工作流模式简介

整理：乔彬

E-mail:qiaobin911@163.com

http://www.workflow.sh.cn

目录

1 引言

我们知道，由于工作流产品（工作流管理系统）众多，而它们之间又缺乏统一的标准，使得不同的产品之间很难实现协同工作。为了解决这一问题，工作流管理联盟（WFMC）于 1993 年成立，并提出了工作流参考模型，制定了 5 个标准接口。其中有一个接口是过程定义接口。几乎每个工作流管理系统都有自己的过程定义语言（也称为工作流语言）。

可以从四个方面（控制流、数据流、资源、操作）来研究流程，“ 工作流模式 ” 只是提及到其中的控制流部分。控制流（control flow）描述了活动在不同结构中的执行顺序。Control flow 对我们有效认识、理解工作流规范具有很大帮助。工作流规范需要不断地扩展，以便满足新的需求，因此有必要对 Control flow 进行基础的认识和分析。目前大多数工作流语言都支持的基本结构是：顺序（sequence）、迭代（iteration）、分支（split）、合并（join）。但是对以上几种基本结构的解释并不一致，而且这些基本结构对更多复杂的结构的支持也不清楚。

存在的多种工作流语言也都基于不同的语义，比如，有的工作流语言支持多实例，有的不支持；有的工作流语言只支持结构化循环，有的可以支持任意循环；有的工作流语言只支持显示终止，而不支持隐式终止。

工作流模式系统化地表述了基本的和复杂的结构（流程）。

模式（pattern）是从具体形式中抽象出来的。面向对象的设计模式，规定了不依赖于具体的实现技术，同时也不依赖于所在领域的基本需求。（**原文：**A pattern “ is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts”. Gamma et al. first catalogued systematically some 23 design patterns which describe the smallest recurring interactions in object-oriented systems. The design patterns, as such, provided independence from the implementation technology and at the same time independence from the essential requirements of the domain that they were attempting to address.）

工作流模式提出了业务需求，但它又不拘泥于具体的工作流语言。但这并不是说，工作流模式是表述业务需求的唯一方法（**原文：**Patterns address business requirements in an imperative workflow style expression, but are removed from specific workflow lanaguages. Thus they don’t claim to be the only way of addressing the business requirements）。目前而言，还没有任何一个工作流管理系统能支持全部的工作流模式。有一点很重要，下面提到的模式，仅限于静态控制流，暂不考虑资源分配、实例控制、异常处理和事务管理。

2 工作流模式

工作流模式并没有采用具体的形式化语言来进行描述，也有一点遗憾（胡长城：大家还是在各自的理解上阐述着对 workflow patterns 的理解和表示。也许理解上来说，基本上都已经达成了一个共识，毕竟这是 Aalst 成果。但是可惜 Aalst 早期没有对如何用一个明确的“语义”或“符号”去表示这个 workflow patterns。—— 如今，bpmi 开始做这个 notation 的统一了，但是却依然有这么大的歧义…）。

在本节介绍的模式中的，凡是 XPDL 语言支持的都给出了采用 JaWE2.2 画出的流程图，并使用工作流引擎 Shark1.1 和 2.0 运行通过的；凡是 XPDL 语言不支持的模式，都会注明。

首先要介绍 XPDL 语言中的几个概念：（XPDL Version1.0）

流程（Process）是由活动（Activity）和子流程（Subflow Activity）按照一定的顺序结构结合而成。

转移（transition） 或者称变迁用于连接活动与活动或者活动与子流程之间的有向弧。转移又分为无条件转移和条件转移。条件转移就是指在执行该转移要满足一定的条件，如设置了某个表达式 ok==true，只要该表达式为真就执行这个转移，否则就不执行这个转移。无条件转移就是指一定会执行该转移，不需要满足任何条件。见图 4-1，在图中，带箭头的比较细的弧是无条件转移，而相对比较粗的带箭头的弧是条件转移。

活动又可以分为四种（当然，子流程也可以认为是活动的一种）：

普通活动（activity without implementation）：不执行应用或者动作的活动。

工具活动（tool activity）：执行应用或者动作的活动。

路径活动（route actibity）：只用于判断选择路径。

块活动（block activity）：由多个活动（以上三种活动）组成活动。

2.1 基本控制流模式(Basic Control Flow Patterns)

Pattern 1 顺序(Sequence)

- **描　述**　在流程中，活动按照预先设定的顺序一个接一个地执行。
- **同义词**　串行
- **例　子**　在网上书店买书，选择“先行付款”方式，只有“付款”活动完成后，才会激活“送货”活动。
- **实　现**　将两个活动用无条件转移连接即可。
- **流程图**　活动 B 必须在活动 A 执行完成以后才能执行。A、B、C 都是普通活动，转移都是无条件转移。

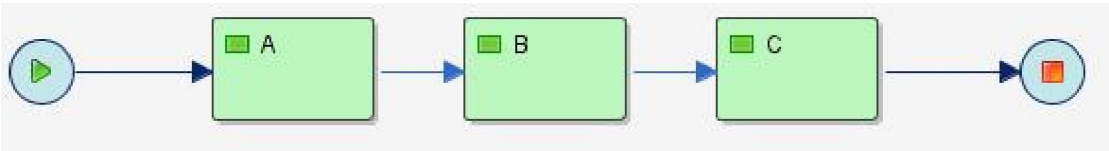

图 1

Pattern 2 并行(Parallel Split)

- **描　述**　流程在某个活动之后产生多个分支，这些分支同时运行或者按任意顺序运行。
- **同义词**　AND-split
- **例　子**　“付款”完成后，激活两个活动分支：“送货”和“告知客户”(通知客户货款已经收到，正在送货)。
- **实　现**　①显式 AND-split，可以定义一个路径活动，当路径活动执行完以后，产生多个并行分支。②隐式 AND-split，不支持路径活动的定义，在普通活动完成后，直接产生多个并行分支。
- **流程图**　AND-split 是一个路径活动活动。A 执行完后会激活 B、C 两个活动，图 2-1 为显式 AND-split，图 2-2 为隐式 AND-split。可以注意到：图 2-1 中的活动 AND-split

和图 2-2 中的活动 A，它们的右边（即 Split—分散类型）都有一点向外突出，并且后面的分支都是无条件转移，表示这个活动后面的分支是并行。

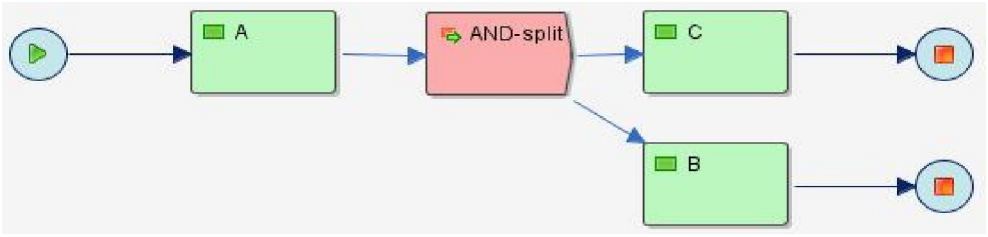

图 2-1

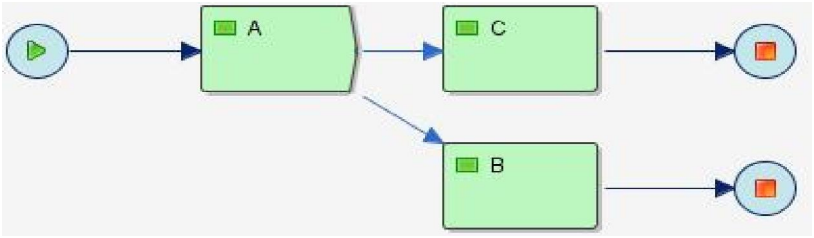

图 2-2

Pattern 3 同步(Synchronization)

