第3章 系统建模

验证系统正确性的首要步骤是建立需求工程，即刻画系统拥有的性质，并理解这些性质在何种抽象层次可以判定为真或假。需求工程通常始于非形式化的规约、模型和流程，然后才是可以应用于验证算法的形式化的规约和模型。所以本章讨论形式化模型，第 4 章将介绍由时序逻辑表示的形式化规约。假设我们想确定某并发程序无死锁，不仅需要精确地刻画无死锁规约，而且也要将系统的并发行为表示得足够充分，比如原子运算概念和针对调度策略的假设。

因此，一旦确定哪些性质重要之后，就要立刻构建系统的形式化模型。为了令自动化的验证有效，模型应该反映出系统中影响这些性质的所有层面。此外，模型也应该分离出与所刻画性质无关的细节，从而使验证简化。所以形式化模型既要表示出某种抽象层次下的系统实况，包含足够多的待验证性质的细节，又要足够简单，以利于验证算法执行。比如在建模同步数字电路时，常常考虑逻辑门和布尔值而不是真实电压值。又比如在建模通信协议时，我们可能更关注信息交换机制，而忽略信息文本的内容，以及特定设备和操作系统下(信息交换)的实现细节。但需注意，建模过程中的“忽略”并不意味着忽略细节，而是映射着约束，这种约束蕴含了验证结果适用于特定的系统实况。

很多数字电路和程序都是反应系统[369]，这种系统频繁地与环境交互，通常不会终止，所以就不能仅使用“输入–输出”行为，而要根据其内部状态来理解并建模这种系统。因此，需要考虑的最重要的反应系统特征就是状态。状态是系统描述的快照或者实例，反映了某个特殊时间点上的系统变量值。为了分析系统行为，也要知道系统动作导致系统状态如何变化，这种变化可以通过将动作发生前后的系统状态关联起来进行描述。状态对确定了系统的一次变迁，系统的行为则根据系统的变迁来定义。路径就是一个(极可能无限长的)状态序列，序列中的每一个状态都根据前态和变迁得出。

可以使用一种称为 Kripke 结构的状态变迁图来描述反应系统的行为，这种结构包含了状态集合、状态间变迁的集合及标记函数，此函数将状态满足的性质集合标记到状态上。Kripke 结构上的路径对应了系统行为，尽管结构非常简单，却充分表达出反应系统推导时序行为所需的细节。本书将在第 4 章讨论刻画反应系统时序行为的方法。

通常而言，真实系统的描述方式是使用如 C 和 Java 的程序设计语言或者使用如 Verilog 和 VHDL 的硬件描述语言。由于语言的丰富性和系统类型的多样性(如同步和异步电路、基于共享变量的程序和基于消息传递的程序)，需要采用统一的形式化机制进行系统建模。本书采用一阶逻辑。将程序翻译为一阶逻辑的描述非常直观，同样，也可以直接将一阶逻辑公式构造成 Kripke 结构。

本章后续内容将首先给出 Kripke 结构的形式化定义，其次介绍一种从一阶逻辑公式表示

的并发系统中提取此结构的方法，最后针对不同程序设计语言描述的系统，展示采用一阶逻辑公式建模的方法。

3.1 变迁系统和 Kripke 结构

一切将始于变迁系统的形式化。变迁系统 T 是一个三元组(S, S_0, R)，其中

1. S 是状态集合。
2. $S_0 \subseteq S$ 是初始状态集合。
3. $R \subseteq S \times S$ 是变迁关系。

要求变迁关系 R 必须是左完全的(left total)，即对于每一个状态 $s \in S$，存在后继状态 $s' \in S$，并满足 $R(s, s')$。

一条从某状态 $s \in S$ 出发的有限路径是一个序列 $s_0, s_1, \cdots, s_n$，其中 $s_0 = s$，且对于所有 $0 \leqslant i < n$，都有 $R(s_i, s_{i+1})$为真。与之类似，一条无限路径即为无限状态序列 $s_0, s_1, \cdots$，且对于所有 $i \geqslant 0$，都有 $R(s_i, s_{i+1})$为真。在谈及路径时，既可能指有限路径，也可能指无限路径，因此本书的路径概念同图论中广泛采用的路径概念一致。要求变迁系统中的 R 是左完全的，每条有限路径都可以扩展为无限路径，所以也可以将有限路径看作无限路径的前缀。

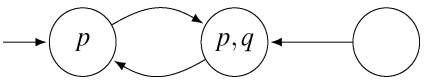
除了第 13 章、第 14 章、第 18 章和第 20 章，本书的其他部分都假设 S 是有限的。第 13 章、第 14 章、第 18 章和第 20 章中虽然假设 S 是无限的，但是这几章的典型方法都可以从有限状态系统归纳出。在本章后续部分还将谈及此话题。

可以定义状态标签集合来观察一些特殊状态，这些标签称为原子命题(atomic proposition)，包含所有原子命题的集合记为 AP。添加了上述状态标签的变迁系统称为 Kripke 结构，记为 M，由五元组 $M = (S, S_0, R, AP, L)$ 构成：

1. S、S_0 和 R 的定义如前所述。
2. AP 是原子命题集合。
3. $L: S \to 2^{AP}$ 是一个函数，将各状态上成真的所有原子命题标记到此状态上。

我们有时也不会关注初始状态集合 S_0，此时可以从上述定义中删除这个状态集合。

通常会将 Kripke 结构描述为有向图，Kripke 结构的状态形成了有向图的节点，状态间的变迁定义出节点间的(有向)边，状态标签将标记到节点中或者节点附近。例如，Kripke 结构 $S = \{s_1, s_2, s_3\}$，$S_0 = \{s_1\}$，$R = \{(s_1, s_2), (s_2, s_1), (s_3, s_2)\}$，$AP = \{p, q\}$，以及 $L = \{s_1 \mapsto \{p\}, s_2 \mapsto \{p, q\}, s_3 \mapsto \emptyset\}$ 可绘制为下图：



注意，初始状态由没有源节点的边来标记。

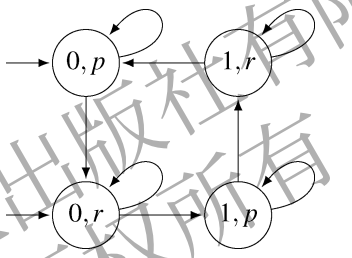
对于“模型”的解读与其出现的位置有关：在模型检测的语境下，Kripke 结构被视为模型，

