内部★　2　年

网格服务流的状态π演算形式化验证技术研究与应用

（申请清华大学工学博士学位论文）

培 养 单 位：自动化系

学　　科：控制科学与工程

研 究 生：许　可

指 导 教 师：吴　澄　　教　授

二〇〇七年四月

网格服务流的状态π演算形式化验证技术研究与应用

许 可

Research on the Theory and Application of the State π Calculus based Formal Verification for Grid Service Flows

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Engineering

by

Xu Ke

(Control Science and Engineering)

Dissertation Supervisor： Professor Wu Cheng

April, 2007

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后遵守此规定）

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘 要

网格服务流是实现复杂网格应用中大规模异构资源协作与共享的重要手段。如何确保其设计和实现上的正确性是系统科学和计算机科学领域的前沿性研究方向，具有重要的理论意义和实用价值。本文通过提出一种称为状态π演算的状态/事件混合形式化工具，采用与模型验证（Model Checking）相结合的手段，系统研究了以状态π演算为基础的网格服务流形式化建模、验证及验证性能优化技术。部分研究成果已在 LIGO 数据网格项目和银行法律法规验证中得到应用。

结合网格中 Web 服务资源框架对有状态资源的管理特点，本文首先提出了一种状态π演算的新形式化工具并分析了其语义和性质，它实现了对系统状态的灵活抽象与管理，有效简化了网格服务流及其业务逻辑的建模和验证。基于状态π演算，本文实现了网格服务及其选择、BPEL4WS、DAGMan 规范和网格服务流中并发与管道模式的形式化语义。这不仅为部分晦涩复杂的网格服务流规范提供了重要的形式化基础，也验证了状态π演算自身的表达能力。

在此形式化语义的基础上，本文从网格服务流的结构验证、规范语义约束验证、业务逻辑验证及其一致性检验的四个方面和静态/动态两个层面对其形式化验证技术进行了研究。通过提出状态π演算的强/弱状态断言和灵活复用模型验证的方法为以上问题提供了系统的解决方法。通过应用实例发现，该方法实现了对状态π演算的状态/事件混合推演和模型验证支持，且有减少重复验证和验证方法独立于特定模型验证技术的优点。

进一步，本文还从服务流验证分解和错误过程模式的两种思路，研究了对网格服务流形式化验证的性能改进。一方面，通过将网格服务流分解为一系列串行域和并发支，提出了基于模块化验证的验证分解策略；另一方面，通过分析网格服务流中可能存在的错误过程模式及其特点，提出了基于错误过程模式搜索向导的快速证伪方法。通过不同复杂度的 LIGO 引力波数据分析服务流实例和数值比较结果，验证了以上两方法在验证效率和内存占用上均有明显提高。

最后介绍了本文工作的系统化集成,即 GridPiAnalyzer 原型系统的模块和应用。它目前已提供了对 Condor DAGMan、BPEL 1.1 规范和 IBM WBI^{TM} 的支持。

关键词：网格；网格服务流；状态π演算；形式化验证；性能改进

Abstract

Grid service flow is an important paradigm for realizing complex heterogeneous resource collaboration and sharing in various grid applications. Correctness verification in the design and implementation of grid service flows is an active research topic in the domain of system science and computer science which enjoys both great theoretical and practical values. Based on realistic applications like LIGO data grid, this dissertation has systematically investigated the formal modeling, verification and its performance issues for grid service flows by proposing a calculus called the state π calculus and the study of its formal verification support.

Following the new feature of stateful resource management introduced in the Web Service Resource Framework, first a new state π calculus is proposed to further strengthen its capability in managing the life-cycle of system states. The proposed calculus not only enables the flexible abstraction and management of historical system events, but also facilitates the modeling and verification of grid service flows. Moreover, based on the state π calculus, the formal semantics of grid service execution and selection, the specification of BPEL4WS, DAGMan and the parallel / pipeline patterns in grid service flows are also rigidly captured, which at the same time has verified the full expressiveness of our state π calculus.

Based on the formal semantics of grid service flows with state π calculus, the dissertation fully studies from both the static and dynamic aspects of its formal verification issues including its structural correctness, specification satisfiability, business logic satisfiability and consistency. Systematic solution is provided for the above issues by the proposal of strong/weak state assertion for our state π calculus with its model checking support. By the gravitational wave data analysis application from LIGO data grid, it is shown the proposed solution enjoys the advantages of: (1) supports the state / event hybrid analysis and model checking of state π calculus models; (2) reduce unnecessary verifications to save the verification cost; (3) be independent of specific model checking engine and thus enjoys a wider choice of new

formal verification techniques.

Furthermore, the improvement of verification performance is also investigated by following the two ideas of grid service flow decomposition and process bug patterns respectively. On the one hand, by decomposing a grid service flow into a set of sequential standard regions with parallel branches, the corresponding verification strategy is also decomposed following the idea of modular verification. On the other hand, by identifying the potential process bug patterns in the structure of grid service flows, the idea of guided search is combined into the formal verification of grid service flow to efficiently falsify its correctness. With the performance evaluation on verifying multiple gravitational wave data analysis applications with different complexity from LIGO, it is illustrated that the proposed approaches can effectively reduce both the required CPU time and memory cost compared to using various traditional verification approaches alone.

Finally, the framework and application background of GridPiAnalyzer, an automatic tool support for the formal verification of grid service flows based on this dissertation, is introduced. It currently provides the support for Condor DAGMan, BPEL4WS 1.1 specification and IBM's Websphere Business Integrator ^{TM} model.

Key words: Grid; Grid Service Flow; State π Calculus; Formal Verification; Performance Improvement

目 录

第 1 章 引言 1
1.1 网格技术的起源和发展 1
1.2 网格服务与网格服务流 1
1.3 网格服务流的正确性保障与形式化验证 3
1.3.1 网格服务流正确性保障的重要性 3
1.3.2 研究挑战及原因 5
1.3.3 本文的网格服务流形式化验证范畴和待解决问题 7
1.4 网格服务流形式化验证技术的研究现状 8
1.4.1 网格服务流的建模 8
1.4.2 形式化建模与验证技术及其必要性 9
1.4.3 π演算的应用与优势 12
1.4.4 Web 服务资源框架与状态/事件混合形式化方法 15
1.4.5 本文方法的不同点 17
1.5 本文研究的主要内容和贡献 19
1.5.1 研究什么和不研究什么 19
1.5.2 本文组织与主要贡献 19
第 2 章 状态π演算及其类型约束的提出与分析 22
2.1 本章引论 22
2.2 状态π演算中的状态模型与状态操作 23
2.2.1 状态模型的静态语义 23
2.2.2 状态π演算的语法和状态操作 24
2.3 状态标号迁移系统、同余和扩展操作语义 27
2.4 状态互模拟关系 29
2.5 状态π演算的类型约束 31
2.5.1 数据类型结构的静态语义 32
2.5.2 带类型约束的状态π演算操作语义 32

2.6 小结 37
第 3 章 基于状态π演算的网格服务流形式化语义 39
3.1 本章引论 39
3.2 服务执行与基本活动的状态π演算形式化 40
3.2.1 服务执行的状态π演算语义 41
3.2.2 服务执行的多实例化语义 43
3.2.3 基本活动的状态π演算语义 44
3.2.4 服务选择的状态π演算语义 45
3.3 DAGMan 控制流结构的状态π演算形式化 47
3.3.1 DAGMan 中的 *NoPostFail* 47
3.3.2 顺序结构的状态π演算语义 48
3.3.3 同步并发结构的状态π演算语义 48
3.3.4 重试结构的状态π演算语义 49
3.4 BPEL4WS 服务流的状态π演算形式化 50
3.4.1 顺序结构的状态π演算语义 50
3.4.2 流结构的状态π演算语义 50
3.4.3 循环结构的状态π演算语义 51
3.4.4 分支结构的状态π演算语义 51
3.4.5 选择结构的状态π演算语义 51
3.4.6 同步联接的状态π演算语义 52
3.4.7 范围限定的状态π演算语义 53
3.4.8 异常处理与补偿的状态π演算语义 54
3.4.9 全局终止的状态π演算语义 55
3.5 服务流的形式化示例 55
3.6 网格服务流中的并发与管道模式 57
3.6.1 静态并发执行模式及其状态π演算语义 58
3.6.2 动态自适应并发执行模式及其状态π演算语义 59
3.6.3 尽力执行的管道执行模式及其状态π演算语义 60
3.6.4 阻塞执行的管道执行模式及其状态π演算语义 61
3.6.5 缓冲执行的管道执行模式及其状态π演算语义 61
3.6.6 多实例的管道执行模式及其状态π演算语义 62

3.6.7 流方式的管道执行模式及其状态π演算语义 63
3.7 基于状态π演算的形式化结果与优势讨论 64
3.8 小结 66
第四章 网格服务流的状态π演算形式化验证 67
4.1 本章引论 67
4.2 从状态π演算到状态标号迁移系统 68
4.3 网格服务流的结构与规范语义约束验证 72
4.4 网格服务流的业务逻辑验证及应用实例 78
4.4.1 业务逻辑验证的服务流实例 78
4.4.2 静态业务逻辑验证 79
4.4.3 动态业务逻辑验证 84
4.5 状态π演算形式化验证方法的优点与特点 88
4.6 业务逻辑的一致性验证 89
4.6.1 冲突业务逻辑与冗余业务逻辑 89
4.6.2 业务逻辑一致性验证方法与实现 90
4.6.3 应用实例与讨论 93
4.7 小结 97
第 5 章 网格服务流形式化验证方法的性能改进 98
5.1 本章引论 98
5.2 基于域分析的验证分解方法 99
5.2.1 网格服务流的建模约定 99
5.2.2 基于标准域分析的服务流分解 101
5.2.3 基于标准域分析的验证分解 104
5.2.4 基于松弛域分析的服务流分解 108
5.2.5 基于域分析的验证分解实现与应用结果讨论 110
5.3 基于错误过程模式的验证向导方法 118
5.3.1 错误过程模式的提出 119
5.3.2 基本错误过程模式 120
5.3.3 高级分支与同步错误模式 124
5.3.4 结构错误模式 126

5.3.5 多实例错误模式 127
5.3.6 基于状态的错误模式 129
5.3.7 取消错误模式 130
5.3.8 带向导的快速错误过程模式验证 131
5.3.9 应用与结果讨论 135
5.4 小结 139
第 6 章 网格服务流验证系统(GridPiAnalyzer)及其实现 141
6.1 本章引论 141
6.2 GridPiAnalyzer 的系统框架与模块 142
6.3 状态π演算模型在 GridPiAnalyzer 中的实现 144
6.4 状态标号迁移系统及验证结果的封装 146
6.5 GridPiAnalyzer 中的 BPSL 可视化业务逻辑描述 148
6.6 GridPiAnalyzer 的已有应用及背景介绍 149
6.6.1 LIGO 数据网格的引力波探测数据分析应用 150
6.6.2 基于 BPEL4WS 的银行开户法律法规验证 151
6.7 小结 153
第 7 章 结束语 154
参考文献 157
致谢与声明 167
附录 A LIGO 引力波探测数据分析实例 SF1~SF3 的完整验证性能及其比较 168
附录 B SF1~SF3 待验证业务逻辑的对应 LTL 公式表 175
附录 C 基于标准域分析的验证策略分解定理证明 176
个人简历、在学期间发表的学术论文与研究成果 178

主要符号对照表

BMC	Bounded Model Checking 算法
BPEL4WS	Business Process Execution Language for Web Services 规范
COI	影响锥方法（Cone Of Influence）
CTL	分支时序逻辑（Branching-temporal logic）
DAGMan	Condor 中的 Directed Acyclic Graph Manager 模块
GridPiAnalyzer	（文中基于状态π演算的）网格服务流形式化验证原型系统
LIGO	Laser Interferometer Gravitational Wave Observatory 项目
LTL	线性时序逻辑（Linear-temporal logic）
NuSMV2	NuSMV 符号模型验证引擎
OGSA	开放式网格服务架构（Open Grid Services Architecture）
OGSI	开放网格服务基础设施（Open Grid Services Infrastructure）
OSG	开放科学网格（Open Science Grid）
PSL	性质描述语言（Property Specification Language）
SF1~3	（文中）LIGO 数据网格引力波探测数据分析的服务流实例 1~3
SMCA	符号模型验证算法（Symbolic Model Checking Algorithm）
UML	统一建模语言（Unified Modeling Language）
WBI™	IBM 的 Websphere Business Integrator 产品
WSBPEL	Web Service Business Process Execution Language 规范
WSRF	Web 服务资源框架（Web Service Resource Framework）
XML	可扩展标记语言（eXtensible Markup Language）

第 1 章　引言

1.1　网格技术的起源和发展

随着科学研究的不断深入和交叉学科的涌现，越来越需要知识技术的结合，相关学科信息的共享和多种科研资源的协同工作。因此，如何协同分散在各地的大量科研资源来完成各种复杂科研问题的求解成为了一个关键问题。网格技术[1,2]的产生正为解决跨组织、跨地域的大规模资源共享与协作提供了一种革新方法。网格最早是借助电力网的概念提出的，其理念是利用互联网或专用网络把地理上广泛分布的、跨组织的各种异构资源（包括 PC 机、计算机集群、存储系统和可视化系统等等）互连在一起，形成一个虚拟的超级计算机，其目标是希望计算机一旦接入网络就能获取源源不断的计算能力。从 1995 年到 1997 年，在美国就研制成第一个现代意义上的网格环境 I-WAY[3]。随着 1998 年“The GRID”[2]一书的出版，网格的概念和意义开始被更多的研究人员所认识接受。目前在很多国家，网格技术已被提升到国家战略高度。近年来的国家网格计划包括如美国的 Teragrid[4]项目、开放科学网格（OSG）①项目，欧洲的 EGEE 项目[5]、D-Grid 项目②和日本的 Naregi[6]等等。中国的国家网格计划包括 ChinaGrid[7], CN-Grid[8]和 CROWN[9]等。

随着网格技术逐渐成为计算机领域的一项主流技术，其内涵开始不再局限在“计算”的狭窄范畴内，很大程度上被上升为一种理念。这其中就包括注重大容量分布数据的存储、传输、复制、分析的数据网格，强调动态资源分配、调度及价格协商的服务网格（也称 On-Demand Computing），注重动态性和安全性的跨领域虚拟组织（Virtual Organization）以及更高层的语义网格和知识网格等。网格的内涵正在被不断拓宽。

1.2　网格服务与网格服务流

在网格技术逐步走向成熟的同时，如何拓展网格用途，为各种科学和工业

① The Open Science Grid consortium; 见 *http://www.opensciencegrid.org/*
② D-Grid Initiative Home page; 见 *https://www.d-grid.de/index.php?id=1&L=1*

应用提供一种通用的基础设施服务，成为了网格研究的一个发展趋势。2002 年 6 月，Globus 联盟首次在全球网格论坛（GGF）上提出了开放式网格服务架构（OGSA[10]）的理念。OGSA 将网格定义为“一个可扩展的网格服务集合，这些网格服务能够通过多种不同方式组合起来满足虚拟组织的不同需要”[10,11]。OGSA 的观念是面向服务的，即将计算资源、存储资源、网络、数据库等全部通过标准的、统一的服务抽象出来。然而 OGSA 只是一个不涉及软件架构的高层规范。针对 OGSA 规范的实现，先后曾提出了开放网格服务基础设施（OGSI）[10]和 Web 服务资源框架（WSRF）[12-14]。其中，WSRF 相对 OGSI 的优势在于它在网格服务中进一步扩展了对有状态资源的定义和管理，从而以一种轻量级的方式将现有的 Web 服务技术与网格技术相结合，确保了两者的兼容性，为异构网格资源的获取和管理提供了标准的统一接口。在实践中 WSRF 被证明是一种有效结合 Web 服务技术的网格服务标准，并在 Globus Toolkit 4[15,16]中得到了实现上的支持。关于 WSRF 和 OGSI 的比较可详见文献[17-19]。正是由于以上工作，使得网格中资源的业务功能能够以一定粒度的服务形式抽象并表示出来，每种服务都清晰地表示其业务流程和价值，用户可通过已发布的业务接口来使用服务，从而透明地对网格资源进行利用。

此外，网格“提供非平凡服务”的标准决定了网格应用将涉及对多服务的组合与协调，以形成具有更强功能的网格服务，从而实现对大规模资源进行管理和对各类复杂任务进行处理的能力。一方面，考虑到业务逻辑的重用性和可维护性，单个网格服务所提供的功能单一，单个资源提供者能提供的资源也有限，因此网格应用不是对单一服务实体的简单调用，而是大量网格服务间的交互、协作和组合；另一方面，高能物理、天文、电子商务、现代集成制造系统[20,21]等科研和工业领域中所涉及的复杂建模、结构分析和数据处理应用也需要网格提供对期望服务流程进行建模、组合和自动运行的能力。对此，OGSA 将网格服务流定义为：“能够完全或者部分自动执行的服务组合，它根据一系列过程规范、文档、信息或任务能够在不同的服务之间进行传递与执行”[17]。注意以上“服务”不只局限于“Web 服务”的概念和技术。为了将更多现有的典型网格工作流[22,23]模型和规范纳入本文的研究范围，本文中网格服务流的范畴进一步扩展为为完成目标领域中的特定业务逻辑而进行的网格中计算/存储任务、服务功能调用和计算模式（如并发模式、管道模式）的流程协作。

目前主要的网格服务流建模和实现方式包括自顶向下方式和自底向上方式

[24]。按照不同学者的理解和习惯[25,26]，也有人将其称为静态方式/动态方式、用户驱动方式/服务驱动方式或者服务编排模型/服务协作模型。在自顶向下方式中，用户往往在网格服务流执行前预定义一个抽象的过程模型。该抽象过程模型描述了用户所期望完成的业务逻辑，它包括所需完成任务的集合以及任务间的数据流/控制流依赖关系。其中，每个任务还可以包含相关的服务信息，以将该任务最终绑定到实际完成操作的具体网格服务上。图结构（如：Condor DAGMan③等）和程序语言（如：Glue[27]）常被用来对这些过程模型做出描述。另一方面，自底向上方式在网格服务的自动选择和绑定之外，强调通过采用谓词演算、进程代数或启发式规则等工具对网格服务流的生成方案进行规划，实现服务流的自动生成。这种自动化可以通过一系列服务间的偏序关系和组合关系（如：服务间的时序约束，输入输出的条件匹配等）实现。然而不论是以上的哪种方式，它们得到的最终结果仍多以网格服务流的过程模型来呈现[17,22]。

因此，作为网格核心技术之一，网格中对服务流的建模、组合及其实现技术已经成为了网格技术的一个重点之一。

1.3　网格服务流的正确性保障与形式化验证

1.3.1　网格服务流正确性保障的重要性

正如前文所述，随着万维网、Web 服务和网格等技术的飞速发展，使得越来越多的商业和科研应用开始借助它们实现对所需功能和资源的共享、发现、重用和组合，以更低的成本、更快的速度、更灵活的方式实现不同的商业目的和科研用途。与此同时，随着实际应用规模和涉及领域的不断扩大，使得现有服务组织的功能、变化、规模和复杂性不断达到新的高度。因此，基于现代信息技术和网络技术的支撑，如何在满足用户实际需求和约束条件下正确、合理地组织服务，检验服务流对用户业务需求的有效实现成为了网格技术和服务科学[28]领域中的一个重要研究问题。

以银行业为例[29]，在其不断加快信息化步伐的同时，也伴随着各种金融规范的出台。这些金融规范和法令不仅体现银行自身的管理规范和业务需求，同时也包括了机构和政府的各种金融法规约束。例如 2003 年中国人民银行出台的金融机构反洗钱规定[30]，就对银行开户服务流中的大额交易需要如何进行上报、

③ Directed Acyclic Graph Manager; 见 *http://www.cs.wisc.edu/condor/dagman/*

结算时必须验证的个人客户身份信息和或代理人信息等做出了明确的规范和约束。任何不满足对大额交易上报和验证要求的开户服务流程，则其信息化实现不仅是不正确的，更将被认定为是违法违规的。这不仅是在中国，在国际上也有对应的新巴塞尔协定[31]、欧洲反洗钱法规定[32]、美国爱国者法案[33]等类似法规约束。因此，虽然现有的 Web 服务和网格技术已经为其服务的整合与协作搭好了基础信息化平台，但如何在其之上根据实际应用需求和约束设计和实现相应的服务流以确保其业务的正确有效性是一项必不可少的任务。

同样在科研领域，由于网格服务流肩负着基于大规模异构资源共享实现复杂网格应用的使命，因此如何确保网格服务流在设计和实现上的正确性[34-44]也是一个关键的任务。例如，LIGO 项目[45-48]中的引力波探测数据分析是对由爱因斯坦广义相对论所预言的引力波信号进行检测的重要科研应用。为了对天体中如中子星和黑洞等致密物质的巨大能量活动所可能产生引力波作出检测，LIGO 数据网格[27,49,50]正为其提供了一个基础研究平台，且自 2000 年起至今它一直随着 Globus Toolkit 技术一起不断成熟。因此，引力波探测数据的分析是目前较典型且具有研究价值的网格应用之一。一方面，由于该应用自身涉及海量数据处理及复杂多服务间的协作，运行一次完整的引力波探测数据分析可以涉及上千个不同任务间对引力波探测数据的计算、转换、复制和管理；另一方面，由于潜在引力波信号异常微弱，为了回避探测过程中的噪声和不可靠的信号源，以上的复杂协作过程必须遵守严格的干涉仪探测器工作状态约束、服务调用间的时序关系约束以及数据的偶然性分析等关键业务需求（详见 4.4 小节）。运行一个有缺陷或是不符合实际业务需求和约束的网格服务流，将造成异常结果的产生和错误数据的利用。这对于以上具有重大科研价值和社会意义的网格应用所造成的负面影响和经济损失将是巨大的。它不仅导致用户的既定需求无法得到满足，更会造成珍贵网格资源自身的浪费，并影响其他用户对网格资源的良好使用。事实上，由于网格中资源的贵重性（如：设备仪器的高昂造价和使用成本）和应用任务的耗时性（如：引力波探测数据分析服务流的一次运行耗时可达 3000 天的 CPU 时间），更使得网格用户无法承受在经济和时间上的损失，凸显了网格服务流正确性保障的意义所在。

但在另一方面，虽然网格服务协作的重要性已被广泛认同，但当前对网格服务流的研究更多的集中在其自动执行、服务绑定、事务处理等“使能技术”上。对于网格服务流在其形式化语义、业务逻辑验证和验证性能提高等方面的

进一步研究工作却仍然是一项亟待解决的任务[37,51-53]。而这方面的形式化验证技术研究工作正是确保整个网格服务流正确工作，保障网格用户既定需求得以满足的关键技术。因此，这更增加了网格服务流正确性保障工作的迫切性。

1.3.2　研究挑战及原因

网格服务流自身的特点及其技术发展也为研究通过形式化验证技术解决其正确性保障问题带来了以下六方面的挑战。

（1） 专业领域的知识差异：网格应用往往用以解决特定科研或工业专业领域中的复杂问题，实现其相应的业务逻辑和约束。但由于领域专家往往不熟悉网格服务流设计和实现技术中涉及的相关技术和理论工具，而应用网格设计人员对专门的应用领域知识又难以有深入的理解，因此这增加了所需应用的业务逻辑和约束在网格服务流中正确实现的难度。

（2） 应用规模的复杂性：网格应用经常涉及多服务间的协调组合和大量数据间的交互传递。例如，图 1.1 中给出了一 LIGO 数据网格的实际引力波探测数据分析服务流示例，它用以分析天体中致密物质的能量变化所可能引发的引力波源④。一方面 LIGO 用户期望整个服务流能够在满足他们既定需求和性质的前提下正确的运行（如：待分析数据必须按一定顺序经过多次必要的偶然性分析，以确保数据处理结果的真实有效），另一方面对于这样一个复杂的网格服务流，他们也难以可靠地以人工方式对如此复杂的服务交互细节、相互依赖和全局状态变化作出详细验证和推理。必须有一种更严格的、更高效的方法来对以上性质和约束作出自动验证。

（3） 网格服务流语义的非形式化：随着现今各类网格服务组合与执行语言，以及网格服务流中相关并发与管道模式的相继提出和应用，网格服务流规范在其语义上正不断变得更复杂、更多样。然而这些复杂规范却急需一个成熟、完整的形式化模型来精确刻画其服务协作过程中的内部行为、交互过程和状态变化。特别是对于其中如出错补偿、全局终止和网格服务流模式等高级特性，形式化语义的缺乏可以导致服务流规范自身语义的晦涩和歧义性。因此容易出现由于规范复杂性和非形式化，导致服务流运行结果出现与用户预期业务逻辑相违背或不曾预见的异常情况。

（4） 网格服务流模型的多样性：现有各应用网格系统往往有着自己特殊的抽

④ 有关该应用的详细信息可参见 4.4 小节中的验证实例和第 6 章中的应用背景介绍。

象过程模型对网格中服务的交互和组合进行描述，它们有着不同的表达能力、模型元素和语法语义，并依赖于不同的程序语言、脚本或可视化语言进行表示。这也使得服务流在建模、重构和网格平台迁移（如目前从 LIGO 数据网格应用到开放科学网格 OSG⑤的迁移）过程中出现疏漏。

（5） 网格服务规范的独特性：Web 服务资源框架（WSRF）引入了有状态资源的概念，使其不仅在传统 Web 服务基础上通过内部行为（收/发 SOAP 消息）和接口事件形成服务间的调用与协作，更使网格服务的运行可以通过对有状态资源的生命周期管理，实现服务行为和应用状态间的动态关联和相互影响。此类形式在形式化方法领域中也被称为是状态/事件混合模型[54-56]。然而目前却尚无直接针对网格服务这一特点的形式化语义和工具，来更好地为其相关正确性分析和验证工作服务。

（6） 网格环境的动态性：网格中倡导的虚拟组织（VO）自身的动态性使得在网格中包括服务获取策略、可用服务等信息在每次网格服务流执行过程中都可能不断地在发生变化。这不仅体现了网格服务流正确性保障的必要性，同时也对相关形式化验证技术的性能提出了更高的要求。

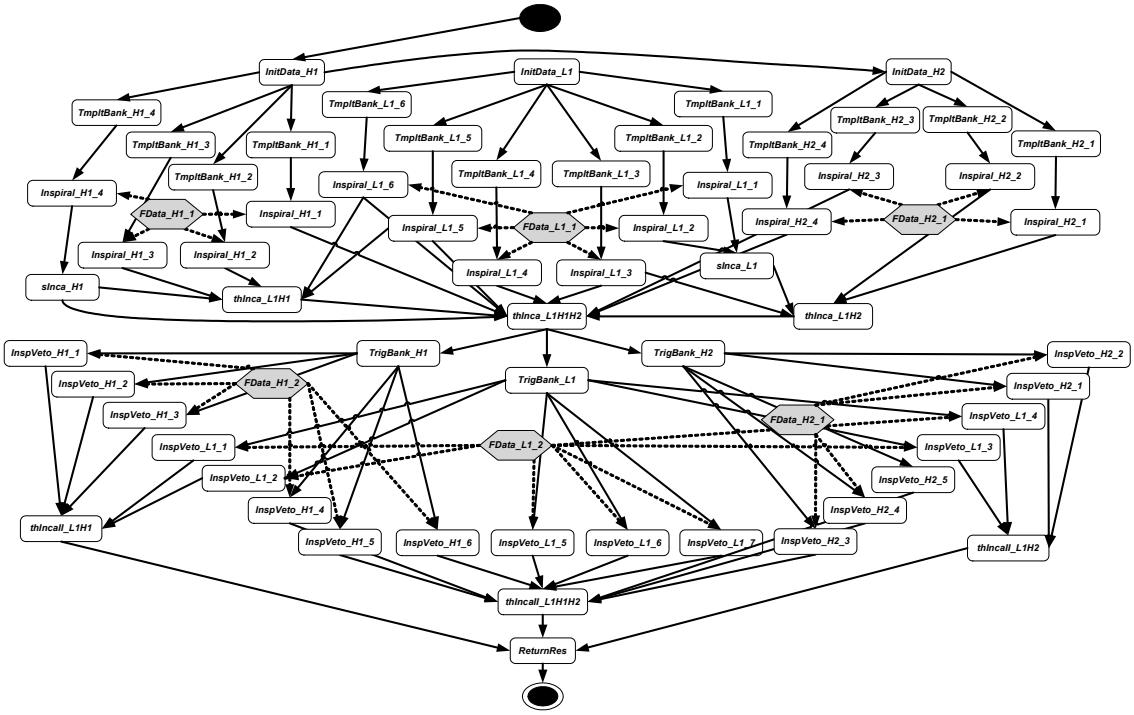


图 1.1 引力波探测数据分析服务流实例

⑤ The Open Science Grid consortium; 见: *http://www.opensciencegrid.org/*

1.3.3　本文的网格服务流形式化验证范畴和待解决问题

IEEE 软件工程标准术语表中对正确性的定义如下：“…... freedom from faults, meeting of specified requirements, and meeting of user needs and expectations”[53]，它表示了对用户期望需求的满足和对规范要求的无过错实现。而从形式化验证（Formal Verification）自身的定义也可以看出，它是“对给定系统模型是否满足规定性质要求的数学验证方法”[57]。这里的“性质要求”主要有着两方面的来源，它可以是由规定的系统规范所提出的约束要求，也可以是由主观上用户所期望的业务逻辑所提出的需求。根据以上两定义，本文中为保障网格服务流正确性所研究的形式化验证工作亦包含以下四方面的范畴（见图 1.2）：

（1） 结构验证：确保网格服务流中服务执行的可达性及其可终止性。这是最基本的要求。

（2） 网格服务流规范的语义约束验证：检验由具体网格服务流规范中所定义的如无消息竞争、变量垃圾回收等语义约束。

（3） 用户业务逻辑需求的验证：确保网格服务流不仅在客观“技术上”正确（满足规范自身的语义约束），同时也满足用户主观的既定业务需求。

（4） 业务逻辑的一致性验证：确保待验证的业务逻辑需求本身不存在相互的冲突和冗余，保证网格服务流业务逻辑验证的有效性。

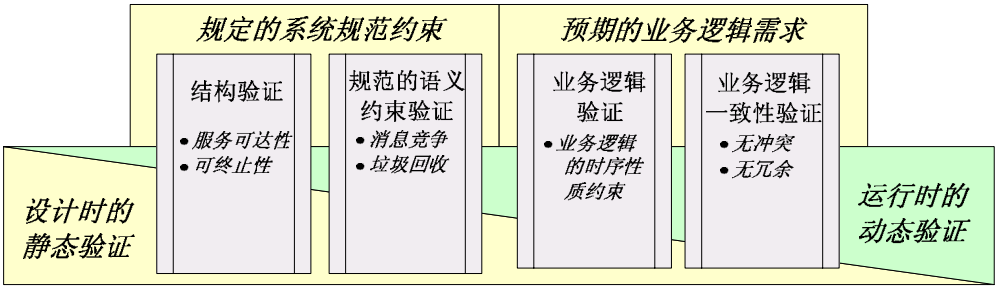


图 1.2 本文中网格服务流的形式化验证范畴

为了在网格服务流中对以上几方面做出分析验证，需要解决以下问题：

（1） 寻找并提出适合于网格服务流相关规范及特点的形式化理论与方法；

（2） 为现有网格服务流规范提供严格的形式化语义；

（3） 基于所提出的形式化工具和模型语义，实现对以上网格服务流正确性的动/静态验证分析方法；

（4） 有效提高网格服务流正确性验证在时间和空间上的整体性能；

（5） 将以上几部分工作相整合，形成网格服务流正确性验证分析的理论体系和自动化工具实现与应用。

1.4　网格服务流形式化验证技术的研究现状

不同于传统对网格服务流相关规范和“使能技术”（Enabling Technique）的研究，如网格工作流的事务处理、服务容器等，本文在其之上针对网格服务流正确性保障的形式化验证问题进行了理论和技术上的研究。本小节从整体上对现有的网格服务流规范和相关形式化验证技术的研究现状做出综述，并阐明了本文研究内容与方法的不同之处。

1.4.1　网格服务流的建模

随着网格工作流成为各类应用网格系统的必备组件之一，众多网格服务流规范与模型也相继被提出。它们不仅在表示和描述能力上互有差异，在模型元素及其语义上也互有不同。从模型表示上，现有网格服务流规范与模型根据不同应用领域的需求可以分为：（1）基于 XML 等的常用标记语言：典型代表包括 GridAnt[58]，BPEL4WS（1.1）[59][⑥] / WSBPEL[60]和 Gridbus 工作流[61]等；（2）可视化模型语言：如 Triana[62]，JOpera[63]以及 BPEL 的可视化建模[⑦]等。（3）其它自定义的脚本语言（如：Condor DAGMan）和程序语言（如：Glue[27]）。从描述能力上，Pegasus[47,64]和 Gridbus[61]等服务流模型主要针对有向无环图进行建模。而包括 Askalon 所使用的抽象网格工作流语言（AGWL）[65]，GridNexus 的 JXPL[66]，全球网格论坛（GGF）组织的网格服务流程语言（GSFL）[67]等同时也支持并发循环等其它流程结构。从模型语义上，各模型也有其一定的特殊性。例如，AGWL 中有着直接构建点对点/广播式数据流通信的模型元素；AGS-Flow[68]的模型中则拥有一类特殊的控制流节点（Adaption Control）允许在一定控制范围内对控制流的触发规则进行定义；DAGMan 则允许通过其任务的 PRE 或 POST 参数所定义的脚本动态重构其后续流程。从模型的通用性上，不同的网格服务流通常有着自己的特殊模型格式。例如，Teuta[69]以 UML 活动图作为其模型描述；I-Lab（GADL，Petri）[70]的模型则是基于 Petri 网的描述；而 GridAnt 的模型则是基于 Apache 的 Ant 工具。

⑥ 除非显示说明，以下本文中所指的 BPEL4WS 特指其 1.1 版本规范，以此和 WSBPEL(2.0)规范相区别。
⑦ 如 ActiveBPEL™ Designer; 见 *http://www.active-endpoints.com/active-bpel-designer.htm*

以上差异性使得系统对不同的网格服务流规范需要有不同的理解和行为定义，增加了不同网格服务流间集成的困难。值得注意的是，由于 Web 服务技术和网格技术相集成的这一重要趋势，将成熟的 Web 服务执行语言与 OGSI，WSRF 标准相结合来定义网格中的服务协作和组合成为了一项重要的任务。特别是随着 BPEL4WS 成为 Web 服务执行语言事实标准，现有工作已经开始对 BPEL4WS 基于 OGSI 和 WSRF 规范的扩展进行了研究和实现[71-73]。

与上述工作不同，本文无意重复定义一种新的网格服务流程模型，而希望通过寻找和提出一种适合于网格服务流特点的形式化建模与验证工具，从而：（1）通过形式化的数学方法，对网格服务协作中的关键特性及操作进行明确定义，严格给出其服务交互过程中的精确执行语义；（2）为典型的网格服务流规范提供一个统一的底层形式化语义基础；（3）基于本文的形式化工具实现对网格服务流业务逻辑完整、自动、高效的验证方法、体系和工具。

1.4.2　形式化建模与验证技术及其必要性

形式化方法（Formal Method）是用于对系统进行规范化描述和可靠性验证的数学语言、技术和工具[57]。由于其严格的数学基础和分析技术的支持，因而在网络通信系统的安全性和信任检查、软硬件系统的设计验证、面向对象的程序语言设计等多个领域得到广泛应用。其中，模型验证（Model checking）[57]是形式化验证方法中的一项主要技术。概括地说，它包括三个步骤：待验证系统的形式化建模、逻辑性质的描述、在系统模型上验证逻辑性质的成立与否并给出反例。典型地，这里的逻辑性质可以用一个分支时序逻辑公式[57,74]（CTL）、线性时序逻辑公式[57,74]（LTL）或μ演算逻辑公式[57]进行描述。其中，基于 CTL 的模型验证和基于 LTL 的模性验证在基本思路上有所不同。前者试图在有限状态系统模型 M（如：Kripke 结构）中找出所有满足 CTL 公式 f 的状态 $\{s \in M \mid M,s \models f\}$。逻辑性质的成立与否则取决于是否能够在系统模型 M 中找到这样的 s。它的复杂度为 O(m*k)，其中 m、k 分别表示 M 和 f 的规模；后者试图判别有限状态系统模型 M（如：有限自动机）所接受语言与待验证 LTL 公式的反“$\neg f$”（同样转换为一个有限自动机后）所接受语言间的交集是否为空，以此表征 M 中不存在违反 f 的行为，它的复杂度为 m*$2^{O(k)}$。相对的，虽然 CTL 和 LTL 均可以视为μ演算的子逻辑，但由于μ演算中不动点算子的语义及其表示上的复杂性，因而在实际应用中较少被直接用于对逻辑性质的描述。实际上，虽然针对如 ATL

（Alternating-Time Temporal Logic）[75]、TLA（Temporal Logic of Actions）[76]和 FIL（Future Interval Logic）[77]等其它时序逻辑以及 CTL、LTL 和μ演算的自身扩展与可视化[78,79]（如：针对π演算等移动进程代数的π Logic[80]和π-μ-Logic[81]、IEEE 标准的 PSL 语言[82]）亦有相应的模性验证算法研究，但 CTL、LTL 仍然是实际中用于性质描述和模型验证的两种最常用逻辑。文献[57,83-90]中分别研究了针对它们的符号模型验证、Bounded Model Checking、组合验证、抽象技术、Partial Ordering 技术和验证向导等性能改进方法的实现。而由于 CTL 和 LTL 在性质描述时的表达能力互不兼容，因此通过两者在模性验证上的优劣比较[74,91-93]，Vardi 等人[74]系统指出了 LTL 相比 CTL 具有：线性时序性质的描述和反例生成更直观易懂、更适应于模块化的组合验证和实现抽象技术的优点。且虽然理论上 LTL 模型验证的复杂度高于 CTL 模型验证的复杂度，但由于两者所能描述的公式 f 的复杂度不同，因此 Vardi 等人也同时强调特别是对于开放系统（Open Systems）[75]，CTL 并不具备比 LTL 更好的验证复杂度。

另一方面，在 Web 服务领域，随着 BPEL4WS 等 Web 服务组合/执行规范语言的不断涌现，利用形式化方法的严密性和分析手段提高此类复杂模型的语义精确性和实用性已经成为了一种共识[94,95]。“它（形式化方法）不仅允许我们对现有的规范及其相关问题进行分析和推理，也使我们能够发现规范中所隐藏的问题”[96]。现有工作中已经尝试了利用有限自动机理论[57]、Petri 网理论、进程代数理论[97]及相关模型验证技术和针对进程代数的互模拟（Bi-simulation）分析[98]对 Web 服务流进行形式化建模和分析。

1) 有限自动机理论：有限自动机是描述系统状态及其变迁的基本模型。通过有限自动机可以以状态驱动的方式对 Web 服务流中的状态进行显示的刻画，通过状态的迁移反应服务流中的行为和约束关系。Andreas[99]等人尝试了将 BPEL4WS 规范转换为确定型有限状态自动机，以支持基于状态的 BPEL4WS 服务匹配；Fu[24,100]等人为了进一步扩充自动机对数据的表达能力，提出了一种守备有限状态自动机（Guided Finite State Automata），并以此对 Web 服务的组合进行了形式化描述。Farahbod 和 Glasser[101]等人则实现了对 BPEL4WS 的抽象操作语义的形式化工作；而 Berardi[102]等人直接将有限状态自动机视为他们所提出的 Web 服务迁移语言（Web Service Transition Language）的概念模型。基于以上各类自动机模型，则现有模型验证技术及引擎（包括 NuSMV2[103]、RuleBase[104]等）可以直接被集成用于 Web 服务

流在时序性质方面的业务逻辑验证。这方面的系统化工作实现包括 WSAT[100] 和 LTSA-WS[105]等。

2) Petri 网理论：Petri 网由于其图形化表示和成熟的分析方法因而也在 Web 服务流的建模与分析中受到关注。通过 Petri 网中的库所和迁移可以有效描述 Web 服务的交互行为和控制流关系。Tang[106]等人为了描述 Web 服务流中的活动与资源的同步，提出了一类扩展 Petri 网—“服务/资源网”，以实现对 Web 服务组合的形式化描述；类似的，Yang[107]等人则基于一类着色 Petri 网实现了对 Web 服务组合的形式化描述。此外，Ouyang[108]和 Lohmann[109] 等人均为 BPEL4WS 提供了其基于 Petri 网的模型语义。由此，Petri 网的相关分析技术可以用于对 Web 服务流中死锁、服务可达性等结构性质作出分析。这方面的系统化工作实现包括 WofBPEL[108]和 Tools4BPEL[109]等。

3) 进程代数理论：与 Petri 网不同，进程代数是一种文本化（Textual）的形式化方法。进程代数可以通过其标准的操作语义对目标系统的行为作出推演。它的互模拟分析理论专门针对于系统间的各类等价性的判别（如可观测等价等）。常用的进程代数，包括 CCS（Calculus of Communicating Systems）和 π演算（π Calculus）[98]，由于其可组合性而被认为是对 Web/网格服务组合、Web/网格服务流进行形式化建模与分析的合适工具。以上进程代数中，通过进程的概念可以对单个的服务实体行为进行建模，通过它们的同步操作语义可以描述服务实体间的交互与组合，通过输入输出端口的操作语义可以对服务与环境间的交互做出描述。Frank 等人首先利用π演算为经典的工作流模式进行了形式化工作[110]；Decker 等人则针对了常用的服务交互模式（Service Interaction Pattern）给出了其基于π演算的形式化语义[111]。Salaűn[112]等人为基于 CCS 的 Web 服务建模与互模拟分析曾作出完整的可行性说明；Camara[113]和 Liu[114]等人则均运用了 CCS 对 Web 服务的动态行为语义做出了描述；另一方面，Wei[115]、Gao[116]和廖军[117]等人则均对 Web 服务组合提供了基于π演算的形式化模型。基于这些形式化基础，以上方法中多利用了相关互模拟分析工具（如：CWB – Concurrency Workbench⑧和 MWB – Mobility WorkBench[118]）对 Web 服务的可替换性做出等价判别。而除了直接利用如 MWB 等工具进行互模拟分析外，目前在基于进程代数对 Web 服务，尤其是网格服务流形式化验证方面的系统化实现工作仍然较少。

⑧ The Edinburgh Concurrency Workbench; 见 *http://homepages.inf.ed.ac.uk/perdita/cwb//*

同样的，在网格系统中有效的形式化语义不仅能够对网格系统的特性做出精确定义，还是加速网格系统分析和设计的重要手段[119,120]。网格领域自身也认识到了形式化方法与验证的重要性。文献[119]首先确立了为网格系统建立形式化语义的重要性。基于抽象状态机（Abstract State Machine），该文从网格的动态运行时语义出发，定义了网格系统中所必须提供的基本服务和功能，从而区分了网格系统和一般分布式系统间的本质特征。类似的，张鹏[121]等人应用逻辑 Petri 网表达了网格体系中的不确定性，并基于给出的网格体系模型分析验证了网格体系的正确性和完整性。Nemeth 等人[122]通过提出一种 Gamma 演算对网格工作流执行引擎的模型进行了形式化，从而为网格工作流执行引擎的实现提供了一个框架。此外，Groth[123]等人从网格中服务溯源的角度，为网格系统中服务溯源的记录协议提供了基于抽象状态机的形势化模型，以便于对其活性做出检测；卢暾[124]等人则展示了如何基于 I/O 自动机对网格服务挖掘中的服务组合进行形式化描述。

1.4.3　π演算的应用与优势

在以上各类形式化理论工具中，π演算[98]由于其可组合性和移动性的特点，对开放通讯系统的自然描述，以及完善的理论和应用支持成为 Web/网格服务流形式化建模与验证的一个有效工具[125,126]。图 1.3 中给出了多项π演算（Polyadic π Calculus，以下统称为π演算）的语法。π演算中最基本的概念是名字（name）和进程（process）。通过名字可以表达系统中的一个最小交互动作，而进程则可以表示一个（子）系统的完整行为。其中，代表着系统行为的进程演化则是通过下列守备（guard）“.”， 组合（composition）“|”，选择（choice）“+”，匹配（match）“[]”，限制（restriction）“*new*”和复制（replication）“!”等基本动作和操作语义来实现的：

- 输出动作（$\bar{x}<\tilde{y}>.P$）：表示通过名字 x 输出 $\tilde{y}$，并使系统其后的行为与进程 P 一致。在这里 x 可以理解为是一个输出端口，而 $\tilde{y}$ 则是输出的数据；
- 输入动作（$x(\tilde{y}).P$）：表示通过名字 x 接收一串名字，并使系统其后的行为与进程 P 根据接收到的新名字进行重命名后的行为一致。在这里 x 也可以被理解为是一个输入端口，而 $\tilde{y}$ 则是通过该端口接收所输入数据的变量集合；
- 不可见动作（$\tau.P$）：表示系统无需与环境交互，直接由内部动作演化为 P；
- 组合（$P|Q$）：表示进程 P 和 Q 的并发组合。它们可以相互独立的执行，也

可以通过两者相同的输入输出端口进行同步通讯；

- 选择（$P+Q$）：表示进程 P 和 Q 执行的不确定选择；
- 匹配（$[x=y]P$）：表示当名字 x 与 y 匹配时，系统行为与 P 一致；否则不发生任何动作；
- 限制（$(new\ x)\ P$）：表示名字 x 是限定在进程 P 中所出现的一个新名字；
- 复制（$!\ P$）：表示进程 P 的无限组合，$P|P|P|\ldots\ldots$。

$$
\begin{aligned}
P &::= \sum_{i=1}^{n} \alpha_i.P_i \mid (new\ x)P \mid\ !P \mid\ P|Q \mid\ \phi P \mid\ A(y_1,...,y_n) \mid\ 0 \\
\alpha_i &::= x<\tilde{y}> \mid x(\tilde{y}) \mid \tau \\
\phi &::= [x=y] \mid \phi \wedge \phi
\end{aligned}
$$

图 1.3 基本π演算语法

需要说明的是，在以上各类基本形式化理论中，本文选择在π演算基础上提出一种新的扩展（称为状态π演算）作为对网格服务流形式化建模、验证以及性能改进的完整形式化工具。本文的选择是由于π演算在 Web/网格服务流的形式化工作中有着以下 4 方面的优势。

（1）网格中虚拟组织的优势和动态性体现在能够通过不同域中的网格服务灵活组合来实现不同目的的复杂科研协作和工业应用。不同的服务提供者可以动态加入或注销所提供的服务，服务的请求者也可以动态地对网格服务做出选择。因此，π演算可组合性（Compositionality）与它自身的移动性（Mobility）为这样的网格服务流提供了一个灵活而自然的描述方式。在这里，可组合性指的是π演算拥有一个原子的组合操作符“|”，使它可以天然的以模块组合方式对一个复杂系统作出建模；而移动性则指的是通过π演算的传名和重命名操作可以对服务交互关系的动态改变和动态的服务选择做出直接描述。这两个特性在基本的 Petri 网和自动机中都没有直接的支持，而是需要通过额外的元素扩展与操作才能实现。事实上，π演算作为服务组合语言的有效性和适应性已经得到了 Lumpe 和 Nierstrasz 等人的专门论证[127-130]。

（2）基本的自动机和基本 Petri 网常被用来描述闭合系统（Closed System），即系统的行为是完全可控的，可以通过系统状态显式地完整确定下来。然而，π演算和 CCS 等进程代数则更适合被用以描述开放系统（Open System）[75,127-129]，即系统的行为由系统自身的可控状态及其和它外界环境（Context）的交互完整

确定[75]。例如：一个基本的 Büchi 自动机模型由于需要显示地对其状态迁移进行完整的描述，因而它建立的是一个行为完全可控的系统；而π演算理论自身就拥有可观测行为和内部行为的概念和语义，且可观测行为可由π演算进程及其与上下文环境的交互进行联合控制。而另一方面，开放系统的建模却对网格服务流非常重要。因为除了可控的服务过程模型外，由于网格环境自身的动态性，网格服务间的交互行为与调用往往需要受到网格环境信息的交叉反馈（如对备选服务的选择、及服务当前可用性等信息的利用）。

（3）随着各种 Web/网格服务流规范的不断提出，在形式化方法研究领域也展开了一种争论，即："哪种方法才是 Web/网格服务流的最佳形式化工具"，而其中π演算则是其中的有力竞争者之一[125,126]。在业务流程管理领域，为了证实π演算自身在应用上的优越性，现有工作中已经尝试了基于π演算对 UML（1.4）的状态图[131]、活动图[132] 、BPMN（Business Process Modeling Notation）[133]和工作流模式[110]等各种模型的形式化。因此，本文选择π演算的一个目的也是响应 Aalst 的号召[125]，即提出并研究一套以本文状态π演算为基础的网格服务流形式化建模与验证的系统理论和方法，从而向形式化方法研究领域展示本文状态π演算作为网格服务流形式化基础的有效性与优越性。

（4）在π演算之后又陆续提出了如 micro K_{LAIM}，Mobile Ambients 和 Seal 演算等其它移动进程代数（Mobile Calculi）[134]。虽然它们通过实现系统拓扑结构间名字、进程甚至是节点的传递使对系统移动性的描述上得到强化，但因其复杂性而目前仍多处于理论研究阶段，实际的系统化工具和应用支持较少。本文的考虑是，形式化方法不仅仅只是单纯的数学描述工具，而更是用来分析和验证实际问题的方法和工具。π演算相对拥有更多实际 Web 服务研究团体的支持和研究。例如，包括 BPEL4WS 和 XLANG 规范自身的设计就是声称基于π演算思想的。而另一方面π演算也有着直接的模型验证和互模拟分析技术及工具支持。其中针对π演算的模型验证包括有 Uppsala 大学的 MWB（Mobility WorkBench[118]）和 PISA 大学形式化方法与工具研究组的 HAL（HD-Automata Laboratory）[80]等典型代表。两者采用了不同的模型验证实现思路。前者基于了 Dam[135]对π演算模型验证的证据系统，而后者则通过π演算早迁移语义将有限π演算进程转化为等价的有限自动机，从而集成 JACK[136]引擎实现了对π演算的模型验证。因此，本文选择π演算同时也是平衡了语言自身的复杂性和理论应用支持的完备性。即一方面，本文研究结果展示了本文所提出的状态π演算已

经可以完整地刻画目标网格服务流语义并实现其系统化验证，另一方面通过本文的选择则可以得到更多实际应用领域的支持和形式化验证技术的选择。

1.4.4　Web服务资源框架与状态/事件混合形式化方法

虽然π演算作为 Web 服务流基本形式化理论工具的有效性已经得到认同，但由于其对系统状态管理能力的缺乏使它不能良好的匹配网格中 Web 服务资源规范（WSRF）的特点。WSRF 是网格服务的一个重要标准。它将网格应用中的各类上下文信息（如：服务运行的当前状态、所涉及的相关数据等）以“状态”的形式进行表示、抽象和管理，并将这种能力融入到了传统 Web 服务中，为实现复杂科研计算和数据处理业务提供了基础。各类支持 WSRF 规范的网格服务流模型和引擎的研究[11,137]，以及对传统 Web 服务流规范（如：BPEL4WS）的 WSRF 扩展[71,72]，也为其自动化实现提供了关键技术。

WSRF 规范的提出，使得在网格服务流中从动态执行语义的角度确立 Web 服务与资源状态间的相互关联和影响、对状态从产生到消亡的生命周期进行管理成为了一项重要工作。网格服务流的执行不只是 Web 服务之间的简单静态交互，而需要涉及服务行为、资源状态和整个网格服务流全局状态的相互影响。服务行为改变着相应的资源状态和服务流全局状态，这些历史状态信息又反过来影响着服务行为和系统事件的触发。然而另一方面，从上一小节π演算的语法（图 1.3）可以看出，π演算对系统的动作及其交互进行了显示的刻画，但对于系统状态的表述却是晦涩的。从解释π演算语义的标号迁移系统来看，它用来隐含识别状态的唯一信息就是当前进程的标识。因此，虽然现有文献[115,117,138]中已经论证了π演算操作和 Web 服务元素间良好的一一对应关系，但随着 WSRF 对有状态资源管理能力的扩展，使得如何保持这种良好的对应关系，实现与 WSRF 特点相匹配的形式化工具成为了一个重要的问题。此外，从对事件驱动形式化方法的研究也可知，π演算由于对系统状态的晦涩表示和管理能力的缺乏，使得它的一个不足在于进程的演化由当前可发生的动作决定，而不利于记录历史事件信息来对将来的进程演化做出约束和分析[139]；另一方面，由于π演算复杂的命名限定和重命名机制，其动作名往往高度抽象了其代表的实际业务含义。因此以上两者增加了π演算对实际系统和业务逻辑建模的复杂度。

实际上从形式化方法自身的研究领域来看，WSRF 的提出使得网格服务流有着状态/事件混合[54-56]的特点。如果将服务流本身视为一个系统，则有状态资源

的引入使得服务的执行可以获取、查询并改变系统的历史（状态）信息，通过对这些有状态资源的生命周期管理，从而实现服务行为和系统状态之间的动态关联和相互影响。形式化语言本身是一项用来描述“当进行系统操作时会发生什么事件，以及系统状态会如何演变”[139]的方法。因此状态和事件是描述系统行为最基本和常用的两个维度[140]。基于状态的形式化方法（如有限自动机等）更看重一个系统内部全局/局部状态结构的表述和演变，它通过状态的变迁对系统的行为进行建模；另一方面，基于事件的形式化方法（如多种进程代数）更看重系统系统间的事件交互，它将系统的行为描述为事件/动作的序列。以上两者从不同的维度看待系统的行为，有着互补的关系和特点[139]。文献[54]和[141]中就分别提出了标号 Kripke 结构和双标号迁移系统两种基本状态/事件混合的系统结构。而文献[56]则从时序逻辑的角度出发，将事件的概念引入了现有时序性质模式[56]中以支持它对系统事件的时序描述。其中，文献[54]更通过实例展示了状态/事件混合建模方法可以有效减少系统的状态空间，降低系统建模的复杂度。本文的已有工作[41]中也基于π演算的互模拟关系对 UML 2.0 状态图和活动图间的形式化语义关联和转换进行了分析，并结合 ATM 业务流程的示例发现状态/事件混合建模方法可以有效简化对业务流程的形式化验证。实际上除了简化系统建模与分析的优点，状态/事件混合形式化方法同时也是集成形式化方法领域（Integrated Formal Methods – IFM）的一个重要研究方向。文献[142]中就提出了 Fluent 的概念，其中每个系统状态被定义为一个布尔变量。每个 Fluent 包含了两个互不相同的系统行为，通过他们的执行分别将一个状态设为真或假。文献[143]则采用了形式化方法集成的思路，整合了 Object-Z 对状态迁移语义的描述和π演算中的进程推演规则，提出了一种状态/行为混合建模语言—PiOZ。

本文正是基于以上两个不同研究领域在相似研究目标上的发现，针对 WSRF 规范对传统 Web 服务实现了有状态资源生命周期管理的这一特点，提出了状态π演算这一形式化工具为网格服务流的形式化建模和验证服务。虽然传统π演算已经被接受为是针对 Web 服务流的有效形式化工具，但状态π演算进一步为π演算这类进程代数扩展了其对系统状态进行管理和动态关联的能力。它不仅是本文网格服务流形式化建模、验证及性能改进的基础形式化工具，同时也拥有简化系统和业务逻辑建模的优点。

本文所提出的状态π演算与文献[142]工作的不同在于，它不仅讨论了如何扩展现有形式化工具对状态/行为混合模型进行静态的表述（即扩展标号迁移系统，

见 2.3 小节），更定义了系统状态变化与系统行为之间的动态语义（即如何将一个基于状态π演算定义的实际系统应用演化为一个扩展标号迁移系统）。从这个角度来看，状态π演算可以看成是文献[142,143]工作的进一步推广。因为它进一步以系统行为交互为驱动，实现对状态的动态关联和注销，以支持系统状态的全生命周期管理。而文献[142]的方法中所有的系统状态仅由布尔变量进行二值定义，且只支持对系统中新资源状态（通过 Fluent 定义）的发现。此外文献[142,143]也没有实现进程演化中对状态结构的逆向约束和系统状态的动态注销。

1.4.5　本文方法的不同点

综上所述，本文整体方法的不同之处主要可以总结为四个方面：

1) 工作目的的不同。形式化方法不仅仅是一种数学语言，其意义更在于它是一种分析手段和工具。因此，本文不仅停留在系统的形式化语义建模上，而进一步讨论了对现有形式化建模与验证技术的结合、扩展和改良，以实现对网格服务流的正确性保障；
2) 方法特点的不同。本文结合 Web 服务资源框架（WSRF）的特点在基本π演算基础上提出了一种称为状态π演算的新形式化工具，以支持对网格服务流的系统化建模与分析，并以此解决 1.3.3 小节中所涉及的 4 类形式化验证问题。状态π演算不仅实现了进程行为交互演化与系统状态生命周期管理能力的相互集成，简化了对系统和业务逻辑的建模，文中基于状态π演算的网格服务流形式化验证方法也有着有效减少无谓验证代价，独立于特定模型验证技术的优点；
3) 工作对象的不同。本文并非针对网格系统及其组件自身的执行语义，而是对构建在网格系统之上以网格服务流为形式的各类网格应用实现相应的动/静态形式化语义和验证方法。同时利用网格服务流自身在串/并行结构的验证分解和典型错误过程模式等结构特征信息，以及通过对冗余和冲突业务逻辑的排除研究了对网格服务流验证性能的改进；
4) 工作范围不同。本文从 5 个递进的层次出发，为保障网格服务流正确性的形式化验证提供了一个完整的系统化解决方法与工具。图 1.4 给出了基于以下五个层次的整体方法流程框架：
   - 理论扩展：针对网格服务流的特性提出并分析了状态π演算形式化工具；
   - 形式化建模：基于状态π演算实现了网格服务流在服务描述、选择、协

作及常用模式的形式化语义；

- 验证分析：以状态π演算为理论基础为网格服务流提供了 1.3.3 小节中 4 方面形式化验证问题的动/静态验证方法和框架；
- 性能提升：基于服务流分解和错误过程模式的搜索向导两个思路提出了网格服务流正确性验证在时间和空间性能上的改进方法；
- 系统化方法集成与应用：最终为网格服务流的正确性验证提供一个完整的系统化方法与原型工具—GridPiAnalyzer[35,37,38,42]。