- **描　述**　在流程中的某个点，多个并行的子流程或者活动，合并成一个流程。流程必须等待所有的分支都执行完以后，才能激活后续活动，这就是“同步”之意。
- **同义词**　AND- join
- **例　子**　在“送货”和“告知客户”这两个并行活动完成后，激活“存档”活动。
- **实　现**　①显式 AND-join，可以在同步的并行分支后面定义一个路径活动，执行路径活动，同步多个并行分支。②隐式 AND- join，不支持路径活动的定义，在后续普通活动中同步多个并行分支。
- **流程图**　A 执行完后会激活 B、C 两个并行活动，B、C 同步后，激活 D。图 3-1 为显式 AND- join，图 3-2 为隐式 AND- join。可以注意到：图 3-1 中的活动 AND- join 和图 3-2 中的活动 D，它们的左边（即 Join—合并类型）都有一点向内凹陷，并且前面的分支都是无条件转移，表示这个活动同步前面的并行分支。

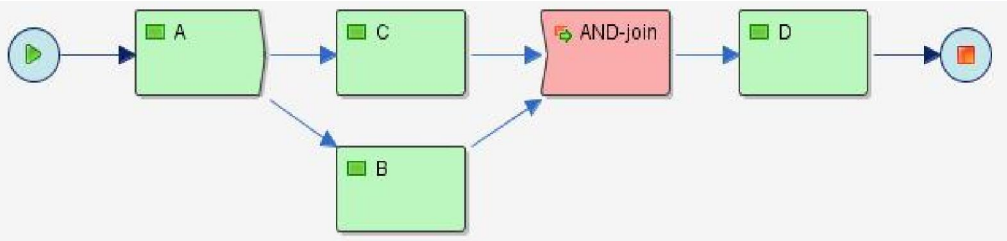

图 3-1

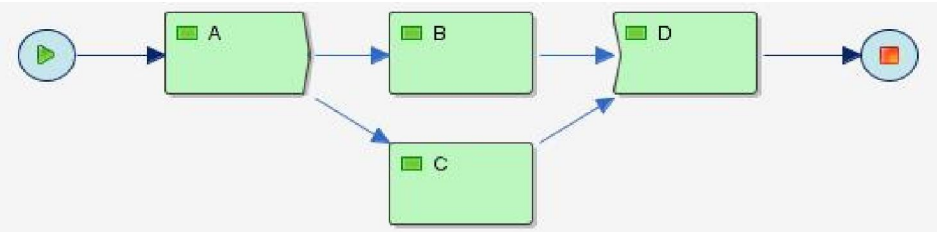

图 3-2

Pattern 4 独占式选择(Exclusive Choice)

- **描　述**　当一个活动完成以后，可以有多个分支进行选择，但是只能选择其中的一个分支，即多选一。
- **同义词**　XOR-split
- **例　子**　在网上购书，“下完订单”后，选择“支付方式”，可以选择“银行卡付款”、“邮局汇款”或者“货到付款”，只要选择一种方式即可。
- **实　现**　①显式 XOR -split，可以定义一个路径活动，当路径活动执行完以后，自动选择某一个分支。②隐式 XOR -split，不支持路径活动的定义，在普通活动完成后，直接选择一个分支。
- **流程图**　A 执行完后可以选择 B、C、D 三个活动中的任意一个。图 4-1 为显式 XOR -split，图 4-2 为隐式 XOR -split。可以注意到 图 4-1 中的活动 XOR -split 和图 4-2 中的活动 A，它们后面的分支都是条件转移（相比无条件转移加粗了），在活动 A 处定义了一个变量 whereToGo，执行 A 时输入它的值，活动 B、C、D 前边的条件转移设置的表达式分别是：whereToGo==”B”， whereToGo==”C”和 Otherwise（否则，即前边两个条件都不满足就会选择这个分支，颜色为桔黄色）。

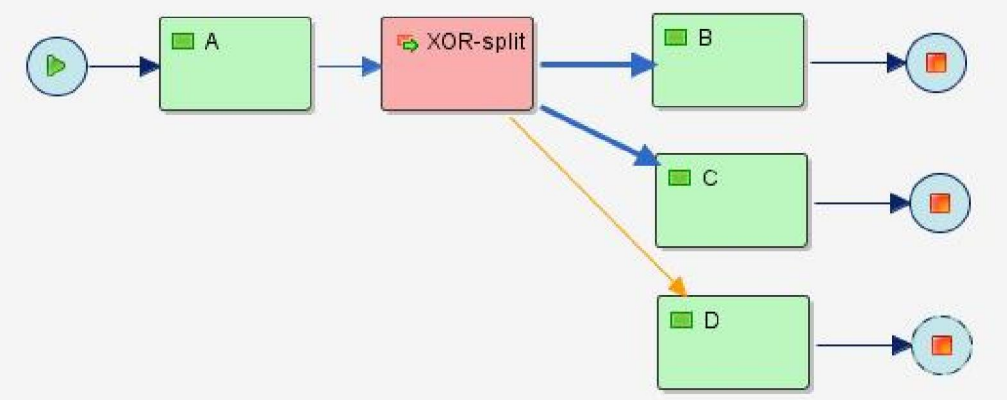

图 4-1

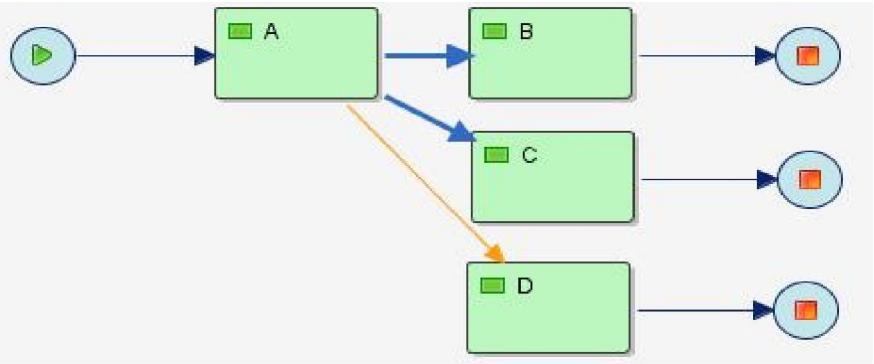

图 4-2

Pattern 5 简单聚合(Simple Merge)

- **描　述**　在流程中，有两个或多个可选择的分支（不允许任意两个或多个分支并行），在某一点处合并成一个分支，但并不是同步合并（与 Pattern 3 的区别）。与 Pettern 4 也有点相似，都是“多选一”，但 Pettern 4 是分散，而 Pettern 5 是合并，一般采用“先进

先出”原则，但是后续活动只产生一次（如果后续活动执行多次产生多实例，就是 Pettern 8）。

- Ø **同义词** XOR- join
- Ø **例 子** 比如在 Pettern 4 中提到的，不论选择了哪种支付方式，只要这个活动完成，就会下一个活动“送货”。
- Ø **实 现** 简单聚合模式前提是：假设可选择的分支不会并行执行（如图 5-1），这样所有的工作流语言都支持这种模式。一些工作流语言强制要求，在简单聚合模式中可选择的分支不能出现并行的情况；如果出现并行分支（如图 5-2），会怎样呢？
- Ø **流程图** A 执行完后可以选择 B、C、D 三个活动中的任意一个，不论选择哪一个分支，该活动完成后都会激活 E。图 5-1，活动 E 前面的转移都是无条件转移。

