摘 要

本文基于主动数据库技术的基本思想并结合面向对象的思想对主动数据库的知识模型和执行模型进行了研究及设计。重点研究了主动规则和执行模型。

在主动规则中，本文在对主动规则分析的基础上针对其中的事件检测论述了一个基于染色 Petri 网的复合事件检测的方法。描述了用于检测复合事件的 Petri 网的定义 Petri-DS，其中用点模拟事件，变迁模拟对组成复合事件的成员事件的各种限制，不同的复合事件对应的守卫表达式不同。本文还给出了该方法的实现。

在执行模型中，本文在对传统执行模型的研究的基础上分析了常见的并发控制协议：两段锁协议、多版本协议和多版本两段锁协议。并对传统的多版本两段锁 MV2PL 协议进行了优化，提出了一种更适合主动数据库的事务执行方法。文中分别对只读事务和更新事务进行了优化。在对只读事务的优化中，降低了查询要求的一致性，使只读事务可以读到更新的数据版本。在对更新事务的优化中，将更新事务细分为普通事务和主动事务。对主动事务的读操作加入了检测是否存在未提交数据版本的函数。通过优化，查询操作可以读到比原来 MV2PL 协议下更新的数据版本。

关键词：ECA 规则，Petri 网，执行模型，MV2PL

Abstract

This paper studied and designed the Knowledge Model and the Excute Model based on the Active Database, combined with the Artificial Intelligence and Object Orient, especially on the Active Rule and the Excute Model.

In the aspect of Active Rule ,this paper described a measure of composite event detection based on Colored Petri-net on analysis of the Active Rule. The paper described the definition of the Petri-DS. The place simulated the event and the transition simulated the limitation put on the primitive events to make up of the composite event. Different composite events have different guard expression. Then the paper discussed the implementation of this measure.

In the aspect of Excute Model , this paper analyzed usual concurrency control protocol : Two-Phase Locking, Multiversion Concurrency Control and Multiversion Two-phase Locking Protocol, optimized conventional MV2PL protocol ,and provided a transaciton model of Active Database. The model optimized the read-only transaction and the update transaction separately. In the optimization of read-only transaction, we decreased the data consistency, thus the read-only transaction could read newer data versions. In the optimization of update transaction,we added a function to the read operation of active transaction to check whether there existed an uncommited data version. After optimization , read operation could get newer data version than MV2PL.

Keywords: ECA rule , Petri-net ,Excution Model , MV2PL

第一章 绪论

数据库技术自上世纪60年代后期问世以来，经历了一个飞速发展的过程。70年代广为流行的是网状模型和层次模型的数据库系统。自80年代以来，由于关系模型有严格的数学基础、概念简单清晰、非过程化程度高、数据独立性强，关系型数据库系统得到飞速发展[36]。随着数据库技术的发展，数据库应用领域已从传统的商务数据处理扩展到许多新的应用领域，关系数据库管理系统很难适应这些新应用领域的应用需求。新的应用需求的主要特征是：

1 数据模型要能描述复杂对象，能表达更丰富的语义。

2 数据类型从单一的结构化数据扩展为多媒体（图象、图形、声音）等半结构化和非结构化的多种类型。

3 支持长事务、版本管理和动态模式修改等。

4 具有各种监视和报警功能，能够主动处理异常情况，选择适当的干预措施，具有自动恢复能力。

尽管传统数据库已经得到了广泛的应用，取得了巨大的成功，但是在实践中，人们也发现了传统数据库是一种典型的“数据支持”服务，其提供的服务完全是被动的，在实际的应用中，如各种过程控制系统和军事系统或先进武器装备中，存在着许多主动性需求，这些系统经常要求其中的应用软件具有高度的“实时性”、“交互性”、“安全性”、“可靠性”，要求数据库具有各种监视和报警功能，能够主动处理异常情况，选择适当的干预措施，具有自动恢复能力。传统数据库系统提供的功能已经无法满足这些需求，因此要研究一种新型数据库来满足这些需求。

主动数据库[25]是数据库技术和人工智能结合的新技术，主动数据库（Active Database）能够主动对事件作出反应。它支持事件－条件－动作(Event-Condition-Action)规则，通过事件监视器监视如数据库事务、时间事件、外部信号等事件的发生，执行规定的动作，使得数据库能够提供主动服务的功能。与传统数据库特别是关系数据库拥有坚实的理论基础和丰富的实际原型相比，主动数据库需要解决大量的新的技术问题，一方面是构造数据库的模型、规则执行模型等问题，另一方面是设计体系结构、条件检测、事务调度等实现的技术问题。

1.1 主动数据库的发展现状

主动数据库满足了人们实践中的多样性需求。随着计算机应用范围的不断扩

大和深入，数据库在各种领域中起着越来越重要的作用，从基本的数据存储，到状态监控、安全监控、故障监控，再到军事或民用部门的协同和工作，以及各种管理信息系统(MIS)和决策支持系统(DSS)，人们对系统中数据库主动性功能要求也越来越高，随着技术的进步，主动数据库也将和其它新一代数据库技术一样得到更大的发展。

软件的主动性概念并非一个全新的概念。在70年代中期就开始设计的Ada语言中，异常处理措施就是一种能根据程序执行中异常的发生而主动触发某些预先设定动作的编程措施。传统数据库系统中的完整性和一致性约束的检查也是主动进行的。从文献上看，“主动数据库”术语出现在80年代初，80年代中后期涉及主动数据库和将产生式（或规则）库容入数据库的论文大量出现。由于大多采用规则库来实现主动性，主动数据库系统又称为“带规则（或产生式）的数据库系统”。关于主动数据库管理系统的研究应该特别提及几个项目：

ETM　　德国卡什鲁研究所（FZI Karlsruhe）在一个CAD OODBMS中为了完整性控制而设计的一个“事件一动作触发器”。

HiPAC　　维斯康新大学（CCA/XAIT，U.Wisconsin）为一个OODBMS开发的，具有“事件一条件一动作规则”（即ECA规则）和事件约束处理功能。

Postgres　　是加州波克莱分校（UC Berkley）在关系型DBMS基础上扩充“条件一动作库”后形成的系统。

Alert　　是IBM公司设计的，可以把一个被动的DBMS变换成一个主动DBMS的一种分层的体系结构，Starburst就是采用这种结构由关系型DBMS作较小的修改和扩充而成的。

CPLEX　　是哈佛大学的一个带有持久性的面向对象的程序设计语言。

RUBIS　　是巴黎大学的一个带有ECA规则触发器的DBMS。

ODE　　是AT&T公司贝尔实验室在持久性C++中增加了约束和触发器后形成的系统。

O_2　　是Altair在OODBMS中增加规则而成。

ATM　　是DEC公司在CRL开发的一个采用规则来组织长期活动的系统。

SAMOS　　是苏黎世大学开发的把ECA规则和OODBMS集成在一起的一个系统。

从上面一些的主动数据库介绍来看，主动数据库的一个突出思想是让数据库系统具有各种主动进行服务的功能，并以一种统一而方便的机制来实现各种主动性需求。统一的机制要求把主动性功能用一种统一的方法与原有的数据库功能集成在一个数据库系统中。目前，这种机制主要通过将一些规则预先嵌入数据库系

统的方法来实现。系统中提供一个自动"监视"模块，它主动地不时检查着这些规则中包含的各种事件是否已发生，一旦发现某事件发生时，就主动触发执行某个动作。

不过，已实现这些带主动功能的数据库管理系统中仍存在两方面的主要缺陷：

1 触发事件的指明方式局限性很大，不能提供用户一种自由构造和设置自己所需事件的机制，更没有从简单事件构造复杂事件的能力。

2 这些主动性设施缺乏一般性和统一性，它基本上是依附在某种特定系统上的。要弥补上述缺陷就要对主动数据库的各个部分进行深入的探讨和研究，主动规则和执行模型是主动数据库中的核心部分，本文将以这两个问题为中心展开讨论。

1.2 本文的工作和组织

本文重点讨论主动数据库的主动规则和执行模型，对主动规则中的事件检测论述了一种基于染色 Petri 网的复合事件的检测方法并给出其实现。对传统的多版本两段锁（MV2PL）协议进行改进，提出一种更适合主动数据库的事务模型。本论文研究的工作由以下几部分构成：

第二章简要论述主动数据库的主要特征及体系结构；

第三章论述了主动数据库的主动规则系统；

第四章针对主动规则中的事件检测论述了一种基于染色 Petri 网的复合事件检测方法并给出了该方法的实现；

第五章分析与研究执行模型；

第六章分析主动数据库事务的特征，并对传统的 MV2PL 协议进行改进，提出了一种更适合主动数据库的事务模型；

第七章总结和展望。

第二章　主动数据库概述

主动数据库是相对于传统数据库的被动性而言的，传统数据库及其管理系统是一个被动的系统，它只能被动地按照用户给出的明确请求执行相应的数据库操作，完成某个应用事务。用户是主动请求者，数据库是被动的执行者。但在许多实际领域的应用中，常常希望数据库在紧急情况下能根据数据库当前状态，主动、适时地做出反应，执行某些操作，向用户提供有关的信息。

传统数据库对这类应用满足甚少，只能用在更新数据库的应用程序中加入对紧急情况的检测的方法，或者由开发人员编写一段程序，用周期性查询数据库的方法来检测条件是否满足。这两种方法都不是系统化的好方法。

近年来，人工智能技术、面向对象的程序设计风范与数据库技术相互结合、相互渗透，为主动数据库的研究和开发创造了条件，使之成为数据库领域中一个活跃的研究领域。人工智能中的产生式系统已经发展成熟，其中产生式规则的一些研究成果可以适当地应用到主动规则当中。面向对象思想使我们可以把主动规则也作为对象考虑，这样便于的对规则进行管理和调用。

2.1　主动数据库功能模块

要使数据库具有各种主动功能，就需要对传统数据库添加支持主动规则的功能模块。大多数主动数据库系统都采用 ECA 规则作为主动规则。ECA 规则的含义是：当某一事件（Event）发生后引发数据库管理系统去检测数据库当前状态，看是否满足设定的条件（Condition），若条件满足，便触发规定动作（Action）的执行。

如图 2.1 所示，为了支持 ECA 规则，主动数据库数据库包括以下几个功能模块：

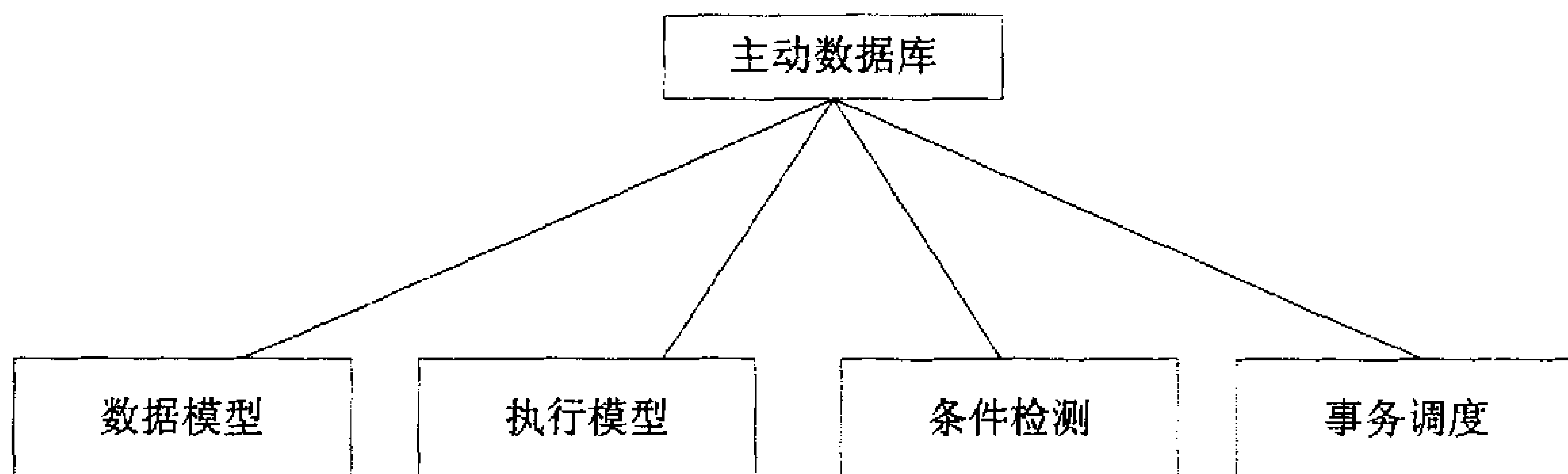


图 2.1 主动数据库功能模块图

数据模型在传统的数据库中主要是指层次模型、网状模型和关系模型等描述和处理实体间联系的方法，而在主动数据库中主要是指知识模型；执行模型是指处理和执行主动规则的方式；条件检测是指如何检测规则的条件；事务调度是指如何控制事务执行的次序，使数据库状态满足完整性、一致性等要求；下面具体论述这些部分的细节。

2.1.1 知识模型

知识模型[25]指在主动数据库管理系统中描述、存储、管理 ECA 规则的方法。为此必须扩充传统的数据模型，使它能支持 ECA 规则的定义、操作，并且保证规则本身的一致性。此外，知识模型还应支持有关时间的约束条件。

传统数据库中，数据模型的描述能力有限，尽管为了实现完整性约束引入了触发器机制，但触发器和主动数据库的主动规则相比表达能力低，只能描述“单个关系的更新”这类事件，而且不区分事件和条件。条件的检查，动作的执行总是在触发后立即执行或事务提交前执行，执行方式简单。所以，主动数据库必须扩充传统的数据模型，增加支持规则部分的知识模型。

在结合面向对象的主动数据库中，为了实现基于面向对象的数据模型，一些主动数据库系统采用面向对象的数据模型，用对象统一描述数据和规则，例如 HiPAC 系统中，规则就是一个实体类，每一条规则是该类中的一个对象实例。在规则对象上可以方便地定义 create，delete，activate，fire 等操作。事件、条件、动作作为规则的构成成分本身又可以是对象。例如条件，作为一个对象又可以定义其耦合方式。该模型具有很强的表达能力。

2.1.2 执行模型

执行模型指 ECA 规则的处理、执行方式，包括 ECA 规则中的事件－条件、条件－动作之间的各种耦合方式及语义描述，规则的动作和用户事务的关系。执行模型是对传统数据库的事务模型的扩充和发展。在主动数据库研究中提出了立即式、延迟式和分离式等多种多样的执行 ECA 规则的方式。丰富多样的执行模型使用户可以灵活地定义主动数据库的行为，克服了数据库管理系统中触发器只能顺序执行其规则的不足。

主动规则的执行分为四个阶段：

1 信号通知阶段指事件源引起事件发生的现象。

2 规则触发阶段，产生事件（包括复合事件）并触发相应的规则。规则和与之相关的事件形成了规则实例。

3 评估阶段对被触发规则的条件进行评估。那些条件评估成功的所有规则实例形成了规则冲突集。

4 规则调度阶段指对规则冲突集进行处理，选出下面将要执行的规则执行其动作。动作执行时可能产生规则的级联触发。

2.1.3 条件检测

主动数据库中条件检测是系统实现的关键技术之一，主动数据库中条件复杂，可以是动态的条件、多重条件、交叉条件。所谓交叉条件可以互相覆盖，即其中某些子条件可以属于其它主条件，因此高效地对条件求值是系统的目标之一。

2.1.4 事务调度

一般地，事务调度是指如何控制事务的执行次序，使得事务满足一定的约束条件。在传统的 DBMS 中并发事务的调度应该满足可串行化要求以保证数据库的一致性。

在主动数据库中，对事务的调度不仅要满足并发环境下的可串行化要求而且要满足对事务时间方面的要求。例如事务中操作的开始时间、终止时间、所需的执行时间等。要同时满足两方面要求的调度是一个困难的技术问题。它要求综合传统数据库的并发控制技术和实时操作系统中与时间要求有关的调度技术。由于主动数据库执行模型的复杂性更增加了事务调度的难度。需要研究一种新的调度模型或框架，以此为基础来建立调度策略，调度算法。由于事务调度需要满足时间方面的要求，因此调度机制常常是时间的谓词，而对执行时间估计的代价模型同样是尚待解决的问题。

2.2 主动数据库的体系结构

主动数据库系统的体系结构[46]应该具有高度的模块性和灵活性。由于目前大部分主动数据库是在传统 DBMS 或面向对象数据库管理系统上研制的，其体系结构大多是扩充 DBMS 的事务管理部件、对象管理部件以支持执行模型和知识模型。此外还要增加：

事件检测部件：当事件发生时发出相应的信号。

条件检测部件：接收要检测的条件并进行优化求值。

规则管理部件：控制规则的执行。

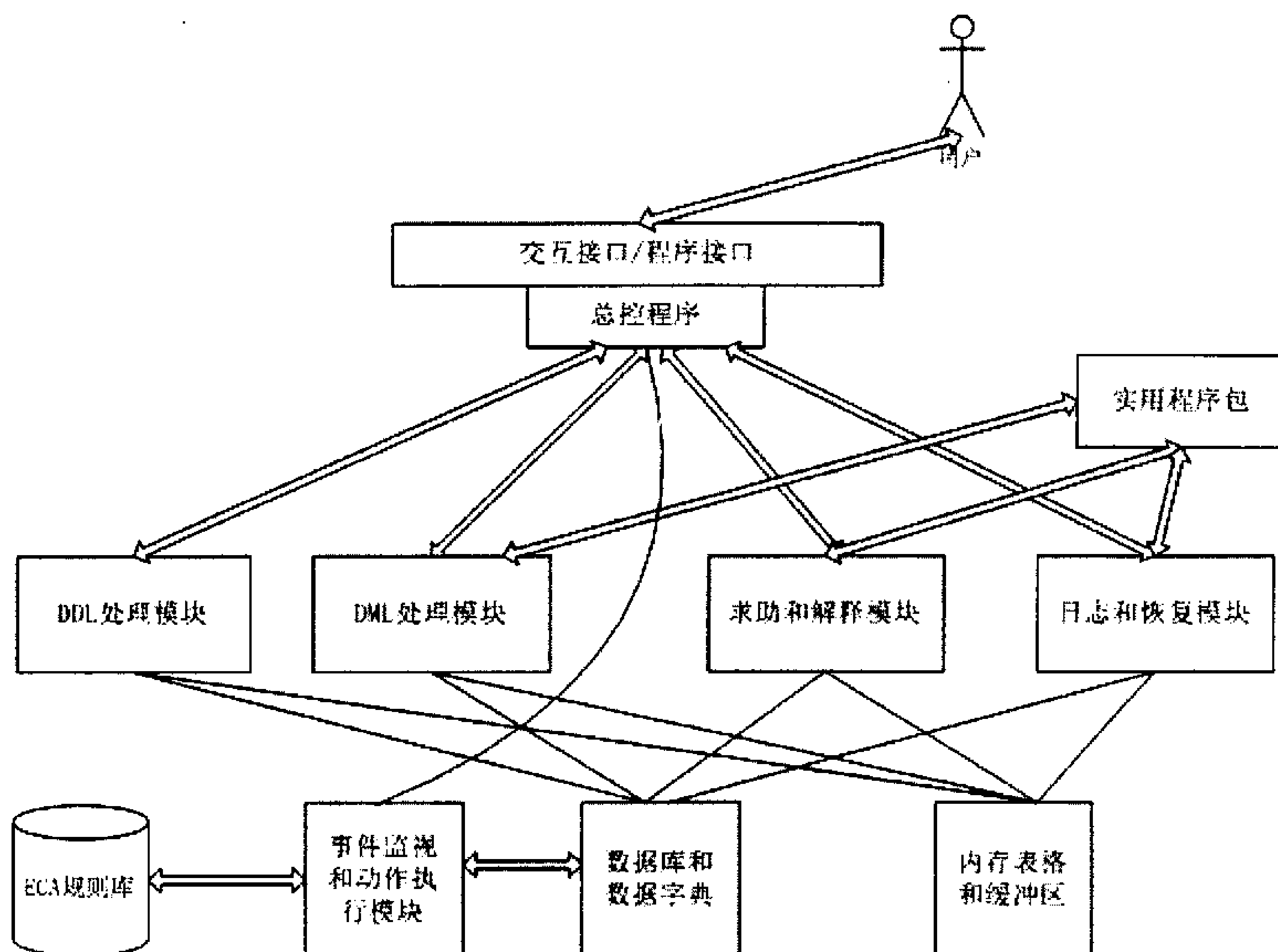


图 2.2 体系结构图

体系结构中的重要问题还有各部件之间接口的设计，尤其是事务管理部件和操作系统之间的接口。因为事务管理中对时间有要求的调度机制必须要有操作系统的有力支持。

体系结构的不同还影响到主动数据库的系统效率。系统效率是指对不同体系结构、算法运行效率的比较和评价。为了提高系统效率，需要把条件计算和动作执行从触发事务中分离出来。

