Nanjing University of Aeronautics and Astronautics
The Graduate School
College of Information Science and Technology

Formal Analysis and Verification of a Transaction Coordination Protocol named WS-TX for Web Services

A Thesis in

Computer Science and Technology Engineering

by

LI Xiang

Advised by

Prof. HUANG Zhi-qiu

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Engineering

December, 2009

承诺书

本人声明所呈交的硕士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：李帅

日　　期：2010.3.15

摘 要

Web 服务作为一种新的计算模型正受到越来越多的关注，保证 Web 服务组合执行结果的一致性和可靠性是 Web 服务面临的重要挑战之一。将事务机制应用到 Web 服务领域是寻求解决 Web 服务组合执行结果一致性和可靠性问题的重要手段。针对 Web 服务自治、跨组织以及长周期等特点，工业界基于传统事务机制提出了新的事务处理协议和标准。其中，Web 服务事务协调协议 WS-TX 由于利用了现有的和正在制订的标准，能够融入 Web 服务架构，得到了业界的广泛关注。

由于 WS-TX 协议采用自然语言描述，缺乏严格的语义，很多学者使用形式化方法对其进行研究，验证协议本身的正确性。然而，协议的正确并不代表使用协议的应用场景也能满足事务特性。本文采用 Pi-演算对 WS-TX 协议应用场景进行建模，使用符号化模型检测工具 NuSMV2 从“安全性”和“活性”两方面对模型进行验证和分析。主要工作分为以下三个方面：

1. 针对 WS-TX 应用场景的并发特性，采用 Pi-演算对其进行建模。给出了 WS-TX 协议应用场景与 Pi-演算元素的映射关系，并在此基础上给出了建模规则。通过对应用了 WS-TX 中 WS-AT 协议的银行转帐场景和 WS-BA 协议的旅行安排场景的协调过程进行 Pi-演算建模，验证了建模规则的可行性。

2. 由于现有 Pi-演算模型检测工具存在检测能力不足的问题，本文将 Pi-演算模型转化为 SMV 程序（有限状态自动机模型），利用 NuSMV2 进行检测。根据标号变迁系统到 Kripke 结构的映射关系找出 Pi-演算模型到 SMV 程序的转换规则，并实现了自动转换工具 PiCal2NuSMV.

3. 分析了 WS-TX 中两个事务协议 WS-AT 和 WS-BA 应用场景的安全性和活性，用计算树逻辑 CTL 进行描述，并采用 NuSMV2 检验银行转帐和旅行安排的业务协调过程是否满足这些性质，验证结果表明应用场景满足事务特性并且协议本身是可靠的。同时，进一步讨论和分析了 NuSMV2 生成的反例。

关键词： Web 服务，事务处理，形式化方法，Pi-演算，协议建模，模型检测

Abstract

As a new computing model, Web Services are becoming more and more attractive. It is one of the most important challenges of Web Services to ensure that the result of running composited web services is consistent and reliable. Transaction mechanism is considered to be applied in web services composition to generate consistent and reliable outcome. Nowadays, new transaction protocols and standards based on traditional transaction mechanisms are proposed in consideration of Web Services' characteristics such as autonomy, heterogeneous, long period and so on. Among the protocols, a coordination protocol named WS-TX is a commonly concerned protocol for it has used the existing and developing standards and can be actioned as a component in the web services architecture.

As we know that WS-TX is described by natural language and lacks of strict semantics, lots of researchers apply formal method to do research on WS-TX. Howerver, the correctness of protocol doesn't imply that the scenarios applying WS-TX meet transaction needs. Thus pi-calculus is used to model the scenarios applying WS-TX and a symbolic model checking tool named NuSMV2 is adopted to check whether the scenarios meet safety and liveness. The main research works can be listed as follows.

First of all, considering the concurrency of WS-TX scenarios, pi-calculus is adopted and the mapping relation of elements of WS-TX scenarios and pi-calculus is proposed. Furthermore, modeling rules of pi-calculus are studied. In order to verify the practicability of mapping rules, a scenario about account transfer applying WS-AT which is a concrete coordination protocol in WS-TX and another scenario about travel arrangement applying WS-BA which is another coordination protocol in WS-TX are presented and modeled by pi-calculus.

Secondly, for existing model checking tools for pi-calculus can't meet our command, pi-calculus models are translated into finite state machine described by SMV language, which is the input of NuSMV2. The translating rules from pi-calculus models to SMV language code are proposed derived from the mapping from labeled transition system to kripke structure. Based on the rules, an automatic translating tool named PiCal2NuSMV is designed and implemented.

Finally, safety and liveness are defined respectively in WS-AT scenarios and WS-BA scenarios, which are then represented in CTL (Computation Tree Logic). NuSMV2 is used to checking whether the coordination processes of account transfer and travel arrangement meet the properties described in CTL. The result of model checking shows that the scenarios applying WS-TX have transaction

properties and the WS-TX protocol itself is reliable. A method about how to analyse conuterexamples generated by NuSMV2 is presented lastly.

Keywords: Web Services, transaction processing, formal method, pi-calculus, protocol modeling, model checking

目 录

图表清单

注释表

Web 服务(Web Services) —— 部署在 Web 上的可以通过 SOAP 协议进行交互的平台无关组件。

SOAP(Simple Object Access Protocol) —— 简单对象访问协议，是一种基于 XML 的分布式计算协议，支持分布式系统中的消息传递和远程过程调用。

WSDL(Web Services Description Language) —— Web 服务描述语言，是一种基于 XML 的 Web 服务接口描述语言。

UDDI(Universal Description, Discovery and Integration) —— 统一描述、发现和集成协议，是一种服务描述和发现的标准规范。

Pi-演算(Pi-Calculus) —— 一种用来描述并发、移动系统的进程代数。

WS-TX —— OASIS 的 Web 服务事务技术委员会制定的 Web 服务事务标准，包括 WS-Coordination、WS-AtomicTransaction 和 WS-BusinessActivity 三部分。

WS-C(Web Services Coordination) —— Web 服务协调，是可扩展的事务协调框架。

WS-AT(Web Services AtomicTransaction) —— Web 服务原子事务，在 WS-C 的基础上定义了一个原子事务协调类型用于协调具有“All-or-None”属性的活动。

WS-BA(Web Services BusinessActivity) —— Web 服务业务活动，在 WS-C 的基础上定义了两种业务活动协调类型用于协调活动，这些活动使用业务逻辑来处理业务流程活动执行过程中发生的异常。

NuSMV2 —— 一种开源的模型检测工具，用于验证有限状态系统是否满足时序逻辑描述的需求，属于符号模型检测系统。

SMV语言 —— 模型检测工具NuSMV2的系统描述语言。

LTL(Linear Temporal Logic) —— 线性时序逻辑。

CTL(Computation Tree Logic) —— 计算树逻辑，属于分支时序逻辑。

第一章 绪论

1.1 课题研究背景及意义

在现代社会中，信息广泛渗透到人类社会和经济生活中，人类经济社会的发展不仅仅依赖物质、能量等资源，同时也越来越多地依赖于信息资源，关于如何使用、管理和保护这些信息的计算机技术对人类和现代社会变得至关重要。这些信息数据集中描述或刻画了现实世界中的某些现象或活动的状态和演化，为了正确地反映现实世界，许多信息数据要求一起被访问和修改，因此必须保证相关数据的一致性，事务处理技术为这些关键信息资源的有效管理和使用提供了必要的机制和措施[1]。事务是指对物理和抽象的应用状态上数据访问和更新的操作集合，要么全部成功，要么全部失败，并保证并发执行的事务彼此互不干扰[2]，这种 “All-or-None” 的语义是传统数据库访问的基础。

随着 Internet 的普及和基于 Internet 应用的延伸，以 Web 服务为基础的面向服务计算体系架构 (Service-Oriented Architecture, SOA) 应运而生。Web 服务作为一种崭新的分布式计算模型，成为 Web 上数据和信息集成的有效机制[3]。为了完成客户的请求，多个 Web 服务需要通过编排或编制[4]的方式组合在一起完成新的功能。由于 Web 服务的最终目标是允许网络上不同类型的服务能够协同合作[5]，这就要求 Web 服务组合执行具有一致性和可靠性，并能及时解决运行时发生的各种异常。这类问题可以借助事务机制来解决[6]。虽然传统的事务技术在数据库系统和分布式应用中得到了广泛的研究和应用，但由于 Web 服务自身的松散耦合性、运行时间长、开放式环境，使其在网络的时延性、系统的可靠性和一致性方面面临新的挑战，使得传统的事务模型无法直接用到 Web 服务中[7]。因此，由 Web 服务组合而成的业务流程如何保证应用一致性，即在失败、异常等情况下，组合服务的执行能结束于一致状态，是 Web 服务组合应用的一个关键问题，也是当前研究的热点[1]。

BPEL(Business Process Execution Language，业务过程执行语言)[8]作为 Web 服务事实上的标准，并没有描述分布于多个供应商和平台的参与者之间的协调一致性[9]，而是建议在这种分布式业务环境中使用 WS-TX 协议[10]来实现对业务的参与者提供注册或撤销等分布式协调的通知。这表明用 WS-TX 协议来解决分布式业务的事务问题是未来的发展方向。WS-TX 协议是 OASIS(Organizatioan for the Advancement of Structured Information Standard，结构化信息标准促进组织)标准，描述了一个为协调分布式应用程序行为提供协议支持的可扩展框架，这种协调协议可被用于支持多个业务流程实例的运行，它们在分布式事务的输出上有一致性的要求。

WS-TX 协议为分布式业务事务处理提供了解决方法。然而，如何保证 WS-TX 协议应用的正确性是将 WS-TX 协议应用于分布式业务中必须解决的问题，本文将对这一问题展开研究。

1.2 Web 服务事务的国内外研究现状

如何保证 Web 服务业务流程的事务特性是 Web 服务研究中的热点问题，得到了学术界和工业界的关注。学术界的研究者主要从以下两个方面对 Web 服务事务问题进行研究：

（1） 学术界的研究者吸取了其它一些松耦合环境下事务模型的做法，适当调整并放松了传统事务 ACID（Atomicity，Consistency，Isolation，Durability）特性的要求，先后提出了各种扩展事务模型，实现松散、长运行系统中的事务管理。面向长事务的 Sagas[11]项目第一次提出了补偿事务的概念，系统在失效时将按照逆序执行所有已安装的补偿处理块回退到初始状态。柔性事务模型[12]是一种对操作的原子性和隔离性要求较低的松耦合扩展事务模型，子事务之间通过偏序和优先两种依赖关系组合在一起。Zhang 等用可重试和可补偿两个属性来刻画服务的事务能力，进一步研究了柔性事务的一致性问题[13]。上海交通大学的唐飞龙博士等提出了一个能够同时协调短事务和长事务的模型，给出了协调算法及恢复机制[14]。国防科学技术大学的任怡博士等提出了事务执行阶段的依赖异常处理机制，保证整个事务不失败[15]。中科院软件研究所的黄涛教授等从保障服务协作可靠性角度，给出了一个基于应用语义的松弛事务模型[16]。

（2）用软件形式化方法对 Web 服务合成和事务进行研究已经成了国际上学术研究的热点。在现有对 Web 服务事务形式化建模研究中，基于自然描述并发系统的进程代数方法的研究较多。Bocchi 等提出了 πt-calculus[17]，对 Pi-演算作了扩展，加入事务模块控制算子以支持长事务。Butler 等在传统的进程代数上加入补偿算子，提出支持描述事务处理的语言 StAC[18]，是最先将补偿和进程控制紧密结合的进程演算之一。

许多年以来，为了解决传统的事务处理的语义和协议不合时宜的问题，工业界由重要 IT 厂商组成的标准化组织 OASIS、OMG 和 W3C 等技术委员会，一直致力于制定分布式业务事务处理协议的业界标准，主要包括 WS-CAF(Web Services Composition Application Framework)[19]、BTP(Business Transaction Protocol)[20]和 WS-TX[19]。WS-TX 用于 Web 服务，利用了现有的和正在制订的标准，比如 WSDL(Web Services Description Language)[21]、WS-Addressing[22]、WS-Security[23] 和 WS-Policy[24]等。这使 WS-TX 集中在 Web 服务环境，并且简化了规范，还可以把它们作为一个组件放在 Web 服务架构中，Web 服务架构在性能优化方面的任何进步都可以自动为它利用。因此，WS-TX 成为业界广泛关注的标准。目前已有对 WS-TX 的实现，其中包括 Apahe 的 Kanlula[25]开源项目、Sun 的 WSIT(Web Services Interoperability Technologies)开源项目[26]和微软的 WCF(Windows Communication Foundation)项目(已包含于.NET Framework 3.0 产品)，它们都已经实现了 WS-Coordination 和 WS-BusinessActivity 协议，并且宣称在不久将加入 WS-BusinessActivity 协议的实现。

工业界提出的 Web 服务事务标准有大的软件厂商的支持，很容易得到推广和应用。然而工业界已有的关于事务的协议标准，大多采用自然语言表达，没有做过形式化的表示和分析，规

范中存在定义不清晰的细节，缺乏精确的语义。针对这个问题，已经有研究用软件形式化方法对工业界的事务协调协议进行了相关研究。Berger 等用异步 Pi-演算对分布式事务处理中经典的两阶段提交协议进行了形式化验证[27]。上海交通大学的戚正伟博士等提出了一种细胞膜演算的形式化方法[28]，利用对象和细胞膜的反应过程，简明地描述了 Web 服务事务处理过程，并分析检验了早期的 WS-AtomicTransaction 和 WS-BusinessActivity 协议。这些说明用形式化方法描述和分析事务协调过程是可行且有意义的。然而，已有研究大部集中在验证事务协议（如 WS-AtomicTransaction 和 WS-BusinessActivity）本身的正确性，并没有验证将事务协议应用到实际场景中是否也能满足需求的事务特性。

为了支持分布式业务事务，传统方法一般如同 Hagen 等通过跨组织工作流协同，组建可靠的工作流管理系统来保证跨组织业务应用的一致性[9]。WS-Coordination 就是这样一种保证业务应用一致性的协议框架，而 WS-AtomicTransaction 和 WS-BusinessActivity 则用来保证协调过程具有事务特性。Curbera 等将各 BPEL 业务作用域之间的补偿对应关系表示成 WS-TX 的协调上下文，提出协调组合 Web 服务的框架[29]。Yang 等以 BPEL 和 WS-TX 为基础，扩充 WS-BusinessActivity 协议，将业务补偿逻辑作为协调逻辑的一部分，实现业务的柔性多补偿机制[30]。Schäfer 等基于向前恢复策略，进一步扩充 WS-TX 的协调框架，支持业务补偿逻辑和协调逻辑相分离的灵活的补偿操作[31]。这些研究都没有提供对协调场景协调正确性的验证方法。

针对上述不足，本文将对使用 WS-TX 协议的应用场景是否满足系统需求的事务特性进行研究，提供 WS-TX 协议应用场景是否满足事务特性的验证方法，以保证使用 WS-TX 协议的应用能够达到系统设计的需求。对 WS-TX 协议应用场景进行形式化分析与验证既可以验证应用场景是否满足事务特性从而指导场景的设计与实现，又可以验证协议本身的可靠性。

1.3 本文研究内容和组织结构

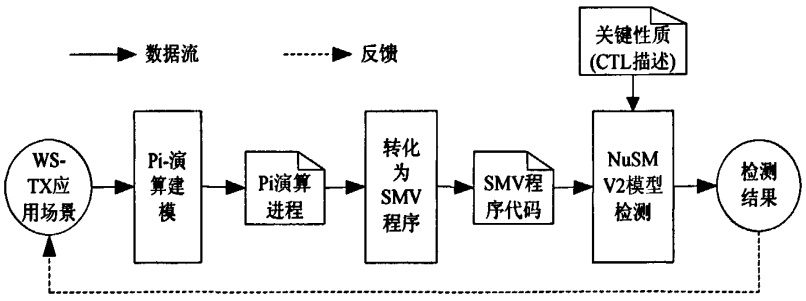

图 1.1 WS-TX 协议应用的 Pi-演算形式化验证方法

WS-TX 协议为分布式业务事务处理提供了解决方法，得到了产业界和学术界的普遍关注。在对 WS-TX 协议的研究和应用中，对 WS-TX 具体应用是否满足事务特性的研究较少，本文将对此进行研究。图 1.1 给出了本文研究的方法：WS-TX 协议应用的 Pi-演算形式化验证方法。

首先对 WS-TX 应用场景进行 Pi-演算建模，再将 Pi-演算进程表达式转化为 SMV（模型检测工具 NuSMV2 的系统描述语言）程序代码，得到使用 SMV 语言描述的 WS-TX 应用场景模型。然后使用 CTL(Computation Tree Logic，计算树逻辑)对 WS-TX 应用场景应满足的性质进行描述，并用 NuSMV2 检测系统是否满足性质。如果不满足，则可反馈到 WS-TX 应用的设计并进行修改，甚至可以查找出 WS-TX 协议的模糊不清或待完善之处。

本文的主要研究内容分为六章，各章节内容概述如下：

第一章 绪论。阐述了本文课题研究的背景及意义。分析了 Web 服务事务的国内外研究现状，并给出了本文的主要研究方法和内容。

第二章 Web 服务事务问题描述。概述了 Web 服务及相关技术，阐述了什么是 Web 服务事务问题，以及 Web 服务事务问题的解决方法。然后对目前广为关注的 WS-TX 协议进行了分析。

第三章 基于 Pi-演算的 WS-TX 协议应用建模。首先分析了 Pi-演算以及为什么要使用 Pi-演算对 WS-TX 协议应用场景进行建模。其次对 WS-TX 协议应用的 Pi-演算建模方法进行了研究，最后根据 Pi-演算建模方法对 WS-TX 协议的应用场景进行建模。

第四章 Pi-演算模型的 SMV 程序表述。论证了使用模型检测工具 NuSMV2 的原因，并对 Pi-演算进程到 SMV 程序代码的转换规则进行了研究，实现了自动转换工具 PiCal2NuSMV.

第五章 基于 NuSMV2 的 WS-TX 协议应用模型检测及分析。分析了 WS-TX 协议应用场景的性质，并用 CTL 进行了描述。将 WS-TX 协议应用场景的 Pi-演算模型转化为 SMV 程序代码，使用 NuSMV2 检测其是否满足性质，并对检测结果进行了分析。

第六章 总结与展望。对本文论点和研究工作进行了总结，并指出本文课题研究的未来工作。

本文的内容组织结构如图 1.2 所示。

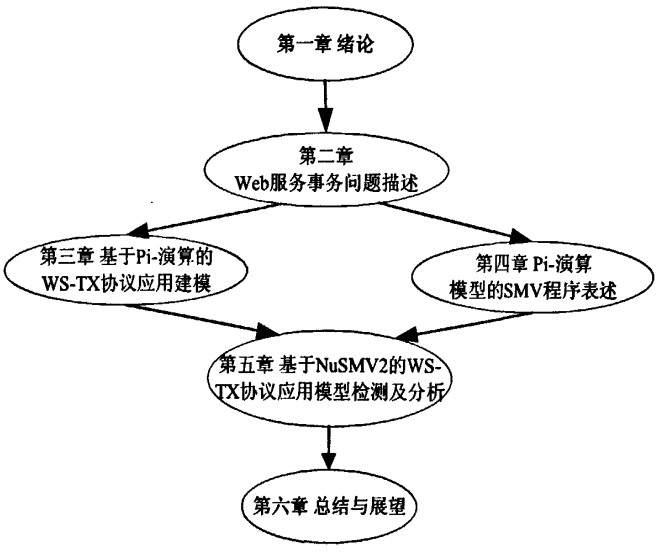

图 1.2 论文组织结构

第二章 Web 服务事务问题描述

Web 服务作为一种新的软件形态将得到越来越广泛的应用，保证 Web 服务组合执行结果的一致性和可靠性是研究 Web 服务的重要课题之一，这类问题可以归为 Web 服务的事务问题。本章将对 Web 服务事务机制进行研究，分析 Web 服务事务的解决方法及面临的挑战。

2.1 Web 服务及其相关技术

维基百科将Web服务（Web Services）定义为：Web服务是一种面向服务的架构的技术，通过标准的Web协议提供服务，目的是保证不同平台的应用服务可以互操作。根据W3C的定义，Web服务应当是一个软件系统，用以支持网络间不同机器的互动操作。Web服务通常是许多应用程序接口（API）所组成的，它们透过网络的远程服务器端，执行客户所提交服务的请求。Web服务可以使用任何编程语言开发，并被部署于任意平台上。Web服务使用XML(Extensible Markup Language，可扩展标识语言)来描述它们的接口和消息，因此相互间可以通信。基于XML的Web服务通过标准的Web协议进行交互。这些协议使用XML接口和消息，使得所有的应用程序都能识别。但是XML本身并不能保证交互的简易性，应用程序需要标准的格式和协议以保证能正确地解释XML。所以三种基于XML的技术应运而生，它们被认为是Web服务事实上的标准：

(1) 简单对象访问协议(Simple Object Access Protocol, SOAP)[32]，定义了Web服务标准的交互协议。

(2) Web服务描述语言(Web Services Description Language, WSDL)[21]，定义了标准的描述Web服务的机制。

(3) 统一描述、发现和集成(Universal Description, Discovery and Integration, UDDI)[33]，提供了注册和发现Web服务的标准机制。

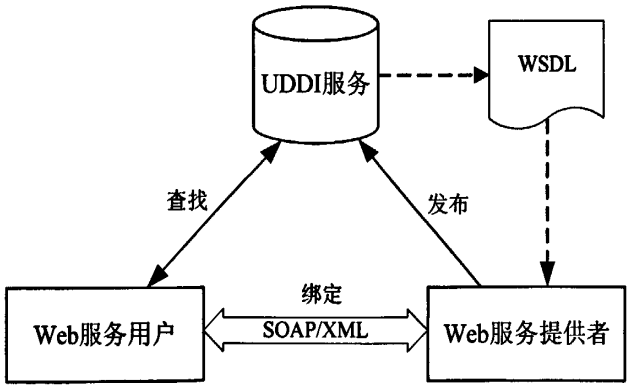

图 2.1 Web 服务基本架构

图 2.1 描述了 Web 服务的基本架构。当一个服务提供者想将服务提供给用户时，它通过

WSDL 描述服务并在 UDDI 服务商处注册服务。UDDI 服务商将提供指向 WSDL 描述和服务本身的地址。当用户想使用一个服务时，它首先查询 UDDI 找到与需求相匹配的服务并获得该服务的 WSDL 描述以及服务的访问点。用户使用 WSDL 描述来构造 SOAP 消息与服务进行交互。

一个 Web 服务表现为业务、应用或系统功能中不相关联的单元，多个 Web 服务可以被组合成具有更强功能的服务。这种模块化的方法给业务流程的设计带来更大的灵活性。通过组合服务，可以创建新的业务服务以完成不同的业务需求。Web 服务组合的方式一般可抽象地划分为静态组合与动态组合两种，其主要区别在于选择被组合服务的时机不同，前者是由设计人员在设计时确定，而后者则是在运行时选择。静态组合方法又可以分为编制（Orchestration）和编排（Choreography）两种方式[34]。编制是一种层次化的请求者/提供者模型，有一个控制中心，如 BPEL；而编排则是一种对等模型，业务流程是通过公共消息交换来完成各服务之间的协作，如 WS-CDL(Web Services Choreography Description Language，Web 服务编排描述语言)。

近年来，已经出现了很多业务流程建模语言并被应用于业务应用领域，比如 XLANG、WSFL、BPEL 和 StAC 等。这些语言提供一系列的基本操作，如交互操作和组合结构。现在，BPEL 已经成为 Web 服务组合事实上的标准。

