第 3 章

原型和图形

3.1 原型

3.1.1 概述

在自然界,“原型”演化是一种重要的生物进化方式。当环境变化显著或发生跃迁时,原来适应的生物面临生存压力,其生存方式必须做一些改变。开始时,生物生存行为上的转变可能仅仅引起生理结构的细微变化,后来经过长期进化的修修补补,生物才产生适应新环境的有效生理结构。与后来定型了的生理结构相比,原始的、简略的最初结构便是后来成熟结构的原型。在功能意义上,我们也可以说原型结构模拟了定型后结构的行为。例如,始祖鸟是现今鸟类的原型,作为生物飞行的最初尝试,它的行为是对飞行的模拟。

在知识界,“原型”构造是一种重要的科学研究方式。科学研究者考查科学现象时,往往对科学对象作出适当的理论假设,随机构造简易模型对自己的假设进行验证。其中主要的验证方式就是观察在一定的条件下,那个简单模型的“行为”是否像科学对象的“行为”一样,产生相同的或相近的自然现象。这个简易的、原始的模型便是科学对象的原型,它的行为便是对科学对象导致的自然现象的一种模拟。于是,通过原型,科学研究者便能证实或推翻自己事前所作的关于特定自然现象的科学推理,甚至促进自己研究进一步深入、细化。

在软件界,原型是一种重要开发且测试手段。软件界都喜欢在软件实现之前,构造一个简易原始模型,用以对即将开发产品的(至少是主要的)功能进行模拟。这个原始模型便是未来软件产品的原型。

采用原型方法的关键思想是模拟,它是软件最早的一个能够运行的版本,是未来产品的雏形。如果软件开发组织采用生物进化理念,把原型模型、进化模型和增量模型综合为一个开发模式,那么就可以在某些产品研制过程中,把原型作为蓝本,不断地增加或修改、进化它的功能。在软件产品的研制过程中,作为产品探索式开发手段,用户使用“构成”原型生存环境,在用户反馈驱动下,迫使原型不断变更,犹如生物进化一般,最后或者弃之不用,或者仅以它实现的主要功能作为核心保留在定型的最终版本中。这种开发方式比事前把什么都规划完整,“按部就班”地生成软件产品方式要快,而且容易应对来自客户对需求的变更以及创

造市场尚未展示的需求“商机”。

即使不以快速原型开发方式开发软件，软件开发过程也会从软件原型构造中受益。因为 Gordon 和 Bieman 通过对 39 个原型系统的调查研究，发现使用原型构造有如下好处：①改善了系统的可用性；②使系统更加贴近用户的需求；③改善了设计质量；④改善了可维护性；⑤减少了开发人力。他们发现，原型构造只是在软件开发的前期阶段需要增加费用，但是会使后期的代价减少。主要原因在于避免了开发过程中很多返工，而这又是由于客户请求系统变更的减少[33]。

一般地，原型系统只是被用来探讨需求和选择设计，作为开发的一种科研方法，原型最后并不移交出去，它是一个抛弃式原型。这是由于构造原型时往往忽略了许多“细节”，特别是非功能需求。例如，目标产品是用于处理应付款、应收款和库存等业务事项，而开发的原型可能只是用于完成数据捕获的屏幕处理和报表打印，并不进行文件处理和错误处理等操作。通常，这些被忽略的功能不能通过调整原型系统得到满足。如果最终软件版本使用了低效的原型代码或甚至就是那个理应被抛弃的原型，那么总的系统性能可能有所退化以致得到一个不完整、质量差、维护困难且昂贵的系统。尽管一般情况如此，但是原型能够快速地模拟系统（至少是主要的）功能，原型构造已经成为一种开发手段和验证手段，在软件界得到广泛使用，并获得很好的效果。

在软件开发早期，客户对自己的需求表述往往模糊不清，甚至需求之间还存在矛盾。有时客户所说的需求并没有表达出他们真正希望的东西，以致客户在整个开发周期中任何时候都会产生新的想法，从而要求系统做出变更，以满足他们心中的真正要求。同样，开发者通常也不能确切地把握客户的真正想法。客户和开发者都无法肯定能够满足客户心中所求的软件能否实现。

这时，最理想的做法是，开发者深入了解了客户的需求后，进行快速分析，理清客户的关键想法，确定初步规格说明，把最重要的功能通过原型构造展示出来。原型是软件最早一个能够运行的版本，客户使用它，就会看到即将开发出来的软件是怎样的“面貌”以及是如何支持他们的工作的。他们就会用眼前的“实现”去印证自己心中真正“期待”，一方面发现自己所提出来的需求中的一些错误和遗漏之处；另一方面也会对需求产生新的想法，从而找出软件中的优点和不足。特别是，用户最初认为自己对需求的功能给出了有用且是完整的描述，但是当某些功能通过原型结合起来运行的时候，用户通常会发现他们的最初想法是不正确的或不完整的。通过用户的反馈，需求描述得到修改，这些修改反映了用户在需求理解上的变化，也是关于开发者在需求工程中的工作的正确性的有效验证测试。通过用户使用原型的反馈信息，对原型进行修改，如此迭代式开发，可以减少花在完成用户文档和今后培训用户使用软件的时间。更重要的是，以此方式写出的需求文档（产品规格说明）会是准确的，且事后被用户要求变更系统的可能性较小，至少在主要功能上用户的需求已经稳定，不再发生变化了。

作为开发方式，原型可以用来演示概念，把客户需求抽象理念真实地、生动地呈现在人们面前，使开发者和客户在需求洽谈中达成一致。这同时增强了客户和开发者对软件制造

的信心。这种在早期阶段就引进的开发和验证的双重活动，促进了随后系统设计和实现阶段的顺利进行，并使开发成本得到很好的控制。

作为科学研究方式，原型也是软件组织在系统设计阶段进行的兼具开发和验证的双重活动。

作为开发活动，原型可以用来演示概念并尝试设计选择，可以用来发现更多的问题和可能的解决方案，明确系统设计和实现的方法和途径。在系统设计阶段，甚至细微到某个具体算法，也可能是通过对算法原型的改进加以实现的。虽然软件界一般主张建立快速原型，在软件开发过程早期就应该抛弃。但是存在允许精炼一个快速原型的做法，特别是快速原型的某些部门。当部分快速原型是由计算机生成的时候，这些部分就可以用在最后的产品中。例如，用户界面经常是快速原型的一个重要方面。因为用户界面的动态性，文字描述和图都难以表达用户的界面需求。这时开发软件系统图形用户界面原型是最有效的方法。当用屏幕生成器和报表生成器等计算机辅助软件工程(Computer-Aided Software Engineering，CASE)工具生成用户界面时，该快速原型的那些部分确实可以用作产品质量软件的一部分[22]。

作为验证活动，系统设计可以利用原型执行设计实验，以检验所提议的设计的可行性。例如，某个数据库设计可以通过原型构造和对原型的测试来检查，看它是否能对绝大多数的普通用户查询提供最高效的数据访问。虽然原型最后抛弃，但是构造原型使我们对即将实现的功能有深切体验，这种直观体验不仅帮助我们设计有效的测试用例去测试今后研制的实际系统，而且还可以实施一种所谓的“背对背”测试。当我们把相同的测试用例既提交给原型，又提交给待测试的实际系统时，如果两个系统给出相同的结果，测试案例可能没有检查出缺陷；如果结果不相同，则意味着系统存在某个缺陷，出现不同的原因有待进一步调查。背对背测试是原型在系统测试中的一种运用[33]。

综上所述，快速构造原型用在需求阶段。它能让用户尽早看到未来系统的概貌，并让客户通过不断反馈参与其中，它是直接捕获用户(恰当的)需求手段。因此，它也是需求验证和确认的一种机制。客户的反馈是对需求的确认机制，也是开发人员验证自己是否“真实地”获取需求的机制，从而能编写正确的需求文档。在系统设计阶段，快速构造的原型可以作为设计选择、算法演示、制定设计方案的手段，这时原型是系统运行的一个蓝本，因而是对系统可行性和正确性的一个“粗略”验证，以后可以应用于“背对背”测试中，作为解决系统测试有效性问题的一种方式。

3.1.2 示例

1. 原型支持用户界面设计

在软件的系统设计中，用户界面设计是最适合且最应该采用原型构造方式进行的开发活动。通常，用户界面的设计和实现都是开发人员的工作。在软件产品质量的因素中，可用性是至关重要的。人机交互界面友好与否，牵涉到人的因素，理所当然地应该以用户体验为准。与其产品完成后由于用户可用性要求(如他们觉得屏幕令人困惑甚至令人烦恼)而不得

不修改人机界面，还不如在用户界面设计之时，让用户参与进来。这时系统设计的最好选择便是快速构造原型，让用户使用用户界面与计算机进行交互，并把自己的感觉告诉开发人员。这样，开发人员便可以发现普通用户的思维逻辑、习惯、直觉和偏好，从而设计出与未来用户所具有的技能、经验和他们的期待一致且容易使用的界面。用户自己的亲身体验是别人代替不了的，开发人员仅凭“深思熟虑”抽象思考企图准确地描述什么是用户所想要的人机界面是极其困难的，而以用户为中心的进化式和探索式原型构造却能容易达到预期目标。由此观之，每个软件产品都要建造与其相应的用户界面的快速原型。

《实战需求分析》[19]对界面设计进行了专门讨论。书中提出设计出的人机界面不要让用户难以学习，不要让用户感到厌烦、恐惧和难以捉摸，进而强调以人为本，优化用户界面，使其具有易学性、易用性、健壮性，使系统在执行中能与用户进行友好沟通，让用户得到准确的自己能理解的消息，并能让计算机对用户提供的消息作出良好反应。在以人为本的理念下，除了详细讨论了界面设计过程以外，这本著作对原型设计方法也进行较完整的介绍，如手画法（即在纸面上构造原型）、使用 Microsoft Office 工具设计法、使用原型设计工具设计法以及使用开发工具设计原型等方法，并对每种方法的实施都列举若干案例加以阐述。

许多文献都讨论了怎样建立用户界面的原型，如《软件工程》（第 8 版）[33]指出，在理想情况下，原型构造可以采取两步原型构造过程。

（1）在过程的最早阶段，应该在纸面上规划“屏幕设计”模型并与未来用户一起探讨这个“原型”。

（2）继而要对设计进行提炼并逐渐地开发复杂的自动化的原型，再将它们呈现给用户，接受测试和活动模拟。

在纸面上构造原型，是既便宜又容易做的事。开发人员无须开发任何可执行软件，只要清楚、直观地画出用户将要与之交互的系统屏幕样式就可以。然而，要想获得“活动”原型起到的演示效果，还必须有一组用来描述系统是如何使用的脚本。

通常，人们理解事物的抽象描述时，喜欢把它与实例联系起来。如果把人如何与软件系统交互用脚本的方法描述，人们就很容易理解并且评论它的好与坏。开发人员从用户对场景的评论中得到信息，不断地修改和增加交互细节，就可以写出需要在屏幕上显示出的信息以及可供用户选择的选项。类似地，可以使用“情节串联图板”表述界面设计。情节串联图板是一系列描述交互序列的草图，虽然实用性较差，但当向一组人而不是单个人表述界面提案时，它是一个较方便的方法。

在初始“纸上谈兵”以后，我们需要实现一个软件界面设计原型。由于我们需要得到某些用户可以交互的系统功能，因此在系统开发的最初阶段就对用户界面“如实”地进行原型构造，往往是不太可能的。避开这个问题的方式，需要使用 Wizard of Oz（人冒充机器演示）原型构造方法。《绿野仙踪》里，有个影子很大的巨人，吓跑了很多人，后来发现他原来是一个瘦小的巫师，站在幕布后用灯放大影子。Wizard of Oz 原型构造方法的基本理念是先做简单的、容易做的事情，而当比较复杂的事情一时难以实现时，就先用一个人在后台冒充机器处理系统。这时，用户的输入被引导到一个隐藏的人，通过这个人仿真系统的响应。用户

认为自己是与计算机系统正在进行交互，在如此“逼真”的界面感觉下，用户愿意给出他们“真实”的意见和想法。实际上，在仿真系统的真实行为时，也可以通过使用某些其他系统计算所要的反应。总之，该方法的要点是除了要提出的用户界面以外，无须拥有任何可执行的软件。实验结束时，除了感谢参与实验的用户以外，还要告知他们真实情况，以示对他们的尊重。

界面设计中会存在缺陷和错误，利用与用户交互设计方法能够帮助开发人员发现问题。快速原型构造方法支持界面设计，不仅使开发人员更多地了解到自己的产品逻辑，从而确定用户界面最好式样，而且帮助开发人员预见未来的设计趋势，如用 Wizard of Oz 方法，现在就去设想当更多传感器和硬件加到设备上以后，怎样设计适应未来用户使用设备行为的用户界面。

