用例TC所在分区被选择的概率，同时减小其它分区被选择的概率，并把缺陷移除。

$$p_j = \begin{cases} p_j - \dfrac{\varepsilon}{m-1}, p_j \geq \dfrac{\varepsilon}{m-1} \\ 0 \qquad\quad , p_j < \dfrac{\varepsilon}{m-1} \end{cases} \quad \text{其中} j \neq 1$$

$$p_i = 1 - \sum_{j \neq i} p_j$$

如果测试用例TC没有找到缺陷，就减少c_i被选中的概率p_i，同时增大其它分区被选中的概率p_j。

$$p_i = \begin{cases} p_i - \delta, p_i \geq \delta \\ 0 \qquad , p_i < \delta \end{cases}$$

$$p_j = \begin{cases} p_j + \dfrac{\delta}{m-1}, p_i \geq \delta \\ p_j + \dfrac{\delta}{m-1}, p_i < \delta \end{cases} \quad \text{,其中} j \neq i$$

检查停止条件，如果不满足，则跳转执行步骤 3，如果满足则停止测试。

3.2 Markov 链理论概述

Markov 随机过程称为 Markov 链，具备“无后效应”，即要确定过程将来的状态，知道它此刻的情况就够了，并不需要对它以往状况的认识。对于有限个或可列个值$E_1, E_2, \cdots, E_n$，以$\{1,2, \cdots, n\}$来标记$E_1, E_2, \cdots, E_n$并称它们为过程的状态，对于任意的$n \geq 0$及状态$i_1, i_2, \cdots, i_{n-1}, i, j$有：$P\{X_{n+1} = j | X_1 = i_1, X_2 = i_2, \cdots, X_n = i_n\} = P\{X_{n+1} = j | X_n = i\}$。可见，一旦 Markov 链的初始分布$P\{X_0 = i_0\}$给定，其统计特性完全由条件概率$P\{X_n = i_n | X_{n-1} = i_{n-1}\}$决定。

假设状态空间$S = \{1,2, \cdots, m\}$。

定义 1（转移概率）：条件概率P$\{X_{n+1}=\mathrm{j}|X_n=\mathrm{i}\}$为 Markov 链的一步转移概率，简称转移概率。

定义 2（时齐 Markov 链）：当 Markov 链的转移概率P$\{X_{n+1}=\mathrm{j}|X_n=\mathrm{i}\}$只与状态i, j有关，而与n无关时，称 Markov 链为时齐的，并记为$p_{ij}=P\{X_{n+1}=j|X_n=i\}(n\geq 0)$。

由定义 2 知道我们可以将$p_{ij}(i,j\in S)$排成一个矩阵的形式，令

$$P=\left(p_{ij}\right)=\begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{pmatrix}$$

则称P为转移矩阵。

转移矩阵 P 具有如下性质：

⑴ $p_{ij}\geq 0,(i,j\in \mathrm{S})$;

(2)$\sum_{j\in s} p_{ij}=1,(\forall i\in S)$;

3.3 MDRT 策略

MDRT 策略合了传统随机算法与分区算法的特点，并引入软件的控制理论。假设软件的输入域有k个测试用例，被划分到m个不相交的分区$C_1,C_2,\ldots,C_m$中并且每一个分区有k_i个测试用例，$k_1+k_2+\cdots+k_m=k$。如果将每个时间点t$(\mathrm{t}=0,1,2,\cdots)$测试用例所在的分区作为时刻 t 测试系统所处的状态，则整个状态空间为$\mathrm{S}=\{s_t,\mathrm{t}\geq 0\}=\{C_1,C_2,\ldots,C_m\}$。根据在$s_t=C_i$状态下测试用例的执行结果可以计算调整到状态$s_{t+1}=C_j$的转移概率。如果$s_t=C_i$状态下检测出软件中存在缺陷，那么就增大下一时刻转移到$s_{t+1}=C_i$的概率，同时减小转移到其它状态的概率；反之，如果在$s_t=C_i$状态下没有检测出软件中存在缺陷那么就减少下一步转移到$s_{t+1}=C_i$状态的概率，同时增大转移到其它状态的概率。另外，根据当前状态下测试用例的执行结果每次增大或者减少的转移概率的幅度应该是不同的。一般情况下，一方面即便某一个分区具有较强的检测出软件存在故障的能力，也不可能每一次在该分区中选择测试用例都会检测出故障。另一方面如果某一个分区被选择的概率比较大说明在以往的测试过程中较多的检测出了软件中的缺陷或者理论上有较大的可能揭示软件中的故障。因此在测试过程中增大或者减少某一

分区被选择的概率幅度时应当与该分区当前被选择概率有关：如果当前分区被选择的概率较大，那么增大该分区被选择的概率幅度就越大，减小该分区被选择的幅度就越小。整个过程的状态转移可以用转移矩阵表示。在整个软件测试过程中，将每一个时间点选取测试用例并执行作为一次决策行动，则行动全体组成整个行动空间$\mathrm{A}=\{a_t,\mathrm{t}\geq 0\}=\{1,2,\cdots,\mathrm{k}\}$，并且每个时间点的状态$S_t$和所采取的行动都会影响到下一个时间点$\mathrm{t}+1$的状态$S_{t+1}$。因此，整个测试过程形成一个 Markov 决策过程。

