计算机科学 1996 Vol. 23 №. 2

论公平转换系统模型的可信性

贾国平 郑国梁
(南京大学计算机系 南京 210093)

摘 要 In this paper, we consider the model of fair transition system (FTS) which is now being popularly applied in the modeling of concurrency. In particularly, we discuss its faithfulbility. We compare the computations which are executed by FTS model with these which are executed by actual concurrent systems, and we present limited-critical reference constraint (LCR condition). This conclusion provides a theoretical basis for the application of FTS model.

关键词 Fair Transition System, Formal Model, Concurrent System, Fairness, Faithfulbility.

1. 引 言

近些年来，并行性(Concurrency)理论发展很快，为正确理解和对分布式计算系统复杂行为的推理方法提供了一个理论基础，然而，如何使大规模并行性实用化却是一个关键问题。现在所研究的并行性模型主要是抽象数学模型，都基于这样一个共同的思想：从实时细节中抽象，并且用一个抽象的不可分活动集作为原子构造块。这一基本思想在许多方面已被证明特别有用。

程序开发中现有的每一种形式化和分析方法其有效性范围均是相对于程序执行的某一数学模型，并非实际系统，尽管对于其模型完全有效，但这些方法对于程序实际的应用却只能由其对应模型表示实际程序执行的可信度来决定，通过一种形式化方法所得到的关于程序执行的结论，在某种程度上，只有当其基本的数学模型是可靠的，它才是可靠的。因此在开发形式化方法过程中，很重要的一点是不仅应该包括构造新的模型和用于分析此模型的技术，而且还应该包括不断对模型之间以及它们与实际系统之间进行比较和评价。

本文我们分析和讨论了当前广泛用于模型化往复(Reactive)和并发系统的公平转换系统模型，此模型最初由文[4]中给出。文[1,2,3,7,8]中，均未对此模型进行讨论，尤其对其可靠性问题。为了讨论此问题，我们考虑了许多在程序实际并行执行中所可能发生的现象，对比并发程序在公平转换系统模型中的执行，我们对程序语句的语法进行约束，提出了限制临界引用约束(LCR 条件)，得到结论，在此约束条件下，公平转换系统模型是可信的。

2. 公平转换系统模型

我们首先提出一般的公平转换系统模型[4]，作为一种抽象的实现语言，给出与此模型有关的一些基本概念，用一阶语言表达公平转换系统的语法，此语言中公式称为断言。

一个公平转换系统 P 是一个六元组(V, Σ, Θ, T, WF, SF)，其中：

$V=\{u_1,\cdots,u_k\}$：状态变量的有穷集合。这些变量是数据变量和控制变量，前者由程序中的语句显式说明及操作；后者指出程序执行过程中将要执行的程序或语句的位置。

Σ：状态集合。每一状态 $s\in\Sigma$ 是 V 的一个解释，给每一变量 $u\in V$ 赋给其对应论域中的一个值。用 s[u]表示。

Θ：初始条件。它是一个表示全体初始状态的断言，也即程序可以开始执行的状态断言。一个状态 $s\in\Sigma$，如果它满足 Θ，即 $s\models\Theta$，则我们称 s 为初始状态。我们要求 Θ 是可满足的，即至少存在一个状态满足 Θ。

T：有穷转换集合。每一转换 $\tau\in T$ 是一函数：$\Sigma\to 2^{\Sigma}$，它将每个状态 $s\in\Sigma$ 映入(可能空)τ-后继状态

贾国平 南京大学计算机系博士生，郑国梁 南京大学计算机系教授，博士生导师。

集 $\tau(s)\subseteq\Sigma$。一个转换在状态 s 下是能行的当且仅当 $\tau(s)\neq\varnothing$，否则 τ 在此状态下是不能行的。我们在 T 中加入一个转换 τ_I，称为空转换，它是一个单位转换，即对每一状态 $s\in\Sigma$，$\tau_I(s)=\{s\}$。每一转换 τ 都由一个断言来表示，记为 $\rho_\tau(V,V')$，称为转换关系。它将状态 $s\in\Sigma$ 与它的 τ-后继状态 $s'\in\tau(s)$ 通过无上撇号和有上撇号的状态变量联系起来。无上撇号的状态变量引用 s 中的值；有上撇号的状态变量引用 s' 中的值。

