1999

Research and implementation of integrated concurrent software development environment based on Object-Oriented Petri Nets

Jinzhong Niu
CUNY Guttman Community College

硕士毕业论文

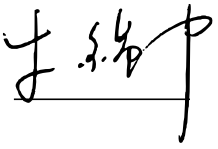
基于面向对象 Petri 网的
并发软件集成开发环境的研究与实现

牛锦中

北京航空航天大学
计算机科学系

本人声明

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者：  日期：1999 年 2 月 28 日

北京航空航天大学学位论文

论文题目：**基于面向对象 Petri 网的并发软件集成开发环境的研究与实现**

计算机软件 专业

九六 级 **硕士** 研究生 **牛锦中** 导师 **任爱华**

摘要

网络技术和环境的日益发展和普及，使对并发软件的需求日益增加，但并发问题的诸多特点使之很难开发。

本论文把面向对象技术和 Petri 网理论相结合，提出了 OOPN 模型以对并发软件进行描述。在此基础上，通过对传统的并发软件开发方法的研究，提出了一种基于 OOPN 模型的集规格说明定义、验证、代码生成于一体的新方法，并开发了相应的以 OOPN 模型为核心的并发软件集成开发环境——OOPN-IDE。它由几个子工具构成：OOPN 模型图形和文本建模工具、多机协同建立/编辑系统模型环境、并发系统仿真和运作工具、死锁检测工具。

本论文的创新工作包括：

- **将面向对象与 Petri 网相结合**

特别阐述了 Petri 网对面向对象方法中问题的弥补。

- **Petri 网支持并发系统建模、仿真、目标系统实际运做全过程**

不同于传统上将网结构转化成程序控制结构的做法，OOPN-IDE 生成的并发程序中仍然保留有网结构。并发程序的执行基于网中 transition 的不断引发而逐步推进，从而可以以 Petri 网的观点和角度对系统进行即时的监视和控制。

- **多机协同**

OOPN-IDE 为 Client/Server 结构，多个 Client 可进行协同建模，为设计大型系统提供了基础。

- **跨平台**

OOPN-IDE 基于 Java 语言开发，可跨平台使用，已通过了在包括 Windows95/98/NT 和 Solaris2.x 在内的异构环境下的测试和鉴定。

- **对进一步的工作方向进行了规划**

关键词： 并发软件，面向对象，面向对象 Petri 网，OOPN 网模型，集成开发工具

答辩日期_____ **导师签字** _____

Master's Dissertation of BUAA

Title of Dissertation: Research and Implementation of Integrated Concurrent Software Development Environment Based on Object-Oriented Petri Nets

Author: Jinzhong Niu

Supervisor: Aihua Ren

Abstract

While network technologies and environments get more and more popular, the demand for concurrent systems becomes stronger and stronger. But the characteristics of concurrency make it difficult to develop them.

This dissertation combines Object-Oriented technology and Petri Nets theory, and put forth OOPN(OO Petri Nets) model to describe concurrent systems. Based on the research of conventional development methods of concurrent software, this article advances a new one, which integrates specification, validation and auto-generating codes. Accordingly, an integrated development environment for concurrent software, i.e. OOPN-IDE, comes into being, which mainly includes the following four tools: the concurrent system modeling tool in either graphics mode or text mode, multi-user modeling coordinator, the concurrent system simulator and enactor, the concurrent system deadlock detector.

This dissertation contains the following innovations:

- **Combine OO and Petri Nets**

  Specially, it is described how OO is benefited from Petri Nets.

- **Petri Nets support the whole cycle of modeling, simulation and enaction.**

  Petri Net structures are kept in generated programs rather than being translated into program structures in the traditional way. The execution of the concurrent programs is based on the firing of the transitions in net models. Thus, we can monitor and control the systems in the view of Petri Nets.

- **Multi users can model coordinately.**

  OOPN-IDE uses a client/server architecture. Multi clients can be involved in a single model simultaneously. The possibility of developing large systems is seen.

- **OOPN-IDE is cross-plat.**

  OOPN-IDE is based on Java and can be used on multi operating systems. It has been tested and appraised by experts in a heterogeneous environment involving Windows 95/98/NT and Solaris 2.x.

- **The future work is discussed.**

Key words: **Concurrent Software, Object-Oriented, Object-Oriented Petri Nets, OOPN, Integrated Development Environment**

Date of Oral Examination: _____ **Signature (Supervisor):** _____

1. 绪论

1.1. 背景

当今社会飞速发展，正在逐步地由工业经济时代进入信息或知识经济时代。这一切变革都与计算机技术的发展和应用息息相关。

一般说来，任何领域都是在需求和技术更新这双重的推动下发展前进的。计算机领域也不例外。正是本世纪中叶计算机的诞生及其以后的发展使社会的方方面面发生着翻天覆地的变革，而社会的进步又创造了更多的需求，从而反过来成为计算机技术发展的巨大动力。

正是在这种循环往复的推进中，**并行计算和处理**方向逐渐成为了一道亮丽的风景。

众所周知，著名的冯.诺依曼串行计算模型对程序的执行顺序约束太多，从它得到更多的性能日益困难。于是，人们把目光转向了大有希望的并行计算。

除了性能上的要求而引入并行外，表达并行还可以从过去的用串行来模拟并行转变到直接反映事物的这一本质上来，从而在提高速度的同时，更加贴近人的思维和符合客观实际。

并行是一种普遍现象，自然界、人类社会、一个组织、一个系统，它们的组成元素的活动都是并行的，元素和元素之间存在着交互作用，有合作，有竞争，而交互作用则依靠通信手段来达到[5]。比如，当前在航空、航天、军事、金融、通讯和机械等领域，计算机发挥着越来越大的作用。数不胜数的设备或系统是由多个嵌入硬件的并行执行且交互的程序来组成的。

要实现并行计算，不仅需要并行的硬件，更重要的是要有相应的软件环境[1]。现在并行计算机(或称为多处理机)尤其是计算机网络的日益发展和普及，各种分布式和并行软件系统的需求更是广泛和强烈。

对于并行计算的软件支持方面，主要应考虑如下几个方面：

1、并行计算程序语言环境：为完成并行计算，需要有用于描述并行计算过程的程序设计语言的支持，还要有调试工具、性能分析器和可视化工具等。

2、并行计算的编译：并行计算程序设计有显式和隐式之分，前者指程序的哪些部分并行执行、如何调度这些并行部分的执行是由程序员利用程序语言来完成的，并行计算对编译几乎没有什么新的要求，只需要一个简单的语言处理程序。对于后者，程

序员并不涉及具体并行执行的细节，程序的并行化由编译自动完成，即并行计算必须有并行编译。

3、并行计算的操作系统：并行计算的操作系统应能支持并行用户程序运行时对多个处理机的要求，在资源管理与调度、进程管理与同步等方面要高效、可靠地运作。

以上的几方面是相辅相成、密切联系的。对于并行编译，由于复杂程序中的并行性探测非常困难，故由其带来的性能上的好处很有限[1，4]。因此现在，一般都在程序的编制上显式地表达并行，除非要实现并行处理的程序是以简单的易于识别的常规数据结构，如向量、矩阵、表等为核心。而并行硬件环境下的操作系统也是属于并行软件系统的范畴。

因此，对于并行计算的软件上的支持现在可以归结到一点，那就是并行或并发软件的开发。

单从并发软件的编程方式上看，可分为：

1、基于操作系统提供的系统调用：UNIX、VM 与 MVS (IBM 大型机操作系统)和 VMS (DEC VAX 小型机操作系统)中均提供了对进程的创建、激活、同步、通信、调度、终结等机制，有些甚至可以表达不确定性。

2、基于程序设计语言：并行 Pascal、Ada、并发 C、occam 等是常见的并发程序设计语言。Java 更是新近出现的可进行多线程编程的流行语言。利用它们可以编写独立于具体硬件和操作系统的的并发程序。

3、基于并行或并发虚拟环境：如 PVM、MPI、Express、NX 等。在这些环境中提供了一些与某种编程语言结合的并行或并发函数库，可在使用相应语言时调用库中的内容来实现并行[23]。与上一种方法不同的是，不需要对程序设计语言进行语法上的更改，从而避免了重新设计编译器的麻烦。

以上三种方法的层次是逐渐提高的，使得并发程序的开发向着容易的方向发展。但是，使用当前的这些开发方法都很难开发高质量的软件，因为：

1、编程困难：如上的方法均需要程序员首先要熟悉程序运行所处环境提供的并发机制，通常是一些琐碎的应遵循的原理或库方法的定义，如线程或进程的定义、创建与启动及进一步地控制。其次，在使用所提供的机制时，必须符合某些原则以避免由于并发性的存在造成某些问题，如由于对资源的占有和申请处理不当出现死锁。

2、调试困难：并发程序具有随机和不确定的性质，程序的某次执行出现的问题很难使之重现。本来对顺序程序来说调试就非常麻烦，再加入多个控制流，无疑是难上加难。

在本文后面将要提到的 OOPN 集成开发工具的开发中，笔者使用 Java 的多线程机制来实现并发系统的仿真运行，即深深感受到用如此低层次的设计实现方法是多么地

困难。

实际上，上述的并发程序的开发方式比顺序程序的开发没有什么提高，根本就没有充分考虑并发程序的特点有针对性地进行改进和采用新的开发方法。

Petri 网是迄今为止最好的并发模型之一。它可以清楚、简捷地表达并发、并行、异步等性质，有图形表示和可以进行数学分析，且是可执行的(operational)。用 Petri 网描述并发系统不仅有可视化的优点，而且可以进行静态的性质分析及动态的仿真检测。所以将 Petri 网引入来辅助并发软件的开发是一个极好的思路。

本论文即是研究如何正确、方便、快捷地基于 Petri 网并结合面向对象思想来进行并发或并行软件的开发的。

1.2. 基本概念

这里首先对本文中所围绕的核心概念进行论述，以期为文章的其它部分奠定基础。

1.2.1. 并发程序、并行程序和分布式程序

由于在许多文献中，并发程序(concurrent program)、并行程序(parallel program)、分布式程序(distributed program)等概念被不加区分地使用，很少有作者对它们进行精确的定义，因此在这里有必要做一下区分。

一般认为：

- 并发程序是指包含有可同时执行的多个动作的程序。这里的“同时”是概念上的，而并不一定要求它在微观的实际执行中是真正的同时。
- 并行程序是指专门在并行硬件上执行的并发程序。
- 分布式程序是指专门在由没有共享的存储器的自主处理器构成的网络上执行的并行程序。

这样的话，我们就可以用并发程序来表示任何一个涉及实际或潜在并行行为的程序。而并行程序和分布式程序则看作是在特定硬件环境下执行的并发程序的特例。

实际上，如果使用单一的处理器来执行并发程序，则其中的并发也被称为伪并行(pseudoparallelism)。

在上面论述的基础上，不难理解并行软件和并发软件及分布式软件的区别。尽管通常认为程序只是软件的一重要组成部分，但在本文中将不加区分地使用程序与软件两个概念。

1.2.2. 并发程序和并发系统

一个并发程序是一个紧密耦合的软件体(a coherent unit of software)。如果两个通信着的程序段并发地执行且在概念上构成了一个整体，我们就认为它们构成了一个并发程序。而如果它们被当作基于预先定义的协议进行通信的两个程序，则它们则构成了一个并发系统(concurrent system)[4]。同样地可以推究出并行系统(parallel system)和分布式系统(distributed system)的含义。

1.3. 课题的来源、目的和意义

1.3.1. 来源与结论

本论文研究的课题来自国家航空科学基金项目——基于面向对象 Petri 网的并发软件开发方法研究。

此项目原计划三年完成：1996 年 10 月——1999 年 10 月。由于整个课题组的艰苦奋斗、共同攻关，现在已提前近一年顺利圆满地并超额完成了研究任务。1999 年 1 月 18 日，本课题通过了由国家航空工业总公司主持召开的鉴定会，由清华大学、北京大学、中国科学院等单位的多位专家组成的鉴定委员会一致认为：“‘以面向对象 Petri 网为基础的并发软件开发方法研究’课题组在理论研究与系统开发方面有所创新，解决了面向对象 Petri 网理论与实践相结合的难题。支持工具 OOPN-IDE 的研制与开发，在跨平台多机协同建模方面以及在并发系统建模与系统运作集成方面处于国际领先水平”[7]。

1.3.2. 目的和意义

如前所述，并行处理是一门综合性的计算机学科，它除了研究并行的硬件体系结构之外，更重要地要包括算法、语言、程序设计方法和支持工具等软件方面的技术。其中，并发软件的开发方法问题是重要的一环。

但是开发并发软件要比开发顺序软件困难得多。并发软件的诸多特性给程序的动态调试带来很大的困难。用传统的顺序程序开发方法(如：Yourdon，Jackson，PAD)无法实现并发和分布控制[8]。

本项目研制开发的基于 OOPN(**O**bject-**O**riented **P**etri **N**et，面向对象 Petri 网)模型的 OOPN-IDE(**OOPN I**ntegrated **D**evelopment **E**nvironment，OOPN 集成开发环境)可用于一般性的并发系统建模和并发程序的生成与并发系统控制，如柔性制造系统的系统建模与控制系统、网络协议分析以及操作系统等。

从航空航天和国防军事应用来看，计算机在高速自动化指挥控制系统中心，在破译技术以及核武器、火箭、航天工具、军用飞机、侦察武器、主战坦克、水下武器等装备的设计和模拟方面的应用中，需要进行并发软件的设计与开发，因此 OOPN 集成开发环境可广泛应用于国防军事和航空航天领域。

从程序员的观点来看，需要研究一种自然、简单和灵活的过程，以一种接近于程序设计者的思维方式来描述并发软件中固有的并发、同步以及通讯现象。由于 Petri 网可以简单而又直观地说明并发系统或分布式系统的结构问题以及各个模块之间的因果关系，既支持数据抽象又支持控制流抽象，因此，凡是具有并发、异步、分布、并行、不确定性和随机性的信息处理系统都可以使用 Petri 网描述其系统模型和进行系统分析。Petri 网本身非常适于描述并发现象，是一种描述并发程序强有力的工具，但是 Petri 网的主要缺点是复杂性问题，因此本课题把面向对象方法与 Petri 网相结合，综合两者的优点，提出了一种基于面向对象 Petri 网的并发系统设计的新方法。该方法可为软件工程提供一种新的思路。

作为本项目的应用，已获准开展国家自然科学基金项目“基于扩展 Petri 网的多处理机操作系统研究”，目前该项目正在实施中。

总之，本项目的应用前景十分广阔。

1.4. 所做的工作

1、将面向对象理论和 Petri 网模型相结合，提出了 OOPN 模型，并对基于其的系统建模及对并发软件开发的支持进行了探索。

发现了[12]中 OPNets 模型死锁检测算法中的问题，并对其进行了详尽的分析。

2、开发了基于 OOPN 模型的集成化支持工具——OOPN-IDE。

1.5. 内容安排

本文的内容安排如下：

第一章　绪论：从计算机科学和技术发展的整个场景逐渐进入本文所着力的并发软件开发的范畴，对研究的课题进行了说明，阐述了研究的意义。

第二章　面向对象 Petri 网模型——OOPN 模型：首先简要地回顾了 Petri 网从传统的经典网模型，到谓词网、着色网、分层网等高级网，进而再到基于对象和面向对象的网模型的发展历程，然后对本文提出的 OOPN 模型所基于的 OPNets (**O**bject-oriented **P**etri **Nets**)模型进行描述，最后给出 OOPN 模型的定义。

第三章　基于 OOPN 模型的并发软件开发方法：在研究了传统的瀑布模式的结构化并发程序设计方法的基础上，提出了基于面向对象 Petri 网的并发软件开发方法。

第四章　OOPN 集成开发环境——OOPN-IDE：首先对常见的几种 Petri 网支持工具做一简单的介绍，然后给出了基于 OOPN 模型的并发软件开发方法的支持工具——OOPN-IDE 的总体描述，对其中使用的数据结构进行了详细阐述。

第五章　OOPN-IDE 模块设计与实现：给出了 OOPN-IDE 中各个模块的设计思路和实现细节。包括：图形和文本建模部分、多用户协同图形建模机制、模型系统仿真运作的机理、OOPN 模型系统的静态死锁检测算法和动态检测的思想等。

结束语：概括全文，总结研究成果和创造性工作，对进一步的工作方向进行了展望。

2. 面向对象 Petri 网模型——OOPN 模型

本章首先回顾 Petri 网的发展历史，对各种已有的网模型进行详尽的分析，论述其优点和缺陷，进而针对并发软件的开发，提出一种新的面向对象 Petri 网模型——OOPN 模型。OOPN 模型是进行系统建模和支持工具开发的基础。

2.1. Petri 网模型

1962 年，C.A.Petri 以其论文“Kommunikation mit Automaten”在达姆施塔特大学获博士学位，标志着 Petri 网的诞生。Petri 网是为了帮助设计和分析并发系统而开发的理论，被认为是第一个通用的并发理论，它是自动机理论的扩充。

2.1.1. 经典 Petri 网

2.1.1.1. 基本定义

下面给出若干定义，以作为进一步讨论的基础[10]：

定义 2.1：三元组 N=(S，T；F)称为有向网(directed net)，简称网(net)，的充分必要条件是：

1、$S \cap T=\Phi$，2、$S \cup T \neq \Phi$，3、$F \subseteq S \times T \cup T \times S$，(“×”为笛卡儿积)

4、$dom(F) \cup cod(F)=S \cup T$，

其中 $dom(F)=\{x|\exists y:(x，y) \in F\}$，

$cod(F)=\{y|\exists x:(x，y) \in F\}$ 分别为 F 的定义域和值域。

S 和 T 分别称为 N 的 place 集和 transition 集，F 为流关系(flow relation)。

通常用圆圈或椭圆表示 place，用方框(□)或粗杠(|、–)表示 transition，用从 x 到 y 的箭头表示流关系中的(x，y)。如图 2.1 所示为有向网的图形表示示例。

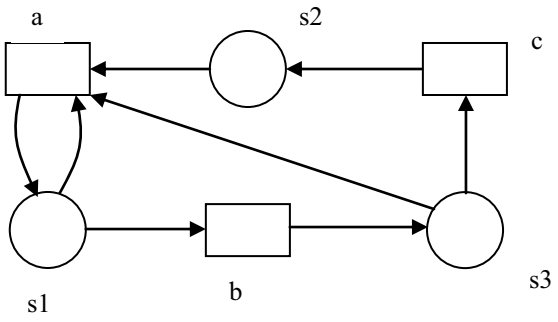


图 2.1 有向网示例

定义 2.2：设 x∈X 为 N 的任一元素，

${}^{*}x=\{y|(y，x)\in F\}$称为 x 的前集(pre-set)或输入集。

$x^{*}=\{z|(x，z)\in F\}$称为 x 的后集(post-set)或输出集。

定义 2.3：若∀s∈S 和∀t∈T，均没有 $s\in {}^{*}t\cap t^{*}$，则称该网为单纯网(pure net)，简称为纯网。

定义 2.4：若∀x，y∈X，${}^{*}x={}^{*}y\wedge x^{*}=y^{*}\Rightarrow x=y$，则称该网为简单网。

定义 2.5：令 N=(S，T；F)为网，X= S∪T 为其元素集。若(X，F)为连通图，则 N 为连通网。

以上给出的有向网仅仅是系统的结构框架。要描述一个完整的系统还必须指明资源的初始分布，规定框架上的活动规则，从而又有了如下的定义：

定义 2.6：对于有向网 N={S，T；F}，记$\prod_0$={0，1，2，.....}，$\prod$={1，2，3，...}，并以ω表示无穷：ω=ω+1=ω-1=ω+ω，则有：

1、K:S→$\prod$∪{ω}称为 N 的容量函数(capacity function)，

2、对给定的容量函数 K，

M:S→$\prod_0$称为 N 的一个标识(marking)的条件是：∀s∈S：M(s)≤K(s)。

3、W:F→$\prod$称为 N 上的权函数，对(x，y) ∈F，W(x，y)=W((x，y))称为(x，y)上的权。

定义 2.7：六元组Σ=(S，T；F，W，M_0)构成网系统的条件是：

1、N=(S，T；F)构成有向网，称为Σ的基网。

2、K、W、M_0依次为 N 上的容量函数、权函数、标识。M_0称为Σ的初始标识(initial marking)。

以上给出了网系统从结构到资源的静态特征，只要再定义 transition 发生的条件和后果，网系统的定义就完整了。网系统的动态规律称为发生规则(firing rule)。

设 M 为网系统Σ=(S，T；F，K，W，M_0)之基网(S，T；F)上的任一标识，t∈T 为任一 transition。

定义 2.8：(transition 发生条件)

1、 ${}^*t^*={}^*t\cup t^*$ 称为 t 的外延(extension)。

2、 t 在 M 有发生权(firable)的条件是

$\forall s\in {}^*t$：$M(s)\geq W(s，t)\wedge\forall s\in t^*$：$M(s)+W(t，s)\leq K(s)$，

t 在 M 有发生权记作 M[t>，也说 M 授权(enables)t 发生或 t 在 M 受权(enabled)发生。

定义 2.9：(transition 发生后果)

若 M[t>，则 t 在 M 可以发生，将标识 M 改变为 M 的后继(successor)M’， M’的定义是，在任何 s∈S：

$$M'(s)=\begin{cases} M(s)-W(s，t) & 若\ s\in {}^*t-t^* \\ M(s)+W(t，s) & 若\ s\in t^*-{}^*t \\ M(s)-W(s，t)+W(t，s) & 若\ s\in {}^*t\cap t^* \\ M(s) & 若\ s\notin {}^*t^* \end{cases}$$

M’为 M 之后继的事实记作 M[t>M’。

在网的图形表示的基础上，每个 place 对应一个圆，标识 M 的图形表示即为在与 s 对应的圆中画上 M(s)个称为 token 的黑点。s 的容量(假定 K(s)=m)则是在这个圆的附近写上 K=m。

由于 F 中的每一有序偶(x，y)用从 x 到 y 的有向弧表示，可以直接把 W(x，y)写在这条弧上。如图 2.2 为网图形示例。

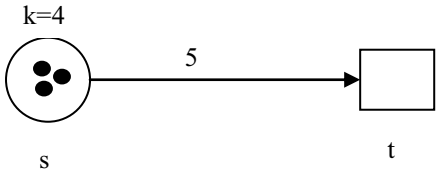


图 2.2 网系统示例

传统上在 K(s)=ω和 W(x，y)=1 时均采用缺省方法，即不标明的容量均为无穷，不标明的权为 1。

对于网系统来说，其主要特点来自外延公理，即：一个 transition 的发生完全由它的外延(前集和后集之和)决定，与系统全局状态无关。因而网系统是异步并发的“自由王国”，没有全局性的中央控制。故网系统的主要应用方向是分布式系统和并行处理。在不同的应用领域或不同的学科给出不同的解释，即可使网系统得到不同的应用。表 2.1 给出了常见的一些网系统的理解：

表 2.1 常见的几种网系统模型的应用

前驱 place	transition	后继 place
前置条件	事件	后置条件
输入数据	计算步骤	输出数据
输入信号	信号处理	输出信号
所需的资源	任务或工作	释放的资源
前提	逻辑子句	结论
缓冲区	处理器	缓冲区

2.1.1.2. 网系统分类

C.A.Petri 在 1962 年使用的系统模型实际上是 K≡ω和 W≡1 的网系统。70 年代 A.Holt 把这种系统称为 Petri 网，Petri 网的名称由此而诞生。同时，把网和网系统不加区分地统称为 Petri 网也由此开始。直到 80 年代中期，人们才明确指出必须把网和网系统区分开。

历史的原因使“Petri 网”一词有三种不同的含义[10]：

1、指定义 2.1 中的有向网；

2、指定义 2.7 中的网系统，但 K≡ω和 W≡1；

3、泛指以有向网为基础的整个学科。

网系统的容量函数 K 和权函数 W 可分为三种情况：

1、K≡1， W≡1。

这时每个 place 只有“有”和“无”两种状态。故网论中将这种 place 称为条件(condition)，只与条件关联的 transition 称为事件(event)。由条件和事件构成的网系统称为基本网系统(elementary net system)或 EN 系统。