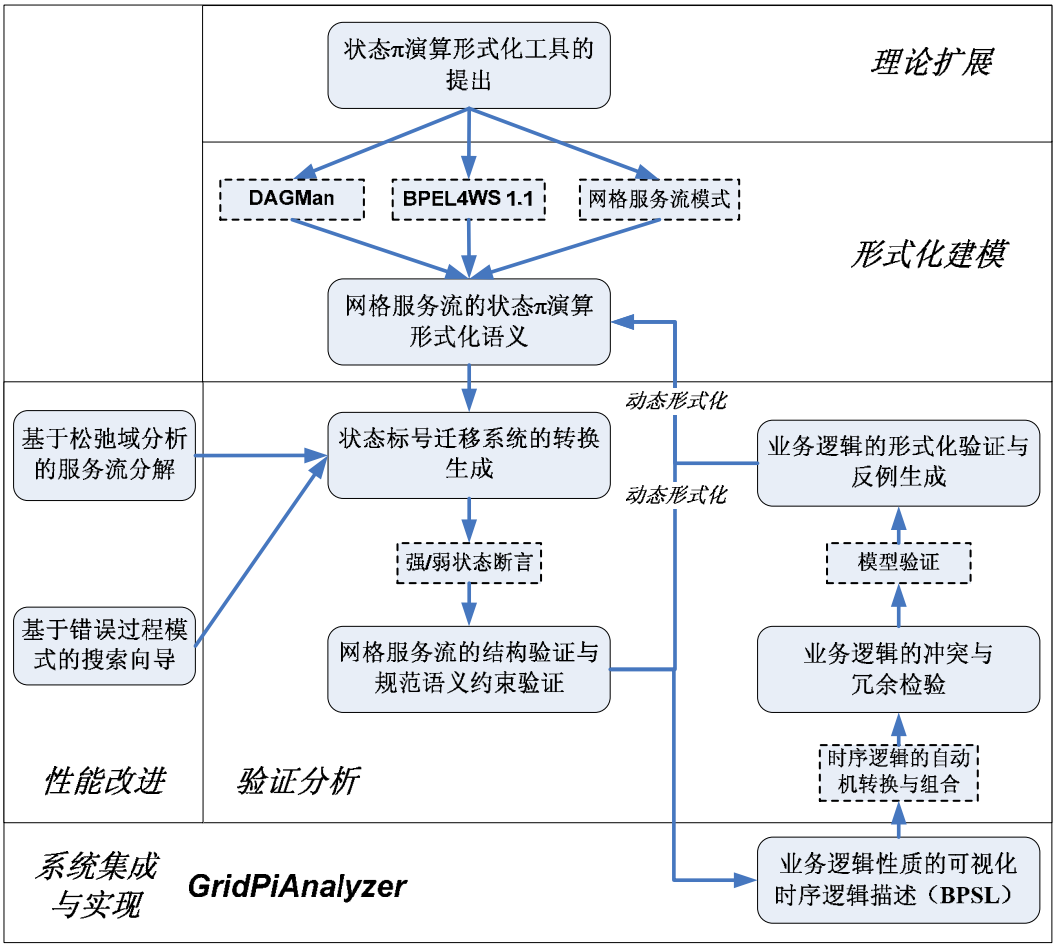


图 1.4　网格服务流形式化建模与验证整体流程框架

1.5　本文研究的主要内容和贡献

1.5.1　研究什么和不研究什么

本文工作是基于以下三个前提而围绕着网格服务流的形式化验证展开的。

（1） 本文研究的形式化验证目标不是网格系统本身，而是基于应用网格系统之上，以网格服务流形式为用户构建的网格应用；

（2） 由于系统正确性和系统验证/检验的定义内涵十分宽泛，因此本文所研究的形式化验证范畴限定在 1.3.3 小节中所给出的 4 方面问题；

（3） 本文的主要研究点在于目前形式化建模与验证领域的理论扩展和方法改进，以及在网格服务流中的实际应用。本文自身不侧重对本体、语义网格领域的具体研究工作，但不排斥其相关工作与本文成果的进一步结合。

1.5.2　本文组织与主要贡献

从理论到应用、从建模到验证、从语言到工具，本文为网格服务流的正确性保障建立了一套系统的自动化解决方法。它围绕着 5 个递进的层次展开，即：形式化理论工具的扩展、网格服务与服务流的形式化语义建模、验证方法的分析与实现、验证性能的改进以及方法的系统集成与应用。每个层次间都有着相互的依赖：理论扩展源于网格服务流规范中的实际特点，为服务流的形式化语义建立了基础并提供了便利；网格服务流的形式化验证方法构建在其形式化语义之上；性能提高则是对该形式化验证方法的进一步改进；而系统集成与应用则将以上四方面工作予以融合，并实现自动化的网格服务流验证工具支持。图 1.5 相应地给出了本文的组织结构。本文后续的第 2–6 章将分别围绕图 1.5 中的 5 个层次内容展开。在第 7 章中将给出全文总结。

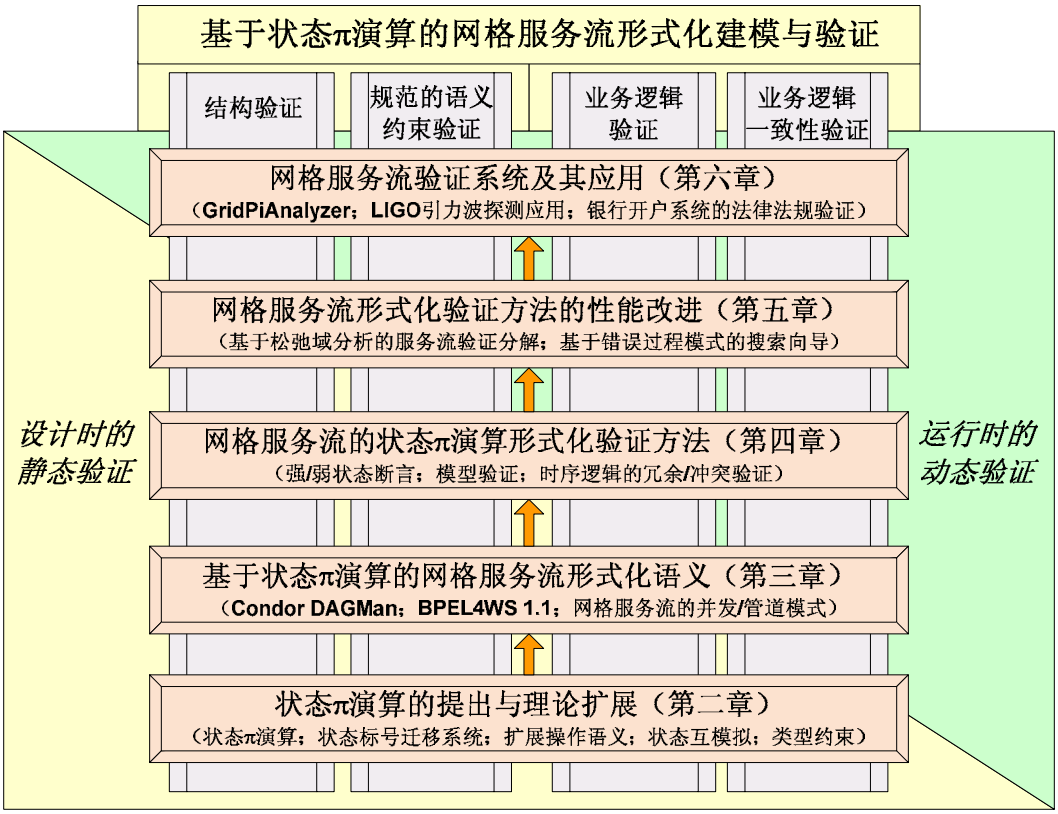


图 1.5 本文组织结构

本文的贡献可以用一句话总结为：为适应和简化网格服务流的形式化建模与验证，通过对状态 π 演算方法的提出与分析，结合模型验证技术实现了系统化的网格服务流正确性保障方法及其性能改进与应用工作。更具体的，它表现在以下 6 方面：

（1） 结合 Web 服务资源框架的特点提出并分析了一种状态π演算的状态/事件混合形式化工具。它扩展了基本π演算对系统状态的生命周期管理和动态关联能力，不仅简化了对系统和业务逻辑的形式化建模，同时也为本文网格服务流的形式化建模与验证奠定了理论基础。

（2） 基于状态π演算实现了对网格服务流中服务描述、选择、多实例化、控制流/数据流约束和网格服务流中常用并发与管道模式的完整形式化模型。在提供了对目前 Condor DAGMan、BPEL4WS 和相关模式的形式化语义同时，也验证了状态π演算自身针对网格服务流有着足够强的表达能力；

（3） 基于网格服务流的状态π演算语义，通过状态标号迁移系统的自动生成、强/弱断言检验的提出及模型验证技术的结合，提出了对 1.3.3 小节中 4 方

面问题（即：结构验证、规范语义约束验证、业务逻辑验证和一致性检验）的动/静态验证分析方法。该方法不仅能有效减少不必要的系统验证代价，同时具有独立于特定模型验证技术的优点；

（4） 基于服务流分解与状态空间搜索向导的两个思想，分别提出了网格服务流中的松弛域分解和错误过程模式搜索向导的方法，有效提升了网格服务流形式化验证在时间和内存占用方面的性能；

（5） 实现了对以上 4 个层次工作和相关技术的系统化融合，为网格服务流的正确性保障提供了自动化验证原型系统 – GridPiAnalyzer。通过对可视化业务逻辑性质描述语言 BPSL 的设计增强了 GridPiAnalyzer 的可用性。

（6） 基于 LIGO 项目引力波观测数据分析等实际网格应用案例，验证了本文方法的可行性和有效性。

第 2 章　状态π演算及其类型约束的提出与分析

2.1　本章引论

正如 1.4.4 小节的综述所述，网格中 Web 服务资源框架（WSRF）实现了对传统 Web 服务有状态资源管理能力的扩展，而这一点正与形式化方法领域中状态/事件混合建模与分析的重要研究思想相呼应。本章在这两个背景下提出了状态π演算这一新形式化工具为网格服务流的形式化建模与验证服务，它是 Milner 基本多项π演算[98]（以下简称为基本π演算）的状态/事件混合扩展。与现有工作相比（见 1.4.4 小节综述），本章提出的状态π演算以系统行为的交互为驱动实现了对系统状态的动态关联和注销，从而支持了对系统状态的生命周期管理能力。这使得状态π演算中实现了进程演化和状态操作语义的集成和相互作用，从而：（1）弥补了基本π演算中难以利用历史事件信息对进程演化进行约束和分析的不足；（2）通过状态命题的管理为进程中的动作和交互提供了灵活的业务含义抽象能力；（3）使状态π演算语义更匹配 WSRF 规范对 Web 服务的扩展思想，为后续基于状态π演算的网格服务流形式化建模、验证和性能改进奠定了理论工具基础，提供了建模和验证上的便利。

本章工作的结构如下：在 2.2.1 小节中，通过对状态π演算中状态、命题及其前后缀标识的静态模型定义，实现了进程交互过程中对状态和命题的识别和获取；在 2.2.2 小节中，通过给出状态π演算中具体行为与状态操作间的语法关联和状态操作语义，实现了对状态生命周期的管理；在 2.3 小节中给出了状态π演算的扩展操作语义，通过状态标号迁移系统的提出解释了状态π演算中系统状态变化与进程演化间的动态交互语义，从而在两者的相互作用中实现了对状态命题的设置与更新；2.4 小节则从状态/事件的混合维度提出并分析了新的状态互模拟和混合互模拟关系及其性质。最后，为了克服状态π演算中进程通讯只依靠简单端口名匹配的弱点，2.5 小节中还为其确立了进程交互中的类型（Sort）约束，从而实现服务交互中潜在的执行条件、数据结构和数据类型约束等重要约束语义的严格刻画。具体的，它包括以下两方面：（1）服务执行约束的考虑：包括服务的调用顺序、执行的前后置条件规则；（2）数据类型的考虑：包括服务交互过程中的复杂数据类型匹配、它们间的继承关系、聚合关系和数量关系。

关于基本π演算已在 1.4.3 小节综述中给出介绍，本章不再重复。本章的状态π演算是本文为网格服务流的形式化验证所提出的基础理论工具。它不仅是后续为不同网格服务流规范和模式建立形式化语义的工具（第 3 章），也是实现其验证方法和性能改进的基础（第 4、5 章），同时为以上建模和验证工作提供了便利。需要补充的是，虽然本章工作集中在形式化方法自身的研究，但通过后续对网格服务流的状态π演算语义、验证方法和系统工具的建立，可以有效屏蔽状态π演算在应用时其操作语义的复杂性。

本章以下内容的相关研究结论主要发表在：

- 网格服务链模型的验证分析技术及应用. *中国科学 F 辑: 信息科学*, 2007, 37(4): 467-485 （**SCI** 检索）

  Formal Verification Technique for Grid Service Chain Model and its Application. *Science in China, Series F: Information Sciences*, 2007, 50(1): 1-20（**SCI** 检索）
- 时间π演算及其弱时间互模拟分析. *计算机集成制造系统–CIMS*, 2006, 12(4): 511-516 （**Ei** 检索）
- Service Provenance based Abstraction of Grid Application Knowledge, In: *Int. Conf. on Semantic and Knowledge Grid*, IEEE Computer Society, 2006

2.2　状态π演算中的状态模型与状态操作

2.2.1　状态模型的静态语义

在状态π演算中，一个状态 *S* 被定义为一系列系统命题 *PROP* 的有限集合。

定义 2.1（系统命题）：一个定义在空间 *D* 上的系统命题 *PROP* 是一个二元组：*PROP*=(*ident, set*)，其中 *ident* 是该命题的唯一标识，而 *set* 是令该命题为真的所有常量的有限集合（$set \in D$）。

因此，$S=\{p_1, ..., p_n\}$，其中 $p_i (i=1,...,n)$是定义在同一空间 *D* 上的命题 *PROP*。记 $ident(p_i)$表示命题 p_i 的标识，$set(p_i)$表示命题 p_i 的当前值集合。在状态π演算中，为了增强状态对实际系统中不同属性的从属关系，系统命题的标识 *ident* 定义为如下的层次关系：

ident ::= *atom* | *atom.ident*

其中 *atom* 代表一任意的字符串标识，“.”表示标识的分隔符。由此我们通过定义标识之间的前后缀来确定其层次关系：

$prefix : ident \times ident' \rightarrow ident''$　　$prefix(ident, ident') = ident''$ IFF $ident = ident''.ident'$

$suffix : ident \times ident' \rightarrow ident''$　　$suffix(ident, ident') = ident''$ IFF $ident = ident'.ident''$

例如状态 S={($AvailableSrv$, {Srv_1, Srv_2}), ($Srv_1.Status$, {$Active$}), ($Srv_2.Status$, {$Input_Pending$})}可以表示系统中存在两可用服务 Srv_1 和 Srv_2，且 Srv_1 当前正在运行，而 Srv_2 则在等待输入。标识 $Status$ 是 Srv_1 和 Srv_2 的后缀。

为了完整的给出状态及其命题的静态操作语义，还需要定义以下三个状态与命题间的关系：

- $range$: $PROP$ → set 用来获得令命题 $PROP$ 为真时的有限常量集合；
- $eval : PROP \times valueset$ → {$true, false$}用来确定命题$PROP$对于给定的有限值集合是否为真，即：$eval(PROP, valueset) = true$ IFF $valueset \subseteq range(PROP)$；
- $proposition : S \times ident$ → {$p_1, ..., p_m$}用来在状态中通过标识$ident$获得相应系统命题。需要注意由于命题中标识的层次关系，因此在 $proposition$ 关系中需要能够根据一个标识的前缀获得所有命题的集合，即：

$proposition(S, ident) = p$　　若 $\exists p = (ident, set) \in S$

$proposition(S, *) = \{p_1, ..., p_k\}$　　若 $\{p_1, ..., p_k\} = S$

$proposition(S, ident.*) = \{p_1, ..., p_m\}$　　若 $\{p_1, ..., p_m\} \subset S$

　　则 $\forall p \in \{p_1, ..., p_m\}, \exists\ ident'$ s.t. $suffix(p, ident) = ident'$

$proposition(S, *.ident) = \{p_1, ..., p_n\}$　　若 $\{p_1, ..., p_n\} \subset S$

　　则 $\forall p \in \{p_1, ..., p_n\}, \exists\ ident'$ s.t. $prefix(p, ident) = ident'$

2.2.2　状态π演算的语法和状态操作

以上先给出了系统状态和命题的静态定义与关联。对于状态π演算，一个关键任务是要确立系统动作的动态执行语义如何对系统状态的创建、消亡与更新实现管理。这是通过其状态操作符实现的。与基本π演算不同，图 2.1 中给出了状态π演算的完整语法。状态π演算的基本思想是通过状态操作符将状态与命题的管理和π演算中的动作建立动态关联，通过状态操作符的自身语义和原有进程操作语义的融合将状态的生命周期管理与进程的演化相关联。因此，在状态π演算中实现了一个新的状态视图与原有行为视图间的相互制约与影响，正是通过两者的松耦合实现了对π演算现有性质和结论的扩展和重用。

如图 2.1 所示，在状态π演算中每一个输入和输出动作都可以关联多个状态表达式（*StateExpr*）。状态表达式表达了系统行为对一个状态所进行的正向操作（*StateOp*）。而在原有名字匹配[x=y]基础上，状态π演算也允许了对当前命题

（*Prop*）在有限取值范围（*valueset*）内的真命题判别（*eval*）[①]，从而反过来允许状态命题对进程演化和动作执行的影响。在状态π演算中提供了 4 类不同的状态操作符：状态创建（+），状态消亡（-），状态关联（++），状态撤销（--），其中状态创建操作兼具了状态更新语义。直观的说，每一类状态操作符实现了一种状态关系$\Re$: *SysState* × *StateOp* × *S* → *SysState*，即根据系统当前状态（*SysState*）和待操作的状态对象（*StateOp* 和 *S*）确定新的系统状态。这也使通过名字限定实现对状态中命题的唯一标识成为可能（例如：可以通过 *new context* 操作使以 *context.identity* 为标识的命题表示不同用户请求的特定上下文中该用户的身份属性）。实际上，当一个动作α所关联的{*StateExpr*}为空时，则状态π演算中的动作退化为一个基本π演算中的动作，即$\alpha\{\}.P \equiv \alpha.P$。

$$
\begin{array}{rcl}
P & ::= & \sum_{i=1}^{n} \alpha_i\{StateExp\}.P_i \mid (new\ x)P \mid !P \mid P \mid Q \mid \phi P \mid A(y_1,...,y_n) \mid 0 \\
\alpha_i\{StateExp\} & ::= & \overline{x}<\tilde{y}>\{StateExp\} \mid x(\tilde{y})\{StateExp\} \mid \tau\{StateExp\} \\
\phi & ::= & [x=y] \mid eval(Prop, valueset) \mid \phi \wedge \phi \\
StateExp & ::= & (StateOp, S) \mid StateExp, StateExp \\
StateOp & ::= & + \mid - \mid ++ \mid -- \\
S & ::= & (iden, trueset) \\
iden & ::= & x \mid iden.iden
\end{array}
$$

图 2.1 状态π演算语法

在状态π演算中，系统当前状态 *SysState* 是所有已创建状态的集合，它在状态π演算的进程演化中根据相应的状态操作符而不断更新。因此，状态操作符的语义决定了系统行为与状态间的耦合关系，并对系统状态的生命周期做出管理。在给出各状态操作符的操作语义前，本节先对冲突状态和状态良构性进行定义。

定义 2. 2（无关状态与冲突状态）：称两状态 S_1 和 S_2 是冲突的（记为 $S_1 \lozenge S_2$），若$\exists p_1=(ident_1,\ set_1)\in S1$、$p_2=(ident_2,\ set_2)\in S_2$，s.t. $ident_1=ident_2$。此时称命题 p_1 和 p_2 相重复（记为 $p_1 \lozenge p_2$）；两无冲突状态 S_1 和 S_2 则称为是无关的（记为 $S1 \nabla S2$）。

对于任意两个冲突状态，可进一步定义它们之间的偏序关系。

定义 2. 3（状态偏序和状态等价）：对于 $S_1 \lozenge S_2$，记 $S_1 \prec S_2$ 若$\forall p=(ident,\ set)\in S_1$，$\exists p'=(ident',\ set')\in S_2$，s.t. $ident=ident'$且 $set \subset set'$；记 $S_1=S_2$ 若 $S_1 \prec S_2$ 且 $S_2 \prec S_1$。

由此，称状态 *S* 是良构的，若$\forall\ p_1$，$p_2 \in S$，$p_1 \lozenge p_2$ 不成立。状态的良构性保证了一个状态中的任意两个命题不会存在歧义。因此，状态π演算中的状态操作

① 注意此处只允许有限集合在 *eval* 结果为 true 时的匹配。

也必须都在保证状态良构性的基础上进行。设当前系统状态 $SysState=\{p_{11},...,p_{1n}\}$，状态操作符 *StateOp* 关联的状态为 $S=\{p_{21},...,p_{2m}\}$，则以下分别给出了各状态操作符的详细语义及意义：

1．**状态创建（+）**：定义一个新状态并将其覆盖至系统当前状态中；

$$CREATE \quad \frac{SysState \nabla S}{+S=\{p_{11},...,p_{1n},p_{21},...,p_{2m}\}}$$

$$UPDATE \quad \frac{SysState \lozenge S \quad \exists p_{1i} \in SysState, p \in S \text{ s.t. } p_{1i} \lozenge p}{+S=\{p_{11},...,p_{1i-1},p,p_{1i+1},...,p_{1n},p_{21},...,p_{2m}\}}$$

2．**状态消亡（-）**：将一个状态从系统当前状态中无条件移除；

$$DESTROY_VOID \quad \frac{SysState \nabla S}{-S=\{p_{11},...,p_{1n}\}}$$

$$DESTROY \quad \frac{SysState \lozenge S \quad \exists p_{1i} \in SysState, p \in S \text{ s.t. } p_{1i} \lozenge p}{-S=\{p_{11},...,p_{1i-1},p_{1i+1},...,p_{1n}\}}$$

3．**状态关联（++）**：将一个状态中的命题集插入到系统当前状态中；

$$INSERT_VOID \quad \frac{SysState \nabla S}{++S=\{p_{11},...,p_{1n},p_{21},...,p_{2m}\}}$$

$$INSERT \quad \frac{SysState \lozenge S \quad \exists p_{1i} =\in SysState, p \in S \text{ s.t. } p_{1i} \lozenge p}{++S=\{p_{11},...,p_{1i-1},p_{1i}',p_{1i+1},...,p_{1n},p_{21},...,p_{2m}\} \quad range(p_{1i}')=range(p_i)\cup range(p)}$$

4．**状态撤销（- -）**：将一个状态中的命题集从系统当前状态中移除；

$$REMOVE_VOID \quad \frac{SysState \nabla S}{--S=\{p_{11},...,p_{1n}\}}$$

$$REMOVE_SHALLOW \quad \frac{SysState \lozenge S \quad \exists p_{1i} =\in SysState, p \in S \text{ s.t. } p_{1i} \lozenge p \quad range(p_{1i}) / range(p) \neq \phi}{--S=\{p_{11},...,p_{1i-1},p_{1i}',p_{1i+1},...,p_{1n},p_{21},...,p_{2m}\} \quad range(p_{1i}')=range(p_i)/range(p)}$$

$$REMOVE_DEEP \quad \frac{SysState \lozenge S \quad \exists p_{1i} =\in SysState, p \in S \text{ s.t. } p_{1i} \lozenge p \quad range(p_{1i}) / range(p) = \phi}{--S=\{p_{11},...,p_{1i-1},p_{1i+1},...,p_{1n},p_{21},...,p_{2m}\}}$$

正如之前提到的，以上的语义规则形成了实现状态关系$\mathfrak{R}$的基础。即：

$\mathfrak{R}$ *: SysState* × *StateOp* × *S* → *SysState'*　　$\mathfrak{R}$(*SysState*, *StateOp*, *S*)=*SysState'*

其中 *SysState'* 由 *CREATE*、*UPDATE*、*DESTROY*、*INSERT* 和 *REMOVE* 规则所确定。从以下性质可以看出，在以上 9 条语义规则中 *UPDATE* 规则本身可以通过 *DESTROY* 规则和 *CREATE* 规则的组合表示出来。为了方便对状态操作的表示和更新操作，因此在状态π演算中仍保留了对 *UPDATE* 规则的直接表述。

性质 2.1：当∃ $p \in SysState$、$p' \in S$ s.t. $p \lozenge p'$，则 $\mathfrak{R}$(*SysState*，+，*S*) =

$\Re(\Re(SysState, -, S)，+，S)$。

证明：根据状态消亡（-）操作的语义可知：$\Re(SysState, -, S)\nabla S$。因此由 *CREATE* 规则和 *UPDATE* 规则可得等式左右两边均等于$(SysState \cup S)/p$。证毕。　□

性质 2.2：除 *UPDATE* 规则外，以上状态操作语义实现了状态间的偏序。

证明：为证明性质 2.2，可以对状态关系$\Re$的定义分情况讨论：

当 $SysState \nabla S$ 时：　$SysState = \Re(SysState,-,S)$；　$SysState \prec \Re(SysState,++,S)$；$SysState = \Re(SysState,--,S)$。

当 $SysState \lozenge S$ 时：　$SysState \succ \Re(SysState,-,S)$；　$SysState \prec \Re(SysState,++,S)$；$SysState \succ \Re(SysState,--,S)$。证毕。　□

2.3　状态标号迁移系统、同余和扩展操作语义

由于在以上的图 2.1 中已经为状态π演算的动作及其各类状态操作间建立了直接的语法关联。因此，以下进一步确定了上小节中状态操作语义和进程演化间的相互影响和制约。这就是状态π演算的扩展操作语义。虽然π演算的行为通常被解释到一个标号迁移系统（Labeled Transition System）上，但在状态/事件混合系统的建模和推理中，需要扩展一般的标号迁移系统使它能够对系统行为（即：迁移标号）和系统状态（即：状态标号）同时进行建模和关联。因此根据 2.2 小节中的状态结构定义，以下定义了针对状态π演算的状态标号迁移系统来对其行为作出解释。

定义 2.4（状态标号迁移系统）：一状态标号迁移系统$(S, M, \{\xrightarrow{a\{StateExpr\}} \mid a \in M\})$由以下元素组成：$S$ 是一个系统进程和状态的二元组集合，M 是迁移标号的集合，而$\{\xrightarrow{a\{StateExpr\}}\}$是迁移$\xrightarrow{a\{StateExpr\}} \subseteq S \times S$的集合（$a \in M$）。

在状态标号迁移系统中，一迁移记为：$(P, SysState) \xrightarrow{a\{StateExpr\}} (P', SysState')$。它表示：“系统当前进程和状态分别由 P 和 *SysState* 表示，通过执行了一个关联状态表达式 *StateExpr* 的活动 a 后，进程演化为 P’且系统状态更新为 *SysState*’”。以上语义可以由统一方法 *transS* 来严格表示：

$$transS : SysState \times StateExpr \to SysState'$$

$$transS(SysState, StateExpr) = \begin{cases} \Re(SysState, Op, S) & StateExpr = (Op, S) \\ SysState & StateExpr\text{为空} \end{cases}$$

由此，在状态π演算的同余关系定义中，称(P, S)和(P', S')是同余的（记为：$(P, S) \equiv (P', S')$），若进程 P 和 P’满足基本π演算的结构同余关系，且对应有状态

S=*S*'成立。以下本节基于π演算的早迁移语义给出了状态π演算的扩展操作语义（见图 2.2）。其中，用*φ*和*δ*分别简化表示 *StateExpr* 和 *SysState*；*fn* 和 *bn* 则分别用来表示所有自由名和受限名的集合②。

$$OUT \quad \frac{}{(\bar{x}<y>\{\varphi\}.P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P,transS(\delta,\varphi))} \qquad INP \quad \frac{}{(x(z)\{\varphi\}.P,\delta)\xrightarrow{x(y)\{\varphi\}}\{y/z\}(P,transS(\delta,\varphi))}$$

$$TAU \quad \frac{}{(\tau\{\varphi\}.P,\delta)\xrightarrow{\tau\{\varphi\}}(P,transS(\delta,\varphi))} \qquad SUM-L \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(P+Q,\delta)\xrightarrow{\alpha}(P',\delta')}$$

$$MAT \quad \frac{(\alpha.P,\delta)\xrightarrow{\alpha}(P',\delta')}{([\phi]\alpha.P,\delta)\xrightarrow{\alpha}(P',\delta')} \qquad \phi表示\ x=x\ 或\ eval(Prop,valueset)=true$$

$$PAR-L \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(P\mid Q,\delta)\xrightarrow{\alpha}(P'\mid Q,\delta')} \qquad bn(\alpha)\cap fn(Q)=\varnothing$$

$$COMM-L \quad \frac{(P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P',transS(\delta,\varphi))\ \ (Q,\delta')\xrightarrow{x(y)\{\varphi'\}}(Q',transS(\delta',\varphi'))}{(P\mid Q,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}(P'\mid Q',transS(transS(\delta'',\varphi),\varphi'))}$$

$$CLOSE-L \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi))\ \ (Q,\delta')\xrightarrow{x(z)\{\varphi'\}}(Q',transS(\delta',\varphi'))}{(P\mid Q,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}(new\ z)(P'\mid Q',transS(transS(\delta'',\varphi),\varphi'))} \qquad z\notin fn(Q)$$

$$RES \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{((new\ z)P,\delta)\xrightarrow{\alpha}((new\ z)P',\delta')} \qquad z\notin n(\alpha)$$

$$OPEN \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi))}{((new\ z)P,\delta')\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta',\varphi))} \qquad z\neq x$$

$$REP-ACT \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(!P,\delta)\xrightarrow{\alpha}(P'\mid !P,\delta')}$$

$$REP-COMM \quad \frac{(P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P',transS(\delta,\varphi))\ \ (P,\delta')\xrightarrow{x(y)\{\varphi'\}}(P'',transS(\delta',\varphi'))}{(!P,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}((P'\mid P'')\mid !P,transS(transS(\delta'',\varphi),\varphi'))}$$

$$REP-CLOSE \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi))\ \ (P,\delta')\xrightarrow{x(z)\{\varphi'\}}(P'',transS(\delta',\varphi'))}{(!P,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}((new\ z)(P'\mid P'')\mid !P,transS(transS(\delta'',\varphi),\varphi'))} \qquad z\notin fn(P)$$

图 2.2 状态π演算扩展操作语义

② 为表示方便，此处用 *y*、*z* 统一表示了单项传值或多项传值 $\tilde{y}$、$\tilde{z}$。下同。

在图 2.2 中，其 *OUT*、*INP* 和 *TAU* 语义中分别展示了在进程通过相应输出、输入和不可见动作做出演化时对当前系统状态所进行的相应操作 *transS*。而在 *COMM-L* 和 *CLOSE-L* 中则详细规定了在通过输入输出动作进行进程同步时，输出动作对当前系统状态的操作将优先于输入动作对当前系统状态的操作。另一方面，*MAT* 语义中则实现了通过当前系统状态的真命题判别来逆向影响进程演化和动作执行。需要指出，通过图 2.2 的状态π演算扩展操作语义表明了状态的操作仅通过进程中所关联的动作进行触发，而不受到重命名的影响。这带来的好处是状态的更新可以作为与系统行为并行的维度进行管理。相比之下由于原有π演算的迁移语义中只定义了系统如何通过行为的交互进行演化，而没有显示地表明系统状态所产生的变化，因此，图 2.2 语义为系统中状态和行为的关联以及对状态的管理做出了进一步的扩展和细化。

作为示例，图 2.3 中给出了基于状态π演算对一服务 *A* 简单执行过程的形式化结果即其对应状态变化。它描述并记录了服务从未开始（*NotStarted*），到通过其输入端口 $port_1$ 进行 *StageIn*，到开始执行（*Active*），到通过其输出端口 $port_2$ 进行 *StageOut* 的简单过程。

$$SimpleServiceA(port_1, port_2, execute, ack) =_{def} \tau\{+\{A.Status, \{NotStarted\}\}\}.$$
$$(port_1(v)\{+\{A.Status, \{StageIn\}\}\}.\overline{execute} < v > \{+\{A.Status, \{Active\}\}\} \mid$$
$$ack(res).\overline{port_2} < res > \{+\{A.Status, \{StageOut\}\}\})$$

图 2.3 简单服务执行过程的状态π演算语义

在下一章中将基于状态π演算给出完整的网格服务流形式化语义。

2.4　状态互模拟关系

互模拟分析是进程代数中检验系统等价性的重要方法，并在服务流的验证分析中常被用于对服务可替换性的等价判别。虽然本文后续基于状态π演算的网格服务流验证工作中并没有直接利用互模拟分析的相关性质（本文研究重点是采用状态断言和模型验证的业务逻辑性质验证，见第 4 章），但互模拟关系仍然是保障本文状态π演算工具完整性，加深其语义理解所不可或缺的重要环节。实

际上，互模拟分析和本文结合的模型验证虽然是两种不同的分析手段，但它们之间也存在联系：若将互模拟分析视为对两系统完整行为的等价关系判别，则模型验证亦可视为对两系统局部行为的包含关系判别[57]。因此，本节将对状态π演算的不同互模拟关系做出定义，并给出其关键的性质分析。

在基本π演算中通过各类强/弱互模拟关系已经定义了对系统的（可观测）行为等价。然而在一个状态/事件混合模型中，还可以分别对系统状态的变化予以考虑。记$\Rightarrow^{\tau}$为一通过不可见动作τ所触发的迁移序列（长度可以为 0）。用$\Rightarrow^{a}$表示$\Rightarrow\xrightarrow{a}\Rightarrow$（以下省去了对动作 a 所关联状态表达式的表示）；用$\Rightarrow^{\hat{a}}$分别表示$\Rightarrow^{a}$（当$\hat{a}\neq\tau$）和$\Rightarrow^{\tau}$（当$\hat{a}=\tau$）；$\Rightarrow$则表示通过任意动作触发的迁移序列（长度可以为 0）。在这里混合互模拟关系就是针对状态π演算中结合系统行为与状态交互的互模拟关系。

定义2. 5（强混合互模拟）：一定义在状态π演算进程上的对称二元关系R是一强混合互模拟关系，当对于$(P, SysState_P)R(Q, SysState_Q)$和任意替换$\sigma$：

若$(P_\sigma, SysState_P)\xrightarrow{a}(P', SysState_P')$ $(bn(a)\notin fn(P_\sigma, Q_\sigma))$，$\exists Q'$ s.t. $(Q_\sigma, SysState_Q)\xrightarrow{a}(Q', SysState_Q')$，$(P', SysState_P')R(Q', SysState_Q')$且 $SysState_P'= SysState_Q'$。

定义2. 6（弱混合互模拟）：一定义在状态π演算进程上的对称二元关系R是一弱混合互模拟关系，当对于$(P, SysState_P)R(Q, SysState_Q)$和任意替换$\sigma$：

若$(P_\sigma, SysState_P)\xrightarrow{a}(P', SysState_P')$ $(bn(a)\notin fn(P_\sigma, Q_\sigma))$，$\exists Q'$ s.t. $(Q_\sigma, SysState_Q)\Rightarrow^{\hat{a}}(Q', SysState_Q')$，$(P', SysState_P')R(Q', SysState_Q')$且 $SysState_P'\prec SysState_Q'$。

此外作为对系统描述的一个独立维度，也可单独以系统状态为主线观察系统间的等价性。实际上，由于通过系统状态可以更灵活地抽象出所关心的系统业务信息，因此以这种方式对系统等价性进行定义是十分实用的。

定义2. 7（状态模拟）：一定义在状态π演算进程上的二元关系R是一状态模拟关系，当对于$(P, SysState_P)R(Q, SysState_Q)$ 和任意替换σ，若$(P_\sigma, SysState_P)\xrightarrow{a}(P', SysState_P')$ $(bn(a)\notin fn(P_\sigma, Q_\sigma))$，$\exists Q'$ s.t. $(Q_\sigma, SysState_Q)\Rightarrow(Q', SysState_Q')$且 $SysState_P'\prec SysState_Q'$。

定义 2. 8（状态互模拟）：一定义在状态π演算进程上的二元关系 R 是一个状态互模拟关系，当 R 和它的逆 R^{-}均为状态模拟关系。

以下总结了状态π演算中互模拟关系的两个简单性质。

性质 2.3：若进程 P 和 Q 是混合互模拟的，则它们必为状态互模拟的。

证明：通过混合互模拟与状态互模拟的定义直接得到。证毕。　□

性质 2.3 说明状态互模拟是比混合互模拟更松弛的一种互模拟关系。实际上从状态模拟的定义可以看出，它不关心进程演化时所发生过的具体动作迁移（用任意动作的迁移序列$\Rightarrow$代替了对指定动作迁移$\xrightarrow{a}$的要求），而只关心在动作迁移后所达到的新状态 *SysState* 间是否存在偏序关系。因此若以状态来表示系统中关心的实际业务信息，则状态模拟关系实现了与这些业务状态无关动作的灵活抽象。特别的，性质 2.3 的逆命题不成立，即若两进程 P 和 Q 状态互模拟且同时强/弱互模拟，则它们未必强/弱混合互模拟。图 2.4 给出了它的一个直接反例。

$$P(x,a,b) =_{def} \overline{x}\{+,\{Srv,\{X\}\}\}.(port_a\{++,\{Srv,\{A\}\}\}.0 + port_b\{++,\{Srv,\{B\}\}\}.0)$$
$$Q(x,a,b) =_{def} \overline{x}\{+,\{Srv,\{X\}\}\}.(port_a\{++,\{Srv,\{B\}\}\}.0 + port_b\{++,\{Srv,\{A\}\}\}.0)$$

图 2.4 状态互模拟且同时强/弱互模拟，但不是强/弱混合互模拟的反例

以下的性质 2. 4 证明了两进程的并发组合与顺序组合间的状态模拟关系。

性质 2. 4：称两状态π演算进程是互相独立的，当它们间不存在相同的输入/输出端口进行交互。则若两个进程 P 和 Q 是独立的，有($P|Q$, *SysState*)状态模拟($P;Q$, *SysState*)成立，反之不然。其中“;”是文献[144]中所定义的进程顺序组合。

证明：由于 P 和 Q 是独立的，因此 $P|Q$ 的迁移序列 TS 是 P 和 Q 各自迁移序列的叉积。因此，设 $P;Q$ 的所有可能迁移序列为 TS’，则 TS’$\subseteq$TS。另外，由于两进程初始时系统状态均为 *SysState*，因此 TS’所对应的二元关系 R 即为（$P|Q$ 到 $P;Q$）的状态模拟关系。而 R^{-}则不是。证毕。 □

2. 5　状态π演算的类型约束

以上状态π演算中的进程通讯仍然只依靠简单的端口名匹配，而无法在进程组合中将如网格服务交互间的执行条件、数据结构和数据类型约束等重要约束信息考虑进来。由于以上信息对于服务的映射与选择有着重要关系，因此本节中将进一步讨论状态π演算中此类类型约束的实现。

2.5.1　数据类型结构的静态语义

为了对服务交互时的交互条件、数据类型和数据结构约束做出描述，以下首先定义了包含上述信息的抽象数据类型结构。

定义 2.9（数据类型结构）：一个数据类型结构是一个三元组：*T-STRUCT*=(*T*，$\mathcal{G}$，$\mathcal{A}$)。其中：

- *T* 是 *T-STRUCT* 中所有数据类型 *t* 的抽象标识集合：*T*={ $t_1, t_2, ..., t_m$ }；
- $\mathcal{G}$是该数据类型结构中所有数据类型的继承关系（*Generalization*）集合：
  $\mathcal{G}: t \rightarrow \{t_1, t_2, ..., t_p\}$；
- $\mathcal{A}$是该数据类型结构中所有数据类型的聚合关系（*Composite Aggregation*）及其数量（[*n*]）的集合：$\mathcal{A}: t[n] \rightarrow \{t_1[n_1], t_2[n_2], ..., t_q[n_q]\}$。

以上定义中，记 $t' \leftrightarrow t''$表示数据类型 *t''*继承 *t'*，即$\exists$ $\{t_1, t_2, ..., t_p\}$，s.t. $t'' \in \mathcal{G}(t_p)$，$t_p \in \mathcal{G}(t_{p-1})$，...，$t_2 \in \mathcal{G}(t_1)$，$t_1 \in \mathcal{G}(t')$；类似的，记 $t' \rightarrow t''[n]$表示通过 *n* 份数据类型 *t''*可以部分实现数据类型 *t'*，即$\exists$ { $t_1, t_2, ..., t_q$}，s.t. $t''[n] \in \mathcal{A}(t_q[n_q])$，$t_q[n_q] \in \mathcal{A}(t_{q-1}[n_{q-1}])$，...，$t_2[n_2] \in \mathcal{A}(t_1[n_1])$，$t_1[n_1] \in \mathcal{A}(t'[1])$。

由此，一个数据类型结构 *T-STRUCT* 代表了一个系统中所有数据类型及其结构关系的一种实现。针对数据类型结构的良构性，必须防止它们出现循环继承或聚合，以从语法上确保它们是正确的。因此，称一个数据类型结构 *T-STRUCT*={*T*，$\mathcal{G}$，$\mathcal{A}$}是良构的，当$\nexists$ $\{t_1, t_2\} \subset T$, s.t. （1）$t_1 \leftrightarrow t_2$和 $t_2 \leftrightarrow t_1$同时成立；（2）$t_1 \rightarrow t_2$ [*n*]和 $t_2 \rightarrow t_1$ [*m*]同时成立，其中 *n*，*m* 为任意自然数。以下的类型约束均基于良构的数据类型结构来讨论的。

2.5.2　带类型约束的状态π演算操作语义

由于状态π演算进程是实现本文中网格服务流形式化语义的基础，因此有必要使状态π演算的进程交互语义同样能遵守网格服务交互中可能存在的执行条件、数据结构和数据类型约束。

与 2.2.2 小节中的语法相比，在考虑类型约束的条件下状态π演算的一个基本输入动作定义为：$x(\widetilde{y:t}; s(PRE))$。它表示当前置条件 *s*(*PRE*)满足时，可以通过输入端口 *x* 接收类型属于 $\tilde{t}$ 的数据集合，并保存在 $\tilde{y}$ 中。这里的 $\widetilde{y:t}$ 是 $\{y_1 : t_1, y_2 : t_2, ..., y_m : t_m\}$的简写；一个基本输出动作定义为：$\bar{x} < \widetilde{y:t}; s(POST) >$。它表示当后置条件 *s*(*POST*)满足时，可以通过输出端口 *x* 发送类型属于 $\tilde{t}$ 的数据集合 $\tilde{y}$。其中，布尔型的前后置条件 *s*(*PRE*)和 *s*(*POST*)可分别用来表达当前网格

自身环境信息的判别，在缺省情况下它们均为真。*s*(*PRE*)和 *s*(*POST*)定义的语法规范可参见以下的图 2.5：

$$
\begin{array}{rl}
PRE & ::= condition \\
POST & ::= condition \\
condition & ::= attr = atom \mid attr > atom \mid attr < atom \mid attr >= atom \mid attr <= atom \mid \\
 & \quad attr \subset atomset \mid attr \supset atomset \mid condition \ \wedge \ condition \mid \\
 & \quad condition \ \vee \ condition \mid \neg condition \mid (condition) \mid TRUE \mid FALSE \\
attr & ::= taskID \mid sessionID \mid timeStamp \mid srvID \mid range(PROP) \mid proposition(PROP, iden) \\
atom & ::= symbol \mid number \\
atomset & ::= (atom)+
\end{array}
$$

图 2.5　*s*(*PRE*)和 *s*(*POST*)的语法规范

图 2.5 中的属性（*attr*）定义可包含任务 ID（*taskID*）、Web 会话 ID（*sessionID*）、时戳（*timestamp*）、服务 ID（*srvID*）等基本信息及状态π演算自身对用户自定义命题所拥有的 *range* 和 *proposition* 判别（见 2.2.1 小节）。

$$
\begin{array}{l}
\overline{port80} < gvdata' : soapmes > \ \mid \\
port80(gvdata : soapmes; s(proposition(SysState, context) \supset \{context.source\} \ \wedge \\
\qquad\qquad range(storage) = 0))
\end{array}
$$

例如：以上进程的例子中描述了通过端口 80 发出一请求引力波（Gravitational Wave）探测数据的 SOAP 消息（*gvdata*:*soapmes*）。但处理该消息的前提是系统中需已提供该数据的来源（*context.source*），且此时待分析数据的存储器容量已为空 *s*(*range*(*storage*) = 0)。此处的 *storage* 和 *context* 均为系统当前状态中的命题，由此通过 *range* 和 *proposition* 实现了对状态π演算状态命题范围和结构的约束。

由此，考虑类型约束的状态π演算完整语法如图 2.6 所示：

$$
\begin{array}{rl}
P & ::= \sum_{i=1}^{n} \alpha_i \{StateExpr\}.P_i \mid (new\ x)P \mid !P \mid P \mid Q \mid \phi P \mid A(y_1, ..., y_n) \mid 0 \\
\alpha_i\{StateExpr\} & ::= \overline{x} < \widetilde{y : t}; s(POST) > \{StateExpr\} \mid x(\widetilde{y : t}; s(PRE))\{StateExpr\} \mid \tau\{StateExpr\} \\
\phi & ::= [x = y] \mid eval(Prop, valueset) \mid \phi \wedge \phi \\
StateExp & ::= (StateOp, S) \mid StateExp, StateExp \\
StateOp & ::= + \mid - \mid ++ \mid -- \\
S & ::= (iden, trueset) \\
iden & ::= x \mid iden.iden
\end{array}
$$

图 2.6　考虑类型约束的状态π演算语法

若记数据类型结构 Γ = *T-STRUCT* = (*T*，$\mathcal{G}$，$\mathcal{A}$)为特定的网格类型环境，则图

2.7 给出了基于状态π演算在带数据类型约束时的扩展操作语义。其中，$\Gamma \triangleright P$ 表示一个带数据类型约束的状态π演算进程 P 在网格类型环境Γ下的行为。

$$T_OUT \quad \frac{t \in T \qquad s(POST) = true}{\Gamma \triangleright (\bar{x} < y : t; s(POST) > \{\varphi\}.P, \delta) \xrightarrow{\bar{x}<y:t;s(POST)>\{\varphi\}} \Gamma \triangleright (P, transS(\delta, \varphi))}$$

$$T_IN \quad \frac{t \in T \qquad s(PRE) = true}{\Gamma \triangleright (x(z : t; s(PRE))\{\varphi\}.P, \delta) \xrightarrow{x(y:t;s(PRE))\{\varphi\}} \Gamma \triangleright \{y/z\}(P, transS(\delta, \varphi))}$$

$$T_TAU \quad \frac{}{\Gamma \triangleright (\tau\{\varphi\}.P, \delta) \xrightarrow{\tau\{\varphi\}} \Gamma \triangleright (P, transS(\delta, \varphi))} \qquad T_SUM_L \quad \frac{\Gamma \triangleright (P, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P', \delta')}{\Gamma \triangleright (P + Q, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P', \delta')}$$

$$T_MAT \quad \frac{\Gamma \triangleright (\alpha.P, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P', \delta')}{\Gamma \triangleright ([\phi]\alpha.P, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P', \delta')} \qquad \phi\text{表示 } x = x \text{ 或 } eval(Prop, valueset) = true$$

$$T_PAR_L \quad \frac{\Gamma \triangleright (P, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P', \delta')}{\Gamma \triangleright (P \mid Q, \delta) \xrightarrow{\alpha} \Gamma \triangleright (P' \mid Q, \delta')} \qquad bn(\alpha) \cap fn(Q) = \varnothing$$

$$T_COMM_L \quad \frac{\begin{array}{l}\Gamma \triangleright (P, \delta) \xrightarrow{\bar{x}<y:t_1;s(POST)>\{\varphi\}} (P', transS(\delta, \varphi)) \qquad s(POST) \to s(PRE) \\ \Gamma \triangleright (Q, \delta') \xrightarrow{x(y:t_2;s(PRE))\{\varphi'\}} (Q', transS(\delta', \varphi')) \qquad t_1 = t_2 \quad t_1, t_2 \in T\end{array}}{\Gamma \triangleright (P \mid Q, \delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} \Gamma \triangleright (P' \mid Q', transS(transS(\delta'', \varphi), \varphi'))}$$

$$T_COMM_GEN \quad \frac{\begin{array}{l}\Gamma \triangleright (P, \delta) \xrightarrow{\bar{x}<y:t_1;s(POST)>\{\varphi\}} (P', transS(\delta, \varphi)) \qquad s(POST) \to s(PRE) \\ \Gamma \triangleright (Q, \delta') \xrightarrow{x(y:t_2;s(PRE))\{\varphi'\}} (Q', transS(\delta', \varphi')) \qquad \neg(t_1 = t_2) \wedge (t_1 \leftrightarrow t_2) \quad t_1, t_2 \in T\end{array}}{\Gamma \triangleright (P \mid Q, \delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} \Gamma \triangleright (P' \mid Q', transS(transS(\delta'', \varphi), \varphi'))}$$

$$T_COMM_AGG_OUT \quad \frac{\begin{array}{l}\Gamma \triangleright (P, \delta) \xrightarrow{\bar{x}<y:t_1;s(POST)>\{\varphi\}} (P', transS(\delta, \varphi)) \qquad s(POST) \to s(PRE) \\ \Gamma \triangleright (Q, \delta') \xrightarrow{x(y:t_2;s(PRE))\{\varphi'\}} (Q', transS(\delta', \varphi')) \qquad t_1, t_2, t' \in T \\ \neg(t_1 = t_2) \wedge \neg(t_1 \leftrightarrow t_2) \wedge (t_2 \to t_1'\{n\}) \text{且} (t_1' = t_1 \vee t_1 \leftrightarrow t_1')\end{array}}{\Gamma \triangleright (P \mid Q, \delta'') \longrightarrow \Gamma \triangleright (P \mid Q'', \delta'') \qquad Q'' = Decompose(Q)}$$

$$T_COMM_AGG_IN \quad \frac{\begin{array}{l}\Gamma \triangleright (P, \delta) \xrightarrow{\bar{x}^u<y:t_1;s(POST)>\{\varphi\}} (P', transS(\delta, \varphi)) \qquad s(POST) \to s(PRE) \\ \Gamma \triangleright (Q, \delta') \xrightarrow{x^Y(y:t_2;s(PRE))\{\varphi'\}} (Q', transS(\delta', \varphi')) \qquad t_1, t_2, t' \in T \\ \neg(t_1 = t_2) \wedge \neg(t_1 \leftrightarrow t_2) \wedge (t_1 \to t_2'\{n\}) \text{且} (t_2' = t_2 \,||\, t_2' \leftrightarrow t_2)\end{array}}{\Gamma \triangleright (P \mid Q, \delta'') \longrightarrow \Gamma \triangleright (P'' \mid Q, \delta'') \qquad P'' = Decompose(P)}$$

图 2.7　带类型约束的扩展操作语义

$$T_CLOSE_L \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<zt_1;s(POST)>\{\varphi\}} (P', transState(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (Q,\delta') \xrightarrow{x(zt_2;s(PRE))\{\varphi'\}} (Q', transState(\delta',\varphi')) \quad t_1 = t_2 \quad t_1,t_2 \in T}{\Gamma \rhd (P \mid Q,\delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} \Gamma \rhd (new\ z)(P' \mid Q', transS(transS(\delta'',\varphi),\varphi'))} \quad z \notin fn(Q)$$

$$T_CLOSE_GEN \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<zt_1;s(POST)>\{\varphi\}} (P', transS(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (Q,\delta') \xrightarrow{x(zt_2;s(PRE))\{\varphi'\}} (Q', transS(\delta',\varphi')) \quad \neg(t_1 = t_2) \wedge (t_1 \leftrightarrow t_2) \quad t_1,t_2 \in T}{\Gamma \rhd (P \mid Q,\delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} \Gamma \rhd (new\ z)(P' \mid Q', transS(transS(\delta'',\varphi),\varphi'))} \quad z \notin fn(Q)$$

$$T_CLOSE_AGG_OUT \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<zt_1;s(POST)>\{\varphi\}} (P', transS(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (Q,\delta') \xrightarrow{x(zt_2;s(PRE))\{\varphi'\}} (Q', transS(\delta',\varphi')) \quad t_1,t_2,t' \in T \quad \neg(t_1 = t_2) \wedge \neg(t_1 \leftrightarrow t_2) \wedge (t_2 \to t_1'\{n\}) \text{且} (t_1' = t_1 \vee t_1 \leftrightarrow t_1')}{\Gamma \rhd (P \mid Q,\delta'') \longrightarrow \Gamma \rhd (P \mid Q'',\delta'') \quad Q'' = Decompose(Q)} \quad z \notin fn(Q)$$

$$T_CLOSE_AGG_IN \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<zt_1;s(POST)>\{\varphi\}} (P', transS(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (Q,\delta') \xrightarrow{x(zt_2;s(PRE))\{\varphi'\}} (Q', transS(\delta',\varphi')) \quad t_1,t_2,t' \in T \quad \neg(t_1 = t_2) \wedge \neg(t_1 \leftrightarrow t_2) \wedge (t_1 \to t_2'\{n\}) \text{且} (t_2' = t_2 \vee t_2' \leftrightarrow t_2)}{\Gamma \rhd (P \mid Q,\delta'') \longrightarrow \Gamma \rhd (P'' \mid Q,\delta'') \quad P'' = Decompose(P)} \quad z \notin fn(Q)$$

$$T_RES \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\alpha} \Gamma \rhd (P',\delta')}{\Gamma \rhd ((new\ z)P,\delta) \xrightarrow{\alpha} \Gamma \rhd ((new\ z)P',\delta')} \quad z \notin n(\alpha)$$

$$T_OPEN \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<z:t;s(POST)>\{\varphi\}} \Gamma \rhd (P', transS(\delta,\varphi)) \quad t \in T}{\Gamma \rhd ((new\ z)P,\delta') \xrightarrow{\bar{x}<z:t;s(POST)>\{\varphi\}} \Gamma \rhd (P', transS(\delta',\varphi))} \quad z \neq x$$

$$T_REP_ACT \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\alpha} \Gamma \rhd (P',\delta')}{\Gamma \rhd (!P,\delta) \xrightarrow{\alpha} \Gamma \rhd (P' \mid !P,\delta')}$$

$$T_REP_COMM \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<yt;s(POST)>\{\varphi\}} \Gamma \rhd (P', transS(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (P,\delta') \xrightarrow{x(yt;s(PRE))\{\varphi'\}} \Gamma \rhd (P'', transS(\delta',\varphi')) \quad t \in T}{\Gamma \rhd (!P,\delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} ((P' \mid P'') \mid !P, transS(transS(\delta'',\varphi),\varphi'))}$$

$$T_REP_CLOSE \quad \frac{\Gamma \rhd (P,\delta) \xrightarrow{\bar{x}<yt;s(POST)>\{\varphi\}} (P', transS(\delta,\varphi)) \quad s(POST) \to s(PRE) \quad \Gamma \rhd (P,\delta') \xrightarrow{x(yt;s(PRE))\{\varphi'\}} (P'', transS(\delta',\varphi')) \quad t \in T}{\Gamma \rhd (!P,\delta'') \xrightarrow{\tau\{\varphi,\varphi'\}} ((new\ z)(P' \mid P'') \mid !P, transS(transS(\delta'',\varphi),\varphi'))} \quad z \notin fn(P)$$

图 2.7（续）　带类型约束的扩展操作语义

图 2.7 中主要的新增语义体现在 4 条 *T-COMM* 和 4 条 *CLOSE-GEN* 上[③]。它

③ 类似的，针对 *T-REP-COMM* 和 *T-REP-CLOSE* 的相似语义则不在此重复给出。

们表示了在网格类型环境Γ下状态π演算的进程交互约束和前提条件。其中，*T_COMM_L*（和 *T_CLOSE_L*）强调进程在输入输出同步时，需要满足以下约束：

1）端口名相匹配；

2）端口中传输的数据项数相等；

3）各数据的类型相匹配；

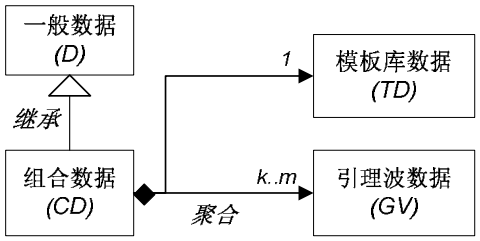
4）输出动作的后置条件是输入动作前置条件的松弛 $s(POST)\rightarrow s(PRE)$。

T_COMM_GEN_OUT（和 *T_CLOSE_GEN_OUT*）则针对类型的继承关系对条件（3）做出了补充，即当数据类型不匹配时，则输出数据类型须继承输入数据类型 $\neg(t_1=t_2)\wedge(t_1\leftrightarrow t_2)$ 以满足进程交互约束。*T_COMM_GEN_OUT*（和 *T_CLOSE_GEN_OUT*）则进一步补充了在类型聚合时的语义，即若输入数据类型和输出数据类型的聚合关系能得到匹配，则相应进程 Q 的待输出数据将被分解为其聚合子类型 *Decompose*(Q)，以满足进程的交互（*T-COMM-GEN-IN* 和 *T-CLOSE-GEN-IN* 是其对应的输入版本）。*Decompose*(P)由聚合语义实现如下：

若 $P=\bar{x}<y:t;s(POST)>.P'$，　$t[1]\rightarrow\{t_1[n_1], t_2[n_2], ..., t_q[n_q]\}\in\mathcal{A}$，则：

$$Decompose(P)=new\ ack(\underbrace{\bar{x}<y:t_1,s(post)>.\ldots.\bar{x}<y:t_1,s(post)>}_{n_1}.\overline{ack}\mid$$
$$\underbrace{\bar{x}<y:t_2,s(post)>.\ldots.\bar{x}<y:t_2,s(post)>}_{n_2}.\overline{ack}\mid.....\mid$$
$$\underbrace{\bar{x}<y:t_q,s(post)>.\ldots.\bar{x}<y:t_q,s(post)>}_{n_q}.\overline{ack}\mid\underbrace{ack.ack....ack}_{q}.P')$$

以 LIGO 引力波探测数据的分析[27,49,50]为例，设一个“组合数据”类型（*CD*）继承了一个“一般数据”类型。一个待分析的“组合数据”（*CD*）由一个用于校验的“模板库数据”（*TD*）和多个（*k..m*）实际探测得到的“引力波数据”（*GV*）组成。则对于待输出“组合数据”的进程 P，*Decompose*(P)将根据该聚合关系不仅试图发送一份“模板库数据”（$\overline{data_{TD}}<d_{TD}:t_{TD},s(post)>$），且会至少发送 k 份“引力波数据”，并在之后尝试发出剩余的 m-k 份“引力波数据”。



记CD，TD和GV分别为组合数据、模板库数据和引力波数据的缩写

且此时 $P=\overline{data_{CD}}<d_{CD}:t_{CD},s(post)>.P'$

则 $Decompose(P)=new\ ack(\overline{data_{TD}}<d_{TD}:t_{TD},s(post)>.\overline{ack}\,|\underbrace{\overline{data_{GV}}<d_{GV}:t_{GV},s(post)>.\ldots.\overline{data_{GV}}<d_{GV}:t_{GV},s(post)>}_{k次}.\overline{ack}.\underbrace{\overline{data_{GV}}<d_{GV}:t_{GV},s(post)>.\ldots.\overline{data_{GV}}<d_{GV}:t_{GV},s(post)>}_{m-k次}|ack.ack.P')\quad(m\geq k)$

以上操作语义的重要特点在于它给出了针对给定网格类型环境Γ = *T-STRUCT* = (T，$\mathcal{G}$，$\mathcal{A}$)下，由状态π演算进程代表的网格服务流在服务交互过程中，若存在数据类型约束时所应满足的完整交互行为约束，即：

（1）输入输出端口名相匹配，且传输的数据项数相等；

（2）传输数据的类型是合法的 $t\in T$；

（3）输入/出动作的前/后置条件能被满足，且同步交互时输出动作的后置条件是输入动作前置条件的松弛 $s(POST)\rightarrow s(PRE)$；

（4）输入输出数据类型（t_1 和 t_2）的匹配，即满足以下互斥条件之一：

- $t_1=t_2$；
- $\neg(t_1=t_2)\wedge(t_1\leftrightarrow t_2)$；
- $\neg(t_1=t_2)\wedge\neg(t_1\leftrightarrow t_2)\wedge(t_2\rightarrow t_1'\{n\})\wedge(t_1'=t_1\ \vee\ t_1\leftrightarrow t_1')$；
- $\neg(t_1=t_2)\wedge\neg(t_1\leftrightarrow t_2)\wedge(t_1\rightarrow t_2'\{n\})\wedge(t_2'=t_2\ \vee\ t_2'\leftrightarrow t_2)$。

通过以上约束，则可以检验特定的网格类型环境对于由状态π演算进程所表示的服务流交互过程在实现服务可达性与可终止性等结构特性和规范约束语义时产生的影响。本文将在第 4 章通过提出状态断言的方法对该问题进行定义和判别。而在下一章中将给出基于状态π演算的网格服务流完整形式化语义。