2002 年 IBM、BEA 和微软一起开发和引入了 BPEL 作为描写协调 Web 服务的语言。这个描写的本身也由 Web 服务提供，可以当作 Web 服务来使用。定义业务协议和定义可执行业务流程所需的概念非常相似，BPEL 被设计成了定义业务协议所需的概念和定义可执行业务流程的所需概念的统一体。BPEL 所定义的模型和语法可被用于描述基于流程和它的伙伴间的交互的业务流程的行为。与每个伙伴的交互是通过 Web 服务接口进行的，在接口级别上关系的结构被封装在服务链接中。BPEL 流程定义了与这些伙伴交互的多个服务交互是怎样协调的以达到业务目的，还定义了这种协调所需的状态和逻辑。BPEL 还引入了一些系统的机制来处理业务异常和流程处理故障，以用于定义在发生异常时或伙伴请求撤销时流程中单个或合成活动是怎样被补偿的。应用 BPEL 基本概念的方式有两种：

(1) BPEL 可以通过使用抽象流程定义业务协议角色。例如，在供应链协议中，买卖双方是两个不同的角色，双方都有自己的抽象流程。它们的关系通常被模拟成服务链接。抽象流程对待数据处理的方式反映了描述业务协议公共部分所需的抽象程度。具体地说，抽象流程仅处理有关协议的数据，使用不确定的数值来隐藏行为的私有部分。

(2) BPEL 也可以被用来定义可执行业务流程。流程的逻辑和状态决定了在每个业务伙伴那里进行的 Web 服务交互的性质和顺序，从而决定了交互协议。虽然从私有实现的角度来看并不需要完整地定义 BPEL 流程，但是 BPEL 为仅依赖于 Web 服务资源和 XML 数据的业务流程有效地定义了可移植的执行格式。此外，这种流程的执行以及与它们的伙伴交互方式是一致的，与托管环境的实现所用的支持平台和编程模型无关。

即便在私有实现部分使用平台相关的功能的情况下(这在许多情况下是很有可能的),BPEL中抽象流程和可执行流程间的基本概念的模型的连续性可能将包含在业务协议中的公共部分作为流程或角色模板进行输出和输入，同时保持协议的目的和结构。从充分利用 Web 服务的角度来看，这是使用 BPEL 最有吸引力的前景，因为它支持大大提高自动化程度的工具和其它技术的开发从而降低了建立跨企业自动的业务流程的成本。

2.2 Web 服务事务研究现状

2.2.1 事务的概念

事务是由相关操作构成的一个完整的操作单元，这些操作要么全做要么全不做，是一个不可分割的工作单位[35]。

事务起源于数据库，是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。事务具有原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持续性(Durability)四个特性，简称为 ACID 特性。

(1) 原子性

事务是数据库的逻辑工作单元，其中包括的操作要么都做要么都不做。在任何操作出现一个错误的情况下，构成事务的所有操作的效果必须被撤消，数据应被回滚到以前的状态。

(2) 一致性

事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。一个事务应该保护所有定义在数据上的不变的属性（例如完整性约束）。在完成了一个成功的事务时，数据应处于一致的状态。比如，在关系数据库的情况下，一个一致的事务将保护定义在数据上的所有完整性约束。

(3) 隔离性

一个事务的执行不能被其它事务干扰。在同一个环境中可能有多个事务并发执行，而每个事务都应表现为独立执行。行发地执行一系列事务的效果应该同于串行地执行它们。

(4) 持久性

一个事务一旦被提交，其对数据库中数据的改变是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

保证事务 ACID 特性是事务处理的重要任务。在数据库中，多个事务并发运行时不同事务的操作交叉执行和事务运行过程中被强行停止都可能破坏事务的 ACID 特性，可以通过数据库系统中的的恢复机制和并发控制机制来保证 ACID 特性。

事务概念不仅应用于数据库，还应用于业务过程。业务过程是多方之间的活动，它作为一个整体必须被完成，并达到一致的结果。比如两个银行间进行转帐，要进行两步操作：①减少

甲银行 A 帐户中的钱；②在乙银行 B 帐户中存入相应的钱。这两步操作要么全部做，要么不做。业务事务常用于分布式环境中。分布式系统由大量用网络连接起来的计算机所组成，常常会遭受其中的任意子部件出现的故障，比如计算机本身、网络连接、操作系统、或者个别的应用程序，以及一些执行时间不能确定的行为活动等。分布式系统与集中型系统不同，不能通过集中管理来保证业务的事务特性。因此，分布式系统需要通过协调机制，扩展传统数据库的事务模型来保证业务运行的事务特性，比如 Web 服务事务处理协议 WS-TX 和 BTP.

2.2.2 Web 服务中的事务问题

Web 服务组合执行的一致性和可靠性要求可以通过事务机制来保障。Web 服务事务与传统的数据库事务不同，一个 Web 服务事务包括多个 Web 服务，它由多个子事务组成。这些子事务可以独立地终止或提交。换而言之，Web 服务事务是松散耦合的。它们很可能包括或跨越多个机构，而且是不可预测的和长周期的。所以，Web 服务事务吸取了其它一些松耦合环境下事务模型的做法，并且放松了传统事务 ACID 特性中的某些特性，以支持长事务（Long-Running Transactions，LRT）的运行[36]。扩展的事务允许将操作分组成层次结构[37, 38]，而放松的事务则可以放宽对 ACID 特性的一些要求。下面将对常见的扩展和放松的事务模型进行介绍[39]：

(1) 嵌套事务（Nested Transactions）

嵌套事务允许事务嵌套在事务内形成事务树[40]。子事务在父事务开始后开始，父事务在子事务结束后才能结束。如果一个父事务中止了，那它所有的子事务也中止。这些提交/中止和资源继承策略被在事务树上进行递归调用。使用嵌套事务有三个方面的优点：①它在全局层面上提供了全隔离性；②它为错误处理提供了很好的粒度；③没有冲突的事务可以并发执行。

(2) 开放嵌套事务（Open Nested Transactions）

开放嵌套事务通过使子事务的提交对顶层事务可见来放松隔离性要求[41]。在开放嵌套事务中，顶层事务的中止要求已经提交的子事务执行补偿操作。也就是说，子事务可以在全局事务完成和提交前提交和释放资源。如果全局事务中止了，可以通过要求已经提交的子事务执行补偿操作来保证一致性。

(3) Saga 事务模型（The Saga Transaction Model）

Saga 事务模型允许将一个长期运行的事务划分成子事务的序列[42]。每个子事务都有一个与之相关联的补偿子事务，当子事务运行失败时调用补偿事务，从语义上撤销其提交所产生的效果。也就是说，一个 saga 事务是由一组具有 ACID 特性的有一定执行顺序的子事务 $T_1,\cdots T_n$ 组成，这些子事务对应一组补偿子事务 $CT_1,\cdots CT_{n-1}$。如果一个 saga 子事务失败了并且不能恢复，其前序子事务所产生的效果将通过执行补偿子事务 $CT_{k-1},\cdots CT_1$ 撤销。

(4) 拆分一合并事务模型（The Split-Join Transaction Model）

拆分一合并事务模型将事务拆分成两个不相关或相关的事务并且在它们结束后连接在一起

[43]。它被设计成 open-ended 活动，这些活动是非确定的，需要持续很长时间，发展过程不可预知，且活动之间能够交互。

(5) 契约（ConTracts）

契约是一种将多个事务组合成一个多业务事务活动的机制[44]。它由一组预定义的活动（步）以及这组活动的执行计划（脚本）组成。契约的执行必须是可向前恢复的。契约一旦行动失败，它将被立刻恢复并且它的执行可以继续。

(6) 长运行活动（Long-Running Activity）

长运行活动由一组执行单元组成，这些执行单元由其它的活动或事务递归组成[45]。活动的控制流和数据流可以由活动的脚本静态地指定或通过 ECA（Event-Condition-Action）动态说明。

Web 服务事务处理与其它分布式事务处理系统相似，但它具有如下两个特点：

(1) Web 服务的事务处理通常要跨越组织边界，这意味着事务的参与者是自治的并且分布于网络[46]。由于事务的参与者分属于不同的组织，因此其事务处理方法也往往不相容。

(2) Web 服务事务可能要运行很长时间，而各组织不能允许其资源在开放的环境中被不可预知地消费。这意味着必须想办法确保资源在长运行期内不被阻塞[46]。然而，这与严格的 ACID 属性相冲突，因为 ACID 要求资源必须被锁定直到事务中止，以保证隔离性和一致性。此外，提交协议很容易受到网络恶意攻击从而导致资源被长时间地加锁。

为了解决以上问题，早期人们定义了基于因特网的事务协议 TIP（Transaction Internet Protocol）用于简化分布式应用程序（如 Web 服务）的事务处理。其后又定义了业务事务协议 BTP(Business Transaction Protocol)用于业务事务处理。BTP 协议要求事务处理支持放松的 ACID 属性和扩展的事务。为了解决资源阻塞的问题，THP 协议(Tentative Hold Protocol)被提出来了。THP 协议允许多个客户端同时尝试预订同一资源，这大大减少了“取消”操作。作者所在团队已经使用 Pi-演算对 THP 协议进行了分析和验证[47, 48]。为了将业务处理和事务处理分离，WS-TX 应运而生。WS-TX 与 Web 服务方案联合制定，与 BPEL 联合发布，得到了业界的广泛关注。

2.3 Web 服务事务协调协议 WS-TX

2002 年，IBM 与微软创立了 WS-Transaction 和 WS-Coordination，作为用于面向 Web 服务的业务流程执行语言 BPEL 的组成部分。其后，WS-Transaction 被分拆为 WS-AtomicTransaction 和 WS-BusinessActivity，并提交国际标准联盟 OASIS. 2007 年 5 月，OASIS 宣布其成员已经正式批准 Web 服务事务处理 WS-TX 1.11 版本成为 OASIS 标准。 WS-TX 描述的是为那些并列的分布式应用程序软件行为的协议提供扩展的协议框架。这种协同的协议可以用来支持各种各样的需要在分布式的结果中拥有统一协议的应用程序软件。

WS-TX 协议是由三个具体部分组成，它们分别是：WS-Coordination（Web 服务协调，简写为 WS-C），WS-AtomicTransaction （Web 服务原子事务，简写为 WS-AT）以及

WS-BusinessActivity（Web 服务业务活动，简写为 WS-BA）。WS-C 能够让应用程序服务创建为其它服务传播所必须的内容。WS-AT 定义了拥有全部或者空属性的短生命周期活动的协定协议。WS-BA 则定义了长生命周期活动所需要的基于调整协定的协议。三者在一起工作，它们能够让已经存在的事务处理进程、工作流以及其它的系统隐藏它们自身的协议，并能够在各种不同的环境下运作。

2.3.1 Web 服务协调 WS-Coordination

WS-C 提供了一个开放的协调框架，每个具体的应用都可以在此基础上设计自己的协调协议以控制整个协调过程，达到各自的应用目的。比如简单的短周期操作的事务协议（如 WS-AT）和复杂的长周期业务活动的事务协议（如 WS-BA）都可以在这个框架中进行定义，以使参与者能够得到一致的结果。

在具体介绍 WS-C 之前，先介绍几个概念：

(1) 活动（Activity）：由一系列 Web 服务为实现某个功能组合而成的单元。

(2) 参与者（Participant）：活动中的某个 Web 服务。

(3) 协调者（Coordinator）：和活动参与者发生协调行为的实体。

(4) 协调类型（Coordination Type）：根据协调过程的目的划分的类型。如 WS-AT 协议中定义的协调类型为 AtomicTransaction。

(5) 协调协议（Coordination Protocol）：协调过程中协调者与参与者消息交互过程。如 WS-AT 规范中定义的 Completion 和 2PC(2 Phrase Commit)两套协议。

(6) 协调上下文（Coordination Context）：协调上下文可以被看作是一个标识符，行为执行在这个标识符之下。协调上下文非常类似于事务上下文，调用这个上下文去确定或回滚已经完成的工作。它被用于在多个应用程序间传递协调信息，包括一个协调标识符、关于协调类型的详细资料以及包括端口信息以便协调服务能够被访问的协调协议。

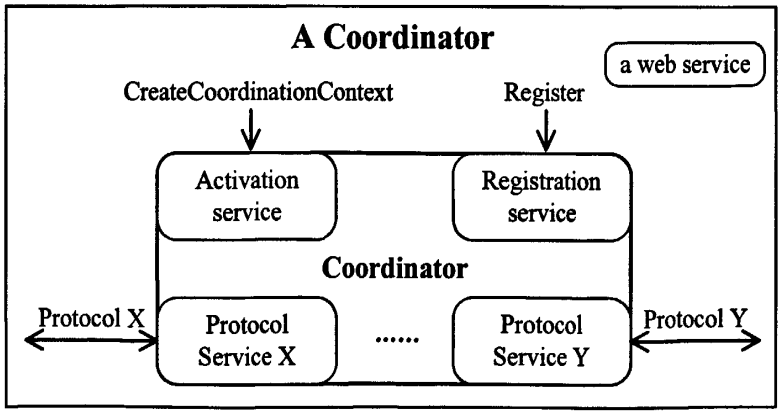

图 2.2 协调服务组成结构

协调者本身就是一个服务——协调服务（Coordination Service），它包含组成协调框架的激活服务，注册服务和特定的协调协议服务。图 2.2 给出了协调服务的组成结构，描述如下：

(1) 激活服务（Activation service）：激活服务提供了根据申请的协调类型创建出相应的协调协议服务和协调上下文的功能。

(2) 注册服务(Registration service)：注册服务为应用程序提供了注册具体协调协议的功能。

(3) 协调协议服务（Coordination Protocol Service）：根据协调协议创建的服务，用于协调消息的传递，以 Web 服务的形式存在。如 WS-AT 协议中为 Completion 和 2PC 协议分别在协调者和参与者之间创建服务，服务中定义了协调消息交换的具体格式。

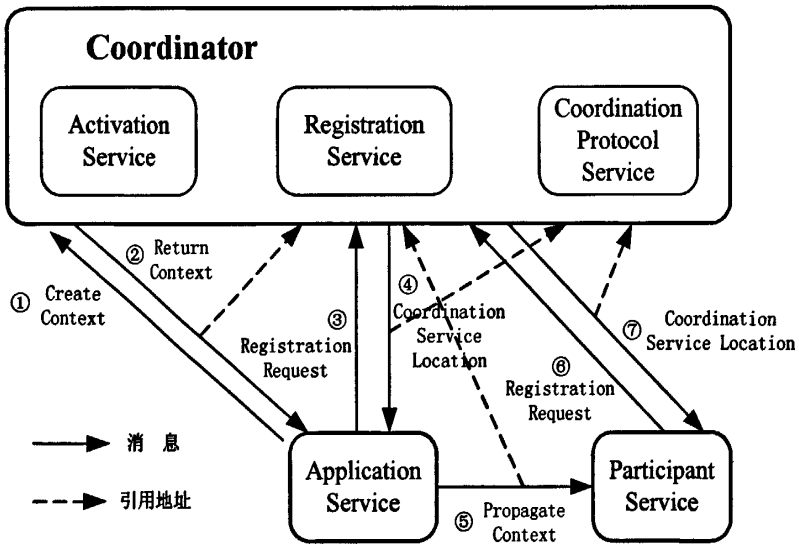

图 2.3 协调工作过程

图 2.3 描述了协调工作的过程。各步骤具体介绍如下：

① 活动发起者（即 Application Service）请求创建 Coordinator（包括 Coordination Context, Registration Service, Coordination Protocol Service）。

② 根据请求中给定的协调类型，返回相应的 Coordination Context，其中包含了 Registration Service 的引用地址。

③ 如果协调类型需要，活动发起者使用注册服务将自己的引用地址注册到相应的 Coordination Protocol Service 上。

④ 注册服务将活动发起者所注册的 Coordination Protocol Service 的引用地址返回给活动发起者。

⑤ 活动发起者依次调用活动中包含的 Web 服务（即 Participant Service），并将在步骤②中获得的 Coordination Context 添加到服务请求的 SOAP Head 中。

⑥ 和步骤③类似，Participant Service 使用 Coordination Context 中包含的 Registration

Service 的引用地址将自己注册到相应的 Coordination Protocol Service 上。通常来说步骤⑥是必须的而步骤③视具体应用而定。

⑦ 注册服务将活动发起者所注册的 Coordination Protocol Service 的引用地址返回给 Participant Service.

通过以上 7 个步骤(其中③, ④可选)，Participant Service 和 Coordinator 之间建立起了双向联系，之后它们就可以根据具体的协调协议开始针对某个应用目的的协调过程。

2.3.2 Web 服务原子事务 WS-AtomicTransaction

WS-AT 在 WS-C 的基础上定义了一个原子事务（Atomic Transaction）协调类型用于协调具有“All-or-None”属性的活动。原子事务通常需要参与者之间在短生命周期内具有高度的信任关系，WS-AT 定义了三种原子事务的协调协议：completion、volatile two-phase commit 和 durable two-phase commit。

原子事务具有“All-or-None”的属性。参与者提交前的活动都是尝试性的，即既不是持久性的，对事务外界也是不可见的。当一个应用程序完成了事务操作，它要求协调者决定事务的结果。协调者通过要求参与者投票来决定是否有处理故障。如果参与者都投票表明自己能够成功地运行，协调者提交所有的活动。如果有参与者投票表明需要中止或参与者没有回应，协调者中止所有的活动。提交（Commit）操作指示参与者将它们的尝试动作持久化并对事务外界可见。中止（Abort）操作指示参与者放弃尝试动作使其看起来没有发生过。原子事务被证明在很多应用中很有用。它提供一致性的错误和恢复语义，所以应用程序不需要达成一致性结果的机制和指定如何从众多的不一致性状态中恢复到一致性状态。

要开始一个原子事务处理，客户端应用程序必须首先定位一个协调服务（WS-C 的实例），这个协调服务支持正确的模型。一旦协调服务定位完成，客户端就会通知它开始一个新的事务，并且取回一个事务上下文环境，这个环境包含了对该事务注册服务的引用。在取得上下文环境以后，客户端应用程序就继续与 Web 服务交互，以完成它的业务级工作。对于业务服务的每次调用，客户端都会传送上下文，这样事务就可以隐式地确定每个调用的范围。

一旦所有必要的应用程序级工作都完成了，客户端就会终止这个事务。事务的中止使用传统的持久性的两阶段提交（Durable2PC）协议；参与者被期望对持久性数据进行操作，比如数据库中的表格或者文件系统中保存的资源。

下面介绍 WS-AT 定义的三个协调协议：

1） Completion Protocol

通过使用 Completion 协议，应用程序告诉协调者是要提交还是中止原子事务。在事务结束后，返回给应用程序一个状态。图 2.4 简单描述了协调者与应用程序之间接受不同消息之后的状态转换图，事务各状态间以消息驱动。

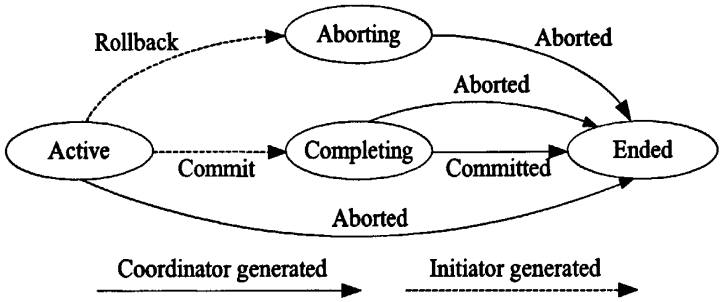

图 2.4 Completion 协议状态转换图

协调者可以发送 Committed 和 Aborted 消息，而应用程序可以发送 Commit 和 Rollback 消息。表 2.1 为消息对于协调者和应用程序蕴含的含义。

表 2.1 Completion 协议中消息的含义

消息	消息发送方	含义
Committed	协调者	收到消息后，应用程序知道协调者决定提交事务。
Aborted	协调者	收到消息后，应用程序知道协调者决定中止事务。
Commit	应用程序	收到消息后，协调者知道应用程序已经完成了业务流程，在其处于 Active 状态时提交事务。
Rollback	应用程序	收到消息后，协调者知道应用程序已经终止了业务流程，在其处于 Active 状态时中止事务。

2）Two-Phase Commit Protocol

Two-Phase Commit(两阶段提交，2PC)协议是一个协调协议，它被用来定义多个参与者如何为原子事务的结果达到一致性状态。两阶段提交协议具有两种类型：Volatile 2PC 和 Durable 2PC。

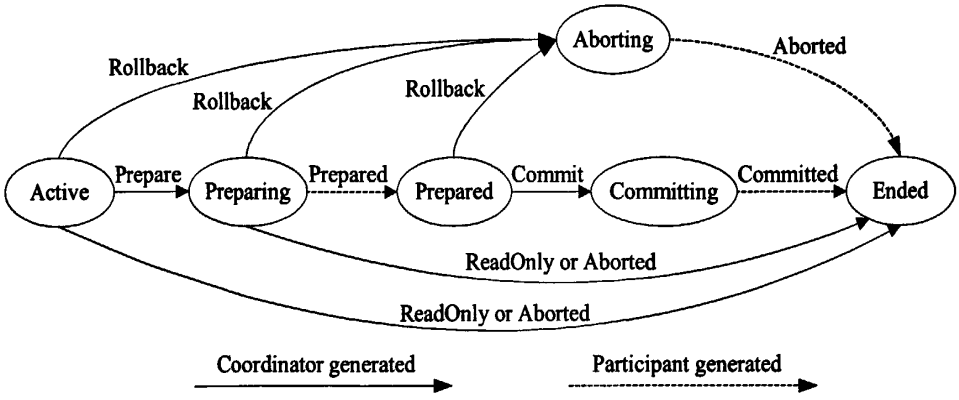

图 2.5 两阶段提交协议状态转换图

图 2.5 描述了两阶段提交协议的状态转换图，说明了协调程序和参与者间的消息交换过程。协调者可以发送 Prepare、Rollback 和 Commit 消息，而参与者可以发送 Prepared、ReadOnly、

Aborted 和 Committed 消息。表 2.2 为消息对于协调者和参与者蕴含的含义。

表 2.2 两阶段提交协议中消息的含义

消息	消息发送方	含义
Prepare	协调者	收到消息后，参与者知道进入第一阶段并且对事务的结果进行表决。处于 Active 状态的参与者可以通过发送 Aborted、Prepared 或 ReadOnly 消息进行表决。如果参与者对当前事务不了解，那它必须发送 Aborted 消息。如果参与者知道它已经进行了表决，则必须重新发送与此前相同的表决。
Rollback	协调者	收到消息后，参与者知道中止并退出事务。处于非提交状态的参与者必须发送 Aborted 消息并且删除当前事务的所有信息退出事务。如果参与者不了解当前事务，则必须向协调者发送 Aborted 消息。
Commit	协调者	收到消息后，参与者知道要提交当前事务。Commit 消息只能在第一阶段过后并且参与者已经投票表决提交后才能发送。如果参与者不了解当前事务，则参与者发送 Committed 消息给协调者。
Prepared	参与者	收到消息后，协调者知道参与者已经准备好并将投票表决提交事务。
ReadOnly	参与者	收到消息后，协调者知道参与者投票表决提交事务并且已经退出事务，参与者并不希望进入第二阶段。
Aborted	参与者	收到消息后，协调者知道参与者已经中止并退出了事务。
Committed	参与者	收到消息后，协调者知道参与者已经提交并退出了事务。