2、K≡ω， W≡1。

这是传统上称为 Petri 网的网系统，又称为 P/T 网(**P**lace/**T**ransition net)。

3、K、W 为任意函数(见定义 2.7)。

这种系统通常称为 P/T 系统(**P**lace/**T**ransition system)。

如果从表达的能力上和表现的性质上考虑，则实际上 P/T 网与 P/T 系统则无实质区别[10]。另一方面，EN 系统又是 P/T 系统的特例。故现在，三种情况可以归结到一种。不妨选其中的表达上最自由的 P/T 系统来代表三者，则得到了通常我们提到“Petri 网”时所意指的对象，即本文所提到的经典 Petri 网。

2.1.1.3. 性质、分析方法与应用

2.1.1.3.1. 性质

网系统有安全性或有界性、活性等方面的性质。而性质的定义离不开可达标识集的定义。

定义 2.10：P/T 系统Σ=(S，T；F，K，W，M_0)的可达标识集[M_0>是满足下列条件的最小集合：

1、$M_0 \in [M_0>$，

2、若有 M’ $\in [M_0>$，t∈T，使 M’[t>M，则 M∈[M_0>。

定义 2.11：

1、若对于所有 M∈[M_0>，存在正整数 k，使得对所有 s∈S，M(s)≤k，就说Σ是有界 P/T 系统，或 k 有界 P/T 系统。k=1 时也称Σ为安全系统。k 有界系统也称 k 安全系统，k 是这种系统的界。

2、对 t∈T，若对任一可达标识 M∈[M_0>，均有从 M 可达的标识 M’ ∈[M>，使得 M’[t>，就说 t 是活的。

3、若所有 t∈T 都是活的，就说系统Σ是活的。

有些系统的界有时不易或无法确定，但可以证明其存在性，这时也可笼统地说该系统是有界的。

2.1.1.3.2. 分析方法

为了得到系统的性质，Petri 网的研究者已经创立了许多方法对建立的网系统进行分析，包括可达树、可达图、transition 发生序列等。

定义 2.12：Σ的可达树(reachability tree)是由下述算法构造的树结构，它的每一节点 x 都有个标记 M_x，M_x 是从 S 到非负整数或ω的映射，即 M_x：S→{0，1，2，...}∪{ω}。

算法步骤为：

(a) T(Σ)的初值只有根节点 r，$M_r=M_0$，即 M_r 以初始标识标记。

(b)令 x 为 T(Σ)的叶节点。若任何的 t∈T 在 M_x 均无发生权，x 为真叶节点；若从根节点 r 到 x 的路径上有另一节点 y，y≠x，但 $M_y=M_x$，则 x 也是真叶节点。若 T(Σ)的所有叶节点均为真叶节点，算法终止。否则执行(c)。

(c)若 T(Σ)有叶节点 x，x 不是真叶节点。于是在 M_x 至少有一个 transition 有发生权。对 M_x 授权发生的每个 t∈T，在 T(Σ)上添加一个新节点 y，y 是 x 的子节点，从 x 到 y 的有向弧用 t 标记。节点 y 的标记 M_y 是如下定义的：首先计算出 M_x 的后继 M'，即对所有 s∈S，$M'(s) = M_x(s) - W(s，t) + W(t，s)$，然后计算 M_y：对所有 s∈S，

$$M_y(s) = \begin{cases} \omega，若从 r 到 y 的路径上有节点 z，使得 M_z<M' 且 M_z(s)<M'(s) \\ M'(s)，否则 \end{cases}$$

(d)回到步骤(b)。

基于可达树有：

定理 2.2：P/T 系统Σ是有界的当且仅当可达树 T(Σ)的任意节点的标记中都不含ω[11]。

但是 T(Σ)有如下明显的不足之处。

1、若在节点 x ∈T(Σ)，M_x 使 t_1 和 t_2 两个 transition 有发生权，那么根据算法，x 会有两个子节点分别与 t_1 和 t_2 的发生对应，而不区分 t_1 和 t_2 是互相冲突还是可以并发。换言之，从树结构看不出冲突和并发的区别。

2、死标识和活标识不能一目了然。有限序列和无限序列也不能从可达树的结构上看出来。T(Σ)中的真叶节点可按标记分为两类：标记不能使任何 transition 发生的一类和标记能使某些 transition 发生的另一类。我们把前一类称为死节点，后一类称为活节点。

为了消除可达树的缺陷，引入可达图的概念。

定义 2.13：如果存在 T(Σ)到 G 的满映射 h：T(Σ)→G，使得：

1、x 为 T(Σ)之节点，则 h(x)为 G 的节点，且 h(x)以 x 在 T(Σ)中的标记 M_x 为标记。

2、(x，y)为 T(Σ)上以 t 为标记的有向弧，则(h(x)，h(y))为 G 上以 t 为标记的有向弧。

3、x≠y 为 T(Σ)的不同节点，则当且仅当 $M_x=M_y$ 且 x 和 y 同在从 T(Σ)根节点 r 出发的同一条路径时，才能有 h(x)=h(y)。

则 G 称为Σ的可达图(reachability graph)，记为 G(Σ)。

定义 2.14：若 x 为 T(Σ)的死节点，则 h(x)称为 G(Σ)的末端节点(terminal node)。

基于可达图有：

定理 2.2：若 G(Σ)有末端节点，则Σ的任何 transition 都不是活的。

在后面的支持工具的实现中，要基于以上的定义和定理给出实现。

其他象 S-不变量、T-不变量、伴随矩阵等方法等也是常用的方法[10，11]。

2.1.1.3.3. 应用

任何具有并发、异步、分布、并行、不确定性或随机性的系统均可用 Petri 网来进行描述[11]。

中国 Petri 网理论研究的先行者袁崇义教授讲课时曾说过一个小故事：Petri 博士有一次力图对一个法律上的案例进行建模，但建到最后发现，必须将两个位置直接连起来才行，而这种情况在 Petri 网中是不合法的。Petri 博士经过仔细研究发现，原来是在法律条文中有一个漏洞。

从这个小故事中，可见 Petri 网理论的应用范围之广。

实际上，Petri 网已广泛地应用在性能评价和通信协议方向。其他一些有希望进行建模和分析的方向是：分布式软件系统、分布式数据库系统、并发和并行程序、柔性制造/工业控制系统、离散事件系统、多处理器共享存储系统、数据流计算系统、容错系统、可编程逻辑和超大规模集成电路阵列、异步电路和结构、编译器和操作系统、办公自动化系统、形式语言和逻辑程序等方面。其他非计算机领域还有：法律系统、人力资源分析、数字过滤和决策模型等。

本文是把并发程序或软件作为研究对象的，故在此分析其性质。

对某一程序系统来说，其动态性质可分为两类：

- ✧ 安全性：要求所有可能的状态具有一定的性质，属于可能性的范畴。
- ✧ 进展性：要求某些特定的状态是一定能达到的，属于必然性的范畴。

无死锁(deadlock-free)是安全性的典型例子，它要求系统的一切可能状态都不会把系统锁死(例如通信双方互相等待)。

无饥饿是进展性的典型例子。描述系统性质的另一个常用的术语是活性(liveness)。一般认为活性是进展性的同义语：不能取得进展的系统即使运行也不能说是活的。它要求保证饥者得食(如对共享资源的申请一定能得到满足)。

应该指出的是：进展性和安全性不是并列的性质。安全性是第一位的，在安全的前提下才谈得上进展。

如前所说，安全性关心“可能性”，死锁的含义是“不可能”(例如不可能通信)，所以是安全与否的问题。进展性是在安全得到保证(有可能)的条件下的必然性。活性指的恰是这一类性质，所以与进展性是一致的。

这里的分析将为本文所阐述的并发软件开发支持工具中的某些算法的设计提供理论依据。

网系统的诸多性质可用来反映模型的面向应用的性质，从而达到服务于应用的目的。

2.1.2. 高级网模型

通常认为：一种模型越普遍适用，则其分析就越难于进行。Petri 网是一种通用的并发模型，它存在的一个主要缺点是复杂性问题，即即使是中等规模的系统，其对应的 Petri 网模型也会变得太大以致无法进行表达，更不用说分析，从而出现了所谓的“状态爆炸问题(state explosion)”[11，12]。

为了解决这一问题，Petri 网的研究者们进行了大量的研究工作。解决的思路不外乎两种：提高网模型的表达能力和进行保持研究的性质不变情况下的化简。本课题的研究属于前者，但后者的某些研究成果也在 OOPN-IDE 的开发中被采用来支持 OOPN 模型的分析。

对于提高网的表达能力的思路可以用图 2.3 表示：

这一做法实际上是借鉴了计算机科学其它方向出现的一些新思想和新技术。

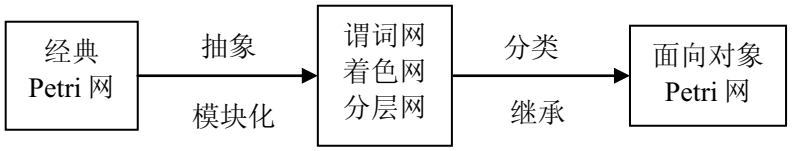


图 2.3 提高网模型描述能力的思路

通过对基于 P/T 系统的模型进行分析得知，其中的某些部分或局部实际上是对同样的或类似的结构进行描述的，完全可以合并，即进行抽象和模块化。但是为了和原来的功用相同，需要适当地处理分布在各个局部的 token。在[10]就提到：如果把作用类似的不同种类的资源用同一 place 表示，那么 place 中的 token 就应该有“个性”，以体现它所属的种类。这样得到的网系统称为个性 token 系统(net system with individual tokens)。基于这样的思路，出现了谓词网、着色网、分层网等。

2.1.2.1. 谓词网

通过引入谓词理论来表达 token 的个性，产生了谓词网(**Pr**edicate/**T**ransition-nets，简称 PrT 网)[13]。

对于谓词网也有相应的 transition 的发生规则的定义，亦可生成可达树及进行不变量分析等。

2.1.2.2. 着色网

谓词网由于基于复杂的谓词演算，至今还未有明确的语义上的分析(中国科学院陆维明教授语)。为了避免谓词的复杂性而又能表达网中 token 的“个性”，出现了着色网，也称有色网。

实际上，着色网就是一种特殊的谓词网。其中的“色”只是一种区分方式，而与通常我们所说的颜色是两回事。决不可以只因为用红笔画了一个网或其中的 token，就称其为着色网。

2.1.2.3. 分层网

由于实际的系统一般总是有一定的层次关系的，而无论是经典 Petri 网还是谓词网、着色网，均是平面结构，无法直接表达层次概念；同时“分而治之”是解决复杂问题的基本方法。基于此认识，出现了分层的网模型，从而使抽象的程度更进了一步[14，15]。

2.1.3. 面向对象 Petri 网模型

引入谓词网、着色网、分层网等模型之后，可描述的系统的规模已大大增加。但是，还不能从根本上解决 Petri 网的复杂性问题。对于复杂的大系统，Petri 网仍是无能为力。国际上就为此在 80 年代后期曾一度冷落了 Petri 网的应用，认为“状态爆炸问题”是 Petri 网的致命缺陷。

而面向对象技术的产生、发展和成熟为 Petri 网的前景带来了曙光。在 90 年代重新兴起的 Petri 网的研究中，与面向对象思想的结合是最有希望的方向。

2.1.3.1. 面向对象思想

面向对象(**O**bject **O**rientation，简称 OO)是当今计算机界关心的重点。面向对象的概念和应用已经超越了最初起源的程序设计和软件开发，而是扩展到很宽的范围。它是一种新的普便适用的方法和技术。

面向对象基于人们对客观世界的认识：客观世界是由许多不同的实体构成的。每个实体具有它自己的行为和内部状态。不同实体间的影响和通信构成了这个世界。在人的认知过程中有两个主要的方法：归纳和演绎。后者基于一个从一般到特殊的过程，它恰是面向对象思想的基础[17]。

面向对象有五大特征：封装、抽象、继承、多态、动态束定[6]。

相关的一些基本概念有[5]：

1、对象

在面向对象系统中，对象是基本的运行时的实体，它既包括数据(属性)，也包括作用于数据的操作(行为)。对象把属性和行为密封成一个整体。

2、消息

对象之间进行通信的一种构造叫做消息。当一个消息发送给某个对象时，包含要求接收对象去执行某些活动的信息。接收到消息的对象经过解释，然后予以响应。这种通信机制叫做消息传递。发送消息的对象不需要知道接收消息的对象如何对请求予以响应，即不需要知道对消息的处理是在何处进行的。

3、类

一个类定义了一组大体上相似的对象。一个类所包含的方法和数据描述一组对象的共同行为和属性。把一组对象的共同特征加以抽象并存贮在一个类中的能力，是面向对象技术最重要的一点。

类是在对象之上的抽象；反之，对象则是类的具体化，是类的实例。

4、继承性

继承性是父类和子类之间共享数据和方法的机制。它是类之间的一种关系，在定义和实现一个类的时候，可以在一个已经定义的内容的基础上来进行。类可以有子类，同样也可以有父类，形成层次结构。

继承性是面向对象的区别于其他的最主要特点。

5、多态

在收到消息时，对象要予以响应。不同的对象收到同一消息可产生不同的结果，这一现象叫做多态。用户可以发送一个通用的消息，而实现的细节则由接收对象自行决定。

6、动态束定

动态束定是程序设计语言所特有的概念。指在运行时将过程调用和响应调用而需要执行的代码结合的过程。

从上所述知道，面向对象提供了诸多的可以提高系统的可重用性、可维护性的手段，能直观、快捷地进行系统的表达和建立。因此将面向对象引入 Petri 网不仅可自然地表达托肯的个性化及层次结构，而还可通过分类、继承等机制进一步提高重用程度，是迄今最有效的 Petri 网复杂性问题的解决办法。

另外，谈一谈并发或并行与面向对象之间的关系。对一个并行系统来说，最本质的方面是：组成并行系统的实体及其特征(包括属性和行为)，实体间的通信，实体间的交互作用(合作和竞争)。而在面向对象中，对象所具有的特点，很适宜于担任并发程序中的实体这一角色。对象有很高的抽象程度，符合信息隐蔽的原则，模块化的特点明显；每个对象的行为一般是顺序的，而它们之间通过消息传递来进行交互，则更是一个并发程序所希望的[5]。可见，面向对象与并发或并行的诸多共性使它们可以很好地结合。

正是基于如上认识，面向对象 Petri 网成为了 Petri 网研究领域的新亮点。

2.1.3.2. 常见的几种对象 Petri 网

应用面向对象思想的 Petri 网模型已出现了很多，但众多模型在引入的面向对象概念的多少及表达的方式上又各有不同，在此统统将它们称为对象 Petri 网，之所以不称为面向对象 Petri 网是因为某些模型中只是涉及了对象或进一步的类的概念，而没有面向对象所特有的继承。

常见的几种对象 Petri 网模型有：G-Net[15，20]、COOPN[17]、LOOPN[18]、PROT[19]、OOCPN[20]等。

1、G-Net

G-Net 表示法将模块及系统结构概念引入 Petri 网，并使模块抽象、封装，模块间是松耦合的关系。前一个特点使 G-Net 适合于说明复杂的软件系统，后一个特点支持复杂系统规格说明的增量修改。

基于 G-Net 的规格说明由一组独立的松耦合的模块组成。在其中，一个模块对另一个模块的访问只能通过一个充分定义了的机制——G-Net 抽象来进行，G-Net 不能直接影响另一个模块的内部结构。

G-Net 的规格说明的执行是由一组并发的进程代理(agent)完成的。一个 G-Net 可取两种执行模式之一：实例化模式或服务器模式。实例化模式中 agent 的作用只是执行一个基于某个方法的 G-Net 的调用的实例。每个实例 G-Net 的实例化过程都创建一个新

的 agent，当实例执行完成时，agent 就终止。在服务器模式，只有一个 agent 与表示服务器的 G-Net 相关。agent 处理所有发给服务器的客户请求。agent 之间的通讯通过标记传递实现。

2、COOPN(**C**oncurrent **O**bject-**O**riented **P**etri **N**ets)

COOPN 对抽象数据类型进行了扩充，使之包括通信结构。系统的说明是由一组对象给出的，每个对象用带有与 transition 相关的方法的抽象数据类型定义，从而构成一个代数网，网中引入了一些参数化的 transition，有三种，分别称为方法、内部 transition 和公理。对象的网结构不再局限于状态机，也可以带有并发发生的多重内部 transition，内部 transition 的发生只依赖于对象的内部状态，而外部 transition 的发生需要与其它对象 transition 的同步机制，对象之间的同步机制是用同步链表示的。为了保留数据封装的特性，对象的访问都是通过它的“方法”进行的。COOPN 的主要贡献是支持模块化的概念，并且可以在真并发的前提下进行对象的细化，但是，这种细化只不过是将对象参数实例化，支持类属对象的定义，而不是使用面向对象语言中的继承性质。另外，COOPN 不具有面向对象技术所支持的层次结构。

3、LOOPN(**L**anguage for **O**bject-**O**riented **P**etri **N**ets)

LOOPN 主要用来描述网络协议，是有色时间 Petri 网的扩展。支持 token 类型的声明。一个 token 类型包括一组数据域和一组相关的函数，可声明为另一种 token 类型的子类型，因而继承父类型的数据域和函数。网中的每个 place 有一个特定的 token 类型。每个 token 到达 place 时被盖上到达时间。这个时间可用来给 place 中的 token 排序。大的 Petri 网可用模块分割。模块的接口是一组常量参数的集合，可用来产生模块的不同实例。一个模块可以通过访问函数向其它模块提供状态信息，可以通过接口 place 在模块间传递 token。LOOPN 还支持 token 类型的多态机制。

4、PROT

PROT 网是一种高级时间 Petri 网，其主要特点是它的操作语义。事实上，它用一种通用的程序设计语言定义 token 的结构以及 transition 的谓词和动作，最后实现时可以将这种通用的程序设计语言转换成目标语言，如：Pascal 语言、C 语言或 Ada 语言等。

该系统中有两类对象：简单对象和复合对象。每个对象的说明包括三部分：对象的 I/O 接口、一组参数及其内部行为。简单对象只需要一个 PROT 网定义即可，而复合对象则包括若干子对象，这些子对象是其它对象的实例，这样，系统便分成了若干子网，子网之间是用链连接的。链将一个 PROT 网的输出 place 连接到另一个 PROT 网的输入 place；进入每个输入 place 的 token 组成一个 FIFO 队列，并且要求它们具有相同的类型；对 transition 而言，对 token 流动的限制是严格的，每个 transition 的具有相同

类型的输入 place 或输出 place 最多只能有一个(但是 nul 类型除外，因为 nul 类型不可区分)。在本系统中，将 token 视为消息，于是，token 的移动便自然地对应着面向对象技术中的消息传递。

5、OOCPN(**O**bject-**O**riented **C**olored **P**etri **N**ets)

OOCPN 是一种面向对象的基于分层有色网(HCPN：**H**ierarchical **C**oloured **P**etri **N**ets)的模型。

HCPN 在有色网的基础上加入了几种层次化的构件，使得大的网模型可由几个小的网组合而成。OOCPN 进一步引入了对象(object)transition、多态(polymorphic) transition 和类属(generic)transition 三种层次化构件。

在 OOCPN 中，有两种类：类(Class)以及聚合类(Aggregate class)，网中每个 place 都与一个类或一个聚合类相关。类定义了 place 中的 token，类包括属性(Attributes)和例程(Routines)，属性是类中的数据成员，例程为类的操作或方法。聚合类的定义中只有属性，没有方法，只能用来定义 place 的颜色集。在 OOCPN 中，证明了三种层次化结构与非层次化的有色网行为的等价性。

对于如上的若干种模型，有以下结论：

1、对象的结构描述及实现方面

有如下两种方式表达对象：

- ✓ 把网中流动的 token 作为对象。
- ✓ 用高级网子网描述一个个对象模板，一个子网就是一个对象。

其中第一种方式表示对象的方式其面向对象的思想是隐含的，没有充分发挥 Petri 网的图形特性。而后者才是提高 Petri 网描述复杂系统能力的主要方式。

2、对象间的通信机制方面

有两种表达方式：

- ✓ 用 transition 的发生来完成消息的传递，消息就用 token 表达。
- ✓ 用赋予额外信息的特殊结构完成。

显然，前者没有引入新的东西，从而保持了原始 Petri 网的结构，为直接应用经典 Petri 网的分析方法提供了保证。

2.2. OOPN 模型

基于上面的阐述和分析，我们就得到了实施项目“基于面向对象 Petri 网的并发软件开发方法研究”的思路：

在现有的面向对象 Petri 网中选出一种普遍适用的描述并发系统的模型，可以根据情况进行适当的修改或扩充，从而形成可以对并发软件进行描述和分析的网模型。在模型的基础上，开发相应的工具来支持模型的建立、仿真、代码生成、分析等功能，从而可完成从规格说明到编码及调试的全过程。

我们选出了 OPNets 模型作为工作的基础，通过进行扩充得到我们的 OOPN 模型。

2.2.1. OPNets 模型

OPNets(**O**bject-oriented high-level **P**etri **Nets**)是韩国 KAIST 的 Yang Kyu Lee 等提出的一种应用于实时系统建模的高级 Petri 网模型。

它的提出是为了能利用面向对象的思想来解决简单 Petri 网的描述系统规模过大的缺点，从而使 Petri 网能广泛地用于系统建模、分析与模拟。

在 OPNets 中，如图 2.4 与 2.5 所示，用高级网子网描述对象的模板即类，通过用方形框把子网括起来表示封装与抽象。对象的外部接口部分由“消息队列” (message queue，简称 mesQueue，用椭圆表示，类似于用圆表示的 place)、“门”(gate，用粗线表示，类似于用方形框表示的 transition)以及它们之间的流关系(arc，用弧线表示)给出。各个类分别可以有若干实例(用黑点表示，即 token)，各实例所处的 place 表示各自的当前状态，故这些 place 特别地称为 state。

对象的内部行为由有限制的谓词网描述。网中的 transition 的前驱和后继中都只能最多有一个 state，这样就禁止了不同的实例不是利用消息传递而是在内部直接进行交互的可能。弧上不加谓词，而每个 transition 则有发生条件和发生时要执行的动作的定义。当 transition 的所有前驱中都有 token 且存在某一 token 的组合使 transition 的发生条件为真时，transition 就可发生。

不同对象之间可以用 gate 把输入 mesQueue 与输出 mesQueue 连接来表示相互的消息传递关系。

类中可以定义属性，transition 的发生条件可以读访问使其发生的实例的属性值，而 transition 的执行动作则可读写属性值。对象相互发送的消息是结构化的数据类型。使 transition 发生的消息将被销毁，而所有后继消息输出队列中则创建一消息，其属性值由 transition 的执行动作来设定。

类有复合类(图 2.4 中的 A)和简单类(图 2.5 中的 AA 和 AB) 之分。在简单类中，不包含并发部分，只用于表示顺序行为；而在复合类中则允许并发，因为复合类由具有独立行为的简单或复合类构造而成，其控制分布在各个类中，并且根据系统要求可以来同步这些内部结构的行为。这些内部的简单类或复合类统称为内部类。

