有限状态机模型测试序列生成算法研究

李元平[1,3] 李 华[1,2] 赵俊岚[3]
(内蒙古大学计算机学院 呼和浩特 010070)[1] (内蒙古大学网络中心 呼和浩特 010070)[2]
(内蒙古财经大学网络中心 呼和浩特 010070)[3]

摘 要 在测试工程学中,应用测试生成树构建测试序列是相关测试方法的基础步骤,在传统测试生成树的基础上加入约束集的概念,使产生的测试生成树符合生产实际。同时在面向状态识别的测试方法中,考虑约束集对所生成状态区分序列的影响,基于带约束的测试生成树产生相应的特征集、状态识别集和 UIO 序列,提出或者改进了相应的算法。同时将测试方法扩展到了 NFSM 的情形下,提出了 NFSM 模型中前缀序列的生成算法和状态识别集的构建算法;结合状态识别矩阵与有限状态机同步乘积,提出在 NFSM 模型中的适应性测试方法,扩展了 FSM 应用于测试理论的完备性。建立了相应的测试方法工具集,实现了上述算法,验证了其可行性。最后给出了下一步的工作。

关键词 DFSM,NFSM,约束集,适应性测试
中图法分类号 TP301.6 **文献标识码** A

Research about FSM Test Sequence Generation Algorithm

LI Yuan-ping[1,3] LI Hua[1,2] ZHAO Jun-lan[3]
(College of Computer Science, Inner Mongolia University, Hohhot 010070, China)[1]
(Center of Network and Information, Inner Mongolia University, Hohhot 010070, China)[2]
(Center of Network and Information, Inner Mongolia University of Finance and Economics, Hohhot 010070, China)[3]

Abstract In the testing engineering, the basic step of the related test methods is to apply the test generated tree to construct test sequence. In this article, we added constraint set on the traditional test generated tree, making the tree more conform to the practice. We also considered the fluence of constraint set on the state separate sequence. We generated character set, states identification set, UIO sequence, and then proposed or advanced the algorithms. At the same time, we extended test method in NFSM, and proposed algorithms to generate prefix sequence and sate separate sets. The algorithm combines state separate matrix and FSM product, and is used in the adapt test of NFSM model, which makes the test theory on FSM completely. We constructed a test tool, which realizes these algorithms, and verifies the capability. Finally, the research work in the future was considered.

Keywords DFSM, NFSM, Constraint set, Adapt test

1 引言

随着软件规模的不断增长及分布式软件与大数据的兴起,人们对软件质量的要求在不断增强。当前开源组件的蓬勃发展,软件集成的门槛在降低,同时人们对质量的保障提出了更高的要求。软件测试是提高软件可靠性、保证软件质量的重要手段。传统的软件测试依赖于手工进行,测试的全面性、覆盖率与正确性无法得到保证,测试过程无法重用,在需要回归测试的场景中,会产生巨大的浪费,属于劳动密集型的活动。因此,提高软件测试效率、保证软件测试质量、降低软件测试费用及测试自动化成为必然的趋势[1]。

自动化测试分为基于模型的测试、基于描述的测试与基于代码的测试[1]。测试工程中,利用 FSM 作为形式描述的手段进行基于模型的测试开展的时间相对较早,在针对确定型有限状态机的测试中建立了相对比较完备的理论。但是其在实际测试工程中应用得相对较少,其中一个重要的原因是所建模型 Ms 与实际系统 Mi 之间尚有差距。文献[2]注意到了此问题,在 FSM 中引入了迁移约束集的概念,限定了测试生成树的行为。本文的主要贡献如下:1)在文献[2]的基础上进一步扩展迁移约束集的使用,在新的数据结构基础上完善并实现了作者提出的算法,其针对文献[1]中的算法提出了等价的算法,其效率与原算法相同,可读性相对清晰;同时提出了基于约束的深度优先测试生成算法,产生对应的基于约束的测试生成树。2)本文提出了基于测试生成树的 W 特征集构建算法,依托产生的基于约束的测试生成树,构建基于约束的 W 特征集。进一步使得在形式建模的基础上推导得到的测试例符合实际需求。3)本文同时扩展了 Wp 方法,提出了基于测试生成树的状态识别集构造算法,提出了在满足假设 3

本文受国家自然基金项目资助项目:面向属性的 CPN 建模及 On the Fly 辅助的测试生成方法研究(61163011),赛尔网络下一代互联网技术创新项目:SDN 环境下 IPv6 网络测试研究(CERNET IPv6 Innovation Project)(NGII20150112)资助。
李元平(1981-),男,博士生,中级实验师,主要研究方向为计算机网络、形式化方法,E-mail:liyuanping_9@163.com;李 华(1965-),女,教授,博士生导师,主要研究方向为形式化方法,E-mail:cslihua@imu.edu.cn。

条件下的 UIO 序列生成算法。4)本文借鉴文献[3,4]中的相关算法与文献[2]中状态识别函数的思想,在 NFSM 模型下给出了前导序列、状态识别序列的构建算法,相应给出了适应性测试的相关步骤,针对文中 NFSM 示例进行了相关测试序列的生成。同时完成了测试方法工具集,实现了上述算法,验证了其可行性。

2 预备知识

协议工程学中应用最广的模型是有限状态机模型,根据输出函数与当前状态是否相关可以分为 Mealy 机和 Moore 机。由于在任何时刻 Mearly 都处于某个状态,与输入输出相结合,适合于协议建模[5],应用比较广泛。其一般形式化定义如下。

定义 1 Mearly 机是一个五元组 $M=(I,O,S,\delta,\lambda)$,其中,I 表示输入符号的集合;O 表示输出符号的集合;S 表示状态的集合;δ 表示状态转移函数,形式化定义为 $S\times I\rightarrow S$;λ 表示输出函数,形式化定义为 $S\times I\rightarrow O$。

在 Mearly 机中输入序列 $x=a_1a_2\cdots a_k$,根据状态转移函数 $\delta(s_i,a_i)=s_i+1(i=1,2,\cdots,k)$ 可以获得状态序列 $s_1s_2\cdots s_k$,本文采用函数 $\delta(s_i,x)$ 表示该组状态序列。同理,采用 $\delta(s_i,x)$ 表示 $o_1o_2\cdots o_k$。本文采用 Ms 表示描述用户需求的 Mearly 机,采用 Mi 表示针对用户需求的实际实现。实际测试中在根据用户需求或者相关描述得到 Ms 模型的基础上,根据测试准则得到包含输入输出的测试序列,将此测试序列的输入信息作用于 M_i 之上,观察所得输出是否与预期输出一致。该种测试称为一致性测试,或者等价性测试[1]。