基于此模型对系统进行分析；但是在逻辑体系中，如果 Kripke 结构 M 满足某规约，那么 M 被视为此规约的模型，这也是模型检测名称的来源。通常联系上下文以后，模型的含义显而易见。

3.2　非确定性和输入

系统建模通常会受制于系统细节和行为的缺失或遗漏。上述变迁系统(或者说 Kripke 结构)的定义并不要求从某状态开始的变迁确定不变。形式化而言，此状态的后继状态可能会有多个。与此类似，也不要求系统的初始状态是唯一的。也就是说，变迁系统具有非确定性特征，通常对细节缺失的系统或者其赋存环境建立非确定性的模型。

电灯开关具有典型的非确定性特征，环境对电灯的输入是非确定的：初始情况下，电灯是关闭的；当按钮按下时，电灯就会打开；下次关灯需要先松开按钮，然后再次按下。我们可以得到如下四个状态的 Kripke 结构：



标记为 1 的状态表示电灯打开，标记为 0 的状态表示电灯关闭，r 表示按钮松开，p 表示按钮按下。因为不需要对按下或者松开按钮的人进行建模，所以采用的非确定模型如下。

1．初始状态是非确定的：刚开始按钮既可能按下也可能松开，所以初始状态有两个。

2．变迁具有非确定性，所以图中每个状态都有两个后继。比如标记为 $(0, r)$ 的状态有两个后继：变迁到 $(1, p)$ 的后继表示按钮按下；变迁到 $(0, r)$ 的后继表示自变迁(self-transition)。

非确定性模型并不仅限于来自环境的输入，3.6.2 节也会介绍系统自身所具有的非确定性。

3.3　一阶逻辑和符号表示法

系统由程序设计语言或者硬件描述语言表示，为了得到其语义，我们使用一阶逻辑公式来刻画初始状态集合和变迁关系。假设读者已具备一阶逻辑的基础知识，熟悉逻辑运算(逻辑与 $\wedge$，逻辑或 $\vee$，逻辑非 $\neg$，蕴含 $\rightarrow$ 等)，并且也理解全称量词($\forall$)和存在量词($\exists$)的意义。

与“数理逻辑”和“程序设计语言的语义学”课程的内容相比，本章所用的逻辑和语义相当“非形式化”，并不考虑一阶逻辑和公理系统本身，只讨论特定数学结构下一阶逻辑公式的赋值计算，此时一阶逻辑用于描述对程序变量的约束。因此，程序变量对应了一阶逻辑

公式的变量，作用在程序变量上的特定数学结构将根据程序变量的数据类型来解释。

比如，整型的程序变量将解释在自然数 N 及其相关运算上，也可能解释为 32 位的位向量。第一种解释下的 Kripke 结构的状态空间是无限的，第二种解释下的状态空间很大，但却是有限的。两种模型都能满足其应用领域的要求：无限状态模型对应了类似算法教材的数学视角；而有限状态模型能精确表示真正的 C 程序。为上述每种解释分别构建正确的实例都是比较容易的，但是很难构建同时满足二者的实例。

从技术的角度来看，通常使用多类型一阶逻辑来描述公式。多类型一阶逻辑解释在特定的一阶逻辑结构之上，这种结构自然地对应了程序变量的数据类型和值域。本章我们限定到简单的变量类型如整型和布尔型，后续将根据需求引入更多的程序语义概念。

因为状态是系统的瞬时描述，因此可以自然地使用系统变量的赋值来标记状态。(正如后面将要讨论的，程序中的程序计数器就是这种变量)。在本书中，设 $V=\{v_1,\cdots,v_n\}$ 为系统变量集合， D_v 是变量 v 的值域。则 V 的赋值是一个把 V 中的变量 v 和 D_v 中的值联系起来的函数，即状态就是映射 $s:V\to\bigcup_{v\in V}D_v$ 。

下面考虑变量集合 $V=\{v_1,v_2,v_3\}$，每个变量的值域 $D_{v_i}=\mathbb{N}$(对于所有的 i)，其状态集合为

$$\mathbb{N}\times\mathbb{N}\times\mathbb{N}$$

或者简写为$\mathbb{N}^3$。对于一组给定的赋值，可以构建仅在这个赋值上成立的公式。例如，赋值 $\langle v_1\mapsto 2,v_2\mapsto 3,v_3\mapsto 5\rangle$ 可以表示成公式

$$(v_1=2)\wedge(v_2=3)\wedge(v_3=5)$$

但通常而言，公式会在多种赋值上为真。而且采用一组赋值上为真的一阶逻辑公式会更方便，可以使用此公式描述状态集合的子集。因此，一阶逻辑公式可被视为状态集合的特征函数或者符号表示。系统中特殊的初始状态集合可以表示为 V 中若干变量的一阶逻辑公式 $\mathcal{S}_0$ 。

继续上述实例，考虑以下状态子集：

$$\begin{aligned}\{\ &\langle v_1\mapsto 2,v_2\mapsto 3,v_3\mapsto 1\rangle,\\ &\langle v_1\mapsto 2,v_2\mapsto 3,v_3\mapsto 2\rangle,\\ &\langle v_1\mapsto 2,v_2\mapsto 3,v_3\mapsto 3\rangle\ \ \}\end{aligned}$$

这个子集可以表述为包含析取运算的一阶逻辑公式：

$$\begin{aligned}&((v_1=2)\wedge(v_2=3)\wedge(v_3=1))\vee\\ &((v_1=2)\wedge(v_2=3)\wedge(v_3=2))\vee\\ &((v_1=2)\wedge(v_2=3)\wedge(v_3=3))\end{aligned}$$

所以对于由 V 中的 3 个变量构成的公式 ϕ， $s\vDash\phi$ 表示 s 属于公式 ϕ 对应的状态集合。

使用一阶逻辑公式时，可以通过公式化简来使集合表示更加紧凑。在上述例子中，使用如下逻辑等价公式表示相同的状态集合：

$$(v_1=2)\wedge(v_2=3)\wedge(v_3\geqslant 1)\wedge(v_3\leqslant 3)$$

用一阶逻辑公式特征化集合后，就可以将常规的集合运算变为一阶逻辑特征函数的变换。令 A 和 B 表示集合 S 的两个子集，$s \in S$ 上的 $\mathcal{A}(s)$ 和 $\mathcal{B}(s)$ 分别代表 A 和 B 的特征函数，则有

$$\begin{array}{lll} A \cup B & & \mathcal{A}(s) \vee \mathcal{B}(s) \\ A \cap B & \text{对应} & \mathcal{A}(s) \wedge \mathcal{B}(s) \\ S \setminus A & & \neg \mathcal{A}(s) \end{array}$$