在用户界面原型构造中，还可以使用脚本驱动、可视化编程语言以及基于因特网的原型构造等方法创建或提供用户界面。当然，我们要在迭代式原型构造及其实验中，寻找改善界面的方向和道路，当原型变得更完善，要对它进行评估。于是，可以采用抛弃式方法或进化式方法进行进一步的原型构造及其实验。

2. 原型支持算法研究

大部分科学活动（如果不是全部的话）都是基于一些“真知灼见”，并首先在极端理想化条件下，构造原型或思想实验，然后将条件尽可能一般化，构造“逼近”自然或实际情况的模型，从而得到理论体系，获得科学结论。

例如，爱因斯坦基于光速不变原理和洛伦兹变换，利用思想实验方法创立狭义相对论；后又基于等效原理（即引力和加速度等效原理）和广义相对性原理，又一次利用思想实验方法创立广义相对论。爱因斯坦基于“光量子”假设，发现光电效应。这些思想和方法引发了 20 世纪两大物理学革命。

在计算机科学中，原始思想和原型方法也是重要的科学研究和开发成熟软件的手段。从广义角度来看，图灵计算模型便是现代计算机的一种“思想实验”，是现代计算机的虚拟“始祖鸟”。德国数学家、计算机科学家 C. A. Petri 在年仅 13 岁时便萌生了现以他名字命名的 Petri 网的想法，他当时只是想用该网描述化学过程。1962 年，Petri 在他的博士论文 *Kommunikation mit Automaten* 中正式提出了 Petri 网理论，当时他是用该网描述自动机通信过程。因此，Petri 网最初只引起自动控制理论工作者的兴趣。后来在性能评估、操作系统以及软件工程等领域，也开始应用 Petri 网描述它们各自的问题，特别是 Petri 网可以有效地描述并发关系活动。目前 Petri 网已经在计算机科学中得到广泛运用，如可以用于设计，它是说明隐含定时问题的系统的一个功能强大的技术。

1975 年，van Emde Boas 提出（现在以他名字命名的）van Emde Boas 树数据结构初步想法。不久，他和 Kaas、Zijlstra 等对该想法加以精炼并发表，随后还得到 Mehlhorn 和 Näher 的扩展以及 Dementiev 等的新实现。Pătraşcu 和 Thorup 得到了查找前驱操作的一个下界，并说明了 van Emde Boas 算法在查找前驱操作上是最优的，即使允许引入随机化方法，仍是最优的。

现以 van Emde Boas 算法为例，说明原型在算法研究中的作用。基本理念是：人们在

研发新算法时，可以遵从一般科学研究方法，先从理想情况着手，构造算法原型，再推至一般情况，形成完整算法。本节取材于《算法导论》[27]一书。

假设只关注存储关系（不允许重复的）关键字，要存储的关键字的全域（Universe）为$\{0,1,2,\cdots,u-1\}$，u 为全域的大小。van Emde Boas 树数据结构维护一个 u 位的数组 $A[0..u-1]$，以存储一个关键字动态集合，其中的位来自全域$\{0,1,2,\cdots,u-1\}$。也就是说，以位向量方式存储一个动态集合，若值 x 属于动态集合，元素 $A[x]$为 1；否则，$A[x]$为 0。例如，$u=16$，全域为$\{0,1,2,\cdots,15\}$，存储关键字的动态集合为$\{2,3,4,5,7,14,15\}$，则在 $A[0..15]$中，当 $x=2,3,4,5,7,14,15$ 时，$A[x]=1$；当 $x=0,1,6,8,9,10,11,12,13$ 时，$A[x]=0$。van Emde Boas 树支持在动态集合上运行时间为 $O(\lg\lg u)$的操作：SEARCH、INSERT、DELETE、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR。

1）原型 van Emde Boas 结构

考虑理想情况，设全域大小 $u=2^{2^k}$，其中 k 为整数。对于全域$\{0,1,2,\cdots,u-1\}$，定义原型 van Emde Boas 结构或 proto-vEB 结构，记作 proto-vEB(u)。由于 $u=2^{2^k}$，使用结构递归方法，每次递归都以平方根大小缩减全域，因此 $u,u^{1/2},u^{1/4},\cdots,4,2$ 都为整数。于是原型结构可以递归定义如下：每个 proto-vEB(u)结构都包含一个指明全域大小的属性 u 和另外若干特征。这些特征如下。

（1）如果 $u=2$，即 $u=2^{2^k}$，$k=0$ 时，那么它是基础大小，只包含一个两位的数组 $A[0..1]$。也就是说，proto-vEB(2)由全域大小属性 2 和数组 $A[0..1]$组成。

（2）如果 $u\neq2$，即对某个整数 $k\geqslant1$，$u=2^{2^k}$，这时有 $u\geqslant4$。除了具有全域大小属性 u 以外，proto-vEB(u)还具有以下属性（见图 3.1）。

- 一个名为 summary 的指针，它指向一个 proto-vEB($\sqrt{u}$)结构。
- 一个数组 cluster$[0..\sqrt{u}-1]$，存储$\sqrt{u}$个指针，每个指针都指向一个 proto-vEB($\sqrt{u}$)结构。

当 $u\geqslant4$ 时，proto-vEB(u)中的数组 cluster$[0..\sqrt{u}-1]$中每个指针都指向一个 proto-vEB($\sqrt{u}$)结构。proto-vEB($\sqrt{u}$)结构的域$\{0,1,2,\cdots,\sqrt{u}-1\}$、它包含的关键字集合以及位向量 $A[0..\sqrt{u}-1]$都与 proto-vEB(u)的全域$\{0,1,2,\cdots,u\}$、包含的关键字集合以及位向量 $A[0..u-1]$有关。proto-vEB($\sqrt{u}$)是 proto-vEB(u)的递归结构，直观上，前者是后者的子结构。也就是说，proto-vEB(u)的所有信息便分别递归存储在由数组 cluster 的$\sqrt{u}$个指针指向的$\sqrt{u}$个簇中，每个簇保留原先 u 个信息中$\sqrt{u}$个信息。于是，全域中的元素 x（无论它是否为关键字），x（$0\leqslant x<u$）递归地存储在编号为 high(x)的簇中，作为该簇中编号为 low(x)的元素。$\text{high}(x)=\left\lfloor\frac{x}{\sqrt{u}}\right\rfloor$，即 high($x$)为不超过$\frac{x}{\sqrt{u}}$的最大整数；$\text{low}(x)=x \bmod \sqrt{u}$，即 low($x$)为 x 除以$\sqrt{u}$的余数。

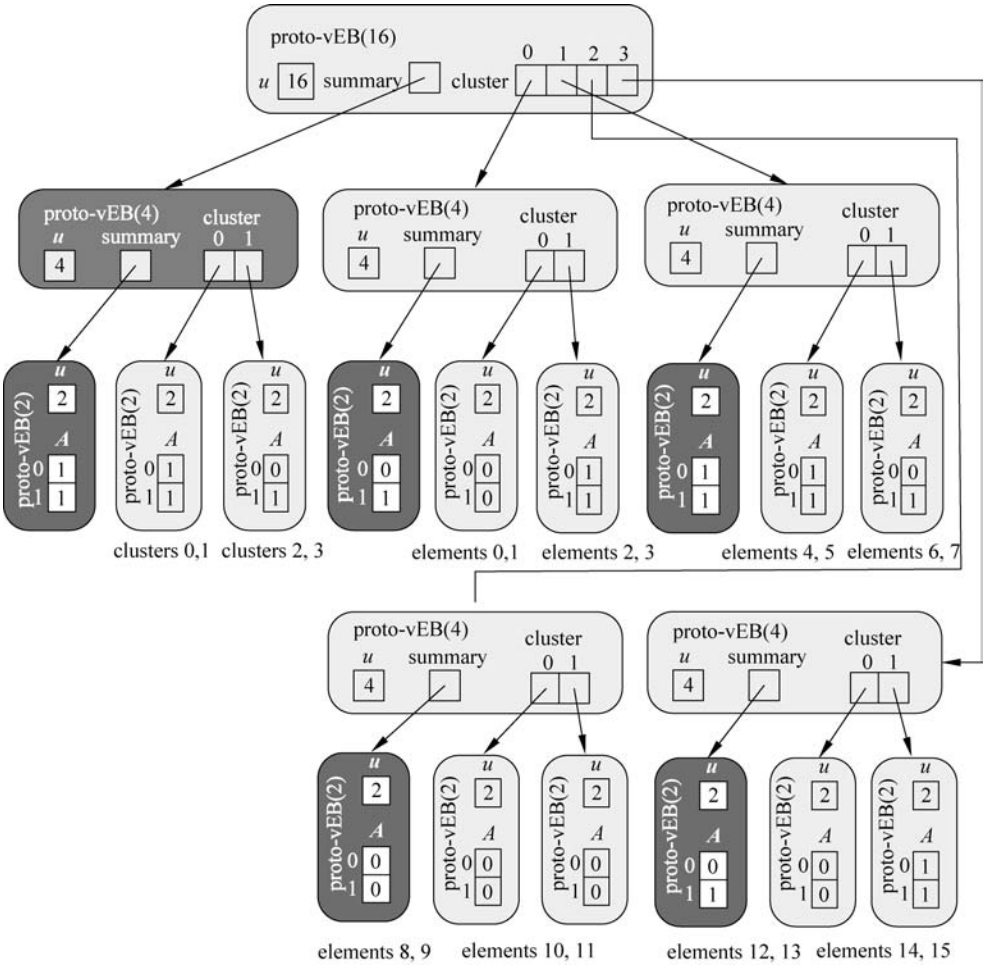


图 3.1　一个完全展开的 proto-vEB 结构①

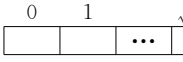
反过来，定义 index(x,y)$=x\sqrt{u}+y$，则 x=index(high(x),low(x))，即已知 x 所在的簇号 high(x)和它在该簇中的编号 low(x)，那么它就由公式$\sqrt{u}$high(x)+low(x)确定。

总之，如果 proto-vEB(u)是对应于全域{0,1,2,…,$u-1$}中动态位向量 A[0.. $u-1$]的结构，那么该结构中的数组 cluster[0..$\sqrt{u}-1$]中的第 0 个，第 1 个，…，第$\sqrt{u}-1$ 个指针指

① 一个 proto-vEB(16)结构表示了集合{2,3,4,5,7,14,15}，cluster[0..3]中的指针指向 4 个 proto-vEB(4)结构，summary 中的指针指向一个 summary 结构，它也是一个 proto-vEB(4)结构。每个 proto-vEB(4)结构在 cluster[0..1]中指向两个 proto-vEB(2)结构，以及指向一个 proto-vEB(2)结构的 summary。每个 proto-vEB(2)结构只包含一个两位的数组 A[0..1]。elements i,j 上方的 proto-vEB(2)结构存储实际动态集合的位 i 和 j，并且 cluster i,j 上方的 proto-vEB(2)结构存储顶层 proto-vEB(16)中的簇 i 和 j 的 summary 位。为清晰起见，深阴影部分表示一个 proto-vEB 结构的顶层，存储它的双亲结构的 summary 信息。(转摘于文献[27]图 20-4)

向的$\sqrt{u}$个 proto-vEB($\sqrt{u}$)结构分别对应于全域中动态位向量$\sqrt{u}$个子域中的动态位向量$A[0..\sqrt{u}-1]$,$A[\sqrt{u}..2\sqrt{u}-1]$,…,$A[(\sqrt{u}-1)\sqrt{u}..u-1]$的结构。虽然每个 proto-vEB($\sqrt{u}$)的域用{0,1,2,…,$\sqrt{u}-1$}表示,位向量用$A[0..\sqrt{u}-1]$表示,但实质上它们有上述对应关系,这是递归结构的精妙所在。由于这种表示法,所以前面我们定义 high(x)、low(x)和 index(x,y)等函数。借助这些函数,不难计算上述对应关系。

顾名思义,当$u\geqslant 4$时,proto-vEB(u)中的另一个属性 summary 指针,它指向的 proto-vEB($\sqrt{u}$)结构是 cluster 数组$\sqrt{u}$个指针指向的$\sqrt{u}$个簇中动态位向量中元素的梗概表示。具体地说,与 summary 指针指向的 proto-vEB($\sqrt{u}$)结构对应的全域{0,1,2,…,$\sqrt{u}-1$}中的动态位向量$A[0..\sqrt{u}-1]$是这样构成的:当且仅当第 x 簇中$\sqrt{u}$个元素中至少有一个是存储的关键字,则$A[x]=1$;否则$A[x]=0$。

