一种可变测试集的协议一致性测试方法

吕欣岩, 赵保华, 屈玉贵
(中国科学技术大学计算机系, 安徽合肥 230027)

摘 要: 目前常用的协议一致性测试的测试方法是首先对协议规范建模, 然后通过模型生成测试集, 最后执行测试集. 这种方法存在执行效率不高和实际测试范围可能被缩小的问题, 为此本文提出一种可变测试集的方法, 通过动态执行测试集提高其执行效率, 同时从与协议实现无关的角度扩大协议的实际测试范围.

关键词: 一致性测试; 固定测试集; 可变测试集; 测试树
中图分类号: TP311 文献标识码: A 文章编号: 0372-2112 (2003) 12A-2179-04

A Protocol Conformance Testing Method about the Alterable Test Suite

LU Xin-yan, ZHAO Bao-hua, QU Yu-gui
(*Department of Computer Science, University of Science and Technology of China, Hefei, Anhui 230027, China*)

Abstract: The common methods of the protocol conformance testing include three steps: create model by the protocol specification; generate test suite from the model; execute this test suite. There are two problems in these methods: the first is low efficiency; the second is the actual range of testing being reduced. In this paper, a method about alterable test suite will be introduced to improve the efficiency by dynamically executing the test suite, and expand the actual range of the protocol conformance testing without depending on the protocol implementation.

Key words: the conformance testing; fixed test suite; alterable test suite; testing tree

1 引言

协议测试的目的是保证协议实现按照协议规范稳定可靠的运行, 协议测试的种类和方法有很多, 一致性测试是其它测试的基础. 协议实现是一个复杂的实体, 完全测试协议在时间和开销上几乎是不可实现的[1], 所以如何在测试的完全性、彻底性和测试的时间、开销上做到平衡是一个需要研究的问题, 事实上实际测试的本质就是在有限的预算范围内要求测试达到给定的覆盖度[2].

按照测试集的执行和生成情况可以将协议的一致性测试分为两类: 固定测试集测试和可变测试集测试. 所谓固定测试集测试就是测试集在整个测试期间不会被改变, 测试集中的每一条测试用例在执行过程中按照一定的顺序相互独立的逐条执行. 而所谓可变测试集测试是说测试集的执行和生成都是随实际测试情况而动态改变的. 当前使用的大部分协议一致性测试方法都是固定测试集的, 在这种测试中人们追求的目标是使用尽可能短的测试用例检查尽可能多的错误, 但由于相关性对测试的影响, 固定测试集的协议一致性测试会存在两个问题: (1) 执行效率低; (2) 可能无法达到理想的实际测试覆盖度. 为解决这两个问题, 本文提出一种可变测试集的测试方法, 它在兼顾了一致性测试时间和开销的前提下, 提高了测试集的实际执行效率, 同时尽可能从与协议实现无关的角度扩大实际测试的覆盖度.

2 相关定义

为讨论方便先给出本文使用的几个定义:

定义 **1** 令 P 为协议规范集合, 则存在一个函数 $f(p)=Ti$, 其中 $p\in P$, 表示集合 P 的任何一部分 p 都可以通过 f 生成测试用例 Ti.

定义 **2** 根据测试的目的, 通常一条测试用例 T_i 可以被划分成三个部分: 引导序列 Lead(T_i)、执行序列 Exe(T_i)、验证序列 Prove(T_i). Lead(T_i) 是将 IUT 从初始状态引导到待测状态的输入/输出序列; Exe(T_i) 是测试目的规定的输入/输出序列; 验证序列 Prove(T_i) 是用于检验 Exe(T_i) 的输入/输出序列.

定义 **3** 设 T 为测试集, 则测试用例 T_i 中事件 E_{ij} 的前缀序列定义如下:

$$\text{prefix}(E_{ij})=\begin{cases}(E_{i1},\cdots,E_{ij,1}), & 1\le i\le m,\ 1<j\le k\\ \text{Empty}, & j=1\end{cases}$$

其中 $E_{ij}\in T_i$, $T_i\in T$. 表示 T_i 在执行 E_{ij} 前需要执行的事件.

收稿日期: 2003-10-24; 修回日期: 2003-12-16
项目基金: 国家自然科学基金(No. 90104010, No. 60241004); 国家 863 项目(No. 2001AA121016); 国家 973 计划项目(No. 2003CB314800); 中国科学院计算机科学重点实验室基础研究课题基金

Empty 表示无需执行任何事件.