2.6　小结

本章的主要贡献在于：基于 Web 服务资源框架（WSRF）对有状态资源的扩展特点，提出了新的状态π演算形式化工具，实现了进程演化和状态管理间的相互作用与约束。本章定义和分析了状态π演算的语法、状态命题结构、状态操作语义、进程的扩展操作语义和状态互模拟关系，并为其进一步设计了服务协

作时的交互条件、数据类型和数据结构约束语义。

本章的工作着眼于形式化理论工具自身的扩展完善。由于状态π演算弥补了基本π演算对状态获取和管理能力缺乏的不足，并为动作的执行和交互提供了灵活的业务含义抽象能力，在后续章节工作中可以发现它为本文中网格服务流的形式化建模（第三章）、验证（第四章）和性能提高（第五章）奠定了理论工具基础，并简化了对服务流的建模和业务逻辑性质的描述。

第 3 章　基于状态π演算的网格服务流形式化语义

3.1　本章引论

本章基于上一章的状态π演算给出了网格服务流的三部分重要形式化语义。正如 1.4 小节的综述所述，建立网格服务流规范形式化语义的重要性已经被广泛接受。它有利于澄清规范中的模糊和歧义信息，区分不同规范在模型元素上的细微语义差别，为它们提供严格的数学描述。更重要的是，它还为实现相应的形式化验证方法以发现网格服务流中潜在的隐含问题奠定了重要的基础。

考虑到网格服务流模型的代表性和在实际应用中的有效性，本章以上一章中的状态π演算为基础，分别对常用的 Condor DAGMan，BPEL4WS（1.1）规范和 JOpera 中所提出的网格服务流模式[23]进行了形式化语义的建模。由于目前对网格服务流中的常用模式等仍然缺乏明确的形式化语义支持，因此本章工作可以有效弥补这一不足。另一方面，以上形式化语义的完整建立过程同时也是对本文状态π演算自身表达能力的一种有效验证。

本章中形式化对象的选择是出于以下的考虑：

- 规范的代表性：DAGMan 是目前以有向无环图方式实现网格服务流的典型工具；BPEL4WS 不仅是 Web 服务组合的事实标准，在网格服务流领域也有着面向 OGSI 和 WSRF 规范的扩展[71,72]；而网格服务流模式则归类了网格服务流中常用的并发与管道执行模式，为相应规范和引擎的实现提供了参照。
- 实际的应用背景和广泛的业界支持：包括 Pegasus 工作流、P-GRADE[145]等均利用 Condor DAGMan 协作其应用的执行，并已在 LIGO[27]等大型网格项目中得到实际应用；而 BPEL4WS 则被包括 CGSP[7,146]在内的网格中间件所扩展利用。且除科研领域外，它在银行等商务领域中也有广泛的应用支持[34]；
- 技术研究与实际系统的支持：以上三者目前都有相应的技术研究[11,146] 和系统实现（如：Condor①，CGSP 和 JOpera[63]）以支持对 Globus Toolkit 4 的集成；例如，Condor 6.8 及其之后版本中不仅提供了对 Globus Toolkit 4 的支持，Condor daemons 自身也正被扩展为一个 WSRF 兼容的网格服务。

基于上一章的状态π演算工具，本章工作组织如下：在 3.2 小节中首先给出

① Condor 6.8.x 之后的版本提供了对 Globus Toolkit 4 的支持。

了服务调用与执行、多实例化、服务选择和消息收发等基本活动的状态π演算形式化语义；在 3.3 和 3.4 小节中则给出了 Condor DAGMan 和 BPEL4WS 1.1 规范中服务协作的控制流结构、错误补偿处理和全局终止等高级特性的形式化语义；3.5 小节给出了完整网格服务流的状态π演算模型示例；3.6 小节中则对网格服务流中的并发和管道模式进行了完整的状态π演算形式化。

通过本章的工作不仅基于有穷状态π演算进程补充了从单一服务模型、服务流协作到相关模式的完整网格服务流形式化语义，也为下一章中的形式化验证方法提供了对象。通过本章结果不仅表明状态π演算自身拥有足够强的服务流描述能力，同时更发现它能够有效简化网格服务流规范中如全局终止等复杂特性的描述和验证时业务逻辑的建模。本章以下内容的相关研究结论主要发表在：

- 网格服务链模型的验证分析技术及应用. *中国科学 F 辑: 信息科学*, 2007, 37(4): 467-485 （**SCI** 检索）

  Formal Verification Technique for Grid Service Chain Model and its Application. *Science in China, Series F: Information Sciences*, 2007, 50(1): 1-20 （**SCI** 检索）
- Pi Calculus based Bi-transformation of State-driven Model and Flow-driven Model. *International Journal of Business Process Integration and Management*, 2006, 1(4): 292 – 306
- Ensuring Secure and Robust Grid Applications – From a Formal Method Point of View. *Advances in Grid and Pervasive Computing, Lecture Notes in Computer Science*, 2006, 3947: 537-546 （**SCI** 检索）
- Study on Pi Calculus Based Equipment Grid Service Chain Model. *Network and Parallel Computing, Lecture Notes in Computer Science*, 2005, 3779: 40-47 （**SCI** 检索）

3.2　服务执行与基本活动的状态π演算形式化

本小节先对一般的网格服务执行与选择，DAGMan 和 BPEL4WS 中的基本活动及变量做出形式化描述。

需要指出的是，网格服务的资源状态是在与实际网格系统的交互中获取的。而本文中状态π演算所实现的状态管理和抽象能力，为各种不同网格服务资源状态的定义和生成提供了一个良好的元数据模型（见 2.2 小节）。因此，在网格服

务及相关活动的形式化语义中，可以通过状态π演算对状态的灵活抽象能力来根据实际网格系统的要求对相应服务资源状态进行定义和声明，并以此开展后续的服务流验证工作。然而另一方面，由于实际网格服务流中所可能涉及服务状态的多样性，因此本节中重点以最常用的两类资源状态为参照进行网格服务执行及其相关活动的形式化：服务执行状态和数据状态（包括它的值和大小变化）[167]。在此基础上，由于状态π演算对系统状态的创建、消亡、关联和撤销具有开放的建模能力，因此针对实际网格系统相关的其它特殊状态也可以直接利用状态π演算在本节现有的服务形式化基础上做出扩展。

3.2.1　服务执行的状态π演算语义

针对网格中服务执行可能涉及的任务状态，图 3.1 首先给出了它的一个一般抽象。在图 3.1 中，服务被调用时首先会进行所需数据的导入和等待（*StageIn* 和 *Pending*），当服务运行（*Active*）成功后则进行数据的导出（*StageOut*）和清理（*Clean*），否则服务的运行失败（*Failed*）。最后服务的执行进行退出（*Exit*），并将服务执行的结果留由具体的服务调用活动或服务流引擎进行处理。其中，数据的导入、导出和运行总是网格服务执行过程中三个经常涉及的基本状态。例如在 Condor DAGMan 中，其 Job 关键字可以以唯一的名字标识定义待调用服务，而通过 PRE、POST 关键字则可以定义在运行（*Active*）前后为执行该服务所需导入的数据（*StageIn*）、导出的数据（*StageOut*）和需要进行的清理活动（*Clean*）。不论是 *StageIn*、*Active* 还是 *StageOut* 阶段 Condor 都以返回值 0 作为执行成功的标志，而以非 0 作为失败的标志（*Failed*）。

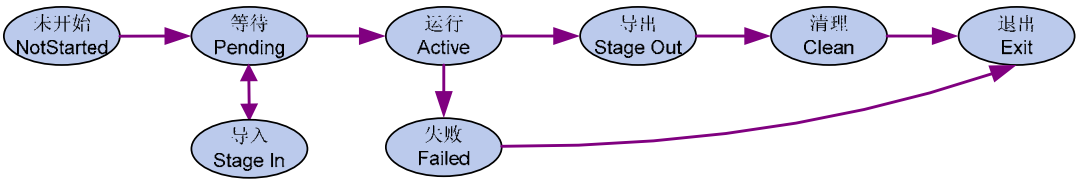


图 3.1　服务执行的状态抽象模型

由此，利用状态π演算对网格服务流中服务执行的形式化语义如图 3.2 所示。不失一般性，在图 3.2 中取待形式化服务为“服务 *A*”（*ServiceA*）。“*#STATE*”关键字代表对相应状态π演算进程中所处理状态的预声明，以利于表示上的清晰。自由名（集合）*port*、*set* 和 *get* 分别是服务调用和变量交互的端口名（关于

服务调用和变量的定义将在 3.2.3 小节详细给出）。通过状态π演算表达式中状态的关联体现了当前服务执行所处的状态（即 *A.Status* 命题）和整个系统中现有执行中的服务（即 *ExecutingSrv* 命题）。图 3.2 中的形式化语义将网格服务 *A* 视为一个独立的实体，其执行过程可以直观地理解为图 3.1 中的状态迁移。更详细的，*NotStarted* 是网格服务 *A* 形式化描述中的初始状态（它的设置方法将在 3.5 小节的网格服务流形式化建模中给出）。当服务 *A* 通过其指定输入端口（$port_1$）收到执行请求及相应参数（$v{:}t_1$）时，它首先进入等待状态（*Pending*），并将服务 *A* 添加入当前执行的服务命题中(++,*execsrv*)；此后 *ServiceA* 进程进入 *StageIn* 阶段（对应于一组并发的 *StageIn* 进程），并不断为服务的执行从指定通道（*get* 端口集合）导入必要的数据。当所有导入完成（第 *n* 个 *ack* 信号被接收到）服务开始进入 *Active* 状态，并开始执行（$\overline{execute} < v:t_1, t, f >$）。之后服务将通过 *StageOut* 进程[②]进入 *StagingOut* 状态，从指定端口导出结果数据（*set*<*res*:t_3>）并完成数据的清理（*Clean*）。最终服务 *A* 实现退出（*Exit*），并从其指定端口返回完成成功信息（$port_2$<*succ*>）。在以上的 *Active* 阶段，一旦出现失败则服务 *A* 将进入 *Failed* 状态，并从其指定端口返回失败信息（$port_2$<*fail*>）。当从端口 $port_2$ 退出时，服务 *A* 也同时从正在执行的服务命题中撤销（--,*execsrv*）。

$$
\begin{aligned}
&\#STATE\ \ srvactive = \{A.Status, \{Active\}\};\ \ \#STATE\ \ srvstagingin = \{A.Status, \{StagingIn\}\}\\
&\#STATE\ \ srvpending = \{A.Status, \{Pending\}\};\ \ \#STATE\ \ srvfailed = \{A.Status, \{Failed\}\}\\
&\#STATE\ \ srvexit = \{A.Status, \{Exit\}\};\ \ \#STATE\ \ srvstagingout = \{A.Status, \{StagingOut\}\}\\
&\#STATE\ \ srvcleaning = \{A.Status, \{Cleaning\}\};\ \ \#STATE\ \ execsrv = \{ExecutingSrv, \{A\}\};\\
&ServiceA(port_1, execute, set, \overrightarrow{get_i}, succ, fail, port_2) =_{def}\\
&\quad new\ ack(port_1(v:t_1)\{(+, srvpending), (++, execsrv)\}.(\prod_{i=1}^{n} StageIn_i \mid\\
&\quad \underbrace{ack.....ack}_{n-1}.ack\{(+, srvactive)\}.new\ t\ f(\overline{execute} < v:t_1, t, f >\mid\\
&\quad (t(res:t_3).(StageOut \mid ack.\overline{port_2}\{(+, srvexit), (--, execsrv)\} < succ >) +\\
&\quad f\{(+, srvfailed)\}.\overline{port_2}\{(+, srvexit), (--, execsrv)\} < fail >))))\\
&StageIn(get, ack) =_{def}\ \ get(x:t_2)\{(+, srvstagingin)\}.\overline{ack}\{(+, srvpending)\}\\
&StageOut(set, ack, res) =_{def}\\
&\quad new\ res\ clean(\overline{set} < res:t_3 > \{(+, srvstagingout)\}.\overline{clean} \mid clean\{(+, srccleaning)\}.\overline{ack}))
\end{aligned}
$$

图 3.2 服务执行的状态π演算形式化语义

② 为简化表示，在本文的进程形式化中，当出现进程嵌套且不需要对该嵌套进程的自由名进行重命名时，则省略了对其自由名集合的表示（如在以上 *ServiceA* 进程中对 *StageIn*、*StageOut* 进程的表示）。下同。

3.2.2　服务执行的多实例化语义

以上 *ServiceA* 的形式化语义中实现的是一种单实例的网格服务执行模型。为了实现服务的并发多实例化，可以通过状态π演算进程的自身嵌套，使得每当服务被调用的同时立即允许新服务实例的生成（如图 3.3 所示）。在图 3.3 的修改语义中，*ServiceA* 进程在它每次通过端口 $port_1$ 被调用时就立即进行了自身的嵌套，以使它继续接受其它存在的服务调用请求。然而这种并发多实例化的问题在于它可能造成进程的无限并发，因此在实际实现中需要对服务流中服务调用次数的有限性和调用端口（*port*）的限定进行严格约束来防止并发进程的无限化。

$$
\begin{aligned}
&ServiceA(port_1, execute, set, \overrightarrow{get_i}, succ, fail, port_2) =_{def} \\
&\quad new\ ack(port_1(v:t_1)\{(+, srvpending),(++, execsrv)\}.(ServiceA \mid \prod\nolimits_{i=1}^{n} StageIn_i \mid \\
&\quad \underbrace{ack.....ack}_{n-1}.ack\{(+, srvactive)\}.new\ t\ f(\overline{execute} < v:t_1, t, f >\mid \\
&\quad (t(res:t_3).(StageOut \mid ack.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < succ >) + \\
&\quad f\{(+, srvfailed)\}.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < fail >))))\\
&StageIn(get, ack) =_{def}\ get(x:t_2)\{(+, srvstagingin)\}.\overline{ack}\{(+, srvpending)\} \\
&StageOut(set, ack, res) =_{def} \\
&\quad new\ res\ clean(\overline{set} < res:t_3 > \{(+, srvstagingout)\}.\overline{clean} \mid clean\{(+, srccleaning)\}.\overline{ack}))
\end{aligned}
$$

图 3.3　服务执行并发多实例化的状态π演算形式化语义

$$
\begin{aligned}
&ServiceA(port_1, execute, set, \overrightarrow{get_i}, succ, fail, port_2) =_{def} \\
&\quad new\ ack(port_1(v:t_1)\{(+, srvpending),(++, execsrv)\}.(\prod\nolimits_{i=1}^{n} StageIn_i \mid \\
&\quad \underbrace{ack.....ack}_{n-1}.ack\{(+, srvactive)\}.new\ t\ f(\overline{execute} < v:t_1, t, f >\mid \\
&\quad (t(res:t_3).(StageOut \mid ack.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < succ > .SeriveA) + \\
&\quad f\{(+, srvfailed)\}.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < fail > .SeriveA))))\\
&StageIn(get, ack) =_{def}\ get(x:t_2)\{(+, srvstagingin)\}.\overline{ack}\{(+, srvpending)\} \\
&StageOut(set, ack, res) =_{def} \\
&\quad new\ res\ clean(\overline{set} < res:t_3 > \{(+, srvstagingout)\}.\overline{clean} \mid clean\{(+, srccleaning)\}.\overline{ack}))
\end{aligned}
$$

图 3.4　服务执行顺序多实例化的状态π演算形式化语义

实际上，若不关心相同服务多实例间执行的时序关系，则可以通过将嵌套

位置的滞后来实现每个 *ServiceA* 实例在执行完毕后才生成新实例的顺序多实例化。图 3.4 中对 *ServiceA* 的嵌套调用就在服务 *A* 已经执行完毕且通过 $port_2$ 返回结果后才进行，从而保证了在新服务实例的生成时，任意时刻同一服务的执行只有一个实例存在。这对于多服务请求下避免服务的同步占用也是有意义的。

3.2.3　基本活动的状态π演算语义

有了以上网格服务执行的形式化语义，下面考虑的是实际发出服务执行请求和接收服务执行结果的服务调用活动（*Invoke*），以及 BPEL4WS 规范中进一步定义的接收（*Receive*）、发送（*Reply*）、赋值（*Assign*）和空活动（*Empty*）。这五类基本活动的形式化语义为建立不同服务执行间的时序约束奠定了基础。其中，活动间的数据传递可以通过变量（*Var*）的共享来实现。在此基础上，通过接收和发送活动可以描述服务流中的直接数据传递或异步消息触发，而赋值活动则可以描述数据的复制转移。它们的状态π演算语义如图 3.5 所示。

$$Invoke(start, get, port_1, port_2, done) =_{def} \ start.get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{done}$$

$$Receive(start, port_2, set, done) =_{def}$$

$$start\{++, \{msgPort, \{port_2\}\}\}.port_2(v:t)\{--, \{msgPort, \{port_2\}\}\}.\overline{set} < v:t > .\overline{done}$$

$$Reply(start, get, port_1, done) =_{def} \ start.get(v:t).\overline{port_1} < v:t > .\overline{done}$$

$$Assign(start, get_1, set_2, done) =_{def} \ start.get_1(v:t).\overline{set_2} < v:t > .\overline{done}$$

$$Empty(start, done) =_{def} \ start.\overline{done}$$

$$Var_V =_{def} \ Var_{V0}(set, get)$$

$$Var_{V0}(set, get) =_{def} \ set(x_1:t)\{++, \{V.bSize, \{x_1\}\}\}.Var_{V1}(set, get, x_1)$$

$$Var_{V1}(set, get, x_1) =_{def} \ set(x_2:t_2)\{++, \{V.bSize, \{x_2\}\}\}.Var_{V2}(set, get, x_1, x_2) \ +$$

$$\overline{get} < x_1:t_1 > \{--, \{V.bSize, \{x_1\}\}\}.Var_{V0}(set, get)$$

.....

$$Var_{Vn}(set, get, x_1, ..., x_n) =_{def} \ \overline{get} < x_n:t_n > \{--, \{V.bSize, \{x_n\}\}\}.Var_{Vn-1}(set, get, x_1, ..., x_{n-1})$$

图 3.5 基本活动的状态π演算形式化语义

图 3.5 中的自由名 *start* 和 *done* 分别代表了对应进程的入口和出口端口。通过后续的 3.3 和 3.4 小节可以发现，正是通过不同状态π演算进程间 *done* 输出和 *start* 输入的同步交互，实现了服务执行间的控制流约束关系。各活动可以通过它们的 *port* 端口进行相应的服务调用和消息收发。这里的 *port* 可以理解为一个

抽象的虚拟服务调用接口，或是一个具体服务的 Endpoint 引用。此外，*Receive* 进程中的 *msgPort* 命题表示该接收活动所占用的消息监听端口，通过对该命题的“++”和“--”操作分别记录了对该消息监听端口的占用和释放情况③。变量的赋值与取值都通过特定的 *set* 和 *get* 端口实现，对传递的数据类型 *t* 的设置是可选的。以上的 Var_V 进程实现了深度为 *n*，取名为 *V* 的变量堆栈，其中通过命题 *V.bSize*（*V* 与变量名对应）记录了当时该变量所包含的值与大小变化④。由此，在以上的状态π演算进程中记录了包括服务当前执行状态和系统中当前变量取值的完整状态信息。注意在这里变量堆栈的实现深度 *n* 必须是有限的，以此确保对应状态π演算进程的有穷性。此外需要注意以上的变量实现中，每次请求完相应数据（如 x_1）后则该数据同时不再于变量中进行保留，即该变量堆栈中的数据是不带共享的一次性存取。而在网格服务流中，由于对数据的请求也可以是它的一个复制（Copy）以实现该数据在不同服务间的共享，因此对此类变量进程 Var_V 的语义则可以实现为（此时变量深度为 1）：

$$\begin{aligned} Var_V(set,get,x) =_{def}\ & \overline{get} < x:t > .Var_V(set,get,x) + \\ & set(y:t)\{++,\{V.CurrentVal,\{y\}\}\}.Var_V(set,get,y) \end{aligned}$$

特别的，由于图 3.5 的形式化语义中都具备相同方式的进程入口（*start*）和出口（*done*）端口，因此若用进程 *Act* 抽象表示以上的任一 *Invoke*、*Receive*、*Reply*、*Assign* 或 *Empty* 进程实现，则记方法 $fn_s(Act)$ 表示 *Act* 进程中除 *start* 端口外所有的自由名集合；记 $fn_d(Act)$ 表示 *Act* 进程中除 *done* 端口外所有的自由名集合；记 $fn_{sd}(Act)$ 表示 *Act* 进程中除 *start* 和 *done* 端口外所有的自由名集合；而记 $fn(P)$ 表示任意状态π演算进程 *P* 的所有自由名集合。例如：$fn(Invoke)=\{start, get, port_1, port_2, done\}$；而 $fn_sd(Invoke)=\{get, port_1, port_2\}$。它们将用来简化后续 3.3 和 3.4 小节中对服务流控制结构和相关模式的状态π演算形式化表示。

3.2.4　服务选择的状态π演算语义

以上考虑的是服务 1 对 1 的显示调用。而在实际的网格服务流中，往往可能存在同时有多个可调用的候选服务来满足相同指定功能的情况。此时需要实现一个额外的服务选择进程，以对服务的选择做出明确的语义刻画（见图 3.6）。

③ 该命题同时为下一章中检验 BPEL4WS 规范语义中的消息竞争约束服务。
④ 该命题同时将在下一章中用于对变量垃圾回收的语义约束进行检验。

$$Selection_1(port_{sel}, port_{11}, port_{12}, ..., port_{n1}, port_{n2}) =_{def} \overline{port_{sel}} < port_{11}, port_{12} > .Selection_2$$

.....

$$Selection_n(port_{sel}, port_{11}, port_{12}, ..., port_{n1}, port_{n2}) =_{def} \overline{port_{sel}} < port_{n1}, port_{n2} > .Selection_1$$

$$Selection =_{def} Selection_1 \qquad n\text{为预定义常数}$$

$$Invoke'(start, get, port_{sel}, done, fail, succ) =_{def}$$

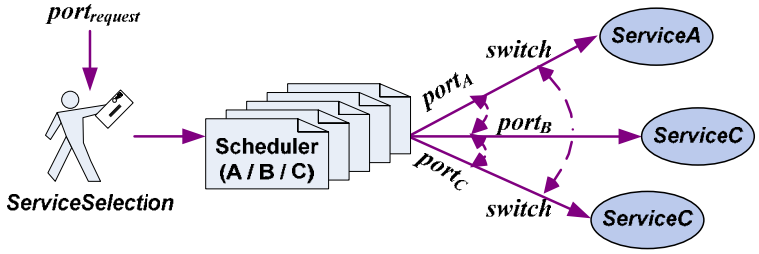
$$start.get(v:t).port_{sel}(p_1, p_2).\overline{p_1} < v:t > .p_2(s).\overline{done}$$

$$ClosedInvocation =_{def} Invoke' \mid Service_1 \mid ..... \mid Service_n \mid Selection$$

图 3.6 简单服务选择的状态π演算形式化语义

此处的 *Selection* 进程存储了实现特定服务的所有 *port* 端口集合，并对这些服务按队列中先后顺序进行选择。而其中服务在队列中的顺序则可以按具体的服务性能指标进行排序，具体策略不在本文的形式化语义范围内。从模型的形式化粒度来看，它将更详细的服务选择策略留到了更底层实现。另外，在这里新的 *Invoke'* 进程不再与某一具体服务直接产生交互，而是通过 $port_{sel}$ 端口进行服务选择的查询，利用命名传递得到所返回的服务端口并赋给 p_1 和 p_2，由此与最终调用的实际服务进行交互。

需要补充的是，当以上服务选择策略无法得到时，（状态）π演算自身的移动性也可以用来实现服务间的动态切换语义。图 3.7 即给出了 3 服务间动态切换的状态π演算形式化描述。此处为了方便表示利用向量 $srvpara_X$ 代表在 3.2.1 小节中服务执行进程 *ServiceA* 中的自由名集合{$port_{X1}$, $execute_X$, set_X, get_X, $succ_X$, $fail_X$, $port_{X2}$}，其中 $X \in \{A, B, C\}$，表示对应服务的相关端口名集合；$port_{X1}$，$port_{X2}$ 则分别特指对应服务的调用和返回端口。此外，状态π演算命题 *selSrv* 被用来记录当前所选择的调用服务。这里假设服务 *A*，*B* 和 *C* 都可以用来完成对某一抽象服务功能的调用请求 $port_{request}$，则在以上状态π演算进程中，通过 $Scheduler_A$、$Scheduler_B$、$Scheduler_C$ 三进程间对各自服务端口 $srvpara_A$，$srvpara_B$，$srvpara_C$ 的相互传递以及对控制器进程 *ServiceSel* 中服务抽象端口 *srvpara* 的重命名实现了对这三个服务选择的动态切换（*switch*）。



$$ServiceSel(\overrightarrow{srvpara}, switch) =_{def} (Service(\overrightarrow{srvpara}) \mid port_2.ServiceSel(\overrightarrow{srvpara}, switch)) + switch(\overrightarrow{srvpara})\{+, \{selSrv, \{port_2\}\}.ServiceSel(\overrightarrow{srvpara}, switch) \quad port_2 \in \overrightarrow{srvpara}$$

$$Scheduler_A(port_{request}, port_{A1}, switch, \overrightarrow{srvpara_B}, \overrightarrow{srvpara_C}) =_{def} port_{request}.(\overrightarrow{port_{A1}}.Scheduler_A + \overrightarrow{switch} < \overrightarrow{srvpara_B} > .\overrightarrow{port_{B1}}.Scheduler_B + \overrightarrow{switch} < \overrightarrow{srvpara_C} > .\overrightarrow{port_{C1}}.Scheduler_C) \quad port_{B1} \in \overrightarrow{srvpara_B} \quad port_{C1} \in \overrightarrow{srvpara_C}$$

$$Scheduler_B(port_{request}, port_{B1}, switch, \overrightarrow{srvpara_A}, \overrightarrow{srvpara_C}) =_{def} port_{request}.(\overrightarrow{port_{B1}}.Scheduler_B + \overrightarrow{switch} < \overrightarrow{srvpara_A} > .\overrightarrow{port_{A1}}.Scheduler_A + \overrightarrow{switch} < \overrightarrow{srvpara_C} > .\overrightarrow{port_{C1}}.Scheduler_C) \quad port_{A1} \in \overrightarrow{srvpara_A} \quad port_{C1} \in \overrightarrow{srvpara_C}$$

$$Scheduler_C(port_{request}, port_{C1}, switch, \overrightarrow{srvpara_A}, \overrightarrow{srvpara_B}) =_{def} port_{request}.(\overrightarrow{port_{C1}}.Scheduler_C + \overrightarrow{switch} < \overrightarrow{srvpara_A} > .\overrightarrow{port_{A1}}.Scheduler_A + \overrightarrow{switch} < \overrightarrow{srvpara_B} > .\overrightarrow{port_{B1}}.Scheduler_B) \quad port_{A1} \in \overrightarrow{srvpara_A} \quad port_{B1} \in \overrightarrow{srvpara_B}$$

$$ServiceSelection(port_{request}) =_{def} new\ \overrightarrow{srvpara_A}\ \overrightarrow{srvpara_B}\ \overrightarrow{srvpara_C}\ switch\ (ServiceSel(\overrightarrow{srvpara_A}, switch) \mid Scheduler_A(port_{request}, port_{A1}, switch, \overrightarrow{srvpara_B}, \overrightarrow{srvpara_C}))$$

图 3.7 服务动态切换的状态π演算形式化语义

3.3 DAGMan控制流结构的状态π演算形式化

本节将讨论 DAGMan 规范中各类控制流关系的状态π演算形式化语义实现。

3.3.1 DAGMan中的*NoPostFail*

在上一小节对单一网格服务执行的实现中，在服务执行时失败时其 *StageOut* 将不会发生，服务执行直接以失败而退出（$\overrightarrow{port_2} < fail >$）。然而 Condor DAGMan 中还支持一类“*-NoPostFail*”开关。当该参数为假时，则 Condor 规定任务的 POST 操作（此处对应为 *StageOut* 进程）即使在服务执行失败时也会进行，且此时整个服务执行的成功与否直接取决于 *StageOut* 的执行成败。如图 3.8 所示，对于此类语义仅需在原有服务实现中统一在服务执行（$\overrightarrow{execute} < v : t_1 >$）后直接进入

StageOut 进程即可，而不需要关心执行（*execute*）的成功（*t*）或失败（*f*）：

$$
\begin{aligned}
&ServiceA(port_1, execute, set, \overrightarrow{get_i}, succ, fail, port_2) =_{def} \\
&\quad new\ ack\ ackfail(port_1(v:t_1)\{(+, srvpending),(++, execsrv)\}.(\prod_{i=1}^{n} StageIn_i \mid \\
&\quad \underbrace{ack.....ack}_{n-1}.ack\{(+, srvactive)\}.new\ t\ f(\overline{execute} < v:t_1 > . \\
&\quad (StageOut \mid (ack.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < succ > + \\
&\quad ackfail\{(+, srvfailed)\}.\overline{port_2}\{(+, srvexit),(--, execsrv)\} < fail >))))) \\
&StageIn(get, ack) =_{def}\ get(x:t_2)\{(+, srvstagingin)\}.\overline{ack}\{(+, srvpending)\} \\
&StageOut(set, ack, ackfail, res, t, f) =_{def}\ new\ res\ clean(\overline{set} < res:t_3 > \{(+, srvstagingout)\}. \\
&\quad \overline{clean}\{(+, srccleaning)\} < t, f > \mid t.\overline{ack} + f.\overline{ackfail}))
\end{aligned}
$$

图 3.8 *NoPostFail* 的状态π演算形式化语义

3.3.2　顺序结构的状态π演算语义

顺序结构（Sequence）表示了服务调用活动间的顺序执行关系。在 DAGMan 中可利用下面的语法直接定义：PARENT $Invoke_{S1}$ CHILD $Invoke_{S2}$。由于 3.2.3 小节中已经提供了包括服务调用 *Invoke* 等基本活动的状态π演算语义，因此以 *Invoke* 为例，可直接利用（状态）π演算中的进程顺序组合“;”[144]对它们的顺序执行约束做出以下形式化（见图 3.9）：

$$
\begin{aligned}
Sequence(fn_d(Invoke_{S1}), fn_s(Invoke_{S2})) &=_{def} (new\ start_{s2})(\{start_{s2} / done_{s1}\} Invoke_{S1} \mid Invoke_{S2}) \\
&=_{def} Invoke_{S1}(start_{s1}, get_{s1}, port_{s11}, port_{s12}, done_{s1})\ ; \\
&\quad Invoke_{S2}(start_{s2}, get_{s2}, port_{s21}, port_{s22}, done_{s2})
\end{aligned}
$$

图 3.9 顺序结构的状态π演算形式化语义

需要指出，图 3.9 的形式化语义中仅关心对 *S1*、*S2* 顺序调用的描述，而不考虑 *S1* 可能存在的调用失败情况（下同）。这是因为：*S1* 的执行失败通过相应服务执行的状态π演算形式化中就已经得到了记录，因此执行失败的语义不会丢失，而不需要在此处重复定义。另一方面，仅考虑顺序执行的语义也可以使本文从整体上更关注于整个网格服务流程中所有服务间的逻辑关联。

3.3.3　同步并发结构的状态π演算语义

在 DAGMan 中可以直接通过指定多 PARENT 元素和多 CHILD 元素来实现

服务调用间的并发与同步结构：

PARENT $Invoke_{Sp1}$ ……$Invoke_{Spm}$ CHILD $Invoke_{Sc1}$ ……$Invoke_{Scn}$

图 3.10 中给出了对并发同步结构的形式化语义（*SynPar*），它进一步推广了顺序结构中的形式化结果。其中的 *SynParImpl* 进程实现了一个同步/并发控制器，它保证了当 $Invoke_{Sp1}$ ……$Invoke_{Spm}$ 均完成之后，可以开始同时发出 $Invoke_{Sc1}$ ……$Invoke_{Scn}$ 的调用请求。

$$SynPar(fn_d(Invoke_{Sp1}),\ldots,fn_d(Invoke_{Spm}),\ldots,fn_s(Invoke_{Sc1}),\ldots,fn_s(Invoke_{Scn})) =_{def}$$
$$(new\ done_{p1}\ .....\ done_{pm}\ start_{c1}\ .....\ start_{cn})$$
$$(Invoke_{Sp1}|.....|Invoke_{Spm}|Invoke_{Sc1}|.....|Invoke_{Scn}|SynParImpl)$$
$$SynParImpl(done_{p1},.....done_{pm},start_{c1},.....start_{cn}) =_{def}$$
$$(new\ ack)(done_{p1}.\overline{ack}|.....|done_{pm}.\overline{ack}|\underbrace{ack.....ack}_{m}.(\overline{start_{c1}}|.....|\overline{start_{cn}}))$$

图 3.10 同步并发结构的状态π演算形式化语义

3.3.4 重试结构的状态π演算语义

DAGMan 中的一类特殊循环结构是服务调用的重试。通过其 Retry 关键词可以对执行失败的服务进行再次尝试，但尝试次数是预先指定的常数。为了实现对重试（*Retry*）结构的形式化，则需对现有 *Invoke* 进程作出如图 3.11 的调整：

$$Invoke_Retry(start,get,port_1,port_2,done,fail,succ) =_{def}$$
$$start.get(v:t).\overline{port_1} < v:t > .port_2(s).([s=fail]Invoke_Retry_1 + [s=succ]\overline{done})$$
$$Invoke_Retry_1(port_1,port_2,v,done,fail,succ) =_{def}$$
$$\overline{port_1} < v:t > .port_2(s).([s=fail]Invoke_Retry_2 + [s=succ]\overline{done})$$
$$.....$$
$$Invoke_Retry_n(port_1,port_2,v,done) =_{def} \overline{port_1} < v:t > .port_2(s).\overline{done} \qquad n\text{为有限常数}$$

图 3.11 同步并发结构的状态π演算形式化语义

图 3.11 中的 $Invoke_Retry_i$ 表示对 *Invoke* 调用的第 *i* 次执行。若在 *i*=*n* 时服务仍无法正确执行完毕，则 *Invoke* 强制退出（$\overline{done}$）。注意此处重试的次数 *n* 是有限的，以满足以上状态π演算进程的有穷性。

3.4　BPEL4WS服务流的状态π演算形式化

BPEL4WS 为服务流的建模提供了更复杂灵活的模型结构，包括其顺序结构（*Sequence*）、循环结构（*While*）、流结构（*Flow*）、选择结构（*Pick*）、分支结构（*Switch*）和同步联接（*Link*）的 6 类控制关系和更复杂的范围限定（*Scope*）、错误补偿（*Fault and Compensation*）、全局终止（*Global Termination*）等高级特性。下面给出针对 BPEL4WS（1.1）规范的状态π演算形式化语义。

3.4.1　顺序结构的状态π演算语义

与 3.3.2 小节的顺序结构类似。但不同的是除 *Invoke* 之外在 BPEL4WS 中还需要考虑接收（*Receive*）、发送（*Send*）、赋值（*Assign*）和空活动（*Empty*）的另外四类基本活动。由于以上基本活动的形式化语义（见 3.2.3 小节）都具备相同的 *start* / *done* 入口和出口，因此若对它们用进程 *Act* 进行统一抽象，则有：

$$\begin{aligned} &Sequence(fn_d(Act_1), fn_s(Act_2)) \\ &\quad =_{def}\ Act_1\ ; Act_2 =_{def}\ (new\ start_{Act_2})(\{start_{Act_2} / done_{Act_1}\} Act_1 \mid Act_2) \end{aligned}$$

正是通过对每个活动和结构入口与出口（即自由名 *start* 和 *done*）的重命名和限定（*new*），使得活动和结构之间能够建立起直接的通信连接，从而确保了两者在执行上严格的顺序关系。

3.4.2　流结构的状态π演算语义

流结构（*Flow*）描述的是服务相关活动及结构间的并发与完成的同步。因此参照 3.3.3 小节的实现可直接得到流结构的状态π演算形式化语义（见图 3.12）。

$$\begin{aligned} &Flow(fn_{sd}(Act_1), ..., fn_{sd}(Act_m), start_{Flow}, done_{Flow}) =_{def} \\ &\quad (new\ start_{Act_1}\ ...\ start_{Act_m}\ done_{Act_1}\ ...\ done_{Act_m}\ ack\ ack') \\ &\quad (start_{Flow}.Starter \mid Act_1 \mid ..... \mid Act_m \mid Acker \mid ack'.\overline{done_{Flow}}) \\ &Starter(start_{Act_1}, ..., start_{Act_m}) =_{def}\ \overline{start_{Act_1}} \mid ..... \mid \overline{start_{Act_m}} \\ &Acker(done_{Act_1}, ..., done_{Act_m}, ack, ack') =_{def}\ done_{Act_1}.\overline{ack} \mid ..... \mid done_{Act_m}.\overline{ack} \mid \underbrace{ack.....ack}_{m}.\overline{ack'} \end{aligned}$$

图 3.12 流结构的状态π演算形式化语义

3.4.3　循环结构的状态π演算语义

循环结构（*While*）表示的是按照循环条件对某一（组）服务的重复调用。若将该循环条件用一状态π演算命题 C 及其分别的真值集合$\{t\}$、$\{f\}$表示，则图 3.13 中给出了循环结构的状态π演算形式化语义：

$$\begin{aligned}&While(fn_{sd}(Act), start_{while}, done_{while}) =_{def} \ new\ start_{Act}\ done_{Act}(start_{while}\cdot\\&\quad([eval(C,\{t\})](\overline{start_{Act}} \mid Act \mid done_{Act}.(\overline{start_{while}} \mid While)) + [eval(C,\{f\})]\overline{done_{while}}))\end{aligned}$$

图 3.13 流结构的状态π演算形式化语义

通过对 $eval(C, \{t\})$和 $eval(C, \{f\})$的评估，则在 *While* 中将选择从 $done_{while}$ 输出端口直接退出循环，或通过进程嵌套重复调用对应的活动进程 *Act*。

3.4.4　分支结构的状态π演算语义

分支结构（*Switch*）表示了一种带条件的选择，其中特别需要严格定义的是当不同分支条件产生冲突时的语义。若将不同分支条件分别用状态π演算命题 C 及其真值集合$\{t\}$、$\{f\}$表示，则图 3.14 中给出了分支结构的状态π演算语义：

$$\begin{aligned}&Switch(fn_{sd}(Act_1), fn_{sd}(Act_2), start_{Switch}, done_{Switch}) =_{def} (new\ start_{Act_1}\ start_{Act_2}\ done_{Act_1}\ done_{Act_2})\\&\quad(start_{Switch}.([eval(C_1,\{t\})]\overline{start_{Act_1}}\{(++,\{Branch,\{Act_1\}\})\} \mid Act_1 \mid done_{Act_1}.\overline{done_{Switch}} +\\&\quad[eval(C_1,\{f\}) \wedge eval(C_2,\{t\})]\overline{start_{Act_2}}\{(++,\{Branch,\{Act_2\}\})\} \mid Act_2 \mid done_{Act_2}.\overline{done_{Switch}}))\end{aligned}$$

图 3.14 分支结构的状态π演算形式化语义

其中，进程 *Switch* 明确定义了当分支条件产生冲突时（如 C_1，C_2 同时满足），则将按照对应分支的定义顺序（先 Act_1 后 Act_2）进行选择。这一点与 BPEL4WS 中对 *Switch* 的规范要求是严格一致的。另外，在命题 *Branch* 中则通过状态π演算记录了进程演化过程中所选择过的分支路径。

3.4.5　选择结构的状态π演算语义

选择结构（*Pick*）描述的是不同服务活动以及结构间基于消息触发的执行选择。根据 BPEL4WS 规范，*Pick* 结构被用来阻塞当前服务流的执行从而：（1）等待接受对应的消息来触发下一步需要调用的服务；或（2）等待接收消息等待的超时事件。进程的选择操作“+”正可以用来描述这一类行为（见图 3.15）：

$$
\begin{aligned}
&Pick(fn_{sd}(Act_1), fn_{sd}(Act_2), port_{p1}, port_{p2}, timeout, start_{Pick}, done_{Pick}) =_{def} \\
&\quad (new\ start_{Act_1}\ start_{Act_2}\ done_{Act_1}\ done_{Act_2})(start_{Pick}. \\
&\quad (port_{p1}\{++, \{msgPort_1, \{port_{p1}\}\}\}. \\
&\qquad start_{Act_1}\{(++, \{Event, \{Act_1\}\}), (--, \{msgPort_1, \{port_{p1}\}\})\} \mid Act_1 \mid done_{Act_1}.\overline{done_{Pick}} + \\
&\quad port_{p2}\{++, \{msgPort_2, \{port_{p2}\}\}\}. \\
&\qquad start_{Act_2}\{(++, \{Event, \{Act_2\}\}), (--, \{msgPort_2, \{port_{p2}\}\})\} \mid Act_2 \mid done_{Act_2}.\overline{done_{Pick}}) + \\
&\quad timeout\{(++, \{Event, \{Timeout\}\})\}.\overline{done_{Pick}})
\end{aligned}
$$

图 3.15 选择结构的状态π演算形式化语义

在 *Pick* 进程中，具体服务活动的调用（Act_1 和 Act_2）将通过不同端口（$port_{p1}$ 和 $port_{p2}$）所接收到的消息进行选择。其中，*msgPort* 命题表示不同选择情况下所占用的消息监听端口，它负责为选择结构记录其各个消息监听端口的占用（++）和释放（--）情况⑤。此外，*timeout* 端口负责监听外界的超时事件。它与消息接收的端口相竞争，若在接收到合适的消息之前就已经检测到超时信号，则命题 *Event* 中将会记录此次选择过程中的超时事件，并让 *Pick* 进程退出。

3.4.6　同步联接的状态π演算语义

在 BPEL4WS 的流结构（*Flow*）中，除了描述服务活动间的并发执行外，也需要约束这些服务活动间可能存在的同步关系。*Flow* 中的同步联接（*Link*）结构正起到这一作用。每个同步联接都有其源活动和目标活动，使得目标活动的执行必须在源活动完成之后进行。此外，当源活动被确定为无法执行时（例如源活动所在的分支没有被选取），则在所有相关的同步联接上需要递归广播拒绝信号（*negative*），以取消对应目标活动对同步信号的等待。这称为“Death-Path Elimination”。通过同步联接可将非结构化的流程引入到网格服务流的建模中，从而使活动间的控流构关系更加灵活。它的状态π演算语义如图 3.16 所示：

⑤ 该命题同时为下一章中检验 BPEL4WS 规范语义中的消息竞争约束服务。

$$Link_i(done_{in}, neg_{in}, ack, nack) =_{def} \; EvalTransCondition_i(done_{in}, neg_{in}, ack, nack)$$

$$EvalTransCondition_i(done_{in}, neg_{in}, ack, nack) =_{def} \; done_{in}.\overline{ack} + neg_{in}.\overline{nack} \qquad i = 1,...,n$$

$$Links(done_{in}, neg_{in}, done_{links}, deathpath) =_{def} \; (new\ ack\ nack)$$
$$(Link_1 \mid ... \mid Link_n \mid \underbrace{ack.(... .(ack}_{n}.\overline{done_{links}} + nack.\overline{deathpath})... + nack.\overline{deathpath}))$$

$$ActWithLinks(freeN) =_{def} \; start.(done_{links}.new\ t\ f(\overline{evaljoin} < t, f > .($$
$$t.(new\ start_{Act}\ done_{Act})(\overline{start_{Act}} \mid Act \mid done_{Act}.\overline{done}.\prod \overline{done_{out}}) +$$
$$f\{++, \{Exception, \{Act\}\}\}.\overline{throw} < joinfailure >)) + deathpath.\prod \overline{neg_{out}})$$

$$freeN = \{start, done, done_{links}, done_{out}, neg_{out}, deathpath,$$
$$evaljoin, fault, joinfailure, fn_{sd}(Act)\}$$

图 3.16 同步联接的状态π演算形式化语义

图 3.16 进程中 *ActWithLinks* 表示在 3.2.3 小节中的五类基本活动当考虑同步联接时的实现。此时，活动的开始不仅需要其前继活动完成的触发（*start*），还需要得到以该活动为目标的所有同步联接的确认（$done_{links}$）。当以上都得到满足之后，进程开始评估活动的执行条件（*evaljoin*），若条件通过则正常执行该活动，并在执行结束时同时出发后续活动（*done*）以及以它为源活动的所有同步联接（$done_{out}$）；否则进程将通过 *throw* 端口抛出 *joinfailure* 异常（有关接收异常处理 *throw* 的进程实现将在 3.4.8 小节详细介绍），并将其记录到 *Exception* 变量中。此外，在以上 *Link* 进程中对于接受到的拒绝信号（neg_{in}）最终都会通过 *deathpath* 通知到该同步联接的目标活动，并且由该目标活动继续将拒绝信号（neg_{out}）广播到以它为源活动的其它同步联接，以此实现同步联接的完整语义。

3.4.7　范围限定的状态π演算语义

范围（*Scope*）在 BPEL4WS 中是一个定义变量、补偿、异常处理和其它服务活动的有效作用域的一个重要概念。例如，可以通过对指定范围或其父范围中补偿活动的调用来处理服务流执行过程中的相应异常。由于 BPEL4WS 中的模型元素可以被关联于某一有效范围，因此为了实现该范围限定语义，记 *FreeNames*(*s*)表示一范围 *s* 内所有服务活动、变量、补偿进程的自由名集合；定义 *PortNames*(*s*)为一个范围 *s* 内所有服务活动进程的 *port* 名集合。由此，状态π演算的限定操作符（*new*）可以被用来根据给定的有效范围来限定其服务活动、变量、补偿中可以与范围外进程相交互的端口名，以此限定它们的行为，即：

$Scope_s(PortNames(s))=(new\ FreeNames(s)\ /\ PortNames(s))(Act_1 \mid Act_2 \mid \ldots\ldots \mid Act_n)$
其中，各 Act_i 代表该范围 s 内的所有活动、变量和异常补偿的相应进程。

3.4.8　异常处理与补偿的状态π演算语义

BPEL4WS 规范中的异常处理和事务补偿对于网格服务流中耗时的计算任务以及潜在的动态性来说也一个必要的保障机制。为了能将服务执行过程中的异常情况正确捕获下来，图 3.17 中首先对调用活动在相应服务执行失败时的异常抛出语义做出补充。其中，此处 *Invoke* 进程的下标 *S* 表示对服务 *S* 的调用。

$$Invoke_S_WithFault(start, get, port_1, port_2, done, throw, invokefailure, fail, succ) =_{def}$$
$$start.get(v:t).\overline{port_1} < v:t > .port_2(s).($$
$$[s = fail]\tau\{(++, \{Exception, \{S\}\})\}.\overline{throw} < invokefailure > +[s = succ]\overline{done})$$
$$ThrowAct(throw, fault) =_{def}\ throw(failtype).\overline{fault} < failtype >$$

图 3.17 异常抛出的状态π演算形式化语义

在图 3.17 中当服务的调用过程中返回失败结果时（*s=fail*），则通过 *throw* 输出端口将抛出 *invokefailure* 异常，并将对应服务 *S* 记录到 *Exception* 状态变量中。与 BPEL4WS 相对应，除了异常的抛出之外，图 3.18 中的 *FaultHandling* 进程则实现了网格服务流执行过程中对所抛出异常的接受和处理。*FaultHandling* 通过 *fault* 端口从 *ThrowAct* 进程获得相应异常。当异常的类型在 *FaultHandling* 所能处理的类型范围内时（这里用 $type_1$，…，$type_n$ 进行抽象表示，它们可以是以上的 *invokefailure* 或 *joinfailure* 等），则调用相关的服务活动 Act_i 进行处理。否则 *FaultHandling* 进程会顺序调用所有可用的补偿活动 Act^c 以补偿失败的服务执行。*FaultHandling* 及补偿（$CompensationHandler_j$）的形式化可参见图 3.18。

$$FaultHandling(faulttype, \overrightarrow{type}, \overrightarrow{compensate}, fn_s(Act_1), \ldots, fn_s(Act_n)) =_{def}$$
$$fault(faulttype).(new\ start_1\ \ldots\ start_n)$$
$$([faulttype = type_1](\overline{start_1} \mid Act_1) + \ldots + [faulttype = type_n](\overline{start_n} \mid Act_n) +$$
$$\sum_{type_i \in \overrightarrow{type},\ i \neq 1..n} [faulttype = type_i](\overline{compensate_1}.\ \ldots\ldots.\overline{compensate_m}))\quad m\text{为有限常数}$$
$$CompensationHandler_j(compensate_j, fn_{sd}(Act_j^c)) =_{def}\ compensate_j.$$
$$((new\ start_j^c\ done_j^c)(\overline{start_j^c} \mid Act_j^c \mid done_j^c.CompensationHandler_j))\quad 1 \le j \le m$$

图 3.18 异常补偿的状态π演算形式化语义

3.4.9　全局终止的状态π演算语义

全局终止是 BPEL4WS 服务流规范的另一个重要语义。与 UML 活动图类似，它要求在满足特定全局终止条件下（如服务流执行发生异常或非正常终止）立即强制中止所有相应正在执行或等待执行中的服务活动。服务活动的全局终止是一个复杂的语义实现。与 1.4.3 小节综述中 Puhlmann 等人[110]为取消模式（Cancellation Patterns）实现的π演算语义不同，它要求每一个服务活动都随时监听终止信号，而不仅仅是撤销对服务调用的等待。此外，需要有一个全局的终止信号通知机制来确保每个服务活动都能及时准确地被终止。状态π演算对系统状态的管理和互操作能力正可用以对这一复杂语义的实现，因为它简化了对进程演化过程中历史动作的记录，以及这些历史动作信息对未来进程演化的约束。以服务执行的异常为例，设实际全局终止请求的发出条件是服务流中有任意服务的执行产生了异常（即：$\{\} \subset range(Exception)$，其中 *Exception* 是上一小节中记录服务执行异常的状态π演算命题），则基于状态π演算只要对现有服务活动进程通过图 3.19 中的简单扩充就可以实现它们相应的全局终止语义：

$$Invoke(start, get, port_1, port_2, done) =_{def} start.[\phi]get(v:t).[\phi]\overline{port_1} < v:t > .[\phi]port_2(s).[\phi]\overline{done}$$

$$Receive(start, port_2, set, done) =_{def} start.[\phi]port_2(v:t).[\phi]\overline{set} < v:t > .[\phi]\overline{done}$$

$$Reply(start, get, port_1, done) =_{def} start.[\phi]get(v:t).[\phi]\overline{port_1} < v:t > .[\phi]\overline{done}$$

$$Assign(start, get_1, set_2, done) =_{def} start.[\phi]get_1(v:t).[\phi]\overline{set_2} < v:t > .[\phi]\overline{done}$$

图 3.19 全局终止的状态π演算形式化语义

图 3.19 中的条件$\phi = eval(Exception, \{\})$。由于根据状态π演算语义，当条件$\phi$不满足时进程的行为表现为空（0 进程），因此通过状态π演算全局状态命题的管理能力直接实现了进程执行的终止。实际上，基于 *eval* 关系还可实现更复杂的全局终止语义的组合。例如，通过$[eval(S.Status, \{Exit\})][eval(Event, \{Timeout\})]$可以表示当某关键服务 *S* 得到执行后若存在 *Pick* 结构接收到等待超时的事件（见 3.4.5 小节的选择结构语义），则此时进行网格服务流的全局终止。

3.5　服务流的形式化示例

有了以上对各个单一服务、服务调用活动及其控制结构的分别语义，一个

完整网格服务流的形式化语义则可以通过状态π演算自身的组合操作符“|”直接构建出来。整个网格服务流的状态π演算语义在本文中定义为其相应流程结构（*FlowStruct*）和流程上下文（*FlowContext*）两部分的并发组合。其中，流程结构为所有服务活动、控制结构和补偿的组合，它反映整个服务流中的控制流与数据流结构；而流程上下文则是网格环境中所有服务和选择策略的组合。以上两者的定义如图 3.20 所示⑥，其中的各个状态π演算进程均已在前文实现。

$$
\begin{aligned}
Act &::= Invoke \mid Send \mid Receive \mid Assign \mid Retry \mid Empty \mid Variable \\
Structure &::= Flow \mid Sequence \mid Switch \mid While \mid Link \mid Pick \mid Retry \mid SynPar \\
Handler &::= FaultHandling \mid CompensationHandler \mid Scope \\
FlowStruct &::= Act \mid Structure \mid Handler \mid (FlowStruct \mid FlowStruct) \\
FlowContext &::= Service \mid Selection \mid (FlowContext \mid FlowContext)
\end{aligned}
$$

图 3.20 服务流结构与上下文的语法定义

由此，若将整个网格服务流用进程 *SrvFlow* 表示，则有：

$$
\begin{aligned}
&\#STATE \quad SysState_{init} = \{(Srv_1.status, \{NotStarted\}), ..., (Srv_n.status, \{NotStarted\})\}; \\
&SrvFlow =_{def} \quad new\ (fn(FlowStruct, FlowContext)) \\
&\qquad\qquad (\tau\{+, SysState_{init}\}.(FlowStruct \mid FlowContext))
\end{aligned}
$$

以上的 *SrvFlow* 整体可以视为一个不包含自由名的闭系统，其中的 *FlowStruct* 按照既定的网格服务流结构与 *FlowContext* 中的服务进行交互。此外，一个前置的不可见动作 τ 用来协助设置整个服务流在开始时的初始状态 $SysState_{init}$（如设置其相应服务在初始时处于“未开始”的 *NotStarted* 状态）。

作为示例，图 3.21 给出了一个 LIGO 数据网格中引力波探测数据分析应用的简单片断。它表示在完成了对探测数据波形匹配结果的阀值过滤（*Inspiral*）后，进行两项偶然性分析工作（*sInca* 和 *thInca*）以进一步保障过滤结果的有效性。图 3.21 中分别给出了它的 Condor DAGMan 描述和对应的状态π演算语义。其中，$Service_{Insp}$、$Service_{sInca}$ 和 $Service_{thInca}$ 进程分别对应了 $Invoke_{Inspiral}$、$Invoke_{sInca}$ 和 $Invoke_{thInca}$ 所调用的服务执行，它们的状态π演算语义已在 3.2.1 小节给出，在此不再重复罗列。以上实际 LIGO 数据网格应用的多个完整服务流实例将分别在第 4 章和第 5 章中给出，并对它们的验证结果及性能做出进一步讨论。

⑥ 为了对状态π演算中的组合操作“|”和语法范式中的“或”标记“|”进行区分，在定义中用“(P | Q)”的形式特指状态π演算中两进程 *P*，*Q* 的组合操作。

JOB *Inspiral inspiral_pipe.inspiral.sub* VARS *VarInsp* …… RETRY *Inspiral* 0
JOB *sInca inspiral_pipe.sinca.sub* VARS *VarsInca* ……RETRY *sInca* 0
JOB *thInca inspiral_pipe.thinca.sub* VARS *VarthInca* ……RETRY *thVar* 0
PARENT *Inspiral* CHILD *sInca thInca*

$$\#STATE\ \ InitS = \{(Inspiral.status, \{NotStarted\}),$$
$$(sInca.status, \{NotStarted\}), (thInca.status, \{NotStarted\})\};$$
$$Invoke_{Inspiral}(start_{Insp}, get_{Insp}, port_{Insp1}, port_{Insp2}, done_{Insp}) =_{def}$$
$$start_{Insp}.get_{Insp}(v).\overline{port_{Insp1}} < v > .port_{Insp2}.\overline{done_{Insp}}$$
$$Invoke_{sInca}(start_s, get_s, port_{s1}, port_{s2}, done_s) =_{def}$$
$$start_s.get_s(v).\overline{port_{s1}} < v > .port_{s2}.\overline{done_s}$$
$$Invoke_{thInca}(start_{th}, get_{th}, port_{th1}, port_{th2}, done_{th}) =_{def}$$
$$start_{th}.get_{th}(v).\overline{port_{th1}} < v > .port_{th2}.\overline{done_{th}}$$
$$Var_{Insp}(set_{Insp}, get_{Insp}, inspd) =_{def} \overline{get_{GWD}} < inspd > .Var_{Insp}(set_{Insp}, get_{Insp}, inspd)$$
$$+ set_{Insp}(y)\{+, \{Insp.CurrentVal, \{y\}\}\}.Var_{Insp}(set_{Insp}, get_{Insp}, y)$$
$$Var_{sInca}(set_s, get_s, sIncad) =_{def} \overline{get_s} < sIncad > .Var_{sInca}(set_s, get_s, sIncad)$$
$$+ set_s(y)\{+, \{SInca.CurrentVal, \{y\}\}\}.Var_{sInca}(set_s, get_s, y)$$
$$Var_{thInca}(set_{th}, get_{th}, thIncad) =_{def} \overline{get_{th}} < thIncad > .Var_{thInca}(set_{th}, get_{th}, thIncad)$$
$$+ set_{th}(y)\{+, \{ThInca.CurrentVal, \{y\}\}\}.Var_{thInca}(set_{th}, get_{th}, y)$$
$$SynPar(freeN) =_{def} (new\ done_{Insp}\ start_s\ start_{th})$$
$$(Invoke_{Inspiral} \mid Invoke_{sInca} \mid Invoke_{thInca} \mid SynParImpl)$$
$$freeN = \{start_{Insp}, get_{Insp}, port_{Insp1}, port_{Insp2}, get_s, port_{s1}, port_{s2},$$
$$get_{th}, port_{th1}, port_{th2}, done_s, done_{th}\}$$
$$SynParImpl(done_{Insp}, start_s, start_{th}) =_{def} done_{Insp}.(\overline{start_s} \mid \overline{start_{th}})$$
$$ServiceFlow =_{def} new\ (fn(SynPar, Var_{Insp}, Var_{sInca}, Var_{thInca}, Service_{Insp}, Service_{sInca}, Service_{thInca}))$$
$$(\tau\{+, InitS\}.(\overline{start_{Insp}} \mid SynPar \mid Var_{Insp} \mid Var_{sInca} \mid Var_{thInca} \mid$$
$$Service_{Insp} \mid Service_{sInca} \mid Service_{thInca} \mid done_s \mid done_{th}))$$

图 3.21 引力波探测数据分析服务流片段的状态π演算语义示例

3.6 网格服务流中的并发与管道模式

除了以上对服务的调用执行、控制流和数据流结构以及异常处理、全局终止等特性语义外，JOpera 小组的研究者们还为网格服务流提出了一组常用的执行模式[23]。它包含两大类：并发执行模式（Parallel Execution）与管道处理模式（Pipeline Execution）（见图 3.22）。与传统工作流模式类似，网格服务流模式总结了 E-Science 领域中常用的控制流和数据流约束及其实现。然而目前对这些网格服务流模式却仍没有一个有效的形式化语义支持。因此，本小节利用状态π演算对网格服务流模式进行形式化，一方面为网格领域中的常见的复杂服务流