2.3 主动数据库的一般模型

一般，一个主动数据库系统（aDBS）功能上是由一个传统数据库（DBS）和一个事件驱动的知识库，简称事件库（EB）及其相应的事件监视器（EM）组成[25]，可用公式表示成：

$$aDBS = DBS + EB + EM$$

其中 DBS 是传统数据库系统，用来存储数据和对数据进行维护、管理与运用。

事件库 EB 是一组由事件驱动的知识的集合，每一项知识表示在相应的事件发生时，如何来主动地执行其中包含的由用户预先设定的动作。

EM 是一个随时监视 EB 中的事件是否已经发生的监视模块，一旦监视到某个事件已经发生时就主动地触发规则，按 EB 中指明地相应知识执行其中预先设定的动作。

可见，根据事件库中知识表示形式的不同，将获得各种不同的主动行为，在目前的实现中往往采取“事件”驱动的“条件一动作”的主动规则来表示这种知识。每条主动规则都指明什么事件发生时，在什么条件下，执行什么样的动作；

“事件”驱动的“条件一动作”规则可具有如下的一般形式：

RULE 〈规则名〉[(〈参数〉，…)]

WHEN 〈事件表达式〉

IF 〈条件 1〉THEN〈动作 1〉

……

IF 〈条件 n〉 THEN〈动作 n〉

END-RULE[〈规则名〉]

〈事件表达式〉可是是基本事件也可以是复合事件；

〈条件 i〉 (i=1，2，…，n)是某种逻辑（例如模糊逻辑）中任意的一个合法逻辑公式；

〈动作 i〉(i=1，2，…，n)既可以是系统预先定义的一些标准动作，也可以是用户定义的一个动作，或是用某种语言编写的一个过程。

上述“事件”驱动的“条件一动作”规则的语义是：一旦〈事件〉发生，计算机就主动触发执行其后的 IF—THEN 规则。即如果〈条件 1〉为真，则执行其后的〈动作 1〉，并且接着逐个检查下一个 IF—THEN 规则，直至执行完为止。注意，这区别于一般程序设计语言的 case 语句，在那里只执行第一个使〈条件〉为真的〈动作〉。

根据实际需要，也可以把上述事件驱动规则的语义定义为，一旦<事件>发生时就主动触发执行其后的规则集合（规则库），按产生式系统执行的模式，（即匹配一解决冲突一执行循环）来解释执行相应的规则集合。

由上可见，在一个主动数据库中，一方面包含了称为“被动数据”的一个传统数据库，另一方面包含了成为“主动规则”的一个能根据事件的发生主动激活执行的事件库。这些主动规则受系统中一个“事件监视器”的监视控制，该事件监视器主动地时刻监视这事件库。这样，用户可以通过设置（或编制）各种不同的事件驱动规则，以一种统一的机制来实现前面所述的多种主动处理功能，从而能满足各种客观需求，它的应用是十分广泛的。

第三章　主动数据库中的主动规则

主动数据库一般是在传统数据库中嵌入“事件”驱动的“条件－动作”，也叫事件－条件－动作（Event－Condition－Action）规则，它是由人工智能中的产生式规则发展而来的，ECA 规则在数据库中定义并存储。这样，规则可以被多个应用程序共享，数据库系统也可以优化规则的执行。

产生式规则范例源于人工智能（AI）和专家系统研究领域。在 AI 系统中，产生式规则通常有如下形式：

condition→action

推理引擎循环地将工作存储器中的数据和规则的条件部分相匹配。对所有已经匹配的规则（规则冲突集），使用冲突解决策略选出一条规则点燃，并执行规则的动作部分。动作部分可以修改工作存储器，循环直到没有可以匹配的规则为止。

这种范例归纳成适用于主动数据库的 ECA 规则，有这样的形式：

on event
if condition
then action

可触发规则的事件可以是数据库操作、出现某个数据库状态和状态的变迁，与产生式规则不同的是主动数据库的规则不去用推理引擎对所有规则进行周期性的循环来评估。当触发规则的事件发生时，评估规则的条件；如果条件满足，就执行动作。规则在数据库中定义和存储，由数据库系统进行条件评估，还要有认证，并发控制和错误恢复机制。

这种 ECA 规则可以强制实现多种数据库任务，如：实施完整性约束，实现触发器和警报，维护取出的数据，强制执行访问限制，贯彻版本控制策略，是一种功能很强的机制。下面具体介绍 ECA 规则中的事件、条件和动作。

3.1　主动规则的描述和定义

ECA 规则可以用事件、条件和动作描述，它的定义可以分成三个部分：事件的定义、条件的定义和动作的定义。

ECARule::=on 〈ECAEvent〉
if 〈ECACondition〉
then 〈ECAAction〉

3.1.1 事件

为了建立一个具有较完善功能地主动数据库系统，必须提供用户一种表达各种复杂事件的能力。事件表达式可以用基本事件和运算符构成事件代数，这大大扩充了事件的表达能力。

1. 基本事件

基本事件是在各种应用中常用的事件，系统将把这些基本事件作为系统预定义的事件，无需用户定义即可使用。

✧ 与时间有关的基本事件

1 事件 $time=t_0$，其中 t_0 表示一个绝对时间，可以是一个常数，也可以是用户或程序设置的变量；还可以是一个事件开始或终止的时间，或者是数据库事务开始、提交或放弃的时间。

2 事件 $time<t_0$，是一个区间事件，在 time 小于 t_0 时都发生。

3 事件 $time \geqq t_0$，是一个区间事件，在 time 大于等于 t_0 时都发生。

4 事件 $time \in [t_0, t_1)$，在 t_0 到 t_1 之间发生。

5 事件 $t_0 \theta t_1$，θ 是一个关系运算符，该事件在 t_0 和 t_1 满足 θ 关系时发生。

这类事件都与时间有关，可用来描述数据库中各种时间限制条件，在实时控制和事件同步等应用中使用。

✧ 与数据库状态有关的基本事件

1 当数据库的一个指定部分的内容 X 发生改变（包括插入、删除和更新等）时发生的事件 change(x)。

2 当数据库的一个指定部分的内容 X 被使用（包括被查询、搜索和读出等）时发生的事件 use(x)。

3 当数据库的一个指定部分的内容 X 处于某个特定状态（包括取特定值或它的值处于或超出某个特定范围）时，或与另一内容 Y 具有某种特定关系时发生的事件。

4 当数据库的一致性遭到破坏时发生的事件。

5 当数据库的容量大于某个限制时，或出现异常状态时发生的事件。

这类事件用来监视数据库的状态，实现一致性和完整性约束。

✧ 与数据库执行有关的基本事件

1 在执行数据库语言或数据处理语言的某个（种）语句时（或执行完成后）引发的事件。

2 当数据处理进入“死循环”时引发的事件。

这类事件用来实现处理的跟踪、自动审计、日志自动建立、例外处理或出错监控等。

✧ 与信号灯有关的基本事件

为了实现同一用户各模块之间或不同用户之间相互传递信息，在系统中设立一些“信号灯”，这里用 S 表示。

1 事件 $S=i$，其中 i 表示一个整数，该事件表示信号灯 S 等于 i 时发生的事件。

2 事件 $S\leq i$，表示信号灯 S 的值小于等于 i 时发生的事件。

3 事件 $S>i$，表示信号灯 S 的值大于 i 时发生的事件。

4 事件 $S_1\theta S_2$，其中 S_1，S_2 为两个信号灯，θ 是一个关系运算符。该事件表示信号灯 S_1，S_2 的值满足关系 θ 时发生的事件。

这些事件可以实现数据库应用程序的同步和通讯，从而实现一些复杂的应用系统。

✧ 与公共变量有关的基本事件

系统还可以设置一些公共变量（或全局变量）X_1，X_2，…X_n，这样包含这些公共变量的任一合法的逻辑表达式 $B(X_1, X_2, \ldots X_n)$都可以用来构成事件。

1 当 $B(X_1, X_2, \ldots, X_n)$为真时发生的事件。

2 当 $B(X_1, X_2, \ldots, X_n)$为假时发生的事件。

用户可以通过程序或键盘对公共变量赋值等办法来构造其需要的事件。

✧ 与机器中断有关的事件

1 键盘或鼠标等外部设备操作引发的各种中断事件。操作系统捕获这些事件，然后转给事件监视器处理。

2 当计算机的外接设备接收到外部信号时发生的事件。这也需要操作系统的帮助才能实现。

3 计算机的各种运算操作中产生的各种硬中断（如运算溢出）发生时的事件。

4 当计算机中可设置的各种陷阱中断发生时的事件。利用这类中断事件可以

灵活地实现对数据库中各种处理对象地实时控制、监视和处理等功能。

以上已经给出了几种基本事件，可以分别用在不同的场合。但是，用户有时往往需要更复杂的事件，它们是这些基本事件的组合，因此，还需要给出一种事件代数，以便用户根据需要从这些基本事件构造出各种丰富的复合事件。

2. 复合事件

为了构造更复杂的复合事件，需要用运算符将基本事件相连接，下面是各运算符以及常用复合事件的定义。

1 同时发生运算∧：两个事件 E_1、E_2，当事件 E_1 和 E_2 同时发生（无论次序如何），$E_1 \wedge E_2$ 才发生。

2 选择发生运算 | ：两个事件 E_1、E_2，有一个且仅有一个事件发生，则 $E_1|E_2$ 发生。

3 合并发生运算∨：两个事件 E_1、E_2，只要有一个事件发生 $E_1 \vee E_2$ 就发生。

4 不发生运算～：E 是一个事件，～(E)[E_1，E_2]表示在全闭区间[E_1，E_2]中事件 E 不发生。

5 相继发生运算(Sequence)：(E_1：E_2)表示事件 E_1 结束后马上发生 E_2 事件。

6 任何运算 Any：一种连接事件，Any(m，E_1，E_2，...，E_n)($m \leq n$)，指 n 个不同事件中的 m 个事件发生，此事件就发生。它与事件发生的相关次序无关。

7 之前发生运算<：设 E 是一个事件，<E 表示当事件 E 开始发生时就终止的一个事件。

8 之后发生运算>：设 E 是一个事件，>E 表示当事件 E 终止时就开始发生的一个事件。

9 周期运算(Period)：如果事件 E 在固定的时间间隔中反复出现，那么称 E 为周期事件。P(E_1，[t]，E_3)指事件 E_1，E_3 在时间间隔 t 发生，则 P 发生。此处时间 t 是确定的常数。

10 非周期运算(Aperiod)：指某事件 E 在两个事件的发生间隔中发生。A（E_1，E_2，E_3）中 E_1，E_2，E_3 为任意事件。每当 E_2 发生在 E_1 和 E_3 的发生间隔中，非周期复合事件 A 发生。此时 A 可能发生 0 次或多次（当 E_1，E_3 间隔中 E_2 不发生或 E_1，E_3 不存在间隔则 A 发生 0 次）。E_1，E_3 间隔为半开区间(E_1，E_3]。

以上所描述的事件运算符集及其定义的事件规约语言可以满足大部分应用的需求。周期与非周期运算符对流程控制、网络管理及计算机集成（CIM）的需求将予满足。

3. 事件的消费策略

在检测复合事件时，可能会有多个同类型事件同时发生。例如，考虑复合事件 CE，CE 为事件 E_1，E_2 的相继发生运算事件。如果事件 E_1 发生了两次，先是 E_1，后是 E_1'，而后又发生了事件 E_2，这就产生了一个问题：CE 选择哪两个事件组成复合事件？可能产生这样一些组合：(E_1; E_2)、(E_1'; E_2)或者(E_1; E_2)∪(E_1'; E_2)。这实际上是一个选择事件消费策略的问题。

以下将描述四种消费策略，根据不同的消费策略（上下文），系统将选择不同的复合事件。

Recent：将最近发生事件的集合组合成复合事件，按照 recent 策略，在上面的例子中，应选择(E_1'; E_2)作为检测到的复合事件。在此之后，在检测复合事件 CE 时，不再考虑 E_1'和 E_2。

Chronicle：以事件发生的顺序构造复合事件。按照 Chronicle 策略，E_2 发生时应选择 E_1 构造复合事件，即(E_1; E_2)。以后不再考虑用 E_1，E_2 构造复合事件 CE。

Continuous：定义一个滑动窗口，对每个基本事件都构成一个复合事件。即对 E_1 和 E_1'都和 E_2 进行组合，分别构成两个复合事件。当 E_2 发生时，将检测到两个复合事件。

Cumulative：一直积累基本事件直到最后生成复合事件，上面的例子，在 E_2 产生时将触发复合事件 CE，CE 的第一个事件参数包含两个事件 E_1 和 E_1'。

4. 事件粒度及作用

由于应用的需要，对事件的运用应采用灵活的手段。事件粒度指明了某事件的应用范围。事件粒度可分成三类：

1 集合（Set）：指该事件适用于集合中的所有对象（如一个类的所有实例）。

2 子集（Subset）：指该事件适用于集合中的部分对象。

3 成员（Member）：指该事件只适用于集合中的某个成员对象（如对某实例的访问予以授权）。

事件的作用（Role）指明了对于主动规则（ECA），事件是否必须显式给定。对于事件的作用也可分为三类：

1 可选（Optional）：指在 ECA 规则中，若没有规定事件，系统也将支持 CA 规则。

2 取消（None）：指系统只支持 CA 规则，不能规定事件。

3 强制（Mandatory）：指系统只支持 ECA 规则，必须规定事件。

综上所述，事件可以根据事件类型、事件的作用及粒度、消费策略等方面进行定义，具体定义如下：

〈ECAEvent〉::=〈EventType〉〈Role〉〈Granularity〉

〈EventType〉::=〈PrimitiveEvent〉|〈CompositeEvent〉

〈PrimitiveEvent〉::=〈TimeEvent〉|〈DBStatus〉|〈DBExecute〉|

〈SignalLamp〉|〈GlobalVar〉|〈i/oInterrupt〉

〈TimeEvent〉::=〈time=t_0〉|〈time<t_0〉|〈time≧t_0〉|

〈time∈[t_0, t_1)〉|〈$t_0 \theta t_1$〉（θ为一个关系运算符）

（和时间有关的事件）

〈DBStatus〉::=〈DataChange〉|〈DataUse〉|〈DataValue〉|

〈ViolateConstraint〉|〈DBException〉

（和数据库状态有关的事件）

〈DataChange〉::=〈DataInsert〉|〈DataDelete〉|〈DataUpdate〉

〈DataUse〉::=〈DataQuery〉|〈DataSearch〉|〈DataDeduce〉|

〈DataRead〉

〈DataValue〉::=〈$x>x_0$〉|〈$x<x_0$〉|〈$x \leqq x_0$〉|〈$x \geqq x_0$〉|

〈$x \in [x_1, x_2]$〉|〈$x\ \theta\ y$〉（θ同上）

〈ViolateConstraint〉::=IntegralityConstraint|ConsistencyConstrainst

〈DBExecute〉::=〈Transaction〉|UnlimitCircle

〈Transaction〉::=Begin | End | Commit | Abort

（和数据库执行状态有关的事件）

〈SignalLamp〉::=〈S=i〉|〈S≦i〉|〈S>i〉|〈$S_1\ \theta\ S_2$〉（S为信号灯，θ同上）

（和信号灯有关的事件）

〈GlobalVar〉::=〈B(X_1 ,X_2, …, X_n)〉（B为包含公共变量$X_1, X_2, \ldots, X_n$的任一逻辑表达式）

（和公共变量有关的事件）

〈i/oInterrupt〉::=Input|HardInterrupt

（和机器中断有关的事件）

〈CompositeEvent〉::=〈EventExpression〉

〈EventExpression〉::=〈ConsumePolicy〉〈Operator〉〈PrimitiveEvent〉|

〈ConsumePolicy〉〈PrimitiveEvent〉〈Operator〉

〈CompositeEvent〉|

〈PrimitiveEvent〉|

〈ConsumePolicy〉〈CompositeEvent〉〈Operator〉

〈CompositeEvent〉

〈ConsumePolicy〉::=Recent |Chronicle |Continuous |Cumulative

〈Operator〉::=∧|||∨|～|；|Any|<|>|Period| Aperiod

〈Role〉::= Mandatory|Optional|None

<Granulrity>::=Member|Subset|Set

3.1.2 条件

在ECA规则中，条件是可选的，就是说可以出现条件也可是省略条件。如果ECA规则中不出现条件，那么ECA规则就演化为EA规则。如果在一个主动数据库系统中事件和条件都是可选的，那么在定义规则时至少要保留事件和条件两个当中的一个。由于条件机制向主动规则和条件检测提供原语，因此条件机制在知识模型中起着关键的枢纽作用。主动规则的条件非常复杂，可以是动态的条件、多重条件和交叉条件。所谓交叉条件是指条件可以互相覆盖，即其中某些子条件可以属于其他主条件。这种条件的复杂性也体现出条件的相互依赖性。

主动规则的各部分是互相联系的，因此对条件的评估不应脱离规则的其它部分以及数据库状态。在处理某个主动规则时，必须和数据库状态相结合[1]。条件的评估环境（上下文）至少可分为以下四种：

DB_T：指数据库处于当前事务开始的状态；

DB_E：指数据库处于事件发生的状态；

DB_C：指数据库处于条件评估状态；

DB_A：指数据库处于动作执行状态。

主动规则系统支持规则中的条件去访问DB_T、DB_E和DB_C三种状态，并且可以访问从触发事件传输过来的绑定结果（$Bind_E$）。图3.1给出规则各部分可获得的信息。

