基于扩展有限状态机的协议测试集生成研究*

王建国，吴建平

（清华大学 计算机科学与技术系，北京 100084）

E-mail: wjg@csnet1.cs.tsinghua.edu.cn

http://www.tsinghua.edu.cn

摘要：与其他测试方法相比，主要解决了自动生成可执行测试序列的问题.首先介绍现有的基于扩展有限状态机的测试生成算法，这些算法虽然结合了控制流和数据流的测试，但是并没有解决测试序列的可执行问题.重点解决了包含有影响循环测试序列的可执行性问题，并通过预先发现循环的中断条件而减少不可用的测试路径的产生.另外，给出了算法的详细说明.

关键词：一致性测试；测试例生成；树表结合表示法

中图法分类号：TP393 **文献标识码**：A

随着计算机网络和通信技术的不断进步，协议的设计和实现变得越来越复杂，而且对于同一协议标准有时会存在多个不同的实现版本.为了保证协议的各种实现版本之间能够完全地相互访问并进行可靠的通信，最有效的手段就是对这些协议实现进行一致性测试.

测试集生成是在协议一致性测试中，专家、学者研究的重点.目前提出的测试例生成方法大都基于有限状态机FSM(finite state machine)、扩展有限状态机EFSM(extended-FSM)等模型.对于EFSM，传统的FSM测试方法不再适合.由于EFSM中扩展的数据流描述部分在更大程度上决定着协议实现的行为，必须重点测试.因为不可满足的谓词条件的存在，将基于FSM的测试生成方法直接应用到基于EFSM的协议会产生不可执行的测试序列.一些学者[2]试图采用数据流测试技术进行测试例的自动生成，但未考虑变迁的谓词条件，也未能有效地解决状态空间爆炸的问题.

本文提出的方法在很大程度上减缓或解决了现有问题.它综合了控制流和数据流测试方法，在测试路径生成的同时，采用分析循环的启发算法进行可执行性的判定，这样就可以避免生成那些最终被抛弃的路径.

本文第1节首先介绍FSM和EFSM数学的模型、数据流和控制流测试的概念.第2节介绍了用于生成可执行测试例和测试序列的主要算法.关于可执行的DU(define-use)路径生成算法也将在第2节中加以介绍.第3节给出了具体协议的生成过程.最后，我们将比较本文提出的方法和其他方法的结果，并总结全文.

1 基本模型和概念

1.1 EFSM模型

扩展有限状态机(EFSM)形式化地表示为一个六元组$\langle S, s_0, I, O, T, V\rangle$.其中：

* 收稿日期：1999-05-20；修改日期：2000-03-24

基金项目：国家自然科学基金资助项目(69682002,69725003)

作者简介：王建国(1972－)，男，山东梁山人，博士，主要研究领域为计算机网络及其协议测试；吴建平(1954－)，男，山西太原人，教授，博士生导师，主要研究领域为计算机网络体系结构与协议测试.

© 中国科学院软件研究所 http://www.jos.org.cn

S 是一个非空的状态集合;

s_0 是初始状态;

I 是非空的输入交互原语集合;

O 是非空的输出交互原语集合;

T 是非空的变迁集合;

V 是变量集合,

T 的每个元素又是一个五元组,$T=\langle Source\_State, Dest\_State, Input, Predicate, Compute\_Block\rangle$. 其中"*Source_State*"和"*Dest_State*"表示的是 T 的首状态和末状态;"*Input*"表示来自于 I 的输入原语或空;"*Predicate*"是关于 V 中变量、输入原语输入的参数和某些约束的类 Pascal 的谓词表达式;"*Compute-Block*"指的是包括类 Pascal 的赋值语句和输出语句的计算模块.

我们假设协议描述的 EFSM 表示是确定性的,而且是完全描述的;初始状态在给定的有效的上下文环境中,EFSM 模型是强连通图.

1.2 控制流测试和数据流分析

控制流测试的目的是为了保证 IUT 的行为与协议 FSM 所描述的相一致. 目前,用于控制流测试的方法通常假设被测系统可被描述为一个 FSM. 我们选择 UIO 序列[4,5]用于控制流测试中的状态判定,这主要是由于每个状态的输入部分一般是不同的,而某一状态的 UIO 序列可以方便地将它与其他状态区别开来.

