第6章　模型检验

对数字电路与系统，无论是硬件、软件或它们的组合，如何检查设计的正确性，即设计验证，已成为一个关键的问题之一。有关设计验证的方法与技术，近年来在工业界有持续增长的需求[28~30]。这一章讨论电路系统验证的模型检验方法。

6.1　验证的建模

一般地，形式验证技术由如下的3个部分组成，或者说需要涉及如下3个方面：

(1)模型系统的一个架构，即对系统的描述是对系统的一些组成部分的结构与关系的描述。

(2)一种特定的语言。它用于描述需要验证的特性。

(3)一种验证方法。它用于确定一个系统的描述是否满足给定的规范。

设计验证常用的两种方法是：基于证明的方法和基于模型的方法[31－33]。

在基于证明的方法中，系统描述是一个由多个公式组成的集合 Γ，而规范是另外一个公式 Φ。这种验证方法试图去找到一种能说明由 Γ 可以推出 Φ 成立的证明。一般地，这需要使用相关的专门知识。

在基于模型的方法中，用一个有限模型 M 来表示一个特定的系统，而规范仍用一个公式 Φ 来表示。这种验证方法是检验模型 M 是否满足 Φ。这经常是自动进行的。下面用表达式 $M \models \Phi$ 说明：模型 M 满足公式 Φ，也可以表述为模型 M 使公式 Φ 成立(为真)。

本章讨论称为模型检验的验证方法。它是一种自动化的，基于模型的验证方法。可以用于具有并发性特征的系统。这种并发性如果使用一些常规的测试方法是难于发现的，因为它们倾向于是不能再生的。因此若有一种验证技术能帮助发现它们，这是非常有价值的。

模型检验基于时态逻辑。时态逻辑的思想是：在模型中，一个公式不是一直为真或为假，因为它是命题和谓词逻辑。在时态逻辑的模型中，包含了很多状态，一个公式在一些状态时为真，而在其他的状态时为假。这样，公式值的静态表示由动态表示所取代，公式值的真或假是随着系统从一个状态转变为另一个状态而改变的。

在模型检验中，模型 M 是迁移系统，而性质 Φ 是时态逻辑中的公式。这里，将一个模型检验算法称为一个模型检验器。为了验证一个系统是否满足某一性质，必须做如下3件事情：

(1) 使用模型检验器的描述语言来对系统进行模型化，进而建立一个模型 M。

(2) 使用模型检验器对性质 Φ 进行编码。

(3) 把 M 和 Φ 作为输入，运行模型检验器。

若 $M \models \Phi$，则模型检验器就输出“是”；否则输出“否”。在后一种情况下，大多数的模型检验器的也会进行对引起这种失效的系统行为的“追踪”。这种追踪的自动生成在系统的设计和调整中是一种重要的工具。

由于模型检验是一种基于模型的方法，后面我们将专注于对满足性的表示，即在一个模型和一个公式之间的满足关系 $M \models \Phi$。

模型检验是在时态逻辑中来讨论的。目前，人们已提出了多种时态逻辑并在许多领域中使用。可以对时态逻辑从特定的“时间”形式来分类。“线型时间”逻辑将时间考虑为时间实例的一个串；“分支时间”逻辑在给定的时间点上提供了很多可选择的将来时间，它在模型化非确定性系统中是很有用的。此外，对“时间”也可以考虑它为“连续”的还是“离散”的，这是人们通常的做法。例如，如果研究模拟电路系统，则可以使用“连续”；如果是对数字电路，则可以使用“离散”。

在本章中，所研究的逻辑在时间上是分支的和离散的，这样的逻辑具有模态逻辑的一些特征，称它为计算树逻辑(CTL)，是由 E. Clarke 和 E. A. Emerson 提出来的。这种逻辑在对硬件和通信协议进行验证等方面，已被证明是相当有效的，并且人们已开始在软件的验证中应用它。

对模型检验，也可以用如下方式来描述：计算 $M, s \models \Phi$ 是否成立。这里 M 是一种模型，是我们所关心的“系统”的一种表示；Φ 是一个逻辑公式，现在是 CTL 的一个公式；s 是这种模型的一种状态；$\models$ 是所满足的关系。当然，Φ 现在是 CTL 的一种，模型 M 是“系统”的一种表示。

这里，对模型 M 做一些说明。不应将模型 M 和实际的物理系统相混淆。模型是抽象化的，它略去了物理系统的很多实际特征，这些特征与对 Φ 的检查无关。这与在微积分或力学中的做法类似，可以允许我们把注意力集中在所关心的一些特定方面。

构建这种模型有广泛的应用，这允许我们开发一种统一的方法，实现对硬件、通信网络、软件等的验证。这种模型的基本组成元素是状态。状态它可以是一些变量的当前值(例如在 C 语言程序中的变量)，也可以是在一种通信网络中的物理器件的实际状态(例如网络中一些计算资源处于“忙”状态或处于空闲的“可使用”状态)。

通过状态这种方式可以表达出模型 M 的动态性。这种动态性行为可以通过状态的迁移来描述。例如，考虑如下的赋值语句：

$$x := x + 1$$

该语句可以用来表示一个状态 s 到另一个状态 s' 的一种状态迁移，这时状态 s' 和状态 s 具有相同的值。

多个状态之间的迁移，就组成一个状态迁移集合，它是 $S \times S$ 的一个子集。对该集合，最好是把它作为一个整体来考虑。可以把它作为状态集合 S 上的一种二元关系，记为“R”如下写法 $s \rightarrow s'$ 的含义是：在一次计算之后系统从状态 s 到达状态 s' 是可能的。

6.2 计算树逻辑的语法

计算树逻辑 CTL 是一种时态逻辑，它允许我们根据当前的情况而对将来的事件做推理。CTL 也是一种分支时间逻辑，这意味着它的时间模型是一种像树一样的结构，而且在

该结构中将来的事件是不能被确定的，即在这种结构中存在多条不同的路径，其中的任何一条路径都可能是将来的“实际”的路径。

这里路径的含义是：对一个系统，所关心的是它的多个状态，也可以是无限多个状态，按照状态之间的转变关系，可以把它们排列成一个状态序列 $s_0, s_1, s_2, \ldots$，称该序列是从 s_0 开始的一条路径。

这里，使用诸如 $p, q, r, \ldots$ 或 $p_1, p_2, \ldots$ 作为对原子公式或原子描述的表示。所谓原子公式或原子描述一般是指不再细分的公式或不再细分的描述。用这些原子公式我们可以实现对一个给定系统的一些具体描述，如：

“打印机忙”或“寄存器 R_3 的当前内容是整数值 8”。

一般地，对原子描述的选择，主要依赖于对系统的哪些特定的方面感兴趣。

下面给出 CTL 公式的定义。

一个 CTL 公式由两部分组成。一部分是路径量词 **A** 和 **E**。**A** 的含义是“对于所有的路径”或“沿着所有路径”；**E** 的含义是“对存在的一些路径”或“沿着存在的至少一条路径”。CTL 公式的另一部分是时态逻辑运算符 **G**、**F**、**U** 和 **X**。**G** 的含义是“总是”或“所有将来的状态”；**F** 的含义是“有时”或“一些将来状态”；**U** 的含义是“直到”；**X** 的含义是“下一状态”或“下一时刻”。

定义 6.1 CTL 公式满足如下的规定：

(1) 每个原子公式是 CTL 公式；

(2) 如果 f_1 和 f_2 是 CTL 公式，则 $\neg f_1$、$f_1 \wedge f_2$、$\mathbf{AX}f_1$、$\mathbf{EX}f_1$、$\mathbf{A}[f_1 \mathbf{U} f_2]$、$\mathbf{A}[f_1 \mathbf{U} f_2]$ 都是 CTL 公式。

其中，$\neg$ 和 $\wedge$ 的含义分别是“否定”和“合取”；$\vee$ 的含义是“析取”。

注意，在 CTL 公式中，运算符 **X**、**F**、**G** 和 **U** 的前面一般要有一个 **A** 或 **E**。因此称 **AX**、**EX**、**AF**、**EF**、**AG**、**EG**、**AU** 和 **EU** 为时态逻辑的连接词。

公式 $\mathbf{AX}f$ 成立，当且仅当在当前状态的所有的直接后继状态公式 f 成立。或者简述为：$\mathrm{AX}f$ 的含义是公式 f 在当前状态的所有的直接后继状态成立。类似地，有：

$\mathbf{EX}f$ 的含义是公式 f 在当前状态的一些后继状态成立。

$\mathbf{AF}f$ 的含义是在从当前状态出发的每一条计算路径中都存在一些状态使得公式成立。

$\mathbf{EF}f$ 的含义是在从当前状态出发的一些计算路径中都存在一些状态使得公式 f 成立。

$\mathbf{AG}f$ 的含义是对所有可能路径上的每一状态公式 f 都成立。

$\mathbf{EG}f$ 的含义是存在一些计算路径，在这些路径上的每一状态公式 f 都成立。

$\mathbf{E}[f \mathbf{U} g]$ 的含义是：存在一些路径使得 f 成立直到 g 成立，即在从当前状态出发的一些计算路径上，存在一个状态使得 g 成立，并且对这些路径上的所有以前状态 f 成立。

$\mathbf{A}[f \mathbf{U} g]$ 的含义是：对所有可能的路径 f 成立直到 g 成立，即在从当前状态出发的每一条计算路径上，都存在一个状态使得 g 成立，并且对这些路径上的所有以前状态 f 成立。

对其他的一些 CTL 公式，可以按下列规则得到：

$$f \vee g = \neg(\neg f \wedge \neg g)$$

$$\mathbf{AF}g = \mathbf{A}(\text{true}\mathbf{U}g)$$
$$\mathbf{EF}g = \mathbf{E}(\text{true}\mathbf{U}g)$$
$$\mathbf{AG}f = \neg\mathbf{E}(\text{true}\mathbf{U}\neg f)$$
$$\mathbf{EG}f = \neg\mathbf{A}(\text{true}\mathbf{U}\neg f)$$

此外，对一个给定的 CTL 公式，我们根据它所包含的连接词的情况，可以形象地用一个图来说明该 CTL 公式的结构。这种图称为语义树。例如，公式 **A**[**AX**¬ p**UE**[**EX**($p \wedge q$)**U**¬ p]] 的语义树如图 6.1 所示。该语义树的根节点被标以连接词 **AU**。

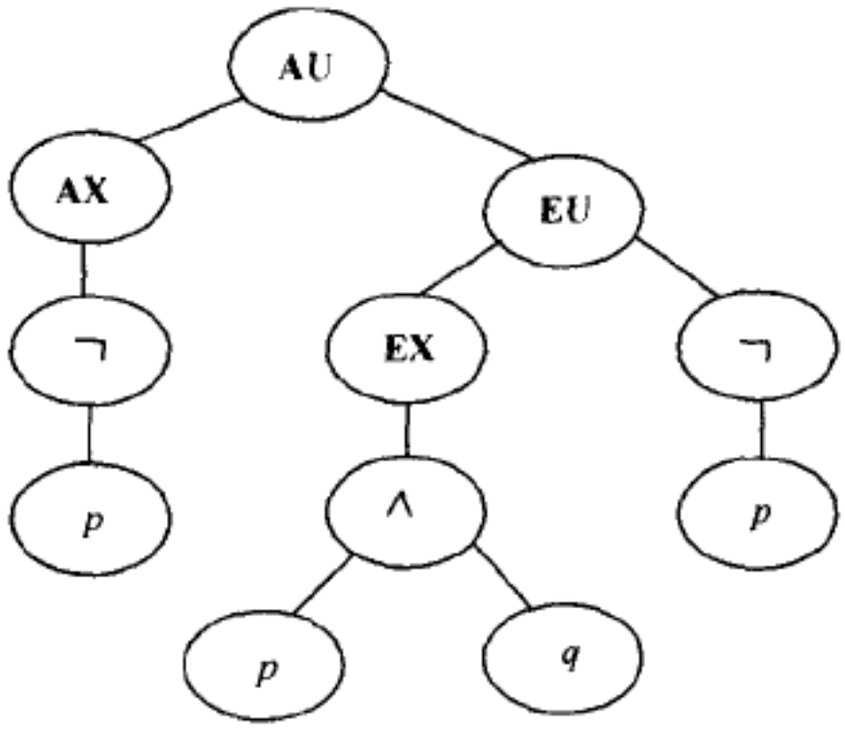


图 6.1　公式 **A**[**AX**¬ p**UE**[**EX**($p \wedge q$)]**U**¬ p] 的语义树

6.3　计算树逻辑的语义

定义 6.2　一个 CTL 模型 $M=(S,R,L)$ 的定义是：M 是状态 S 的一个集合；它具有一种迁移关系 R，该关系是 S 上的二元关系，$R \subseteq S \times S$；并且存在一个标记函数 L 为

$$L: S \rightarrow\!\!\!\!\rightarrow \psi$$

其中，符号 $\rightarrow\!\!\!\!\rightarrow$ 表示通常的函数映射关系。ψ 是系统 M 的所有原子描述的一个幂集。例如，集合 $\{p,q\}$ 的幂集是 $\{\varnothing,\{p\},\{q\},\{p,q\}\}$。

以上这种对模型 M 的定义看起来相当数学化。可以简单地理解为：

(1) 存在状态 S 的一个集合；

(2) 存在一个关系 R，它说明系统怎样从一个状态向另一个状态变化；

(3) 与每一个状态 s 相关，存在一个原子命题的集合 $L(s)$，该集合中的原子命题在特定的状态为真。

可以按如下的方式来理解函数 L：它仅仅是对所有原子命题分配一种值（为真或假）。对系统的一个状态 s，把与该状态有关的一些原子命题所组成的集合命名为 $L(s)$，使得在 $L(s)$ 中的原子命题在状态 s 时为真。

也可以用下面的五元组结构来说明 CTL 模型。定义 $K=<S,S_0,R,\text{AP},L>$，其中：

S 是状态的集合；

$S_0 \subseteq S$ 是初始状态的集合；

R 是状态之间的迁移关系；

AP 是所有原子命题和它们的否定命题的集合；

L 是从 S 到 AP 的幂集的一个标记函数，该函数把 S 中的状态 s 映射为在 AP 中的为真的原子命题集合。

这种五元组结构也称为 Kripke 结构。

也可以用关系 R 来对“路径”做定义。如果对于任意的 $i \in \{0,1,2,\cdots\}$，在状态 s_i 和 s_{i+1} 之间都存在迁移关系 $s_i \rightarrow s_{i+1}$，即 $(s_i, s_{i+1}) \subseteq R$，则称一个无限状态序列 $\pi = \{s_0, s_1, s_2, \cdots\}$ 是从 s_0 开始的一条路径。

对 CTL 公式，可以按多种方式来分类，例如有典型的“状态公式”和“路径公式”两种类型。一个 CTL 公式如果在一个特定的状态下为真，则称它为状态公式；如果对一条特定路径上的所有状态都为真，简述为对一条路径为真，则称它为路径公式。显然，有如下的一些结论：

若 $p \in$ AP，则 p 是一个状态公式；

若 f 和 g 是状态公式，则 $\neg f, f \wedge g, f \vee g$ 是状态公式；

若 f 是一个路径公式，则 $\mathbf{E}f$ 和 $\mathbf{A}f$ 是状态公式；

若 f 和 g 是路径公式，则 $\neg f, f \wedge g, f \vee g, f\mathbf{U}g$ 是状态公式。

对一个CTL模型及其给定的初始状态 $s_0 \in S$，可以通过如下的“计算树”来说明状态之间的迁移关系。若状态 s 和 t 之间有迁移 $s \rightarrow t$，则在它们之间画一条弧。图 6.2 说明了一个 CTL 模型和它的从状态 s_0 开始的计算树。

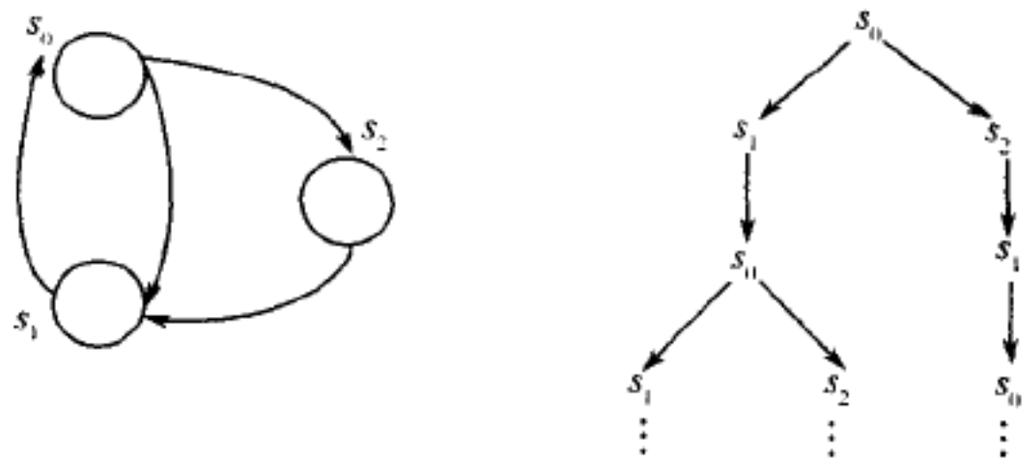


图 6.2　一种 CTL 模型的计算树

也可以用有向图方便地表示一个CTL模型M的所有信息。在这种图中，一个节点（它包含一个状态）包含了在该状态下为真的所有原子命题。例如，对一个仅仅有 3 个状态 s_0、s_1 和 s_2 的系统，若状态之间所有可能的迁移为 $s_0 \rightarrow s_1$、$s_0 \rightarrow s_2$、$s_1 \rightarrow s_0$、$s_1 \rightarrow s_2$ 和 $s_2 \rightarrow s_2$；并且 $L(s_0) = \{p,q\}$、$L(s_1) = \{q,r\}$ 和 $L(s_2) = \{r\}$。这时，可以用图 6.3 来表达该模型的所有信息。

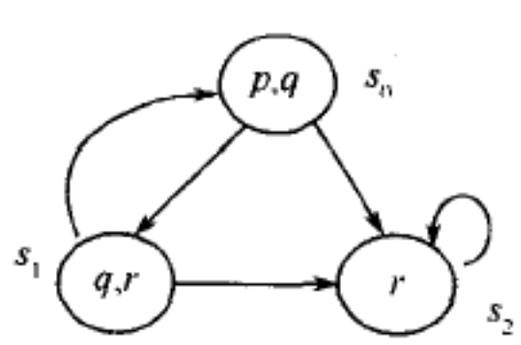


图 6.3　一种 CTL 模型的有向图表示

在如上的**定义 6.2** 中对每一个状态 $s \in S$，要求至少存在一个 $s' \in S$，并且 $s \rightarrow s'$。这个要求的目的是使得系统的状态不会发生“死锁”。若不满足此要求时，可以添加一个额外的状态 S_d 来表示死锁，并增加迁移 $s_d \rightarrow s_d$；也为发生死锁的状态 s 增加新的迁移 $s \rightarrow s_d$，

图 6.4 是这种情况的一个例子。图 6.4 中左边的图在状态 s_4 时没有任何的状态迁移，在右边的图中添加了状态 s_d。

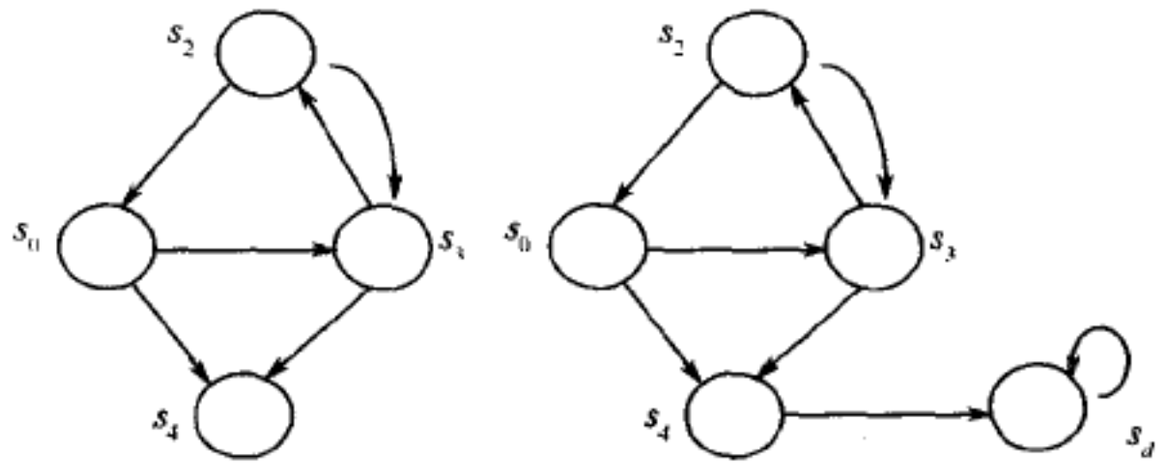


图 6.4　对具有死锁的模型的处理

用如下较标准的形式来说明 CTL 公式的值。

定义 6.3　设 $M=(S,R,L)$ 是一个 CTL 模型。对 S 中一个给定的状态 s，一个 CTL 公式 Φ 在状态 s 是否成立，用满足关系来表达：

$$M,s\models\Phi$$

在不产生误解的情况下，可以省略 M，即把“$M,s\models\Phi$”简写为“$s\models\Phi$”。

使用满足关系 $\models$，可以递归地定义一些常用的 CTL 公式，例如：

$s\models p$，当且仅当 $p\in L(s)$，这里 p 是原子命题。

$s\models\neg f$ 成立，当且仅当 $s\models f$ 不成立。

$s\models f\wedge g$ 成立，当且仅当 $s\models f$ 和 $s\models g$ 都成立。

$s\models f\vee g$ 成立，当且仅当 $s\models f$ 或 $s\models g$ 成立。

$s\models\mathbf{AX}f$ 成立，当且仅当对所有有迁移关系 $s\to t$ 的状态 t，有 $t\models f$ 成立；简称为“在 s 的每一个直接后继状态 f 成立”。

$s\models\mathbf{EX}f$ 成立，当且仅当对一些有迁移关系 $s\to t$ 的状态 t，有 $t\models f$ 成立。简称为“在 s 的一些直接后继状态 f 成立”。

$s\models\mathbf{AG}f$ 成立，当且仅当对所有的路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$ 和这种路径上的所有状态 s_i，使得 $s_i\models f$ 成立，$i=1,2,3,\cdots$。简称为“在所有路径上 f 成立”。

$s\models\mathbf{EG}f$ 成立，当且仅当存在一些路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$ 和这种路径上的所有状态 s_i，使得 $s_i\models f$ 成立，$i=1,2,3,\cdots$。简称为“存在一些路径使得 f 成立”。

$s\models\mathbf{AF}f$ 成立，当且仅当对所有的路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$ 和在这种路径上存在一些状态 s_i，使得 $s_i\models f$ 成立。简称为“在所有路径上存在一些状态使 f 成立”。

$s\models\mathbf{EF}f$ 成立，当且仅当存在一些路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$ 和在这种路径上存在一些状态 s_i，使得 $s_i\models f$ 成立。简称为“存在一些路径，在该路径上存在一些状态使 f 成立”。

$s\models\mathbf{A}[f\mathbf{U}g]$ 成立，当且仅当对所有的路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$，使得 $f\mathbf{U}g$ 成立。即在每一条这种路径上存在一些状态 s_i，使得 $s_i\models g$ 成立，并且对每一个 $j<i$，有 $s_j\models f$ 成立。简称为“所有路径对 f 成立直到 g 成立”。

$s\models\mathbf{E}[f\mathbf{U}g]$ 成立，当且仅当存在一些路径 $s_1\to s_2\to s_3\to\cdots$，这里 $s_1=s$，使得 $f\mathbf{U}g$ 成立。即在这种路径上存在一些状态 s_i，使得 $s_i\models g$ 成立，并且对每一个 $j<i$，有 $s_j\models f$ 成立。简称为“存在一些路径，在该路径上存在一些状态使得 f 成立直到 g 成立”。