现有基于 FSM 的测试理论建立在一些假设之上。

假设 1 Ms 是被约简的或最小化的。对于任意两个状态 s 和 t,存在一个输入序列 x,使得 x 能依据输出函数 $\lambda(s,x)$ 和 (t,x) 的不同,区分 s 和 t,形式描述为 $\forall s,t:S,\exists x:I^*\cdot(\lambda(s,x)\neq\lambda(t,x))$。

假设 2 Ms 是完全描述的。S 中的每一个状态和 I 中的每个输入符号都能应用状态迁移函数 δ 和输出函数 λ。形式描述为 $\forall s_i:S,\forall x:I\cdot(\exists s_j\in S,\exists y\in O^*\cdot(\delta(s_i,x)=s_j)\wedge\lambda(s_i,x)=y))$。

假设 3 Ms 是强连通的。Ms 中的每一个状态都可经过一条或多条连续的迁移由其他状态到达。形式描述为 $\forall s_i,s_j:S,\cdot(\exists p:I^*\cdot\delta(s_i,x)=s_j))$。

某些基于 FSM 的测试方法并不严格遵守假设 3,仅要求 Ms 中的每一个状态都能由初态到达。

假设 4 在测试过程中,M_i 不会改变,且 M_i 与 Ms 拥有相同的输入和输出符号集合,形式描述为 $I_{Ms}=I_{M_i}\wedge O_{Ms}=O_{M_i}$。

一般的测试活动中,上述 4 种假设必须要遵循。其余部分的假设更多是为了简化测试,突出问题本质。根据实际问题,不同的假设条件需适当放开。

假设 5 初始状态。Ms 和 M_i 都有一个初始状态 s_1,且测试之前 M_i 已经处在他的初始状态。

假设 6 状态数目相同;Ms 和 M_i 有相同数目的状态,Ms 和 M_i 之间的不一致性错误并不会造成 M_i 中状态的增加。

此种假设条件下,测试所包含错误分为两类[2,6]:1)输出错。输出函数 λ 在相同输入的情况下所得输出与预期不一致。2)状态转移错。状态转移函数 δ 在相同输入的情况下所得状态与预期不一致。实际情况中,假设 6 一般是不满足的,其中 Mearly 机 M_i 中所包含的状态一般要大于等于 Ms 中所包含的状态,此种错误称为额外状态错误。由额外状态错误会导致相应额外迁移错误,具体表现为 Ms 中某些输入在一定状态的条件下是非法的,但是在 M_i 中可能就是可以接受的。此种错误利用第 3 节中的扩展 Wp 方法可以检测。

假设 7 重置消息。Ms 和 M_i 都有一个重置消息 *reset*(简称 r),使得状态机在任何状态时都能回到初始状态 s_1 且不会产生任何输出。

假设 8 状态位消息。Ms 和 M_i 都有一个特定的状态位(status),使得在任意时刻,它们所处的状态都能被状态位标识,例如状态 s_i 表明了状态所处的状态位是 i。

假设 9 设置消息。输入符号集合 I 包含一个特殊的集合 $set(s)$,当系统处于初始状态时,自动机获得一个 $set(s)$ 消息后能够移动到状态 s,同时不产生任何输出。

3 基于约束集的 DFSM 测试序列生成

定义 2(区分序列) 对于 S 中的任意两个状态 s 和 t,如果存在序列 x 使得 $\lambda(s,x)\neq\lambda(t,x)$,则此序列称为状态 s 与状态 t 的区分序列。形式描述为:$\forall s,t:S,\exists x:I^*\cdot(\lambda(s,x)\neq\lambda(t,x))$。

如果系统满足假设 8,可以利用状态覆盖、变迁覆盖等方法进行测试,或者采用中国乡村邮递员算法进行遍历。其中状态覆盖无法探测某些输出错误与转换错误。如果状态无法利用观察手段得到,则利用区分序列进行相关测试,比较常用的测试方法有 D 方法、U 方法、W 方法及其改进版本 WP 方法等。上述测试方法各有优缺点,D 方法相对方便,但是不是所有的 FSM 都存在 D 序列。可以证明在最小化的 FSM 中,总是可以运用 W 方法找到用于区分其状态的特征集[2],但是利用 W 方法找到的测试序列存在测试效率不高的缺点,WP 方法是基于 W 方法的改进。一般比较常用的方法是 U 方法。测试工程中,为了使所建模型与被测实际更加拟合,需要在模型中加入约束条件,针对迁移活动进行相关限定。

3.1 基于约束的广度优先测试生成树算法

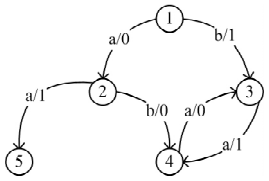
存在如下的测试场景:学生选课与学生缴费,一般情况下这两个活动没有依赖关系,在特定要求下,只有在缴费完成之后才可以进行选课。此时选课与缴费存在约束关系。

定义 3(迁移约束集[2]) 在 FSM 中,t_i 和 t_j 为两条迁移且 t_j 的存在取决于 t_i 的存在,形式描述为 $t_i\rightarrow t_j$,称为迁移约束。$RS=\{(t_i,t_j)\mid t_i\rightarrow t_j\}$ 称为迁移约束集。

一般来说,存在两种方式针对上述问题进行处理:1)生成全部的测试序列集,利用迁移约束集对该测试序列集进行过滤。2)构建测试序列生成算法之时即保留满足迁移约束的测试序列。第一种方式中测试空间会变得非常巨大,状态爆炸,测试爆炸的问题会导致测试工作量巨大,测试成本无法降低,测试效率无法得到提升。第二种方式的测试空间相对较小,尽早过滤掉无效测试序列,提升了测试效率,降低了测试成本,此种思路在基于属性的测试中体现得更为明显[7]。本文重点关注第二种处理方式。

在利用区分序列方法中,无论是构建输入序列覆盖集,还是在某些方法中构建区分序列本身都要用到测试生成树。文献[2]中测试生成树采取邻接矩阵存储的方式,时间复杂度为

$O(n^2)$。本文以邻接表的形式对上述算法进行了实现，较原算法，可读性更好。该算法时间复杂度为 $O(n*m)$，其中 n 为节点个数，m 为迁移的个数。采用邻接表的存储方式，程序可以进行动态的扩展，避免在起始阶段限定程序规模，同时在生成测试路径之时可以快速与边信息相结合，获取输入输出信息。本文以图 1(a)所示的有限状态机为例说明通过广度优先遍历得到基于约束的测试生成树与通过深度优先遍历得到基于约束的测试生成树。XML 文件中对应的迁移约束如图 1(b)所示。