除了传统的 Durable2PC(持久性的两阶段提交)协议，原子事务处理协议还支持 Volatile2PC（短时间两阶段提交）协议，它的准备阶段运行于整个 Durable2PC 之前，而其提交或回滚阶段运行于 Durable2PC 完成之后。这个协议背后的原因，是因为访问持久性存储器常常是性能瓶颈。因此，对一个对象的状态（例如整个数据库表格）的缓存和事务周期内对缓存下来的状态进行操作，相对于持续地从数据库往返交互的方式，可以显著地提高性能。然而，在事务处理提交之前很显然需要强制将状态更改为最初的持久化存储。

Volatile2PC 把两阶段提交协议变成了一个四阶段协议：

(1) 在事务处理开始 Durable2PC 之前，所有用 Volatile2PC 注册的参与者都被通知到，并可以清空缓存数据。这时的任何失败都会导致事务处理的回滚。

(2) 协调程序执行完整的 Durable2PC 协议。

(3) 一旦事务处理中断，Volatile2PC 协议的第二段就被执行。但是，因为这里有些错误被忽视了，所以认为事务中断没有什么影响。

原子事务处理模型在 Web 服务里成功地模拟出传统的 ACID 事务处理协议。这被意味着

可以匹配已存在的事务处理标准，这些标准对原子性、隔离性和持久性等有很好的定义行为。

2.3.3 Web 服务业务活动 WS-BusinessActivity

WS-BA 在 WS-C 的基础上定义了两种业务活动(Business Activity)协调类型用于协调活动，这些活动使用业务逻辑来处理业务流程活动执行过程中发生的异常。参与者的动作会被持久化，因此活动在发生错误时补偿操作会被触发。WS-BA 定义了两种协调类型和两种协议类型，任何一种协调类型可以与任何一种协议类型组合使用。

两种协调类型分别为 AtomicOutcome（原子输出）和 MixedOutcome（混合输出）：

(1) 具有 AtomicOutcome 协调类型的协调器必须指示所有的参与者要么终止要么补偿。

(2) 具有 MixedOutcome 协调类型的协调器必须指示所有的参与者到达一致性状态，但是可以指示单个的参与者终止或补偿。

两种协调协议分别为：BusinessAgreementWithParticipantCompletion（参与者主动完成）AgreementWithCoordinatorCompletion（协调者主动完成）。下面进行简单地介绍：

1）BusinessAgreementWithParticipantCompletion 协议

在 BusinessAgreementWithParticipantCompletion 协议下，子活动开始以一个活动状态被创建，如果它结束了需要创建它来处理的工作，并且在业务活动范围（比如在活动操作常数时）内没有更多的参与者，那么子活动就可以单方面脱离父范围。但是，如果子活动结束了，并且想在业务活动里继续，那么它必须能补偿它已经执行的工作。

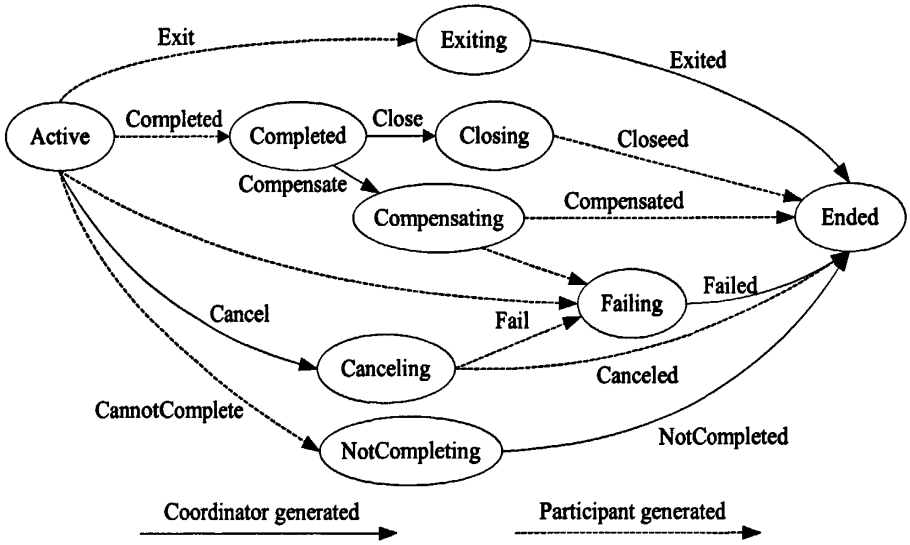

图 2.6 BusinessAgreementWithParticipantCompletion 状态转换图

图 2.6 描述了 BusinessAgreementWithParticipantCompletion 协议的状态转换图。协调者可以发送 Close、Cancel、Compensate、Failed、Exited 和 NotCompleted 消息，而参与者可以发送

Completed、Fail、Compensated、Closed、Canceled、Exit 和 CannotComplete 消息。同时，协调者和参与者都可以发送 GetStatus 和 Status 消息。表 2.3 为消息对于协调者和参与者蕴含的含义。

表 2.3 BusinessAgreementWithParticipantCompletion 协议中消息的含义

消息	消息发送方	含义
Close	协调者	收到消息后，参与者知道协议实例将成功结束，必须发送 Closed 消息来结束协议实例。
Cancel	协调者	收到消息后，参与者知道完成的工作已经被取消，必须发送 Canceled 或 Fail 消息。如果工作被成功地取消，参与者应该发送 Canceled 消息，这将同时结束协议实例；如果工作没有被成功取消，参与者应该发送 Fail 消息。
Compensate	协调者	收到消息后，参与者知道已完成的工作应该被补偿，必须发送 Compensated 或者 Fail 消息。如果工作被成功地补偿，参与者发送 Compensated 消息，这将同时结束协议实例；如果工作未被成功地补偿，参与者发送 Fail 消息。
Failed	协调者	协调者发出消息后忘记参与者。参与者收到消息后知道协调者意识到已经失败并且不需要再做什么，参与者退出当前活动。
Exited	协调者	协调者发出消息后忘记参与者。参与者收到消息后知道协调者意识到参与者将不再参与当前活动，参与者退出当前活动。
NotCompleted	协调者	协调者发出消息后忘记参与者。参与者收到消息后知道协调者意识到参与者不能完成与协议实例相关的所有数据处理，参与者将不再参加当前活动，退出当前活动。
Completed	参与者	收到消息后，协调者知道参与者已经完成了与协议实例相关的数据处理，协调者必须发送Close或者Compensate消息表示协调实例的最终结果。发送完消息后，参与者不再参与当前活动。
Fail	参与者	收到消息后，协调者知道参与者在Active、Canceling和Compensating状态下失败，参与者的操作状态未确定，协调者必须发送Failed消息，并附带失败的原因。
Compensated	参与者	参与者发送消息后退出当前活动。协调者收到消息后知道参与者已经成功补偿了所有相关的数据处理，忘记参与者的状态。
Closed	参与者	参与者发送消息后退出当前活动。协调者收到消息后知道参与者已经成功地结束了协议实例，协调者忘记参与者的状态。

表 2.3（续）

消息	消息发送方	含义
Canceled	参与者	参与者发送消息后退出当前活动。协调者收到消息后知道参与者已经成功取消了所有相关的数据处理，忘记参与者的状态。
Exit	参与者	收到消息后，协调者知道参与者将不再参与业务活动，参与者未决定的工作都将被放弃，参与者已经完成的与协议实例相关的工作已经被成功取消。对于下一个协议消息，协调者必须发出Exited消息。只有处于Active或者Completing状态时参与者才能发送Exit消息。
CannotComplete	参与者	收到消息后，协调者知道参与者将不再参加业务活动，参与者未决定的工作都将被放弃，参与者已经完成的与协议实例相关的工作已经被成功取消。对于下一个协议消息，协调者必须发送NotCompleted消息。在发送CannotComplete消息后，参与者不允许参与当前活动的任何工作。只有处于Active状态时参与者才能发送CannotComplete消息。
GetStatus	协调者、参与者	发送方发送消息表明是要得到对方当前的状态，而对方则返回当前状态。GetStatus消息不改变对方的状态。
Status	协调者、参与者	Status消息是作为GetStatus请求消息的响应消息。

2）BusinessAgreementWithCoordinatorCompletion 协议

除了子活动不能自主决定在业务活动中终止它的参与者，BusinessAgreementWithCoordinatorComplete 协议和 BusinessAgreementWithParticipantComplete 协议是一样的，尽管它可以被补偿。在子活动收到所有给它的请求时，它依赖父范围通知它，然后执行父范围通过给子活动发送完全消息通知它处理。然后子活动开始像它在 BusinessAgreementWithParticipantComplete 协议里做的一样，开始工作。本文给出的 WS-BA 应用场景使用的是 BusinessAgreementWithParticipantCompletion 协议，因此这里不对 BusinessAgreementWithParticipantComplete 协议作详细介绍，其详细介绍可参见文献[10].

2.4 本章小结

本章首先介绍了 Web 服务及其相关技术。其次对 Web 服务中的事务问题进行了分析，比较了 Web 服务事务与传统事务处理方法的异同。然后对现有 Web 服务事务处理方法和技术进行了分析。最后重点介绍了 Web 服务事务协调协议 WS-TX.

第三章 基于 Pi-演算的 WS-TX 协议应用建模

WS-TX 协议为分布式业务事务处理提供了解决方法。然而，如何保证 WS-TX 协议应用的正确性是将 WS-TX 协议应用于分布式业务中必须解决的问题。用形式化方法对 Web 服务事务进行研究可以帮助人们更好地理解 Web 服务事务的行为特性，其语义模型也可以提供对分布式业务事务的推理基础，基于模型检查的分析技术还可以允许自动地检验分布式业务系统是否满足事务特性，从而帮助消除设计缺陷，提高系统的可靠性。

由于 Pi-演算的结构特性、移动特性，以及其理论的完整性，Pi-演算常被用来对 Web 服务及其事务协议进行建模以验证其正确性。本章将使用 Pi-演算对 WS-TX 应用场景进行建模。

3.1 Pi-演算建模分析

Pi-演算由 Robin Milner 提出，它以进程间的移动通信为研究重点，是对 CCS(Calculus of Communication System)的发展，其基本计算实体为名字（name）和进程（process），进程之间通过传递名字来完成通信[49]。与 CCS 不同，Pi-演算除了可以传递与 CCS 中相对应的变量和值以外，还可以传递通道名，这是因为 Pi-演算中并不将它们进行区分，而是统称为名字。Pi-演算传递通道的能力使得 Pi-演算可以被用来描述结构变化的并发系统。

用 x、y、z 等小写字母表示名字，P、Q 等表示进程表达式，A、B、C 等大写字母表示进程标识符，则 Pi-演算的语法结构描述如下：

$$P=\mathbf{0}\,|\,\alpha.P\,|\,[x=y]P\,|\,P+Q\,|\,P\,|\,Q\,|\,(\nu\, x)P\,|\,!P\,|\,A(y_1,y_2,\cdots,y_n)$$

$$\alpha=\tau\,|\,a(x)\,|\,\overline{a}<x>$$

其中：

(1) $\mathbf{0}$ 表示空进程，什么也不做；

(2) $\alpha.P$ 表示经过动作 α 后表现为进程 P 的行为。Pi-演算里定义了 3 种前缀动作，前缀 τ 表示一个进程外部不可见的内部动作(或称为哑元动作)；输入前缀 $a(x)$ 表示从通道 a 接受一个名字；输出前缀 $\overline{a}<x>$ 表示名字 x 沿着通道 a 输出，该名字将替换 P 中的名字 x；

(3) $[x=y]P$ 表示当名字 x 和 y 相等时执行进程 P；

(4) $P+Q$ 表示选择执行进程 P 或 Q；

(5) $P|Q$ 表示由 P 和 Q 并行运行的进程；

(6) $(\nu\, x)P$ 表示名字 x 被限制在进程 P 内，只在进程 P 内部可见；

(7) $!P$ 表示进程 P 的无数次复制；

(8) $A(y_1,y_2,\cdots,y_n)$为进程标识符。每个进程标识符必须有其定义 $A(x_1,x_2,\cdots,x_n) \overset{def}{=} P(i \neq j \Rightarrow x_i \neq x_j)$，其含义是当用 y_i 替换 x_i 时，$A(y_1,y_2,\cdots,y_n)$ 的行为表现为进程 P。

Pi-演算的迁移语义通过标签迁移系统（Labeled Transition System，LTS）给出，如图 3.1 所示。Pi-演算的迁移语义通过定义进程能够执行的动作，表明进程的行为能力，能够描述系统内部活动以及系统与环境的交互活动。在 Pi-演算中，用基于互模拟的等价关系来刻画进程之间的行为等价关系。互模拟等价关系有强互模拟等价、弱互模拟等价和观察同余等。

$$\text{STRUCT}\ \frac{P' \equiv P, P \xrightarrow{\alpha} Q, Q \equiv Q'}{P' \xrightarrow{\alpha} Q'} \qquad \text{PREFIX}\ \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{SUM}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{MATCH}\ \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'}$$

$$\text{PAR}\ \frac{P \xrightarrow{\alpha} P', bn(\alpha) \cap fn(Q) = \varnothing}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{COM}\ \frac{P \xrightarrow{a(x)} P', Q \xrightarrow{\bar{a}\langle u \rangle} Q'}{P|Q \xrightarrow{\tau} P'\{u/x\}|Q'}$$

$$\text{RES}\ \frac{P \xrightarrow{\alpha} Q, x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \qquad \text{OPEN}\ \frac{P \xrightarrow{\bar{a}\langle x \rangle} P', a \neq x}{(\nu x)P \xrightarrow{\bar{a}\langle \nu x \rangle} P'}$$

图 3.1 Pi-演算迁移语义

3.2 面向 WS-TX 协议应用场景的 Pi-演算建模方法

3.2.1 WS-TX 协议应用场景元素与 Pi-演算元素的映射关系

用 Pi-演算描述 WS-TX 应用场景的本质问题是如何把 WS-TX 应用场景的元素和结构用 Pi-演算来表示，共有三个问题必须解决：

(1) WS-TX 应用场景参与者的元素如何与 Pi-演算的基本元素对应；

(2) WS-TX 应用场景的各参与者如何用 Pi-演算表示；

(3) 整个 WS-TX 应用场景作为一个整体如何用 Pi-演算表示。

Pi-演算的基本元素就是进程和名字，进程间的交互是通过通道进行。而WS-TX应用场景各参与者是通过彼此并发地交互以完成相互的请求，而彼此间的交互是通过通信和交换信息来完成的[50]。WS-TX应用场景中的各参与者是自治的和跨组织域的，因此将各参与者作为Pi-演算中一个独立的进程。Pi-演算只描述进程间的通信，并不描述进程的内部逻辑。参与者的内部实现对于其它参与者而言是透明的，即参与者状态的变化都来源于消息的触发，所以不需要关注参与者的内部逻辑，这与Pi-演算通过消息的传递来表示变化的思想相一致，参与者之间的交互可以被抽象为彼此之间沿着通道发生的输入输出动作，因此可以使用Pi-演算进程间通信来描述参与者间的交互关系，从而表示活动中的每个参与者。参与者是自治的，参与者间的消息约束了

参与者的行为，多个参与者可以协同工作，完成事务协议的协调过程。因此，整个WS-TX应用场景的Pi-演算进程用参与者进程的并发来表示。

参与者间的通信通过通道进行，消息和通道映射为 Pi-演算的名字。两个参与者间通道的个数由以下方法决定：当消息在两个服务间以非并行的顺序出现时，为这两个服务引入一条通道；当消息在服务间以并行方式传递时，引入多条通道，通道的个数为最大的消息并发数。这样做的目的是为了进行通道复用，减少 Pi-演算中名字的数量，从而减少模型检测时的状态空间，提高检测效率。

表 3.1 WS-TX 应用场景元素到 Pi-演算元素的映射关系

WS-TX 应用场景	Pi-演算
参与者	进程
通信通道	名字（通道）
消息	名字（消息）

通过对“用 Pi-演算来描述 WS-TX 应用场景三个问题”解决方法的探讨，得出 WS-TX 应用场景元素和结构与 Pi-演算元素的映射关系，如表 3.1 所示。表中的 WS-TX 应用场景的通信通道为虚拟概念，因为各参与者通过 SOAP 协议进行通信时并不显式地建立端到端的连接。表 3.1 蕴含了使用 Pi-演算对 WS-TX 应用场景建模的三个原则：

(1) 在参与者间引入通道，通道的多少由消息的并发数决定。将通道和消息映射为 Pi-演算中的名字。

(2) 参与者作为 Pi-演算中一个独立的进程，使用 Pi-演算进程间通信来描述参与者之间的交互关系，作为参与者的 Pi-演算进程。

(3) 整个 WS-TX 应用场景作为 Pi-演算中一个独立的进程，其 Pi-演算进程用参与者进程的并行来表示。

在 WS-TX 应用场景的 Pi-演算模型中传递的消息是一个抽象的概念，它并不一定表示某个具体的消息，而有可能是消息的集合。另外，在 Pi-演算中传递的消息并不是包含实际应用场景中所有的消息，它忽略了对于模型而言无关紧要的消息，比如不改变系统状态的消息。

3.2.2 WS-TX 协议应用场景的 Pi-演算建模规则

通过 Pi-演算的迁移语义可以看出，Pi-演算的状态迁移通过动作来触发。这些动作包括输入动作、输出动作、哑元动作、选择和并行五种。WS-TX 应用场景参与者活动的外在表现正好可以用 Pi-演算的这五种动作来表示，动作的组合描述了参与者的动态运行过程。下面以参与者状态改变的触发条件为基础，总结 WS-TX 协议应用的 Pi-演算建模规则。

为了详细说明如何使用 Pi-演算对 WS-TX 应用场景进行建模，假定存在参与者 A 和 B，A

和 B 之间存在通信通道 c。根据表 3.1 的映射关系，将 A 和 B 的当前状态分别映射为进程 P 和 Q，将通信通道 c 映射为名字 x。

规则 1 参与者 A 通过通道 c 输出消息给 B，表现为 A'，将 A'映射为 Pi-演算的进程 P'，输出的消息映射为 M，则对 A 进行 Pi-演算建模为：

$$P = \overline{x} < M > .P'$$

这里的 M 为标识，它对应一组实际消息。比如 WS-TX 应用场景各参与者间通过 SOAP 协议传送失败消息，除了发送 Failed 消息外还发送失败的原因，这些信息统一映射为 M。后续规则中 Pi-演算通道中传递的消息都是标识，对应一组实际消息。

规则 2 参与者 A 收到 B 通过通道 c 发来的消息 msg，如果 msg 为消息 m_1，则表现为 A'；如果 msg 为消息 m_2，则表现为 A"；……。将 A'映射为 Pi-演算的进程 P'，A"映射为 $P"$，m_1 映射为 M_1，m_2 映射为 M_2，则对 A 进行 Pi-演算建模为：

$$P = x(msg).\left([msg = M_1]P' + [msg = M_2]P" + \cdots\right)$$

msg 并不是通道 x 传递的消息，它是引入的内部名字。在通道 x 中传递的消息为 M_1 或 M_2. 在标准 Pi-演算中，通道 x 可以传递通道，但在这里并不使用这一能力。因为 WS-TX 应用场景的各参与者在设计时就已经确定，相应的各参与者间的通信通道也确定，所以不需要传递通道。

规则 3 参与者 A 经过了一个内部动作表现为 A'，将 A'映射为 Pi-演算的进程 P'，内部动作映射为哑元动作 τ，则对 A 进行 Pi-演算建模为：

$$P = \tau.P'$$

由于内部动作是不可见的，在对 WS-TX 应用场景进行 Pi-演算建模时，可以忽略内动作，将 P 和 P' 等同看待。

规则 4 参与者 A 经过一系列内部动作后表现为 A'或 A"，将 A'映射为 Pi-演算的进程 P'，A"映射为 $P"$，则对 A 进行 Pi-演算建模为：

$$P = P' + P"$$

参与者 A 表现为 A'还是 A"是由内部执行结果决定的，并不是任意的。但从外在表现看，A 要么表现为 A'要么表现为 A"，因此选用 Pi-演算的选择动作对其进行建模。

规则 5 参与者 A 表现为 A'和 A"的并发执行，将 A'映射为 Pi-演算的进程 P'，A"映射为 $P"$，则对 A 进行 Pi-演算建模为：

$$P = P' \mid P"$$

并行动作一般用于对参与者的组合的建模。比如以上 Pi-演算进程中的 P 对应 WS-TX 应用场景，P' 和 $P"$ 等对应参与者。

Pi-演算通过消息的传递来模拟参与者的行为，因此消息的传递是 Pi-演算对 WS-TX 应用场景进行建模的基础。在 Pi-演算模型中，传递的消息都具有确切的含义，一个进程的输出必然为

另一个进程的输入。在 WS-TX 应用场景的整个模型中，所有参与者的输出的消息的集合与输入的消息的集合相等。

3.3 WS-TX 协议应用场景建模

3.2 节给出了 WS-TX 应用场景的 Pi-演算建模方法，本节将通过一个具体的实例来说明如何使用 Pi-演算根据建模规则对 WS-TX 应用场景进行建模。

3.3.1 WS-TX 协议应用场景

在本节中，将看到这样一个场景：为会议安排旅行和住宿。这里假定会议出席者需要预订到会议举办地的航班和当地酒店的一个房间。在这一场景中，用户还要给旅行代理商付费。

假定用户在交通银行有一帐户 A，旅行代理商在农业银行有另一帐户 B。付费的过程就是把帐户 A 里面的部分金额转到帐户 B 中。由于在转帐过程中帐户 A 与帐户 B 必须被锁定，且“A 帐户减少金额”与“B 帐户增加金额”这两个操作要么全部完成要么都不做，所以将使用 WS-AT 协议来完成银行转帐的协调过程。WS-AT 支持 Durable2PC 和 Volatile2PC 两种提交方式，只是先后顺序不同，整个流程是一样的，所以这里不区分两者的差别，协调过程将使用 Durable2PC 方式进行提交。这里假定交通银行和农业银行在经过认证后都提供包含增减帐户金额的 Web 服务。图 3.2 为银行转帐参与者交互图。

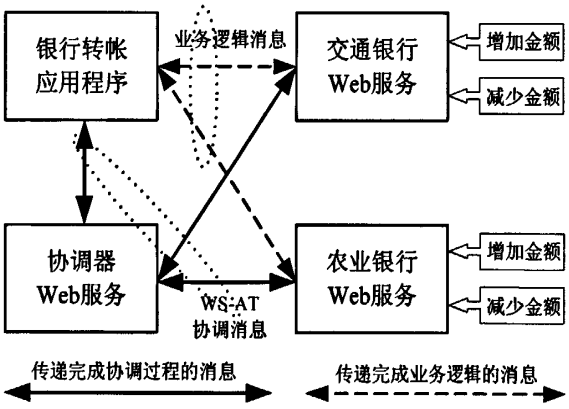

图 3.2 银行转帐参与者交互图

在预订航班和宾馆时，需要同时为整个旅行确定必要预订组别，以确保合适的选择可被预约。除了要考虑会议出席者的需要，服务提供商也需要保留一定自主权，并且保持对它们自己资源（本例中是航班和房间的预约）的控制。在这个业务领域中，每一个预约的基本要素都互相关联，很显然，航班没有订下来的话，预订宾馆房间就显得毫无意义。因而，在单独的原子事务处理中进行整个旅行的安排是不合适的，因为在那种情况下，要么所有的工作都发生，要么就什么都不发生，且资源必须被锁定。因此在这个旅行代理业务场景中，WS-BA 比 WS-AT 处理更为合适。WS-BA 有两种完成协议，参与者主动完成和协调者主动完成，两者有细微差别，