综上所述,当$u\geqslant 4$时,proto-vEB(u)结构由 3 部分组成:u □,表示该结构大小属性;cluster 

,由$\sqrt{u}$个指针组成,其指向$\sqrt{u}$个 proto-vEB($\sqrt{u}$)结构,直观上,它们表示已经将原结构中全域动态位向量细分为$\sqrt{u}$个子域子动态位向量,并在其上重新构造子结构;而 summary 指针指向的是 cluster 指针指向的数据梗概结构。于是,proto-vEB(u)就凭借 cluster 和 summary 两个属性递归展开,直到基础结构 proto-vEB(2)为止。在每个递归结构中,既包含梗概,也包含细分数组,从而为动态存储集合的操作提供了快捷算法。

图 3.1 显示了一个完全展开的 proto-vEB(16)结构,它表示集合{2,3,4,5,7,14,15}。如果 i 在由 summary 指向的 proto-vEB 结构中,那么第 i 个簇包含了被表示集合中的某个值。cluster[i]表示 $i\sqrt{u}\sim(i+1)\sqrt{u}-1$ 的值,这些值形成了第 i 个簇。实际上,第 0 簇位向量[0,0,1,1]表示{2,3};第 1 簇位向量[1,1,0,1]表示{4,5,7};第 2 簇位向量[0,0,0,0]指明元素 8、9、10、11 都不在表示的集合中;第 3 簇位向量[0,0,1,1]表示{14,15}。于是,summary 结构位向量为[1,1,0,1],表示第 0 簇、第 1 簇、第 2 簇和第 3 簇的 summary 值分别为 1、1、0、1。

在基础层,实际动态集合的元素被存储在一些 proto-vEB(2)结构中,而余下的(见图 3.1 中深阴影部分)结构则存储 summary 位。在每个非 summary 基础结构的底部,数字表示它存储的位。例如,标记为 elements,6,7 的 proto-vEB(2)结构在$A[0]$中存储位 6(0,因为元素 6 不在集合中),并在$A[1]$存储位 7(1,因为元素 7 在集合中)。

与簇一样,summary 只是一个全域大小为$\sqrt{u}$的动态集合,而且 summary 表示为一个 proto-vEB($\sqrt{u}$)结构。图 3.1 中主 proto-vEB(16)结构的 4 个 summary 位都在最左侧的 proto-vEB(4)结构中,并且它们最终出现在两个 proto-vEB(2)结构中。例如,标记为 cluster2,3 的 proto-vEB(2)结构有$A[0]=0$,含义为 proto-vEB(16)结构的簇 2(包含元素 8,9,10,11)都为 0;并且$A[1]=1$,说明 proto-vEB(16)结构的簇 3(包含元素 12,13,14,15)至少有一个为 1。注意,每个 proto-vEB(4)结构都有指向自身的 summary,而 summary 自

已存储为一个 proto-vEB(2)结构。例如，查看标记为 elements0,1 左侧的那个 proto-vEB(2)结构，因为 $A[0]=0$，所以 elements0,1 结构都为 0；由于 $A[1]=1$，所以 elements2,3 结构至少有一个 1。

在 proto-vEB 结构上可以执行一些操作，包括判断一个值是否在集合中、查找最小数、查找最大数、查找后继和前驱等一系列查询操作，它们并不改变 proto-vEB 结构。还有两个修改 proto-vEB 结构的操作：插入一个元素和删除一个元素。如果全域大小为 u，则判断一个值是否在集合中操作的运行时间为 $O(\lg\lg u)$，不过其他操作在最坏的情况下，运行时间都超过 $O(\lg\lg u)$，许多操作的运行时间都为 $\Theta(\lg u)$。由此看来，不仅许多操作没有达到最优，而且该结构对全域大小 u 还做了很强的限制，即要求 $u=2^{2^k}$，k 为整数。因此，对于科学研究，我们势必要对该原型算法加以精化，得到更好的结构。

2) van Emde Boas 树

proto-vEB 结构已经接近运行时间为 $O(\lg\lg u)$的目标，缺陷是大多数操作要进行多次递归。此外，它对全域大小 u 做了很大限制，即 $u=2^{2^k}$，k 为整数。现在，就以它作为算法原型，设计一个类似于 proto-vEB 结构的数据结构，不仅要放宽对全域大小的规定，而且要去掉原型操作中的一些递归需求，从而达到运行时间为 $O(\lg\lg u)$的目标。修改后得到的结构便是 van Emde Boas 树数据结构，简称为 vEB 树。

首先，允许全域大小 u 为 2 的任何幂。如此放宽条件，$\sqrt{u}$ 可能不为整数。例如，当 $u=2^{2k+1}$(其中 $k\geqslant 0$ 为某个整数)为 2 的奇次幂时，$\sqrt{u}=2^{\frac{2k+1}{2}}$ 就不为整数。这时，把该数的 $\lg u$ 位分割成高 $\lceil \lg u/2\rceil$(即 $\lceil (2k+1)/2\rceil$)位和低 $\lfloor \lg u/2\rfloor$(即 $\lfloor (2k+1)/2\rfloor$)位。其中，$\lceil \lg u/2\rceil$($\lfloor \lg u/2\rfloor$)表示超过(不超过)$\lg u/2$ 的最小(最大)整数。为方便起见，把 $2^{\lceil \lg u/2\rceil}$ 记为 $\uparrow\sqrt{u}$，并称它为 u 的上平方根；把 $2^{\lfloor \lg u/2\rfloor}$ 记为 $\downarrow\sqrt{u}$，并称它为 u 的下平方根。显然，有 $u=\uparrow\sqrt{u}\downarrow\sqrt{u}$。特别地，当 $u=2^{2k}$($k\geqslant 0$ 为整数)为 2 的偶次幂时，有 $\uparrow\sqrt{u}=\downarrow\sqrt{u}=\sqrt{u}$。一般地，只假设 u 为 2 的任意一个非负整数幂。于是，重新定义前面介绍过的有用函数如下。

$$\text{high}(x)=\lfloor x/\downarrow\sqrt{u}\rfloor$$

$$\text{low}(x)=x \bmod \downarrow\sqrt{u}$$

$$\text{index}(x,y)=x\downarrow\sqrt{u}+y$$

其次，对 proto-vEB 结构进行修改，增加一些属性，以便存储更多信息。当全域大小为 u 时，修改后得到的 vEB 树记为 vEB(u)。

现在，我们讨论 vEB(u)树的结构。如果 u 不为 2 的基础情形，那么属性 summary 指向一棵 vEB($\uparrow\sqrt{u}$)树，数组 cluster[0..$\uparrow\sqrt{u}-1$]指向 $\uparrow\sqrt{u}$ 个 vEB($\downarrow\sqrt{u}$)树，如图 3.2 所示。并且，一棵 vEB 树还增添了 proto-vEB 结构中没有的以下两个属性。

(1) min 存储 vEB 树中的最小元素。

(2) max 存储 vEB 树中的最大元素。

值得强调的是，存储在 min 中的元素并不出现在任何递归的 vEB($\downarrow\sqrt{u}$)树中，这些树是

由 cluster 数组指向它们的。因此，在 vEB(u)树 V 中存储的元素为 V.min 再加上由 V.cluster[0..$\uparrow\sqrt{u}-1$]指向的递归存储在 vEB($\downarrow\sqrt{u}$)树中的元素。如此一来，当一棵 vEB 树中包含两个或两个以上元素时，实际上我们是以不同方式处理 min 和 max 属性的：存储在 min 中的元素不出现在任何簇中，而存储在 max 中的元素却不是这样的。

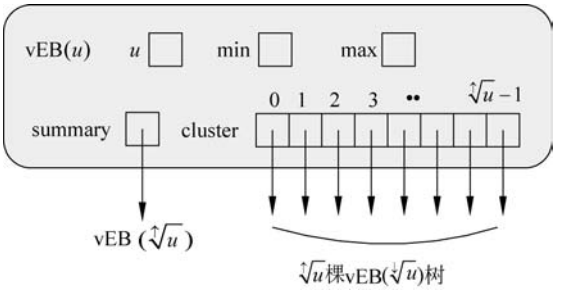


图 3.2　当 $u>2$ 时，一棵 vEB(u)树中的信息①

因为基础情形 u 的大小为 2，一棵 vEB(2)树中对应的 proro-vEB(2)结构并不需要数组 A。这是由于可以通过其 min 和 max 属性确定它的元素。在一棵不包含任何元素的 vEB 树中，不管全域的大小 u 如何，min 和 max 均为 NIL(无值)。

图 3.3 所示为一棵 vEB(16)树 V，包含集合{2,3,4,5,7,14,15}。因为最小的元素是 2，所以 V.min=2。根据上述构造方法，它不出现在 cluster 数组任意指针指向的簇中。也就是说，即使 high(2)=0，元素 2 也不会出现在由 V.cluster[0]所指向的 vEB(4)树中。注意到这时 V.cluster[0].min=3，果然元素 2 不在这棵 vEB 树中。

类似地，因为 V.cluster[0].min=3，虽然在 V.cluster[0]中“理应”包含元素 2 和 3，但根据结构表示约定，V.cluster[0]内的 vEB(2)簇为空。

min 和 max 属性以及关于 min 元素存储的“古怪”做法是减少 vEB 树上一些操作的递归调用次数的关键。这些操作原先在 proto-vEB 结构上大都要进行多次递归。min 和 max 属性有以下 4 方面的作用。

(1) 查找最小数和最大数操作甚至不需要递归，因为可以直接返回 min 和 max 的值。

(2) 查找后继和前驱操作也得到简化。例如，查找 x 的后继操作可以避免一个用于判断 x 的后继是否位于 high(x)簇中的递归调用。这是因为 x 的后继位于 x 簇中，当且仅当严格小于 x 簇的 max。

(3) 通过 min 和 max 的值，可以在常数时间内知晓一棵 vEB 树是否为空、仅含一个元素或两个以上元素。这种能力将在插入一个元素和删除一个元素操作中发挥作用。如果

① 结构包含大小为 u 的全域元素 min 和 max、指向一棵 vEB($\uparrow\sqrt{u}$)树的指针 summary，以及指向 vEB($\downarrow\sqrt{u}$)树的 $\uparrow\sqrt{u}$ 个指针数组 cluster[0..$\uparrow\sqrt{u}-1$]。(转摘于文献[27]图 20-5)

min 和 max 都为 NIL，则 vEB 树为空；如果 min 和 max 都不为 NIL 但相等，则 vEB 树仅含一个元素；如果 min 和 max 都不为 NIL 但并不相等，这时 vEB 树则包含两个或两个以上元素。

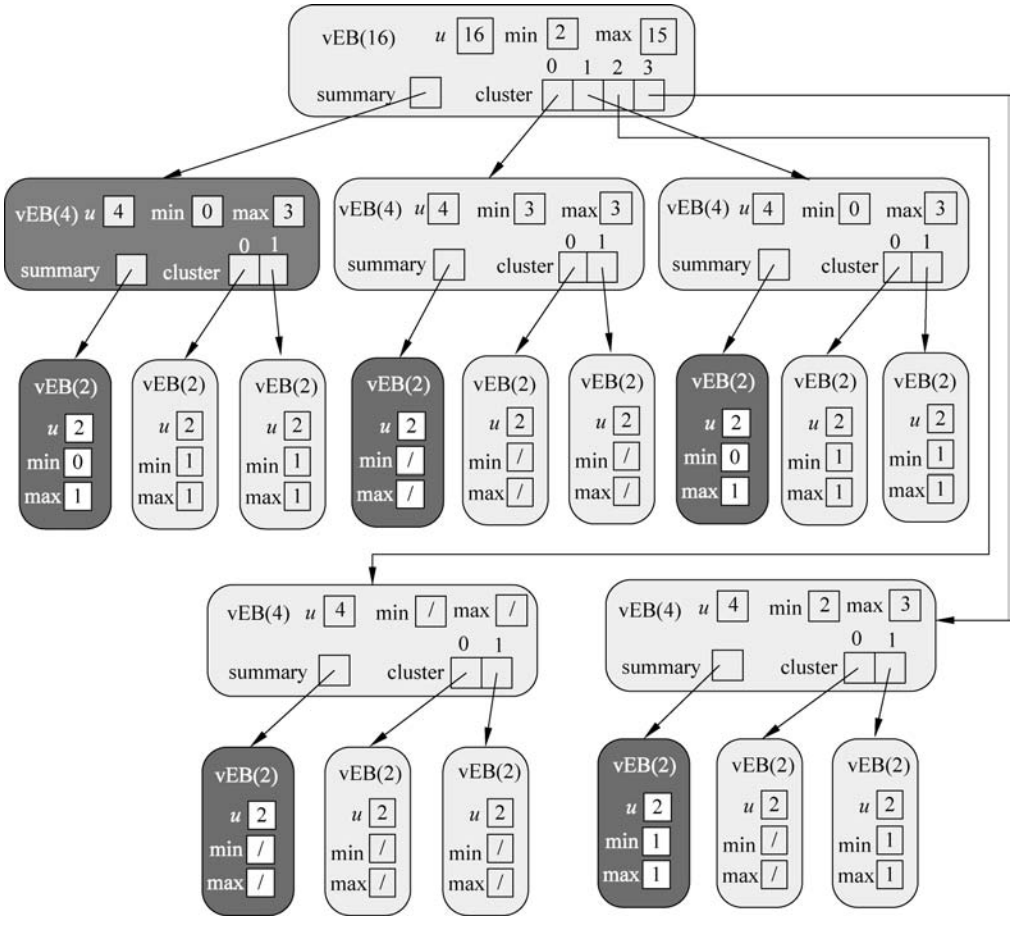


图 3.3　对应于图 3.1 中 proto-vEB 树的一棵 vEB(16)树[①]