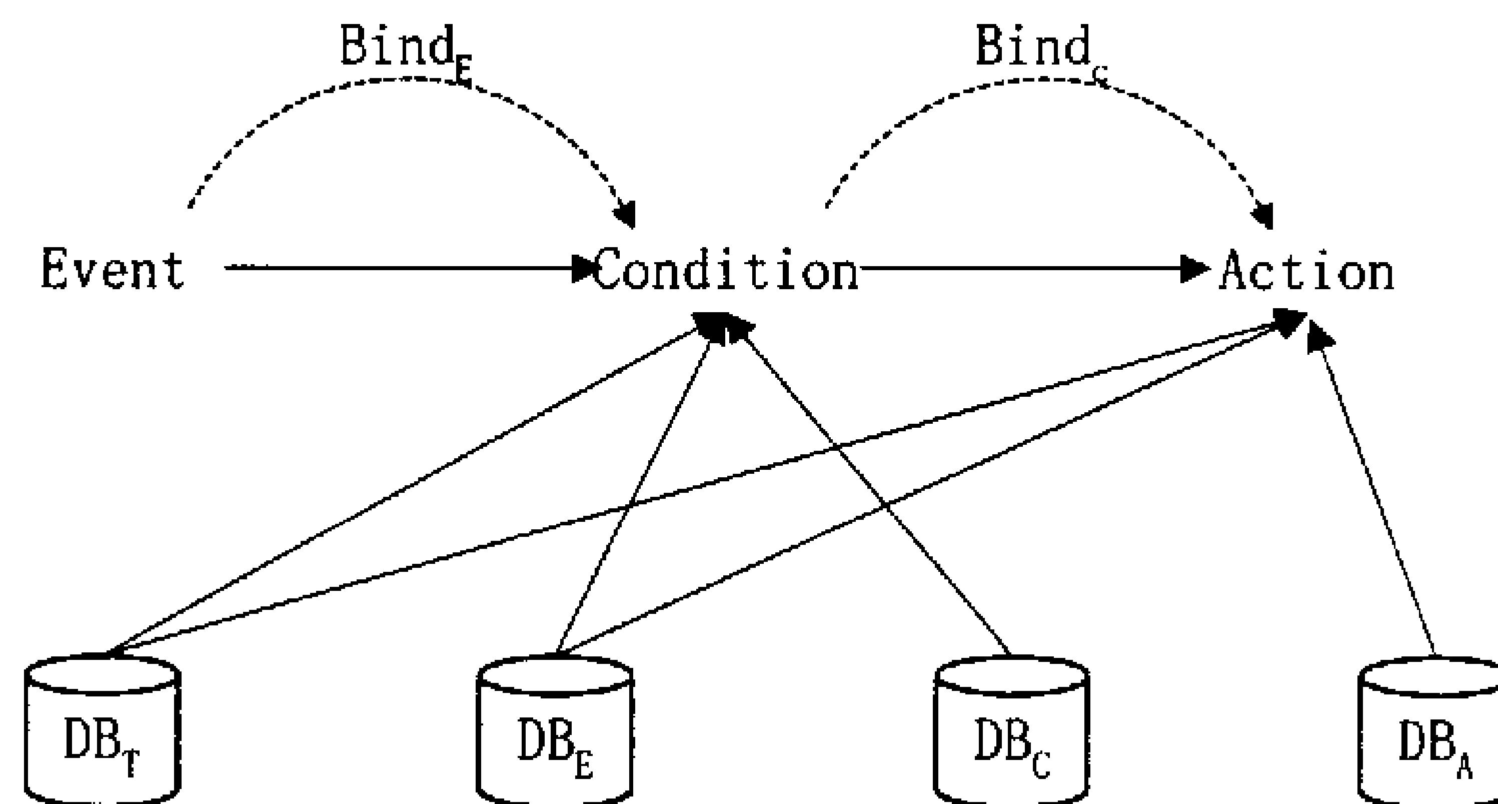


图3.1 规则的处理流程

实际上，由于在事件发生的前后数据库的各个状态并不相同，而且可能有多条规则被触发并且在执行某一单一动作时执行完结，因此实际的情形远比图中所表示的复杂。

主动规则的条件一般是谓词，它是关于数据库状态、状态变迁或关于时间的条件。条件可分为以下几类：

1 简单条件：只涉及单个数据对象。这种条件只与对象事件相关，易于评价，它相当于简单查询处理。

2 统计条件：涉及导出数据。

3 结构条件：与语义结构联系有关。

4 时间条件：一种时间限制。它总是与时间相关联，其评价需要系统的“识时”机制。

5 复杂条件：各种条件的布尔表达式，跨多个事务的条件，涉及数据经历而不是单个数据值的条件等。

由于条件自身的复杂性以及在ECA规则实现时要解决事件与条件之间及条件与动作之间的复杂匹配问题，因此在实时应用环境中常采用状态—动作（Situation-Action）模型。在S－A模型中，状态是事件和条件的统一，消除了E－C匹配。与事件的作用相似，条件的作用也可分为二类，它指明了是否必须给定条件。在ECA规则中，条件一般是可选的（Optional）；在事件驱动（Event-Action）的规则中，可以取消（None）条件。在事件和条件都可选的主动规则系统中，它总是要求事件和条件至少有一个必须给定。

综上所述，条件的具体定义如下：

〈ECACondition〉::=〈Boolean〉〈Role〉|〈Condition-Expression〉〈Role〉

〈Boolean〉::=True|False

〈Role〉::= Mandatory|Optional|None

〈Condition-Expression〉::=〈Value〉|

〈Value〉〈Operator〉〈Value〉|

〈Value〉〈Operator〉〈Condition-Expression〉

〈Value〉::=Number|Symbol

〈Operator〉::=>|<|≧|≦|OR|AND|NOT|IN

3.1.3 动作

动作是主动规则ECA的最后部分，也是主动规则的最终目的。主动数据库管理系统的核心思想是由系统“主动”负责对各种事件或状态作出响应，并执行预先设定好的动作，因此，主动数据库管理系统中的主动规则又可称为行为规则。动作不仅是主动规则的重要部分，而且动作的触发及其与条件的耦合关系对将来的执行模型也产生着重大的影响。

所谓动作就是一系列操作，这一系列操作是系统对事态作出评价后“主动”

发出的，它从效果上体现了系统的主动性。动作所执行的任务范围大致可分为以下几类：

1 Structure：指修改数据库或规则集结构的操作。

2 Behavior invocation：指调用数据库内部的方法。

3 External：指调用数据库外部的方法或功能。

4 Inform：指把当前的状态通知用户或系统管理员。

5 Abort：指放弃数据库的某个事务。

6 Do-instead：指用户自定义的某种操作替代将要执行的动作。

动作执行的环境（上下文）与条件评估的环境大致相同，同样指明了动作所能获得的信息。在 E-C-A 的一系列相接的评价中，有时信息可能作为数据库的事件状态（DB_E）或条件评价结果（$Bind_C$）从规则的条件部分传到动作部分。

综上所述，动作采用下面的形式进行定义：

<ECAAction>::=<Acts>| Do-Instead<Acts>

<Acts>::=<Operation>|<Operation><Acts>

<Operation>::=<Structural>|<Behavior>|<Inform>|< Transaction>

<Structural>::= Create|Drop|Insert|Delete|select

<Behavior>::=Command|Procedure|Function

<Inform>::=Message

<Transaction>::= Begin|End|Commit|Abort

3.2 主动机制的实现途径

想要将主动规则变成切实可行的可用之物，需要对传统数据库添加：事件发生时发出相应的信号的事件检测部件、接收要检测的条件并进行优化求值的条件检测部件和控制规则的执行的规则管理部件。图 3.2 为主动规则系统结构图。

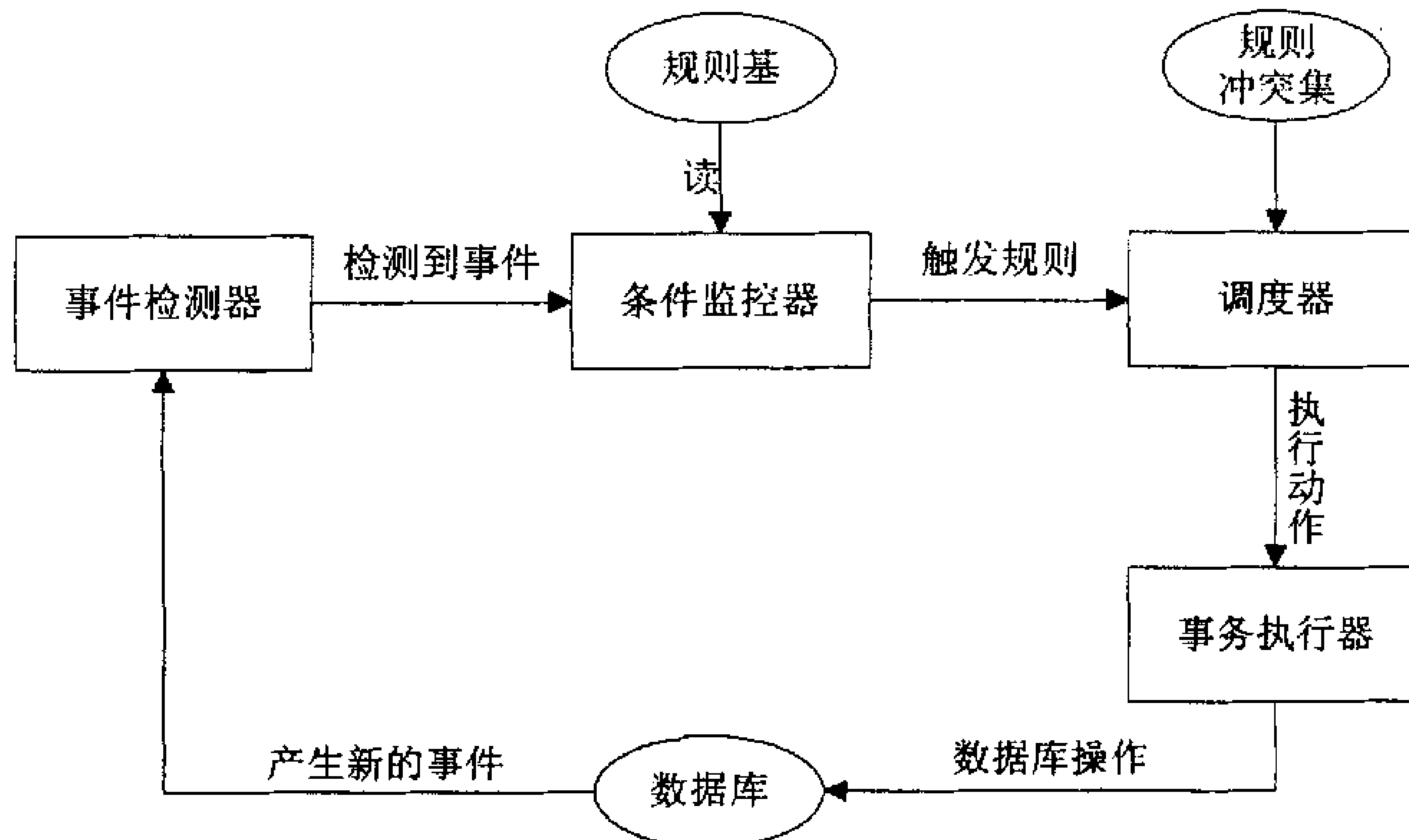


图 3.2 主动规则系统结构图

上面各个主动组件的功能和主动数据库的知识模型、执行模型有很大关系，很大程度上受到主动数据库开发环境的影响。总的来说，主动组件和数据库有两类构造关系：

层次型：主动组件作为被动数据库系统上一层的软件出现。这种方法的优点是不需要修改被动数据库，生成的主动数据库便于移植。该方法的缺点是：缺乏对底层数据库内核的访问，会影响性能并且降低了检测基本事件的能力。

集成型：通过修改被动数据库系统的源代码开发主动组件。这种方法突破了层次型方法的诸多限制，是这两种方法种更为优秀的一个。下面就上述功能的两个主要部分事件的检测和条件的监测进行论述。

3.2.1 事件的检测

主动数据库系统中用事件监视器来检测规则系统感兴趣的事件是否发生。事件检测的实现主要包括两个方面：对基本事件的检测和对可以构成复合事件累积事件信息的检测。

基本事件的检测通常要对数据库系统内核做内部检查。总的来说，有三种方法：

1 在应用程序中添加发送“事件信号”消息的功能；

2 系统对用户的应用程序进行扩展，应用程序触发什么事件就对该程序添加一段程序使它能够向事件监视器发送“事件信号”；

3 在数据库系统中放置可发送“事件信号”的代码，这种方法可以检测事件的种类多，性能超过了前两种方法。它需要修改数据库系统内核，只能采用集成

型的主动组件构成方式。

复合事件则需要将当前发生的基本事件和以往发生的基本事件信息组合在一起，构造成复合事件。复合事件的检测需要一定的算法支持，本文将在下一章中论述复合事件的检测方法。

3.2.2 条件的监控

条件的监控由主动数据库中的条件监控器进行处理。在一些主动数据库中，事件和条件可以组合起来描述了ECA规则中的状态（Situation）。实际上，事件的检测和条件监控两者之间存在这密切的联系。事件监视器对条件监控器中的处理过程进行初始化，并从事件监视器将已经发生事件的信息传递到条件事件器中。事件监视器越复杂，传递给条件事件器的信息就越丰富。例如，对Student表进行插入数据（Insert）的操作引发的基本事件，只要将被插入的元组传给条件监控器就行了。而如果事件是复合事件，则和事件相关的信息也会很复杂。例如，在一个用 “同时发生运算”（也就是“与”运算）构成的复合事件中，需要向条件监控器提供所有构成该复合事件的所有基本事件的信息。

条件的监控是对指对那些被检测到事件已经发生了的规则进行条件评估的处理过程。条件检测采用下面的循环进行处理：

```
match
while (conflict set not empty) do
conflict   resolution
act
match
end-while
```

该循环中，匹配(Match)阶段确定在当前数据库状态下哪些规则的条件为真，然后把它加入冲突集（冲突集实际上是等待点燃的已经触发规则）。冲突解决（Conflict Resolution）这一步是要从冲突集中选出一条规则，并进一步处理该规则的动作部分。

在匹配阶段中，原始的评估方法是对每条规则的条件都进行评估，找出下一步要处理的规则。这种方法开销很大，也没有必要，因为数据库中只有一部分数据发生了变化。现在许多课题研究的是怎样才能不重复计算规则的全部条件。可以采取在循环之间将部分已计算的条件存储起来的方法。这当中出现了许多与人工智能领域的产生式系统结合的处理方法。如Rete算法，TREAT算法和Gator算法。

Rete网有几种不同类型的结点：根结点是网的一个入口结点；根结点的下面

是代表基本表的结点；这些结点的下面会有零个或多个过滤结点，每个过滤结点都有一个单独的条件。满足过滤节点中条件的元组就存储到一个α存储区内。如果一个结点有两个父结点，就意味着该结点是两个父结点基本表的连接，该结点的过滤结果存储到β存储区内。这样，树的底部就形成了支持冲突集的结点。

假设数据库中有三个关于学生信息的基本表：

Student (name，SNO#)
Course (CNO#，Cname，SNO#)
Score (CNO#，level)

有一主动规则的条件为：

Student.name="Mike" and
Student.SNO#=Course.SNO# and
Course.CNO#=Score.CNO# and
Score.level= "good"

按照前面 Rete 算法的说明，图 3.3 表示了上面的条件匹配过程。Rete 匹配阶段[42]信息从 Rete 网的根（root）向下传递。例中，如果 Score 元组进入数据库，则测试其属性 level 是否等于 good，如果等于，将它存储在带有α存储器的结点 alpha1 中，该元组然后有和另一个α连接，连接后生成的结果存储到β存储器结点 beta1 中。这个新产生的β结点再和α结点 alpha3 进行连接最后产生一个数据加入冲突集。

Rete 主要用在产生式系统中，将它用在数据库系统中，会产生一些问题，因为数据库的数据量远远大于产生式系统。采用α和β存储区存储中间结果，同时还要维护这些中间结果，会给加上巨大的系统开销。

对于中间结果有两种极端的做法：一种是不存储任何中间结果；另一种是存储所有的中间结果。Rete 的做法是存储所有的中间结果。

而算法 TREAT 就是对 Rete 算法的变异，它只有α存储器，但是没有β存储器。不存储任何中间结果。在许多情况下，TREAT 的性能超过了 Rete 算法。

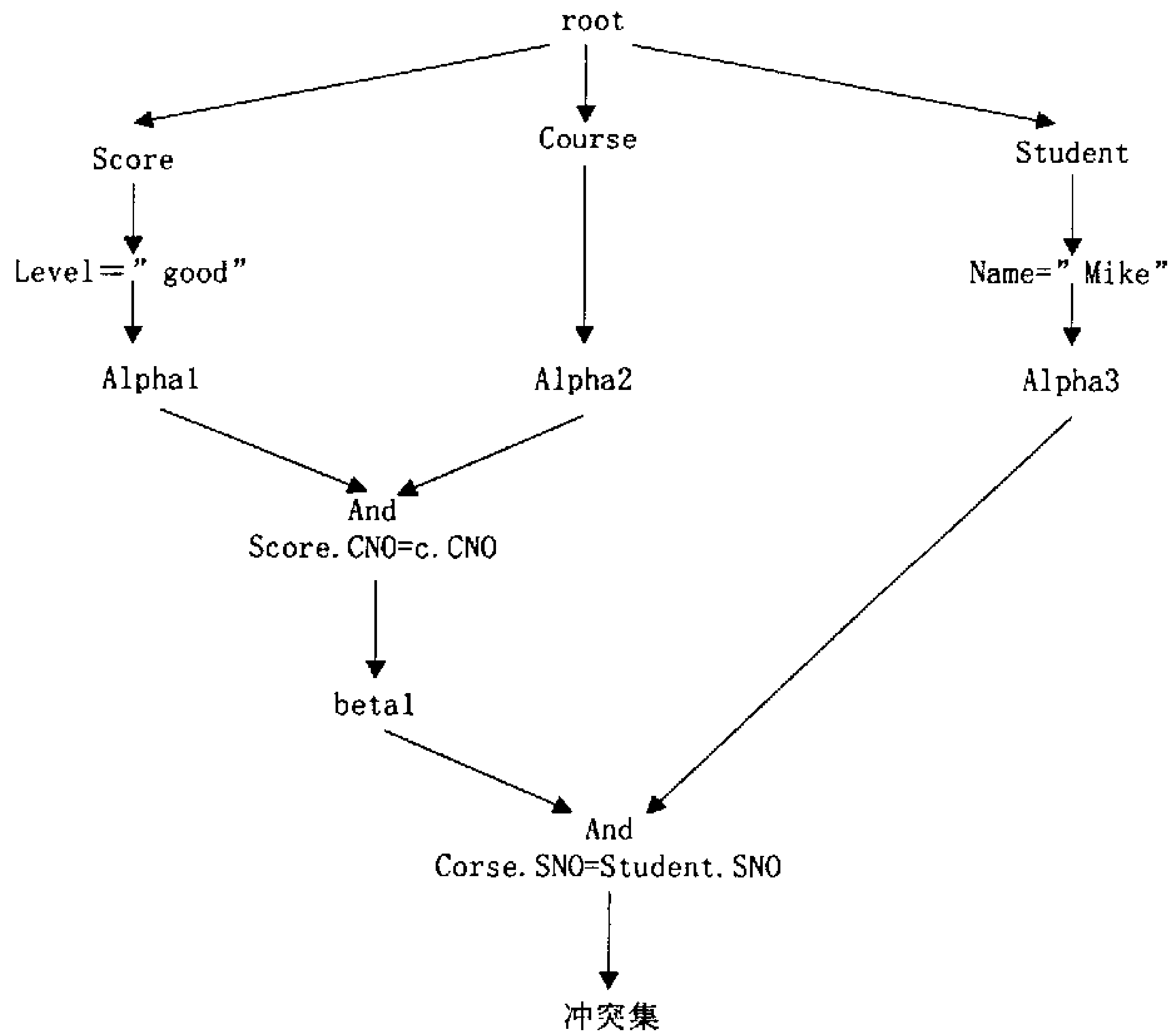


图 3.3 Rete 识别网

Gator 网[43]是以上两种算法的折衷，它同时具有α存储区结点和β存储区结点。β结点的输入可以不局限于两个，有多输入的β结点的输入可以是α结点也可以是具有多个输入β结点。