差别在于谁先发送完成消息，旅行安排场景将使用协调者主动完成协议。WS-BA 提供了两种协调类型：原子输出和混合输出。为了简化模型，旅行安排场景将采用原子输出协调类型。图 3.3 为旅行安排参与者交互图。

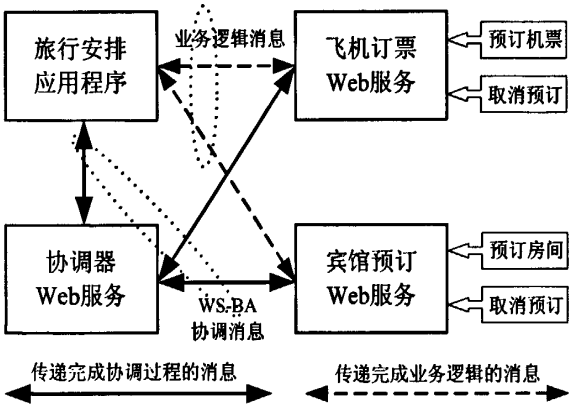

图 3.3 旅行安排参与者交互图

3.3.2 WS-TX 协议应用场景的 Pi-演算模型

本节将对银行转帐和旅行安排中的协调过程进行建模，而忽略业务逻辑部分。同时，只考虑引起参与者状态改变的消息，而忽略不引起状态改变的消息，比如 WS-BA 中的 GetStatus 和 Status 消息。由于参与者注册协调器的过程是固定的，因此本文将不对 WS-C 的工作过程进行建模。为了简化模型，这里不进行超时处理，假定消息发送后一定能在规定时间内被对方接收。

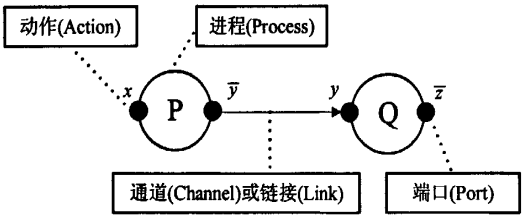

图 3.4 Pi-演算流图

为了表述的方便，借用 Pi-演算流图[51]的概念来描述应用场景。图 3.4 给出了应用场景参与者与 Pi-演算基本元素的对应关系，它描述了一个最简单的交互动作，P 进程沿着通道发出消息，Q 进程从通道接收消息。

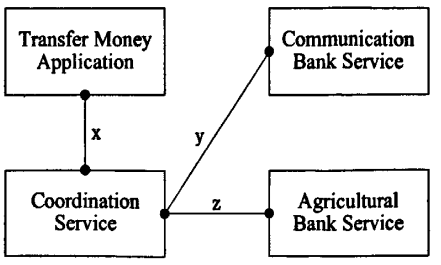

图 3.5 银行转帐 Pi-演算流图

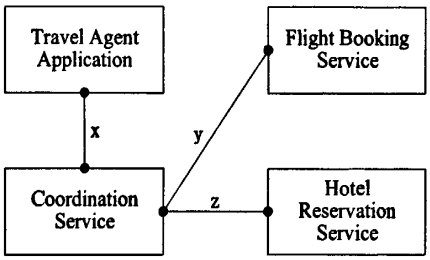

图 3.6 旅行安排 Pi-演算流图

根据表 3.1 的映射关系，将银行转帐和旅行安排抽象成 Pi-演算流图，分别如图 3.5 和图 3.6 所示。

表 3.2 给出了图 3.5 中的进程、通道含义和缩写形式。

表 3.2 银行转帐进程、通道含义和缩写形式对照表

进程、通道名称	缩写形式	含义
Transfer Coordinating	TransferCoord	整个转帐协调过程
Transfer Money Application	Transfer	银行转帐应用程序
Coordination Service	Coordinator	协调器 Web 服务
Communication Bank Service	ComBank	交通银行 Web 服务
Agricultural Bank Service	AgrBank	农业银行 Web 服务
x	x	银行转帐应用程序与协调器之间的通道
y	y	交通银行 Web 服务与协调器之间的通道
z	z	农业银行 Web 服务与协调器之间的通道

表 3.3 给出了图 3.6 中的进程、通道含义和缩写形式。

表 3.3 旅行安排进程、通道含义和缩写形式对照表

进程、通道名称	缩写形式	含义
Travel Coordinating	TravelCoord	整个旅行安排协调过程
Travel Agent Application	Travel	旅行代理应用程序
Coordination Service	Coordinator	协调器 Web 服务
Flight Booking Service	Flight	飞机订票 Web 服务
Hotel Reservation Service	Hotel	宾馆预订 Web 服务
x	x	旅行代理应用程序与协调器之间的通道
y	y	飞机订票 Web 服务与协调器之间的通道
z	z	宾馆预订 Web 服务与协调器之间的通道

下面将根据图 2.4 Completion 协议状态转换图和图 2.5 两阶段提交协议状态转换图对银行转帐的协调过程进行 Pi-演算建模。模型中传递的消息与状态转换图中的含义和写法一致，这里不

再赘述，只给出各通道中传递的消息。

在 x 通道中传递的消息有：Rollback, Commit, Aborted, Committed.

在 y 与 z 通道中传递的消息相同，有：Rollback, Commit, Prepared, ReadOnly, Aborted, Committed.

(1) 银行转帐应用程序

银行转帐应用程序是整个转帐过程的发起者，在协调过程中，应用程序可以告诉协调器是要提交还是中止原子事务。在事务结束后，返回给应用程序一个状态。应用程序与协调器的交互遵循 Completion 协议，其对应的 Pi-演算进程表述如下：

$\tilde{a} = \{Rollback, Commit, Aborted, Committed\}$,

$Transfer(\tilde{a}) = \overline{x} < Rollback > .x(msg).[msg = Aborted]\mathbf{0} + \overline{x} < Commit > .x(msg2).$

$([msg2 = Aborted]\mathbf{0} + [msg2 = Committed]\mathbf{0})$

(2) 协调器 Web 服务

协调器包括激活服务、注册服务和协调协议服务，对协调器进行建模是一个抽象的过程，只关注协调过程，并不关注某个消息由哪个具体的服务发送和接收的。因此将激活服务、注册服务和协调协议服务抽象成一个整体——协调器 Web 服务——对外提供协调服务。协调器负责事务的协调过程，它接收应用程序发来的提交或中止消息，然后与参与者（交通银行 Web 服务和农业银行 Web 服务）交互完成两阶段提交协议，并给应用程序返回事务完成的情况，其对应的 Pi-演算进程表述如下：

$\tilde{b} = \{Rollback, Prepare, Commit, Prepared, ReadOnly, Aborted, Committed\}$,

$Coordinator(\tilde{b}) = x(msg).([msg = Rollback]\overline{y} < Rollback > .y(msg2).[msg2 = Aborted]$

$\overline{z} < Rollback > .z(msg3).[msg3 = Aborted]\overline{x} < Aborted > .\mathbf{0} + [msg = Commit]$

$\overline{y} < Prepare > .y(msg4).([msg4 = Aborted]\overline{z} < Rollback > .z(msg5).$

$[msg5 = Aborted]\overline{x} < Aborted > .\mathbf{0} + [msg4 = ReadOnly]\overline{z} < Rollback > .z(msg6).$

$[msg6 = Aborted]\overline{x} < Aborted > .\mathbf{0} + [msg4 = Prepared]\overline{z} < Prepare > .z(msg7).$

$([msg7 = Aborted]\overline{y} < Rollback > .y(msg8).[msg8 = Aborted]\overline{x} < Aborted > .\mathbf{0} +$

$[msg7 = ReadOnly]\overline{y} < Rollback > .y(msg9).[msg9 = Aborted]\overline{x} < Aborted > .\mathbf{0} +$

$[msg7 = Prepared]\overline{y} < Commit > .y(msg10).[msg10 = Committed]$

$\overline{z} < Commit > .z(msg11).[msg11 = Committed]\overline{x} < Committed > .\mathbf{0})))$

(3) 交通银行 Web 服务

交通银行 Web 服务作为参与者参与两阶段提交协议的完成。在第一阶段，它接收来自协调器的消息，如果是中止则直接中止。在第一阶段结束时，协调器会收到它根据自身状态发出的消息。如果它已经准备好，在收到第二阶段的提交请求后提交已完成的任务，并向协调器发送

确认信息。它的 Pi-演算进程表述如下：

$\tilde{c} = \{Rollback, Prepare, Commit, Prepared, ReadOnly, Aborted, Committed\}$,

$ComBank(\tilde{c}) = y(msg).([msg = Rollback]\overline{y} < Aborted > .0 + [msg = Prepare]$

$(\overline{y} < Aborted > .0 + \overline{y} < ReadOnly > .0 + \overline{y} < Prepared > .y(msg2).$

$([msg2 = Rollback]\overline{y} < Aborted > .0 + [msg2 = Commit]\overline{y} < Committed > .0)))$

(4) 农业银行 Web 服务

农业银行 Web 服务进程和交通银行 Web 服务一样作为参与者参与两阶段提交协议的完成。它的 Pi-演算进程表述如下：

$\tilde{d} = \{Rollback, Prepare, Commit, Prepared, ReadOnly, Aborted, Committed\}$,

$AgrBank(\tilde{d}) = z(msg).([msg = Rollback]\overline{z} < Aborted > .0 + [msg = Prepare]$

$(\overline{z} < Aborted > .0 + \overline{z} < ReadOnly > .0 + \overline{z} < Prepared > .z(msg2).$

$([msg2 = Rollback]\overline{z} < Aborted > .0 + [msg2 = Commit]\overline{z} < Committed > .0)))$

(5) 整个银行转帐协调过程

整个转帐协调过程是各参与者之间交互过程的总和，其 Pi-演算进程可用参与者进程的并行表示：

$TransferCoord(\tilde{a} \cup \tilde{b} \cup \tilde{c} \cup \tilde{d}) = Transfer(\tilde{a}) \mid Coordinator(\tilde{b}) \mid ComBank(\tilde{c}) \mid AgrBank(\tilde{d})$

下面将根据图 2.6 BusinessAgreementWithParticipantCompletion 状态转换图对旅行安排中的协调过程进行 Pi-演算建模。模型中传递的消息与状态转换图中的含义和写法一致，这里不再赘述，只给出各通道中传递的消息。

在 x 通道中传递的消息有：Aborted, Committed.

在 y 与 z 通道中传递的消息相同，有：Aborted, Committed, Close, Cancel, Compensate, Failed, Exited, Completed, NotCompleted, Fail, Compensated, Closed, Canceled, Exit, CannotComplete.

(1) 旅行安排应用程序

在 WS-AT 协议中，Completion 协议被定义来驱动协调过程的开始，而在 WS-BA 协议中并没有定义类似的协议，事实上也并不需要，因为 WS-BA 适用于长事务的协调，应用程序无需驱动事务提交。因此，本文规定应用程序只接收协调器的协调结果消息，其 Pi-演算进程表述如下：

$\tilde{e} = \{Committed, Aborted\}$,

$Travel(\tilde{e}) = x(msg).([msg = Committed]0 + [msg = Aborted]0)$

(2) 协调器 Web 服务

与银行转帐场景中的协调器一样，旅行安排中的协调器也是一个抽象概念，是将激活服务、注册服务和协调协议服务抽象成的一个整体。协调器负责事务的协调过程，如果参与者（飞机

订票 Web 服务和宾馆预订 Web 服务）都完成，则向应用程序发送成功提交的消息；如果只有一个成功提交，则让补偿已经提交的参与者执行补偿操作，同时向应用程序发送提交失败消息；如果两个参与者均不能成功提交，则向应用程序发送提交失败消息。其对应的 Pi-演算进程表述如下：

$$\tilde{f} = \{Aborted, Committed, Close, Cancel, Compensate, Failed, Exited, Completed,$$
$$NotCompleted, Fail, Compensated, Closed, Canceled, Exit, CannotComplete\},$$
$$Coordinator(\tilde{f}) = y(msg).([msg = Exit]\overline{y} < Exited > .\overline{z} < Cancel > .$$
$$z(msg2).([msg2 = Exit]\overline{z} < Exited > .\overline{x} < Aborted > .\mathbf{0} + [msg2 = Canceled]$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg2 = Fail]\overline{z} < Failed > .\overline{x} < Aborted > .\mathbf{0} +$$
$$[msg2 = CannotComplete]\overline{z} < NotCompleted > .\overline{x} < Aborted > .\mathbf{0} +$$
$$[msg2 = Completed]\overline{z} < Compensate > .z(msg3).([msg3 = Compensated]$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg3 = Fail]\overline{z} < Failed > .\overline{x} < Aborted > .\mathbf{0})) +$$
$$[msg = Fail]\overline{y} < Failed > .\overline{z} < Cancel > .z(msg4).([msg4 = Exit]\overline{z} < Exited > .$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg4 = Canceled]\overline{x} < Aborted > .\mathbf{0} + [msg4 = Fail]$$
$$\overline{z} < Failed > .\overline{x} < Aborted > .\mathbf{0} + [msg4 = CannotComplete]\overline{z} < NotCompleted > .$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg4 = Completed]\overline{z} < Compensate > .z(msg5).$$
$$([msg5 = Compensated]\overline{x} < Aborted > .\mathbf{0} + [msg5 = Fail]\overline{z} < Failed >$$
$$\overline{x} < Aborted > .\mathbf{0})) + [msg = CannotComplete]\overline{y} < NotCompleted > .\overline{z} < Cancel > .$$
$$z(msg6).([msg6 = Exit]\overline{z} < Exited > .\overline{x} < Aborted > .\mathbf{0} + [msg6 = Canceled]$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg6 = Fail]\overline{z} < Failed > .\overline{x} < Aborted > .\mathbf{0} +$$
$$[msg6 = CannotComplete]\overline{z} < NotCompleted > .\overline{x} < Aborted > .\mathbf{0} +$$
$$[msg6 = Completed]\overline{z} < Compensate > .z(msg7).([msg7 = Compensated]$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg7 = Fail]\overline{z} < Failed > .\overline{x} < Aborted > .\mathbf{0})) +$$
$$[msg = Completed]z(msg8).([msg8 = Exit]\overline{z} < Exited > .\overline{y} < Compensate > .$$
$$y(msg9).([msg9 = Compensated]\overline{x} < Aborted > .\mathbf{0} + [msg9 = Fail]$$
$$\overline{y} < Failed > .\overline{x} < Aborted > .\mathbf{0}) + [msg8 = Fail]\overline{z} < Failed > .$$
$$\overline{y} < Compensate > .y(msg10).([msg10 = Compensated]\overline{x} < Aborted > .\mathbf{0} +$$
$$[msg10 = Fail]\overline{y} < Failed > .\overline{x} < Aborted > .\mathbf{0}) + [msg8 = CannotComplete]$$
$$\overline{z} < NotCompleted > .\overline{y} < Compensate > .y(msg11).([msg11 = Compensated]$$
$$\overline{x} < Aborted > .\mathbf{0} + [msg11 = Fail]\overline{y} < Failed > .\overline{x} < Aborted > .\mathbf{0}) +$$
$$[msg8 = Completed]\overline{y} < Close > .y(msg12).[msg12 = Closed]\overline{z} < Close > .$$
$$z(msg13).[msg13 = Closed]\overline{x} < Committed > .\mathbf{0}))$$

(3) 飞机订票 Web 服务

飞机订票 Web 服务主动向协调器发送自身完成的情况，如果完成不了则直接退出当前业务活动；如果成功完成则等待协调器的进一步指令。如果是补偿指令，则进行补偿操作，并向协调器发送完成的情况；如果是完成指令，则表明宾馆预订 Web 服务也成功完成了，事务成功结束。它的 Pi-演算进程表述如下：

$$\tilde{g} = \{Close, Cancel, Compensate, Failed, Exited, NotCompleted, Completed, Fail,$$
$$Compensated, Closed, Canceled, Exit, CannotComplete\},$$
$$Flight(\tilde{g}) = \overline{y} < Exit > .y(msg).([msg = Exited]\mathbf{0} + [msg = Cancel]$$
$$\overline{y} < Canceled > .\mathbf{0}) + \overline{y} < Fail > .y(msg2).([msg2 = Failed]\mathbf{0} +$$
$$[msg2 = Cancel]\overline{y} < Canceled > .\mathbf{0}) + \overline{y} < CannotComplete > .y(msg3).$$
$$([msg3 = NotCompleted]\mathbf{0} + [msg3 = Cancel]\overline{y} < Canceled > .\mathbf{0}) +$$
$$\overline{y} < Completed > .y(msg4).([msg4 = Close]\overline{y} < Closed > .\mathbf{0} +$$
$$[msg4 = Compensate](\overline{y} < Compensated > .\mathbf{0} + \overline{y} < Fail > .y(msg5).$$
$$[msg5 = Failed]\mathbf{0}) + [msg4 = Cancel]\overline{y} < Canceled > .\mathbf{0})$$

(4) 宾馆预订 Web 服务

宾馆预订 Web 服务与飞机订票 Web 服务完成的工作相同，它主动向协调器发送自身完成的情况，如果完成不了则直接退出当前业务活动；如果成功完成则等待协调器的进一步指令。如果是补偿指令，则进行补偿操作，并向协调器发送完成的情况；如果是完成指令，则表明飞机订票 Web 服务也成功完成了，事务成功结束。它的 Pi-演算进程表述如下：

$$\tilde{h} = \{Close, Cancel, Compensate, Failed, Exited, NotCompleted, Completed, Fail,$$
$$Compensated, Closed, Canceled, Exit, CannotComplete\},$$
$$Flight(\tilde{h}) = \overline{z} < Exit > .z(msg).([msg = Exited]\mathbf{0} + [msg = Cancel]$$
$$\overline{z} < Canceled > .\mathbf{0}) + \overline{z} < Fail > .z(msg2).([msg2 = Failed]\mathbf{0} +$$
$$[msg2 = Cancel]\overline{z} < Canceled > .\mathbf{0}) + \overline{z} < CannotComplete > .z(msg3).$$
$$([msg3 = NotCompleted]\mathbf{0} + [msg3 = Cancel]\overline{z} < Canceled > .\mathbf{0}) +$$
$$\overline{z} < Completed > .z(msg4).([msg4 = Close]\overline{z} < Closed > .\mathbf{0} +$$
$$[msg4 = Compensate](\overline{z} < Compensated > .\mathbf{0} + \overline{z} < Fail > .z(msg5).$$
$$[msg5 = Failed]\mathbf{0}) + [msg4 = Cancel]\overline{z} < Canceled > .\mathbf{0})$$

(5) 整个旅行安排协调过程

整个旅行安排协调过程是各参与者之间交互过程的总和，其 Pi-演算进程可用参与者进程的并行表示：

$$TravelCoord(\tilde{e} \cup \tilde{f} \cup \tilde{g} \cup \tilde{h}) = Travel(\tilde{e}) \mid Coordinator(\tilde{f}) \mid Flight(\tilde{g}) \mid Hotel(\tilde{h})$$

通过对银行转帐和旅行安排两个应用场景进行 Pi-演算建模，验证了本章给出的 Pi-演算建

模规则的可行性。

3.4 本章小结

本章对基于 Pi-演算的 WS-TX 协议应用建模进行了研究。首先介绍了 Pi-演算，分析了为什么要使用 Pi-演算进行建模。然后对 WS-TX 协议应用场景中元素与 Pi-演算元素的映射关系进行了研究，并根据 Pi-演算的五类动作讨论了 WS-TX 协议应用场景的 Pi-演算建模规则。最后，给出应用 WS-AT 的银行转帐场景和应用 WS-BA 的旅行安排场景，并对它们进行了 Pi-演算建模，验证了建模规则的可行性。

第四章 Pi-演算模型的 SMV 程序表述

第三章已经使用 Pi-演算对 WS-TX 应用场景进行了建模，但是现有对 Pi-演算模型进行形式化验证的系统如 MWB（Mobile Workbench）[52]和 HAL (History Dependant Automata Laboratory) [53]都无法满足本文模型检测的要求。因为 MWB 是一种证据系统，可以验证 Pi-演算模型的正确性，主要关注并发、死锁等结构特性，缺乏对模型逻辑需求的验证。而 HAL 虽然可以进行行为等价的检测和模型性质的检测，但由于 HAL 运行时将 Pi-演算表达式转化为 FSM（Finite State Machine，有限状态机），很容易出现状态爆炸的问题。由于 NuSMV2 使用 on-the-fly 技术实现边组合边验证，在一定程度上解决了状态爆作问题，本文将选用它作为模型检测工具。本章的工作是把 Pi-演算转化为 SMV 程序代码以便利用 NuSMV2 进行模型检测。

4.1 基于 NuSMV2 的模型检测方法

NuSMV2[57]是一个开源的模型检测工具，用于验证有限状态系统是否满足时序逻辑（Temporal Logic）的说明，属于符号模型检测系统。在使用 NuSMV2 进行模型检测时，首先将要验证的系统使用 SMV 语言表述成有限状态迁移系统，然后使用 LTL(Linear Temporal Logic，线性时序逻辑)或 CTL(Computation Tree Logic，计算树逻辑)对需求说明进行描述。NuSMV2 进行工作时，通过探索状态迁移系统的所有状态空间来自动检测系统是否满足需求说明。NuSMV2 并不在内存中构建系统的自动机，而是使用 on-the-fly 技术实现边组合边验证，这在一定程度上解决了状态爆作问题。与其它模型检测工具一样，在性质得不到满足时，NuSMV2 会给出反例。

下面分别介绍描述系统的 SMV 语言和描述需求的 LTL/CTL。

NuSMV2 的输入语言（简称 SMV 语言）被设计用于描述有限状态系统。它提供如下数据结构：Booleans，bounded integer subranges 和 symbolic enumerated types。它还允许定义基本数据类型的有界数组。在使用 SMV 语言进行表述时，可以把复杂的系统分为若干模块（modules），并且每个模块可被实例化多次，这一功能为模块化设计提供了可能。一个 SMV 程序从顶层的模块——Main 模块——开始，Main 模块没有参数。一般模块由三个基本元素组成：VAR，ASSIGN 和 SPEC。VAR 部分用于定义变量，ASSIGN 部分用于说明变量的初始值和变迁关系，SPEC 部分用于说明系统要验证的性质，使用 LTL/CTL 描述。

CTL 和 LTL 是两种非常有用的时序逻辑，其主要差别在于如何处理展开 Kripke 结构所对应的计算树分支。在 CTL 中，时态运算符限定于从一个给定的状态开始的所有可能路径上；而在 LTL 中，时态运算符仅限定于描述从一个给定的状态开始的某条路径上的事件。

LTL 主要用于描述两种特性：安全性和活性。安全性通常用于描述某种不良性质 φ 永不出现 $(\mathbf{G}\neg\varphi)$，而活性则描述某种良好的性质一直保持 $(\mathbf{GF}\psi \text{ or } \mathbf{G}(\phi \rightarrow \mathbf{F}\psi))$。LTL 是 CTL 的

子集。CTL 允许给出路径修饰符，整个系统的演化也是从某个起始状态开始的，但可以有不同的分支，即未来发展是不确定的。

CTL 表达式的语法规则如下[58]：

(1) 任何原子命题是一个 CTL 表达式；

(2) 如果α和β是 CTL 表达式，那么$\alpha \bullet \beta$和$\neg\alpha$也是 CTL 表达式，其中$\bullet$为任何布尔连接符（$\wedge,\vee,\cdots$）；

(3) 如果α和β是CTL表达式，那么$\mathbf{EX}\alpha$，$\mathbf{EG}\alpha$，$\mathbf{E}[\alpha\mathbf{U}\beta]$也是CTL表达式。