(4) 如果一棵 vEB 树为空，那么只要更新它的 min 和 max 值就可以在常数时间内实现插入一个元素。类似地，如果一棵 vEB 树仅含一个元素，也只要更新 min 和 max 值就可以在常数时间内删除这个元素。这些性质可以缩减递归调用链。

3) 两个结构操作对比示例

vEB(u)树上的所有操作，在最坏情况下，运行时间至多为 $O(\lg \lg u)$。这与 proto-vEB(u)相比，确实达到了最优。现在以查找最小数和查找后继两个操作为例在两个结构上进行

① 它存储集合{2,3,4,5,7,14,15}。斜杠(/)表示 NIL 值。存储在 vEB 树中的 min 属性的值不会出现在它的任何一个簇中。这里的深阴影与图 3.1 的表示一样。(转摘于文献[27]图 20-6)

对比。

(1) 在原型 van Emde Boas 结构上查找最小数。

过程 PROTO-vEB-MINIMUM(V)返回 proto-vEB 结构中 V 中的最小元素,如果 V 代表的是一个空集,则返回 NIL。

```
PROTO - vEB - MINIMUM(V)
   1.  if V.u = = 2
   2.       if V.A[0] = = 1
   3.            return 0
   4.       elseif V.A[1] = = 1
   5.            return 1
   6.       else return NIL
   7.   else min - cluster = PROTO - vEB - MINIMUM(V.summary)
   8.       if min - cluster = = NIL
   9.            return NIL
   10.      else offset = = PROTO - vEB - MINIMUM(V.cluster[min - cluster])
   11.           return index(min - cluster, offset)
```

第 1 行判断是否为基础情形;第 2～6 行平凡地处理基础情形;第 7～11 行处理递归情形。首先,第 7 行查找包含元素的第 1 簇号。因为 V.summary 是一个 proto-vEB($\sqrt{u}$)结构,它包含 V.cluster 指向的各簇中是否有存储集合中元素的信息,所以在 V.summary 上递归调用 PROTO-vEB-MINIMUM 过程就可以找到最小数所在的簇号,第 7 行表明把递归调用的结果(即最小数所在的簇号)赋值给变量 min-cluster,如果集合为空,那么递归调用返回 NIL;如果集合非空,第 7 行递归调用的结果表明集合的最小元素就存在于编号为 min-cluster 的簇中。第 10 行中的递归调用是查找最小元素在这个簇中的偏移量,该偏移量指明最小数在它所在簇中的“位置”。最后,第 11 行调用函数 index(),通过最小数所在的簇号以及它所在簇中的偏移量计算出它的值,并返回。

查询 summary 信息允许我们快速地找到包含最小元素的簇,这是 summary 属性在结构中重要性的体现。虽然如此,由于查询最小数操作需要两次调用(第 7 行和第 10 行)proto-vEB($\sqrt{u}$)结构,所以根据算法理论,在最坏情况下运行时间会超过 $O(\lg \lg u)$。可以推出,这个操作的运行时间为 $\Theta(\lg u)$[①]。

(2) 在 vEB 树上查找最小数。

因为最小数存储在 min 属性中,所以查询最小数操作只有一行代码。

```
vEB - TREE - MINIMUM(V)
    1.  return V.min
```

顺便提一下,因为最大数存储在 max 属性中,所以查询最大数操作也只有一行代码。

① 具体推导参见《算法导论》[27]第 313 页。

```
vEB-TREE-MAXIMUM(V)
    1.  returnV.max
```

显然,这两个操作的运行只耗费常数时间。

(3) 在原型 van Emde Boas 结构上查找后继。

查找后继操作取一个参数 x 和一个 proto-vEB 结构 V 作为过程 PROTO-vEB-SUCCESSOR 的输入参数,过程 PROTO-vEB-SUCCESSOR(V,x)便返回 proto-vEB 结构 V 中大于 x 的最小元素;或者当 V 中不存在大于 x 的元素时,返回 NIL。注意,过程不要求 x 一定属于该集合,但假定 $0 \leqslant x < V.u$。

```
PROTO-vEB-SUCCESSOR(V,x)
   1.  ifV.u = = 2
   2.       if x = = 0 and V.A[1] = = 1
   3.            return 1
   4.       else return NIL
   5.  else offset = = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
   6.       if offset≠NIL
   7.            return index(high(x), offset)
   8.       else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
   9.            if succ-cluster = = NIL
   10.                return NIL
   11.           else offset = PROTO-vEB-MINIMUM(V.cluster[succ-cluster]
   12.                return index(succ-cluster, offset)
```

第 1 行判断是否为基础情形;第 2～4 行平凡处理:当 $x=0$ 且 $A[1]=1$ 时,才能在 proto-vEB(2)结构中找到 x 的后继;第 5～12 行处理递归情形。因为 x 所在簇的簇号为 high(x),在该簇内它的位置是 low(x),所以第 5 行表明是在 x 所在的簇内查找其后继,并将结果赋给变量 offset,offset 的直观意义是 x 的后继在该簇中的偏移量。第 6 行判断 x 所在的这个簇中是否存在 x 的后继。若存在,第 7 行计算 x 的后继值并返回它;否则,必须在其他簇中查找。这时要借用属性 summary 中的关于各簇是否包含元素的信息决定在哪个簇中查找 x 的后继。第 8 行通过递归调用本过程,在 V.summary 中查找 high(x)的后继簇号,并将它赋给变量 succ-cluster。第 9 行判断 succ-cluster 是否为 NIL,如果是,就意味着所有后继簇是空的,第 10 行返回 NIL;如果 succ-cluster 不为 NIL,第 11 行将编号为 succ-cluster 的簇中第 1 个元素(即该簇中最小元素)赋值给变量 offset,并且第 12 行计算并返回这个簇中的最小元素,它即为 x 的后继。

综上所述,在最坏情况下,PROTO-vEB-SUCCESSOR 在 proto-vEB($\sqrt{u}$)结构上做两次(第 5 行和第 8 行)自身递归调用和一次 PROTO-vEB-MINIMUM 调用(第 11 行),所以查找后继操作比查找最小元素操作运行时间更长。可以推得,运行时间为 $\Theta(\lg u \lg\lg u)$,因此查找后继操作渐近地慢于查找最小数操作。

(4) 在 vEB 树上查找后继,过程如下。

```
vEB-TREE-SUCCESSOR(V,x)
1.  if V.u = =2
2.      if x = =0 and V.max = =1
3.          return 1
4.      else return NIL
5.  elseif V.min≠NIL and x <V.min
6.      return V.min
7.  else max-low = vEB-TREE-MAXIMUM(V.cluster[high(x)])
8.      if max-low≠NIL and low(x) < max-low
9.          offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10.         return index(high(x), offset)
11.     else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
12.         if succ-cluster = =NIL
13.             return NIL
14.         else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
15.             return index(succ-cluster, offset )
```

这个过程有 6 个返回语句和几种情形处理。第 1 行判断是否是基础情形,若是,第 2～4 行便处理它。这时如果查找的是 0 的后继并且 1 在集合中,那么第 3 行返回 1;否则第 4 行返回 NIL。

如果不是基础情形,第 5 行接着判断 x 是否严格小于最小元素。若是,那么第 6 行返回这个最小元素。

如果不是基础情形,并且 x 大于或等于 vEB 树 V 中的最小元素值,则第 7 行把 x 簇中的最大元素赋值给 max-low。顺便提一下,由于 min 属性的巧妙规定,max-low 若存在,就绝不会等于 vEB 树 V 中的最小值。如果 x 簇中存在大于 x 的元素,那么可确定 x 的后继必在 x 簇中,这时 max-low 肯定存在。实际上,第 8 行测试这种情况。如果测试结构是肯定的,那么第 9 行便确定 x 的后继在该簇中的位置,接着第 10 行进行计算并且返回计算值。

如果 x 大于或等于 x 簇中的最大元素,意味着 x 的后继不在 x 簇中。于是程序进入第 11 行,利用 summary 属性包含的信息,查找 x 簇的后继簇,并把它赋给变量 succ-cluster。如果 succ-cluster 经第 12 行判断为 NIL,于是第 13 行返回 NIL,表示集合中没有 x 的后继;否则第 14 行计算簇号为 succ-cluster 的簇中最小值,并把它赋给变量 offset,最后第 15 行便从 x 的后继所在的簇号及其在该簇中的位置计算出后继数并返回。

回忆在原型上的查找后继操作,PROTO-vEB-SUCCESSOR(V,x)要进行两个递归调用:一个是判断 x 的后继是否和 x 一样被包含在 x 的簇中,如果不包含,另一个递归调用就是要找出包含 x 后继的簇。在最坏的情况下,这两次递归调用都必须执行。

但是在 vEB 树上执行后继操作就不同了。虽然表面上看起来程序中也有两处标明了递归调用,但是由于能在 vEB 树中很快地访问最大值(第 7 行),这样就可以避免进行两次

递归调用。实质上，过程 vEB-TREE-SUCCESSOR(V,x)只进行一次递归调用，或是在簇上的(第 9 行)，或是 summary 上的(第 11 行)，并非两者同时进行或是如同原型结构上在最坏情况下那样两者都要执行。也就是说，根据程序第 9 行测试的结果，x 的后继或者在 x 簇内，或者在其他簇内，两种情况互斥。于是，过程不是在第 9 行(在全域大小为 $\sqrt[\downarrow]{u}$ 的 vEB 树上)就是在第 11 行(在全域大小为 $\sqrt[\uparrow]{u}$ 的 vEB 树上)对自身进行递归调用。不管哪种情况，一次递归调用是在全域大小至多为 $\sqrt[\uparrow]{u}$ 的 vEB 树上进行，至于过程的剩余部分，包括调用 vEB-TREE-MINIMUM 和 vEB-TREE-MAXIMUM，耗费时间为常数时间。所以 vEB-TREE-SUCCESSOR 的最坏情况运行时间为 $O(\lg \lg u)$。

查找前驱过程和查找后继过程是对称的，不过查找前驱比查找后继要多处理一个附加情况。因为 x 的后继不在 x 的簇中，就在有更高编号的簇中，查找后继只考虑这两种情况。同样，查找前驱也要考虑 x 的前驱可能在 x 的簇中，或者在更低编号的簇中，但是由于 min 的规定，如果 x 的前驱是 vEB 树中的最小元素，那么 x 的前驱就不存在于任何一个簇中，必须检查这个条件，并返回其前驱值。由于树中明确存储了 min 值，所以这个多处理的情况并不影响它的渐近运行时间，这样查找前驱操作最坏情况运行时间也为 $O(\lg \lg u)$。这就是我们在开始讨论这个专题时，提到的 Pătraşcu 和 Thorup 表明的事实。

3.2 图形

在 3.1 节示例中，我们看到在开发过程中，抛弃式原型并不一定要求原型是可执行的，如纸上的系统用户界面模型和 Wizard of Oz 原型。前者只是在纸上演绎脚本，后者虽然开发了用户界面，可使用户与此界面交互，但真实交互却是由一个隐藏的人在幕后代替系统回答客户请求才得以进行的。这就在实质上说明，原型方法在软件开发中发挥的作用，也可以通过图形方法得到。

3.2.1 图形在需求分析中的作用

规格说明文档是客户和开发组织之间的合同，它明确规定产品必须做什么、产品必须要满足的约束以及产品的验收标准。在需求分析阶段，开发组织必须给出完整详细且能被客户清晰理解的产品规格说明书。

将图形应用于软件的规格说明是 20 世纪 70 年代的一项重要技术。有 3 种使用图形的技术非常流行：DeMarco 方法、Gane 和 Sarsen 方法、Yourdon 和 Constantine 方法。采用这些图形技术，获取产品规格说明书曾给软件业带来好处。例如，Gane 和 Sarsen 方法采取 4 种符号(见图 3.4)绘制数据流图(Data Flow Diagram，DFD)，借此确定系统的逻辑数据流(Logical Data Flow)以及与它相对应的物理数据流(即发生了什么与如何发生相对应)。