假设有一条规则的条件用下面的条件图表示为：

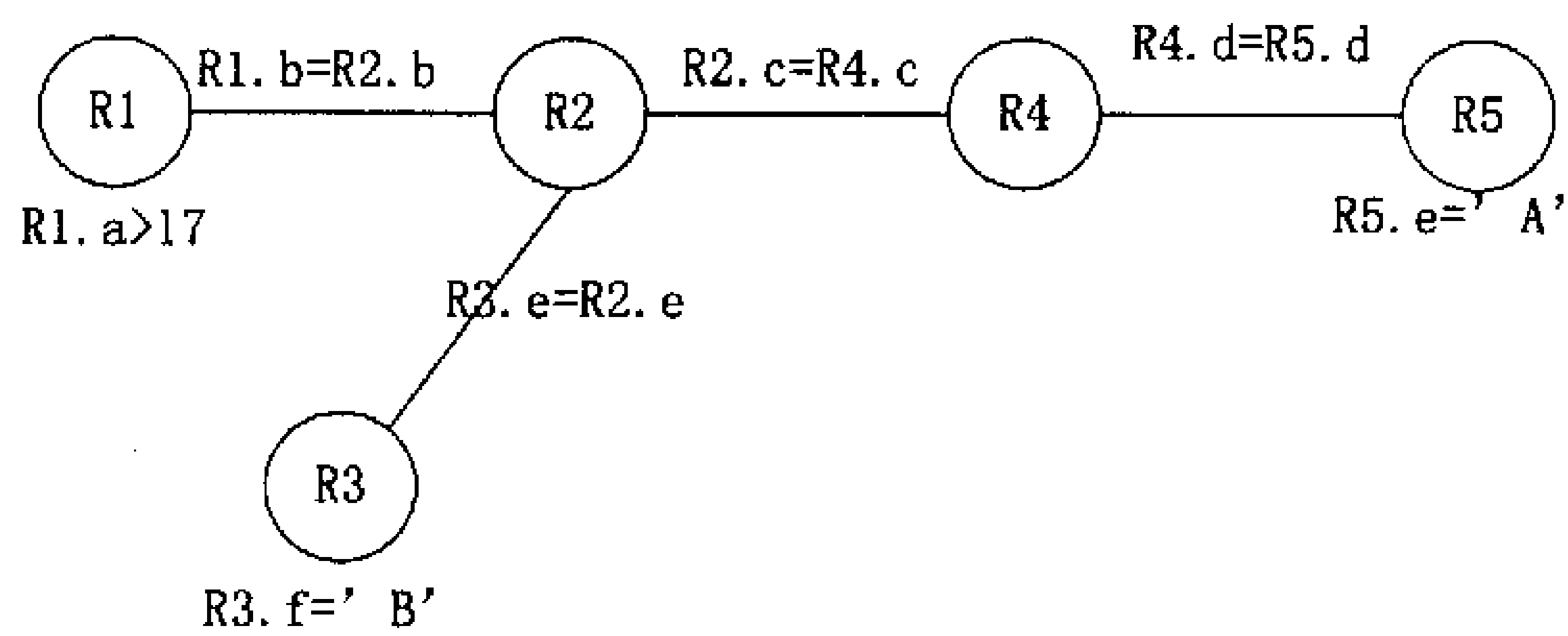


图 3.4 规则条件图

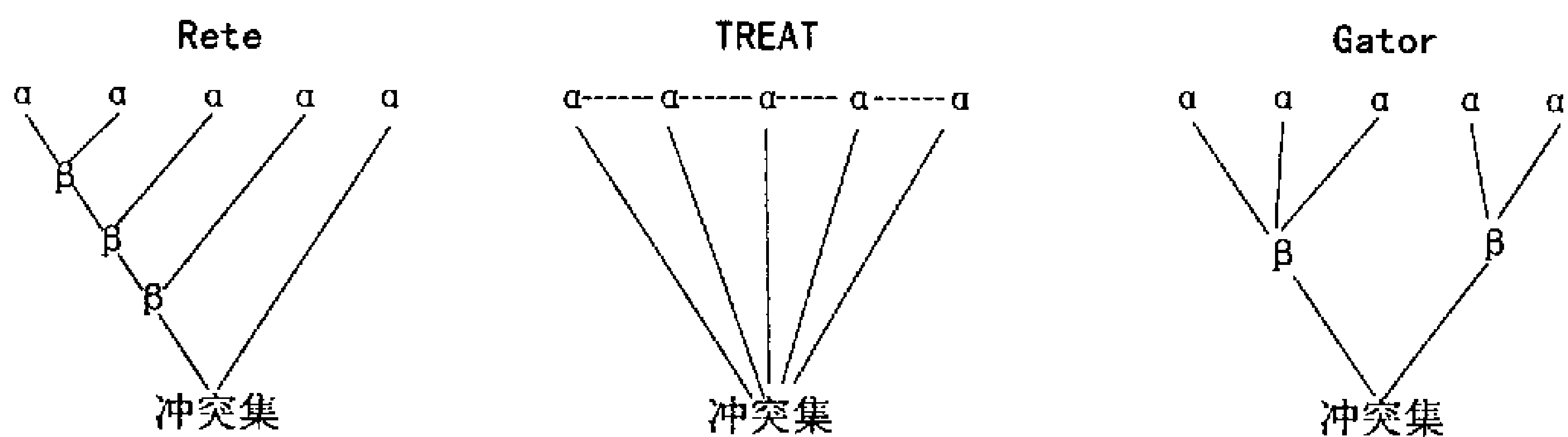


图 3.5 Rete、TREAT 和 Gator 识别网结构

和前面两种识别网相比，Gator 灵活性很强，因为一个规则条件的识别网可以是没有存储区的的实体化结点，也可以是二叉树结构。它可以作为识别优化的目标。

第四章　复合事件的检测

如前所述，在ECA规则的事件中，出现两类事件：基本事件和复合事件。那么事件监视器怎样检测复合事件呢？为了支持复合事件的检测，首先需要基本事件的探测器。无论何时检测到基本事件，都要检查该基本事件是否会构成复合事件。一个很直接但是效率不高的检测复合事件的方法是这样的：

1 确定发生的基本事件可以构成哪个复合事件。

2 对每个确定的复合事件，检查构成该复合事件的其他所有基本事件是否已经出现。

这种方法要检测所有已发生的基本事件的信息，需要重复检测。每当一个基本事件发生时这些步骤都要重新再做一次。显然效率不高。本文中论述的检测方法利用了染色Petri网对复合事件进行检测，下面详细叙述该检测方法。

4.1　数据结构定义

首先对基于染色Petri网的检测法使用的数据结构进行定义，本文把它定义为Petri-DS。

定义：Petri-DS是一个元组（P，P_s，P_e，T，A，C，CF，E，G）

P：有穷点集

P_s：属于P，一个输入点的有穷集。

P_e：属于P，一个输出点的有穷集。

T：变迁的有穷集合。

$A \subset (P \times T \cup T \times P)$ 弧的集合，弧表示点和变迁之间的连接。

C：一个token的有限集。

CF：P→C。token类型函数。把每个点映射到一个属于C的token类型。

E：弧函数。每个弧映射一个表达式（叫做弧表达式）。

G：守卫函数。把每个变迁t映射到一个布尔表达式（叫做守卫表达式）。

注释：

输入点（P_s）模拟事件，输出点（P_e）模拟构成的复合事件。除了代表事件的点，一个Petri-DS还可能包含辅助点，用来模拟事件发生之间的“依赖”关系，如事件E_1必须在E_2之前发生。

对每个变迁t，它的输入点为集合{p|(p，t)∈A}，他的输出点为集合{p|(t，p)∈A}，输入弧为集合$\{a \mid a \in P \times T and \exists p : a = (p,t)\}$，输出弧定义为集合

$\{a \mid a \in T \times P and \exists p : a = (t, p)\}$。

每个点可能包含几个 token，Petri-DS 的状态是由将 token 分配给它对应的点来确定的。当一个点中加入一个 token，Petri 网就开始动态操作 token，也就是将 token 从一个点移动到另一个点。在一个 Petri-DS 中，token 可以携带复杂信息。每个分配到对应点的 token 都是一个特定的 token 类型，它携带的信息是一种特定结构的信息。也就是说，每个点都有一个相对应的 token 类型，每个用在该点的 token 都必须是这种 token 类型。

点模拟的是事件模式，有一个事件发生时就在相应的点就做一个标记，这样 token 的值实际就是事件。我们可以为每个事件定义一个 token 类型。对每种基本事件类型只需要一个 token 类型。输出弧表达式包含一个函数，该函数的参数为输入弧表达式中的变量。可以采用各种不同的函数确定怎样从输入点的 token 创建输出点的 token。

变迁 t 的守卫表达式用来模拟对成员事件（组成复合事件的基本事件）的限制。具体说，Petri-DS 中的守卫表达式总是 t 的输入点的 token 的值的关系表达式。本文采用这样一个约定：如果一个变迁没有用守卫表达式注释出来，那么守卫表达式的值总是为真。

4.2 Petri-DS 的动态行为

在某一个状态下，Petri-DS 的每个点都包含一定数量的 token。该状态下，每个点的 token 集合都是由标记（Mark）操作来添加的，标记操作把 token 集合定义为函数 M: p→T。其中，T 是一个集合，它的每个元素都是一个 token 集合。这样，M(p)表示所有驻留在点 p 的 token。M_0 是 Petri-DS 的初始化 mark 操作，它对应创建 Petri-DS 时的 mark 操作；M_0 还定义了辅助点的 token。

如果有事件信号发出时，Petri-DS 就对模拟该事件的点标记一个 token，该点获得一个新的标记 M_1，Petri-DS 便开始动态操作 token，在操作 token 的过程中要考虑弧表达式和守卫表达式给出的附加要求。token 操作的目的是：找出可以被点燃的变迁，且该变迁的输入点必须已经被标记了 M_1。对于所有这些变迁 t，需要进行下列步骤：

1 检查是否 t 所有的输入点都标记了 M_1。

2 如果是，则变迁 t 的所有输入弧的弧表达式中的变量值一定是相应的 token 的值。

3 选择守卫表达式等于真的变迁 t 进行点燃。

4 从输入点中删除 token，在输出点中加入 token。选择哪个 token 进行添加或删除由对应的弧表达式（在输入或输出弧上）的值决定。

4.3 用图表示 Petri-DS

用图表示 Petri-DS 需要用到以下的元素：

点 Place，用圆圈表示，带有两个属性：点的名称，也就是该点所模拟的事件模式的名称（如，E1）；该点对应的 token 类型（如，图 4.2 中的[DBE]）。

变迁 Transition，用长方形表示，它也带有两个属性：变迁的名称，如 t；变迁的守卫表达式。

弧 Arc，由从点到变迁（或从变迁到点）的带箭头的弧表示，带有属性弧表达式。

4.4 使用 Petri-DS 构造复合事件

下面用“同时发生事件”和“历史事件”作为例子说明 Petri-DS 模型怎样模拟复合事件的构造。前面已经给出了 Petri-DS 的定义，基于以上定义可以对每个用户定义的复合事件构造一个 Petri-DS 实例。对每种 Petri-DS 可能会有多个 Petri-SD 实例。这些 Petri-DS 实例共同构成复合事件监视器。

4.4.1 同时发生事件

复合事件(E1∧E2)，即基本事件 E1 与 E2 的同时发生运算构成的复合事件。如图 4.1 所示，图中标识了三个点 E1，E2 和(E1∧E2)，分别模拟事件 E1，E2 和复合事件(E1∧E2)。弧 E_1→T 上的弧表达式只有一个变量，它用来将点 E1 的值传递到变迁 t。变迁 t 有守卫表达式 x.occ_tid=y.occ_tid，这表示对复合事件的构成加上的限制，该限制是：两个基本事件 E1 和 E2 必须在同一事务中发生（也就是同时发生）。从变迁 t 到复合事件(E1∧E2)的弧表达式是 f(x，y)，它的含义是：构成复合事件(E1∧E2)的 token 值必须为 token 值为 x 和 y 的联合，也就是说构成复合事件(E1∧E2)的基本事件必须是 E1，E2。

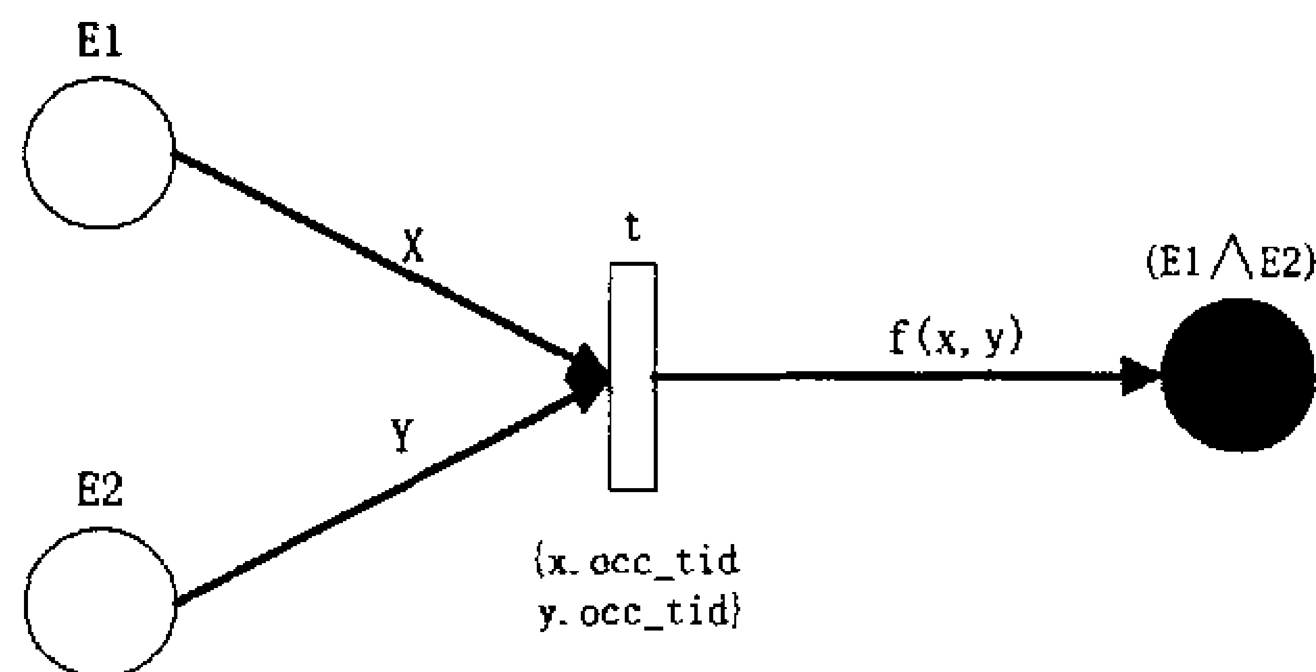


图 4.1 复合事件（E1∧E2）

现在假设用户定义了复合事件 update(update_value1，update_value2)：same transaction，这里 update_value1 和 update_value2 都是数据库操作产生的基本事件 same transaction 表示 update_value1 和 update_value2 必须在同一个事务中发生。图 4.2 表示了运用上面给出的 Petri-DS，对具体的用户定义的复合事件事件 update 生成的 Petri-DS 实例。该图中的点携带了具体的 token 类型，对每个变量都有一个声明，用来表示该变量和 token 之间的关系。点 update_value1 和 update_value2 的 token 类型都是 DBE，它们模拟的是数据库操作形成的基本事件；点 update 的 token 类型是 CE_1，它模拟同时发生事件。