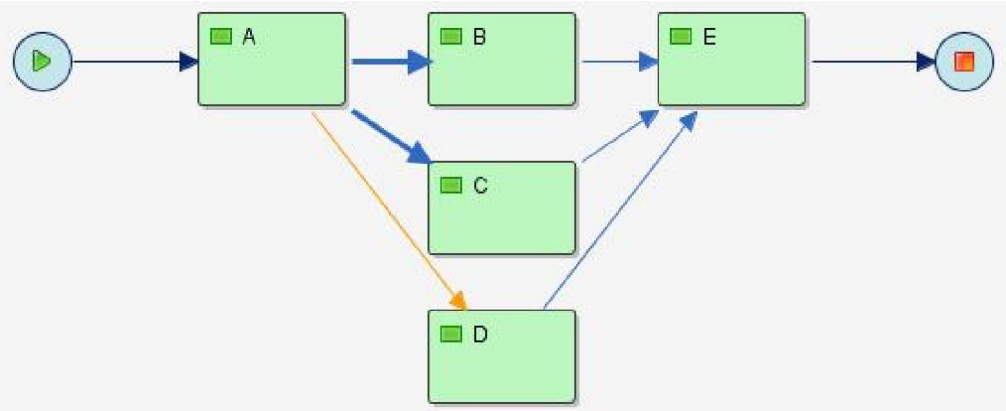

图 5-1

在下图中，A 执行完后产生 B、C 两个并行分支，B、C 在 XOR- join 聚合后激活 D。在 XPDL 中，这种模式被当做多重聚合（Pattern 8），具体运行情况请参见 Pattern 8。

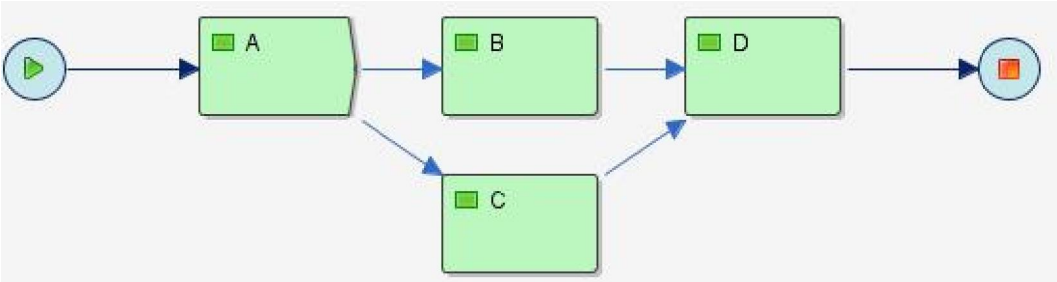

图 5-2

以下的高级复杂模式都是由这几种基本模式组合而成，只给出隐式模式，不再给出显式模式，以力求简洁。

2.2 高级分支同步模式(Advanced Branching and Synchronization Patterns)

Pattern 6 多重选择(Multiple Choice)

- Ø **描 述** 在流程中，当一个活动完成以后，有多个分支进行选择，可以选择其中的一个或者多个分支，即“多选多”。Pattern 4 独占式选择是“多选一”模式。选择的多个分支可能存在并行执行的情况。

Ø **同义词** OR-split

Ø **例 子**

Ø **流程图** A 执行完后可以选择 B、C、D 三个活动中的一个或者多个。活动 A 后面的转移是条件转移，转向 B、C、D 的转移上设置的表达式分别是：whereToGo.indexOf('B') != -1，whereToGo.indexOf('C') != -1，whereToGo.indexOf('D') != -1。如果在 A 中设置 whereToGo 的值为“ BC ”，则激活 B 和 C，这两个分支可能并行执行。

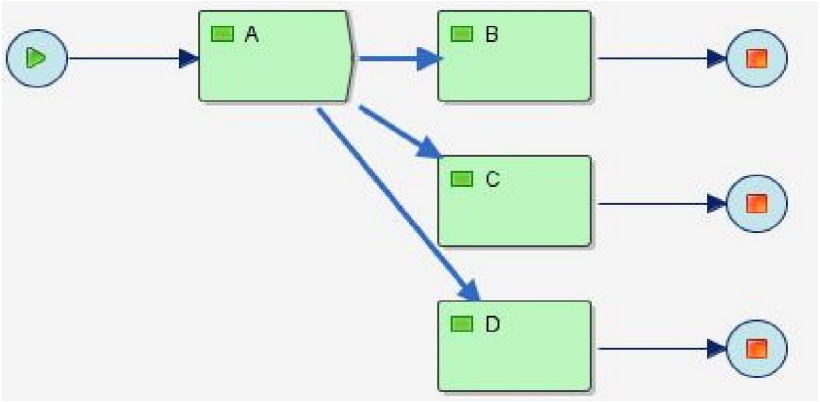

图 6-1

Ø **问 题** 有的工作流语言并不支持多重选择。可以将 OR-split 转化成先 AND-split 再 XOR-split 的形式或者是先 XOR-split 再 AND-split 的形式。如图 6-2：

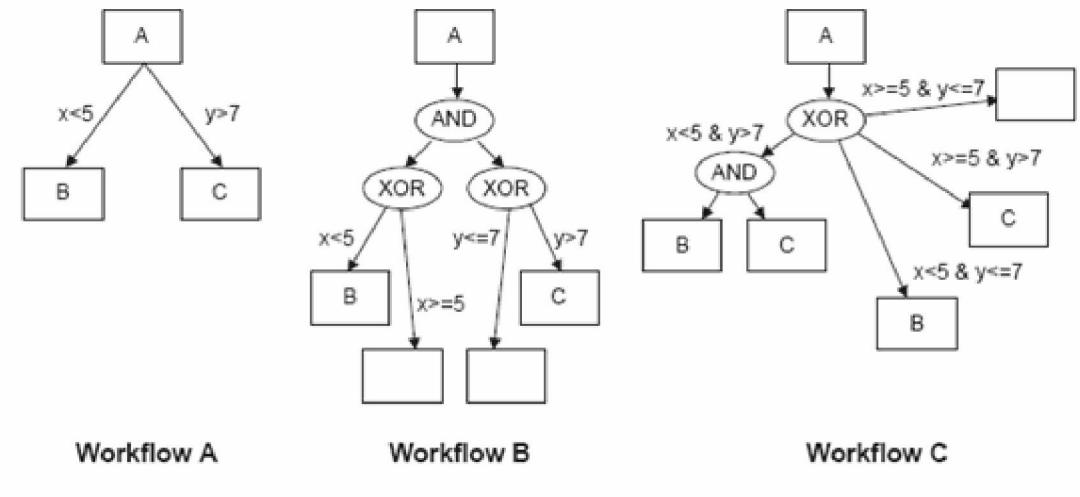

图 6-2

Pattern 7 同步聚合(Synchronizing Merge)

Ø **描 述** 在流程中的某个聚合点，多个分支路径合并成一个路径。在聚合点，流程会等待所有的分支到来，才能激活发续的活动。这个模式可以选择分支路径，如果只选择一个分支，实现的功能类似于 Pattern 5 简单聚合模式；如果选择两个及以上的分支，实现的功能类似于 Pattern 3 同步模式。注意：这种模式不能出现在循环结构中!