(a)

```
〈Test_Constraint〉
 〈Graph_Edge〉
 〈From_Graph_Vertex〉state_02〈/From_Graph_Vertex〉
 〈To_From_Graph_Vertex〉state_04〈/To_From_Graph_Vertex〉
 〈Input_Data〉b〈/Input_Data〉
 〈Output_Data〉0〈/Output_Data〉
 〈/Graph_Edge〉
 〈Graph_Edge〉
 〈From_Graph_Vertex〉state_04〈/From_Graph_Vertex〉
 〈To_From_Graph_Vertex〉state_03〈/To_From_Graph_Vertex〉
 〈Input_Data〉a〈/Input_Data〉
 〈Output_Data〉0〈/Output_Data〉
 〈/Graph_Edge〉
〈/Test_Constraint〉
```

(b)

图 1　有限状态机与迁移约束集

算法 1　基于迁移约束的广度优先测试生成树构建算法

输入：XML 格式的 FSM 描述以及约束集描述信息

输出：XML 格式的满足约束集的广度优先测试生成树

1. 根据配置文件初始化 FSM 信息，约束集信息
2. 有限状态机初始定点入队列
3. While(队列不空)
4. 当前队列头出队列，if 当前节点尚未被处理，找到节点对应的边信息
5. If 边信息集合为空，设置节点状态为已处理
6. Else 对每一条边信息进行处理
7. 每一条边的末端节点入队列
8. 检查该边是否需要检查
9. If 不需要，按照一般流程处理
10. Else 是否符合约束集要求，如果符合，检查该边的前导序列是否也符合要求，如果符合要求，按照一般流程处理；否则跳过本次边处理
11. If 该边处理正确，设置边为测试生成树分支
12. If 每一条边都处理正确，设置当前节点为已处理
13. 跳转 3
14. 信息写入 XML 文件

由图 2 可以看到生成的广度优先与深度优先测试生成树中约束集 $\{t_1 \rightarrow t_2\}$，其中 $t_2: s_4 \xrightarrow{a/0} s_3$，$t_1: s_2 \xrightarrow{b/0} s_4$ 满足要求。基于约束集的深度优先测试生成树算法实现将队列部分替换为堆栈得到即可，如图 3 所示，正常情况下测试序列为{a0a1，a0b0，b1a1a0}，满足约束的测试序列集为{a0a1，a0b0a0，b1a1}。

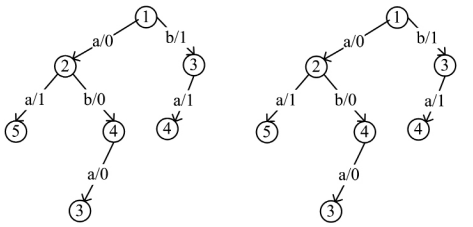


图 2　广度优先测试生成树与带约束的广度优先测试生成树

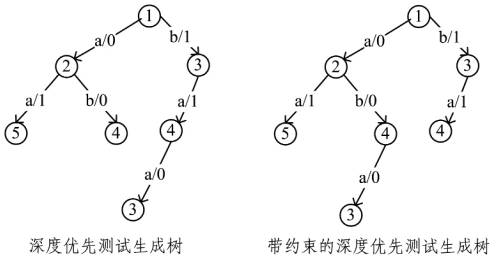


图 3　深度优先测试生成树与带约束的深度优先测试生成树

3.2　基于约束的特征集构建

定义 4(特征集[2])　Ms 的特征集是由分离序列构成的集合，以便对 S 中的任意状态 s 和 t，W 中都存在一个输入序列 x 使得 $\lambda(s,x) \neq \lambda(t,x)$。形式描述为：$\forall s,t: S, \exists x: W \cdot (\lambda(s,x) \neq \lambda(t,x))$。

基于确定型有限状态机的测试方法中，W 方法是重要且基本的方法。已经存在的 W 方法可以在文献[2,5]中找到相应的解释。可以利用状态矩阵的方式通过状态识别函数构建可识别矩阵得到特征集[2,8]。但是上述算法产生的特征集中都没有考虑约束的成分。为了使得依据状态机模型产生的测试序列更加符合系统的实际运行情况，需要在产生的状态特征集中考虑是否符合约束。本文提出在已经产生的基于约束的测试生成树中构建 W 集，具体请见算法 2。其中针对每个顶点构建测试生成树的部分请见算法 1。算法 2 的时间复杂度为 $O(n^3)$，与经典的特征集构造算法的时间复杂度相同。

算法 2　基于约束的特征集构建算法

输入：XML 格式的 FSM 描述以及约束集描述信息

输出：特征集 W

1. 初始化 W=Φ；
2. 从 FSM 状态集合中选取未处理状态 S；
3. Ws=Φ；
4. 以 S 为根节点广度优先遍历构建基于约束的测试生成树，具体见算法 1，生成迁移覆盖集，选择迁移覆盖集中的序列 Px 区分 S 与各个未处理的状态；
5. Ws= Ws∪Px；
6. 设置 S 为处理状态；
7. W=W∪Ws；
8. 转步骤 2，直到所有状态都处理完毕。

一般情况下，利用算法 1 得到的测试生成树构建迁移覆盖集 P。同时根据算法 2 中得到的状态特征集 W 得到测试集合 T，形式描述为：$T=\{t \mid : \forall p \in P, \forall w \in W, t=r \cdot p \cdot w\}$。在基于假设 6 的前提下，$W$ 方法可以发现所有的输出错与状态转移错[6]。

3.3　基于约束的 Wp 方法

定义 5(状态覆盖集[2])　状态覆盖集 Q 是由输入序列组成，且对于任意的 $s \in S$，都存在一个输入序列 $x \in Q$，使得自动机迁移到 s，即 $\delta(s_1,x)=s$，形式描述为：$\forall s: S \cdot (\exists x: Q \cdot \delta(s_1,x)=s)$。

W 方法具有比较良好的错误探测能力。但是在实际的测试环境中，W 方法存在测试序列过长的问题，本文提出在基于约束的测试生成树的基础之上构建基于约束的状态识别集的算法，具体见算法 3，其时间复杂度为 $O(n^3)$。

算法 3　基于约束的状态识别集构建算法

输入：XML 格式的 FSM 描述以及约束集描述信息

输出：状态识别集 W_i

1. 初始化 $W_i=\Phi$；
2. 从 FSM 状态集合中选取未处理状态 S_j；
3. 以 S_i 为根节点广度优先遍历构建基于约束的测试生成树，具体见算法 1，生成迁移覆盖集，选择迁移覆盖集中的序列 P_x 区分 S_i 与各个未处理的状态 S_j；
4. $W_i=W_i\cup P_x$；
5. 设置 S_j 为处理状态；
6. 转步骤 2，直到所有状态都处理完毕；
7. W_i 即为所求。