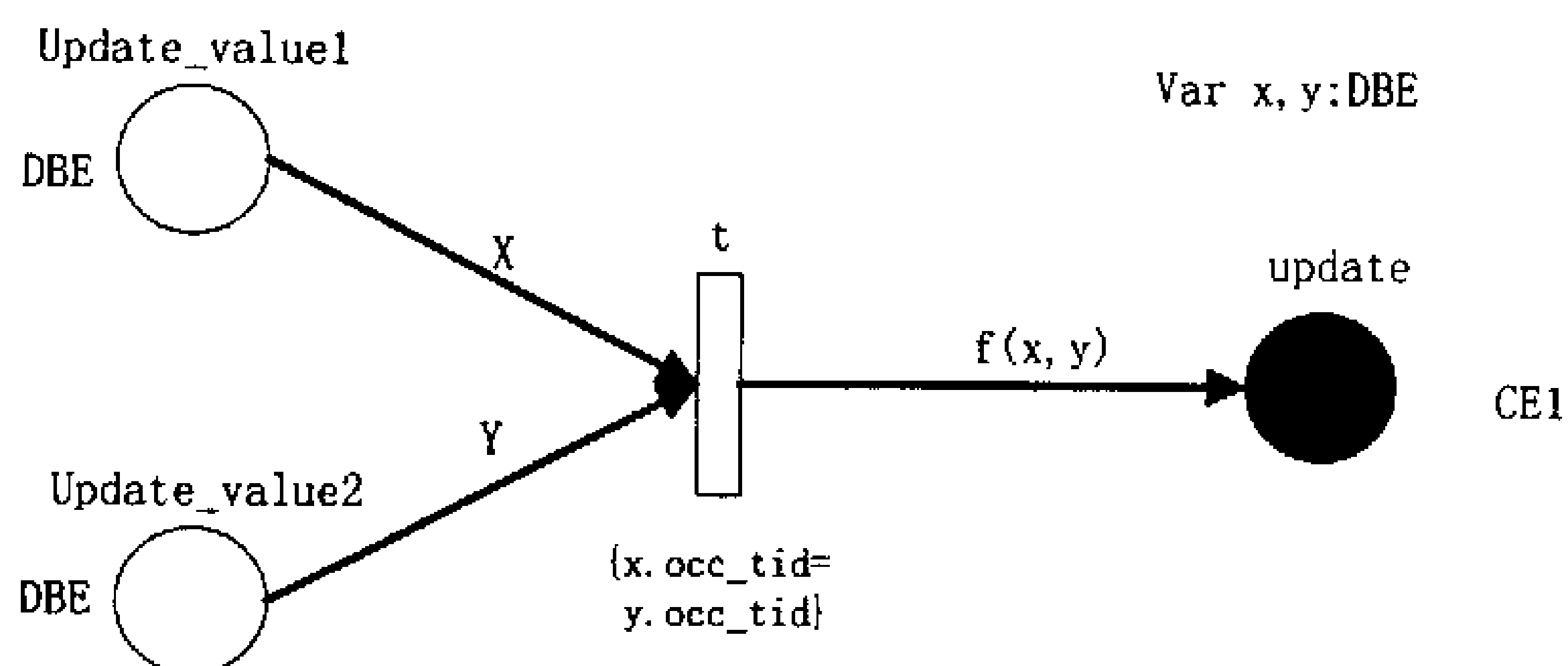


图 4.2 “同时发生”复合事件实例 update

4.4.2 不发生复合事件

复合事件(～E)是指基本事件 E 没有发生构成的复合事件，确切地说不发生复合事件是指在某个时间段[s，e]内不发生该事件，涉及时间段的复合事件可以采用添加“开始时间事件”和“结束时间事件”的方法来表示，则时间点 s 和 e 用时间事件表示就是 TS 和 TE。为了表示“不发生”语义，这里引入一个特殊的弧，叫做禁止弧。禁止弧的含义是输入点条件为真导致变迁不能发生[47]。在图中用带有小圆圈的有向弧表示。禁止弧携带的信息可以导致变迁 t 不能被点燃。

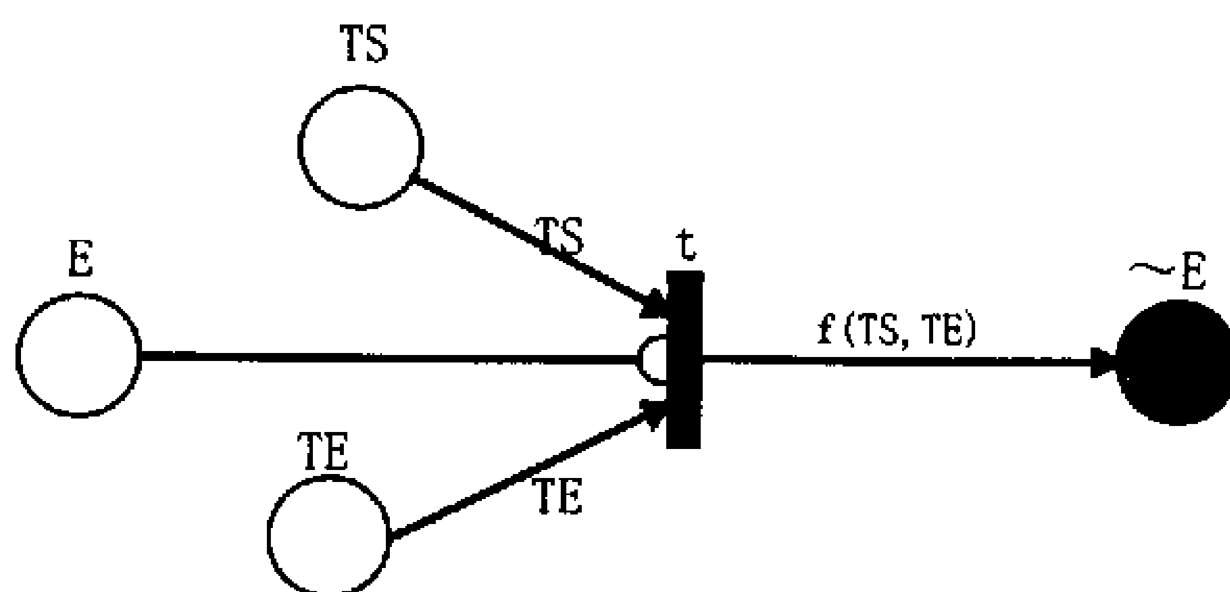


图 4.3 不发生复合事件

4.4.3 n 次发生复合事件

复合事件 TIMES(n，E) IN [s-e]，该复合事件表示：在时间段[s-e]闭区间内基本事件 E 发生 n 次。图 4.4 中显式了模拟该复和事件的 Petri-DS。为了简化，不考虑其中传递的参数。

采用代表发生频率的常数 n 标识从点 E 到变迁 t_3 的弧。每当 E 发生时，E 得到一个新的 token，token 的数量表示该事件发生的次数。当点 E 中含有的 token 数量达到 n 时，就点燃变迁 t_3。

这里对 Petri 网和自动机作一下比较。用 Petri 网模拟复合事件，它对每个事件模式只定义一个点，该事件每发生一次，该点就记录一个 token。而采用自动机模拟复合事件，则一个自动机状态对应一个事件的发生，这样一共要定义 n 个自动机状态。例如，复合事件 TIMES(3，E）转换成((E；E)；E)，每个事件 E 都对应一个自动机状态，则这三个状态都要在自动机中定义。可以看出，采用自动机模拟复合事件会造成一定程度的浪费。

事件定义 TIMES(n，E) IN [s-e]可以转化成[TS；～TE]；TIMES（n，E），图 4.4 模拟了该复合事件。这里，H 是一个辅助点，它是用来忽略时间事件 TS 之前所发生的 E 事件。用这种方法可以确保是在 TS 发生之后 TE 发生之前事件 E 发生了 n 次。同样，TS'也是一个辅助点，添加了这个辅助点后，触发了一次复合事件 TIMES(n，E) IN [s-e]时，后面又出现可以再次形成该复合事件的 n 个 E 事件，这个 Petri－DS 仍能继续记录复合事件。综上所述，每当在 TS 和 TE 之间 n 个 E 事件发生后，事件监视器就发出复合事件 TIMES(n，E) IN [s-e]的信号。

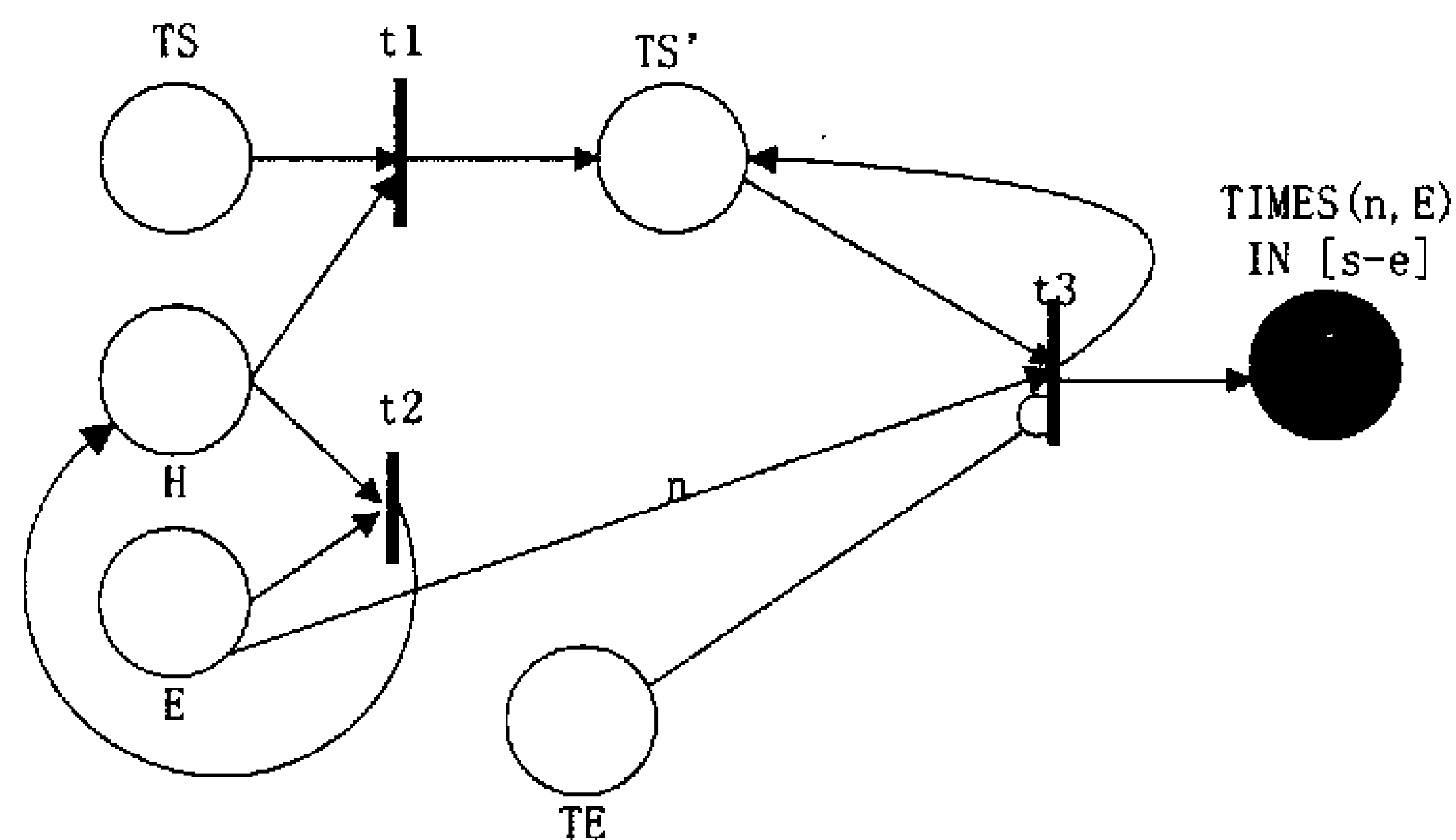


图 4.4 复合事件 TIMES(n，E) IN [s-e]

4.4.4 复合事件的组合

每当用户用事件构造器定义一个复合事件时，事件监视器就创建一个新的Petri-DS。系统组件负责搜索Petri-DS类型，并创建Petri-DS实例。这样，复合事件的事件监视器就是一个包含所有Petri-DS的组合。如果复合事件不仅仅是由基本事件构成的而是由多个复合事件构成，则需要将多个独立的Petri-DS组合成一个Petri-DS来模拟该复合事件。组合可能是下面的情况：

1 复合事件继续参加下一个复合事件如，E=(E1∧E2) and EE=(E|E3).则复合事件E的输出点是另一个复合事件EE的输入点。图4.5说明了E和EE的Petri-DS

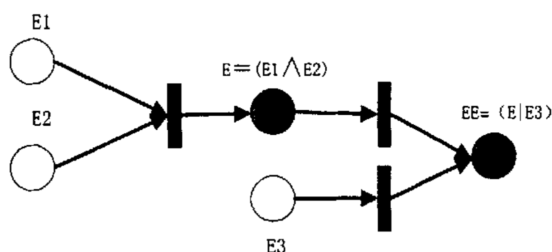


图4.5 复合事件参与别的复合事件

2 事件参与多个复合事件，如E1参加了事件E=(E1|E2)，又参加了事件EE=(E1∧E3)中的E1。图4.6显示了E和EE怎样组合成一个Petri-DS。E1复制成E1'和E1"，而E1只占一个点。

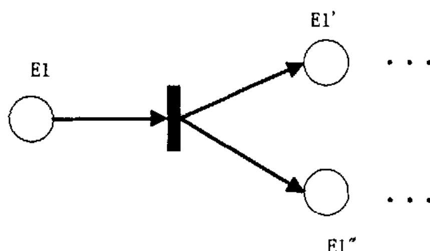


图4.6 事件参与多个复合事件

4.4.5 事件监视器工作方式

下面说明Petri-DS在基本事件的事件监视器发出基本事件信号后要做的工作：

1 Petri-DS的输入点得到一个token，该token含有实际发生的事件参数。

2 检查是否可以点燃一个变迁。然后，可能有一个或多个变迁会被点燃，并

且对一个或多个输出点（这些点模拟复合事件）进行标记操作。

3 发出复合事件信号，基本事件作为参数在各自的 token 中被传递。

4 如果这些输出点中的一个是 Petri-DS 所谓的终点，则删除 token。

4.5 实现

在实现过程中，本文采用了面向对象的方法，所以选择面向对象化的 JAVA 语言。Petri－DS 中的每个点都是一个 Place 类的实例。Place 类有两个属性，分别是：index 和 list_of_tokens。index 是一个系统内部分配的数字，list_of_tokens 是指向类 token 的多个对象的引用列表；这些 token 是对该点的进行了标记操作的 token。当用户定义一个复合事件时，系统就会创建适当的对象。因为初始化时 Petri－DS 中只有辅助点的包含 token 信息，这些 token 信息都存放在 list_of_tokens 属性中。在需要进行标记操作时，就会根据增加或删除 token 的情况对属性 list_of_tokens 进行修改。在本文实现的 place 类中通过 addTokens()和 removeTokens()两个成员函数实现类 token 的引用添加和删除，其具体实现如下：

```
Integer currentIndex=1;
public class Place extends Node {
  private ListOfToken listOfToken;
  private Integer index;
  private List listeners = new ArrayList();
  private List fusionPlaces = new ArrayList();
  private List inputs = new ArrayList(MAXINPUT);
  private List outputs = new ArrayList(MAXOUTPUT);
  /* Creates a new   Place   instance.    */
  public Place() {
     this.index = currentIndex;
     currentIndex = currentIndex + 1;
     this.listOfToken = new ListOfToken();
  }
  /* Adds an input Arc. @return this place.      */
  public Place addInput(final InputArc anArc){
     this.inputs.add(anArc);
     return this;
  }
  /* Adds an output Arc. @return this place.      */
  public Place addOutput(final OutputArc anArc){
     this.outputs.add(anArc);
     return this;
  }
  /* Returns list of all input arcs from this place.      */
  public List inputArcs(){ return this.inputs; }
```

```
  /* Returns list of all output arcs from this place.    */
  public List outputArcs(){ return this.outputs; }
  /* Returns all the tokens for this place.     */
  public Multiset getTokens() {
    return new ListOfToken(this.listOfToken);
  }
  /* Adds given tokens to this place.   */
  public void addTokens(final Collection aListOfToken) {
    addTokensQuietly(aListOfToken);
    notifyFusionAdded(new TokensAddedEvent(this, new ListOfToken(aListOfToken)));
  }
  /* Removes all tokens from the given listOfTokens from this place.    */
  public void removeTokens(final Collection aListOfToken) {
    removeTokensQuietly(aListOfToken);
    notifyFusionRemoved(new TokensRemovedEvent(this, new ListOfToken(aListOfToken)));
  }
}
```

Petri－DS 中的变迁有两种：带有守卫表达式的变迁、不带有守卫表达式的变迁。对每个不带守卫表达式的变迁用类 transition 的实例表示，该 transition 只有一个属性 index；对带有守卫表达式的变迁采用从父类 transition 继承而来的子类表示。不同的守卫表达式需要定义不同 transition 子类。例如，带有守卫表达式 same transaction（即限制基本事件必须在同一事务中发生）的变迁的子类可以定义为 tr_same_transaction 子类。

在本文中定义了一个类 transition 用来实现上文中提到的功能，其定义如下：

```
public class Transition extends Node implements PlaceListener {
  private List listeners = new ArrayList();
  private List inputs = new ArrayList(MAXINPUT);
  private List outputs = new ArrayList(MAXOUTPUT);
  private AdbContext context = new AdbContext();
  private Guard guard;
  private Action action = new Action();
  public Transition() ;
  public Transition addInput();
  public Transition addOutput();
  public List inputArcs();
  public List outputArcs();
  public Guard setGuard();
  public Action setAction();
  public AdbContext getContext();
  public void setContext();
  public Transition fire();
  public Transition unfire();
```

```
    public void addTransitionListener();
    public void removeTransitionListener();
    public void notify();
    public NetElement apply();
    public abstract class Guard;
    public class Action;
}
```

下面详细介绍一下其中一些比较重要的成员函数的功能。在该类中本文通过成员函数 fire()实现变迁的点燃。其具体实现如下：

```
/* Fire this transition. */
  public Transition fire() {
    /* Notifies   listener just before this transition fire.   */
    notify();
    ......
    return this;
  }
public void notify(final PlaceEvent anEvent) {
  }
  /* Registers a given TransitionListener with this place.   */
  public void addTransitionListener(final TransitionListener aListener) {
    this.listeners.add(aListener);
  }
  /* Deregisters a given TransitionListener from this place.   */
  public void removeTransitionListener(final TransitionListener aListener) {
    this.listeners.remove(aListener);
  }
  /* Notifies all listeners just before this transition fire.   */
  public void fireTransitionStartedEvent() {
    final TransitionStartedEvent event = new TransitionStartedEvent(this);
    final Iterator l = this.listeners.iterator();
    while (l.hasNext()) {
      ((TransitionListener) l.next()).notify(event);
    }
  }
```

成员函数 unfire()用于输入弧为禁止弧时禁止变迁被点燃，并将变迁状态重置，具体实现如下：

```
/* Unfire this transition. */
  public Transition unfire() {
    enabled = false;
    //remove all prepared tokens from input places
    final Iterator iter = this.inputs.iterator();
    while (iter.hasNext()) {
      final InputArc arc = ((InputArc) iter.next());
```

```
        arc.place().removeTokens(arc.getContext().getBinding().values());
    }
    ......
    return this;
}
```

在 transition 类中通过定义了两个内部类 Guard 和 Action，来实现守卫表达式和发出复合事件的信号，其中 Guard 类通过其成员函数 evaluate()来实现守卫表达式，Action 类通过其成员函数 execute()来实现发送复合事件的信号，其类具体实现如下：

```
/* Represents this transition guard.   */
public abstract class Guard implements Context {
    /* Different composite event holds different guard expression */
    public abstract boolean evaluate();
    guard
  }// Guard
  /* Represents this transition action.   */
  public class Action implements Context {
    public void execute(){
        /* signal composite event */
    }
    ......
  }// Action
}
public Guard setGuard(final Guard aGuard) {
    final Guard old = this.guard;
    this.guard = aGuard;
    return old;
  }
  /* Sets the action for this transition.   */
  public Action setAction(final Action anAction) {
    final Action old = this.action;
    this.action = anAction;
    return old;
  }
```

添加输入输出弧的成员函数和列出输入输出弧的实现如下：

Petri－DS 中的弧由类 arc 的实例表示。Petri－DS 中的每条弧都是相应 arc 子类的对象。类 arc 有两个属性：place 和 transition，分别表示该弧对应的点和变迁的索引。为了实现上文的定义，首先定义一个类 arc，其定义如下：

```
public interface Arc extends NetElement {
  /* Returns the transition for this arc. */
  Transition transition();
  /* Returns the place for this arc. */
```

```
  Place place();
  /* Returns the context for this arc. */
  AdbContext getContext();
  /* @param ArcType value "normal" "forbid" */
  String AcrType;
} // Arc
```