下面用两个例子来对如上的 CTL 公式进行说明。

【例 6.1】　图 6.5 说明了 **EF**f、**EG**f、**AG**f 和 **AF**f 的特征。

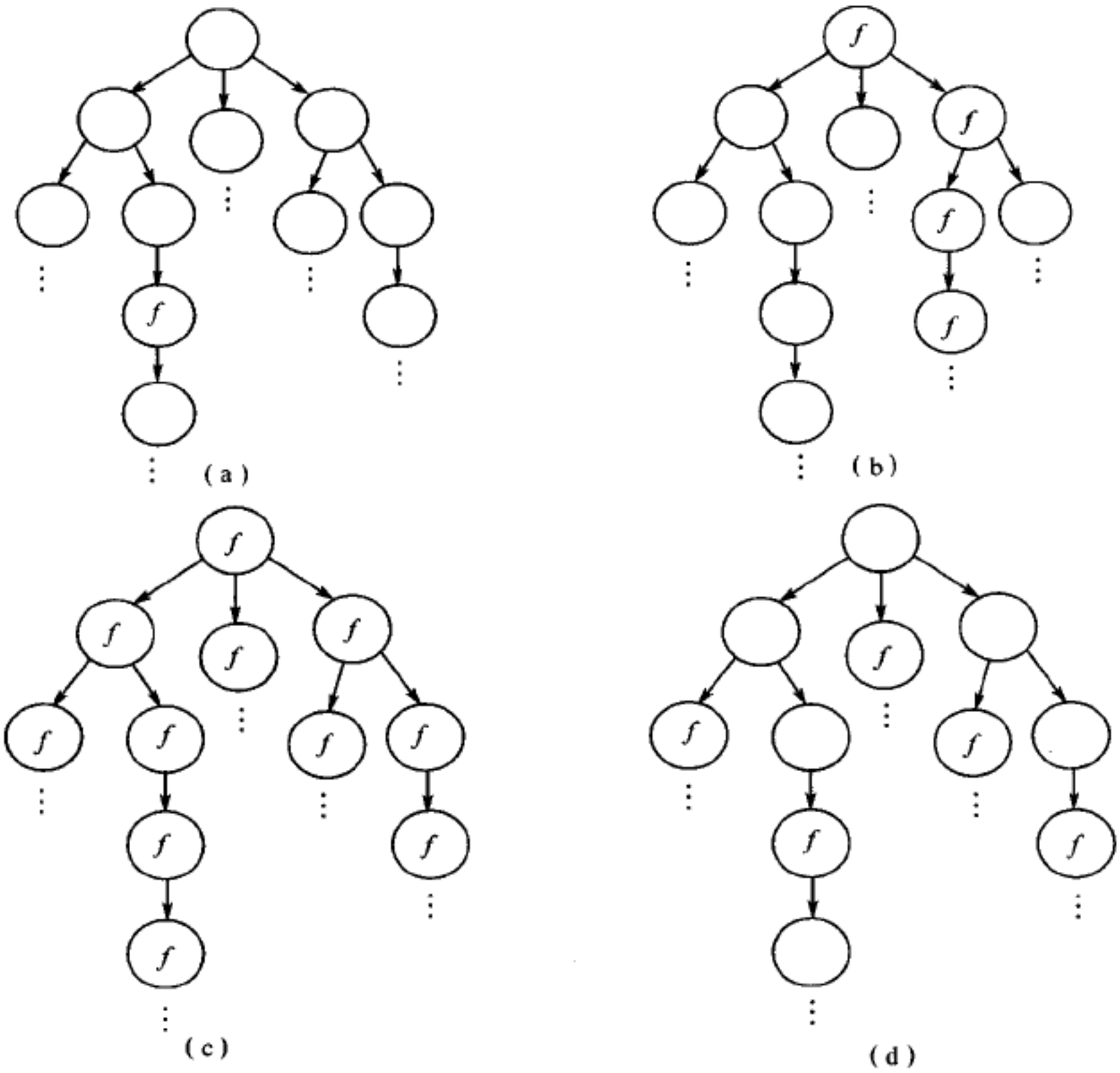


图 6.5　开始状态满足 **EF**f、**EG**f、**AG**f 和 **AF**f 的例子

(a) 开始状态满足 **EF**f；(b) 开始状态满足 **EG**f；(c) 开始状态满足 **AG**f；(d) 开始状态满足 **AF**f。

【例 6.2】　对图 6.3 的系统，从状态 s_0 开始的计算树如图 6.6 所示。由图 6.6 可知，对图 6.3 的系统，有如下的结论：

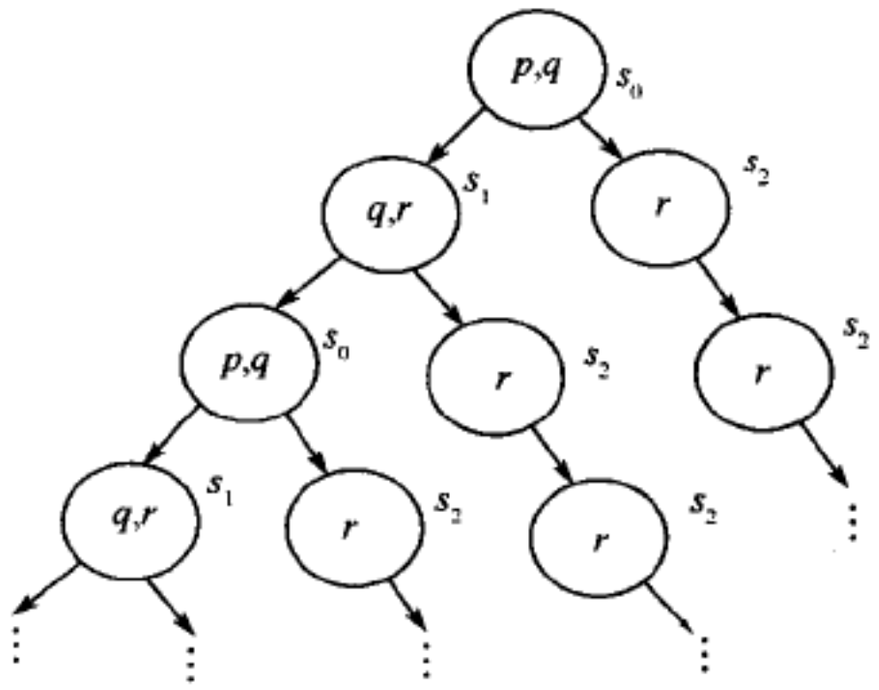


图 6.6　CTL 模型图 6.3 的计算树

(1)$s_0 \models p \wedge q$ 成立，因为在 s_0 的节点中包含了原子命题 p 和 q。

(2)$s_0 \models$ **EX**$(q \wedge r)$ 成立，因为存在路径 $s_0 \to s_1 \to s_0 \to s_1 \to \cdots$，第二个节点 s_1 包含 q 和 r。

(3)$s_0 \models \neg$ **AX**$(q \wedge r)$ 成立，因为有路径 $s_0 \to s_2 \to s_2 \to \cdots$，第二个节点 s_2 只包含 r，

但不包含 q。

(4)$s_0 \models \neg \mathbf{EF}(p \wedge r)$ 成立,因为不存在以 s_0 开始的路径,使得可以达到一个使 $p \wedge r$ 成立的状态。

(5)$s_2 \models \mathbf{EG}r$ 成立,因为存在一条以 s_2 开始的路径 $s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \cdots$,使得 r 在所有将来状态都成立。类似地,$s_2 \models \mathbf{AG}r$ 也成立。

(6)$s_0 \models \mathbf{AF}r$ 成立,因为以 s_0 开始的所有路径,系统可以达到一个状态(s_1 或 s_2)使得 r 成立。

(7)$s_0 \models \mathbf{E}[(p \wedge q)\mathbf{U}r]$ 成立,因为存在一条以 s_0 开始的路径 $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \cdots$,第二个节点 $s_2(i=1)$ 满足 r,而所有以前的节点($i=0$,即节点 s_0)满足 $p \wedge q$。

(8)$s_0 \models \mathbf{A}[p\mathbf{U}r]$ 成立,因为在 s_0 时 p 成立,且对始的路径 $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow s_2 \rightarrow \cdots$,第二个节点 $s_2(i=1)$ 满足 r,而所有以前的节点($i=0$,即节点 s_0)满足 $p \wedge q$。

【例 6.3】 用 CTL 公式可以检查系统的一些基本特征。下面列出几个常用的例子。假定原子描述包括“开始”(start)、“忙”(busy)、“请求”(request)、“准备就绪”(ready)、“认可”(ack) 等一些事件。可以用如下公式实现相关的功能。

(1)用 **EF**(start∧¬ ready)可以获得一个“开始”成立但“准备就绪”不成立的状态:

(2)用 **AG**(request→**AF** ack)可以表示对所有的计算路径,如果在当前状态发出 request 请求,则系统在将来的某个时刻,总能到达使 ack 满足的某个状态。这里 request→AF ack 说明若在初始状态有个请求 request,则在将来的某个时刻,系统能够进入 ack 为真的状态。

(3)用 **AG**(**AF** ack)可以表示对每一条路径 ack 可以被无限次满足,或者表示从系统的任何一个状态可以无限多次进入 ack 为真的状态。

(4)用 **AF**(**AG** ack)可以表示从任何一个状态出发,系统最终都将停留在 ack 为真的状态。

(5)用 **AG**(**EF** start)可以表示从任何状态出发,都可以获得一个开始(start)状态。

【例 6.4】 在例 6.3 的基础上,可以用 CTL 公式来描述一个实际的例子:电梯的运行规则。楼层、运行方向、乘客所按下的电梯中的楼层选择按钮等分别用 floor,direction,buttonPressed 来表示;电梯向上运行其方向的值为 up,即 direction=up。

(1)如下公式可以说明一部向上运行的电梯如果有乘客希望去六楼时,该电梯运行到三楼时不能改变它的运行方向:

AG(floor=3∧direction=up∧buttonPressed=6→**A**[direction=up**U** floor=6])

(2)对电梯若没有按下任何楼层选择按钮,这时称它处于空闲状态(idle)。处于空闲状态的电梯若门(door)是关闭的(closed),则它可以停留在任何一个楼层。如下的公式说明处于空闲状态的电梯可以停留在第四层。

AG(floor=4∧idle∧door=closed→**EG**(floor=4∧idle∧door=closed))

6.4 CTL 公式间的等价性

定义 6.4 对两个 CTL 公式 ϕ 和 ϕ,如果在任何模型中的任何状态它们都互相满足,则称它们在语义上是等价的,记为 $\phi \equiv \varphi$。

在CTL公式中，有一些公式不可能为真，例如，$\varphi \wedge \neg \varphi$，因此，用$\perp$来代表它们；而有一些公式就一定为真，例如，$\varphi \vee \neg \varphi$，因此用$\top$来代表它们。

A是对“所有”路径的修饰词，**E**是对“存在”路径的修饰词。而且，**G**和**F**也是针对“所有”和“存在”判断的修饰词，与**A**和**E**不同的是，它们只是沿着特定的路径来说明状态的变化范围。根据这一点，不难发现笛摩根定理对**A**和**E**、**F**和**G**仍然成立：

$$\neg \mathbf{AF}\phi \equiv \mathbf{EG} \neg \phi$$

$$\neg \mathbf{EF}\phi \equiv \mathbf{AG} \neg \phi$$

在任何特定的路径上，每一个状态都有一个唯一的后继。因此，**X**是它自己在计算路径上的对偶，有

$$\neg \mathbf{AX}\phi \equiv \mathbf{EX} \neg \phi$$

也有如下等式：

$$\mathbf{AF}\phi \equiv \mathbf{A}[\top \mathbf{U}\phi] \qquad \mathbf{EF}\phi \equiv \mathbf{E}[\top \mathbf{U}\phi]$$

可以根据**U**的含义来检验如上等式的正确性。对一条路径的一个给定状态，如果$\phi_1 \mathbf{U} \phi_2$为真，则当ϕ_2在将来的一些状态为真时，ϕ_1在直到这些状态的每一个状态都为真。在如上的等式中，让ϕ_1为$\top$。因为$\top$在每一个状态都为真，因此ϕ_2在将来的某个状态为真时，ϕ_1在直到这个状态的每一个状态都为真。

自然地，任何在命题逻辑中成立的等价性在CTL中也成立，即使这种等价性涉及CTL公式。例如，由于命题逻辑公式$\phi \vee \neg \phi$等于$\top$，因此有$\top$和$(\mathbf{AX}p) \vee (\neg \mathbf{AX}p)$是等价的。

在CTL中一些其他的等价性如下：

$$\mathbf{AG}\phi \equiv \phi \wedge \mathbf{AX}\ \mathbf{AG}\phi$$

$$\mathbf{EG}\phi \equiv \phi \wedge \mathbf{EX}\ \mathbf{EG}\phi$$

$$\mathbf{AF}\phi \equiv \phi \vee \mathbf{AX}\ \mathbf{AF}\phi$$

$$\mathbf{EF}\phi \equiv \phi \vee \mathbf{EX}\ \mathbf{EF}\phi$$

$$\mathbf{A}[\phi \mathbf{U} \varphi] \equiv \varphi \vee (\phi \wedge \mathbf{AX}\ \mathbf{A}[\phi \mathbf{U} \varphi])$$

$$\mathbf{E}[\phi \mathbf{U} \varphi] \equiv \varphi \vee (\phi \wedge \mathbf{EX}\ \mathbf{E}[\phi \mathbf{U} \varphi])$$

例如，对如上的第三个等式做如下分析：为了使$\mathbf{AF}\phi$在一个特定状态成立，ϕ在沿着从该状态出发的每一条路径的一些状态上一定为真。为了得到这种特性，或者在当前状态使ϕ为真；或者推迟它，使得在每一个下一状态必须有$\mathbf{AF}\phi$成立。

注意，第三个等式中的等价性是按照**AX**和**AF**自身来定义**AF**，这是一种明显地循环定义。事实上，对上述的6种等价性，可以按照**AX**和**EX**，以一种非循环的方式来定义每种等价性中的左边。这被称为CTL的固定特征，它是在后面所讨论的模型检验算法的数学基础，在后面会再讨论它。

6.5 CTL验证的例子——进程互斥

在下面，讨论使用CTL来验证一个规模较大的例子：进程互斥。

在计算机系统中，将可供进程使用的一些数据、软件和硬件等统称为资源，例如在磁盘上的一个文件或在一个数据库中的一条记录。当多个并发进程共享一个资源时，必须保证这些并发进程不在同一时刻对共享资源进行存取，比如不希望多个进程同时对同一文

件进行编辑。

为此，把每一个进程的程序代码中的一部分作为它的“临界区”。所谓“临界区”它是在进程的程序代码中与存取共享资源有关的那些程序代码。在管理这些临界区时，使得在某一时刻只有唯一一个进程处于它的临界区执行。

现在面临的问题是找到一种协议以决定在给定的时刻哪个进程可以进入它的临界区。一旦找到了一种可以工作的协议，就可以通过一些性质来检查它，使得进程可在进入它的临界区时满足如下的 4 点：安全性、许可性、非阻塞、非排序。

安全性：协议在任何时刻仅仅允许一个进程处于它自己的临界区。这种安全性是不充分的，因为永久性地排除每一个进程进入它的临界区将是安全的。因此，也要求协议具有如下的性质。

许可性：任何进程无论何时想进入它自己的临界区，它最终都将被许可。

非阻塞：每一个进程可以一直要求进入它的临界区。

一种简单的协议是采用使进程轮流循环地进入临界区。而在一般情况下，一些进程比另一些进程要更多地存取共享资源，即不同的进程进入临界区的次数有多有少，是不相等的。因此应保证协议具有如下性质。

非排序：进程不按事先所规定的次序进入临界区。

下面讨论两个进程的情况。假定进程被命名为进程 1 和进程 2。用 n 来表示一个进程“处于它的非临界区”这种状态；用 t 表示一个进程“试图进入它的临界区”这种状态；用 c 表示一个进程“处于它的临界区”这种状态。例如 n_1 和 t_2 分别表示进程 1 处于非临界区和进程 2 试图进入临界区。

每一个进程按循环 $n \to t \to c \to n \to t \to \cdots$ 经历状态的迁移。而在两个进程时，则是和另一进程交替进行，例如 $n_1 \to t_2 \to c_2 \to t_1 \to \cdots$。

考虑图 6.7 中由迁移系统 M 所给出的协议。在图中圆圈里的表达式的含义是：它的每个原子命题在该状态都为真。例如，$t_1 n_2$ 表示在状态 s_1 时 t_1 和 n_2 都为真，即进程 1 处于试图进入它的临界区这种状态，进程 2 处于它的非临界区这种状态。

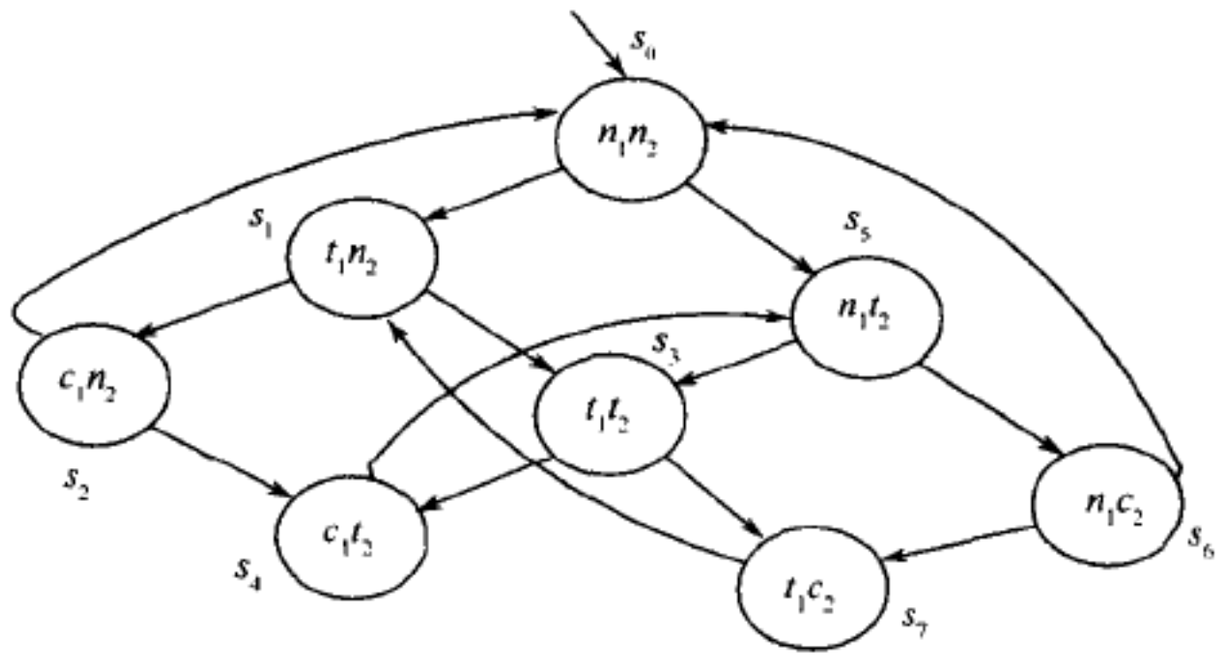


图 6.7　进程互斥的第一种模型

两个进程都开始于它们的非临界区状态（全局状态 s_0）。状态 s_0 是唯一的初始状态，用一条没有源的边来指向它。两个进程中的任何一个现在都可能向“试图进入它的临界区”这种状态移动，但只有唯一的一个进程在一给定时刻能发生状态迁移，因此从 s_0 到 s_1 和 s_5

都存在一个迁移(箭头)。在 s_1 时,进程 1 试图进入临界区,而进程 2 处于非临界区。从 s_1 出发,这时会再一次发生两件事:或者进程 1 移动到 s_2,或者进程 2 移动到 s_3。

注意每一个进程不是在每一个状态都能移动。例如,进程 1 在状态 s_7 就不能移动,它不能进入它的临界区直到进程 2 已退出了临界区。

对安全性、许可性、非阻塞、非排序这 4 个性质,首先用 CTL 公式来对它们进行描述,然后进行验证。

安全性:定义 ϕ_1 为 $\phi_1 = \mathbf{AG}\neg(c_1 \wedge c_2)$。显然,$\mathbf{AG}\neg(c_1 \wedge c_2)$ 在初始状态是被满足的。事实上,在每一状态都满足。

非阻塞:定义 ϕ_3 为 $\phi_3 = \mathbf{AG}(n_1 \rightarrow \mathbf{EX}t_1)$。显然,$\phi_3$ 是被满足的。因为每一个 n_1 状态(即 s_0、s_5 和 s_6)都有一个直接的 t_1 后继状态。

非排序:定义 ϕ_4 为 $\phi_4 = \mathbf{EF}(c_1 \wedge \mathbf{E}[c_1\mathbf{U}(\neg c_1 \wedge \mathbf{E}[\neg c_2\mathbf{U}c_1])])$。这是被满足的。例如,使用路径 $s_5 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow \cdots$。

许可性:定义 ϕ_2 为 $\phi_2 = \mathbf{AG}(t_1 \rightarrow \mathbf{AF}c_1)$。在初始状态 ϕ_2 是不满足的。这是因为,可以找到一个经初始状态迁移之后的状态,例如 s_1,在 s_1 时 t_1 为真,但 $\mathbf{AF}c_1$ 为假。$\mathbf{AF}c_1$ 为假是由于存在一条路径 $s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow s_1 \rightarrow \cdots$,在此路径上 c_1 一直为假。

这里,“许可性”不满足的原因是在于存在一些非确定性。这意味着所给出的协议可能偏爱某一个进程。在图 6.7 的协议中,这表现为状态 s_3 不能区分哪一个进程首先进入“它的试图进入临界区”状态。下面把 s_3 划分成两个状态,就可解决此问题。

在图 6.8 中,状态 s_3 和 s_8 的圆圈里的表达式都是 t_1t_2,这两个状态与图 6.7 中的状态 s_3 相对应。这两个状态记录了“处于试图进入临界区”状态的两个进程:s_3 记录了进程 1,而 s_8 则记录了进程 2。

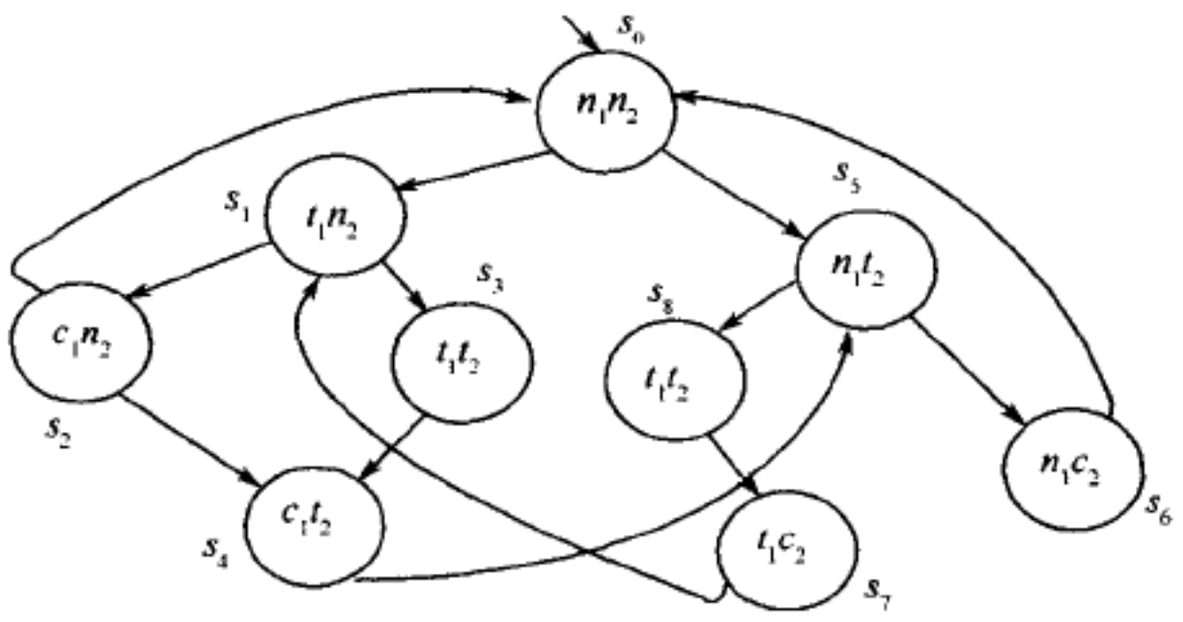


图 6.8　进程互斥的第二种模型

仍使用如上定义的 CTL 公式 $\phi_1 \sim \phi_4$,可以得到如下的结论:在图 6.8 中,安全性、许可性、非阻塞、非排序这 4 个性质都满足。

在图 6.8 所给出的协议中,迁移系统仍然是较为简单的,主要的原因是:这里假定在时钟的每一次跳动时进程将移向下一个不同的状态;对同一状态不存在迁移。有时,我们希望能模型化一个进程,它处于临界状态的时间是连续多次的时钟跳动。一种实现方法是用指向自身的箭头。例如,在图 6.8 中从 s_5 到它自身也存在一个箭头。如果用这种方法,则将再次使“许可性”得不到满足。这个问题将来在本章后面考虑“公正性约束”时来解决。

6.6 模型检验算法

一般地，人们所感兴趣的迁移系统往往会有上万个状态，在检查时所关心的公式可能是相当长的。因此，找到一种有效的模型检验算法是非常有价值的。

对模型 M 的表示法中已包括了有向图，也包括有向图所展开成的树和一个给定的初始状态。对树而言，由于所有可能的路径是明显可见的，因此对它的检查容易进行。然而，如果打算在计算机上完成一个模型检验方法，则事实上不可能将迁移系统都展开成树，而需要对有限的数据结构做检查。这就是对 CTL 语义进行深入和详细讨论的原因，它将为设计有效的模型检验算法提供基础。该算法的目标是(称为第一个问题)：

给定 $M, s \in S$ 和 ϕ，计算：$M, s \models \phi$ 是否成立。

若不成立，则这种算法能够产生一条系统的实际运行路径，以说明 M 不能满足 ϕ。基于此，通过确定是什么原因造成系统运行时不满足性质 ϕ，我们可以对系统的结构做出修改或调整。

特别地，对如下问题(称为第二个问题)：$M, s_0 \models \phi$ 是否成立。可以把它作为一个计算问题来处理。

例如，假定我们已经有了模型 M、公式 ϕ，并且把 s_0 作为系统的一个输入。如果 $M, s_0 \models \phi$ 成立，则就期望有一个肯定的回答“是”；若不成立，就有否定的回答“否”。此外，系统的输入也可以正好是 M 和 ϕ，而输出则是模型 M 中满足 ϕ 的所有状态 s。

若解决了这两个问题中的第二个问题，则会自动地获得第一个问题的解，这是由于能够简单地检查 s_0 是否是输出集中的一个元素。反过来，为了求解第二个问题，对第一个问题的一种给定算法，将简单重复地对每一个状态调用这个算法，以决定该状态是否进入了输出集。在后面将给出一个算法以求解第二个问题。

6.6.1 标签算法

下面讨论具有如下功能的算法：给定了一个模型和一个 CTL 公式，求出满足该公式的状态集合。该算法不需要直接对每一个 CTL 连接词进行处理，它使用了一个递归程序称为 Trans。程序 Trans 的功能是：把任意的一个 CTL 公式 ϕ，转换成一个由对集合{⊥，¬，∧，**AF**，**EU**，**EX**} 中的元素进行操作的等价的另外一个 CTL 公式。