$WF\subseteq T$：弱公平性转换集。直观上，对转换 $\tau\in WF$ 的弱公平性要求不允许一个计算，其中转换 τ 从某一点开始一直能行，但只有有穷多次被执行。

$SF\subseteq T$：强公平性转换集。直观上，对转换 $\tau\in SF$ 的强公平性要求不允许一个计算，其中 τ 无穷多次能行，但只有有穷多次被执行。

我们说转换关系 $\rho_\tau(V,V')$ 标识出 s' 是 s 的一个 τ-后继，是指 $\langle s,s'\rangle\models\rho_\tau(V,V')$，其中 $\langle s,s'\rangle$ 是联合解释，将 $x\in V$ 解释为 $s[x]$，$x'\in V'$ 解释为 $s'[x]$。

对于 Σ 中的无穷状态序列 $\sigma:s_0,s_1,\cdots$ 及一个转换 $\tau\in T$，我们称 τ 在 σ 中位置 j 以后一直能行，是指对每一 $k\geqslant j$，τ 在 s_k 是能行的。称 τ 在 σ 中无穷多次能行，是指对无穷多个 k，τ 在 s_k 是能行的。

我们称 Σ 中的无穷状态序列 $\sigma:s_0,s_1,\cdots$ 是公平转换系统 P 的一个计算(Computation)，是指 σ 满足：1)初始化条件：s_0 是初始状态，即 $s_0\models\Theta$。2)连续性：对每一 $j=0,1,\cdots$，状态 s_{j+1} 是状态 s_j 的 τ-后继状态，即存在转换 $\tau\in T$，$s_{j+1}\in\tau(s_j)$。此时我们称转换 τ 在 σ 的 j 位置被执行。注意，在一给定位置可以有多个转换被执行。3)弱公平性：对每一转换 $\tau\in WF$，不会发生 τ 在 σ 中某一位置以后一直能行，但只有有限多次被执行。4)强公平性：对每一转换 $\tau\in SF$，不会发生 τ 在 σ 中无穷多次能行，但只有有限多次被执行。

我们常常将计算表示为由带标号的箭头连接而成的状态序列，用使系统转移到下一状态的转换作为箭头上的标号，即计算由 $s_0\xrightarrow{\tau_0}s_1\xrightarrow{\tau_1}s_2\cdots$ 来表示。它除了指出 $s_0,s_1,\cdots$ 是一个计算，同时也标识出转换 $\tau_i\in T$ 使系统从状态 s_i 转移到状态 s_{i+1}。

3. 程序设计语言

本节我们给出一简单程序设计语言，其中进程通过共享变量进行通信。先给出每个基本语句形式及其它们的直观解释，然后给出它们与一般的公平转换系统模型之间的对应，即给出一般模型中的每一抽象实体到我们所定义的具体语言模型相应结构中的映射。关于许多其它一般的程序设计语言到公平转换系统模型之间的对应关系，可参见文[1]及[6]。

3.1 语法

赋值语句：$y_1,\cdots,y_k:=(e_1,\cdots,e_k)$。其中 $y_1,\cdots,y_k$ 是一变量表，$e_1,\cdots,e_k$ 为相应类型的表达式表。当 $k=1$，简写为 $y:=e$，当 $k=0$，写为 skip。

等待语句：await c，其中 c 为一布尔表达式。我们称 c 为语句的卫士。等待语句的执行并不改变任何变量的值，其唯一目的是等待直到卫士条件 c 为真，此时语句终止。

条件语句：if c then S_1 else S_2，其中 S_1 和 S_2 为语句，c 为布尔表达式。条件语句的第一步是对 c 进行求值以及对 S1 或 S_2 的选择，尔后继续执行所选择的子语句。

合并语句：$S_1;\cdots;S_k$，其中 $S_1,\cdots,S_k$ 为语句。合并语句是一种顺序复合，第一步是语句 S_1 执行，其后步是继续执行 S_1 的剩余步。当 S_1 终止，继续执行 $S_2,S_3,\cdots,S_k$。

有了等待语句及合并语句，我们可以定义 when 语句：when c do $S\equiv_{df}$ await c ; S。

选择语句：S_1or$\cdots$orS_k，其中 $S_1,\cdots,S_k$ 为语句。选择语句执行的第一步是语句 $S_1,\cdots,S_k$ 中当前的一个能行的语句被选择且第一步被执行，其后步继续执行所选子语句的剩余语句。如果 $S_1,\cdots,S_k$ 中多于一个能行，则选择是非确定的，