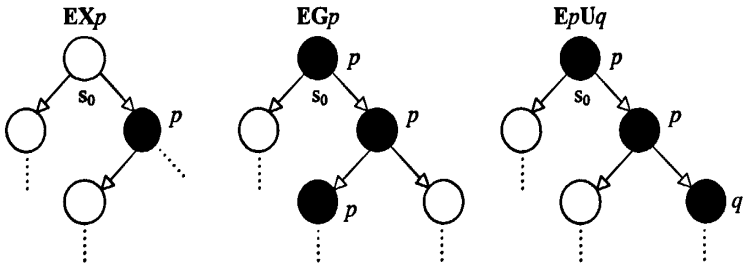

图 4.1 CTL 表达式算子含义

图 4.1 给出了 CTL 表达式直观上的含义。$\mathbf{EX}\alpha$ 表示存在（E）一条从状态 s_0 开始的路径，在这条路径中的下一个（X）状态α满足。$\mathbf{EG}\alpha$ 表示存在一条从状态 s_0 开始的路径，在这条路径中α全部（G）满足。$\mathbf{E}[\alpha\mathbf{U}\beta]$表示存在一条从状态 s_0 开始的路径，在这条路径中直到（U）β满足前α都满足。其它的 CTL 算子（比如$\mathbf{AF}\alpha$表示在所有的路径中最终α满足）可以用以上三个算子表示，如表 4.1 所示。

表 4.1 CTL 算子定义

算子	含义
$\mathbf{AX}\alpha = \neg\mathbf{EX}(\neg\alpha)$	对所有的路径，下一状态α满足
$\mathbf{EF}\alpha = \mathbf{E}[\mathrm{TU}\alpha]$	存在一条路径，使得α最终满足
$\mathbf{AG}\alpha = \neg\mathbf{EF}(\neg\alpha)$	对所有的路径，α全部满足
$\mathbf{A}[\alpha\mathbf{U}\beta] = \neg\mathbf{E}[\neg\beta\mathbf{U}\neg\alpha \wedge \neg\beta] \wedge \neg\mathbf{EG}\neg\beta$	对所有的路径，直到β满足前α都满足
$\mathbf{AF}\alpha = \mathbf{A}[\mathrm{TU}\alpha]$	对所有的路径，α最终满足

本文将使用 CTL 来描述系统的性质。LTL 比 CTL 缺少了两个路径算子 A 和 E，关于 LTL 的详细介绍见文献[58]。

4.2 Pi-演算模型到 SMV 程序代码的转换

由于 MWB 缺乏对模型逻辑需求的验证，而 HAL 存在状态爆作问题，很多研究[54-56]尝试将 Pi-演算模型转化为 SMV 程序代码，使用 NuSMV2 进行模型检测。文献[56]为了解决 UML 状态图缺少严格的语义而无法进行形式化分析和推理的问题，将 UML 状态图转化为 Pi-演算表

达式，再基于 UML 状态图的语义信息将 Pi-演算表达式转化为 SMV 程序代码，并进行模型检测，验证系统性质。该方法从 LTS（Labelled Transition System，标号变迁系统）与 Kripke 结构的映射关系着手进行转换，是文本系统到文本系统之间的转换，在转换时不会涉及到状态爆炸问题。然而在转换时，使用了 UML 状态图的语义作为转换条件。

文献[54]将 BPEL 代码转化为 Pi-演算，再把 Pi-演算转化为 SMV 程序代码进行模型检测。然而在将 Pi-演算转化 SMV 程序代码时，先将 Pi-演算转换为 FSM（其转换方法同文献[53]），再将 FSM 转化为 SMV 程序代码。该方法的缺点在于构建 FSM 时容易引起状态爆作。

以上方法各有优缺点，文献[56]的方法不会引起状态爆作，但不够通用；文献[54]的方法通用但会有状态爆炸的问题。下面将尝试找到一种不使用 Pi-演算模型所对应应用的语义信息作为转换条件、又不会引起状态爆炸的转换规则，将 Pi-演算进程表达式转化为 SMV 程序代码。

4.2.1 理论基础

Pi-演算属于 LTS（Labelled Transition System，标号变迁系统），而 SMV 程序属于 KS（Kripke structure，Kripke 结构）。LTS 与 KS 的区别在于 LTS 强调通过动作来表示系统，而 KS 通过状态的变化来表达系统。但 LTS 和 KS 的共同特点是它们都属于状态自动机（State Automata），本节将从研究 LTS 到 KS 的转化开始找出 Pi-演算进程到 SMV 程序的转换规则。

下面将分别给出 LTS 和 KS 的定义，以及从 LTS 到 KS 的转换方法[59]。

定义4.1 标号变迁系统

标号变迁系统是一个四元组 $M=(S,I,A,R)$，其中：

- S为状态的集合；
- I为初始状态的集合；
- A为有限、非空动作的集合，内部动作$\tau \notin A$；
- R表示变迁关系，$R \subseteq S \times (A \cup \{\tau\}) \times S$，元素$(r,\alpha,s) \in R$称为一个变迁，常用$r \xrightarrow{\alpha} s$表示。

定义4.2 Kripke结构

假定AP为原子断言的集合，Kripke结构是一个四元组 $M_k=(S_k,I_k,R_k,L_k)$，其中：

- S_k为有限状态集；
- $I_k \subseteq S_k$表示初始状态集；
- $R_k \subseteq S_k \times S_k$表示状态间的变迁关系，即$\forall s_k \in S_k$，$\exists s_k' \in S_k$，使得$(s_k,s_k') \in R_k$；
- L_k：$S_k \to 2^{AP}$为标记函数，即对每个状态$s_k \in S_k$，$L_k(s_k)$返回在该状态成立的原子断言的集合。

定义4.3 从标号迁移系统LTS到Kripke结构KS的映射

给定标号迁移系统$M=(S,I,A,R)$，并且符号$\perp$不在A中出现，那么Kripke结构可以定义

为 $M_k=(S_k,I_k,R_k,L_k)$，其中：

- $S_k=S\cup\{(r,a,s)\mid a\in A,r\in S,s\in S,r\xrightarrow{a}s\}$；
- $I_k=I$；
- $R_k=\{(r,s)\mid r\xrightarrow{\tau}s\}\cup\{(r,(r,a,s))\mid r\xrightarrow{a}s\}\cup\{((r,a,s),s)\mid r\xrightarrow{a}s\}$；
- $AP=A\cup\{\perp\}$，对于 $r,s\in S$，$a\in A$：$L_k(s)=\{\perp\}$ 且 $L_k((r,a,s))=\{a\}$。

LTS到KS的转换是将LTS中的动作在KS中用状态表示，即在LTS中的变迁关系 (r,a,s) 在KS中用两个变迁关系 $(r,(r,a,s))$ 和 $((r,a,s),s)$ 表示，(r,a,s) 为KS中新增加的状态。

3.2 节给出了使用 Pi-演算对系统进行建模的建模规则，这些建模规则是 Pi-演算建模方法的子集。使用这些子集所建的模型同样是属于 LTS，称之为“基于 Pi-演算的标号变迁系统”（Pi-Calculus Based Labeled Transaction System，记为 LTS_π），下面给出 LTS_π 的定义。

定义 4.4 基于 Pi-演算的标号变迁系统

基于 Pi-演算的标号变迁系统是一个四元组 $M_\pi=(S_\pi,I_\pi,A_\pi,R_\pi)$，其中：

- S_π 为状态的集合；
- $I_\pi\subseteq A_\pi$ 为初始状态的集合；
- $A_\pi=A_{in}\cup A_{out}\cup\{\tau\}\cup\{\varsigma\}\cup\{\rho\}$ 为动作的集合，A_{in} 为Pi-演算中的输入动作并紧跟匹配操作 $a(x).[x=Var]$ 的集合，A_{out} 为Pi-演算中的输出动作 $\overline{a}<x>$ 的集合，τ 为Pi-演算中的内部哑元动作，ς 为Pi-演算中的选择操作，ρ 为Pi-演算中的并行操作。
- $R_\pi\subseteq S_\pi\times A_\pi\times S_\pi$ 是当前状态与其后续状态的变迁关系，变迁标记为一个动作。

与文献[59]中的 LTS 定义不同，LTS_π 的动作 A_π 带有参数。

状态集合 S_π 中的状态用 $P_i(i\in N^+)$ 标识。$\forall P_i,P_j(i,j\in N^+,i\neq j)$，$P_i\neq P_j$。标识 Pi-演算进程状态的过程如下：

(1) 画出 Pi-演算进程表达的状态变迁图，构成一棵树；

(2) 将树的叶子结点加入到 S_π 中；

(3) 对于非叶子结点，从底层向上、从右到左依次使用 $P_i(i\in N^+)$ (i 从小到大，步长为 1，不可重复)标识，并将 P_i 加入 S_π 中；

(4) 将 S_π 中的状态 **0** 用 NIL 替换。

下面将通过一个实例来说明 Pi-演算进程的状态求解过程。

例： 求出 Pi-演算进程 $P(a,b,Var,Var1,Var2)=\overline{a}<Var>.a(msg).([msg=Var1].\mathbf{0}+[msg=Var2]\overline{b}<Var2>.P(a,b,Var,Var1,Var2))$ 的状态集合 S_π.

画出该 Pi-演算进程的状态变迁图，如图 4.2 所示。

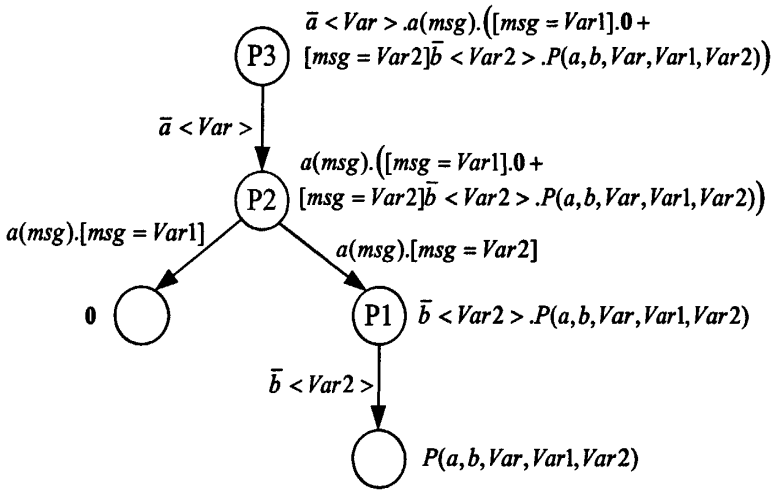

图 4.2 Pi-演算进程的状态变迁图

经过对状态图中的节点进行标识，得出 $S_\pi=\{NIL,P,P1,P2,P3\}$。

LTS到KS的转换是将LTS中的动作映射为KS的状态。在基于Pi-演算的标号变迁系统 LTS_π 中，动作分为五类：输出、输入、哑元、选择和并行。下面讨论在这五类动作发生的情况下如何将 LTS_π 映射为KS。

定义4.5 从标号变迁系统 LTS_π 到Kripke结构 KS 的映射

给定标号变迁系统 LTS_π 定义为 $M_\pi=(S_\pi,I_\pi,A_\pi,R_\pi)$，其相应的Kripke结构 KS 定义为 $M_k=(S_k,I_k,R_k,L_k)$，满足如下条件：

（1）

$$\begin{aligned}&\forall s_1,s_2\in S_\pi,\forall\bar{a}<Var>\in A_{out}\subseteq A_\pi.\\&\quad(s_1,\bar{a}<Var>,s_2)\in R_\pi\\&\Rightarrow s_1,s_2,a=Var\in S_k\\&\wedge\ (s_1,s_2)\in R_k\\&\wedge\ (s_1,a=Var)\in R_k\\&\wedge\ (a=Var,s_2)\in R_k\end{aligned}$$

条件（1）描述了 LTS_π 中两个状态的变迁是输出动作触发的情况下如何映射为 KS 中的状态和变迁关系。这里使用 $a=Var$ 表示输出动作对应的 KS 中的状态。

为了说明如何将 KS 转化为 SMV 程序，规定存在变量 state，使得 $\forall s\in S_\pi$ 均为变量 state 的枚举值，用 $P_i(i\in N^+)$ 表示（$P1$ 对应 s_1，$P2$ 对应 s_2，$P3$ 对应 s_3，……）；同时规定在存在通道 a 的情况下引入变量 a，在通道 a 中传递的消息均为变量 a 的枚举值。在条件（1）下，将会得到如下 SMV 程序片段：

VAR

$state:\{P1,P2,\cdots\}$

$a:\{Var,\cdots\}$

$ASSIGN$

$next(state):=case$

$state=P1\,\&\,a=Var:P2;$ (1)

$\cdots$

$esac;$

$next(a):=case$

$state=P1:Var;$ (2)

$\cdots$

$esac;$

在以上程序片段中，$a=Var$ 既看作布尔表达式，也可以看作赋值语句。这取决于 $a=Var$ 作为前置状态还是后置状态。由于语句(1)中的 $state=P1$ 蕴含着 $a=Var$ (由语句(2)可以推导出)，因此可将语句(1)简写为：

$$state=P1:P2;$$

从以上分析可以看出，在输出动作下，$a=Var$ 为赋值语句，其含义是新产生消息 Var。

（2）

$$\begin{aligned}&\forall s_1,s_2\in S_\pi,\forall a(x).[x=Var]\in A_{in}\subseteq A_\pi.\\&\quad(s_1,a(x).[x=Var],s_2)\in R_\pi\\&\Rightarrow s_1,a=Var,s_2,\in S_k\\&\wedge\ (s_1,s_2)\in R_k\\&\wedge\ (s_1,a=Var)\in R_k\\&\wedge\ (a=Var,s_2)\in R_k\end{aligned}$$

条件（2）描述了 LTS_π 中两个状态的变迁是输入动作触发的情况下如何映射为 KS 中的状态和变迁关系。这里使用 $a=Var$ 表示输入动作对应的 KS 中的状态。在 Pi-演算模型中，一个进程的输入为另一个进程的输出，因此在 Pi-演算模型对应的 KS 中，新引入的动作状态在条件（1）下已经考虑过了，在条件（2）下转化为的 SMV 代码片段将不包含这些状态，$a=Var$ 在这里看作布尔表达式。在条件（2）下，从 KS 会得到如下 SMV 程序片段：

$$
\begin{aligned}
&VAR\\
&\quad state:\{P1,P2,\cdots\}\\
&ASSIGN\\
&\quad next(state):=case\\
&\qquad state=P1\,\&\,a=Var:P2;\\
&\qquad \cdots\\
&\quad esac;
\end{aligned}
$$

在输入动作下，$a=Var$ 为布尔表达式，其含义是在进程接收到一个新的消息，如果消息为 Var 则表现为另一个进程。

（3）

$$
\begin{aligned}
&\forall s_1,s_2\in S_\pi,\tau\in A_\pi.\\
&\quad (s_1,\tau,s_2)\in R_\pi\\
&\Rightarrow s_1,s_2,\tau\in S_k\\
&\wedge\ (s_1,s_2)\in R_k\\
&\wedge\ (s_1,\tau)\in R_k\\
&\wedge\ (\tau,s_2)\in R_k
\end{aligned}
$$

在 Pi-演算中，τ 表示内部哑元动作，表示对外界而言系统没有发生任何改变。为了简化模型的复杂度，将 KS 中的 s_1、s_2 和 τ 三个状态看作是一个状态 s，$s\in S_k$。s 满足如下条件：

$$
\begin{aligned}
&\forall s'\in S_k.(s',s_1)\in R_k\Rightarrow(s',s)\in R_k\\
&\forall s'\in S_k.(s_2,s')\in R_k\Rightarrow(s,s')\in R_k
\end{aligned}
$$

（4）

$$
\begin{aligned}
&\forall s_1,s_2,s_3\in S_\pi,\varsigma\in A_\pi.\\
&\quad (s_1,\varsigma,s_2)\in R_\pi\\
&\vee\ (s_1,\varsigma,s_3)\in R_\pi\\
&\Rightarrow s_1,s_2,s_3,\varsigma\in S_k\\
&\wedge\ (s_1,\varsigma)\in R_k\\
&\wedge\big(\big((s_1,s_2)\in R_k\wedge(\varsigma,s_2)\in R_k\big)\\
&\vee\ \big((s_1,s_3)\in R_k\wedge(\varsigma,s_3)\in R_k\big)\big)
\end{aligned}
$$

在 Pi-演算中，选择动作并不是真实发生的动作，是虚拟的，它表示两个进程选择执行。因此在 KS 结构中，状态 s_1 与状态 ς 可以合并成一个状态。因此条件（4）可以改写成：

$$\begin{aligned}
&\forall s_1, s_2, s_3 \in S_\pi, \varsigma \in A_\pi. \\
&\quad (s_1, \varsigma, s_2) \in R_\pi \\
&\vee (s_1, \varsigma, s_3) \in R_\pi \\
&\Rightarrow s_1, s_2, s_3, \varsigma \in S_k \\
&\wedge \left((s_1, s_2) \in R_k\right. \\
&\vee \left.(s_1, s_3) \in R_k\right)
\end{aligned}$$

在条件（4）下，将会得到如下 SMV 程序片段：

```
VAR
  state:{P1,P2,P3…}
ASSIGN
  next(state):=case
    state=P1:{P2,P3};
    …
  esac;
```

（5）

$$\begin{aligned}
&\forall s_1, s_2, s_3 \in S_\pi, \rho \in A_\pi. \\
&\quad (s_1, \rho, s_2 \mid s_3) \in R_\pi \\
&\Rightarrow s_1, s_2 \mid s_3, \rho \in S_k \\
&\wedge (s_1, s_2 \mid s_3) \in R_k \\
&\wedge (s_1, \rho) \in R_k \\
&\wedge (\rho, s_2 \mid s_3) \in R_k
\end{aligned}$$

在 Pi-演算中，并行动作并不是真实发生的动作，是虚拟的，它表示两个进程并行执行。因此在 KS 中，状态 s_1 与状态 ρ 可以合并成一个状态。因此条件（5）可以改写成：

$$\begin{aligned}
&\forall s_1, s_2, s_3 \in S_\pi, \rho \in A_\pi. \\
&\quad (s_1, \rho, s_2 \mid s_3) \in R_\pi \\
&\Rightarrow s_1, s_2 \mid s_3, \rho \in S_k \\
&\wedge (s_1, s_2 \mid s_3) \in R_k
\end{aligned}$$

在 SMV 程序中，并行执行有同步方法和异步方式。由于 WS-TX 应用场景参与者是异步运行的，在 SMV 程序中也使用异步方法执行进程。如下为条件（5）所对应的 SMV 程序片段：

$$
\begin{aligned}
&VAR\\
&\quad P1Process : process\ P1(a,c);\\
&\quad P2Process : process\ P2(b);\\
&\cdots\\
&MODULE\ P1(a,c)\\
&\cdots\\
&MODULE\ P2(b)\\
&\cdots
\end{aligned}
$$

上述片段中的 a、c 为模块 P1 的参数，b 为模块 P2 的参数。这里的参数为模块中用到的通道。

4.2.2 转换规则

下面将根据定义 4.5 通过一系列的规则来说明如何用 SMV 代码表述 Pi-演算进程表达式。在下面的规则中符号“$\rightarrow$”的左边为 Pi-演算进程表达式，右边的为对应的 SMV 代码表述。

规则 1 整个 WS-TX 应用场景协调过程的 Pi-演算进程映射成 SMV 程序的主函数。

$$P = P_1 \mid P_2 \mid \cdots \rightarrow MODULE\ main$$

这里 $P1$ 和 $P2$ 为 WS-TX 应用场景中的参与者进程。

规则 2 WS-TX 应用场景及其参与者进程中的通道映射为 SMV 程序主函数中的变量，变量名为通道名，每个变量的枚举值为相应的通道发送的消息的集合，并将 null 值加入到集合中表示变量的初始值。

$$
\begin{aligned}
&P = P_1 \mid P_2\\
&P_1 = \cdots \overline{a} < Var1 > \cdots\\
&P_2 = \cdots \overline{b} < Var2 > \cdots
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
&MODULE\ main\\
&VAR\\
&\quad a : \{Var1, \cdots, null\};\\
&\quad b : \{Var2, \cdots, null\};\\
&ASSIGN\\
&\quad init(a) := null;\\
&\quad init(b) := null;
\end{aligned}
$$

规则 3 WS-TX 应用场景参与者进程映射为 SMV 程序中并发执行的子模块，子模块的名称为参与者进程的名称，子模块的参数为相应进程中涉及到的通道名。

$$P = P_1 \mid P_2$$
$$P_1 = \cdots \overline{a} < Var1 > .c(msg) \cdots \rightarrow$$
$$P_2 = \cdots \overline{b} < Var2 > \cdots$$

$$\begin{array}{l} VAR \\ \quad P1Process : process\ P1(a,c); \\ \quad P2Process : process\ P2(b); \\ \cdots \\ MODULE\ P1(a,c) \\ \cdots \\ MODULE\ P2(b) \\ \cdots \end{array}$$

规则 4 一个 Pi-演算进程映射为一个变量 state，变量的枚举值为该进程状态集合 S_π 中的所有元素，变量 state 的初始值为进程名。

规则 5 变量 state 的值在遇到动作时发生改变，其先决条件为当前状态标识和匹配值。

$$P_2 = \overline{a} < V1 > .P_1,$$
$$P_4 = a(x).[x = V2]P_3,$$
$$P_7 = a(x).([x = V3]P_6 + [x = V4]P_5) \rightarrow$$
$$P_{10} = P_9 + P_8,$$

$$\begin{array}{l} ASSIGN \\ \quad next(state) := case \\ \quad\quad state = P_2 : P_1; \\ \quad\quad state = P_4 \,\&\, a = V2 : P_3; \\ \quad\quad state = P_7 \,\&\, a = V3 : P_6; \\ \quad\quad state = P_7 \,\&\, a = V4 : P_5; \\ \quad\quad state = P_{10} : \{P_9, P_8\}; \end{array}$$

规则 6 任何变量 x 的 next(x)语句中均包含：

$$\begin{array}{l} ASSIGN \\ \quad next(x) := case \\ \quad\quad \cdots \\ \quad\quad 1 : x; \\ \quad esac; \end{array}$$

以上代码表示在默认情况下变量的值不会发生改变。

规则 7 通道变量 a 的值（状态）在遇到输出动作时发生改变，其下一状态为输出的消息值，先决条件为该输出动作的前序状态标识。

$$P_2 = \overline{a} < V1 > .P_1 \rightarrow$$

$$\begin{array}{l} ASSIGN \\ \quad next(a) := case \\ \quad\quad state = P_2 : V1; \end{array}$$

将 Pi-演算进程转化 SMV 程序大体分为两步。首先是对 Pi-演算进程进行标识，然后按照规则 1 到规则 7 进行转换。4.3 节将根据以上七规则设计并实现自动转换工具 PiCal2NuSMV.