数据流测试通常基于有向数据流图. 在理想情况下,测试所有可能的输入数据将提供程序行为的最完全信息,而在实际测试中,通常选择一个可以代表整个输入域子集. 下面,我们将给出在后续章节中需要用到的一些定义.

定义 1(变量的使用集合).

A-USE:当变量 x 出现在变迁 t 中赋值语句的左首时,则有 $x\in A\text{-}USE(t)$.

I-USE:当变量 x 出现在变迁的输入列中时,则有 $x\in I\text{-}USE(t)$.

P USE:当变量 x 出现在变迁的谓词表达式中时,则有 $x\in P\text{-}USE(t)$.

C-USE:如果变量 x 出现在赋值语句的右首或输出原语中,则有 $x\in C\text{-}USE(t)$.

如果 $x\in A\text{-}USE(t)\cup I\text{-}USE(t)$,则称变量 $x\in DEF(t)$.

例:如图 1 所示,对于变迁 t_3,$I\ USE(t_3)=\{sdu,n,h\}$,$C\_USE(t_3)=\{n,h\}$,$P\ USE(t_3)=\varnothing$,$A\_USE(t_3)=\{number, no\_of\_segement, blockbound, counter\}$.

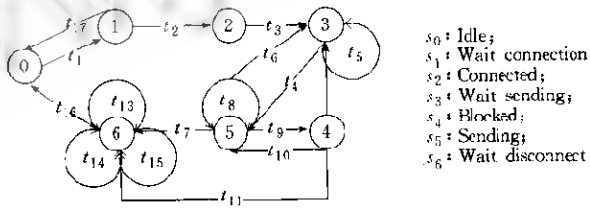

Fig. 1 EFSM based protocol specification
图1 基于EFSM的协议描述示例

定义 2(Def-Clear-Path 和 Du-Path). *Def-Clear-Path*$(t_1,t_2,\ldots,t_{k-1},t_k)$:对于变量 x,$x\in DEF(t_1)$ 且 $x\notin DEF(t_i)$,$i=2,3,\ldots,k$.

Du-Path$(t_1,t_2,\ldots,t_{k-1},t_k)$:对于变量 x,如果 $x\in DEF(t_1)\cup C\text{-}USE(t_k)\cup P\text{-}USE(t_k)$,且

© 中国科学院软件研究所 http://www.jos.org.cn

$(t_2,\ldots,t_{k-1},t_k)$是 Def-Clear-Path. *Du-Path*(x)称为对于变量 x 的 Du-Path.

定义 3(数据分析标准). 设 G 是一张 Def-Use 图,P 是 G 的全路径集合,I 是 G 中的节点集合.那么:

如果$\forall\, i\in I,\forall\, x\in DEF(i)$,$P$ 包含全部的 *Du-Path*(x),则说 P 满足 All-Du-Path 标准.

如果 P 包含 G 的所有路径,则称 P 满足 All-Path 标准.

在选择某种标准时,必然存在妥协.选择标准越强,就越能定位实现中的错误.另一方面,如果采用较弱的标准,则用到的测试例较少,生成代价较低.我们将采用次强标准 All-Du-Path 作为我们的生成标准.All-Du-Path 生成满足这一标准的所有路径.由于变量值的可选择性,这些路径并不保证检测到所有错误的存在.

2 可执行测试序列生成算法

我们的方法结合了控制流和数据流测试技术,并可生成出标准的树表结合表示法 TTCN[3]测试例.

2.1 可执行测试例生成过程

在 Du-Path 的生成过程中,进行测试序列的可执行性的判定.下列算法说明了如何自动生成可执行测试例的过程.