选择语句常常应用于 when 语句，它们的组合称为条件选择语句。Dijkstra 的一般的卫士命令语言中的条件命令 if $c_1\rightarrow S_1\ \square\ c_2\rightarrow S_2\ \square\ \cdots\ \square\ c_k\rightarrow S_k$ fi 可表为下述条件选择语句：[when c_1 do S_1]or[when c_2 do S_2]or$\cdots$or[when c_k do S_k]。

while 语句：while c do S，其中 S 为一语句，c 为一布尔表达式。此语句的执行首先对 c 进行求值。如果 c 为假，则语句执行终止。否则，其后步继续执行 S。当 S 终止，c 重新被估值。

Request 语句：request(y)。其中 y 为一函数变量，它是一信号语句。此语句能行当 y 是正的。它的执行是从 y 中减 1。

Release 语句：release(y)。其中 y 为一函数变

量，它也是一信号语句。此语句能行当控制到达此语句之前。它的执行是将 y 加 1，等价于赋值语句 y：=y+1。

在上述程序设计语言中程序 P 的形式是：

P::〔说明；[$P_1::S_1 \| \cdots \| P_m::S_m$]〕。

其中，$P_1::S_1,\cdots,P_m::S_m$ 是有名进程。每一 S_i 是一个语句。P_i 是此进程的名字，程序和进程的名字是可选的。说明由一系列说明语句组成，每一说明语句具有形式：模式：变量…，变量：类型 where φ。其中，模式可能为 in，local 或 out，分别表示其后变量对程序的输入、局部的或程序的输出。变量，…，变量：类型说明这些变量均具有其后所说明的类型。φ 为一断言，它对说明语句中说明的某些变量的初始值施加约束。

令 $\varphi_1,\varphi_2,\cdots,\varphi_n$ 是出现在一个程序的说明语句中的所有断言，我们称合取式 $\varphi:\varphi_1\wedge\varphi_2\wedge\cdots\wedge\varphi_n$ 为程序的数据前置条件。

对一个程序的每个语句都用一标号来标识，我们要求出现在程序中的这些标号是不同的。关于这些标号的进一步讨论见文[6]。

3.2 语义

下面给出公平转换系统中每一单元到我们所定义的程序设计语言中的结构的映射。

状态变量 V 和状态集 Σ 对于上述定义的程序设计语言，状态变量 V 包含数据变量和一个控制变量 π。数据变量是由程序显式说明和控制的，它包括输入，输出和局部变量。它们在其所说明的数据论域中变化。π 在位置集上变化。在一个状态中 π 的值表示所有当前控制所在的程序位置。

我们将所有给状态变量赋给其相应论域中值的赋值解释作为我们的状态。Σ 即为所有这些状态的集合。

转换 T 对程序中的每一语句 S，归纳定义一个标号集合 After(S)，表示语句 S 执行结束后控制所在的位置。

- 对 S 是程序的一个进程，$After(S)=\varnothing$。
- $S=[l_1:S_1;\cdots;l_k:S_k]$，$After(S_k)=After(S)$。
- $S=$ if c then S_1 else S_2，$After(S_1)=After(S_2)=After(S)$。
- $S=[S_1 or\cdots or S_k]$ $After(S_i)=After(S)$，对 $i=1,\cdots,k$。
- $S=l:$while c do S，$After(S')=l$

对语句 $l:S$，常记 After(S) 为 After(l)。

下面我们定义与每个语句相联的转换及其转换关系。

赋值语句：$l:(y_1,\cdots,y_k):=(e_1,\cdots,e_k)$，相联转换 τ_l，其转换关系为：

$\rho_l:(\{l\}\subseteq\pi)\wedge(\pi'=(\pi-\{l\})\cup After(l))\wedge(y'_1=e_1)\wedge\cdots\wedge(y'_k=e_k)$。