关系给出形式化实现，另一方面也验证状态π演算作为网格服务流形式化语义的基础有着足够强的表达能力。以下分别给出每种对应模式的状态π演算语义。

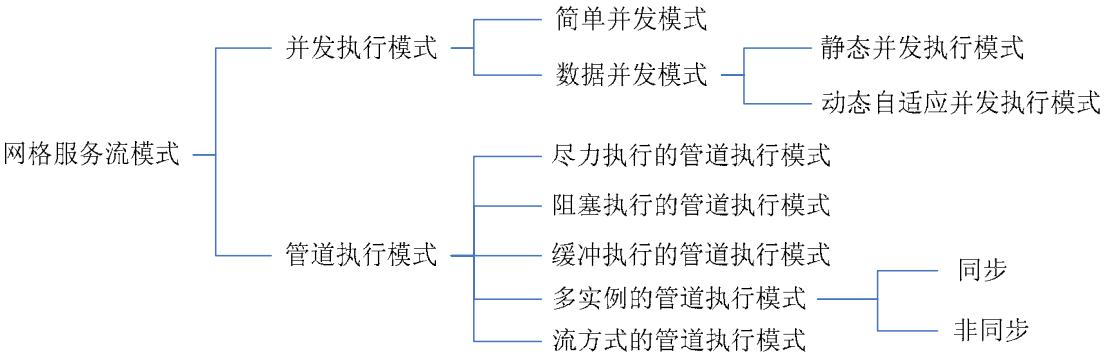


图 3.22　网格计算中的并发与管道模式

3.6.1　静态并发执行模式及其状态π演算语义

在图 3.22 中，由于简单并发执行模式的语义与 3.4.2 小节中的流结构实现一致，因而本节关心的是两类数据并发执行模式，即：静态并发执行模式和动态并发执行模式。数据并发执行模式的由来是出于在科学计算中频繁出现的各类参数试验（Parametric Experiment）。例如在引力波探测的 LIGO 数据网格应用中，对引力波数据的波形匹配服务往往将在不同的海量引力波探测数据上分组进行并发检查，以找出真正有分析价值的引力波数据。

首先在静态并发执行模式中，它给定的数据参数组大小是已知不变的。因此不失一般性，假定有 n 个参数分别存储在变量中待并发处理，则静态并发执行模式可以通过以下图 3.23 的状态π演算语义实现对这些数据参数的并发操作。

在图 3.23 中，*StaticDataParallel* 进程将通过交错并发（Interleaving）方式令 n 个服务调用活动分别读取变量中的 n 个数据，并交由 *port* 端口所指定的服务实例对这些数据进行处理。服务的调用过程是完全并发进行的⑦。此外，根据调整以上 *Acker* 进程中对执行完毕确认信号（*ack*）数量的统计（m），更可以灵活实现该并发执行模式中的不同同步策略：Wait-for-All（即 $m=n$）；Wait-for-One（即 $m=1$）和 m-out-of-n（即 $1 \le m \le n$）。注意以上实现中变量的深度 n 是有限的，以确保对应状态π演算进程的有穷性。变量中的命题 *bSize* 记录了该变量的大小变化，它将在下一章中用于对变量垃圾回收的规范语义约束进行检验。

⑦ 与服务调用活动的并发多实例相应，服务执行的多实例化语义实现可参见 3.2.2 小节（下同）。

$$Var_0(set,get) =_{def}\ set(x_1)\{++,\{bSize,\{x_1\}\}\}.Var_1(set,get,x_1)$$

$$Var_1(set,get,x_1) =_{def}\ set(x_2)\{++,\{bSize,\{x_2\}\}\}.Var_2(set,get,x_1,x_2)\ +$$

$$\overline{get} < x_1 > \{\text{--},\{bSize,\{x_1\}\}\}.Var_0(set,get)$$

.....

$$Var_n(set,get,x_1,.....,x_n) =_{def}\ \overline{get} < x_n > \{\text{--},\{bSize,\{x_n\}\}\}.Var_{n-1}(set,get,x_1,.....,x_{n-1})$$

$$Var =_{def}\ Var_n(set,get,x_1,.....,x_n)$$

$$Invoke_i(start_i,get,port_1,port_2,done_i) =_{def}\ start_i.get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{done_i}$$

$$StaticDataParallel(port_1,port_2,start_{DP},done_{DP}) =_{def} \qquad i = 1,2,...,n$$

$$(new\ start_1\ .....\ start_n\ done_1\ .....\ done_n\ get\ set\ ack\ ack')$$

$$(start_{DP}.Starter \mid Invoke_1 \mid ..... \mid Invoke_m \mid Var \mid Acker \mid ack'.\overline{done_{DP}})$$

$$Starter(start_1\ .....\ start_n) =_{def}\ \overline{start_1} \mid ..... \mid \overline{start_n}$$

$$Acker(done_1\ .....\ done_n\ ack\ ack') =_{def}\ done_1.\overline{ack} \mid ..... \mid done_n.\overline{ack} \mid \underbrace{ack.....ack}_{m}.\overline{ack}'$$

图 3.23 静态并发执行模式的状态π演算语义

3.6.2 动态自适应并发执行模式及其状态π演算语义

动态自适应并发执行模式相对静态并发执行模式的复杂性在于给定数据参数的数量不是事先已经确定的常数，而所需执行的服务实例需要动态地灵活生成。因此，它不能像静态并发执行模式那样预定义 *n* 个服务调用活动，而需要根据进程嵌套完成多服务调用的动态生成（见图 3.24）。在图 3.24 的服务调用进程（$Invoke_i$）中，每当探测到有新数据的存在（*get*）时将立即自发创建一新的服务调用进程实例（$Invoke_{i+1}$），并在调用相应服务做出处理（$port_1$）的同时，继续探测其它新数据的到来。此外，在这里进程间的并发使得 $Invoke_i$ 和 $Invoke_{i+1}$ 对 $port_1$ 的并发调用也成为可能。需要额外注意的是：

（1） 为了防止进程数的无限生成，在以上变量的 *Var* 进程中，*k* 表示变量中初始的数据量，而 *n* 用来限定了变量的最大存储上限；

（2） *p* 用来限定服务调用 *Invoke* 进程的最大并发数；而相应的在 *Acker* 进程中 *m*（$1 \le m \le p$）与 3.6.1 小节一样用来控制并发执行模式的最终同步策略；

（3） 与 3.6.1 小节不同，*AdaptiveDataParallel* 进程中对于变量的设置端口 *set* 进行了开放，以保证数据的动态获得并对应出发新的并发服务执行进程。

$$Var_0(set,get) =_{def}\ set(x_1)\{++,\{bSize,\{x_1\}\}\}.Var_1(set,get,x_1)$$

$$Var_1(set,get,x_1) =_{def}\ set(x_2)\{++,\{bSize,\{x_2\}\}\}.Var_2(set,get,x_1,x_2)\ +$$

$$\overline{get} < x_1 > \{--,\{bSize,\{x_1\}\}\}.Var_0(set,get)$$

.....

$$Var_n(set,get,x_1,.....,x_n) =_{def}\ \overline{get} < x_n > \{--,\{bSize,\{x_n\}\}\}.Var_{n-1}(set,get,x_1,.....,x_{n-1})$$

$$Var =_{def}\ \tau\{+,\{bSize,\{x_1,.....,x_k\}\}\}.Var_k(set,get,x_1,.....,x_k)$$

$$Invoke_i(start_i,get,port_1,port_2,done_i,\ ...,\ done_p) =_{def} \qquad i = 1,...,p-1$$

$$start_i.get(v:t).((new\ start_{i+1})(\overline{start_{i+1}} \mid Invoke_{i+1}) \mid \overline{port_1} < v:t > .port_2(s).\overline{done_i})$$

$$Invoke_p(start_p,get,port,done_p) =_{def}\ start_p.get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{done_p}$$

$$AdaptiveDataParallel(port_1,port_2,set,start_{DP},done_{DP}) =_{def}$$

$$(new\ start_1\ done_1\ .....\ done_p\ get\ ack\ ack')$$

$$(start_{DP}.Starter \mid Invoke_1 \mid Variable \mid Acker \mid ack'.\overline{done_{DP}})$$

$$Starter(start_1) =_{def}\ \overline{start_1}$$

$$Acker(done_1\ .....\ done_p\ ack\ ack') =_{def}\ done_1.\overline{ack} \mid ..... \mid done_p.\overline{ack} \mid \underbrace{ack.....ack}_{m}.\overline{ack'}$$

图 3.24 动态并发执行模式的状态π演算语义

3.6.3 尽力执行的管道执行模式及其状态π演算语义

与并发执行模式相对的，管道执行模式描述的是数据经过一系列服务顺序处理的场景。该模式同样也在科学计算中普遍存在。例如在引力波探测的 LIGO 数据网格应用中，在确认给定引力波数据的有效性之前需要经过数据波形的模板库生成、波形的匹配和偶然性分析等多步序列操作。

然而由于管道的执行可能发生拥堵（Pipeline Collision），即在当前服务执行完毕并准备触发执行下一服务时，后续服务可能因尚未完成而无法及时做出响应。因此，网格服务流模式将管道执行模式分为了 5 类不同的管道执行语义（见图 3.22）。本小节先讨论第一种“尽力执行的管道执行模式（Best Effort）”。

在尽力执行的管道执行模式中，当管道执行由于后续服务尚未完成而发生拥堵时，则前一服务应直接放弃尝试对后续服务发出执行请求。由于之前的3.2.1小节已经实现了对服务执行状态的刻画，因此利用状态π演算可以通过以下方式方便地基于这些信息实现对后续服务执行请求的终止：

$$Invoke_k(start,get,port_1,port_2,done) =_{def}$$

$$start.get(v:t).\overline{port_1} < v:t > .port_2(s).([\phi]\overline{done} \mid Invoke_k)$$

以上用 $Invoke_k$ 表示管道执行中的第 k 步任务，ϕ表示对下一步调用服务（设其对应服务为 S_{k+1}）当前执行是否终止的判断，即：$eval(S_{k+1}.Status, Exit)$。因此根据状态π演算的扩展操作语义此处的 *done* 信号仅在ϕ成立时发出，否则将被丢弃且同时 $Invoke_k$ 开始重新等待对新数据的处理。此外，通过 3.5 小节中网格服务流的整体实现语义中可以看出，此处服务调用活动的 *start* 和 *done* 端口将被进行限定以控制由于进程嵌套所可能引发的并发进程数量。

3.6.4　阻塞执行的管道执行模式及其状态π演算语义

在阻塞执行的管道执行模式（Blocking）中，当管道执行发生拥堵时，前一服务将阻塞其对后续服务执行请求的发出，并一直等待后续服务执行完毕后才发出该执行请求。对于这一模式，状态π演算自身的守备操作“.”可以直接用于实现这一阻塞语义：

$$Invoke_k(start, get, port_1, port_2, done) =_{def} start.get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{done}.Invoke_k$$

3.6.5　缓冲执行的管道执行模式及其状态π演算语义

在缓冲执行的管道执行模式（Buffered）中，当管道执行发生拥堵时，前一服务对后续服务发出的执行请求将被缓冲，而后续服务在执行完毕后将从缓冲中继续实现累积的未完成请求。因此，为了实现缓冲执行的管道执行模式，有必要对缓冲队列做出额外的形式化。其状态π演算语义如图 3.25 所示。图 3.25 中的 *Buffer* 进程实现了一先入先出的服务执行请求队列。每个 $Invoke_k$ 不再直接通过其 *start* 和 *done* 端口进行服务调用活动的触发，而是从 *Buffer*[*k*]（代表管道执行的第 k 步服务调用所读取的缓冲）中获取所需的服务执行请求，并在服务调用完成后在 *Buffer*[*k+1*]中存入对后续服务的调用请求。注意此处的缓冲深度 n 和 m 是有限的，以满足状态π演算进程的有穷性。其中，*bSize* 命题记录了当前缓冲的大小变化，它将在下一章中用于检验变量垃圾回收的规范语义约束。

$$Buffer_{[K]0}(set_k, get_k) =_{def} set_k(done_1)\{++, \{bSize_k, \{done_1\}\}\}.Buffer_{[K]1}(set_k, get_k, done_1)$$

$$Buffer_{[K]1}(set_k, get_k, done_1) =_{def} \overline{get_k} < done_1 > \{--, \{bSize_k, \{done_1\}\}\}.Buffer_{[K]0}(set_k, get_k)$$
$$+ set_k(done_2)\{++, \{bSize_k, \{done_2\}\}\}.Buffer_{[K]2}(set_k, get_k, done_1, done_2) +$$
$$.....$$

$$Buffer_{[K]n}(set_k, get_k, done_1, ..., done_n) =_{def}$$
$$\overline{get_k} < done_1 > \{--, \{bSize_k, \{done_1\}\}\}.Buffer_{[K]n-1}(set_k, get_k, done_2, ..., done_n)$$

$$Buffer_{[K]} =_{def} Buffer_{[K]0}(set_k, get_k)$$

$$Buffer_{[K+1]0}(set_{k+1}, get_{k+1}) =_{def} set_{k+1}(done_1)\{++, \{bSize_{k+1}, \{done_1\}\}\}.Buffer_{[K+1]1}(set_{k+1}, get_{k+1}, done_1)$$

$$Buffer_{[K+1]1}(set_{k+1}, get_{k+1}, done_1) =_{def}$$
$$\overline{get_{k+1}} < done_1 > \{--, \{bSize_{k+1}, \{done_1\}\}\}.Buffer_{[K+1]0}(set_{k+1}, get_{k+1})$$
$$+ set_{k+1}(done_2)\{++, \{bSize_{k+1}, \{done_2\}\}\}.Buffer_{[K+1]2}(set_{k+1}, get_{k+1}, done_1, done_2) +$$
$$.....$$

$$Buffer_{[K+1]m}(set_{k+1}, get_{k+1}, done_1, ..., done_n) =_{def}$$
$$\overline{get_{k+1}} < done_1 > \{--, \{bSize_{k+1}, \{done_1\}\}\}.Buffer_{[K+1]m-1}(set_{k+1}, get_{k+1}, done_2, ..., done_n)$$

$$Buffer_{[K+1]} =_{def} Buffer_{[K+1]0}(set_{k+1}, get_{k+1})$$

$$Invoke_k(get_k, set_{k+1}, get, port_1, port_2, done) =_{def}$$
$$get_k(d).get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{set_{k+1}} < done > .Invoke_k$$

图 3.25 缓冲执行管道模式的状态π演算语义

3.6.6　多实例的管道执行模式及其状态π演算语义

在多实例管道执行模式（Superscalar）中，当管道执行发生拥堵时，则后续服务将动态生成新的实例以接受前一服务对自己发出的执行请求。这意味着需要对服务调用活动也实现如图 3.26 所示的多实例化描述：

$$Invoke_1(start, get, port_1, port_2, done, syn, ack) =_{def}$$
$$start.(Invoke_2 \mid get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{syn}.ack.\overline{done})$$
$$Invoke_2(start, get, port_1, port_2, done, syn, ack) =_{def}$$
$$start.(Invoke_3 \mid get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{syn}.ack.\overline{done})$$
$$.....$$
$$Invoke_k(start, get, port_1, port_2, done, syn, ack) =_{def} \quad k\text{为有限常数}$$
$$start.get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{syn}.ack.\overline{done}$$

图 3.26 多实例执行管道模式中服务调用活动的多实例化描述

与 3.2.2 小节中的服务多实例方式类似，图 3.26 通过进程的前置嵌套实现了对服务调用活动请求（*start*）的并发监听，并根据收到的新请求进行相应的

服务执行操作（*port*）。为了满足并发进程的有穷性，此处限定 *k* 为有限常数。另外在以上 *Invoke* 进程中，新增了对 *syn* 和 *ack* 端口的顺序操作。它们将与额外的同步器（*Synchonizer* / *NonSynchonizer*）进程通讯，从而进一步区分同步和非同步多实例管道执行模式在语义的细微差别。在同步多实例管道执行模式中，下一（多实例）服务调用活动必须在本次服务调用活动所有实例的完成得到同步后才能进行。而非同步多实例管道执行模式则无此约束。下面的 *Synchonizer* 和 *NonSynchonizer* 进程分别实现了这两种不同语义，其中的自由名 *syn* 和 *ack* 应在它们与以上 *Invoke* 进程的组合中进行限定，以防止 *syn* 和 *ack* 信号的丢失。

$$Synchronizer(syn,ack) =_{def} \underbrace{syn.....syn}_{k}.\underbrace{\overline{ack}.....\overline{ack}}_{k}.Synchronizer$$

$$NonSynchonizer(syn,ack) =_{def} syn.\overline{ack}.NonSynchonizer$$

与以上服务调用的多实例化实现相对应的，对实际服务执行的多实例化和共享变量进程的语义实现则可以参见已在 3.2.2 小节中给出。

3.6.7　流方式的管道执行模式及其状态π演算语义

与以上管道执行模式不同，流方式的管道执行模式（Streaming）要求当管道执行发生拥堵时：（1）后续服务的执行请求接收不能受阻（Non-Blocking）；（2）后续服务不能等到当前调用完毕后才开始处理下一个调用请求（Non-Buffering）；（3）后续服务需要有在同一实例的执行过程中接收多个请求并产生相应输出的能力（Non-Superscalar）。这意味着服务调用活动需要有更强大的输入输出收集机制，而不是简单的 *start*，*done* 触发。由此，在图 3.27 对流方式管道执行模式的形式化中，进一步将服务调用活动的状态π演算语义分成三部分：输入集合（*InPinSet*）；内部调用（*InternalInvoke*）和输出集合（*OutPinSet*）。

图 3.27 中状态π演算语义与缓冲执行管道模式的区别在于在 *InternalInvoke* 进程的同一次调用过程中，服务调用活动的输入集合仍可以不断收集相应的服务调用请求。且此时输入集合将主动地通知（*notify*）*InternalInvoke* 进程是否有新服务调用请求的到达，并等待 *InternalInvoke* 进程的确认（*ack*）。当 *InternalInvoke* 进程完成对当前请求的处理后，会向输入集合发出重新初始化的信号（*reinit*）以方便输入集合再次通知（*notify*）是否还存在新的未处理服务调用请求。整个输出集合则始终监听 *InternalInvoke* 进程对服务的当前调用状态并

时刻准备向后续服务发出调用请求。同样，此处的输入集合大小 n 是有限的，以满足以上状态π演算进程的有穷性。其中，*bSize* 命题用于记录输入集合的大小变化，它将在下一章中用于对变量垃圾回收的规范语义约束进行检验。

$$InPinSet_0(in, notify, reinit) =_{def}$$
$$in(inp_1)\{++, \{bSize, \{inp_1\}\}\}.\overline{notify} < inp_1 > .InPinSet_1(in, inp_1) + reinit.InPinSet_0(in)$$
.....
$$InPinSet_n(in, notify, reinit, ack, inp_1, ..., inp_n) =_{def}$$
$$ack\{--, \{bSize, \{inp_n\}\}\}.InPinSet_{n-1}(in, inp_1, ..., inp_{n-1}) +$$
$$reinit.\overline{notify} < inp_n > .InPinSet_n(in, inp_1, ..., inp_n) +$$
$$in(inp_{n+1})\{++, \{bSize, \{inp_{n+1}\}\}\}.InPinSet_{n+1}(in, inp_1, ...inp_{n+1})$$
$$OutPinSet(receive, done) =_{def} \ receive.(\overline{done} \mid OutPinSet)$$
$$Invoke(in, get, port_1, port_2, done) =_{def} \ (new\ notify\ ack\ reinit\ receive)$$
$$(InPinSet_0 \mid InternalInvoke \mid OutPinSet)$$
$$InternalInvoke(get, port_1, port_2) =_{def} \ notify(inp).\overline{ack}.$$
$$(get(v:t).\overline{port_1} < v:t > .port_2(s).\overline{receive} \mid \overline{reinit}.InternalInvoke)$$

图 3.27 流方式执行管道模式的状态π演算语义

3.7 基于状态π演算的形式化结果与优势讨论

以上基于本文所提出的状态π演算给出了网格服务流中从单一服务模型到服务流协作到相关并发与管道模式的完整形式化语义。从另一个角度来看，这些工作不仅检验了状态π演算自身有着足够强的表达能力来对以上网格服务流规范和模式做出对应的形式化，同时也展示了状态π演算的以下两个优点。

1) **形式化描述的简化：**相对于基本π演算的纯行为描述，状态π演算由于实现了对状态命题的灵活抽象及其对进程演化的逆向约束，因此更方便了对网格服务流规范与模式中状态约束语义的形式化。以 3.6.3 小节中尽力执行的管道模式为例，它要求“当管道执行由于后续服务尚未完成而发生拥堵时，则前一服务应直接放弃尝试对后续服务发出执行请求”。在这里“后续服务尚未完成”是对当前状态的约束，而“放弃尝试对后续服务发出执行请求”则是在该约束下所应执行的动作或产生的事件。因此，正由于状态π演算对状态语义的结合使它可以通过$[\phi]\overline{done}$（ϕ表示 $eval(S_{k+1}.Status, Exit)$）的简单方式直接对它的语义做出形式化。相反的，同样为了对以上状态约束进行

刻画，使用基本π演算的一种可行方法则需要额外增加一个甚至多个变量或堆栈进程对后续服务的完成作出“记录”和“查询”，并根据查询结果控制对后续服务调用请求 *done* 的输出。与 3.6.3 小节的简单实现相比，此处需要额外增加的查询进程和控制逻辑如下。其中，$port_1^{k+1}$ 和 $port_2^{k+1}$ 分别指后续服务（$k+1$）的两个 port 端口，$Exited_{k+1}$ 和 $Occupied_{k+1}$ 进程分别用来记录后续服务的当前状态，而 *query* 则是状态查询的端口。

$$Invoke_k(start, get, port_1, port_2, done, query, free, occ) =_{def} start.\ldots.(\overline{query}.(free.\overline{done}.Invoke_k + occ.Invoke_k))$$

$$Exited_{k+1}(query, free, occ, port_1^{k+1}, port_2^{k+1}) =_{def} query.\overline{free}.Exited_{k+1} + port_1^{k+1}.Occupied_{k+1}$$

$$Occupied_{k+1}(query, free, occ, port_1^{k+1}, port_2^{k+1}) =_{def} query.\overline{occ}.Occupied_{k+1} + port_2^{k+1}.Exited_{k+1}$$

可以理解，当以上状态语义约束更为复杂时（如有多个命题需要查询和控制），则对应的额外进程实现和控制管理将会更为复杂。

2) **业务逻辑描述的简化：**与基本π演算语义中进程根据“当前”动作做出演化相对，由于状态π演算所实现的状态管理能力，使得在针对状态π演算形式化模型做出时序性质约束时，可以良好地利用到状态π演算进程演化过程中的历史动作信息。这些历史信息抽象了动作间隐含的时序关系，以状态的形式有效简化了对网格服务流中控制流/数据流业务逻辑约束的描述。以 3.2.1 和 3.2.2 小节中服务执行的任务状态和多实例化为例，我们期望描述两服务间如下的业务逻辑关系：“在任意情况下服务 *A* 的执行退出时，则服务 *B* 必定也在退出状态”。由于服务的退出（*Exit*）受动作 $port_1$ 和 $port_2$ 的影响，即服务执行通过端口 $port_2$ 退出，通过端口 $port_1$ 再次生成新的实例并开始等待执行（*Pending*），因此对于 3.2.1 和 3.2.2 小节实现的状态π演算语义只需用简单的分支时序逻辑就可以描述：AG (*A.Status* = *Exit* → *B.Status* = *Exit*)。它只需要了一个 AG 算子。但若没有对服务执行状态的形式化语义支持，则只能通过 $port_1$ 和 $port_2$ 动作间的时序关系实现以上业务逻辑的描述。由于服务的执行允许多实例化（即 $port_1$ 和 $port_2$ 动作可能重复发生多次），且此时无法利用原有 *Exit* 状态对 $port_1$ 和 $port_2$ 动作间隐含时序关系的表示（见图 3.28），因此对相同业务逻辑描述的复杂度将相应上升。实际上，若记服务 *A* 进程的 *port* 端口为 $port_{A1}$、$port_{A2}$，服务 *B* 进程的所有动作集合为 $\eta=\{port_{B1}, execute_B, get_B, set_B, port_{B2}\}$，则从动作执行的角度该业务逻辑的描述需要转化为：“任意情况下在 $port_{A1}$ 和 $port_{A2}$ 动作产生的时序间隔内，不会

发生任何服务 B 的执行动作 η”。因此根据时序性质模式[56]（见 1.4.4 小节综述）中的 *Absence* 模式定义可知，它所对应的π演算时序逻辑[80]（π Logic）描述将会变为（其中，A[x W y]等价于¬E[¬y U (¬x ∧ ¬y)]）：

$$\mathrm{AG}(port_{A1} \wedge \neg port_{A2} \rightarrow \mathrm{A}[(\neg(port_{B1} \vee execute_{B}, \vee get_{B}, \vee set_{B}, \vee port_{B2}) \vee \mathrm{AG}(\neg port_{A2})) \mathrm{W}\ port_{A2}])$$

以上描述既不利于对网格服务流业务逻辑自身的理解，同时也变得更复杂和难以理解。这里的复杂性源于此时瞬态的动作信息成为了唯一可用于描述系统行为关系的信息。可以理解，当以上业务逻辑中涉及更多服务执行的任务状态时（如 AG ((*A.Status* = *Exit* ∧ *C.Status* = *Exit*)→ *B.Status* = *Exit*)），则若没有状态π演算对状态信息进行灵活抽象的帮助，其对应基于动作的业务逻辑描述将会进一步复杂化。

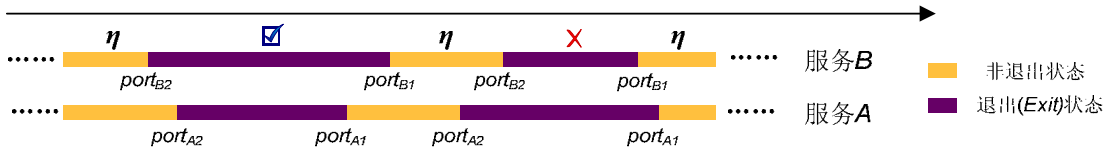


图 3.28 服务执行关系的业务逻辑描述示例

3.8 小结

本章的主要贡献可以总结为两方面。一方面，本章中基于有穷状态π演算进程给出了网格服务流从单一服务模型到服务流协作到相关并发与管道模式的完整形式化语义。这其中不仅包括服务执行、多实例化、服务选择、Condor DAGMan 和 BPEL4WS (1.1)中的基本服务活动、变量、堆栈及其控制流和数据流关系，还包含了错误补偿处理、全局终止等复杂特性语义。此外，本章基于状态π演算，更对网格服务流中目前缺乏形式化语义支持的各类并发与管道执行模式建立了形式化语义基础。另一方面，通过本章工作结果不仅验证了状态π演算对于网格服务流的形式化建模确实拥有足够强的表达能力，同时也揭示了它拥有简化服务流形式化建模和相应业务逻辑描述的优点。

本章工作是对上一章状态π演算理论模型的实际应用，它同时也为下一章所建立的网格服务流形式化验证方法提供了待验证的形式化对象。

第四章　网格服务流的状态π演算形式化验证

4.1　本章引论

形式化方法不仅是系统规范化描述的数学语言，更是实现其性质验证的技术和工具。通过第 2 章对状态π演算的提出和第 3 章对网格服务流的有穷状态π演算形式化，本章在此基础上继续研究对网格服务流的动/静态形式化验证方法。如 1.3.3 小节的引言所述，此处的验证工作涵盖了以下四方面内容：结构验证；服务流规范的语义约束验证；用户的业务逻辑需求验证和业务逻辑的一致性检验。本章实现了状态π演算语义在状态标号迁移系统上的解释和强/弱状态断言方法的提出。此外，通过实现状态π演算自身的模型验证支持和重用，解决了网格服务流的动/静态业务逻辑验证，及待验证业务逻辑自身的一致性检验。

正如 1.4.3 小节的综述所述，Dam 的证据系统[135]和等价自动机的转换[80]是实现基本π演算的模型验证支持及其在 Web 服务组合等领域进行应用的基本思路和手段。而本章中基于状态π演算所实现的网格服务流形式化验证方法则不仅有着能独立于特定模型验证技术的优点，且通过事先的强/弱状态断言和业务逻辑一致性检验可以有效避免不必要的验证代价。而状态π演算自身对系统状态的灵活抽象同时也简化了对本方法中对业务逻辑性质的描述。

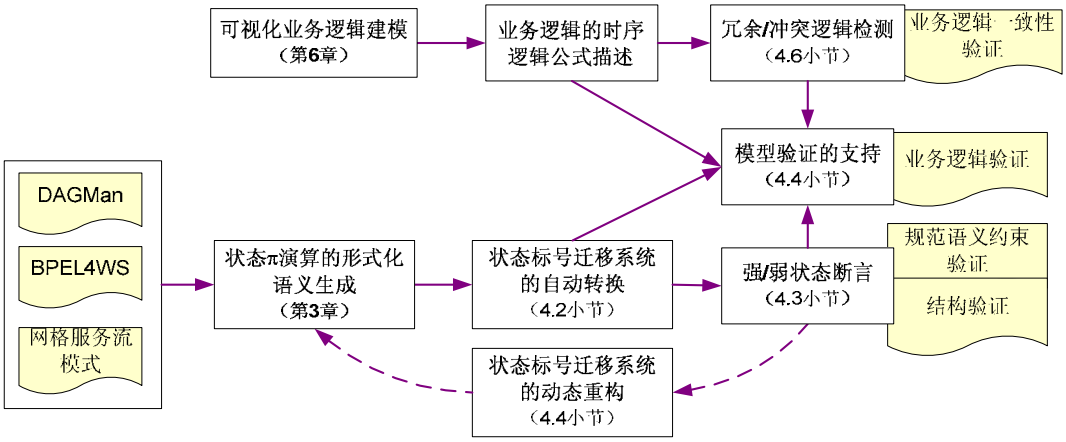


图 4.1　网格服务流的状态π演算形式化验证方法框架

如图 4.1 的方法框架所示，在已有网格服务流的状态π演算形式化语义基础

上，本章工作将围绕以下关键部分展开。4.2 和 4.3 小节中根据状态π演算语义实现了对应状态标号迁移系统的生成，并通过状态断言的提出实现了网格服务流的结构验证和服务流规范的语义约束验证；4.4 小节通过对状态π演算语义的动态重构和模型验证支持实现了对网格服务流的动/静态业务逻辑验证；4.5 小节总结了本章形式化验证方法的整体优点和特点；4.6 小节则基于时序逻辑与（扩展）Büchi 自动机的映射关系，进一步灵活复用了模型验证的思路对业务逻辑间潜在的冲突和冗余进行了检验；4.7 小节是本章工作的小结。

与本章结果对应的原型系统实现 GridPiAnalyzer 将在第 6 章给出详细介绍。

本章以下内容的相关研究结论主要发表在：

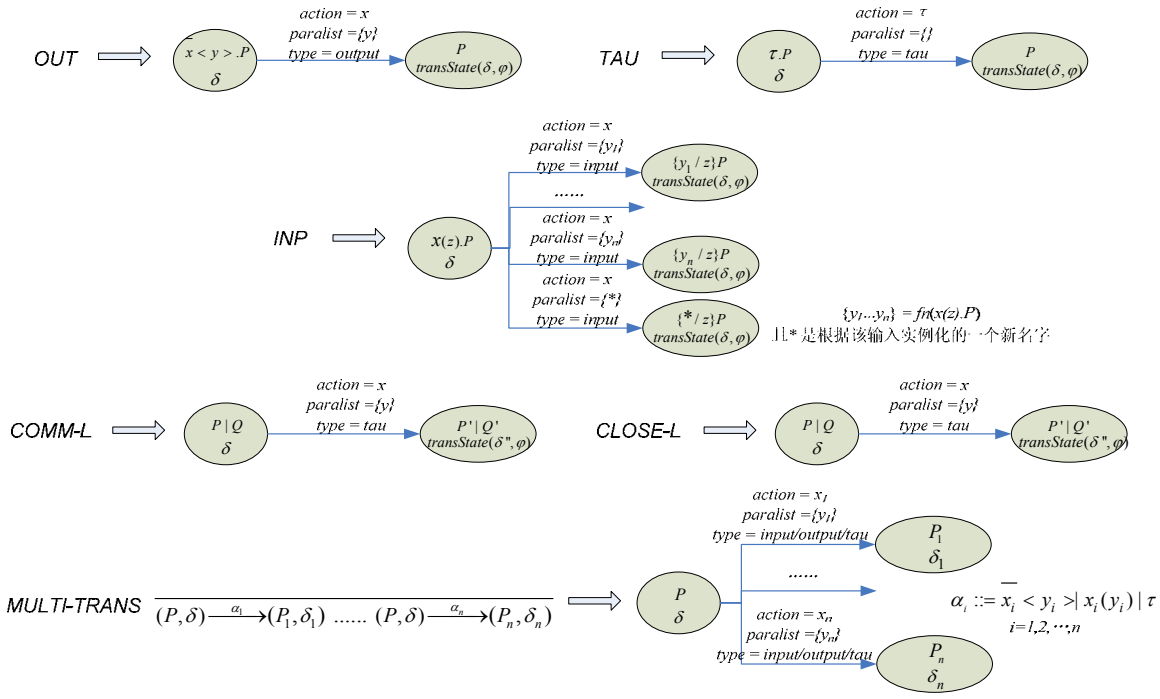
- A Static Compliance Checking Framework for Business Process Models. *IBM Systems Journal*, 2007, 46(2), 335-362 （**SCI** 检索）
- 网格服务链模型的验证分析技术及应用. *中国科学 F 辑: 信息科学*, 2007, 37(4): 467-485 （**SCI** 检索）

  Formal Verification Technique for Grid Service Chain Model and its Application. *Science in China, Series F: Information Sciences*, 2007, 50(1): 1-20（**SCI** 检索）
- A Three Layered Method for Business Process Discovery and its Application. *Computers in Industry*, 2007, 58(3): 265-278 （**SCI** 检索）
- BPSL Modeler - Visual Notation Language for Intuitive Business Property Reasoning. In: *Graph Transformation and Visual Modelling Techniques*, To appear in: *Electronic Notes in Theoretical Computer Science*, 2006: 205-214

4.2　从状态π演算到状态标号迁移系统

通过 2.3 小节的结论可知，状态π演算中进程的行为和状态的管理是解释在一个状态标号迁移系统上的。对于网格服务流的状态π演算形式化语义，它在状态标号迁移系统上的解释相当于实现了对服务流自身的完整行为及其状态空间的推演。因此，在图 4.1 所示的本章验证方法中，首先的一个关键步骤就是从网格服务流的状态π演算形式化语义到其对应状态标号迁移系统的自动转换。该步骤的作用不仅在于通过语义的推演过程完成后续对网格服务流自身命题性质的分析，同时也使现有各类模型验证技术可以无缝地集成到本章状态π演算的形式化验证框架中来。

在 1.4.3 小节的综述中已经提到，基本π演算进程的行为可以解释在一个普通标号迁移系统上，且 Ferrari[80]和 Pistore[147]等人已经证明了任何有限π演算进程通过其早迁移语义可以对应转换为一个等价的普通标号迁移系统。这在本质上是从一个基于名字的形式化理论（Name-based formal theory）到一个无名字的形式化理论（Nameless formal theory）的过渡。然而对于状态π演算，虽然其操作语义同样基于了π演算的早迁移语义（见 2.3 小节），但此处仍然需要额外处理其对状态标号迁移系统的扩展，即在表示进程演化的动作标号创建过程中同时完成相应对状态标号的创建与管理。这本身也是由状态标号迁移系统的双重标号特性（动作标号+状态标号）所决定的。因此，根据 2.3 小节中状态π演算的扩展操作语义，以下归结了状态标号迁移系统转换的关键迁移转换规则。图 4.2 中的各操作语义（即：*OUT*、*INP* 等）可详见 2.3 小节，此处不再赘述。



SUM-L，*PAR-L*，*RES*，*OPEN*，*REP-ACT*，*REP-COMM*，*REP-CLOSE*可由以上规则类推得到，在此不再赘述。

图 4.2 状态标号迁移系统生成的语义转换规则

图 4.2 中的迁移转换规则完整遵守了 Ferrari 对关键的输入前缀 *INP* 的语义转换，即在输入迁移过程中同时考虑当前进程所包含的所有自由名及由该输入所引入的一个新名字的实例化。状态标号迁移系统中的每一个迁移由状态π演算的相应动作所触发。其中，三个额外的状态π演算保留命题*#action*、*#paralist*

和*#type* 被用来记录必要的动作信息。*#action* 命题用来记录动作的端口名 *x*；*#paralist* 命题记录了通过端口所传输参数的集合{*y*}；*#type* 则用来记录相应的动作是一个输入动作（*input*）、输出动作（*output*）还是一个不可见动作（*tau*）。

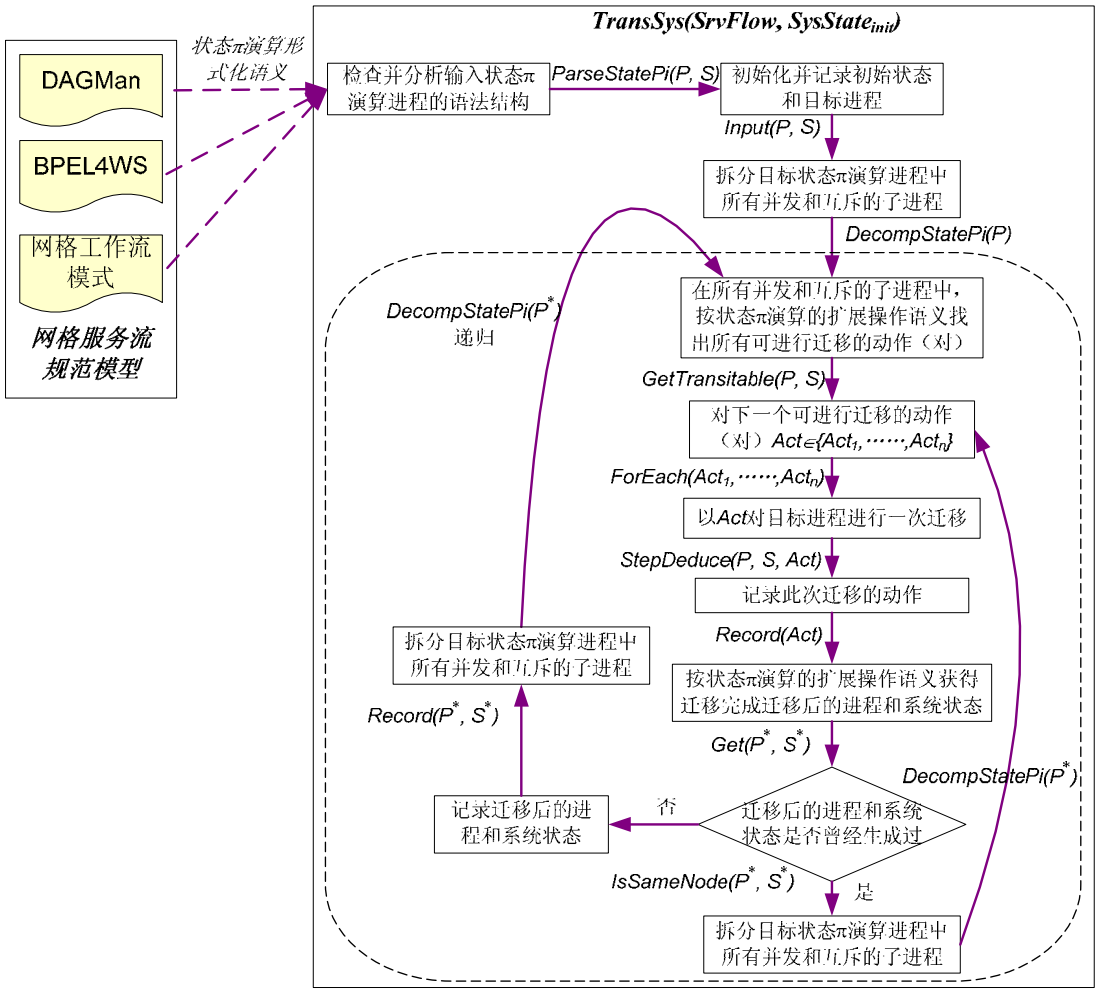


图 4.3　状态标号迁移系统转换算法流程

基于以上结果，图 4.3 由此给出了完整的状态标号迁移系统转换算法流程 *TransSys(SrvFlow, SysState$_{init}$)*。在图 4.3 中，*ParseStatePi* 步骤负责根据状态π演算语法结构对输入的状态π演算进程进行语法解析；*DecompStatePi* 步骤负责对当前状态π演算进程中可并行（|）和互斥（+）的子进程进行拆分；*GetTransitable* 步骤负责解析出当前状态π演算进程中所有可能的执行动作；*StepDeduce* 步骤遵循图 4.2 的语义转换规则根据状态π演算当前可能的执行动作为状态标号迁移系统生成对应的新节点；*IsSameNode* 步骤负责判断当前 *StepDeduce* 为状态标号迁

移系统生成的节点是否已经生成过。

需要注意，图 4.3 与 Ferrari 等人方法的不同点是在状态标号迁移系统中，它不单纯依赖当前进程来唯一标识标号迁移系统中的（状态）节点，而同时需要结合当前进程所关联的状态命题。因此在图 4.3 的 *IsSameNode* 步骤中不仅需要考虑进程间的同余，而且需要判别其状态是否同时相等。这一点是状态π演算对状态生命周期管理能力扩充后的一个直接结果。因此，为了检验以上对状态扩展部分转换的有效性，下面进一步讨论了状态π演算中由于其状态操作扩展所生成对应状态标号迁移系统的有穷性。

性质 4.1：记状态标号迁移系统 $SLTS=(S, M, \{\xrightarrow{a\{StateExpr\}} \mid a \in M\})$，则有穷状态π演算进程所对应 *SLTS* 的规模 *n*（即图 4.3 算法所生成的节点数）也将是有限的。

证明：由 Pistore[147]等人的结论，有穷π演算进程其可达的并发子进程数必是有限的，因而在进程演化过程中产生的中间进程 *P* 的个数和对应的等价标号迁移系统亦有限。然而在状态π演算中即使针对同一个中间进程 *P*，由于演化到 *P* 的动作序列不同而对于同一个 *P* 会关联不同的当前状态，即：∃ (*P, SysState*), (*P', SysState'*) ∈*SLTS*，s.t. $P \equiv P'$但 $SysState \neq SysState'$。记$\rightarrow^{\alpha}$表示由图 4.2 中迁移转换规则所形成的一有限动作序列$\alpha=\{\pi_1\{StateExpr_1\},..., \pi_n\{StateExpr_n\}\}$。则以下讨论了对于有限的中间进程 *P* 在不同情况下所可能关联的有限状态数。

1）非嵌套进程：若∃不同的α_1，α_2，……α_m（$m \geq 1$），s.t. (*P, SysState*) $\rightarrow^{\alpha i}$ (*P', SysState'*)且 $1 \leq i \leq m$、$P \neq P'$，则由有穷π演算进程对应的有限等价标号迁移系统可知，此处 *m* 必为有限值。且由于状态状态π演算中状态操作和动作在语法上有着静态的一一对应关联，因此在最坏情况下 *P'*关联 *m* 种可能状态；

2）串行嵌套进程：若存在唯一的α，s.t. (*P, SysState*) $\rightarrow^{\alpha}$ (*P, SysState'*) $\rightarrow^{\alpha}$……$\rightarrow^{\alpha}$(*P, SysState"*)，则由状态π演算中状态操作和动作在语法上的一一对应以及状态操作的语义可知，必有 *SysState'*= *SysState"*成立。因此在最坏情况下 *P* 关联 2 种可能状态 *SysState* 和 *SysState'*；

3）并行嵌套进程：若∃α_1，α_2，……α_m（$m \geq 1$），s.t. (*P, SysState*) $\rightarrow^{\alpha i}$ (*P, SysState'*) 且 $1 \leq i \leq m$，同时$\nexists \alpha' \subset \alpha_i$，s.t. (*P, SysState*) $\rightarrow^{\alpha'}$ (*P, SysState"*)，此时称α_1，α_2，……α_m为进程 *P* 形成嵌套的最短动作序列。由此则∃(*P, SysState*) $\rightarrow^{\alpha_1}$ (*P, SysState*$_1$) $\rightarrow^{\alpha_2}$……$\rightarrow^{\alpha_m}$(*P, SysState*$_m$)。然而通过有穷π演算进程对应的有限等价标号迁移系统可知，*m* 自身必定有限，且α_i（$1 \leq i \leq m$）均为有限长动作序列。因此根据状态π演算的状态操作语义可知，α_i所能操作的状态命题（*prop*）

的最大数量（p）和命题的取值范围集合 *range*(*prop*)大小（r）定为有限的，即在最坏情况下 P 关联 $\prod_{i=1}^{p} r_i$ 种可能状态。证毕。 □

由以上讨论可知，针对第 3 章中网格服务流的（有穷）状态π演算形式化语义，其对应的状态标号迁移系统规模将由状态π演算进程演化过程中的中间进程 P 及其可能关联的不同系统状态数同时决定。关于状态标号迁移系统转换的实例及其性能的讨论将在后续的 4.4 小节与第 5 章的验证性能改进中详细给出。

4.3　网格服务流的结构与规范语义约束验证

网格服务流在客观上的结构与规范语义约束检验是本章形式化验证内容中的一个基本环节。网格服务流的结构验证保证了其在可达性（Reachability）和可终止性（Termination）两方面的基本性质。更具体的，给定一个网格上下文环境（*Context* 进程），网格服务流的可达性用于检验在一组给定需要执行的服务集合 *SS* 中，是否存在某服务由于该网格服务流在控制流/数据流上的约束而始终无法得到执行；网格服务流的可终止性用于检验对于既定的服务流终止条件 *TC*，它在网格服务流的执行过程中在任意情况下最终是否一定会得到满足。网格服务流的可终止性本质上是一种活性性质。

另一方面，网格服务流的规范语义约束验证确保了网格服务流模型自身不会违反其所依赖的服务流规范所要求的语法和语义约束。在这些约束中，由于部分约束可以从语法层面就得到直观而方便的检验（如表 4.1 所列举的），因此本章工作主要关注两类尚不能从语法层面得到直接验证的规范语义约束：消息竞争冲突和变量垃圾回收约束。更具体的，BPEL4WS 规范中显示地声明了对消息竞争冲突的防止:“任一服务流实例不得同时触发两个或两个以上接收活动来监听由同一端口发出的事件”。这里的接收活动包括 3.2.3 小节中的基本接收活动（*Receive*）和选择结构（*Pick*）中的事件触发。而“端口”则指在它们的状态π演算进程中由 partnerLink、portType、operations 所组成的接收动作“*port*”；变量垃圾回收约束指的是在网格服务流的执行过程中，所有的临时变量进程（如：3.2.3 小节中的变量堆栈进程；3.6.5 小节缓存管道执行模式中的缓存进程，3.6.7 小节流方式管道执行模式中的输入集合）在服务流终止时其大小均应为空，以保证在执行过程中不会产生多余的，不被任何其它进程所消化的消息和数据。

表 4.2 常用的规范语法约束列表

名称	说明	来源
连接类型约束	不同活动类型所能关联的控制流/数据流关系约束	UML2.0 活动图 / WBI模型①
数据流类型和数量约束	数据流关系在传输资源类型和数量上的匹配约束	UML2.0 活动图 / WBI模型
冗余输入输出约束	检查活动中是否存在无控制流/数据流关系的输入输出	UML2.0 活动图 / WBI模型
活动开始条件约束	从语法上静态检验服务活动的开始条件（如输入数据、资源的数量和类型要求等）	WBI模型
连通路径约束	检查流程中是否存在不连通的路径	WBI模型
循环路径约束	检查流程中是否存在循环路径	WBI模型
分支条件的互斥约束	检查特定分支条件的互斥性	WBI 模型 / BPEL4WS
*Link*的自环约束	*link*结构不能形成自环的约束	BPEL4WS
*Scope*范围约束	特定活动和结构类型（如*link*等）不能跨范围定义	BPEL4WS

在给出网格服务流可达性、可终止性和规范语义约束的验证之前，本节先对其对应状态π演算进程的执行做出必要的定义。

定义 4.1（执行）：二元组(α, β)被称为是一个状态π演算进程(P, S)的执行，若：

- α是一个有限的有序状态π演算动作序列$\alpha=\{\pi_1\{StateExpr_1\}, \pi_2\{StateExpr_2\}, ..., \pi_n\{StateExpr_n\}\}$；
- β是与α对应的一个有限状态序列$\beta=\{S_1, S_2, ..., S_n\}$；
- 有 $(P,S) \xrightarrow{\pi_1\{StateExpr_1\}} (P_1,S_1) \xrightarrow{\pi_2\{StateExpr_2\}} (P_2,S_2) ..... \xrightarrow{\pi_n\{StateExpr_n\}} (P_n,S_n)$ 成立，其中$(P_i, S_i) \neq (P_j, S_j)$，即：$P_i \not\equiv P_j$或$S_i \neq S_j$（$1 \le i \le n$，$1 \le j \le n$，$i \neq j$）。

此时，称 P 和 S 分别为执行(α, β)的初始进程与状态，P_n和S_n分别为(α, β)的结束进程与状态。

定义 4.2（可接受的执行）：对于状态π演算进程(P, S)，(α, β)是它的一个可接受执行，若(α, β)是它的执行，且不存在其它执行(α', β')使得有$\alpha \subset \alpha'$，$\beta \subset \beta'$成立。

① IBM WebSphere Business Integration Modeler ™; 见: *http://www-306.ibm.com/software/integration/wbimodeler/library/*

由此，一个状态π演算进程(P, S)的可接受执行表示了它所对应状态标号迁移系统中，在不包含循环迁移情况下的一次以(P, S)为起点的最长迁移过程。以下性质给出了一个状态π演算进程的执行是其可接受执行的三个条件。

性质 4.2：一个状态π演算进程(P, S)的执行(α, β)是其可接受执行，当且仅当(α, β)的结束进程 P_n 满足：（1）是一个空进程“0”；或（2）P_n 无法再根据状态π演算的扩展操作语义进行任何动作迁移；或（3）P_n 的任一后续迁移形成到该执行(α, β)的中间进程 P_i（$1 \le i \le n$）的循环。

证明：由定义 4.2 直接得出。证毕。 □

以下将满足性质 4.2 中三条件的进程记为Φ，而记“—”表示一个任意的状态π演算进程。对于代表网格服务流形式化语义的（有穷）状态π演算进程，由于 4.2 小节已经完整刻画了其进程演化和状态操作的推演过程，因此通过判别状态π演算进程可接受执行中的状态命题，可以直接分析相应网格服务流的结构特性。本章中将该方法称为状态π演算的强/弱状态断言方法。

定义 4.3（强状态断言）：记$(P, S) \models \lceil Sc \rfloor_{(P', S')}$，若对($P$, S)的任意可接受执行(α, β)，$\forall (P, S) \rightarrow^{\alpha^*} (P^*, S^*)$，其中$\alpha^* \subset \alpha$，使得当 $P^* \equiv P'$和 $S^* \rightarrow S'$成立时，必有 $S^* \rightarrow S_C$ 成立。此时称状态π演算进程(P, S)满足定义在目标(P', S')上的一个强状态断言 Sc。

定义 4.4（弱状态断言）：记$(P, S) \models \langle Sc \rangle_{(P', S')}$，若存在($P$, S)的一个可接受执行(α, β)，且$\exists (P, S) \rightarrow^{\alpha^*} (P^*, S^*)$，其中$\alpha^* \subset \alpha$，使得当 $P^* \equiv P'$和 $S^* \rightarrow S'$成立时，必有 $S^* \rightarrow S_C$。此时称状态π演算进程(P, S)满足定义在目标(P', S')上的一个弱状态断言 Sc。

例如，在下面的简单状态π演算进程示例(P, {})中，易得由于存在一可接受执行使得$(P, \{\}) \rightarrow^{\alpha^*} (0, \{InvokedSrv=\{Subtract\}\})$，因此$(P, \{\}) \not\models \lceil InvokedSrv = \{Sum\} \rfloor_{(0, true)}$，但$(P, \{\}) \models \langle InvokedSrv = \{Sum\} \rangle_{(0, true)}$成立。

$$P(port_{8001}, port_{8002}) =_{def}$$
$$\tau.(\overline{port_{8001}}\{+, \{InvokedSrv, \{Sum\}\}\}.0 + \overline{port_{8002}}\{+, \{InvokedSrv, \{Subtract\}\}\}.0)$$

可以看出，强/弱状态断言间的区别类似于时序逻辑中的全局路径算子 A 和存在路径算子 E。易得：$(P, S) \models \lceil Sc \rfloor_{(P', S')}$当且仅当$(P, S) \not\models \langle \neg Sc \rangle_{(P', S')}$。

以上状态断言的一个重要作用在于，一方面在图 4.3 的状态标号迁移系统转换的基础上可以方便地实现对它们的判别，另一方面通过状态π演算自身的强/弱状态断言判别也可以直接对相应网格服务流的可达性、可终止性和消息竞争冲突、变量垃圾回收的两类规范语义约束作出表示和检验。

定义 4.5（服务可达性）：给定网格服务流的对应状态π演算进程 *SrvFlow* 及其初始状态 $SysState_{init}$（详见 3.5 小节定义），则其中一组给定服务 $SRV=\{Srv_1,\ldots, Srv_n\}$ 为可达的，当$\forall Srv\in SRV$，$(SrvFlow, SysState_{init}) \vDash \langle Srv.Status = Exit\rangle_{(\Phi, true)}$均成立。

定义 4.6（可终止性）：给定网格服务流的对应状态π演算进程 *SrvFlow* 及其初始状态 $SysState_{init}$，则在终止条件 *TC* 下该服务流为可终止的，当$(SrvFlow, SysState_{init})\vDash \lceil Tc \rfloor_{(\Phi, true)}$成立。

定义 4.7（消息竞争冲突）：给定网格服务流的对应状态π演算进程 *SrvFlow* 及其初始状态 $SysState_{init}$，则其执行中不存在消息竞争冲突，当对于 *SrvFlow* 进程演化过程中的所有 *msgPort* 命题，有$(SrvFlow, SysState_{init}) \vDash \lceil |range(*.msgPort)|=0 \vee |range(*.msgPort)|=1 \rfloor_{(—, true)}$成立。

定义 4.8（变量垃圾回收）：给定网格服务流的对应状态π演算进程 *SrvFlow* 及其初始状态 $SysState_{init}$，则其执行中不存在垃圾变量，当对于 *SrvFlow* 进程演化过程中的所有 *bSize* 命题，有$(SrvFlow, SysState_{init})\vDash \lceil |range(*.bSize)|=0 \rfloor_{(\Phi, true)}$成立。

定义 4.9：称给定网格服务流的对应状态π演算进程 *SrvFlow* 及其初始状态 $SysState_{init}$ 在给定网格类型环境Γ（见 2.5 小节）下为可达、可终止、无消息竞争冲突和无垃圾变量的，当$\Gamma \rhd (SrvFlow, SysState_{init})$分别满足定义 4.5 至定义 4.8。

以上定义中，服务的可达性通过弱状态断言表示了所有给定的服务必会至少在一个网格服务流的状态π演算可接受执行中出现；而可终止性通过强状态断言表示了网格服务流的终止条件必会在其状态π演算进程的所有可接受执行中得到满足。另一方面，消息竞争冲突通过强状态断言表示了网格服务流中由 *msgPort* 命题（参见 3.2.3 和 3.4.5 小节中的形式化语义）所标识的任意相同消息端口不会被多个接收活动或选择结构同时监听；而变量垃圾回收通过强状态断言表示网格服务流的状态π演算进程经过任意可接受执行后，其所有的缓冲、变量和输入集合大小必为空（通过 *bSize* 命题得到，参见 3.2.3 和 3.6 小节中的形式化语义）。由此，以下的图 4.4 中在图 4.3 状态标号迁移系统自动生成的基础上扩展了其对状态π演算强/弱状态断言的判别，从而实现了对以上服务可达性、可终止性和两类规范语义约束的自动检验。其中，新的 $IsSatisfied((P^*, S^*), (P', S'))$和 $IsSatisfied(S^*, Sc)$分别用于实现对目标进程(P', S')和断言命题 S_C 的追加判别。类似的，关于强/弱状态断言方法的实例及其性能的讨论将在后续的 4.4 小节与第 5 章的验证性能改进中详细给出。