算法 1. 自动生成可执行测试例主算法.

```
Algorithm EFSM-TestGen
Begin
  读取协议 EFSM 的 EBE 描述;
  从 EBE 描述生成 EFSM 状态图 G;
  为可能影响控制流的变量和定时器选定值;
  Generate-Executable-Du-Path(G);
  删除被包含在其他路径中的较短路径;
  为可执行的 Du-Path 增加状态判定序列;
  再续加后缀序列,形成一条完整的测试路径;
  For 每条完整的测试路径 P
    If P 不可执行
      Executablization(P)
    Endif
    Else 丢弃 P;
    Endif
  Endfor
  For 每条不被包含在所有路径中的变迁 t
      增加前缀序列;
      后续状态判定序列;
      再续加后缀序列,形成一条完整的测试路径;
  EndFor
(接左列)
  For 每条可执行的测试路径
      利用代数求值技术,生成序列中用到的 PDU 的约束;
      将输入/输出序列转变成 TTCN 格式,形成完整的测试例;
  EndFor
  End;
  Procedure Generate Executable-Du-Path(G)
  Begin
    为图 G 中的每条变迁 t,
    计算出 A-Uses,I-Uses,C-Uses 和 P-Uses 集合;
    为图 G 中的每条变迁 t,生成出最短的可执行前缀(Preamble);
    For 图 G 中的每条变迁 t
      For 变迁 t 中每个变量 v∈A-use(t)
        For 每条变迁 u,有变量 v∈P-Use(u)∪C-Use(u)
            Find-All-SubPaths(t,u,v)
        EndFor
          EndFor
    EndFor
  End.
```

当找到各个状态的前缀(preamble)和后缀(postamble)序列时,我们首先去选择那些不包含任何谓词的最短路径(计算代价最小).如果无法得到这样的路径,那么我们就选择其中次优的最短路径,并逐步使其可执行.

© 中国科学院软件研究所 http://www.jos.org.cn

2.2 Du-Path 的生成过程

为了克服由于某些变迁上的谓词条件无法得到满足,许多已生成的路径因无法执行而终被丢弃的问题,我们采用在生成的同时进行可执行判定.下面的算法主要用于发现两条变迁之间的全部路径.

算法2. 发现两条变迁之间的全部路径的过程.

```
Procedure Find-All-SubPaths(Transition t1, Transition t2, Variable var)
 Begin
 If 生成前缀,后缀或循环
   Preamble:=t1;
 Else Preamble:=从 t0到变迁 t1的最短可执行前缀;
 EndIf
 Generate-All-Du-Paths (t1,t2,var,Preamble)
 End;
```

下面的算法用于根据 t_1上定义的变量 var,得到变迁 t_1,t_2之间全部的可执行的前缀序列和 Du-Path.

算法3. 生成所有的可执行的前缀序列和 Du-Path 的算法.

```
Procedure Generate-All-Du-Paths(t1,t2,var,
          Preamble)
Begin
If (t 是 t1的直接后继)
 If (t=t2) or (t 是 t1的后继 and t2是 t 的后继)
  If 正在生成新的路径 Path
    Proceed:=最后一条生成的 du-path(不包含前缀)
    If (t1∈Proceed)
      Common:=Proceed 中 t1之前的变迁序列;
    EndIf
    If 正在生成新的路径 Path
      加入路径的前缀,
      将 var 加入到该路径的测试目的列表中;
    EndIf
    If Common 不空
      将 Common 加入到 Path;
    EndIf
    If (t=t2)
      将 t 加入到 Path,Make-Executable(Path);
    Else
      If t∉Path and var∉A-Use(t)
        将 t 加入到 Path;
(转右列)
```

```
        Generate-All-Du-Paths(t,t2,t0,var,Preamble)
      EndIf
    EndIf
  EndIf
  t1:=图中下一条变迁;
  If (t 非空)
    Generate-All-Du-Paths(t1,t2,t,var,Preamble)
  Else
  If (Path 非空)
    If (Path 中的最后一条变迁不是 t2的直接前驱)
      将 Path 中的最后一条变迁删除;
    Else
      If (添加上 t2之后,Path 将和其他生成的路径相同)
        丢弃 Path;
      EndIf
    EndIf
  Endif
Endif
End.
```

生成后缀和循环的算法与之相似,只是不再调用过程 *Make-Executable*(*Path*).

设 $P_1=(t_1,t_2,\ldots,t_{k-1},t_k)$. 如果 *Make-Executable*($P_1$)检测到不可执行变迁 t_k,而且,若存在另一条可执行路径 $P_2=(t_1,t_2,\ldots,t_{k-1},\ldots,t_k)$的话,则丢弃 P_1. 否则,就调用 *Executablization*(P_1)进一步解决 P_1的可执行化问题. 这种判别省去了多次生成相同路径或同等路径(即具有不同的循环的相同的 Du-Path)的时间和空间浪费. *Executablization*(*Path*)对 Path 中的变迁的可执行性逐一判定. 对于每条变迁,必须对相应的谓词条件进行代数求解,直到只包含常量项和输入参数项为

© 中国科学院软件研究所 http://www.jos.org.cn

止.通过该过程可以确定指定变迁的可执行性(特别是对于简单谓词条件).