4.3 转换工具 PiCal2NuSMV 的设计与实现

本节将实现一个从 Pi-演算模型到 SMV 程序代码的自动转换工具 PiCal2NuSMV. 如图 4.3 所示为转换工具 PiCal2NuSMV 的基本架构。PiCal2lNuSMV 共分为三大组件：Pi-演算文本解析器、转换适配器和 SMV 程序产生器。Pi-演算文本解析器用于把 Pi-演算表达式的文本形式转化为内存表示形式，转换适配器基于此形式分析并转换 Pi-演算进程为 SMV 程序的内存表示形式，SMV 程序产生器将 SMV 程序内存表示方式转换为 SMV 程序代码文本。

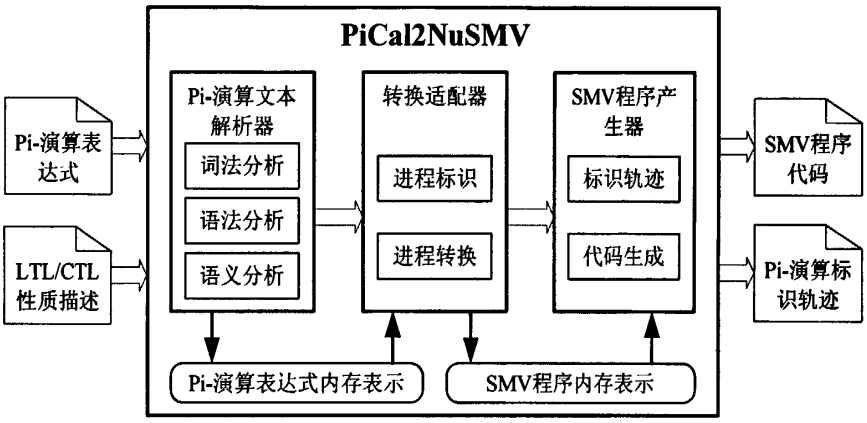

图 4.3 PiCal2NuSMV 基本架构

PiCal2NuSMV 的输入为两个文件：Pi-演算表达式文件和基于 LTL/CTL 的性质描述文件。Pi-演算表达式文件给出了系统的 Pi-演算模型的文本形式，其中第一条进程表达式对应 SMV 程序的 main 模块；性质描述文件每一行给出系统的一条性质，以 LTLSPEC 或 SPEC 开头。PiCal2NuSMV 的输出也为两个文件：SMV 程序代码文件和 Pi-演算标识轨迹文件。SMV 程序代码文件为 Pi-演算表达式对应的 SMV 程序的文本表述，其中也包含了性质描述文件中的内容；Pi-演算标识轨迹文件为标识 Pi-演算进程状态的过程信息，作为分析 NuSMV2 产生的反例的辅助信息，同时为后续集成 PiCal2NuSMV 和 NuSMV2 工具作准备。

PiCal2NuSMV 的可执行文件、使用方式以及本文中的例子可以到 CSDN 空间 http://download.csdn.net/source/1976669 下载。

4.3.1 Pi-演算文本解析器

Pi-演算文本解析器用于把 Pi-演算表达式的文本形式转化为内存表示形式，分为三步：词法分析、语法分析和语义分析。ANTLR（Another Tool for Language Recognition）[60]将被用来辅助完成这一功能。

ANTLR 是一个软件工具，可以基于特定语法产生词法分析程序和语法分析程序。为了产生 Pi-演算的词法和语法分析程序，采用与 MWB 输入语言相似的 Pi-演算语法，其 EBNF(Extended Backus-Naur Form，扩展的巴科斯范式)如图 4.4 所示。

Pi-演算 EBNF 源文件以.g 为后缀名，行 1 表示语法文件的名称为 PiCal.g. 行 2 定义了可选参数部分，如果未定义则使用默认值。ANTLR 使用 LL(k)文法，k=5 定义了每一个 token 向前搜索的字符个数为 5. 行 3 为 token 命令（tokens command），用于为字符串赋予一个符号名称，要使用这个字符串时只要调用对应的符号名称即可。归约时符号名称的优先级高于一般的变量名。行 4 到行 19 为 Pi-演算语法规则的定义，行 20 到行 25 为 Pi-演算词法规则的定义。Pi-演算的 EBNF 中还会嵌入语义分析的代码用于把 Pi-演算文本转化为具有特定含义的内存结构，限于篇幅图 4.4 中没有给出。

```
1 grammar PiCal;
2 options {k=5;}
3 tokens {AGENT  ='agent';  EQ  ='=';  LPAREN  ='(';  RPAREN  =')';
   LSQUARE ='[';  RSQUARE ='];  LANGLE ='<';  RANGLE ='>';
   COMMA  =',';  PLUS  ='+';  PARALLEL='|';  DOT  ='.';
   REST  ='^';  OUT  ='\';  NIL  ='0';  SILENT ='t';}
  /*PARSER RULES*/
4 multiDefn  : agentDefn+;
5 agentDefn  : AGENT agentId EQ process;
6 agentId  : NAME LPAREN nameList RPAREN;
7 nameList  : NAME(COMMA NAME)*;
8 process  : sumExpr;
9 sumExpr  : expr(PLUS expr)*;
10 expr   : inputExpr | outputExpr | silentExpr | parallelExpr  | LPAREN sumExpr RPAREN | NIL;
11 inputExpr  : inputAction DOT match;
12 outputExpr : outputAction DOT expr;
13 silentExpr  : silentAction DOT expr;
14 match   : matchExpr | (LPAREN v_matchExpr=matchExpr
          (PLUS v_matchExpr=matchExpr)+ RPAREN);
15 matchExpr  : LSQUARE NAME EQ v_match=NAME RSQUARE expr;
16 parallelExpr : agentId (PARALLEL v_agentId=agentId)*;
17 inputAction : NAME LPAREN v_value=NAME RPAREN;
18 outputAction: OUT NAME LANGLE NAME RANGLE;
19 silentAction : SILENT;
  /*LEXER RULES*/
20 WHITESPACE   : ('\t'|' '|'\r'|'\n'|'\u000C')+ {$channel=HIDDEN;};
21 NAME          : (LOWER_CASE | CAPITAL_LETTER) BASE_NAME;
22 fragment BASE_NAME :  (LOWER_CASE|CAPITAL_LETTER|DIGIT|'_'|'$'|'\')*;
23 fragment DIGIT                : '0'..'9';
24 fragment CAPITAL_LETTER :  'A'..'Z';
25 fragment LOWER_CASE     : 'a'..'z';
```

图 4.4 Pi-演算的 EBNF

Pi-演算的内存结构如图 4.5 所示。一个 Pi-演算模型(PiCal)由若干条 Pi-演算表达式(Agent)组成，而一条 Pi-演算表达式由一个代理标识（AgentId）和一条表达式（Expr）组成。根据 Pi-演算中的五类动作，PiCal2NuSMV 将表达式分为五类：输入表达式（InputExpr）、输出表达式（OutputExpr）、选择表达式（SelectExpr）、并行表达式（ParallelExpr）和哑元表达式（SilentExpr），它们均继承自 Expr 类。此外， LeafAgentExpr 类和 LeafNilExpr 类也继承自 Expr 类，它们均表

示叶子表达式，即不存在子表达式，标识当前进程结束。LeafAgentExpr 类表示新进程标识，LeafNilExpr 类表示空进程标识。

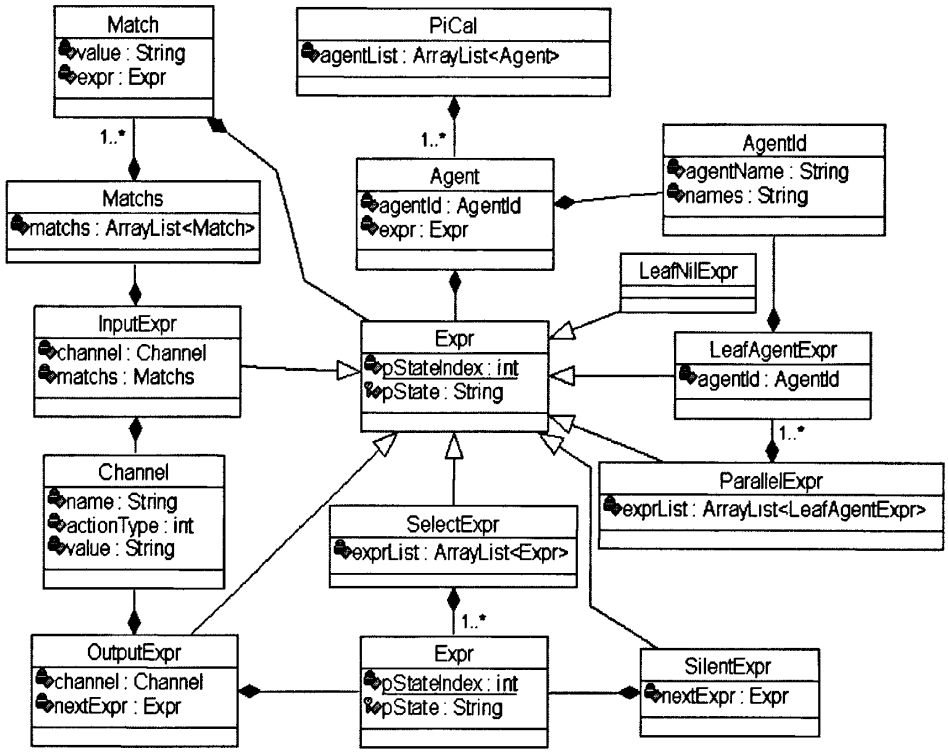

图 4.5 Pi-演算内存结构类图

4.3.2 转换适配器

转换适配器用于把 Pi-演算内存结构转化为 SMV 程序内存结构，为进一步转化为 SMV 程序代码打下基础。转换适配器工作过程在 4.2.2 节已经详细给出，共分为两部分，首先是对 Pi-演算进程进行标识（进程标识），然后按照规则 1 到规则 7 进行转换（进程转换）。

Expr 中提供了 getPState()方法进行 Pi-演算进程的标识，其工作原理是：Expr 中存在一个静态的计数器 pStateIndex，标识的过程是字符‘P’和计数器的数值拼接成一个新的字符串，然后计数据自动加 1，以备对另一表达式标识。

Pi-演算元素与 SMV 程序之间存在映射关系，根据 4.2.2 节的规则 1 到规则 7 将转换代码分布于每一个 Pi-演算元素的内存结构类中，作为一个方法存在，供转换适配器调用。

图 4.6 描述了 SMV 程序的内存结构。一个 SMV 程序（NuSMV）由三部分构成：主模块（MainModule）、子模块（SubModule）和性质描述（Property）。一个 SMV 程序必然存在一个主模块作为入口函数，可能存在零个或多个子模块，也可能存在或不存在性质描述。MainModule 和 SubModule 都继承自 AbstractModule 类。AbstractModule 类有四个属性：模块名称（name）、参数列表（paramList）、变量列表（varList）和包含的子模块列表（subModuleList）。变量类

（Variable）描述了一个变量的所有可能的枚举值、初始值和变迁条件。通过对 SMV 程序内存结构的文本化就可以得到 NuSMV2 可以识别的 SMV 程序文本。

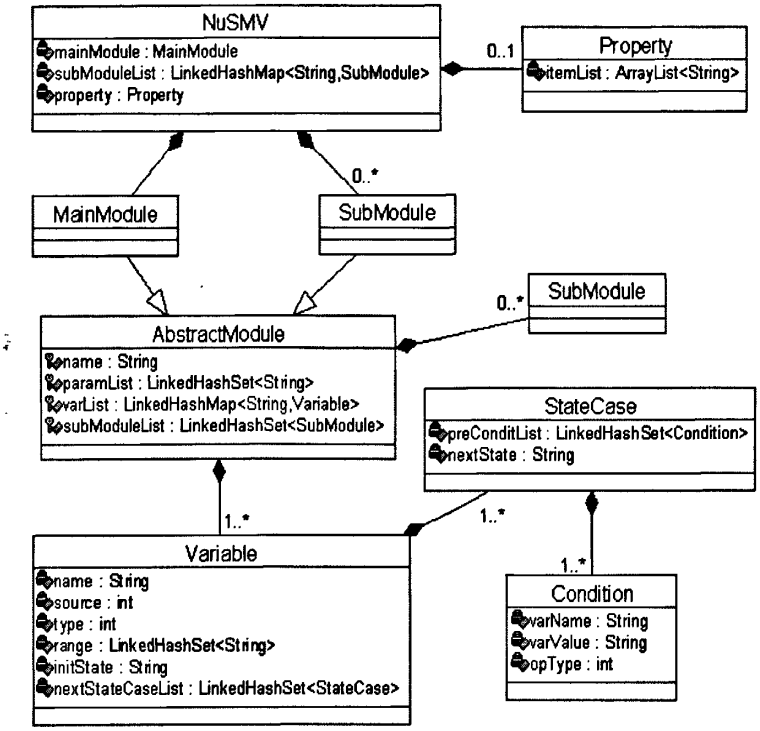

图 4.6 SMV 程序内存结构类图

4.3.3 SMV 程序产生器

SMV 程序产生器将 SMV 程序内存表示方式转换为 SMV 程序代码文本，并可同时产生 Pi-演算进程标识过程信息，为分析模型检测的反例提供辅助信息。

```
/**
 * 将该类字符串化，即以SMV语言的格式表示该Module.
 */
public String toString(){
    ......
    this.genModuleDecl();    //产生Module申明文本
    this.genVarDecl();       //产生变量申明文本
    this.genProDecl();       //产生并行进程申明文本
    this.genAssignStmt();    //产生ASSIGN部分文本
    this.genFairnessStmt(); //产生FAIRNESS部分文本

    return this.moduleString.toString();
}
```

图 4.7 模块的 SMV 代码本文生成程序

图 4.6 中类与 SMV 程序元素相互对应，只要对每个类重载 toString()方法即可，用以完成

对该类对应 SMV 程序元素的文本输出。比如 AbstractModule 类的 toString()方法如图 4.7 所示。其中，行 6 到行 10 分别生成 SMV 程序中一个模块各段的文本。

4.4 本章小结

本章对如何使用 SMV 程序表述 Pi-演算模型进行了研究。首先讨论了为什么要把 Pi-演算模型转化为 SMV 程序代码。其次分析了 NuSMV2 模型检测工具及其输入语言 SMV 程序的特点，并介绍了 NuSMV2 的性质描述语言 CTL。然后从 LTS 到 KS 的转换关系为出发点找出 Pi-演算进程到 SMV 程序代码的转换规则。最后根据转换规则设计并实现了自动转换工具 PiCal2NuSMV.

第五章 基于 NuSMV2 的 WS-TX 协议应用模型检测及分析

模型检测是为了对系统进行性质检验以发现系统设计存在的问题并为纠错提供指导。作为解决事务问题的标准，WS-TX 协议得到了学术界和工业界的广泛关注。一方面，在人们基于 WS-TX 协议设计分布式应用时，不可避免地会出现错误，而这些错误又不易被人发现，对这些错误的检测是分布式系统应用的前提；另一方面， WS-TX 协议的规范说明中并没有对存在多个参与者时如何协调进行说明，因此对 WS-TX 协议的应用有可能存在模糊不清的地方，这种模糊不清将导致 WS-TX 协议应用设计存在先天性不足，验证这种先天性不足是否存在是 WS-TX 协议应用的前提。本章将使用第三章、第四章的研究成果，对第三章中的银行转帐和旅行安排场景实例进行模型检测，检测协调模型是否满足事务特性以及协议本身是否可靠。具体分为如下四步：

(1) 对银行转帐和旅行安排场景协调过程进行 Pi-演算建模；

(2) 将 Pi-演算模型转化为 SMV 程序代码；

(3) 使用 CTL 描述应用场景的性质；

(4) 使用 NuSMV2 进行模型检测并分析。

第一步在第三章已经完成，第二步可以使用第四章实现的 PiCal2NuSMV 转换工具完成。本章着重讨论第三步和第四步。

5.1 WS-TX 协议应用性质分析

对 WS-TX 协议应用场景的验证分为“安全性验证”和“活性验证”[61, 62]，如图 5.1 所示。“安全性验证”从主观上验证 WS-TX 协议的应用场景是否满足事务特性，而“活性验证”则是从客观上验证 WS-TX 协议本身协调参与者的可靠性。

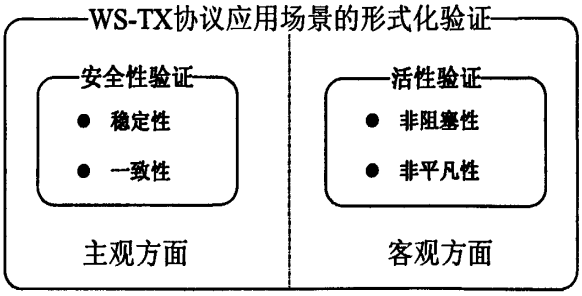

图 5.1 WS-TX 协议应用场景形式化验证方法框架

WS-AT 协议是传统的两阶段提交协议在分布式环境下的应用，具有严格的 ACID 事务特性，而 WS-BA 应用于长事务，不具有严格的 ACID 特性，因此对于 WS-AT 协议与 WS-BA 协议应用场景关于“安全性”与“活性”的解释也不一样，下面将分别介绍。

5.1.1 WS-AT 应用场景的性质分析

定义 5.1 WS-AT 安全性

- 稳定性（Stability）：事务一旦提交或中止，就会永远保持这个状态。
- 一致性（Consistency）：事务的所有参与者要么都提交，要么都中止，不可能一部分提交而另一部分中止。

对 WS-AT 应用场景安全性的验证是为了验证应用场景是否满足事务特性。这里的稳定性指事务的持续性。在银行转帐场景中，稳定性指当协调器向转帐应用程序发送 Committed 消息或者 Aborted 消息后，协调器不能再向参与者（交通银行 Web 服务和农业银行 Web 服务）发送 Rollback 消息。这里的一致性蕴涵了事务的原子性和一致性。在银行转帐场景中，一致性指交通银行 Web 服务和农业银行 Web 服务要么都向协调器发送 Committed 消息，要么都发送 Aborted 消息，不可能一个发送 Committed 消息，而另一个发送 Aborted 消息。

定义 5.2 WS-AT 活性

- 非阻塞性（Non-Blocking）：参与者与协调者都可以达到协议的终止状态。
- 非平凡性（Non-Triviality）：如果所有的参与者都准备好，协调者可能会让所有的参与者提交事务。

对 WS-AT 应用场景活性的验证是为了验证 WS-AT 协议本身是否可靠。非阻塞性保证了协议是会终止的，不会陷入死循环或其它非终止状态，也就是说参与者（交通银行 Web 服务和农业银行 Web 服务）最终都会向协调器发送 Committed、Aborted 或者 ReadOnly 消息，而协调器最终都会向转帐应用程序发送 Committed 或者 Aborted 消息。非平凡性是说每个参与者都会终止，但不希望总是进入 Aborted 终止状态，因为这是一个满足一致性的最简单的情况，这里希望如果可以提交，那么协议会达到 Committed 终止的状态，即参与者均向协调器发送 Committed 消息。

5.1.2 WS-BA 应用场景的性质分析

WS-BA 协议作为 Web 服务事务中业务事务的协调接口，要求该协议具有一系列良好的性质。虽然业务事务没有像原子事务那样具有清晰的 ACID 性质，但是根据一般协议的要求和参照 WS-AT 协议应用场景的安全性和活性的定义，下面相应的给出 WS-BA 应用场景的性质。

定义 5.3 WS-BA 安全性

- 稳定性（Stability）：一旦进入终止状态，就会永保持在这个状态。
- 一致性（Consistency）：所有的参与者要么都进入正常结束状态，要么都进入补偿结束状态，不可能一部分正常结束，而另一部分为补偿结束。

对安全性这两条性质的验证是为了验证使用 WS-BA 协议的应用场景是否满足事务特性。

WS-BA 协议虽然放宽了 ACID 事务特性，但其最终运行结果还是应该保证一致性的。在旅行安排场景中，稳定性是指一旦协调器向旅行安排应用程序发送 Committed 或 Aborted 消息后，协调器不再向参与者发送 Cancel 消息；一致性是指在协调器收到飞机订票 Web 服务和宾馆预订 Web 服务的 Completed 消息后，协调器要么向它们发送 Compensate 消息，要么发送 Close 消息。

定义 5.4 WS-BA 活性

- 非阻塞性（Non-Blocking）：参与者与协调者均可达到协议的终止状态。
- 非平凡性（Non-Triviality）：如果所有的参与者都参与业务活动，那么协议参与者有可能进入正常结束状态。

WS-BA 的活性与 WS-AT 的活性含义一样，都是用来描述协议本身是否可靠。在 WS-BA 的应用场景中，非阻塞性表述为参与者（飞机订票 Web 服务和宾馆预订 Web 服务）最终都会向协调器发送 Closed、Compensated 或 Canceled 消息，或者受到协调器发送的 Exited、Failed 或 NotCompleted 消息，而协调器最终都会向旅行安排应用程序发送 Committed 或 Aborted 消息。非平凡性是指每个参与者都会终止，但不希望错误终止，所以可能正常终止，即协调器可能向参与者均发送 Close 消息。

5.2 基于计算树逻辑 CTL 的 WS-TX 应用模型的性质归约

NuSMV2可以使用时序逻辑CTL和LTL来描述系统的性质。在CTL中，时态运算符限定于从一个给定的状态开始的所有可能路径上；而在LTL中，时态运算符仅限定于描述从一个给定的状态开始的某条路径上的事件。LTL是CTL的子集。由于CTL的表述能力更强，因此本文选用CTL来描述Pi-演算模型的性质。

5.2.1 WS-AT 应用场景性质的 CTL 描述

根据5.1.1节对安全性和活性的定义，下面给出银行转帐场景应该满足的性质：

性质1 稳定性：协调器向转帐应用程序发送Committed消息或者Aborted消息后，协调器不能再向参与者（交通银行Web服务和农业银行Web服务）发送Rollback消息。使用CTL描述为：

EX ((x=Committed | x=Aborted) -> EG !(y=Rollback & z=Rollback))

性质2 一致性：交通银行Web服务和农业银行Web服务要么都向协调器发送Committed消息，要么都发送Aborted消息，不可能一个发送Committed消息，而另一个发送Aborted消息。使用CTL描述为：

EG !((y=Committed & z=Aborted) | (y=Aborted & z=Committed))

性质3 非阻塞性：交通银行Web服务最终都会向协调器发送Committed、Aborted或者ReadOnly消息。使用CTL描述为：

AF (y=Committed | y=Aborted | y=ReadOnly)

性质4 非阻塞性：农业银行Web服务最终都会向协调器发送Committed、Aborted或者ReadOnly消息。使用CTL描述为：

AF (z=Committed | z=Aborted | z=ReadOnly)

性质5 非阻塞性：协调器最终都会向银行转帐应用程序发送Committed或者Aborted消息。使用CTL描述为：

AF (x=Committed | x=Aborted)

性质6 非平凡性：有可能参与者均向协调器发送Committed消息。使用CTL描述为：

EX (y=Committed -> EX z=Committed)

5.2.2 WS-BA 应用场景性质的 CTL 描述

根据5.1.2节对安全性和活性的定义，下面给出旅行安排场景应该满足的性质：

性质7 稳定性：一旦协调器向旅行安排应用程序发送Committed或Aborted消息后，协调器不再向参与者发送Cancel消息。使用CTL描述为：

EX ((x=Committed | x=Aborted) -> EG !(y=Cancel & z=Cancel))

性质8 一致性：在协调器收到飞机订票Web服务和宾馆预订Web服务的Completed消息后，协调器要么向它们发送Compensate消息，要么发送Close消息。使用CTL描述为：

EX ((y=Completed & z=Completed) -> EG !((y=Compensate & z=Close) | (y=Close & z=Compensate)))

性质9 非阻塞性：飞机订票Web服务最终都会向协调器发送Closed、Compensated或Canceled消息，或者受到协调器发送的Exited、Failed或NotCompleted消息。使用CTL描述为：

AF ((y=Closed | y=Compensate | y=Canceled) | (y=Exited | y=Failed | y=NotCompleted))