整个模型检验算法的描述如下。

输入：一个 CTL 模型 $M = (S, \rightarrow, L)$ 和一个 CTL 公式 ϕ。

输出：满足 ϕ 的 M 的状态集合。

算法的实现步骤是：首先，把 CTL 公式 ϕ 作为到 Trans 的输入，变换成另一种等价的形式；方法是使用在前面所给出的等价关系，按照连接词 **AF**、**EU**、**EX**、∧、¬和 ⊥ 来重写公式 ϕ。下一步是，用满足 ϕ 的一些子公式来标签 M 的状态；从最小的子公式开始，直到整个公式 ϕ。

假定 φ 是 ϕ 的一个子公式，并且满足 ϕ 的所有直接子公式的这些状态都已被标签。这里，ϕ 的一个直接子公式是指任何最大长度的子公式，且不是 ϕ 自身。通过对如下几种情况的分析，来决定对状态的标签方法。

(1) 若 φ 是 $\perp$，则没有状态用 $\perp$ 来标签。

(2) 若 φ 是 p，这里 $p \in L(s)$，则用 p 来标签 s。

(3) 若 φ 是 $\psi_1 \wedge \psi_2$，如果已经同时用 ψ_1 和 ψ_2 来标签了 s，则用 $\psi_1 \wedge \psi_2$ 来标签 s。

(4) 若 φ 是 $\neg\psi_1$，如果 s 没有用 ψ_1 来标签，则用 $\neg\psi_1$ 来标签 s。

(5) 若 φ 是 $\mathbf{AF}\psi_1$，则

① 若一个状态 s 是用 ψ_1 来标签的，则重新用 $\mathbf{AF}\psi_1$ 来标签它。

② 若一个状态的所有后继状态是用 $\mathbf{AF}\psi_1$ 来标签的，则重新用 $\mathbf{AF}\psi_1$ 来标签这个状态。

重复进行这种操作，直到不存在变化为止。这可以用图 6.9 来说明。在图 6.9(a) 中，状态 s 的 3 个后继状态是用 $\mathbf{AF}\psi_1$ 来标签的，因此对状态 s 用 $\mathbf{AF}\psi_1$ 来标签，如图 6.9(b) 所示。

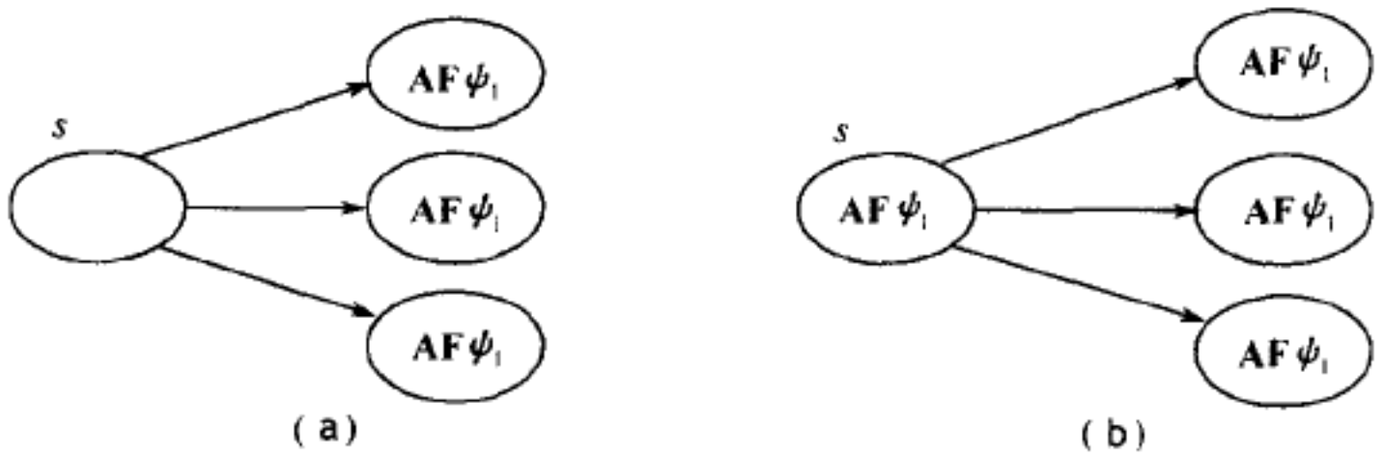


图 6.9　对一个状态用 $\mathbf{AF}\psi_1$ 来标签

(6) 若 φ 是 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$，则

① 若一个状态 s 是用 ψ_2 来标签，则重新用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签它。

② 若一个状态是用 ψ_1 来标签，并且它的所有后继状态是用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签的，则重新用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签这个状态。重复进行这种操作，直到不存在变化为止。这可以用图 6.10 来说明。在图 6.10(a) 中，状态 s 是用 ψ_1 来标签的，而它的一个后继状态是用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签的，因此对状态 s 用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签，如图 6.10(b) 所示。

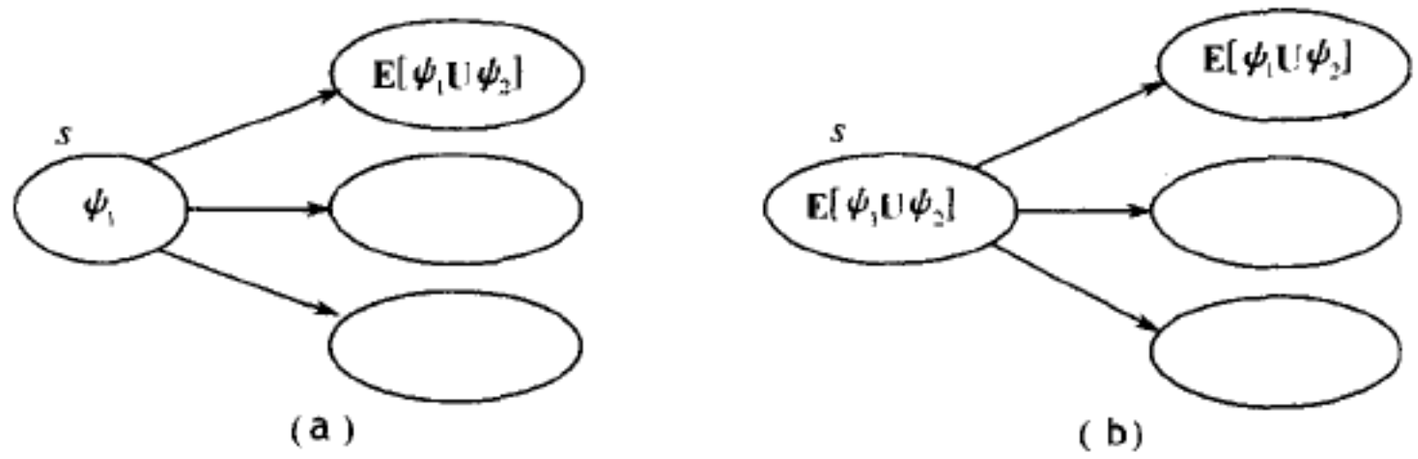


图 6.10　对一个状态用 $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ 来标签

(7) 若 φ 是 $\mathbf{EX}\psi_1$，则对一个状态，如果它的后继状态中有一个被标签为 ψ_1，则重新用 $\mathbf{EX}\psi_1$ 来标签该状态。

使用如上的步骤，如果对 ϕ 的所有子公式(包括 ϕ 自身) 都已经完成了标签，这时就获得了标签为 ϕ 的状态，就可以输出该状态。这种算法的复杂性是 $O(f \cdot V \cdot (V+E))$，其中 f 是公式中的连接词的个数，V 是状态的个数，E 是迁移的个数。该算法的复杂性与公式的规模成线性关系，与模型的规模成平方关系。

如上的程序 Trans 是把任意的一个 CTL 公式转换成等价的另外一个 CTL 公式，在此基础上来进行状态的标签。事实上，也可以直接对一些连接词进行处理。

下面介绍对 $\mathbf{EG}\psi_1$ 进行直接处理的算法，它分为如下 3 步：

(1) 用 $\mathbf{EG}\psi_1$ 标签所有的状态。

(2) 若一个状态 s 以前不是由 ψ_1 来标签，则删除该标签 $\mathbf{EG}\psi_1$；

(3) 若一个状态它的后继中没有一个被标签为 $\mathbf{EG}\psi_1$，则删除标签 $\mathbf{EG}\psi_1$。重复进行这种操作，直到不存在变化为止。

这里首先用子公式 $\mathbf{EG}\psi_1$ 标签所有的状态，然后从所得的这种标签集合中去掉一些元素。实际上，就关心的最后结果而言，**EG** 的这一过程和把它转换成 $\neg\mathbf{AF}\neg$ 之后再进行处理，不存在实质差别。

【例 6.5】 我们现在讨论如上标签算法的一个具体例子。考虑图 6.8 的进程互斥的第二种模型和 CTL 公式 $\varphi = \mathbf{E}[\neg c_2 \mathbf{U} c_1]$。

使用标签算法中的(6) 来进行，其结果如图 6.11 所示。在图中“0:” 表示原来的状态，例如“$0: n_1 n_2$” 表示原来的初始状态。算法的(6) 中的 ① 首先对满足 c_1 的所有状态用 $\varphi = \mathbf{E}[\neg c_2 \mathbf{U} c_1]$ 进行标签，即对 s_2 和 s_4 进行标签，在图中用“$1:\varphi$” 来表示。然后算法的(6) 中的 ② 对不满足 c_2 并且有一个后继状态被标签的所有状态进行标签，即对状态 s_1 和 s_3 用 φ 进行标签，在图中用“$2:\varphi$” 来表示。进一步，对状态 s_0 由于它不满足 c_2 并且有一个后继状态 s_1 已经被标签，因此对状态 s_0 也用 φ 进行标签，在图中用“$3:\varphi$” 来表示。此后，由于再没有其他的状态能够进行标签，算法终止。

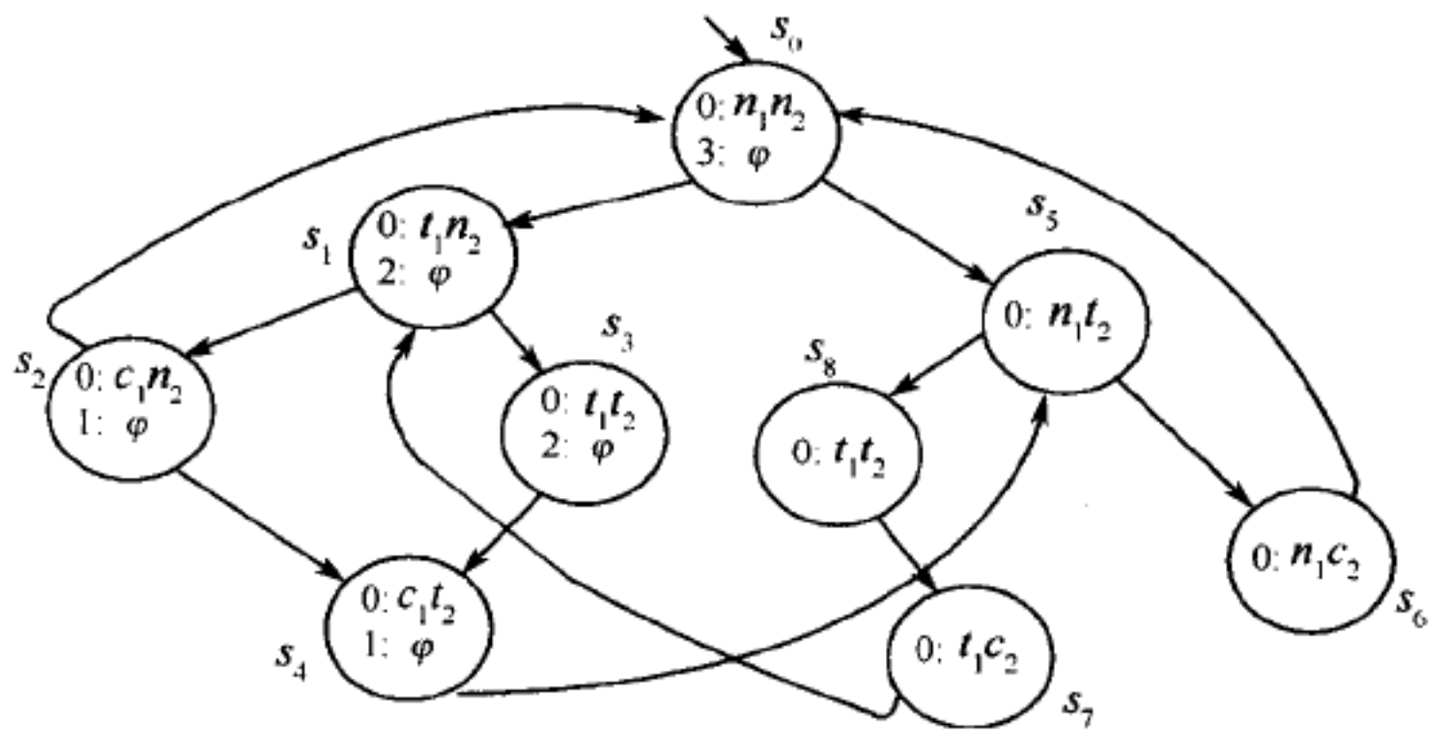


图 6.11 使用公式 $\mathbf{E}[\neg c_2 \mathbf{U} c_1]$ 对进程互斥的第二种模型应用标签算法

6.6.2 模型检验算法的程序实现

下面首先给出基本标签算法的程序代码。主函数 SAT 的功能是“满足”，它以一个CTL公式作为输入，它的组成代码类似于C语言或Pascal语言的程序代码。在一个函数中使用关键字 return 来说明该函数的返回值。有时也使用文字来说明对公式 ϕ 的语义树的根节点的分析。

local var 语句是说明一些局部变量，repeat until 是循环语句，它重复执行循环体，直到条件为真。此外，也使用了一些集合操作，像求交、求补等；实际上，将需要一种称为集合(Sets) 的抽象数据类型，它包含了对这些操作的实现。但在这里只对 SAT 算法的原理与实现进行讨论，对集合的操作是容易实现的，在此略去。

假定主函数 SAT 已经成功地存取了模型的所有相关部分：S、$\rightarrow$ 和 L。特别地，忽略 SAT 将需要 M 的一个描述作为输入这个事情，而是简单地假定 SAT 直接对任何一种给

定的这种模型进行操作。

下面首先给出 SAT 函数的代码。它对容易处理的情况直接处理，而对复杂的情况则用特定的过程逆归调用 SAT 函数，之后，给出 SAT 函数的子函数。这些子函数使用了 Sets 类型的变量 X、Y、V 和 W。

```
Function  SAT(ϕ)      // 确定满足公式 ϕ 的状态的集合
{
    switch(ϕ)
    { case  "⊤": return S;
      case "⊥": return ϕ;
      case "atomic"(ϕ 是一个原子描述): return{s∈S|ϕ∈L(s)};
      case "¬ϕ1":  return S-SAT(ϕ1);
      case "ϕ1 ∧ ϕ2": return SAT(ϕ1)∩SAT(ϕ2);
      case "ϕ1 ∨ ϕ2": return SAT(ϕ1)∪SAT(ϕ2);
      case "ϕ1→ϕ2": return SAT(¬ϕ1 ∨ ϕ2);
      case "AXϕ1": return SAT(¬EX¬ϕ1);
      case "EXϕ1": return  SAT_EX(ϕ1);
      case "A[ϕ1 Uϕ2]": return  SAT(¬(E[¬ϕ2 U(¬ϕ1 ∧¬ϕ2)]∨EG¬ϕ2));
      case "E[ϕ1 Uϕ2]": return  SAT_EU(ϕ1,ϕ2);
      case "EFϕ1": return  SAT(E(⊤Uϕ1));
      case "EGϕ1": return  SAT(¬AF¬ϕ1);
      case "AFϕ1": return  SAT_AF(ϕ1);
      case "AGϕ1": return  SAT(¬EF¬ϕ1);
    }
}
```

函数 SAT 调用了 3 个子函数：当 **EX**、**EU** 或 **AF** 是所输入的 CTL 公式所对应的语义树的根节点时，就分别调用子函数 SAT_{EX}、SAT_{EU} 和 SAT_{AF}。

```
Function  SAT_EX(ϕ)      // 确定满足 EXϕ 的状态的集合
{
  local var  X,Y;
  X=SAT(ϕ);
 Y={s0∈S|对一些 s1∈X,有 s0→s1成立};
 return  Y;
}
```

这里，函数 SAT_{EX} 首先调用 SAT 计算满足 ϕ 的状态；然后找到满足 **EX**ϕ 的状态。

```
Function  SAT_AF(ϕ)      // 确定满足 AFϕ 的状态的集合
{
    local var  X,Y;
    X=S;
    Y=SAT(ϕ);
```

```
    repeat until   X=Y      // 循环执行,直到 X 与 Y 相等为止
     {     X=Y;
           Y=Y∪{s| 对满足 s→s′的所有 s′,都有 s′∈Y 成立};
           }
  return   Y;
}
```

函数 SAT_{AF} 首先调用 SAT 计算满足 ϕ 的状态;然后按照基本标签算法中的方法找到满足 **AF**ϕ 的这些状态。

```
Function   SAT_EU(ϕ,φ)      // 确定满足 E[ϕUφ]的状态的集合
{
  local var   W,X,Y;
  W=SAT(ϕ);
   X=S;
  Y=SAT(φ);
  repeat until   X=Y    // 循环执行,直到 X 与 Y 相等为止
       {     X=Y;
             Y=Y∪(W∩{s|存在 s′,使得 s′→s 并且 s′∈Y};
       }
  return   Y;
}
```

这里,函数 SAT_{AF} 首先调用 SAT 计算满足 ϕ 的状态;然后按照基本标签算法中的方法找到满足 $E[\phi U \varphi]$ 的这些状态。

关于上面 4 个函数的功能正确性将在后面的 6.9 节中说明。下面讨论一下验证问题的复杂性。

尽管标签算法与模型的规模成线性关系,然而不幸的是模型自身的规模是与变量数和部件数成指数关系。这意味着,向验证程序中增加一个变量,将可能使验证的复杂性增大 1 倍。这种状态空间的急剧变化趋势(变为非常大)就是著名的状态爆炸问题。人们研究了很多方法来处理这个问题,例如采用如下 3 种方法:

(1)使用高效的数据结构。编序二元判定图,它能够表示状态的一个集合,以取代对单个状态的表示。

(2)抽取。把模型中的与所检查的公式无关的变量提取出来。

(3)分解。把验证问题划分成很多简单的验证问题。

6.7 符号模型验证系统

到目前为止,这一章的内容是相当理论化的,然而,对模型检验它也有实用的方面。在目前的实际应用中,已有很多有效的完成方法,它们能在可接受的时间内验证大型系统。在本节中,讨论较著名的一个 CTL 模型检验器,称它为符号模型验证系统(SMV)。它提供了一种语言来描述模型,并且可以直接检查 CTL 公式对模型的有效性。

6.7.1 验证系统的结构

SMV 是自由软件，这里仅说明 SMV 的设计思想。

SMV 的输入是一个文本。该文本是描述一个模型及其规范(CTL 公式)的程序。若对所有初始状态规范都成立，它就输出“真”；若规范不成立时，它就输出相关的原因。

SMV 是由多个程序模块组成的。它是用 C 语言或 Java 语言编写的，有一个称为 main 的主程序。每个程序模块可以声明一些变量并且可以对它们进行赋值，开始时是给一些变量赋初始值，而变量的将来值常常是当前变量的一个表达式。

下面是对 SMV 的一种输入。

```
Module  main
var
    request:  boolean;
    status:  {ready,busy};
assign
    init (status)=ready;
    next (status)=case
                    request:  busy;
                      1:  {ready,busy};
                      esac;
spec
    AG(request->AF status=busy);
```

该输入由一段程序和一个规范组成。程序部分有两种变量：request 类型是布尔型；status 类型是枚举型，它有两种取值 ready 和 busy。0 代表 F(假 False)，1 代表 T(真 True)。

在这段程序中，变量 request 的值没有被确定，这意味它可以由用户确定，或者由程序环境决定。变量 status 的值是部分地被确定：开始时它是 ready；在 request 为真时，它就变为 busy；若 request 的值为假，则 status 的值是没有被确定的。

注意 case1 是默认情况。如果 case request 不执行时，才执行该语句。

在这段程序中，由关键字 spec 和 CTL 公式来引入规范。在处理规范时，SMV 是使用字符 &、|、->、和! 来分别代替 ∧、∨、→和¬。因为这 4 种字符在标准键盘上是一定有的。

如上这段程序所表示的迁移系统模型如图 6.12 所示。

在图中，“busy”是“status=busy”的缩写；“request”是“request=ture”的缩写。在此模型中共存在 4 种状态，每一种状态与两个变量的一种可能值对应。此外，该迁移系统是非确定性的，即迁移系统中一个状态的下一个状态不能唯一地被定义。例如，状态“¬ request,busy”可以迁移至 4 种状态(它自身和 3 个其他状态)。

从图 6.12 中也可以看出：变量 request 随环境变化的方式；基于 stutas 的任何状态迁移的后继状态都是成对出现的：后继状态中的 request 的值分别是真或假。

用图 6.12 可以容易地验证模块 main 中的规范是成立的。

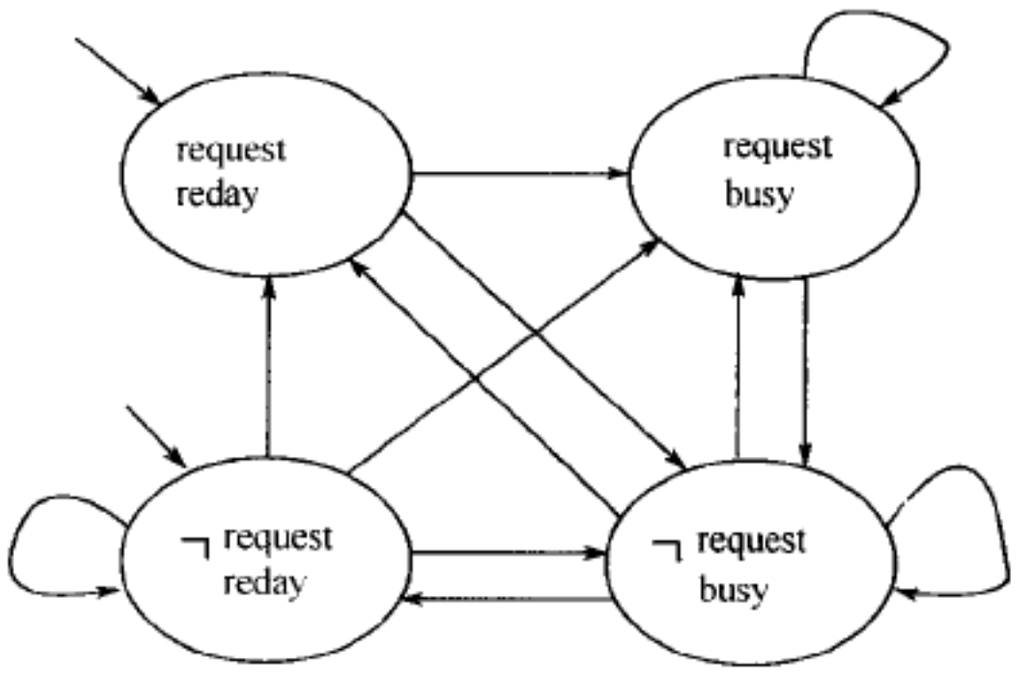


图 6.12　一个程序段的迁移系统模型

6.7.2　验证系统中的模块

SMV 支持将一个系统划分为多个模块，以增加程序的可读性和验证模块间的相互作用。当一个模块的名字在变量类型的声明中被说明时，该模块就被实例化。下面用一个例子来说明。

```
module  main
var
    bit0:  counter_cell(1);
    bit1:  counter_cell(bit0.carry_out);
    bit2:  counter_cell(bit1.carry_out);
spec
    AG AF  bit2.carry_out;
module  counter_cell(carry_in);
var
    value:  boolean;
assign
    init(value)=0;
    next(value)=(value+carry_in)  mod  2;
define
    carry_out=value & carry_in;
```

该程序的功能是实现一个计算器。它是由 3 个 1 位计数器组成，计算范围是从 000 到 111。模块 counter_cell 被实例化 3 次，每次所用的名字分别是 bit0、bit1 和 bit2。

计算器模块有一个形式参数 carry_in，在 bit0 中该参数所赋的值为 1；在 bit1 中赋的值为 bit0. carry_out，即 bit1 的 carry_in 是模块 bit0 的 carry_out。

这里使用了带点“.”的表达例如 m. v，其含义是指模块 m 的变量 v。这种表示方法在很多程序设计语言中使用，以实现对一条记录或一个对象的一些域的操作(存取)。

关键字 define 用于为 carry_out 分配一个表达式 value&carry_in，这种 define 声明的效果相当于声明一个变量并给它赋值，即

```
var
    carry_out:  boolean;
assign
    carry_out=value&carry_in;
```

下面讨论一下 SMV 中模块的运行方式。在默认情况下，SMV 中的模块是按照同步方式运行的，即存在一个全局时钟，每一次时钟跳动，各个模块都并行执行。也可以通过使用关键字 process，使得模块异步地执行。这时，模块可以以不同的速度自由运行；时钟的每一次跳动，它们中的一个被选择并被执行一个时钟周期。这种异步运行方式在一些方面是有用的，例如描述通信协议和描述异步电路和系统，这些协议或系统的行为针对时钟而言是异步的，而不是同步的。

6.7.3 验证系统在互斥访问中的应用

对符号模型验证系统 SMV 讨论一个实际例子即进程的互斥。对此，下面首先给出进程互斥访问协议的 SMV 代码。

```
Module  main
var
    turn: boolean;
    pr1: process prc(pr2.st,turn,0);
    pr2: process prc(pr1.st,turn,1);
assign
    init(turn)=0;
SPEC  AG! ((pr1.st=c)&(pr2.st=c))                    // 安全性
SPEC  AG((pr1.st=t)->AF(pr1.st=c))                   // 许可性
SPEC  AG((pr2.st=t)->AF(pr2.st=c))                   // 许可性
SPEC  EF(pr1.st =c&E[pr1.st=c U (! pr1.st=c&
                E[! pr2.st=c U pr1.st=c])])          // 非排序
Module  prc(other_st,turn,myturn);
var
    st:{n,t,c};
assign
    init (st)=n;
    next (st)=
        case
            (st=n):  {t,n};
            (st=t) & (other_st=n):  c;
            (st=t) & (other_st=t)&(turn=myturn):  c;
            (st=c):  {c,n};
            1:  st;
        esac;
```

```
        next (turn):=
                case
                    turn=myturn & st=c:  ! turn;
                    1:  turn;
                esac;
    fairness  running;
    fairness  ! (st=c);
```