开始时刻初始化测试剖面p'，则初始转移矩阵$\mathrm{P}=p_{ij}=(p',p',\ldots,p')'$其中$(\mathrm{i}=1,2,\ldots,\mathrm{m};\mathrm{j}=1,2,\ldots,\mathrm{m})$。

MDRT 测试策略可以分为如下步骤：

步骤 1：根据当前分区到其它分区所对应的转移概率p_{ij}随机选取一个分区C_i(第一次根据测试剖面选择分区)，在这里$p_{i1}+p_{i2}+\cdots+p_{im}=1$。转步骤 2。

步骤 2：等概率随机地从分区C_i中选取一个测试用例 TC, 转步骤 3。

步骤 3：执行选中的测试用例 TC：

如果测试用例 TC 揭示了软件中的故障，就增大测试用例 TC 所在状态转移到本身的概率p_{ii}，同时减小转移到其它状态的概率$p_{ij}(i\neq j)$，并把缺陷移除：

$$p_{ij}=\begin{cases}p_{ij}-\dfrac{\gamma\times p_{ii}}{m-1}, & p_{ij}\geq\dfrac{\gamma\times p_{ii}}{m-1}\\ p_{ij}, & p_{ij}<\dfrac{\gamma\times p_{ii}}{m-1}\end{cases}\quad(i\neq j)$$

$$p_{ii}=1-\sum_{i\neq j}p_{ij}$$

如果测试用例 TC 没有找到缺陷，就减少测试用例 TC 所在状态转移到本身的概率p_{ii}，同时增大转移到其它状态的概率$p_{ij}(i\neq j)$：

$$p_{ij}=p_{ij}+\frac{\tau\times p_{ij}}{m-1}$$

$$p_{ii}=p_{ii}-\frac{\tau\times(1-p_{ii})}{m-1}$$

更新转移矩阵：

$$\begin{pmatrix}*&*&*&\cdots&*\\ \vdots&\vdots&\vdots&\cdots&\vdots\\ p_{i1}&\cdots&p_{ii}&\cdots&p_{im}\\ \vdots&\vdots&\vdots&\ldots&\vdots\\ *&*&*&\cdots&*\end{pmatrix}$$

步骤 4：检查测试停止条件，如果不满足则转步骤 1；如果满足则停止测试。

3.4 RDRT 策略

在软件中不存在难检测的故障时，Yang 在[16]中提出的 A-DRT 策略的测试效率比传统的 DRT 策略有明显的提高；但是软件存在难检测的故障时，效果不理想。当软件的失效率高时，软件内的缺陷很多测试策略都能用较小的代价揭示出来。但是当软件的失效率低时，不同检测策略的效率差别很大。在以往的测试活动中发现，当失效率很低时 DRT 策略的测试效率相对于 RT 策略没有提高或者提高不明显。直觉地，引起故障的输入在输入域中趋向于聚簇在连续的区域，即存在一个或者少数分区具有较高的检测能力。因此在软件输入域的失效率较低时，往往一些分区内不具备揭示软件中缺陷的能力或者具备较小的检测能力。另一方面由于每次调整概率的幅度很小，并且某一个分区被选择的概率易受到其它分区的测试结果的影响，使得那些不具备或者具备很小的检测能力的分区仍然被不断的选择，最终具有较高检测能力的分区不容易在短时间内突显出来. 因此 DRT 策略在软件输入域的失效率低时，测试效率不高。为了缓解这一问题，本文提出了基于奖惩机制的动态随机测试策略(RDRT)，该策略旨在加速测试的过程：如果分区C_i内的测试用例揭示了软件中的缺陷，下一次仍在该分区内选择测试用例并且该分区绑定的奖励因子自增一次，对应的惩罚因子清 0，直到该分区中的测试用例没有检测出软件中的缺陷，奖励因子清 0，惩罚因子+1。奖励因子越大该分区对应的概率增加的越多。相反地，如果存在这样一个分区：累计 n 次选中该分区，但是该分区中的测试用例均没有揭示出软件中存在缺陷，那么就认为该分区具有较低的检测能力，甚至不具备检测能力，让该分区对应的选择概率为 0。

假定软件测试的输入域中的测试用例划分到 m 个不相交的分区中，输入域中共有k个测试用例，用$C_1, C_2, \ldots, C_m$来表示这 m 个分区，每一个分区有 k_i个测试用例。初始每个分区的奖励因子$reward_i = 0$，惩罚因子$\mathrm{punishment}_i = 0$，惩罚上限$\mathrm{boundary} = \mathrm{Z}$。