基于约束的 Wp 方法分为两个阶段进行测试。测试准备阶段根据基于约束的测试生成树得到迁移覆盖集 P、状态覆盖集 Q、特征集 W 和状态识别集 W_i。

第一阶段执行测试集合 $T_1=\{t\mid \forall q\in Q,\forall w\in W,t=r\cdot q\cdot w\}$。

第二阶段执行测试集合 $T_2=\{t\mid \forall r'\in R,\forall w_i\in W_i,t=r\cdot r'\forall w_i\}$，其中 $R=P-Q$。

根据上述执行步骤以及算法 2 与算法 3 之间的不同可以得知，在已经确认 Ms 中的状态在 M_i 中存在的情况下，用状态识别集替代特征集可以得到等价的测试效果[9]。同时由于状态识别集为特征集的子集，由理论推演可知 Wp 方法可以在测试效果等价的前提下减少测试序列长度，降低测试成本，提升测试效率。

3.4　基于约束的 UIO 方法

定义 6（UIO 序列[2]）　在 Mearly 机中，状态 s 的 UIO 序列是一条输入序列 x，满足任意其他状态 $t\in S$，都有 $\lambda(s,x)\neq\lambda(t,x)$。

Wp 方法中，如果状态识别集中只包含一条序列，称此序列为状态签名或 UIO 序列。UIO 方法中，测试序列集合为 $T=\{t\mid \forall p\in P,t=r\cdot p\cdot UIO_i\}$。一般情况下 UIO 序列都存在[10]，并且 UIO 方法的序列要小于 W 方法产生的测试序列。但是某些场景中，如果 Ms 中的状态 S 与错误实现 M_i 中的状态 S' 的 UIO 序列相同，则无法通过 UIO 方法检测得到该错误。

在所建模型 Ms 满足假设 3 的前提下，本文提出了根据基于约束的状态识别集 W_i 得到基于约束的 UIO_i 的算法。具体内容见算法 4，其时间复杂度为 $O(n^4)$，本算法较文献[2]中的算法相对简洁与清晰，同时加入了约束信息，与实际系统的拟合度更高。

算法 4　基于状态识别集构建 UIO 算法

输入：XML 格式的 FSM 描述，状态识别集 $W_1,W_2,\cdots,W_n$

输出：$UIO_1,UIO_2,\cdots,UIO_n$

1. 初始化 $UIO_i=\Phi$，sequence_s$=\Phi$；
2. For(i=1;i<=n;i++)
3. If |Wi|=1,UIOi=Wi;
4. Else
5. 根据状态识别集 W_i 中的每个序列 t_i 与 FSM 描述信息，得到序列 t_i 到达的状态 s_j，形成序列集合 $T=\{t_i\}$；
6. While(序列集合 T 不空)
7. 取得 t_i，根据第一步信息得到对应状态 s；以 s 为根节点，广度优先遍历构建基于约束的测试生成树，具体见算法 1，得到从 s 到状态 i 的序列 sequence_s；
8. $UIO_i=UIO_i\cdot t_i\cdot$ sequence_s；
9. 转步骤 6；
10. $UIO_i=$Substring(UIO_i,0,length(UIO_i)−length(sequence_s))；
11. End for；
12. Return $UIO_1,UIO_2,\cdots,UIO_n$

3.5　基于约束的扩展 W 方法

前文提到在假设 6 满足的前提下，有两类错误模型。在实际的测试工程中，M_i 中的状态一般要多于 Ms 中的状态，即必须考虑假设 6 不满足的情形。为此需要引入区分集[9]的概念。

定义 7（区分集）　在 Mearly 机中，区分集被定义为 $Z=X[m-n]\cdot W$。其中 $X[m-n]=\{\varepsilon\}\cup I^1\cup I^2\cdots\cup I^{m-n}$，$I$ 为 Mearly 机的输入符号集，n 为 Ms 中的状态数目，m 为 M_i 中的状态数目，且 $m\geqslant n$，W 为特征集。

本文采用算法 1 构建特征集，则此时区分集是满足迁移约束的区分集。实际的测试工程中，采用区分集 Z 代替特征集，可以发现 M_i 中的额外状态错与额外迁移错。扩展 W 方法的检错能力要强于 T 方法、W 方法、Wp 方法、U 方法与 D 方法。扩展 W 方法中，测试集合 T 的形式描述为：$T=\{t\mid :\forall p\in P,\forall z\in Z,t=r\cdot p\cdot z\}$。当 $m-n$ 越大时，扩展 W 方法的测试序列越长。

可以在基于约束的扩展 W 方法的基础上进一步得到基于约束的扩展 Wp 方法，此时测试过程分为两个阶段。测试准备阶段根据基于约束的测试生成树得到迁移覆盖集 P、状态覆盖集 Q、特征集 W 和状态识别集 Wi 和区分集 Z。

第一阶段执行测试集合 $T1=\{t\mid \forall q\in Q,\forall z\in Z,t=r\cdot q\cdot z\}$。

第二阶段执行测试集合 $T2=\{t\mid \forall r'\in R,\forall w_i\in W_i,t=r\cdot r'\cdot X[m-n]\cdot w_i\}$，其中 $R=P-Q$。

文献[11]提出了 R-Wp 方法。在保证与扩展 Wp 方法有相同错误探测能力的前提下减少测试输入序列长度。此时，第二阶段执行测试集合 $T2=\{t\mid \forall r'\in R,\forall w_i\in W_i,t=r\cdot r'\cdot w_i\}$，其中 $R=P-Q$。由约束集的含义可知，R-Wp 方法也可改写加入此约束。

3.6　基于切分树构建适应性测试序列方法

基于切分树构建测试序列的过程中，经常要用到的概念为后继树与切分树，以表 1 所列的有限状态机为例，则其对应后继树与切分树如图 4 所示。本文给出利用广度优先方式由经过规约的有限状态机构建切分树的算法，以及通过切分树得到适应性测试序列的算法。

表 1　有限状态机

	a	b
S1	S1,0	S2,1
S2	S2,1	S3,1
S3	S3,0	S1,0

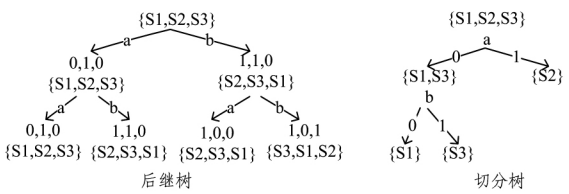


图 4　有限状态机对应后继树与切分树

算法 5　基于有限状态机构建切分树算法

输入:经过规约的有限状态机

输出:切分树 ST

1. 初始化 S={S1,S2,…,Sn}
2. S 入队列
3. While(队列不空){
4.　S=出队列;
5. If(S 不是单节点){
6.　　按照算法 8 产生切分序列,将 S 依据此序列进行切分,依据输出的不同产生新节点;
7.　　新节点入队列;
8.　　}
9. }