等待语句：$l:$await c，相联转换 τ_l，其转换关系为：

$\rho_l:(\{l\}\subseteq\pi)\wedge c\wedge(\pi'=(\pi-\{l\})\cup After(l))$。

条件语句：$l:$if c then $[l_1:S_1]$else$[l_2:S_2]$，相联转换 τ_l^T 和 τ_l^F，其转换关系为：

$\rho_l^T:(\{l\}\subseteq\pi)\wedge c\wedge(\pi'=(\pi-\{l\})\cup\{l_1\})$及

$\rho_l^F:(\{l\}\subseteq\pi)\wedge\sim c\wedge(\pi'=(\pi-\{l\})\cup\{l_2\})$

While 语句：$l:$[while c do $[l':S']$]，相联两个转换 τ_l^T 和 τ_l^F，其对应转换关系为：

$\rho_l^T:(\{l\}\subseteq\pi)\wedge c\wedge(\pi'=(\pi-\{l\})\cup\{l'\})$

$\rho_l^F:(\{l\}\subseteq\pi)\wedge\sim c\wedge(\pi'=(\pi-\{l\}\cup After(l))$

Request 语句：$l:$request(y)，相联转换为 τ_l，其转换关系为：

$\rho_l:(\{l\}\subseteq\pi)\wedge y>0\wedge(\pi'=(\pi-\{l\})\cup After(l))\wedge(y'=y-1)$。

Release 语句：$l:$release(y)，相联转换 τ_l，其转换关系为：

$\rho_l:(\{l\}\subseteq\pi)\wedge(\pi'=(\pi-\{l\}\cup After(l))\wedge(y'=y+1)$。

我们称与信号语句 request 和 release 相联的转换为信号转换。

初始条件 对程序 P：〔说明：[$P_1::[l_1:S_1]\| \cdots \| P_m::[l_m:S_m]$]〕。令 φ 为程序的数据前置条件，则初始条件为：$(\pi=\{l_1,\cdots,l_m\})\wedge\varphi$。

弱公平性和强公平性 对我们定义的程序设计语言，它们分别为：弱公平性集：$T-\{\tau_l\}$；强公平性集：所有的信号转换集。

4. 交替模型评价

前面介绍了公平转换系统模型，其关键的一点是并行性在此模型中是由交替(Interleaving)来表示的。两个并行执行进程永远不会在同一时间点同时执行它们各自的语句，而总是交替执行原子转换。形式上，当其中一个进程执行一个原子转换时，其它的进程均处于非活动状态。这个计算模型易用于并发

程序的形式化、分析及处理，也是时序逻辑能够很好地用于程序规约和验证的基础。

然而，实际的并发系统常常由许多独立的处理器组成，每个处理器执行它们各自的进程。这样一个系统中，在不同处理器上执行的语句常常是重迭(Overlapping)的而不是交替的。本节要解决的一个关键问题就是对比公平转换系统模型定义的交替计算形式与实际系统执行中的重迭执行形式之间的不同，找出达到统一的方法。

我们必须解决的一个问题就是干扰问题。交替执行模型对干扰提供了比重迭模型更多的保护。交替执行要求当一个转换被执行时，所有其它转换都是非活动的，因此在一个转换之中无干扰，而对于重迭执行却不都如此。考虑语句 when y=y do S 在交替计算中，条件 y=y 在一个原子步中判定，因此它总是为真，而在重迭执行中(假设没有进行程序优化)，此条件语句的执行将两次引用 y，如果恰好在这两次引用中另一个重迭并行语句改变了 y 值，则判断结果可能为假。

4.1 交替与并行性

在具体比较重迭执行与公平转换系统模型中的交替执行行为之前，我们先来看看一个赋值语句，如 y:=y+1 的执行过程。假设系统在多处理器上执行，并发进程是共享内存的。对于其它情况，不难由此得到。一般，语句 y:=y+1 的执行由三个不同执行步组成：①取值步：取出对应于 y 的共享内存位置的值，将它存放于局部寄存器中。②计算步：寄存器增加，所得结果可能被存放在另一个局部寄存器中。③存贮步：结果寄存器中的值被存放于与 y 对应的共享内存位置上。

对其它的语句，这种分划是类似的。更复杂的赋值语句可能有多个取值步，这依赖于在语句右端所出现的变量个数。

从上述赋值语句的三个不同执行步的划分，可以看到，对共享内存位置的取值步和存贮步是赋值语句执行中的两个临界事件步，因为它们之间的相对执行顺序将决定执行的最终结果，而对于计算步，由于它的操作只是在内部寄存器中进行，它对其它进程不可存取，因此它并不是临界事件步。