该程序由两个模块 main 和 prc 组成。主模块有一个变量 turn，其作用是：当有两个程序模块都试图进入临界区时，该变量用来决定这两个模块各自进入临界区的先后次序。

主模块也有程序模块 prc 的两个实例。在实例中，变量 st 是表示进程的状态，以说明进程是处在临界区、不在临界区或者是试图进入临界区。变量 other_st 是表示其他进程的状态。

变量 st 的值的变化方式如下：

(1) 当它的值为 n 时，它可以仍然停留为 n，或者向 t 移动变为 t；

(2) 当它的值为 t 时，若其他一个进程的 st 值是 n，则它会直接变为 c，但是如果另外一个进程的 st 值是 t 时，在变为值 c 之前，它将检查进入临界区的次序。

(3) 当 st 的值为 c 时，它可以移动回到 n，即 st 的值可以变为 n。

因此每一个 prc 实例当它进入它的临界区时，就给出其他一个进程进入临界区的次序。

SMV 的一个重要特征是：能够对它的搜索树进行限制，以沿着一个 CTL 公式 ϕ 为真的路径执行。这一点经常用于对资源的公平存取进行模型化，也称为公平约束，并由关键字 fairness 来说明。因此，fairness ϕ 的发生就意味着：当检查一个规范 ϕ 时，SMV 将忽略仅发生有限多次的任何一条路径。

在程序模块 prc 中，限制了变量 st 是无限次不等于 c 的那种计算路径，这是因为程序代码允许一个进程停留在它的临界区只要它自己愿意。这时，就可能出现不满足“许可性”的情况：如果进程 2 永远停留在它的临界区，则进程 1 将永远不能进入临界区。这是相当不公平的，为了避免这种情形的出现，规定了公正性约束：!(st = c)。

如果一个子程序模块已经用 process 关键字作了声明，这时在每一个时间点上，SMV 将决定是否选择它来执行。

有时，希望忽略一些路径，在这些路径上程序模块没有必要花费处理时间，对此可以使用 running。保留字 running 在公平约束中可以用来代替一个 CTL 公式。用 fairness running 可以完成对执行路径的限制，即对仅发生有限多次的任何一条路径该程序模块都不执行(运行)。此外，在 prc 中，对这种路径也进行限制。这样做的理由在于：在没有这种限制的条件下，如果每个程序模块都未被选中来运行，则将很容易造成对“许可性”的不满足。

一般情况下，都假定进程的调度是公平的。在上面的程序中，这种假定是由两个 fairness 语句来表达的。

接下来分析如上程序所对应的迁移系统，如图 6.13 所示。

在图 6.13 中，一个圆圈表示一个状态。圆圈中的表达式都是由 3 个字符组成，每个

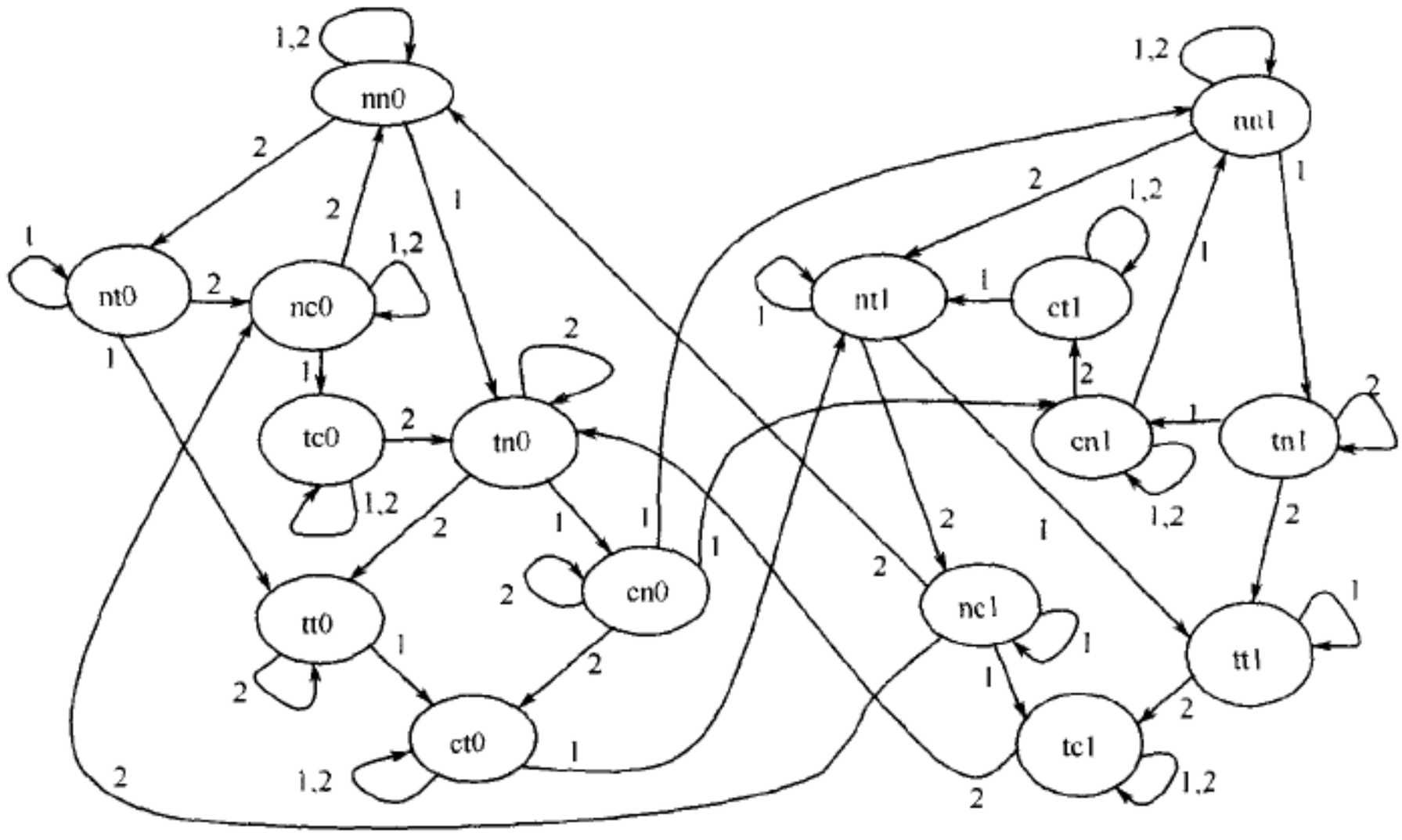


图 6.13　进程互斥程序的迁移系统模型

字符分别说明进程 1 和进程 2 的状态以及变量 turn 的取值情况。例如，ct1 是一种状态，在该状态，进程 1 是处于临界区(由 c 表示)，而进程 2 试图进入临界区(由 t 表示)，并且 turn=1。在迁移上的数字说明了哪一个进程被选中执行。一般地，每个状态有多种迁移方式，在一种特定的迁移中，或者是进程 1 移动，或者是进程 2 移动。

图 6.13 中给出的模型与图 6.8 中给出的互斥模型有一些不同，主要有如下两个原因：

(1)图 6.13 的模型引入了布尔变量 turn，以区分 s_3 和 s_8 这两种状态。现在需要区分一些状态(例如 ct0 和 ct1)，它们在以前(图 6.8 中)是相同的。然而，如果仅仅从迁移关系来看，这些状态是没有差别的，它们满足相同的 CTL 公式；而在这种 CTL 公式中不涉及 turn。为了对这些状态做区分，唯一的方法是把它们提取出来进行处理。

(2)在图 6.8 中，已删去了一些简单的迁移方式。例如，假定系统在时钟的每一次跳动都将移向不同的状态，但从一个状态到它自身不存在迁移。在图 6.13 中，就允许每一个状态从它自身迁移到自身，这代表了一个进程被选中执行并做一些局部计算，但是它不移进或移出它的临界区。通过这种方法，也为一个进程停留在它的临界区引入了迁移路径。当然，根据需要，也可以删去这种路径，一种常用方法是使用公正性约束。

6.8　具有公正性的模型检验

由于模型 M 可能包含不现实的一些行为，或者在实际系统中不满足这些行为时，对 $M, s_0 \models \phi$ 的验证可能会失败。处理这种问题的方法是对模型 M 进行提炼；或者仍然使用原来的模型，但增加一个过滤器到这个模型中。这样，取代模型检验 $s_0 \models \phi$，验证 $s_0 \models \psi \rightarrow \phi$，其中，$\psi$ 是一个规范，它是对原模型进行提炼之后的表达。

然而，对 CTL 模型检验，并不是所有对模型的提炼都可以用这种方式来完成。在前面

几节中看到，SMV 允许对它所描述的迁移系统增加公正性约束。这种公正性约束说明一个给定的公式沿着每一个计算路径都是成立的，称这种路径为公正性计算路径。公正性约束的出现就意味着：当计算在规范中 CTL 公式是否成立时，连接 **A** 和 **E** 的范围仅仅局限于公正性路径。

例如，在互斥的情况下，为了表示进程 prc 能停留在它的临界区（st = c），对此可以用非决定性分配来模型化：

```
next(st):=
    case
      ...  ...
       (st=c):  {c,n};
     ...  ...
    esac;
```

然而，如果实际上允许进程 2 停留在它的临界区，则有一条违背约束 $\mathbf{AG}(t_1 \rightarrow \mathrm{AF}_{c1})$ 的路径。这是因为，如果进程 2 永远停留在它的临界区，则 t_1 为真而不管 c_1 是否曾经变为真。

也可以忽略这条路径，即可以假定进程能停留在它的临界区，但它在有限的时间内最终将退出临界区，因此经常增加公正性约束 !st = c。这意味着，无论进程处于什么状态，在将来将存在一个状态该进程不处于临界区。

简单的公正性约束的形式是：

性质 ϕ 为真。

还有其他的一些形式，例如：

如果 ϕ 为真，则 ψ 也为真。

SMV 能够处理简单的公正性约束。它如何处理呢？为对此进行说明，下面解释如何对模型检验算法进行修改，使得 ***A*** 和 ***E*** 的范围仅仅局限于公正性计算路径。

定义 6.5　设 $C = \{\psi_1, \psi_2, \cdots, \psi_n\}$ 是一个由 n 个公正性约束所构成的集合。对这些公正性约束，称一条计算路径 $S_0 \rightarrow S_1 \rightarrow \cdots$ 是公平的，如果对每一个 i，存在无穷多个 j 使得 $S_j \models \psi_i$，即每一个 ψ_i 沿着这条路径都为真。把操作 **A** 和 **E** 受限于公正性路径的情况写为 $\mathbf{A}_C$ 和 $\mathbf{E}_C$。

例如，$M, S_0 \models \mathbf{A}_C\mathbf{G}\phi$ 成立，当且仅当沿着公正性路径的每一个状态 ϕ 为真。也可以简单地直接表示为 $\mathbf{A}_C F$、$\mathbf{A}_C U$ 等。注意这些操作依赖于公正性约束所选择的集合 C。已经知道 $\mathbf{E}_C U$、$\mathbf{E}_C\mathbf{G}$ 和 $\mathbf{E}_C\mathbf{X}$ 形成一个合适的集合，这一点可以使用没有公正性约束的与谓词连接相同的方式来说明。因此也有如下的等式，即

$$\mathbf{E}_C[\phi \mathbf{U} \psi] \equiv \mathbf{E}[\phi \mathbf{U}(\psi \wedge \mathbf{E}_C\mathbf{G} \top)]$$

$$\mathbf{E}_C\mathbf{X}\phi \equiv \mathbf{EX}(\phi \wedge \mathbf{E}_C\mathbf{G} \top)$$

为了说明两个等式，可以观察得到，一条计算路径是公平的，当且仅当它的任何下标是公平的，因此，仅需要对计算 $E_C G\phi$ 提供一个如下的算法，该算法与对计算 **EG** 的**方法**类似：

（1）对图施加约束使它的状态满足 ϕ；对得到的结果图，弄清在什么状态下存在公正性路径。

(2) 找到约束图的最大的强连通分量。

(3) 一个强连通分量，对一些 ψ_i，它不包含满足 ψ_i 的状态时就去除它。这样，所得的结果强连通分量就是公正性强连通分量。可以达到的受限图中的任何状态都有一条公正性路径。

(4) 使用广度优先搜索去找到约束图中能够达到一个公正性强连通分量的状态。

该算法的复杂性是 $O(n \cdot f \cdot (V+E))$，即仍然与模型和公式的规模成线性增长关系。图 6.14 说明了计算满足 $\mathbf{E}_C\mathbf{G}\phi$ 这种状态的过程。一个状态满足 $\mathbf{E}_C\mathbf{G}\phi$，当且仅当对图施加约束使它的状态满足 ϕ 时，该状态有公正性路径。一条公正性路径使得一个强连通分量至少通过满足公正性约束的一个状态。在图 6.14 的例子中，C 等于 $\{\psi_1, \psi_2, \psi_3\}$。

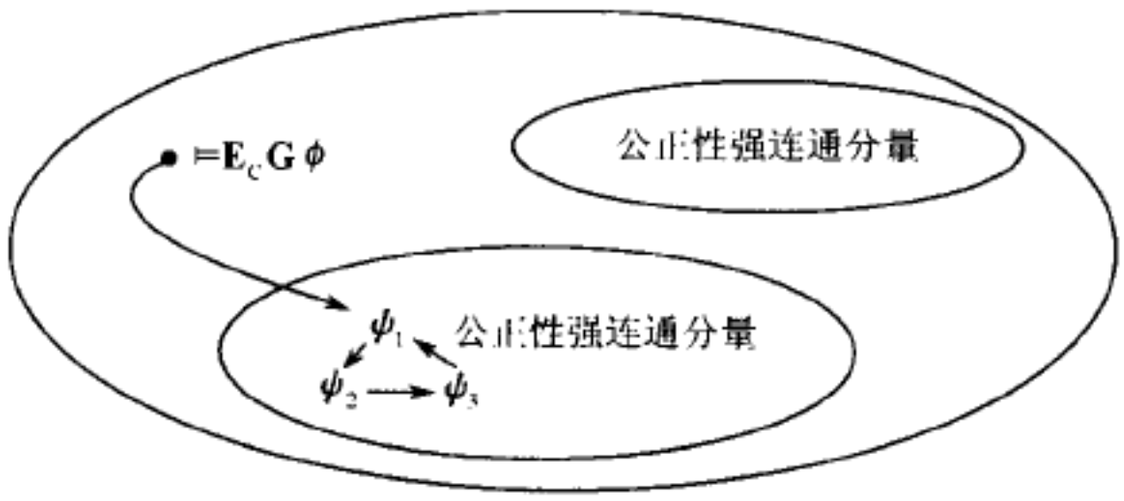


图 6.14　计算满足 $\mathbf{E}_C\mathbf{G}\phi$ 的状态

6.9　CTL 的不动点特性

在 6.6.2 小节中所讨论的一个算法是：给定一个 CTL 公式 ϕ 和一个 CTL 模型 $M=(S, \rightarrow, L)$，去计算满足 ϕ 的所有状态 s，这里 $s \in S$。将这种状态组成的集合写为 $[\![\phi]\!]$。算法对 ϕ 的结构采用递归方式来操作。对只有一个元素的公式 ϕ（例如 $\bot$、$\top$ 或 p）直接计算 $[\![\phi]\!]$。其他的公式一般是由一些较小的子公式通过 CTL 的连接所构成的，例如，若 ϕ 是 $\psi_1 \vee \psi_2$，则算法计算集合 $[\![\psi_1]\!]$ 和 $[\![\psi_2]\!]$，并以一定的方式对它们进行组合，以获得 $[\![\psi_1 \vee \psi_2]\!]$。

由**定义 6.3**，可以得到如下等式：

(1) $[\![\top]\!] = S$

(2) $[\![\bot]\!] = \varnothing$；

(3) $[\![\neg\phi]\!] = S - [\![\phi]\!]$

(4) $[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$

(5) $[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!]$

(6) $[\![\phi_1 \rightarrow \phi_2]\!] = (S - [\![\phi_1]\!]) \cup [\![\phi_2]\!]$

(7) $[\![\mathbf{AX}\phi]\!] = S - [\![\mathbf{EX}\neg\phi]\!]$

(8) $[\![\mathbf{A}(\phi_1 \mathbf{U} \phi_2)]\!] = [\![\neg(\mathbf{E}(\neg\phi_1 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)) \vee \mathbf{EG}\neg\phi_2)]\!]$

当处理一个公式比如 $\mathbf{EX}\psi$，它涉及一个谓词操作，这时就会出现更有意义的情况：算法计算集合 $[\![\psi]\!]$，然后计算有迁移进入 $[\![\psi]\!]$ 中一个状态的所有这些状态的集合。这是与语义 $\mathbf{EX}\psi$：$M, s \models \mathbf{EX}\psi$ 相一致的，当且仅当存在一个状态 s'，有 $s \rightarrow s'$，且 $M, s' \models \psi$。

对大多数这种逻辑操作，可以类似地进行讨论，可以说明算法能够正常工作。然而，对 **EU**、**AF** 和 **EG**，需要迭代一定的次数直到它稳定，这在推理方面不是那么明显。下面的讨

论是设计一些语义方面的操作以允许对算法的正确性提供一个完整的说明。

考虑 6.6.2 节中 FunctionSAT(ϕ) 的程序，可以看到这些语句中的大多数按照 CTL 的语义是显然正确的。例如，可以看出用 $\phi_1 \rightarrow \phi_2$ 调用它时 SAT 应做的工作。在该程序中也存在一些操作过程不是十分明显的语句，例如，若输入公式是 $\mathbf{A}[\phi_1 \mathbf{U} \phi_2]$，则可以看到 SAT 调用它自身，并以一个相当复杂的公式作为输入。若对一些 ψ，输入公式是 $\mathbf{AF}\psi$，则函数 SAT_{AF} 被调用并且将使用一个 while 循环。需要对这个 while 语句的正确执行提供说明。

在 6.6.2 节中给出了构建 SAT_{EU} 和 SAT_{AF} 的方法，它依赖于具有循环语句的迭代过程。现在仍然缺少的是一种计数方式，它可以保证所有的这种迭代能正常停止，并且所得的计算结果是真正所希望的。

下面提供 SAT_{AF} 和 SAT_{EU} 的正确性说明。此外，也给出了一个过程 SAT_{EG}，并且说明了它的正确性。

过程 SAT_{EG} 如下，它基于在 6.6 节中给出的方法：当没有一个后继状态被标明时，把具有相同后继状态的标签集进行求交来删除标签。

```
Function   SAT_EG(ϕ)      // 确定满足 EGϕ 的状态的集合
    {
      local var   X,Y;
      Y=SAT(ϕ);
     X=∅;
     repeat until   X=Y      // 循环执行，直到 X 与 Y 相等为止
       {   X=Y;
         Y=Y∩{s|存在 s′，使得 s′→s 并且 s′∈Y};
        }
    return   Y;
    }
```

$\mathbf{EG}\phi$ 的语义说明 $s_0 \models \mathbf{EG}\phi$ 成立，当且仅当存在一条计算路径 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$，使得 $s_i \models \phi$ 对所有 $i \geqslant 0$ 成立。

对此可以用另一种表达：$\mathbf{EG}\phi$ 成立，如果 ϕ 成立并且 $\mathbf{EG}\phi$ 在当前状态的一个后继状态成立。这相当于如下等式：

$$\mathbf{EG}\phi = \phi \wedge \mathbf{EX}\,\mathbf{EG}\phi$$

根据连接的语义定义可以很容易地证明这个等式。

可以得

$$[\![\mathbf{EX}\psi]\!] = \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{并且 } s' \in [\![\psi]\!]\}$$

因此，可以推出

$$[\![\mathbf{EG}\phi]\!] = [\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{并且 } s' \in [\![\mathbf{EG}\phi]\!]\}$$

这显然不是对计算 $\mathbf{EG}\phi$ 的一种非常有效的方法，这是由于为了计算出右边而需要知道 $\mathbf{EG}\phi$ 的值。这里，根据$[\![\mathbf{EG}\phi]\!]$ 的如上这种表达形式，对 $\mathbf{EG}\phi$ 的计算可以用处理不动点问题的方法来进行。

6.9.1 单调函数

定义 6.6 设 S 是一个状态集合，令 $P(S)$ 表示集合 S 的超集。设 $F:P(S)\to P(S)$ 是一个定义于 S 的超集上的函数。

(1) 称 F 是单调，如果由 $X\subseteq Y$，可推出 $F(X)\subseteq F(Y)$ 对所有 S 的子状态 X 和 Y 都成立。

(2) 对 S 的一个子集 X，如果有 $F(X)=X$，则称 X 是 S 的一个不动点。

单调函数的例子。设 $S=\{s_0,s_1\}$，且 $F(Y)=Y\cup\{s_0\}$ 对 S 的所有子集 Y 成立。由于由 $Y\subseteq Y'$ 可以推出 $Y\cup\{s_0\}\subseteq Y'\cup\{s_0\}$，因此可以得到 F 是单调的。F 的不动点是 S 中包括 s_0 的所有子集。这样，F 有两个不动点，即 $\{s_0\}$ 和 $\{s_0,s_1\}$。这里，也可以说 F 有一个最小的不动点 $\{s_0\}$ 和一个最大的不动点 $\{s_0,s_1\}$。

下面是一个非单调函数的例子。函数 $G:P(S)\to P(S)$ 由如下方式定义：若 $Y=\{s_0\}$ 则 $G(Y)=\{s_1\}$；否则 $G(Y)=\{s_0\}$。因此 G 将 $\{s_0\}$ 映射到 $\{s_1\}$，将其他的集合映射到 $\{s_0\}$。函数 G 是非单调的，这是因为有 $\{s_0\}\subseteq\{s_0,s_1\}$，但 $G(\{s_0\})=\{s_1\}$ 不是 $G(\{s_0,s_1\})=\{s_0\}$ 的一个子集；同时 G 也没有任何不动点。

在说明 SAT 的正确性时使用 $P(S)$ 上的单调函数，其主要原因是：

(1) 单调函数一定存在一个最小的和一个最大的不动点。

(2)**EG**、**AF** 和 **EU** 的含义可以通过使用 $P(S)$ 上的单调函数的最大或最小不动点来表达。

(3) 这些不动点是容易计算的，且能够对过程 SAT_{EU} 和 SAT_{AF} 进行修改以进行不动点计算。

在下面的讨论中，表达式 $F^i(x)$ 是指函数 F 是对 X 应用了 i 次，例如 $F^3(x)=F(F(F(X)))$。此外，对上面定义的函数 $F(Y)=Y\cup\{s_0\}$，有 $F^2(Y)=F(F(Y))=(Y\cup\{s_0\})\cup\{s_0\}=Y\cup\{s_0\}=F(Y)$，此时，$F^2=F$。可以进一步证明 $F^i=F$ 对所有 $i\geqslant 0$ 成立。显然，对一个任意的函数 G，$G^i=G$ 并不一定成立。

下面的定理说明了单调函数的收敛性。

定理 6.1 设 S 是一个具有 $n+1$ 个元素的集合，$S=\{s_0,s_1,\ldots,s_n\}$。若 $F:P(S)\to P(S)$ 是一个单调函数，则 $F^{n+1}(\varnothing)$ 是 F 的最小不动点，$F^{n+1}(S)$ 是 F 的最大不动点。

证明：由于 $\varnothing\subseteq F(\varnothing)$，且 F 是单调的，因此有 $F(\varnothing)\subseteq F(F(\varnothing))$，即 $F^1(\varnothing)\subseteq F^2(\varnothing)$。可以用数学归纳法说明 $F^1(\varnothing)\subseteq F^2(\varnothing)\subseteq F^3(\varnothing)\subseteq\cdots\subseteq F^i(\varnothing)$ 对所有 $i\geqslant 1$ 成立。

特别地，取 $i=n+1$，可以说如上的一表达 $F^k(\varnothing)$ 已经是 F 的一个不动点，这里 $0\leqslant k\leqslant n+1$。否则，$F^1(\varnothing)$ 应至少包含一个元素(这时有 $F^1(\varnothing)\neq\varnothing$)；用同样的表示，$F^2(\varnothing)$ 应至少包含有两个元素，因为它比 $F^1(\varnothing)$ 大。继续进行，就有 $F^{n+2}(\varnothing)$ 将包含有至少 $n+2$ 个元素。这最后的一种情况是不可能的，因为 S 只有 $n+1$ 个元素。因此，$F(F^k(\varnothing))=F^k(\varnothing)$ 对一些 k 成立，$0\leqslant k\leqslant n+1$。该等式也说明了 $F^{n+1}(\varnothing)$ 是 F 的一个不动点。

现在假定 X 是 F 的另外一个不动点，下面去证明 $F^{n+1}(\varnothing)$ 是 X 的一个子集，从而说明 $F^{n+1}(\varnothing)$ 是 F 的最小不动点。

由于 $\varnothing \subseteq X$，F 是单调的，并且 X 是 F 的一个不动点，因此有 $F(\varnothing) \subseteq F(X) = X$。由归纳法，可以获得 $F^i(\varnothing) \subseteq X$ 对所有 $i \geqslant 0$ 成立。因此，对 $i = n+1$，有 $F^{n+1}(\varnothing) \subseteq X$。