类似的公式变换均可应用到相应的集合关系运算上。比如，根据对于所有 s 来确定公式 $\mathcal{A}(s) \Rightarrow \mathcal{B}(s)$ 的真值，就能检测 $A \subseteq B$。

上述例子也展示出用一阶逻辑足以符号化表示变迁关系，但还要扩展上述办法来符号化表示状态集合，即采用一阶逻辑公式表示有序的状态对的集合。状态对指的是系统变量的两次赋值，所以仅使用 V 就无法表示这个对，需要引入第二个变量集合 V'。故 V 中的变量是现态变量，而 V' 中的变量是次态变量，现态变量集合 V 中的每个变量 v 都对应了 V' 中的一个次态变量 v'。$V \cup V'$ 集合元素中变量的赋值就是有序的状态对，即变迁。采用与以上类似的方法就能表示这些赋值组成的集合。令 $R \subseteq S \times S$ 是一个变迁关系，则 $\mathcal{R}(V, V')$ 表示了此变迁对应的一阶逻辑公式。比如，以下公式

$$(v_1' = v_1) \wedge (v_2' = v_2 + 1)$$

表示的变迁关系是：v_1 保持不变，v_2 在每一步都增加，v_3 不受约束故不确定。给定包含了 $V \cup V'$ 中自由变量的一阶逻辑公式 ϕ，$s, s' \vDash \phi$ 说明状态对 (s, s') 对应的变迁关系的符号表示为 ϕ。

原子命题集合 AP 也由类似的机制定义。AP 是标记的有限状态集合，这些标记由系统状态的相关信息构成，对于任意的标记 $l \in AP$，$s \vDash l$ 表示 $l \in L(s)$；而 $s \nvDash l$ 表示 $l \notin L(s)$。而且一定要注意到，AP 可以包含相当复杂的标记：特别是 AP 可以包含任意形式的性质，此性质是真是假由状态唯一确定，也就是说经过变量赋值才能确定，如 AP 中的标记可以是 $v = d$ 形式的公式，其中 $v \in V$ 且 $d \in D_v$。如果 $s(v) = d$，命题 $(v = d)$ 为真；此时 $(v = d) \in L(s)$，即 $s \vDash (v = d)$。以上分析可以泛化为，命题可以由包含 V 中自由变量的一阶逻辑公式表示。比如，更复杂的命题可以有以下形式：

$$v_1 > v_2 \wedge v_2 > v_3$$

v 是一个布尔域 $\{true, false\}$ 上的变量，那么 AP 并不需要包含 $v = true$ 及 $v = false$。$s \vDash v$ 表示 $s(v) = true$，$s \vDash \neg v$ 表示 $s(v) = false$。

现在就可以得出由一阶逻辑公式 $\mathcal{S}_0$ 和 $\mathcal{R}$ 表示并发系统的 Kripke 结构 $M = (S, S_0, R, AP, L)$ 的方法。

- 状态集合 S 是变量集合 V 上所有赋值构成的集合。
- 初始状态集合 S_0 是 V 上满足公式 $\mathcal{S}_0$ 的赋值 s_0 构成的集合，即 $s_0 \vDash \mathcal{S}_0$。
- 令 s 和 s' 为两个状态，则 $R(s, s')$ 为真当且仅当 $s, s' \vDash \mathcal{R}$。
- 标记函数 $L: S \to 2^{AP}$ 定义为 $L(s)$ 是所有满足 s 的标记构成的集合，即 $s \vDash l$。

因为要求 Kripke 结构的变迁关系一定是完全的，所以当状态 s 没有后继时，必须对其进行扩展，即修改 R，使之包含 $R(s, s)$。

例 3.1 为了说明本节的概念，考虑一个包含 x 和 y 两个变量的系统，变量的值域为 $D=\{0,1\}$，这时变量 x 和 y 的赋值就是一个对 $(d_1,d_2)\in D\times D$，其中 d_1 是 x 的值，d_2 是 y 的值。此系统还包含一个变迁关系：

$$x:=(x+y) \bmod 2$$

系统从 $x=1$ 和 $y=1$ 对应的状态开始，由两个一阶逻辑公式表示，一个是初始状态集合

$$\mathcal{S}_0(x,y)\equiv x=1\wedge y=1$$

另一个是变迁关系集合的一阶逻辑公式

$$\mathcal{R}(x,y,x',y')\equiv x'=(x+y) \bmod 2\wedge y'=y$$

从这些公式抽取出来的 Kripke 结构 $M=(S,S_0,R,AP,L)$ 如下所示：

- $S=D\times D$
- $S_0=\{(1,1)\}$
- $R=\{((1,1),(0,1)),((0,1),(1,1)),((1,0),(1,0)),((0,0),(0,0))\}$
- $AP=\{x=0,x=1,y=0,y=1\}$
- $L((1,1))=\{x=1,y=1\}$, $L((0,1))=\{x=0,y=1\}$, $L((1,0))=\{x=1,y=0\}$, $L((0,0))=\{x=0,y=0\}$

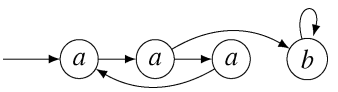
此 Kripke 结构中从初始状态开始的路径只有一条，即

$$(1,1)(0,1)(1,1)(0,1)\cdots$$

3.4 布尔编码

命题逻辑中的公式由命题变量和逻辑运算符(连接符)如与、或、非构成，命题变量的值可能是真(*true*)，也可能是假(*false*)。通常将真写作 1，将假写作 0。在进行布尔编码之后，就可以采用诸如 BDD、命题满足性等命题逻辑的推理技术，因此需要一种能将特征函数 $\mathcal{S}_0$ 和 $\mathcal{R}$ 翻译为命题逻辑的方法。如果状态集合 S 是有限的，那么这种方法通常是可行的，后续将以实例的形式阐述此方法。