下面我们考虑如下一个简单程序：Program 1：

out y : integer where y=1

$P_1::[l_0: y:=y+1; l'_0] \| P_2::[m_0: y:=y-1; m'_0]$

其中两个进程 P_1 和 P_2 均由上述的简单赋值语句组成。

令 $r_i(m)$ 表示 P_i 从 y 中取到值 m，$W_i(m)$ 表示 P_i 将值 m 存入 y 中(i=1,2)。因此得到 Program1 中四个临界事件步。假设它们是原子的，即它们的执行是不重迭的。对于简单的数据类型，这一假设常常可以由硬件来保证。

上述程序的重迭执行产生{0,1,2}作为 y 的可能结果集。下列是某些执行序列：

E1: $r_1(1), r_2(1), w_1(2), w_2(0)$ 产生 y=0

E2: $r_1(1), w_1(2), r_2(2), w_2(1)$ 产生 y=1

E3: $r_1(1), r_2(1), w_2(0), w_1(2)$ 产生 y=2

应该注意，此处我们所讨论的例子具一般性，在任何并发程序中，总可以找到程序执行中的某些事件，将它们作为临界事件。它们在时间上的执行顺序唯一决定了程序执行以及可观测行为的最终结果。例如对于通过共享变量进行通信的程序，其临界事件就是对共享变量的存和取。下面我们考虑程序 Program1 在公平转换系统模型中的可能计算结果。

对应于进程 P_1 和 P_2 中的赋值语句，模型首先给程序赋给两个转换 τ_{l_0} 和 τ_{m_0}(简记为 l_0 和 m_0)，因此在交替模型中可能的程序计算是下列序列：

$$(\{l_0, m_0\}, 1) \xrightarrow{l_0} (\{l'_0, m_0\}, 2) \xrightarrow{m_0} (\{l'_0, m'_0\}, 1) \cdots\cdots$$

$$(\{l_0, m_0\}, 1) \xrightarrow{m_0} (\{l_0, m'_0\}, 0) \xrightarrow{l_0} (l'_0, m'_0\}, 1) \cdots\cdots$$

即在公平转换系统模型中只有一个可能结果，即 y=1。

由此，我们看到公平转换系统模型似乎并不能完全刻画实际程序执行所得的行为集。然而，事实并非如此，上述问题并不是交替模型本身的问题，而在于对程序的语句赋给原子转换的赋值本身。我们在程序设计语言一节中已给出了映射规则：对每一个赋值语句都相联一个原子转换。正是这一规则导致了上述不希望有的结果。如对 l_0: y:=y+1，赋给原子转换 τ_{l_0}，它就迫使两个临界事件 r_1 和 w_1 在一步中发生，因此也就排除了其它临界步如 r_2 及 w_2 发生在它们之间的可能性。事实上，正是这些可能性才产生了上例中的结果 0 及 2。这也正是交替模型无法产生的根本原因。

基于上述分析讨论，一种解决办法是对每一赋值语句，如 l_0: y:=y+1，赋给两个原子转换。如 τ'_{l_0} 和 τ''_{l_0}，转换 τ'_{l_0} 执行取值步，转换 τ''_{l_0} 执行存贮步，而

计算步含于 τ'_{t_0} 或 τ''_{t_0} 中。这种方法大大地增加了转换的个数，使得模型本身非常复杂，分析、操作起来很不方便。另一解决办法是从程序语句本身的修改做起。程序中任一原子转换的产生仍然遵从前述规则，即每一个赋值语句对应一个原子转换，唯一的要求是每一原子转换至多包含一个临界事件。因此对程序 Program1 修改可得如下 Program2：

Out y: integer where y=1

$$P_1::\begin{bmatrix}\text{local} & t_1 & \text{intger}\\ l_0: & t_1:=y & l_0\\ l_1: & y:=t_1+1 & l_1\end{bmatrix}$$

$$\|\,P_2::\begin{bmatrix}\text{local} & t_2 & \text{intger}\\ m_0: & t_2:=y & m_0\\ m_1: & y:=t_2+1 & m_1\end{bmatrix}$$

Program1 中的每一个赋值语句被分成相应的 Program2 中的两个连续赋值语句，第一个语句执行取值步，而第二个执行存贮步。不难看出，Program2 的交替执行产生其对应的重迭执行的所有结果。