步骤 1：根据当前各个分区所对应的概率p_i选取分区C_i，其中$p_1 + p_2 + \cdots + p_m = 1$。

步骤 2：等概率随机地从分区C_i中选取一个测试用例 TC。

步骤 3：执行选中的测试用例 TC。如果测试用例 TC 揭示了软件中的故障转步骤 4，反之转步骤 5。

步骤 4：分区C_i的奖励因子$reward_i = reward_i + 1$，惩罚因子$\mathrm{punishment}_i =$

0，并移除缺陷。转步骤 2。

步骤 5：分区C_i的惩罚因子$\text{punishment}_i = \text{punishment}_i + 1$。如果$reward_i \neq 0$就增大测试用例 TC 所在分区对应的概率$p_i$，同时减小其它分区被选择的概率$p_j(i \neq j)$：

$$p_j = \begin{cases} p_j - \dfrac{(1+\ln reward_i)\times\varepsilon}{m-1}, & p_j \geq \dfrac{(1+\ln reward_i)\times\varepsilon}{m-1} \\ 0, & p_j < \dfrac{(1+\ln reward_i)\times\varepsilon}{m-1} \end{cases}$$

$$p_i = 1 - \sum_{i \neq j} p_j$$

如果$reward_i = 0$，就减少该分区对应的概率p_i，同时增大其它分区对应的概率$p_j(i \neq j)$，如果该分区的惩罚因子$\text{punishment}_i \geq boundary$则该分区对应的概率$p_i = 0$：

$$p_i = \begin{cases} p_i - \delta, & p_i \geq \delta \\ 0, & p_i < \delta \ \ or \ \ \text{punishment}_i = boundary \end{cases}$$

$$p_j = \begin{cases} p_j + \dfrac{\delta}{m-1}, & p_i \geq \delta \\ p_j + \dfrac{p_i}{m-1}, & p_i < \delta \ \ or \ \ \text{punishment}_i = boundary \end{cases}$$

步骤 6：检查测试停止条件，如果不满足则转步骤 1；如果满足则停止测试。

3.5 Remarks

[10]中的实验可以看出增大ε/δ的值可以提高 DRT 策略的效率，因此让$\varepsilon/\delta = r_M + (r_\Delta - r_M) * K(r_M = 1/\theta_M - 1, r_\Delta = 1/\theta_\Delta - 1$，并且 K 值为 0.8 时 DRT 策略具有更高的故障检测效率。保守地，ε 被设置一个相当小的数$\varepsilon = 0.05$，本实验中θ_M、θ_Δ是根据真实的故障检测率确定的大小。每一个实验对象，DRT 策略和 RDRT 策略的参数大小相同。所有的实验对象 MDRT 策略的参数均设置为$\gamma = \tau = 0.1$。

RDRT 策略的惩罚上限根据具体实验对象的不同设置的具体值也可能不同。在 grep 实验中，由于测试用例的数目比较多并且本实验中的故障较难检测因此惩罚上限设置为 50。基于相似地考虑 gzip 实验的惩罚上限为 30；flex 实验的惩

罚上限为 10；bash 实验的惩罚上限为 50；make 实验的惩罚上限为 10。

初始的测试剖面$\{p_1, p_2, \ldots, p_m\}$应该有测试工程师根据以往的测试经验设定。本文采取两种方式设定初始测试剖面。第一种方式为$p_1 = p_2 = \cdots = p_m = 1/m$；第二种设置方式为$p_i = k_i/k$，$k_i$表示$C_i$分区内的测试用例数目，k表示 SUT 输入域的测试用例总数。这两种分区方式应用于 RPT、DRT、MDRT、RDRT 四种测试策略中。

在 MDRT 策略中通过将当前测试用例所在的分区当成此时所处的状态，根据该测试用例的执行结果计算调整到下一个状态的转移概率。通过转移矩阵使得某一个分区的故障检测能力仅由该分区内的测试用力执行情况决定，不受其它分区测试结果的影响，从而加速了找到具有高故障检测能力分区的进程。通过步骤 3 使得每一次根据某一分区内的测试用例执行情况调整分区概率的幅度由当前分区被选择的概率大小决定：如果当前分区被选择的概率大，增加的幅度就比较大，减少的幅度就比较小；如果当前分区被选择的概率小，增加的幅度相对较小，增加的幅度相对较大。

4 实验设置

4.1 实验对象

为了避免测试策略在特定的程序中具有更高的故障检测能力，本文在 Software artifact Infrastructure Repository(SIR)网站下载了五个真实的程序，每一个程序附带测试用例集以及故障。为了模拟实际的软件测试，本文选取的程序代码行数均大于 5K。

在 SIR 网站中，每一个程序都有不同的版本以及对应的测试用例集和故障。例如 bash 程

序有 6 个可测的版本，本文将 V1 作为 SUT。实验对象的基础信息展示在表 1。

4.2 测试用例和分区

TSL[17]是用来书写测试规格说明书的语言。基于测试规格说明书中的信息产生大量的测试帧。为每一个测试帧中的 choice 指定一个具体的值，可以得到一个具体的测试用例。根据等价类划分策略将规格说明书中规定的相同处理方式组成

一个分区，这样得到一种分区方式。然后根据已经得到分区方式，将该方式下的

表 4.1 实验对象信息

源程序	代码行数	测试版本	测试用例数目	故障数目	测试的故障数目	分区数目
bash	59846	V1	1061	6	6	5，4
flex	10459	V1	567	19	3	6，3
grep	10068	V1	809	19	5	5，4
gzip	5680	V1	217	16	5	4，3
make	35545	V1	793	19	2	5，3

某一分区进行更细力度地划分得到分区数目更多地分区方式。每一个实验具体的分区数目展示在表 4.1 的最后一列。

4.3 测试的故障选择

对于每一个实验对象，本实验先用随机测试策略对每一个故障进行测试，重复 50 遍得到每一个故障被检测到时用的测试用例数目的平均值。比较每一个故障用到的测试用例数目然后取相对难杀死的故障(检测到该故障用到的测试用例数目较多)，每一个实验对象测试的故障数目展示在表 4.1 的倒数第二列。