Ø **同义词** Synchronizing join

Ø **例 子**

Ø **实 现** 这个模式实现的困难在于决定什么时候同步，什么时候合并。它即有同步的功能，又有简单聚合的功能。XPDL 是采用图 7 所示的方式实现的。

Ø **流程图** 在图 7 中，在活动 A 中设置 whereToGo 的值，B、C 前边的转移上设置的表达式分别是：whereToGo.indexOf('B') != -1，whereToGo.indexOf('C') != -1。如果在 A 中设置 whereToGo 的值为“ B ”，则激活 B；如果设置为“ C ”，则激活 C；如果设置为“ BC ”，则激活 B 和 C，产生同步。

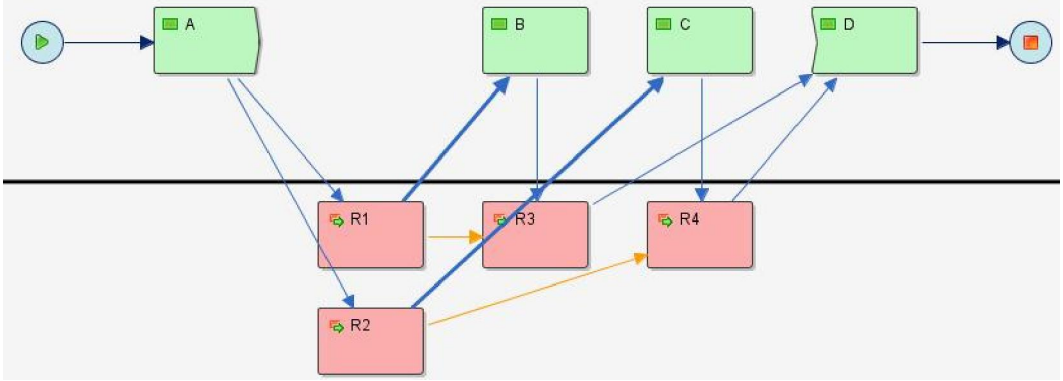

图 7

Pattern 8 多重聚合(Multiple Merge)

Ø **描 述** 在流程中多个分支（可能是 Pattern 6 多重选择的一个或多个分支；也可能是 Pattern 2 并行中的多个分支），在聚合时每个分支执行完都会激活后面的活动或流程（也就是会产生多实例）。与 Pattern5 简单聚合的区别在于：简单聚合的分支只有一个可执行并且后续活动只激活一次；而多重聚合是多个分支可执行，后续活动激活多次，产生多实例。

Ø **例 子**

Ø **实 现** 如果多重聚合不出现在循环中，也就说多实例的个数是确定的，那么就把后续活动复制到每个分支后边就可以了（如图 8-1）；如果多重聚合出现在循环中，多实例的个数是不确定的，就要用到 2.4 节中介绍的包含多实例的模式。

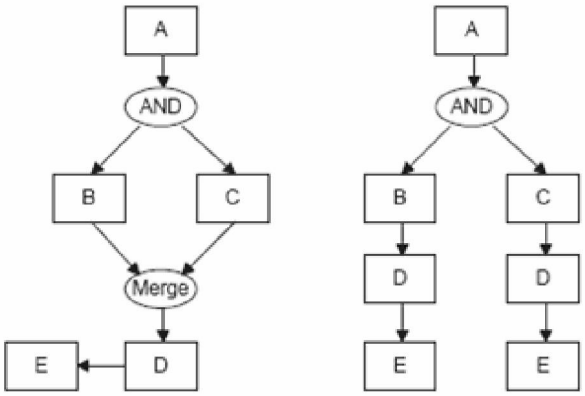

图 8-1

Ø **问 题** 有的工作流引擎不支持多重聚合；有的不会产生第二个实例，也就是不论前边有几个可以激活的分支，都不会再产生第二个实例；有的在第一个实例未结束前，不会产生第二个实例。

Ø **流程图** 图 8-2 中，S1 是子流程（如图 8-3）。A 执行完后多重选择 B、C、D 三个活动中的一个或多个，假如选择了 B 和 D，B 执行完成后，会激活产生 S1 的第一个实例，

同样在 D 执行完成后，会产生 S1 第二个实例，这样就是多实例了。注意：在 XPDL 中，S1 的执行方式要选择“ Asynchronization ”，即异步。

下面说一下后续活动执行方式 Asynchronization、Synchronization （即异步或同步）对多重聚合的影响（仅限 XPDL）：

①在图 8-2 中，子流程 S1 的执行方式是设置为“ Asynchronization ”，那么 B 执行完成产生子流程 S1 的一个实例，C 执行完成仍然产生 S1 的第二个实例，这两个实例可以同时存在；如果 B 激活产生的第一个实例执行完后，才执行 C，仍然能产生 S1 的第二个实例。

②现在将图 8-2 中，子流程 S1 的执行方式是设置为“ Synchronization ”，如果 B 执行产生的 S1 的第一个实例未结束，那么 C 执行后就不会再产生 S1 的第二个实例；如果 B 执行后产生 S1 的第一个实例运行结束了，再执行 C，才会产生 S1 的第二个实例。

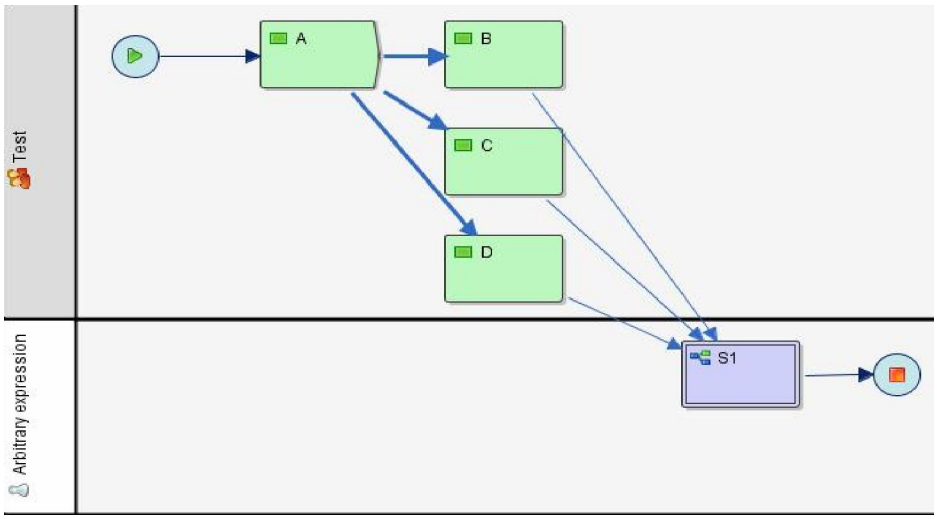

图 8-2

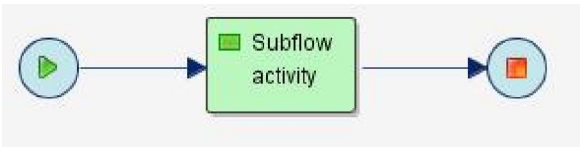

图 8-3

对于子流程可以设置它的执行方式是同步还是异步，但是对于普通活动，在多重聚合中它的执行方式默认为 Synchronization。

如图 8-4（同图 5-2），A 活动 AND-Split 执行完后产生 B、C 两个并行分支，B、C 在 XOR- join 聚合后激活 D。也就是先并行分支然后再多重聚合，这个流程的执行结果与上面②的结果是一样的：只要运行 B 产生 D 的第一个实例未结束，就不会再产生第二个实例；第一个实例结束了，再执行另一个并行分支活动 C，仍然会产生 D 的第二个实例。

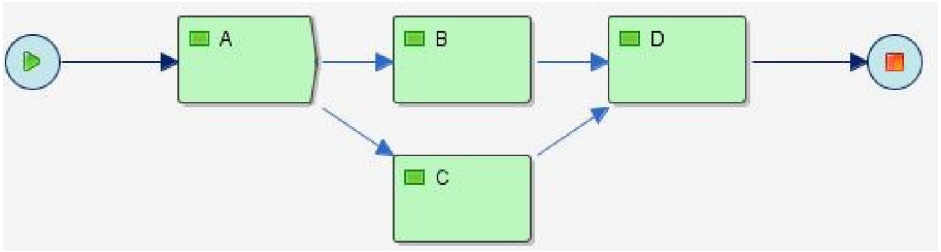

图 8-4

在图 8-5 中，先并行分支然后再多重聚合（S 是一个子流程），与图 8-4 流程相比它含有子流程，可以设置子流程的执行方式。如果设置成“ Asynchronization ”，执行结果同①；如