性质10 非阻塞性：宾馆预订Web服务最终都会向协调器发送Closed、Compensated或Canceled消息，或者受到协调器发送的Exited、Failed或NotCompleted消息。使用CTL描述为：

AF ((z=Closed | z=Compensate | z=Canceled) | (z=Exited | z=Failed | z=NotCompleted))

性质11 非阻塞性：协调器最终都会向旅行安排应用程序发送Committed或Aborted消息。使用CTL描述为：

AF (x=Committed | x=Aborted)

性质12 非平凡性：协调器可能向参与者均发送Close消息。使用CTL描述为：

EX (y=Close -> EX z=Close)

在 SMV 程序中，CTL 性质使用 SPEC 关键字申明。以上 12 条性质将使用 SPEC 申明并根据所描述的不同系统（银行转帐和旅行安排）分别放于两个文本文件中，作为 PiCal2NuSMV 的输入。

5.3 基于 NuSMV2 的模型检测及其分析

本节将对 5.2 节的性质进行模型检测。首先将 3.3 节所建的 Pi-演算模型根据图 4.4 Pi-演算的 EBNF 进行改写，改写成 PiCal2NuSMV 工具能够识别的文本格式。然后使 PiCal2NuSMV 生成包含系统描述和性质描述的 SMV 程序代码，作为 NuSMV2 的输入进行模型检测。银行转帐和旅行安排的性质将被分开进行检测。

本文实验的硬件条件为：CPU Pentium Dual E2180 2.00GHz，内存 1GB。软件条件为：Windows XP 系统，NuSMV2.4。

5.3.1 WS-AT 应用场景性质检测及分析

使用 PiCal2NuSMV 工具生成银行转帐 SMV 程序代码见附录 1. 使用 print_reachable_state 打印出银行转帐系统 FSM 状态数如图 5.2 所示。系统共有 284 个可达状态。使用 check_ctlspec 可以检测 5.2 节定义的性质 1 到性质 6，检测结果如图 5.3 所示。

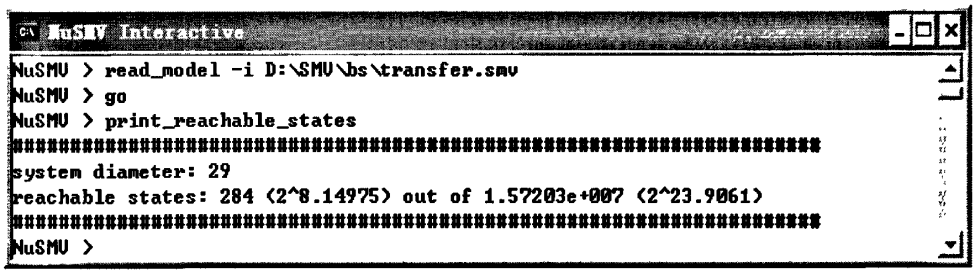

图 5.2 银行转帐模型 FSM 状态数

图 5.3 的模型检测结果说明使用 WS-AT 协议的银行转帐场景满足安全性和活性的要求。这一方面说明了银行转帐系统能够满足事务特性，系统运行能够得到一致性的结果；同时也说明了 WS-AT 协议本身是可靠的。

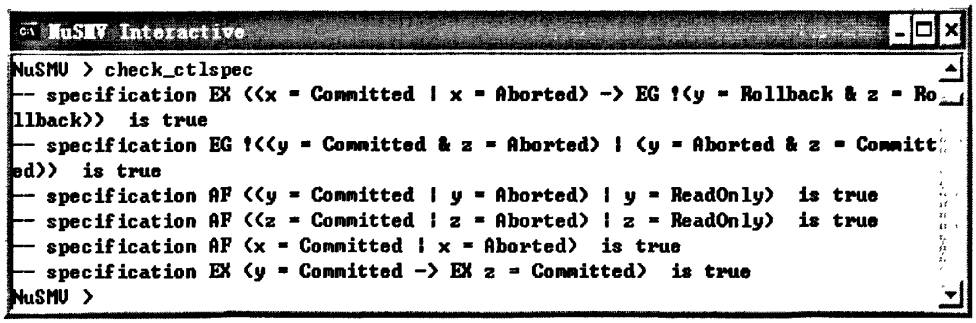

图 5.3 银行转帐模型检测结果

5.3.2 WS-BA 应用场景性质检测及分析

使用 PiCal2NuSMV 工具生成旅行安排 SMV 程序代码见附录 2. 使用 print_reachable_state 打印出旅行安排系统 FMS 状态数如图 5.4 所示。系统共有 2874 个可达状态。使用 check_ctlspec

可以检测 5.2 节定义的性质 7 到性质 12，检测结果如图 5.5 所示。

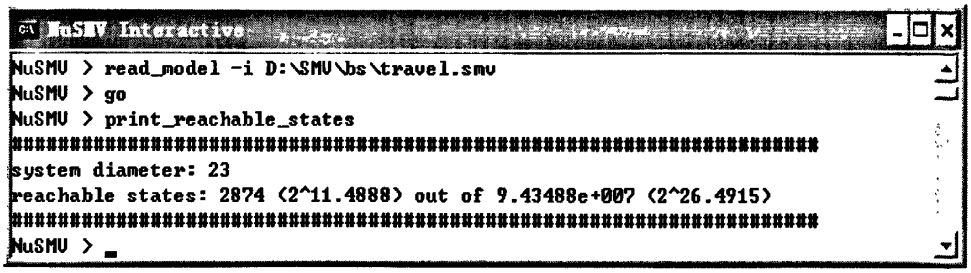

图 5.4 旅行安排模型 FSM 状态数

图 5.5 的模型检测结果说明使用 WS-BA 协议的旅行安排场景满足安全性和活性的要求。这一方面说明了旅行安排系统能够满足放松的事务特性，系统运行能够得到一致性的结果；同时也说明了 WS-BA 协议本身是可靠的。

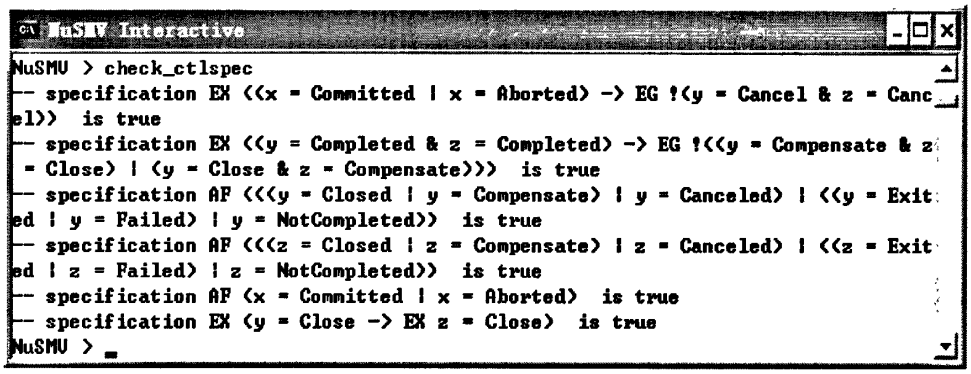

图 5.5 旅行安排模型检测结果

5.3.3 进一步思考

模型检测的一大特点是在系统不满足性质的情况下能够给出反例。反例的产生使得定位系统设计缺陷变得容易。下面将通过一个实例来说明如何分析反例。

WS-BA 放宽了 ACID 事务特性，并不满足严格的原子性。旅行安排场景中的原子性可以定义为：参与者一旦提交就不能撤消。飞机订票 Web 服务的原子性可以使用 CTL 描述为：

$$AG\ (y = Completed \rightarrow !(EX\ y = Compensate))$$

对以上性质的模型检测结果如图 5.6 所示，具体的反例见附录 3. 通过分析发现，在飞机订票 Web 服务向协调器发送 Completed 消息后，宾馆预订 Web 服务向协调器发送 Failed 消息表明任务失败，那么协调器就要向飞机订票 Web 服务发送 Compensate 消息，让其撤消已完成的任务，这表明旅行安排场景不满足严格的原子性。

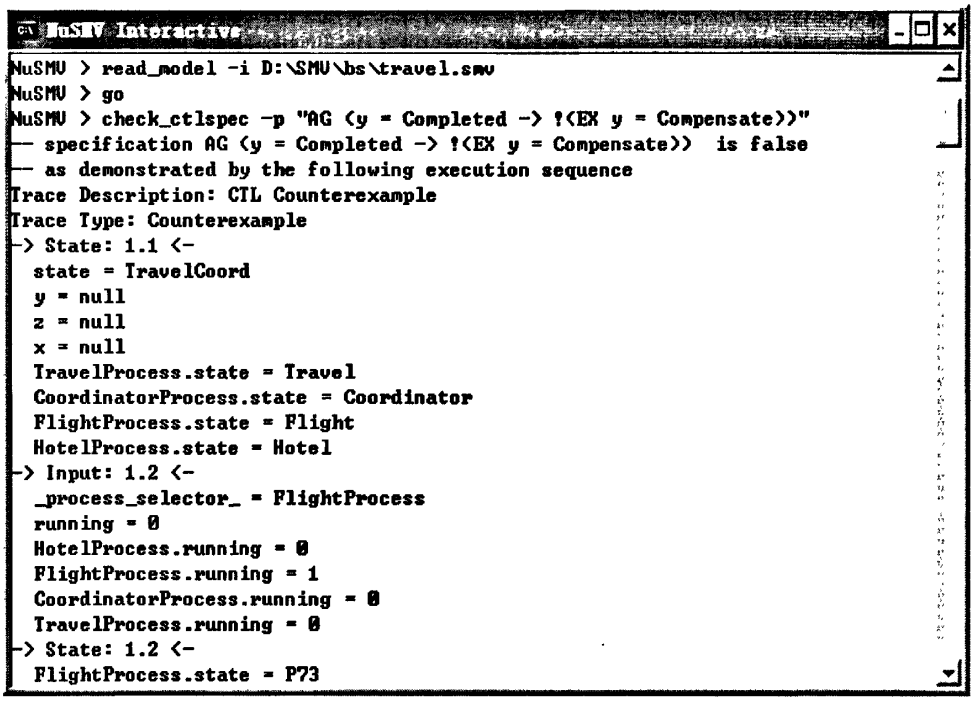

图 5.6 旅行安排场景原子性检测结果

本文中，反例是根据 SMV 代码给出的，因此要对照 PiCal2NuSMV 生成的 Pi-演算标识轨迹文件进行分析。在本文以后的工作中，将试图把这种分析工作用工具自动完成，给出针对 Pi-演算进程的反例代码，而不是基于 SMV 程序的，使得对反例的分析工作更加方便。

5.4 本章小结

本章使用模型检测工具 NuSMV2 对 WS-TX 协议的应用场景进行了模型检测。首先分析了 WS-TX 协议应用场景应具有的性质：安全性和活性。安全性验证从主观上验证应用场景是否满足事务特性，活性验证从客观上验证协议本身是否可靠。其次对 WS-AT 和 WS-BA 协议应用场景的安全性和活性分别用 CTL 进行了表述。然后使用 NuSMV2 验证了 WS-TX 应用场景是否满足安全性和活性，并对检测结果进行了分析。最后对 NuSMV2 产生的反例分析方法进行了思考。

第六章 总结与展望

6.1 论文工作总结

BPEL 2.0 规范中说到："BPEL 所描述的长事务概念发生在单个业务流程实例中，并且业务参与者之间的分布式协调一致性的实现是超出 BPEL 范围的问题，将来可以结合 WS-Transaction 规范解决这个问题。"[8]这里的 WS-Transaction 规范就是指 WS-TX 协议。WS-TX 协议作为解决分布式业务事务的标准受到了学术界和产业界的青睐。目前有很多学者使用形式化方法对 WS-TX 协议本身的正确性进行了研究，但缺少对 WS-TX 协议实际应用场景中是否满足事务特性的研究。基于此，本文提出了对 WS-TX 协议应用场景进行形式化验证分析的框架：使用 Pi-演算对 WS-TX 协议的应用场景进行建模，将模型转化为 SMV 程序代码，并使用模型检测工具 NuSMV2 对 CTL 表述的模型的安全性和活性进行检测和分析。模型检测的结果表明 WS-TX 协议应用场景满足安全性和活性。具体研究内容和研究成果如下：

（1）给出了 WS-TX 协议应用场景的 Pi-演算建模方法。首先对 WS-TX 协议应用场景中元素与 Pi-演算元素的映射关系进行了研究，并在此基础了提出了 WS-TX 应用场景的建模规则。然后给出应用了 WS-AT 的银行转帐场景和应用了 WS-BA 的旅行安排场景，使用建模方法对这两个场景进行 Pi-演算建模，验证了建模方法的实用性。

（2）给出了从 Pi-演算模型到 SMV 程序代码的转换方法。首先分析了现有转换方法通用性不够和状态爆炸的缺点，然后根据 Pi-演算属于 LTS 而 SMV 程序属于 KS 的特点，从 LTS 到 KS 的转换关系出发寻找 Pi-演算到 SMV 程序代码的转换方法，提出了如何使用 SMV 代码表述 Pi-演算进程表达式的七条规则。

（3）设计并实现了 Pi-演算表达式到 SMV 程序代码的自动转换工具 PiCal2NuSMV. PiCal2lNuSMV 共分为 Pi-演算文本解析器、转换适配器和 SMV 程序产生器三大组件。其中转换适配器为工具的核心，实现了 SMV 代码表述 Pi-演算进程表达式的七条规则，完成了 Pi-演算模型到 SMV 程序的转换。

（4）验证了 WS-TX 协议应用场景的安全性和活性。首先对应用了 WS-AT 的银行转帐场景和应用了 WS-BA 的旅行安排场景的安全性和活性进行了定义。安全性指协议应用场景的事务特性，而活性指协议本身的可靠性。然后使用 CTL 对这些性质进行了描述。最后使用 PiCal2NuSMV 工具将两个应用场景的协调过程转化为 SMV 程序代码，使用 NuSMV2 对它们分别进行了模型检测和分析。验证结果表明 WS-TX 协议应用场景满足安全性和活性。

6.2 进一步的工作

在本文的基础上，还可以开展以下几方面的进一步的研究：

（1）进一步完善 Pi-演算模型到 SMV 程序代码的转换方法。由于 WS-TX 协议具体应用中协调的参与者是固定的，不需要传递通道，因此本文中的 WS-TX 协议应用场景的建模规则并不能传递通道，在此基础上提出的 Pi-演算模型到 SMV 程序的转换方法也缺乏对含传递通道进程的考虑。而传递通道保证了 Pi-演算对并发进程间通信的可移动性的描述能力，考虑将没有任何建模方法限制的 Pi-演算模型到 SMV 程序代码的转换方法是本文未来工作之一。

（2）进一步完善 PiCal2NuSMV 工具，使其支持反例分析功能。使用 NuSMV2 进行模型检测得到的反例是基于 SMV 程序代码生成的，将反例映射到 Pi-演算模型将有利于对系统设计存在问题的分析。事实上，本文在实现 PiCal2NuSMV 工具时已经为反例分析做了考虑，PiCal2NuSMV 工作时记录了转换的重要痕迹，这是分析反例的基础和前提。

（3）将 PiCal2NuSMV 和 NuSMV2 整合在一起，实现一个对 Pi-演算模型进行验证分析的集成化的工具。

参考文献

[1] Greenfield, P., et al.. Consistency for web services applications. in Proceedings of the 31st international conference on Very large data bases. 2005, VLDB Endowment: Trondheim, Norway.

[2] Gray, J. and A. Reuter. Transaction processing : Concepts and techniques. 1993: Morgan Kaufmann.

[3] 岳昆, 王晓玲, 周傲英. Web 服务核心支撑技术：研究综述. 软件学报, 2004, 15(3): 428-442.

[4] Su, J., et al.. Towards a Theory of Web Service Choreographies. in Web Services and Formal Methods. 2008: 1-16.

[5] Bruni, R., H.a. Melgratti, and U. Montanari. Theoritical foundations for compensation in flow composition languages. in 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2005, ACM: Long Beach, California, USA. p. 209 - 220.

[6] 官荷卿. Web 服务事务的研究综述. 计算机科学, 2005. 32(5): 13-16.

[7] 龚松杰. Web 服务事务处理模型研究. 计算机工程, 2007. 33(10): 283-285.

[8] OASIS. Web Services Business Process Execution Language Version 2.0. 2007; Available from: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[9] Hagen, C. and G. Alonso. Exception handling in workflow management systems. Software Engineering, IEEE Transactions on, 2000, 26(10): 943-958.

[10] OASIS. WS-TX 1.2. 2009; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[11] ALONSO, R., H. GARCIA-MOLINA, and K. SALEM. Concurrency control and recovery for global procedures in federated database systems. in Q. Bull. IEEE-CS TC Data Eng. I0, 3 (Sept.). 1987: 5-11.

[12] Elmagarmid, A.K., et al.. A Multidatabase Transaction Model for InterBase. in Proceedings of the 16th International Conference on Very Large Data Bases. 1990, Morgan Kaufmann Publishers Inc.

[13] Aidong, Z.. Global Scheduling for Flexible Transactions in Heterogeneous Distributed Database Systems. IEEE Transactions on Knowledge and Data Engineering, 2001, 13: 439-450.

[14] 唐飞龙, 李明禄, 曹健, 一个 Web 服务事务处理模型: 结构、算法和事务补偿. 电子学报, 2003, 21(12A): 2074-2078.

[15] 任怡, 等. 一种基于 QoS 的事务工作流并发调度算法. 电子学报, 2007, 35(4): 621-628.

[16] Huang, T., X. Ding, and J. Wei. An application-semantics-based relaxed transaction model for internetware. Science in China Series F: Information Sciences, 2006. 49(6): 774-791.

[17] Bocchi, L., C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. in FMOODS 2003. 2003, Springer Berlin / Heidelberg: Paris, France. p. 124-138.

[18] Butler, M. and C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. in Coordination Models and Languages: 6th International Conference. 2004, Springer Berlin / Heidelberg: Pisa, Italy. p. 87-104.

[19] OASIS. Web Services Composite Application Framework (WS-CAF). 2007; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf.

[20] OASIS. Business Transaction Protocol (BTP) 1.0. 2002; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction.

[21] W3C Group. Web Services Description Language (WSDL) 1.1. 2001; Available from: http://www.w3.org/TR/wsdl.

[22] OASIS. Web Services Addressing (WS-Addressing). 2004; Available from: http://xml.coverpages.org/ws-Addressing.html.

[23] OASIS. Web Services Security(WS-Security) 1.1. 2006; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[24] OASIS. Web Services Policy (WS-Policy) 1.5. 2007; Available from: http://xml.coverpages.org/ni2003-06-04-a.html.

[25] Apache Software Foundation. Apache Kandula. Available from: http://ws.apache.org/kandula/.

[26] Sun. Web Services Interoperability Technologies (WSIT). Available from: https://wsit.dev.java.net/.

[27] Berger, M. and K. Honda. The Two-Phase Commitment Protocol in an Extended Pi-Calculus. Electronic Notes in Theoretical Computer Science, 2003, 39(1): 21-46.

[28] 戚正伟, 尤晋元. 基于细胞膜演算的 Web 服务事务处理形式化描述与验证. 计算机学报, 2006, 29(7): p. 1137-1141.

[29] Curbera, F., et al.. The next step in Web services. Commun. ACM, 2003, 46(10): 29-34.

[30] Zaihan, Y. and L. Chengfei. Implementing a Flexible Compensation Mechanism for Business Processes in Web Service Environment. in Web Services, 2006, ICWS '06. International Conference on. 2006.

[31] Schfer, M., P. Dolog, and W. Nejdl. An environment for flexible advanced compensations of Web service transactions. ACM Trans. Web, 2008, 2(2): 1-36.

[32] W3C Group. Simple Object Access Protocol(SOAP) 1.2. 2007; Available from: http://www.w3.org/TR/SOAP.

[33] OASIS. Universal Description, Discovery and Integration(UDDI) 3.0. 2004; Available from: http://uddi.org/pubs/uddi_v3.htm.

[34] Zhao, X., H. Yang, and Z. Qiu. Towards the formal model and verification of web service choreography description language. in WS-FM 2006. 2006.

[35] 萨师煊, 王珊. 数据库系统概论（第三版）. 2000, 北京: 高等教育出版社.

[36] 王勇, 张煜, 尹瑞, Web 服务组合中商业事务处理的研究. 小型微型计算机系统, 2006, 27(01): 121-125

[37] Zhang, Y. and X. Jia. Transaction Processing, in Wiley's Encyclopedia of Electrical and Electronics Engineering. Webster, 1999, 22: 298-311.

[38] Zhang, Y., et al.. On Interactions between Co-existing Traditional and Cooperative Transactions. International Journal of Cooperative Information Systems, 1999, 8(2).

[39] Limthanmaphon, B. and Y. Zhang. Web service composition transaction management. in Proceedings of the 15th Australasian database conference - Volume 27. 2004, Australian Computer Society, Inc.: Dunedin, New Zealand.

[40] Moss, E.B.. Nested Transactions: An Approach to Reliable Distributed Computing. 1981, Massachusetts Institute of Technology.

[41] Weikum, G. and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. in Database transaction models for advanced applications, 1992, Morgan Kaufmann Publishers Inc. p. 515-553.

[42] Garcia-Molina, H. and K. Salem. SAGAS. in Proceedings of ACM SIGMOD Conference on Management of Data, 1987.

[43] Pu, C., G.E. Kaiser, and N.C. Hutchinson. Split-Transactions for Open-Ended Activities. in Proceedings of the 14th International Conference on Very Large Data Bases, 1988, Morgan Kaufmann Publishers Inc.

[44] Reuter, A. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. in Proceedings of the 3rd International Workshop on High Performance Transaction Systems, 1989.

[45] Dayal, U., M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. in Proceedings of the 17th International Conference on Very Large Data Bases, 1991, Morgan Kaufmann Publishers Inc.

[46] Mani, A. and A. Nagarajan. Understanding quality of service for Web services. 2002.

[47] 赵健, 袁敏, 黄志球. 基于 Pi-演算的 THP 协议形式化建模与验证. in NASAC2008: 沈阳, 2008, 计算机科学: 228-232.

[48] Yuan, M., et al.. Modeling and Verification of Automatic Multi-business Transactions. in 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), 2009: Boston, Massachusetts, USA, p. 274-279.

[49] Robin, M.. A calculus of mobile processes, parts. I and II. Information and Computation, 1992.

[50] 辜希武, 卢正鼎. 基于 Pi-演算的 BPEL4WS Web 服务组合形式化模型. 计算机科学, 2007, 34(3): 69-74.

[51] 廖军, 谭浩, 刘锦德. 基于 Pi-演算的 Web 服务组合的描述和验证. 计算机学报, 2005, 28(4): 635-643.

[52] Victor, B. and F. Moller. The mobility workbench — A tool for the π-Calculus. in Computer Aided Verification, 1994: 428-440.

[53] Ferrari, G-L., et al.. A model-checking verification environment for mobile processes. ACM Trans. Softw. Eng. Methodol., 2003, 12(4): 440-473.

[54] Liu, Y., S. Mller, and K. Xu. A static compliance-checking framework for business process models. IBM Syst. J., 2007, 46(2): 335-361.

[55] 许可, 王跃宣, 吴澄. 网格服务链模型的验证分析技术及应用. 中国科学, 2007, 37(4): 467-485.

[56] Lam, V.S.W. and J. Padget. Symbolic model checking of UML statechart diagrams with an integrated approach. in Engineering of Computer-Based Systems, 2004, Proceedings. 11th IEEE International Conference and Workshop on the. 2004.