类似地，可以说明 $F^{n+1}(S)$ 是 F 的最大不动点。

定理证毕。

定理 6.1 不仅说明了最小和最大不动点的存在性，而且也提供了一种计算它们的方法。例如，在计算 F 的最小不动点时，所要做就是对空集 $\varnothing$ 应用 F，之后再对 $F(\varnothing)$ 应用 F，继续进行，直到在 F 的应用之前已变为常数。**定理 6.1** 也说明了这一过程的迭代次数，假定 S 有 $n+1$ 个元素，则在最坏的情况下达到不动点的迭代次数上限为 $n+1$。

6.9.2 SAT_{EG} 的正确性

在上一节中我们获得了

$$[\![\mathbf{EG}\phi]\!] = [\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{并且 } s' \in [\![\mathbf{EG}\phi]\!]\}$$

这说明了 $\mathbf{EG}\phi$ 是如下函数的一个不动点，即

$$F(X) = [\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{并且 } s' \in X\}$$

事实上，F 是单调的，$\mathbf{EG}\phi$ 是它的最大不动点，因此可以使用**定理 6.1** 来计算 $\mathbf{EG}\phi$。

定理 6.2 设 F 具有如上的形式，即 $F(X) = [\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{并且 } s' \in X\}$。令 $n = |S|$，即 n 是 S 中所含元素的个数，则：

(1) F 是单调的。

(2) $[\![\mathbf{EG}\phi]\!]$ 是 F 的最大不动点。

(3) $[\![\mathbf{EG}\phi]\!] = F^{n+1}(S)$。

证明：(1) 为了说明 F 是单调的，取 S 的任意两个子集 X 和 Y，使得 $X \subseteq Y$，需要说明 $F(X)$ 是 $F(Y)$ 的子集。

给定一个 s_0 使得存在一些 $s_1 \in X$ 且有 $s_0 \rightarrow s_1$，由于 X 是 Y 的子集，因此就有 $s_0 \rightarrow s_1$，这里 $s_1 \in Y$。这样就说明了 $\{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in X \text{ 成立}\} \subseteq \{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in Y \text{ 成立}\}$，从而就有

$$F(X) = [\![\phi]\!] \cap \{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in X\} =$$
$$[\![\phi]\!] \cap \{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in Y\} = F(Y)$$

(2) 我们已经看到 $[\![\mathbf{EG}\phi]\!]$ 是 F 的一个不动点，为了证明它是最大的不动点，需要说明对任何满足 $F(X) = X$ 的集合 X，它都被 $[\![\mathbf{EG}\phi]\!]$ 所包含。设 s_0 是这种不动点集合 X 的一个元素，需要说明 s_0 也在集合 $[\![\mathbf{EG}\phi]\!]$ 中。为此，使用如下的事实，即

$$s_0 \in X = F(X) = [\![\phi]\!] \cap \{s \in S \mid s \rightarrow s_1 \text{ 对一些 } s_1 \in X\}$$

可以推出 $s_0 \in [\![\phi]\!]$，且对一些 $s_1 \in X$，成立 $s_0 \rightarrow s_1$。

此外，因为 s_1 在 X 中，可以得到

$$s_1 \in X = F(X) = [\![\phi]\!] \cap \{s \in S \mid s \rightarrow s_2 \text{ 对一些 } s_2 \in X\}$$

由此可以推出 $s_1 \in [\![\phi]\!]$，且对一些 $s_2 \in X$，成立 $s_1 \rightarrow s_2$。

由数学归纳法，可以构造一条路径 $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s_{n+1} \rightarrow \cdots$，使得 $s_i \in [\![\phi]\!]$ 对所有 $i \geqslant 0$ 成立。由 $[\![\mathbf{EG}\phi]\!]$ 的定义可知，这说明了 $s_0 \in [\![\mathbf{EG}\phi]\!]$。

(3) 由于 $[\![\mathbf{EG}\phi]\!]$ 是 F 的最大不动点，根据**定理 6.1**，直接可以得到

$$[\![\mathbf{EG}\phi]\!] = F^{n+1}(S)$$

定理证毕。

现在来说明过程 SAT_{EG} 是正确的。

首先，把在过程 SAT_{EG} 中的行

$$Y := Y \cap \{s \mid \text{存在 } s', \text{使得 } s \rightarrow s', \text{且 } s' \in Y\}$$

改写为

$$Y := SAT(\phi) \cap \{s \mid \text{存在 } s', \text{使得 } s \rightarrow s', \text{且 } s' \in Y\}$$

改写这种形式之后不改变过程过程 SAT_{EG} 的功能与效果。为说明这一点，注意到第一次循环，Y 是 $SAT(\phi)$，而在随后的循环中，有 $Y \subseteq SAT(\phi)$。因此，我们是用 Y 还是 $SAT(\phi)$ 来求交这是无关紧要的。

根据对过程 SAT_{EG} 中行的这种改写，可以清楚地知道 SAT_{EG} 就是所计算的 F 的最大不动点。因此，从**定理 6.2** 可知，它是正确的。

6.9.3 SAT_{EU} 的正确性

说明 SAT_{EU} 的正确性也与证明 SAT_{EG} 的正确性类似。首先，有如下的等式，即

$$\mathbf{E}[\phi \mathbf{U} \psi] = \psi \vee (\phi \wedge \mathbf{EX}\, \mathbf{E}[\phi \mathbf{U} \psi])$$

该等式也可写成

$$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!] = [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{且 } s' \in [\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]\}$$

这说明$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$ 是如下函数的一个不动点，即

$$G(X) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s \mid \text{存在 } s' \text{ 使得 } s \rightarrow s', \text{且 } s' \in X\}$$

下面说明函数 $G(X)$ 是单调的，可以得到$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$ 是函数 SAT_{EU} 的最小不动点，它可以按**定理 6.1** 的方式来进行计算。

定理 6.3 设 $G(X)$ 是按如上方式定义的函数，令 $n = |S|$，则

(1)G 是单调的。

(2)$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$ 是 G 的最小不动点。

(3)$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!] = G^{n+1}(\varnothing)$。

证明：(1) 当 $X \subseteq Y$ 成立时，需要说明 $G(X) \subseteq G(Y)$ 也成立。首先，将集合 X 映射到集合$\{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in X\}$ 的函数是单调的；其次 $G(X)$ 所做的只是把此集合与常数集$[\![\phi]\!]$ 和$[\![\psi]\!]$ 求交和求并。因此，G 是单调的。

(2) 若 S 有 $n+1$ 个元素，由**定理 6.1**，则 G 的最小不动点等于 $G^{n+1}(\varnothing)$。下面说明该集合等于$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$。对空集 $\varnothing$ 应用 G，有

$$\begin{aligned} G^1(\varnothing) &= [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in \varnothing \text{ 成立}\} \\ &= [\![\psi]\!] \cup ([\![\phi]\!] \cap \varnothing) = [\![\psi]\!] \cup \varnothing = [\![\psi]\!] \end{aligned}$$

因此 $G^1(\varnothing)$ 是包含了所有的状态 $s_0 \in [\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$。这里按照直到(until) 的定义，选取 $i = 0$。

类似地，$G^2(\varnothing) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s_0 \in S \mid s_0 \rightarrow s_1 \text{ 对一些 } s_1 \in G^1(\varnothing) \text{ 成立}\}$

这说明 $G^2(\varnothing)$ 的元素是所有的状态 $s_0 \in [\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$，此时选取 $i \leqslant 1$。

由数学归纳法，有 $G^{K+1}(\varnothing)$ 是所有状态 s_0 的集合，它满足 $i \leqslant k$，且 $s_0 \in [\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$。

由于这对所有的 k 成立，因此$[\![\mathbf{E}[\phi \mathbf{U} \psi]]\!]$ 是所有集合 $G^{K+1}(\varnothing)$ 的并集($k \geqslant 0$)。但是由于 $G^{n+1}(\varnothing)$ 是 G 的一个不动点，因此可以得到：所有集合 $G^{k+1}(\varnothing)$ 的并集正好就是

$G^{n+1}(\varnothing)$。故$[\![\mathbf{E}[\phi\mathbf{U}\psi]]\!]$是$G$的最小不动点。

(3) 由如上(2)的讨论可知，$[\![\mathbf{E}[\phi\mathbf{U}\psi]]\!] = G^{n+1}(\varnothing)$。

定理证毕。

程序 $\mathrm{SAT}_{\mathrm{EU}}$ 的正确性说明类似于对 $\mathrm{SAT}_{\mathrm{EG}}$ 的说明，把如下的行

$$Y := Y \cup (W \cap \{s \mid \text{存在 } s' \text{ 使得 } s \to s', \text{且 } s' \in Y\})$$

改写为

$$Y := \mathrm{SAT}(\psi) \cup (W \cap \{s \mid \text{存在 } s' \text{ 使得 } s \to s', \text{且 } s' \in Y\})$$

这样做并不改变程序 $\mathrm{SAT}_{\mathrm{EU}}$ 的运行结果。这是因为第一次循环时，Y 是 $\mathrm{SAT}(\psi)$，而且，由于Y是一直递增的，这使得是对Y还是对 $\mathrm{SAT}(\psi)$ 来执行集合的并操作的结果是一样的。

通过使用这种对行的形式的改变，很容易看出程序 $\mathrm{SAT}_{\mathrm{EU}}$ 就正好是使用**定理 6.3** 来计算 G 的最小不动点。

下面通过一个例子来说明上面对函数 F 和 G 的相关讨论结果。考虑图 6.15 所示的系统。从计算集合$[\![\mathbf{EF}p]\!]$开始。由 **EF** 的定义可知，该集合正好是$[\![E(\top \mathbf{U}p]\!]$。因此定义 $\phi_1 = \top$ 和 $\phi_2 = p$。根据图 6.15 的结构，有$[\![p]\!] = \{s_3\}$，当然有$[\![\top]\!] = S$。因此，函数G等于

$$G(X) = \{s_3\} \cup \{s \in S \mid s \to s' \text{ 对一些 } s' \in X \text{ 成立}\}$$

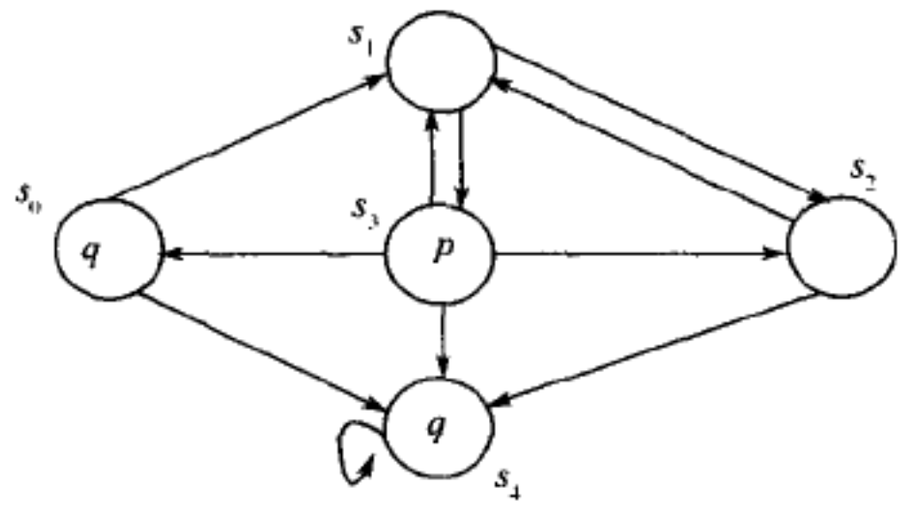


图 6.15　一个例子系统

由于$[\![\mathbf{E}(\top \mathbf{U}p]\!]$等于$G$的最小不动点，需要对 $\varnothing$ 应用G并进行迭代，直到这一过程稳定。

$$G^1(\varnothing) = \{s_3\} \cup \{s \in S \mid s \to s' \text{ 对一些 } s' \in \varnothing \text{ 成立}\} = \{s_3\}$$

$$G^2(\varnothing) = G(G^1(\varnothing)) = \{s_3\} \cup \{s \in S \mid s \to s' \text{ 对一些 } s' \in \{s_3\} \text{ 成立}\} = \{s_1, s_3\}$$

$$G^3(\varnothing) = G(G^2(\varnothing)) = \{s_3\} \cup \{s \in S \mid s \to s' \text{ 对一些 } s' \in \{s_1, s_3\} \text{ 成立}\}$$
$$= \{s_0, s_1, s_2, s_3\}$$

$$G^4(\varnothing) = G(G^3(\varnothing)) = \{s_3\} \cup \{s \in S \mid s \to s' \text{ 对一些 } s' \in \{s_0, s_1, s_2, s_3\} \text{ 成立}\}$$
$$= \{s_0, s_1, s_2, s_3\}$$

由**定理 6.3** 知，$\{s_0, s_1, s_2, s_3\}$ 是 G 的最小不动点，它等于$[\![\mathbf{E}(\top \mathbf{U}p]\!]$。由于$[\![\mathbf{E}(\top \mathbf{U}p]\!] = [\![\mathbf{EF}p]\!]$，因此有$[\![\mathbf{EF}p]\!] = \{s_0, s_1, s_2, s_3\}$。

另一个例子是计算集合$[\![\mathbf{EG}q]\!]$。由**定理 6.2**，该集合是函数 F 的最大不动点，这里定义 $\phi = q$。从图 6.15 可知$[\![q]\!] = \{s_0, s_4\}$，并且有

$$F(X) = [\![q]\!] \cap \{s \in S \mid s \to s' \text{ 对一些 } s' \in X \text{ 成立}\}$$
$$= \{s_0, s_4\} \cap \{s \in S \mid s \to s' \text{ 对一些 } s' \in X \text{ 成立}\}$$

由于$[\![\mathbf{EG}q]\!]$等于 F 的最大不动点，因此需要对 S 应用 F 并进行迭代，直到这一过程稳

定。首先，有

$$F^1(S) = \{s_0, s_4\} \cap \{s \in S \mid s \to s' \text{ 对一些 } s' \in S \text{ 成立}\}$$

由于对每一个 s 都有一些 s' 存在，并满足 $s \to s'$，因此 $F^1(X)$ 可写成

$$F^1(S) = \{s_0, s_4\} \cap S = \{s_0, s_4\}$$

其次，有

$$F^2(S) = F(F^1(S)) = \{s_0, s_4\} \cap \{s \in S \mid s \to s' \text{ 对一些 } s' \in \{s_0, s_4\} \text{ 成立}\} = \{s_0, s_4\}$$

这说明 $\{s_0, s_4\}$ 是 F 的最大不动点，由**定理 6.2** 可知，它等于$[\![\mathbf{EG}q]\!]$。

6.10 符号模型检验

在模型检验中使用二元判定图(BDD) 可以对具有大量状态空间的系统进行验证，从而使得验证方法的性能有较大的提高。在本节中，详细描述在前面几节所讨论的模型检验算法如何通过使用 BDD 作为基本数据结构来实现。

在 6.6.2 节中所讨论的算法的程序代码以 CTL 公式 ϕ 作为输入，并且返回在给定模型中满足 ϕ 的状态集合。对这些程序代码进行分析之后可以发现，算法是通过操作一系列的中间状态集合来完成的。下面将说明如何用编序二元判定图(OBDD) 来存储给定的模型及其中间状态的集合，并且说明如何用对 OBDD 的操作来完成这些程序代码的操作。

首先以用 OBDD 来表示状态的集合开始，然后将这种表示扩展到迁移系统，最后对所需要的一些操作的完成方法进行说明。称使用 OBDD 的模型检验为符号模型检验，它不是单独表示每个状态，而是强调同时表示多个状态所构成的集合。

6.10.1 状态集的表示

设 S 是一个有限集，这里暂时不把它作为状态集。我们的任务是用 OBDD 来表示 S 的多个子集。由于 OBDD 是针对布尔函数的，因此需要把 S 的元素编码成布尔值。一般的做法是：对 S 的每一个元素 $s \in S$ 分配一个唯一的布尔值矢量 $(v_1, v_2, \cdots, v_n)$，这里每一个 $v_i \in \{0,1\}$。我们用函数 f_T 来说明 S 的一个子集 T，f_T 的特征是：若 $s \in T$，则 f_T 将 $(v_1, v_2, \cdots, v_n)$ 映射到 1，否则映射到 0。

对长度为 n 的布尔矢量 $(v_1, v_2, \cdots, v_n)$，存在 2^n 个这种矢量。因此，应选择 n 使得 $2^{n-1} < |S| \leqslant 2^n$，这里 $|S|$ 是表示 S 中元素的个数。若 $|S|$ 不是 2 的幂，则会存在一些矢量它们不对应于 S 中的任何元素，这时就忽略这些矢量，不使用。

函数 $f_T: \{0,1\}^n \to \{0,1\}$，它说明了对每一个 s，可以由一个对应的矢量 $(v_1, v_2, \cdots, v_n)$ 来表示，也说明了 s 它是否在集合 T 中。称 f_T 为集合 T 的特征函数。

对 S 是 CTL 模型 $M = (S, \to, L)$ 的状态集合的情况，把 S 表示为布尔矢量也是一种自然的方法。标签函数 $L: S \to \psi$(原子) 给出了编码。假定对原子集合有固定的编序，例如 $x_1, x_2, \cdots, x_n$。然后，由矢量 $(v_1, v_2, \cdots, v_n)$ 来表示 S 中的元素 $s \in S$。对每一个 i，若 $x_i \in L(s)$，则 $v_i = 1$，否则 $v_i = 0$。

为了满足对每一个 s 有唯一的布尔矢量表示，这里要求：对所有 $s_1, s_2 \in S$，由 $L(s_1) = L(s_2)$ 可推出 $s_1 = s_2$。如果这一点不满足的话，这时可以增加额外的原子以达到能够进行区分。

从现在开始，对一个状态 $s \in S$，用它的布尔矢量$(v_1, v_2, \cdots, v_n)$来代表它。这里，若 $x_i \in L(s)$ 则 $v_i = 1$，否则 $v_i = 0$。当用 OBDD 时，状态可以由布尔函数 $L_1 \cdot L_2 \cdot L_3 \cdot \cdots \cdot L_n$ 所对应的 OBDD 来表示。此时，若 $x_i \in L(s)$，则 $L_i = x_i$；否则 $L_i = \overline{x}_i$。

状态集合$\{s_1, s_2, \cdots, s_m\}$由如下布尔函数

$$L_{11} \cdot L_{12} \cdot L_3 \cdot \cdots \cdot L_{1n} + L_{21} \cdot L_{22} \cdot L_{23} \cdot \cdots \cdot L_{2n} + \cdots + L_{m1} \cdot L_{m2} \cdot L_{m3} \cdot \cdots \cdot L_{mn}$$

所对应的 OBDD 来表示。这里，$L_{i1} \cdot L_{i2} \cdot L_{i3} \cdot \cdots \cdot L_{in}$ 表示状态 s_i。

用这种表示的一个突出特点是：表示状态集合时 OBDD 的规模可以很小。

【例 6.6】　考虑图 6.16 中的 CTL 模型。

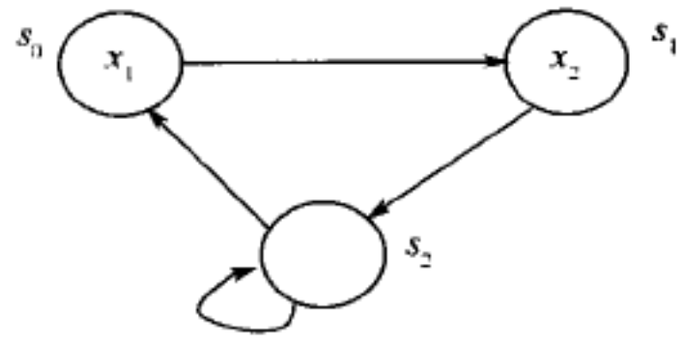


图 6.16　一个 CTL 模型

在该例子中，定义：$S = \{s_0, s_1, s_2\}$；

$$\rightarrow = \{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\};$$

$$L(s_0) = \{x_1\}; L(s_1) = \{x_2\}; L(s_2) = \varnothing。$$

该系统有如下性质：对所有状态 s_1 和 s_2，由 $L(s_1) = L(s_2)$ 可推出 $s_1 = s_2$，即一个状态完全由原子公式的真值所决定。状态的集合可以由布尔值表示，也可以由布尔函数表示。对编序 $x_1 < x_2$，表 6.1 给出了布尔值和布尔函数的表示。

表 6.1　在图 6.16 中状态的子集表示

状态集	布尔值表示	布尔函数表示
$\varnothing$		0
$\{s_0\}$	(1,0)	$x_1 \cdot \overline{x}_2$
$\{s_1\}$	(0,1)	$\overline{x}_1 \cdot x_2$
$\{s_2\}$	(0,0)	$\overline{x}_1 \cdot \overline{x}_2$
$\{s_0, s_1\}$	(1,0),(0,1)	$x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot x_2$
$\{s_0, s_2\}$	(1,0),(0,0)	$x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot \overline{x}_2$
$\{s_1, s_2\}$	(0,1),(0,0)	$\overline{x}_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2$
S	(1,0),(0,1),(0,0)	$x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2$

注意，矢量(1,1)和它对应的函数 $x_1 \cdot x_2$ 是不使用的，在对 S 的子集的表示中可以包括它，也可以不包括它。因此，在 OBDD 的操作中，为了使 OBDD 的规模较小，可以选择包括它或不包括它。例如，子集$\{s_0, s_1\}$可以由布尔函数 $x_1 + x_2$ 来较好地表示，因为它的 OBDD 要小于函数 $x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot x_2$ 的 OBDD，如图 6.17 所示。

为了说明对S的子集的如上OBDD表示方法是否适合于6.6.2节中所给出的算法，需要把在算法中使用的对子集的操作按所定义的 OBDD 的操作来完成。算法所定义的操作是：

(1) 求集合的交、并和补。显然这分别可以用布尔函数的 $\cdot$、$+$、$-$ 来表示。这也可以通过 OBDD 的相关操作来表示。

(2) 函数 $\mathrm{pre}_{\exists}(X) = \{s \in S \mid$ 存在 $s', (s \rightarrow s'$ 且 $s' \in S)\}$

$$\mathrm{pre}_{\forall}(X) = \{s \mid \text{对所有 } s', (s \rightarrow s' \text{ 可推出 } s' \in X)\}$$

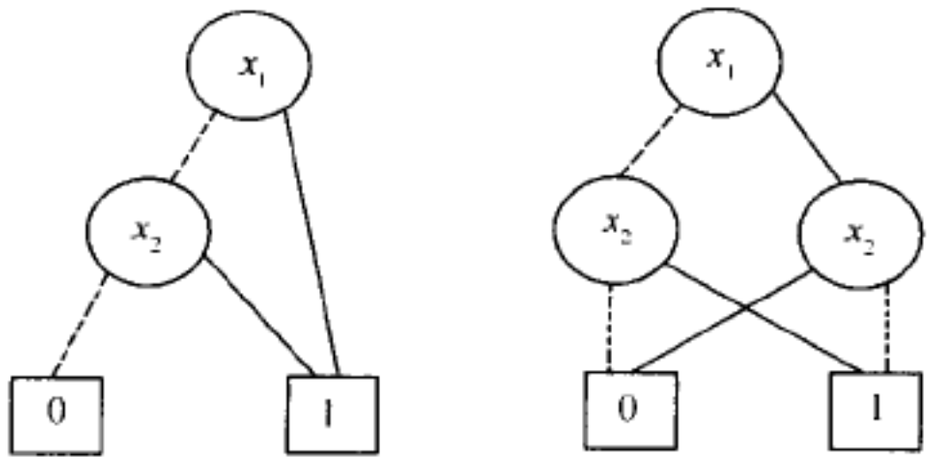


图 6.17　集合$\{s_0, s_1\}$的两种 OBDD 表示

函数 $\text{pre}_\exists$ 有助于 SAT_{EX} 和 SAT_{EU} 的现实，它作用于状态的子集 X，并且返回能发生一次迁移为 X 的状态集合。函数 $\text{pre}_\forall$，在 SAT_{AF} 中使用，它作用于子集 X，并且返回只能发生一次迁移为 X 的状态集合。为了说明用 OBDD 如何来完成这两种函数，首先需要说明如何表示迁移关系自身。

6.10.2　迁移关系的表示

模型 $M=(S, \rightarrow, L)$ 的迁移关系 $\rightarrow$ 是 $S\times S$ 的一个子集。由前面的讨论可知，一个给定有限集的子集可以由一种二元编码的特征函数来表示，并用 OBDD 表示。

对 S 的子集情况，二元编码可以由标签函数 L 给出。由于 $\rightarrow$ 是 $S\times S$ 的一个子集，因此需要布尔矢量的两个备份。这样，链接 $s\rightarrow s'$ 由一对布尔矢量$((v_1, v_2, \cdots, v_n), (v_1', v_2', \cdots, v_n'))$来表示。这里，若 $x_i \in L(s)$，则 $v_i=1$；否则 $v_i=0$。类似地，若 $x_i \in L(s')$，则 $v_i'=1$；否则 $v_i'=0$。对 OBDD 而言，链接可由布尔函数$(L_1 \cdot L_2 \cdot L_3 \cdot \cdots \cdot L_n) \cdot (L_1' \cdot L_2' \cdot L_3' \cdot \cdots \cdot L_n')$的 OBDD 来表示。而对链接的集合，例如，全局关系 $\rightarrow$，这时的 OBDD 对应于这些布尔函数相加。

【例 6.7】　为了计算图 6.16 的迁移关系的 OBDD，首先说明它的真值表，见表 6.2。

在表中最后一列的每一个 1 与迁移关系的一个链接相对应；每一个 0 对应于不出现一个链接；最后一列值为“—”是表示“不关心”的链接。根据需要可以包括或不包括这些链接。有时包括它，可以用于使最后的 OBDD 的规模更小。