果设置成“ Synchronization ”，执行结果同②。

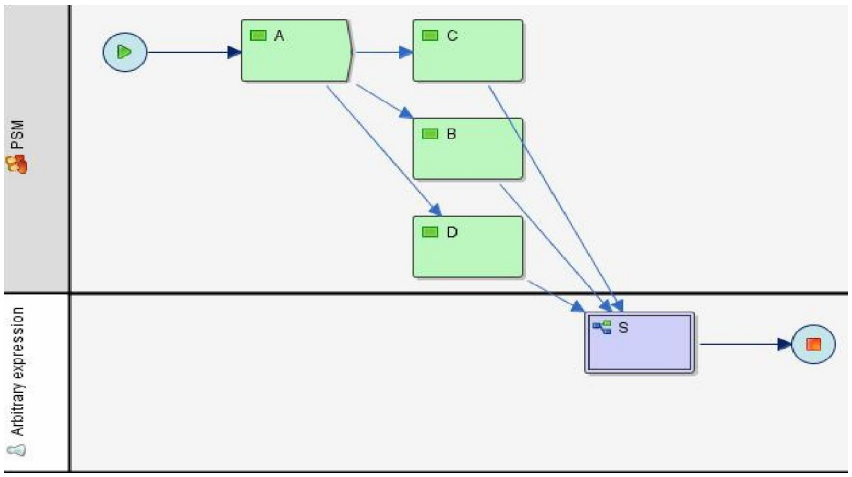

图 8-5

Pattern 9 鉴别器(Discriminator)

Ø **描 述** 在流程中的某个聚合点，等待所有的分支（可能是并行分支，也可能是多重选择分支）中的第一个分支执行到达后，就立刻激活后续活动；与此同时，流程仍然要等待其余的分支执行完成，并忽略它们（这些分支不会再激活后续活动了）。注意：在其余分支未全部完成前，第一个到达的分支激活的后续活动是无法执行的。胡长城在《工作流模型分析》中，将这个模式看成是一个简单的路径活动，这是不对的。

Ø **例 子** 东华大学硕士研究生申请助教的例子：“ 个人申请提交 ”后，并行分支给“ 导师审批 ”和“ 辅导员审批 ”，只要两人有一个审批了，那么就会提交给“ 学院审批 ”。

Ø **流程图** 在 XPDL 中，鉴别器模式是按图 9-1 中的流程实现的。A 执行完成后，多重选择 B、C、D，第一个活动执行后，就会激活 E，等待其余的活动执行完成后，执行 E 方能结束。在 shark1.1 中执行无任何问题(在 Shark1.1 中应该将工具活动放在自由角色中，否则会报错)，在 2.0 中运行有点问题（活动 E 会出现隐式终止的情况，这里就不再具体讨论)。

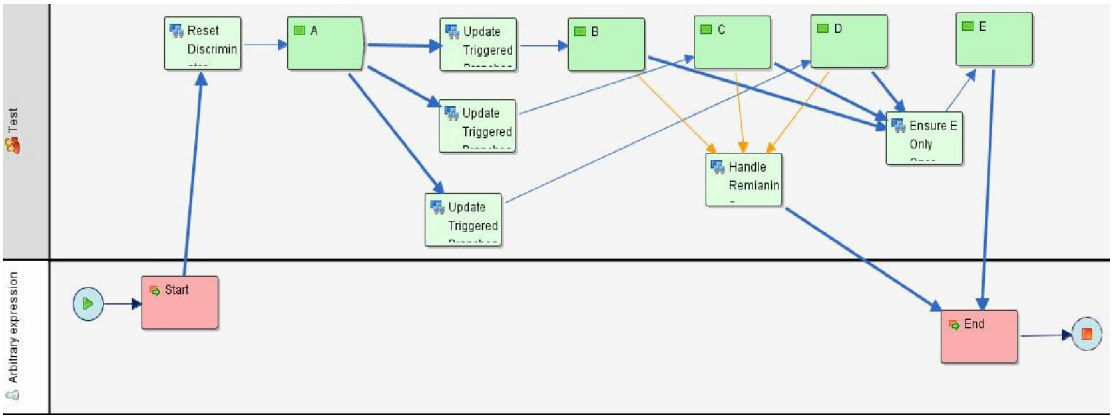

图 9-1

上面的鉴别器模式是等待 M 个分支中的一个到达，如果要等待 M 个分支中的 N（N 的最小值为 1，最大值为 M）个分支到达，应该如何实现呢？图 9-2 是一个 2-out-of-3 的实现方式。通过这个图不难理解 N-out-of-M 模式的实现方法。

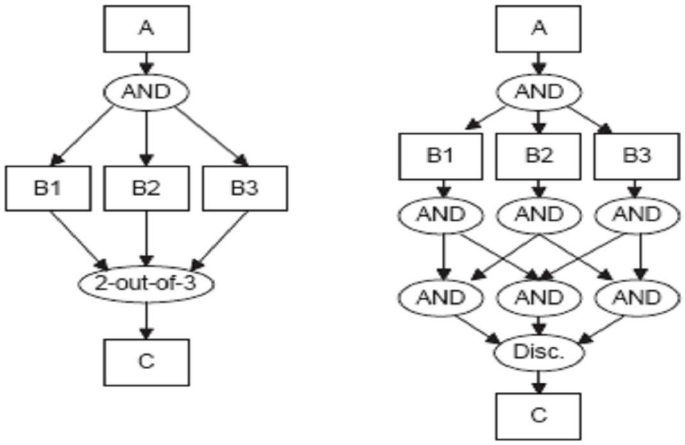

图 9-2

◆ N-out-of-M 鉴别模式

Ø **描　述**　在流程中的某个聚合点，等待所有的 M 个分支(可能是并行分支，也可能是多重选择分支）中的前 N 个分支执行到达后，就立刻激活后续活动；与此同时，流程仍然要等待其余的 M－N 个分支执行完成，并忽略它们（这些分支不会再激活后续活动了)。注意：在其余 M－N 个分支未全部完成前，激活的后续活动是无法执行的。

Ø **例　子**　“个人申请提交”后，并行提交给“第一导师审批”、“第二导师审批”和“辅导员审批”，只要三个人有两个审批了，那么就会提交给“学院审批”。

Ø **流程图**　在 XPDL 中，该模式是按图 9-3 中的流程实现的。（不再具体讨论，如果有兴趣可以自己下载相关软件进行验证)。

下载地址：http://download.csdn.net/source/287186

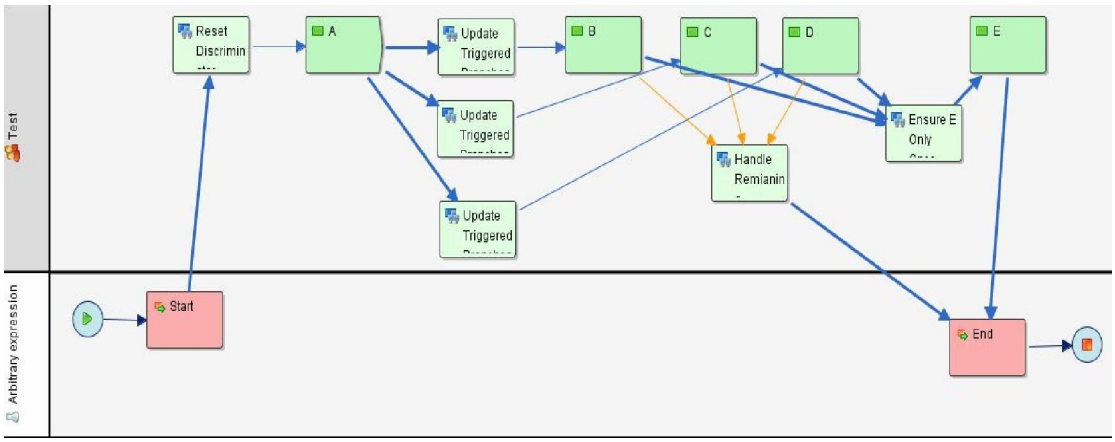

图 9-3

2.3 结构化模式(Structural Patterns)