定义 **4** 设 $T_i=\{E_{i1}, \cdots E_{ik}\}$ 其中 $k>0$, 则测试序列 $ti=\{E_{i1}, \cdots, E_{ij}\}$ 其中 $j\in[1, k]$ 的执行函数 Run(ti) 表示将 ti 应用到待测实体 IUT, 这个函数有两个返回值: *Pass*、*Fail*, 分别表示 IUT 通过 ti 的测试; IUT 与 ti 的测试不一致.

定义 **5** 设 T 是测试集, 则测试用例 T_i, T_j (其中 $i>j$) 的最大公共事件前缀定义如下

$$\text{MaxPrefix}(Ti, Tj)=\begin{cases}\text{Empty}, & E_{i1}\neq E_{j1}\\ \text{Prefix}(E_{ik})+E_{ik}, & k=\text{Max}(\{k\mid \text{Prefix}(E_{ik})=\text{Prefix}(E_{jk})\wedge E_{ik}=E_{jk}\})\end{cases}$$

其中 $T_i, T_j\in T$, $E_{iu}\in T_i$, $E_{ju}\in T_j$, $u\in[0, k]$. 表示 T_i、T_i 的最大公共事件序列, Empty 表示 T_i、T_i 没有公共事件.

3 采用可变测试集测试的原因

采用可变测试集测试有两方面原因.

首先, 固定测试集存在执行效率低和实测范围小的缺点. 在传统的协议一致性测试中测试集不会随实际测试情况而动态调整, 但在实际的测试过程中经常遇到: 协议的多个测试不同部分的测试用例因为同一个原因而执行失败, 这导致对协议产品的测试不够全面. 例如设 T 是一个固定测试集, 令 $T_i\in T$(其中 $i>0$)是 T 中的一条测试用例, 通常 T_i 对应于一个事件(通常是输入/输出对)序列($E_{i1}..E_{ik}$), 由此可以将 T_i 按照E_{ij}进行重新组合构造成一颗测试树, 如图 1 所示, 则测试用例的执行过程就是从树根 E_0 出发到达每一个叶子节点的遍历过程. 在测试中如果 Run($E_0, \cdots, E_{ij-1}, E_{ij}$) = *Fail*, 则对于$\forall T_u\in T$, 如果MaxPrefix($T_u$, ($E_0, \cdots, E_{ij-1}, E_{ij}$)) = ($E_0, \cdots, E_{ij-1}, E_{ij}$)则必有 Run($T_u$) = *Fail*, 也就是说图1 中所有处于虚线圈中的测试用例都将执行不通过, 固定测试集的测试不处理这种情况, 所以会造成重复执行序列($E_0, \cdots, E_{ij-1}, E_{ij}$), 效率低下. 同时, 如果$\exists$ Exe(T_u) $\notin$ ($E_0, \cdots, E_{ij-1}, E_{ij}$)则 T_u 的实际测试目的也没有达到, 因此实测范围因 Run($E_0, \cdots, E_{ij-1}, E_{ij}$) = *Fail* 而缩小了. 可变测试集的测试可以有条件的解决这两个问题.

其次, 由于协议本身的模块性、层次性、依赖性, 通常上层协议模块的测试依赖于下层协议模块的正确[3], 例如OSPF 协议, 当下层协议模块有不通过测试的部分出现时会对上层协议的测试产生影响, 文[3]提出一种基于 IUT 实现的方法解决这个问题, 这种方法的缺点是测试与实现相关, 所以测试集的通用性差而且测试的自动化程度被限制, 而可变测试集测试可以在测试范围不变的条件下尽可能的避免这两个缺点.

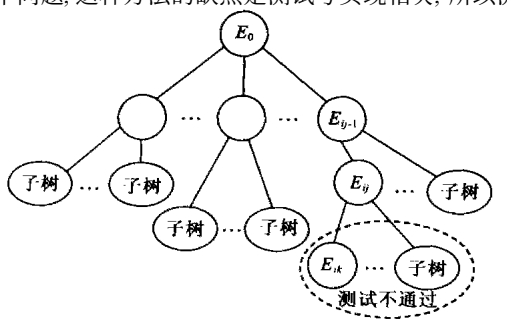

图 1 测试树

4 可变测试集测试的原理、方法及其分析

本节将详细论述可变测试集测试. 首先给出测试假设: 待测实体 IUT 的行为是确定的, IUT 对应的协议规范可以由确定的有限状态机 DFSM 描述.