4.4 测试策略

本实验检测了五个策略：传统的随机测试策略(RT)，随机分区测试策略(RPT)，动态随机测试策略(DRT)，以及本文提到的基于 Markov 链的动态随机测试策略(MDRT)和基于奖惩机制的动态随机测试策略(RDRT)。在测试过程中，如果一个故障被检测到就立即移除该故障。测试的停止条件为所有的故障都被揭示和移除。

4.5 指定随机数种子

计算机产生的随机数是伪随机,如果不指定随机数种子，它将以当前时间为种子随机产生随机数。如果这样做一方面导致实验不可重复，另一方面也使得不同测试策略的差异是随机数产生的还是策略本身产生的无法确定。因此对于同一个实验对象的某一次实验不同测试策略的随机数种子相同。

4.6 度量方法

为了反映五个测试策略的故障检测效率，在实验中运用了 3 个度量标准：

1. F，揭示第一个故障需要的测试用例数目。F的均值用$\bar{F}$表示
2. NF，揭示第一个故障之后到揭示第二个故障需要的测试用例数目。NF的均值用$\overline{NF}$表示。
3. T，揭示软件中所有的故障需要的测试用例数目。T的均值用$\bar{T}$表示

接下来的章节展示了不同实验的各个度量标准的值。本文没有考虑执行每一个测试用例花费的代价，假设同一个实验中，每一个测试用例花费的代价是相同的。在这种情况下，F,NF,T三种度量标准可以用来比较不同测试策略检测故障的效率。

4.7 实验结果

表 4.2(a)，(b)，(c)分别展示了在F,NF,T度量标准下，RT 策略的测试结果以及 RPT、DRT、MDRT、RDRT 策略的相对于 RT 策略的提升率。

5 数据分析和讨论

5.1 数据分析

1. 表 4.1 中除了 make 实验对象，DRT、MDRT、RDRT 在其它测试对象上几乎所有的测试结果显示数目较多的分区方式比数目较少的分区方式具有更高的故障检测效率。make 实验能够检测出软件故障的测试用例集中在一个分区之中，分区数目少时有更高的概率选中该分区，因此 make 实验分区数目少时基于分区测试的测试策略具有更高的测试效率。
2. 本文的五个实验对象的故障均是相对难杀死的，grep、bash、make 实验输入域的失效率很低在 1%左右；flex 实验输入域的失效率很高达到 22.93%；gzip 实验输入域的失效率处于两者之间达到 2.34%。(1)当软件输入域的失效率高(如 flex 实验)时，即输入域中很多测试用例能够检测软件的故障，因此分区中测试用例数目多的分区，包含能够检测故障的测试用例数目可能也较多，并且该分区的失效率也可能较高。因此软件输入域失效率高时基于分区的测试策略初始剖面为不均等概率分布比均等概率分布可能具有更高的故障检测能力。(2)当软件输入域的失效率低(如 grep、bash、make 实验)时，即有很少的测试用例能够揭示软

表 4.2 实验结果(a)

实验项目	分区数目及初始概率分布		RT	RPT	DRT	MDRT	RDRT
bash	4 个分区初始概率分布	均等	36.15	-24.48%	-96.13%	-82.85%	-16.60%
		不均等	36.15	-45.37%	-20.19%	-18.12%	-32.50%
	5 个分区初始概率分布	均等	36.15	26.28%	30.43%	35.41%	36.38%
		不均等	36.15	-29.60%	7.19%	-62.10%	-56.02%
flex	3 个分区初始概率分布	均等	3.75	-110.67%	-109.33%	-104.00%	-77.33%
		不均等	3.75	-13.33%	-69.33%	-9.33%	-10.67%
	6 个分区初始概率分布	均等	3.75	-37.33%	-12.00%	-33.33%	-41.33%
		不均等	3.75	-6.67%	4.00%	4.00%	-10.67%
grep	4 个分区初始概率分布	均等	97.95	12.05%	14.50%	32.01%	36.50%
		不均等	97.95	-35.78%	22.26%	27.41%	20.62%
	5 个分区初始概率分布	均等	97.95	45.74%	34.05%	38.74%	53.34%
		不均等	97.95	48.65%	23.99%	18.12%	36.24%
gzip	3 个分区初始概率分布	均等	38.65	-8.67%	-6.47%	-4.79%	-1.42%
		不均等	38.65	-8.15%	8.54%	-3.36%	34.54%
	4 个分区初始概率分布	均等	38.65	57.31%	60.80%	65.46%	72.32%
		不均等	38.65	8.15%	41.53%	23.54%	50.84%
make	3 个分区初始概率分布	均等	90.80	89.92%	90.31%	90.86%	90.86%
		不均等	90.80	-21.04%	1.76%	36.56%	75.83%
	5 个分区初始概率分布	均等	90.80	81.50%	81.94%	85.68%	85.19%
		不均等	90.80	-22.41%	-11.56%	41.30%	65.25%