算法 6　基于切分树构建适应性测试算法

输入:完全切分树 ST

输出:适应性区分序列

1. I 初始状态集,C 当前状态集;
2. 设定初始输入;
3. While(|I|>1){
4.　依据当前输入与当前状态集,更新初始状态集与当前状态集;
5.　在切分树中查找包含当前状态集的最小节点,将其对应输入序列更新为当前输入;
6. }

由算法 6 可得图 4 所对应的适应性测试序列为{ab}。

4　基于 NFSM 的 Adapt 测试序列生成

定义 8　部分非确定有限状态机(PNFSM)是一个五元组 $M=(I,O,S,\delta,S_0)$,其中:I 表示输入符号的集合;O 表示输出符号的集合;S 表示状态的集合;δ 表示迁移函数,形式化定义为 $S\times I\rightarrow powerset(S\times O)\mid\{\emptyset\}$;其中 *powerset* 表示幂集,$\emptyset$ 表示空集。给定集合 A,B,则 $A\mid B=\{x\mid x\in A\wedge x\notin B\}$。$S_0$ 是初始状态。

本文中,非确定有限状态机需要用到一些形式化定义符号,其具体含义如表 2 所列。

表 2　PNFSM 记号

记号	含义
L	l∈L 表示输入输出对
Ø	空序列
L*	L 上的序列集,X 代表这样的序列(X∈L*)
P|l→	P,Q∈S,(∃Q(P−l−Q))为假
P=ε=>Q	P=Q
P=a/b=>Q	P−a/b→Q
P=x=>Q	∃P1,…,Pk∈S,(P=P0=l1=>P1…lk=>Pk)x=l1…lk
P=x=>	∃Q∈S(P=x=>Q)
Tr(P)	Tr(P)={x|P=x=>}为源状态 P 的所有迁移上的输入/输出对
X^{in}	x^{in}代表输入序列
$Tr^{in}(P)$	$Tr^{in}(P)=\{x^{in}\mid P=x=>\}$
V^{in}	对 $V\subseteq L^*$,$V^{in}=\{x^{in}\mid x\in V\}$

定义 9　可观察部分确定有限状态机(OPNFSM)。$\forall s\in S$,$\forall l\in L(S-l-S_i\wedge S-l-S_j\rightarrow i=j)$,则 PNFSM 是可观察的。OPNFSM 的输入输出对唯一确定下一状态,但是 OPNFSM 依然是非确定型的,因为它的输入和状态不能唯一确定输出和下一状态。其中 L 是有穷的输入输出集。

定义 10(可区分状态)　如果 $\exists x\in Tr(S_i)\oplus Tr(S_j)(x^{in}\in Tr^{in}(S_i)\cap Tr^{in}(S_j))$,则称状态对 S_i,S_j 是可区分的,记为 $S_i\neq S_j$. $Tr(S_i)\oplus Tr(S_j)=(Tr(S_i)\cup Tr(S_j))\mid Tr(S_i)\cap Tr(S_j)$。两个状态是可区分的当且仅当两个状态同时接受 x^{in},但是只有一个状态接受 x。

定义 11(最小化 PNFSM)　如果 $\forall S_i,S_j:S,(i\neq j\rightarrow S_i\neq S_j)$。PNFSM 是最小的,当且仅当每对状态均是可区分的。

定义 12(等价性)　PNFSM 内的状态 P,Q 间的等价关系 $P\equiv Q$ 成立 iff $Tr(P)\equiv Tr(Q)$。给定两个 PNFSM S 和 I 以及对应的初始状态 S_0 和 I_0,如果 $S_0\equiv I_0$,则 $S\equiv I$。称实现 I 等价于它的规范 S 当且仅当 $S\equiv I$。

定义 13(单输入有限状态机)　PNFSM 迁移函数 δ 中,如果 $(s,i_1,o_1,s_1),(s,i_2,o_2,s_2)\in\delta$,则 $i_1=i_2$。

4.1　NFSM 状态前导序列

基于 NFSM 的测试,与传统的测试生成树方法不同,本文采用逆向分析的手段,由终态出发,以初态结束,建立每个状态的单输入前导序列。应用算法 7,针对图 5 所示非确定有限状态机,构建每个状态的前导序列子状态机,同时根据算法规约到单输入有限状态机。各个状态对应的前导序列子状态机分别如图 6−图 8 所示,其前导序列集为{ε,$a1$,$a0c1b1$,$c0$,$c1a1$,$c1$,$c0c1$}。

算法 7　NFSM 状态前导序列构建算法

输入:FSM M,s∈S,s≠s_0

输出:如果状态 s 初始可达,产生其前导序列

1. 初始化构建 FSM R(I,O,R,δ,r_0)其中
　R:={s};
　δ:=Φ;
2. While 存在状态 s′∉R 且 s′≠s_0
3.　如果存在迁移序列(s′,i,o,s) ∈δ_m
4. R:=R∪s′;
5. δ:=δ∪(s′,i_m,o_m,s);
6. End While;
7. If 存在迁移序列(s0,i,o,s) ∈δ_m,则
　R:=R∪s0;δ:=δ∪(s0,i_m,o_m,s);
8. If $s_0\notin$R then terminate and stop;
9. Else FSM R 即为所求。
10. 从 FSM R 的初始状态开始,如果状态的输入信息个数大于 1,则删除相应的输入信息。将信息所在边一并删除。直到该 FSM R 为单输入有限状态机。
11. 如果存在状态∈R,由 r_0 不可达,删除该状态。
12. FSM R 即构成所求前缀。

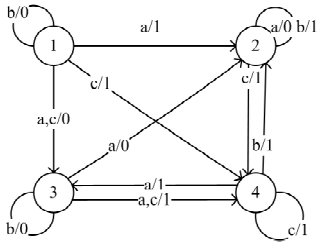


图 5　非确定有限状态机

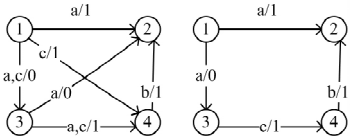


图 6　状态 2 前导序列子状态机及其单输入状态机

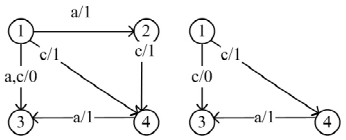


图 7　状态 3 前导序列子状态机及其单输入状态机

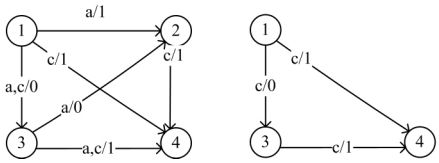


图 8　状态 4 前导序列子状态机及其单输入状态机

4.2　NFSM 状态区分序列

定义 14(特征集 W)　给定 OPNFSM,特征集 $W\subseteq I^*$ 是最小集,使 $\forall S_i,S_j:S(S_i\neq S_j\rightarrow\exists x\in Tr(S_i)\oplus Tr(S_j)(x^{in}\in Tr^{in}(S_i)\cap Tr^{in}(S_j)))$。