目标系统的模型将是互连的多个定义了对象实例的类的组合。

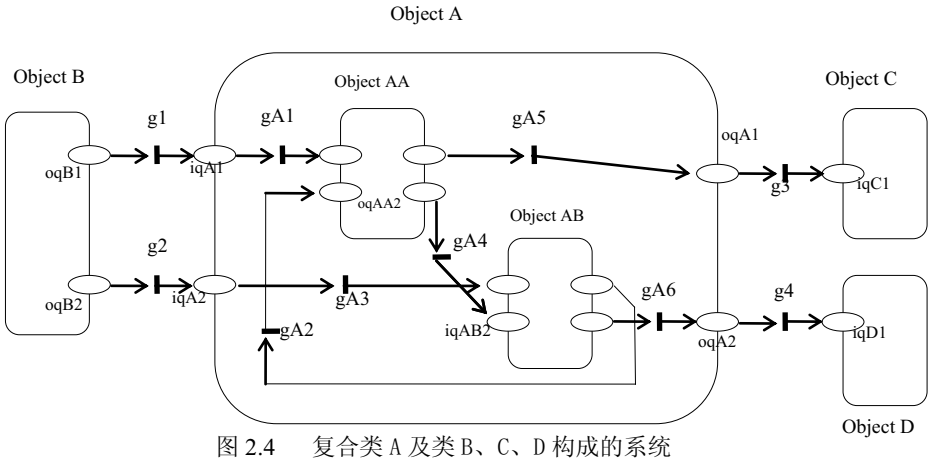


图 2.4　复合类 A 及类 B、C、D 构成的系统

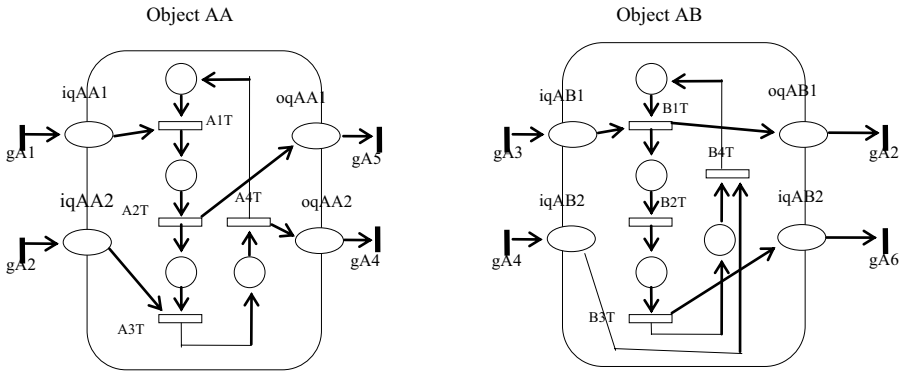


图 2.5　类 A 的内部结构：简单类 AA 和 AB

2.2.2.　对 OPNets 模型的扩充——OOPN

要把 OPNets 用于并发软件的开发，必须对原始的 OPNets 模型进行扩充。经过如下的取舍和增加，我们得到了支持并发软件开发的面向对象 Petri 网模型——OOPN。

■ 基于 Java

要使一模型应用于实际系统的描述和开发，必须使用某种正文语言来进行语义的定义，单靠图形元素是无法有充分描述能力的。

Java 语言可在以下几方面起作用：

- 类的属性的定义，属性可以是 Java 中的简单类型也可以是 Java 类；
- transition 的发生条件和执行动作；
- mesQueue 中可存放的消息类型，定义成 Java 类。

将来要把 OOPN 类转化成 Java 类来进行底层的实现，而且 Java 类中仍然保留网结构，即系统的执行仍然按照网的引发规则来进行，而非将网结构转化成语言中的控制结构来实现。这样做有两个好处：

1、可以直观地通过 Petri 网的执行获知并发程序系统的运作。

2、可以以 Petri 网的观点和角度来对系统进行控制。

■ 支持对象实例的动态性

支持两种特殊 transition：前驱无、后继有唯一 state 的 transition 可在网执行(即系统运行)时动态创建对象实例；而后继无、前驱有唯一 state 的 transition 可动态地销毁对象实例。前者称为源泉(generator) transition，后者称为坟墓(tomb) transition。两种结构如图 2.6 所示。

图 2.6 源泉 transition 与坟墓 transition

■ 对于继承性的处理

原始 OPNets 模型号称是支持继承的，但没有任何的说明和体现。

在继承关系中的父类通常是几个子类的属性和动作的抽象，对于各个动作或行为之间的逻辑关系则并无定义和约束。而 Petri 网擅长于描述控制流和数据流，所表达的是存在实际行为逻辑的东西，故不便定义抽象的父类。进一步地，继承性是影响封装不便于分布式处理的[5]，因此在 OOPN 模型和已开发的 OOPN-IDE 工具中，并未支持继承性，而留待日后扩展。但是由于如后面所述的 OOPN 模型的实现基于 Java 语言，而 Java 本身支持继承性，所以排除了行为逻辑的继承的考虑之后，将来仍可增加对能带来重用的继承性的支持。

■ 对时间的支持

为了表达时间的概念，增加时间到 OPNets 中。具体地说有两种：

- place 加时间，表示进入其中的 token 要被锁定相应的时间之后才能进一步流动。此举可描述如网络延迟等概念。
- transition 加时间，表示 transition 的执行所需要的时间。

■ 引入抑制弧

为使 OPNets 网有“零”测试能力，故引入抑制弧(inhabit arc)。

但是只允许抑制弧从 mesQueue 连到 transition，而不允许出现从 state 连到 transition。这是因为，在 OPNets 中，既然以对象为主体，则对象不存在时，即 state 为空时，就谈不上什么动作，要动作就要有对象的参与。因此为了符合面向对象的原则施加了这条限制，毕竟经典 Petri 网中的 token 是没有自主性的。

2.2.3. OOPN 模型的形式化定义

1、系统

在 OOPN 模型中，系统由多重层次包含了实例的类及它们之间的消息传递关系组成：

SYSTEM = (O,R)

其中， O： 类集合

R： 关系集合

例如：图 2.4 所示系统：

O = {A,B,C,D}

R = {<oqB1,g1,iqA1>,
<oqB2,g2,iqA2>,
<oqA1,g3,iqC1>,
<oqA2,g4,iqD1>}

2、 类的外部结构

类 $O_i \in O$ 可表示为六元组：

$O_i = (H_i, IM_i, OM_i, F_i)$

其中，

H_i : 类层次，指定父类

IM_i : 输入 mesQueue 集合

OM_i : 输出 mesQueue 集合

F_i : 流关系集合

输入/输出 gate 用来连接相应的输出和输入 mesQueue，并进行必要的消息类型转换。

输入/输出 mesQueue 是对象的窗口，通过它们可以向外部发送消息要求服务并接收应答或从外部接收消息提供服务并返回应答。这样的消息传递机制有效地将对象的内部与外部相分离，提高了可维护性和可重用性。

流关系集合指示输入 gate 与输入 mesQueue、输出 gate 与输出 mesQueue 的连接关系。

3、 对象的内部结构

设 PO_i 表示简单类 i，CO_j 表示复合类 j，则整个系统的对象集合为：

$$O = PO \cup CO$$

其中，

$$PO = \bigcup_i PO_i \qquad CO = \bigcup_j CO_j$$

进一步地，我们用 IPO_i 和 ICO_j 分别表示 PO_i 和 CO_j 的内部结构：

●对复合类，其内部结构定义了该类所包含的内部类以及它们之间的相互关系：

$$ICO_j = (X,Y,R_j)$$

其中，

$X \in \rho(CO)$, $CO_j \notin X$

$Y \in \rho(PO)$

R_j: 关系集合

这里的ρ(CO)、ρ(PO)分别表示 CO、PO 的幂集。比如，图 1 中的类 A 的内部结构：

ICO_A =(Φ,{AA,AB},{<oqAA2,gA4,iqAB2>})

●对简单类，其内部结构明确指定了其静态特性与动态行为。静态特性是用代数方法来描述对象的某些属性或状态；而动态行为则用高级网来表示，因而具有更强的表达分析能力。

简单类的内部结构可定义为：

$$IPO_i = (D_i,SV_i,S_i,AT_i,LF_i,IN_i,M_0)$$

其中，

D_i : 属性集合

SV_i : 状态变量集合

S_i : state 集合

AT_i : 活动 transition 集合

LF_i：局部流关系集合

IN_i：实例集合

M_0：初始标识

属性与状态变量意思上很相似，都刻画简单类实例的当前状态，但前者常表示较长久的量(如人的年龄等)，而后者则常随实例状态的变化而变化(如机床的忙、闲)。

每个 state 都对应有刻画状态变量的一元状态谓词，定义状态谓词是通过将特定状态映射为简单类的状态值元组的一个函数来完成。

活动 transition(action transition)表示类实例的活动或动作，故特地加了“活动”二字，以区别同属 transition 的 gate， 起着同步作用，在它的先决条件满足时，就可以发生，去执行预先定义的活动。这些活动代表了顺序程序的执行。活动分为内部活动和外部活动， 主要取决于该活动是否为其他对象提供了服务

局部流关系表示了简单类的内部控制流，指示输入/输出 mesQueue 或 state 与活动 transition 之间的连接关系。

模型中有两种 token：一种代表对象的具体实例，只在对象内部流动，不允许在网执行期间被创建或销毁。实例由实例标识符唯一确定和引用。实例的初始状态由初始时存放实例的 state 表示；另一种 token 代表对象之间通讯的信息，可在对象之间流动，允许动态创建或销毁。

2.2.4. OOPN 模型的特点

- 提高表达能力的同时，保持了经典 Petri 网的原有特征(如图形表示、可执行)，使已有的一些分析方法和算法能继续使用。
- 利用了面向对象的抽象，有类和对象，能实现高度的共享。
- 有复合类和简单类之分，产生了层次和结构化，使系统建模直观，易于表达和理解，为分而治之地进行模型分析避免状态爆炸问题提供了可能。
- 引入 Java 语言与网结构有机地结合起来，可以定义具体的语义，为表达实际系统提供了可能。

正是由于 OOPN 模型具有这样的特点，我们基于它进一步研究了并发软件的开发方法。

3. 基于 OOPN 模型的并发软件开发方法

本章将在分析传统并发软件开发方法的基础上，提出基于 OOPN 模型的并发软件开发方法。

3.1. 传统开发方法

首先对传统的结构化的瀑布开发模式做简单的回顾[24]：

3.1.1. 阶段描述

其开发阶段大致可以分成三个，即分析阶段、设计阶段、实现阶段。

3.1.1.1. 分析阶段

本阶段，利用某种规格说明语言(specification model)来描述系统。这种模型是由一些逻辑上并发的单元组成的，称为分析并发单元(**A**nalysis **C**oncurrent **U**nit，简称 ACU)。ACU 有两种：基于状态的 ACU 和功能 ACU。前者通常是用状态转换图来建模，后者则是基于功能上的分解。

本阶段主要是将一个并发应用系统逻辑地分解成若干 ACU。对于系统的体系结构，如各部分之间的交互，要推迟到设计阶段。

3.1.1.2. 设计阶段

本阶段是将规格说明模型转化成设计模型，并为进一步地编码打基础。这就要求设计阶段的形式化结构与最终应用环境所提供的机制之间有严格的一一对应关系。后者包括基于优先级的进程调度及通讯同步机制。

设计模型由一系列交互的设计并发单元(**D**esign **C**oncurrent **U**nit，简称 DCU)组成。在实现阶段，每个 DCU 将转化成独立的进程。

在规格说明模型与设计模型间存在如下区别：

前者是层次化的，因其是采用自顶向下的分解；而后者则不然，因为进程结构是平面的(flat)。

ACU 与 DCU 之间没有一一对应关系。某些优化会使若干 ACU 集合成一个 DCU，或为处理通讯而引入没有 ACU 对应的 DCU。

另外，DCU 对应的进程在应用环境中的分布及交互机制也在本阶段确定。

3.1.1.3. 实现阶段

本阶段，每个 DCU 都要转化成一个顺序程序，程序中包含对操作系统服务的调用，从而来实现并发与通讯。而且，通讯机制的实施也要依赖于进程的分配。

3.1.2. 存在的缺点

瀑布模式虽曾风光一时，但其具有严重的缺陷使之已逐渐被人抛弃。如下从两个角度对其缺点进行讨论：

3.1.2.1. 从普通软件的角度

■ 需求分析不易充分

在需求分析中，分析员和用户是不同的思路来看待同一问题，很难充分地相互沟通，因而需求分析不易充分，最终可能导致项目部分地不能达到要求，进行弥补势必拖延工期和增加成本，甚至导致项目失败。

■ 功能分解具有随意性和缺少灵活性、可维护性差

采用自顶向下的方法很难构造合理的系统模型，因为自顶向下的“顶”是一个空中楼阁，缺乏坚实的基础，而且功能分解有相当大的任意性。尽管结构化分析和设计方法已相当地成熟，使软件的可维护性有了较大的改进，但从本质上讲，基于功能分解的软件是不易维护的。因为功能一旦有变化就会使开发的软件系统产生较大的变化，甚至推倒重来。更严重的是，在这种软件系统中，修改是困难的。由于种种原因，即使是微小的修改也可能引入新的错误。

■ 形式化验证困难

首先，规格说明与设计模型均不是可运行的(operational)，因而只能进行静态的检查；而且，由于从设计模型到实际的程序没有一种形式化的对应关系，使得各阶段的结果必须分别验证(validate)。

3.1.2.2. 从并发软件的角度

■ 调试困难

并发软件是多进程或多线程的，自然不如单控制流的程序易调试。一方面是由于多控制流使原本不方便的调试更显笨拙；另一方面并发软件的执行是不可重现的，出现的问题很难精确定位。

■ 运行代价高

由于基于状态的 DCU 容易理解，因此系统通常是由若干 DCU 组成的。而从 DCU 到进程的一一映射使进程数目过多，自然会增加运行时的代价。

■ 程序结构不清晰

最终得到的并发应用程序由两部分组成：行为部分(behavioral part)和与进程相关的部分(process-related part)。前者来自应用系统的固有逻辑，而后者则与同底层的操作系统或各种函数库进行交互有关。这两部分在系统中很难分开，这就使得：

很难对 DCU 在计算单元上的分布进行重新安排，因为这要改进与进程相关的那部分。

很难将该并发系统移植到其他的应用环境中，重用代价很大。

与进程相关的部分的开发需花费相当大的代价。因为由底层操作系统提供的各种机制通常难于理解且需要一些专门的技巧。

3.2. 面向对象的优势与问题

面向对象思想和技术的发展，对计算机科学各个领域的发展都是一个惊喜。面向对象技术应用于软件开发，使目标系统具有抽象能力强、可重用性好、易于修改和维护等优点。

3.2.1.对软件开发的改进

面向对象思想对软件开发上的促进表现在：

■ 用户和开发人员易于沟通，需求分析充分

面向对象软件开发方法学是基于对象模型的。对象模型中的类和对象是从实际系统中直观地抽取出来的。这样开发人员与用户就能够以相同或相似的思路进行讨论。用户与开发人员之间的共同语言一定程度上避免了传统需求分析中可能产生的种种问题，如需求分析不彻底、与用户要求不符和功能分解较随意等。

■ 可维护性、可重用性、可靠性好

应用面向对象技术，实际系统的功能是通过对对象的使用实现的。尽管功能是依赖于应用的细节的，并在开发过程中不断变化，但由于对象是客观抽取的，因此当需求变化时对象的性质要比对象的使用更为稳定；对象模型的抽象、封装和继承等机制更使系统易于调整和维护，从而使建立在对象结构上的软件系统较为稳定，提高了软件的可靠性与健壮性。

3.2.2.存在的不足

虽然面向对象对软件开发有了极大的促进，但是现有的面向对象思想、面向对象程序设计语言或面向对象方法学仍然存在一定问题。

■ 不完全的封装使系统错综复杂

面向对象的一大特征就是封装，各个对象是通过定义其外部可见的属性和方法来实现的。但是这种封装是不完全的。因为虽然通过界面的定义对外部的访问进行了约束，但是其他对象对其可见部分的访问是无法控制的。正是这种访问的随意性造成了对象间关系的复杂。

虽说对象间是仅通过消息来进行联系的，但是例如在面向对象程序设计语言中，这种消息传递对各个对象来说是透明的。

总而言之，不完全的封装造成了对象间的复杂访问关系和对象对访问其的操作的不可控。

■ 对象内部的固有逻辑没有清晰的描述和约束

对象中的全部属性构成了对象状态的描述，通过外部的访问，其属性值的改变使其状态发生改变。一般来说，一个对象的状态的改变是有其固有逻辑的，尤其是那些对应现实事物的对象，因而对此逻辑的清晰明确的描述和控制是需要的。但面向对象中并不支持这种描述和控制。

■ 继承性影响对象的封装及其固有的并发性

继承是面向对象机制所特有的，对对象的重用性起着决定作用。但是子类对父类的继承即访问则某种程度上破坏了对象的封装[5]。

对象的概念及消息传递的通信方式，在本质上是支持并发的，并发自然也需要共享。但是在分布式的环境中却很难实现继承所要求的父类代码的直接访问和父类的先期可见性，尽管在单机单用户的情况下很容易满足之。

■ 缺乏验证的手段

通过引入对象机制使系统得以层次化和模块化，一定程度上提供了进行形式化验证的可能。但是，现有的各种流行的面向对象方法学并不支持此种手段；即使退而求其次，也没有提供非形式化的但却友好方便的其他手段。

3.2.3.Petri 网的帮助

对于 OO 的上述问题，引入一种直观的形式化的东西来补充之不失为一种好的思路。结合并发软件开发的任务，我们考虑并发模型 Petri 网即为一种可行的选择。

在上一章中阐述的 OOPN 模型是从 Petri 网模型的发展的角度来引入的。目的是通过利用面向对象来解决 Petri 网模型存在的问题。实际上，两者进行结合，OO 施作用于 Petri 网的同时，Petri 网亦会反作用于 OO。这一点完全类似于物理学中的牛顿第三定律。

OOPN 模型中的 Petri 网部分是如何对 OO 进行优化的呢？

首先，OOPN 模型中引入的完全封装使对象可以对外界的请求进行控制。对象对外的接口表现在输入/输出 mesQueue 上，外部若要访问就必须在网结构上用 gate 将有关的 mesQueue 连接起来完成消息的传递。这种表达方式要比面向对象理论本身中的对对象外部可见接口的访问毫无控制的方式清晰得多，严谨得多。

其次，OOPN 模型中用网结构对对象的行为方式进行描述，对象与外界的交互中使用的消息传递亦是遵循网的引发规则的，从而使对象的固有逻辑得以保证。已有一些

模型对此做了一些工作，如并发对象语言 PROCOL 中使用“协议”来对访问者所调用的操作进行次序上的控制[5]，但明显不如 Petri 网来的自然和直观。

最后，OOPN 模型本来属于 Petri 网的高级形式，自然地可以通过仿真执行和静态分析来进行系统验证。

对面向对象中继承性的处理已在 2.2.2 节讲述。

3.3. 基于 OOPN 模型的并发软件开发模式

既然 OOPN 模型作为 Petri 网理论与面向对象思想的结合，将二者的优点集于一身，且能扬长避短，互补缺陷，因此基于 OOPN 模型来进行并发软件的开发就成为了自然的选择。

3.3.1.开发过程

由于没有必要且从当前的研究基础上看也没有可能用它去支持软件开发的整个生命周期及各个领域的应用，故完全可以利用已有的软件开发方法和技术实现 Petri 网不擅长的部分。

为了把 Petri 网在软件开发过程中的地位和作用说清楚，首先参照 Coad/Yourdon 方法把大致的开发过程描述如下[25]：

1. 首先进行如下活动：

- 抽取类及对象。从应用领域开始找出类及对象，形成整个应用的基础。
- 识别结构。该阶段分为两个步骤。第一，识别一般－特殊结构，该结构捕获了识别出的类的层次结构；第二，识别整体－部分结构，该结构用来表示一个对象如何成为另一个对象的一部分，以及多个对象如何组装成更大的对象。
- 定义属性。其中包括定义类的实例(对象)之间的实例连接。
- 定义服务。其中包括定义对象之间的消息连接。

2. 完成以上活动，即可建立一个由类及对象图表示的问题域模型，进一步地以此为基础，区分下列部分：

- 人机交互部分。这部分的活动包括描述人机交互的脚本，设计命令层次结构，设计详细的交互。
- 任务管理部分。这部分的活动包括识别任务(进程)、任务所提供的服务以及任务与其它进程和外界如何通信。

- 数据管理部分。这一部分依赖于存储技术，采用的是文件系统，还是关系数据库管理系统，还是面向对象数据库管理系统。

现在考虑基于 OOPN 模型的实现方法：

- 对于任务管理部分，基于前面分析的结果，即可建立简单的 OOPN 模型，然后进行仿真执行和静态分析加以验证。未发现问题时，即可进一步丰富该模型，使之能完全表达实际系统，从而得到任务管理部分的 OOPN 模型描述。
- 对于人机交互部分，可利用已有的可视化工具进行构造。
- 对于数据管理部分，也有诸多工具和技术可用。

在各部分实现的基础上即可利用如构件技术等手段进行集成，完成整个系统的实现。

如上各步有些只是简单的轮廓，尚有许多细节需要处理。但其中的如何应用 OOPN 模型来进行并发部分的表示和控制管理则已比较明确。在实际系统中，并发的任务管理部分所占的比重越大，OOPN 模型的作用越大。

其它的面向对象开发方法与 Coad/Yourdon 方法从思想上大同小异，不再赘述。

3.3.2.基于 OOPN 模型的并发系统建模

对于一个应用系统，建立 OOPN 模型分以下几步：

⑴确定组成系统的各个对象的结构。

①从研究的问题域里出现的名词中选出用来构成系统的对象的类。

②对每一个类确定其自身行为及与其它类的联系。

③对较复杂类，可进一步提取内部类，对每一内部类进行②中的分析；对已比较简单的类，用 OOPN 简单类构造出它的内部行为。

④用 mesQueue 表示类的外部接口。

⑤将内部控制中的 transition 与相应的 mesQueue 相连。

⑵构成系统静态结构。

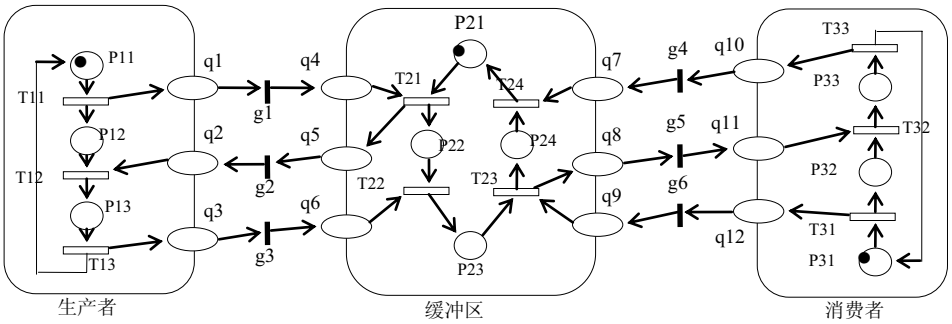
①确定类间的通讯关系。

②用 gate 将类间对应的输入 mesQueue 与输出 mesQueue 相连。

⑶确定系统的初始标识。

①确定构成系统的各类的对象实例。

②确定各实例的初始状态。



P11：生产者空闲。
P12：等待生产许可。
P13：正在生产。
T11：提出生产请求。
T12：获得生产许可。
T13：结束生产。

P21：缓冲区为空。
P22：等待生产结束。
P23：缓冲区为满。.
P24：等待消费结束。

P31：消费者空闲。
P32：等待消费许可。.
P33：正在消费。
T31：提出消费请求。
T32：获得消费许可。
T33：结束消费。

T21：收到生产请求并应答。
T22：缓冲区变满。
T23：收到消费请求并应答。
T24：缓冲区变空。

图 3.1　　生产者/消费者问题模型

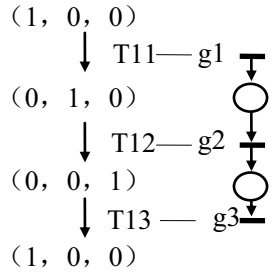


图 3.2 生产者的可达树与 IE 网

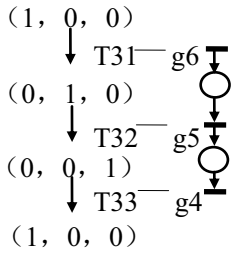


图 3.3 消费者的可达树与 IE 网

③将代表实例的各个 token 放入与各自初始状态对应的 state 中。

经过以上三步，就可以建立一个 OOPN 系统模型。图 3.1 给出了生产者/消费者问题的 OOPN 模型, 其中包含一个生产者、一个消费者和一个容量为 1 的缓冲区。