Pattern 10 任意循环(Arbitrary Cycles)

Ø **描　述**　某一个或多个活动可以反复执行。

Ø **同义词**　Loop，iteration，cycle

- **问　题**　有的工作流语言只支持结构化循环：即循环只有一个输入和一个输出。而任意循环可以有多个输入和输出。结构化循环好比 while，而任意循环就是 goto。
- **实　现**　在支持循环模式的工作流语言中，采用独占式选择很容易就可以实现任意循环。
- **流程图**　在图 10 中，活动 A 运行完成后激活 B，B 执行完成后激活 C，在 C 中设置 whereToGo 的值为“ B ”，运行后流程又返回 B，否则流程激活 D ;在 D 中设置 whereToGo 的值为“ C ”，运行后流程又返回 C，否则流程激活 E。

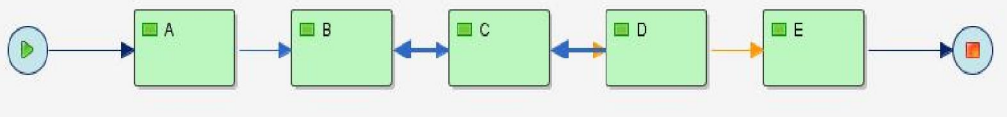

图 10

Pattern 11 隐式终止(Implicit Termination)

- **描　述**　隐式终止就是指在一个流程中，如果没有活动可执行了那么流程就会终止。换句话说，在工作流中没有 active 状态的活动了，而且也没有活动会被激活，这就是隐式终止。（前提：工作流不能处于死锁状态）
- **问　题**　有的工作流只支持显式终止，一旦显式终止了，其他正在执行的活动也会被放弃。
- **流程图**　在图 11 所示的流程中，A 运行完成后多重选择 B、C，活动 B 设置 whereToGo 的值为“ D ”，则激活 D，否则上面的分支就会终止；同样，活动 C 设置 whereToGo 的值为“ E ”，则激活 E，否则下面的分支也会终止；两个分支都隐式终止了，流程也就隐式终止。在种模式在 XPDL 中很常见，有时错误的输入经常会导致整个流程的隐式终止。

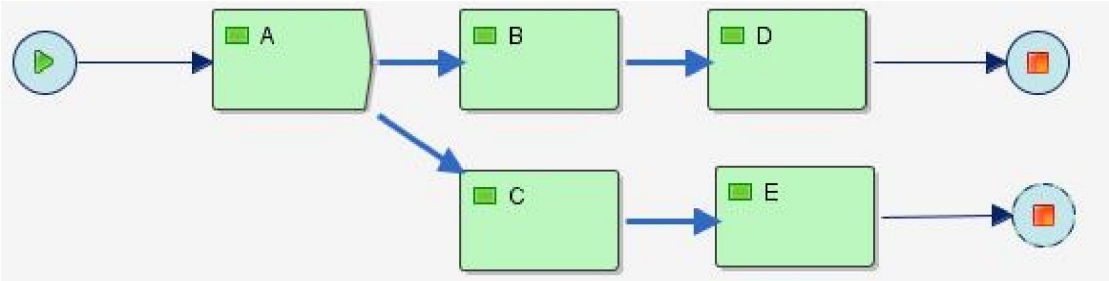

图 11

2.4 包含多实例的模式(Patterns involving Multiple Instances)

在前面的 Pattern 8 中已经提到了多实例，这里在详细说明一下。

从理论的观点来看，“ 多实例 ”的概念比较简单，和执行共享同一个定义的多个线程差不多；从实践的观点来看，“ 多实例 ”是指在流程图中，一个活动在同一时刻拥有多个可运行的、处于活动状态的实例。这种模式实现的主要问题是：由于设计的约束和实际需求的不可预期性，大多数工作流引擎并不允许同一个活动在同一个时刻拥有多个处于活动状态的实例。以下我们用 MI 来代替 Multiple Instances.

根据以下两种需求来考虑多实例的问题：⑴要具有激发一个活动或者子流程产生多个实例的能力；⑵要具有在多实例操作完成后同步它们，并激发后续流程的能力。所有的多实例模式都必须满足第一个条件；如果多实例不需要同步，那么第二个条件就不是必须的。比如我们前面提到的 Pattern 8 多重聚合和 Pattern 11 隐式终止：在多重聚合中，允许产生多个实例，但是并不需要同步它们；如果产生的多实例不需要同步，那么每个实例的终止都是隐式的，不需要和主流程协同。

多实例的同步和实例的个数是高度相关的。如果实例的个数是固定的，并且在设计时就已经知道了，那么这种同步是很容易实现的。如果实例的个数在运行时才确定，或者在操作实例过程中会改变，那么这种同步的实现是非常困难的。在这节中将介绍三种多实例同步模式。不需要同步的多实例与实例的个数关系不大，在这里只介绍一种。

Pattern 12 无同步的多实例(MI without Synchronization)

- **描　述**　在流程中，一个活动可以激活多个实例，也就是说可以把一个活动分发成几个控制线程。每个控制线程之间都是相互独立的，并不需要同步它们。
- **同义词**　Multiple threading Without Synchronization
- **例　子**　在网上书店买书，同时订购了多本书。支付账单、更新客户记录都是以订单为单位进行的；如果以书为单位进行，每一本书都会产生一个实例，进行更新库存、邮寄等，要是不需要同步的话，可以采用该模式。
- **实　现**　如果工作流引擎支持无交互的合并，并且允许一个活动同时产生多个实例，这种模式的采用循环和并行分支结构来实现是非常容易的（如图 12-1 所示）。还有的工作流语言支持这样一种结构：能够从主流程分裂出一个子流程，并且可以和主流程并行运行（XPDL 就是支持这种结构）。

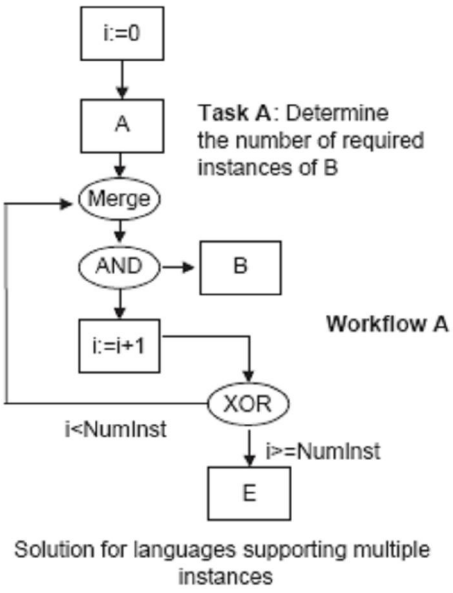

图 12-1

- **流程图**　在图 12-2 中，S1、S2、S3 是图 8-3 所示的子流程，这三个子流程的执行方式都是“Asynchronization”。活动 A 运行完成后并行激活 S1、S2、S3 三个子流程（实际上是同一个子流程的三个实例），同时活动 B 也被激活。子流程和主流程之间并行执行，相互独立，主流程结束了子流程仍然运行；而三个子流程之间也是互相独立的。