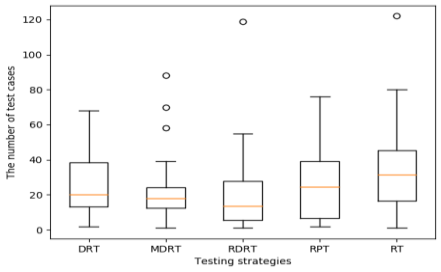
表 4.2 实验结果(b)

实验项目	分区数目及初始概率分布		RT	RPT	DRT	MDRT	RDRT
bash	4 个分区初始概率分布	均等	60.35	-93.29%	0.66%	-39.93%	-44.16%
		不均等	60.35	-33.89%	-14.17%	-45.40%	5.72%
	5 个分区初始概率分布	均等	60.35	-19.97%	-28.83%	17.56%	1.99%
		不均等	60.35	-44.74%	-37.20%	-5.39%	54.85%
flex	3 个分区初始概率分布	均等	4.00	-111.25%	-103.75%	-93.75%	-65.00%
		不均等	4.00	-16.25%	-83.75%	-2.50%	-75.00%
	6 个分区初始概率分布	均等	4.00	-120.00%	-45.00%	-43.75%	-33.75%
		不均等	4.00	-25.00%	-20.00%	-1.25%	-17.50%
grep	4 个分区初始概率分布	均等	132.50	17.47%	28.04%	32.75%	42.72%
		不均等	132.50	-10.45%	-5.92%	-47.25%	40.04%
	5 个分区初始概率分布	均等	132.50	41.21%	46.08%	47.81%	53.09%
		不均等	132.50	44.60%	-38.38%	11.02%	33.74%
gzip	3 个分区初始概率分布	均等	86.15	-73.24%	-130.59%	-108.13%	-34.30%
		不均等	86.15	24.26%	-9.00%	2.79%	5.05%
	4 个分区初始概率分布	均等	86.15	69.41%	64.94%	78.87%	73.88%
		不均等	86.15	7.72%	71.62%	69.53%	72.72%

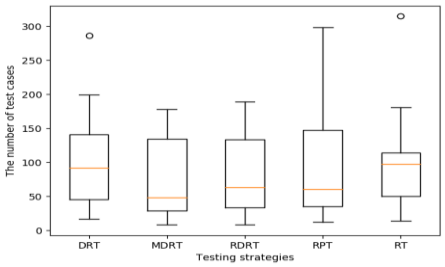
表 4.2 实验结果(c)

实验项目	分区数目及初始概率分布		RT	RPT	DRT	MDRT	RDRT
bash	4 个分区初始概率分布	均等	1582.95	-19.76%	-9.56%	2.35%	10.67%
		不均等	1582.95	0.17%	2.07%	5.04%	8.27%
	5 个分区初始概率分布	均等	1582.95	3.96%	8.39%	10.81%	12.06%
		不均等	1582.95	23.44%	21.87%	26.79%	31.64%
flex	3 个分区初始概率分布	均等	26.65	-3.38%	-18.57%	-6.00%	-16.51%
		不均等	26.65	-0.38%	-20.08%	18.20%	13.13%
	6 个分区初始概率分布	均等	26.65	12.95%	-12.76%	1.31%	24.77%
		不均等	26.65	-14.45%	-23.83%	21.76%	15.95%
grep	4 个分区初始概率分布	均等	1486.95	-67.03%	-28.20%	-12.82%	-3.28%
		不均等	1486.95	11.43%	-24.37%	-8.81%	-24.32%
	5 个分区初始概率分布	均等	1486.95	-58.00%	-57.36%	-29.11%	-52.43%
		不均等	1486.95	-48.21%	-71.04%	-35.34%	-60.66%
gzip	3 个分区初始概率分布	均等	332.15	-66.88%	-66.28%	-39.88%	-65.24%
		不均等	332.15	-0.41%	-25.55%	-20.40%	-21.80%
	4 个分区初始概率分布	均等	332.15	65.87%	65.08%	73.34%	78.19%
		不均等	332.15	8.35%	69.02%	73.16%	73.91%
make	3 个分区初始概率分布	均等	278.70	91.51%	91.84%	92.21%	93.04%
		不均等	278.70	-8.72%	49.10%	50.79%	88.12%
	5 个分区初始概率分布	均等	278.70	85.77%	86.51%	88.30%	88.57%
		不均等	278.70	5.87%	36.58%	39.68%	83.42%

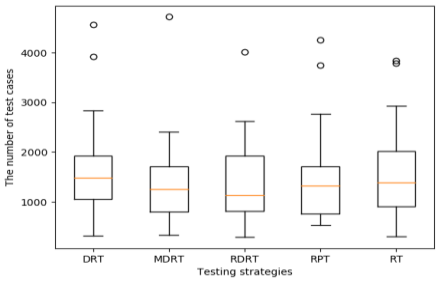
件中的故障，并且由于能够造成故障的输入趋向于聚簇在连续的区域。因此可能只有少数的分区具有故障检测能力。即便在理论上，含有较多测试用例的分区可能包含更多的能够揭示故障的测试用例，但是由于能够揭示故障的测试用例数目少，这样的分区的失效率也往往很低。在这种情况下，很难判断基于分区的测试策略的初始剖面为不均等概率分布与均等概率分布哪一个更好。(3)当软件输入域的失效率处于以上两者之间时，即软件中不存在“特别难”检测的故障(如 gzip 实验)时，基于分区的测试策略的初始剖面为不均等概率分布和均等概率分布具有相似的故障检测能力。但是由于传统的 DRT 策略以及基于 DRT 策略的改进策略具有动态调整测试剖面的能力，当测试剖面为均等概率分布时，可以根据测试的历史信息更快找到具有高故障检测能力的分区。