[57] Cimatti, A., et al.. NuSMV2: An OpenSource Tool for Symbolic Model Checking. in Computer Aided Verification, 2002: 241-268.

[58] Cimatti, A., et al.. NUSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer (STTT), 2000, 2(4): 410-425.

[59] De Nicola, R. and F. Vaandrager. Action versus state based logics for transition systems. in Semantics of Systems of Concurrent Processes. 1990: 407-419.

[60] Parr, T.. ANTLR: ANother Tool for Language Recognition. Available from: http://www.antlr.org.

[61] Gray, J. and L. Lamport. Consensus on transaction commit. ACM Trans. Database Syst., 2006, 31(1): 133-160.

[62] 戚正伟. 细胞膜演算: 一种新的事务处理形式化方法研究, [博士学位论文]. 上海: 上海交通大学, 2005.

致 谢

论文完成之际也是我研究生学习接近尾声的时间，本文是在我的导师黄志球教授的悉心指导和殷切关怀下完成的，他给予了我大量有益的帮助，使得我的研究工作和论文有计划、有步骤地不断向深层次推进。黄老师渊博的知识、敏锐的思维、远见卓识的判断力使我倾倒，严谨的治学作风、正直的人品是我一生学习的榜样。在此谨向黄老师表示最衷心的感谢和最诚挚的敬意。

特别感谢实验室的袁敏博士，在研究过程中他给了我很大的帮助，从他身上我学到了严谨的做事态度以及做事的方法，跟他讨论课题时连饭都顾不上吃的情景使我终生难忘。

感谢南京航空航天大学 EM 软件研究开发中心的全体老师和同学，他们给我提供了温暖而良好的研究、学习氛围，这是一个融洽的、勇于进取的集体，给了我不断前进的力量。感谢吕樟权老师、沈国华老师、张定会老师给予我各方面的帮助。感谢李春、徐万江、刘亚萍、黄凤和徐丙凤同学，在三年共同的学习和生活中，我们互帮互助，共同进步，并克服了许多困难，是你们给予了我成长的动力和支持，在此再次表示感谢。感谢周航、朱小栋、祝义、肖芳雄、刘林源、于瑞强、范大娟、方元康、李婧、吴其展、赵剑、苏焕程、汪文颖、项高友、陈圣青、吴博、仲晶等师兄师姐们，在共同学习、生活的日子里，你们给了我很多的指导和帮助。从你们身上，我学到了很多非常有价值的东西。

感谢闫艳、洪宏、靳玲、刘通和吉鸣等师弟师妹们给实验室带来的欢声笑语以及良好的学习研究气氛，很怀念和你们在一起并肩战斗开发项目的日子。

感谢信息科学与技术学院计算机专业的所有老师，在我攻读硕士学位期间，他们给了我的各个方面的指导和帮助，令我终身难忘。

感谢所有帮助、关心过我的朋友们。

最后，将本文献给我的父亲、母亲，感谢他们对我长期以来的关怀、帮助和鼓励，他们对我无私的爱是我勇往直前的动力源泉。

在学期间的研究成果及发表的学术论文

攻读硕士学位期间发表（录用）论文情况

1. 李祥, 黄志球, 袁敏. 基于 Pi-Logic 约束的服务组合验证与分析. 2009 全国软件与应用学术会议, 计算机科学, 2009, 36(9): 10-15
2. 李祥, 李春, 徐万江. 面向对象软件度量数据 3D 可视化映射规则研究. 2008 南京地区通信年会, 2008
3. Min Yuan, Zhiqiu Huang, Jian Zhao, Xiang Li. Modeling and Verification of Automatic Multi-business Transactions. In The 21st International Conference on Software Engineering and Knowledge Engineering, 2009, SEKE 2009: 274-279

攻读硕士学位期间参加科研项目情况

1. 中石油测井公司数字岩心实验数据库系统。
2. 日本益盟医疗保险机关（医院）支援系统版本升级项目。

附录

附录 1 银行转帐 SMV 程序代码

```
MODULE main
VAR
    state : {P1, TransferCoord};
    x : {Rollback, Commit, Aborted, Committed, null};
    y : {Rollback, Prepare, Commit, Aborted, ReadOnly, Prepared, Committed, null};
    z : {Rollback, Prepare, Commit, Aborted, ReadOnly, Prepared, Committed, null};
    TransferProcess : process Transfer(x);
    CoordinatorProcess : process Coordinator(x, y, z);
    ComBankProcess : process ComBank(y);
    AgrBankProcess : process AgrBank(z);
ASSIGN
    init(state) := TransferCoord;
    next(state) := case
        state = TransferCoord : P1;
        1 : state;
    esac;
    init(x) := null;
    init(y) := null;
    init(z) := null;
SPEC EX ((x=Committed | x=Aborted) -> EG !(y=Rollback & z=Rollback))
SPEC EG !((y=Committed & z=Aborted) | (y=Aborted & z=Committed))
SPEC AF (y=Committed | y=Aborted | y=ReadOnly)
SPEC AF (z=Committed | z=Aborted | z=ReadOnly)
SPEC AF (x=Committed | x=Aborted)
SPEC EX (y=Committed -> EX z=Committed)

MODULE Transfer(x)
VAR
    state : {P2, P3, P5, Nil, P4, P6, Transfer};
ASSIGN
    init(state) := Transfer;
    next(state) := case
        state = Transfer : P2;
        state = P2 : {P3,P4};
        state = P3 : P5;
        state = P5 & x = Aborted : Nil;
        state = P4 : P6;
        state = P6 & x = Aborted : Nil;
        state = P6 & x = Committed : Nil;
        1 : state;
    esac;
    next(x) := case
        state = P3 : Rollback;
        state = P4 : Commit;
        1 : x;
    esac;
FAIRNESS
    running

MODULE Coordinator(x, y, z)
VAR
    state : {P7, P8, P10, P11, P12, P13, Nil, P9, P14, P15, P18, P19, P16, P20, P21, P17, P22, P23, P26, P27, P24, P28, P29, P25, P30, P31,
P32, P33, Coordinator};
ASSIGN
    init(state) := Coordinator;
    next(state) := case
        state = Coordinator : P7;
        state = P7 & x = Rollback : P8;
        state = P7 & x = Commit : P9;
        state = P8 : P10;
        state = P10 & y = Aborted : P11;
        state = P11 : P12;
        state = P12 & z = Aborted : P13;
        state = P13 : Nil;
        state = P9 : P14;
```

```
        state = P14 & y = Aborted : P15;
        state = P14 & y = ReadOnly : P16;
        state = P14 & y = Prepared : P17;
        state = P15 : P18;
        state = P18 & z = Aborted : P19;
        state = P19 : Nil;
        state = P16 : P20;
        state = P20 & z = Aborted : P21;
        state = P21 : Nil;
        state = P17 : P22;
        state = P22 & z = Aborted : P23;
        state = P22 & z = ReadOnly : P24;
        state = P22 & z = Prepared : P25;
        state = P23 : P26;
        state = P26 & y = Aborted : P27;
        state = P27 : Nil;
        state = P24 : P28;
        state = P28 & y = Aborted : P29;
        state = P29 : Nil;
        state = P25 : P30;
        state = P30 & y = Committed : P31;
        state = P31 : P32;
        state = P32 & z = Committed : P33;
        state = P33 : Nil;
        1 : state;
    esac;
    next(y) := case
        state = P8 : Rollback;
        state = P9 : Prepare;
        state = P23 : Rollback;
        state = P24 : Rollback;
        state = P25 : Commit;
        1 : y;
    esac;
    next(z) := case
        state = P11 : Rollback;
        state = P15 : Rollback;
        state = P16 : Rollback;
        state = P17 : Prepare;
        state = P31 : Commit;
        1 : z;
    esac;
    next(x) := case
        state = P13 : Aborted;
        state = P19 : Aborted;
        state = P21 : Aborted;
        state = P27 : Aborted;
        state = P29 : Aborted;
        state = P33 : Committed;
        1 : x;
    esac;
FAIRNESS
    running

MODULE ComBank(y)
VAR
    state : {P34, P35, Nil, P36, P37, P38, P39, P40, P41, P42, ComBank};
ASSIGN
    init(state) := ComBank;
    next(state) := case
        state = ComBank : P34;
        state = P34 & y = Rollback : P35;
        state = P34 & y = Prepare : P36;
        state = P35 : Nil;
        state = P36 : {P37,P38,P39};
        state = P37 : Nil;
        state = P38 : Nil;
        state = P39 : P40;
        state = P40 & y = Rollback : P41;
        state = P40 & y = Commit : P42;
        state = P41 : Nil;
        state = P42 : Nil;
        1 : state;
    esac;
    next(y) := case
```

```
        state = P35 : Aborted;
        state = P37 : Aborted;
        state = P38 : ReadOnly;
        state = P39 : Prepared;
        state = P41 : Aborted;
        state = P42 : Committed;
        1 : y;
    esac;
FAIRNESS
    running

MODULE AgrBank(z)
VAR
    state : {P43, P44, Nil, P45, P46, P47, P48, P49, P50, P51, AgrBank};
ASSIGN
    init(state) := AgrBank;
    next(state) := case
        state = AgrBank : P43;
        state = P43 & z = Rollback : P44;
        state = P43 & z = Prepare : P45;
        state = P44 : Nil;
        state = P45 : {P46,P47,P48};
        state = P46 : Nil;
        state = P47 : Nil;
        state = P48 : P49;
        state = P49 & z = Rollback : P50;
        state = P49 & z = Commit : P51;
        state = P50 : Nil;
        state = P51 : Nil;
        1 : state;
    esac;
    next(z) := case
        state = P44 : Aborted;
        state = P46 : Aborted;
        state = P47 : ReadOnly;
        state = P48 : Prepared;
        state = P50 : Aborted;
        state = P51 : Committed;
        1 : z;
    esac;
FAIRNESS
    running
```

附录 2 旅行安排 SMV 程序代码

```
MODULE main
VAR
    state : {P1, TravelCoord};
    y : {Exited, Failed, NotCompleted, Compensate, Close, Exit, Canceled, Fail, CannotComplete, Completed, Closed, Compensated, null};
    z : {Cancel, Exited, Failed, NotCompleted, Compensate, Close, Exit, Canceled, Fail, CannotComplete, Completed, Closed, Compensated,
null};
    x : {Aborted, Committed, null};
    TravelProcess : process Travel(x);
    CoordinatorProcess : process Coordinator(y, z, x);
    FlightProcess : process Flight(y);
    HotelProcess : process Hotel(z);
ASSIGN
    init(state) := TravelCoord;
    next(state) := case
        state = TravelCoord : P1;
        1 : state;
    esac;
    init(y) := null;
    init(z) := null;
    init(x) := null;
SPEC EX ((x=Committed | x=Aborted) -> EG !(y=Cancel & z=Cancel))
SPEC EX ((y=Completed & z=Completed) -> EG !((y=Compensate & z=Close) | (y=Close & z=Compensate)))
SPEC AF ((y=Closed | y=Compensate | y=Canceled) | (y=Exited | y=Failed | y=NotCompleted))
SPEC AF ((z=Closed | z=Compensate | z=Canceled) | (z=Exited | z=Failed | z=NotCompleted))
SPEC AF (x=Committed | x=Aborted)
SPEC EX (y=Close -> EX z=Close)

MODULE Travel(x)
VAR
```

```
    state : {P2, Nil, Travel};
ASSIGN
    init(state) := Travel;
    next(state) := case
        state = Travel : P2;
        state = P2 & x = Committed : Nil;
        state = P2 & x = Aborted : Nil;
        1 : state;
    esac;
FAIRNESS
    running

MODULE Coordinator(y, z, x)
VAR
    state : {P3, P4, P8, P9, P10, P15, Nil, P11, P12, P16, P13, P17, P14, P18, P19, P20, P21, P5, P22, P23, P24, P29, P25, P26, P30, P27, P31,
P28, P32, P33, P34, P35, P6, P36, P37, P38, P43, P39, P40, P44, P41, P45, P42, P46, P47, P48, P49, P7, P50, P54, P55, P56, P57, P58, P51,
P59, P60, P61, P62, P63, P52, P64, P65, P66, P67, P68, P53, P69, P70, P71, P72, Coordinator};
ASSIGN
    init(state) := Coordinator;
    next(state) := case
        state = Coordinator : P3;
        state = P3 & y = Exit : P4;
        state = P3 & y = Fail : P5;
        state = P3 & y = CannotComplete : P6;
        state = P3 & y = Completed : P7;
        state = P4 : P8;
        state = P8 : P9;
        state = P9 & z = Exit : P10;
        state = P9 & z = Canceled : P11;
        state = P9 & z = Fail : P12;
        state = P9 & z = CannotComplete : P13;
        state = P9 & z = Completed : P14;
        state = P10 : P15;
        state = P15 : Nil;
        state = P11 : Nil;
        state = P12 : P16;
        state = P16 : Nil;
        state = P13 : P17;
        state = P17 : Nil;
        state = P14 : P18;
        state = P18 & z = Compensated : P19;
        state = P18 & z = Fail : P20;
        state = P19 : Nil;
        state = P20 : P21;
        state = P21 : Nil;
        state = P5 : P22;
        state = P22 : P23;
        state = P23 & z = Exit : P24;
        state = P23 & z = Canceled : P25;
        state = P23 & z = Fail : P26;
        state = P23 & z = CannotComplete : P27;
        state = P23 & z = Completed : P28;
        state = P24 : P29;
        state = P29 : Nil;
        state = P25 : Nil;
        state = P26 : P30;
        state = P30 : Nil;
        state = P27 : P31;
        state = P31 : Nil;
        state = P28 : P32;
        state = P32 & z = Compensated : P33;
        state = P32 & z = Fail : P34;
        state = P33 : Nil;
        state = P34 : P35;
        state = P35 : Nil;
        state = P6 : P36;
        state = P36 : P37;
        state = P37 & z = Exit : P38;
        state = P37 & z = Canceled : P39;
        state = P37 & z = Fail : P40;
        state = P37 & z = CannotComplete : P41;
        state = P37 & z = Completed : P42;
        state = P38 : P43;
        state = P43 : Nil;
        state = P39 : Nil;
```

```
        state = P40 : P44;
        state = P44 : Nil;
        state = P41 : P45;
        state = P45 : Nil;
        state = P42 : P46;
        state = P46 & z = Compensated : P47;
        state = P46 & z = Fail : P48;
        state = P47 : Nil;
        state = P48 : P49;
        state = P49 : Nil;
        state = P7 & z = Exit : P50;
        state = P7 & z = Fail : P51;
        state = P7 & z = CannotComplete : P52;
        state = P7 & z = Completed : P53;
        state = P50 : P54;
        state = P54 : P55;
        state = P55 & y = Compensated : P56;
        state = P55 & y = Fail : P57;
        state = P56 : Nil;
        state = P57 : P58;
        state = P58 : Nil;
        state = P51 : P59;
        state = P59 : P60;
        state = P60 & y = Compensated : P61;
        state = P60 & y = Fail : P62;
        state = P61 : Nil;
        state = P62 : P63;
        state = P63 : Nil;
        state = P52 : P64;
        state = P64 : P65;
        state = P65 & y = Compensated : P66;
        state = P65 & y = Fail : P67;
        state = P66 : Nil;
        state = P67 : P68;
        state = P68 : Nil;
        state = P53 : P69;
        state = P69 & y = Closed : P70;
        state = P70 : P71;
        state = P71 & z = Closed : P72;
        state = P72 : Nil;
        1 : state;
    esac;
    next(y) := case
        state = P4 : Exited;
        state = P5 : Failed;
        state = P6 : NotCompleted;
        state = P54 : Compensate;
        state = P57 : Failed;
        state = P59 : Compensate;
        state = P62 : Failed;
        state = P64 : Compensate;
        state = P67 : Failed;
        state = P53 : Close;
        1 : y;
    esac;
    next(z) := case
        state = P8 : Cancel;
        state = P10 : Exited;
        state = P12 : Failed;
        state = P13 : NotCompleted;
        state = P14 : Compensate;
        state = P20 : Failed;
        state = P22 : Cancel;
        state = P24 : Exited;
        state = P26 : Failed;
        state = P27 : NotCompleted;
        state = P28 : Compensate;
        state = P34 : Failed;
        state = P36 : Cancel;
        state = P38 : Exited;
        state = P40 : Failed;
        state = P41 : NotCompleted;
        state = P42 : Compensate;
        state = P48 : Failed;
        state = P50 : Exited;
```

```
        state = P51 : Failed;
        state = P52 : NotCompleted;
        state = P70 : Close;
        1 : z;
    esac;
    next(x) := case
        state = P15 : Aborted;
        state = P11 : Aborted;
        state = P16 : Aborted;
        state = P17 : Aborted;
        state = P19 : Aborted;
        state = P21 : Aborted;
        state = P29 : Aborted;
        state = P25 : Aborted;
        state = P30 : Aborted;
        state = P31 : Aborted;
        state = P33 : Aborted;
        state = P35 : Aborted;
        state = P43 : Aborted;
        state = P39 : Aborted;
        state = P44 : Aborted;
        state = P45 : Aborted;
        state = P47 : Aborted;
        state = P49 : Aborted;
        state = P56 : Aborted;
        state = P58 : Aborted;
        state = P61 : Aborted;
        state = P63 : Aborted;
        state = P66 : Aborted;
        state = P68 : Aborted;
        state = P72 : Committed;
        1 : x;
    esac;
FAIRNESS
    running

MODULE Flight(y)
VAR
    state : {P73, P74, P78, Nil, P79, P75, P80, P81, P76, P82, P83, P77, P84, P85, P86, P88, P89, P90, P87, Flight};
ASSIGN
    init(state) := Flight;
    next(state) := case
        state = Flight : P73;
        state = P73 : {P74,P75,P76,P77};
        state = P74 : P78;
        state = P78 & y = Exited : Nil;
        state = P78 & y = Cancel : P79;
        state = P79 : Nil;
        state = P75 : P80;
        state = P80 & y = Failed : Nil;
        state = P80 & y = Cancel : P81;
        state = P81 : Nil;
        state = P76 : P82;
        state = P82 & y = NotCompleted : Nil;
        state = P82 & y = Cancel : P83;
        state = P83 : Nil;
        state = P77 : P84;
        state = P84 & y = Close : P85;
        state = P84 & y = Compensate : P86;
        state = P84 & y = Cancel : P87;
        state = P85 : Nil;
        state = P86 : {P88,P89};
        state = P88 : Nil;
        state = P89 : P90;
        state = P90 & y = Failed : Nil;
        state = P87 : Nil;
        1 : state;
    esac;
    next(y) := case
        state = P74 : Exit;
        state = P79 : Canceled;
        state = P75 : Fail;
        state = P81 : Canceled;
        state = P76 : CannotComplete;
        state = P83 : Canceled;
```

```
        state = P77 : Completed;
        state = P85 : Closed;
        state = P88 : Compensated;
        state = P89 : Fail;
        state = P87 : Canceled;
        1 : y;
    esac;
FAIRNESS
    running

MODULE Hotel(z)
VAR
    state : {P91, P92, P96, Nil, P97, P93, P98, P99, P94, P100, P101, P95, P102, P103, P104, P106, P107, P108, P105, Hotel};
ASSIGN
    init(state) := Hotel;
    next(state) := case
        state = Hotel : P91;
        state = P91 : {P92,P93,P94,P95};
        state = P92 : P96;
        state = P96 & z = Exited : Nil;
        state = P96 & z = Cancel : P97;
        state = P97 : Nil;
        state = P93 : P98;
        state = P98 & z = Failed : Nil;
        state = P98 & z = Cancel : P99;
        state = P99 : Nil;
        state = P94 : P100;
        state = P100 & z = NotCompleted : Nil;
        state = P100 & z = Cancel : P101;
        state = P101 : Nil;
        state = P95 : P102;
        state = P102 & z = Close : P103;
        state = P102 & z = Compensate : P104;
        state = P102 & z = Cancel : P105;
        state = P103 : Nil;
        state = P104 : {P106,P107};
        state = P106 : Nil;
        state = P107 : P108;
        state = P108 & z = Failed : Nil;
        state = P105 : Nil;
        1 : state;
    esac;
    next(z) := case
        state = P92 : Exit;
        state = P97 : Canceled;
        state = P93 : Fail;
        state = P99 : Canceled;
        state = P94 : CannotComplete;
        state = P101 : Canceled;
        state = P95 : Completed;
        state = P103 : Closed;
        state = P106 : Compensated;
        state = P107 : Fail;
        state = P105 : Canceled;
        1 : z;
    esac;
FAIRNESS
    running
```

附录 3 飞机订票 Web 服务的原子性模型检测反例

```
-- specification AG (y = Completed -> !(EX y = Compensate))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  state = TravelCoord
  y = null
  z = null
  x = null
  TravelProcess.state = Travel
  CoordinatorProcess.state = Coordinator
  FlightProcess.state = Flight
  HotelProcess.state = Hotel
```

-> Input: 2.2 <-
_process_selector_ = FlightProcess
running = 0
HotelProcess.running = 0
FlightProcess.running = 1
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.2 <-
FlightProcess.state = P73
-> Input: 2.3 <-
_process_selector_ = FlightProcess
running = 0
HotelProcess.running = 0
FlightProcess.running = 1
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.3 <-
FlightProcess.state = P77
-> Input: 2.4 <-
_process_selector_ = FlightProcess
running = 0
HotelProcess.running = 0
FlightProcess.running = 1
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.4 <-
y = Completed
FlightProcess.state = P84
-> Input: 2.5 <-
_process_selector_ = HotelProcess
running = 0
HotelProcess.running = 1
FlightProcess.running = 0
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.5 <-
HotelProcess.state = P91
-> Input: 2.6 <-
_process_selector_ = HotelProcess
running = 0
HotelProcess.running = 1
FlightProcess.running = 0
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.6 <-
HotelProcess.state = P93
-> Input: 2.7 <-
_process_selector_ = HotelProcess
running = 0
HotelProcess.running = 1
FlightProcess.running = 0
CoordinatorProcess.running = 0
TravelProcess.running = 0
-> State: 2.7 <-
z = Fail
HotelProcess.state = P98
-> Input: 2.8 <-
_process_selector_ = CoordinatorProcess
running = 0
HotelProcess.running = 0
FlightProcess.running = 0
CoordinatorProcess.running = 1
TravelProcess.running = 0
-> State: 2.8 <-
CoordinatorProcess.state = P3
-> Input: 2.9 <-
_process_selector_ = CoordinatorProcess
running = 0
HotelProcess.running = 0
FlightProcess.running = 0
CoordinatorProcess.running = 1
TravelProcess.running = 0
-> State: 2.9 <-
CoordinatorProcess.state = P7
-> Input: 2.10 <-

```
  _process_selector_ = CoordinatorProcess
  running = 0
  HotelProcess.running = 0
  FlightProcess.running = 0
  CoordinatorProcess.running = 1
  TravelProcess.running = 0
-> State: 2.10 <-
  CoordinatorProcess.state = P51
-> Input: 2.11 <-
  _process_selector_ = CoordinatorProcess
  running = 0
  HotelProcess.running = 0
  FlightProcess.running = 0
  CoordinatorProcess.running = 1
  TravelProcess.running = 0
-> State: 2.11 <-
  z = Failed
  CoordinatorProcess.state = P59
-> Input: 2.12 <-
  _process_selector_ = CoordinatorProcess
  running = 0
  HotelProcess.running = 0
  FlightProcess.running = 0
  CoordinatorProcess.running = 1
  TravelProcess.running = 0
-> State: 2.12 <-
  y = Compensate
  CoordinatorProcess.state = P60
```