定义 15(可识别函数[2])　$\Phi:S\rightarrow I/O$,其中 S 是有限状态机状态集,I/O 表示输入/输出集合,且满足 $\forall S_i:S,\exists x/y:I/O,\exists S_j$。$(S_i,x,y,S_j)\in\delta$。

依据定义 15,图 4 中的可识别函数为 $\Phi(1)=\{b/0,a/1,c/1,a/0,c/0\}$,$\Phi(2)=\{c/1,a/0,b/1\}$,$\Phi(3)=\{b/0,a/1,c/1,a/0\}$,$\Phi(4)=\{c/1,b/1,a/1\}$。

定义 16(区分序列)　给定 OPNFSM S 与其中的可区分状态 $s1$ 和 $s2$。其交集 $S/s1\cap S/s2=(I,O,Q,\delta_{S/s1\cap S/s2},s1s2)$。$FSM\ P=(I,O,P,\delta_P,s1s2)$,其中 $P=Q\cup\{s1,s2\}$,$\delta_P=\delta_{S/s1\cap S/s2}\cup\{(ss',i,o,s1)\mid ss'\in Q,o\in out(s,i)\backslash out(s',i)\}\cup\{(ss',i,o,s1)\mid ss'\in Q,o\in out(s',i)\backslash out(s,i)\}$。则 P 为区分序列对应的有限状态机。

依据定义 16,图 5 所示的非确定有限状态机各个状态之间的区分序列状态机如图 9、图 10 所示。

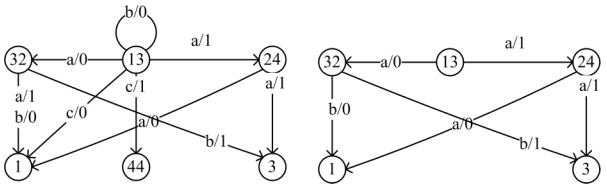


图 9　状态 1 与状态 3 区分序列状态机及其单输入状态机

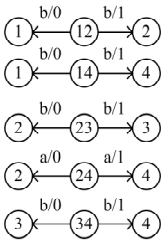


图 10　其他状态区分序列单输入状态机

定义 17(状态识别矩阵)　在有限状态机中,可识别函数的状态识别矩阵由 D_{ij} 构成,其中

$$D_{ij}=\begin{cases}\Phi(s_i)-\Phi(s_j), & \Phi(s_i)\not\subset\Phi(s_j)\\ \varnothing, & \text{Others}\end{cases}$$

图 5 所示的非确定有限状态机对应的三角矩阵为:

$$\begin{array}{c}\\ s_1\\ s_2\\ s_3\\ s_4\end{array}\begin{array}{c}\begin{array}{cccc}s_1 & s_2 & s_3 & s_4\end{array}\\ \begin{bmatrix}\varnothing & \{b/0,a/1,c/0\} & \{c/0\} & \{b/0,a/0,c/0\}\\ \{b/1\} & \varnothing & \{b/1\} & \{a/0\}\\ \varnothing & \{b/0,a/1\} & \varnothing & \{b/0,a/0\}\\ \{b/1\} & \{a/1\} & \{b/1\} & \varnothing\end{bmatrix}\end{array}$$

由有限状态机可识别函数三角矩阵得到对应的单输入有限状态机可识别函数三角矩阵,如下所示:

$$\begin{array}{c}\\ s_1\\ s_2\\ s_3\\ s_4\end{array}\begin{array}{c}\begin{array}{cccc}s_1 & s_2 & s_3 & s_4\end{array}\\ \begin{bmatrix}\varnothing & \{b/0\} & \{c/0\} & \{b/0\}\\ \{b/1\} & \varnothing & \{b/1\} & \{a/0\}\\ \varnothing & \{b/0\} & \varnothing & \{b/0\}\\ \{b/1\} & \{a/1\} & \{b/1\} & \varnothing\end{bmatrix}\end{array}$$

文献[2]提出了状态可识别函数的概念,给出了有限状态机分离序列识别的三角矩阵定义,该方法具有简单、直观的特点,但是该定义局限在确定性有限状态机的范围之内。文献[3]提出了在非确定性有限状态机的条件下,通过有限状态机的交集运算,同时结合定义 16 与算法 7,得到区分状态之间的区分序列有限状态机。该方法相对复杂,且程序运行效率不高。在节点数量增加的情况下,复杂度呈现爆炸性递增。本文结合文献[2,3]中的相应算法,将文献[2]中的相应算法扩展到非确定有限状态机下,得到该非确定有限状态机的单输入有限状态机,同时针对扩展后的算法中难以区分的状态节点对,采用文献[3]中的相应算法对其进行处理。根据实际情况,大部分的区分状态都可以采用扩展后的算法处理,提高了处理效率,减少了时间复杂度。

由单输入有限状态机可识别函数三角矩阵可知,$D_{31}=\varnothing$,则将其单独处理,由图 9 可知 $D_{31}=\{a/0b/0\}$,$D_{13}=\{a/0b/1\}$。更新相应状态对,最终状态对如下所示。

$$\begin{array}{c}\\ s_1\\ s_2\\ s_3\\ s_4\end{array}\begin{array}{c}\begin{array}{cccc}s_1 & s_2 & s_3 & s_4\end{array}\\ \begin{bmatrix}\varnothing & \{b/0\} & \{a/0b/1\} & \{b/0\}\\ \{b/1\} & \varnothing & \{b/1\} & \{a/0\}\\ \{a/0b/0\} & \{b/0\} & \varnothing & \{b/0\}\\ \{b/1\} & \{a/1\} & \{b/1\} & \varnothing\end{bmatrix}\end{array}$$

其完整算法描述如下。

算法 8　NFSM 状态区分序列构建算法

输入:NFSM M

输出:NFSM M 各状态对区分序列

1. 对 NFSM M 中的各个状态,依据定义 15 得到状态识别函数;
2. 依据定义 17,得到状态识别三角矩阵;
3. 对每个状态对依据 $x^{in}\in D_{ij}\cap D_{ji}$ 进行剪枝,则 x 为所求单输入有限状态机状态区分序列;
4. 在单输入有限状态机查找包含 $\varnothing$ 的位置,其中排除对角线位置;
5. 依据定义 16,对包含 $\varnothing$ 的位置,分别以此位置对为初始状态,构建区分序列有限状态机;
6. 根据得到的区分序列,更新步骤 3 中得到的输入有限状态机状态识别矩阵;