生产者(消费者)在进行生产(消费)之前，须先通过 g1(g6)向缓冲区提出请

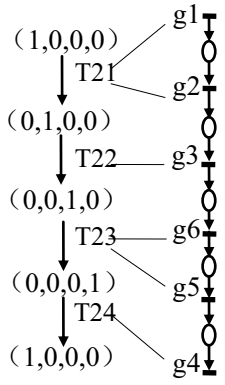


图 3.4 缓冲区的可达树与 IE 网

求，待应答消息通过 g2(g5)返回许可之后，才能开始生产(消费)。结束时，还要通过 g3(g4)告诉缓冲区。很显然，图 3.1 清楚地表示了系统涉及的生产者、消费者、缓冲区的结构及相互通讯关系。

3.3.3.OOPN 模型的死锁检测

3.3.3.1. 原始 OPNets 模型的死锁检测思想

建立了系统的 Petri 网模型之后，必须进行各种性能分析以确定该系统模型是否可靠及符合实际，这其中最重要的是死锁检测。

根据[12]中所述原则和思想，可得到 OPNets 模型的死锁检测过程：首先根据对象的内部结构，提取出对其输入/输出 gate 发生次序的要求，构造出接口等价网（Interface Equivalent Net,简称 IE 网），然后将不同对象的 IE 网合并，构成整个系统的 IE 网，通过建立 IE 网的可达树，分析其中是否存在死锁。

具体说分以下几步：

(1)局部分析

①对每一基本对象的内部行为进行可达树分析。

②利用可达树中各 transition 的发生次序来确定通过 mesQueue 与各 transition 相联系的输入/输出 gate 的发生次序。

如：与生产者相连接的几个 gate 的发生序列可由如下方法获得：

i) g1：通过 q1 与 T11 相连

ii) g2：通过 q2 与 T12 相连

iii) g3：通过 q3 与 T13 相连

③根据 gate 的发生次序构造 IE 网。

(2)同步分析

①将各 IE 网中出现的相同的 gate 合并，得到对象间的 IE 网。

②对得到的 IE 网进行可达树分析。

(3)层次抽象分析

①通过内层的 IE 网确定与内层各 gate 相连的外层复合对象的各 gate 的发生次序，构造出 IE 网。

②对得到的 IE 网进行可达树分析。

在上面三步中，若在某处发现有死锁，则建立的模型需要改进以消除死锁；否则，说明无死锁。以图 3.1 中的生产者/消费者系统为例进行分析，图 3.2、3.3、3.4 分别描述了该系统中的三个对象—生产者、消费者、缓冲区的 IE 网的构造过程，得到的整个系统的 IE 网与缓冲区的 IE 网恰好相同，显然该网无死锁。

3.3.3.2. 检测方法的两疑点

提出 OPNets 模型的文献[12]中的死锁分析方法有两处需要商榷：

1、违背了构成系统的基本单位是对象实例的原则。

在 OPNets 中，各个 transition 是与一定的语义即谓词相关的。不同的实例执行谓词描述相同的动作很可能会涉及不同的其它实例。尽管它们执行该动作时都是由于同一个 transition 的发生。比如在哲学家就餐问题中，两个哲学家执行“拿左筷子”的动作时涉及的筷子的实例便不同。但在[12]中，利用 IE 网来检测死锁的过程中，只是考虑了各个类之间的消息传递关系。实际上每个对象才是参与运作的独立实体，因此应该以对象为单位生成输入/输出 gate 的顺序关系，而不应以类为单位，只有这样才能反映系统的真实情况。

事实上，在有谓词参与的情况下，对象实例之间的关系是很难静态确定的，故也很难进行准确的静态死锁分析。但是，研究如何对谓词进行限制可以在检测时不考虑它或者研究不考虑时造成的后果如何则是可能的方向。

2、在对低层进行抽象时，丢掉了有用的信息。

在 OPNets 中，为了得到对象的 IE 网，实际上是用对象内部 transition 的发生次序来约束与它们相连的各个 gate 的发生次序。这种做法其实是没有根据的。这个道理可以很清楚地用如下的几个式子表示出来：

A→B、C→D、B→D

其中的→表示导出关系。即 A 成立可导出 B，C 成立可导出 D，B 成立可导出 D，但是显然不能依据如此条件推出来 A→C。这样的关系恰反应了两个一先一后的 transition 及与它们相连的 gate 的关系。更直接地说就是，不能因为 transition B 必须在 transition D 前发生就推出与 B 相连的 gate A 必须在与 D 相连的 gate C 之前发生。当然当是 D→C 成立而非 C→D 时，有 A→C。

这一问题的出现恐怕是没有对 Petri 网的发生规则深入分析的缘故。

寻找一个不丢失信息且能逐层抽象的解决方法还有待进一步的研究。

3.3.4.语言和工具的支持

任何一种软件开发模式，都要有与之相适应的具体的开发方法、语言和工具。方法为模式的实现提供指导原则和处理过程，语言用以描述这一处理过程，工具则是支持方法的实现和语言的描述的[8]。

前面已对 OOPN 模型在并发软件开发中的地位和作用进行了描述，而要应用它需要有相应的语言和工具支持。

考虑到 Java 语言的诸多优点，尤其是它的跨平台特性，以及大多数并发系统均要求分布的特点，结合 OOPN 模型中的语义描述，决定采用 Java 语言来支持并发软件开发。其中，Java 的多线程机制被用来表达并发。

有了语言，还要有开发工具。根据当前的研究情况，我们确定去且也已完成了一个以 OOPN 模型为核心的并发软件集成开发环境。

这一工具支持系统规格说明的建立，表现为 OOPN 模型的建立；规格说明的验证，表现为 OOPN 模型的动态与静态分析；规格说明的程序实现，表现为基于 OOPN 模型的 Java 语句代码的自动生成。

诚然在该工具中，并未显式地支持用户界面、数据库访问等，虽然这些部分或许是一个实际应用所必不可少的。要说明的是，本论文所做的工作只是整个方法研究与实现的第一步，还有很长的路要走。关于此方面可见本文最后的未来工作的展望部分。

尽管如此，该工具所生成的 Java 程序，仍可由用户直接嵌入程序语句去实现上述的功能。对于只包括任务管理成分的系统，如网络协议等，该工具则是提供了充分的支持。

关于该集成开发环境的设计与实现，将在如下的几章中介绍。

4. OOPN 集成开发环境——OOPN-IDE

上一章已把开发支持工具的必要性阐述得非常地清楚，本部分将在对几种已有的 Petri 网工具调研的基础上，提出开发 OOPN 集成开发环境的思想和原则，即软件需求，并进一步地对该工具的实际设计和实现进行阐述。

4.1. 常见的 Petri 网支持工具

当今知名的 Petri 网支持工具有以下几种：

- Design / CPN

Design/CPN 是由丹麦 Aarhus 大学和 MetaSoft 公司合作开发的一个商业化的工具。它基于有色网和 ML 语言来实现对系统进行设计、规格说明、仿真和验证，适用于通讯协议、分布式系统、自动制造系统、工作流和集成电路等领域。

它提供了丰富的手段用于仿真和形式化分析，但是使用它需要掌握较深的理论知识，且不是基于面向对象的 Petri 网模型，故只能适用于较小规模的系统。

- LOOPN

LOOPN 是由澳大利亚 Tasmania 大学计算机系开发的一个基于第二章中已提及的同名的面向对象 Petri 网语言(LOOPN)的仿真器。它主要是为网络协议的建模和仿真设计的。

它提供了较为方便的仿真和调试机制，但是，使用它要掌握较多的数学理论和复杂的定义，网的层次化不直观。

- PESIM

PESIM 是由捷克 Brno 技术大学计算机工程系研制开发的 Windows 下处理 P/T 网的专用工具，界面友好，操作简便，但功能很弱。

更多的 Petri 网支持工具的详细列表可在[26]中获得。

对所有工具的功能进行总结，从支持并发软件开发的角度考虑有下列结论：

1、绝大多数工具只支持某一领域的应用建模和分析，而不具有普遍性。

2、所有的工具都不能直接用于实际系统的开发，而是间接地通过提供形式化的描述来辅助开发。

3、所有工具均不能跨平台运行，只支持 Windows，或只支持 UNIX。

4、有些工具采用基于复杂数学理论的模型，使用难度大。

5、多数工具所基于的 Petri 网本身并不能简洁、清晰、高效地表达系统，比如所基于的只是低级的 Petri 网。

基于如此认识，对要开发的 OOPN 集成开发环境就有了一个原则上的需求描述：

1、当今的并发软件通常是分布式的，而且通常是异构平台，为了使该工具有充分的支持能力，需要其能跨平台运行。

2、为具有强大的建模能力，需要基于好的模型，能直观、高效地描述目标系统，掌握它应不要求有很深的理论基础。

3、工具应提供丰富的手段使之简便易用。

4、支持普遍的建模，不只是专用的系统。尽管这样会使得效率等有所下降，但既然已有了一个通用的框架和基础，完全可以进一步地为特定应用量身裁制。

基于如此描述，我们开发了基于 OOPN 模型的集成开发环境(**OOPN I**ntegrated **D**evelopment **E**nvironment，简称 OOPN-IDE)。

4.2. OOPN 集成开发环境——OOPN-IDE

4.2.1.软件描述

OOPN-IDE 基于 Sun 公司的 JDK(**J**ava **D**evelopment **K**it)开发工具包环境开发，并随着 JDK 版本的提高使用最新版，最后的开发与运行环境是 Java2 平台(曾被称为 JDK1.2)。由于 Java 的跨平台能力及设计和实现上的详尽考虑，OOPN-IDE 可在 Microsoft Window 95/NT 及各种 UNIX 操作系统上运行。

OOPN-IDE 可完成图形编辑、文本编辑、Java 类的生成、图形编辑的多机协同、仿真运作以及静态与动态死锁检测等功能。

4.2.2.需求描述

按照 3.3.2 节阐述的基于 OOPN 模型的目标系统建模过程，使用 OOPN-IDE 的流程如下所示，同时也是 OOPN-IDE 的技术实现的简单需求描述：

1、利用图形编辑工具或文本编辑工具新建 OOPN 简单类。

2、在图形编辑窗口或文本编辑窗口中为该简单类添加组件，并为添加的组件建立连接。

3、使用 Java 语句为该简单类及其中的组件定义属性。

4、定义完毕后保存该简单类，OOPN-IDE 将对其进行编译并生成 OOPN 实现类(Java 源码及相应字节码)。

5、在已创建的简单类的基础上，根据需要，以图形或文本方式新建 OOPN 复合类。

6、选择已经创建的模型类作为复合类的内部类，并利用 gate 在内部类之间建立连接。

7、定义完毕后保存该复合类，OOPN-IDE 将对其进行编译并生成 OOPN 实现类。

到此，可定义完毕目标系统中所需的所有对象模板即类，而这些只是属于目标系统的静态部分，其中的一个最顶层的包容其它的类作为整个系统的模板。

8、以图形或文本方式新建 OOPN 模型系统，建立时要指定所基于的模板，即上述的最顶层的类。

9、为该模型系统添加 token，并为 token 定义属性，定义的 token 是系统中的动态部分，每个 token 均表示它所在的类的实例。

10、模型系统定义完毕编译保存后，基于生成的 Java 代码对应的字节码即可进行仿真和运作。

另有几点需要说明：

1、消息类：

为了进行对象间的消息传递，可以针对某 mesQueue 定义其中可存放的消息的类型，表现形式即为 Java 类，用该类的名字表示，作为 mesQueue 的属性。

2、OOPN 实现类：实际就是 Java 类，用来支持 OOPN 模型的实现，故称为实现类。

3、多机协同：为支持多用户对较大系统的协同建模，提供多机协同功能，多个用户可同时对某一系统中的不同的类或同一个类进行编辑。

4.2.3.工作机理

如图 4.1 所示为本工具的工作机理。整个环境的结构为 Client/Server 模式，以支持同一模型系统的分布式协同建模和管理。Client 端完成与用户的交互，Server 端负责多个 Client 的协同。

用户可采用两种建模方式：图形和文本。文本是有预定义格式的，输入的文本需要进行语法检查。

对模型中用 Java 语言描述的部分调用 Java 编译器进行语法检查。通过检查后，即可生成 OOPN 实现类。OOPN 模型类及其实现类均要放到位于 Server 上的相应类库中。

OOPN 网中的各种图形元素都设计成 Java 类，组成了 Java 类库的 OOPN 扩展部分，在图形建模及仿真中调用。

由网络通讯模块完成 Client 与 Server 的通讯。

Server 负责存储和管理用户建立的 OOPN 模型类及相应实现类。

Server 对 Client 希望得到的各种信息提供服务，对 Client 反馈的信息进行应答和控制，对多 Client 的协同建模操作进行必要的一致性检查。

4.2.4.模块设计

如图 4.2 所示为本软件的总体设计结构，分为六部分：初始化部分、用户界面部分、网络通讯 Client 部分、网络通讯 Server 部分、OOPN 模型支持类部分、系统环境配置部分。

各部分均与若干 Java 程序包相对应，各个包所包含的 Java 类的列表见附录一。

初始化部分负责系统的启动和初始化。在启动时，用户可通过参数指定是启动客户还是启动服务器还是两者都启动，而本部分正是通过对参数的判定来决定初始化哪些部分的。

用户界面部分在整个系统中占的份量最重，负责可视化界面的显示、控制和配置，与用户直接交互。对涉及多机协同的文件操作和编辑操作，本部分是把一定的信息传递给网络通讯的 Client 部分，由后者对信息进行指令编码后再行传递。

网络通讯的 Client 部分负责将用户协同建模型的各种操作命令传至 Server 端，同时接收 Server 端传来的各种响应，并将其分派到适当的部分进行处理。

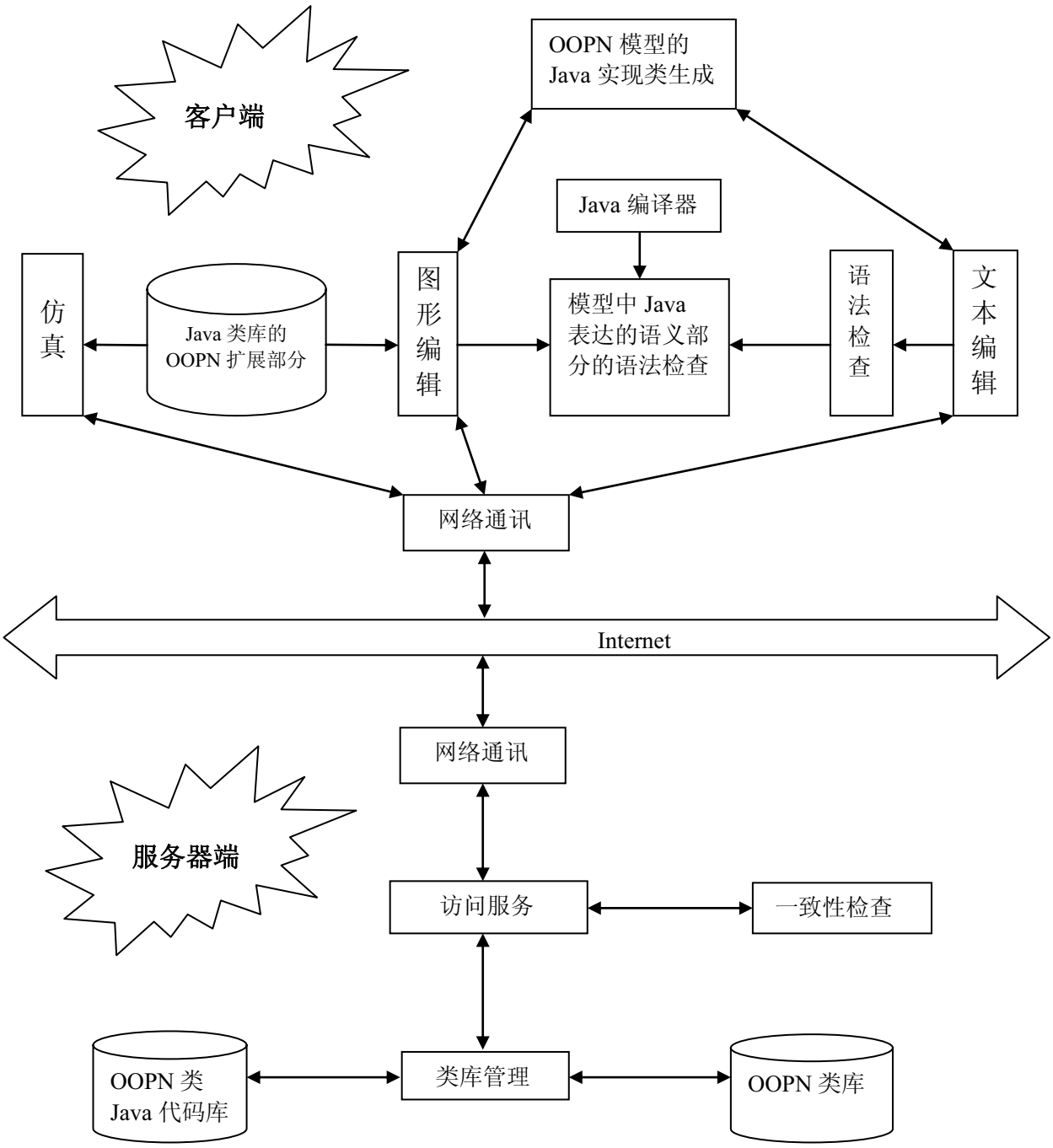


图 4.1　并发软件开发 OOPN 支持工具

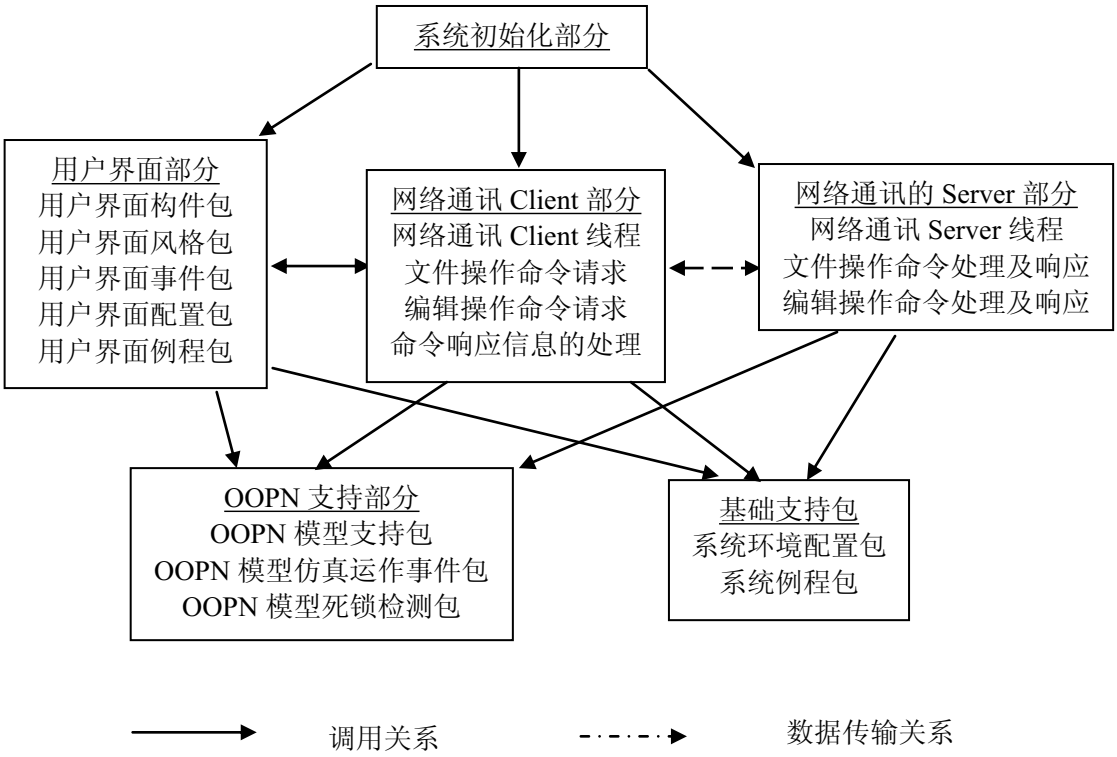


图 4.2　功能模块划分

网络通讯的 Server 部分负责监听 Client 方传来的各种文件操作和编辑操作命令，然后执行之，若执行成功，则给提交者返回成功信息，否则返回失败的信息。若是编辑操作成功，Server 还要发送给每一个正在当前模型上进行协同编辑的客户一条命令，通知他们进行协同改动。

OOPN 支持部分中定义若干 Java 类来描述 OOPN 模型中的各种组件的属性和行为，并在此基础上对 OOPN 模型的数据结构进行了设计。另外，还提供了各种消息传递和控制机制来完成仿真与运作，及提供了若干个类来完成对静态死锁检测中所需数据结构和算法的支持。

基础支持包中提供了各种系统环境常量及对各种环境配置手段的支持。另外为整个系统的设计提供例程支持。

4.2.5.数据结构设计

由于涉及较复杂的模型的定义，系统设计中数据结构的设计至关重要。

4.2.5.1. 逻辑结构设计

系统中逻辑数据结构的设计主要分以下几方面：

4.2.5.1.1. 用于 OOPN 模型描述的数据结构

在系统中的许多部分都用到 OOPN 模型，实际上 OOPN 模型的结构的设计可以说是本工具的核心。

在 OOPN 模型系统中，同一个模型类是可能在目标系统中出现多次的，而且用来表示对象实例的 token 是成组地存在于某一处出现的模型类的网结构中的，它们共享该结构。我们把这样的一处网结构称为模型类的一次**出现**。为了实现这样的关系，设计了 OPNClassShow 与 OPNClassInst 两个类，分别用来抽象用户定义的具体的模型类的两个方面：网结构，属性与动作语义。

通过对 OOPN 模型的分析并尽可能地提高重用程度，设计了如图 4.3 所示的类层次。

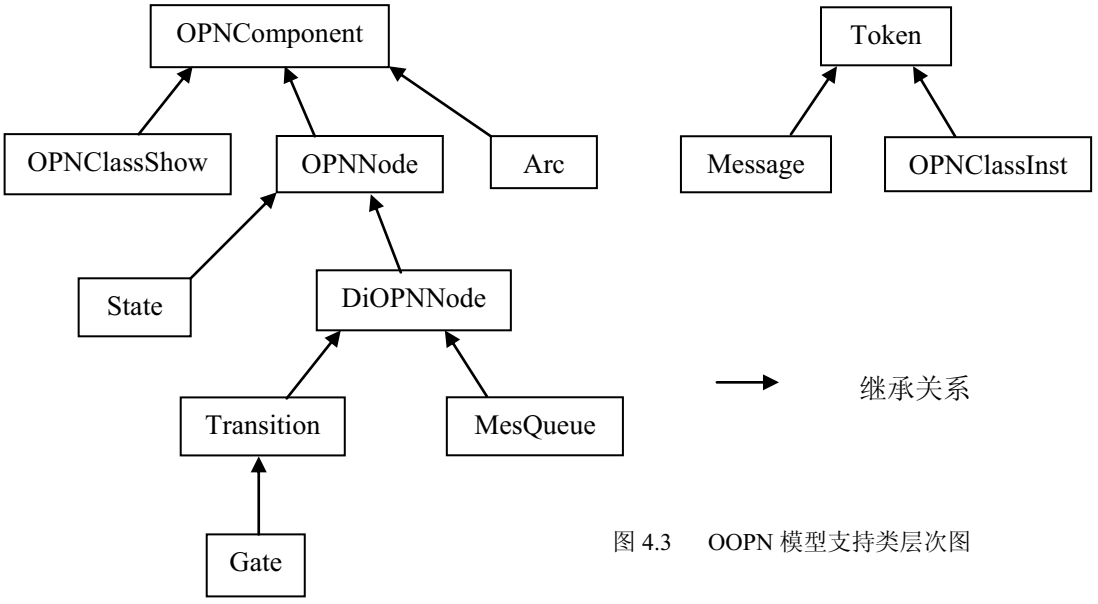


图 4.3　OOPN 模型支持类层次图

1、OPNComponent 是 OOPN 模型中涉及到的所有组件的父类。其中定义了所有的组件所共有的属性和方法。

属性包括：名字、颜色、是否被选中、属性是否可被修改、是否显示名字、所属的 OOPN 模型类实例等。

方法包括：名字的设置和获取、颜色的设置和获取、被选中属性的设置和获取、显示名字属性的设置和获取、热点框的获取(热点框是指要选中某个组件，必须去点击的其所在的方形范围。通过判定是否点中某个热点框即可知道是否选中了对应的组件。)、移动、绘制、选中状态下的绘制、显示名字等。有些方法只是给出了缺省的内容，具体到某种组件，还要具体情况具体定义。