我们知道，编写完全正确的产品规格说明书是一件很不容易的事情，自然语言不是一个规定产品的好方法，用图形方法进行结构化系统分析替代它可能是一个好主意。Gane 和

Sarsen 利用图形技术进行结构化系统分析，它是 9 个步骤的技术：①画数据流图；②决定哪部分计算机化以及如何计算机化(批处理或联机处理)；③确定数据流的细节；④定义处理的逻辑；⑤定义数据存储；⑥定义物理资源；⑦确定输入-输出规格说明；⑧确定大小(如文件的大小)；⑨确定硬件要求。

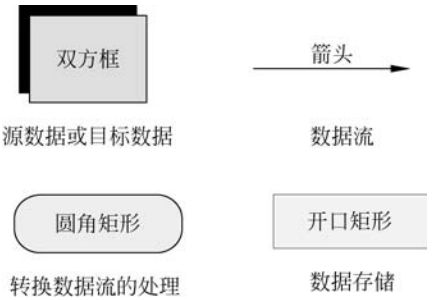


图 3.4　Gane 和 Sarsen 的结构化系统分析符号
(转摘于文献[22]图 11-1)

当然，Gane 和 Sarsen 方法并不能解决所有问题，它和其他用于分析或设计的方法一样，都具有一些明显缺陷，如不能用于确定响应时间和定时问题。但是，利用该方法，可以分析客户的要求。最重要的一点是，在这 9 个步骤中多次用到逐步求精，如步骤③～步骤⑤，深化对客户要求的阐明，从而明确系统今后设计的方向[22]。

按照 Dart 等的分类，Gane 和 Sarsen 方法利用图形化方法对待开发系统做了结构化系统分析，属于半形式化规格说明技术。还有其他利用图形化方法的半形式化规格说明技术，如 Ross 的结构化分析与设计技术(Structured Analysis and Design Technique，SADT)。SADT 由两个相互关联的部分组成：一部分是被称为结构化分析(SA)的方框-箭头图形化语言；另一部分是设计技术(DT)。SADT 中蕴含的逐步求精的程度比 Gane 和 Sarsen 方法更高，它已经成功地用于范围广泛的产品规格说明，特别是大型和复杂的项目，不过它不太适用实时系统[22]。

使用形式化的规格说明技术比使用半形式化或非形式化技术更可能得到精确的规格说明。Meyer 建议使用数学术语形式化地表达规格说明。因此，利用图形化技术进行形式化表述自然是有效方式，在某种意义上，诸如状态图(State Chart)和 Petri 网等都是图形化形式化技术，状态图的功能非常强大，并且由一个 CASE 工作平台 Rhapsody 支持。这个方法是有穷状态机(Finite State Machine，FSM)的扩展，已经成功地应用于一些大型实时系统。众所周知，说明并发系统的一个主要困难是处理定时问题，如果对定时没有正确地拟制规格说明，会引起不好的设计，进而引发错误的实现。Petri 网是一个功能强大的技术，它不仅能够很好地说明隐含有定时问题的系统，而且还具有一个大优点，即可以用于设计[22]。

20 世纪 50 年代，认知心理学家乔治·米勒有一个著名发现：人类只能记住和处理 7 项左右内容，即 7±2，这通常称为“米勒魔数”。人们有意识地努力拥护米勒法则。例如，提出 SADT 技术的 Ross 指出：“对值得表述的任何事情所表述的每件事情，必须用 6 个或更少

的词表达。”人们也有意识地努力绕过米勒法则，解决人脑的基本限制。其中图形表示是最好的方法，图形是信息的视觉表现方式，如前所述，在软件界，它已经获得很好的应用。

1997 年，Booch、Jacobson 和 Rumbaugh 联合工作，推出 UML 1.0 版，在软件工程领域掀起风暴。在这之前，还没有开发出软件产品一致接受的符号。几乎一夜之间，全世界都在使用 UML。随着 UML 国际标准的制定，UML 已成为无可争辩的表示面向对象软件产品的国际标准符号。软件开发领域如同音乐领域一样，在其中文字描述不能代替图示。现今，“统一软件开发过程”采用的最好建模语言便是 UML[22]。

为了得到对需求更深的理解，以及为了按需求描述得到的设计和实现易于维护，统一过程采用用例驱动方式进行需求分析。用例以软件产品的类描述。统一过程有 3 种类：实体类、边界类和控制类。实体类为长期存在的信息建模；边界类为软件产品和它的参与者之间的交互行为建模，通常与输入和输出相关；控制类为复杂的计算和算法建模。UML 并不是一种方法，它只是一种专门用于以可视化方式设计软件系统的语言。于是，我们可以使用 UML 用例、类图、注解、用例图、交互图、活动图、包、组件图和部署图建模软件。UML 的优点是允许定义额外的结构，该结构不是 UML 的一部分，却是准确地为待定系统建立模型所必需的。换言之，作为一种语言，与所有语言一样，它只是表达思想的工具，并不会限制该语言可以描述的思想的类别和被描述的方式，所以它是可以与任何方法结合使用的可视化符号体系。甚至，文献[22]指出，UML 不是一般的符号，它就是我们所需要的符号，很难想象，一本现代的关于软件工程的书不使用 UML 描述软件。

虽然 UML 为需求建模奠定了合理基础，但是乔伊·贝迪(Joy Beatty)和安东尼·陈(Anthony Chen)认为，UML 不满足需求建模的全部要求，原因是它缺少有关需求与业务价值的模型，缺少从最终用户的角度展示系统结构的模型。此外，UML 在技术上过于复杂，业务项目干系人难以掌握，原因是它的模型侧重于软件系统的架构建模。他们认为，当一个模型只聚集于解决问题的一个或两个方面时是最有用的。而当一个模型具有许多类型的信息或模型的语法规则过于复杂难以理解，项目干系人就绝对不会用。模型的复杂性是造成大型企业不用一些现成建模语言的主要原因之一[20]。

鉴于 UML 只是用于描述系统的技术设计和结构，而且过于复杂难以掌握，乔伊和安东尼创立了需求建模语言(RML)，它是为建立需求视觉模型而专门设计的语言。RML 不是一种学术上的建模语言，它的开发，一是为了弥补现有模型在功能上的缺陷；二是为了易用性，便于企业管理、业务和技术等项目的干系人使用。应用一个模型只聚集于解决问题的一个或两个方面的原则，乔伊和安东尼专门为软件需求建模设计一套完整的模型，这对于常常搞不懂复杂模型的项目干系人更容易接受。乔伊和安东尼的主要想法是用不同的可视化模型发现、检验和分析需求，为了确认软件需求，他们创造了图形化解决方案，帮助项目干系人理解解决方案交付什么结果和不包括什么。这套完整模型主要有 22 种可视化模型，分为 4 个主要需求类别，即目标、人员、系统和数据。

在每个类别中，都有一个绑定模型，它可能捕获用于创建该类模型的所有信息。目标、人员、系统和数据 4 类模型的绑定模型分别为业务目标模型(目标)、组织结构图(人员)、生

态系统图(系统)和业务数据图(数据)。合理使用每个类别的绑定模型,界定分析范围,创建全面信息的基础,然后随着分析进展到更详细的 RML 模型。

目标模型描述了系统的业务价值,并根据系统的价值设置功能和需求的优先级。在概念上,目标模型最接近传统的需求。在项目早期,目标模型帮助项目干系人认同项目的业务价值,在项目进行中,这些模型可以帮助找出哪些业务目标没有对应的需求,以便增加这些需求;也可以帮助找出没有创造明显价值的要求,以便缩减项目规模。人员模型描述了系统的干系人,以及他们的业务流程和目标。与用户的交流会创建更多人员模型,描述用户希望如何使用系统以及期望得到的输出结果。系统模型描述存在什么系统、用户界面样式、怎样与系统互动以及系统如何表现。这些模型确定在"生态系统"(即多系统环境)中运行的主要应用程序,以及系统之间、人机之间的接口、界面和系统自动化过程等事件。数据模型描述从最终用户的角度看待的业务数据对象之间的关系,包括数据的生命周期以及如何利用数据做决定。其中有关的数据操作可以有规律地定义需求,进而定义用户能够与数据交互的具体方式[20]。

因为 4 个类别的模型是从不同的视角分析解决方案,所以分析大多数解决方案通常需要所有 4 种模型。特别地,现今软件界共同的问题是在软件开发中没有针对整体价值进行功能分析,资金浪费在增加没人用的功能上,成本远大于收益。由于在每个 RML 类别的模型开发上,能确保对解决方案清晰了解,因此可以最大限度地增加团队开发出正确软件的机会。综上所述,综合运用 4 种类型需求模型,比一般传统分析方法更能正确获得需求。

Ian Alexander 为《软件需求与可视化模型》一书写了推荐序。在序中,他指出,对于软件需求工作,学术界所想的与工业界实际所做的之间存在巨大差距。由于乔伊和安东尼是实干家,又熟悉研究人员的工作。对于需求工程,他们正视传统的分析,认为旧的分析也许不全面但未必是错误的。他们开发 RML,用于建立需求的可视化模型,收集和规范了工业界中普遍使用的最佳实践模型。"需求模型之间存在必然的复杂性,它们相互依存。目标与功能有关;功能与流程有关;流程与用例有关;用例与用户接口有关。乔伊和安东尼展示如何调整需求模型结构(也可称为元结构)以适应不同的项目。他们已经无数次地检验并证明他们是成功的。"

3.2.2 图形在形式表示中的应用

通过 3.2.1 节,我们看到在软件开发方法学中存在可视化表示法的趋势,如软件界开发了 UML 和 RML 等可视化建模语言。实际上,这种趋势早就存在了。软件是一个形式系统,它高度抽象,以代码方式存在。认知科学告诉我们,人们思考形式问题时,总是以一种抽象和直观的混合方式进行着[35]。对于软件系统也不例外,在其整个开发流程中,人们并不局限在严格的逻辑框架下,总是以某种直观方式考查软件程序。在此背景下,计算机科学和软件工程领域便发明了许多可视化方法。可视化方法的好处是不言而喻的,一旦我们把呆板的语法映射成图形化对象之间的关系,原来静止的代码便能提供一些生动的、不同的信息,如不同的程序对象以及它们之间的关联、控制流和通信模式等。

以 Petri 网为例说明图形在形式表示中的作用。Petri 网融合了严格的数学表述和直观的图形表达，被证明可有效地描述并发关系活动，并在系统性能分析、通信协议验证等方面都得到广泛应用。

一般地，一个 Petri 网可以表示为 4 元组 $N=(P,T,F,M)$。

$P=\{p_1,p_2,\cdots,p_m\}$是一个有穷集合，$m\geqslant 0$。P 中的元素 p_i 称为“库所”(Place)，在图中用圆圈表示。

$T=\{t_1,t_2,\cdots,t_n\}$是一个有穷集合，$n\geqslant 0$。T 中的元素 t_j 称为“变迁”(Transition)，在图中用细线表示。

$F\subseteq(P\times T)\cup(T\times P)$是有向边的集合。有向边用来连接库所和变迁，在图中用带箭头的弧线表示。

当有一条边连接库所 p_i 到变迁 t_j 时，即箭头从 p_i 指向 t_j，p_i 称为 t_j 的输入库所；当有一条边连接变迁 t_j 到库所 p_i 时，即箭头从 t_j 指向 p_i，p_i 称为 t_j 的输出库所。

注意，库所和变迁是两类不同的元素，有向边只建立了从库所到变迁、从变迁到库所的单方向联系，即同类元素之间是不能直接联系的。此外，还要求每个库所或变迁必须通过有向边和其他(不同类的)元素关联，即 N 中不能有孤立元素。

$M:P\rightarrow\{0,1,2,\cdots\}$是从库所集到非负整数集的一个函数。在图中，$M$ 的具体含义是：每个库所可以用令牌(Token)标记或不标记，即每个库所可能不拥有令牌，也可能拥有令牌，如有令牌，令牌数为任意正整数。库所中的令牌用小的实心圆圈表示。

标记(Marking)一个 Petri 网是给该网分配令牌。为了简单起见，我们只讨论基本网络系统，它在每个库所的令牌数量不多于一个。在这个情况下，通常当一个变迁的所有输入库所都被标记且它的输出库所都没有被标记时，这个变迁将被允许(Enabled)。只有被允许后，变迁才能被激活。若变迁被激活(即得到执行)，这时，输入库所上的令牌被消耗，同时为输出库所产生令牌。Petri 网是不确定的，也就是如果能够激活多个变迁，那么它们中的任意一个都可以被激活。

Petri 网可以用于规格说明，也可以用于设计，甚至可以用于程序验证。Petri 网利用 3 个基本元素：库所、变迁和边，将系统的行为表示为状态及其变化，变迁描述了改变系统状态的事件。