4.3　NFSM 迁移覆盖集合

类比于确定型有限状态机中的迁移覆盖集,非确定性有限状态机的迁移覆盖集为定义 7 中的 δ 集合。

4.4　NFSM 适应性测试

NFSM 适应性测试步骤分为 3 个阶段。其中第一阶段是测试准备阶段,第二阶段是前导序列测试阶段,第三阶段是迁移集测试阶段。

测试准备阶段根据 NFSM 的有限状态机描述将其化简为可观察的 ONFSM,同时利用算法 7、定义 15 与算法 8 分别得到前导序列集 P、迁移覆盖集 S、状态识别集 W_i。

前导序列测试阶段执行测试集合 $T_1=\{t\mid\forall p\in P,\forall w\in W_i,t=r\cdot p\cdot w\}$。通过该测试步,得到测试例集合{ε,b0,a1a0,a1,a1b1,c1a1,c1a1a0b1,c1a1b0,c1b1}。

迁移集测试阶段执行测试集合 $T2=\{t\mid \forall p\in P,\forall s\in S,\forall w\in W_i,t=r\cdot p\cdot s\cdot w\}$。由定义 15 知，$\Phi(1)=\{b/0,a/1,c/1,a/0,c/0\}$，此时在状态 1 输入 a，观察输出的迹为 $a1$，执行状态 2 的状态识别集，则对应的迹为 $a1b1,a1a0$；在状态 1 输入 b，观察输出的迹为 $b0$，执行状态 1 的状态识别集，对应的迹为 $b0b0,b0a1a0$；在状态 1 输入 c，观察输出的迹为 $c1$，执行状态 4 的状态识别集，对应的迹为 $c1a1,c1b1$。此时状态 1 对应的测试集为$\{a1b1,a1a0,b0b0,b0a1a0,c1a1,c1b1\}$。同理得到状态 2、状态 3 与状态 4 的对应测试集。全部的测试集为$\{a1a0a0,a1a0b1,a1b1a0,a1b1b1,a1c1a1,a1c1b1,b0a1a0,b0b0,c1a1a0a0,c1a1a0b1,c1a1b0a0b1,c1a1b0b0,caa1c1a1,caa1c1b1,c1b1a0,c1b1b1,c1c1a1,c1c1b1\}$。

5 工具的设计与实现

测试工程中，为了实现测试的自动化，测试工具是其中的关键部分，是自动化得以实现的重要载体，测试工具的开发是测试工程的重要课题。

本文以 MyEclipse Enterprise Workbench 2014 为开发环境，在 1.8.0_60 版本的 64 位 java 虚拟机下构建了 FSM 测试算法工具集。该工具由配置模块得到 FSM、约束集等相关信息，通过配置信息解析模块在内存中构建顶点、边以及相应 FSM 的图形表示，由测试生成树产生模块按照遍历算法与约束集要求构建测试生成树。此测试生成树以 XML 格式存储，可以方便地得到树中节点到另一节点的正向与反向序列。在基于约束的测试生成树的基础上，应用前文中提到的算法产生特征集、状态识别集与 UIO 序列，将其与状态覆盖集、迁移覆盖集相结合产生测试序列。测试工具的模块构建图如图 11 所示。

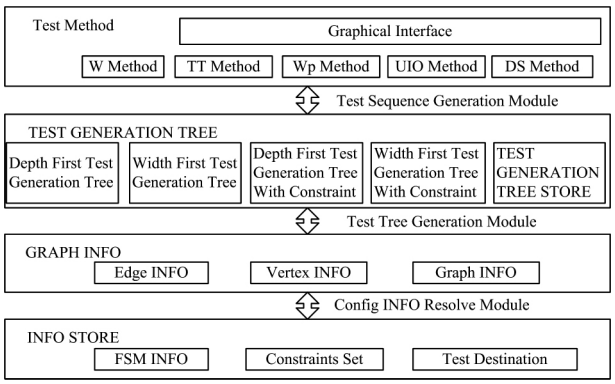


图 11 测试工具模块构建图

5.1 FSM 构建模块

本文所构建的工具中，配置信息从 XML 文件中读取。其中 XML 文件需满足一定的格式要求，如图 12 中 XSL 格式定义所示。

```
〈xs:element name="FSM_Graph"〉
  〈xs:complexType〉
    〈xs:sequence〉
      〈xs:element name="Graph_Vertex_List" type="Graph_Vertex_List_Type" minOccurs="1" maxOccurs="1"/〉
      〈xs:element name="Graph_Edge_List" type="Graph_Edge_List_Type" minOccurs="1" maxOccurs="1"/〉
      〈xs:element name="Test_Constraint" type="Test_Constraint_Type" minOccurs="0" maxOccurs="1"/〉
      〈xs:element name="Test_Destination" type="Test_Destination_Type" minOccurs="0" maxOccurs="1"/〉
    〈/xs:sequence〉
  〈/xs:complexType〉
〈/xs:element〉
```

图 12 XSL 格式定义

构建模块基于当前主流的 DOM4J 开源软件实现，整合了 DOM 和 SAX 的优点，性能超过 sun 公司官方的 DOM 技术。在 init_grahp_structure 函数调用中根据配置信息在内存中重构 FSM。数据结构采用逻辑的链式存储结构，以 Map〈graph_vertex，ArrayList〈graph_edge〉〉具体完成实际顶点与边的关联配置。

5.2 测试生成树构建与存储模块

测试生成树构建模块实现了前文所提到的算法 1，加入了未带约束的广度与深度优先遍历生成树。本模块同时支持基于约束的深度与广度优先遍历生成树。本模块可以将得到的测试生成树导出成 XML 文件，其树节点信息如图 13 所示。

```
〈Tree_Node〉
  〈Edge_Node_Id〉86e0fb1e-58f0-4dac-953f-c18f58b6c129〈/Edge_Node_Id〉
  〈Parent_Edge_Node_Id〉627c271b-348d-4aaf-b202-0db1c8e475b6〈/Parent_Edge_Node_Id〉
  〈Graph_Edge〉
    〈From_Graph_Vertex〉state_02〈/From_Graph_Vertex〉
    〈To_From_Graph_Vertex〉state_04〈/To_From_Graph_Vertex〉
    〈Input_Data〉b〈/Input_Data〉
    〈Output_Data〉0〈/Output_Data〉 〈/Graph_Edge〉
  〈Succeed_Edge_Node〉
    〈Edge_Node_Id〉587e6b51-c864-44cd-830e-9a07725c7def〈/Edge_Node_Id〉
  〈/Succeed_Edge_Node〉
〈/Tree_Node〉
```