2.3 对测试例中可执行化处理

可执行问题属于不确定性问题,但是,在大多数情况下都可以得到解决.文献[1]通过采用静态循环分析自环应该重复的次数.但若自环中的有影响变量并未得到修改,该方法就不再合适.另外,如果自环的循环次数未知,它也无能为力.故此,我们采用了下面的启发式算法来生成适当的循环,插入到不可执行的路径中,使之可执行.

算法4. 可执行化处理算法.

```
Procedure Executablization(Path P)
Begin
令 Cycle 非空;
Process(P);
If P 仍然不可执行
  删除 P;
Endif
End;

Procedure Process(Path P)
Begin
t:=Path P 中的第1条变迁;
While (t 非空)
  If (t 不可执行)
    Cycle:=Extract-Cycle(P,t)
Endif
If (Cycle 非空)
  Trial:=0;
  While (t 不可执行,并且 Trial<Max_cycle) Do
      Precedent:=Path P 中的第1条变迁;
      将 Cycle 插入到 Path P 中 Predcent 之后;
      从 Cycle 的第1条变迁开始,
        检查变迁上的谓词条件是否得到满足;
      Trial:=Trial+1;
    EndWhile
Else
  Return(Null);
EndIf
  t:=P 中的下一条变迁;
EndWhile
End.
```

启发过程"Executablization"用来判定测试路径是否可执行化,并调用"*Extract-Cycle*(P,t)"寻找最短的循环,如果存在这样的循环,就把它插入到不可执行路径中,通过更新有影响变量,实现路径的可执行化.在EFSM模型的描述中,主要有两种谓词表示:一元谓词 $f(x)$(这里,F 是一个布尔函数,如("*equl_zero*(x)")和二元谓词,形式如"*var1* R *var2*",这里,R 是一个关系操作符,如"<"和">"等.我们从路径 P 中定位到第1条不可执行的变迁 t,如果 t 中的一元谓词无法满足,我们就试图从 t 前面的变迁中找到含有相同的一元谓词(但具有不同的值)的 t_k,从而生成含有 t_k 的有影响的循环,并把它插入到 t 之前.如果不是一元谓词,我们就利用代数求值的技术,找出导致该变迁不可执行的变量,并初步判断通过变量值的增加或减少是否可使 t_k 得到执行.这样,我们就在 t 之前的变迁中搜索到适当改变该变量值的变迁 t_k,生成包含该变量的循环,并插入路径中.如果经过尝试,路径仍然不可执行化,则丢弃.

3 实例应用和分析

图1为简化的INRES协议状态机,该EFSM描述在文献[1,2]中作为实例进行了生成分析,为了便于进行算法生成能力之间的比较,我们也采用该协议进行测试生成分析.利用我们开发的测试生成工具TUGEN[6].首先将协议进行EBE描述,根据TUGEN的执行结果,下面我们进行具体的分析.

算法5. 变迁说明部分.

```
T1:?U.sendrequest
  !L.cr
t2:?L.cc
t7:?L.ack()
  if(number=no_of_segment)
  !U.moniter_complete(counter)
t11:if(counter>blockbound)
  !L.token_release
  !U.mointor_incomplete(number)
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
    !U.sendconfirm                    !L.token_release                  !U.dis_request
t3:?U.datarequest(sdu,n,b)            !L.disrequest                 t12:if timeout
    number:=0;                    t8:!L.ack()                           and counter<=blockbound
    counter:=0;                       if(number<no_of_segment)          !L.token_release
    no_of_segment:=n;                 and(not timeout)              t13:?L.resume
blockbound:=b;                        !L.dt(sdu[number])            t14:?L.block
t4:?L.tokengive                       number:=number+1              t15:?L.ack
    !L.dt(sdu[number])            t9:?L.block                       t16:?L.dis_request
start timer;                          if (not timeout)              !U.disindication  t17:?L.disrequest
number:=number+1;                       counter:=counter+1;         !U.disindication
t5:?L.resume                      t10:?L.resume
t6:if timeout                     if (not timeout)
    !L.token_release                  and (counter<=blockbound)
```

3.1 输入参数值的选择

输入参数值的选择对于测试例有直接的影响.这些值的选择可影响变迁的分支走向以及循环的重复次数.在TUGEN的协议EBE描述的变量声明中,用户可以为每个输入参数指定合法和非法的值,也可以根据算法提供取值策略,从变量的定义域或多个推荐值中进行选取.如果未指定参数值,当输入参数影响到控制流变化时,就要求用户为该参数输入合法的值.