类 arc 有两个子类，inputarc 和 outputarc，本文通过定义 inputarc 和 outputarc 继承该类实现输出弧和输入弧。禁止弧用一个标识位 ArcType 进行标记，如果是禁止弧则 ArcType 的值为 forbid。

第五章 主动数据库中的执行模型

执行模型是指主动规则的执行方式。它包括主动规则中各部件之间的耦合方式及其语义描述，即系统在运行时如何处理规则集。传统数据库主要是由可被用户显式调用的应用程序组成。这些应用程序执行时被处理为原子的、可恢复的事务序列。系统采用并发控制及恢复技术来保证事务的可串行化及一致性。而在主动数据库中，系统必须执行附加在用户事务上的已被触发的程序或动作。因此，要求系统对这些与用户事务相关的触发动作加以处理，以达到可串行化与一致性的要求。主动数据库的执行模型描述了主动数据库系统中规则执行对于事务的相关语义。主动规则系统的执行模型与低层的数据库管理系统有着密切的联系。执行模型是对传统事务模型的发展和扩充。在主动数据库中，规则的执行可分为以下几个阶段：

1 信号通知阶段，指事件源引起事件发生的现象。

2 触发阶段，产生事件（包括复合事件）并触发相应的规则。规则和与之相联系的发生事件形成了规则实例。

3 评估阶段，对被触发规则的条件进行评估。条件为真的规则实例形成了规则冲突集。

4 规则调度阶段，指对规则冲突进行处理。

5 执行阶段，指执行所选出的规则实例的动作。动作执行时还可能产生其他事件，称为规则的级联触发。

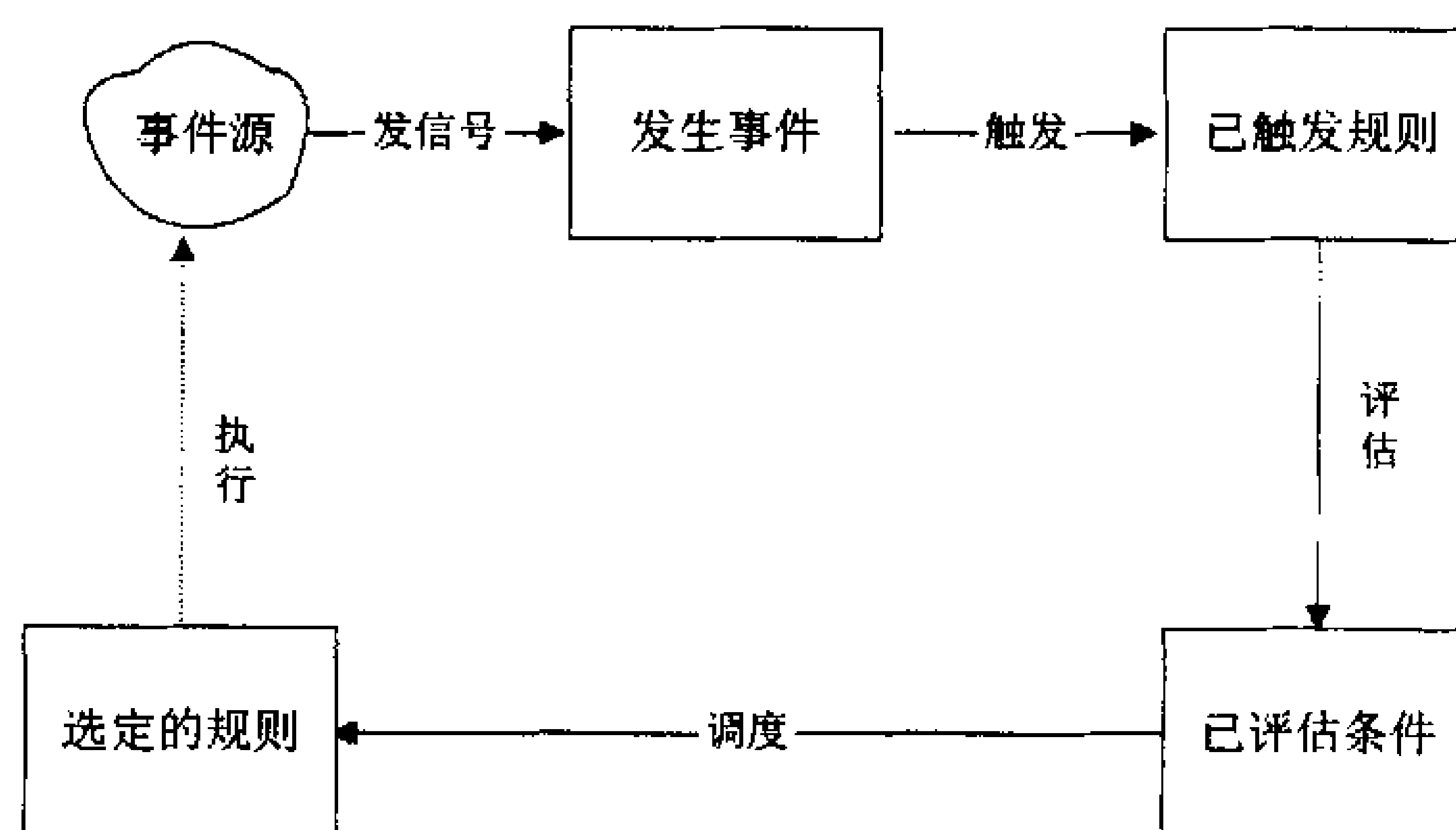


图 5.1 规则执行过程图

5.1 耦合方式

规则执行是按照事件与条件、条件与动作之间的耦合方式执行的。事件与条件的耦合方式是描述事件发生与评估条件之间的关系；条件与动作的耦合方式是用来描述条件评估与动作执行之间的关系的。

事件与条件的耦合方式有三种：

1 立即式（Immediate）：事件发生后立即进入条件评估阶段，就是说在引起事件发生的事务（触发事务）中进行条件评估。事件的发生和条件的评估在同一个事务中。

2 延迟式（Deffered）：事件发生后，不立即进行条件评估，但是条件也在触发事件的事务中进行，但不是很快就执行条件评估，通常是等到事务执行结束之前执行条件评估。

3 分离式（Detached）：事件发生和条件评估在两个不同的事务中进行。

同样，条件与动作的耦合方式也有三种：

1 立即式（Immediate）：评估条件为真后立即执行动作。

2 延迟式（Deffered）：动作的执行不必在条件评估成功后立即执行，但必须在评估事务结束前执行。

3 分离式（Detached）：动作的执行与条件评估不在同一个事务内，动作的执行可以独立于或依赖于事件发生事务或条件评估事务的提交。

在 ECA 规则中，事件、条件与动作三者之间是触发与被触发关系，因此从事务的观点看，耦合方式又可被认为是在同一个嵌套事务中，只是被触发事务的执行与提交的时间点不同。在立即式耦合方式下，被触发事务必须在触发事务发生时立即执行；而延迟式耦合方式下，被触发事务可延迟到触发事务提交前执行；在分离式耦合方式下，触发事务与被触发事务可认为是两个相对独立的事务，根据两个相对独立事务的关系又可分为因果依赖事务和独立事务。因果依赖事务指当且仅当触发事务已提交且被触发事务已串行化在触发事务之后，则被触发事务可以提交。因果依赖事务又可细分为序列因果依赖和独占因果依赖方式。序列因果依赖指只有在触发事务提交后被触发事务才能开始。独占因果依赖指当且仅当触发事务失败后被触发事务才能提交。独立事务指触发事务与被触发事务的提交相互独立，没有制约关系。

无论耦合方式如何细分，立即式和分离式是耦合方式的上线和下线，在执行的时间点上决定了耦合方式的低线。通常，用户可以定义事件条件之间及条件动作之间的模式为立即式、延迟式、分离式三种，根据三种模式及其对触发事务与被触发事务之间关系的影响，E—C 和 C—A 可能的组合方式有：（立即式，立即式）、（立即式，延迟式）、（立即式，独立式）、（延迟式，延迟式）、（延

迟式，独立式）、(独立式， 独立式)。模式组合方式的选择必须根据特定应用领域的需求。

耦合方式的形式定义如下：

<Coupling Mode>::=Immediate|Deferred|Detached

5.2 触发粒度

触发粒度是指事件与事件所触发的规则之间的关系。触发粒度受转变粒度（Transition Guanularity）与净效策略（Net Effect policy）的影响。转变粒度是指事件的发生与触发的规则实例是一对一关系还是多对一关系。系统的状态转变是由不可分的操作序列产生的。如果转变粒度为元组（Tuple），那么单个事件触发单条规则；如果转变粒度为集合（Set），那么一个发生事件的集合被用于触发一条规则。例如：如果条件—动作耦合方式为延迟式的一条规则 R，其监测的事件为 E，而在一事务内事件 E 的三个实例 e_1、e_2、e_3 分别发生，那么转变粒度就决定了在触发阶段将产生多少条 R 规则实例。如果转变粒度为元组，那么相对每个 e_1、e_2 和 e_3 将分别产生 R 规则实例。如果转变粒度为集合，那么相应事件集{ e_1、e_2、e_3}只产生一条 R 规则实例。净效策略表明了系统是考虑多事件发生后的净效果，还是考虑每一独立事件产生的效果。这两种策略的不同处源于对同一数据项的多次修改可被认为是一次修改的情形。此情形具体为：

1 如果有一实例被修改后删除，那么净效果就是删除原始实例。

2 如果有一实例插入后又被修改，那么净效果可认为是被修改实例的插入。

3 如果有一实例插入后又被删除，那么净效果可认为是不发生操作。

由于转变粒度不同，因此相对不同的转变粒度要有相应的绑定模式。所谓绑定模式就是指事件中的变量如何作为参数传递给条件或者动作。发生事件一般都与其发生的数据项绑定。绑定模式不但决定了事件的粒度，而且决定了与事件相关联的条件和动作部分所能获得的数据项。以下给出两种绑定模式：

1 面向个体绑定：指事件发生所在的特定个体（如元组、单个对象）能够被条件和动作部分参引。如果在一个集合内，相同的事件被绑定在多个个体上，那么每一个体各自执行起相联的规则。

2 面向集合绑定：指一个发生的事件与其发生所在的个体的集合（如一个关系中所有被修改的元组）相关联。在此情形下，只有当事件所发生的所有个体被绑定，其相关联的规则执行。条件和动作部分可以参引个体集。

如果在这两种绑定模式中设置优先级别，那么条件部分可以优先于事件的发生而参引个体的状态。在理想的主动数据库中，系统在每个规则基中均支持两种绑定模式的选择。在二者只能取其一的情形下，绑定模式是系统中牢靠的线索。

5.3 循环策略

循环策略是指当发出如果事件信号在评估条件或动作执行期间发出，系统的处理方法。这种情况通常是在规则动作作为事件而触发新的规则，从而引起级联触发时出现。级联触发涉及的各条规则具有明确的因果关系，因而其执行顺序也很明确。级联触发的一个潜在危险是其所构成的规则链会导致循环的级联触发，从而使系统处于无限或有限循环之中。这种级联触发可能是某条规则的自我触发直接产生的，也可能是多条规则的相互触发而间接产生的。由于级联触发产生的循环将影响系统的效率，因此我们须采取一定的策略加以控制。有两种循环策略，一种是重复策略，另一种是递归策略。

重复策略：即规则自身被触发后，前一条规则实例仍然执行，被触发的规则实例根据自身的优先权值顺序执行。所以如果采用重复策略，在规则条件评估或动作执行期间发出了事件信号，将该事件信号与各个处理阶段中发生的其他事件信号放在一起，形成一个大的事件信号集合，然后规则从这个事件信号集合中选择事件进行消费。这就意味着条件评估和动作评估不会因为有新的事件发生而暂停运行，对这些事件信号作出反映。

递归策略：即规则自身被触发后，前一条规则实例的执行事务被挂起，转而执行被触发的规则实例。因此耦合方式为立即式的规则可在第一时间被处理。如果采用的是递归策略，在条件评估和动作执行阶段发出的事件信号会使得条件和动作暂停运行，这样监视该事件信号的立即式规则可以在最早的时刻得到处理。实际上，有两种支持支持递归循环策略的系统，一种是支持立即式规则处理方式的系统，另一种是对立即式规则采用递归策略而对延迟式规则采用重复策略的系统

递归策略一般都被用于支持立即式规则的处理，重复策略被用于支持延迟式规则的处理。对于规则间产生的级联触发则可采用超时、超触发规则数目等监控策略加以控制。

5.4 规则调度

规则调度是指系统对规则的动态管理。被触发的规则应该如何执行，是立即执行还是到触发事务结束时执行。是否给被触发规则的执行创建一个单独的事务，触发事务与被触发事务是并发执行还是顺序执行，如何有效地协调事务之间的关系，以保证整个系统信息畅通、高效和正确，等等。所有这些都是规则调度的任务。在主动数据库管理系统中，规则调度模块是保证各规则被正确执行的关键。

规则评价的调度阶段确定如何处理在同一时刻触发的多条规则。以下将从两个基本主题出发进行讨论：

第一，选择哪一条规则作为下一条要触发的规则，即规则冲突处理。该课题曾得到专家系统研究学者的很大重视，他们把规则冲突处理看作是理解和控制规则集行为的基础。规则的执行顺序会极大的影响到执行结果，并且可以反映系统所服从的推理机的类型。一般，能高效率的处理大量数据的主动数据库系统都支持优先级调度机制。突解决策略的优先级机制包括两类：动态优先级方法和静态优先级方法。

动态优先级方法有两种定义方式：一，就是检查更新操作是否为一个新的更新操作，具有越新更新操作（事件出现越晚）的规则，优先级越高；二，看条件的复杂程度，具有越复杂条件的规则，优先级越高。使用最新更新操作策略的系统重点放在推理，最近修改的数据和最近点燃的规则有很大联系（也就是说，搜索方法是深度优先搜索）。使用复杂条件优先策略的系统假设条件的复杂性反映规则的特性（也就是说，越复杂的条件适合当前数据库状态的程度越高）。

静态优先级方法通常也有两种定义方式，一种由系统根据规则的创建时间定义，另一种作为规则的属性由用户自己定义。分别有两种不同的优先级机制：绝对优先级：用数字调度顺序排列规则，这样每条规则优先级的值都是一个确切的数字；相对优先级：同时触发的两条规则，对其中一条规则进行显式声明该规则必须在另一条规则之前点燃。

第二，点燃规则的数量。有四种可能的情况：

1 用串行的方式依次点燃所有的规则实例；

2 用并行的方式点燃所有的规则实例；

3 用带饱和度的方式点燃规则实例，也就是说：先考虑一部分规则实例，并将这部分规则全部点燃，然后再考虑剩下的那部分规则实例；

4 仅点燃规则实例中的一个或几个。

以上何种方法最适合须考虑规则要支持什么样的任务。第一种方法适用于对完整性进行维护的任务，如：必须所有的约束都检验有效一个修改操作才算成功。第二种方法适用于需要高效地对规则进行处理的任务。第三种方法适用于专家系统，因为它可以比其它几种方法更适合推理。第四种方法可以用来支持导出数据，每条规则支持一种导出准则，因此当某条规则被触发时，此种准则即被采用。

5.5 错误处理

规则在被激活后就做好了被执行的准备。规则何时执行以及用什么方式执行完全靠主动数据库管理系统中的规则调度和管理模块来决定。在执行规则时，规

则调度和管理模块所做的事情包括：

1 从规则库中选择某条已被激活的规则执行；

2 根据需要创建被触发事务；

3 管理各种嵌套事务的执行；

4 负责发送和接收各嵌套事务之间以及触发事务与被触发事务之间的各种消息；

5 处理级联触发。

规则的执行是一种非常复杂的过程，因此异常处理就显得尤为重要，异常处理是否得当将极大地影响规则执行的可靠性和效率。对异常的处理可采用以下四种方法：

放弃：即当前事务若发生异常就放弃，转向其它事务。

忽略：若正执行的某条规则产生错误，系统忽略之，并继续执行其它规则处理。

跟踪：若某条规则产生错误，就回归到开始出路规则的状态，并重新开始处理规则或者继续当前事务。

预防：即采用某种防范策略，尽可能地使系统从错误状态恢复，此种方法可借助底层数据库的异常机制。

在这四种方法里，第一种方法在传统数据库系统中被广泛采用，后三种方法则在主动数据库系统中采用起来显得更为方便。

第六章　主动数据库中的事务

在主动数据库中，对事务的调度不仅要满足并发环境下的可串行化要求而且要满足对事务时间方面的要求。它要求综合传统数据库的并发控制技术和实时操作系统中与时间要求有关的调度技术。本章中将提出一个基于扩展的多版本两段锁协议的事务模型。

6.1　基本概念

数据库是对象的集合。事务 T_i 是序偶$(\Sigma_i,<_i)$，这里Σ_i是 T_i 中的操作集，$<_i$是操作的执行顺序。T_i 的读或写操作分别表示成 $r_i[x]$，$w_i[x]$。事务的终止或者提交（c_i for T_i）和放弃（a_i for T_i）。令 $T=\{T_1, T_2...T_n\}$是事务的集合。T 的历史 H 是一个偏序关系（$\Sigma,<_H$），这里Σ是 T 中事务执行操作的集合，$<_H$是这些操作的执行顺序。

如果：1) H_1 和 H_2 定义在同样的事务集合上，2) 操作 o_1 和 o_2 是两个冲突的操作，且$o_1 <_{H_1} o_2$则$o_1 <_{H_2} o_2$，则 H_1 和 H_2 是冲突等价（conflict equivlent）的。如果：两个事务 T_i 和 T_j，T_i 所有的操作都在 T_j 所有操作之前，或者 T_i 的所有操作都在 T_j 的所有操作之后，则称这两个事务历史 H 是串行（serial）的；如果历史 H 和一个串行的历史相等，则称历史 H 是可串行化的。通过对 H 的串行化图 SG（H）进行分析，可以判断 H 是否为可串行化的。串行化图（serialization graph）是一个有向图，该图由 SG（H）=(V，E)对组成。其中 V 是顶点集，E 是边集，顶点集由 H 中提交了的事务组成，边集由满足下列条件之一的边$T_i \to T_j$组成[24]：

1 在 T_j 执行了 read(x)之前，T_i 执行了 write(x)。

2 在 T_j 执行了 write(x)之前，T_i 执行了 read(x)。

3 在 T_j 执行了 write(x)之前，T_i 执行了 write(x)。

如果历史 H 的串行化图中有环，则历史 H 不是冲突可串行化的；如果图中不存在环，则 H 是可串行化的。

在多版本数据库中，每个对象 x 上的写操作都产生一个 x 的新版本。这样每个对象 x 都有一个版本列表记为：x_i，x_j，...下标是对 x 进行写操作事务的索引序号（x_i 表示 x 的这个版本由 T_i 创建）。多版本历史（MV）是指在事务提交的这些多个版本上进行操作的序列。每个写操作 $w_i[x]$映射成 $w_i[x_i]$（事务 Ti 对 x 进行写操作，创建了版本 x_i），每个读操作 $r_i[x]$映射成 $r_i[x_k]$（事务 T_i 对 x 进行读操作，读到前面事务创建的 x 的某个版本 x_k）。如果有 $r_j[x_i]\in H$，这说明事