2、OPNClassShow 类是 OPNComponent 的三个直接子类之一。其实例是一个具有完整独立意义的实体，用来表达一个 OOPN 模型类的一次出现。即在实际的实现中，目标系统中的同一模型类的多处出现对应了 OPNClassShow 的多个实例。

OPNClassShow 中含有一定的数据结构来描述对应的模型的结构，具体包括：

graphStruc 用来保存模型中的各个 Transition、State、Gate(当然根据 OOPN 模型的定义，Gate 与 Transition 不可能同时出现在其中)、MesQueue。

mesQueues 保存所有的 MesQueue，虽然在 graphStruc 中已含有了 MesQueue 组件，但在很多时候需要仅去访问 MesQueue。这种情况下，为了提高性能和效率，故专门设置 mesQueues 来存放所有 MesQueue 的引用。

arcs 中保存了所有的 Arc。

innerClasses 中保存了所有的内部类，即由当前的模型类所包容的更小的模型类。

noPreStatesTran 中保存了所有的无前驱 State 的 Transition 或 Gate 的引用。在仿真运作过程中，无前驱 State 的 Transition 由模型类本身来判定发生，相当于 Java 类代码中的静态方法，不存在实例的情况下即可发生。而有前驱 State 的 Transition 或 Gate 的引发则由存在于 State 中的模型类的实例来驱动引发。

以上的各个结构为了访问的方便，均设计成哈希表的形式，其中均是由组件名和组件自身构成的值对，当要访问某个组件时，通过提供该组件的名字来实现。

allInstances 中保存了属于当前模型类的所有的实例。对简单类来说，其中都是自身的实例；而对复合类来说，则是其所有的分布于各个子层的模型类的实例的集合。allInstances 的存储结构是线性的。

OPNClassShow 中虽然提供了对模型类中的所有组件及其他信息的访问手段，但是类中的各个组件之间还有关联。这一点则由各个组件来完成相互之间的引用。

3、OPNNode 是 OPNComponent 的另一直接子类。它用来表示传统 Petri 网中的节点。该类中的几乎所有属性和方法正是基于此来进行设计的。

OPNNode 中的 pres 属性是一个线性结构，用来保存当前组件的所有前驱。据此即可访问组件的前驱。

OPNNode 中的 posts 属性是一个线性结构，用来保存当前组件的所有后继。据此即可访问组件的后继。

为了支持对相互引用关系的管理，提供了如下方法：

getPreNodes：　获取所有的前驱节点的列表；

getPostNodes：　获取所有的后继节点的列表；

addPreCom：　增加一个前驱组件到 pres 中；

addPostCom：　增加一个后继组件到 posts 中；

remPreCom：　从 pres 中删除一个前驱组件的引用；

remPostCom：　从 posts 中删除一个前驱组件的引用；

remAllComs：　删除所有的前驱组件和后继组件。

为表示节点的具体位置，用 x、y 两个变量定义当前节点的中心点坐标。用 boundingBox 表示当前节点的热点框。

相应地设置了若干方法来支持对位置的访问：

setLocation：　设置中心点的坐标；

getBoundingBox：获取热点框的信息，热点框是一个矩形；

setBoundingBox：用来设置热点框，还有待各子类“因地制宜”；

canBeSelected：　用来判定提供的坐标点是否可选中当前节点。

对于节点来说，是用弧线连接起来的，即 State、Transition 或 Gate 是通过 Arc 连在一起的。而用户在两个节点之间建立连接时进行的输入，仅仅是选中两个节点，而弧线则是要自动生成的。而弧线为了美观，应该是恰好连接两个节点的边缘。这就要求将用户输入的仅仅用来选中组件的点调整到适宜的节点的边缘。故在 OPNNode 中定义了方法 adjustPoint，来将指定的线段中的起始点，调节到节点的边缘。

对于节点来说，在可视化的界面上，名字是要显示出来的。名字的显示相对于节点有九个位置：

SW：　左下；

S：　正下；

SE：　右下；

W：　正左；

C：　正中；

E：　正右；

NW：　左上；

N：　正上；

NE：　右上。

以上也是为了提供此种支持定义的所有常量。为了表示某个节点的具体的名字的显示位置，定义了 nameLoc，其值即在如上说明的常量中取。

为提供对时间的支持，在 OPNNode 上引入时间，定义了 time 和 timeUnit 两个属性，分别表示时间的具体数值和单位。时间的作用表现在两方面：对 State 来说，是指示存在于其中的实例必须停留相应时间之后才允许进一步地流动；而对 Transition 来说，是指示其的发生要花费相应长短的时间。

4、Arc 是 OPNComponent 的最后一个直接子类。它用来表示 Petri 网中节点之间的连接关系，即流关系。

从逻辑关系上讲，每一条弧都有唯一的前驱和后继。由于这样的两个端点必然是 OPNNode 的实例，因此在 Arc 中定义了 pre 和 post 两个 OPNNode 类型的变量，分别表示当前弧的前驱和后继。

相应地，为了访问和设置前驱和后继，定义了如下的方法：

setPre： 设置前驱；

getPre： 获取前驱；

setPost： 设置后继；

getPost： 获取后继。

在 OOPN 模型中，弧有两种：普通的弧和抑制弧。对于两者逻辑意义上的区别是在系统仿真的时候才显示出来的；而在图形表示上，唯一的区别是前者的末端是箭头，而后者的末端是一实心圆。Arc 中定义了两个常量来表示两种类型：

Common：表示普通弧；

Inhabit： 表示抑制弧。

定义了一个变量 type 来表示当前弧是属于何种类型的。

在图形表示上，Arc 的位置由用户输入的若干个点的坐标来确定。原始的用户输入的点的坐标值存放在名为 ps 的变量中。

对于弧来说，有两种图形显示方式：折线和圆滑曲线。对于前者，是直接在各个输入点之间连直线段来完成的；而对于后者，则是根据一定算法生成一条通过所有输入点的贝齐尔曲线来绘制显示的。为了提高显示的速度，Arc 中专门为贝齐尔曲线的显示定义了一个变量 psAdjust，用来存放该曲线的所有控制点。

为了支持用户对弧的输入过程，定义了方法 addPoint 来增加一个新点到弧中。

另外 Arc 中还对继承的若干抽象方法给出了具体的定义。

5、State 是 OPNNode 的直接子类。

为了表示实例信息，定义了 instances，其中包含了所有当前存在于 State 中的实例的引用。为支持实例增加和消去，定义了如下的方法：

getInstance：　获取具有指定名字的实例；

addInstance：　增加一实例到 State 中；

removeInstance：消去具有指定名字的实例；

getInstances：　获取所有实例信息；

removeInstances：消去所有实例；

setInstances：　设置 State 中的所有实例为指定的实例集合。

在图形表示上，为了表达 State 的大小，定义了变量 r，指示该 State 的半径大小。

6、DiOPNNode 是 OPNNode 的直接子类，作为 Transition 和 MesQueue 的父类，用来表示一个有方向的节点。

在 DiOPNNode 中，只定义了 xRad 和 yRad 两个变量，分别表示 Transition 或 MesQueue 的 x 方向上的半径和 y 方向上的半径。

7、Transition 是 DiOPNNode 的直接子类，用来表示传统 Petri 网中的 transition。

Transition 中为了方便地访问其前驱的和后继的 MesQueue，专门定义了两个变量：preMesQueues 和 postMesQueues。其中分别存放当前 Transition 的所有前驱 MesQueue 和后继 MesQueue。

为了表达 Transition 的语义，定义了 preCondContent 和 actionContent 两个变量。其中以正文形式保存用户输入的用来确定 Transition 语义的前驱发生条件和发生时执行的动作。

另外，Transition 可有几种类型：

Gate：　表示当前的 Transition 是一个输入/输出 gate。

Generator：表示当前的 Transition 只有一个后继 State，而无前驱 State。

Tomb：　表示当前的 Transition 只有一个前驱 State，而无后继 State。

Common：表示当前的 Transition 有一个后继 State，且有一个前驱 State。

以上类型的划分在系统仿真运作的处理中是非常有用的。

8、MesQueue 是 DiOPNNode 的直接子类，用来表示 OOPN 模型中所特有的输入/输出 mesQueue。

MesQueue 用来存放消息实例，定义了变量 mes 保存所有当前的消息。

MesQueue 可有两种方向类型：输入的和输出的。为此定义了三个常量：

In：　表示是输入的；

Out：　　表示是输出的；

Unknown：表示传递方向待定。

另外，MesQueue 中允许存放的消息亦是有一定类型的，称为消息类型。为了表示消息类型，定义了 mesType 变量，用来表示其中存放的消息的类型。mesType 实际上是一个字符串，其中存放某个 Java 类的名字，而正是这个 Java 类定义了消息的类型。

9、Gate 是 Transition 的直接子类，用来完成 OOPN 模型中所特有的对应的输入 mesQueue 和输出 mesQueue 之间的消息传输关系。

在 Gate 中并未有进一步的属性的定义，而只是在本身的绘制方法上进行了重新的定义，以区别于普通的 Transition。

10、Token 类用来描述系统中所有的实例，包括对象实例和消息实例，从而自然地成为描述对象实例的 OPNClassInst 类和描述消息实例的 Message 类的父类。

每个实例都有自己的名字，为此定义了变量 name，相应地定义了改变和获知的方法：setName 和 getName。

为了进行图形上的表示，定义了：

x、y：　表示当前实例所处的位置；

r：　　表示实例的半径大小；

c：　　表示实例的颜色。

11、Message 类用来描述消息实例。

用户可以继承本类并进一步定义具有特定属性的消息实例。Message 类本身没有定义什么实质性的方法和属性。

12、OPNClassInst 类用来描述对象实例。

定义了 show 来指示其所在的 OPNClassShow，即所属模型类，或说是谁的实例。

定义了 curState 来指示实例当前所处的 State，从而使得每个对象实例均能知道其在 OOPN 模型类的网结构中的位置，这是在仿真和运作时所必需的。

定义了 initAction 来描述该实例所特有的初始化动作。在仿真运行时，该动作将在创建了具体的 OPNClassInst 实例之后首先得到执行。

对以上属性，又定义了如下的方法：

setClassShow：　设置当前实例所属的 OPNClassShow 实例；

getClassShow：　获取当前实例所属的 OPNClassShow 实例；

setInitAction：　设置本实例特有的初始化动作；

getInitAction： 获取本实例特有的初始化动作；
setCurState： 设置所处的 State；
getCurState： 获取所处的 State。

总之，OPNClassShow 用来描述一个完整的 OOPN 模型类，通过 OPNClassShow 中的数据结构可以访问模型中的各种组件，而各个组件之间又有相互的引用，从而构成了网结构。OPNClassShow 和 State，均可以访问各自所辖范围的对象实例，而对象实例亦可方便地访问其所在的模型类及 State。如上的各个类即共同定义了一个完整的既有内部结构又有外部接口的 OOPN 模型类，为整个集成开发环境的实现打下了基础。

4.2.5.1.2.用于类库的数据结构

使用本工具可以对某个系统进行建模，整个目标系统是一个多层次的网结构。只有在设计好低层类的基础上，才能进行上一层的类的建模。在整个系统的建模过程中，已建立的可用做模板的类需要在一定机构的管理下进行使用，可以做为更大的类的内部类，或做为定义系统时选用的模板。为了实现这样的支持，考虑设置一个类库，对已建立的可用做模板的类进行管理。

类 ClassLib 的设置即是上述思想的产物。在 ClassLib 中，定义了变量 classPairs，用来保存类库中的类(模板)，形式上表现为一系列的值对，各个类的名字作为键值，而相应的类作为被管理的数值对象。

当需要在界面上提供类库的信息时，构造一下拉式选择框 JComboBox，其中存放有各个模板类的名字，通过选择，即可从类库中获得相应的类做进一步的操作。

4.2.5.1.3.用于消息库的数据结构

与类库的设置相类似，由于建立的模型系统中的 MesQueue 可能需要能够存放携带特定信息的消息实例，因此要提供对特定消息类的定义与管理。

对于前者，将在用户界面设计部分进行说明；而为了实现管理，即引入消息库，相应的类为 MessageChoice，其中定义了变量 MessagePairs，用来存放可用的消息类的信息。在用户界面上，表现为在对 MesQueue 的属性进行定义时有一下拉式选择框供选择，达到设置该 MesQueue 可容纳的消息的类型的目的。

4.2.5.1.4.用于环境配置信息的数据结构

对于集成开发环境来说，有些信息是需要在连续的两次系统执行中进行传递的。这些信息反映了用户对系统的配置项目的值的修改。可以概括为如下四个方面：

- OOPN 模型组件属性方面：

对于模型中的属性，如缺省的大小、颜色等都是有系统初始值的，但是必须提供给用户一种能对其进行改变的方式，且改动后的值应该能传递到系统的执行中。

GraphOptions 类即用来完成如上目的的。它实现了定义有各种组件属性的系统初始值的接口类 ComPropertyCollI，进而为了能够改动，又定义了一系列的变量，存放用户自定义的值。

- 主窗口的位置和大小信息方面：

用 XyFrameOptions 类定义了变量 frameSize 和 frameLocation，分别用来表示和记录窗口的大小和位置。

- 文件对话框的当前目录和过滤器(filter)信息方面：

FileDialogOptions 类完成此功能，定义了 dir 和 filter 分别存放目录信息和当前的过滤器。

- 仿真运作中使用的若干配置信息方面：

在仿真中，用户可以选择是否同时进行动态的死锁检测，以及进行检测所用的观察时间段(如果系统在指定的时间内无进展而又没有结束，则认为已发生死锁)。

定义了 SimulationOptions 类，其中定义了 DeadlockInterval 和 DeadlockDetect。用来分别表达观察时间段和是否进行死锁检测。

以上的四个方面，构成了整个系统的配置信息。为了进行管理，定义了 OptionsMngr 类，间接地通过哈希表式的结构，将各个方面对应的配置信息类的实例存入其中。

4.2.5.1.5.多机协同数据传输的逻辑结构

在多机协同中，为了表达 Client 与 Server 之间的多种操作命令与响应，设置一个 Hashtable(哈希表，Java 类库中的类)的实例来表达一条信息，表达完整的一次文件或编辑操作。具体的指令格式见附录三。

4.2.5.1.6. 对 OOPN 模型系统进行死锁分析的数据结构

为了对 OOPN 模型系统进行死锁分析，特设计了如下的类(由于本部分是在完成了建模和仿真之后又新增加的内容，因此不便再使用 4.1.1 中的图 3 所示的结构，因为那里使用的类中定义了太多的属性，将为死锁分析带来太多的开销)：

SimpleNetCom：相当于 OPNComponent。定义了 name 和 show 两个变量，分别表示名字和所属的 OOPN 类。另外，定义了一个 Object 类型的变量 tag，可以赋上任意的 Java 对象实例，用在死锁检测算法中记录某些信息，或者打标记。

SimpleNetNode：相当于 OPNNode。定义了类似于 OPNNode 中的 pres 和 posts，还有若干方法定义。

SimpleClassShow：相当于 OPNClassShow，

SimpleState：相当于 State。只是定义了一个名为 tokenNum 的 int 型的变量，表示当前 SimpleState 中存在的 token 数目。

SimpleTransition：相当于 Transition 或 Gate。没有定义任何属性，存在的若干方法亦是为死锁检测服务的。

SimplePetriNet：用来表示一个传统的 Petri 网结构。其中定义了两个变量 states 和 transitions，分别存放网中的 place 元素和 transition 元素。而它们之间则通过引用进行访问，构成了一个网结构。

4.2.5.1.7. 用户建立的模型的编译错误显示所用的逻辑结构

用户建立的 OOPN 模型系统或其中使用的模型类均是要转变成 Java 语言来进行实现的，由于模型中的具体语义是用户通过 Java 语言来输入的，因此错误在所难免，必须提供某种机制提示用户在哪些地方出现了错误。

对 Java 源码的编译是利用未公开使用的 sun.tools.javac.Main 类来进行的，通过调用该类实例的 run 方法进行编译，即会得到生成的 Java 字节码类文件或失败时提供错误信息。错误信息是有一定格式的，但其本是用于显示在 DOS 窗口中的。为了能友好地交互式地浏览错误信息，专门设计了一个窗口处理之，即 VisCompile 类。

在 VisCompile 中，每条错误的点击都会使对应错误语句位置加亮。

另外，设计如下的类：

```
class InfoSet{
        public int fileNameIndex = -1;
        public int lineNo = -1;
        public StartEnd startEnd = null;
```

```
}

class StartEnd{
        public int s;
        public int e;
}
```

其中均略去了方法的定义。InfoSet 用来表示一条错误信息。fileNameIndex 表示当前的错误来自哪个文件，因为有时同时要编译多个文件；lineNo 指示错误语句所在的行；startEnd 则可以记录错误语句所在行的起始和终止位置。

利用与某条错误相关的 InfoSet 实例即可很方便地找到错误出现的位置，从而方便地显示出来。

4.2.5.2. 物理结构设计

本系统中物理结构的设计包括如下所示的三项：

4.2.5.2.1. 保存的 OOPN 模型的正文形式表示

用户建立的 OOPN 模型类和系统是按照预定义的正文脚本来保存的。而用户定义的消息类则是在已有的程序框架的基础之上，通过填充 Java 语句片段来完成的，保存时也是按正文形式保存到文件中的。

对于模型类和系统的正文脚本的格式定义见附录二。

本环境规定所有定义的消息类都是 Message 类的派生类。在创建消息类时，键入有效的消息类名之后，即会自动生成如下所示的程序框架：

```
package opn.opnmessage;
public class New-Message extends opn.baseclass.Message{
}
```

然后用户在“{}”之间输入 Java 语句即可定义消息类的属性。

对于正文形式表达的 OOPN 模型类和系统的生成与载入是通过两个类来实现的：OPNInput 与 OPNOutput。生成相当于将程序形式的 OOPN 模型按上面的脚本语言生成

正文文件，而载入则是对某个脚本文件读入进行语法分析和词法分析，还原成内存中的程序形式的 OOPN 模型。

4.2.5.2.2. 保存的程序结构形式的 OOPN 模型

上面的脚本语言及 OPNInput 类、OPNOutput 类一起解决了 OOPN 模型类和系统的保存与读入问题，但是可能的问题是进行读入时词法与语法分析将耗费大量的时间。这一问题对于单个文件来说，可能还不是太明显，但若是要读多个文件，则会出现用户无法忍受的延迟。类库的加载恰存在这种情况。为此在本工具启动时，要先将库中的可用作模板的类读入，以便将来被使用。

为解决这一问题，充分发挥 Java 语言的优势，采用直接将程序形式的类库内容序列化(serialize)存到硬盘上的方法。Java 语言中提供了序列化的强大支持。在硬盘上用一文件存着类库的内容。当系统重新启动时，直接读入该文件，然后反序列化即可迅速得到原类库，从而解决了装入速度慢的问题。

4.2.5.2.3. 保存的系统配置信息的结构

传统上，一个软件系统的配置信息是以正文形式存放于某个文件中的，即使有时不是正文形式，也要一个一个地将有关配置值写入文件中，非常麻烦。

既然 Java 有序列化机制，而从上面的 4.2.5.1.4 节可知，所有程序形式的配置信息是由 OptionsMngr 统一管理的，故在系统退出时将 OptionsMngr 实例直接序列化存到盘上，当系统再次启动时，再反序列化，即可还原配置信息，这就避免了过多的细节处理。

当然在系统初次运行或配置文件被删去后，该文件都会找不到，这时系统会按缺省值生成所有配置信息，而不会出错。

4.2.6.运行设计

为了提供用户充分的选择及尽量减少运行的负载，运行模块的组合有四种方式：只启动 Client 部分、只启动 Server 部分、Client 及 Server 两部分均启动及独立的仿真环境。这里把集成开发环境看作由 Client 和 Server 两部分组成。当然这里的 Client 与 Server 的概念与上面提到的网络通讯部分的 Client 端和 Server 端是不同的。可以描述如下：

1、集成环境只启动 Client 部分：

这种情况下的运行模块包括：用户界面部分和 Client 线程部分。进行多机协同时必须连接到其它机器的 Server 上。

2、集成环境只启动 Server 部分：

这时的运行模块包括：Server 部分中的 Server 线程部分及对各个 Client 传来的命令进行处理的模块。该部分只负责协同。

3、既启动 Client 部分也启动 Server 部分：

运行模块包括：用户界面部分、网络通讯部分中的 Client 端和 Server 端。

4、独立的仿真环境：

运行模块包括：用户界面中的仿真主窗口及仿真运作程序。此种情况用来直接运行指定的目标系统，而不必进入集成环境。

4.2.7.系统出错处理设计

4.2.7.1. 出错情况

系统在运行中可能会有两种预想不到的问题出现，分别是：

- 由于系统设计的不周造成异常或错误的发生，比如可能数组越界、空引用操作、找不到类等问题。
- 由于用户在建模时输入了不合法的 Java 语句造成模型系统在运行时出现异常或错误。

4.2.7.2. 补救措施

对于如上列出的两种可能的问题或错误，分别采用如下的方法解决之：

- 工具本身的问题：

由于出现的问题的确切地点无法预先知道，但是可以在极可能出现问题的地方利用 Java 语言的抛出异常的机制来使异常在程序的控制下进行向上的传递。这样在一定的

层次上通过对捕获的异常的处理，即可避免由于预料之外的异常的出现而使整个系统停顿。

- 由于用户输入引起异常的语句而造成的问题：

用户极有可能输入一些仅仅通过语法检查和编译不能发现问题但实际却有问题的语句。这就相当于用户在系统的设计上有 bug，例如上面所提到的数组越界、空引用操作等问题。

由于用户输入的 Java 语句片段是在系统仿真运作过程的某个已经确定的位置去调用的，因此只要在相应位置使用 Java 语言中的 catch 语句捕获可能的异常并进行处理即可。

5. OOPN-IDE 模块设计与实现

在上一章所述的逻辑结构和物理结构设计的基础上，下面依次讲述各模块的设计：

5.1. 用户界面部分

用户界面部分可分为：主窗口布局、图形编辑方式、代码自动生成、仿真运作过程几方面，下面逐一讲述。有关的可视效果见附录四。

5.1.1. 主窗口的布局

MainFrame 类实现了主窗口，其布局自上而下分别是：

主菜单：由 MainMenu 类实现。

快捷工具条：由于其上按钮与菜单项有对应关系，故将其也放到 MainMenu 中，为统一的事件处理带来了方便。

文档窗口区：用 Desktop 类实现，它提供文档窗口的增、删、检索等功能。

编辑工具条：由 ToolBarManager 类来实现。由于不同内容的编辑所需工具不同，因此设计了三种编辑工具条分别用于：简单类、复合类、模型系统。当在不同的文档窗口间进行切换时，对应的工具条应进行更新，以与要编辑的饿内容相一致。

状态条：由 InfoPane 实现，可以显示一行文本。当文本长度超过可视的范围时，应提供一定方式对可见部分进行移动，以获知完整信息。

5.1.2. 文档窗口的布局

本工具可以对多种类型的内容按多种方式进行编辑。为方便管理，特将各种文档窗口的内容，即布局与其它编辑上的属性，抽象成一个 DocPane 类。任何一个文档窗口均是在 DocPane 基础上外包一个 JInternalFrame 类实例完成的。

DocPane 中包括了对不同编辑内容与方式的支持。

当以文本方式编辑模型类、模型系统、消息类时，DocPane 中将会有一个 JTextArea 实例(文本编辑区类，属 Java 类库)，可进行文本编辑。

而当以图形方式编辑模型类时，DocPane 中间接地创建一个 OPNCanvas 类实例来显示 OOPN 模型类的图形结构，并完成编辑操作与处理。

当对模型系统进行图形编辑时，DocPane 将一分为二，左面为一树形目录结构，右面则是以 CardLayout 方式布局的一系列 OPNCanvas 类实例，树形结构中的每个节点均对应一个 OPNCanvas 实例。这样用户即可选择模型系统各个层次中的任一模型类进行编辑操作。