4.1 测试原理

概括的说可变测试集测试就是测试期间测试集的执行和生成过程可以被动态调整以适应实际情况的一种测试. 从这个表述可以知道"可变"有两层含义: (1) 执行意义上的可变, 称为测试集的动态执行; (2) 测试用例生成意义上的可变, 称为测试用例的动态生成.

执行意义上的可变用于解决固定测试集执行效率低的问题. 由第 3 节的论述知, 设 T 为一个测试集, 令 $T_i, T_j\in T$ (其中 $i>j$), 如果 Run(MaxPrefix(T_i, T_j)) = *Fail*, 则 T_j 已经没有实际执行的必要了. 为此在测试集执行期间, 将每一条执行失败的序列提取出来构成一个集合 *ET*, 设后续将要执行的序列是 T_i, 则可以得到这个结论:

定理 **1** 如果$\exists$ $ETi\in ET$ 使得 MaxPrefix(T_i, ETi) = ETi, 则可以跳过 T_i 的执行并且判定 Run(T_i) = *Fail*. 证明略.

动态执行测试集可以提高执行效率, 但并不能扩大实际的测试范围; 动态生成测试用例可以扩大实际测试范围, 但由于无法消除测试用例间的重复, 所以不能保证测试集的执行效率. 因此在实际测试时要将两者结合起来使用.

4.2 测试方法

由上面的论述所以可变测试集测试的原理可以描述如下: $AT=ST\cup DT$, 其中 *AT* 为可变测试集, *ST* 为*AT* 的静态部分, *DT* 为*AT* 的动态部分. 测试前用常规测试用例生成方法产生 *ST*, 而 $DT=\Phi$. 测试时, 首先调用算法1 动态执行 *ST*, 并收集 *ST* 执行的信息. 在执行算法 1 的过程中调用算法 2 标记已经被测试的形式化模型部分. 算法 1 结束后, 对于未被测试的部分, 在保证 Run(Lead(DT_i))和 Prove(DT_i) 正确的条件下, 调用算法 3 动态生成新的测试用例 DT_i. 新生成测试用例的全体构成 *DT*. 算法 3 结束后, 如果仍然存在未被测试的部分, 则这一部分无法在实现无关的条件下进行测试, 此时只能使用文[3]的方法进行测试.

算法 **1** **ST** 的动态执行

输入: *ST*, *G*

输出: *ET*, *ST* 的执行结果, 经过标记的 *G*

初始化 *ET* = *NULL*;

for($\forall$ $STi\in ST$)

sequence = *NULL*;//存储序列

if($\neg\exists$ $ETi\in ET$ 使得MaxPrefix(STi, ETi) = = ETi)

if (Run(STi) = = *Fail*)

sequence = Max(($E_0, \cdots, E_k$) | ($E_0, \cdots, E_k$) $\subseteq$ STi 且 Run($E_0, \cdots, E_k$) = = *Fail*));

endif

else

```
        sequence = ETi;
      endif
      MarkG(G, STi, sequence);        //调用算法 2
      ET = ET ∪ sequence;
      ST = ST - STi;
    endfor
End.
```

算法 **2**　标记 **G**

输入: *G*, *STi*, *sequence*

输出: 经过标记的 *G*

```
if (sequence) = = NULL
  // Trans(STi) 函数返回 STi 能测试的协议规范集.
  对∀p ∈ Trans(STi), 标记 p 为已测试(Pass);
else
  if (∃p ∈ Trans(STi)) ∧ IsFront(p, sequence)
  // IsFront: 判断 p 是否在 sequence 前.
    标记 p 为已测试(Fail);
  endif
endif
End.
```