3.2 可执行前缀序列的生成

表1列出了如图1所示的EFSM中的最短可执行前缀序列(设输入参数 n 和 b 等于2).

Table 1 Some of executabal preambles in Fig. 1

表1 图1所示EFSM中部分变迁的可执行前缀

Transition①	Executable preamble②	Transition	Executable preamble
t_2	t_1,t_2	t_{10}	$t_1,t_2,t_3,t_4,t_9,t_{10}$
t_3	t_1,t_2,t_3	t_{12}	$t_1,t_2,t_3,t_4,t_9,t_{12}$
t_4	t_1,t_2,t_3,t_4	t_{16}	$t_1,t_2,t_3,t_4,t_8,t_7,t_{16}$
t_5	t_1,t_2,t_3,t_5	t_{17}	t_1,t_{17}

①变迁,②可执行前缀.

我们首先寻找最短可执行前缀序列.假设我们生成了 t_3 和 t_7 之间的全部可执行Du-Path,就必须为 t_3 找到一条前缀序列,否则,从 t_3 到 t_7 的任一路径都是不可行的.

3.3 Du-Path的生成

表2给出了从 t_9 到 t_{10} 的部分Du-Path(前缀为 t_1,t_2,t_3,t_4),并给出了丢弃的原因.

Table 2 Some Du-Path from t_9 to t_{10}

表2 从 t_9 到 t_{10} 的部分Du-Path

Du-Path	Discard?③	Causes④
$t_1,t_2,t_3,t_4,t_9,t_{10},t_6,t_3,t_7$	No⑤	–
$t_1,t_2,t_3,t_4,t_9,t_{10},t_6,t_4,t_7$	Yes⑥	Predicate of transition t_7 becomes (3=2)⑦
$t_1,t_2,t_3,t_4,t_9,t_{10},t_8,t_5,t_4,t_7$	No	–
$t_1,t_2,t_3,t_4,t_9,t_{10},t_7$	Yes	After being executabal, it is same as the first Du-Path⑧

①丢弃否,②丢弃原因,③否,④是,⑤t_7上的谓词变成了(3=2),⑥在可执行化后与第1条Du-Path相同.

我们定义变量 n 和 b 等于2.对于变迁 t_{11},最短的前缀为($t_1,t_2,t_3,t_4,t_9,t_{11}$),但 t_{11} 不可执行,因为它的谓词"counter>2"经过一次重复执行后,成为"1>2"了.根据我们的算法,有影响的变量是"counter",在 t_{11} 之前的变迁中,t_9 使变量"counter"得到了增加.生成循环"t_{10},t_9",并在变迁 t_9 之后

© 中国科学院软件研究所 http://www.jos.org.cn

插入两次.这样,路径就成为(t_1,t_2,t_3,t_4,t_9,t_{10},t_9,t_{10},t_9,t_{11}).

同时,在过程"Executablization"中,我们还引入了循环阈值 Max_Cycle.可以通过适当地调节 Max_Cycle 来控制程序的执行时间和生成循环的长度.

3.4 可执行测试序列的生成

表3给出了部分可执行测试序列(不含状态判定序列).

Table 3 Some executable test sequences

表3 部分可执行测试序列

Executable test sequences①	Test purposes②
t_1,t_2,t_3,t_5,t_4,t_8,t_7,t_{16}	Number,counter,no_of_segement
...	...
t_1,t_2,t_3,t_4,t_9,t_{10},t_9,t_{10},t_9,t_{11},t_{16}	Number,counter,blockbound
t_1,t_2,t_3,t_4,t_8,t_7,t_{13},t_{14},t_{15},t_{16}	Number,counter,blockbound

①可执行测试序列,②测试目的.

3.5 TTCN 测试例的生成

最后,在生成可执行测试路径后,还需要将测试序列转换成 TTCN 的机器处理(MP)格式,基于可执行测试路径,我们可以构造出含有正确性、不确定性和防护性分支的测试树,并对树中的每一个叶节点赋予3种测试判决之一.测试树可以完整地映射到一个 TTCN 格式的测试例.将测试例按照一定的规则予以分组,并附加上 TTCN 测试集的定义说明部分,便可以得到完整的 TTCN 格式的测试集.