通过访问 DocPane，可以获知其中的编辑内容的信息，从而主窗口可以对编辑工具条进行及时刷新，及确定是否应保存其中的内容等。

5.1.3.图形编辑方式

此部分的实现在 OPNCanvas 类中。

进行图形编辑时，关键是在用户的操作与程序的执行动作之间建立对应关系。

用户在进行编辑时首先要选择一种工具，然后再进行操作，使该工具的作用反映到图形上。本工具中定义了如下的映射关系：

- 单击鼠标左键：

  Transition 工具被选中时增加一 Transition；

  State 工具被选中时增加一 State；

  MesQueue 工具被选中时增加一 MesQueue；

  Gate 工具被选中时增加一 Gate；

  InnerClass 工具被选中时增加一 InnerClass；

  Arc 工具被选中时增加一个点到已开始画的 Arc 中或开始画一个 Arc；

  AddToken 工具被选中时增加一 token 到当前点击的 State 中；

  RemToken 工具被选中时删去一 token 从当前点击的 State 中；

  Select 工具被选中时使点击的组件变成被选中状态；

  Remove 工具被选中时删除被点击的组件；

- Shift+左键单击：

  Transition 工具被选中时按与通常走向不同的方向增加一 Transition；

  MesQueue 工具被选中时按与通常走向不同的方向增加一 MesQueue ；

  Arc 工具被选中时使一正在输入的 Arc 在指定组件上结束。

  其它工具被选中时与不按 Shift 的情况相同。

- 按下左键拖动鼠标：
  Select 工具被选中时移动当前选中的组件。
  其它工具被选中时不进行任何操作。
- 单击右键：
  弹出快捷菜单，对组件和栅格的属性进行修改。

据此映射，即可在 OPNCanvas 中设计出相应处理流程。

在图形建模中，为了使输入的组件对齐，在背景上增加了栅格。当通过鼠标的点击输入组件时，新组件将自动定位到最近的栅格点上，从而达到各种组件整齐排列、连接弧横平竖直的效果。栅格的单元宽度和高度都可以由用户自定义。

为了显示 State 中的 token，采用的方法是：从 State 的中央排起，先在最中间排上一个，然后在半径为四倍于 token 半径的圆上均匀地分布若干个 token，然后再在 8 倍于 token 半径的圆上均匀分布若干个 token，直至 token 完全分布完或者在 State 中放不下全部的 token，则显示一个数字代替。

半径为 nr(r 为 token 的半径)的圆，周长为 2πnr，则近似地可以排列 2πnr/2r 个 token，假设圆周上的一半空间空余而只用一半空间来放置 token，则得到 πn/2 个 token，而因为 n=4m(m=1，2，3，…)，则得 n=4，8，12，16，…，即在半径为 4mr 得圆上可以放 2mπ 个 token。基于如上的思想，可以在 State 类中定义方法来完成 token 的显示。类似地也可完成 MesQueue 中的 token 的显示。

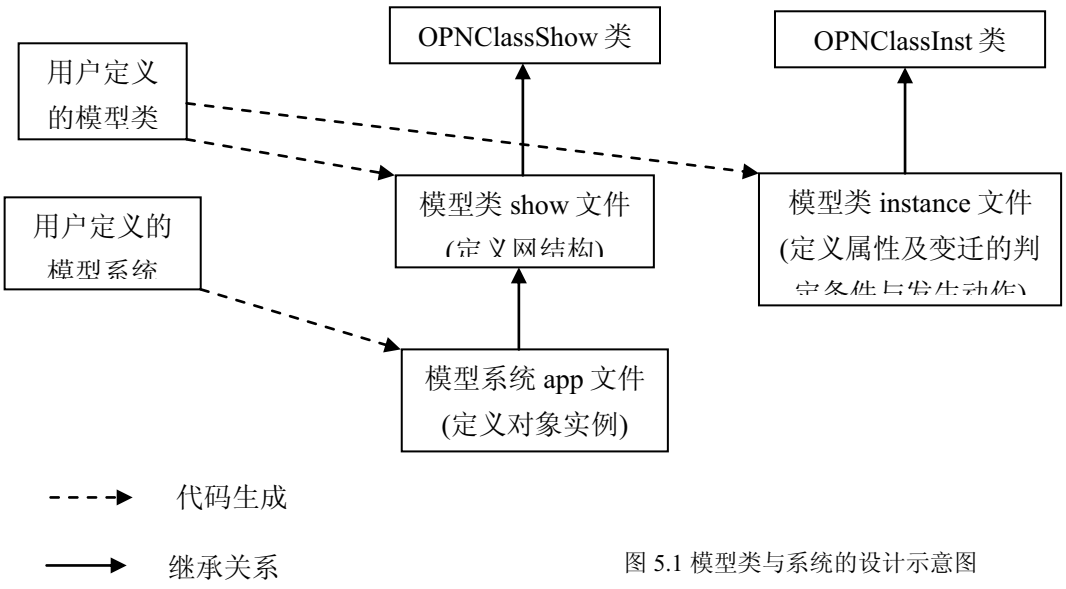


图 5.1 模型类与系统的设计示意图

5.1.4.代码的自动生成

根据构想，是将用户输入的模型转化成 Java 代码，进而编译成字节码，最后使用 Java 解释器来解释字节码从而达到目标系统运作的目的。

如图 5.1 所示为用户建立的模型类与系统同程序内部实现中使用的 Java 类之间的关系。在 4.2.5.1.1 中已阐述了分别用 OPNClassShow 和 OPNClassInst 来表达模型类的两个方面，但是这两个类只是对所有用户创建的模型类的抽象，实际的某个模型类则有其特定的网结构、属性及动作语义。为了表达之，对于每一个用户定义的模型类都生成两个文件：show 文件和 instance 文件。它们均为 Java 类，并分别继承了 OPNClassShow 类和 OPNClassInst 类。它们的公共特性通过继承来获得，而定制的信息则在自身中定义。

对应模型类的 show 文件和 instance 文件合作才构成一个完整的包括状态转换特征(即网结构)、属性、动作的具体语义等在内的 OOPN 模型类的描述。在实例化的时候，并非是一个 OOPN 模型类 show 文件实例与一个 instance 文件实例一一对应地构成一个完整意义上的实例，而是若干 instance 文件实例共享一处 show 文件实例表达的网结构。总之，可以用 instance 文件实例即 OOPN 模型中的对象实例 token 来表达实际的一个对象实例，因为它们之间有一一对应关系。

基于如上的认识再进行细节上的定义与处理，将使结构非常清晰、条理。

由 Petri 网结构获得程序代码的传统做法是将 Petri 网的结构映射成程序结构。但是很明显，在生成程序代码之后，Petri 网将再无任何影响力，从而不能继续发挥 Petri 网的优点。正是在这一步中，出现了明显的“脱钩”。

因此我们考虑在目标系统的实现程序中，仍保留 Petri 网结构，依赖于网中 Transition 的发生来进行目标系统的运作。这样用户定义的网对应的程序中只需要把表示网的数据结构建立起来即可，网的 transition 发生规则则完全可以抽象出来，定义在某个类中，而目标系统中模型类的实现类通过继承该类，即可自动地完成执行规则与方式的设置，从而进行运作。

根据驱动来源不同，网的发生规则分别在 OPNClassShow 和 OPNClassInst 两个类中进行定义。对于前者，是来判定无前驱 State 的 Transition (即源泉 Transition)或 Gate 是否可发生的，而对于后者，则相当于由对象实例主动地通过 Transition 的发生完成自己的执行或状态变换。两者中发生规则的表达均用一线程，通过线程的创建，即可去启动网的执行。

如前所述，Transition 有 preCond 与 action 两种属性，分别表示发生条件和执行的动作，且由用户用 Java 语句块来表达。在自动生成的 Java 程序框架中，通过将 Java 语

句块嵌入适当的位置，并动态地调用之，即可表达一个真正目标系统的实现，而非只是系统的框架式描述。

在 show 文件中，包括一系列的用来建立程序形式的网结构的 Java 语句，执行之，即可完成网结构的建立。

在 instance 文件中，包含有用户针对模型类定义的属性。除此，还有成对的方法定义，一对对应一个 Transition 或 Gate。其一返回值为 boolean，方法体为用户输入的该 Transition 的发生条件，另一个返回值为 void，方法体为用户输入的执行动作。

基于如上的思想，用户创建的每个模型类均自动生成两个文件，分别是 show 文件和 instance 文件，两文件均是 Java 类，类名由模型类的名字加上一定的前缀和后缀构成。

用户创建的模型系统，会自动生成一个 app 文件，该文件亦是 Java 类，且是该系统的模板所对应的 show 文件的子类。在 app 文件中，会生成一系列的语句，其执行的结果即是在模型类中加入对应的对象实例。

按如上设计，即可确定自动生成代码的过程与方法。show、instance、app 文件的示例见附录五。

5.1.5.仿真运作过程

在上节中相当篇幅的内容实际上是与仿真运作相关的。这里还需要说明的，就是一个目标系统的仿真运作是如何启动并被监控的。

要启动一个目标系统，用户必须指定系统名，本环境根据该名字即可以找到与上面的 app 文件(Java 源码文件)相对应的 Java 字节码文件，利用它，即可创建实例，从而创建一个目标系统。

而一个系统要运作，就要启动引发过程。而系统中已自含了网的发生规则，表现为 OPNClassShow 及 OPNClassInst 中定义的线程。由于一个目标系统是一个层次结构，因此稍微定义几个递归方法，即可完成自顶向下基于结构的各层上线程的依次创建和启动，从而使目标系统进行运作。

系统运作起来了只是一方面，进一步地还希望对运作过程进行监控。既然系统是一个层次结构，且 Java 语言中提供了很方便的事件听者注册机制，因此很容易将上一层的 OPNClassShow 注册为下一层及其所含的 OPNClassInst 实例的听者，而听者又可以向自己的听者继续发消息，这样的最终结果，会使网引发运作的所有事件均可被有选择地送至系统层，而系统层的听者若定义成一个专门负责监控工作的管理器，对听到的事件进行处理并返回响应，则整个目标系统的运作就被完全控制起来了。

通过这一机制，很容易实现连续、单步仿真运作及系统的运作是否有进展的判定。

5.2. 网络通讯部分

5.2.1.设计思想

在整个工具的实现中，多机协同部分是较繁杂的部分。在设计思想上，从以下两点入手是成功的关键：

1、抓住中心概念：对每个 **Client** 和进行协同的每个**文件**均设置一个线程，从而可以进行分散的集中控制。这样即避免了整个集中控制的复杂与混乱，也克服了完全分散处理的低效率。这里的整个集中控制指 Server 方只有一个控制流，由其来负责全部的处理。而完全分散是指 Client 与协同的文件的完全组合中的每一种情况都设置一线程，过多的控制流会使负载过重。

2、采用阻塞/唤醒机制：多机协同涉及大量的线程，它们之间要进行大量的交互。为提高效率，参考操作系统中的 P/V 操作，应用阻塞/唤醒机制，避免了轮讯造成的低效。

5.2.2.Client 部分

客户端主要由以下服务线程组成：

5.2.2.1. Client 线程

主要负责客户的连接与断开连接：

1、阻塞自己。 当被唤醒时，准备分析接收到的命令。

2、分析接收到的指令。有如下三种指令码：

(1) LOCAL_CONNECT，与本地的服务器创建连接。

(2) REMOTE_CONNECT，根据用户输入的 IP 地址或域名与远端的服务器创建连接。

(3) DISCONNECT，断开与服务器的连接。

3、连接建立的过程：

(1) 首先在服务器的周知口 6540 处创建 socket 连接，读取服务器传来的新端口号，断开这个连接。

(2) 在新的端口处与服务器创建 socket 连接。创建完成后，取得输入/输出流。

(3) 创建 Import、Export、Receive 等线程，并初始化它们的输入/输出流。Import、Export 和 Change 的输出流为这个 socket 的输出流。Receive 的输入流为这个 socket 的输入流。最后使 Import、Export、Receive 等线程开始运行。

4、断开连接的过程：

(1) 关闭所有在服务器上打开的文件。

(2) 断开 socket 连接。

5.2.2.2. Import 线程

主要负责打开服务器上的文件这一操作：

1、阻塞自己。 当被唤醒时，准备打开文件。

2、打开文件的操作：

(1)首先向服务器发送 import 命令，准备接收文件列表。

(2)接收到文件列表后，由用户选择一个要打开的文件。

(3)向服务器发送 openfile 命令，等待服务器的返回。

(4)接收到服务器返回的 OPNClassShow 类实例，根据它生成 DocPane 实例。

3、操作完成后，阻塞自己。

5.2.2.3. Export 线程

主要负责保存文件、关闭文件这些操作：

1、阻塞自己。 当被唤醒时去判断指令类型。

2、保存文件的操作：

(1)若为 export，则当前文件信息可由当前编辑窗口得到。

(2)若为 exportas，还要由用户输入新文件名。

(3)将指令类型和可能涉及的新旧文件名发给服务器，等待服务器的回答。

(4)若服务器返回 saveok，表示文件正确保存；若为 saveerror，则文件保存出错；若服务器在 30 秒内没有返回，则认为连接或传输出错，给出提示信息。

3、关闭文件的操作，若本客户为该文件的最后一个客户，会被询问是否保存该文件，用户回答后进行相应处理，关闭该文件。

5.2.2.4. Change 线程

负责用户对文件的修改。每次的改动都会生成一个线程来执行：

1、根据用户的各种操作的指令名和组件对象的名字生成关键字，在 Receive 的 change_list 变量中登记，值为本 Change 线程的引用。

2、将用户要添加、移动、删除的组件对象和指令名传递给服务器，等待返回。

3、变量 ok 初始值为 null， 在 30 秒内，如果 Receive 线程接收到服务器的返回信息，会将 ok 置为“ok”。Change 线程每隔 1 秒判断一次 ok 的值，若不为“ok”则转向睡眠。 过了 30 秒，则认为出错，操作失败，给用户提示信息。

5.2.2.5. Receive 线程

主要负责接收服务器的返回信息，并根据指令类型, 传递给其它线程处理或自行处理：

1、若是接收到与文件打开有关的指令，则传递给 Import 线程解决。

2、若是接收到与文件存储和关闭有关的指令，则传递给 Export 线程解决。

3、其余的指令为文件被修改后，服务器的返回信息。

(1)先判断该指令是否是由本用户发出的，根据接收到的指令类型和部件名形成关键字在 change_list 中查找，若能找到，则取出该关键字对应的 Change 线程，将其中的变量 ok 置为“ok”。

(2)根据得到的指令执行相应的操作。

5.2.3.Server 部分

服务器端主要有以下服务线程：

5.2.3.1. Server 线程

在周知口 6540 处提供连接服务：

1、 监听 6540 端口(在没有客户连接时，该线程为阻塞状态)。

2、 有客户连接时被激活。

3、 为该客户分配一端口号，创建对应的 ClientServer 线程建立 socket 连接。

4、 转 1。

数据结构：

1、Hashtable clientList： 连接的客户表。 KEY： 端口号，VALUE： 端口对应的 ClientServer 线程。

2、Hashtable fileList： 打开的文件列表。 KEY： 文件名，VALUE： 该文件对应的 FileServer 线程。

5.2.3.2. ClientServer 线程

对应一个客户连接，接收客户的指令，并根据指令，做相应的工作：

1、取得输出流，为该客户分配一个端口(大于 5000)。

2、将端口号通过输出流告知客户，断开当前连接。

3、在分配的端口处建立一个 ServerSocket(Java 类库中的类)，监听该端口。

4、当客户连接时，取得输入/输出流。

5、将自身阻塞等待客户的指令。

6、被激活后，分析指令，有如下情况：

(1) import，找到所有的可编辑模型类文件，将文件名列表返回。

(2) openfile，根据文件名在 Server 线程的 fileList 中查找该文件是否已经打开。若该文件尚未打开，则为该文件新建一个下述的 FileServer 线程，否则取出该 FileServer 实例引用。调用此 FileServer 实例的 addClient 方法，将本 ClientServer 线程加入到其 clientList 中。将此 FileServer 线程加入到本线程的 fileList 中。

(3) 其他类型的指令，必为图形编辑指令。 根据文件名，在 fileList 中查找出对应的 FileServer 线程。然后判断该线程是否忙，忙(标志是该线程的 cmd 变量非空)则等待(每 0.5 秒判断一次)。 不忙时将该指令及 this 指针传给它。

如果在 fileList 中没有找到相应的 FileServer 线程，则指令错误，返回错误信息。

7、若该文件线程忙超过 30 秒，则认为该线程已经陷入了死循环，调用该 FileServer 线程的 beDestroied 方法，将其关闭，并通知连接到该文件上的客户，将该文件先关闭，再重新打开。

数据结构：

1、Hashtable fileList： 记录客户打开的文件。

2、Socket 类型的变量和一些流： 用于网络通信。

对外接口：

1、send(Hashtable cmd)： 将参数 cmd 传送给客户。 该方法主要由 FileServer 调用。

2、shutdown()： 若服务器关闭，则由 Server 类调用该方法。

5.2.3.3. FileServer 线程

创建时传入参数为一文件名，打开相应的文件，接收从客户传来的对图形结构进行修改的指令，做相应处理：

1、打开文件。 将自身阻塞等待指令。

2、被激活后，分析指令，有如下的指令类型 (cmd 为指令，client 为发出指令的客户对应的 ClientServer 实例)：

(1) remove，若发指令者不是最后一个客户，直接将 client 从 clientList 中移出。若该客户为最后一个客户，则询问用户是否保存当前文件，在该用户回答了保存或否后，执行相应操作，并将该 client 从 clientList 中移出，接着关闭该文件。

(2) export，客户发出的存盘指令。 若保存成功则调用 client.send，发出成功信息,否则发出错误信息。

(3) state、gate、transition、innershow、mesqueue、arc 等，先调用 improve 方法对 cmd 处理。 在 cmd 中取出对应的组件，(对 arc，还要根据取出的前驱,后继的名字找到前驱和后继)，添加到 show 中。若添加成功，调用 sendAll(cmd)将信息传给编辑本文件的所有客户；否则，将错误信息传给 client 对应的客户。

(4) move，取出组件名，新位置信息。 根据组件名找到该组件，若该组件存在且移动成功，调用 sendAll(cmd)将信息传给所有相关客户，否则，将错误信息传给 client 对应的客户。

(5) delete，取出组件名， 根据组件名找到该组件，若该组件存在且删除成功，调用 sendAll(cmd)将信息传给所有相关客户，否则，将错误信息传给 client 对应的客户。

(6) lock 或 unlock，取出组件名，根据组件名找到该组件，将该组件的 canBeChanged 分别置为 false 或 true，表示该部件是否可修改。

3、处理完毕后，阻塞本线程，转 2。

4、数据一致性的保证： 文件服务器每次只处理一个指令。在处理该指令时，由于 cmd 不为 null，则其它需要本线程服务的 ClientServer 线程要等待(每隔 0.5 秒后查询一次 cmd 是否为 null)。在处理完成后，将 cmd 置为 null，其它线程就有了得到服务的机会。

数据结构：

1、Hashtable clientList： 记录协同编辑这个文件的所有客户。

2、Hashtable cmd：存放指令。

3、ClientServer client：为这个线程的引用。

4、Monitor monitor： 本线程调用 monitor 的 suspend 与 resume 方法，将自己阻塞或唤醒。

5、DocPane pane 和 OPNClassShow show： 打开的文件的数据结构。

对外接口：

1、beDestoried()： 当本线程陷入死循环时，由 ClientServer 线程调用，进行的处理为：

(1) 判断标志 toBeDestroied 是否为真，若否，则说明该方法已经由另外一个 ClientServer 线程调用，退出。

(2) 若 toBeDestroied 为真，先储存该文件，根据文件的客户列表找到相关的所有客户，给它们发消息。 客户端接到消息后，将打开的这个文件关闭，并通知用户重新打开该文件。

2、 shutdown()： 当服务器要关闭的时候，调用该方法关闭该文件。

5.3. 静态死锁分析

基于 3.3.3 节所述的死锁分析的思想，用如下的设计来完成死锁检测的功能：

基于 4.1.6 节的数据结构，设计了一个 ReachableTreeNode 类，表示可达树节点，而通过一个根节点即可访问整个可达树，从而为进行死锁分析提供基础。

当对某个模型系统进行死锁分析时，首先针对 OPNClassShow 生成对应的 SimpleClassShow 实例，其中的网结构也递归地生成。

在 SimpleClassShow 类中定义了 containsDeadlock 方法，用来检测其中是否含有死锁，其具体的过程在上一节中已清楚地进行了阐述。

为了支持该过程设计了如下的方法：

SimpleClassShow 中的 getIENet 用来获得当前对象的接口等价网。

SimpleClassShow 中的 IENetPart 内部类用来表示与一个 Transition 相关的接口等价网部分，通过将多个相互连接的 Transition 对应的接口等价网片段连接起来，完成整个接口等价网的生成。

SimplePetriNet 中的 containsDeadlock 用来检测在一个经典的 Petri 网中是否存在死锁，其算法严格按照基于可达树的死锁检测算法来进行。此方法用在对接口等价网进行死锁检测。

SimplePetriNet 中的 getReachableTree 方法用来生成当前 Petri 网对应的可达树，返回该树的根节点。

ReachableTreeNode 中的 containsDeadlock 用来检测以当前可达节点为根的可达树中是否含有死锁。

5.4. 动态死锁分析

动态死锁分析工具的实现基于如下认识：

在整个目标系统运作未结束时，若经过一定时间，仍未有进展发生，则认为发生了死锁。

在这句话中，有几点需要说明：

1、运作结束：系统运作的结束是要有判定条件的。这里把系统中的对象实例分为三种：活动实例、非活动实例、死实例。

- 活动实例指该实例所在的 State 有后继 Transition，即还有流动的可能。
- 非活动实例指该实例所在的 State 无后继 Transition，即已无法继续流动。
- 死实例指那些曾经在系统中出现但现在已销毁的实例。

我们认为，只要系统中尚有活动实例，则系统还未结束运作；否则认为已结束。

为了判定结束，利用消息传递来进行信息的收集。对于每一个实例来说，当其由活动实例变为非活动或销毁时，发送一特定消息给其所属的类(必为简单类)。系统中的每一个类在接收到该种消息时，均去判定在它所辖范围内是否还有活动实例，若已无活动实例，则去发送一同样的消息通知其包容类(必为复合类)。复合类在已接收到它的所有内部类发来的此种消息后，即向它所属的类发同样的消息。最终这些信息会汇集到目标系统的顶层。在顶层类已接收到它的所有内部类的消息后，系统运作即告结束。

2、进展：系统运作的进展体现在 Transition 的发生上，有 Transition 发生，则认为取的了进展，否则认为没有进展。

而 Transition 发生的信息也是采用类似于上述的事件传递的方式由局部传递到顶层的。

3、判定时间段：该时间可由用户定义。由于软件所运行的环境的变化，整个运作的过程的速度也会发生大的变化。在运作较慢时，应将该时间段值设得较大，否则应较小。

在认为运作过程已发生了死锁时，系统将以对话框的形式提示用户。

结束语

现在对全文作一总结。

1．完成的工作

在对相关领域或方向——Petri 网、面向对象、并行计算等进行细致研究和认真分析的基础上，本文提出**基于面向对象 Petri 网支持并发软件的开发**。

本课题的**研究成果**包括：

1、在纵观了 Petri 网模型的发展的基础上，选择了一种面向对象 Petri 网模型——OPNets 为蓝本，进行取舍和扩充，提出了 OOPN 模型。

2、完成了如下工具：

- OOPN 模型图形和文本建模工具。
- 多机协同建立/编辑系统模型。
- 并发系统仿真和运作工具。
- 死锁检测工具。

上述四项功能已被集成在统一的交互式用户界面上。

OOPN-IDE 集成开发工具具有如下优点：

- **跨平台**

本工具基于 Java 语言开发，可以跨平台使用。OOPN-IDE 已顺利通过项目测试专家组在 WindowsNT/95/98 及 Solaris2.x 异构网络环境下的跨平台运行测试。