算法 **3**　测试用例的动态生成

输入: 经过标记的 *G*, *ET*

输出: *DT*, 经过标记的 *G*

```
if(!(∃p ∈ G 且 p 未被标记)) exit;
for(∀p ∈ G 且 p 未被标记)
  生成 f(p), 使得∀ETi ∈ ET 都有 MaxPrefix(f(p), ETi) ≠ETi.
  if (f(p) = = NULL) countinue;
  if (Run(f(p)) = = Fail)
    sequence = Max((E0, …, Ek) | (E0, …, Ek) ⊆ Ti 且 Run(E0, …, Ek) = = Fail)
    if (IsFront(p, sequence))
      MarkG(G, f(p), sequence);
    endif
    ET = ET ∪ sequence;
  else
    MarkG(G, f(p), NULL);
  endif
endfor
End.
```

4.3　应用分析

可变测试集测试的过程由三个算法组成, 下面分别对每个算法进行应用分析. 算法 1 可以单独使用, 它是对测试集执行过程的一个改进. 算法中如果对 *ST* 进行分组并且 *STi* 与 *ET* 的比较通过错误用例树完成, 则由于协议测试的行为主要是物理链路上的数据传输, 而这个时间开销远大于 CPU 的处理开销, 因此虽然在最坏情况下算法 1 的时间复杂度与现有的测试集执行过程相同(不考虑测试用例的执行和其他操作的时间差异), 但如果测试树的主干上存在错误, 则算法 1 可以极大的提高效率.

算法 2 由算法 1 调用, 它可以与 *ST* 执行同步, 所以算法 2 对算法 1 的时间没有数量级上的影响.

算法 3 也可以单独使用, 以获得最大的测试范围, 但时间开销太大, 所以通常是与算法 1 结合使用的. 它的时间主要消耗在生成 $f(p)$ 上, 这个过程与模型相关. 如果协议规范由状态机描述, 并且每个状态都有 UIO 序列, 则动态生成一条 DTi ($DTi \in DT$) 的过程就简化为寻找一条满足 $\neg \exists ETi \in ET$ 且 MaxPrefix(DTi, ETi) $\neq ETi$ 的 Lead(DTi). 在最坏情况下, 这个过程的时间复杂度与在一个图中寻找两点间所有无环路径的时间复杂度相同. 如果采用深度优先搜索(DFS)算法在完全图上的复杂度是 $O(n^n)$, 其中 n 为图的节点数. 但在实际测试中基于以下三点考虑: (1) 由于 *ET* 的影响, 可以对搜索进行剪枝; (2) 实际协议的状态机并非一个完全图; (3) *DT* 是对 *ST* 的测试范围的补充. 所以仍然使用 DFS 算法获取新的 $f(p)$, 鉴于篇幅搜索算法略.

对算法 2、3 的一种改进是, 算法 2 采用文[4]中给出的错误定位方法, 标记模型, 对于可能的错误, 删除模型中相应的边, 算法 3 使用经过标记删除的模型生成测试用例, 此时算法 3 的时间复杂度可以下降到 $O(n^3)$, 但文[4]的方法定位范围比较宽泛, 有可能导致最终 *DT* 为空. 相应的两个算法的实现并不困难, 限于篇幅本文不再给出.

5　实例

如图 2 所示, 左边为一个状态机, 右边是每个状态的 UIO 序列, 经过优化的 UIO 测试集如表 1 所示, 这个测试集构成 *ST*. *ST* 中标注了每条测试用例要测试的转移. 图 3 是 *ST* 构成的测试树. 现在假设转移 $s_4 \xrightarrow{} e_2 \xrightarrow{} s_5$ 发生错误, 可以分两种情况. 如果这个错误是输入/输出错, 则动态执行 *ST* 时序列(e_0, e_2, e_4, e_2, e_4, e_1) 执行失败, 导致失败的序列是(e_0, e_2, e_4, e_2), 转移($s_4 \xrightarrow{} e_2 \xrightarrow{} s_5$) 被标记为已测试, 序列($e_0$, e_2, e_4, e_2, e_4, e_2, e_1) 和序列(e_0, e_2, e_4, e_2, e_5, e_4, e_5) 没有被实际执行, 而是通过算法 1 的第 2 步直接判定 $Run(Ti) = = Fail$ 转入算法 2 的调

