Data structure

**//状态机状态**

st {

id // 节点id

status // 一个n维向量 (s1,…,sn）每个元素 si 表示第i个承诺目前的状态值

}

**//状态转移矩阵**

M=[m][m] // m\*m 的矩阵，其中m表示的是状态机的状态个数 M(i,j)有如下定义

// M(i,j)= 空 表示第i个st不能转移到 第j个st

// 不为空时 表示第i个st在接收到该事件后转移到 第j个st

**//承诺*cc***

cc {

x, // 承诺人

y, // 被承诺人

p, // 前提

r, // 结果

tc // 时间

}

List CC // 里面记录了n 个承诺的合理状态列表

queue.append(root\_st) // 使用广度优先搜索 遍历所有的合理状态 因此需要将初始状态入队

while (len(queue)>0){ //当状态队列长度不为0 就遍历合理状态

st <- queue.pop(); //st 是多个承诺的状态组合 st=（*cc*1,…,*cc*n）

List children <- createChildrenNodes(st) //查找所有合理子状态

for (each (st’ in children){ //将所有合理子状态入队

queue.append(st);

}

}

//根据目前状态节点 st，返回所有合理的子状态，并将状态转移记录到M中

createChildrenNodes( st){

List rlt <- *Φ* //建立列表存储所有的子状态

for(i=1:n){ //遍历状态中的所有承诺的状态 跟据承诺的定义只有1 2 才能转变

if (st.status[i]==1）

List act<- hadel\_act(st,i); //当状态为1时 查找所有的子状态

rlt.add(st’); // 将子状态加入列表

if (st.status[i]==2）

List bas<- hadel\_bas(st,i); // 当状态为2时 查找所有的子状态

rlt.add(st’); // 将子状态加入列表

}

//处理st.status[i]==1的情况 输入合理的状态组合，以及状态为1 的下标

//返回该状态组合的子状态

hadel\_bas(st,i){

List rlt <- *Φ* // 建立空列表存储所有的子状态

st’ <- // 创建新的状态节点，st’会被自动分配一个id

st’.status <- st.status; // 将状态组态赋值给新的st

st’.status[i] <- 2; // 当承诺的前提及时完成时则置为2

//检查生成的状态是否重复

M(st.id, st’.id) <- “TURE(CC[i].p)” // 将触发的动作填入转移矩阵

rlt.add(st’); // 将子状态加入列表

st’.status <- st.status; // 将状态组态赋值给新的st

st’.status[i] <- 5; // 当承诺的前提未及时完成时则置为5

M(st.id, st’.id) <- “Timeout(CC[i].p)” // 将触发的动作填入转移矩阵

rlt.add(st’); // 将子状态加入列表

//检查生成的状态是否重复

}

//处理st.status[i]==2的情况 输入合理的状态组合，以及状态为2 的下标

//返回该状态组合的子状态

hadel\_bas(st,i){

List rlt <- *Φ* // 建立空列表存储所有的子状态

st’ <- // 创建新的状态节点，st’会被自动分配一个id

st’.status <- st.status; // 将状态组态赋值给新的st

st’.status[i] <- 3; // 当承诺的后果及时完成时则置为3

//检查生成的状态是否重复

M(st.id, st’.id) <- “TURE(CC[i])” // 将触发的动作填入转移矩阵

rlt.add(st’); // 将子状态加入列表

st’.status <- st.status; // 将状态组态赋值给新的st

st’.status[i] <- 4; // 当承诺的后果未及时完成时则置为4

M(st.id, st’.id) <- “Timeout(CC[i])” // 将触发的动作填入转移矩阵

rlt.add(st’); // 将子状态加入列表

//检查生成的状态是否重复

}