利用本工具所建立的模型经代码生成后得到的源程序也是基于 Java 语言的，从而亦可直接跨平台使用。

- **以面向对象 Petri 网为基础模型**

本工具所基于的 OOPN 模型是在相关文献中可见到的最好的面向对象 Petri 网模型。

OOPN 模型中封装和抽象的表示简洁明了。基于类之间的 part-of (复合) 关系，可以实现方便的重用，并可以构造无限层次的复合类。

- **界面友好，编辑方便**

应用本工具可以方便地进行建模、仿真和系统运作。

集成环境中提供了各种工具按钮可用于进行各种文件操作、编辑操作和运作控制。

- **多机协同**

本工具的设计遵循 Client/Server 体系结构。多个 Client 可以同时连接到一个 Server 上对同一个系统进行建模。为设计大型系统提供了基础。

- **Petri 网支持并发系统建模、仿真、目标系统实际运做全过程**

应用本工具不仅可以进行常规的系统建模和仿真，还可以进行系统的实际运做。不同于传统上将网结构转化成程序控制结构的做法，本工具生成的并发程序中仍然保留有网结构。实际上并发程序的执行即是基于网中 transition 的不断发生而逐步推进的。这样做有两个好处：

1、可以直观地通过 Petri 网的运作获知并发程序系统的执行情况。

2、可以以 Petri 网的观点和角度对系统进行即时控制。

- **在一定条件下可进行避免状态爆炸问题的死锁分析**

本工具所基于的 OOPN 模型，在一定的条件限制下，可以仅用类的外部通讯接口代替该类参加其所在的包容类的死锁检测。这样即可按从低到高的次序进行逐层的死锁检测，从而可在一定程度上避免通常应用可达树对大系统进行分析而造成的状态爆炸问题。

2．进一步的工作

OOPN-IDE 工具是一个集成开发环境，其中包括多个子工具。但是 OOPN-IDE 仅仅是为进一步的工具开发进行的初步探索，可以说奠定了一个基础，与已有的各种专门 Petri 网工具相比在细节上，还有许多需要细化的地方：如支持模型仿真的双向进行、性能分析的结果以正交形式或图形形式(曲线图，柱状图，统计表)给出等。

这里撇开小细节不谈，从宏观上讲，进一步的工作可分为以下三部分：

1、开放性：基于构件技术，使 Petri 网与非 Petri 网部分无缝连接。

本质上，Petri 网擅长于表达各种任务、操作与服务，而对其它方面则不易实现。为了能真正描述实际应用，应使 OOPN 模型与非 Petri 网部分相互协作完成系统的构造

实现。要协作，则需要在两部分之间定义简洁、方便的接口。该接口的实现可有如下考虑：

- 要使 OOPN 对象能被外界所访问，可以在相应 OOPN 网的底层实现 Java 类中提供若干方法用来支持对该 OOPN 网的 mesQueue 中的消息的存取。

进一步地，为了避免外界以“忙等待”的方式访问 OOPN 对象，可将网中某 OOPN 的消息的变化作为一种事物的发生看待，在事先进行事件监听注册的情况下，可以把事件送到收听者来触发一定的动作，即利用事件的驱动来传递信息。

- 非 OOPN 部分也可以进行一定的封装来实现与 OOPN 对象外部完全相同的感观，如有几个可见的 mesQueue，可以进行消息的读写，有图形表示(只有外框与 mesQueue 的图形表示，内部细节是不可见的，实际也是不存在的)。这样一来就可以与对 OOPN 对象相同的方式处理和对待这种“伪”OOPN 对象。
- 现在软构件是重要的软件发展方向，是非常有希望的。应把 OOPN 对象设计成构件的形式，以便于进行系统插装式的构造。考虑以 Java Beans 为标准来封装各个 OOPN 对象，而前述的事件驱动的思路即参考了 Java Beans 中的消息模型。

2、伸缩性：可应用于单机的目标系统也可应用于多机，即支持分布式环境。

应用 Petri 网的原因，是由于它在表达并发上有优势。而分布是与并发紧密相连的，既然把面向对象应用到 Petri 网上形成了一个个的对象，自然也就要考虑如何进行对象的分布与相互通信。

现在有的几种分布式对象技术有 CORBA 与 DCOM，新出现的有 Java 的 RMI。由于 RMI 简洁易用，可考虑用 RMI 作为 OOPN 模型系统进行分布式处理的基础来构筑整个 OOPN 模型分布式体系。这一体系的内容包括：

- 设计好的系统如何进行不同机器的分布，即将系统中的对象分散到各个不同机器上。
- 分布到各处的对象何时执行、如何启动及由谁启动。
- 分布的对象如何实现相互的定位，从而进一步访问。

这些内容都尚未仔细考虑，只是对系统的分布与执行有如下想法：

分布式软件可能是动态分布的或是静态分布的。所谓动态是指完成系统功能的相互通信的对象不相关地分别在不同的机器上启动，对此应提供某种机制能简洁方便地建立它们的相互连接。所谓静态是指集中地控制一个 OOPN 类的实例的各个并发部分去分布地并发执行，可以指定某一部分到某一机器上运行，自然相应的机器上要有底层支持。

3、友好性：在 Petri 网上增加一面向应用的图符语言层，使 Petri 网结构透明化。

尽管 Petri 网在表示上简单、直观，但对于大多数人来说仍认为是形式化的东西，视其为畏途。用贴近应用的各种图符作为最外层，内部用 Petri 网实现，则可达到外强而且中实的效果。表现在两个方面：编辑时，用图符语言来面向用户，而系统自动地将其转化成 Petri 网实现；在系统运作时，Petri 网中发生的事件，则解释成图符表示层的某种变化。

基于上述三个方向，不妨进行如下的展望：

若实现了伸缩性的要求，一个大的 OOPN 模型即分布到一个广大的网络环境中。各个部分之间通过 Petri 网的拓扑连接进行信息传递。而在每一部分处，如果实现了开放性，即可有非 Petri 网部分参与，从而构成一个以 Petri 网结构为骨架，以非 Petri 网部分为内容的整体，而这个整体在上述友好性实现的支持下即可将整体内发生的变化以非常友好的方式提供给用户。从这个意义上讲，Petri 网成为了一个使网络透明的实体，这实际上是一种新的“中间件”。

众所周知，中间件现已风靡世界。Petri 网与现有的中间件相比，其图形化的表示与可进行数学分析将是突出的优势。

可以预见，如果完成了上面的三项工作，一个基于 OOPN 模型和 Java 的集成化的工具集将可用来控制，大到全国范围内的现代化战争的仿真模拟，小至一个钟表来定时去控制一个洗衣机的开启 (基于 Sun 公司推出的使网络无处不在的 Jini 技术将使我们不久就会看到类似的一切。实际上，就在本文的写作过程中，Microsoft 即推出一种称为“家庭网”的演示产品。) 。

因此可以说，基于本文的基础，继续进行“基于 OOPN 模型的并发软件或系统的开发方法研究”意义重大，它将会在国民经济建设中发挥巨大作用。

附录一 OOPN 软件包构成

■ 初始化部分与用户界面部分

opn.ide: 系统初始化部分与用户界面构件包

ArcPropertyPane
ClassShowPropertyPane
ComPropertyPane
Desktop
DocPane
FileChooser
FileManager
GlobalRefs
GridOptionsDialog
HtmlHelpPane
IdeEngine
InfoPane
Main
MainFrame
MainMenu
MqPropertyPane
MyCheckBoxMenuItem
MyInternalFrame
OPNCanvas
OPNGraph
OPNText
PictureWindow
SimFrame
StatePropertyPane
TextChangeListener
ToolBarManager
TranPropertyPane

opn.ide.lookandfeel： 用户界面风格包

AquaMetalTheme
BigContrastMetalTheme
ContrastMetalTheme
DemoMetalTheme
GreenMetalTheme
KhakiMetalTheme
MetalThemeMenu
PropertiesMetalTheme
UIStyleManager

opn.ide.options： 用户界面配置包

CreateDOptionsListener
DefaultOptions
DuplicateOptionNameException
GraphOptions
OptionsColl
OptionsEventEngine
OptionsMngr
OptionsSuper

OptionsSuperI
XyFrameOptions

opn.ide.event： **用户界面事件包**

IdeEvensList
IdeEvensListColl
IdeEventListenerAC
IdeEventListenerI
IdeEventsListItem
IdeSuperEvent
IdeSuperEventI

opn.ide.tools： **用户界面例程包**

ArgumentException
ButtonPane
ColorSelectPane
FaceControl
JIntegerSliderLabel
JIntegerSliderTextField
JIntegerTextField
MultiItemChooser
MultiOptionsSelectPane
NameLocSelectPane
NameSelectPane
SizeSelectPane
TimeSelectPane

■ 网络通讯 Client 部分

opn.net.tools： **网络通讯例程包**

MyObjectPipe

opn.net.client： **Client 端处理包**

Change
Client
Export
Import
Receive

■ 网络通讯 Server 部分

opn.net.server： **Server 端处理包**

ClientServer
FileServer
Server

■ OOPN 模型支持类部分

opn.baseclass.petrinet： OOPN 模型死锁检测包

NetAnalysisException
ReachableTreeNode
SimpleClassShow
SimpleMesQueue
SimpleNetCom
SimpleNetNode
SimplePetriNet
SimpleState
SimpleTransition

opn.baseclass： OOPN 模型支持包

Arc
ClassChoice
ComGlobals
DiOPNNode
FileFormatException
Gate
GraphTopologyException
Lockable
MesQueue
Message
MessageChoice
OPNClassInst
OPNClassShow
OPNComponent
OPNInput
OPNNode
OPNOutput
RemoteClassChoice
State
Token
Transition

opn.baseclass.event： OOPN 模型仿真运作事件包

ClassThreadAdapter
ClassThreadEvent
ClassThreadListener

opn.baseclass.tools： OOPN 模型支持例程包

ComPropertyCollI
DefaultComPropertyColl

■ 系统环境配置部分

java.lang： 调试及信息显示例程包

AssertException
Debug
MS

opn.tools： 专用例程包

ColorName
Delay
ExampleFileFilter
ExampleFileView
FileNameFilter
JavaCompiler
Monitor
MyGraphics
MyMonitor
StringList
Strings
StringsListener
VisCompile

opn.env： 系统环境配置包

Globals
InfoShowerI
MyClassLoader
ResMan
Utilities

附录二 模型类与系统的正文保存形式脚本定义

■ 简单类的脚本说明

简单类的基本框架定义为：

< Name: … >
[Type: Simple]
[Color: …]
[Pos: …]

< MesQueue: >
…
< #MesQueue >

< Transition: >
…
< #Transition >

< State: >
…
< #State >

< Arc: >
…
< #Arc >

[Attr:]
…
[#Attr]

[Init]
…
[#Init]

<##>

在上述定义中，带有“<>”的项为必选项，带有“[ ]”的项为任选项，符号“…”表示具体的定义。在后面的文本定义中，这些符号也具有同样的含义。下面将详细介绍简单类的具体定义。

- 类的属性定义

简单类的属性定义包括上述定义中的“Name”、“Type”、“Color”、“Pos”、“Attr”和“#Attr”、“Init”和“#Init”，如下详述：

“Name:”：简单类的名字标志符，为必选项，用来定义简单类的名字，并规定名字的第一个字符为英文字母，后跟英文字母、数字或下划线。各组件名字的定义都有同样的规定。

“Type:”：类型标志符，为任选项。类型的定义可取三种值：“Simple”、“Compound”、“Application”，用来表示所定义的是简单类、复合类还是模型系统。缺省定义为“Simple”，所以在简单类的定义中可省略该项。

“Color:”：颜色标志符，为任选项，用来定义简单类外框的颜色。颜色的标准定义为：在“R”、“G”、“B”字母后分别跟 0~255 之间的数字，用来表示各 R、G、B 分量组合成的颜色。另外，也可以用下述十三个英文单词来定义颜色：“black”、“blue”、“cyan”、“darkGray”、“gray”、“green”、“lightGray”、“magenta”、“orange”、“pink”、“red”、“white”和“yellow”。缺省值为“yellow”。

“Pos:”：位置标志符，为任选项，用来定义简单类外框的位置及大小。后面可跟六个用逗号隔开的数字，其中前两个表示外框左上角的位置，中间两个表示外框的宽度和高度，最后两个表示外框角弧的宽度和高度。缺省值为“50，50，1000，1000，20，20”。

“Attr:”和“#Attr”：属性变量定义标志符，为任选项，用来定义简单类的属性变量描述。“Attr:”表示属性变量定义的开始，“#Attr”表示属性变量定义的结束，在这两个标志符之间允许以 Java 语句形式为简单类定义任意的属性变量。若不想为简单类定义属性变量，则可以省略此项内容。

“Init:”和“#Init”：属性变量初始化标志符，为任选项，用来初始化简单类的属性变量。“Attr:”表示属性变量初始化的开始，“#Attr”表示属性变量初始化的结束，在这两个标志符之间允许以 Java 语句初始化所定义的属性变量，若不想为简单类初始化属性变量，则可以省略此项内容。

- MesQueue 的定义

“Message:”和“#Message”分别表示 MesQueue 定义的开始和结束，在它们之间可以定义任意多个 MesQueue，也可以不定义，但是不能省略这两个标志符。单个 MesQueue 的基本框架定义为：

< …: >
< Pos: … >
[Color: …]
[NameLoc: …]
[Type: …]

具体含义如下：

“ …: ”：表示 MesQueue 的名字，为必选项。注意：名字后面必须跟冒号。

“Pos:”：位置标志符，为必选项，用来定义 MesQueue 的位置和大小。后面可跟四个用逗号分开的数字，其中前两个表示 MesQueue 的位置，后两个分别表示消息队列 X、Y 方向上的半径，并且前两个必须定义，后两个可以省略，缺省值为 30，15。

“Color:”：颜色标志符，为任选项。标准定义同上，缺省值为“red”。

“NameLoc:”：名字位置标志符，为任选项，用来定义 MesQueue 名字的位置。位置的可能定义有九种：“N”、“EN”、“E”、“ES”、“S”、“WS”、“W”、“WN”和“C”，分别表示名字位于 MesQueue 标志符的上方、右上方、右方、右下方、下方、左下方、左方、左上方和中心，缺省值为“N”。

“Type:”：类型标志符，为任选项，用来定义消息队列所能接受的消息类型。所选的消息类型必须是已经定义的消息类，缺省值为“Message”。

若想定义多个 MesQueue，只须按照上述基本框架重复定义即可。

- Transition 的定义

“Transition:”和“#Transition”分别表示 Transition 定义的开始和结束，在它们之间可以定义任意多个 Transition，也可以不定义，但是不能省略这两个标志符。单个 Transition 的基本框架定义为：

< …: >
< Pos: … >
[Color: …]
[NameLoc: …]
[Time: …]

[PreCond:]
…
[#PreCond]

[Action:]
…
[#Action]

具体含义如下：

“…: ”：表示 Transition 的名字，为必选项。注意：名字后面必须跟冒号。

“Pos:”：位置标志符，为必选项，用来定义 Transition 的位置和大小。后面可跟四个用逗号分开的数字，其中前两个表示 Transition 的位置，后两个分别表示 Transition 在 X、Y 方向上长度的一半，并且前两个必须定义，后两个可以省略，缺省值为 30，15。

“Color:”：颜色标志符，为任选项。标准定义同上，缺省值为“blue”。

“NameLoc:”：名字位置标志符，为任选项，用来定义 Transition 名字的位置。详细定义同上。

“Time:”：时间标志符，为任选项，用来定义 Transition 发生的持续时间。后跟用逗号隔开的数字和时间单位，数字范围为 0~10000，时间单位有七种：“MilliSecond”、“Second”、“Minute”、“Hour”、“Day”、“Month”和“Year”。缺省值为“0，MilliSecond”。

“PreCond:”和“#PreCond”：发生条件定义标志符，为任选项，分别表示发生条件定义的开始和结束。这两个标志符之间可以定义一个合法的返回值为“boolean”的方法体，若不想为 Transition 定义发生条件，则可以省略此项内容。

“Action:”和“# Action”：动作定义标志符，为任选项，分别表示动作定义的开始和结束。这两个标志符之间可以定义一个合法的返回值为“void”的方法体，若不想为 Transition 定义动作，则可以省略此项内容。

若想定义多个，只须按照上述基本框架重复定义即可。

- State 的定义

“State:”和“# State”分别表示 State 定义的开始和结束，在它们之间可以定义任意多个 State，也可以不定义，但是不能省略这两个标志符。单个 State 的基本框架定义为：

< …: >
< Pos: … >
[Color: …]
[NameLoc: …]

具体含义如下：

“ …: ”：表示 State 的名字，为必选项。注意：名字后面必须跟冒号。

“Pos:”：State 的位置标志符，为必选项，用来定义 State 的位置和大小。后面可跟三个用逗号分开的数字，其中前两个表示 State 的位置，最后一个表示 State 半径，并且前两个必须定义，最后一个可以省略，缺省值为 30。

“Color:”：颜色标志符，为任选项。标准定义同上，缺省值为“red”。

“NameLoc:”：名字位置标志符，为任选项，用来定义 State 名字的位置。详细定义同上。

若想定义多个 State，只需按照上述基本框架重复定义即可。

- Arc 的定义

“Arc:”和“# Arc”分别表示 Arc 定义的开始和结束，在它们之间可以定义任意多个 Arc，也可以不定义，但是不能省略这两个标志符。单个 Arc 的基本框架定义为：

< 连接关系及类型 >
< Pos: … >
[Color: …]

具体含义如下：

“ 连接关系及类型 ”：为必选项，用来定义 Arc 的连接关系和类型。格式为“(源组件，目的组件)：Arc 的类型，线条的类型”，其中源组件和目的组件分别表示该 Arc 连接的源组件和目的组件；Arc 的类型有两种：“Common”和“Inhabit”，分别表示普通 Arc 和抑制 Arc；线条的类型也有两种：“Line”和“Curve”，分别表示直线和曲线。后两项可缺省，缺省值为“Common，Line”。

“Pos:”：位置定义标志符，为必选项，用来定义 Arc 的位置。后面至少跟四个用逗号隔开的数字，前两个表示该 Arc 与源组件的连接点坐标，最后两个表示与目的组件的连接点坐标。对于“Line”类型的 Arc，中间可以有任意多对数字，表示中间转折点的坐标；对于“Curve”类型的弧，中间最多只能有两对数字，因为在曲线中间最多只允许有两个转折点，作控制点用。

“Color:”：颜色标志符，为任选项。标准定义同上，缺省值为“black”。

若想定义多个弧，只须按照上述基本框架重复定义即可。

- 结束定义

简单类定义的最后一个标志符为“##”，表示简单类定义的结束，在该标志符后面的所有文本都将被忽略。

■ 复合类的脚本说明

复合类的基本框架定义为：

```
< Name: … >
< Type: Compound >
[ Color: … ]
[ Pos: …]

< MesQueue: >
 …
< #MesQueue >

< Gate: >
  …
< #Gate >

< InnerClass: >
              …
< #InnerClass >

< Arc: >
  …
< #Arc >

[ Attr: ]
  …
[ #Attr ]

[ Init ]
 …
```

[#Init]

<##>

下面将详细介绍复合类的具体定义。

- 类的属性定义

复合类的属性定义包括上述定义中的“Name”、“Type”、“Color”、“Pos”、“Attr”和“#Attr”、“Init”和“#Init”。其详细定义与简单类的属性定义相似，但“Type”的定义有所不同，在这里该项为必选项，必须将其定义为“Type: Compound”。其它各项的详细定义见“简单类的脚本说明”一节。

- MesQueue 的定义

MesQueue 的定义与简单类的完全相同，详见“简单类的脚本说明”一节。

- Gate 的定义

“Gate:”和“#Gate”分别表示 Gate 定义的开始和结束，在它们之间可以定义任意多个 Gate，也可以不定义，但是不能省略这两个标志符。单个 Gate 的定义与单个 Transition 的定义完全相同，详见“简单类的脚本说明”一节。

- InnerClass 的定义

“InnerClass:”和“# InnerClass”分别表示 InnerClass 定义的开始和结束，在它们之间可以定义任意多个 InnerClass，也可以不定义，但是不能省略这两个标志符。单个 InnerClass 的基本框架定义为：

< … : … >
< Pos: … >
[Color: …]

< MesQueue: >
…
< #MesQueue >

具体含义如下：

“…:…”：为必选项，冒号之前填写 InnerClass 在该复合类中的名字，冒号之后为被引用的类的原始名字。

“Pos:”：InnerClass 的位置标志符，为必选项，用来定义 InnerClass 的位置和大小。其具体定义与简单类的“Pos:”完全相同。

“Color:”：颜色标志符，为任选项。标准定义同上，缺省值为“yellow”。

“MesQueue:”和“#MesQueue”：MesQueue 定义标志符，为必选项，用来定义 InnerClass 所带的 MesQueue。这两个标志符分别表示 MesQueue 定义的起始和结束，中间可以定义任意多个 MesQueue。由于在被引用的类中已经定义了 MesQueue 的详细信息，所以在这里只定义 MesQueue 的名字、位置及颜色，如下所示：

“…:”：定义 MesQueue 在该复合类中被引用的名字，为必选项。

“Pos:”：定义 MesQueue 的位置，为必选项。

“Color:”：定义 MesQueue 的颜色，为任选项。

在这里定义的 MesQueue 要与被引用类中的 MesQueue 一一对应。

若想定义多个 InnerClass，只须按照上述基本框架重复定义即可。

- Arc 的定义

Arc 的定义与简单类中 Arc 的定义十分相似。但是，对于 Arc 所连接的 MesQueue，除了要标明其名字外，还要指出它所属的 InnerClass 的名字，定义格式为：InnerClass 的名字.MesQueue 的名字。具体定义参见“简单类的脚本说明”一节。

- 结束定义

复合类定义的最后一个标志符为“##”，表示复合类定义的结束，在该标志符后面的所有文本都将被忽略。

■ 模型系统的脚本说明

模型系统的基本框架定义为：

< Name: … >
< Type: Application >
< InstanceOf: … >
[Pos: …]

< Instance: >
…
< #Instance >

<##>

具体定义详述如下：

- 系统属性定义

模型系统的属性定义包括上述定义中的“Name”、“Type”、“InstanceOf”和“Pos”。“Name”和“Pos”的定义与简单类的完全相同，详见“简单类脚本定义”一节。“Type”项为必选项，且必须为“Application”。“InstanceOf”项也是必选项，后跟已经创建的模型类的名字，指示该模型系统的模板类。

- 实例的定义

“Instance:”和“#Instance”分别代表实例定义的起始和结束标志符，为必选项。本环境要求同一个 State 内的实例必须定义在一起，在这两个标志符之间可以为任意多个 State 定义实例，也可以不定义，但是标志符不能省略。State 中实例的基本定义框架为：

< InnerClass 的名字.State 的名字: >
< Token: >
< 实例的名字: >
< Init: >
…
< #Init >
< #Token >

开始应表明 InnerClass 和 State 的名字，然后在起始标志符“Token:”和结束标志符“#Token”之间为该 State 定义实例，并且实例的个数不限，但在上述定义框架中只给出了一个实例的定义。对于实例的定义，开始应给出实例的名字，然后在标志符“Init:”和“#Init”之间初始化属性值。

■ 脚本示例

由于篇幅所限，这里只列出一个简单类的脚本文件：

Name:buffer

Type:Simple

Color:orange

Pos:50,50,359,424,20,20

MesQueue:

mesqueue5:
Pos:409,100
Color:red
Type:Message

mesqueue4:
Pos:409,240
Color:red
Type:Message

mesqueue3:
Pos:409,340
Color:red
Type:Message

mesqueue2:
Pos:50,340
Color:red
Type:Message

mesqueue1:
Pos:50,180
Color:red

Type:Message

mesqueue0:
Pos:50,120
Color:red
Type:Message

#MesQueue

Transition:

transition4:
Pos:120,300
Color:R0G0B255

transition3:
Pos:320,160
Color:R0G0B255

transition2:
Pos:320,300
Color:R0G0B255

transition1:
Pos:120,160
Color:R0G0B255

#Transition

State:

state3:
Pos:240,380
Color:red

state2:
Pos:320,240
Color:red

state1:
Pos:120,240
Color:red

state0:
Pos:240,100
Color:red

#State

Arc:

(transition2,mesqueue4):Common, Line
Pos:334,290,391,252
Color:black
(mesqueue2,transition4):Common, Line
Pos:69,328,101,310
Color:black
(state1,transition4):Common, Line
Pos:120,270,120,290
Color:black
(transition2,state2):Common, Line
Pos:320,290,320,270
Color:black
(mesqueue5,transition3):Common, Line
Pos:391,112,334,150
Color:black
(transition1,state1):Common, Line
Pos:120,170,120,210
Color:black
(state0,transition1):Common, Line
Pos:213,113,139,150
Color:black
(state3,transition2):Common, Line
Pos:261,358,309,310
Color:black
(transition3,state0):Common, Line

Pos:306,150,263,118
Color:black
(mesqueue0,transition1):Common, Line
Pos:69,131,102,150
Color:black
(transition1,mesqueue1):Common, Line
Pos:90,168,75,172
Color:black
(mesqueue3,transition2):Common, Line
Pos:386,329,342,310
Color:black
(state2,transition3):Common, Line
Pos:320,210,320,170
Color:black
(transition4,state3):Common, Line
Pos:135,310,215,363
Color:black

#Arc

##

附录三 多机协同指令格式

为了实现多机协同环境下数据的传送，特设计了一套指令，其中可包含各种各样的信息满足协同下的文件操作和编辑操作的要求。

指令的类型和具体的内容设计如下：

(Request：C 表示由客户提出的请求的内容；而 Response：S 表示由服务器返回的响应的内容)

==========

获得 Server 端的可访问的文件的列表：

Request： C

"command" -- "import"

Response： S

"command" -- "import"

"message" -- "filelist"

1 -- (第一个文件名)

2 -- (第二个文件名)

...