4.2 限制临界引用约束：LCR 条件。

现在我们对前面所定义的一般程序设计语言给出其临界事件的严格定义，然后给出临界引用定义，最后给出我们的限制临界引用约束。

定义 1 （读引用和写引用定义） 1)语句 skip 既无写引用又无读引用；2)赋值语句 $(u_1,\cdots,u_k):=(e_1,\cdots,e_k)$ 定义为对每一变量 $u_1,\cdots,u_k$ 有一写引用，而对出现在 $e_1,\cdots,e_k$ 中的所有变量有读引用；3)语句 await c，when c do S，if c then S_1 else S_2，while c do S，对所有出现在 c 中的变量有读引用。 □

注意，上述定义将读引用和写引用归于语句本身，而并不归于其中子语句的引用。

定义 2 一个对语句 S 中所出现变量的引用是临界的，如果：1)它是一个写引用，而在与 S 并行的语句中存在对此变量的读引用或写引用，或 2)它是一个读引用，而在与 S 并行的语句中存在对此变量的写引用。 □

定义 3 （限制临界引用约束）我们称一个语句 S 满足限制临界引用约束（即 LCR 条件），是指与 S 相联的每一转换至多执行一个临界引用。 □

注意，LCR 条件是指与语句 S 相联的每一转换中临界引用数目多少而言的，它并不是语句 S 中所包含的临界引用数目的多少。

定义 3 将 LCR 条件表示为对与一个语句相连的转换上的约束，下面将 LCR 条件形式化为对语句本身的语法约束：

- 语句 skip 总是满足 LCR 条件。
- 对于赋值语句 $\bar{u}:=\bar{e}$，LCR 条件是 $\bar{u}$ 和 $\bar{e}$ 一共只含有至多一个临界引用。
- 对语句 await c，when c do S，if c then S_1，else S_2，while c do S，LCR 条件是 c 至多包含一个临界引用。
- 对于合并语句及选择语句，由于并没有自己唯一的转换，它们总是被认为满足 LCR 条件。
- 对同步语句 request 和 release，并无显式约束。由于它们内在的保护机制阻止干扰，因此都被认为只有一个临界引用。
- 对于消息传递程序，情况特别简单。除了通信语句 Send 和 Receive 外，所有基本语句都认为没有临界引用。而 Send 和 Receive 语句，即使在带条件情况下，也只有单个临界引用。在此程序中，任何两个引用同一变量的并发进程，只有对此变量的读引用。

我们称一个语句 S 是一个 LCR 语句，如果它的所有子语句都满足 LCR 条件。我们称一个程序是 LCR 程序，如果它所有的语句都是 LCR 语句。

显然，由上述讨论，我们有下列结论：

命题 1 如果 P 是一个 LCR 程序，则 P 的交替计算与 P 的重迭执行产生相同的行为集。 □

由前述讨论，我们能够将一个非 LCR 程序转化为一个 LCR 程序。粗略地讲，可以通过将每一违反 LCR 条件的语句细化为一个更小的 LCR 语句序列，必要时，可以引入辅助变量。下面我们给出转化算法。这里只考虑我们的程序设计语言所允许的语句形式，对于其它语句形式，不难由此扩展而得到。

转化算法：

步骤 1：首先移去除赋值语句之外，所有其它语句中的多个临界引用。引入一个新的局部布尔值变量 t，对语句结构实施转化：

- 对语句 if c then S_1 else S_2，其中 c 不满足 LCR 条件，代换为语句[t:=c;if t then S_1 else S_2]。
- 对语句 while c do S，其中 c 不满足 LCR 条件，代换为语句[t:=c;while t do[S;t:=c]]。
- 对语句 when c do S，它不是选择语句的子语句且 c 不满足 LCR 条件，代换为语句[t:=F;while ~t do[t:=c];S]。
- 对语句

$$\begin{bmatrix}\text{when} & c_1 & \text{do} & S_1\\ & \text{or} & & \\ & \vdots & & \\ & \text{or} & & \\ \text{when} & c_k & \text{do} & S_k\end{bmatrix}$$

其中 $c_1,\cdots,c_k$ 一起包含多于一个临界引用，代换为

语句：

$$\left[\begin{array}{l} t_1 := F, \cdots, t_k := F \\ \text{while} \sim(t_1 \vee \cdots \vee t_k) \text{do} \\ (t_1, \cdots t_k) := (c_1, \cdots, c_k) \\ \left[\begin{array}{lll} \text{when} & t_1 & \text{do } S_1 \\ & \text{or} & \\ & \vdots & \\ & \text{or} & \\ \text{when} & t_k & \text{do } S_k \end{array}\right] \end{array}\right]$$