图 13 测试生成树存储结构

5.3 迁移覆盖集与测试方法模块

本模块实现了前文中的状态集识别算法，如 W，Wp，UIO，扩展 Wp 等，同时与迁移覆盖集和状态覆盖集相结合，产生了测试序列，其执行过程见日志信息，如图 14 所示。

```
2016-08-25 23:24:28] INFO main test_tree_construct_with_constraint.java:154-Depth First Traversal with Constraint Begin
==========
[2016-08-25 23:24:28] INFO main graph.java:36-vertex graph_vertex [is_visited=false, vertex_label=state_01]is not exist.
[2016-08-25 23:24:28] INFO main graph.java:43-add vertex graph_vertex [is_visited=false, vertex_label=state_01]
[2016-08-25 23:24:28] INFO main graph.java:36-vertex graph_vertex [is_visited=false, vertex_label=state_02]is not exist.
[2016-08-25 23:24:28] INFO main graph.java:43-add vertex graph_vertex [is_visited=false, vertex_label=state_02]
[2016-08-25 23:24:28] INFO main graph.java:36-vertex graph_vertex [is_visited=false, vertex_label=state_03]is not exist.
[2016-08-25 23:24:28] INFO main graph.java:43-add vertex graph_vertex [is_visited=false, vertex_label=state_03]
[2016-08-25 23:24:28] INFO main graph.java:36-vertex graph_vertex [is_visited=false, vertex_label=state_04]is not exist.
[2016-08-25 23:24:28] INFO main graph.java:43-add vertex graph_
```

vertex [is_visited=false, vertex_label=state_04]
[2016-08-25 23:24:28] INFO main graph.java:36-vertex graph_vertex [is_visited=false, vertex_label=state_05]is not exist.
[2016-08-25 23:24:28] INFO main graph.java:43-add vertex graph_vertex [is_visited=false, vertex_label=state_05]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_01], to=graph_vertex [is_visited=false, vertex_label=state_02], input_data=a, output_data=0]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_01], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=b, output_data=1]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_02], to=graph_vertex [is_visited=false, vertex_label=state_05], input_data=a, output_data=1]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_02], to=graph_vertex [is_visited=false, vertex_label=state_04], input_data=b, output_data=0]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_03], to=graph_vertex [is_visited=false, vertex_label=state_04], input_data=a, output_data=1]
[2016-08-25 23:24:28] INFO main graph.java:66-add edge graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_04], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=a, output_data=0]
[2016-08-25 23:24:28] INFO main graph.java:173-the graph has been constructed as below:
graph [graph_edge_test_constraint_list=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_02], to=graph_vertex [is_visited=false, vertex_label=state_04], input_data=b, output_data=0], graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_04], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=a, output_data=0]], graph_edge_test_destination_list=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_04], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=a, output_data=0]], graph_struct={graph_vertex [is_visited=false, vertex_label=state_04]=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_04], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=a, output_data=0]], graph_vertex [is_visited=false, vertex_label=state_03]=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_03], to=graph_vertex [is_visited=false, vertex_label=state_04], input_data=a, output_data=1]], graph_vertex [is_visited=false, vertex_label=state_01]=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_01], to=graph_vertex [is_visited=false, vertex_label=state_02], input_data=a, output_data=0], graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_01], to=graph_vertex [is_visited=false, vertex_label=state_03], input_data=b, output_data=1]], graph_vertex [is_visited=false, vertex_label=state_02]=[graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_02], to=graph_vertex [is_visited=false, vertex_label=state_05], input_data=a, output_data=1], graph_edge [from=graph_vertex [is_visited=false, vertex_label=state_02], to=graph_vertex [is_visited=false, vertex_label=state_04], input_data=b, output_data=0]]}][2016-08-25 23:24:28] INFO main test_tree_construct_with_constraint.java:163-The current processed vertex is : graph_vertex [is_visited=false, vertex_label=state_01]

图 14　测试序列生成日志信息

结束语　本文在基于 FSM 的测试中，加入约束集的情形，使之与被测实现进一步拟合。分别通过广度优先和深度优先进行了相应测试生成树的构造，并在测试树的基础上加入约束集，对生成树算法进行了优化。在此测试生成树的基础上针对依据状态识别算法的测试方法进行了改进，提出或者改进了基于约束的 W，Wp，UIO，$UIOv$ 等方法。同时针对很多被测对应非确定型的有限状态机[11]，提出在 NFSM 模型中构建前导序列与状态识别集的算法。该算法以状态识别函数为基础，构建状态识别函数矩阵，同时结合有限状态机的同步乘积算法，在算法效率与准确性方面对当前算法[16]进行了优化。

如何在本文构建的测试方法工具集的基础上考虑效率最大化的测试工程[12]，使得整个测试可控，以及使测试人员与测试资源等成本管理与测试产出达到合适的比例是下一步的工作重点。

参 考 文 献

[1] Bertolino A. Software testing research: Achievements, challenges, dreams[C]// 2007 Future of Software Engineering. IEEE Computer Society, 2007: 85-103

[2] 刘攀，缪淮扣，曾红卫，等. 基于 FSM 的测试理论、方法及评估[J]. 计算机学报，2011，34(6)：965-984

[3] Petrenko A, Yevtushenko N. Adaptive testing of deterministic implementations specified by nondeterministic FSMs[C]// IFIP International Conference on Testing Software and Systems. Springer Berlin Heidelberg, 2011: 162-178

[4] Petrenko A, Yevtushenko N. Adaptive testing of nondeterministic systems with FSM[C]// 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering. IEEE, 2014: 224-228

[5] Chow T S. Testing software design modeled by finite-state machines[J]. IEEE Transactions on Software Engineering, 1978, 4(3): 178

[6] 蒙移发，徐惠民，高强. 协议验证与一致性测试方法[J]. 计算机科学，2002，29(5)：40-42

[7] Machado P D L, Silva D A, Mota A C. Towards property oriented testing[J]. Electronic Notes in Theoretical Computer Science, 2007, 184: 3-19

[8] 刘攀. 基于 FSM 的测试用例生成和测试优化[D]. 上海：上海大学，2010

[9] Fujiwara S, Bochmann G, Khendek F, et al. Test selection based on finite state models[J]. IEEE Transactions on Software Engineering, 1991, 17(6): 591-603

[10] 毕军，吴建平. 基于 FSM 的形式化测试序列生成方法[J]. 软件，1995(8)：15-21

[11] Luo G, Petrenko A, Bochmann G. Selecting test sequences for partially-specified nondeterministic finite state machines[M]// Protocol Test Systems. Springer US, 1995: 95-110

[12] Juristo N, Moreno A M, Vegas S. Reviewing 25 years of testing technique experiments [J]. Empirical Software Engineering, 2004, 9(1/2): 7-44