把表 6.2 最后一列的值为 1 的行取出，就可以写出布尔函数为

$$f^{\rightarrow}=\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}'_1 \cdot \bar{x}'_2+\bar{x}_1 \cdot \bar{x}_2 \cdot x'_1 \cdot \bar{x}'_2+x_1 \cdot \bar{x}_2 \cdot \bar{x}'_1 \cdot x'_2+\bar{x}_1 \cdot x_2 \cdot \bar{x}'_1 \cdot \bar{x}'_2$$

实验结果说明对链接 $\rightarrow$，在 OBDD 中将变量交叉会更有效，例如，使用编序 $x_1<x_1'<x_2<x_2'$ 就比使用 $x_1<x_2<x_1'<x_2'$ 有效。表 6.3 给出了用这种交叉的变量编序对应的真值表。

在考虑应包括哪些“不关心”的行时，把一行与它的相邻行做成组，以形成多个 0 串和多个 1 串。例如，对表 6.3，把第二框中的第一个和第二个“—”分别当做 1 和 0 来处理，这就使得在 OBDD 中计算关系 $\rightarrow$ 的值 1，1，0，0 时，不会涉及到变量 x_2'。如果把表 6.3 中的最后 3 个“—”当做 0 来处理，这时第四个框就独立于 x_2 和 x_2'，即与 x_2 和 x_2' 的值无关。获得的结果 OBDD 如图 6.18 所示。

下面讨论函数 $\text{pre}_\exists$ 和 $\text{pre}_\forall$ 的实现方法。对迁移关系的处理，当给定了 X 和 $\rightarrow$ 的 OBDD 之后，设它们所对应的 OBDD 分别为 B_X 和 $B_{\rightarrow}$，剩下的问题是怎样来计算 $\text{pre}_\exists(X)$

表 6.2　在图 6.16 中迁移关系的真值表

x_1	x_2	x_1'	x_2'	→
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	—
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	—
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	—
1	1	0	0	—
1	1	0	1	—
1	1	1	0	—
1	1	1	1	—

表 6.3　交叉的变量编序对应的真值表

x_1	x_1'	x_2	x_2'	→
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	—
0	1	1	0	0
0	1	1	1	—
1	0	0	0	0
1	0	0	1	1
1	0	1	0	—
1	0	1	1	—
1	1	0	0	0
1	1	0	1	—
1	1	1	0	—
1	1	1	1	—

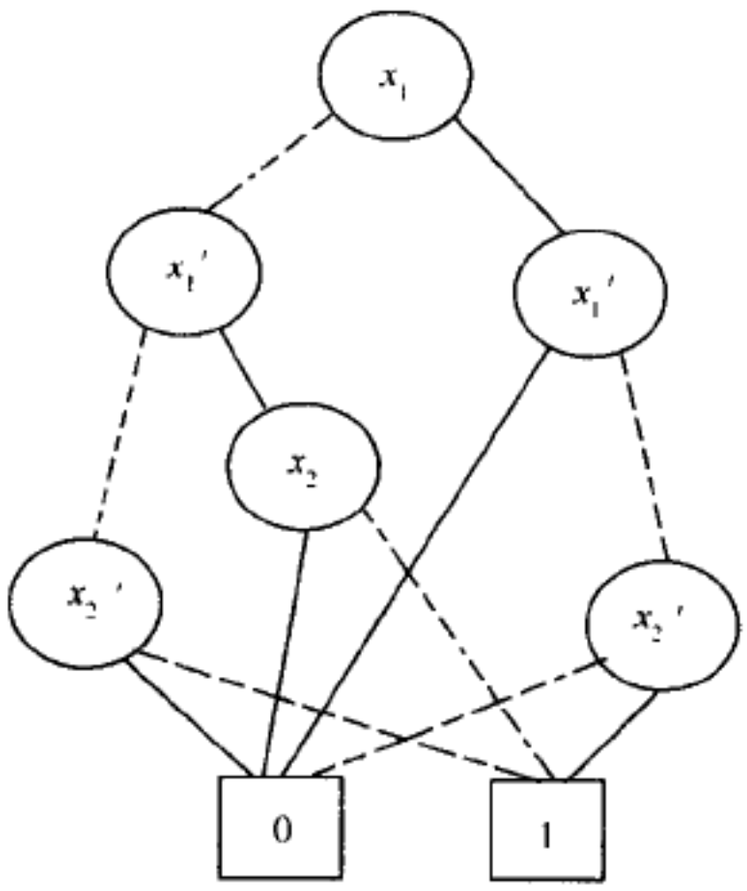


图 6.18　在图 6.16 中迁移关系对应的 OBDD

和 $\mathrm{pre}_\forall(X)$ 所对应的 OBDD。首先，不难发现 $\mathrm{pre}_\forall$ 可以由 $\mathrm{pre}_\exists$ 及其补来表示，即

$$\mathrm{pre}_\forall(X) = S - \mathrm{pre}_\exists(S - X)$$

这里的表达式 $S-X$ 是表示由所有的 $s \in S$，但 $s \notin X$，所有具有这种性质的元素 s 所构成的集合。因此，仅需要说明由 B_X 和 $B_{\to}$ 如何计算 $\mathrm{pre}_\exists(X)$ 的 OBDD。根据

$$\mathrm{pre}_\exists(X) = \{s \in S \mid \text{存在 } s', (s \to s', \text{且 } s' \in S)\}$$

由此可知，可以按如下方式计算 $\mathrm{pre}_\exists(X)$ 所对应的 OBDD：

(1) 对 B_X 中的变量重新命名，都使用原始变量，例如 $x_1, x_2, \cdots, x_m, \cdots$。称结果 OBDD 为 $B_{X'}$。

(2) 使用 apply() 和 exists() 算法，计算 exists($\hat{x'}$,apply($\cdot, B_{\to}, B_{X'}$)) 所对应的 OBDD。该 OBDD 即为 $\mathrm{pre}_\exists(X)$ 所对应的 OBDD。这里 $\hat{x'}$ 代表($x_1', x_2', \cdots, x_n'$)。

说明：如上由迁移关系来生成 OBDD 的方法是首先计算真值表，然后计算 OBDD，此时的 OBDD 可能不是约简的，因此需要调用约简“reduce”函数。对于具有很多或大量变量的实际系统，真值表的规模随布尔变量的增加而成指数增长，根据真值表来计算 OBDD 的方法就不能适应这种情况。此时就需要以系统的描述为基础，直接生成 OBDD，而不使用与规模成指数关系的中间表示（比如真值表）。

作为 OBDD 在电路验证中的进一步应用，下面简要说明怎么对电路进行表示可以容易生成其 OBDD。

（1）同步电路。图 6.19 所示的时序电路是一个同步电路，即所有的状态变量是并行地同步更新。

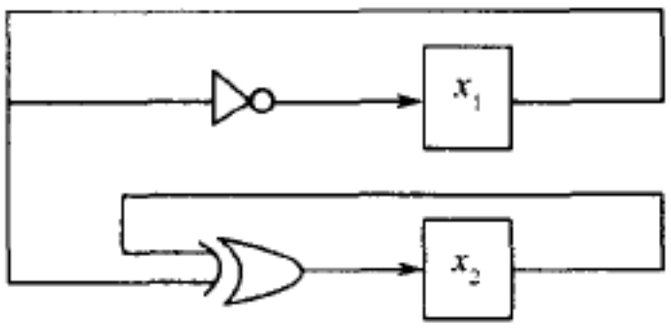


图 6.19　具有两个寄存器的一个同步电路

该电路的功能可以通过对寄存器 x_1 和 x_2 在下一个状态的值来说明。函数 $f^{\rightarrow}$ 说明了电路的可能的下一个状态，其形式为

$$(x_1' \leftrightarrow \overline{x}_1) \cdot (x_2' \leftrightarrow x_1 \oplus x_2)$$

容易将该函数 $f^{\rightarrow}$ 转换成它的一个 OBDD 表示。

（2）异步电路。与同步电路比较，异步电路的逻辑结构表示非常类似于 $f^{\rightarrow}$ 的 CTL 模型。可以构建函数 f_i，它是对电路中逻辑部件的可能的下一个状态所进行的描述。对异步电路与系统，存在两种构造这些函数并组成全局系统行为的方法：

① 在同时性模型中，对一次全局迁移，任意多个部件都可以产生它们各自的局部迁移。这可以模型化为

$$f^{\rightarrow} = \prod_{i=1}^{n} ((x_i' \leftrightarrow f_i) + (x_i' \leftrightarrow x_i))$$

② 在交叉模型中，对一次全局迁移，仅只有一个部件产生一次局部迁移，所有其他的部件都停留在它们的局部状态。这可以模型化为

$$f^{\rightarrow} = \sum_{i \neq 1}^{n} (x_i' \leftrightarrow f_i) \cdot \prod_{j \neq i} (x_j' \leftrightarrow x_j))$$

对这两种方法进行比较，同时性模型是乘积表达；而交错模型是求和表达。

第 7 章　二元判定图的结构

本章介绍布尔函数的二元判定图表示方法，对它的结构及其建立方法做了详细论述。二元判定图 BDD 目前在国内没有统一的名称，也被称做二元判定图、二元判断图或二叉判定图。本书中用二元判定图。

由于二元判定图在电路的设计与测试领域有着十分广泛的应用，因此在本章讨论二元判定图的结构之后，在第 8 章将继续讨论它的一些重要性质，并介绍它在电路设计及综合中的应用。在第 9 章和第 10 章将分别讨论组合电路的验证与时序电路的验证，并就二元判定图在这两个方面的应用做了说明。

7.1　二元判定图的概念

首先讨论布尔函数的图表示方法。设所要表示的布尔函数为 $f(x)$，它有 n 个变量 x_1，$x_2,\cdots,x_n$，即 $f(x_1,x_2,\cdots,x_n)$。令 $X=\{x_1,x_2,\cdots,x_n\}$。用一种有向无环图 $G=(V,E)$ 来表示布尔函数，其中 V 是图 G 的所有节点构成的集合（即节点集），E 是 G 的所有边构成的集合（即边集）。这里的名称“节点”也可用“结点”，在本书中使用“节点”这一名称。

这里 V 中的节点有两类：终节点和非终节点。用方形节点来表示终节点。终节点的值域取值为 0 或 1，如图 7.1 所示。

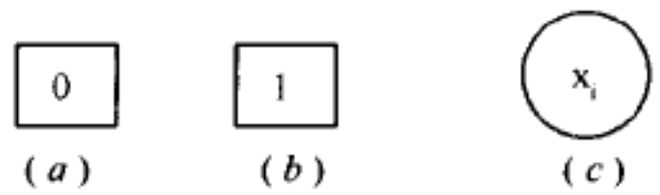


图 7.1　节点类型

在图 7.1 中(a) 所示节点为取值为 0 的终节点，简称为 0 终节点。图 7.1 中(b) 所示节点为取值为 1 的终节点，称为 1 终节点。在图 7.1 中(c) 为非终节点，它用圆圈表示。非终节点的值域取值为一个布尔变量 x_i，圆圈内标有该节点对应的变量。

定义 7.1　一个定义在布尔变量集合 X 上的有向无环图 G 若满足如下的条件，则称为判定图。

(1) G 的节点集合 V 非空，$V\neq\varnothing$。

(2) 图中的任一节点要么为终节点，要么为非终节点。

(3) 图中任一非终节点都有两个后继节点。

(4) 任一终端节点没有后继节点。

图 7.2 就是一个判定图的例子。

从图 7.2 可知，判定图的结构为树形结构，故有时也称它为判决树。在图 7.2 中，节点 x_1 为根节点。

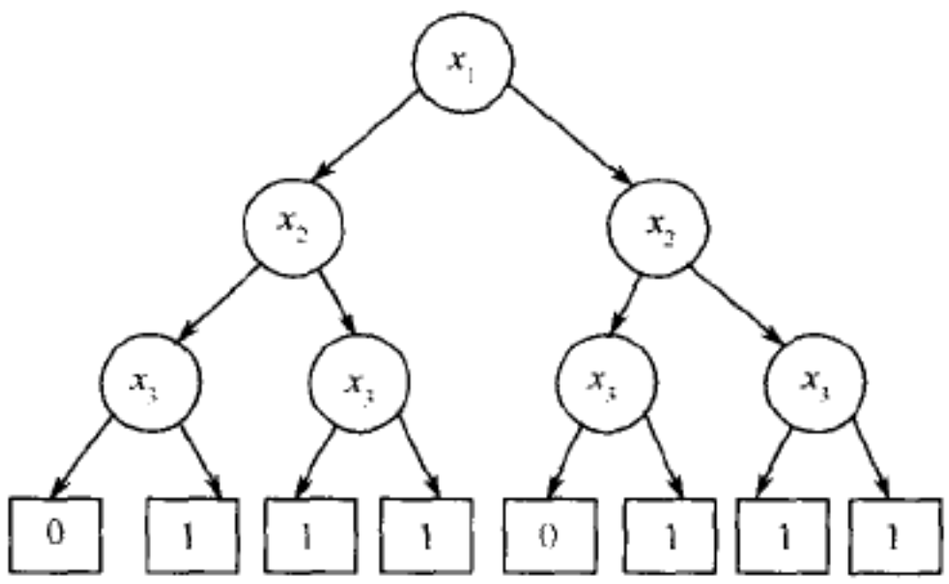


图 7.2　一个判定图

定义 7.2　对一个判定图 G，如果 G 中的任一条从根节点到一个终节点的路径上所有 n 个节点 $x_1, x_2, \cdots, x_n$ 都存在，且每一个只出现一次，则称判定图 G 是完全的。

例如图 7.2 就是一个完全判定图。

为了说明判定图中节点和边的性质，引入如下的表示。

(1) 将节点的取值(值域)称为节点的属性值，用 value 表示，例如，终节点 v 的属性值 value(v) = 0 或 1，非终节点 v 的属性值 value(v) = x_i，$i \in \{1, 2, \cdots, n\}$。特别地，对非终节点 v，把 value(v) 称为节点 v 的编序；同时为与终节点相区别，有时将 value(v) 用 index(v) 表示。

(2) 把非终节点 v 的两个子节点(后继节点)分别用 low(v) 和 high(v) 表示。这样，有时也可把节点 v 直接表示为(index(v), high(v), low(v)) 的形式。

下面给出二元判定图 BDD(Binary Decision Diagram) 的一般定义，并且将 BDD 与逻辑函数联系起来。

定义 7.3　二元判定图 G 是满足如下条件的一种判定图。设 G 有根节点 v，则 G 能表示按如下方式定义的逻辑布尔函数 f_v：

(1) 当 v 是终节点时，若 value(v) = 1，则 $f_v = 1$；若 value(v) = 0，则 $f_v = 0$。

(2) 当 v 是非终节点，且 v 的变量编序为 x_i 即 index(v) = x_i，则 f_v 使

$$f_v(x_1, x_2, \cdots, x_n) = \bar{x}_i \cdot f_{\text{low}(v)}(x_1, x_2, \cdots, x_n) + x_i \cdot f_{\text{high}(v)}(x_1, x_2, \cdots, x_n)$$

也称 x_i 为是节点 v 的判决变量，high(v) 是节点 v 的 1 节点，low(v) 是节点 v 的 0 子节点；称从节点 v 指向节点 high(v) 和 low(v) 的边为 v 的 1 边和 0 边。

对 $f_{\text{low}(v)}(x_1, x_2, \cdots, x_n)$ 和 $f_{\text{high}(v)}(x_1, x_2, \cdots, x_n)$ 按**定义 7.3** 进行递归定义，可以获得二元判定图 G 的所有节点。这些节点按 G 所表示的逻辑函数功能排成了许多从根节点开始到终节点的路径。在一条路径上，编序为 i 的节点 v 指向它的子节点 low(v) 和 high(v)，而节点 v 的判决变量 x_i 取值为 0 或 1。这样，一条路径上的所有节点的取值形成一个变量集的取值，在这个取值下函数的值就等于该路径所到达的终节点的值。

例如，对由真值表 7.1 给出的逻辑函数 f，按**定义 7.3** 所构造的 BDD 如图 7.3 所示。

在图 7.3 中，0 边用虚线画出，1 边用实线画出，所有边的方向都是从上到下。从图 7.3 中可以看出，每个判决变量 x_i 取值为 0 或 1。从根节点 x_1 到终节点的路径共有 8 条。每条路径上经过的节点顺序都为 x_1、x_2、x_3，且路径的终节点为 0(或 1)。这 8 条路径与表 7.1 中函数的 8 种取值组合相对应。例如，最左边的那一条路径：节点 x_1 至节点 x_2，节点 x_2 至节点 x_3，节点 x_3 至 0 终节点，这 3 条虚线组成的路径，是说明当判决变量 x_1、x_2、x_3 取值都为

0 时，函数 f 的取值为 0。

表 7.1　函数 f 的真值表

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

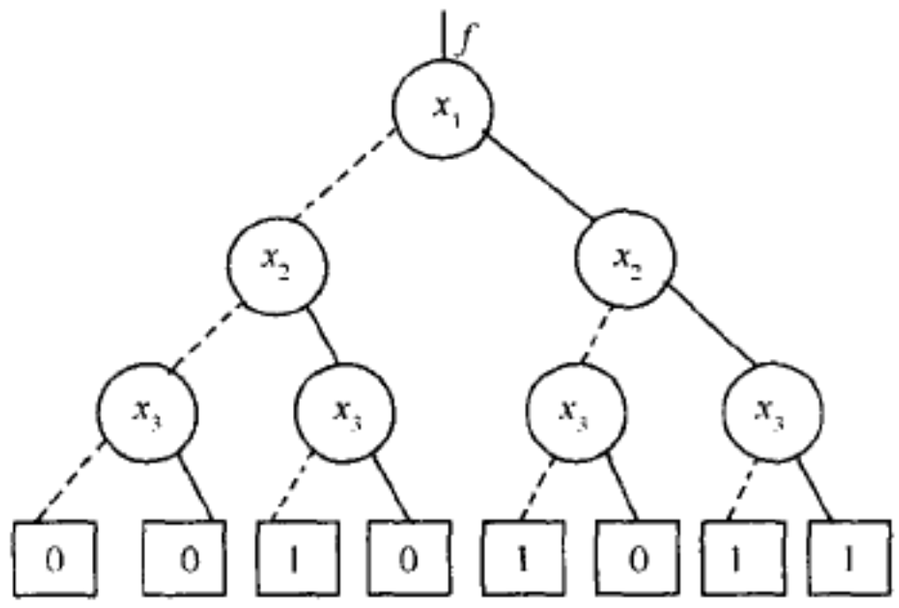


图 7.3　函数 f 的 BDD 表示

从上面的例子可以看出，给定了一个逻辑函数的真值表，可以构建它对应的一种 BDD；反之，由函数的 BDD 表示，也可以获得函数的真值表形式。

逻辑函数的表示除了真值表之外，还有一种常见的形式就是用函数表达式的形式给出。下面讨论此时的 BDD 表示。

对任意的一个逻辑函数 $f(x_1,x_2,\cdots,x_n)$，它可以写成如下的形式：

$$f = \overline{x}_i \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1}$$

这就是著名的香农(Shannon)展开公式。称 $f|_{x_i=1}$ 为 f 在 x_i 上的香农展开因子，$f|_{x_i=0}$ 为 f 在 x_i 上的补香农展开因子。

定义 7.4　设 $f(x_1,x_2,\cdots,x_n)$ 是 n 元布尔函数，f 的二元判定图 BDD 递归地定义如下：

(1) 若 $f(x_1,x_2,\cdots,x_n)$ 的值是常数(0 或 1)，则其 BDD 仅由一个终节点构成，终节点的值为该常数；否则，执行如下的步骤(2)。

(2) 选变量 x_i，生成非终节点 x_i。由该变量 x_i 对布尔函数 f 进行香农展开，并从该节点出发引出两条有向边，一条边上的权为 0 即 0 边，另一条边上的权为 1 即 1 边。

(3) 在 0 边上对布尔函数 $f|_{x_i=0}$ 重复进行(1) 和(2)；在 1 边上对布尔函数 $f|_{x_i=1}$ 也进行同样处理，直至出现终节点为止。

例如，对逻辑函数 $f = x_1\overline{x}_3 + x_1x_2 + x_2\overline{x}_3$，由**定义 7.4** 构造它所对应的 BDD 如图 7.4 所示。

其中，
$$g_1 = f|_{x_1=0} = x_2\overline{x}_3, \qquad g_2 = f|_{x_1=1} = x_2 + x_3$$
$$h_1 = g_1|_{x_2=0} = 0, \qquad h_2 = g_1|_{x_2=1} = \overline{x}_3$$
$$h_3 = g_2|_{x_2=0} = \overline{x}_3, \qquad h_4 = g_2|_{x_2=1} = 1$$
$$L_1 = h_2|_{x_3=0} = 1, \qquad L_2 = h_2|_{x_3=1} = 0$$
$$L_3 = h_3|_{x_3=0} = 1, \qquad L_4 = h_3|_{x_3=1} = 0$$

对逻辑函数 $f = x_1\overline{x}_3 + x_1x_2 + x_2\overline{x}_3$ 进行香农展开构造二元判定图时，依次选定的变量顺序为 x_1、x_2、x_3，即先用 x_1 进行，然后用 x_2；最后用 x_3。用小于符号“<”来表示这种顺序，即 $x_1 < x_2 < x_3$。

一般地，对同一布尔函数，选取不同的变量顺序按香农展开来构造出的二元判定图是

不相同的，例如对函数 $f = x_1\bar{x}_3 + x_1x_2 + x_2\bar{x}_3$，选取变量顺序为 $x_3 < x_1 < x_2$，则所对应的 BDD 如图 7.5 所示。

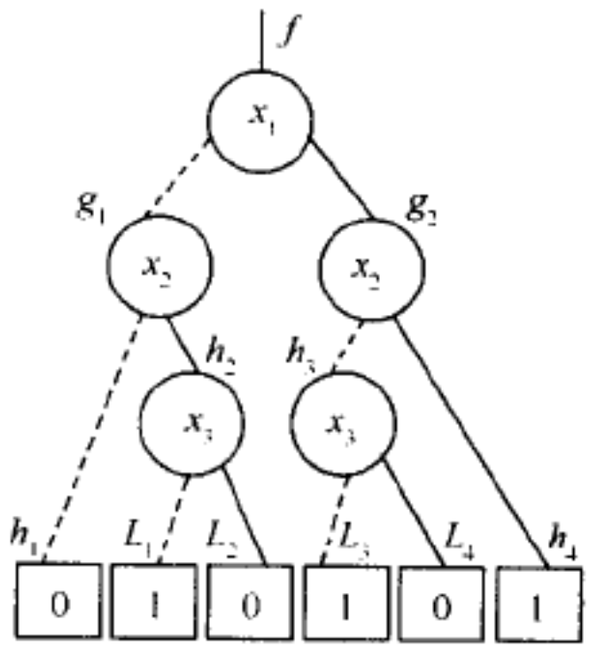


图 7.4　由香农展开式获得 f 的 BDD

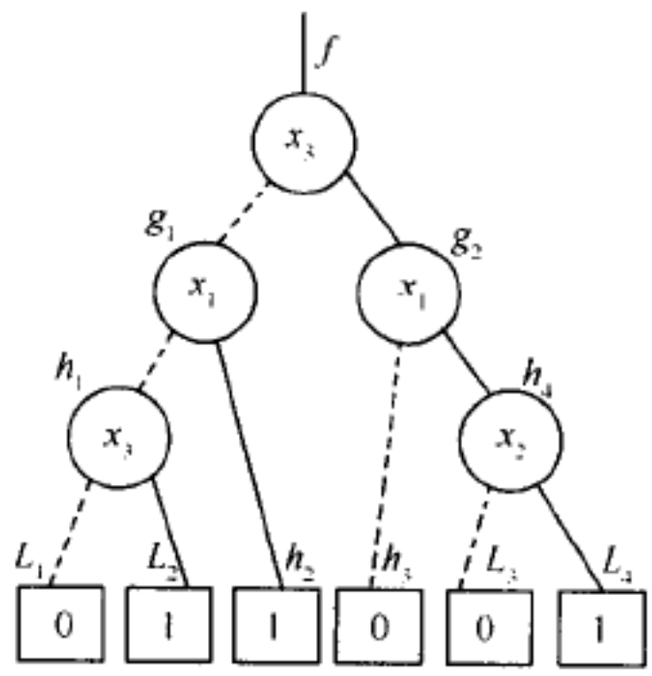


图 7.5　函数 f 在编序 $x_3 < x_1 < x_2$ 下的 BDD

其中，$g_1 = f\,|_{x_3=0} = x_1 + x_1x_2 + x_2 = x_1 + x_2$，　$g_2 = f\,|_{x_3=1} = x_1x_2$

$h_1 = g_1\,|_{x_1=0} = x_2$，$h_2 = g_1\,|_{x_1=1} = 1$，$h_3 = g_2\,|_{x_1=0} = 0$，$h_4 = g_2\,|_{x_1=1} = x_2$

$L_1 = h_1\,|_{x_2=0} = 0$，$L_2 = h_2\,|_{x_3=1} = 0$，　$L_3 = h_4\,|_{x_2=0} = 0$，　$L_4 = h_3\,|_{x_3=1} = 0$

对照图 7.4 和图 7.5，这两种二元判定图的结构是不相同的。因此，选取不同的变量顺序所构造出来的二元判定图也不同，这使二元判定图的运算和比较产生了很大的困难。为此，引入如下的编序二元判定图的概念。

定义 7.5　设对布尔函数 $f(x_1, x_2, \cdots, x_n)$ 的 n 个变量 $x_1, x_2, \cdots, x_n$ 的一个给定顺序为 $x_i < x_j < \cdots < x_k$。若函数 f 的二元判定图中，从根节点经任意路径到终节点所经历的变量顺序与所给定的顺序一致，则称此二元判定图为在该顺序下的编序二元判定图 OBDD(Ordered Binary Decision Diagram)。