或者

"command" -- "refuse"

"message" -- (错误信息)

==========

打开一个 Server 端的文件：

Request： C

"command" -- "openfile"

"filename" -- (文件名)

Response： S

"command" -- "openfile"

"filename" -- (文件名)

"show" -- (OPNClassShow 实例)

或者

"command" -- "openrefuse"
"message" -- (错误信息)
==========
关闭一个 Server 端的文件：

Request： C
"command" -- "remove"
"filename" -- (文件名)

Response： S
"command" -- "askforsave"
or
"command" -- "closeok"
"message" -- (关闭的信息)
==========
保存一个 Server 端的文件或另存为一新文件：

Request： C
"command" -- "export"
"filename" -- (文件名)
或者
"command" -- "exportas"
"filename" -- (文件名)
"newfilename" -- (新文件名)

Response： S
"command" -- "exportok"
or
"command" -- "exporterror"
"message" -- (错误信息)
==========
在当前的 OPNClassShow 中增加一个 OPNComponent 实例：

Request： C
"command" -- (可在"state"|"transition"|"gate"|"mesqueue"|"innerclass"中取值)
"filename" -- (文件名)

"com" -- (OPNNode 实例)
"show" -- (OPNClassShow 实例)
"id" -- (Change 实例的 id)
或者
"command" -- "arc"
"filename" -- (文件名)
"com" -- (Arc 实例)
"id" -- (Change 实例的 id)
"pre" -- (OPNNode 实例)
"post" -- (OPNNode 实例)

Response： S
"com" -- (OPNNode 实例)
或者
与 Request 的内容相同。(当 Request 是要求增加 Arc 且 Arc 被成功增加上时)
或者
"command" -- "addarcerror"
"message" -- (错误信息)
"filename" -- (文件名)
==========
删除 OPNComponent 实例：

Request： C
"command" -- "delete"
"filename" -- (文件名)
"name" -- (OPNComponent 实例名)
"id" -- (Change 实例的 id)

Response： S
"command" -- "deleteerror"
"message" -- (错误信息)
"filename" -- (文件名)
"name" -- (OPNComponent 实例名)
==========

移动 OPNComponent 实例：

Request： C
"command" -- "move"
"filename" -- (文件名)
"name" -- (OPNComponent 实例名)
"x" -- (x 坐标值)
"y" -- (y 坐标值)
"id" -- (Change 实例的 id)

Response： S
与 Request 的内容相同。
==========
锁定 OPNComponent 实例：

Request： C
"command" -- "lock"
"filename" -- (文件名)
"name" -- (OPNComponent 实例名)

Response： S
"command" -- "lockok"
"message" -- (成功信息)
or
"command" -- "lockerror"
"message" -- (错误信息)
==========
对 OPNComponent 实例解锁：

Request： C
"command" -- "unlock"
"filename" -- (文件名)
"name" -- (OPNComponent 实例名)

Response： S

无。

附录四 可视化界面浏览

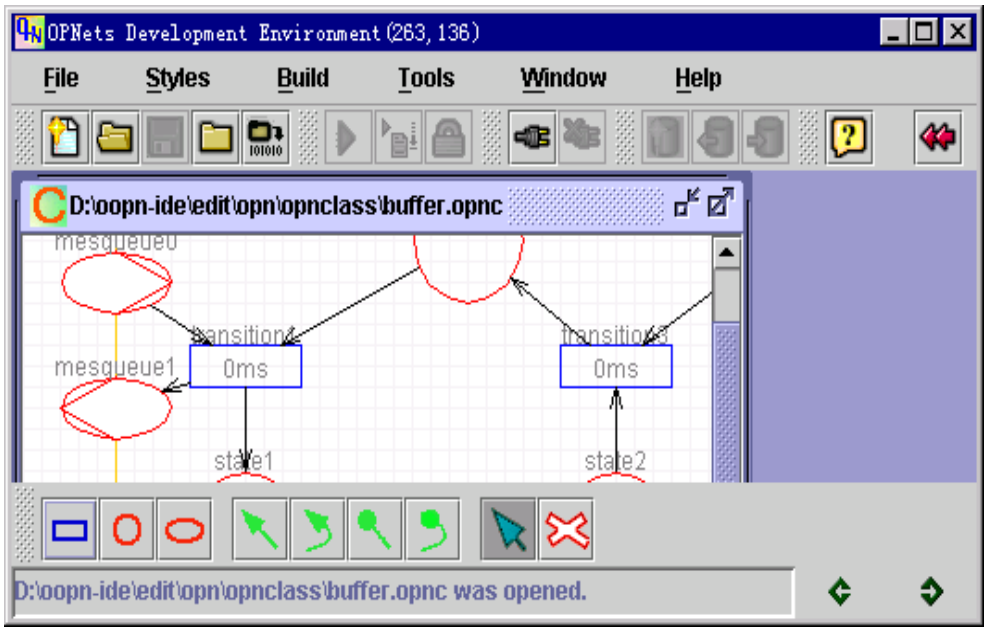


图 1 主窗口布局

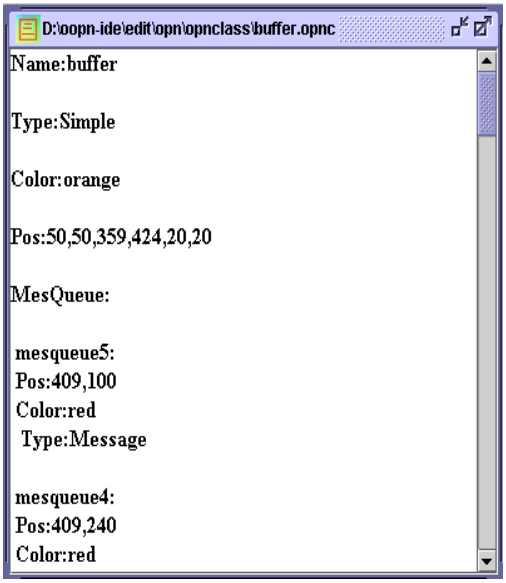


图 2 文本方式下的编辑窗口

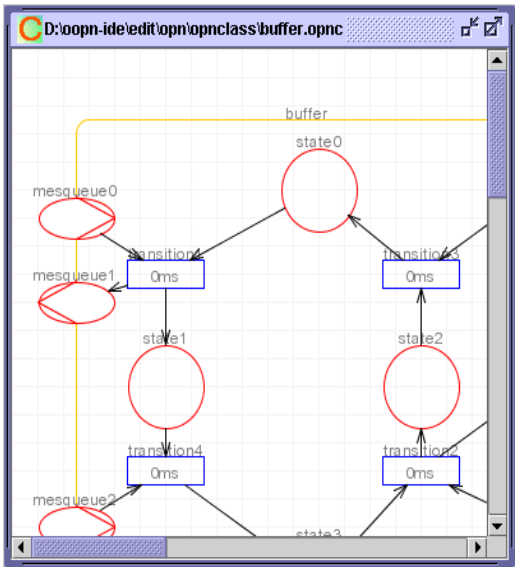


图 3 图形方式下的编辑窗口

图 3 简单类的编辑工具条

图 4 复合类的编辑工具条

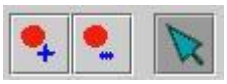


图 5 模型系统的编辑工具条

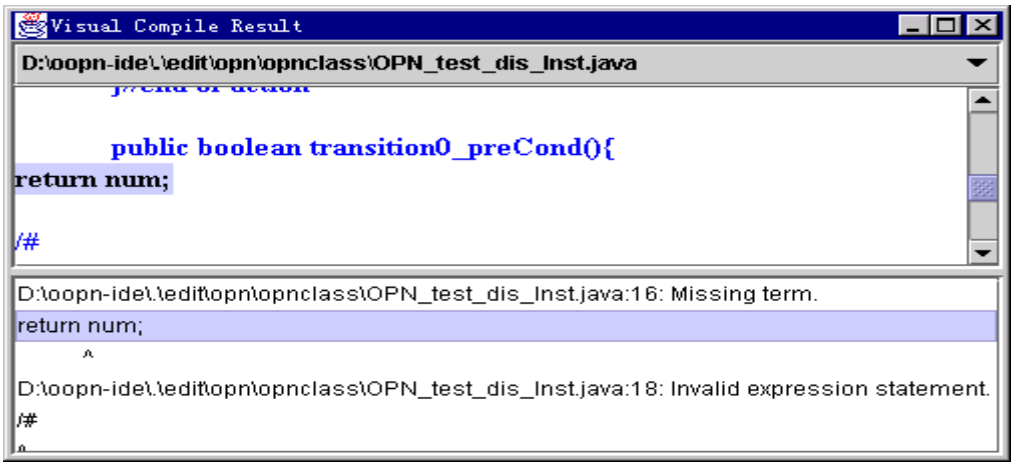


图 6 编译错误显示窗口

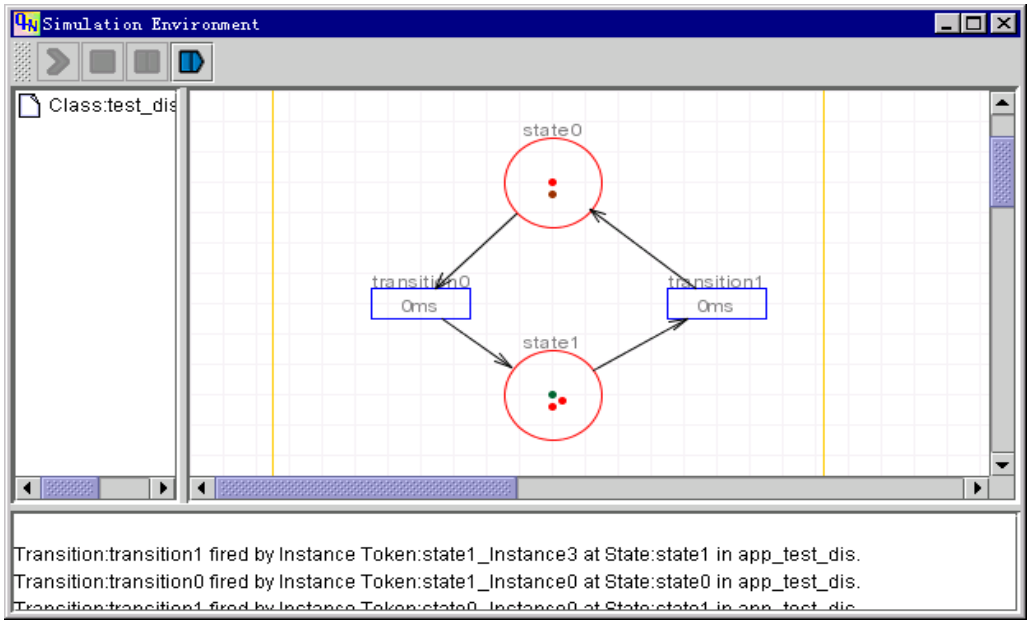


图 7 动态仿真运作窗口

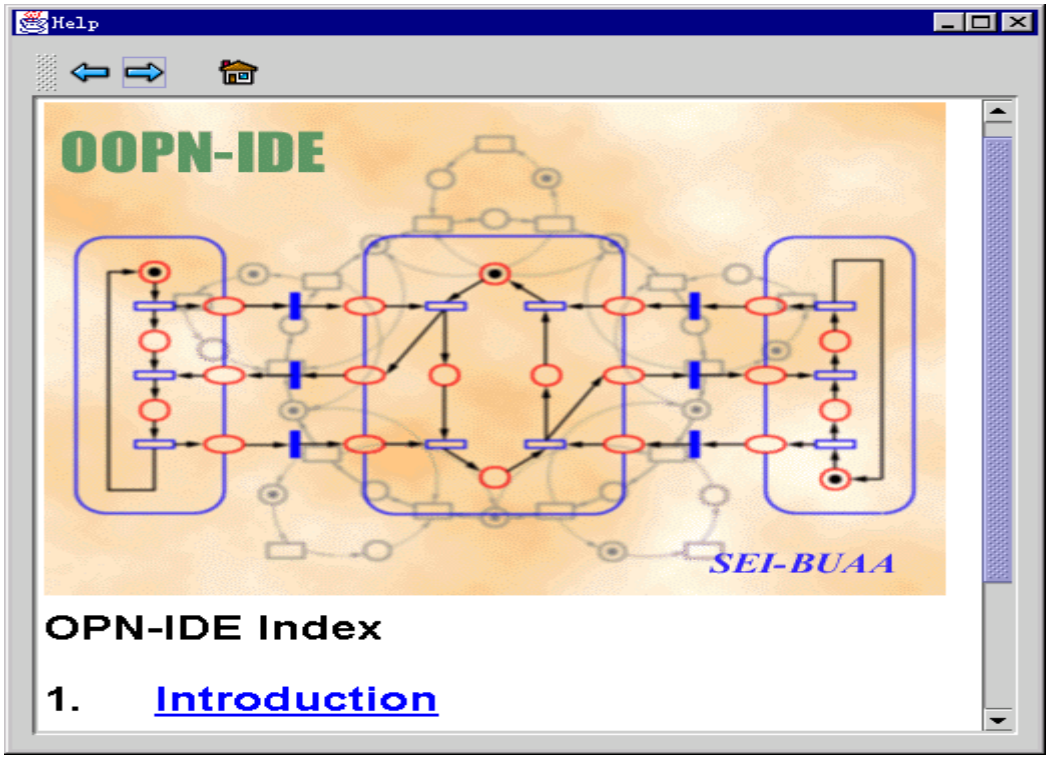


图 8 联机帮助窗口

附录五 OOPN 模型生成代码示例

以下为 show 文件、instance 文件和 app 文件示例(有删节)：

■ show 文件

```
//////////////////////////////////////////////////
OPN_test_Show.java
//////////////////////////////////////////////////
package opn.opnclass;

import java.awt.*;
import opn.baseclass.*;
import opn.opnmessage.*;
import opn.env.Globals;

public class OPN_test_Show extends OPNClassShow{
    public Message mesqueue2;//与名为 mesqueue2 的 MesQueue 对应的缓冲区

    public void setupGraphStructure(){//begin of setupGraphStructure.
            MesQueue mq=null;

            setClassName("test");//设置类名
            setShape(50,50,339,339,0,0);//设置位置

            //initialize the state part.
            State s=null;
            //创建并增加一 State 到网中
            addCom(s=new State("state1",this,220,310));
            //end of the state part.

            //initialize the transition part.
            Transition t=null;
            //创建并增加一 Transition 到网中
            addCom(t=new Transition("transition1",this,290,230));
```

```
            //end of the transition part.

            //initialize the mesQueue part.
            //创建并增加一 MesQueue 到网中
            addCom(mq=new MesQueue("mesqueue2",this,389,230));
            //end of the mesQueue part.

            //initialize the arc part.
            Arc a=null;
            OPNNode b=null,e=null;
            //创建并增加一 Arc 到网中
            addCom(a=new Arc("arc6",this,320,230,359,230));
            b=getNode("transition1");
            a.setPre(b);//设置 Arc 的起始组件
            e=getNode("mesqueue2");
            a.setPost(e);//设置 Arc 的结束组件
            b.addPostCom(a);//起始组件存放 Arc 的引用
            e.addPreCom(a);//结束组件存放 Arc 的引用
            a.setCurveStatus(false);//设置线条形式为直线
            //end of the arc part.
    }//end of setupGraphStructure.
}
```

- **instance 文件**

```
////////////////////////////////////////////////
OPN_test_Inst.java
////////////////////////////////////////////////
package opn.opnclass;

import opn.baseclass.*;
import opn.opnmessage.*;
import opn.env.Globals;

public class OPN_test_Inst extends OPNClassInst{

    //名为 transition1 的 Transition 的发生条件方法的定义
```

```
    public boolean transition1_preCond(){
            return true;
    }//end of preCond

    //名为 transition1 的 Transition 的执行动作方法的定义
    public void transition1_action(Message mesqueue2){
    }//end of action

}
```

■ app 文件

```
////////////////////////////////////////////////////////
OPN_app_App.java
////////////////////////////////////////////////////////
package opn.opnsystem;

import opn.baseclass.*;
import opn.opnmessage.*;
import opn.opnclass.*;
import opn.env.Globals;

import java.awt.*;

public class OPN_app_App extends OPN_test_Show{
    public OPN_app_App(){
            setTypeBit(Globals.Application);//设置“模型系统标志”
            setName("app");//设置名字
            setupToken();//初始化实例
    }

    void setupToken(){//setup the instances in this application.
            State state = null;
            OPNClassInst inst = null;

            state=getState("state0");//取得名为 state0 的 State 实例
            //增加一名为 OPN_app_state0_Instance0_Token 的实例
```

```
            inst = createInst(state, "OPN_app_state0_Instance0_Token");

        }
    }
```

参考文献

[1] 朱福民，并行计算的软件环境与硬件结构，计算机科学，1992，Vol.19，No.4，P23

[2] Geoffrey C. Fox，并行计算的基本问题与现状，http://www.dlut.edu.cn/idq/zhuanti/cpswt/HPCC/1/overview.htm

[3] Peter C Patlon，Multiprocessors Architecture and Applications，Computer，June 1985，Vol.18，No.6

[4] David W. Bustard，Concepts of Concurrent Programming，Carnegie Mellon University Software Engineering Institute，CURRICULUM MODULE SEI-CM-24

[5] 蔡希尧、陈平，面向对象技术，西安电子科技大学出版社，1995

[6] 麦中凡，高级程序设计语言，北京航空航天大学硕士研究生学位课讲义，1998

[7] 中国航空工业总公司，OOPN 集成开发环境鉴定证书，1998 年 1 月 18 日

[8] 杨文龙、姚淑珍、谢佩君、任爱华，基于 Petri 网的并发软件开发方法及其支持工具的研究，科学技术文献出版社，1993 年

[9] 刘海燕、陈火旺，并发模型分析，计算机科学，1995，Vol.22，No.3

[10] 袁崇义，Petri 原理，电子工业出版社，1998

[11] Tadao Murata，Petri Nets:Properties, Analysis and Applications，Proceedings of IEEE，Vol.77，No.4，April 1989

[12] Yang Kyu Lee、Sung Joo Park，OPNets:An Object-Oriented High-Level Petri Net Model for Real-Time System Modeling，J. of Systems and Software，Jan 1993，Vol.20，No.1， pp69-86

[13] H.J.Genrich、K.Lautenbach，System Modelling with High-level Petri Nets，Theoretical Computer Science，1981，Vol.13，pp109-136

[14] Guat Yew Tan、Gurdeep Singh Hura，MASE:A User-Friendly Performance Tool，Microelectron. Reliab.，1996，Vol.36，No.6，pp821-841

[15] Yi Den、S.K.Chang、Jorge C.A. de Figueired、Angelo Perkusich，Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems，Applications and Theory of Petri Nets，1993，14th International Conference Proc.， pp206-223

[16] Tang Fagen、Yao Shuzhen、Yang Wenlong，The Research and application of the general Object-Oriented Petri Net(ONet)，Proc. of TOOLS Asia’97 & OOT China ’97，1997，pp13-17

[17] Olivier Biberstein、Didier Buchs，An Object-Oriented Specification Language Based on Hierarchical Petri Nets，IS-CORE Workshop(ESPRIT)，Amsterdam，Sep.1994

[18] C.D. Keen、C.A. Lakos，A Methodology for the Construction of Simulation Models Using Object-Oriented Petri Nets，Proc.of the European Simulation Multiconference 1993，pp267-271

[19] Sarah L Englist，Colored Petri Nets for Object-Oriented Modelling，a dissertation of Ph.D.，Univ. of Brighton，June 1993

[20] 吴芸、杨汉瑜、任爱华、杨文龙，Petri 网与面向对象技术结合应用的发展，计算机世界报，1995 年 9 月 20 日，Petri 网专版

[21] 胡剑玲，面向对象有色 Petri 网(OOCPN)和 OOCPN 系统设计方法的研究，博士毕业论文，中国科学院数学研究所，1996 年 7 月

[22] Michael B. Feldman，Language and System Support for Concurrent Programming，Carnegie Mellon University Software Engineering Institute，CURRICULUM MODULE SEI-CM-25

[23] Ian Foster，Designing and Building Parallel Programs，http://www.mcs.anl.gov/dbpp/text/book.html，1995

[24] G.Bruno、R.Agarwal、A.Castella、M.P.Pescarmona，CAB:an Environment for Developing Concurrent Application，Proc. of the 16th International Conference on Application and Theory of Petri Nets，1995，pp141-160

[25] 陈小群，Coad/Yourdon 方法，水木清华 BBS 站软件工程版精华区，1998

[26] http://www.daimi.au.dk/PetriNets/tools/，Internet 上的 Petri 网工具列表，1999

[27] 袁崇义，C.A.Petri 与计算机科学，计算机科学，1987，pp62-64

[28] 任爱华、牛锦中、张永鸣，一种基于面向对象 Petri 网的并发程序建模方法，北航学报，1998，Vol.24，No.4

[29] 任爱华、牛锦中，面向对象 Petri 网建模方法，TOOLS Asia’97&OOT China’97

[30] 任爱华、牛锦中、孙自安、林仕鼎，多机协同建立并发系统模型实现原理，计算机科学，1999(待发表)

[31] 任爱华、牛锦中、孙自安、杜悦冬，OOPN 集成开发环境，计算机科学，1999(待发表)

致谢

首先感谢导师任爱华副教授。正是她的严格要求、正确引导，才促成了本文。任老师的关心与爱护使我始终感到温暖。是任老师指导我掀开了充满诱惑与激情的计算机科学宏篇巨制的第一页。

我要特别感谢周伯生教授。与周老师的短暂接触，使我能跳出小圈子去看大世界，所学的东西将使我受益终生。

我还要特别感谢麦中凡教授。麦老师的细致入微、充满哲理的教导是我的许多想法与思路的源泉。

我还要感谢刘又诚教授、张莉副教授、王雷老师及周宇辰、葛科、车向东、谭文安、陆伯鹰、马云静、王云、孙自安、杜悦冬、杨顺祥、尹春梅等师兄弟姐妹，没有他们的指导、关心与帮助，也不会有我的工作的顺利完成。

感谢陈羽𠈌、吕良权同学，与他们的讨论使我们能够共同进步。

最后要感谢我的父母、姐弟、我的爱人，是他们与我共担或苦亦甜的一切。