Petri 网的(全局)状态 G 是 P 中有标记的库所组成的子集，也就是说，不在 G 中的库所没有令牌。在 Petri 网所有可能的状态中，其中某个状态作为初始状态。若激活一个变迁，随着令牌由一个库所向另一个库所转移，Petri 网的状态也就相应地发生变化。Petri 网的一次执行就是从初始状态开始的最大状态序列，它通过一次次变迁激活，获得这个状态序列中相继状态的更替。一旦确定了初始状态，虽然只有变迁促使序列从初始状态变更，但是由于变迁的激活并不是确定的，因此我们可以得到 Petri 网所有不同的执行过程。

举个例子，当多个进程争夺同一个资源时，往往会发生互斥情况。Dijskstra 阐述了如何通过互斥协议的临时尝试序列合理地解决进程间的互斥问题[16]。在这个问题中，两个(或两个以上)进程竞争进入一个临界区(Critical Section)，在程序中，即它们访问某个互斥

资源的代码段，这个访问过程是排他的。例如，临界区涉及打印，显然，两个进程不能同时访问同一台打印机。

如果一个进程不需要访问临界区，则它会进行任意的本地计算，本地计算位于非临界区中。当进程要进入其临界区，或者已经获取了访问临界区的权限，该进程都可能会加入(不同的)临界区协议，这些协议要保证达到以下两个目的。

(1) 斥性(Exclusiveness)：任意两个进程不能同时进入其临界区。

(2) 活性(Liveness)：如果一个进程想要进入其临界区，则该进程能在有限时间内被允许进入。

下面是一个这种协议的临时尝试。

```
                    boolean c1,c2 initially 1;
P1 :: m1 : while true do                  P2 :: n1: while true do
      m2 : ( * noncritical section 1 * )          n2  : ( * noncritical section 2 * )
      m3 : c1 := 0;                               n3 : c2 := 0;
      m4 : wait until c2 = 1;                     n4 : wait until c1 = 1;
      m5 : ( * critical section 1 * )             n5 : ( * critical section 2 * )
      m6 : c1 := 1                                n6 : c2 := 1
end                                       end
```

这个用抽象形式描述的临时互斥协议所表达的安全属性可以用 Petri 网模型可视化。Petri 网如图 3.5 所示，被标记的库所表示系统的初始状态。使用 Petri 网可以很方便地检验协议中的安全属性，即检验协议能否保证系统做到：无法从初始状态到达“critical section 1 和 critical section 2 都有令牌”的状态。

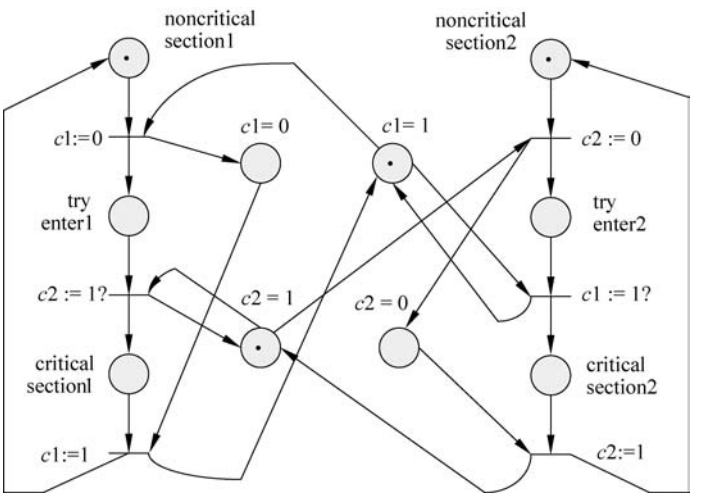


图 3.5　临时互斥算法的 Petri 网模型

(转摘于文献[16]图 11.11)

根据 Petri 网激活规则，可以通过移动令牌演示这个算法死锁的可能性。由于死锁将在没有可以激活的变迁时发生，因此我们检验这种情况能否出现。当被标记为 $c1:=0$ 的变迁被激活时，去掉 noncritical section 1 和 $c1=1$ 库所的令牌，并在 try enter 1 和 $c1=0$ 库所加入令牌。与之对等，当被标记为 $c2:=0$ 的变迁被激活时，去掉 noncritical section 2 和 $c2=1$ 库所的令牌，并在 try enter 2 和 $c2=0$ 库所加入令牌。至此，此 Petri 网中不存在可以激活的变迁了，如图 3.6 所示。

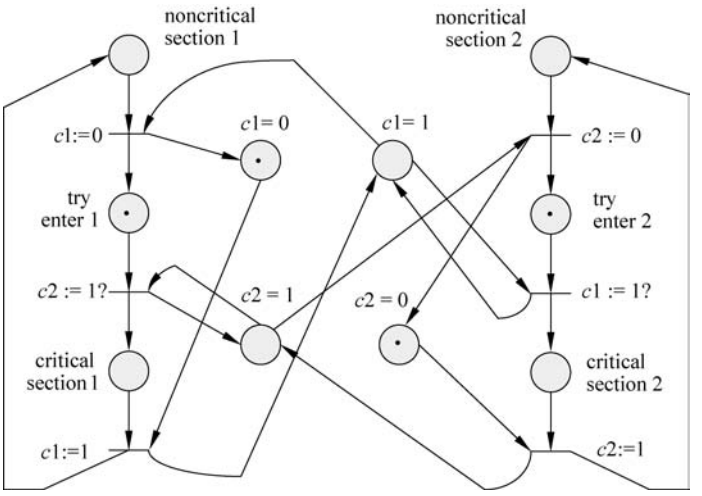


图 3.6　临时互斥算法的 Petri 网的死锁检验图

注意，Petri 图中 $c2:=1$ ？($c1:=1$?)变迁意指当 $c2=1$($c1=1$)事件发生时，系统的状态才可能发生改变。图 3.5 显示，因为 $c2=1$ 库所没有令牌，所以标记为 $c2:=1$ ？的变迁不能执行。同样，因为 $c1=1$ 库所没有令牌，所以标记为 $c1:=1$? 的变迁也不能执行。这样，两个进程都无法进入各自的临界区。因此，临时互斥协议可以保证系统的安全属性，即系统将不会从初始状态到达“critical section 1 和 critical section 2 都有令牌”的状态。由此观之，我们要设计算法，解决上述会出现的死锁问题。例如，荷兰数学家 Dekker 给出一个解，如图 3.7 所示[16]。进一步，在理论上，我们不仅要保证这些临界区和非临界区从不终止，即都可以无限次执行，还要考虑公平性原则，即两个进程中任意一个进程都不会被另一个进程的行为而永远推迟自己的转换。

除了 Petri 图，Peled 在其著作《软件可靠性方法》[16]中介绍了其他可视化方法在形式表述中的应用。

1. 消息序列图

每个消息序列图(Message Sequence Chart，MSC)描述了一个涉及进程间通信的场景，这些场景描述了消息的发送和接收及其顺序。

图 3.8 所示为一个简单消息序列图示例。在序列图中每个进程被表示成一个竖线，竖线上方的方框内标记进程名。消息被标记为横向箭头，它从消息发送方指向接收方。注意，

消息序列图刻画了一些消息发送和接收事件，一般情况下，它忽略对其他事件（如判定和赋值事件）的刻画。

```
                boolean c1 initially 1;
                boolean c2 initially 1;
            integer(1..2) turn initially 1;

P1 :: while true do                    P2 :: while true do
  begin                                  begin
    noncritical section 1                  noncritical section 2
    c1 : = 0;                              c2 : = 0;
    while c2 = 0 do;                       while c1 = 0 do;
      begin                                  begin
      if turn = 2 then                       if turn = 1 then
        begin                                  begin
          c1 : = 1;                              c2 : = 1;
          wait until turn = 1;                   wait until turn = 2;
          c1 := 0;                               c2 := 0;
        end                                    end
      end;                                   end;
    critical section 1                     critical section 2
    c1 := 1;                               c2 := 1;
    turn := 2                              turn := 1
end                                    end
```

图 3.7　Dekker 的互斥解

（转摘于文献[16]图 4.4）

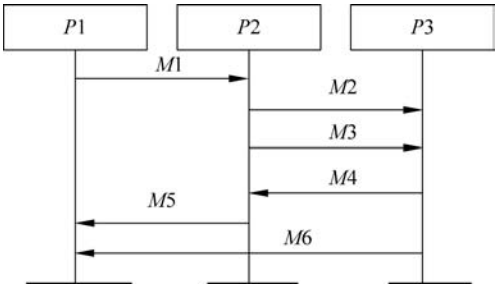


图 3.8　一个简单的消息序列图

（转摘于文献[16]图 11.1）

每个消息序列图对应一个图$(S,<)$，其语义解释如下：假定消息序列图中有两个事件$p,q\in S$，$p<q$ 表示 p 在 q 之前，其中序关系可以是因果关系（如发送事件 p 和相应的回复事件 q）、控制关系（如在同一进程队列中 q 是一个消息发送事件，但它必须等待 p 事件发生才可执行）和先进先出（First In First Out，FIFO）顺序（如在同一进程序列，p 和 q 是两个接收事件，且 p 出现在接收事件 q 之前；而它们相应的发送事件 p' 和 q' 也在同一进程队列中，且 p' 在 q' 之前）。如此，单独的消息序列图表示了事件的一个偏序集。

消息序列图可以用于描述一个系统典型或异常执行的通信结构以及在测试或模型检验中找到的反例。在消息序列图上还可以应用一些简单的验证算法，如检查消息序列图中是

否存在竞争条件。例如，在图 3.8 中，进程 $P1$ 包含了两个消息接收事件 $M5$ 和 $M6$。进程队列(Process Line)都是一维的，$P1$ 必须确定选择先处理其中一个接收事件，而把另一个放在后面。由于 $M5$ 和 $M6$ 分别是由 $P2$ 和 $P3$ 两个不同进程发送的，它们的到达时间顺序可能无法确定，因此我们并没有理由确定接收的消息会以特定的顺序到达。这便是竞争条件产生的直观背景，即它是包括至少一个接收事件的一组事件，我们只对其顺序进行了有限的控制。形式上，可以将竞争定义为消息序列图中有这样两个事件 p 和 q：①p 和 q 出现在同一个进程队列中；②p 出现在 q 之前；③在图$(S,\prec)$中不存在从 p 到 q 的路径。根据定义，在消息序列图中检测竞争情况就很简单。只要计算消息序列图对应图$(S,\prec)$中所有事件关于$\prec^*$关系的传递闭包。对于事件 $p,q\in S$，仅当在图$(S,\prec)$中存在从 p 到 q 的路径时，有 $p\prec^* q$。然后，只须比较在同一进程队列中的两个事件的传递闭包关系。如今，在描述通信协议的执行方面，消息序列图方法十分流行。它是众多用来设计通信系统的标准描述技术之一，越来越多的工具提供了消息序列图接口。

2. 程序形式化表示的可视化

1) 可视化流程图和状态机

程序形式化表示的可视化将一个程序的运行过程用状态机或流程图表示，演示从一个状态(或节点)到另一个状态(或节点)的转换过程，是理解程序并检验其正确性的有效方式。

通过高亮当前节点(改变节点的颜色或色调)可以仿真执行过程。当执行一个转换时，当前节点的后继节点会替代前者高亮显示出来。

有许多不同的方式可以构建一个系统的图形化表示。例如，可以使用编辑器将系统的代码翻译成图。目前已出现一些工具，它们把生成图和修改图作为系统设计的一部分供开发人员使用。

在系统设计、测试和验证的许多阶段，都可以使用程序的可视化表示。例如，系统的设计可以从使用一些可视化工具开始，有些工具可以自动生成可执行代码。即使无法直接使用这些自动生成的代码，也可以通过修改它开始对目标系统的开发。例如，图中节点或边可以包含附加信息，这些信息可以是给变量赋值的实际代码、判定谓词或发送/接收的消息。

2) 层次状态图

使用层次状态图可以表示系统的层次化结构以及系统的并发组合特征。当状态空间巨大时，使用层次状态图更能达到可视化直观的效果。已经有图符号体系描述系统的层次状态图。STATECHARTS 是其中一种，它允许将若干状态(此时应叫作子状态)组合成超状态(Super State)，因此在刻画系统的层次化结构、并发等特征的同时，能简化普通状态图表示。例如，利用超状态之间的转换代替它们内部子状态间的转换，减少了普通状态图冗余表示，尤其适用于指定中断。

3) 程序文本着色可视化

虽然上述许多图形式化方法能显示程序的许多属性，但往往由于状态空间过于庞大，致使图形复杂，降低了直观性。这从图 3.5 就可以看出，一个简单的临时互斥协议画成 Petri 网，线条就很多，这还是初始状态图。因此，图的显示是一个困难的问题。关于图的显示，一