例 3.2 使用 4 个状态的 Kripke 结构来解释命题编码：



将状态标记为 a 或者 b，因此可以使用两个布尔变量 v_0 和 v_1 来编码模型的 4 个状态，其值域为 $D_{v0}=D_{v1}=\{true,false\}$。最左端的状态编码为 $v_0=false$ 和 $v_1=false$，第二个状态为

$v_0 = ture$ 和 $v_1 = false$，以此类推。前面已经介绍了公式可表示为满足此公式的所有变量赋值的集合，所以在编码之后，得到表示初始状态的公式如下：

$$S_0(v_0, v_1) = \neg v_0 \wedge \neg v_1$$

变迁关系如下：

$$\begin{aligned} R(v_0, v_1, v_0', v_1') = \quad & \neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1' \\ \vee \quad & v_0 \wedge \neg v_1 \wedge v_1' \\ \vee \quad & \neg v_0 \wedge v_1 \wedge \neg v_0' \wedge \neg v_1' \\ \vee \quad & v_0 \wedge v_1 \wedge v_0' \wedge v_1' \end{aligned}$$

其中 R 的第二个子句表示两个变迁。

3.5　数字电路建模

3.5.1　状态保持单元

本节介绍采用公式描述电路的方法。电路中通常会采用一些特殊的单元存储数据，可称之为状态保持单元。为了简单起见，假设每个状态保持单元取值为 0 或者 1，令 V 为电路状态保持单元的集合。对于同步电路，集合 V 通常只包含电路中所有寄存器的输出及主输入。而对于异步电路，通常认为电路中所有的导线都是状态保持单元。如果将 V 中的各元素作为一个布尔变量，那么状态就可以表示为由 1 或 0 构成的对所有变量的一次赋值。每一给定的赋值都可视为特定布尔表达式，而此表达式在这个赋值下也为真，例如，给定 $V = \{v_1, v_2\}$ 和赋值 $\langle v_1 \mapsto 1, v_2 \mapsto 0 \rangle$，可得到布尔公式 $v_1 \wedge \neg v_2$。如前所述，我们可以将公式视为使之为真的赋值集合，因此描述电路并不需要用表达能力强的一阶逻辑公式，使用布尔逻辑公式就足够了。定义布尔表达式 $\mathcal{S}_0(V)$ 和 $\mathcal{R}(V, V')$ 分别表示电路的初始状态集合和变迁关系。

3.5.2　同步电路

同步电路工作时包含一系列具有时序的步骤。在每一步，电路的输入发生变化，电路进行响应而达到稳定，而后时钟脉冲到来，电路状态保持单元发生改变。

以下简单例子可以说明获取同步电路变迁关系的方法。图 3.1 所示的电路是一个同步模 8 计数器。设 $V = \{v_0, v_1, v_2\}$ 为此电路的现态变量集合，并设 $V' = \{v_0', v_1', v_2'\}$ 为次态变量集合，此计数器的变迁可表示为

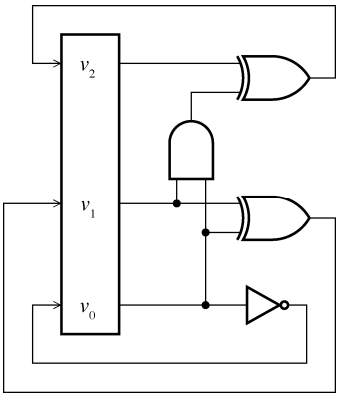


图 3.1　同步模 8 计数器

$$v_0' = \neg v_0$$
$$v_1' = v_0 \oplus v_1$$
$$v_2' = (v_0 \wedge v_1) \oplus v_2$$

$\oplus$ 是异或操作。以上等式可以用来定义下述关系：

$$\mathcal{R}_0(V, V') \equiv (v_0' \Leftrightarrow \neg v_0)$$
$$\mathcal{R}_1(V, V') \equiv (v_1' \Leftrightarrow v_0 \oplus v_1)$$
$$\mathcal{R}_2(V, V') \equiv (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

这些公式描述了每个 v_i' 在合法变迁后必须满足的约束条件。因为同步电路中所有的改变都同时发生，所以可以合取约束条件来构造变迁关系：

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V')$$

通常对具有 n 个状态保持单元的同步电路，可以设 $V = \{v_0, \cdots, v_{n-1}\}$ 且 $V' = \{v_0', \cdots, v_{n-1}'\}$。与模 8 计数器的分析方法类似，对于每个状态变量 v_i'，定义布尔函数 f_i 使得

$$v_i' = f_i(V)$$

上述等式就能定义变迁关系

$$\mathcal{R}_i(V, V') \equiv (v_i' \Leftrightarrow f_i(V))$$

不需要为电路主输入对应的变量也构造上述函数，通常并不约束这些变量，即

$$\mathcal{R}_i(V, V') \equiv true$$

再从模 8 计数器的例子继续类推，以上这些公式的合取就构成了整个模型系统的变迁关系

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \cdots \wedge \mathcal{R}_{n-1}(V, V')$$

因此，同步电路对应的变迁关系可表示为单个变量对应的变迁关系的合取。

3.5.3 异步电路

这里仅讨论一种简单和非定时的异步电路模型。异步电路的变迁关系可以很自然地表示为析取形式。以下由易到难地介绍了变迁关系的获取方法，首先假设异步电路的每个组件都只有一个输出，且没有内部状态变量，此时组件就能定义为函数 $f_i(V)$，只要知道 V 中当前状态变量的值，就可以通过 $f_i(V)$ 计算出这个组件的输出。这种方法可直接扩展到多输出的电路。

由于各组件的值可以快速变化，两个组件几乎不可能同时改变，因此习惯上使用交织语义来描述每个时刻仅有一个组件变化，由此可得出以下析取形式：

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \vee \cdots \vee \mathcal{R}_{n-1}(V, V')$$

其中

$$\mathcal{R}_i(V, V') \equiv \left(v_i' \Leftrightarrow f_i(V)\right) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j)$$

仔细观察上述公式会发现，可能出现某些组件反复变化而其他组件却无法执行的情况。

但在实际中这种情况几乎不可能发生，可以添加公正性约束来防止这种情况的发生，这方面的内容将在第 4 章讨论。

例 3.3　以下例子用以区别同步和异步模型。设 $V=\{v_0,v_1\}$、$v_0'=v_0\oplus v_1$、$v_1'=v_0\oplus v_1$，且状态 s 表示为 $v_0=1\wedge v_1=1$。在同步模型中，所有的赋值同时执行，所以此时 s 只有一个后继状态 $v_0=0\wedge v_1=0$。而在异步模型中，s 有两个后继状态：