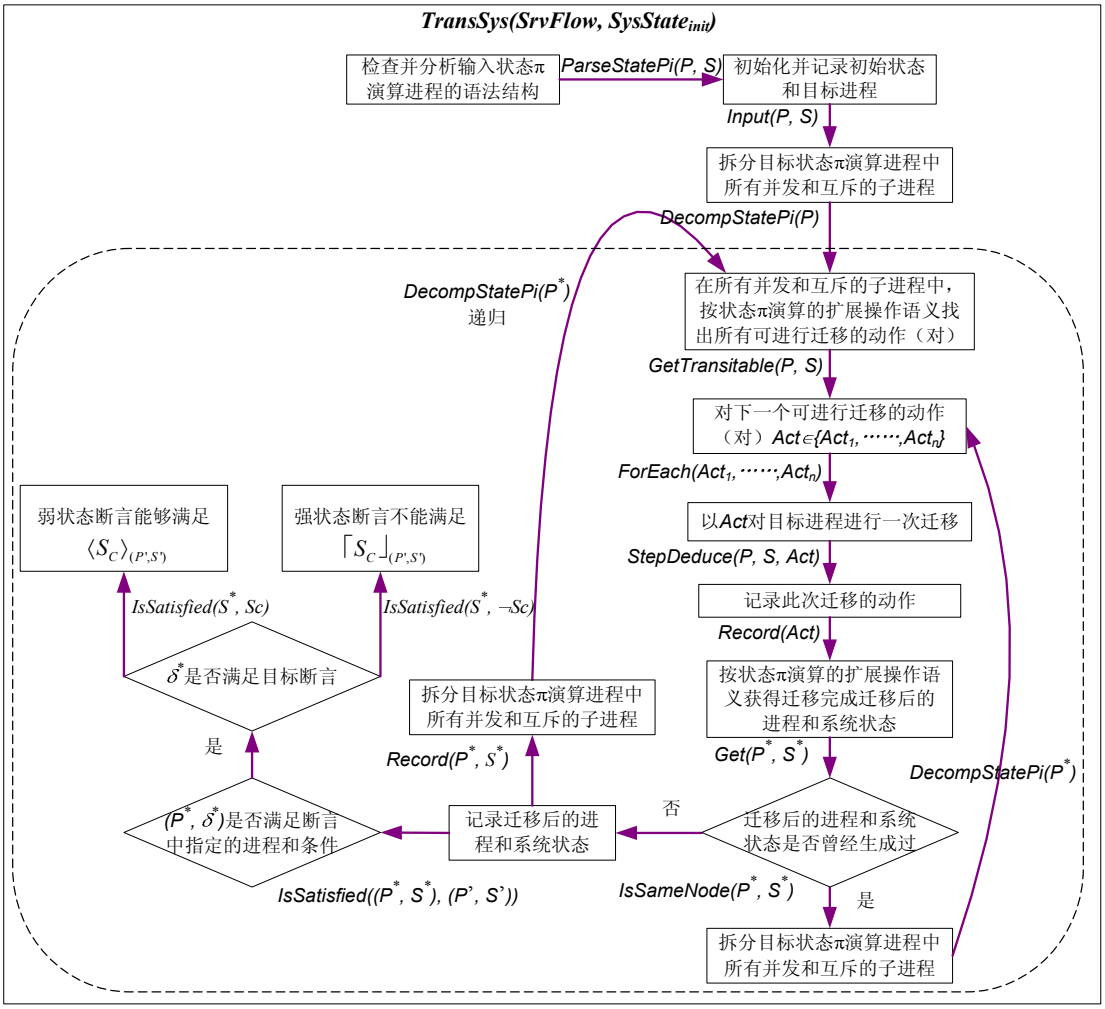


图 4.4　带状态断言的状态标号迁移系统生成

```
JOB initdata initdata.sub
RETRY initdata 0
JOB tmpltbankl1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankl1 0
VARS tmpltbankl1 macroframecache="cache/L-791592854-791607098.cache" macrochannelname="L1:LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB tmpltbankh1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh1 0
VARS tmpltbankh1 macroframecache="cache/H1-791592855-791607099.cache" macrochannelname="H1;LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB tmpltbankh2 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh2 0
VARS tmpltbankh2 macroframecache="cache/H2-791592856-791607100.cache" macrochannelname="H2:LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB inspiall1 inspiral_pipe.inspiral.sub
RETRY inspiall1 0
VARS inspiall1 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="L1-TMPLTBANK-791592862-2048.xml" macrochannelname="L1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/L-791592854-791607098.cache" macrosnrthreshold="7.0"
JOB trigbankh11 inspiral_pipe.trig.sub
RETRY trigbankh11 0
VARS trigbankh11 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H1:LSC-AS_Q"
JOB trigbankh12 inspiral_pipe.trig.sub
RETRY trigbankh12 0
VARS trigbankh12 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H1:LSC-AS_Q"
JOB inspiralh11 inspiral_pipe.inspiral.sub
RETRY inspiralh11 0
VARS inspiralh11 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H1-TMPLTBANK-791592863-2049.xml" macrochannelname="H1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H1-791592855-791607099.cache" macrosnrthreshold="7.0"
JOB inspiralh12 inspiral_pipe.inspiral.sub
RETRY inspiralh12 0
VARS inspiralh12 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H1-TMPLTBANK-791592864-2050.xml" macrochannelname="H1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H1-791592855-791607099.cache" macrosnrthreshold="7.0"
JOB sincalih1 inspiral_pipe.sinca.sub
RETRY sincalih1 0
VARS sincalih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB thincalih1 inspiral_pipe.thinca.sub
RETRY thincalih1 0
VARS thincalih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB trigbankh21 inspiral_pipe.trig.sub
RETRY trigbankh21 0
VARS trigbankh21 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2;LSC-AS_Q"
JOB trigbankh22 inspiral_pipe.trig.sub
RETRY trigbankh22 0
VARS trigbankh22 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2:LSC-AS_Q"
JOB trigbankh23 inspiral_pipe.trig.sub
RETRY trigbankh23 0
VARS trigbankh23 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2:LSC-AS_Q"
JOB InspVeto inspiral_pipe.veto.sub
RETRY InspVeto 0
VARS InspVeto macrocalibrationcache="cache_files/calibration.cache" macrochannelname="L1:LSC-AS_Q"
JOB inspiralh21 inspiral_pipe.inspiral.sub
RETRY inspiralh21 0
VARS inspiralh21 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H2-TMPLTBANK-791592865-2051.xml" macrochannelname="H2:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H2-791592856-791607100.cache" macrosnrthreshold="7.0"
JOB inspiralh22 inspiral_pipe.inspiral.sub
RETRY inspiralh22 0
VARS inspiralh22 smacrocalibrationcache="cache_files/calibration.cache" macrobankfile="H2-TMPLTBANK-791592866-2052.xml"
macrochannelname="H2:LSC-AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H2-791592856-791607100.cache" macrosnrthreshold="7.0"
JOB thinca2lih1 inspiral_pipe.thinca2.sub
RETRY thinca2lih1 0
VARS thinca2lih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB thinca2lih2 inspiral_pipe.thinca2.sub
RETRY thinca2lih2 0
VARS thinca2lih2 macroframecache="cache/L-791592857-791607101.cache, cache/H1-791592855-791607099.cache"
JOB returnres returnres.sub
RETRY returnres 0

PARENT initdata CHILD tmpltbankl1 tmpltbankh1 tmpltbankh2
PARENT tmpltbankl1 tmpltbankh1 tmpltbankh2 CHILD inspiall1
PARENT inspiall1 CHILD trigbankh11 trigbankh12 thincalih1
PARENT trigbankh11 CHILD inspiralh11
PARENT trigbankh12 CHILD inspiralh12
PARENT inspiall1 inspiralh11 inspiralh12 CHILD sincalih1
PARENT sincalih1 CHILD thincalih1 trigbankh21
PARENT thincalih1 CHILD trigbankh22 returnres
PARENT trigbankh21 CHILD inspiralh21
PARENT trigbankh22 CHILD inspiralh22
PARENT inspiralh21 inspiralh22 CHILD thinca2lih1
PARENT thinca2lih1 thincalih1 CHILD returnres
```

图 4.5 引力波探测数据分析网格服务流实例 SF1 及其 DAGMan 脚本

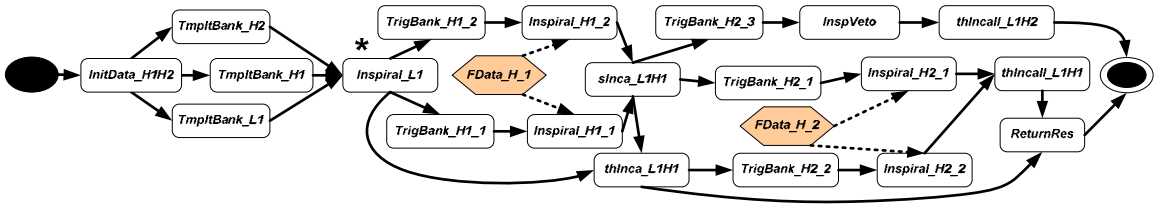


图 4.5（续） 引力波探测数据分析网格服务流实例 SF1 及其 DAGMan 脚本

4.4　网格服务流的业务逻辑验证及应用实例

如图 4.1 中的本章验证方法框架可知，通过上两个小节的工作一方面实现了对状态π演算在进程演化和状态操作语义的推演，完成了相应网格服务流形式化语义中基本命题性质的断言；另一方面同时也为模型验证技术的引入提供了重要的形式化模型输入。本节将通过实例讨论如何为状态π演算提供模型验证技术的支持，并由此实现对网格服务流的动/静态业务逻辑验证。

4.4.1　业务逻辑验证的服务流实例

为了帮助对以上验证步骤和本节的网格服务流业务逻辑验证方法进行理解，本小节首先给出了一个实际应用背景的实例：即基于 LIGO 数据网格的引力波（Gravitational Wave）探测数据分析服务流[27,49,50]。

引力波是物理学中大量的致密物质能量发生剧烈移动时所产生时空结构的波动，它的探测肩负着分析天体中未知物质活动与形成的重大科研使命。然而对引力波探测数据的分析是一个涉及多任务和海量数据的复杂过程。LIGO 数据网格正为引力波探测过程中多服务协作的自动化和海量引力波探测数据的处理提供了基础功能平台。图 4.5 中给出了实际 LIGO 项目中引力波探测数据分析的一个简单服务流实例[27,50]（记为 SF1）。它展示了该服务流的实际 DAGMan 脚本，并给出了其可视化表示。针对该应用更复杂的两个服务流实例（分别记为 SF2 和 SF3）及其详细的验证性能将在第 5 章中完整给出。

图 4.5 中的引力波探测数据分析包含以下的几个关键步骤。首先三个 LIGO 激光干涉仪(以下简称为干涉仪)分别进行各自的引力波信号数据采集(*FData*)。由于引力波信号多似正弦波，且随信号源的不同有着不同的频率，因此待分析的数据将通过傅立叶变换分解为更小的数据块，将原有的完整波形分解为更小

的，具有不同频率的正弦波。该过程称为数据的初始化（*InitData*）。根据信号源所产生的不同频率，通过现有的理论模型可以产生一系列与待探测的期望波形相匹配的模板库（*TmpltBank*），通过将分解的数据块与模板库中期望波形的分别比较来过滤不符合匹配阀值要求的数据（*Inspiral*）；对于满足匹配阀值的数据，需要进一步和其它干涉仪的探测结果进行偶然性分析（*sInca* 和 *thInca*），以检测尚未被过滤出的噪声数据；通过偶然性分析的信号则再次被放入波形模板库用于针对后续的分析操作进行性能上的优化（*TrigBank*）；由于实际的期望波形匹配方法在针对干涉仪稳定状态下采集的数据才能有较好的效果，因此在偶然性分析之后得到的信号还可能需要根据当时干涉仪的运作情况（如：环境条件，干涉仪控制参数等）来否决其中的不良信号（*InspVeto*）。具体的引力波探测数据分析中，该否决通常通过分析 L1 干涉仪[②]中额外信道的脉冲波形干扰来实现的；最后的信号将再次通过一次额外的偶然性分析（*thIncaII*）来确定最终可能是检测到引力波的候选信号。借由图 4.5 中服务流的执行，LIGO 用户可以获得选定天体范围内潜在的引力波数据信号，并通过进一步分析得出对相关天体活动现象的解释。

4.4.2　静态业务逻辑验证

引力波的探测数据分析是一项复杂的任务（该应用的两个完整的复杂服务流实例可参见第 5 章）。为了实现潜在引力波信号与噪声信号的区分，保证分析过程的有效性，整个引力波探测数据分析过程必须满足以下的关键业务逻辑：

业务逻辑 1（模板库生成的后续操作）：在任意情况下，一旦模板库生成服务（*TmpltBank*）执行完毕，则为了保证数据分析的有效性，一定会相应进行期望波形的匹配（*Inspiral*）和波形匹配的优化（*TrigBank*）这两个关键分析操作；

业务逻辑 2（干涉仪的工作状态约束）：由于不同干涉仪有着不同的灵敏度，因此当 H1 和 H2（Handford 的两个干涉仪）同时工作（*InitData_H1H2*）时，要求对 H2 数据的期望波形匹配（*Inspiral_H2*）必须挂起，直到 L1H1 的数据通过了偶然性分析处理（*sInca_L1H1* 和 *thInca_L1H1*）；

业务逻辑 3（偶然性分析的完整性）：对于三个干涉仪的收集数据，必须确保它们一定会经过所有的偶然性分析（*sInca*、*thInca* 和 *thIncaII*），以尽可能地减少最终分析的信号中噪声；且 *sInca* 和 *thInca* 分析必须都在 *thIncaII* 之前先进行；

② 以下的 L1、H1、H2 分别指 LIGO 项目中位于 Livingstone 和 Hanford 的三个激光干涉仪。

业务逻辑 4（偶然性分析的必然性）：在任意情况下，一旦数据进行期望波形匹配或模板库生成后，则相应的最终一定会进行最后的偶然性分析的处理。

本节中通过实现对状态π演算的模型验证支持来完成对以上 LIGO 应用中特定业务逻辑的验证。正如 1.4.2 小节的综述所述，模型验证技术是在有限状态系统模型上检查期望的时序逻辑性质是否成立的一种重要形式化验证技术。而在本章的网格服务流业务逻辑验证问题中，对应的有限状态系统模型则是由有穷状态π演算进程所表示的形式化模型，对应的时序逻辑性质则是以上的业务逻辑约束。由此，正如引言中图 4.1 的方法框架所示，本章通过状态π演算作为网格服务流的形式化工具（第 2 章），通过网格服务流的状态π演算形式化语义作为网格服务流形式化验证方法的基础（第 3 章），通过状态标号迁移系统的生成和强/弱状态断言作为对状态π演算模型提供模型验证技术支持的桥梁（4.2、4.3 小节），从而实现了一个基于状态π演算的网格服务流自动业务逻辑验证方法，并在原型系统 GridPiAnalyzer 中进行了实现（见第 6 章）。

更具体的，以下将基于状态π演算的网格服务流业务逻辑验证方法简称为 GridPiAnalyzer 方法，则 GridPiAnalyzer 方法首先根据本文第 3 章中已给出的形式化研究结果完成图 4.5 中 LIGO 引力波探测数据分析服务流的状态π演算语义形式化（具体的状态π演算语义可参见第 3 章，在此不再赘述）。进一步的，通过 4.2、4.3 小节方法则可以从该网格服务流的状态π演算语义转换得到对应的状态标号迁移系统。由于状态标号迁移系统显示地刻画了相应状态π演算进程的有限状态命题迁移过程，因而它直接作为 GridPiAnalyzer 方法中模型验证引擎③的有限状态系统模型输入。另一方面，GridPiAnalyzer 方法同时接受相应的时序逻辑公式作为输入，从而对以上 4 组业务逻辑约束的进行表述，并最终完成基于状态π演算的完整网格服务流业务逻辑验证流程。根据 1.4.2 小节综述中对线性时序逻辑（LTL）的优点分析，本节中将以上 4 组业务逻辑约束表达为 LTL 的逻辑公式。结合强/弱状态断言和模型验证技术，以下给出了对应的 4 组 LTL 业务逻辑公式和待查的服务流结构验证与规范语义约束描述：

1)　**业务逻辑 1：**

1.1）G ($TmpltBank_H1.Exit \rightarrow$ ((F $TrigBank_H1.Exit$) $\wedge$ (F $Inspiral_H1.Exit$)))

1.2）G ($TmpltBank_H2.Exit \rightarrow$ ((F $TrigBank_H2.Exit$) $\wedge$ (F $Inspiral_H2.Exit$)))

③ GridPiAnalyzer 方法中当前所集成的模性验证引擎为 NuSMV2。有关 NuSMV2 的输入模型语法及从状态标号迁移系统到它的语法转换实现将在第 6 章的 GridPiAnalyzer 系统介绍中详细给出。

2) **业务逻辑 2：**

$$\mathrm{G}\ ((InitData_H1H2.Active) \rightarrow ((\neg\ Inspiral_H2.Exit\ \mathrm{U}\ sInca_L1H1.Active) \wedge (\neg\ Inspiral_H2.Exit\ \mathrm{U}\ thInca_L1H1.Active)))$$

3) **业务逻辑 3：**

$$((\mathrm{F}\ sInca_L1H1.Active \wedge (\neg thIncaII_L1H1.Exit\ \mathrm{U}\ sInca_L1H1.Active)) \vee (\mathrm{F}\ thInca_L1H1.Active \wedge (\neg thIncaII_L1H1.Exit\ \mathrm{U}\ thInca_L1H1.Active))) \wedge \mathrm{F}\ thIncaII_L1H1.Exit$$

4) **业务逻辑 4：**

4.1）$\mathrm{G}\ (Inspiral_H1.Exit \rightarrow (\mathrm{F}\ thIncaII_L1H1.Exit))$

4.2）$\mathrm{G}\ (Inspiral_H2.Exit \rightarrow (\mathrm{F}\ thIncaII_L1H1.Exit))$

4.3）$\mathrm{G}\ (TmpltBank_H1.Exit \rightarrow (\mathrm{F}\ thIncaII_L1H1.Exit))$

4.4）$\mathrm{G}\ (TmpltBank_H2.Exit \rightarrow (\mathrm{F}\ thIncaII_L1H1.Exit))$

5) **服务可达性：**$(SrvFlow，SysState_{init}) \vDash \langle Srv.Status = Exit \rangle_{(\Phi, true)}$，$Srv \in \{initData_H1H2，tmpltBank_L1，tmplt\ Bank_H1，tmplt\ Bank_H2，Inspiral_L1，Inspiral_H1，Inspiral_H1，TrigBank_H1，TrigBank_H2，sInca_L1H1，thInca_L1H1，thIncaII_L1H1，thIncaII_L1H2，InspVeto，ReturnRes\}$；

6) **可终止性：**$(SrvFlow，SysState_{init}) \vDash \lceil ReturnRes.Status=Exit \rfloor_{(\Phi, true)}$；

7) **消息竞争：**DAGMan 中无此约束；

8) **变量垃圾回收：**$(SrvFlow，SysState_{init}) \vDash \lceil |range(*.bSize)|=0 \rfloor_{(\Phi, true)}$；

其中，业务逻辑 1、3、4 的 LTL 公式分别对应为时序性质模式[56]中典型的响应（Response Pattern）模式和优先模式（Precedence Pattern）。而 *Inspiral_H1.Exit* 则是 *Inspiral_H1_1.Exit* ∨ *Inspiral_H1_2.Exit* 的简写，它表示 *Inspiral_H1* 服务的当前状态（Status）为 *Exit*（对于 *Inspiral_H2.Exit*、*TrigBank_H1.Exit*、*TrigBank_H2.Exit* 亦是如此）。在 GridPiAnalyzer 方法通过 4.2、4.3 小节进行状态标号迁移系统的转换生成后，以上引力波探测数据分析服务流的状态π演算形式化语义所对应的标号迁移系统共包含 932（$2^{9.8642}$）个可达状态，总状态命题数为 26 个（20 个服务活动的 *Status* 变量，1 个 *ExecutingSrv* 变量，1 个 *Exception* 变量，2 个 *CurrentVal* 变量，分别针对 *Inspiral_H1* 和 *Inspiral_H2*）。整个服务流的状态π演算形式化语义生成耗时 78 毫秒，状态标号迁移系统的自动生成和状态断言的检验耗时 2297 毫秒，基于现有 NuSMV2[103]引擎的可达状态计算耗

时 1750 毫秒，最大内存占用 37.412Mb。对以上 8 条时序逻辑公式的验证总时间可参见表 4.3，其中包括反例生成的时间 1339 毫秒。具体性能见表 4.3。本文的验证硬件环境为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。

针对该应用的两个更复杂服务流实例的完整验证性能亦可参见附录 A 第 5 章中对验证方法性能改进的研究结果。

表 4.3 各业务逻辑的验证性能（ms：毫秒）

状态π演算的形式化	78 ms	状态断言的检验	2297 ms
可达状态计算	1750 ms	反例生成	1339 ms
业务逻辑1.1的验证	2125 ms	业务逻辑1.2的验证	2688 ms
业务逻辑2的验证	4094 ms	业务逻辑3的验证	3156 ms
业务逻辑4.1的验证	2328 ms	业务逻辑4.2的验证	2500 ms
业务逻辑4.3的验证	2313 ms	业务逻辑4.4的验证	2359 ms
业务逻辑1.1的验证	37.237 Mb	业务逻辑1.2的验证	37.412 Mb
业务逻辑2的验证	36.512 Mb	业务逻辑3的验证	37.410 Mb
业务逻辑4.1的验证	37.352 Mb	业务逻辑4.2的验证	37.389 Mb
业务逻辑4.3的验证	36.887 Mb	业务逻辑4.4的验证	37.837 Mb

验证结果显示，图 4.5 中的 LIGO 引力波探测数据分析服务流中的所有服务（*InitData*、*TmpltBank*、*Inspiral*、*sInca*、*thInca*、*TrigBank*、*TrigVeto*、*thIncaII*、*ReturnRes*）均可达；在终止条件 *TC*=“*ReturnRes.Status=Exit*”（即：最终分析结果的返回必能完成）下该服务流必为可终止的；且整个服务流在终止时不会产生垃圾变量信息。此外，对于 4 组指定的业务逻辑约束，图 4.5 中的 LIGO 引力波探测数据分析服务流可以满足业务逻辑 1（模板库生成的后续操作）、业务逻辑 3（偶然性分析的完整性）和业务逻辑 4（偶然性分析的必然性），但验证结果显示它并不满足业务逻辑 2（干涉仪的工作状态约束）。GridPiAnalyzer 方法中模型验证引擎给出了针对性的反例，它包含 51 个状态迁移。其中的重要迁移部分可见图 4.6，它对各服务状态（Status）和下一步动作迁移（action）的两个关键命题进行了表示。

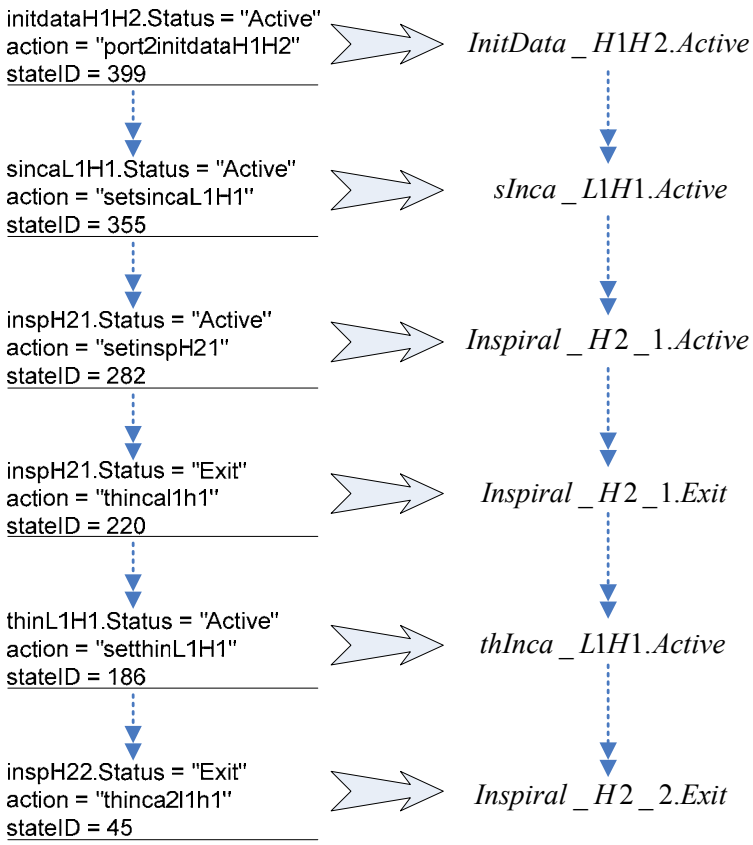


图 4.6 业务逻辑 2 验证结果的反例片断

从该反例中可以发现，在该服务流中确实存在 Hanford 的两干涉仪同时工作（*InitData_H1H2.Exit*）的情况，且此后有部分对 H2 数据的期望波形匹配工作（*Inspiral_H2_1*）可能在偶然性分析 *thInca_L1H1* 进行之前完成。该反例表明了图 4.5 服务流不满足针对 *thInca_L1H1* 的干涉仪工作状态约束，即：

$$G\ ((InitData_H1H2.Active) \rightarrow (\neg\ Inspiral_H2.Exit\ U\ thInca_L1H1.Active))$$

不能成立。由此，为了弄清 *Inspiral_H2* 与 *thInca_L1H1* 和 *sInca_L1H1* 间存在的相互约束关系，我们进一步对以下额外的时序性质公式做出验证：

1）服务流是否满足针对 *sInca_L1H1* 的干涉仪工作状态约束？

$$G\ ((InitData_H1H2.Active) \rightarrow (\neg\ Inspiral_H2.Exit\ U\ sInca_L1H1.Active))$$

2）*Inspiral_H2_1* 是否可能在 *thInca_L1H1* 完成之后进行？

$$F\ (thInca_L1H1.Exit \rightarrow F\ Inspiral_H2_1.Active)$$

两者的验证结果均给出了肯定的答案（总耗时分别为：2381 毫秒和 1937 毫秒；内存占用分别为 36.922Mb 和 33.692Mb）。这表明：一方面该服务流能够

满足针对 *sInca_L1H1*（而不是 *thInca_L1H1*）的干涉仪工作状态约束；另一方面造成以上约束违背的原因是偶然性分析 *thInca_L1H1* 和部分 H2 的期望波形匹配 *Inspiral_H2_1* 之间实际上可以以任意顺序执行（因为 *Inspiral_H2_1* 也可能在 *thInca_L1H1* 完成之后才开始）。这正提醒了 LIGO 的引力波探测数据分析用户需要进一步为这两者建立必要的执行约束（见图 4.7 的修正）。

需要注意的是，以上基于 LIGO 数据网格中引力波探测数据分析应用的一个简单服务流实例讨论了 GridPiAnalyzer 方法中对网格服务流的业务逻辑验证，其验证复杂度等同于 LTL 模型验证的复杂度 $m*2^{O(k)}$，m 为待验证网格服务流状态π演算形式化语义的状态标号迁移系统规模，而 k 则为待验证 LTL 逻辑公式的规模。关于图 4.7 的修正以及该应用更复杂的两个服务流实例的完整验证结果与性能分析可参见附录 A 和下一章中对本验证方法性能改进的研究结果。

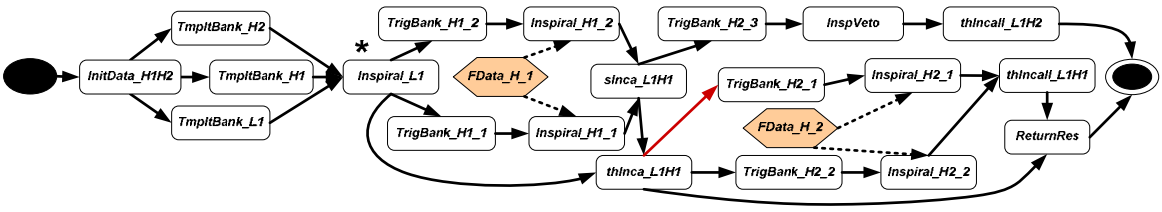


图 4.7　修正的 LIGO 引力波探测数据分析网格服务流 SF1

4.4.3　动态业务逻辑验证

以上工作中给出了针对一静态给定的网格服务流状态π演算形式化语义，如何通过强/弱状态断言方法以及模型验证的支持来完成其服务流结构、规范语义约束和业务逻辑验证。然而由于虚拟组织的动态性，服务提供者可以根据自身的意愿添加或注销相应的服务，并改变其获取策略。因此无论网格环境还是网格服务流自身都始终处于一个动态变化的上下文中。

正如本章引言中图 4.1 的整体验证方法所示，本小节中讨论的则是如何实现对相应状态标号迁移系统的动态重构，从而将网格环境或服务流模型自身的动态变化及时反映到其状态π演算形式化语义中④，使以上基于状态π演算的网格服务流形式化验证方法可以将网格环境的动态性考虑进来。为了防止状态标号迁移系统不间断的连续重构，此处将网格服务流在需要进行其状态标号迁移

④ 对动态性处理的另一个考虑则是进一步提升网格服务流的形式化验证性能，从而在仍然合理的上下文环境中尽快给出有效的验证结果。对于这部分研究工作将在下一章详细展开。

系统动态重构时对应的状态π演算进程称为观测点进程（*CheckPointProcess*），以使得在观测点进程处可以针对更新的网格环境信息做出有效的重验证。需要说明的是，对于观测点（Check Point）自身的设置策略是网格服务流中服务映射与服务执行约束的另一个研究方向[48,148,149]，它并不属于本章的研究范围。本节中关心的是在已设置的观测点基础上，针对网格服务流所进行的形式化语义动态重构及重验证。

由此，在以上 GridPiAnalyzer 方法中还需要在观测点进程处根据更新的网格环境和服务流上下文信息，能够动态重构生成其新的状态标号迁移系统，以保障图 4.1 中本章验证方法对网格服务流在新环境下验证的有效性。更具体的：一个网格服务流的状态π演算形式化语义（*SrvFlow* 进程）由其流程结构（*FlowStruct*）和流程上下文（*FlowContext*）两部分并发组成（*SrvFlow*= *FlowStruct*|*FlowContext*，参见 3.5 小节定义）。而每个观测点进程（*CheckPointProcess*）本质上则是 *SrvFlow* 进程演化过程中的中间进程，它表示了对应网格服务流的形式化语义需要进行动态重构的时机。因此，给定网格服务流的 $SrvFlow=FlowStruct_{old}|FlowContext_{old}$ 进程及其相应的状态标号迁移系统 $TransSys(SrvFlow, SysState_{init})$，若已知观测点进程 *CheckPointProcess*，则对应的网格服务流状态标号迁移系统的动态重构回答了以下问题：$\forall(P, S) \in TransSys(SrvFlow, SysState_{init})$ s.t. $P \equiv CheckPointProcess$，如何根据此时的已知状态集合 *S*，在新的流程结构（$FlowStruct_{new}$）和流程上下文（$FlowContext_{new}$）下为 $FlowStruct_{new}|FlowContext_{new}$ 生成新的对应状态标号迁移系统。

根据以上思路，图 4.8 中给出了网格服务流状态π演算形式化语义的动态重构算法流程 *TransSysDynamic*。图 4.8 中的动态重构中重用了图 4.3 和图 4.4 中的状态标号迁移系统生成方法 *TransSys*。其关键步骤在于首先找出网格服务流的原状态标号迁移系统中在对应观测点进程处的所有可能状态集合 *states*；以当前 *states* 中的每一个状态为初始，*TransSysDynamic* 进一步基于新的流程结构和流程上下文动态局部更新对应的新状态标号迁移系统，并将重构后分别的子状态标号迁移系统最终在统一状态下予以合并。图 4.9 中给出了其过程示意图。

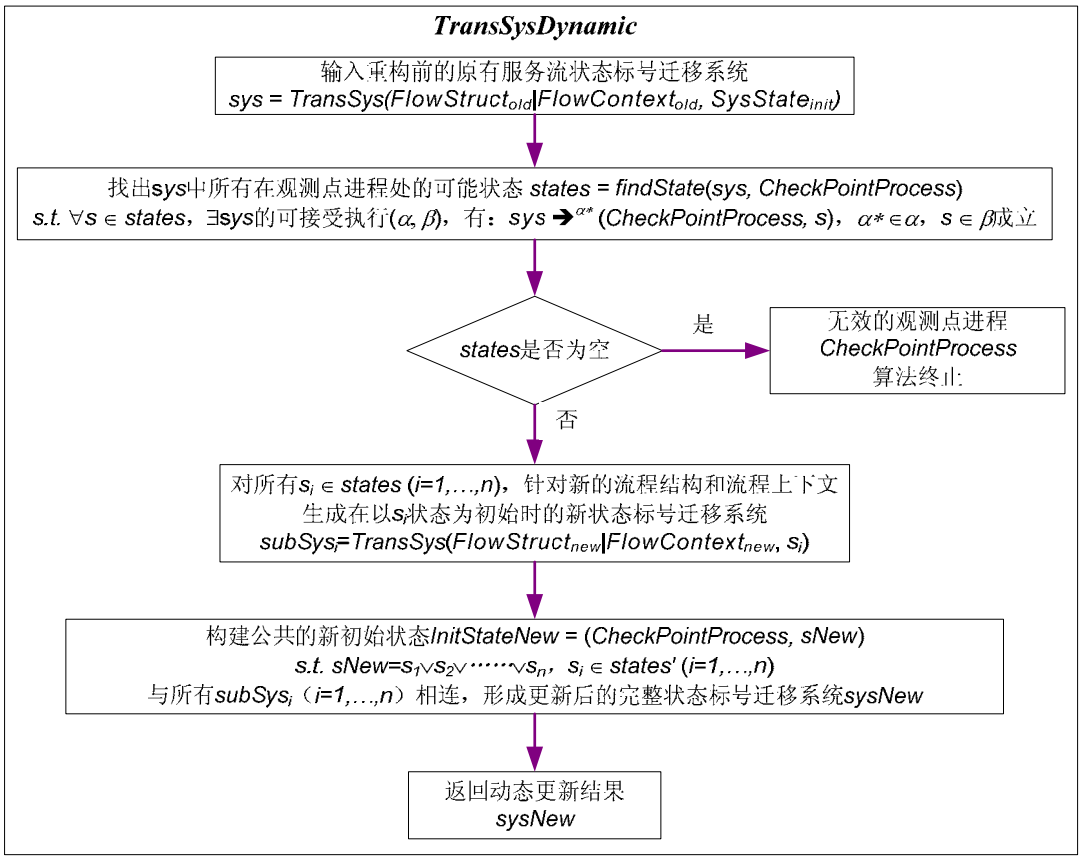


图 4.8 网格服务流状态π演算形式化语义的动态重构

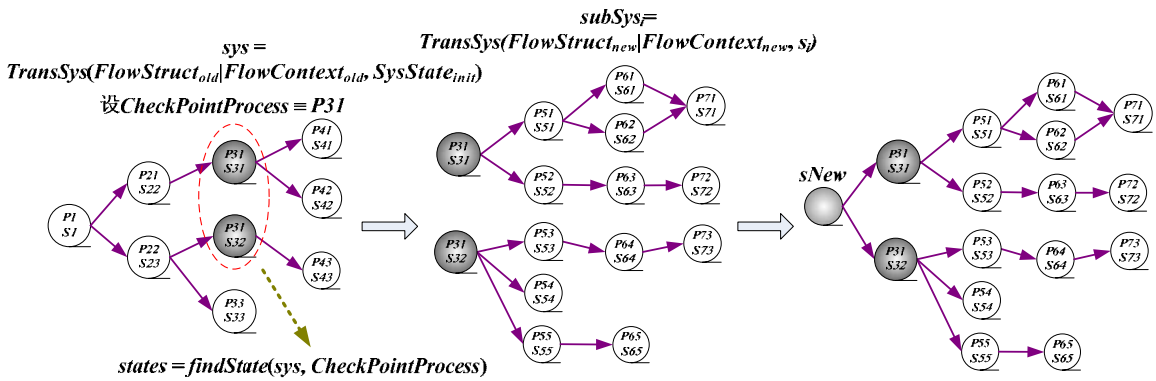


图 4.9 网格服务流形式化模型的动态重构过程示意

仍以 4.4.2 小节中 LIGO 数据网格的引力波探测数据分析为例，在图 4.5 的服务流实例中在 *Inspiral_L1* 处设置了一观测点（以*号表示）。对应的，设此时其 DAGMan 脚本中 *Inspiral_L1* 有着如下的 PRE 脚本设置，使得整个引力波探

测数据分析服务流（记为 *GWFlow*）可以在运行到 *Inspiral_L1* 对 4.4.2 小节中 4 组必要业务逻辑（记为 *GWProperties*）做出自动验证（记为 *GridPiAnalyzer*）：

SCRIPT PRE *Inspiral_L1 GridPiAnalyzer GWFlow GWProperties*。

此时由于对业务逻辑 2“干涉仪的工作状态约束”的违反（见 4.4.2 小节验证结果及数据），通过验证结果将发现对该约束不满足的错误信息。为了使当前服务流能按既定业务逻辑需求准确执行完毕，通过 DAGMan 自身基于 POST 脚本的动态流程重写，为 *Inspiral_L1* 之后的子服务流按图 4.7 进行了相应的修正。则在此情况下针对按图 4.7 修正后的服务流结构，按 *TransSysDynamic* 整个状态标号迁移系统的对应重构结果共包含 888 ($2^{9.7944}$)个可达状态，总状态命题数不变。整个服务流的状态标号迁移系统动态重构耗时 1844 毫秒。在重构后基于本章 GridPiAnalyzer 方法中所集成的现有 NuSMV2[103]引擎，其可达状态计算耗时 1516 毫秒，对 4.4.2 小节 8 条时序逻辑公式的验证过程总时间见表 4.4，最大内存占用 35.696Mb。验证的硬件环境仍然为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。

通过以上对图 4.7 的修正服务流，重构后的验证发现由于对 *thInca_L1H1* 和 *Inspiral_H2_1* 间执行约束的建立，使得重构后服务流在仍保证满足原有业务逻辑 1、3、4 的基础上，进一步满足了业务逻辑 2 的干涉仪工作状态约束。

表 4.4 重构后的业务逻辑验证性能（ms：毫秒）

动态重构	1844 ms	可达状态计算	1516 ms
业务逻辑1.1的验证	1781 ms	业务逻辑1.2的验证	2172 ms
业务逻辑2的验证	1775 ms	业务逻辑3的验证	2531 ms
业务逻辑4.1的验证	2068 ms	业务逻辑4.2的验证	2182 ms
业务逻辑4.3的验证	1999 ms	业务逻辑4.4的验证	2015 ms
业务逻辑1.1的验证	32.796 Mb	业务逻辑1.2的验证	35.696 Mb
业务逻辑2的验证	31.356 Mb	业务逻辑3的验证	35.318 Mb
业务逻辑4.1的验证	32.080 Mb	业务逻辑4.2的验证	33.912 Mb
业务逻辑4.3的验证	31.336 Mb	业务逻辑4.4的验证	31.336 Mb

注意由于以上的 *TransSysDynamic* 是基于图 4.3 和图 4.4 中状态标号迁移系统的生成方法 *TransSys* 实现的，因此它的复杂度和 *TransSys* 一样随着待验证网格服务流π演算语义的状态标号迁移系统规模线性增长，且该状态标号迁移系统的规模同时取决于相应状态π演算进程演化过程的中间进程 P 及其可能关联的不同状态数。注意在此处基于 *TransSysDynamic* 的重构过程中，由于只需对观测点进程后所对应的局部网格服务流进程做出动态形式化，因此它处理的状态数要小于完整状态标号迁移系统生成时所处理的状态数。这解释了为什么在表 4.4 中的业务逻辑验证时间、可达状态计算时间和最大内存占用均小于表 4.3 中对应网格服务流的完整静态验证性能。

4.5　状态π演算形式化验证方法的优点与特点

正如 1.4.3 小节的综述所述，在本章的状态π演算形式化验证方法（简称 GridPiAnalyzer 方法）中，对状态π演算的模型验证支持采用了与 Uppsala 大学的 MWB[118,135]不同的思路。它不是在相应移动进程代数证据系统上对时序逻辑验证的直接实现，而是通过状态π演算的扩展操作语义将系统模型转换为状态标号迁移系统，并在此基础上通过具体验证引擎（如：NuSMV2）所实现的时序逻辑验证。而这一点和 PISA 大学所研究的 HAL（HD-Automata Laboratory）[80]在思路上是一致的。本章采用该思路的原因和优点在于：

（1） 在实际运用业务逻辑验证前就可以基于状态断言方法对状态π演算进程的基本命题性质做出验证，从而可以减少对它们进行重复的模型验证工作，降低整个验证的系统开销；

（2） 可以使整个状态π演算的业务逻辑验证方法独立于特定的模型验证技术。这使得更多在实际中成熟有效的验证算法（如：Bounded Model Checking）可以直接被用于本章的网格服务流形式化验证中。

然而另一方面，本章方法相对 HAL 仍然有着以下的不同点和优点：

- GridPiAnalyzer 方法实现了对状态π演算进程演化和状态操作的推演及模型验证的支持。正如第 2、3 章中的研究结果表明，除了状态π演算自身对系统状态生命周期管理能力的扩充外，它还有利于简化对业务逻辑性质的描述。
- GridPiAnalyzer 方法实现了对状态标号迁移系统的动态重构，以将网格环境的动态上下文信息及时反映到其对应的状态π演算形式化语义中。

- GridPiAnalyzer 方法中通过进一步实现状态π演算的强/弱状态断言（以及下一小节将要介绍的业务逻辑一致性验证），可以降低对模型验证方法的重复依赖。此外它更实现了（第 5 章中）基于域分析的服务流分解和基于错误过程模式的搜索向导两种方法以进一步降低由状态标号迁移系统生成和业务逻辑验证的性能花费。具体性能比较亦可参见第 5 章结果。

4.6　业务逻辑的一致性验证

4.6.1 冲突业务逻辑与冗余业务逻辑

正如本章引言中图 4.1 的方法框架所示，在实际针对网格服务流进行业务逻辑验证之前，可以额外地进行待查业务逻辑性质间的一致性验证。一致性验证的目的在于事先判别：（1）对某业务逻辑的验证努力是否是不必要的，因为它的成立已经可以通过网格服务流所满足的其它业务逻辑直接推出（即：存在冗余业务逻辑）；（2）对给定业务逻辑的验证努力是否是徒劳的，因为网格服务流必定无法同时满足所有给定的业务逻辑（即：存在冲突业务逻辑）。业务逻辑的一致性验证使得在针对实际网格服务流进行复杂的业务逻辑验证之前，对无谓的验证工作做出预判，从而节约网格服务流的整体验证代价。

更严格的，对冗余和冲突业务逻辑可以定义如下：

定义 4. 10（冗余业务逻辑）：给定一组待查业务逻辑 $A=\{A_1, A_2, \ldots\ldots, A_n\}$，称 $A_j \in A$ 在 A 中是冗余的，当$\exists$ $\{A_{i1}, A_{i2} \ldots, A_{ik}\} \in A$（$0<k<n, i_k \neq j$），s.t. 对于任意网格服务流模型 M，$M \vDash A_{i1} \wedge A_{i2} \ldots \wedge A_{ik} \rightarrow M \vDash A_j$ 成立。

定义 4. 11（冲突业务逻辑）：给定一组待查业务逻辑 $A=\{A_1, A_2, \ldots\ldots, A_n\}$，称 A_i，$A_j \in A$（$i \neq j$）是相互冲突的，当对于任意网格服务流模型 M，$M \vDash A_i \rightarrow M \nvDash A_j$ 成立；反之亦然。

然而由于本章中描述业务逻辑约束的 LTL 时序逻辑公式自身的复杂性，其时序算子的嵌套组合及其逻辑非语义间往往不能形成明显的偏序关系，因而增加了分析各业务逻辑间关联的难度。特别是随着待验证业务逻辑数的增多，使得其中的冗余和冲突验证成为一项必要而困难的任务。为了解决这一问题，本文的思路是复用 LTL 的模型验证原理将业务逻辑的一致性验证纳入本章中网格服务流形式化验证的整体框架中。与传统人工智能和数据库系统中基于决策树/

表[150,151]和逻辑规划[152,153]的业务规则冲突检测方法相比，这样做的优点在于：

（1） 可以进一步实现对业务逻辑间（LTL）时序关系的处理，从而将严格的时序关系考虑到冗余和冲突逻辑的检测中；

（2） 无需枚举或预定义业务逻辑间可能存在的偏序关系，使方法更一般化；

（3） 在冲突检测的同时能够兼顾对业务逻辑间冗余的判断。

然而正如 1.4.2 小节的综述所述，模型验证技术自身的典型应用是对待验证系统时序逻辑性质的检验。为了将其重用到本小节问题的求解，本文注意到在描述系统所应满足的业务逻辑同时也是在对该系统的局部行为进行刻画[112,132]。换言之，针对 LTL 模型验证，一个期望满足的 LTL 时序逻辑公式本身可以等价转换为一个（扩展） Büchi 自动机，该自动机包含了所有系统模型所应该接受的语言。因此，本文将业务逻辑性质自身同时作为待检验系统的局部状态模型看待，通过该思路的调整在以下研究了 LTL 业务逻辑性质间的冲突、冗余关系，并给出了相应的方法证明与实现。

4.6.2　业务逻辑一致性验证方法与实现

本节先给出了扩展 Büchi 自动机及其语言的必要定义，在此基础上将分析基于 LTL 模型验证的业务逻辑一致性验证方法和关键性质。

定义 4.12（扩展 Büchi 自动机）：一个扩展 Büchi 自动机是一个 5 元组 $B = <\Sigma, Q, \Delta, Q^0, F>$，其中：

- Σ是一个有限的命题集合；
- Q是一个有限的状态集合；
- $\Delta \subseteq Q \times \Sigma \times Q$是一组迁移关系；
- $Q^0 \in Q$是一个初始状态；
- $F \subseteq 2^S$是一个可接受状态的集合。

定义 4.13（运行）：一个扩展 Büchi 自动机在无限字符序列 $w= a_1a_2a_3\ldots$（a_i 定义在字符集Σ中）上的一次运行是一个无限状态序列$\eta = Q_0Q_1Q_2\ldots$（Q_i 定义在状态集 Q 中），s.t. $\forall i \in N^+, Q_i \in \Delta(Q_{i-1}, a_i)$。

定义 4.14（可接受的运行）：定义在字符集Σ中的一个无限字符序列 w 上的一次运行可以被一个扩展 Büchi 自动机接受，当且仅当它无限次包含 F 中可接受状态集合内的至少一个元素。

称一个扩展Büchi自动机 B 所有可接受运行的集合为其可接受的语言，并记为 $\Psi(B)$。除了 LTL 时序逻辑公式与扩展Büchi自动机的一一对应（记为：$Trans(f)$），以下定义了本文中对应于多条同时期望满足的 LTL 时序逻辑公式，所需进行的扩展Büchi自动机的相交（*Intersection*）。

定义 4.15（自动机的相交）：定义一组（LTL）时序逻辑公式$\{A_1，……，A_n\}$所对应扩展Büchi自动机的交（记为 $Intsec(Trans(A_1), ..., Trans(A_n))$），为这组时序逻辑公式的逻辑与所对应的扩展Büchi自动机：$Trans(A_1 \wedge \ldots \wedge A_n)$。

LTL 模型验证的基本原理告诉我们：给定系统模型 M（如一个网格服务流的状态π演算形式化语义）及其命题集合$\sum_M$，则 M 能够满足时序逻辑性质 A，当$\Psi(M) \subseteq \Psi(A)$。它可以理解为模型中 M 的所有行为均为待验证性质 A 所允许的行为。由此，对冲突业务逻辑性质的验证思路可以转换为判别：一部分业务逻辑中所能允许的行为在另一部分业务逻辑中是否一定是不允许的；即是否一定有$\Psi(M) \subseteq \Psi(A_1) \subseteq / \Psi(A_2)$成立，其中$/ \Psi(A_2)=\sum_M^{\varpi} - \Psi(A_2)$表示性质 A_2 所接受语言在命题集合$\sum_M$下语言全集中的补集。另一方面，冗余业务逻辑性质的验证思路可以转换为判别：一部分业务逻辑中所能允许的行为在另一部分业务逻辑中是否一定是允许的；即是否一定有$\Psi(M) \subseteq \Psi(A_1) \subseteq \Psi(A_2)$成立。由此，业务逻辑公式间冗余和冲突验证可实现为对扩展Büchi自动机所接受语言的检验。

性质 4.3（冲突检测）：一组 LTL 时序逻辑公式$\{A_1，……，A_n\}$在$\sum_M^{\varpi}$下存在相互冲突的逻辑公式，若 $Intsec(Trans(A_1), ..., Trans(A_n))$没有任何可接受的运行，即$\Psi(Trans(A_1 \wedge \ldots \wedge A_n))=Empty$。

证明：若∃非空网格服务流状态π演算语义的状态标号迁移系统 M，s.t. $\Psi(M) \neq Empty$ 且 $M \vDash A_1 \wedge \ldots \wedge A_n$，则可得 $Empty \subset \Psi(M) \subseteq \Psi(Trans(A_1 \wedge \ldots \wedge A_n))=Empty$。矛盾。因此根据 LTL 模型验证的定义，∄网格服务流的非空形式化模型 M，s.t. $M \vDash A_1 \wedge \ldots \wedge A_n$（即同时满足所有给定业务逻辑）。证毕。 □

性质 4.4（冗余检测）：一 LTL 时序逻辑公式 A_1 在一组 LTL 时序逻辑公式$\{A_1，……，A_n\}$中是冗余的，若在$\sum_M^{\varpi}$下 $Intsec(Trans(\neg A_1), ..., Trans(A_n))$没有任何可接受的运行，即$\Psi(Trans(\neg A_1 \wedge \ldots \wedge A_n))=Empty$。

证明：由$\Psi(Trans(\neg A_1 \wedge \ldots \wedge A_n))=Empty$可得，对于非空网格服务流状态π演算模型的状态标号迁移系统 M，$Empty \subset \Psi(M) \subseteq / \Psi(Trans(\neg A_1 \wedge \ldots \wedge A_n)) = \sum_M^{\varpi}$。因此 $M \vDash \neg(\neg A_1 \wedge \ldots \wedge A_n) = \neg (A_2 \wedge \ldots \wedge A_n) \vee A_1$。则由业务逻辑冗余的定义，若当已知 $M \vDash (A_2 \wedge \ldots \wedge A_n)$时，根据以上关系必有 $M \vDash A_1$ 成立，即 $M \vDash A_2 \wedge \ldots \wedge A_n \rightarrow M$

$\models A_1$ 成立。证毕。 □

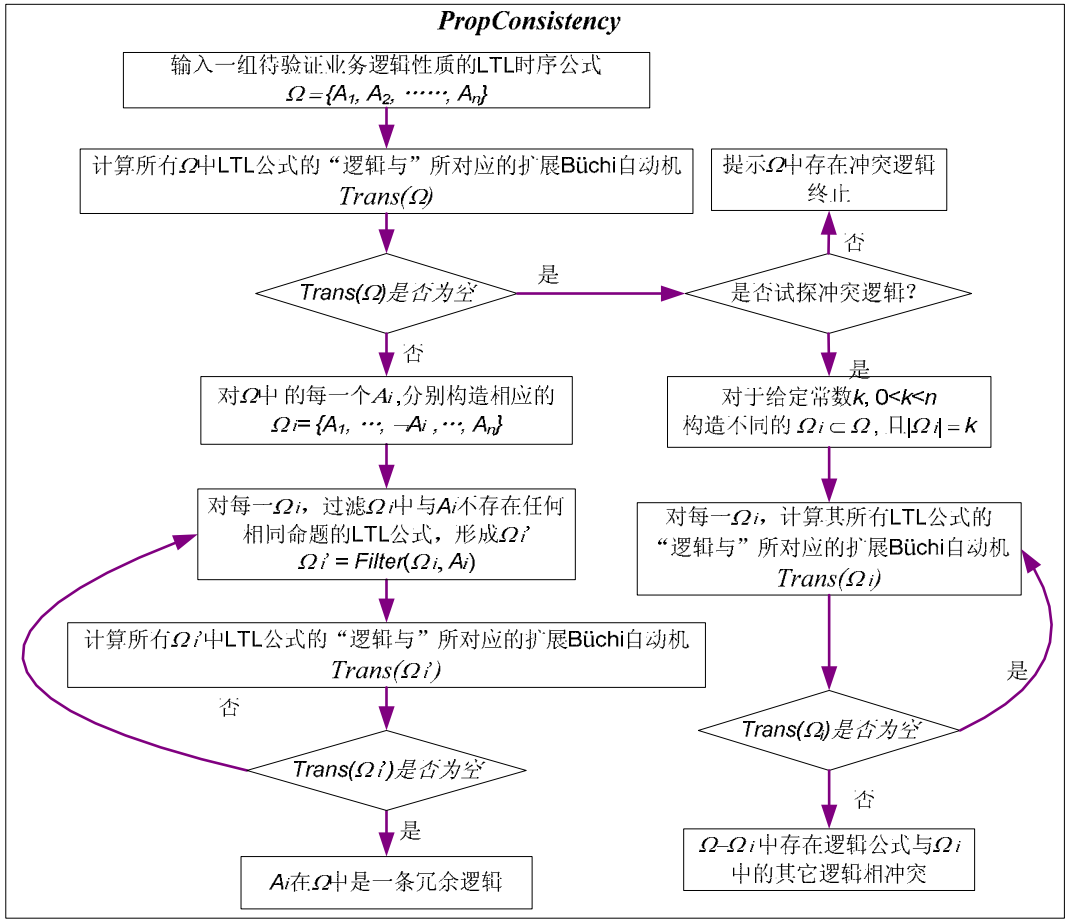


图 4.10 业务逻辑性质的冗余和冲突检测方法流程

至此，通过以上两性质可以完整的将 LTL 业务逻辑性质间的冗余和冲突验证转换为一个 LTL 模型验证问题。根据以上结论，图 4.10 中分别给出了对 LTL 业务逻辑性质的冗余和冲突检测方法。其中，步骤 *Trans*(*Ω*)负责进行 LTL 业务逻辑性质（如：4.4.2 小节中的时序逻辑公式）到扩展 Büchi 自动机的转换，它利用文献[154]中的 LTL2 Büchi 算法实现。图 4.10 中 *PropConsistency* 严格遵循了性质 4.3 和性质 4.4 中的结论。首先在冲突检测部分，*PropConsistency* 通过对所有待验证业务逻辑*Ω*与所对应扩展 Büchi 自动机是否为空来进行判断。若确实为空，则 *PropConsistency* 可以进一步通过从*Ω*中移除 *n-k* 条业务逻辑（*0<k<n*）来试探该冲突是否是由于这 *n-k* 条业务逻辑所产生的。此时 *k* 可以理解为冲突业务逻辑的“冲突试探”。默认情况下当逐条试探冲突业务逻辑时 *k* 为一固定常

数 *n-1*。另一方面，若Ω中不存在冲突，则 *PropConsistency* 将进入冗余检测部分。此时通过对Ω中业务逻辑的逐一取反及其逻辑与之后所对应扩展 Büchi 自动机是否为空来进行判断。其中，步骤 *Filter* 负责过滤Ω_i中与 A_i 不存在任何相同命题的 LTL 公式。这是因为在不考虑具体待验证网格服务流自身信息的情况下，我们无法预知公式中两个不相同原子命题间可能存在的逻辑关联，因而在图 4.10 算法中默认不认为Ω_i中与 A_i 不存在公共命题的 LTL 公式会对 A_i 的冗余判别造成影响，从而降低在业务逻辑一致性验证过程中所需生成的对应自动机规模。例如：在 4.4.2 小节的所有待验证逻辑公式中，业务逻辑 2 和 4.1 中就不存在任何相同的原子命题，因而当判别业务逻辑 4.1 的冗余性时，业务逻辑 2 可以从Ω_i中除去（相应业务逻辑的 LTL 公式可参见 4.4.2 小节，在此不再赘述）。

4.6.3　应用实例与讨论

本节将以上的业务逻辑一致性验证结论及方法在 LIGO 引力波探测数据分析应用的业务逻辑中进行了应用。表 4.5 和表 4.6 中列出了针对 4.4.2 小节中 8 条 LTL 逻辑公式的业务逻辑一致性验证结果及性能。验证的硬件环境仍然为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。基于图 4.10 的方法结果显示：这 8 条 LTL 业务逻辑间不存在冲突，其逻辑与之后所对应的扩展 Büchi 自动机在最小化后共包含 65 个状态和 4731 个迁移，整个自动机生成共耗时 625 毫秒[⑤]，其最小化约简耗时 437 毫秒，最大内存占用 33.662Mb。另一方面，若此时 LIGO 的引力波数据分析人员进一步给出了以下额外期望满足的偶然性分析业务逻辑需求：

- **业务逻辑 5：**

$\mathrm{F}\ \mathit{InitData_H1H2.Exit} \wedge \mathrm{G}\ (\mathit{InitData_H1H2.Exit} \rightarrow \mathrm{G}(\neg\ \mathit{thInca_L1H1.Exit}))$

/ 引力波的探测过程中 H1、H2 两干涉仪必将同时工作，且在任意情况下当两者同时工作时必不会完成对 L1H1 数据的 thInca 偶然性分析 */*

则此时基于图 4.10 方法得：在加上业务逻辑 5 之后的 9 条 LTL 逻辑公式将对应产生一空的扩展 Büchi 自动机，且当移除 4.4.2 小节中业务逻辑 2 或以上业务逻辑 5 中的一条时（即 *k=8*），剩余业务逻辑间则不存在任何冲突。该自动机

⑤ 此处与自动机生成的相关性能基于了文献[154]中的 LTL2Buchi 算法。下同。

生成共耗时 203 毫秒，自动机的最小化约简耗时 0 毫秒（因其大小为空），最大内存占用 20.565Mb。此时对比两业务逻辑的要求，实际上可以发现冲突的原因在于业务逻辑 2 和业务逻辑 5 分别要求了在 H1、H2 两激光干涉仪同时工作后，L1H1 的数据会通过 *thInca* 偶然性分析和肯定不会完成对 L1H1 数据的 *thInca* 偶然性分析。这部分语义造成了两者的矛盾。

另一方面，通过一致性验证的结果还发现，4.4.2 小节中业务逻辑 4.3 的语义在整个业务逻辑集合中是冗余的，它在取反后使得整个扩展 Büchi 自动机成为了空。该自动机生成共耗时 79 毫秒，自动机的最小化约简耗时 0 毫秒（因其大小为空），最大内存占用 7.046Mb。在进行 *Filter* 后发现，业务逻辑 4.3 的语义其实已经包含在了业务逻辑 4.1 和业务逻辑 1.1 中了，即：“在任意情况下，一旦模板库生成服务执行完毕，则相应的一定会进行期望波形匹配，以保证数据分析的有效性”（业务逻辑 1.1）和“一旦数据进行期望波形匹配后，则相应的最终一定会进行最后的偶然性分析的处理”（业务逻辑 4.1）已经可以得出“一旦数据进行模板库生成后，则相应的一定也会进行最后的偶然性分析的处理”（业务逻辑 4.3）这一结论。因此，当目标网格服务流已经满足业务逻辑 4.1 和业务逻辑 1.1 时，可以安全地放弃对冗余业务逻辑 4.3 的验证，从而节约不必要的网格服务流业务逻辑验证花费。相同的结论也发生在业务逻辑 4.4 上。实际上针对以上得到结论，正如 4.4.2 小节所述由于这里的业务逻辑 1.1、4.1 和 4.3 都是时序性质模式中的典型全局响应模式（Global Response），因此通过本节的业务逻辑一致性验证方法实际上在另一方面也同时证实了全局响应模式之间所具备的传递性。

此外，图 4.11 至图 4.13 中还分别给出了以上业务逻辑一致性验证与基于 4.4 小节方法针对 LIGO 引力波探测数据分析的三组实际服务流实例（SF1、SF2 和 SF3）进行业务逻辑验证的验证规模、验证时间和内存占用上的比较。其中，验证规模指待验证服务流的状态π演算形式化模型状态数与待验证逻辑公式所对应状态数的比较。待验证的业务逻辑仍然是 4.4.2 小节中的 8 条 LTL 逻辑公式。服务流实例 SF1 已在 4.4 小节的图 4.7 给出，而其更复杂的两个实例 SF2 和 SF3 将在下一章的 5.2.5 小节详细给出。此处预先给出它们的验证性能是为了与本节中业务逻辑一致性验证的性能做出参考的比较。以上完整的验证性能数据和通过 4 种不同验证方法的性能对比亦可参见附录 A。

表 4.5 业务逻辑一致性验证结果及性能 1 （ms：毫秒）

冲突验证	合成自动机		自动机生成耗时	自动机最小化耗时	最大内存占用	结果
	状态数	迁移数				
不含业务逻辑5	65	4731	625 ms	437 ms	33.662Mb	无冲突
包含业务逻辑5	空	空	203 ms	0 ms	20.565Mb	冲突

表4.6 业务逻辑一致性验证结果及性能2 （ms：毫秒）

冗余验证	状态数	迁移数	自动机生成耗时	自动机最小化耗时	最大内存占用	结果
业务逻辑1.1	8	39	63 ms	15 ms	2.146 Mb	不冗余
业务逻辑1.2	49	893	110 ms	32 ms	5.033 Mb	不冗余
业务逻辑2	97	4280	250 ms	359 ms	18.121Mb	不冗余
业务逻辑3	151	3990	156 ms	47 ms	12.720Mb	不冗余
业务逻辑4.1	12	161	62 ms	16 ms	2.690 Mb	不冗余
业务逻辑4.2	89	3743	140 ms	384 ms	9.747 Mb	不冗余
业务逻辑4.3	空	空	79 ms	0 ms	7.046 Mb	冗余
业务逻辑4.4	空	空	110 ms	0 ms	5.897 Mb	冗余

从图 4.11 至图 4.13 的性能对比可以看出，业务逻辑一致性验证的性能明显小于对 3 组不同复杂度服务流实例的业务逻辑验证性能。特别是针对更复杂的服务流实例 SF2 和 SF3，由于其验证规模的大大增加，使得业务逻辑一致性验证不论在验证时间还是内存占用上都远小于实际的业务逻辑验证。实际上从 1.4.2 小节的综述可知，LTL 模型验证自身的复杂度为 $m*2^{O(k)}$。而在针对相同 LTL 逻辑公式进行网格服务流的业务逻辑验证和一致性验证时，待验证业务逻辑的规模 k 是同一量级的。但另一方面正如 Vardi 所指出的[74]：在实际中待验证系统的模型规模 m（此处为网格服务流的状态π演算模型规模）一般却远大于待验证逻辑公式的规模（此处对于服务流实例 SF2 和 SF3 尤为明显）。这使得针对相同的业务逻辑公式，其一致性验证代价在实际中往往能远小于它在复杂系统中的业务逻辑验证代价，且一致性验证的复杂度仅取决于待验证业务逻辑自身的规模和数量，而不随待验证系统规模的变化而变化。因此，本小节的业务逻辑一致性验证方法不仅能有效检验待验证 LTL 逻辑公式中潜在的冗余/冲

突逻辑，通过对它们的发现同时也能防止无谓的业务逻辑验证工作。这在本章网格服务流的状态π演算形式化验证方法中对于节约验证代价（特别是针对复杂的网格服务流）是一项经济而有意义的工作。