例如，若变量的序取为 $x_1 < x_2 < x_3$，对函数 $f = x_1\bar{x}_3 + x_1x_2 + x_2\bar{x}_3$，图 7.4 所示 BDD 为满足编序二元判定图的定义，是一种 OBDD。不难看出，函数 $f = x_1\bar{x}_3 + x_1x_2 + x_2\bar{x}_3$ 的等价真值表形式就是表 7.1，因此图 7.3 的二元判定图仍为函数 f 的一种 OBDD 表示。显然，图 7.3 和图 7.4 的判定图的结构是不相同的，图 7.3 有 15 个节点，而图 7.4 有 11 个节点。

7.2　二元判定图的约简

由 7.1 节的讨论可知，对一个布尔函数在给定了变量的编序之后，在此编序下可能存在多个二元判定图能表示该布尔函数。显然，具有较少节点数的二元判定图在其操作中所用的时间和存储空间较少；对此，在表示逻辑函数的多个二元判定图中，是否存在一种二元判定图它含有的节点数是最少的?这种二元判定图的结构是否是唯一的?本节将讨论这些问题。为此，首先给出二元判定图中子图和同构的概念。

定义 7.6　对二元判定图 $G = (V, E)$ 中的任一节点 v，定义以 v 为根节点的子图 G' 如下：设 $G' = (V', E')$，$V' \subseteq V$，$E' \subseteq E$。这里 V' 包含了节点 v 和满足下列条件的节点 v_s：$v_s \in V$，且从节点 v 开始到节点 v_s 存在一条路径。E' 是 V' 中所有父节点到子节点的连接。

简单地说，在二元判定图中，以一个非终节点为根的子图是从该节点到所有终节点的

有向路径构成的子图，例如，图 7.6(a) 中以 A 点为根的子图如图 7.6(b) 所示。

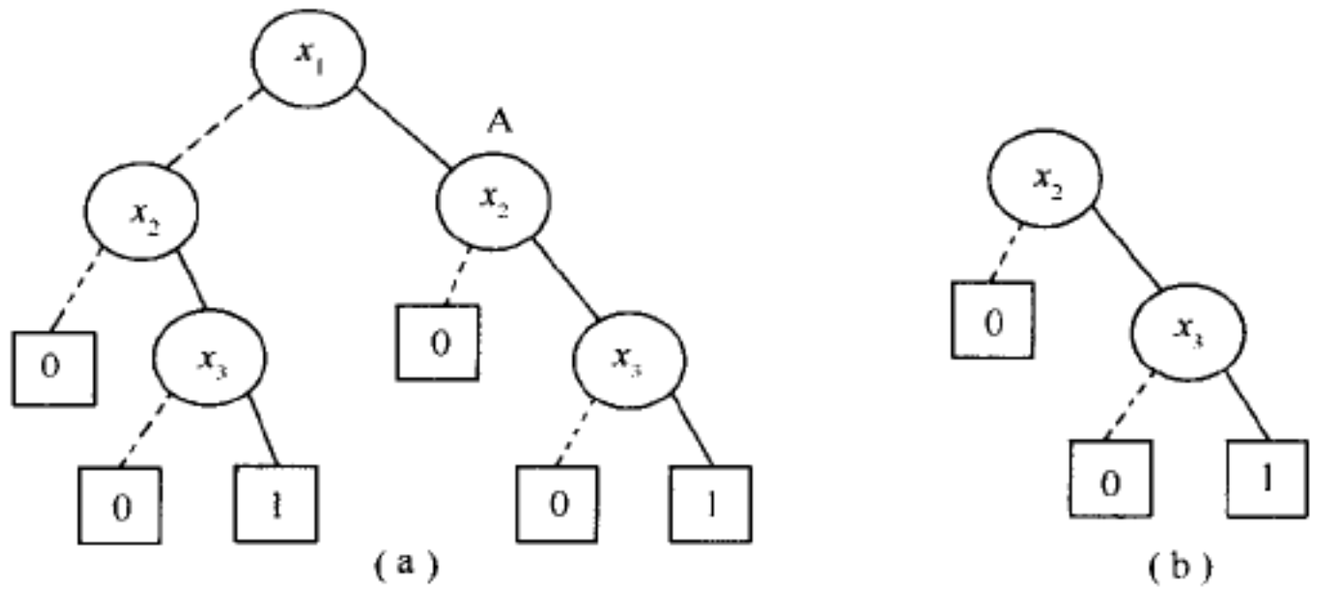


图 7.6　根子图

定义 7.7　对两个二元判定图 $G=(V,E)$ 和 $G'=(V',E')$，如果在节点集 V 与 V' 之间存在一种一一对应的映射 $\delta: V \to V'$，使得：对任意的 $v \in V$，在 V' 中存在 v' 满足 $\delta(v)=v'$，并且：

(1) 当 v 和 v' 都是终节点时，有 $\text{value}(v)=\text{value}(v')$；

(2) 当 v 和 v' 都是非终节点时，有 $\text{index}(v)=\text{index}(v')$，$\delta(\text{low}(v))=\text{low}(v')$，$\delta(\text{high}(v))=\text{high}(v')$。则称 G 和 G' 是同构的。

例如在图 7.3 中，在以 x_3 节点作为根节点的 4 个子图中，中间两个子图是同构的。由**定义 7.7**，可以得到引理 7.1。

引理 7.1　如果二元判定图 G_1 和 G_2 同构，且它们之间的一一映射关系为 δ，则对于 G_1 中的节点 v，以 v 为根的子图与以 $\delta(v)$ 为根的子图是同构的。

显然，在一个 BDD 中若存在多个同构子图，则会造成存储空间的浪费，这时只需要保留一个同构子图。为不影响该 BDD 所表示的逻辑函数，此时对相关节点的边应做修改，这样，可以定义一种更为精简的 BDD 形式。

定义 7.8　二元判定图 $G=(V,E)$ 如果满足：

(1) 任意两个节点 $v, v' \in V$，且 $v \neq v'$ 时，则以 v 和 v' 为根的两个子图不同构；

(2) 对任意非终节点 $v \in V$，成立 $\text{low}(v) \neq \text{high}(v)$。则称 G 是约简的二元判定图 (Reduced Binary Decision Diagram)。

由**定义 7.8**，可以使用如下的 3 条规则来把普通的二元判定图转化为约简二元判定图。

规则 1　删除相同的终节点。如果不同的终节点具有相同的值，则分别由这些终节点所构成的子图就是同构子图。这不满足**定义 7.8** 的条件(1)，因此，无论二元判定图中有多少个终节点，应只留下两个终节点 0 和 1，并把指向原来终节点的边转移到相应的 0 或 1 终节点上。

规则 2　删除同构子图。根据**定义 7.7**，如果非终节点 v 和 v' 具有 $\text{value}(v)=\text{value}(v')$、$\text{low}(v)=\text{low}(v')$ 和 $\text{high}(v)=\text{high}(v')$，则以 v 和 v' 为根节点的两个子图同构。根据**定义 7.8** 的条件(1)，应删除这两个节点 v 和 v' 之一，并把所有指向被删除节点的边转移到所保留的节点上。

规则 3　删除冗余节点。由**定义 7.8** 的条件(2)，如果非终节点 v 满足 $\text{low}(v)=\text{high}(v)$，即 v 的两条边指向同一节点，则可以删除节点 v，并将所有指向 v 的边转到 $\text{high}(v)$ 节点上。

下面以图 7.3 所示的二元判定图为例，说明约简的过程。

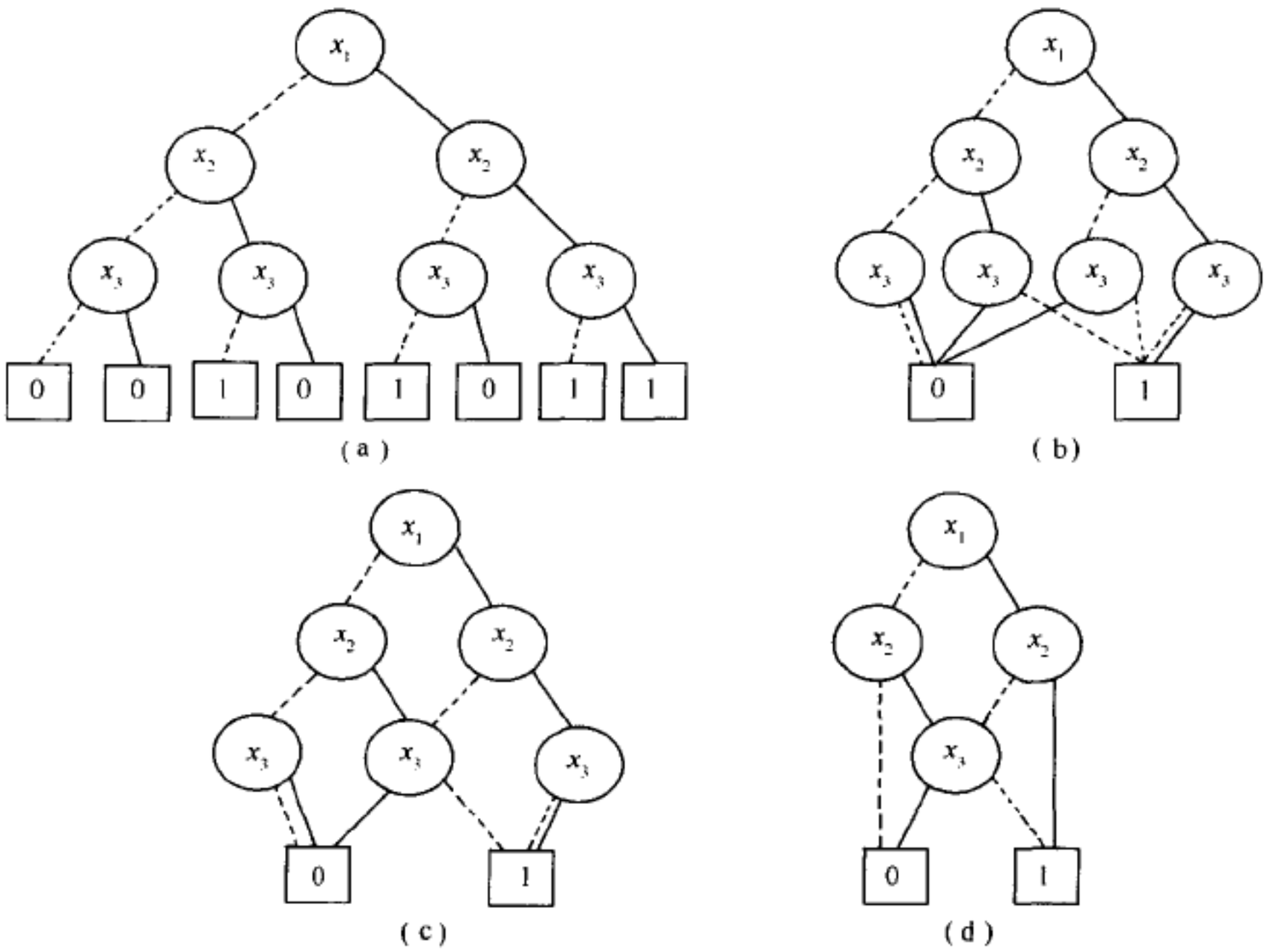


图 7.7　二元判定图的约简

(a) 函数 f 的 BDD 表示；(b) 删除相同节点；(c) 删除同构子图；(d) 删除冗余节点。

图 7.7(b) 是把图 7.7(a) 中的 8 个终节点删去 6 个，只保留了值为 0 和 1 的两个终节点。图 7.7(c) 中的 BDD 删去了判决变量为 x_3 的两个节点之一，这两个节点是图 7.7(b) 中判决变量为 x_3 的中间两个节点，因为以这两个节点为根节点的子图是同构的。图 7.7(d) 是在图 7.7(c) 的基础上删除去了两个冗余节点。这样，图 7.7(d) 为约简的二元判定图。

同样地，对图 7.4，采用如上过程进行约简，得到的 BDD 仍为图 7.7(d)。

由前面的讨论可知，图 7.3 和图 7.4 的二元判定图所表示的布尔函数相同，都为 $f = x_1\bar{x}_3 + x_1x_2 + x_2\bar{x}_3$，且符合编序 $x_1 < x_2 < x_3$，这两种二元判定图经过约简后得到了同一个二元判定图。这种情况其实具有一般性，即对任何逻辑函数 f，在给定的变量编序下，所有能表示 f 的约简二元判定图的形式是一样的，或者说都是同构的。

此外，在对二元判定图进行约简时，可能需要多次重复使用**规则 2** 和**规则 3**，这是因为其中的任意一个步骤都可能产生新的同构子图或冗余节点。例如，在图 7.8 中，对图 7.8(a) 中的两个同构子图删除一个，就产生了冗余节点，如图 7.8(b) 所示；在图 7.9 中，对图 7.9(a) 删除一个冗余节点，就产生了两个同构子图，如图 7.9(b) 所示。

定义 7.9　如果二元判定图 G 既是编序二元判定图(满足**定义 7.5**)，又是约简二元判定图(满足**定义 7.8**)，则称这种二元判定图为约简编序二元判定图，简称为 ROBDD(Reduced Ordered Binary Decision Diagram)。

图 7.7 中的图 7.7(d) 就是一个约简编序二元判定图的例子。对任意一个逻辑函数，可以先构造它的编序二元判定图，再经过约简得到其约简编序二元判定图。如下的**定理 7.1** 说明了编序二元判定图 ROBDD 的一个重要特性。

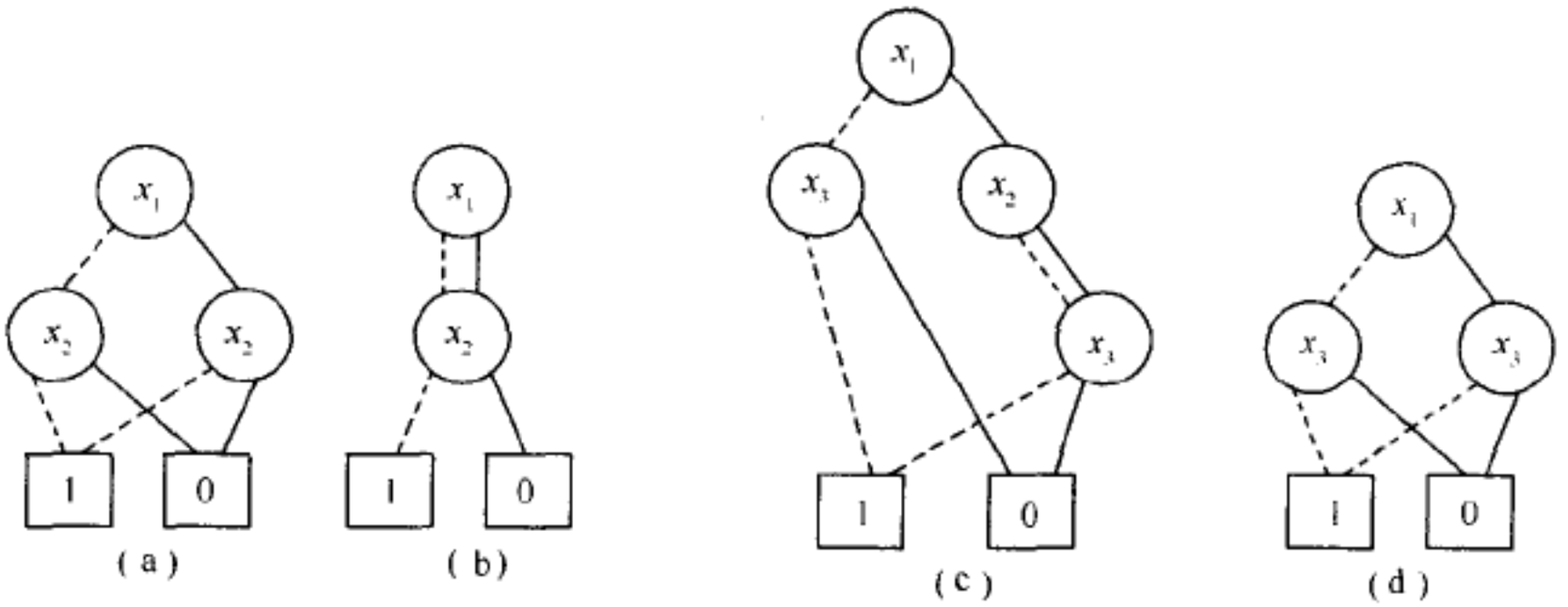


图 7.8　约简同构子图产生冗余节点　　图 7.9　删除冗余节点产生同构子图

定理 7.1　对于任意布尔函数，在给定的变量编序情况下，存在唯一一个 ROBDD 能表示该布尔函数，且它是能表示该布尔函数的所有二元判定图中所包含节点数最少的。

证明：令 $B=\{0,1\}$。设逻辑布尔函数为 $f(x_1,x_2,\cdots,x_n)$，$f:B^n\to B$。若一个具有根节点 u 的 ROBDD能表示布尔函数 f，则用 f^u 这种形式来代表该 ROBDD，这时就有 $f^u=f$。这里可以把 f^u 看成是该 ROBDD对应的布尔函数，因此自然就有 $f^u=f$。也可以直接简称为 u 是 f 的 ROBDD 表示。此外，设 f 的 ROBDD 表示所用的编序是 $x_1<x_2<\cdots<x_n$。

下面对 f 的参数个数 n 用归纳法。对 $n=0$ 时，仅存在两个布尔函数，也就是值为 0 或 1 的这两个常值函数 $f=0$ 或 $f=1$。由于包含非终节点的任何 ROBDD 一定不是常值函数，因此对于常值函数都有唯一的一个 ROBDD，即终端节点 0 和 1。

假定定理的结论对具有 n 个参数的所有布尔函数都成立，下面去证明对具有 $n+1$ 个参数的所有布尔函数也成立。设 $f:B^{n+1}\to B$ 是一个有 $n+1$ 个参数的函数。把第一个参数 x_1 的值固定为 0 和 1，定义如下的两个函数：

$$f_0=f_0(x_2,x_3,\cdots,x_n)=f_0(0,x_2,x_3,\cdots,x_n);$$
$$f_1=f_1(x_2,x_3,\cdots,x_n)=f_1(1,x_2,x_3,\cdots,x_n).$$

由于 f_0 和 f_1 都有 n 个参数，因此由归纳假定，存在根节点分别为 $u0$ 和 $u1$ 的 ROBDD 与 f_0 和 f_1 对应，并且这种 ROBDD 是唯一的，即有 $f^{u0}=f_0$，$f^{u1}=f_1$。下面考虑两种情况。

第一种情况。若 $u0=u1$，则 $f^{u0}=f^{u1}$。且 $f_0=f^{u0}=f^{u1}=f_1$。根据香农展开公式，有如下的等式成立，即

$$f(x_1,x_2,\cdots,x_n,x_{n+1})=\bar{x}_1\cdot f|_{x_1=0}+x_1\cdot f|_{x_1=1}=$$
$$\bar{x}_1\cdot f_0+x_1\cdot f_1 \tag{7-1}$$

因此，可以得到 $f_0=f^{u0}=f^{u1}=f_1=f$。因此 $u0=u1$，$u0$ 就为 f 的一种 ROBDD 表示。下面去说明这也是唯一的 ROBDD 表示。若此时 f 还有其他的一个 ROBDD，设该 ROBDD 的根节点 v 对应的变量编序为 x_1，此时有 $f=f^v$，因此 $f_0=f^v|_{x_1=0}=f^{\mathrm{low}(v)}$，$f_1=f^v|_{x_1=1}=f^{\mathrm{high}(v)}$。因为等式 $f_0=f^{v0}=f^{v1}=f_1$ 成立，故 v 的两个子节点 $\mathrm{low}(v)$ 和 $\mathrm{high}(v)$ 相同，这与 ROBDD 的约简规则相矛盾，因此 f 只有唯一的一种 ROBDD 表示。

第二种情况。若 $u0\neq u1$，由归纳假定则有 $f^{u0}\neq f^{u1}$。构造一个节点 v，使得节点 v 对应的变量编序为 x_1，$\mathrm{index}(v)=x_1$，且 $\mathrm{low}(v)=u0$ 和 $\mathrm{high}(v)=u1$。根据归纳假定有 $f^{u1}=f_1$ 和 $f^{u0}=f_0$ 成立，由式(7-1) 可得到 $f=f^v$。下面去说明节点 v 的唯一性。假定 w

是使得 $f^w = f$ 的另一个节点，显然 f^w 必须依赖变量 x_1，即 $f^w|_{x_1=0} \neq f^w|_{x_1=1}$，且 $\text{index}(w) = x_1$。从 $f^w = f$ 可得到：$f^{\text{low}(w)} = f_0 = f^{u0}$ 且 $f^{\text{high}(w)} = f_1 = f^{u1}$。由归纳假定，因此 $\text{low}(w) = u0 = \text{low}(v)$，$\text{high}(w) = u1 = \text{high}(v)$。依据 ROBDD 的约简规则，得到 $w = v$。故节点 v 是唯一的。

定理证毕。

定理 7.1 说明约简编序二元判定图是布尔函数的范式，或者说是布尔函数的正则表达式，即每个布尔函数可由ROBDD唯一表示。这里“唯一”的含义是在“同构”的意义下来说的，也就是，对任意逻辑函数，在给定的变量编序情况下，所有能表示该函数的约简编序二元判定图都是同构的。

在实际应用中，二元判定图的这个性质是非常重要的，例如，为了检查两个布尔函数是否等价，可以通过检查与它们对应的 ROBDD 是否同构来进行。与二元判定图相关的大多数工作都是基于这种 ROBDD 形式，在本书的后面内容中，若不加以特别说明，所说的二元判定图 BDD 都是指约简编序二元判定图 ROBDD。

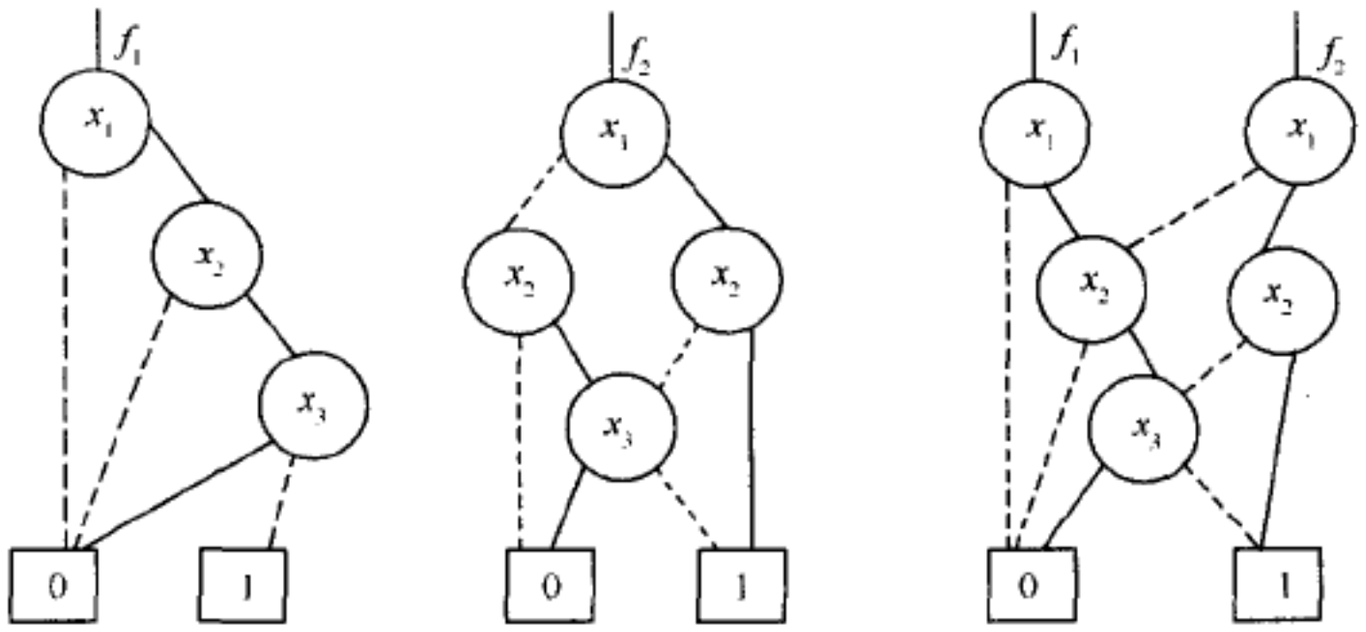
虽然 ROBDD 是节点数最小的编序二元判定图，但在二元判定图的许多应用中，需要对多个逻辑函数同时构造出 ROBDD 表示，然后在这些 ROBDD 之间进行各种操作。如果对每个函数都用一个单独的判定图来表示，所花费的存储空间会较大。为了在表示多个逻辑函数时节省更多的存储空间，因此引入了共享二元判定图。

定义 7.10 共享二元判定图是具多个根节点的有向无环图 $G = (V, E)$，它满足如下两个条件：

(1) 对 V 中的任意一个节点 v，以 v 为根节点的子图是约简编序的二元判定图。

(2) 在 G 中不存在同构子图，即不存在两个节点 $u, v \in V$，使得分别以 u 和 v 为根节点的两个子图同构。

把共享二元判定图简称为 SBDD(Shared Binary Decision Diagram)。图 7.10 是共享二元判定图的一个例子。



(a) (b) (c)

图 7.10 共享二元判定图

图 7.10(a) 和图 7.10(b) 分别是函数 $f_1 = x_1x_2\bar{x}_3$ 和 $f_2 = x_1\bar{x}_3 + x_1x_2 + x_2\bar{x}_3$ 的 ROBDD表示，图 7.10(a) 有 5 个节点，图 7.10(b) 有 6 个节点，这两个图总共有 11 个节点。图 7.10(c) 是这两个函数 f_1 和 f_2 的共享二元判定图表示，它把两个 ROBDD 放到了一个有向无环图中。并根据约简的规则对该图进行了简化，删除了同构的子图，使得一些节点