1. $v_0=0\wedge v_1=1$（先对 v_0 赋值）。
2. $v_0=1\wedge v_1=0$（先对 v_1 赋值）。

3.6　程序建模

3.6.1　顺序进程

以下从基本的顺序程序开始介绍，然后将方法扩展到异步交织语义下的并发程序。本书所用的建模方法与 Manna 和 Pnueli 的方法[369]类似，用于对理想的程序设计语言建模。第 14 章将介绍处理工业界使用的程序设计语言的细节。

程序由彼此之间有顺序关系的语句组成，我们引入一个翻译算法 $\mathcal{C}$，它将顺序程序 P 翻译为一阶逻辑公式 $\mathcal{R}$ 表示的程序变迁的集合。为了不失一般性，假设每条语句都有唯一的入点和出点。标签转换算法将一个无标签程序 P 转化为标签程序 $P^{\mathcal{L}}$，从而极大地简化翻译过程。我们将这些标签称为程序位置。

标签转换算法在程序 P 中每条语句的入点附上唯一标签，但不标记整个程序 P。在顺序程序中，一条语句的出点就是下一条语句的入点，所以只标记入点就足够了。程序 P 的入点和出点也将标记，这样程序 P 中所有语句的入点和出点都唯一标记了。

不失一般性，标签转换算法不能局限于具体的程序设计语言，以下选用常见的语句类型来定义其标记过程，可以很容易地将这种方法扩展到其他语句类型。给定一条语句 P，标记后的语句 $P^{\mathcal{L}}$ 定义如下：

- 如果 P 不是复合语句(例如，P 是 $x:=e$，**skip**，**wait**，**lock**，**unlock**，等等)，那么 $P^{\mathcal{L}}=P$。
- 如果 $P=P_1;P_2$，那么 $P^{\mathcal{L}}=P_1^{\mathcal{L}};l_1:P_2^{\mathcal{L}}$，其中 l_1 是新标签。
- 如果 $P=$ **if** b **then** P_1 **else** P_2 **end if**，那么 $P^{\mathcal{L}}=$ **if** b **then** $l_1:P_1^{\mathcal{L}}$ **else** $l_2:P_2^{\mathcal{L}}$ **end if**，其中 l_1 与 l_2 都是新标签。
- 如果 $P=$ **while** b **do** P_1 **end while**，那么 $P^{\mathcal{L}}=$ **while** b **do** $l_1:P_1^{\mathcal{L}}$ **end while**，其中 l_1 是新标签。

本节后续部分均假设 P 已标记，其入点和出点分别标记为 m 和 m'。定义 pc 为程序计数器，其取值范围是由所有标签和附加的 **susp** 构成的集合。并发程序系统中 **susp** 表示程序暂停，$pc=$ **susp** 表示此程序目前处于非活跃态。

设 V 为程序变量集合，相应地设 V' 为次态变量集合，其中每个 $v \in V$ 都对应一个 $v' \in V'$，pc 所对应的次态记为 pc'。如前所述，现态对应变迁前变量的赋值，而次态对应变迁后的赋值。一般来说，每一个变迁只会更改少量的程序变量，所以定义 $same(Y)$ 表示公式

$$\bigwedge_{y \in Y} (y' = y)$$

首先给出程序 P 的初始状态集合对应的一阶公式的生成方法，设 $pre(V)$是程序 P 若干变量的初始值，则初始状态集合可以表示如下：

$$\mathcal{S}_0(V, pc) \equiv pre(V) \wedge pc = m$$

翻译算法 $\mathcal{C}$ 取决于三个参数：入点标签 l，已标记的语句 P，以及出点标签 l'。翻译算法为该语言的每种语句类型递归定义了一条规则。$\mathcal{C}(l, P, l')$ 描述了 P 中一系列的变迁，由多个变迁析取表示。在每个析取式中，程序计数器条件确保了各个变迁何时执行。

- 赋值

$$\mathcal{C}(l, v := e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \setminus \{v\})$$

- 跳过

$$\mathcal{C}(l, skip, l') \equiv pc = l \wedge pc' = l' \wedge same(V)$$

- 顺序组合

$$\mathcal{C}(l, P_1; l'' : P_2, l') \equiv \mathcal{C}(l, P_1, l'') \vee \mathcal{C}(l'', P_2, l')$$

变迁 $P_1; l'' : P_2$ 的公式是 P_1 的变迁公式和 P_2 的变迁公式的析取。因为 P_2 入点标签为 l''，所以语句 P_2 将在语句 P_1 之后执行。

- 条件

$\mathcal{C}$ (l, **if** b **then** l_1:P_1 **else** l_2:P_2 **end if**, l') 是以下四个公式的析取：

$$\begin{aligned}
&(pc = l \wedge pc' = l_1 \wedge b = true \wedge same(V)) \\
&\vee (pc = l \wedge pc' = l_2 \wedge b = false \wedge same(V)) \\
&\vee \mathcal{C}(l_1, P_1, l') \\
&\vee \mathcal{C}(l_2, P_2, l')
\end{aligned}$$

第一个公式对应条件 b 为真的情况，在这种情况下，语句 P_1 将随后执行。第二个公式对应条件 b 为假的情况，这时随后执行语句 P_2，这两个公式表示的都是程序计数器变化所导致的变迁。第三个和第四个析取公式分别是 P_1 和 P_2 的自身变迁。值得注意的是，l' 既表示 P_1 的出点，也表示 P_2 的出点。只要从公式中删除 b，**if** 语句的翻译就可以扩展到多分支不确定选择的情况。

- 循环

$\mathcal{C}$ (l, **while** b **do** l_1:P_1 **end while**, l') 是以下三个公式的析取：

$$(pc = l \wedge pc' = l_1 \wedge b \wedge same(V))$$
$$\vee(pc = l \wedge pc' = l' \wedge \neg b \wedge same(V))$$
$$\vee C(l_1, P_1, l)$$

第一个析取公式对应 b 为真的情况，随后将执行语句 P_1。第二个析取公式对应 b 为假的情况，这时 **while** 语句执行终止。第三个析取公式是 P_1 本身的变迁公式。值得注意的是，P_1 的出点与 **while** 语句的入点相同。因此，P_1 执行终止后，**while** 语句会重新开始执行。