直是计算机界研究的课题。

使用颜色使程序代码可视化是一种有趣的方法。该方法使用不同的色调表示不同等级的代码。例如，黄色表示低等级值；红色表示中等级值；棕色表示高等级值；等等。而代码等级划分主要依据软件界长期实践积累的经验和直觉，根据某个观念对程序代码进行分级。一般地，我们是从软件测试和验证角度制定等级标准的。如果依据代码块改变的频繁程度和容易出现错误的可能性大小对程序的代码块进行等级划分，深颜色代码块和浅颜色代码块相比，是改变更频繁、更容易出错的代码块，这就预示我们要特别注意对这些代码块的正确性进行验证。另外，如果我们按能被测试用例覆盖的多寡数量着色程序代码块，那么对于浅颜色代码块，因为覆盖它们的测试用例数量相对较少，当程序错误原因一时难以确定时，我们应当为这些代码块设置新的测试用例。这样看来，对程序文本着色，是对软件界长期经验的有效应用，这对软件开发、验证和测试都是有益的。

3.2.3 图形在形式证明中的应用

1. Manna-Pnueli 演绎规则方法

Z. Manna 和 A. Pnueli 在 20 世纪 80 年代初提出了并发程序安全性（不变性、优先性等）、活性（响应性、反应性等）的演绎推理规则[36]。

1）不变性规则

基本不变式规则（INV-B）为

$$\frac{\begin{array}{l}\text{B1. } \Theta \rightarrow \varphi \\ \text{B2. } \{\varphi\}\,\mathcal{T}\,\{\varphi\}\end{array}}{\Box\varphi}$$

B1：Θ 为初始条件，φ 为一个断言，$\Theta\rightarrow\varphi$ 表示 φ 在初始状态成立。

B2：$\{\varphi\}\mathcal{T}\{\varphi\}$ 表示 φ 在执行过程中的某个状态成立，$\mathcal{T}$ 为某个迁移（即转换）集，这时无论 $\mathcal{T}$ 中哪个迁移，φ 在下一状态依然成立。

有这两个前提的保证，就可以得出 φ 在所有状态成立，即 $\Box\varphi$ 成立。符号 $\Box$ 是时态运算符，表示“总是”（Always）。这时称 φ 为不变式。

基本不变式规则还可以扩充或变形为其他类型的不变式规则。

2）优先性规则

$$\frac{\begin{array}{l}\text{N1. } p \rightarrow \bigvee_{i=0}^{r}\varphi_i \\ \text{N2. } \varphi_i \rightarrow q_i, \quad i=0,1,2,\cdots,r \\ \text{N3. } \{\varphi_i\}\,\mathcal{T}\,\{\bigvee_{j\leqslant i}\varphi_j\}, \quad i=1,2,\cdots,r\end{array}}{p \Rightarrow q_r \omega q_{r-1}\cdots q_1 \omega q_0}$$

N1：如果 p 在状态 s_k 成立，那么 $\bigvee_{i=0}^{r}\varphi_i$ 在这个状态也成立，即存在 j_k 使 φ_{j_k} 在状态 s_k 成立（$0\leqslant j_k\leqslant r$）。

N2：φ_i 在某状态或状态区间成立，那么 q_i 在这一状态或状态区间也成立，$i=0,1,2,\cdots,r$。

N3：如果 φ_{j_k} 在状态 s_k 成立，那么 $\varphi_{j_{k+1}}$ 在状态 s_{k+1} 成立，且 $j_k \geqslant j_{k+1} \geqslant \cdots$。

由 N1 可知若 p 在状态 s_k 成立，则得到 φ_{j_k} 在状态 s_k 成立。若 $j_k=0$，则可得到结论，即 $p \Rightarrow q$。若 $j_k>0$，还需要考虑 s_k 的下一个状态 s_{k+1}，由 N3 可知存在 $j_{k+1}(0 \leqslant j_{k+1} \leqslant j_k)$ 在 s_k 的下一个状态 s_{k+1} 成立，对 s_{k+1} 不断重复这一过程，从逻辑角度考虑一般情况，可以推知：$\varphi_r, \varphi_{r-1}, \cdots, \varphi_1$ 在状态区间序 $I_r, I_{r-1}, \cdots, I_1$ 上分别成立，且它要么在一个 φ_0 成立的状态终止，要么为无限序列。再结合 N2，便可以得到结论。

注意，$p\omega q$ 意为 p“等待/除非”q；$p \Rightarrow q$ 意为由 p 可以推出 q。

3）响应性规则

响应性规则比较多，我们只介绍单步响应性规则和链式规则，其他规则只是这两个规则的扩充或变形。

（1）单步响应性规则如下。

$$\begin{array}{l} \text{J1. } p \rightarrow (q \vee \varphi) \\ \text{J2. } \{\varphi\}\ \mathcal{T}\ \{q \vee \varphi\} \\ \text{J3. } \{\varphi\}\ \tau\ \{q\} \\ \dfrac{\text{J4. } \varphi \rightarrow \mathrm{En}(\tau)}{p \Rightarrow \Diamond q} \end{array}$$

其中，$\tau \in \mathcal{T}$。

J1：$p \rightarrow (q \vee \varphi)$ 是状态有效的，即在所有状态上，$p \rightarrow (q \vee \varphi)$ 成立。

J2：在 φ 成立的某个状态，执行 $\mathcal{T}$ 中任意一个迁移，$q \vee \varphi$ 在下一个状态成立。

J3：在 φ 成立的某个状态，执行迁移 $\tau(\in \mathcal{T})$，q 在下一个状态成立。

J4：在 φ 成立的所有状态，τ 是使能的，即 $\mathrm{En}(\tau)$ 表示 τ 能够执行。

考虑结论 $p \Rightarrow \Diamond q$，其中 $\Diamond$ 是时态运算符，指“终将”，即 $\Diamond q$ 意为“某个时刻”q。于是，从上述前提可以看出，如果 p 在状态 s_i 成立$(i \geqslant 0)$，q 也在状态 s_i 成立，那么可以直接得出结论。如果 p 在状态 s_i 成立，但是 q 在状态 s_i 并不成立，这时根据 J1，φ 必在状态 s_i 成立。然后根据 J2，只要 φ 在状态 s_i 成立，经过 $\mathcal{T}$ 中任意迁移，在其转换到的下一个状态，$q \vee \varphi$ 必定成立。但是根据 J3，$\mathcal{T}$ 中有一个迁移 τ，却能从使 φ 成立的状态转换到使 q 成立的某个状态，再根据 J4，凡在 φ 成立的状态，τ 是能使的。综上可以得出结论。

（2）链式规则如下。

$$\begin{array}{l} \text{J1. } p \rightarrow \vee_{j=0}^{m} \varphi_j \\ \text{J2. } \{\varphi_i\}\ \mathcal{T}\ \{\vee_{j \leqslant i} \varphi_j\} \\ \text{J3. } \{\varphi_i\}\ \tau_i\ \{\vee_{j<i} \varphi_j\} \\ \dfrac{\text{J4. } \varphi_i \rightarrow \mathrm{En}(\tau_i)}{p \Rightarrow \Diamond q} \end{array}$$

其中，$\varphi_0 \rightarrow q$；迁移 $\tau_1, \tau_2, \cdots, \tau_m \in \mathcal{T}$；J2、J3、J4 中的 $i=1,2,\cdots,m$。

J1：如果 p 在某个状态成立，则 $\vee_{j=0}^{m} \varphi_j$ 也在这个状态成立。

J2：在 φ_i 成立的某个状态，执行$\mathcal{T}$中任意迁移，则有 φ_j 在下一个状态成立，其中 $j \leqslant i$。

J3：在 φ_i 成立的某个状态，执行迁移 τ_i，则有 φ_j 在下一个状态成立，其中 $j < i$。

J4：在 φ_i 成立的所有状态，τ_i 是使能的。

现在考虑结论。用反证法，假设 p 在状态 s_t 成立($t \geqslant 0$)，q 在所有状态 s_m($m \geqslant t$)上不成立。这样一来，由 J1 可知，p 在状态 s_t 成立，若 φ_j 在状态 s_t 成立，则 $j > 0$。因为 $\varphi_0 \rightarrow q$，若 $j = 0$，就与假设矛盾。由 J2 可知，φ_j 在状态 s_t 成立，执行任意迁移后，便有 φ_k 在状态 s_{t+1} 成立，其中 $k \leqslant j$，而且 $k > 0$(与 $j > 0$ 同理)。重复此过程，为清楚起见，把上述迭代过程产生的 φ 序列记为 $\varphi_{j_k}, \varphi_{j_{k+1}}, \cdots$，其下标满足 $j_k \geqslant j_{k+1} \geqslant \cdots > 0$。因此，由 J2 可以推知，必定存在一些 j_k($j_k \geqslant t$)，序列不再递减。这样，由 J2 便得 φ_{j_k} 在(某)状态 s_k 后一直成立。由 J3 可知，因为序列是递减的，故 τ_{j_k} 在状态 s_k 后是不能执行的。但由 J4 可知，τ_{j_k} 在状态 s_k 后是使能的，这违反了 τ_{j_k} 的公平性，所以反证得到结论成立。

2. Manna-Pnueli 验证图方法

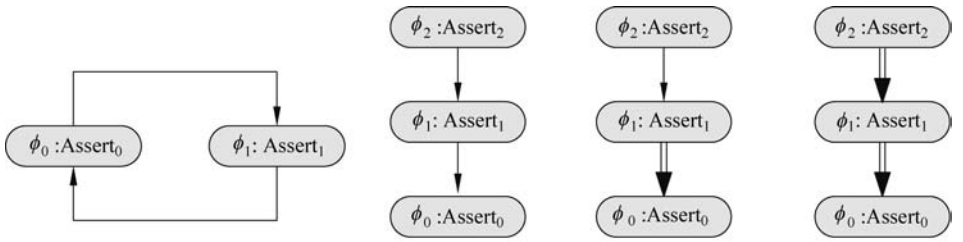
为了并发程序演绎验证可视化，1983 年，Z. Manna 和 A. Pnueli 首次提出并发程序演绎证明规则的图形表述，即验证图(Verification Diagrams)。1994 年，他们系统地阐述了验证各类时序属性的验证图，这就为并发程序演绎验证提供了一种直观、可视化的方法[36]。

验证图是一个带标记的有向图，由节点和边组成。节点与断言绑定，表示一个验证条件。边表示节点的转移。验证图中可能存在这样的节点，没有边再从它引出。这种节点称为终止节点，与它绑定的是“目标断言”，指示演绎证明的结束。在验证图中，边用有向弧表示，节点用长圆角矩形表示，但终止节点的矩形要加粗。我们用 $\phi_m, \cdots, \phi_0$ 表示节点，对应的断言为 $\text{Assert}_m, \cdots, \text{Assert}_0$，其中 $m > 0$。

验证图可分为下几种类型。

1) 不变图

没有终止节点的验证图，可以包含循环。不变图可应用于不变性的证明。图 3.9(a)所示的验证图即为一个不变图。



(a) 不变图示例　(b) Wait图示例　(c) Chain图示例　(d) Rank图示例

图 3.9　4 类验证图示例

2）Wait 图

ϕ_0 是图中终止节点，并且图是非循环的，即无论何时从节点 ϕ_i 引一条边到节点 ϕ_j，$i \geqslant j$ 时，称该验证图为 Wait 图。Wait 图用于优先性证明。图 3.9(b)所示的验证图即为一个 Wait 图。

3）Chain 图

当验证图满足以下条件时，称该验证图为 Chain 图。

(1) ϕ_0 是终止节点；

(2) 从节点 ϕ_i 引一条单线边到节点 ϕ_j，则 $i \geqslant j$；

(3) 从节点 ϕ_i 引一条双线边到节点 ϕ_j，则 $i > j$；

(4) 从节点 $\phi_i(i>0)$有一条双线边从 ϕ_i 引出，该边称为断言 ϕ_i 的“帮助”；

(5) 同一变迁不允许既是单线边又是双线边。

Chain 图可应用于响应性证明。图 3.9(c)所示的验证图即为 Chain 图。

4）Rank 图

当验证图满足以下条件时，称验证图为 Rank 图。

(1) ϕ_0 是终止条件；

(2) 对于每个节点 $\phi_i(i>0)$，有一条双线边从节点 ϕ_i 引出，该边称为断言 ϕ_i 的“帮助”；

(3) 同一变迁不允许既是单线边，又是双线边。

Rank 图也可应用于响应性证明。图 3.9(d)所示的验证图即为 Rank 图。

注意，在 Rank 图中，当 $j>i$ 时，也许节点 ϕ_j 连接到节点 ϕ_i，这一点不同于 Chain 图。

3. Manna-Pnueli 证明规则和验证图应用示例

例 3.1　响应性规则和验证图应用示例[36]