务 T_j 读取的 x 的版本为 T_i 创建的版本，记做 T_j reads-x-from T_i。H 中唯一可能产生冲突的操作是事务 T_i、T_j 在某个 x 上的 $w_i[x_i]$和 $r_j[x_i]$操作。如果两个 MV 历史的操作相同，则这两个 MV 历史等价。

MV 历史 H 的多版本串行化图 MVSG（H）是一个有向图，该图由 MVSG（H）=(V，E)对组成：结点表示已提交过的事务；边的构造方法如下：

如果，对某个 x 有 $r_j[x_i] \in H$（也就是说 T_j reads-x-from T_i），则有一条 $T_i \rightarrow T_j$ 的边。

对每个对象 x，所有对 x 进行写操作事务有一个总顺序（记为 $\prec_x$）。添加一条边 $T_i \rightarrow T_j$ 到 MVSG（H）中，当且仅当：

对某个 x 和 T_k，T_i reads-x-from T_k 且 $x_k \prec_x x_j$

对某个 x 和 T_k，T_k reads-x-from T_j 且 $x_i \prec_x x_j$

其他这些边叫做版本顺序边。如果多版本串行图不含有环的，则 MV 历史是单拷贝串行化的。

下面介绍一下数据库的一致性，数据库存在四个级别的一致性[40]，其描述如下：

表 6.1 数据的一致性

一致性名称	描述	串行化图的要求
严格一致性	每个查询看到的更新事务的一个串行顺序与更新事务的提交顺序序一样	更新事务顺序和提交顺序一样，禁止有环
强一致性	每个查询看到的更新事务的一个普通的串行顺序，它可以和提交顺序不一致	禁止有环
弱一致性	每个单独的查询看到的更新事务的串行顺序（其他查询看到的顺序不一定和这个查询看到的顺序相同）	禁止查询有环；禁止更新事务有环
更新一致性	每个查询都可以看到一个事务一致的数据库状态（该查询和它所看到的更新事务都是可串行化的）	如果去掉更新事务中的一个 read-wirte 边，不能消除查询环，则禁止查询环；禁止更新事务环

6.2 并发控制常用协议分析

在阐述了上文的基本概念后，本文将重点讨论并发控制问题。数据库管理系统必须控制事务的并发执行。并发控制的作用是正确协调同一时间里多个事务对数据库的并发操作，以保证数据库的一致性和完整性。并发控制是通过调度来确保事务的并发执行的效果等同于没有并发执行时的执行效果，也就是使事务的并发执行调度等价于事务的某个串行调度，从而确保数据库的一致性。

在传统数据库系统中，并发控制常采用两段锁协议、时间戳协议来确保并发调度的冲突可串行化。

6.2.1 两段锁协议

两段锁协议（Two-Phase Locking）协议的基本思想是：

一个事务加锁和解锁被严格地安排在两个阶段进行，分别称为增长段和萎缩段。

1 增长段（Growing Phase）：事务可以获得锁（对需存取数据项加锁），不允许释放锁（对已存取数据项解锁）；

2 萎缩段（Shrinking Phase）：事务可以释放锁，不允许再获得锁。

两段锁协议能确保并发调度是冲突可串行化的。这里称事务已获得它所需要的所有锁的时间点（即增长段的末端）为事务的锁点。因此，按锁点的先后顺序对并发调度事务集中的事务进行排序便可以得到并发事务集的可串行化排序。

两段锁协议的并发调度可能导致死锁，也可能导致串联夭折。为此，进一步又提出了严格的两段锁（Strict Two-phase Locking）协议和残酷的两段锁（Rigorous Two-phase Locking）协议，他们都是以牺牲并发程度为代价来避免串联夭折现象的。所谓严格的两段锁协议是指排他锁一旦获得则必须到事务提交时才能释放；而残酷的两段锁协议是指各种类型的锁（排他锁或共享锁）一旦获得都必须到事务提交才能释放。残酷的两段锁协议可以按事务的提交顺序对并发调度事务集中的事务进行排序，来实现并发事务集的可串行化。

锁协议的核心是对于冲突事务的每一对冲突的读写操作（不相容锁模式）谁先获得锁谁先执行，没有获得锁的事务只有等待已获得锁的事务释放锁后再去获得所需的锁。

6.2.2 多版本机制

在多版本并发控制（Multiversion Concurrency Control）机制中，每个 write(x) 操作都创建一个 x 的新版本。当事务发出一个 read(x)操作时，并发控制器选择 x 的一个版本进行读取。并发控制必须保证用于读取的版本的选择能保持可串行性。

对于每个数据项 x，有一个版本序列 $\langle x_1, x_2, \ldots, x_m \rangle$ 与之关联，每个版本 x_k 包含三个数据字段：

1 Content 是 x_k 版本的值。

2 W-timestamp(x)是创建 x_k 版本的事务的时间戳。

3 R-timestamp(x)所有成功地读取 x_k 版本的事务的最大时间戳。

下面的多版本时间戳排序机制（Multiversion Timestamp-ordering Scheme）保证可串行性。该机制运作如下：假设事务 T_i 发出 read(x)或 write(x)操作，令 x_k 表示 x 的版本，它具有小于或等于 $TS(T_i)$的最大时间戳。

1 如果事务 T_i 发出 read(x)操作，则返回值是版本 x_k 的内容。

2 如果事务 T_i 发出 write(x)操作且若 $TS(T_i) < R\text{-}timestamp(x_k)$，则系统回滚事务 T_i；另一方面，若 $TS(T_i) = W\text{-}timestamp(x_k)$，则系统覆盖 x_k 的内容；否则，创建 x 的一个新版本。

6.2.3 多版本两段锁

多版本两段锁协议（Multiversion Two-phase Locking Protocol）希望将多版本并发控制的优点与两段锁的优点结合起来。该协议对只读事务（Read-only Transaction）与更新事务（Update Transaction）加以区分。

更新事务强制执行残酷两段锁协议，即它们持有所有锁到事务结束。因此，他们可以按提交事务的次序进行串行化。数据项的每个版本都有一个时间戳，这种时间戳不是真正的基于时钟的时间戳，而是一个计数器，称为 ts-counter，这个计数器在事务提交时增加计数。

只读事务在开始执行之前，读取 ts-counter 的当前值来作为该事务的时间戳。只读事务在执行数据库操作时遵从多版本时间戳排序机制。因此，当只读事务 T_i 发出 read(x)时，返回值是时间戳小于 $TS(T_i)$的最大时间戳的版本的内容。

当更新事务读取一个数据项时，它在获得该数据项上的共享锁后读取该数据项的最新版本的值。当更新事务想写一个数据项时，它首先要获得该数据项上的排他锁，然后为数据项创建一个新版本。写操作在新版本上进行，新版本的时间戳最初置为∞，它大于任何可能的时间戳值。当更新事务 T_i 完成其任务后，它按如下方式完成提交：首先 T_i 将它创建的每一版本的时间戳设为 ts-counter 的值加 1；然后，T_i 将 ts-counter 增加 1。一次只允许有一个更新事务进行提交。

6.3 主动数据库事务特征

主动机制的引入使得数据库事务有了触发事务与被触发事务之分。触发事务与被触发事务之间存在三种关系[31]：

1 被触发事务是触发事务的部件（Is-Part-Of）。

2 被触发事务是触发事务的子事务（Is-Subtransaction-Of）。

3 被触发事务独立于触发事务（Is-Independent-Of）。

触发事务与被触发事务的匹配模式亦有三种：

1 立即执行（Immediate）；

2 延迟执行（Deferred）；

3 分离执行（Separate）。

立即执行意味着，不经过事务调度立即驱动被触发事务运行或经过事务调度并立即获得最高优先级而优先投入运行；延迟执行意味被触发事务由调度系统在一定的时间间隔内调度执行；分离执行意味着被触发事务的运行时机完全由它自身的特征决定，并由调度系统按通常的调度原则统一处理。上述三种关系与三种匹配模式的结合可得出如下表所示的五种执行方式：

表 6.2 匹配模式的结合

Coupling Relationship	Immediate	Deferred	Seperate
Is-Part-Of	IP	DP	×
Is-Subtransaction-Of	IS	DS	×
Is-Independent-Of	×	×	SI

×表示不存在这种结合方式

主动规则包括三个部分：触发规则的事件，规则被触发时要检查的条件，规则的触发条件为真时要执行的动作。事件可以是某关系上的数据修改（插入、删除或更新），条件是任意数据库查询，动作可能是产生警报执行数据操作、检索或回滚操作。假设所有的规则都是延迟执行的，也就是说，被触发的规则在事务结束时但在本事务中进行。这样，事务执行有两个连续部分：处理事务程序文本语句的部分，执行被触发规则的部分。方便起见，第一部分叫做常规部分，第二部分叫做触发部分。

6.4 采用传统并发控制机制存在的问题

考虑实际生活中的一个例子，假设有两个关系 Account(account-id，name，balance，...)表示存储银行帐户，Withdraw(account-id，date，...)用来跟踪银行帐户的取款记录。有两个事务程序 purchase 和 debit。当客户进行一项交易时 purchase 事务在关系 Withdraw 中插入一个元组，debit 事务读取 Withdraw 关系上从某日期开始的数据，并更新相应银行帐户的余额数。最后，定义一个主动规则：当对 Withdraw 进行插入时，取款金额大于银行帐户余额时，事务必须回滚。

无论何时执行 purchase，都会触发规则，评估条件，最后都要对 Account 进行读操作。这样 purchase 在执行时首先要获得 Withdraw 上的“排他锁”（purchase 本身要对 Withdraw 进行写操作），然后要获得 Account 上的“共享锁”。purchase

和 debit 一个要读 Account 一个要写 Account，如果两个同时执行就会产生冲突。MV2PL 中的更新事务采用残酷两段锁协议，只读事务采用严格两段锁协议，如果两个事务产生冲突，按照以上两种两段锁协议的处理方法，要先暂停一个事务直到另一个事务释放它的锁（提交或放弃）为止。这样，运行 purchase 事务增加了 Account 上的锁争用，阻碍了主动数据库系统的事务流量。采用 MV2PL 还存在对多个版本进行维护的问题，当数据库中数据存储量很大而且一个数据又有很多个版本要维护时，系统的负担将会很重。

在严格两段锁（S2PL）协议中，读数据操作在更新事务 T 的第二阶段，如果在 T 进行读数据操作的同时发生了事务 T'，T'要对这些数据项进行写操作，但 T'不能访问 T。这样，T'就要等到 T 释放锁才能进行些操作。假设 T 更新一个关系 R_1，并触发了一条需要读关系 R_2 的规则。同时运行的更新 R_2 的事务就会和 T 起冲突。这种情况下，关系 R_2 变成该应用程序中的锁争用资源，降低了应用程序的吞吐量。在以任务为中心的应用中，如联机事务处理（OLTP）中，这是不能承受的。

6.5 扩展的多版本两段锁协议

扩展的多版本两段锁协议对传统的 MV2PL 进行了优化和扩展，下面将其称之为 EMV2PL。本协议中，事务分为只读事务（查询）和更新事务，协议分别对只读事务和更新事务中的查询操作进行优化。只读事务记为 Read-only 事务，更新事务再细分为两个部分，第一部分执行事务程序文本中读或写操作的常规部分；第二部分在事务结尾，执行事务前面的部分触发的操作的触发部分。本文对 MV2PL 的查询操作分两个层次进行优化：

1 对只读事务中的查询操作的优化；

2 对更新事务触发部分中的查询操作的优化。

对于只读事务中的查询，通过降低数据的一致性的方法对其进行优化。对于更新事务，优化了其触发部分的查询。在执行查询操作前先检测是否存在一个未提交版本，如果存在则等待该版本提交后对其进行读取。

6.5.1 EMV2PL 协议

在本协议中分别对只读事务和更新事务中进行了优化，下面将分别描述这两个优化过程。

1. 只读事务的优化

在本协议中对更新事务划分为只读前更新事务和只读后更新事务两种类型。在串行顺序中，发生在查询（只读）之前的更新事务放在只读前更新事务集合中，串行顺序中发生在查询后的更新事务放在只读后更新事务集合中。只读事务可以读取只读前更新事务集里的事务创建的对象版本，从而实现读取对象的更新版本并减少对象的存储空间。

为了实现对对更新事务的划分，本文定义了如下规则，符合规则之一的更新事务 U 将被放入只读后更新事务集合，具体规则定义如下：

1 R 正在读一个数据项，U 要对该数据项加排他锁；

2 U 对一个数据项进行写操作，即对该数据项加了排他锁，R 要对该数据项进行读操作；

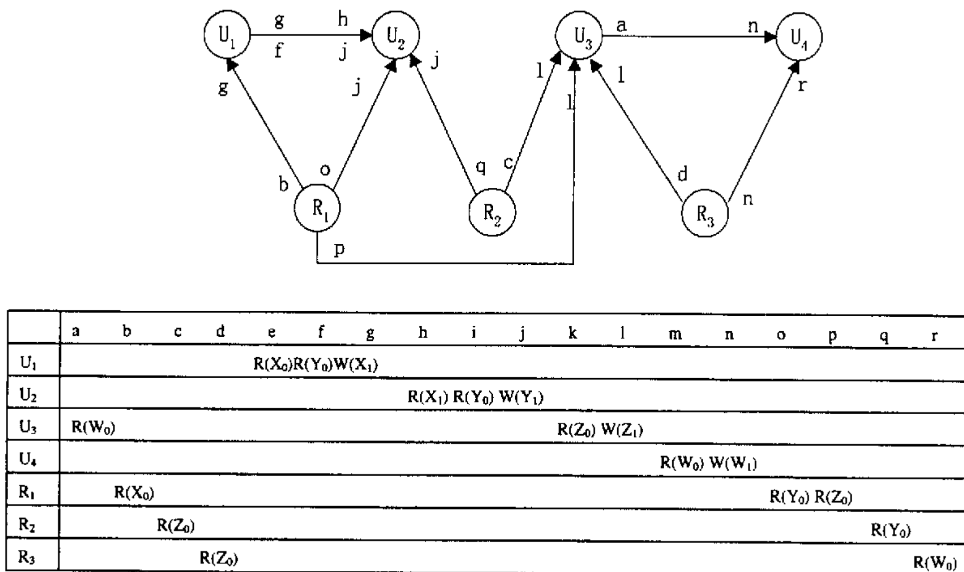
3 U 读取了一个由另一更新事务 U'创建的数据项版本，并且 U'是 R 的只读厚更新集成员。

这三条规则保证了 R 可以可以看到一个事务一致的数据库状态。其中，规则 1，2 保证了只读事务不会直接看到更新事务的局部影响，规则 3 保证了只读事务不会间接看到局部影响。规则 3 中包含的事务一定可以将更新事务的任何成员的局部影响传递出去，其原因如下：1）当更新事务 U 没有提交时，则在串行顺序中 U 一定出现在查询之后；如果更新事务 U 已经提交，且 U 不在只读后更新事务集合中，则 U 以后也不会出现在只读后更新事务集合中。2）一个更新事务 U 只能读到已经提交的数据，当更新事务读到一个版本时，通过检查该版本的时间戳，它立刻就能知道该版本的创建者（更新事务）是在每个（处于生命周期中的）查询操作之前创建的还是在它之后创建的。

基于这些规则的事务集可以实现事务的更新一致性，更新一致性保证每个查询都可以看到一个事务一致的数据库状态，也就是说，更新一致性保证该查询和它所看到的更新事务是可串行化的，而不保证所有事务都是可串行化的。

分析一下传统的 MV2PL 协议，可以认为 MV2PL 将所有在只读事务生命周期内发生的更新事务都放到只读后更新集中，它是遵循严格一致性的。

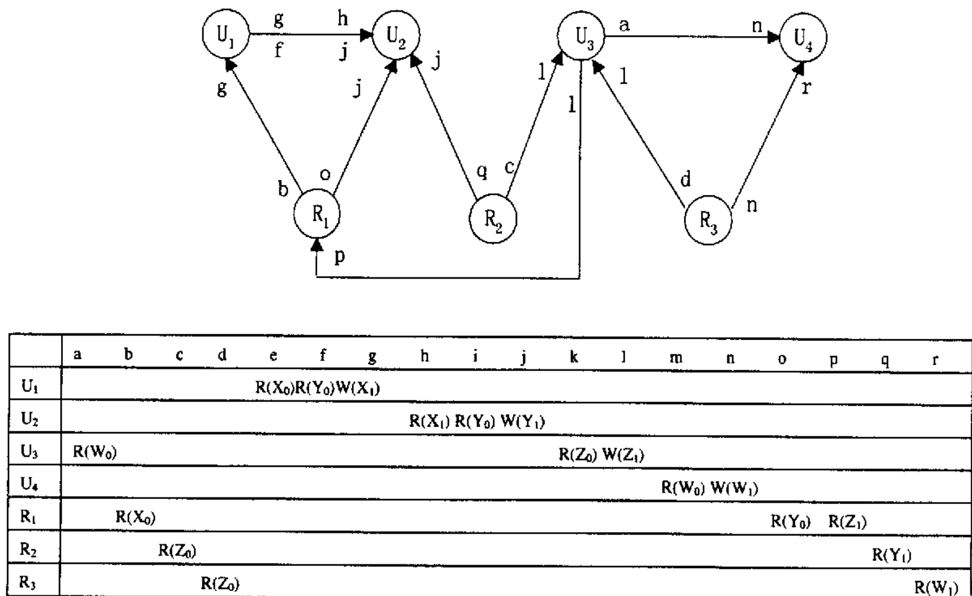
下面举例说明同样的只读事务在 EMV2PL 协议中读到了比 MV2PL 协议更新的版本，这里定义四个更新事务分别为 U_1、U_2、U_3、U_4，三个只读事务 R_1、R_2、R_3，使用图来表示以上事务的操作流程，从左到右按时间顺序排列。首先是 MV2PL 协议中的流程和串行化图：



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					$R(X_0)$	$R(Y_0)$	$W(X_1)$											
U_2								$R(X_1)$	$R(Y_0)$	$W(Y_1)$								
U_3	$R(W_0)$										$R(Z_0)$	$W(Z_1)$						
U_4													$R(W_0)$	$W(W_1)$				
R_1		$R(X_0)$													$R(Y_0)$	$R(Z_0)$		
R_2			$R(Z_0)$														$R(Y_0)$	
R_3				$R(Z_0)$														$R(W_0)$

图 6.1 MV2PL 串行化和操作流程图

在 MV2PL 协议中，因为只读事务 R_1 在更新事务 U_1 前创建，所以它读的是在 U_1 之前创建的 Z 的旧版本 Z_0，而在本文提出的 EMV2PL 协议中，其事务执行的流程和串行化图如下：