下面，我们将讨论并发进程建模的多种风格，然后在描述并发程序模型时介绍软件建模方法。

3.6.2　并发进程建模

多个同时执行的组件构成了并发系统，这些组件常常借助一些机制彼此通信，系统执行和通信的模型有可能会不同。本节考虑两种执行模型：同一时刻仅允许一个组件运转一步的异步或者交织执行模型，以及所有组件同时运转一步的同步执行模型。对于通信模型，当然也可以采用多种方式进行建模，比如组件之间可以通过改变共享变量的值进行通信，也可以使用队列或其他握手协议交换消息来实现通信。因为本书的主要目的不是介绍建模方法，所以仅介绍改变共享变量这种通信机制。

以下我们将介绍一些重要的并发系统类型，介绍如何用一阶公式来描述它们，依据这些公式，就可以采用类似于 3.3 节的方法，将其表示为 Kripke 结构。

并发程序由多个并行执行的进程构成，进程由前面介绍的顺序语句组成。本节将考虑异步进程，即在任意时刻仅有一个进程执行变迁。首先定义术语，V_i 是进程 P_i 引起赋值变化的变量集合，不要求各进程的变量是唯一的(变量集合的交集为空)，V 是所有进程变量的集合，相应地，pc_i 是进程 P_i 中的程序计数器，而 PC 是所有程序计数器的集合。

并发程序 P 表示为

$$\textbf{cobegin } P_1 \| P_2 \| \cdots \| P_n \textbf{ coend}$$

其中 $P_1, P_2, \cdots, P_n$ 是进程。并发程序的标签转换算法由顺序程序的标记算法扩展而来，具体做法是给每一个进程都加上入点和出点标签。与顺序程序不同的是，这里的所有出点标签都是唯一的。如前所述，各进程中的所有标签都是不同的，并发程序 P 的出点和入点标签分别为 m 和 m'。

- 如果 $P = \textbf{cobegin } P_1 \| P_2 \| \cdots \| P_n \textbf{ coend}$
  则 $P^{\mathcal{L}} = \textbf{cobegin } l_1 : P_1^{\mathcal{L}} l_1' \| l_2 : P_2^{\mathcal{L}} l_2' \| \cdots \| l_n : P_n^{\mathcal{L}} l_n' \textbf{ coend}$

描述并发程序 P 的初始状态的公式如下：

$$S_0(V, PC) \equiv pre(V) \wedge pc = m \wedge \bigwedge_{i=1}^{n} (pc_i = \textbf{susp})$$

其中，$pc_i = \textbf{susp}$ 表明进程 P_i 没有激活，因此不能在目前的状态下执行。

翻译算法 $\mathcal{C}$ 按照如下方法扩展到并发程序。$\mathcal{C}(l, \textbf{cobegin}\ l_1 : P_1 l_1' \| \cdots \| l_n : P_n l_n'\ \textbf{coend}, l')$ 的结果为如下三个公式的析取：

$$\begin{aligned}&(pc = l \wedge pc_1' = l_1 \wedge \cdots \wedge pc_n' = l_n \wedge pc' = \textbf{susp})\\ \vee\ &(pc = \textbf{susp} \wedge pc_1 = l_1' \wedge \cdots \wedge pc_n = l_n' \wedge pc' = l' \wedge \bigwedge_{i=1}^{n} (pc_i' = \textbf{susp}))\\ \vee\ &\bigvee_{i=1}^{n} \left(\mathcal{C}(l_i, P_i, l_i') \wedge same(V \setminus V_i) \wedge same(PC \setminus \{\, pc_i \,\})\right)\end{aligned}$$

第一个公式描述了并发系统的初始化，此时变迁由 **cobegin** 语句的入点迁移到若干子进程的入点，创建子进程的程序暂停。第二个公式描述并发系统的终止，变迁由子进程的出点迁移到 **cobegin** 语句的出点，只有当所有进程都终止时，这个变迁才会执行。

第三个公式描述了并发进程的交织执行，我们采用交织语义，也就是说，任何时刻仅有一个进程执行变迁。进程 P_i 的变迁关系公式将同以下公式做合取：

$$same(V \setminus V_i) \wedge same(PC \setminus \{pc_i\}))$$

这不但保证了 P_i 中的变迁只能改变 V_i 中的变量，还能确保每次只有一个进程执行变迁。进程执行变迁是非确定的。

共享变量

进程 P_i 会引起 V_i 中变量赋值的变化。集合 V_i 中有重叠的并发程序称为共享变量程序。下面我们扩展解释算法 $\mathcal{C}$，使之能够处理进程同步语句，这种语句提供了多进程互斥访问共享变量的机制，在标签转换算法中，这种语句具有原子性。假设进程 P_i 中包含这种语句。

- 等待　因为本书的重点在于有限状态程序，所以仅考虑忙等待的语句，也就是实际上并不会涉及诸如进程队列的复杂数据结构。语句 **wait**(b)将一直测试布尔变量 b 的值是否为真，当其为真时该变迁执行，进程从而可以运行到下一个标签。
  翻译算法 $\mathcal{C}(l, \textbf{wait}(b), l')$ 是如下两个公式的析取：

  $$(pc_i = l \wedge pc_i' = l \wedge \neg b \wedge same(V_i))$$
  $$\vee(pc_i = l \wedge pc_i' = l' \wedge b \wedge same(V_i))$$

- 锁定　语句 **lock**(v)类似于 **wait**($v = 0$)，只是在该语句执行过程中当 $v = 0$ 为真时，变迁会把 v 的值设置为 1。这条语句常用来保证互斥，也就是阻止多个进程同时进入临界区。翻译算法 $\mathcal{C}(l, \textbf{lock}(v), l')$ 是如下两个公式的析取：

  $$(pc_i = l \wedge pc_i' = l \wedge v = 1 \wedge same(V_i))$$
  $$\vee(pc_i = l \wedge pc_i' = l' \wedge v = 0 \wedge v' = 1 \wedge same(V_i \setminus \{v\}))$$

- 解锁　**unlock**(v)语句将 v 的值赋为 0。执行该语句后，进程就可以退出临界区，从而使其他进程能够进入临界区。

$$\mathcal{C}(l, \textbf{unlock}(v), l') \equiv pc_i = l \wedge pc_i' = l' \wedge v' = 0 \wedge same(V_i \setminus \{v\})$$