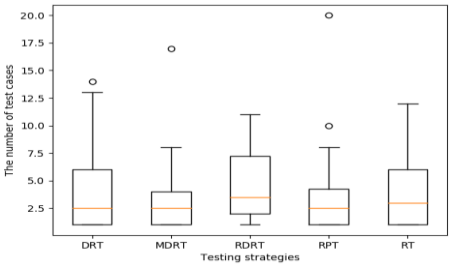
5.1.1(a) bash 实验 F 度量标准实验数据



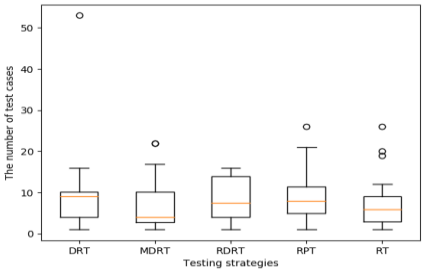
5.1.1(b) bash 实验 NF 度量标准实验数据



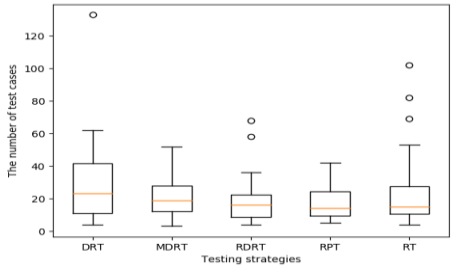
5.1.1(c) bash 实验 T 度量标准实验数据



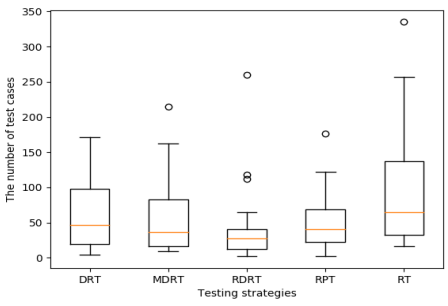
5.1.2(a) flex 实验 F 度量标准实验数据



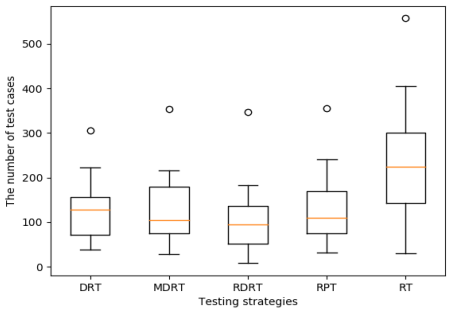
5.1.2(b) flex 实验 NF 度量标准实验数据



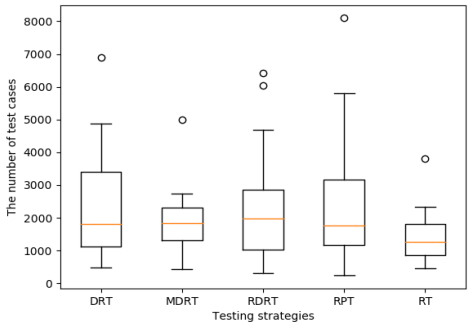
5.1.2(c) flex 实验 T 度量标准实验数据



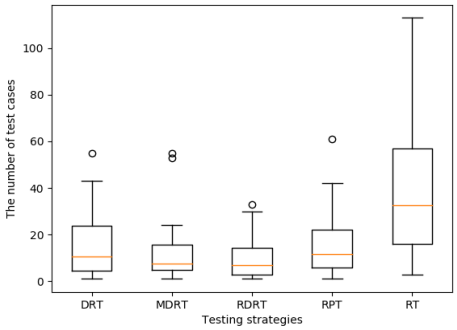
5.1.3(a) grep 实验 F 度量标准实验数据



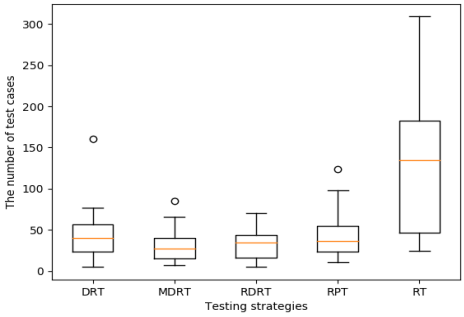
5.1.3(b) grep 实验 NF 度量标准实验数据



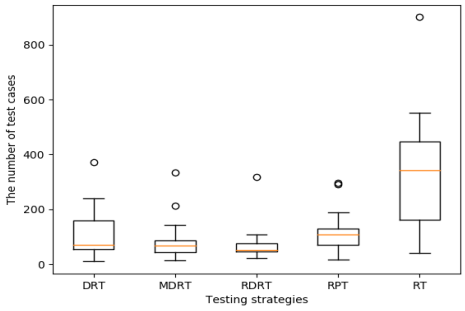
5.1.3(c) grep 实验 T 度量标准实验数据



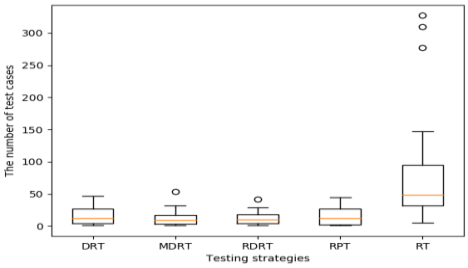
5.1.4(a) gzip 实验 F 度量标准实验数据



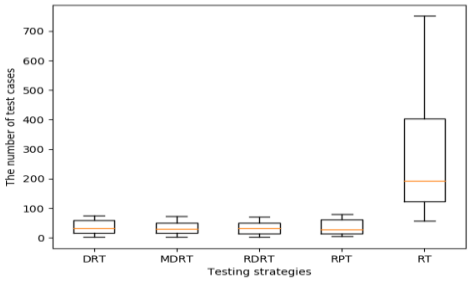
5.1.4(b) gzip 实验 NF 度量标准实验数据



5.1.4(c) gzip 实验 T 度量标准实验数据



5.1.5(a) make 实验 F 度量标准实验数据



5.1.5(b) make 实验 T 度量标准实验数据

5.1 不同实验对象在分区数目较多均等分布的初始剖面下的测试结果

3. 在表 4.2 以及图 5.1 中，除了 flex 的每一个实验对象在均等的初始概率分布以及分区数目较多的情况下，几乎所有的测试结果表明：MDRT、RDRT 策略揭示第一个和第二个故障的测试效率比 RT、RPT、DRT 策略的测试效率高。在 flex 实验中，DRT 策略的测试效率在F, NF, T三个度量指标下均不如 RT 策略，MDRT 策略以及 RDRT 策略在F, NF度量指标下不如 RT 策略，但是T度量相对 RT 策略表现更好。原因可能如下：(1) 正如第一部分提到的引起故障的输入趋向于集簇在连续的区域。DRT、MDRT、RDRT 策略都是根据这一个思想改进 RT 测试策略。然而在对 flex 对象测试时，能够揭示软件故障的测试用例数目很多，导致引起故障的输入分散到整体输入域中。这就意味着每一个分区的故障检测能力