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					$R(X_0)$	$R(Y_0)$	$W(X_1)$											
U_2								$R(X_1)$	$R(Y_0)$	$W(Y_1)$								
U_3	$R(W_0)$										$R(Z_0)$	$W(Z_1)$						
U_4													$R(W_0)$	$W(W_1)$				
R_1		$R(X_0)$													$R(Y_0)$	$R(Z_1)$		
R_2			$R(Z_0)$														$R(Y_1)$	
R_3				$R(Z_0)$														$R(W_1)$

图 6.2 EMV2PL 的串行化和操作流程图

对只读事务 R_1 而言，U_1 符合上文提到的规则 1，U_2 符合规则 3，所以对 R_1 而言，其只读前更新事务集为{U_3}，只读后更新事务集为{U_1，U_2}，因此 R_1 读数据 Y 时只能读到 Y_0 版本而读数据 Z 时可以读 Z_1 版本，此时对所有的事务而言，Z_0 版本都不再被使用，所以其可被丢弃，这样节约了存储空间。从例子中可以看出 EMV2PL 协议能使只读事务读到了更新的数据的版本。

2. 更新事务的优化

本协议中，将更新事务细分为两个部分：常规部分和触发部分。对于常规部分中的查询，按照 MV2PL 的方式执行；对于触发部分的查询，添加一个函数用来在欲读数据的未提交版本刚刚提交时就及时获取该版本，而不会直接读取原本就存在的旧版本。在这里本文将只有常规部分的事务称为普通事务，包含常规部分和触发部分的事务称为主动事务。

处理普通事务时，读操作服从传统的严格两段锁协议 S2PL 协议。这里 write(x) 创建了一个 x 的新版本（如果是第一次写 x）。在提交之前，普通事务获得它的事务时间戳（记为 tn），将每个它的版本都和该时间戳相关联，释放它的锁。如图 6.3 所示：

操作调用	操作执行
begin(T)	φ
……	
read(x)	得到 x 上的共享锁 /*根据严格两段锁协议 S2PL 可能需要等待*/ 返回 x 最近的版本
……	
write(y)	得到 y 上的排他锁 /*根据严格两段锁协议 S2PL 可能需要等待*/ 创建一个 y 的新版本
……	
end(T)	从 TM 中获取 tn(T) commit(T)： 用时间戳为 tn(T)的版本更新数据库 释放锁

图 6.3 普通事务的执行

图 6.4 说明主动事务的处理过程。主动事务的常规部分处理过程和普通事务的处理过程一样。当执行到主动事务常规部分的末尾时，事务将触发部分开始的

信号发送到 TM，同时，得到一个事务时间戳 tn。以后的读和写操作按照下列步骤进行：read(x)调用函数 check_read(x)检查是否存在一个由小于调用者时间戳 tn 的事务创建的 x 的未提交版本；如果存在，则 check_read(x)等到未提交的事务执行结束再返回；主动事务读取时间戳小于或等于 tn 的最近版本的 x。提交之前，主动事务将其创建的每个版本和它的 tn 相关联，并释放它拥有的锁。

操作调用	操作执行
begin(T)	φ
……	
read(x)	得到 x 上的共享锁 /*根据严格两段协议锁 S2PL 可能需要等待*/ 返回 x 最近的版本
……	
write(y)	得到 y 上的排他锁 /*根据严格两段锁协议 S2PL 可能需要等待*/ 创建一个 y 的新版本
……	
start_trigger(T)	从 TM 中获取 tn(T)
……	
read(z)	check_read(z) /*等待*/ 返回时间戳≦tn(T)的 z 最晚版本
……	
end(T)	commit(T)： 用时间戳为 tn(T)的版本更新数据库 释放锁

图 6.4 主动事务的执行

本文给出的实现方法主要通过在触发部分中执行读操作之前通过一个检测函数判断是否可以读更新的版本。

check_read(x)函数需要的信息在锁表中可以找到。检查是否存在一个 x 的未提交版本需要检查 x 上是否存在排他锁。如果 check_read(x)必须等到创建 x 的事务提交以后，则它仅仅是等待锁的释放。这里说明这些机制可以很容易的在 MV2PL 锁管理器的基础上实现。

本文采用和文献[19]中描述的相似的锁管理器来实现。锁管理器的两个基本接口是 lock 和 unlock。他们只在 lock 管理器的数据结构中改变，该数据结构是一个和每个锁头有关系处理 id 的线性表，记为 list_pid。该线性表存放出

check_read()阻碍的主动事务的 pid(s)。

check_read(x)执行下列步骤：检查和 x 有关的锁头，看 x 是否为排他锁。如果是，则存在一个 x 的未提交版本，check_read()检查这个锁的拥有者的 tn 是否比调用者的 tn 小。check_read()将调用者的 pid 追加到锁头的 list_pid 上，并等候排他锁的释放。在它被唤醒之后，或者如果它没有受到阻碍，check_read 返回 ok 消息。这些操作不需要需要遍历锁请求队列，也不需要在锁请求队列中加入新的锁请求。

最后，对 unlock 接口做细微的修改，使事务释放排他锁，唤醒所有锁头的 list_pid 中 pid 中指示的进程，并将 list_pid 置为空。lock 接口没有改变。实际上，在队列遍历和存储空间上，check_read 比 lock 的开销小，因为它不在队列中插入新的锁请求，而是在线性表上追加调用者的 pid。系统中 lock 的数量减少了。

为了实现上文中的功能，本文定义了一个结构体 LOCK_HEAD 用来存储锁中信息，其结构体定义如下：

```
typedef struct lock_head {
    char Xsem; /* X means eXclude lock ,S means Share lock */
    char name[SIZE];
    pid_list pidList;
    lock_request * queue;
    struct lock_hash * next;
}LOCK_HEAD;
LOCK_HEAD *lockHash[HASHSIZE];
```

本文只考虑共享锁（S）和排他锁（X）方式。为了实现锁的根据名称快速查找，在文中定义了一个哈希算法并通过 getHashIndex()函数构造哈希函数，通过链表的方式解决冲突，下面是 check_read()函数的核心代码：

```
LOCK_REPLY check_read(char *name){
    long hashIndex;            /* the hash index of lock */
    LOCK_HEAD *head;   /* pointer to lock header block*/
    LOCK_REQUEST *request; /* this lock request block*/
    TransCB *me;       /* pointer to caller's transaction descriptor*/
    me=MyTransCB();
    hashIndex=getHashIndex(name);  /* find hash chain*/
    Xsem_get(&lockHash[hashIndex].Xsem);    /* get semaphore on it */
    head=lockHash[bucket].next;                          /* tracerse hash chain*/
    while((head!=NULL)&&(head->name!=name))
        head=head->chain;
    /* lock already taken in X mode*/
    if ((head!=NULL)&&(head->mode==X)) {
        sem_get(&head->Xsem);
        Xsem_give(&head_hash[bucket].Xsem)
        request=head->queue
```

```
        if (request->trane==me)
          return(LOCK_OK);
        else if (request->tran->tm<me->tn) {
          append(me->pid,head->pid_list);
              Xsem_give(&head->Xsem);         /* release semaphore on lock head*/
              wait();                 /* wait*/
          return(LOCK_OK);
        }
    else
        return(LOCK_OK);
    }
}
```

下面是unlock()函数的实现，在本文中增加了唤醒锁头中的等待进程，并将pid_list清空。

```
void unlock(struct *lockHead) {
    /* the common operator */
    ......
    lock_wake(&lockHead->pid_list); /* wake up the pid in the queue */
    lock_free(&lockHead->pid_list);  /* clean the pid_list */
}
```

6.5.2 死锁分析

显然，EMV2PL会产生死锁现象，因为它使用S2PL序列化普通事务和主动事务的常规部分，并且check_read()函数根据具体需要也会存在等待某个写操作完成，这些都是死锁产生的潜在因素。

如果存在一个事务集，该集合中的每个事务在等待该集合中的另一个事务，那么我们说系统处于死锁状态[24]。更确切地说，存在一个等待事务集{T_0，T_1，……，T_n}，其中T_0正等待被T_1锁住的数据项，T_1正在等待被T_2锁住的数据项。在这种情况下，没有一个事务能取得进展。

但是，一旦一个主动事务开始执行它的触发部分，它就不可能再涉及死锁问题了。这可以很简单的说明：

假设T_i是执行主动事务触发部分的事务。已经获得了它的tn。死锁意味着在事务的等待图（存在一条$T_i \to T_j$边，当且仅当T_i受到T_j阻挡）中存在环。如果T_i属于一个环或者等待一个有环的事务，则它肯能产生死锁。首先说明T_i不属于环：令$T_i \to T_{i1} \to T_{i2} \to ...T_{ik} \to T_i$为一个环。因为$T_i$等待$T_{i1}$，已经获得它的tn且tn($T_i$) > tn($T_{i1}$)（只有这种情况下check_read才有可能阻塞T_i）。然后，T_{i1}不

可能是一个普通事务，因为一旦普通事务获得其 tn（也就是说，已经超出它的锁的生命周期），它不可能等待另一个事务 T_{i2}。这样，T_{i1} 就是一个执行触发部分的主动事务。将此结论应用于环的每个结点，我们得到所有节点都是主动事务，且 $tn(T_i) > tn(T_i)$，矛盾，所以 T_i 不可能属于环。

现在假设 T_i 不属于环，如果 T_i 和一个属于环的事务 T_j 之间有一条路径，则 T_j 也是一个执行到触发部分的主动事务。因为 T_j 属于环，所以 T_j 不可能是执行到触发部分的主动事务。

第七章　总结和展望

7.1　本文总结

本文基于主动数据库技术的基本思想并结合面向对象的思想对主动数据库的知识模型、执行模型进行研究及设计。重点讨论了主动数据库的主动规则和执行模型。

在主动规则方面，本文在对ECA规则分析的基础上针对主动规则中的事件检测论述了一种基于染色Petri网的复合事件检测的方法。首先描述了用于检测复合事件的Petri网的定义Petri-DS，其中用点Place模拟事件模式，变迁Transition及守卫表达式模拟对组成复合事件的成员事件的各种限制，不同的复合事件具有不同的守卫表达式。接着论述了Petri-DS的动态行为。为了说明该检测方法，本文给出了三种复合事件来说明，分别是同时发生复合事件、不发生复合事件和多次发生复合事件。此外，本文用面向对象编程语言给出了该检测方法的实现。

在执行模型方面，本文在对传统执行模型的基础上分析了常见的几种并发控制协议，分别为：两段锁协议、多版本机制和多版本两段锁协议。对传统的MV2PL协议进行改进，提出了一种更适合主动数据库的事务模型。在该事务模型中，分别对只读事务和更新事务进行了优化。在对只读事务的优化中，将查询所要求的一致性从原来的严格一致降低至更新一致，使只读事务可以读到更新的数据版本。在对更新事务的优化中，本文根据主动数据库的事务特点重新定义事务的分类，将更新事务细分为普通事务和主动事务。对主动事务的读操作加入了检测是否存在未提交数据版本的函数。通过这样的优化，查询操作可以读到比原来MV2PL协议下更新的数据版本，旧版本的数据可以提前丢弃，不但节省了存储空间还降低了维护过量旧版本数据的开销。本文针对主动数据库中大量的主动事务提出优化的设计方案并给出其中的部分实现。

7.2　工作展望

由于时间和客观条件等因素，本文的染色Petri网检测复合事件的实现部分仅仅是实现了一个简单模型，要形成一个成熟可用的模型还需要更进一步的研究和细化。

主动数据库从一出现就不断飞速的发展，因为主动数据库以统一且方便的机制提供主动服务功能，能适应现实世界应用中存在着的各种主动性需求。主动数

据库技术目前正处于研究与实现阶段，主动规则的实现是一个复杂的过程，它包括规则管理，并发控制、效率评估、规则编译、应用交互等复杂技术。

主动数据库存在以下研究方向如事件代数的应用及实现，主动规则的优化等。随着人工智能和面向对象思想的进一步研究，主动数据库也将步入一个新的研究发展阶段。

参考文献

[1] Norman W.Paton, Oscar Diazf . Active Database System

[2] S.Chakravarthy, D. Mishra. An Event Specification Language (Snoop) For Active Databases and its Detection; Techn. Report UF-CIS-TR-91-23, University of Florida, September 91.

[3] U.Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraints. ACM Sigmod Record, 17(1), March 88.

[4] K.R.Dittrich, S.Gatziu. Time Issues in Active Database Systems. Proc. Intl. Workshop on an Infrastructure for Temporal Databases..Arlington, Texas, June 93.

[5] IEEE Bulletin of the TC on Data Engineering, Vol. 15, No. 1-4, Special Issue on Active Databases, December 92.

[6] S.Gatziu, A.Geppert, K.R. Dittrich. Integrating Active Concepts into an Object-Oriented Database System; Proc. of the 3. Intl. Workshop on Database Programming Languages, Nafplion, August 91.

[7] S.Gatziu, K.R.Dittrich. SAMOS: an Active Object-Oriented Database System

[8] S.Gatziu, K.R.Dittrich. Events in an Active Object-Oriented Database System; to appear in Proc. of the 1. Intl. Workshop on Rules in Database Systems, Edinburgh, August 93.

[9] N.H.Gehani, H.V.Jagadish, O.Shmueli. Event Speci-fication in an Active Object-Oriented Database. Proc.ACM SIGMOD, June 92.

[10]N.H.Gehani, H.V.Jagadish, O.Shmueli. Composite Event Specification in Active Databases: Model & Implementation. Proc. of the 18th Intl. Conf. on Very Large Data Bases, Vancouver, August 92.

[11]K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol.1, EATCS 92.12. S.B.Navathe, A. Tanaka, S.Chakravarthy; Active Database Modelling and Design Tools: Issues, Approach, and Architecture.

[12]R.Agrawal, M.J.Carey, M.Livny. Concurrency control performance model:Alternatives and implication. ACM Transactions on Computers and Systems,12(4):609-654,December 1987.

[13]D.Adler, B.Dageville, Kam-Fai Wong.SIM: A C-based SIMulition Package. ECRC-92-27i,1992.

[14]D.Agrawal, V.Krishnaswamy. Using mutiversion data for noninterfering execution of write-only transactions.Proc.ACM SIGMOD Internationa Conference on Management of Data, Denver, Colorado.

[15]Agrawal, S.Sengupta. Modular synchronistion in multiversion database:Version control and concurrency control. Proc. ACM SIGMOD Internatioal Conference on Management of Data,Portland,Oregon, pages 408-417

[16]P.M Bober, M.J.Carey. On mixing queries and transaction via multiversion locking .Proc International Conference on Data Engineering,Tempe,Arizona,pages535-545,February 1992

[17]P.A.Bernstein, V,Hadzilacos, N.Goodman. Concurrency Control and Recovecry in Database Systems.Addison-Wesley Publishing Company,1987

[18]Michael J. Carey, On Transaction Boundaries in ActiveDatabases: A Performance Perspective IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 3, NO. 3, SEPTEMBER 1991

[19]J.Gray, A.Reuter. Transaciton Processing .Morgan Kaufman ,1993

[20]Umeshwar DayalActive Database System，1994.9

[21]N. Bassiliades, I. Vlahavas.DEVICE Compiling Production Rules into Event-Driven Rules Using Complex Events

[22]S.Chakravarthy. ECA Rule Integration into an OODBMS.Architecture and Implementation，1994.5

[23]Jennifer Widom. Active Database Rule System，1992

[24]Avraham Siberschatz 等.《数据库系统概念》，机械工业出版社 2002

[25]李昭原 .《数据库技术新进展》，清华大学出版社，1997

[26]宗平. 基于关系型数据库的主动信息系统研究与实现，计算机工程与应用，2001.37

[27]万常选. 一种实时数据库的多版本两段锁并发控制协议，江西师范大学学报，2000.4

[28]陈宁，董继润，陈安. 面向对象主动数据库 SDAOODB 的技术与实现，计算机工程，1998.3

[29]张杰敏. 有关高级数据库系统中的并发控制问题，华北工学院学报，1998.19.3

[30]左万利. 事务框架下主动规则的并发控制算法，东北大学自然科学学报，2001.04

[31]卢炎生. 主动实时数据库事务及其处理，计算机研究与发展，1998.35.2

[32]徐长醒. 基于图的主动数据库规则模型 ERG，华中理工大学学报，2000.28.11

[33]蒋长俊. Perti 网理论与方法研究综述，控制与决策，1997.11

[34]周志逵等. 主动面向对象数据库系统中主动规则的研究，北京理工大学学报，1998.12

[35]奚峡. 主动数据库系统的规则机制及应用，Computer Era 2003.7

[36]冯玉才. 《数据库系统基础（第二版）》，华中理工大学出版社，1993.7

[37]杨芙清等. 基于事件驱动的主动对象模型，软件学报，1996.3

[38]卢炎生等. 一个主动实时数据库管理系统的事务处理子系统，计算机工程与应用，1996.1

[39]竺卫东等. 主动的面向对象数据库管理系统中的事件和规则的设计，计算机研究与发展，1998.5

[40]http://www.postgres.org

[41]http://www.ccf-dbs.org.cn/

[42]Pandurang Nayak，Anoop Gupta. Comparison of the Rete and Treat Production Matchers for Soar

[43]Eric N.Hanson. Gator: A Discrimination Network Structure for Active Database Rule Condition Maching，1993

[44]孔兵等. 事件驱动系统的 Petri 网建模，云南大学自然科学版，2003.25（13－16）

[45]王珊等. 《数据库大词典》 第 22 章主动数据库，上海科技文献出版社，1995

[46]林琪，贺松云. 逻辑推理中的 Petri 网应用，计算机应用研究，1998
[47]袁崇义.《Petri 网原理》，电子工业出版社，1998

致谢

衷心感谢我的导师宗平教授！感谢导师在我研究生阶段中所给予悉心关怀和耐心指导，并提供了很好的学习、研究的环境和机会，我的每一点进步都与导师的言传身教分不开。导师求实的治学态度、缜密的思维以及广博的学识将使我受益终生。

感谢计算机学院的其他老师，特别是费玉奎老师为我的论文提出了宝贵意见和珍贵资料，他们在平时的学习中给予很多帮助和启发。谨在此向他们表示衷心的感谢。

感谢同学卞海红、吉建峰、高军、朱辉、印梅、李婷婷、吕小燕等同学，谢谢他们为论文的写作提供了许多有价值的建议和修改意见，同时对他们三年以来的相互勉力和无私帮助表示感谢。

感谢计算机及信息工程学院 2002 级研究生周叶丹、邓赛峰、孟庆强、田震生等同学。文中的许多思想源于平时和他们的交流和讨论。

感谢所有的师妹、师弟们，与他们一起愉快地参与项目的开发，相互交流，使我受益非浅，给了我很多帮助和启发。感谢他们和我们共同创造了一个活泼、团结、向上的学术氛围，在此对他们表示最诚挚的谢意。

最后，我还要特别感谢我的父母，在我三年艰苦的学习和研究期间，他们一直对我的学习和生活等各方面无微不至的关心和支持，我的成绩也凝聚了他们的心血。

潘劼

2004．3 月于南京