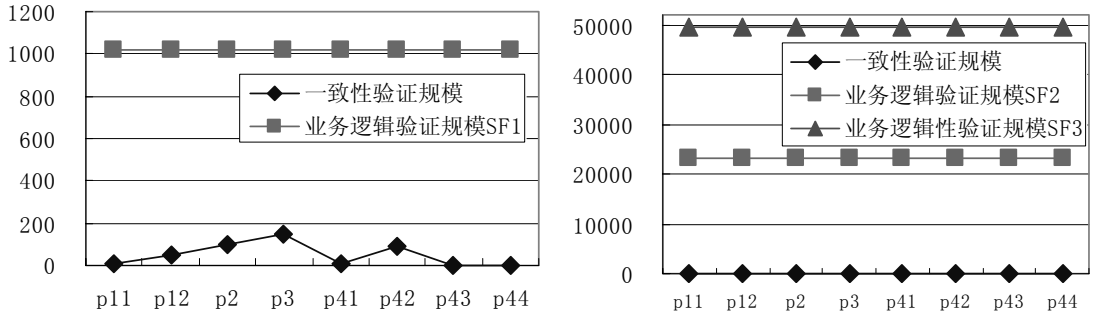


图 4.11 一致性验证与业务逻辑验证的规模比较

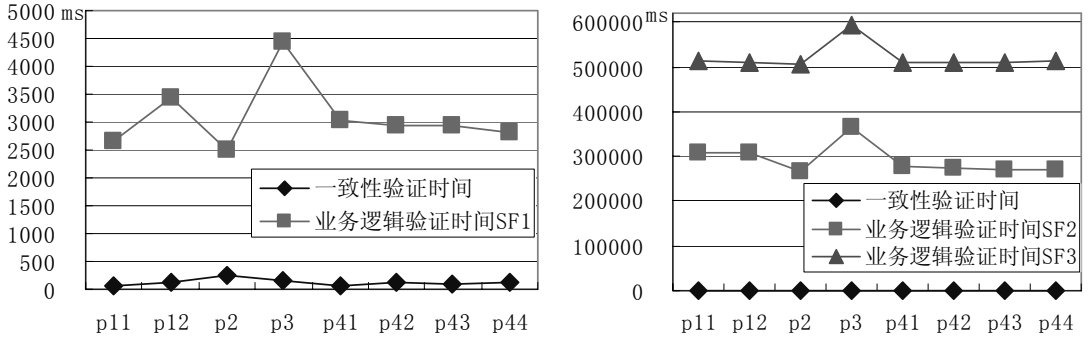


图 4.12 一致性验证与业务逻辑验证的时间性能比较（单位：ms）

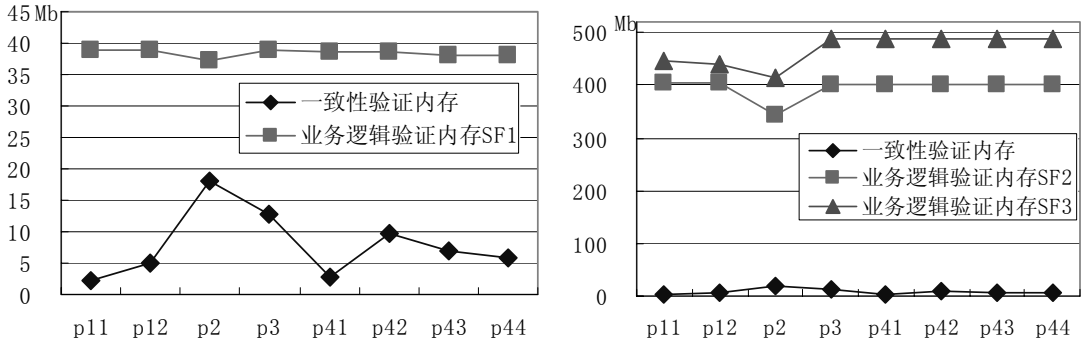


图 4.13 一致性验证与业务逻辑验证的内存占用比较（单位：Mb）

4.7　小结

基于前两章对状态π演算和网格服务流形式化语义的研究结果，本章进一步研究了对网格服务流四方面性质的形式化验证工作。本章的工作与贡献可以总结如下：

1) 基于状态π演算的扩展操作语义给出了其有穷状态标号迁移系统的语义转换。通过提出状态π演算的强/弱状态断言方法实现了网格服务流的结构验证与规范语义约束的验证；
2) 通过状态标号迁移系统的动态重构与状态π演算模型验证支持的实现完成了对网格服务流的动/静态业务逻辑验证。此外，通过 LIGO 数据网格中引力波探测数据分析的实际服务流应用结果展示了本章方法的有效性和优越性。
3) 分析并证明了 LTL 业务逻辑间冗余和冲突的条件，通过对业务逻辑一致性验证方法的实现表明它不仅能有效完成对冗余和冲突逻辑的验证，同时相对实际的业务逻辑验证还能有效减少无谓的验证代价。

本章的网格服务流状态π演算整体形式化验证方法具有以下优点：

- 实现了对状态π演算进程演化和状态操作的推演及模型验证的支持。正如第 2、3 章中的研究结果表明，除了状态π演算自身对系统状态生命周期管理能力的扩充外，它也有利于简化对业务逻辑性质的描述；
- 在实际运用业务逻辑验证前可以基于强/弱状态断言和业务逻辑一致性验证减少重复和无谓的模型验证工作，降低验证的代价；
- 整个验证方法独立于特定的模型验证技术。这使得更多成熟有效的验证算法可以直接用于我们的网格服务流正确性保障中。

在下一章中将进一步研究针对网格服务流正确性验证技术的性能改进。

第 5 章　网格服务流形式化验证方法的性能改进

5.1　本章引论

为了使本文中基于状态π演算的形式化验证方法能更有效地运用于实际的复杂网格服务流，本章进一步对验证方法的性能改进进行了研究。网格服务流形式化验证的性能改进不仅是本文整体工作中的重要一环，同时也是形式化方法自身的重要研究方向。正如 No-Free-Lunch-Theorem (NFLT) for Optimization 原理[155]所述，对实际问题自身特征信息的利用是实现对该问题求解方法改善的关键。因此，不同于现有对模型验证技术自身的复杂性与优化方法研究（见 1.4.2 小节综述），本章重点在于从网格服务流及其状态π演算语义这两个层次着手，通过将其中具体的结构信息和领域知识与相应模型验证优化思路相结合的手段，改进本文中网格服务流状态π演算形式化验证方法的时间和空间复杂度。具体的，本章工作分别包含以下两部分内容：

1) 基于域分析（Region Analysis）的验证分解，它是一种基于服务流分解思想的验证方法；
2) 基于错误过程模式（Process Bug Pattern）的验证向导，它是一种基于搜索向导的快速服务流证伪方法。

本章后续内容组织如下：在 5.2.1 和 5.2.2 小节中首先给出了文中基于“标准域”性质的服务流分解方法；5.2.3 和 5.2.4 小节则分别根据标准域及其并发枝间保持的严格串行与并发关系，给出了对应网格服务流在验证策略分解上的性质与实现。通过 5.2.5 小节对 LIGO 数据网格中三组不同复杂度服务流实例的应用结果，证实了域分析的验证分解方法是提高网格服务流验证效率和内存占用的一种有效方法；另一方面，从 5.3.1 到 5.3.7 小节分别提出了针对传统工作流的 6 组反模式，本文称之为错误过程模式；在 5.3.8 小节中则根据错误过程模式的 IEEE 标准 PSL（Property Specification Language）[82]语义给出了其基于搜索向导的快速检验方法。5.3.9 小节中同样通过多组实际 LIGO 数据网格应用对本章基于错误过程模式的验证向导方法进行了有效性检验和数值结果讨论。

本章以下内容的相关研究结论主要发表和投稿在：

◆ A Static Compliance Checking Framework for Business Process Models. *IBM*

Systems Journal, 2007, 46(2): 335-362 （**SCI** 检索）

- Guided Reasoning of Complex E-Business Processes with Business Bug Patterns. In: *Int. Conf. on E-Business Engineering*, IEEE Computer Society, 2006: 195-202 （**Ei** 源刊）
- Aspect Oriented Region Analysis for Efficient Equipment Grid Application Reasoning. In: *5th Int. Conf. on Grid and Cooperative Computing* (*GCC 2006*), IEEE Computer Society, 2006: 28-31 （**Ei** 源刊）
- Performance Improvement of Temporal Reasoning for Grid Workflows Using Relaxed Region Analysis. *Future Generation Computer Systems* (Submitted), 2007

5.2　基于域分析的验证分解方法

基于域分析（Region Analysis）的验证分解方法的基本思路是：通过获取网格服务流中存在的串行与并发子模块信息（本文称之为包含“并发枝”的“标准域”），将一个完整网格服务流上的待查业务逻辑分解到对其各个标准域的局部验证上。每个标准域上的局部验证将同时利用在其它标准域上的已知验证结果信息。由此通过对每个标准域的局部验证来推出对完整网格服务流上待查业务逻辑的全局验证结果。由于可以将全局验证拆分为在更小状态空间规模下的子模块验证，因此基于域分析的验证分解方法无需一次产生整个网格服务流所对应的完整状态标号迁移系统，而可以按照验证的分解策略对各个标准域进行逐一的处理，因而它可以有效降低验证时的内存占用和时间花费。

为了具体实现以上思路，本章需要解决以下三个关键问题，即：如何对一个网格服务流进行结构拆分，并获得拆分后各子服务流之间有用的时序关系信息？对应于服务流的分解，如何实现相应的验证分解策略？如何基于验证分解策略从局部验证的结果推理出原有服务流上的全局验证结论？

以下工作将围绕着以上三个关键问题详细展开，并给出相应域分解方法的自动化实现与应用结果。

5.2.1　网格服务流的建模约定

正如 1.4.1 小节的综述所述，由于如 BPEL4WS 或 Condor DAGMan 等各类常用规范模型不论在语法和表示格式上都各不相同，因此为了便于以下对网格服务

流中标准域的定义与分析，协助其理解，本小节首先对所涉及的网格服务流进行必要的统一建模约定。一方面，为了兼容本文中 LIGO 数据网格基于 Condor DAGMan 的引力波探测数据分析应用，该约定的目的在于以 DAGMan 规范为基础为其表示做出统一的规范化；另一方面，为了弥补 DAGMan 规范自身在表达能力上的限制，该约定中也进一步允许了对数据参数节点、服务分支与汇合的控制节点和条件迁移的显示建模，从而使本节域分析的工作结论可以兼容到更多的其它服务流模型中。

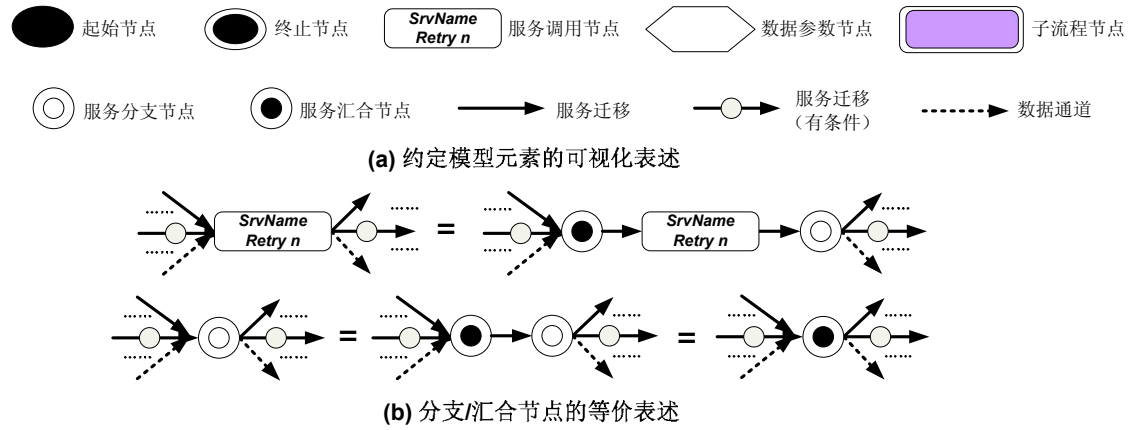


图 5.1 服务流建模元素的可视化描述

如同第三章中对 DAGMan 规范状态π演算的形式化语义建立，图 5.1 中给出了本节中对 DAGMan 模型元素及其可视化表示。其中的网格服务调用节点（包含 Retry 关键字）、数据参数节点（可视为 VARS 关键字的设置）、子流程节点、服务迁移和数据通道均可与原有 DAGMan 输入语言对应。相应的约定如下：

约定1. 引入了虚拟的服务流起始节点和终止节点。每个服务流都有着唯一的一个起始节点和终止节点（在后续的松弛域分析方法中将对此约束做出松弛），使它们标志着整个服务流执行过程的起始和终止；

约定2. 由于每个网格服务调用节点允许有着多输入多输出的关系，因此额外的引入了虚拟的分支/汇合节点来显示表示这些并发与同步关系。它们的可视化表述和等价关系可参见图 5.1（b）；

约定3. 称一个节点是一个服务控制节点，若它是一个网格服务调用节点、子流程节点或控制节点；称一个节点是一个服务流节点，若它是一个服务控制节点或数据参数节点；

约定4. 每个服务控制节点必须都与其起始节点和终止节点通过服务迁移相连通。注意此处的连通性和 4.3 小节中服务可达性的本质区别，即连通性仅在语法层面要求存在相应的服务迁移和数据通道使得一个服务控制节点可以与起始节点和终止节点相连。

约定5. 每个服务迁移只有唯一的一个源/目标服务控制节点。另一方面，每一个数据通道至多有一个源/目标流节点，且其中有一个必须是数据参数节点。因此，称一个数据通道是通畅的（Fluent），当其源/目标数据参数节点与网格服务流的一个终止/起始节点相连通；

约定6. 允许网格服务调用间的循环，但前提是该循环不会造成服务流的非结构化[156]。

5.2.2 基于标准域分析的服务流分解

在以上的统一约定下，本小节首先研究域分析方法中的第一个关键问题，即服务流的分解。为了从分解的思路改善网格服务流的验证效率和花费，首先需要在服务流中寻找特定的结构，使得通过利用该结构间隐含的时序关系，可以将一个网格服务流复杂的全局验证化解为在其子结构上的快速验证。本节中将这样的结构称为一个“域”。记 $N_1 \rightarrow N_2 ... \rightarrow N_m$ 为在网格服务流中从节点 N_1 到 N_m 的一个有限的有向连通路径。$N_1 \rightarrow N_2$ 表示在语法上 N_1 和 N_2 通过一个服务迁移或数据通道形成了连通，且 N_1 、N_2 是该服务迁移或数据通道的源、目标节点。由此，通过服务流中节点的连通信息可以给出以下对域的定义：

定义 5.1（域）： 称两个不同的服务控制节点、起始或终止节点 N_{head} 和 N_{tail} 形成了网格服务流$\mathbb{F}$中一个域，记为$\{N_{head}, N_{tail}\}$，当：（1）$\nexists N_{head} \rightarrow N_1 \rightarrow ... N_m \rightarrow End$ 且 *End* 是$\mathbb{F}$的一个终止节点，使得 $N_i \neq N_{tail}$ 和 N_{head}（$i=1,...,m$）；（2）$\nexists Begin \rightarrow N_1 \rightarrow ... N_m \rightarrow N_{tail}$ 且 *Begin* 是$\mathbb{F}$的起始节点，使得 $N_i \neq N_{head}$ 和 N_{tail}（$i=1,...,m$）。

一个域$\{N_{head}, N_{tail}\}$实际上定义了$\mathbb{F}$中的这样一种结构，使得 N_{tail} 总能包含在从 N_{head} 到$\mathbb{F}$终止节点的任意连通路径中，而 N_{head} 总能包含在从$\mathbb{F}$起始节点到 N_{tail} 的任意连通路径中。例如在后续的图 5.3 中，$\{TrigBank_H2_3, thIncaII_L1H2\}$就是一个域而$\{sInca_L1H1, thIncaII_L1H2\}$则不是。易得整个网格服务流自身也是一个域。称任一服务控制节点 N'包含在一个域$\{N_1, N_2\}$中，记为 $N' \in \{N_1, N_2\}$，当有 $N_1 \rightarrow ... \rightarrow N' \rightarrow ... \rightarrow N_2$ 存在。可以证明域的定义满足以下的有用性质：

性质 5. 1： 一个网格服务流$\mathbb{F}$中若能找到$\{N_1, N'\}$和$\{N'', N_3\}$两个不同的域，且

$N_1 \neq N_3$、$N' = N''$，则$\{N_1, N_3\}$也是$\mathbb{F}$中的一个域。

证明：不失一般性设以上 $N' = N'' = N_2$，由于$\{N_1, N_2\}$和$\{N_2, N_3\}$均为$\mathbb{F}$的域，因此：（1）在$\mathbb{F}$中有 $N_1 \rightarrow \ldots \rightarrow N_2 \rightarrow \ldots \rightarrow N_3$ 存在；（2）对于$\mathbb{F}$中任一 $N_1 \rightarrow \ldots \rightarrow End$，$N_2$ 和 N_3 必同时包含于该连通路径；（3）对于$\mathbb{F}$中任一 $Begin \rightarrow \ldots \rightarrow N_3$，$N_1$ 和 N_2 必同时包含于该连通路径。因此$\{N_1, N_3\}$形成$\mathbb{F}$的一个域。证毕。 □

为了进一步控制在一个网格服务流中可以被找出的域的个数，以对每个域的标准粒度进行控制，需要进一步强化域的语义，并使得所有能找出的域可以覆盖整个网格服务流。这就是下面定义最大域和标准域的作用。

定义 5. 2（最大域）：称两个服务控制节点、起始或终止节点 N_{head} 和 N_{tail} 形成了网格服务流$\mathbb{F}$中一个最大域，当对于$\mathbb{F}$中任意从起始到终止节点的路径 $Begin \rightarrow N_1 \rightarrow \ldots N_m \rightarrow End$，$N_{head}$ 和 N_{tail} 都在该连通路经中，且 $N_{head} \neq N_{tail}$。

从定义 5. 2 易得，一个最大域同时必也是一个域，因此它也必满足性质 5. 1。

定义 5. 3（全分解）：网格服务流$\mathbb{F}$中一个最大域$\{N_1, N_2\}$称为是可分解的，若：（1)有多条从 N_1 到 N_2 的连通路径 $N_1 \rightarrow \ldots \rightarrow N' \rightarrow \ldots \rightarrow N_2$，且$\exists N' \in \{N_1, N_2\}$，s.t. $\{N_1, N'\}$或$\{N', N_2\}$是一个最大域；或（2）有唯一一条从 N_1 到 N_2 的连通路径 $N_1 \rightarrow \ldots \rightarrow N' \rightarrow \ldots \rightarrow N_2$，且对于$\mathbb{F}$中任意的 $N_0 \rightarrow N_1$ 和 $N_2 \rightarrow N_3$，$\{N_0, N_2\}$，$\{N_1, N_3\}$或$\{N_0, N_3\}$是一个最大域。由此，对于 $N'_1, \ldots N'_m \in \{N_1, N_2\}$，一个最大域的集合 $\{ \{N', N''\} \mid \{N', N''\} = \{N_1, N'_1\}$或$\{N'_1, N'_2\} \ldots$或$\{N'_m, N_2\}\}$称为是域$\{N_1, N_2\}$的一个全分解（Total Decomposition），当所有$\{N', N''\}$均为不可分解的最大域。

定义 5. 4（标准域）：给定一网格服务流$\mathbb{F}$，由于$\mathbb{F}$自身也形成一个（最大）域，因此称$\{N_1, N_2\}$是$\mathbb{F}$的一个标准域，当$\{N_1, N_2\}$属于$\mathbb{F}$的全分解。

由于$\mathbb{F}$自身也形成一个最大域，因此由定义 5. 2 到定义 5. 4 可知一个网格服务流的全分解（及其标准域）总能存在（最差情况下唯一能找到的标准域就是$\mathbb{F}$自身）。例如在后续的图 5.3 中，虽然$\{TrigBank_H2_3, thIncaII_L1H2\}$是一个域，但它既不是一个标准域也不是一个最大域；而$\{Begin, Inspiral\}$则是该服务流的一个标准域，其中 *Begin* 代表该服务流的起始节点。

通过以下性质的证明可以发现，不同标准域的节点间存在着重要的顺序连接关系，这为将要进行的网格服务流验证策略分解工作提供了重要的时序信息。

性质 5. 2：对于网格服务流$\mathbb{F}$中任意两个不同的标准域$\{N1', N2'\}$和$\{N1'', N2''\}$，可以断言以下两组时序关系中必有一个能够成立：

- 对于任意服务控制节点 $N'' \in \{N_1'', N_2''\}$（包含 N_1''和 N_2''）和任意服务控制节

点 $N' \in \{N_1', N_2'\}$（包括 N_1'和 N_2'），𝔽中总有 $N' \rightarrow \ldots \rightarrow N''$，使得该连通路径由 0 个或多个服务迁移或数据通道组成。此时称 N'为 N''的前继；

- 对于任意服务控制节点 $N' \in \{N_1', N_2'\}$（包含 N_1'和 N_2'）和任意服务控制节点 $N'' \in \{N_1'', N_2''\}$（包括 N_1''和 N_2''），𝔽中总有 $N'' \rightarrow \ldots \rightarrow N'$，使得该连通路径由 0 个或多个服务迁移或数据通道组成。此时称 N''为 N'的前继。

证明：记 $N' \rightarrow N''$表示 N'是 N''的前继。由性质 5. 2 中前继的定义可知它满足若 $N_1 \rightarrow N_2$、$N_2 \rightarrow N_3$ 则 $N_1 \rightarrow N_3$。因此由（最大）域的定义和 5.2.1 小节中的约定 4，对任意服务控制节点 $N' \in \{N_1', N_2'\}$和 $N'' \in \{N_1'', N_2''\}$，分别有 $N_1' \rightarrow N' \rightarrow N_2'$和 $N_1'' \rightarrow N'' \rightarrow N_2''$成立。此外，基于（最大）域的定义和性质 5. 1 可知，由于$\{N_1', N_2'\}$和$\{N_1'', N_2''\}$是两个标准域，因此$\{N_2', N_1''\}$或$\{N_2'', N_1'\}$也能形成一个域。这意味着必有 $N_2' \rightarrow N_1''$或 $N_2'' \rightarrow N_1'$成立。因此，对应于$\{N_2', N_1''\}$或$\{N_2'', N_1'\}$所形成的域，$N_1' \rightarrow N' \rightarrow N_2' \rightarrow N_1'' \rightarrow N'' \rightarrow N_2''$或 $N_1'' \rightarrow N'' \rightarrow N_2'' \rightarrow N_1' \rightarrow N' \rightarrow N_2'$中必有一个关系成立。证毕。 □

基于以上对标准域的定义和性质，图 5.2 中给出了对相应网格服务流全分解的寻找算法 *TotalDecomposition*。该算法的核心思想是：（1）试图找出服务流𝔽中所有不可分解的最大域(Ns, N^*)；（2）使任意找到的两个最大域(N_1, N_2)和(N_3, N_4)都是相邻的（即：$N_2=N_3$），从而形成𝔽的全分解。该算法相当于是对网格服务流在语法层对所有服务流节点的一次遍历，通过 5.2.1 小节中的约定 4 确保了$|Nc|$最终定可为 0，以结束算法中的循环。需要指出的是，由于系统的实际状态空间（在这里即为网格服务流的状态π演算语义所对应的状态标号迁移系统大小）往往随其规模（在这里即为网格服务流中的服务流节点数和迁移数）指数增长[85,88,157]，因此 *TotalDecomposition* 在语法层对网格服务流全分解的寻找代价可以远小于对该网格服务流自身的（LTL）业务逻辑验证代价（$m*2^{O(k)}$）。因为这里的服务流状态标号迁移系统规模 m 将远大于其服务流节点数和迁移数（*TotalDecomposition* 在 LIGO 引力波探测数据分析服务流 SF1~SF3 的应用性能及其与实际业务逻辑验证时间的对比可参见附录 A 中的验证数据结果）。

由此，重新针对第 4 章中图 4.7 的 LIGO 引力波探测数据分析服务流，如图 5.3 所示，它基于 *TotalDecomposition* 方法所得到的全分解大小为 2（即可以找到 2 个标准域）。在后续的 5.2.4 小节中将进一步讨论松弛域分析的分解方法，以强化现有对服务流的分解能力，获得更好的分解结果。更复杂的两个 LIGO 引力波探测数据分析服务流的标准域分解也将在 5.2.5 小节给出。以下先进一步讨论根据已

有网格服务流分解结果而需要进行的相应验证策略分解。

```
Procedure TotalDecomposition
  Define Nc, N*          /*Nc：服务流中所有节点的队列；N*：当前正在处理的服务流节点
       b，Ns=Begin  /*b：计算寻找最大域的标志；Ns：当前寻找最大域的起始节点
       Ns, lev = 1, in, out; Nc.add(Ns); Ns.setVisited, b=true;
  While (|Nc|>0)
    If(b== true)        N*=Nc.removeHead();          /*取出队列中的头服务流节点
    Else                                  /* removeFirstSingleIn用来得到Nc中第一个没有
    N*= Nc.removeFirstSingleIn();        多输入的服务流节点
       If (N* == null)       N*= Nc.removeHead();
    out=|srcTrans(N*)| + |srcFlData(N*)|          /* srcFlData和tarFlData用来获得当前节点的
    in=|tarTrans(N*)| + |tarFlData(N*)|           通畅（Fluent）数据通道（见5.2.1小节约定5）
    If (out <= 1 && in <= 1)
       Nc.add(next_fl(N*) - Visited);  Visited = Visited + next(N*);
              /* next_fl用来获得当前节点通过服务迁移或通畅数据通道的所有后继节点
    Else
       If (b == true)              /* (Ns, N*)形成一个最大域
          在其变为可分解之前记录找到的最大域(Ns, N*);
          b = false; lev = lev + out – in; Ns = N*;
          Nc.add(next_fl(N*) - Visited);  Visited = Visited + next(N*);
       Else
          If (out > 1 && in > 1)     lev' = lev + 1 – in;   /*  处理5.2.1小节的约定2;
          If (lev' == 1)                                    /* (Ns, N*)形成一个最大域
             在其变为可分解之前记录找到的最大域(Ns, N*);
             If (out <= 1)    b = true; Ns = N*;
          Nc.add(next_fl(N*) - Visited);  Visited = Visited + next(N*);
    End while
End Procedure
```

图 5.2 网格服务流全分解算法

5.2.3 基于标准域分析的验证分解

针对以上对服务流的分解结果，本节将继续研究：

（1） 如何利用各标准域间隐含的顺序串行关系，实现对应的网格服务流验证策略的分解？

（2） 如何从标准域的局部验证结果推导出原有网格服务流的全局验证结论？

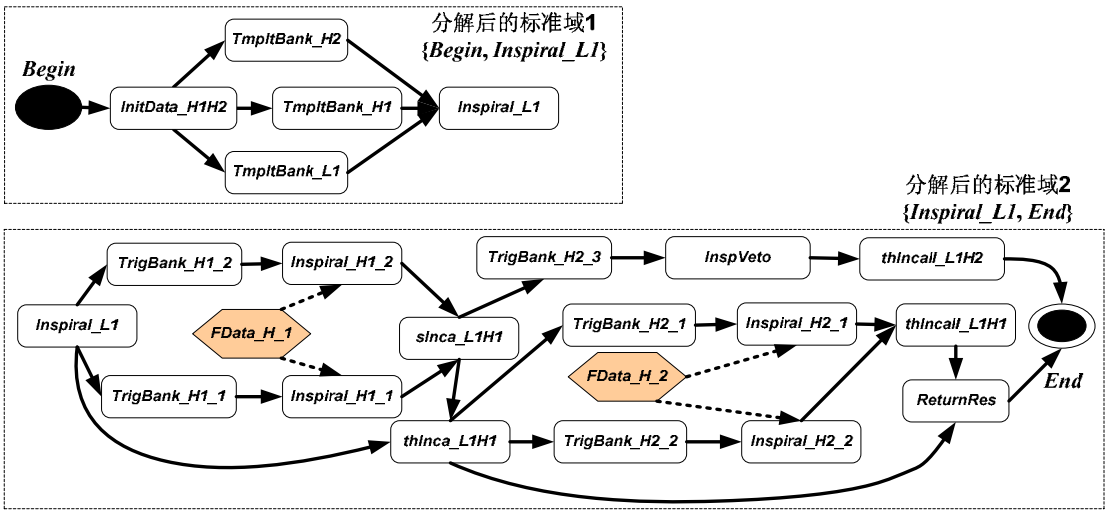


图 5.3 基于标准域分析对服务流 SF1 的分解结果

以上问题实际上是一个模块化模型验证问题[85,86,157,158]的特例。典型的模块化模型验证过程可以表现为以下形式的一种验证推理[85]：

$$\frac{<TRUE>M<\varphi> \qquad <\varphi>M'<\psi>}{<TRUE>M \,||\, M'<\psi>} \tag{5–1}$$

该推理表示：若模块 M 满足性质 φ（即：$<TRUE>M<\varphi>$），而模块 M' 在其环境满足 φ 的前提下满足性质 ψ（即：$<\varphi>M'<\psi>$），则期望得到 M 和 M' 的并发组合“||”能够满足性质 ψ 的结论。而在本节问题中，若将以上的模块 M、M' 视为目标网格服务流中所分解出的标准域，则根据性质 5. **2** 中标准域间隐含的顺序关系特性，以上推理 5–1 可以转变为分析两个标准域 M 和 M' 的顺序组合（$M;M'$）是否能够满足性质 ψ 的过程。即：

$$\frac{<TRUE>M<\varphi> \qquad <\varphi>M'<\psi>}{<TRUE>M;M'<\psi>} \tag{5–2}$$

由此，记图 5.2 算法所得网格服务流 $\mathbb{F}$ 的全分解为 $\{M_1, M_2, \ldots, M_n\}$，其中 $M_i=\{N_i, N_{i+1}\}$，N_i、$N_{i+1}\in\mathbb{F}$（$i=1,2,\ldots,n$），则基于标准域分析的业务逻辑验证分解思路在于：期望借助以上原子推理 5–2 的成立，将待验证业务逻辑 ψ 在 $\mathbb{F}$ 的各个标准域上按从 M_n 到 M_1 的逆向顺序逐一进行分别验证。其中在 M_i 上对 ψ 的验证将在已知其后续标准域 M_j（$i<j\le n$）对 ψ 的验证结果基础上进行，从而最终推理出整个服务流 $\mathbb{F}$ 是否满足 ψ 的结论（见推理 5–3）。

$$\frac{<TRUE>M_{i+1};.....;M_n<\psi_{i+1}> \qquad <\psi_{i+1}>M_i<\psi_i>}{<TRUE>M_i;M_{i+1};.....;M_n<\psi_i>} \quad 1\le i\le n-1 \quad (5\text{–}3)$$

以上过程中的关键环节在于如何验证$<\psi_{i+1}>M_i<\psi_i>$，以保证推理过程 5–3 的成立。由于本文 LIGO 数据网格应用中的业务逻辑是基于 LTL 逻辑公式进行的刻画（更严格的，为了保证待验证逻辑的 Stuttering Closedness [56,141]，它采用的是 LTL-X，即不包含 Next 操作“*X*”的 LTL），因此在$<\psi_{i+1}>M_i<\psi_i>$中ψ_i和ψ_{i+1}均针对 LTL 逻辑公式进行。正如 1.4.2 小节的综述所述，利用 LTL 逻辑的一个优势在于它对模块化模型验证的直接支持与简化。即由于 LTL 逻辑公式可以等价转换为接受相同状态序列的自动机，因此对$<\psi_{i+1}>M_i<\psi_i>$的验证可直接通过判别$Trans(\psi_{i+1})||M_i \vDash \psi_i$来实现①[85,158]。然而该方法中却没有有效利用到本问题中各标准域 M_i 间所隐含的顺序关系特性，因此它仍然需要额外占用对自动机组合的操作代价。以下则将讨论如何利用已知标准域间的顺序组合关系，从而基于 LTL(-X) 在模型路径和状态上的语义给出$<\psi_{i+1}>M_i<\psi_i>$的实现。

定义 5. 5（衔接状态）：给定一网格服务流$\mathbb{F}$的全分解为$\{M_1, M_2, ..., M_n\}$，其中$M_i=\{N_i, N_{i+1}\}$，N_i、$N_{i+1}\in\mathbb{F}$（$i=1,2,...,n$）。记 $TransSys(\mathbb{F},\Phi)$为$\mathbb{F}$在初始状态集$\Phi$下对应的状态标号迁移系统$(S, M, \{\xrightarrow{a\{StateExpr\}} | a\in M\})$②；记 $SPi(M_i)$为标准域 M_i 所对应的状态π演算形式化模型。则由于 M_{i-1} 和 M_i 共享了节点 N_i，称以 $SPi(M_{i+1};M_{i+2};...;M_n)$ 为状态π演算进程标识的状态集合$\{s \mid (P, s)\in S, P \equiv SPi(M_{i+1};M_{i+2};...;M_n)\}$ 为在 $TransSys(\mathbb{F},\Phi)$ 中从 $SPi(M_i;M_{i+1};...;M_n)$ 迁移到 $SPi(M_{i+1};M_{i+2};...;M_n)$的衔接状态集合 $Im(\mathbb{F}, M_i)$。

衔接状态集合表示了以上 5–3 的逆向递归推理中，前一次推理所涉及的标准域和本次待推理标准域间分别的“域初始状态”和“域终止状态”。

定义 5. 6（域初始/终止状态）：给定一网格服务流$\mathbb{F}$的全分解为$\{M_1, M_2, ..., M_n\}$，其中 $M_i=\{N_i, N_{i+1}\}$，N_i、$N_{i+1}\in\mathbb{F}$（$i=1,2,...,n$）。称 $Im(\mathbb{F}, M_i)$分别为标准域 M_i 的域终止状态集$\mathbb{E}(M_i)$和 $M_{i+1};M_{i+2};...;M_n$ 的域初始状态集$\mathbb{S}(M_{i+1};M_{i+2};...;M_n)$。

引理：记 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$和 $TransSys(M_{i+1}, \mathbb{S}(M_{i+1};...;M_n))$分别为网格服务流$\mathbb{F}$中标准域 M_i 和 M_{i+1} 对应的标号迁移系统$(S_i, M_i, \{\xrightarrow{a\{StateExpr\}} | a\in M_i\})$和$(S_{i+1}, M_{i+1}, \{\xrightarrow{a\{StateExpr\}} | a\in M_{i+1}\})$，则 $S_i \cap S_{i+1} = Im(\mathbb{F}, M_i) \neq \text{Empty}$。

该引理描述了在一网格服务流$\mathbb{F}$的全分解$\{M_1, M_2, ..., M_n\}$中，M_i和 M_{i+1}所对

① $Trans(\psi)$表示公式ψ所对应的扩展 Buchi 自动机（见 4.6 小节），下同。
② 状态标号迁移系统的实现方法 *TransSys* 已在上一章的 4.2 和 4.3 小节中详细给出。

应的状态标号迁移系统在状态上的唯一非空交集为 $Im(\mathbb{F}, M_i)$，而不存在 M_{i+1} 中状态到 M_i 中状态的环。它是本章中最大域的定义和约定 6 的直接结果。

性质 5. 3：记 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$为网格服务流$\mathbb{F}$中标准域 M_i 对应的状态标号迁移系统$(S, M, \{\xrightarrow{a\{StateExpr\}} | a \in M\})$。给定 LTL-X 公式$\Psi$及其规范化（Canonical Form）$\varphi$，若已知$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$，则$<TRUE>M_i;M_{i+1};M_{i+2};...;M_n<\varphi>$成立，当：

- 若$\varphi=p \in AP$（AP 为所有原子命题集合），则$<\varphi>M_i<p>$若 $M_i, \mathbb{S}(M_i;...;M_n) \models p$；
- 若$\varphi=\varphi_1 \vee \varphi_2$，则$<\varphi>M_i <\varphi_1 \vee \varphi_2>$若 $M_i, \mathbb{S}(M_i;...;M_n) \models \varphi_1$ 或 $M_i, \mathbb{S}(M_i;...;M_n) \models \varphi_2$；
- 若$\varphi=\neg\varphi_1$，则$<\varphi>M_i <\neg\varphi_1>$若 $M_i, \mathbb{S}(M_i;...;M_n) \not\models \varphi_1$；
- 若$\varphi=\varphi_1$ U φ_2，则$<\varphi>M_i<\varphi_1$ U $\varphi_2>$若对任意以$\mathbb{S}(M_i;...;M_n)$为初始的状态序列 $\pi=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots\sigma_k$，有：
  - $\exists i \in N^+$，s.t. $\sigma_i \in \pi$且$\sigma_i \in \mathbb{S}(M_{i+1};...;M_n)$，$M_{i+1};...;M_n, \sigma_i \models \varphi_1$ U φ_2。$\forall$无限状态序列$\pi' \in Trans(\varphi_1$ U $\varphi_2) = \sigma^0, \sigma^1, \sigma^2, \ldots$，对于任意这样的$\sigma_i$和$\pi''=\mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma^1, \sigma^2, \ldots$，有 $M_i, \pi'' \models \varphi_1$ U φ_2 成立；否则：
  - $M_i, \pi^* \models \varphi_1$ U φ_2，其中$\pi^*=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots$；
- 若$\varphi=$G φ_1，则$<\varphi>M_i<$G $\varphi_1>$若对任意以$\mathbb{S}(M_i;...;M_n)$为初始的状态序列 $\pi=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots\sigma_k$，有：
  - $\exists i \in N^+$，s.t. $\sigma_i \in \pi$且$\sigma_i \in \mathbb{S}(M_{i+1};...;M_n)$，$M_{i+1};...;M_n, \sigma_i \models$ G φ_1。$\forall$无限状态序列$\pi' \in Trans($G $\varphi_1) = \sigma^0, \sigma^1, \sigma^2, \ldots$，对于任意这样的$\sigma_i$和$\pi''=\mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma^1, \sigma^2, \ldots$，有 $M_i, \pi'' \models$ G φ_1 成立；否则：
  - $M_i, \pi^* \models$ G φ_1，其中$\pi^*=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots$；
- 若$\varphi=\varphi_1$ W φ_2，则$<\varphi>M_i<\varphi_1$ W $\varphi_2>$若$<\varphi>M_i<$G $\varphi_1 \vee (\varphi_1$ U $\varphi_2)>$；
- 若$\varphi=\varphi_1$ R φ_2，则$<\varphi>M_i<\varphi_1$ W $\varphi_2>$若$<\varphi>M_i<$G $\varphi_2 \vee (\varphi_2$ U $(\varphi_2 \wedge \varphi_1))>$；
- 若$\varphi=$F φ_1，则$<\varphi>M_i<$F $\varphi_1>$若$<\varphi>M_i<TRUE$ U $\varphi_1)>$；

以上性质的证明可参见附录 C，它蕴含了基于标准域的业务逻辑验证分解的重要充分条件：即可以通过已知标准域的已满足性质与待验证标准域的顺序组合来逆向推导对整个网格服务流的性质验证结果。更直观的，该验证分解策略可以由以下推论直接表示。

推论：记 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$为网格服务流$\mathbb{F}$中标准域 M_i 对应的状态标号迁移系统$(S, M, \{\xrightarrow{a\{StateExpr\}} | a \in M\})$。给定 LTL-X 公式$\Psi$及其规范化$\varphi$，若已知$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$，则$<TRUE>M_i;M_{i+1};M_{i+2};...;M_n<\varphi>$成立，当：

$(TransSys(M_i, \mathbb{S}(M_i;...;M_n));Trans(\varphi)), \mathbb{S}(M_i;...;M_n) \vDash \varphi$ 。

该推论是性质 5. 3 和推理 5–3 的直接结果。其中 $(TransSys(M_i, \mathbb{S}(M_i;...;M_n));Trans(\varphi))$表示模型 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$在其终止状态下与性质$\varphi$自动机的顺序组合结果：$\mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma^1, \sigma^2, \ldots$（即通过$\sigma_i \in \mathbb{S}(M_{i+1};...;M_n)$的状态序列顺序组合）。至此本章形成了对应于标准域分析的网格服务流验证分解策略以及相应的服务流证实方法。

5.2.4　基于松弛域分析的服务流分解

以上基于标准域分析的服务流及验证分解仍有的一个不足在于对网格服务流模型自身的约定太强，从而在网格服务流$\mathbb{F}$中得到的标准域仍不能小到使验证的时间性能和内存耗费有足够满意的提高。以 5.2.2 小节中的分解结果为例，其标准域{*Inspiral_L1*，*End*}相对原服务流仍然有着较大的规模。因此在 5.2.2 小节中标准域分解的基础上，本节试图对 5.2.1 小节中的约定 1 进行进一步松弛，允许网格服务流中多个终止节点的存在，从而寻找其中可能存在的并发执行分支。

约定（松弛）：网格服务流的起始节点和终止节点标志着整个服务流执行过程的起始和执行路径的终止，其中，每个服务流都有唯一的一个起始节点，并松弛为允许拥有多个终止节点。称松弛后拥有的新终止节点为副终止节点（*VEnd*）。

后续的图 5.5 中给出了对主终止节点（*End*）和副终止节点（*VEnd*）进行显示上区分的例子。对于一个标准域 M_i，记 $M_i / (N_1 \rightarrow ... \rightarrow N_m)$（$N_j \in M_i$，$j=1,...,m$）表示从 M_i 中删除一条连通路径和相应的服务迁移/数据通道。称“/”为对标准域的剪枝操作。则期望在对标准域进行适当的临时剪枝后，可以使网格服务流中能够找出更多的标准域，且使 5.2.3 小节中业务逻辑验证分解的性质同样有效。与以上的标准域分析方法（Standard Region Analysis）相对应，本节称这种基于临时剪枝的域分析方法为松弛域分析方法（Relaxed Region Analysis）。

定义 5. 7（待剪并发枝）：在全分解为$\{M_1, M_2, ..., M_n\}$的网格服务流$\mathbb{F}$中，称 M_n 中的一个连通路径$\mathbb{CP}=N_1 \rightarrow^* N_2 \rightarrow \ldots \rightarrow VEnd$ 是一个待剪并发枝，当：（1）在 $M_n / (N_2 \rightarrow \ldots \rightarrow Vend)$的全分解$\{M'_1, M'_2, ..., M'_m\}$中，若 $N_1 \in M'_i$（$i=1,2,...m$），则$\forall i < j \leq m$，$\notin M_j$ 中的连通路径 $N_{j1} \rightarrow ... \rightarrow N_{jk}$，s.t. $\exists N_{jk} \in N_2 \rightarrow \ldots \rightarrow VEnd$；（2）$\notin M'_i$ 中的其它连通路径 $N_0 \rightarrow^* N_1 \rightarrow N_2 \rightarrow \ldots \rightarrow VEnd$ 使得它也满足条件（1）；（3）$\rightarrow^*$是一个不带条件的迁移。

以上的待剪并发枝$\mathbb{CP} = N_1 \rightarrow^* N_2 \rightarrow \ldots \rightarrow VEnd$ 表达了一条到副终止节点

（*VEnd*）的连通路径，使得$\mathbb{CP}' = N_2 \rightarrow \ldots \rightarrow Vend$ 中的所有节点与节点 N_1 后能找到的新标准域间存在着并发组合的关系（没有任何服务迁移或数据通道的控制约束）。例如，在后续图 5.5 中的引力波探测数据分析服务流中，*sInca_L1H1* →* *TrigBank_H2_3* → *InspVeto* → *thIncaII_L1H2* → *VEnd* 就是一个待剪并发枝。对于 $M_n / (N_2 \rightarrow \ldots \rightarrow Vend)$的全分解$\{M'_1, M'_2, \ldots, M'_m\}$中，若 $N_1 \in M'_i$，则称待剪并发枝$\mathbb{CP}$属于 M'_i，记为 $M'_i(\mathbb{CP})$。而称$\{M_1, M_2, \ldots, M'_1, M'_2, \ldots, M'_i(\mathbb{CP}), \ldots, M'_m\}$为一个全分解是$\{M_1, M_2, \ldots, M_n\}$的网格服务流$\mathbb{F}$在进行$\mathbb{CP}=N_1 \rightarrow^* N_2 \rightarrow \ldots \rightarrow Vend$的并发剪枝操作后形成的全分解。因此对于 $M'_i(\mathbb{CP})$，由于属于 M'_i 的待剪并发枝和剩余标准域（$M'_{i+1},\ldots, M'_m$）存在相同的并发关系特性，这使得对于 $i \le k < m$ 和相同的初始状态集合，仍有$\mathbb{E}(M'_k \| \mathbb{CP}') = \mathbb{S}(M'_{k+1}, \ldots, M'_m \| \mathbb{CP}') \neq \text{Empty}$。因此对于松弛域分析的验证分解仍可以基于上一小节结论表达为以下四个阶段的推理过程：

$$\begin{cases} \dfrac{<TRUE>(M'_{j+1};.....;M'_m) \| \mathbb{CP} <\psi'_{j+1}> \quad <\psi'_{j+1}> M'_j \| \mathbb{CP} <\psi'_j>}{<TRUE> M'_j; M'_{j+1};.....;M'_m \| \mathbb{CP} <\psi_i>} & i < j \le m \\ \dfrac{<TRUE>(M'_{i+1};.....;M'_m) \| \mathbb{CP} <\psi'_{i+1}> \quad <\psi_{i+1}> M'_i(\mathbb{CP}) <\psi'_i>}{<TRUE> M'_i; M'_{i+1};.....;M'_m \| \mathbb{CP} <\psi_i>} & \\ \dfrac{<TRUE> M'_{l+1};.....;M'_m <\psi'_{l+1}> \quad <\psi'_{l+1}> M'_l <\psi'_l>}{<TRUE> M'_l; M'_{l+1};.....;M'_m <\psi'_l>} & 1 \le l \le i-1 \\ \dfrac{<TRUE> M_{k+1};.....;M_{n-1};M'_1;.....;M'_m <\psi_{k+1}> \quad <\psi_{k+1}> M_k <\psi_k>}{<TRUE> M_k; M_{k+1};.....;M_{n-1};M'_1;.....;M'_m <\psi_k>} & 1 \le k \le n \end{cases} \tag{5–4}$$

以上推理 5–4 的前两种情况是针对需要考虑待剪并发枝的标准域进行的（不失一般性，设$\mathbb{CP}$属于 M'_i），而后两者则处理了 5.2.3 小节中的一般情况。图 5.4 中基于标准域分析的 *TotalDecomposition* 算法给出了松弛域分析的对应算法流程。

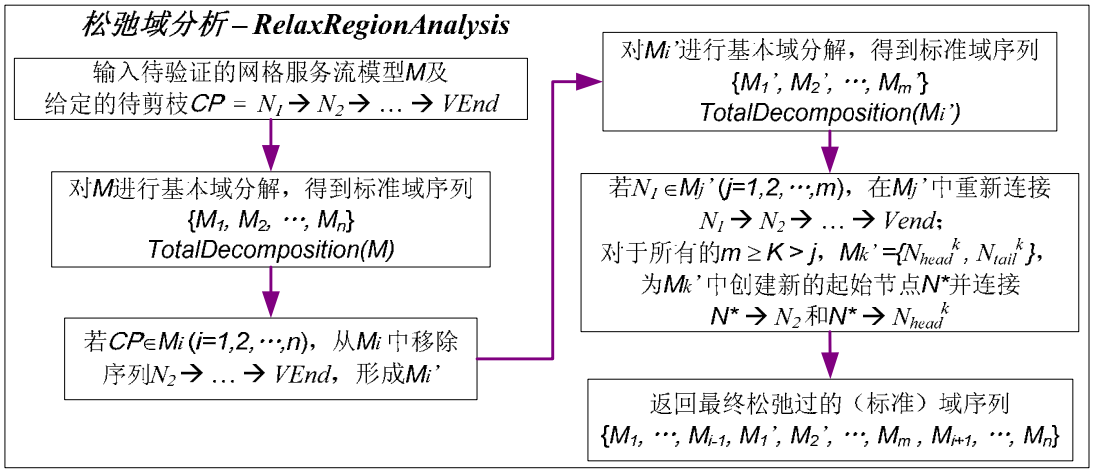


图 5.4 松弛域分析算法流程

重新以 LIGO 引力波探测数据分析的服务流 SF1 为例，通过以 *sInca_L1H1* →* *TrigBank_H2_3* → *InspVeto* → *thIncaII_L1H2* → *VEnd* 为待剪并发枝，该服务流在标准域分析的分解结果基础上可以进一步对原有标准域{*Inspiral_L1*，*End*}进行松弛，从而将其拆分为两个更小的域：{*Inspiral_L1*，*thInca_L1H1*}和{*thInca_L1H1*，*End*}（见图 5.5）。

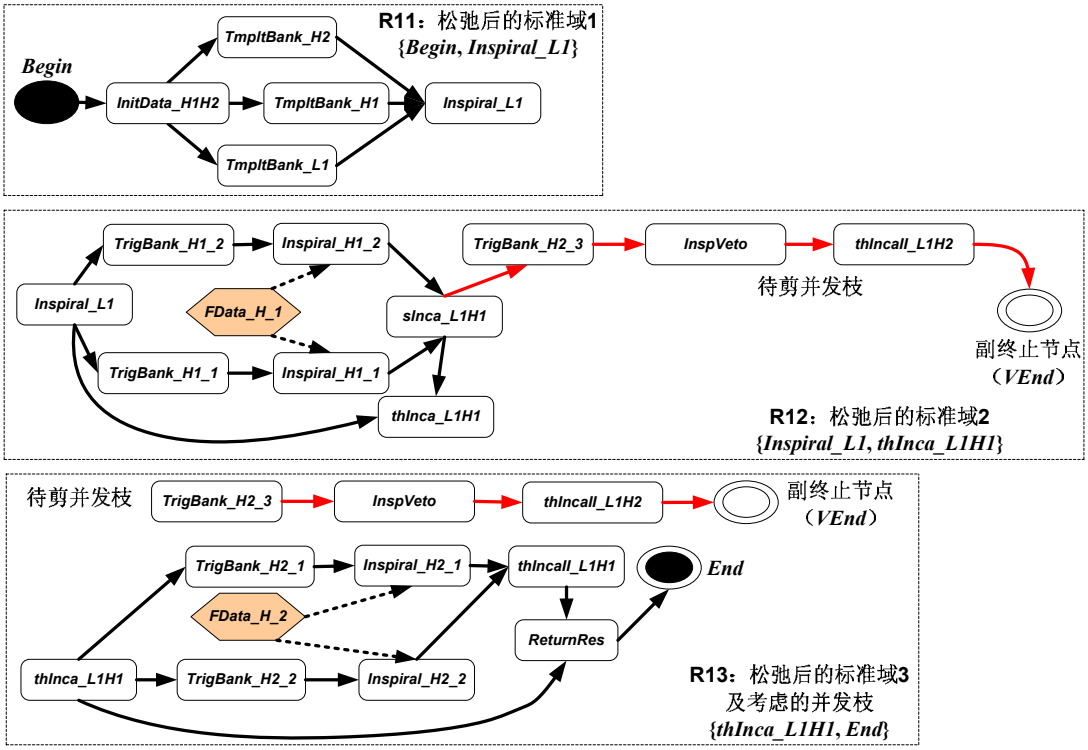


图 5.5 基于松弛域分析对服务流 SF1 的分解结果

5.2.5　基于域分析的验证分解实现与应用结果讨论

在以上服务流和业务逻辑验证分解的研究结果基础上，图 5.6 中完整给出了基于标准域和松弛域分析的验证分解方法实现步骤。通过该实现步骤使得本节中网格服务流基于域分析方法的验证分解能在本文原型系统 GridPiAnalyzer（见第 6 章）中进行集成和自动化的验证实现。

图 5.6 中实现了对网格服务流进行证实的分解策略。它根据目标服务流的各个标准域，基于 5.2.3 小节中的推理结果，分别实现了对其状态标号迁移系统生成过程的逐一前向迭代，和业务逻辑验证过程的逆向迭代分解。其中，在逐一对各标准域的状态标号迁移系统进行前向迭代生成的同时，获得了后续标准域在状态标号迁移系统生成时的初始状态集合$\mathbb{S}$；而逆向的验证迭代分解则基于了 5.2.3 和

5.2.4 小节的研究结果。这使得对整个网格服务流的验证工作从其状态标号迁移系统的生成到业务逻辑验证都可以分别在各个标准域上分别进行，而不需要一次生成完整的状态π演算形式化语义并在整个形式化模型上进行全局验证，从而节约对复杂网格服务流在验证时的内存消耗，并同时有效提高验证的效率。

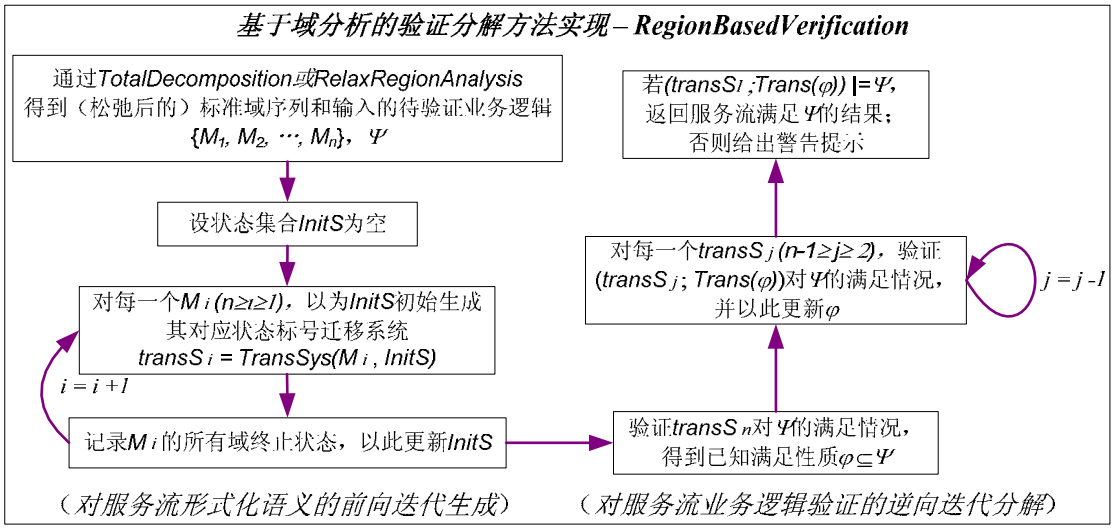


图 5.6 基于域分析的验证分解方法实现

为了验证以上方法的有效性，本文将其应用于 LIGO 数据网格引力波探测数据分析的服务流验证中。待分析服务流除了图 5.5 中的 SF1 外（其原始模型可参见第四章图 4.7），还包括两个复杂的完整实例[27,49]（分别记为 SF2 和 SF3）。图 5.7 至图 5.10 分别给出了 SF2 和 SF3 以及它们的域分解结果。从附录 A 对这三个实例的完整验证性能可以看到，SF2 和 SF3 相对 SF1 具有明显的更大复杂度，其可达状态分别达到 23096（$2^{14.4954}$）和 43073（$2^{15.3945}$）。此处待验证的业务逻辑仍是第 4 章的 8 条关键业务逻辑。针对 SF1、SF2 和 SF3 的具体待验证 LTL 公式可参见附录 B（其中对 *Inspiral_H1.Exit* 等的定义与 4.4.2 小节相同）。

为检验本章方法的有效性，本文将以上基于标准域/松弛域的验证分解方法与经典的符号模型验证算法[103]（记为 SMCA）、带影响锥（Cone of Influence）的符号模型验证算法[103]（记为 SMCA+COI）、带二元决策树（BDD）变量动态排序的符号模型验证算法[103]（记为 SMCA+Dynamic）和 Bounded Model Checking 算法[84]（BMC，其步长设置为 k）相比较。图 5.11 至图 5.16 以及附录 A 中分别给出了对 SF1~SF3 在验证时间和最大内存耗费上的完整性能比较数据。图中的各缩写代表含义如下：

- R_{ij}：服务流 SF_i 中所分解出的第 j 个标准域；
- p_i 和 p_{ij}：待验证的 LTL 业务逻辑公式；
- SMCA’：符号模型验证算法 SMCA 的可达状态计算时间；
- SMCA：符号模型验证算法 SMCA 的验证总时间；
- COI：SMCA+COI 的验证总时间；
- Dynamic：SMCA+Dynamic 的验证总时间；
- BMC（k）：BMC 的验证总时间，k 为其步长设置；

其中，验证时间和内存占用分别设置了相应的阀值（超过 10 分钟或小于 10Mb）。本文的验证硬件环境为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。