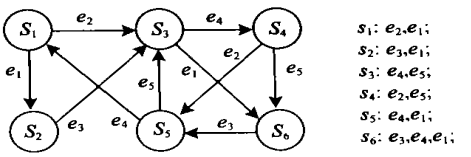

图 2　状态机及其状态的 UIO 序列

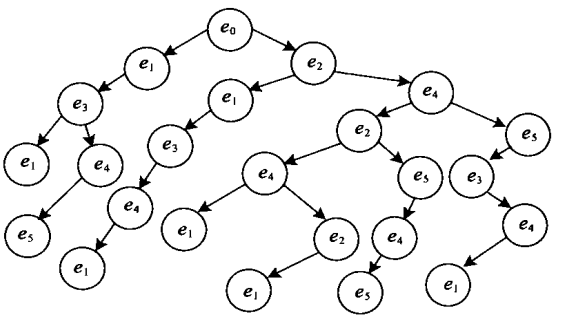

图 3　测试树

用, 此时转移 $s_3 \rightarrow e_4 \rightarrow s_4$ 被标记为已测试, $s_5 \rightarrow e_4 \rightarrow s_1$ 和 $s_5 \rightarrow e_5 \rightarrow s_3$ 标记为未测试到, 所以 *ST* 的动态执行完毕后需要对它们生成新的测试用例, 由算法 3 所以到达 s_5 的一条新的引导序列变成(e_2, e_1, e_3), 所以最终生成的两条测试用例为$\{s_5 \rightarrow s_1: e_2, e_1, e_3, e_4, e_1; s_5 \rightarrow s_3: e_2, e_1, e_3, e_5, e_4, e_5\}$. 而固定测试集需要实际执行全部 7 条测试用例, 并且无法测试转移 $s_5 \rightarrow e_4 \rightarrow s_1$ 和 $s_5 \rightarrow e_5 \rightarrow s_3$.

表 **1** 测试集

测试目的	测试用例	测试目的	测试用例
$s_1 \rightarrow s_2$:	e_1, e_3, e_1	$S_2 \rightarrow s_3$:	e_1, e_3, e_4, e_5
$s_1 \rightarrow s_3, s_4 \rightarrow s_6$:	$e_2, e_4, e_5, e_3, e_4, e_1$	$s_3 \rightarrow s_4, s_5 \rightarrow s_3$:	$e_2, e_4, e_2, e_5, e_4, e_5$
$s_4 \rightarrow s_5$:	e_2, e_4, e_2, e_4, e_1	$s_5 \rightarrow s_1$:	$e_2, e_4, e_2, e_4, e_2, e_1$
$s_3 \rightarrow s_6, s_6 \rightarrow s_5$:	e_2, e_1, e_3, e_4, e_1		

如果这个错误是转移错, 比如转移变成 $s_4 \rightarrow e_2 \rightarrow s_2$, 则分析过程同上一段的论述.

6 总结

目前常用的协议一致性测试的测试方法存在两方面不足: 重复执行导致的执行效率不高; 实际测试范围可能被缩小. 本文给出了一种可变测试集的协议一致性测试方法, 它通过对固定测试集的动态执行, 有效的改善了测试集的实际执行效率, 同时对于协议中未被测试的部分利用 *ET* 或其它错误定位算法生成新的测试用例, 以扩大实际测试范围.

参考文献:

[1] Petrenko A, Bochmann G v, Yao M. On fault coverage of tests for finite state specifications[J]. Computer Networks and ISDN Systems special issue on Protocol Testing, 1996, 29(1): 81- 106.

[2] Jonathan P Bowen, Kirill Bogdanov. FORTEST: Formal methods and testing (2002)[A]. Proc COMPSAC 02: 26th IEEE Annual International Computer Software and Applications Conference[C]. Oxford, UK, 2002. 91- 101.

[3] 杨晶, 赵保华, 屈玉贵. 基于层次结构的 OSPF 一致性测试[J]. 通信学报, 2002, 23(8): 87- 92.

[4] Miller R E, Arisha K A. Fault identification in networks by passive testing[A]. Proceedings of the 34th Annual Simulation Symposium [C]. Seatte, WA USA, 2001. 277- 284.

作者简介:

吕欣岩 男, 1976 年 5 月生于新疆, 博士研究生, 研究方向: 通信软件测试理论与方法.

赵保华 男, 1947 年 8 月生于安徽, 教授, 博士生导师, 研究方向: 通信软件测试理论与方法, 软件工程.

屈玉贵 女, 1945 年 5 月生于安徽, 教授, 博士生导师, 研究方向: 计算机体系结构, 通信软件和协议工程.