例 3.4　接下来研究双进程 P_0 和 P_1 互斥的程序：

$$P = m : \textbf{cobegin } P_0 \parallel P_1 \textbf{ coend } m'$$

其中，

$P_0 ::$ l_0 : **while** *true* **do**
　　NC_0 : **wait**$(turn = 0)$;
　　CR_0 : $turn := 1$
　end while;
　l'_0

$P_1 ::$ l_1 : **while** *true* **do**
　　NC_1 : **wait**$(turn = 1)$;
　　CR_1 : $turn := 0$
　end while;
　l'_1

P 的程序计数器 pc 有三个值：P 的入点标签 m，P 的出点标签 m'，以及当 P_1 与 P_2 同时未激活时 pc 的值 **susp**。每个进程 P_i 有一个独立的程序计数器 pc_i，其取值范围是 l_i、l'_i、NC_i、CR_i 和 **susp**。两个进程共享唯一的变量 *turn*。因此有 $V = V_0 = V_1 = \{turn\}$，$PC = \{pc, pc_0, pc_1\}$。当进程 P_i 的程序计数器的值为 CR_i 时，进程进入它们的临界区，但两个进程不能同时进入临界区。当程序计数器的值为 NC_i 时，进程处于非临界区，这时进程一直等到对应的 $turn = i$ 才能获准进入临界区。

P 的初始状态表示如下：

$$S_0(V, PC) \equiv pc = m \wedge pc_0 = \textbf{susp} \wedge pc_1 = \textbf{susp}$$

我们不对 *turn* 的值进行限制，可以取 0 或 1 的任意值。应用翻译算法 $\mathcal{C}$ 后，可以得到 P 的变迁关系公式 $\mathcal{R}(V, PC, V', PC')$ 是如下四个公式的析取：

$$\begin{aligned}
&(pc = m \wedge pc'_0 = l_0 \wedge pc'_1 = l_1 \wedge pc' = \textbf{susp}) \\
\vee\ &(pc_0 = l'_0 \wedge pc_1 = l'_1 \wedge pc' = m' \wedge pc'_0 = \textbf{susp} \wedge pc'_1 = \textbf{susp}) \\
\vee\ &(\mathcal{C}(l_0, P_0, l'_0) \wedge same(V \setminus V_0) \wedge same(PC \setminus \{pc_0\})) \\
&\textit{(which is equivalent to } \mathcal{C}(l_0, P_0, l'_0) \wedge same(pc, pc_1)\textit{)} \\
\vee\ &(\mathcal{C}(l_1, P_1, l'_1) \wedge same(V \setminus V_1) \wedge same(PC \setminus \{pc_1\}) \\
&\textit{(which is equivalent to } \mathcal{C}(l_1, P_1, l'_1) \wedge same(pc, pc_0)\textit{)}
\end{aligned}$$

对于任意进程 P_i，$\mathcal{C}(l_i, P_i, l'_i)$ 是以下公式的析取：

$$\begin{aligned}
&(pc_i = l_i \wedge pc'_i = NC_i \wedge true \wedge same(turn)) \\
\vee\ &(pc_i = NC_i \wedge pc'_i = CR_i \wedge turn = i \wedge same(turn)) \\
\vee\ &(pc_i = CR_i \wedge pc'_i = l_i \wedge turn' = (i+1) \bmod 2) \\
\vee\ &(pc_i = NC_i \wedge pc'_i = NC_i \wedge turn \neq i \wedge same(turn)) \\
\vee\ &(pc_i = l_i \wedge pc'_i = l'_i \wedge false \wedge same(turn))
\end{aligned}$$

图 3.2 的 Kripke 结构忽略了不可达状态，由 3.3 节讨论的公式 $\mathcal{S}_0$ 和 $\mathcal{R}$ 推导可得。分析

Kripke 结构的状态空间，可以很容易发现这两个进程不会同时进入它们的临界区，因此该程序的互斥特性是满足的。但是它不能保证不出现“饥饿”现象，即可能出现一个进程反复尝试进入临界区，另一个进程却一直占据此临界区而不退出的情况。在本书的后续章节中，我们将说明如何用公式表达诸如此类的系统性质并在模型上进行检测。

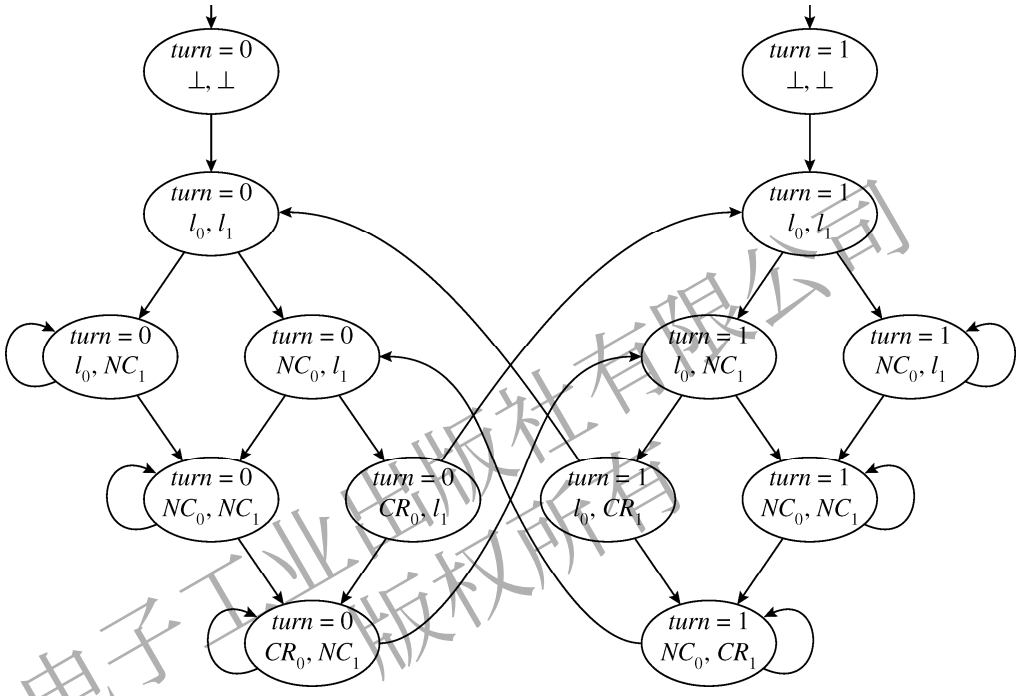


图 3.2　互斥程序的 Kripke 结构的可达状态

变迁的粒度

确定变迁的粒度是并发系统建模的一个关键问题。确保一个变迁部分执行后不出现可见的状态，也就是变迁的原子性非常重要。常见的错误是把变迁关系定义得过于粗糙，此时 Kripke 结构就会遗漏一些可见的系统状态。从而使包括模型检测在内的各种验证技术无法检测到某些重要的错误。但是变迁的粒度过细也会导致一些问题，比如在多个变迁关系的作用下，系统可能会到达一个实际不会出现的状态。也就是说，模型检测找到了一个实际不会出现的伪错误。