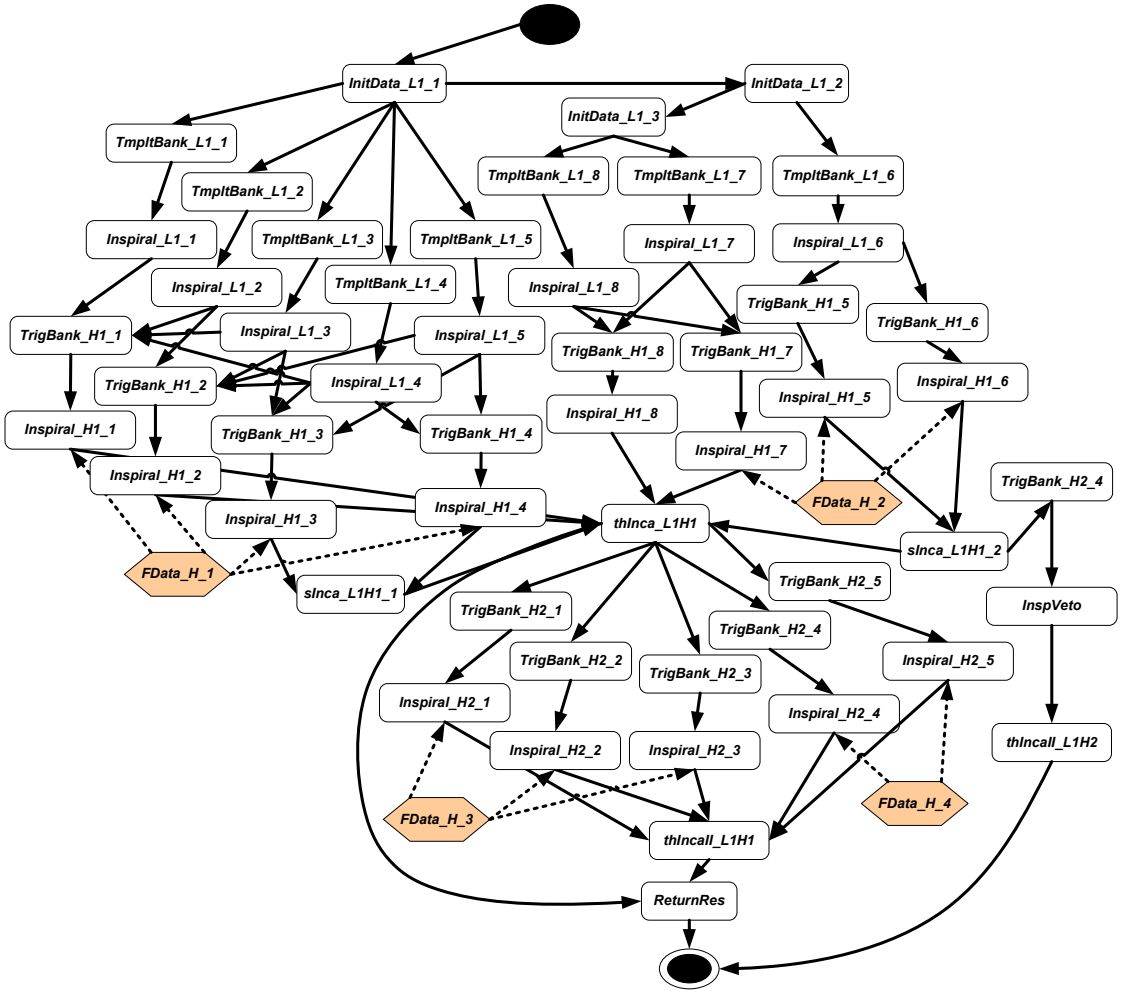


图 5.7 引力波探测数据分析的服务流实例 SF2

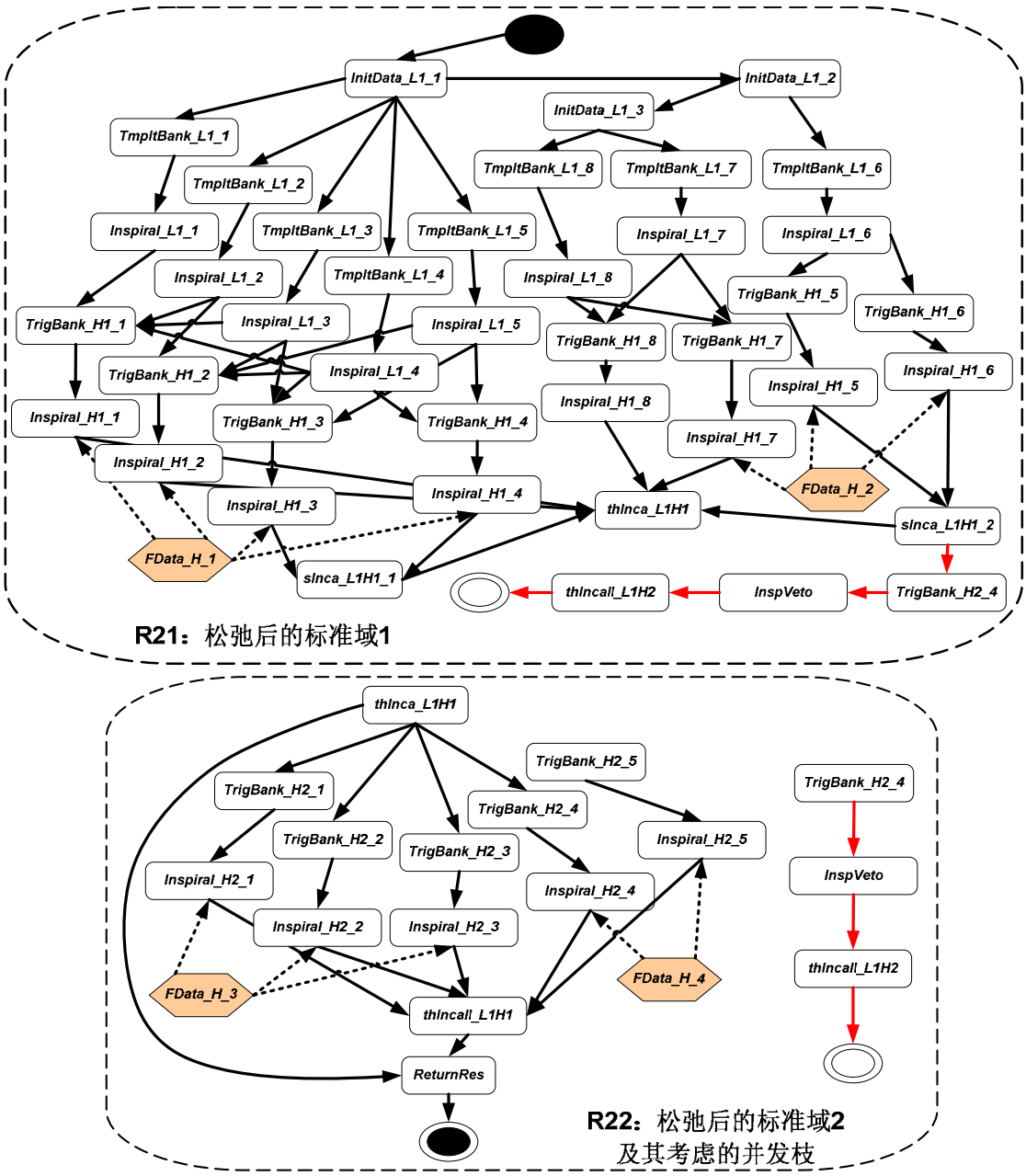


图 5.8　SF2 的域分解结果

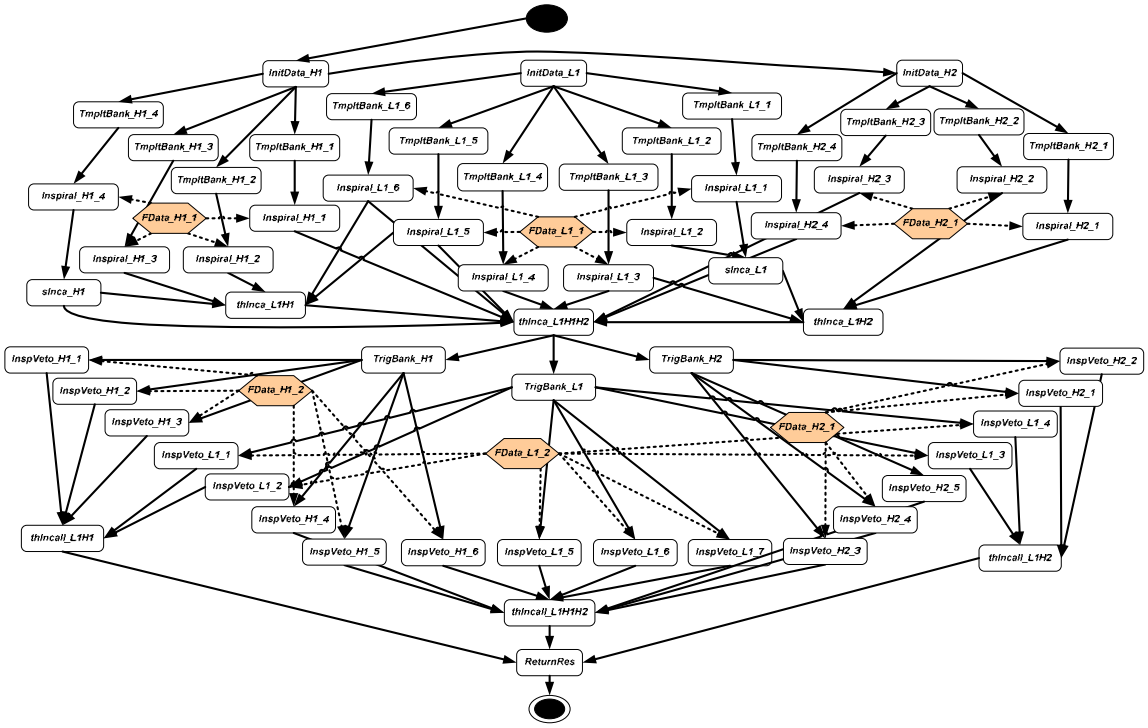


图 5.9 引力波探测数据分析的服务流实例 SF3

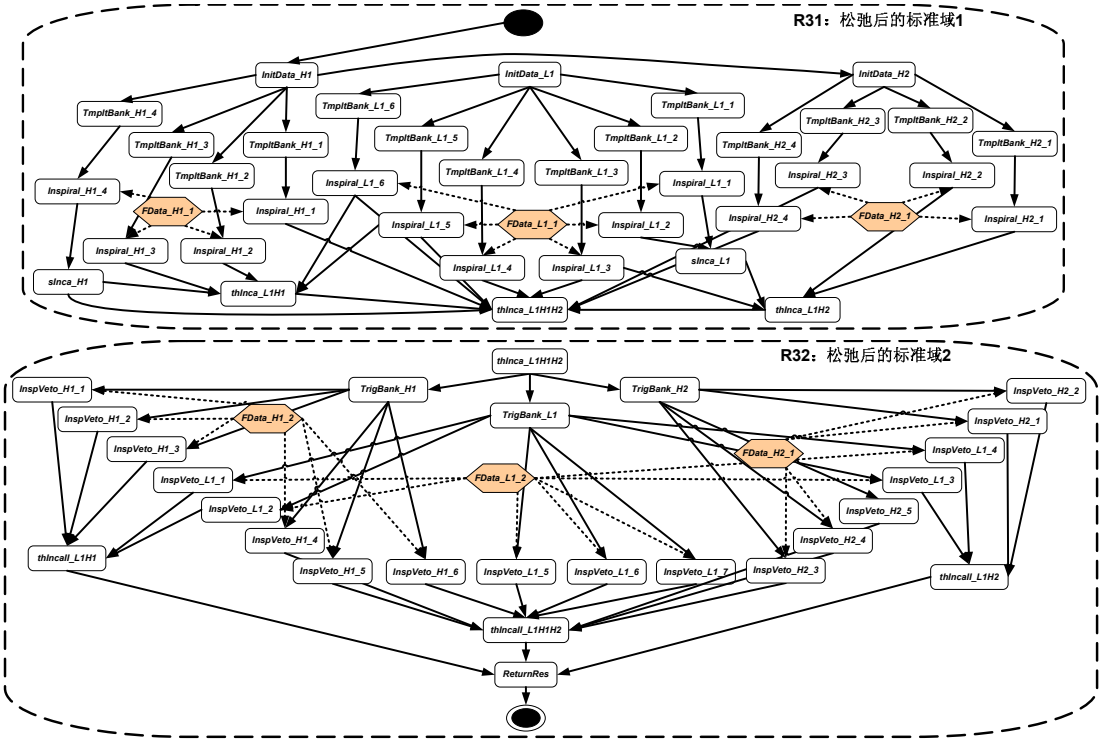
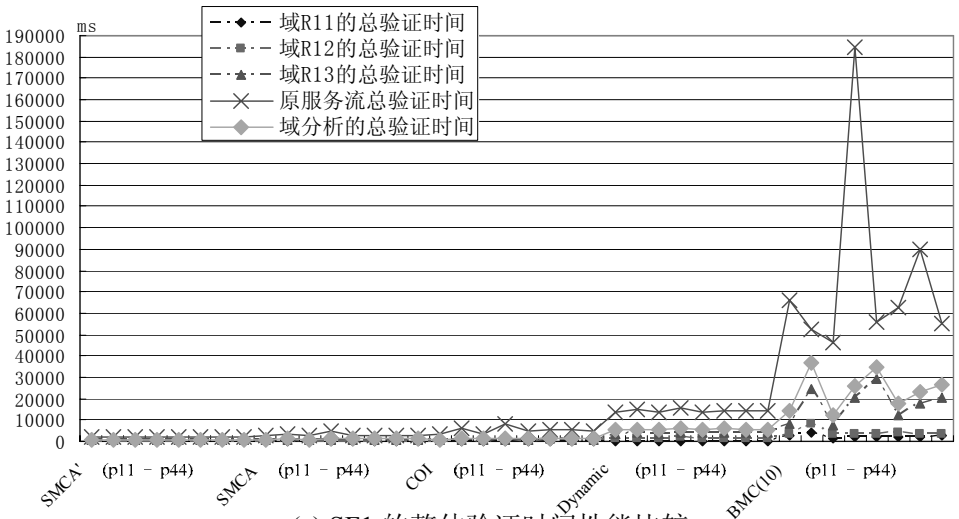
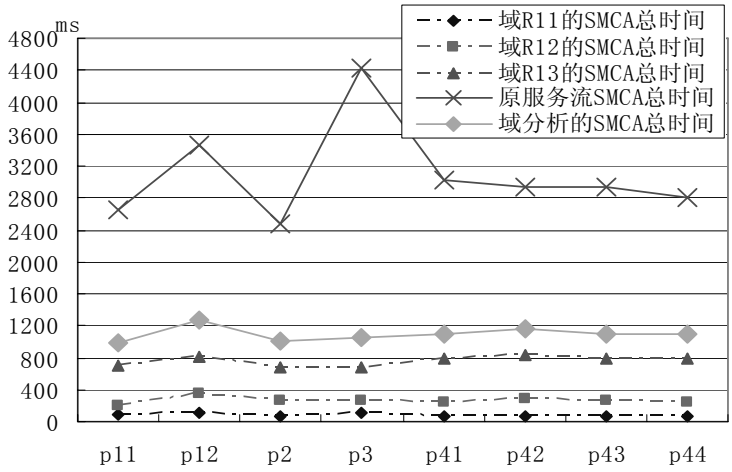


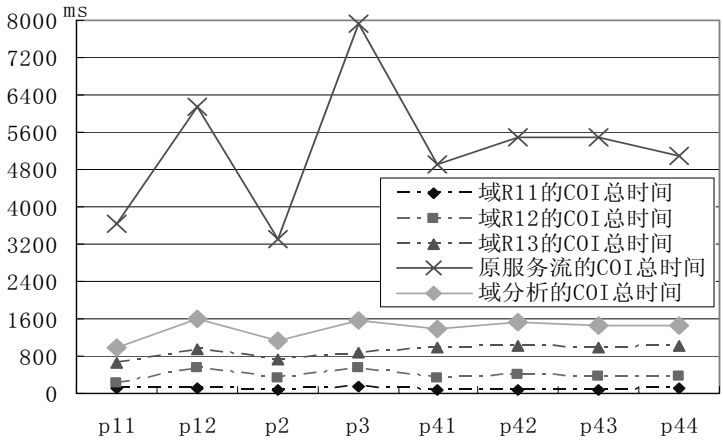
图 5.10　SF3 的域分解结果



(a) SF1 的整体验证时间性能比较



(b) SF1 基于 SMCA 方法的验证时间性能比较



(c) SF1 基于 SMCA+COI 方法的验证时间性能比较

图 5.11 SF1 的验证时间性能比较（单位：ms）

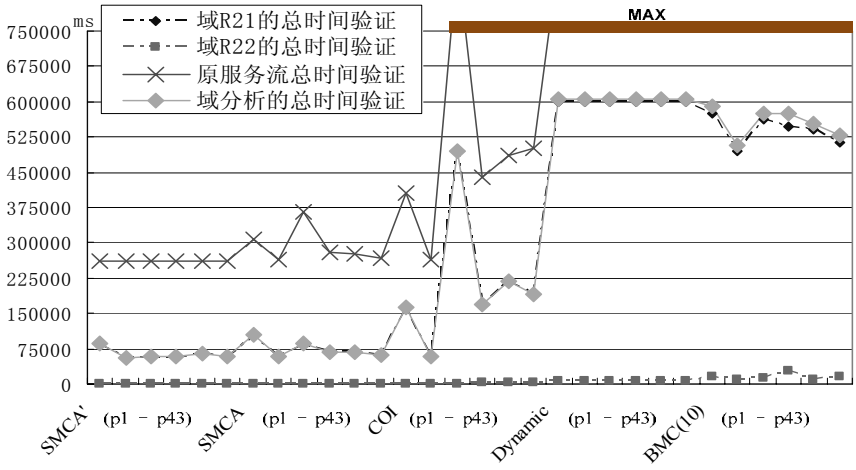


图 5.12 SF2 的验证时间性能比较（单位：ms）

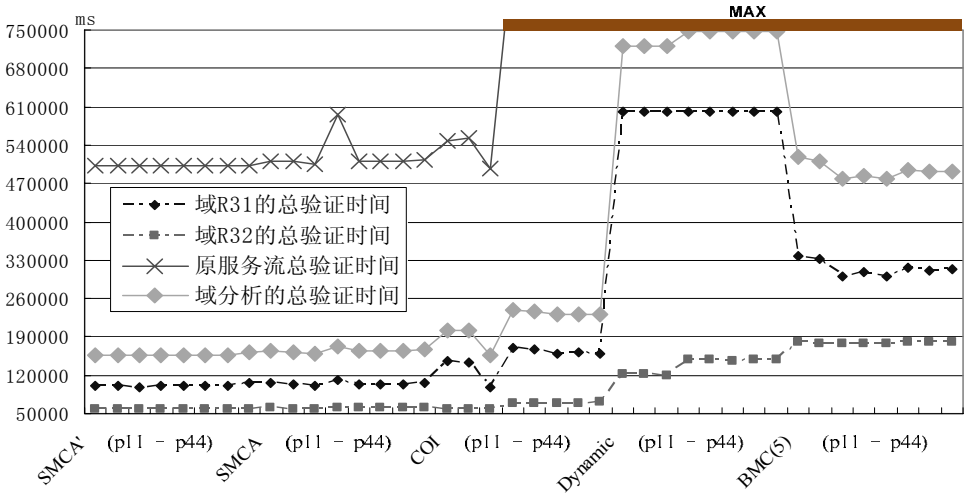


图 5.13 SF3 的验证时间性能比较（单位：ms）

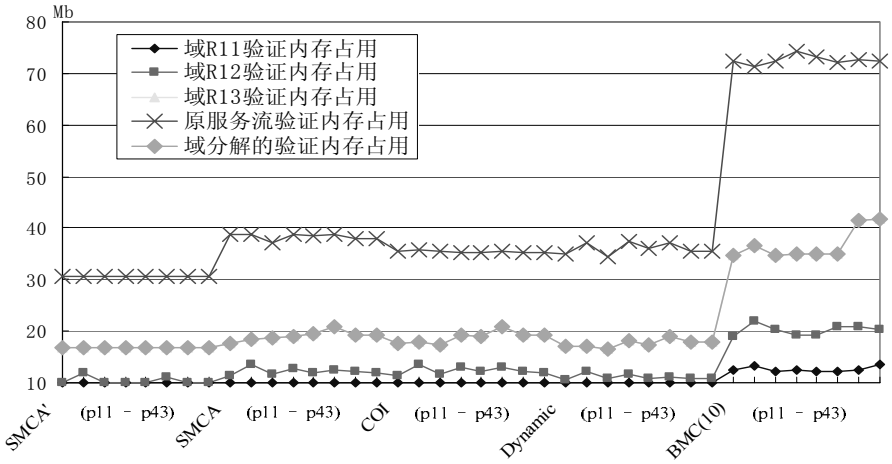


图 5.14 SF1 的验证内存占用性能比较（单位：Mb）

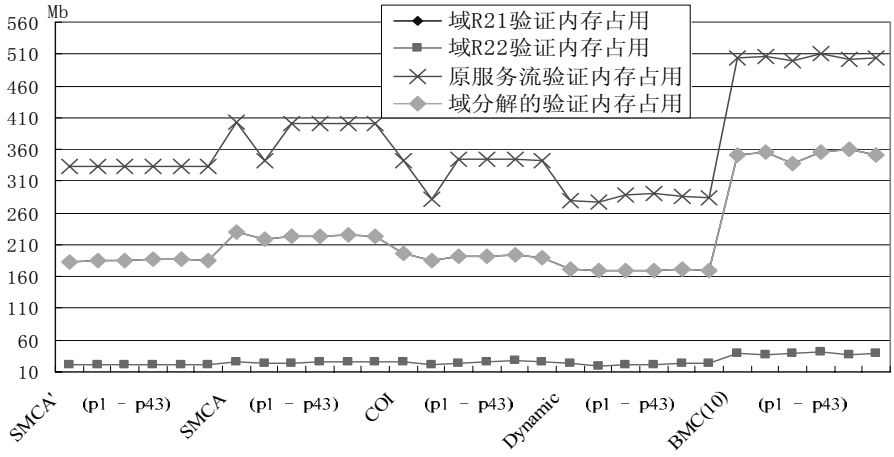


图 5.15 SF2 的验证内存占用性能比较（单位：Mb）

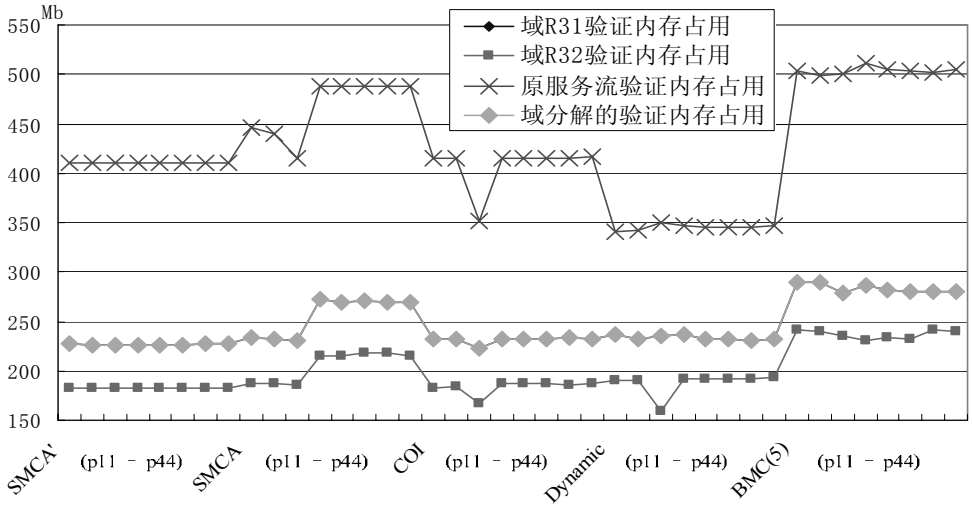


图 5.16 SF3 的验证内存占用性能比较（单位：Mb）

图 5.11~图 5.16 中基于域分析的验证分解方法对业务逻辑 p 的验证总时间等于各个域上分别对 p 进行验证的总时间之和：$t_p=\sum_{i=1}^{n} t_p(R_i)$；而验证最大内存占用等于各个域上分别对 p 进行验证时的最大内存占用：$m_p=Max(m_p(R_i))$，$i=1,\ldots,n$。其中图 5.11(b)和(c)分别是对 SF1 验证性能中 SMCA 和 SMCA+COI 方法性能对比的放大。从图 5.11~图 5.16 以及附录 A 中的详细数据可以看出，由于 SF2 和 SF3 相对 SF1 具有较大的复杂度，因而在直接对它们进行业务逻辑验证时的验证时间和内存占用均较大，其中在基于 SMCA 和 SMCA+COI 方法对两服务流的验证时间分别至少达到 250000 毫秒和大于 500000 毫秒；最大内存占用分别达到 400Mb 和 480Mb。而在基于 SMCA+Dynamic 和 SMCA+BMC 方法对两服务流的验证时

间则更超过了验证时间阀值（10 分钟），最大内存占用分别达到 290Mb、510Mb 和 349Mb、511Mb（以上数据中 BMC 方法所使用的 SAT Solver 是 MiniSat[159]，此处对 SMCA 和 BMC 方法自身的性能比较不是本文的讨论范围）。而在基于本文的验证分解方法进行验证时，不论在验证时间还是最大内存耗费上都得到了明显的改善。其中，在域分解的验证分解方法下基于 SMCA 和 SMCA+COI 对两服务流的验证时间分别降为 60000 毫秒和 160000 毫秒；最大内存占用分别降为 230Mb 和 270Mb。而在基于 SMCA+Dynamic 和 SMCA +BMC 方法对两服务流的验证时间分别降为 590000 毫秒和 480000 毫秒，最大内存占用分别降为 171Mb、360Mb 和 236Mb、290Mb。而此处 SMCA+Dynamic 对两服务流第一个标准域的验证总时间仍然超过了阀值，这一方面是由于 SMCA+Dynamic 方法本质上是一种注重降低内存占用的优化方法，而其额外的计算量本身也会造成验证时间的变大，另一方面也是因为分解出的第一个标准域仍具有一定的复杂度所造成的。在以上结果中，基于域分析的网格服务流验证分解方法在内存占用上拥有明显改进的原因是直观的，因为它实现了对整个服务流的局部读入和局部验证。而它在验证时间上拥有的改进则是由于各分解后的标准/松弛域在状态数和所考虑命题数的下降。因此在各个分解后的标准/松弛域上进行业务逻辑验证不仅降低了实际的业务逻辑验证时间，同时能在实际中减少验证的初始化和内存操作时间，从而在整体上得到验证性能的有效提升。

5.3　基于错误过程模式的验证向导方法

除了对网格服务流实现业务逻辑验证分解的思想，本小节中还通过引入“错误过程模式”的概念，对网格服务流进行带向导的模型验证（Guided Model Checking），从而加速寻找网格服务流中可能存在的错误行为。带向导的模型验证其基本思路是在目标系统的状态空间搜索过程中将每次迭代限定于最“感兴趣的”状态上，从而能够更有效地验证系统的期望性质。文献[160]首先在模型验证基础上测试了如目标放大（Target Enlargement）等一般性启发式向导策略的可行性；Seppi 等人[161]则进一步将随机因素考虑进来，提出了基于经验贝叶斯法的向导策略；另一方面，文献[162]和[89]则分别研究了基于符号化方法对不变量验证和 CTL 模型验证的向导搜索。其中在每一步不动点迭代中，通过对系统行为所设置的限定（该文中称为“Hints”），从而产生系统模型的上下近似（over / under

approximation）。Prakash 等人[90]则尝试了将向导搜索与 Bounded Model Checking 技术的结合。然而遵照 No-Free-Lunch-Theorem (NFLT) for Optimization 的思想，对实际问题自身特征信息的利用是实现对该问题求解方法改善的关键[155]。因此，本小节中基于错误过程模式的验证向导方法与以上工作的不同点在于，它从传统工作流领域的现有经验出发，为经典的工作流模式（Workflow Pattern）[163]提出了一套相应的反模式，本文称之为“错误过程模式”（Process Bug Pattern），并同时研究了在网格服务流中加速寻找潜在错误过程模式的验证向导方法。本小节方法同样在本文的网格服务流状态π演算形式化验证原型系统 GridPiAnalyzer 中进行了实现，并在三组不同复杂度的 LIGO 引力波探测数据分析服务流中进行了应用和有效性验证。本小节中的研究工作可以归纳如下：

1) 基于经典的工作流模式提出了一套它的反模式 – 错误过程模式，用来表述网格服务流规范模型中可能出现的违例行为；
2) 基于 IEEE 标准规范的性质描述语言（PSL）[82]为本节中总结的错误过程模式定义了完整的形式化语义；
3) 基于错误过程模式，为网格服务流模型的证伪建立了带向导的业务逻辑验证方法，以提高对错误过程模式的验证效率。整个方法的有效性和优越性通过三组不同复杂度的实际 LIGO 引力波探测数据分析服务流进行了检验。

5.3.1　错误过程模式的提出

正如传统面向对象系统中的设计模式一样，经典的工作流模式识别并整理了在一般工作流设计与实现过程中常用的各类控制流关系。由于其在工作流领域的显著地位和广泛研究，因此对于网格服务流的设计与实现亦有着重要的参考价值。它根据控制流关系的复杂度与类型的不同，共分成了 5 大类模式：基本控制流模式、高级分支与同步模式、结构模式、多实例模式、基于状态的模式和取消模式。有关工作流模式的介绍可详见文献[110,163,164]，在此不再赘述。与之对应的，本小节中关注的则是在网格服务流模型中的相应控制流关系中（即各个服务为完成特性应用目的而实现的既定协作顺序与约束）所可能出现的违例情况。

因此，与工作流模式对工作流模型中期望行为的定义与总结不同，本小节中提出其反模式 – “错误过程模式”的目的在于反过来借鉴并总结网格服务流模型中可能存在的违例行为和结构。正如 Yang 和 Dill 曾指出的[160]，由于在实际的形式化验证应用中，试图寻找目标系统中潜在的错误较往往较证明其正确性更为实

用和有效，因此本节中对潜在违例行为和结构的研究（故命名为错误过程模式）对于复杂网格服务流的业务逻辑验证也是一项重要而有意义的工作。

为了对各类工作流模式及相应的错误过程模式做出精确的语义定义，图 5.17 首先回顾了 3.2.1 小节中的网格服务执行状态和调用活动关系。

每个服务在被调用时都先进行所需要数据的输入和等待（*StageIn* / *Pending*），服务运行（*Active*）在成功后进行数据的导出（*StageOut*）和清理（*Clean*），否则其运行则可能失败（*Failed*）。服务的执行最终将实现退出（*Exit*）。在此本文合理地假设网格服务流中每次服务调用实例都可以通过服务 ID 进行唯一的标识，用大写字母记为：*A, B, …*。每个网格服务的执行根据服务流规范中的各种结构关系进行交互：如顺序、同步/并发、分支、选择和同步联接等。每个结构控制了一个服务执行的状态变化如何相互影响并约束着其它服务的执行状态。关于这方面工作，第 3 章和第 4 章中已经完整给出了网格服务流基于状态π演算的形式化语义，并实现了该网格服务流对应状态标号迁移系统的转换生成。

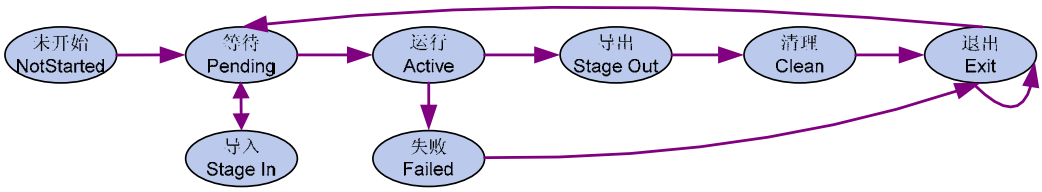


图 5.17 服务调用的状态抽象

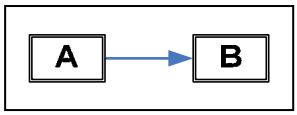
由此，以下小节中将分别根据各工作流模式提出相应的错误过程模式，并实现其带向导的快速验证方法。为了对它们有一个直观和精确的理解，本节利用了 IEEE 标准的性质描述语言 PSL[82]实现了其形式化语义。选择 PSL 的原因在于：

1) 它是一个国际标准并拥有广泛的业界支持（包括 NuSMV2 2.3.1[103]和 RuleBase[104]等主流模型验证引擎）；
2) 它具有直观的符号化表达方式，有利于其语义的理解；
3) 它的语义可以与 LTL 逻辑（以及 CTL 逻辑的大部分语义）完整兼容，从而可以直接纳入本文对网格服务流的状态π演算形式化方法中。

5.3.2 基本错误过程模式

5.3.2.1　顺序错误模式（*SequentialBug*）③

***SequentialBug*(*A*, *B*) =**
***SimultaneousStart*(*A*, *B*) ∨ *NoResponse*(*A*, *B*)**



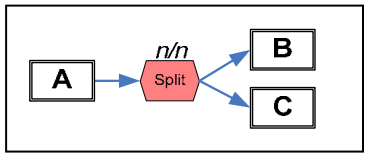
一个基本的顺序工作流模式表示了一服务 *B* 的执行将受限于另一服务 *A* 的执行完成。为了对这一简单关系进行证伪，需要寻找以下两种关系的存在：（1）服务 *B* 的执行在 *A* 尚未完成时就已经开始（*SimultaneousStart*）；（2）在 *A* 完成之后 *B* 从未被执行过（*NoResponse*）。以上两方面可以用两个原子错误过程模式分别进行描述，记为：*SimultaneousStart*(*A*, *B*)和 *NoResponse*(*A*, *B*)，它们的 PSL 语义定义④和相应的解释分别如下：

SimultaneousStart(A, B)={[*]; ¬*A.Exit* ∧ *B.Active*}
/* *经过服务流的多步执行后，其状态标号迁移系统可以到达这样的一个状态，其中服务 B 已经开始执行（Active）而服务 A 尚未执行完毕* */
NoResponse(A, B)= {[*]; *A.Clean; A.Exit*} ↦ {*B.Active*[=0]}
/* *当服务 A 在网格服务流中成功执行完毕后，永远都不会有服务 B 的执行* */

由此称网格服务流𝔽中可以找到一个顺序错误模式 *SequentialBug*(*A*, *B*)，若 $\exists\pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots\in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash$ *SimultaneousStart*(*A*, *B*) ∨ $\pi \vDash$ *NoResponse*(*A*, *B*)。其中 $Trans(\mathbb{F}, S_{init})$为𝔽对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。注意以上顺序错误模式的语义并不严格要求在服务 *B* 后不会执行服务 *A*，因为这种情况是可以接受的（如：当服务 *B* 通过 5.3.4.1 小节的 *Arbitrary Cycle* 与 *A* 的执行形成回环）。

5.3.2.2　并发错误模式（*ParallelSplitBug*）

***ParallelSplitBug*(*A*, {*B*, *C*})=**
***SequentialBug*(*A*, *B*) ∨ *SequentialBug*(*A*, *C*)**



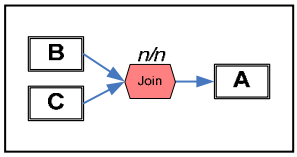
并发工作流模式中将流程的执行从一个服务（*A*）分支为后续多服务（*B*、*C*）的并发执行。由于并发工作流模式中并未规定并发服务调用间的执行顺序（即 *B*、*C* 可以以任意顺序执行，甚至是同时执行），因此并发工作流模式的反模式可以

③ 在以下对错误过程模式的提出过程中，分别使用左、右两框图显示了该模式的定义与可视化表示。下同。
④ 为统一本文的逻辑符号，本节中使用∧、∨、¬分别表示 PSL 中的与&、或|、非!关系。下同。

分别使用两个顺序错误模式的或来表达。并发错误模式描述了 *A*, *B* 和 *A*, *C* 间可能违反顺序关系的情况（即：在 *A* 执行完毕后是否能得到 *B*，*C* 执行的响应；*B*，*C* 是否会在 *A* 执行完毕之前就已经开始了执行），而并未对 *B*，*C* 间的执行顺序进行约束。这一点是和以下 5.3.6.2 小节中交错并发错误模式（*InterleavedParallelRoutingBug*）的区别所在，它根据相应的交错并发工作流模式语义而明确禁止了服务 *B*，*C* 间同时执行的情况。

5.3.2.3　同步错误模式（*SynchronizationBug*）

***SynchronizationBug*({*B, C*}, *A*) =**
***MultipleNoResponse*({*B, C*}, *A*) ∨**
***SimultaneousStart*(*B, A*) ∨ *SimultaneousStart*(*C, A*)**



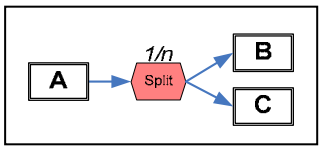
与并发工作流模式相反，同步工作流模式将所有并发服务（*B*、*C*）在执行完毕后同步到另一服务（*A*）的开始上，并继续整个流程的执行。因此，它不仅要求了当 *A* 开始执行时 *B*、*C* 必须已经执行完毕（即不存在以上的 *SimultaneousStart* 情况），且当 *B* 和 *C* 均完成执行后，*A* 最终会开始其执行。因此若将此处考虑的并发服务更一般化为 *B*, *C*, …, *Z* 的多服务组合，则为了对后一种情况做出刻画，以下相应对新的原子错误过程模式 *MultipleNoResponse* 进行了定义和解释：

MultipleNoResponse({*B, C, …, Z*}, *A*)={[*]; *B.Clean* ∨ *C.Clean* ∨…∨ *Z. Clean;*
B.Exit ∧ *C.Exit* ∧…∧ *Z. Exit*} ↦ {*A.Active*[=*0*]}
/* *若所有对应的并发服务 B, C, …, Z 能在网格服务流中成功执行完毕，则在此之后将不会有服务 A 的执行* */

由此，整个同步错误模式由相应 *MultipleNoResponse* 和 *SimultaneousStart* 错误过程模式的或组成。称网格服务流 $\mathbb{F}$ 中可以找到一同步错误模式 *SynchronizationBug*({*B, C*}, *A*)，若$\exists\pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots\in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash$ *MultipleNoResponse*({*B, C*}, *A*) ∨ $\pi \vDash$ *SimultaneousStart*(*B, A*) ∨ $\pi \vDash$ *SimultaneousStart*(*C, A*)。其中 *Trans*($\mathbb{F}$, S_{init})为$\mathbb{F}$对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。注意在以上同步错误模式中并不检验 *B*, *C*, …, *Z* 是否在流程中最终一定都会正常执行完毕，因为这是一种可以接受的情况（例如：当它们的执行根据以下 5.3.7 小节中的取消模式被强制取消时）。

5.3.2.4　单选错误模式（*ExclusiveChoiceBug*）

***ExclusiveChoiceBug*(*A*, {*B*, *C*}) =**
***OverExecute* (*A*, {*B*, *C*}) ∨**
(*NoResponse*(*A*, *B*) ∧ *NoResponse*(*A*, *C*)) ∨
***SimultaneousStart*(*A*, *B*) ∨ *SimultaneousStart* (*A*, *C*)**

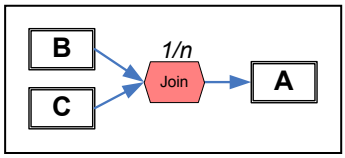


单选工作流模式中将流程的执行从一个服务（*A*）分支为后续多服务的调用（*B*、*C*），并只选取其中的一个服务继续流程的执行。与之相反，一个单选错误模式定义了这样的一种情况：（1）在 *A* 执行完毕后，*B* 和 *C* 都开始了其执行（*OverExecute*）；或（2）在 *A* 完成执行后存在 *B* 或 *C* 不会开始其执行的情况（*NoResponse*）；或（3）存在 *A*，*B* 或 *A*，*C* 会同时进行执行（*SimultaneousStart*）的情况。因此，单选错误模式由以上三种情况下对应错误模式的或组成。此时称一网格服务流𝔽中可以找到一单选错误模式 *ExclusiveChoiceBug*({*B*, *C*}, *A*)，若 $\exists\pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots\in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash$ *OverExecute*(*A*, {*B*, *C*}) ∨ $\pi \vDash$ (*NoResponse*(*A*, *B*) ∧ *NoResponse*(*A*, *C*)) ∨ $\pi \vDash$ *SimultaneousStart*(*A*, *B*) ∨ $\pi \vDash$ *SimultaneousStart*(*A*, *C*)。其中 $Trans(\mathbb{F}, S_{init})$为𝔽对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。“∧”则表示为了找到一个单选错误模式，*NoResponse*(*A*, *B*)和 *NoResponse*(*A*, *C*)错误都要得到满足。以上新的原子错误 *OverExecute* 的 PSL 语义定义及其相应解释如下：

OverExecute(*A*, {*B*,*C*})={[*];*A.Clean; A.Exit*} ↦ {{[*]; *B. Active*} ∧ {[*]; *C. Active*}}
/ 当 A 在流程中完成执行后，B 和 C 都会相应地开始进行执行 */*

5.3.2.5　简单合并错误模式（*SimpleMergeBug*）

***SimpleMergeBug*({*B*, *C*}, *A*)=**
***PrematureStart*({*B*, *C*}, *A*) ∨ *InclusiveExit*(*B*, *C*)**
∨ (*NoResponse*(*B*, *A*) ∧ *NoResponse*(*C*, *A*))



简单合并工作流模式将单选工作流模式对应的多分支服务（*B*、*C*）在执行完毕后汇合到另一服务（*A*）的开始，并继续整个流程的执行。为了对简单合并工作流模式的语义进行证伪，在简单合并错误模式中试图寻找流程中的这样一种执行路径，它满足：（1）当 *A* 正在执行时 *B* 和 *C* 都没有执行完毕；或（2）*B* 和 *C* 都执行完毕；或（3）*B*，*A* 和 *C*，*A* 都满足了 *NoResponse* 错误（由于可选路径间的单选关系）。由于第三种情况是直接的 *NoResponse* 描述，因此若将此处考虑的分

支服务更一般化为 *B, C, …, Z* 的多服务组合，则对前两种情况可以分别用以下的 *PrematureStart* 和 *InclusiveExit* 错误过程模式作出定义：

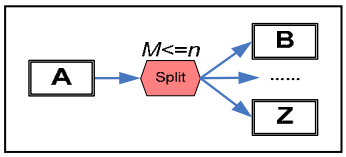
PrematureStart ({*B, C, …, Z*}, *A*)= {[*]; *A.Active* ∧ (¬*B.Exit* ∧ ¬*C.Exit* ∧ …∧ ¬*Z.Exit*)}
/* *当 A 在流程中开始执行时，其前置服务 B, C …和 Z 都没有执行完毕* */
InclusiveExit(*B, C, …, Z*)= {[*]; *B.Exit* ∧ *C.Exit* ∧…∧ *Z.Exit* }
/* *B, C…和 Z 在流程中均可以同时执行完毕* */

由此，称网格服务流𝔽中可以找到一简单合并错误模式 *SimpleMergeBug*({*B, C*}, *A*)，若∃$\pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots \in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash$ *PrematureStart*({*B, C*}, *A*) ∨ $\pi \vDash$ *InclusiveExit*(*B, C*) ∨ $\pi \vDash$ (*NoResponse*(*B, A*) ∧ *NoResponse*(*C, A*))。其中 $Trans(\mathbb{F}, S_{init})$为𝔽对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。

5.3.3　高级分支与同步错误模式

5.3.3.1　多选错误模式（*MultiChoiceBug*）

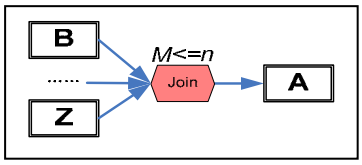
MultiChoiceBug(A, {B,…,Z})=
$\wedge_{Act\in\{B,\ldots,Z\}}$ ***NoResponse(A, Act)*** ∨
$\vee_{Act\in\{B,\ldots,Z\}}$ ***SimultaneousStart (A, Act)***



与单选工作流模式不同，多选工作流模式允许根据不同分支运行时条件的满足情况选择一个或多个分支上的服务进行执行(*m*-out-of-*n*，$m \le n$)。因此，为了确定服务流的执行与以上语义相违背的场合，必须对多个分支间所有可能的被选择情况进行考虑。由于所有分支上的服务 *B, C, …, Z* 都可能被执行甚至是同时执行，因此在多选错误模式中定义了：（1）在 *A* 执行完毕后，没有任何分支上的服务（*B, C, …, Z*）在流程中被选择执行：$\wedge_{Act\in\{B,\ldots,Z\}}$*NoResponse*(*A, Act*)，或（2）存在某一分支上的服务，使它可以和 *A* 同时开始执行：$\vee_{Act\in\{B,\ldots,Z\}}$ *SimultaneousStart*(*A, Act*)。

5.3.3.2　同步合并错误模式（*SynchronizingMergeBug*）

SynchronizingMergeBug({B,C,……,Z}, A)=
PrematureStart({B,C,……,Z}, A)

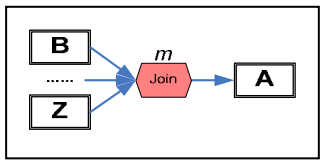


同步合并工作流模式的作用是将多选工作流模式形成的分支进行相应的合并。类似的，为了在一个服务流中寻找可能的同步合并错误，也需要对不同分支所有可能的被选择情况进行考虑（即：所有分支上的服务 *B, C, …, Z* 都可能被执行甚至是同时执行）。因此，同步合并错误中试图寻找以下的必要情况，即当 *A* 开始准备执行时，不同分支中没有任何服务已经完成了其执行，即：*PrematureStart*({*B, C,……, Z*}, *A*) 。

注意在同步合并错误中，并没有要求 *A* 必须根据其任意前继 $Act \in \{B,...,Z\}$ 的完成而做出开始执行的响应。这是由于在服务流设计时无法预知在多选工作流模式中哪几个分支是被选择执行并会最终得到了完成，因此无法相应的判别在同步合并工作流模式中 *A* 的开始执行究竟受到哪几个选择分支的约束。

5.3.3.3　多合并错误模式（*MultiMergeBug*）

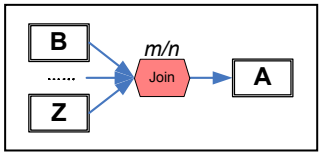
***MultiMergeBug*({*B,C,……,Z*}, *A*)=**
***SequentialBug*(*B, A*) ∨ …∨ *SequentialBug*(*Z, A*)**



与同步合并工作流模式类似，多合并工作流模式也用来对一个或多个选择分支的执行进行合并。然而它们的不同点在于同步合并工作流模式将会等待所有被选择分支执行完毕后才继续后续服务流的执行；反之，在多合并工作流模式中每一个被选择分支在其执行完毕后都可以独立地触发后续服务流（以 *A* 为初始）的执行。因此，为了对多合并语义进行证伪，多合并错误模式试图在网格服务流$\mathbb{F}$中寻找某服务 *Act*，使得 $Act \in \{B,...,Z\}$，且 *Act* 可以和 *A* 同时开始执行或在 *Act* 执行完毕后得不到 *A* 开始执行的响应（即：*SequentialBug*(*Act, A*)）。整个多合并错误模式的结果即为对各个分支 $Act \in \{B,...,Z\}$ 和服务 *A* 的顺序错误模式的或组合。

5.3.3.4　复杂合并错误模式（*ComplexJoinBug*）

***ComplexJoinBug*({*B,C,……,Z*}, *A, m, n*)=**
$\vee_{Acti \in \{B,...,Z\}}$ ***PrematureStart*({Act_1,…,Act_{n-m+1}}, *A*)** ∨
$\wedge_{Acti \in \{B,...,Z\}}$ ***MultipleNoResponse*({Act_1,…,Act_n}, *A*)**



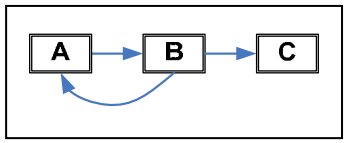
复杂合并工作流模式（亦称 *m*-out-of-*n* 模式，或其特例的鉴别器模式 *Discriminator*）负责合并 *n* 条并发路径中最先完成的 *m* 条路径上的服务，并忽略

剩余路径上服务的完成。因此，复杂合并错误试图断言以下两种情况的产生：（1）*A* 在足够数量 *m* 的分支服务执行完毕之前就已开始准备执行，即：$\vee_{Acti\in\{B,\ldots,Z\}}$ *PrematureStart*($\{Act_1,\ldots,Act_{n-m+1}\}$, *A*)；或（2）在 *m* 个分支的服务被执行完毕后没有 *A* 的开始执行作为响应，即：$\wedge_{Acti\in\{B,\ldots,Z\}}$*MultipleNoResponse*($\{Act_1,\ldots,Act_n\}$, *A*)。复杂合并错误即为以上两部分语义的或组合。注意在以上的复杂合并错误并不排除有超过 *m* 个分支上的服务被执行完毕的情况，因为复杂合并工作流模式中本身并不阻塞后 *n-m* 条并发路径的执行。

5.3.4　结构错误模式

5.3.4.1　任意循环错误模式（*ArbitraryCycleBug*）

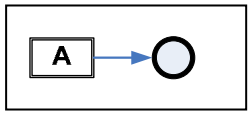
***ArbitraryCycleBug*(*A, B, C*)=**
***SequentialBug*(*A, B*) ∨ (*NoResponse*(*B, A*) ∧**
***NoResponse*(*B, C*)) ∨ *SimultaneousStart*(*B, C*)**



任意循环工作流模式简单地在一个服务（*B*）结束之行后循环回一个前置服务（*A*）进行执行，或继续对后续服务（*C*）进行执行。为了对该语义进行证伪，在任意循环错误模式中试图断言以下的情况：（1）*B* 完成执行之后，*C* 和 *A* 实际上均没有开始准备执行；或（2）*A*，*B* 或 *B*，*C* 间满足可能的顺序错误模式（*SequentialBug*）。注意在以上任意循环错误的定义中，使用了 *SimultaneousStart*(*B, C*)而非 *SequentialBug*(*B, C*)进行定义，这是因为此处需要同时判别 *NoResponse*(*B, A*) ∧ *NoResponse*(*B, C*)的成立。

5.3.4.2　隐含终止错误模式

无对应错误模式

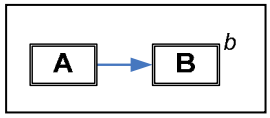


隐含终止是在本节错误过程模式中唯一没有找到对应反模式的一个工作流模式。这是因为隐含终止工作流模式的一个主要是为了帮助减少在流程中对终止节点描述的冗余，松弛在一个流程中只能有唯一终止节点的约束。它在本质上没有对服务流的建模增加表达能力上的贡献和实现上的约束。

5.3.5　多实例错误模式

5.3.5.1　无同步错误模式（*WithoutSynchronizationBug*）

***WithoutSynchronizationBug* (*b, A, B*)=**
***RedundantInstance*(*b, A, B*) ∨**
***SequentialBug*(*A, B*)　*b* 为一布尔条件**



无同步多实例工作流模式允许（在 *A* 执行完毕之后）实现服务（*B*）的多实例执行，且各实例的执行完成不需要同步。正如 Havey[164]所指出的，它的典型实现是一个带循环的并发分支。若记循环条件为 *b*，则无同步多实例模式不仅要求 *A*、*B* 间的顺序执行，同时 *B* 可以在条件 *b* 被满足时重复多次执行。因此，在无同步错误模式中使用了顺序错误 *SequentialBug* 对前一部分语义进行证伪，并定义了新的 *RedundantInstance* 错误过程模式来刻画后一部分语义的反例：

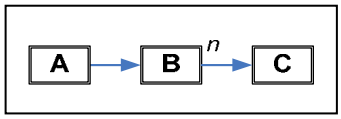
$RedundantInstance(b, A, B) = \{[*]; A.Exit \wedge b\} \mapsto \{[*]; \neg b \wedge B.Exit; \neg b \wedge B.Pending\}$

/* *当 A 执行完毕后且条件 b 不再满足时，B 仍然被再次重复执行* */

由此，称一个网格服务流 $\mathbb{F}$ 中可以找到一个无同步错误模式 $WithoutSynchronizationBug(b, A, B)$，若 $\exists \pi = \sigma_1, \sigma_2, \ldots \sigma_k, \sigma_k, \ldots \in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash RedundantInstance(b, A, B) \vee \pi \vDash SequentialBug(A, B)$。其中 $Trans(\mathbb{F}, S_{init})$为$\mathbb{F}$对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。

5.3.5.2　带设计时知识错误模式（*WithDesignTimeKnowledgeBug*）

***WithDesignTimeKnowledgeBug*(*A, B, n, C*)=**
***ParallelSplitBug*(*A*, {$B_1, B_2, \ldots, B_n$}) ∨**
***SynchronizationBug*({$B_1, B_2, \ldots, B_n$}, *C*)**



带设计时知识的多实例工作流模式（在某服务 *A* 执行完毕之后）实现了服务（*B*）的 *n* 个多实例执行，并将它们的完成进行同步，以继续流程的继续（以 *C* 为第一个服务）执行。如 Havey[164]所指出的，该模式可以简单地通过一个并发分支工作流模式和同步工作流模式的组合实现，其中两模式共享了所有的分支服务。因此，带设计时知识错误模式则被定义为相应并发错误模式 *ParallelSplitBug* 和同步错误模式 *SynchronizationBug* 的或。注意这里 *ParallelSplitBug*(*A*, {$B_1, B_2, \ldots, B_n$})

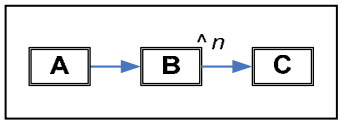
和 *SynchronizationBug*($\{B_1, B_2, \ldots, B_n\}$, *C*)的语义是直接基于 5.3.2.2 和 5.3.2.3 小节的分别扩展，其中 $B_1, B_2, \ldots, B_n$ 分别表示已知的 *n* 个服务 *B* 实例：

$$ParallelSplitBug(A, \{B_1, B_2, \ldots, B_n\}) = \vee_{Act \in \{B1, B2, \ldots, Bn\}} SequentialBug(A, Act)$$

$$SynchronizationBug(\{B_1, B_2, \ldots, B_n\}, C) = NoResponse(\{B_1, B_2, \ldots, B_n\}, C) \vee \vee_{Act \in \{B1, B2, \ldots, Bn\}} SimultaneousStart(Act, C)$$

5.3.5.3　带运行时知识错误模式（*WithRunTimeKnowledgeBug*）

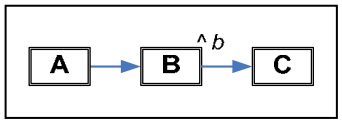
***WithRunTimeKnowledgeBug*(*A, B, n, C*)=**
***MultiChoiceBug*(*A*, {$B_1, B_2, \ldots, B_n$}) ∨**
***SynchronizingMergeBug*({$B_1, B_2, \ldots, B_n$}, *A*)**



带运行时知识的多实例工作流模式和之前带设计时知识的多实例工作流模式描述了类似的控制约束，但前者对于服务实例的实际数量是在运行时才确定的。因此，带运行时知识错误模式与带设计时知识错误模式的区别在于它利用了对应的多选错误（*MultiChoiceBug*）和同步合并错误（*SynchronizingMergeBug*）分别替换了并发错误和同步错误。该替换的目的是为了将任意可能数量的服务实例考虑进来，因为带运行时知识的多实例决定了实际的实例数量在设计时是无法预知的。需要注意以上的 *n* 表示了所允许的最大实例数，因为文中要求服务流所对应的状态π演算进程是有穷的。

5.3.5.4　不带运行时知识错误模式（*WithoutRunTimeKnowledgeBug*）

***WithoutRunTimeKnowledgeBug*(*A, B, b, n, C*)=**
***WithRunTimeKnowledgeBug*(*A, B, n, C*) ∨**
***RedundantInstance*(*b, A, B*)　　*b* 为一布尔条件

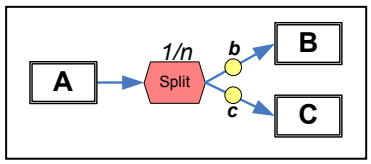


不带运行时知识的多实例工作流模式进一步一般化了 5.3.5.3 小节中的多实例语义，此时所需的服务实例数可以在实际服务流的执行过程中依据条件尽可能晚地被确定下来。它的实现与 5.3.5.3 小节类似，但此时确定服务实例数的要素需要依据一个循环条件 *b* 在运行时的动态判别结果[164]。因此，不带运行时知识错误模式不仅需要检查可能存在的带运行时知识错误模式，同时还需要额外地寻找 *RedundantInstance*(*b, A, B*)的反例情况（其定义参见 5.3.5.1 小节），即判断是否在 *A* 已经执行完毕且条件 *b* 已经不被满足的情况下 *B* 还可以再次被实例化执行。

5.3.6　基于状态的错误模式

5.3.6.1　延迟选择错误模式（*DeferredChoiceBug*）

DeferredChoiceBug(A, {b, B, c, C})=
OverExecute(A, {B, C}) ∨
(NoResponseOnEvent(A, b, B)
　∧ ***NoResponseOnEvent(A, c, C))*** ∨
SimultaneousStart(A, B)∨
SimultaneousStart(A, C)　***b, c*** 为布尔条件

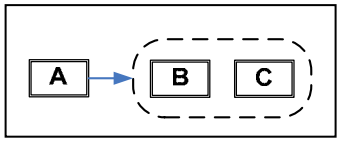


延迟选择工作流模式与单选工作流模式类似，但它在选择分支执行时并非马上确定待执行的分支服务，而是等待相应分支上事件的发生以触发该分支（记 *b*、*c* 分别为分支活动 *B* 和 *C* 所对应的事件）。因此，为了在网格服务流中寻找一个可能的延迟选择错误，需要进一步修改单选错误模式以对其语义进行定义。为了实现事件等待的选择语义，以下基于顺序错误模式中的 *NoResponse* 实现了相应 *NoResponseOnEvent* 错误的定义，从而给出了延迟选择错误的 PSL 语义。

NoResponseOnEvent(A, b, B)= {[*]; *A.Clean; A.Exit*} ↦
　　{[*]; *b; B. Active*[=*0*]}
/* *若 A 在服务流中执行完毕后，事件 b 被满足了但服务 B 却从未被执行* */

5.3.6.2　交错并发错误模式（*InterleavedParallelRoutingBug*）

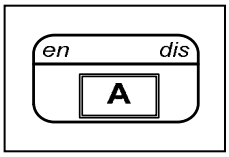
InterleavedParallelRoutingBug (A, {B, C})=
SimultaneousStart(B, C) ∧
SimultaneousStart(C, B) ∨
SequentialBug(A, B) ∨ ***SequentialBug(A, C)***



交错并发工作流模式的目的在于实现多服务（*B*、*C*）以任意次序（但不能同时）进行的顺序执行。自然的，在交错并发错误模式中，两个顺序错误（*SequentialBug*）错误用来确保是否 *B* 或 *C* 在 A 执行完毕后会最终进行执行，且它们的开始执行是否会和 *A* 同时进行。同时，*B* 和 *C* 间的 *SimultaneousStart* 错误则用来识别 *B* 和 *C* 是否可能违背“以任意次序，但不能同时执行”的“交错”约束。

5.3.6.3　里程碑错误模式（*MilestoneBug*）

MilestoneBug (en, dis, A)=
PrematureStart*(en, A) ∨
***RedundantInstance*(¬dis, en, A) en, dis* 为布尔条件**



里程碑工作流模式定义了某服务（*A*）在某使能条件 *en* 满足后可以开始被执行，但在某失效条件 *dis* 满足后则无法再执行。里程碑错误模式的定义因此非常直观，通过 *PrematureStart**错误，它判断了服务执行过程中可能违背使能条件的语义；通过 *RedundantInstance**错误，它判断了服务执行过程中可能违背失效条件的语义。称一网格服务流$\mathbb{F}$中可以找到一里程碑错误 *MilstoneBug*(*en, dis, A*)，若$\exists\pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots\in Trans(\mathbb{F}, S_{init})$，s.t. $\pi \vDash PrematureStart^*(en, A) \vee \pi \vDash RedundantInstance^*(\neg dis, en, A)$。其中 $Trans(\mathbb{F}, S_{init})$为$\mathbb{F}$对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。注意在此处的 *RedundantInstance**错误和 5.3.5.1 小节中定义的细微差异，因为在里程碑错误的语义中不仅仅要求 *A* 在失效条件满足后无法重新开始新的执行，更严格要求了在失效条件被满足时即使 *A* 正在执行过程中同样也是不能接受的：

$RedundantInstance^*(\neg dis, en, A)=\{[*]; en \wedge \neg dis\} \mapsto \{[*]; dis \wedge A.Active\}$；

另一方面，此处 $PrematureStart^*(en, A)$的定义与简单合并错误模式类似，为：

$PrematureStart^*(en, A)= \{[*]; A.Active \wedge (\neg en)\}$

5.3.7　取消错误模式

取消错误模式实际上在本文的网格服务流执行中并不考虑，因为正如 3.2.1 小节所定义的服务执行状态模型，它并不包含服务执行取消的情况。然而为了使本节方法更具有一般代表意义，因此本节仍然将取消模式考虑进来，使网格服务在等待或执行过程中可以被强制取消并退出（如图 5.18 所示）。

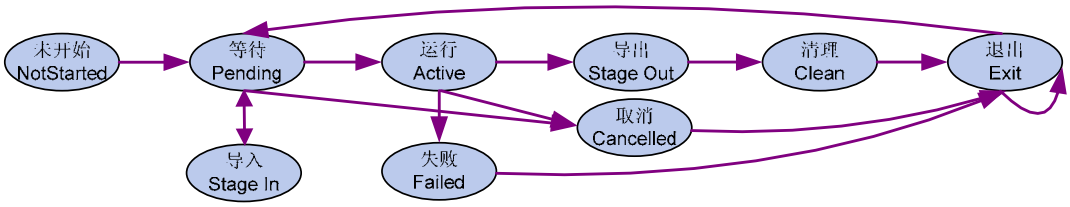
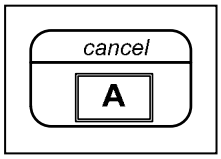


图 5.18　带取消的服务执行状态抽象

5.3.7.1　取消活动错误模式（*CancelActivityBug*）

CancelActivityBug(cancel, A)
***cancel* 为一布尔条件**



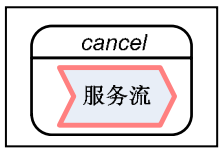
在取消活动工作流模式中，一服务（*A*）的执行可以通过特定的触发事件 *cancel* 进行取消。由于先前定义的错误过程模式中没有可以直接用来协助取消活动错误模式的定义，因此在这里将一个取消活动错误过程模式直接定义如下：

$$CancelActivityBug(cancel, A)=$$
$$\{[*]; cancel \wedge (A.Active \vee A.Pending)\} \mapsto \{A.Cancel[=0]\}$$