其中，$t_1, \cdots, t_k$ 为新的局部布尔值变量。

步骤 2：细化赋值语句。令 $(u_1, \cdots, u_k) := (e_1, \cdots, e_k)$ $(k \geq 1)$ 是一多重赋值语句，包含多个临界引用。令 $\bar{v} = v_1, \cdots, v_m$ 是赋值语句中有临界读引用的所有变元表。我们代换上述语句为：

$$\left[\begin{array}{c} t_1 := v_1, \cdots, t_m := v_m \\ u_1 := e_1[t/\bar{v}], \cdots, u_k := e_k[t/\bar{v}] \end{array}\right]$$

表达式 $e[t/\bar{v}]$ 表示 e 中 v_i 的每一出现均由 t_i 代替 $(i = 1, \cdots, m)$。每一 t_i 是与 v_i 同类型的新的局部变量。

步骤 3：对步骤 2 得到的代换语句，我们引入信号量语句，代换为：

$$\left[\begin{array}{c} \text{request}(r) \\ t_1 := v_1, \cdots, t_m := v_m \\ u_1 := e_1[t/\bar{v}], \cdots, u_k := e_k[t/\bar{v}] \\ \text{release}(r) \end{array}\right]$$

其中 r 为信号量变量。注意，步骤 3 主要目的是确保赋值语句细化的原子性，即保证整个细化语句序列的执行不受干扰，步骤 3 是保证原始程序 P 与经过转化算法细化后的 LCR 程序 P′ 等价的关键步。

由转化算法过程，我们可得下述结论：

命题 2 对每一程序 P，存在一个 LCR 程序 P′，它与 P 等价。 □

由此我们可知，公平转换系统模型在 LCR 条件下是与实际系统的重迭执行模型一致的，也即此模型在 LCR 条件下是可信的。这对公平转换系统模型的应用提供了一个理论基础。

参 考 文 献

[1]K. M. Chandy and J. Misra, Parallel Program Design: a Fundation, Reading, MA: Addision-Wesley, 1988

[2]L. Lamport, Specifying Concurrent Program Modules, ACM Trans. Prog. Lang. Sys., 5, 1983

[3]L. Lamport, What Good is Temporal Logic?, Information Processing 83, R. E. A. Mason(ed), North-Holland, 1983

[4]Z. Manna and A. Pnueli, How to Cook a Temporal Proof System for Your Pet Language, Proc. 10th ACM Symp. Princ. of Prog. Lang., 1983

[5]Z. Manna and A. Pnueli, On the Faithfulness of Formal Models, LNCS-520, MFCS, 1991

[6]Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent System: Specification, Spinger-Verlag, NY, 1991

[7]S. Owicki and L. Lamport, Proving Liveness Properties of Concurmet Programs, ACM Trans. Prog. Lang. Sys., 4(3)1982

[8]A. K. Singh, Program Refinement in Fair Transition Systems, Acta Informatica, 30, 1993

（上接第 86 页）

(2)现有交换式局域网的不足：a)现有网络协议(如以太网)软件开销及时延不小。b)缺乏交换式局域网管理软件，对于流量分析及控制端口到端口连接虚拟网管理(虚拟网是指不同网段上几个节点组成一个工作组)等的管理软件都在开发中。c)交换是网路的核心，而控制又是网路核心的核心，ATM 在交换结构及控制方式上还需做很多的工作，可能要花几年甚至更长的时间才能有完善的 ATM 地区网、世界网。

(3)交换式局域网的发展趋势 a)交换式以太网将在近几年内迅速发展和应用，b)将在几年内发展低速 ATM 交换机和建设 ATM 局域网。同时，深入研究 ATM 交换结构，从而为建设完善的 ATM 广域网和世界性的 ATM 网作准备。c)新的交换式局域网结构及技术将得到广泛研究，这些研究包括：并行数据线(即由串行数据传输发展到位数据位校验位并行传输)；用网络硬件来完成网络协议，发送信息、建立链路、实现交换及接收信息，这些工作都与节点计算机的操作并行进行；0-Copy TCP/IP，它能充分利用内存作为网络信息的缓冲区，从而将信息在产生、发送、接收和使用各步骤中的转换次数做到最少，以此减少网络协议的软件开销。这些技术可能在 SAN(System Area Network，系统区域网)中首先得到应用，从而建立 Gbps 级吞吐量和 us 级软件开销的小型网，并逐步发展到局域网。