ANY-Y 程序如图 3.10 所示。程序包含两个进程 P_1 和 P_2，共享变量为 x，初始值为 0。进程 P_1 中，只要 $x=0$，y 就增加 1；进程 P_2 中，只有 $x:=1$ 这一条语句。显然，一旦将 x 赋值为 1，进程 P_2 就终止；同样，只要 $x \neq 0$，进程 P_1 也会终止。

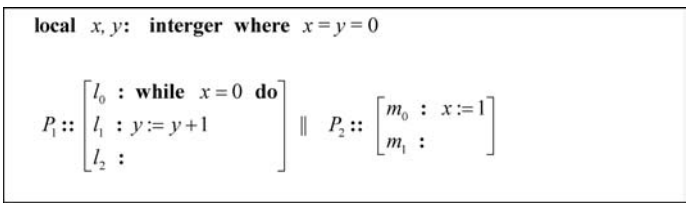


图 3.10　ANY-Y 程序

（转摘于文献[36]图 8-6）

现在，利用链式规则和验证图证明 $\Theta \Rightarrow \Diamond(\text{at\_}l_2 \wedge \text{at\_}m_1)$。其中，初始条件 Θ 为：$\text{at\_}l_0 \wedge \text{at\_}m_0 \wedge x=0 \wedge y=0$，为了使用链式规则证明它成立，建立该规则的 4 个前提。为此，首先建立 4 个断言及其相应的迁移集：

$$\varphi_3: \text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0 \quad \tau_3: m_0$$

$$\varphi_2: \text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1 \quad \tau_2: l_1$$

$$\varphi_1: \text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1 \quad \tau_1: l_0$$

$$\varphi_0: \text{at\_}l_2 \wedge \text{at\_}m_1 \quad \mathcal{I}=\{m_0, l_1, l_0\}$$

如果令 $p=\text{at\_}l_0 \wedge \text{at\_}m_0 \wedge x=0 \wedge y=0$，显然 $p \rightarrow \varphi_3$ 成立，由此得到链式规则中的前提 J1：$p \rightarrow V_{j=0}^{3} \varphi_j$。

下面再来验证链式规则中前提 J2～J4 也是成立的。

关于 φ_3：$\text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0$，验证 J2～J4 成立如下。

$$\underbrace{\{\text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0\}}_{\varphi_3} \tau \underbrace{\{\text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0\}}_{\varphi_3}, \quad \forall \tau \neq m_0$$

$$\underbrace{\{\text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0\}}_{\varphi_3} m_0 \left\{ \underbrace{\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1}_{\varphi_2} \vee \underbrace{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1}_{\varphi_1} \right\}$$

$$\underbrace{\{\text{at\_}l_{0,1} \wedge \text{at\_}m_0 \wedge x=0\}}_{\varphi_3} \rightarrow \underbrace{\text{at\_}m_0}_{\text{En}(m_0)}$$

关于 φ_2：$\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1$，验证 J2～J4 成立如下。

$$\underbrace{\{\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_2} \tau \underbrace{\{\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_2} \quad \forall \tau \neq l_1$$

$$\underbrace{\{\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_2} l_1 \underbrace{\{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_1}$$

$$\underbrace{\{\text{at\_}l_1 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_2} \rightarrow \underbrace{\text{at\_}l_1}_{\text{En}(l_1)}$$

关于 φ_1：$\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1$，验证 J2～J4 成立如下。

$$\underbrace{\{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_1} \tau \underbrace{\{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_1} \quad \forall \tau \neq l_0$$

$$\underbrace{\{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_1} l_0 \underbrace{\{\text{at\_}l_2 \wedge \text{at\_}m_1\}}_{\varphi_0}$$

$$\underbrace{\{\text{at\_}l_0 \wedge \text{at\_}m_1 \wedge x=1\}}_{\varphi_1} \rightarrow \underbrace{\text{at\_}l_0}_{\text{En}(l_0)}$$

因此，根据链式规则，可以得到结论 $\Theta \Rightarrow \Diamond(\text{at\_}l_2 \wedge \text{at\_}m_1)$。验证图如图 3.11 所示。

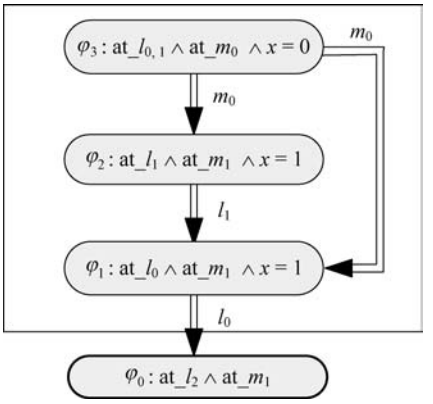


图 3.11　证明 $\Theta \Rightarrow \Diamond(\text{at\_}l_2 \wedge \text{at\_}m_1)$

（转摘于文献[36]图 8-7）

例 3.2　优先性规则和验证图的应用示例[36]

图 3.12 所示为 Peterson 算法，该算法主要用于解决互斥问题。假设有两个进程 P_1 和 P_2，Peterson 算法通过布尔变量 y_1 和 y_2 控制它们访问一个共享的单用户资源行为，从而避免发生访问冲突情况。算法基本机制如下：若进程 $P_i(i=1,2)$试图进入临界区，则将该进程相应的 y_i 设为 T；离开临界区时，将 y_i 设为 F。但是当两个进程同时处于等待状态，即算法中 P_1 和 P_2 分别在 l_4 和 m_4 处，这时 $y_1=y_2=$T，若只有 y_i 控制进程，那么会出现死锁情况。因此，还需要变量 $s(s=\{1,2\})$作为签名。当 $y_1=y_2=$T 时，在下一个语句中，每个进程将自己的下标数字赋给 s 作为签名。也就是说，P_1 执行 $s:=1$；P_2 执行 $s:=2$。增添签名机制，当两个进程都处于等待状态时，若 $P_i(i=1,2)$首先到达等待区，则 $s\neq i$，设 j 为另一个进程的下标，则 $s=j$，它是最后到达等待区的进程，因此 P_i 具有优先权，即可以进入临界区。

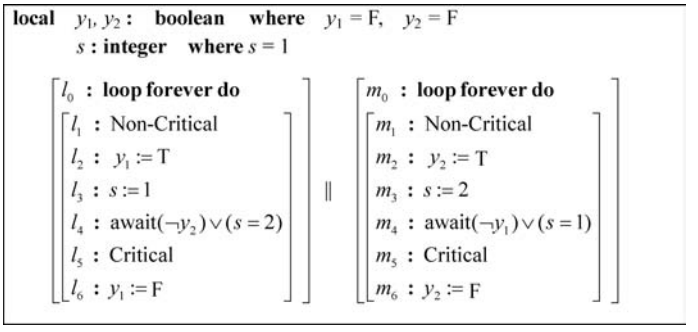


图 3.12　Peterson 算法

（转摘于文献[36]图 8-8）

现在，我们尝试证明

$$\text{at\_}l_4 \Rightarrow (\neg \text{at\_}m_{5,6})\omega(\text{at\_}m_{5,6})\omega(\neg \text{at\_}m_{5,6})\omega(\text{at\_}l_{5,6})$$

上述公式表达的程序属性可以直观描述为：P_1 在 l_4 位置，P_2 可能先于 P_1 进入临界区，但最多只能出现一次。也就是说，如果 P_1 正在 l_4 位置，那么有可能一段区间内 P_2 不在 $m_{5,6}$，接着一段区间 P_2 在 $m_{5,6}$，然后 P_2 又不在 $m_{5,6}$，P_1 进入 $l_{5,6}$。任何一个区间都有可能为空，特别是 P_2 在 $m_{5,6}$ 的区间内，在 P_2 没有先到 $m_{5,6}$ 时，允许 P_1 进入 $l_{5,6}$。另外，任何一个区间都有可能是无限的，在这种情况下，不能保证有紧接着的区间和 P_1 进入 $l_{5,6}$。然而，这种情况是不会发生的。实际上，上述公式也是进程之间公平性属性的表述。由于这个公式牵涉到进程执行的次序问题，是一个优先公式，所以我们用优先性规则证明它。

令 $P=\text{at\_}l_4$，显然，若把上式作为结论，对照前面所述的优先性规则，有：$q_0=\text{at\_}l_{5,6}$，$q_1=\neg \text{at\_}m_{5,6}$，$q_2=\text{at\_}m_{5,6}$，$q_3=\neg \text{at\_}m_{5,6}$。下面建立 4 个断言，让它们满足优先性规则的 3 个前提条件。令 4 个断言为

$$\varphi_0: \text{at\_}l_{5,6}$$

$$\varphi_1: \text{at\_}l_4 \wedge (\text{at\_}m_{0..3} \vee (\text{at\_}m_4 \wedge s=2))$$

$$\varphi_2: \text{at\_}l_4 \wedge \text{at\_}m_{5,6}$$

$$\varphi_3: \text{at\_}l_4 \wedge \text{at\_}m_4 \wedge s=1$$

注意，$P=\text{at\_}l_4$ 表示进程 P_1 处于等待区间，而在整个等待区间内 P_1 从位置 l_4 开始，到 l_5 处结束。在此等待区间，进程 P_2 可在任意处，所以上述 φ_1、φ_2、φ_3 都采取了与 $\text{at\_}l_4$ 合取的形式，表明在 P_1 等待时，P_2 进程的“作为”。同样，q_1、q_2、q_3 也可采取与 $\text{at\_}l_4$ 合取的形式。

φ_0 是用 $\text{at\_}l_{5,6}$ 本身表示的，因为很显然它将终止等待。φ_0 也是进程 P_1 处在等待区间时的“目标”，因此 $q_0=\varphi_0$。

显然，$\varphi_1 \rightarrow q_1$，即 $\varphi_1 \rightarrow \neg \text{at\_}m_{5,6}$。然而，$\varphi_1$ 断言在 P_1 处在等待区间时，P_2 不是处在 $m_{0..3}$ 就是在 m_4，但由于 $s=2$，它比 P_1 后“签名”，以致无法进入 $m_{5,6}$。现在，φ_1 是 P_2 到达 $\text{at\_}m_4 \wedge s=2$ 状态的完整形式，于是它便体现了 φ_1 的作用，在所有状态中，只要 φ_1 成立，则下一个进入临界区的将是 P_1，即在所有状态中，P_1 比 P_2 具有绝对优先权。于是 N3 中 $\varphi_1 \mathcal{I}\{\varphi_1 \vee \varphi_0\}$ 成立。实际上，从 φ_1 成立的状态唯一能转换到的状态是 $\text{at\_}l_{5,6}$。

显然，$\varphi_2 \rightarrow \text{at\_}m_{5,6}$，即 $\varphi_2 \rightarrow q_2$。从 φ_2 成立状态，程序下一个状态是使 $\varphi_1 \vee \varphi_0$，即 $\varphi_2 \mathcal{I}\{\varphi_2 \vee \varphi_1 \vee \varphi_0\}$ 成立。

因为 φ_1、φ_2、φ_3 表示在 P_1 处于等待区间时，P_2 所有的状态，也是程序所有状态。鉴于 φ_1 和 φ_2 的直观意义，φ_3 的直观意义是，在等待位置 l_4，P_2 比 P_1 具有优先权，于是虽然 $\varphi_3 \rightarrow q_3$，即 $\varphi_3 \rightarrow \neg \text{at\_}m_{5,6}$，但是，下一状态可能是 φ_2，即 $\varphi_3 \mathcal{I}\{\vee_{j\leqslant 3}\varphi_j\}$ 成立。

显然，$p \rightarrow \varphi_0 \vee \varphi_1 \vee \varphi_2 \vee \varphi_3$，所以优先性规则前提全部成立，根据这个规则，得到结论：

$\text{at\_}l_4 \Rightarrow (\neg \text{at\_}m_{5,6})\omega(\text{at\_}m_{5,6})\omega(\neg \text{at\_}m_{5,6})\omega(\text{at\_}l_{5,6})$。

最后用验证图验证上述属性，如图 3.13 所示。

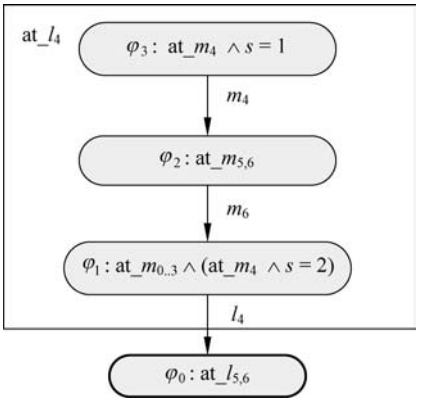


图 3.13　证明 $\text{at\_}l_4 \Rightarrow (\neg \text{at\_}m_{5,6})\omega(\text{at\_}m_{5,6})\omega(\neg \text{at\_}m_{5,6})\omega(\text{at\_}l_{5,6})$

（转摘于文献[36]图 8-9）