可以被共享。结果只用了 7 个节点所构成的判定图就能同时表示函数 f_1 和 f_2。

共享二元判定图仍然是逻辑布尔函数的一种范式表示，这是因为对一个逻辑函数而言，在共享二元判定图中都只有唯一的根节点与之对应。

共享二元判定图不单在表示多个函数时减少了节点数目，降低了对存储空间的要求，它还对一些运算的性能有很大提高。例如，下面分析一下在 ROBDD 和 SBDD 表示下判定两个布尔函数相等的时间复杂度。假设函数 f_1 和 f_2 的 ROBDD 表示分别为 G_1 和 G_2，要验证 f_1 是否与 f_2 的逻辑功能相同，就是要验证 G_1 和 G_2 是否同构。在 ROBDD 表示时，所需的时间复杂度为 $O(\min(|G_1|,|G_2|))$，这里 $|G_1|$ 和 $|G_2|$ 分别表示图 G_1 和 G_2 的节点数；而在 SBDD 表示的情况下，函数 f_1 和 f_2 逻辑功能相同的唯一条件是 f_1 和 f_2 都由 SBDD 的同一子图表示，即是否对应于同一根节点，这种验证操作是常数时间复杂度的。

至此，对二元判定图的一些基本特点做了介绍。二元判定图的种类及形式近些年仍在不断发展之中。

7.3 二元判定图的程序实现

在 7.2 节中，讨论了二元判定图的一些基本概念，在本节中，讨论它的程序实现的几种典型方法。这些方法是以伪 Pascal 语言来描述的。

首先把节点定义为如下的一条记录：

```
type   vertex = record
         low,high:vertex;
         index:1…n+1;
         val:(0,1,X);
         id:integer;
         mark:Boolean;
       end
```

非终节点和终节点都由同一类记录来表示。对于节点 v，它的记录中的各个域的值由表 7.2 按节点的类型给出。

表 7.2 节点记录中各个域的取值

域	终节点	非终节点
low	null	low(v)
high	null	high(v)
index	$n+1$	index(v)
val	value(v)	x

在节点记录中的 id 和 mark 的作用是算法所用的辅助信息。id 是一个整数标志，取值为 1 至 $n+1$ 之间的一个整数，可以用它来唯一地对应于图中的一个节点。用 mark 来标记在图的遍历过程中已被访问过的节点。

7.3.1 BDD 的遍历

遍历就是从 BDD 的根节点出发，按照一定的搜索方法对图中的所有节点做一次访问的过程。为了说明一个节点是否已被访问，就用 mark 的值来说明。开始之前，把所有节点的 mark 域的值都设置为真(true)。当一个节点被访问之后，就将它对应的 mark 域的值设

置为假(false)。因此，当所有节点的 mark 域的值都为假时，则遍历过程结束。下面是对 BDD 进行遍历的程序。

```
procedure   traverse(v:vertex)
begin
    v.mark:= not v.mark;
    ———对节点 v 进行某些操作
    if v.index≤n        // 非终节点
    then begin
        if v.mark≠v.low.mark   then   traverse(v.low);
        if v.mark≠v.high.mark   then   traverse(v.high);
    end
end
```

该过程由根节点作为参数来调用，然后访问以两个子节点为根的子图，这样递归地进行就可以访问图中的所有节点。在一个节点被访问时，就对其 mark 域的值求反。这样在以后判断一个节点是否被访问时，便可根据该节点的 mark 域的值是否求反来决定。

7.3.2 BDD 约简的程序实现

一个 BDD 可能含有冗余节点和同构子图，因此需要对其进行约简，以减小节点数和缩短图中的路径长度。约简是按下列两个化简规则进行的：

规则 1(合并规则)　两个同构子图可以合并。

规则 2(删除规则)　对一个节点，若它的两个分支(子节点)指向同一节点，则可以删除该节点。

规则 1 就是删除同构子图，**规则 2** 就是删除 BDD 中的冗余节点。

约简算法将一个函数的任意 BDD 转换成表示同一函数的一个约简 BDD。在从终节点到根节点的处理过程中，对每一个唯一的子图的根赋予一个唯一的整数标志。也就是说，对每个节点 v 赋予标志 id(v)；对于两个节点 u 和 v，当且仅当 $f^u = f^v$ 时，才有 id(u) = id(v)。给出此标志后，约简算法对每个标志都以一个节点来构造图。

从终节点到根节点，可用下面的方法来标志节点。首先，当且仅当两个终节点具有相同的值时，此两个节点有相同的标志。现在假定索引比 i 大的所有终节点和所有非终节点都已经标记，在处理节点索引为 i 的节点时，当且仅当下列两个条件之一满足时，节点 v 的 id(v) 等于已标记过的某个节点的标志。

第一，如果 id(low(v)) = id(high(v))，则节点 v 冗余，并且将 id(v) 设置为 id(low(v))，即 id(v) = id(low(v))。

第二，如果有某已标记的节点 u 有满足：index(u) = i，id(low(v)) = id(low(u))，id(high(v)) = id(high(u))，则以此两个节点 v 和 u 为根的子图是同构的，并且记为 id(v) = id(u)。

下面是对 BDD 进行约简的程序。它的参数是节点 v，即待约简的 BDD 的根节点；它的返回值是一个节点，即经过约简之后的 BDD 的根节点。

```
function   Reduce(v:vertex):vertex
```

```
    var subgrapg:array[1··|G|]of vertex;
    var vlist:array[1··n+1]  of  list;
begin
    put each vertex u on list vlist[u. index];                    // 把节点 u 放入表中
    nextid:=0;
    for i:=n+1 downto 1 do
    begin
        Q:=∅;                                                     // 把 Q 设置为空集
    for each u in vlist [i] do
        if  u. index=n+1                                          // 当为终节点
        then add <key,u>to Q where key=(u. value)
        else if u. low. id=u. high. id  then u. id=u. low. id;   // 冗余节点
            else add<key,u> to Q where key=(u. low. id,u. high. id);
    sort elements of Q by keys;                                   // 用键的值对 Q 中的元素进行分类
    oldkey=(-1,-1);                                               // 不匹配的键
    for each<key,u>in Q removed in order do
        if  key=oldkey  then  u. id=nextid;                       // 匹配存在的节点
          else  begin
                nexitid:=nextid+1;  u. id=nextid;  subgraph[nextid]:=u;
                u. low:=subgraph[u. low. id];u. high:=subgraph[u. high. id];
                oldkey:=key;  end;
      end;
    return(subgraph[v. id]);
  end.
```

如上约简程序的实现过程如下：首先，根据节点的索引把图中的节点列为若干个表，这种处理可以由类似于遍历 traverse()的方法来实现，在访问一个节点时就把此节点加到相应的表中。然后，从含有终节点的表开始对这些表进行处理，一直到处理完含有根节点的表。对表中的每个节点，按如下方法生成一个键(key)。若为终节点就生成 value 型的键，若为非终节点就生成(lowid，highid)型的键，这里 lowid＝id(low(v))，highid＝id(high(v))。如果 lowid＝highid，则 id(v)＝lowid。余下的节点根据其键值来分类，然后对每类表进行处理，对于具有相同键值的所有节点赋予一个标志。之后，对每个唯一标志选择一个节点记录，并且在以该标志作为索引的数组中存储一个指向这个顶点的指针。这些选择的节点最终将形成一个约简图。用这种方法去修改节点的记录以使得节点的两个子节点在约简图中、在整个过程结束时返回约简图的根节点。注意，在约简过程中赋给一个节点的标志在下面的其他过程中可看作唯一标志。

7.3.3　由布尔表达式经运算生成 BDD

运算提供了根据布尔表达式或逻辑电路的结构生成它们所对应的 BDD 的方法。它以两个函数 f_1 和 f_2 所对应的BDD为基础，使用一个布尔操作＜op＞，去生成函数 f_1＜op＞

f_2 所对应的 BDD。函数 $f_1 < \text{op} > f_2$ 的定义为

$$[f_1 < \text{op} > f_2](x_1, x_2, x_3, \cdots, x_n) = f_1(x_1, x_2, \cdots, x_n) < \text{op} > f_2(x_1, x_2, \cdots, x_n)$$

这里布尔操作 $< \text{op} >$ 可以取常见的逻辑与、或、非、异或，对一个函数取反(用 $f \oplus 1$)以及其他的各种运算。

算法的处理过程是由对两个待处理的图的根节点开始向下进行，在两个待处理的图的分支节点处获得结果图中的节点。在进行过程中，反复地使用如下的香农展开式：

$$f_1 < \text{op} > f_2 = \overline{x}_i \cdot (f_1 |_{x_i=0} < \text{op} > f_2 |_{x_i=0}) + x_i \cdot (f_1 |_{x_i=1} < \text{op} > f_2 |_{x_i=1})$$

为了用以 v_1 和 v_2 为根节点的两个布尔函数对应的 BDD 图进行运算，必须考虑如下几个问题：

(1) 假设 v_1 和 v_2 为都为终节点，则结果图也是由终节点组成，此图的终节点的值为 $\text{value}(v_1) < \text{op} > \text{value}(v_2)$。

(2) 假设 v_1 和 v_2 两个节点中至少有一个为非终节点，如果 $\text{index}(v_1) = \text{index}(v_2) = i$，则生成的节点 u 的索引为 i，并且对 $\text{low}(v_1)$ 和 $\text{low}(v_2)$ 递归地应用此算法并生成以 $\text{low}(u)$ 为根的子图；对 $\text{high}(v_1)$ 和 $\text{high}(v_2)$ 递归地应用此算法并生成以 $\text{high}(u)$ 为根的子图。

(3) 假设 $\text{index}(v_1) = i$，而 v_2 为非终节点或 $\text{index}(v_2) > i$，则以 v_2 为根的子图所表示的函数(设为 f_2)与 x_i 无关，即 $f_2 |_{x_i=0} = f_2 |_{x_i=1} = f_2$。这时则可生成一个索引为 i 的节点 u，并递归地在 $\text{low}(v_1)$ 和 v_2 上应用本算法并生成以 $\text{low}(u)$ 为根的子图；在 $\text{high}(v_1)$ 和 v_2 上递归地应用本算法并生成以 $\text{high}(u)$ 为根的子图。如果 v_1 和 v_2 两个节点的关系与此相反，也可应用类似地方式来解决。

一般地，用如上算法求得的图不是约简的图，因此应对所得图进行约简。此外，该算法的时间复杂度将是指数复杂型，即可以把时间复杂度写成 n 的一种指数函数形式，例如 a^n，这里 a 是大于 1 的正常数。这是因为，对任一个待处理图的每次调用如果为非终节点的话，则要产生两次递归的调用。

因此，为了降低时间复杂度，可采用下面的两个规则。

规则 1　本算法对给出的每对子图的计算不超过一次。可用如下方法来完成这一点：首先，建立一张表，该表的每一项的形式为 (v_1, v_2, u)，其含义是由以 v_1 和 v_2 为根节点的子图用本算法求得以 u 为根节点的结果图。然后，在对一对节点 w_1 和 w_2 用本算法操作之前，首先在表中检查是否含有这两个节点所对应的这一项 (w_1, w_2, w)。如果有，则可马上返回结果即返回以 w 为根节点的结果图；否则，就用前面所述方法进行处理，并在表中新增一项。

这个规则可将算法的时间复杂度降低到 $(|G_1| \cdot |G_2|)$，这里 G_1 和 G_2 分别代表用本算法进行操作 $< \text{op} >$ 的两个初始 BDD 图。这个规则说明了怎样使用共享子图以使算法达到较好的效果。这里把在表中找到了对应的一项称为“命中”。如果两个初始 BDD 图有许多共享子图，则表的命中率就可以很高。从实验中发现，表的命中率在 40% ～ 50%。当表的命中率为 50% 时，算法的速度会大大地超过人们原先所期待的两倍 。在这里，算法避免了在构建以 u 为根节点的子图时潜在的递归调用。

规则 2　对两个节点应用此算法时其中一个节点如 v_1 为终节点。在此时，可以把

value(v_1)作为是一个控制值。即对所有的a,要么 value(v_1)$<$op$>a=1$,要么 value(v_1)$<$op$>a=0$。例如,1 为两个变量 OR 运算的控制值;0 为两个变量 AND 运算的控制值。在这种情况下,并不需要做进一步地计算便可方便地生成对应的终节点值。此规则虽然对算法的时间复杂度没有改善,但在许多实际应用中却相当有用。在实验中发现有 10% 的时间会发生这种情况。

算法的具体程序实现是写成了如下的函数 Apply():

```
function  Apply(v1,v2:vertex,<op>:operator):vertex
    var T: array[1..|G1|,1..|G2|]of vertex;
    function apply-step(v1,v2:vertex):vertex;   // 下面递归调用函数 apply-step
    begin
      u:=T[v1.id,v2.id];
      if u≠NULL then return(u);                  // u 已计算过
      u:=new vertex record;   u.mark:=false;
      T[v1.id,v2.id]:=u;                         // 在表中增加一个节点
      u.value:=v1.value<op>v2.value;
      if u.value≠X then begin                    // 创建终节点
        u.index:=n+1;u.low:=NULL;u.high:=NULL;   end
      else begin                                 // 创建非终节点,并做进一步处理
        u.index:=min(v1.index,v2.index);
        if v1.index=u.index then begin vlow1:=v1.low;vhigh1:=v1.high; end
                            else begin vlow1:=v1;   vhigh1:=v1;   end
        if v2.index=u.index then begin vlow2:=v2.low;vhigh2:=v2.high; end
                            else begin vlow2:=v2; vhigh2:=v2; end
          u.low:=apply-step(vlow1,vlow2);
          u.high:=apply-step(vhigh1,vhigh2); end
        return(u);   end
begin                                            // 主程序
    Initialize all elements of T to NULL;   // 初始化表 T 中的所有元素
    u:=apply-step(v1,v2);
    return(Reduce(u));                           // 对根节点为 u 的图进行约简
end
```

在实际应用中,由于在应用 Apply 算法之后得到的 BDD 一般并不是约简的,因此还需要执行约简算法 Reduce 以获得约简二元判定图。

例如,建立图 7.11(a)中电路的原始输出 f 所对应的 BDD 如下。首先,对电路的原始输入建立 BDD;其次,从原始输入到原始输出,利用 apply 和 reduce 逐级计算电路内部节点的 BDD 表示,直至到达电路的原始输出。

图 7.11(b)、图 7.11(c) 和图 7.11(d) 分别是信号线 a、b 和 c 所对应的 BDD;图 7.11(e)、图 7.11(f) 和图 7.11(g) 分别是信号线 d、e 和 f 所对应的 BDD;图 7.11(h) 为电路的原始输出信号线 g 对应的 BDD。

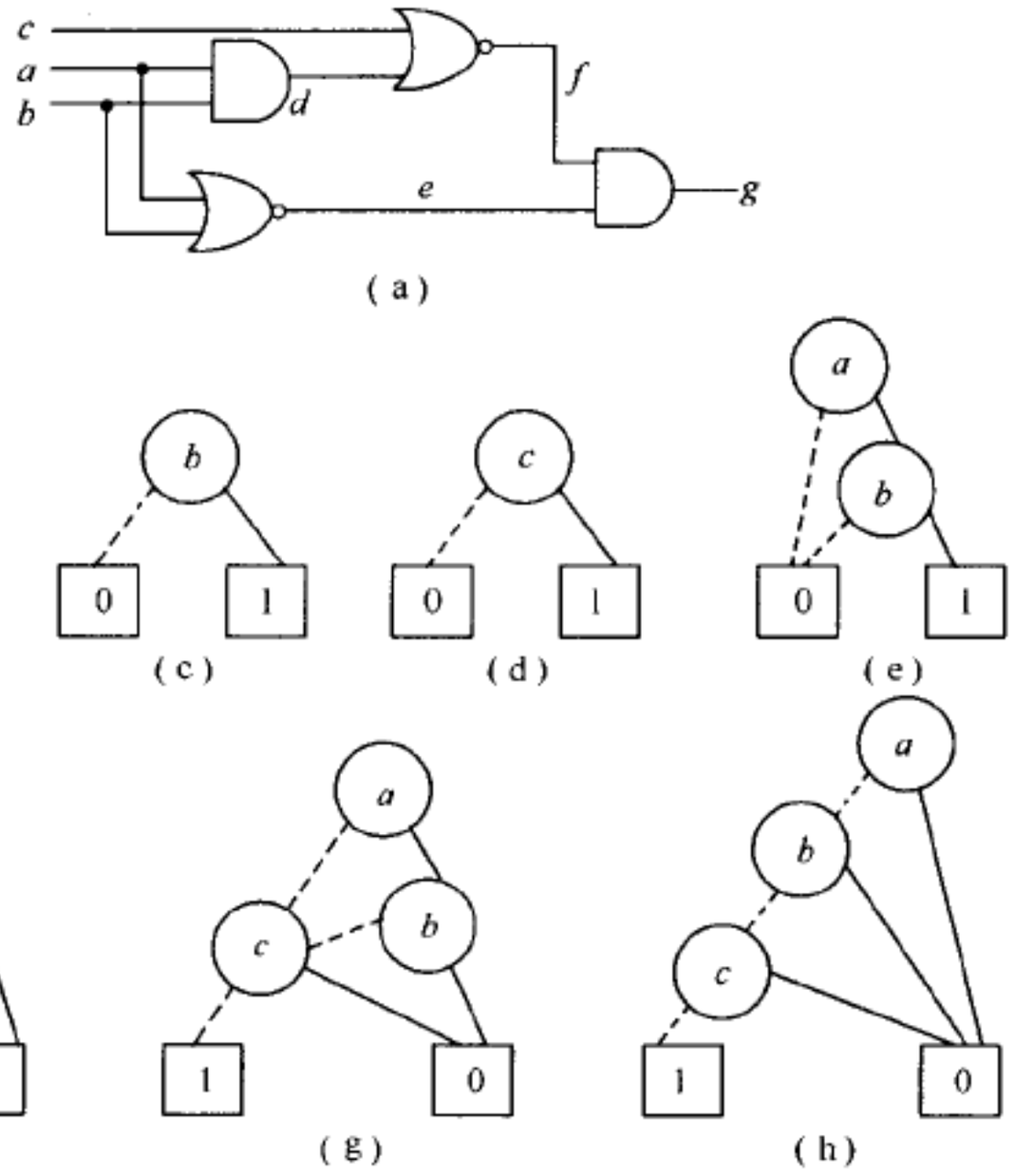


图 7.11　计算电路的原始输出所对应的 BDD

7.3.4　生成 BDD 的 ITE 算法

布尔函数的基本运算有：一元运算(求函数的“补”) 和二元运算(求两个函数的“与”、“或”、“异或” 等)。ite 运算符能用一种统一的形式来表示布尔函数的各种基本运算，它是一个三元逻辑运算符。ite 是 if-then-else 的简称。对 3 个布尔函数 f、g、h，ite 运算符定义为

$$\text{ite}(f,g,h) = f \cdot g + \overline{f} \cdot h \tag{7-2}$$

布尔函数所有的一元运算和二元运算都可以用 ite 运算符来实现，例如：

$\overline{f} = \text{ite}(f,0,1)$；$f \cdot g = \text{ite}(f,g,0)$；$f + g = \text{ite}(f,1,g)$；$f \oplus g = \text{ite}(f,\overline{g},g)$

式中：符号 $\oplus$ 表示异或。

设布尔函数 f、g、h 各自的 BDD 表达为 F、G、H。在计算 $\text{ite}(F,G,H)$ 时不能直接使用 ite 的定义式(7－2)，对此可使用如下方法。当函数 $f(x_1,x_2,\cdots,x_n)$ 用二元判定图 F 表示时，香农展开式为

$$F = v \cdot F_v + \overline{v} \cdot F_{\overline{v}} \tag{7-3}$$

其中 $v \in \{x_1,x_2,\cdots,x_n\}$，$F_v$ 和 $F_{\overline{v}}$ 是 $F|_{v=1}$ 和 $F|_{v=0}$ 的简写，分别对应于 F 在给定的输入变量编序下，变量 v 取 1 和取 0 后产生的 BDD。根据式(7－3)，可得到如下的递归公式：

$$\text{ite}(F,G,H) = \text{ite}(v,\text{ite}(F_v,G_v,H_v),\text{ite}(F_{\overline{v}},G_{\overline{v}},H_{\overline{v}})) \tag{7-4}$$

这是因为：

$$\text{ite}(F,G,H) = F \cdot G + \overline{F} \cdot H = v \cdot (F \cdot G + \overline{F} \cdot H)_v + \overline{v} \cdot (F \cdot G + \overline{F} \cdot H)_{\overline{v}} =$$
$$v \cdot (F_v \cdot G_v + \overline{F}_v \cdot H_v) + \overline{v} \cdot (F_{\overline{v}} \cdot G_{\overline{v}} + \overline{F}_{\overline{v}} \cdot H_{\overline{v}}) =$$
$$\text{ite}(v,\text{ite}(F_v,G_v,H_v),\text{ite}(F_{\overline{v}},G_{\overline{v}},H_{\overline{v}}))$$

由式(7－4) 计算 $\text{ite}(F,G,H)$ 的 BDD 时分为如下两步：第一步，求 F 和 G 的香农展开

因子；第二步，在求出 ite(F_v,G_v,H_v) 和 ite($F_{\bar{v}}$,$G_{\bar{v}}$,$H_{\bar{v}}$) 的结果 BDD 后，由 ite(v,ite(F_v,G_v,H_v),ite($F_{\bar{v}}$,$G_{\bar{v}}$,$H_{\bar{v}}$)) 生成相应的 BDD 图。

下面首先讨论计算 F 和 G 的香农展开因子的方法。假定 F 具有如下形式：$F=(v,G,H)$，即 F 的根节点的判决变量为 v，其 1 子节点为 G，0 子节点为 H。对变量 w 而言，计算 F_w 和 $F_{\bar{w}}$ 在一般情况下比较复杂，有时甚至需要遍历 F 的全部节点。但是在如下的假定条件：变量 w 的编序小于变量 v 的编序的情况下（记为 $w<v$），计算过程却很简单，即当 $w<v$ 时，有 $F_w=F_{\bar{w}}=F$；当 $w=v$ 时，$F_w=G$，$F_{\bar{w}}=H$。称 $w<v$ 为快速香农展开条件。这种条件在 ite 运算中很容易得到满足。

对计算 ite(v,ite(F_v,G_v,H_v),ite($F_{\bar{v}}$,$G_{\bar{v}}$,$H_{\bar{v}}$)) 的 BDD，假设 F、G、H 的根节点的判决变量分别为 v_1、v_2、v_3，从 $x_1,x_2,\cdots,x_n$ 中选取一个变量 v，使得 $v<v_i$ 成立，$i\in\{1,2,3\}$。这时就满足快速香农展开条件，因此首先计算 F_v,G_v,H_v 和 $F_{\bar{v}}$,$G_{\bar{v}}$,$H_{\bar{v}}$ 然后计算 $T=$ ite(F_v,G_v,H_v) 和 $E=$ ite($F_{\bar{v}}$,$G_{\bar{v}}$,$H_{\bar{v}}$)；最后由式(7-4) 构造出 ite(F,G,H) 的 BDD，此时 ite(F,G,H) = ite(v,T,E)。

于是得到如下结论：设函数 g_1、g_2、g_3 的 BDD 分别为 G_1、G_2、G_3，如果 G_1、G_2、G_3 中节点判决变量的编序互不交叉且 G_1 的节点编序最小，则 ite(G_1,G_2,G_3) 可以按如下方式生成：把 G_1 中所有指向终节点 1 的边重定向到 G_2 的根节点，把 G_1 中所有指向终节点 0 的边重定向到 G_3 的根节点。

由如上的讨论可知，ite 运算符能有效地使用递归来进行运算，其条件是：对 ite(F,G,H)，选择一个变量 v，使 v 小于或等于 F、G、H 中所有节点的序，由于选择小于是无效的，所以应该选择 v 等于 F、G、H 中节点的编序的最小者，即选取 F、G、H 的根节点的编序的最小者。此时，ite 迭代公式可以表示为

$$\text{ite}(F,G,H)=\text{ite}(v,\text{ite}(F_v,G_v,H_v),\text{ite}(F_{\bar{v}},G_{\bar{v}},H_{\bar{v}}))$$

由 ite 算符的定义可得这个迭代公式的结束条件为

$$\text{ite}(F,1,0)=\text{ite}(1,F,G)=\text{ite}(0,G,F)=F$$

下面是使用 ite 算符的两个例子。

【例 7.1】 在变量不交叉时的 BDD 构造。在图 7.12 中，BDD 图 G_1、G_2 和 G_3 的变量集分别为 $\{x_1\}$、$\{x_2,x_3\}$ 和 $\{x_4,x_5\}$，这 3 个变量集的交集为空集。这里 G_1 和 G_2 所表示的函数分别为 $\bar{x}_1$ 和 $\bar{x}_2\bar{x}_3$。按前面的讨论，可以容易地构造出 ite(G_1,G_2,G_3) 的 BDD，如图 7.12 所示。

【例 7.2】 在一般情况下的 BDD 构造。图 7.13 中，F 的变量集与 G 的变量集有交叉，交集为 $\{a\}$；F 的变量集与 H 的变量集也有交叉，交集为 $\{b\}$。这时 ite(F,G,H) 的计算就复杂些。

在图 7.13 中，把 BDD 图 F 的 $a=0$ 的分支用根节点为 B 的子图来表示；把图中 $a=1$ 的分支用根节点为 C 的子图来表示；H 的 $b=0$ 的分支用根节点为 D 的子图来表示。这里就有

$$\begin{aligned}\text{ite}(F,G,H)&=(a,\text{ite}(F_a,G_a,H_a),\text{ite}(F_{\bar{a}},G_{\bar{a}},H_{\bar{a}}))=(a,\text{ite}(1,C,H),\text{ite}(B,0,H))=\\&(a,C,\text{ite}(B,0,H))=(a,C,(b,\text{ite}(B_b,0_b,H_b),\text{ite}(B_{\bar{b}},0_{\bar{b}},H_{\bar{b}})))=\\&(a,C,(b,\text{ite}(1,0,1),\text{ite}(0,0,D)))=\\&(a,C,(b,0,D))\end{aligned}$$