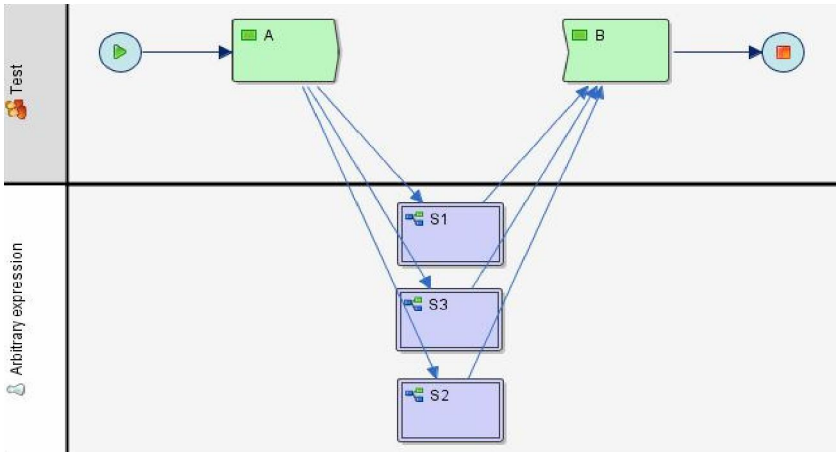

图 12-2

根据图 12-1，在 XPDL 中实现了一个在运行时设置实例个数的无同步多实例模式（如图 12-3 所示）。不再详细说明。

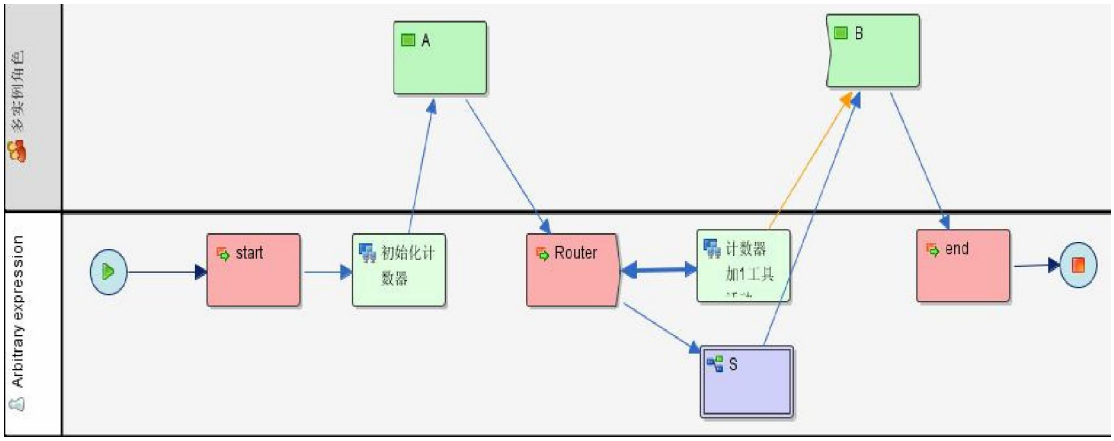

图 12-3

以下三种模式都是同步多实例模式。

Pattern 13 设计时确定的多实例(MI with a Priori Design Time Knowledge)

- Ø **描　述**　一个活动可以激活多次产生多个实例。而产生的实例的个数在流程设计时就事先知道了。一旦所有的实例都执行完成，就会激活其他活动。
- Ø **实　现**　如果设计时就确定了多实例的个数，最简单的实现方法就是把后续活动复制到每个分支后边就可以了（如图 8-1 所示）。在所有实例完成后，采用一个标准的同步模式（Pattern 3）就可以把所有实例同步。
- Ø **流程图**　在图 13 中，S1、S2、S3 是图 8-3 所示的子流程，这三个子流程的执行方式都是“ Synchronization ”。活动 A 运行完成后并行激活 S1、S2、S3 三个子流程，则主流程就会停下来，等待三个子流程全部都结束后，激活活动 B。

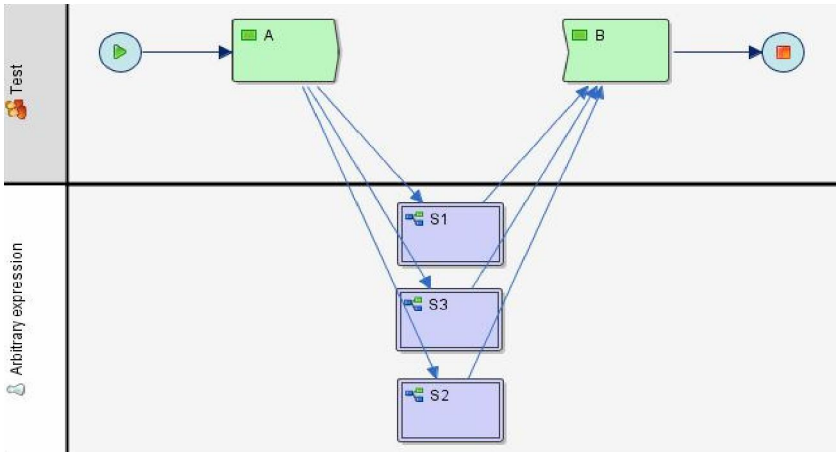

图 13

目前 XPDL 能够实现前 13 种模式，在后面提到的 7 种模式，XPDL 并没有实现，我们只是简单介绍一下这些模式的概念，不再详细探讨。

Pattern 14 执行时确定的多实例(MI with a Priori Runtime Knowledge)

- Ø **描　述**　一个活动可以激活多次产生多个实例。而产生的实例的个数是变化的，取决于实例的特点或者可用资源数目，但是在流程执行过程的某个时期，在这个活动的实例产生以前，要产生的实例个数是能确定的。所有的实例都运行完成后，激活后续活动。
- Ø **例　子**　处理一个订单，订单中有多本书，要分别检查每一本都有库存，所有的书都检查完成后才开始进入送货。
- Ø **问　题**　目前只有很少数工作流引擎支持该模式。
- Ø **实　现**　在图 14-1 中，直接提供了一个结构 Bundle 来实现该模式。图 14-2 中，产生的多实例不是并行执行的，而是按某一固定顺序执行。总的来说，该模式实现非常困难。

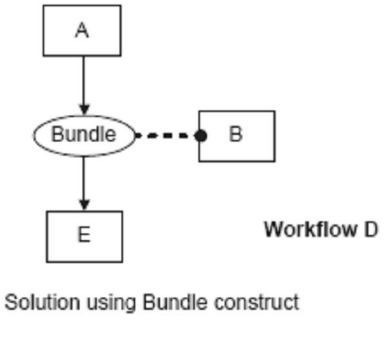

图 14-1

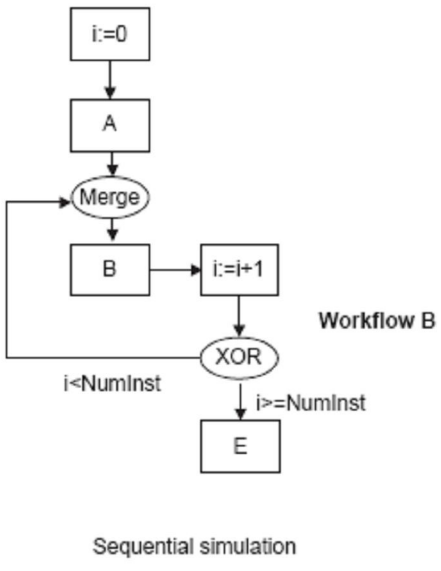

图 14-2

Ø **流程图** XPDL 不支持此模式。

Pattern 15 执行时不确定的多实例(MI without a Priori Runtime Knowledge)

Ø **描 述** 一个活动可以激活多次产生多个实例。而产生的实例的个数在设计流程时既不知道，在运行时也不知道。所有的实例都运行完成后，激活后续活动。该模式与 Pattern 14 执行时确定的多实例的区别是：在产生的实例执行时或者已经执行完时，仍然有新的实例产生。
Ø **例 子** 订购 100 台电脑，涉及多个供应商，但是每个供应商供应多少台电脑是不知道的，因此供应商的数量事先也不确定。但是当每次供应商送货后，就会将现在所拥有的电脑数量和所需的 100 台进行比较，来决定是否要下一个供应商继续送货。
Ø **问 题** 事先确定的多实例就好比 for 循环，而事先不确定的多实例就好比 while 循环。
Ø **实 现** 该模式实现非常困难。
Ø **流程图** XPDL 不支持此模式。