考虑如下粒度过粗和过细的例子，假设一个系统有两个变量 x 和 y，以及两个并发执行的变迁 α 和 β，

$$\alpha: \quad x := x + y$$

$$\beta: \quad y := y + x$$

在初始状态下，$x = 1 \wedge y = 2$。当采用汇编语言指令在内存和寄存器之间实现如下读取、相加和存储操作时，粒度会过细：

$$\alpha_0:\ \text{load}\ R_1, x \qquad \beta_0:\ \text{load}\ R_2, y$$
$$\alpha_1:\ \text{add}\ R_1, y \qquad \beta_1:\ \text{add}\ R_2, x$$
$$\alpha_2:\ \text{store}\ R_1, x \qquad \beta_2:\ \text{store}\ R_2, y$$

先执行 α 后再执行 β 的情况下，得到的结果是 $x=3 \wedge y=5$；但在 α 前执行 β 时，得到的结果却是 $x=4 \wedge y=3$。不仅如此，采用这种过细粒度并以 $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$ 为执行顺序时，则会得到 $x=3 \wedge y=3$。

假设 $x=3 \wedge y=3$ 违反了真实系统的某些期望属性。如果系统的实现符合变迁关系 α 和 β，就不可能同时得到 $x=3$ 和 $y=3$（系统正确）；但如果用过细粒度的变迁关系 α_0、α_1、α_2、β_0、β_1、β_2 对系统进行建模，则可能产生误判。反之，假设真实系统需要使用过细粒度变迁关系 α_0、α_1、α_2、β_0、β_1、β_2 来描述，这时它就可能到达状态 $x=3$ 和 $y=3$，但如果用粗粒度变迁关系 α 和 β 对系统建模，又会漏判系统存在的错误。

可以把从程序代码和电路图中抽取一阶逻辑公式的过程看成编译，此过程必须考虑上述的粒度问题。

3.7　公平性

前面已经介绍了缺少模型细节时就要使用非确定性，比如考虑 3.6.2 节中的并发程序 P，其形式如下：

$$\textbf{cobegin}\ P_1 \| P_2 \| \cdots \| P_n\ \textbf{coend}$$

真实系统可以采用调度器来实现交织特性，以保证这种确定的 P_1 到 P_n 的交织执行。尽管如此，我们会避免对调度器建模，一方面因为引入调度器后模型复杂度升高，另一方面为了验证 P 是否能在各种系统调度器下正常工作。因此，我们的模型允许进程中任何语句的交织，这对应了真实系统的过近似行为。由于过近似的边界效应，有时会得到模型非预期的行为。例如，一个模型中包含了真实系统中不具有的错误路径，即一个进程一直执行但其他进程永远暂停，这就是所谓的进程饥饿。

有很多通过添加约束来消除 Kripke 结构非预期行为的方法，下述内容将关注公平性。形式化而言，公平的 Kripke 结构 M 是一个六元组 $M=(S, S_0, R, AP, L, F)$，其中：

1．S、S_0、R、AP 和 L 同前面定义的一样。

2．$F \subseteq 2^S$ 是公平性约束。

令 $F=\{F_1, F_2, \cdots\}$，F_j 是 S 的子集，$s_0, s_1, \cdots$ 表示 M 中路径 π 的状态。如果对于每一个公平性约束 $F_j \in F$，路径 π 上有无限个 $s_i \in F_j$，那么路径 π 就是公平的。因此之后仅考虑 M 上的公平路径。

回顾上述例子，阻止进程饥饿的方法是，选择进程 P_i 的一个状态及对应变迁 σ_i，构建以下公平性约束：

$$\{s \in S \mid \sigma_1 \in L(s)\}, \cdots, \{s \in S \mid \sigma_n \in L(s)\}$$

在公平的 Kripke 结构中，每个进程都可以调度无限多次。

本章文献注释

3.6 节中采用一阶公式对程序进行形式化建模的方法类似于 Manna 和 Pnueli 的方法[369]。

系统模型涉及很多额外的概念，定时系统的建模将推迟到第 19 章和第 20 章再讨论。并发系统的形式化建模方法有很多种，比如通信顺序进程(CSP)、通信系统演算(CCS)、π 演算和 Petri 网等。

本章并没有讨论混杂系统和概率模型，*Handbook of Model Checking*[138]一书讨论了这些概念。有关概率模型的深入研究请参考Baier和Katoen的著作[35]。

习题

习题 3.1(特征函数) 特征函数用于定义变迁关系和状态集合。

a．为如下变迁关系写出一个简单的特征函数。

$$T:\{1,\cdots,10\}^2=\{(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,5)\}$$

b．T 是左完全的关系吗？

c．T 的自反传递闭包是什么？

习题 3.2(Verilog 建模) 给出这道习题的目的在于增强读者的信心，能够建模变迁系统。以下这个电路由 Verilog 硬件描述语言(HDL)表示。首先要读懂 Verilog HDL 的非形式化语义。

a．Verilog HDL 区分了两类赋值：使用=的阻塞赋值和使用<=的非阻塞赋值(请勿与≤混淆)。请解释二者的区别。

b．定义如下 Verilog 代码段的变迁系统：

```
input clk;
reg[31:0]A, B;

always@(posedge clk) begin
        A=B;
        B=A;
end
```

假设时钟在出现每一个变迁时触发。

c．将=赋值替换为<=赋值，根据以上 Verilog 代码段，定义一个变迁系统。

d．关键字 **reg** 并不一定生成模型的状态变量，为什么？请给出一个例子。