3.6 测试生成的验证

在对测试生成方法的研究中,我们特别注重测试生成的验证,并根据 ISO/IEC 的 FMCT,将测试集错误覆盖率作为评测测试集的一个标准.我们采用 ITEX 工具提供的分析错误覆盖率的功能来完成我们的验证工作.

ITEX 是 Telelogic 公司(http://www.telelogic.se)提供的用于软件开发的集成软件环境,它支持 SDL 语言,并能由 SDL 描述生成实现仿真,提供接口良好的 TTCN 编辑器,可编译成专用的测试程序,与仿真实现进行通信,达到虚拟测试的目的,并可根据测试的结果,给出测试集的错误覆盖率评价.

在 TUGEN 中,影响测试集错误覆盖率的因素主要有两个:协议描述的正确程度和输入参数的选择策略,两者都与用户有关,需要用户根据 ITEX 分析的反馈结果,对协议描述和约束集描述进行修改,以期得到更优化的协议测试集.

4 总结和展望

本文提出的协议测试集自动生成方法生成的测试例兼顾了协议数据流测试和控制流测试,具有更高的测试覆盖.另外,该方法通过采用启发式循环分析等新的算法,不仅可以在测试路径生成的同时进行可执行性的判定,避免无用路径的生成,而且进一步节省了算法空间和执行时间.目前,根据本文给出的算法思想,我们已经完成了测试生成工具 TUGEN[6]的编制,并已经应用到 TCP/IP 协议(包括 TCP,PPP,OSPF)的测试集的自动生成工作之中,不仅有力地支持了国家"九五"科技攻关项目"多功能协议集成测试系统"的测试任务,而且加速了"高性能路由器"项目的研制开发,取得了令人满意的效果.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Chanson, S. T., Zhu, Jin-song. A unified approach to protocol test sequence generation. In: Harris, C. T., ed. Proceedings of the IEEE INFOCOM. San Francisco: IEEE Publishers, 1993. 106~114.

[2] Huang, Chung-ming, Lin, Yuan-chuen, Jang, Ming-yuhe. Executable data flow and control flow protocol test sequence generation for EFSM-specified protocol. In: Roman, T., ed. Proceedings of the International Workshop on Protocol Test Systems (IWPTS). Evry: Kluwer Academic Publishers, 1995. 273~283.

[3] International Organization for Standardization. ISO 9646: Conformance Testing Methodology And Framework Part 3: The Tree And Tabular Combined Notation (TTCN). Paris: ISO/IEC Publishers, 1991.

[4] Sabnani, K., Dahbura, A. A protocol test generation procedure. Computer Networks and ISDN Systems, 1988,15(2):285~297.

[5] Shen, Y. N., Lombardi, F., Dahbura, A. T. Protocol conformance testing using multiple UIO sequences. IEEE Transactions on Communications, 1992,40(8):323~335.

[6] Wang, Jian-guo, Hao, Rui-bing, Wu, Jian-ping. TUGEN: an automatic test case generator integrating data-flow and control-flow test methods. In: Cari, J. Y., ed. Proceedings of the IEEE International Conference on Communications. Piscataway, NJ: IEEE Publishers, 1998. 286~290.

An Extended Finite State Machine Based Generation Method of Test Suite*

WANG Jian-guo, WU Jian-ping

(*Department of Computer Sciences and Technology, Tsinghua University, Beijing* 100084, *China*)

E-mail: wjg@csnet1.cs.tsinghua.edu.cn

http://www.tsinghua.edu.cn

Abstract: Compared with other test generation methods, the problem of automatically generating executable test cases is addressed in this paper. First, the existing test generation methods for EFSM——specified systems are also presented. These methods combine both control and data flow techniques, but the executability problem has not been solved. For this purpose, a methodology which will solve, if not completely, partially the executability problem and mainly the problem of including all the influencing loops in the test sequences and finding how many times an unbounded loop must be executed is proposed in this paper. In addition, the detailed algorithms are presented.

Key words: conformance testing; test case generation; tree and tabular combined notation

* Received May 20, 1999; accepted March 24, 2000

Supported by the National Natural Science Foundation of China under Grant Nos. 69682002, 69725003

© 中国科学院软件研究所 http://www.jos.org.cn