2.5 状态模式(State-based Patterns)

在本节中提到的三个模式都是基于状态的模式，而工作流产品实现这类模式是非常困难的。

Pattern 16 延迟选择(Deferred Choice)

Ø **描 述** 流程在某个点可以有多个分支进行选择。与 XOR-Split（独占式选择）模式相比，不是基于简单的数据或者决定就可以很明显地做出选择，而是它会向系统或者执行环境提供多种可选择的分支；但是这又不同于 AND-Split 模式，延迟选择只能选择一个分支执行，一旦选中了其中的一个分支，那么其它分支就会被撤消。这种延迟会一直持续到第一个选择分支开始实际运行。
Ø **同义词** Implict choice, deferred XOR-split
Ø **例 子** 将收到的一批商品运送到各个部门，到底选择什么样的运送方式，要看资源的可用性。
Ø **问 题** 大多数工作流系统只支持 Pattern 4 定义的独占式选择，而不支持延迟选择。
Ø **流程图** XPDL 不支持此模式。

Pattern 17 交叉存取并行路由(Interleaved Parallel Routing)

Ø **描 述** 该模式叫“任意顺序”更准确些。有几个活动它们是按顺序执行的，但是它们执行的顺序又是任意的，不会出现并行运行的情况。
Ø **同义词** Unordered sequence
Ø **例 子** 体检一般分为常规检查和血液检查，哪个在先哪个在后都可以，但是不可能同时检查。

Ø **问　题**　大多数工作流系统只支持并行分支和同步。

Ø **实　现**　在图 17-1 中，Workflow A 描述了任意顺序的功能；Workflow B 是事先将所有可能的顺序都列举出来，但是太复杂而且很难决定选择哪一个顺序；Workflow C 是用 Pattern 16 延迟模式实现，也太复杂。在图 17-2 中，是任意顺序的 Petri 网描述。

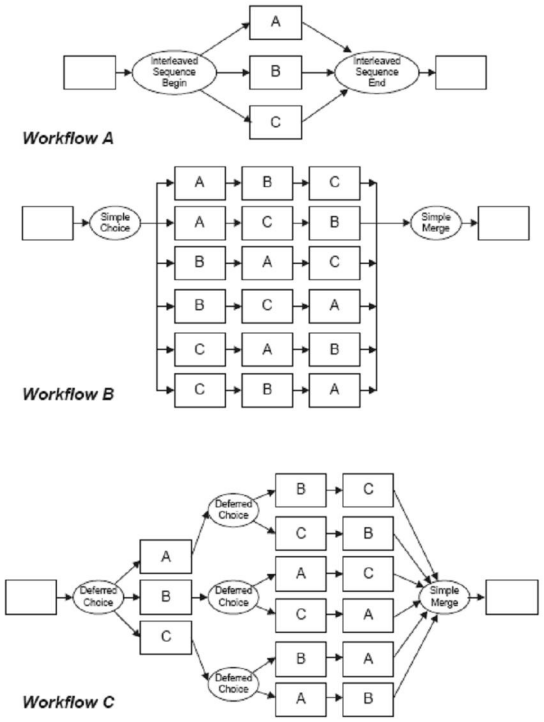

图 17-1

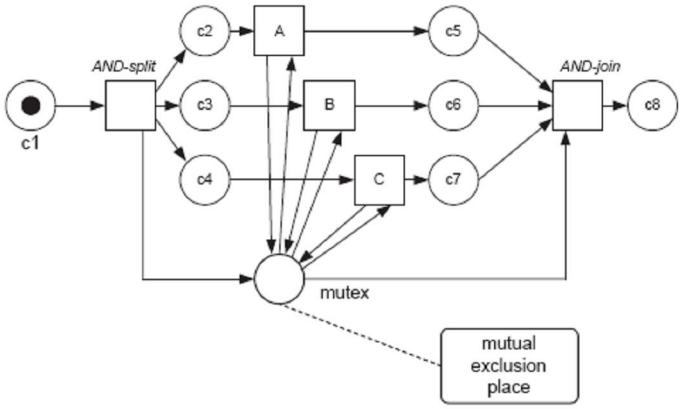

图 17-2

Ø **流程图**　XPDL 不支持此模式。

Pattern 18 转折点(Milestone)

Ø **描　述**　一个活动的使能需要一个具体的状态，只有当流程到达某个转折点的时候，这个活动才能使能。比如，有三个活动 A、B 和 C，A 只有在 B 执行完成且 C 还未执行的时候才能执行；B 执行前或者 C 执行后，A 都不能使能。

Ø **同义词**　Deadline, withdraw message

Ø **例　子**　客户在确定交付的前两天是可以取消订单的。

- **问　题**　大多数工作流系统都不支持这种模式。
- **实　现**　在图 18 中，是一个转折点模式的 Petri 网描述。注意：变迁 A 并不从库所 M 中取出 token，它只是用来测试 token 是否存在。

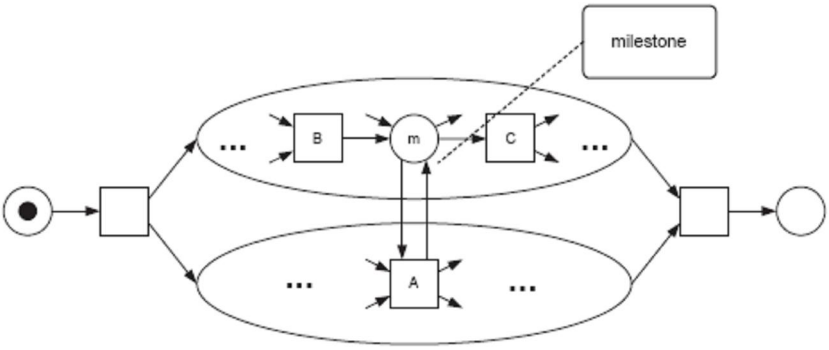

图 18

- **流程图**　XPDL 不支持此模式。

2.6 取消模式(Cancellation Patterns)

Pattern 19 取消活动(Cancel Activity)

- **描　述**　将一个处于使能状态（enabled）的活动变成不可用（disabled）。也就是将一个正在等待运行的活动移走。
- **同义词**　Withdraw activity
- **例　子**　一个顾客在网上订购书，已经下了订单，“支付货款”活动在等待运行，这时顾客取消了订单，那么相应的“支付货款”活动也要取消。
- **问　题**　只有很少的工作流语言支持取消活动。
- **实　现**　许多工作流管理系统可以通过调用 API，将活动的切入点从数据库中移除来实现活动的撤消。
- **流程图**　XPDL 不支持此模式。

Pattern 20 取消实例(Cancel Case)

- **描　述**　如果一个活动产生了多实例，那么仅仅撤消这个活动是不行的，要将这个活动的所有后代（实例）都移除才行。
- **同义词**　Withdraw case
- **问　题**　大多数工作流语言不支持取消实例。
- **实　现**　像 Pattern 19 一样，许多工作流管理系统可以通过调用 API，将活动的所有实例从数据库中移除。
- **流程图**　XPDL 不支持此模式。

3 参考文献

[1] W.M.P van der Aalst,et. Workflow Patterns. http://www.workflowpatterns.com/
[2]**胡长城. 工作流模型分析**. http://www.javafox.org/

4 研究机构

[1] W.M.P van der Aalst, Department of Technology Management, Eindhoven University of Tcchnology, The Netherlands. **（荷兰埃因霍温科技大学）**
http://w3.tue.nl/en/
[2] School of Information System, Distributed Systems Technology, The University of Queensland. Australia. **（澳大利亚昆士兰大学）**
http://www.uq.edu.au/
[3] http://www.workflowpatterns.com/

注：由于个人水平有限，对其中个别模式的理解可能存在问题，希望大家批评指正。