几乎相同。此时根据历史信息更新测试剖面很有可能不能提高软件的测试效率。(2)根据表 4.2(a)，(b)中的数据显示揭示软件中的前两个故障 RT 策略平均需要 7.75 个测试用例。这表明 flex 程序中的故障“很容易”被揭示。在对这样很容易揭示软件中故障的程序 DRT、MDRT、RDRT 策略没有更多的机会调整测试剖面，测试过程就已经结束了。MDRT 策略不受其它分区测试结果的影响，并且根据所处分区当前被选择概率大小调整概率的幅度，使得该策略可以更快速的识别每一个分区的故障检测能力，因此 MDRT 策略在揭示 flex 程序所有的故障时比 DRT、RPT、RT 策略具有更高的测试效率。RDRT 策略由于具有奖惩机制，因此在对这种“很容易”被检测出故障的程序在经过短暂的测试剖面调整之后更快的辨别每一个分区的故障检测能力，从而在T度量指标下比 DRT、RPT、RT 具有更高的故障检测效率。

4. 除了 grep 的每一个实验对象在均等的初始概率分布以及分区数目较多的情况下，MDRT、RDRT 策略揭示所有故障的测试效率均比 RT、RPT、DRT 策略的测试效率高。在 grep 实验中，RPT、DRT、MDRT、RDRT 策略的测试效率不如 RT 策略。原因可能如下：根据表 4.2(a)，(b)，(c)中的数据，杀死前两个故障 RT 策略平均需要 270.45 个测试用例，杀死所有的故障平均需要 1486.95 个测试用例。表 4.1 中可以看出 grep 实验的测试用例数目为 809 个，因此 grep 实验中的故障很难被揭示，也就是说有很少的测试用例能够揭示软件中的故障，很有可能使得只有个别分区具有故障检测能力，并且这些分区的失效率很低。如果引起故障的输入分散到整个输入域中，那么能够揭示软件故障的分区的失效率更低。当每一个分区的失效率都很低时，即便选中有可能揭示故障的分区，由于失效率很低，揭示软件中故障的概率也很小。因此当存在“不容易”揭示的故障时，RPT、DRT、MDRT、RDRT 策略不如 RT 策略。但是 MDRT、RDRT 策略的测试效率仍然高于 RPT、DRT 策略。

6 结论和将来的工作

动态随机测试是一个旨在利用历史的测试信息动态改变测试剖面的测试策略。DRT 策略的主要优点是测试剖面不断变化，使得较高失效率的分区具有更高的被选择概率。但是 DRT 策略的测试效率受分区数目、初始剖面这些外部因素的影响。同时 DRT 策略的测试效率也受内部机制的影响：该策略根据某一个分区的执行结果调整所有分区被选择的概率并且所有的分区调整概率的幅度都相

同。本文结合 Markov 链的状态转移矩阵提出了 MDRT 策略解决 DRT 策略的内部机制的不恰当问题。由于传统的 DRT 策略的参数取值普遍很小，并且分区被选择的概率容易受其它分区测试结果的影响使得找出具有较高故障检测能力的分区的速度较慢。本文提出基于奖惩机制的 RDRT 策略解决这一问题。针对 DRT 策略的两个外部影响因素本文为每一个实验设置了不同数目的分区，并且为每一种分区方式设置均等的初始概率分布和不均等的初始概率分布作为初始剖面。通过对 5 个真实的程序进行测试，均等的初始概率分布作为初始剖面以及数目较多的分区方式 DRT、MDRT、RDRT 具有更高的故障检测效率。在均等的初始概率分布作为初始剖面以及分区数目较多的分区方式下各个策略的测试效率结果分为三个方面：1. 当软件中存在“很难”检测到的故障时， DRT 策略具有很高的提升空间，MDRT、RDRT 策略的测试效率相对较低，但仍比 DRT 策略的测试效率高。2. 当软件中的故障都很容易被检测出来时，DRT、MDRT、RDRT 的测试效率收到限制，但是 MDRT、RDRT 策略的测试效率仍然比 DRT、RPT 高。3. 当软件中的故障“不是很难”被检测到时，DRT、MDRT、RDRT 策略比 RT、RPT 策略具有更高的故障检测效率，并且 MDRT、RDRT 策略比 DRT、RPT 策略的测试效率更高。因此可以总结出 MDRT、RDRT 策略比 DRT 策略具有更高的故障检测能力。

但是 MDRT 策略中参数τ,γ应当满足$\gamma > \tau$，因为在实际情况中输入造成的故障要比输入没有造成故障少。RDRT 策略中惩罚上限的设置不同的实验可能取值不同，对策略的测试效率也有影响。将来的重点工作是研究 MDRT 策略中的参数以及 RDRT 策略中的惩罚上限进一步提高 MDRT、RDRT 策略的测试效率。

7 参考文献

[1] R. Hamlet, “Random Testing,” Encyclopedia of Software Engineering, J. Marciniak(ed), Wiled, pp. 970-978, 1994.

[2] W. J. Gutjahr, “Partition Testing vs Random Testing: The Influence of Uncertainty,” IEEE Transactions on Software Engineering, vol. 25, issue 5 pp: 661-674, Setember/October 1999.

[3] R. Hamlet and R. Taylor, “Partition Testing Does Not Inspire Confidence,” IEEE Transactions on Software Engineering, vol. 16, no. 12, pp. 1, 402-1, 411, December 1990.

[4] T. Y. Chen and Y. T. Yu, “A more general sufficient condition for partition testing to be better than testing,” Information Processing Letters, vol.57, issue 3, pp. 145-149,

February 1996.

[5] K. Y. Cai, B. Gu, H. Hu, and Y. C. Li, “Adaptive Software Testing with Fixed-Memory Feedback,” Jounal of Systems and Software, vol. 80, issue 8, pp. 1328-1348, August 2007.

[6] K. Y. Cai, T. Jing, and C. G. Bai, “Partition Testing with Dynamic Partitionint,” in Proceedings of the 29th COMPSAC, Edinburgh, Scotland, July 2005, pp. 113-116.

[7] A. G. koru, K. E. Emam, D. S. Zhang, H. Liu, D. Mathew, “Theory of relative defect proneness,” Encyclopedia of Software Engineering, vol. 13, no. 5, pp. 473-498, 2008.

[8] P. E. Ammannn and J. C. Knight, “data diversity: an approach to software fault tolerance,” IEEE Transactions on Computers, vol. 37, no. 4, pp. 418-425, April 1988.

[9] G. B. Finelli, “NASA Software Failure Characterization Experiments,” Reliability Engineering and System Safety, vol. 32, pp. 155-169, 1991.

[10] K. Y. Cai, “Optimal software testing and adaptive software testing in the context of software cybernetics,” Information and Software Technology, vol. 44, pp. 841-855, November 2002.

[11] K. Y. Cai, H. Hu, C. H. Jiang, and F. Ye, “Random testing with dynamically updated test profile,” in Proceedings of the 20th ISSRE 2009, Fast Abstract 198, Mysuru, Karnataka, India, November 2009.

[12] G. J. Myers, The Art of Software Testing, Wiley, New York, 1979.

[13] J. W. Duran and S. C. Ntafos, “An evaluation of random testing,” IEEE Transactions on Software Engineering, 10(4): 438-444, July 1984.

[14] E. J. Weyuker and B. Jeng, “Analyzing partition testing strategies,” IEEE Transactions on Software Engineering, 17(7): 703-711, July 1991.

[15] Junpeng Lv, Hai Hu, and Kai Yuan Cai, “A Sufficient Condition for Parameters Estimation in Dynamic Random Testing,” International Computer Software and Application Conference, 2011 IEEE 35th Annual, Munich, 2011, pp. 19-24.

[16] Z. J. Yang, B. B. Yin, J. P. Lv, K. Y. Cai, S. S. Yau, and J. Yu, “Dynamic Random Testing with Parameter Adjustment,” COMPSACW, 2014 IEEE 38th Annual, 2014, pp. 37-42.

[17] T. J. Ostrand, M. J. Balcer, “The Category-Partition Method for Specifying and Generating Functional Tests,” Communications of the ACM, vol. 331, pp. 676-686, June 1988.