以上定义断言了当 *A* 在等待或正在执行过程中接受到相应取消事件 *cancel* 时，此后 *A* 的执行可能并没有被取消过。

5.3.7.2　取消过程错误模式（*CancelCaseBug*）

CancelCaseBug(cancel, Process)=
$\vee_{Act \in \mathbb{F}}$***CancelActivityBug(cancel, Act)***



取消过程工作流模式描述了根据特定取消触发事件 *cance*，对整个（子）服务流的执行进行取消的行为。直观的，在一网格服务流$\mathbb{F}$中可以找到一取消过程错误模式，若$\exists A \in \mathbb{F}$，s.t. $\exists \pi=\sigma_1, \sigma_2, \ldots\sigma_k, \sigma_k, \ldots\in Trans(\mathbb{F}, S_{init})$，且 $\pi \vDash CancelActivityBug(cancel, A)$。其中 $Trans(\mathbb{F}, S_{init})$为$\mathbb{F}$对应的状态标号迁移系统，其定义和实现已在 4.2 和 4.3 小节给出，而π则代表它的一个状态迁移路径。它表示：存在$\mathbb{F}$范围内的一服务，它正在等待或执行过程中，但在收到一相应取消触发事件 *cancel* 后它没有被成功取消（即在$\mathbb{F}$中可以找到满足取消活动错误模式的服务）。注意为了使以上取消过程错误的定义更一般化，它没有涵盖每个服务在取消后的具体行为（如它们何时通过何种方式是否会进行重新执行），因为这部分语义同样也没有包含在取消过程的工作流模式中。

5.3.8　带向导的快速错误过程模式验证

以上总结的错误过程模式有着两个特点：（1）为了确定网格服务流$\mathbb{F}$中是否能找到一特定错误过程模式，只需保证$\mathbb{F}$中存在某条执行路径能够满足相应模式的

语义既可（而不需要在所有执行路径上满足）；（2）所有错误过程模式中涉及的语义条件判别有着时序上的递增关系（即在判别过程中不需要在服务流的执行路径上进行回溯）。此外，错误过程模式中还为我们提供了可以用来断言在网格服务流中可以找到对应错误的目标状态命题（如：在 *SimultaneousStart* 错误中的 $\neg A.Exit \wedge B.\ Active$）。本节中称此类状态为“确认状态”（Commitment States）。出于以上特性的考虑，以下研究了针对错误过程模式的向导搜索（Guided Search）方法，从而在网格服务流验证过程中提高寻找其潜在违例行为的效率。

本节向导搜索的基本思想是在服务流状态π演算语义所对应的状态空间搜索过程中始终寻找并跟随最“有趣”的状态集合，从而更快探测到网格服务流中可能存在的错误过程模式。若给定确认状态 *CS*，则在给定状态集合 *SS* 中的有趣状态（Interesting States）被定义为在 *SS* 中能通过最少状态迁移次数到达 *CS* 的状态。更具体的，记：

- ***M***(𝔽)：表示一网格服务流𝔽的完整状态空间（即其状态π演算语义对应的状态标号迁移系统），其中在它的初始状态中所有服务均处于 *NotStarted* 状态；
- ***S***(𝔽)={***s***(***srv***$_1$),***s***(***srv***$_2$),…}：表示在 *M*(𝔽)中的一个服务流全局状态，其命题包含了该服务流中所有服务的当前执行状态；其中 $srv_i \in \mathbb{F}$ 且 $s(srv_i)$可以是图 5.17 或图 5.18 中所示的任意一种执行状态。

同一服务的两种不同执行状态间的距离 $D(s(srv)_1,\ s(srv)_2)$定义为在图 5.17 中从 $s(srv)_1$ 到 $s(srv)_2$ 的最少状态迁移次数。例如，*D*(*srv.Active, srv.Clean*)=*2*。该定义存在两个特例。一是当 $s(srv)_1$ 不存在能迁移到 $s(srv)_2$ 的路径时，两者距离定义为无穷。例如：*D*(*srv.Active, srv.NotStarted*)=∞；二是当考虑的目标状态命题是一个布尔条件时（如：*en*, *dis*, *cancel* 等），则其距离定为 1 或 0，分别表示该布尔条件满足和不满足时的情况。由此，网格服务流𝔽中两个服务流全局状态间的距离 *D_S* 则定义为其所有服务执行状态距离 *D* 的均值。图 5.19 中给出了它的一个例子。

$$\boldsymbol{D_S(S(\mathbb{F})_1,\ S(\mathbb{F})_2)} = \sum_i D(s(srv_i)_1, s(srv_i)_2) \Big/ |S(\mathbb{F})| \qquad srv_i \in \mathbb{F}$$

给定状态集合 *SS* 中的一确认状态 *CS*，则面向 *CS* 的有趣状态集合 *SS_CS* 为：

$$\boldsymbol{SS_CS} = \{S(\mathbb{F}) | S(\mathbb{F}) \in SS,\ \text{且} \forall S'(\mathbb{F}) \in SS,\ D_S(S(\mathbb{F}),\ CS) \le D_S(S'(\mathbb{F}),\ CS)\}$$

该定义表明了面向一确认状态 *CS* 的有趣状态总是那些可以通过最短路经的状态迁移到达 *CS* 的状态。进一步，对于状态集合 *SS* 中的确认状态 *CS* 还可以定

义相关的有趣程度。若称 $S(m)_CS_1 = S(m)_CS$ 的有趣程度为 1（记为 $Lv(S(\mathbb{F})_CS_1)=1$），则 $S(m)_CS_n$ 在 SS 中的有趣程度为等级 n，当 $S(m)_CS_n$ 在 $\{SS / (S(m)_CS_1 \cup ... \cup S(m)_CS_{n-1})\}$ 中的有趣程度为 1。

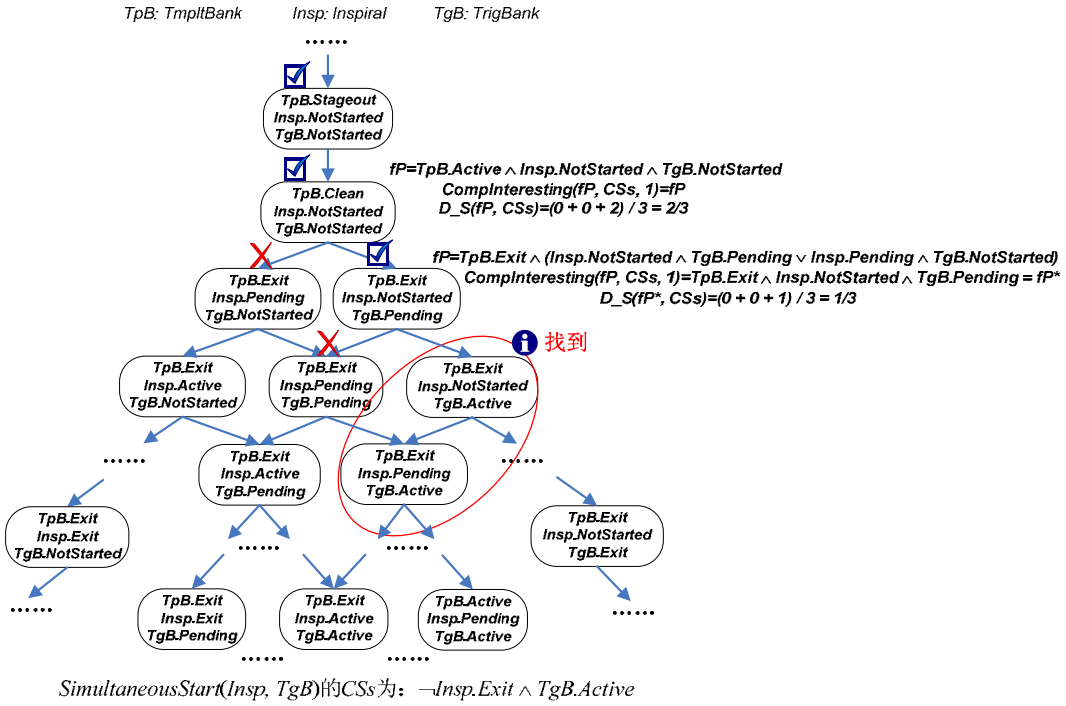


图 5.19 服务流的状态距离和搜索过程示例

在实际检测可能存在的错误过程模式中，往往需要考虑多确认状态的情况。例如，在 5.3.2.4 的 *OverExecute* 错误中，条件 *B.Active* 和 *C.Active* 需要在一个服务流状态上同时被满足才能确认该错误的存在。因此，以下针对多确认状态 *CSs* 的情况进一步定义了：

SS_CS' $= min\{SS_CS_1, SS_CS_2\}$，若：$CS' = CS_1 \vee CS_2$；

SS_CS' $= max\{SS_CS_1, SS_CS_2\}$，若：$CS' = CS_1 \wedge CS_2$。

在以上定义基础上，本节基于广度优先符号模型验证算法[84]进一步增加了在网格服务流中对错误过程模式的向导搜索机制。如图 5.20 所示，在新实现的 *GuidedBugHunting* (GBH)和 *GuidedBugHuntingBackTrace* (GBH-BT)算法中，存在两个不同的修改：

1) 实现了一个前向的状态搜索迭代过程，以对确认状态作出探测；
2) 额外的距离信息在状态搜索的迭代过程中被用来限定其中的有趣状态。

```
Init: 给定的初始状态;
CSs: 目标确认状态的逻辑描述;
MLv: 给定允许最大的有趣程度;
Gate: 存储非有趣状态的堆栈大小;
Procedure GuidedBugHunting (GBH)
1: Define P=Init=Reached, Lv=MLv
2: Set fP=Image(P)
3: Set fI=CompInteresting(fP, CSs, Lv)
4: Set Guided= fP ∧ fI
5: If Witness(Guided)
6:    提示找到了确认状态，返回Guided;
7: If fP = Guided
8:    /*若所有后续状态都是有趣的*/
9:    If Lv>1    Lv--, goto 3;
10:         /*降低考虑的最大有趣程度*/
11:   CSs=PreImage(CSs)
12: Set Union=Guided ∨ Reached
13: If Union=Reached
14:   提示没有找到对应对应错误模式;
15: P= Guided ∧ ¬Reached
16: Reached=Union, goto 2;
```

```
Procedure GuidedBugHuntingBackTrace (GBH-BT)
1: Define P=Init=Reached, Lv=MLv, k=0
2: Define stack = Empty /*保存非有趣状态的堆栈*/
3: Set fP=Image(P)
4: Set fI=ComInteresting(fP, CSs, Lv)
5: Set Guided= fP ∧ fI
6: If Witness(Guided)
7:    提示找到了确认状态，返回Guided;
8: If fP= Guided
9:    /*若所有后续状态都是有趣的*/
10:   If Lv>1  Lv--, goto 3;
10:         /*降低考虑的最大有趣程度*/
11:      CSs=PreImage(CSs)
12: Set UnInterested= fP ∧ ¬fI
13: If stack.size < Gate
14:   将UnInterested入栈stack
15: Set Union=Guided ∨ Reached
16: If Union=Reached
17:   If stack ≠ Empty
18:      P= stack.get(k++);
19:      Union=Reached=P ∨ Reached;  goto 3;
20:   Else 提示没有找到对应对应错误模式;
21: P= Guided ∧ ¬Reached
22: Reached=Union, goto 3;
```

图 5.20 GBH 和 GBH-BT 算法

GBH 和 GBH-BT 方法的关键步骤是 *CompInteresting*(*fP, CSs, Lv*)，它用来计算在预期的最大有趣程度 *Lv* 下，当前状态集合 *fP* 的后续状态集中，所有面向目标确认状态 *CS*s 的有趣状态集所对应的逻辑公式。因此，整个算法总是首先关注每次状态迭代过程中的有趣状态集，从而缩小网格服务流业务逻辑验证时在状态迭代过程中每次需要考虑的状态规模。以上 GBH 和 GBH-BT 的区别在于 GBH 在每次状态迭代过程中总是（并且仅是）考虑所有的有趣状态集。因此，它不是对网格服务流对应状态标号迁移系统的完整搜索，而只是在它的下近似（Under-Approximation）中试图寻找可能的错误过程模式。另一方面，GBH-BT 方法则允许先按有趣状态进行迭代搜索，并在没有找到目标错误模式时再回溯迭代非有趣状态以重新寻找可能的错误过程模式。此外，在两种方法中当在当前状态集的下一次迭代中无法找到任何有趣状态时（*fP=Guided*），则方法将自适应地通过降低预期的有趣程度 *Lv*，或通过计算当前确认状态集合的 *PreImage* 更新所需的目标确认状态集合来进一步限定有趣状态的判别条件。

易得此处 *CompInteresting* 操作的计算复杂度为 *O*(*m***n*)，*m* 是当前状态集 *fP*

的大小而 n 是目标确认状态集 *CSs* 的大小。其中，在整个网格服务流的状态迭代搜索过程中 n 是一个相对固定的常数；而 m 则可以通过每次有趣状态集合的限定而控制其实际大小。

5.3.9　应用与结果讨论

根据上一小节结果，GBH 和 GBH-BT 方法均在本文的网格服务流状态π演算形式化验证系统 GridPiAnalyzer（见第 6 章）中进行了实现。为验证以上方法的有效性并保持本文应用的连续性，仍以 5.2 小节中三组不同复杂度的 LIGO 引力波探测数据分析服务流 SF1~SF3 为性能比较的实例，它们已在 5.2.5 小节中给出。虽然在 4.4 和 5.2 小节中已证实了该服务流所满足的 8 条业务逻辑性质，本节在此基础上进一步关心在引力波探测数据分析中更细节的重要逻辑信息，包括：

- B1：对于满足偶然性分析必然性的服务流$\mathbb{F}$（即一旦数据进行期望波形匹配或模板库生成后，则最终一定会进行最后的偶然性分析处理，见 4.4 小节的详细描述），该结论是由于$\mathbb{F}$最终一定会进行最后的偶然性分析处理而成立的，还是由于服务流中可能根本不会进行任何波形匹配或模板库生成操作所形成的误判？
- B2：对于满足偶然性分析完整性的服务流$\mathbb{F}$，即所有偶然性分析操作定会执行，且 *sInca* 和 *thInca* 分析都将在 *thIncaII* 之前先进行，那所有可能存在的 *sInca* 和 *thInca* 之间在$\mathbb{F}$中又存在怎样的执行约束关系？
- B3：4.4 小节中的 8 条业务逻辑性质没有涵盖对干涉仪工作异常状态下的否决操作（*InspVeto*）语义。而根据 4.4 小节 LIGO 引力波探测数据分析的需求，实际的期望波形匹配方法对于干涉仪正常状态下采集的数据才能有较好的效果。因此偶然性分析 *sInca* 之后得到的信号还需要通过 L1 干涉仪中脉冲波形干扰，做出对不良信号的否决 *InspVeto*。这在$\mathbb{F}$中是否能得到满足？

由此，根据本章中的错误过程模式语义，图 5.21 中分别基于以上的进一步需求给出了对应于服务流实例 SF1~SF3 的待验证错误模式。在图 5.21 的可视化描述中，“*”表示对同类型服务的任一选择。表 5.1~表 5.3 分别给出了经典的符号模型验证算法（记为 SMCA）、带影响锥的符号模型验证算法（记为 SMCA+COI）以及本节的 GBH 和 GBH-BT 方法（*Mlv*=1，*Gate* 分别设为 0、1、3）在验证性能上的比较。以上选择 SMCA（+COI）作为比较对象是由于在已有的详细验证结果中（见 5.2.5 小节图 5.11~图 5.13 和附录 A），这两种方法在本应用案例中有着较

好的验证性能。本文的验证硬件环境仍然为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。

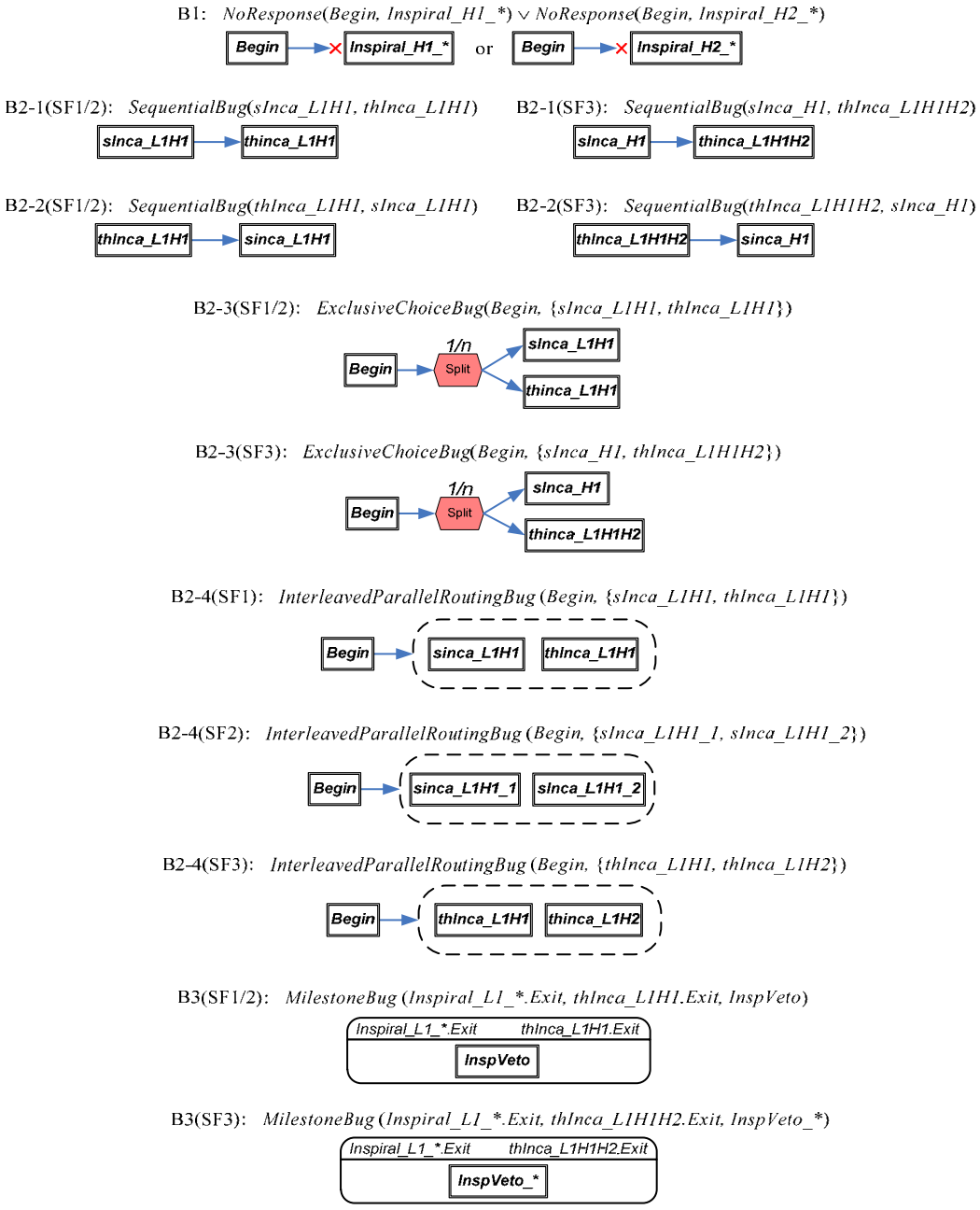


图 5.21　待验证错误行为模式及其可视化描述

正如 5.2.5 小节所示，以上选取的网格服务流实例 SF1-3 有着不同的复杂度（参见附录 A 中它们的详细规模）。表 5.1 至表 5.3 中 B1~B3 的六个错误行为模式分别对应于以上本节中所描述的需求和图 5.21 中的可视化描述，“true”结果表示对应错误过程模式能在网格服务流中被找出，“false”结果表示对应的错误行为模式不存在于目标网格服务流中。对于 SF1，GBH 方法和 GBH-BT 方法相较 SMCA 和 SMCA+COI 方法并没有大的性能改进，而除了对 B2-2(SF1/2)的验证外，它们的性能更是落后于这两种方法。这是由于该网格服务流规模过小，从而使 GBH 和 GBH-BT 方法中对有趣状态的额外计算代价变得相对昂贵。然而，类似小规模网格服务流显然不是本节进行网格服务流形式化验证性能改进的关注点。本节方法的优势在更复杂的 SF2 和 SF3 中被体现出来。从验证性能可以看出，虽然 SMCA 和 SMCA+COI 对所有错误过程模式的验证性能相当，但 GBH 和 GBH-BT 方法总能比它们在更快的时间内找出对应错误。因为 GBH 和 GBH-BT 方法不仅能始终关注于最可能检测到错误的状态上，同时剔除了对以非有趣状态为起始的执行分支的检测，从而降低了验证的规模。此外，GBH 方法对于所有应用实例的性能都比 GBH-BT 方法好，因为 GBH 方法仅关注于有趣状态集并在每次状态迭代过程中不会浪费时间对非有趣状态进行检验。而 GBH-BT 方法则会在堆栈大小允许的情况下进一步作出部分的状态回溯，以进行非有趣状态的计算和栈操作。然而，GBH 方法的缺点在于它是针对目标错误过程模式，在待验证网格服务流状态标号迁移系统的下近似（Under Approximation）上进行的判别。因此它是用来基于目标错误对给定网格服务流进行证伪的方法，即当其返回 false 结果时能够断言该网格服务流中必能找到目标错误过程模式 B，但当其返回 true 时不代表该网格服务流中一定不存在 B。实际上，从表 5.1~表 5.3 中可以看出，GBH 方法是 GBH-BT 方法在其栈（*stack*）大小为 0 时的特例。根据以上讨论本节可以得到如下结论：

（1）　GBH 和 GBH-BT 方法通过指导有趣状态的迭代，可以有效提高在复杂网格服务流中进行错误过程模式的检验效率；

（2）　特别当网格服务流规模非常庞大而使得经典的模型检测算法无法正常验证完毕时，GBH 方法可以通过对其下近似的检验而得到是否存在潜在错误的重要信息；

（3）　整个网格服务流状态空间的搜索完整性和效率间的平衡可以根据实际待检测网格服务流的规模，通过对 GBH 方法中的栈大小来进行调节。

表 5.1 SF1 的验证时间性能比较（单位：ms）

GBH / -BT (0)		**GBH-BT (1)**		**GBH-BT (3)**		**SMCA**		**SMCA+COI**	
结果	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**
B1: *NoResponse*(*Begin, Inspiral_H1_**) ∨ *NoResponse*(*Begin, Inspiral_H2_**)									
false	3126	false	3320	false	3360	false	2514	false	2467
B2-1(SF1/2): *SequentialBug*(*sInca_L1H1, thInca_L1H1*)									
false	3609	false	3750	false	3802	false	2288	false	2318
B2-2(SF1/2): *SequentialBug*(*thInca_L1H1, sInca_L1H1*)									
true	1500	true	1516	true	1547	true	2065	true	2034
B2-3(SF1/2): *ExclusiveChoiceBug*(*Begin,* {*sInca_L1H1, thInca_L1H1*})									
true	3922	true	4667	true	4813	true	2678	true	2569
B2-4(SF1): *InterleavedParallelRoutingBug* (*Begin,* {*sInca_L1H1, thInca_L1H1*})									
false	3203	false	3703	false	3813	false	2345	false	2412
B3(SF1/2): *MilestoneBug* (*Inspiral_L1_*.Exit, thInca_L1H1.Exit, InspVeto*)									
true	3453	true	3844	true	3875	true	2237	true	2347

表 5.2 SF2 的验证时间性能比较（单位：ms）

GBH / -BT (0)		**GBH-BT (1)**		**GBH-BT (3)**		**SMCA**		**SMCA+COI**	
结果	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**
B1: *NoResponse*(*Begin, Inspiral_H1_**) ∨ *NoResponse*(*Begin, Inspiral_H2_**)									
false	112327	false	113594	false	126140	false	259387	false	255375
B2-1(SF1/2): *SequentialBug*(*sInca_L1H1, thInca_L1H1*)									
false	67672	false	92437	false	97797	false	252547	false	249687
B2-2(SF1/2): *SequentialBug*(*thInca_L1H1, sInca_L1H1*)									
true	22938	true	29375	true	30062	true	256844	true	250609
B2-3(SF1/2): *ExclusiveChoiceBug*(*Begin,* {*sInca_L1H1, thInca_L1H1*})									
true	115438	true	172047	true	175484	true	247906	true	250641

表 5.2（续） SF2 的验证时间性能比较（单位：ms）

B2-4(SF2): *InterleavedParallelRoutingBug* (*Begin, {sInca_L1H1_1, sInca_L1H1_2}*)									
true	66500	true	89531	true	95031	true	246380	true	251265
B3(SF1/2): *MilestoneBug* (*Inspiral_L1_*.Exit, thInca_L1H1.Exit, InspVeto*)									
true	89562	true	101078	true	102328	true	248266	true	248063

表 5.3 SF3 的验证时间性能比较（单位：ms）

GBH / -BT (0)		**GBH-BT (1)**		**GBH-BT (3)**		**SMCA**		**SMCA+COI**	
结果	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**	**结果**	**时间**
B1: *NoResponse*(*Begin, Inspiral_H1_**) ∨ *NoResponse*(*Begin, Inspiral_H2_**)									
false	196172	false	198032	false	201312	false	491328	false	487201
B2-1(SF3): *SequentialBug*(*sInca_H1, thInca_L1H1H2*)									
false	273542	false	280938	false	283527	false	499903	false	496109
B2-2(SF3): *SequentialBug*(*thInca_L1H1H2, sInca_H1*)									
true	236328	true	237594	true	238109	true	492390	true	486739
B2-3(SF3): *ExclusiveChoiceBug*(*Begin, {sInca_H1, thInca_L1H1H2}*)									
true	104282	true	222593	true	229032	true	491306	true	495281
B2-4(SF3): *InterleavedParallelRoutingBug* (*Begin, {thInca_L1H1, thInca_L1H2}*)									
true	236188	true	237438	true	238719	true	495875	true	497322
B3(SF3): *MilestoneBug* (*Inspiral_L1_*.Exit, thInca_L1H1H2.Exit, InspVeto_**)									
false	268532	false	272250	false	274187	false	495654	false	503276

5.4　小结

本章分别从网格服务流验证分解和错误过程模式的两个思路，对上一章中网格服务流的状态π演算形式化验证方法进行性能上的改进，从而使其能够适用于更复杂的服务流验证中。其中，基于松弛域分析的方法将网格服务流分解为一系列顺序的串行标准域及其并行待剪枝。由此，根据模块化验证的原理以逆向顺序组合的策略，实现了根据局部已知的标准域验证结果推理整个网格服务流全局验证结果的验证分解方法，并给出了其分析过程和方法实现。另一方面，在基于错误

过程模式的搜索向导方法中则根据经典工作流模式提出了其对应的反模式，用以描述网格服务流中可能存在的常见违例结构和行为，并基于 IEEE 标准的性质描述语言 PSL 给出了它们的形式化语义。此外，基于所总结的错误过程模式特点，本章进一步实现了基于有趣状态（Interesting States）的验证向导方法，以加速网格服务流中错误模式的检验。根据 LIGO 数据网格中引力波探测数据分析的 3 组不同复杂度的实际服务流应用结果，以上两种方法相对直接使用各类符号模型验证方法和 Bounded Model Checking 方法在网格服务流的验证效率上均有较好的改进。此外，由于松弛域分析的验证分解方法允许了对网格服务流的局部递增分析，因而在验证内存占用上也有明显改善。

第 6 章　网格服务流验证系统(GridPiAnalyzer)及其实现

6.1　本章引论

基于以上的各章工作，本章将它们在本文的原型系统 GridPiAnalyzer 中进行了统一的集成与实现，为网格服务流基于状态π演算的形式化验证提供了自动化的工具支持。GridPiAnalyzer 的特点在于其自动化和直观性，即：

- **自动化**：网格服务流的各验证环节，包括具体网格服务流建模结果的状态π演算语义形式化、状态标号迁移系统的转换生成和状态断言过程、待验证业务逻辑的一致性检验、以及业务逻辑验证和反例生成过程，都可以通过集成本文各章工作结果而自动进行，从而减少用户需要的手工干预；
- **直观性**：GridPiAnalyzer 通过验证过程的自动化，充分将复杂的状态π演算语义和形式化验证技术细节对用户进行了屏蔽，从而降低了用户在使用 GridPiAnalyzer 时的难度和对专业知识的需求。此外，GridPiAnalyzer 中还设计了可视化的业务逻辑描述方式，我们称之为业务性质描述语言（BPSL）[43]，从而方便了用户对复杂时序逻辑性质的建模和理解。

本章中 6.2~6.5 小节将首先对 GridPiAnalyzer 的整体设计框架及其实现模块进行详细介绍，在 6.6 小节中则介绍了 GridPiAnalyzer 现有在 LIGO 数据网格和银行法律法规验证中的两个应用及其背景。

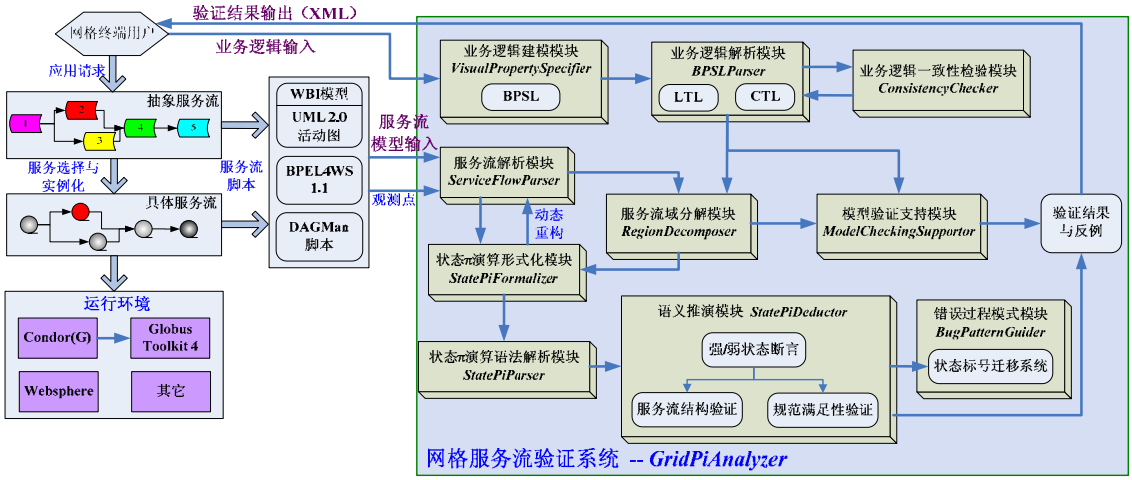


图 6.1 GridPiAnalyzer 的系统框架及其模块

6.2　GridPiAnalyzer的系统框架与模块

GridPiAnalyzer 是基于本文研究结果，在 Eclipse 平台（*http://www.eclipse.org*）上实现的一个基于状态π演算的网格服务流形式化验证原型系统。选择 Eclipse 平台的原因在于其开放性。作为一个开放插件平台，Eclipse 允许不同的研发人员以插件的形式开发、部署、并相互共享和复用各自的不同功能插件。而目前在 SOA 和网格领域中，在 Eclipse 上也已经有着相应的开源和商用插件，包括：Web 服务开发与部署插件（如 Web Standard Tools – WST）、基于 WSRF 的网格服务开发与部署插件（如 Globus Service Building Tools – GT4IDE）、不同的 BPEL4WS 建模插件（如 ActiveBPEL™ Designer 和 Eclipse BPEL Project）和服务流执行引擎插件（如 ActiveBPEL™ Engine）等。GridPiAnalyzer 自身也作为 Eclipse（3.0.2 版本）的一个插件实现。图 6.1 中给出了其整体系统框架。它共由 9 个主要模块组成：服务流解析模块、状态π演算形式化模块；状态π演算语法解析模块；语义推演与错误过程模式模块；模型验证支持模块；业务逻辑建模模块；业务逻辑解析模块；业务逻辑一致性检验模块和服务流域分解模块。

- **服务流解析模块：**服务流解析模块负责接收具体网格服务流规范的模型输入，并在根据其语法进行解析后转换为状态π演算形式化模块所能识别的元数据结构。服务流解析模块是整个 GridPiAnalyzer 在实际网格服务流规范模型与其运行环境之间的接口模块，为具体服务流的状态π演算形式化提供了基础。目前 GridPiAnalyzer 共支持三种不同运行环境下的服务流规范模型：
  - Condor DAGMan 脚本 → Condor(-G)运行环境 + Globus Toolkit；
  - BPEL4WS 1.1 规范脚本[①] → ActiveBPEL™ Engine（针对 WSRF 的）扩展运行环境；
  - Websphere Business Integrator ™（WBI）服务流模型[②]（与 UML2.0 活动图部分兼容，它能自动转换至 BPEL4WS 1.1 规范脚本） → Websphere Integration Developer 运行环境；
- **状态π演算形式化模块：**状态π演算形式化模块负责对经过解析的输入网格服务流模型进行相应状态π演算形式化语义的自动生成。生成的依据即为第 3 章中不同网格服务流规范和模式的状态π演算形式化语义结果。

① 目前版本的 GridPiAnalyzer 中支持由 ActiveBPEL™ Designer (2.0 版)所建模生成的 BPEL4WS 脚本。
② IBM WebSphere Business Integration Modeler ™; 见: *http://www-306.ibm.com/software/integration/wbimodeler/library/*

- **状态π演算语法解析模块：**GridPiAnalyzer 中除了允许直接按支持的格式对输入网格服务流自动生成其形式化语义并进行验证，也支持在遵循状态π演算语法的条件下接收手写的状态π演算模型脚本。因此状态π演算语法解析模块在这里的作用有两方面：（1）对输入状态π演算脚本进行语法检查；（2）通过解析输入的状态π演算脚本，在内存中形成相应的状态π演算元模型实例（参见 6.3 小节），供下面的语义推演模块使用。
- **语义推演与错误行为模式模块：**语义推演模块是 GridPiAnalyzer 中的一个关键模块。它的功能包括三部分：（1）根据 4.2 小节结论，为网格服务流的状态π演算语义实现自动的状态标号迁移系统转换；（2）实现 4.2 小节中的强/弱状态断言判断，在根据扩展操作语义进行推演的同时完成对服务流结构和规范语义约束的验证；（3）若在 GridPiAnalyzer 的参数设置中激活了错误过程模式的向导搜索选项，则错误行为模式模块进一步根据 5.3 小节方法，将服务状态间的距离信息予以考虑，对给定错误过程模式进行检验。
- **模型验证支持模块：**作为 GridPiAnalyzer 的后端，模型验证支持模块负责将网格服务流的状态π演算语义解释到具体的模型验证引擎上，从而实现业务逻辑的自动验证。目前 GridPiAnalyzer 已经实现了对 NuSMV2[103]这一主流开源引擎的支持。此外，该模块的另一功能在于对模型验证结果进行封装。每次针对不同网格服务流的不同验证操作，它都会形成单独的 XML 文件封装包括该服务流模型的规模、待验证的业务逻辑、验证的性能和结果、以及存在的反例信息（见 6.4 小节）。
- **业务逻辑建模模块：**除了直接手写待验证业务逻辑的时序逻辑公式外，为了克服时序逻辑语言自身的复杂度，降低其建模难度和符号化表达的晦涩，在 GridPiAnalyzer 中还设计并实现了可视化的时序逻辑的建模环境，我们称之为可视化业务性质规范语言（BPSL）。BPSL 的语义同时兼容了 LTL 和 CTL，其介绍亦可参见 6.5 小节③。
- **业务逻辑解析模块：**负责将以上 BPSL 建模器中对业务逻辑的可视化建模结果自动解释为 LTL 或 CTL 时序逻辑性质。
- **业务逻辑一致性检验模块：**若在 GridPiAnalyzer 中激活了业务逻辑一致性验证选项，则在选定一组待验证业务逻辑性质后，业务逻辑一致性检验模块会在实际进行形式化验证操作前根据 4.6 小节中的工作检验这组业务逻辑性质

③ 在目前版本的 GridPiAnalyzer 中对错误过程模式的描述仍然只提供纯文本的建模方式。

中存在的冲突和冗余性质，并给出高亮标识。

- **服务流域分解模块：**服务流域分解模块包含两个子模块：服务流分解模块和业务逻辑推理分解模块。通过该模块使 GridPiAnalyzer 可以在整个网格服务流的业务逻辑验证过程中依据 5.2 小节的工作进行域分解操作，并基于模型验证支持模块的功能在不同的域上进行待验证业务逻辑的验证推理。

图 6.2 中给出了 GridPiAnalyzer 的标准运行界面示例。

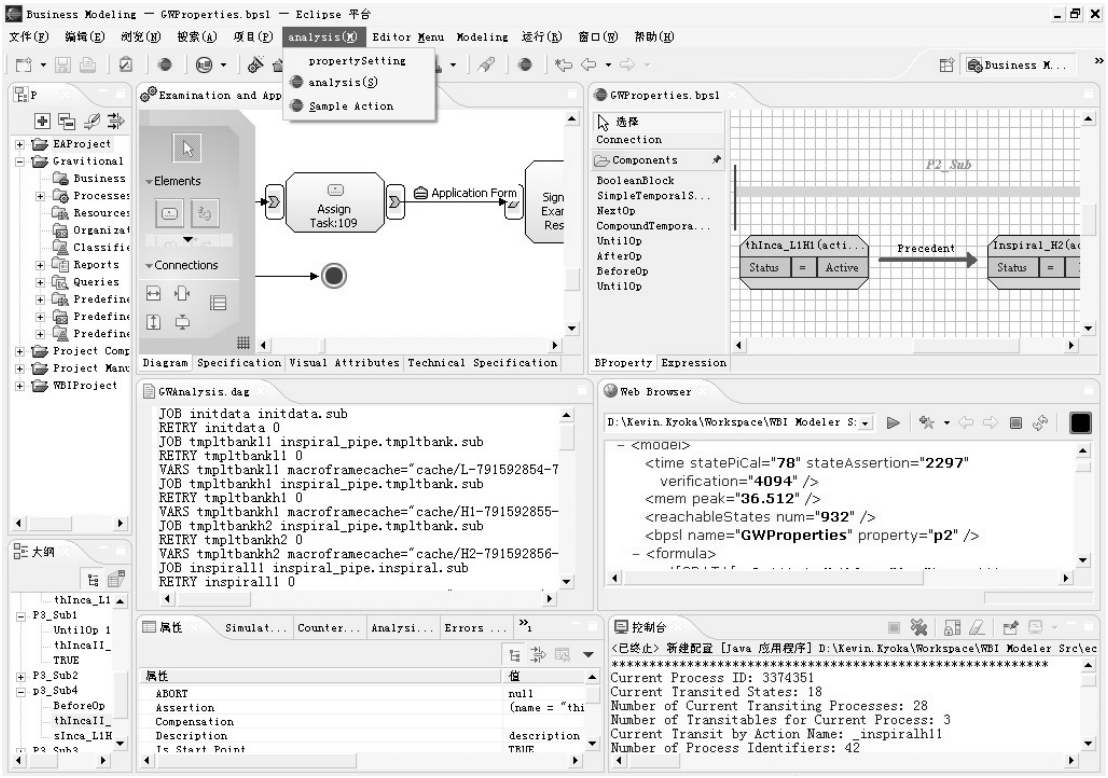


图 6.2　GridPiAnalyzer 的运行界面示例

6.3　状态π演算模型在GridPiAnalyzer中的实现

根据状态π演算的语法（见 2.2 小节），在 GridPiAnalyzer 中通过 JavaCC[④]实现了其语法检查和编译模块，即上一小节中的状态π演算语法解析模块。对应的，当该语法解析模块完成对输入状态π演算脚本进行编译后，GridPiAnalyzer 将按图 6.3 中的状态π演算元模型在内存中缓存网格服务流的对应状态π演算语义。

④ Java Compiler Compiler; 见 *https://javacc.dev.java.net/*

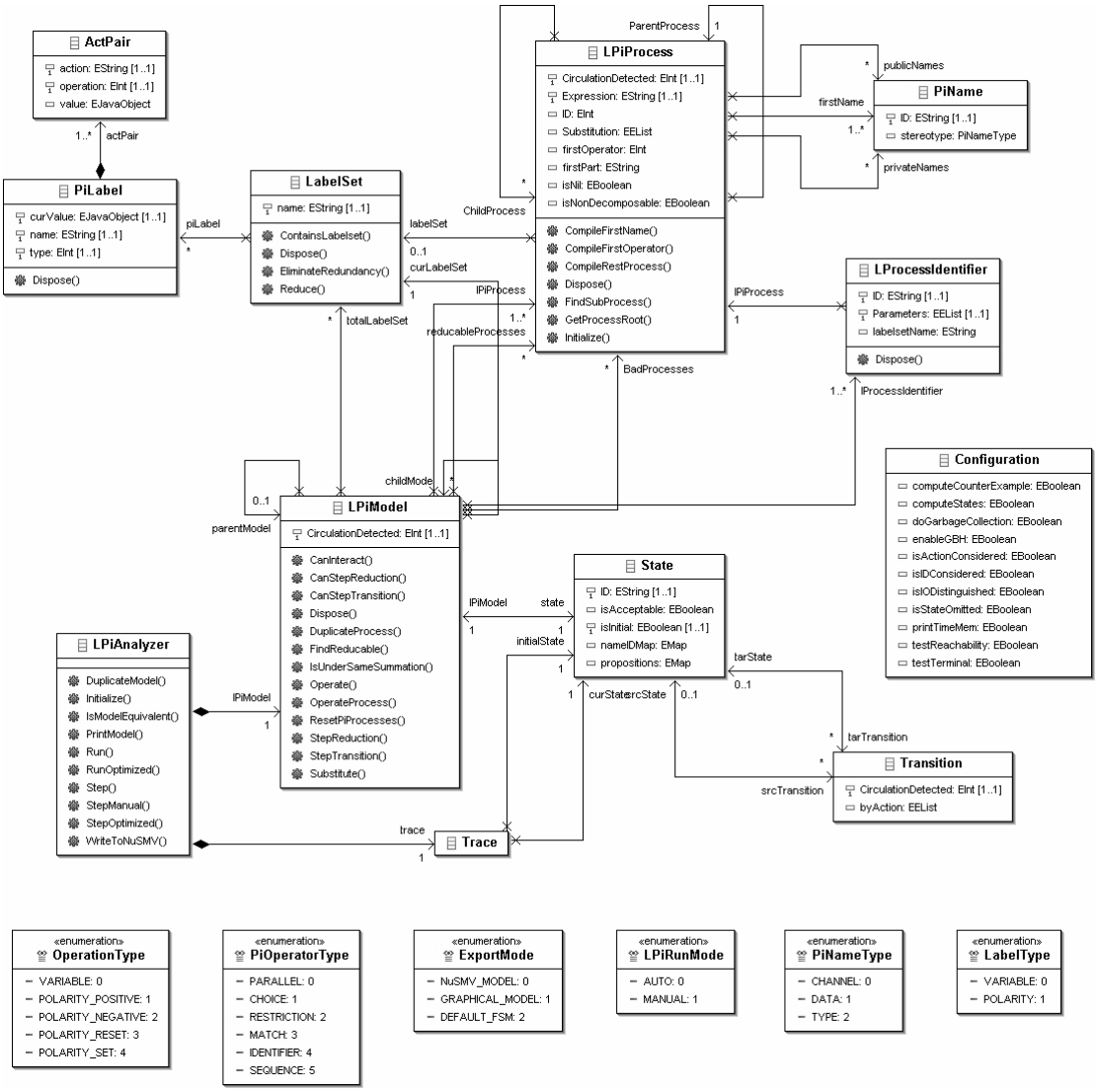


图 6.3 状态π演算语法实现的元模型

需要注意，为了避免在使用常用词“*State*”和“*Proposition*”时对其它系统可能带来的意义混淆，在以上的元模型实现中使用了保留字“*LabelSet*”和“*Label*”来分别代表状态π演算中对状态和命题声明与描述。

在网格服务流状态π演算模型的语法编译完成之后，则可以根据 GridPiAnalyzer 自身的不同工作参数设置（见图 6.4），在第 4 章工作的基础上进行具体的状态标号迁移系统转换、状态断言和形式化验证工作。

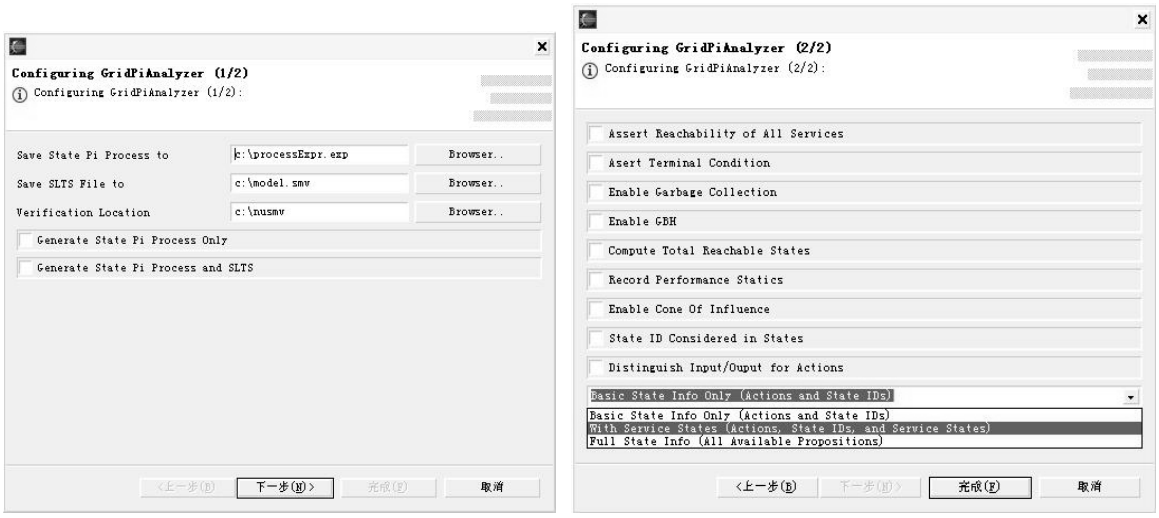


图 6.4 GridPiAnalyzer 的工作参数设置

6.4　状态标号迁移系统及验证结果的封装

在现有 GridPiAnalyzer 中基于 NuSMV2[103]这一主流开源模型验证引擎的输入语法对状态标号迁移系统进行了表示和存储。文中网格服务流形式化语义所对应的状态标号迁移系统采用了 NuSMV2 语法中的一个主模块（MODULE main）进行表示。更具体的，在 GridPiAnalyzer 中利用了该主模块的以下 3 个主要语法部分和一个可选的业务逻辑描述部分：

- 变量声明：包含了在输入状态标号迁移系统中存在的变量及它们的取值范围声明。这些变量的来源是网格服务流的状态π演算语义中已定义的状态命题。在 GridPiAnalyzer 的设置中默认能够直接支持的命题包括：
  - name / action：负责记录当前状态迁移通过哪个状态π演算动作进行；
  - paralist：负责记录相关状态π演算动作所关联的输入/输出，当存在多元输入/输出时使用“_”进行分隔；
  - type：负责记录相关状态π演算动作的类型，即：输入、输出、或不可见；
  - stateID：根据状态标号迁移系统转换过程中每个进程及其关联状态的不同为其设定的唯一标识；
  - serviceStatus：分别负责记录各服务的当前执行状态；这里的“service”将由具体服务的状态π演算进程名所替代，以表示是哪个服务的当前状态；

- ExecutingSrv：负责记录服务流中当前正在执行的服务，当存在多个时使用“_”进行分隔；
- varSize：分别负责记录各相应变量或堆栈的当前大小。这里的“var”将由具体变量或堆栈的状态π演算进程名所替代，以表示是哪个变量或堆栈的当前大小；
- recentBranch：负责记录服务流语义推演过程中最近选择的分支动作；
- recentException：负责记录服务流语义推演过程中最近曾出现的异常；

- 初始化声明：确定在输入状态标号迁移系统中每个变量的初始值，其声明格式为：*init*(*var*) *:=* *val;*。其中 *var* 表示变量声明中所定义的变量名；*val* 表示在变量声明中属于 *var* 有限取值范围内的初始取值；
- 迁移声明：负责描述输入状态标号迁移系统中的状态迁移关系。与状态标号迁移系统中的状态迁移相对应，若当前通过动作α已到达状态 S，则对能从 S 出发的各个迁移 $\xrightarrow{\alpha_1}(P_1,S_1)$，$\xrightarrow{\alpha_2}(P_2,S_2)$，……，$\xrightarrow{\alpha_n}(P_n,S_n)$，在迁移声明中存在对应的：

$$\begin{aligned} var = val\ (\&\ var = val)^*\ : \quad & next(var) = val_1\ (\&\ next(var) = val_1)^*\ | \\ & next(var) = val_2\ (\&\ next(var) = val_2)^*\ | \\ & \ldots\ldots \\ & next(var) = val_n\ (\&\ next(var) = val_n)^*; \end{aligned}$$

以上所有的名值对(*var, val*)、(*var, val*$_1$).....和(*var, val*$_n$)分别属于 S、S_1...和 S_n。“*&*”和“|”分别是 NuSMV2 输入语法中变量值判别的与、或操作；“()*”则表示对括号内语法块的 0 次或多次重复。

而对于形式化验证的结果输出，GridPiAnalyzer 的模型验证支持模块对不同网格服务流的不同验证结果用单独的 XML 文件进行了封装，它包含了待验证服务流模型的规模、待验证的业务逻辑、验证的性能和结果、以及存在的反例等信息。图 6.5 中给出了一个验证结果的 XML 示例。需要指出，在验证结果的封装中对所生成反例的状态序列进行了进一步的过滤。为了增加反例序列的可读性，在封装的 XML 反例信息中将用户不关心的底层状态信息进行了剔除。例如：在每个服务活动的状态π演算进程中，其 *start*、*done* 等动作（见 3.2.3 小节结果）只是在形式化模型层次为了刻画不同服务执行的约束关系而建立的，它们并不反映用户所关系的实际业务和服务信息。因此在相关的反例序列中，对于此类动作的相关命题信息将被删除，从而增加整个反例的可读性和可理解性。

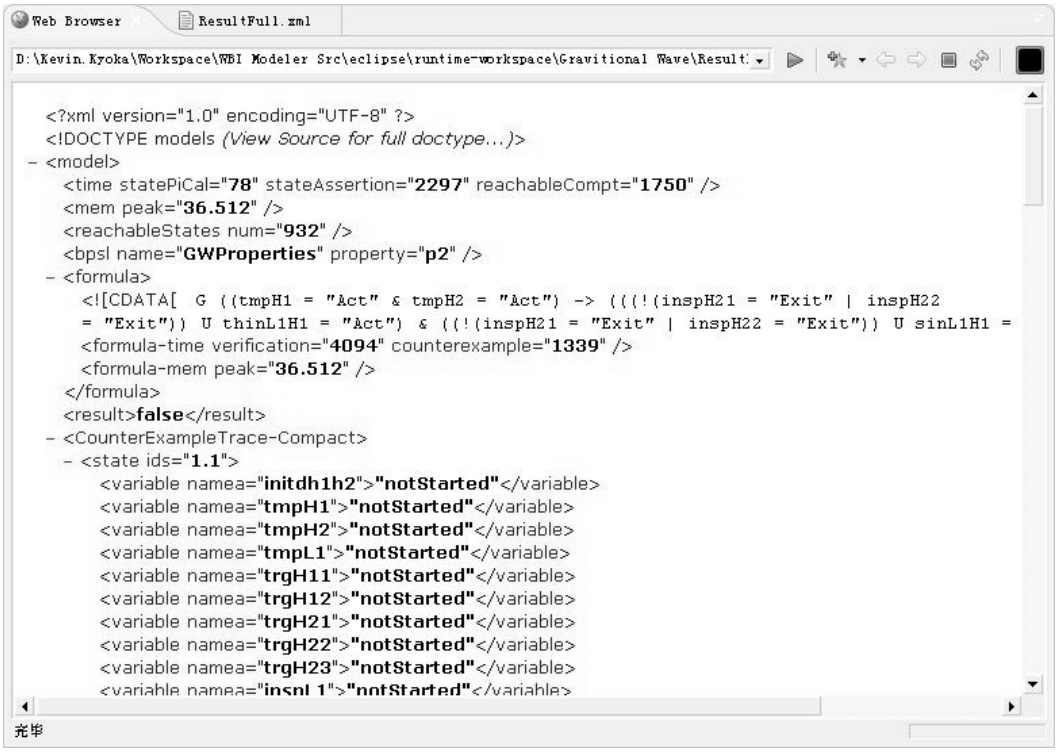


图 6.5　GridPiAnalyzer 中验证结果的 XML 示例

6.5　GridPiAnalyzer中的BPSL可视化业务逻辑描述

由于时序逻辑自身的复杂性及对不直观的符号化语义，成为了网格服务流形式化验证方法在实际应用时的一个瓶颈。为了向实际 GridPiAnalyzer 的用户屏蔽在业务逻辑性质描述时的复杂性，减少对专业逻辑知识的要求，GridPiAnalyzer 中设计并实现了一类可视化业务逻辑建模方式，即我们的业务性质规范语言（BPSL）[43]。BPSL 保证了对 LTL 和 CTL 逻辑的兼容，它是对以上两时序逻辑的可视化。通过 BPSL 能够以可视化的方式描述所需的业务逻辑，并通过业务逻辑解析模块实现从它到 LTL、CTL 逻辑公式的自动转换，并作为待验证性质写入 NuSMV2 的输入模型中。其中，BPSL 建模器中更集成了文献[154]中的 LTL2 Büchi 模块，以实现从 LTL 逻辑公式到对应扩展 Büchi 动机的自动转换，为业务逻辑的一致性验证提供服务。图 6.6 中分别给出了在第 4、5 章中本文所研究的 LIGO 引力波探测数据分析应用中 8 条待验证业务逻辑的 BPSL 可视化描述示例。

为了突出本文在主题上的重点，关于 BPSL 的完整语法、语义、图例以及详

细设计可参见我们在文献[43]中的工作，在此不再详细介绍。而基于 BPSL 对《金融机构反洗钱规定》[30]的条例建模亦可参见我们在文献[34]中的工作。

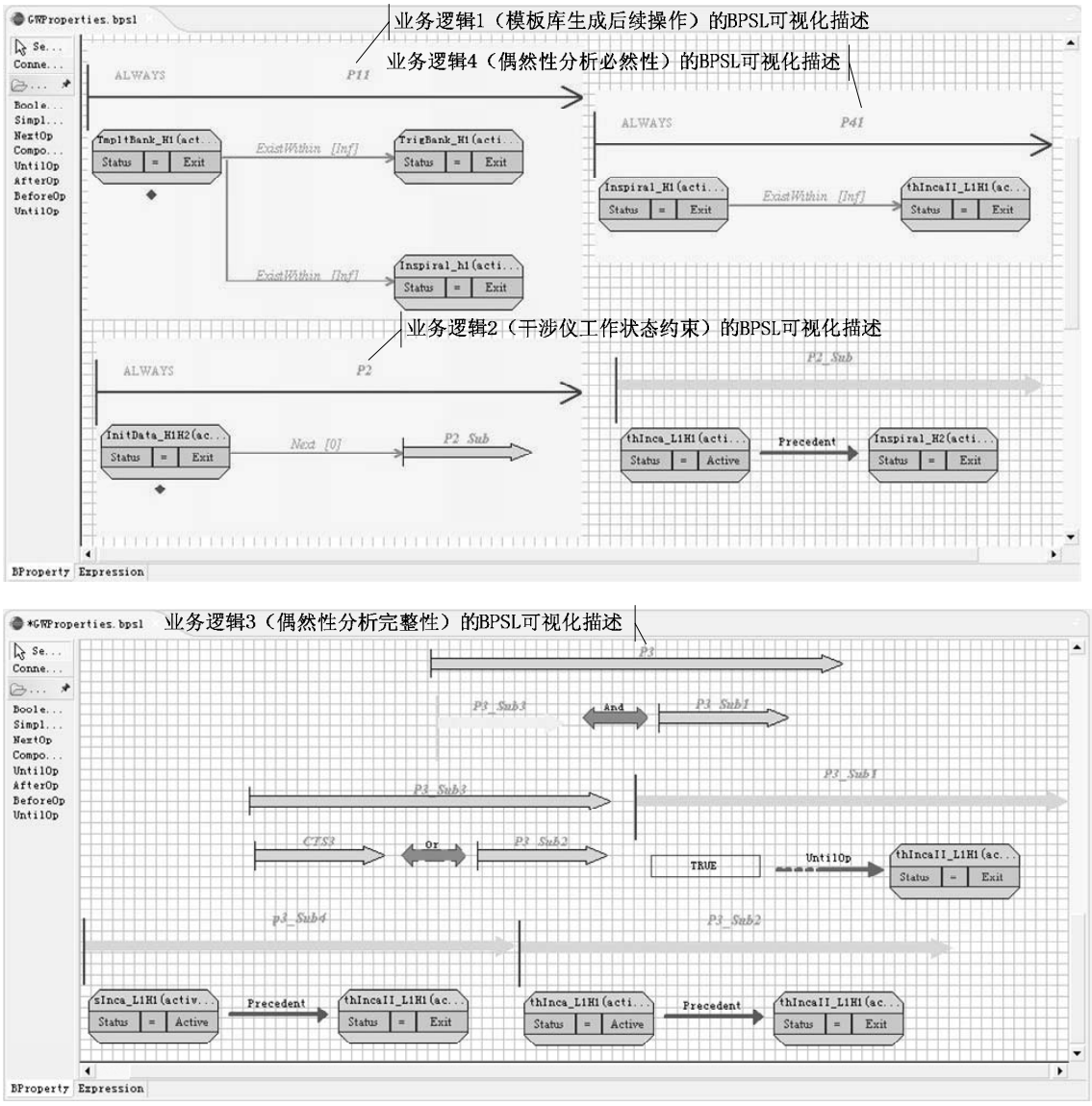


图 6.6 基于 BPSL 的引力波探测数据分析业务逻辑描述

6.6　GridPiAnalyzer的已有应用及背景介绍

本节将给出目前 GridPiAnalyzer 在科研和商用领域中已有的两个应用背景介绍。

6.6.1　LIGO数据网格的引力波探测数据分析应用

根据爱因斯坦广义相对论的预言，大量的致密物质能量发生剧烈移动时（例如中子星的对撞，两个极重的黑洞对撞等），能产生强烈的时空扭曲，并以光速向外传播。这种时空结构的波动称之为引力波。尽管引力波至今尚未能被直接观测到，但引力波的探测和分析对了解天体中未知的物质活动与形成有着深远的科研意义。1993 年 Russell Hulse 和 Joseph Taylor 仍然凭借发现双脉冲星 PSR1913+16 [165]，并显示其互相旋入速率于广义相对论中引力波反作用效应之预测的符合而获得诺贝尔物理学奖。

然而引力波信号相当微弱而难以探测。活跃的天体运动所产生的引力波在传达到地球附近时，其造成地表各处相对距离改变的数量级仅为 10^{-21} 或更小。LIGO（Laser Interferometer Gravitational wave Observatory）项目[45-48]是为了在地面进行引力波探测而建立的激光干涉仪引力波探测台科研项目。LIGO 探测台拥有三套干涉仪来满足对引力波探测所需的灵敏度。其中两套位于美国华盛顿的 Hanford（简记为 H1、H2），一套位于路易斯安娜州的 Livingston（简记为 L1）。多干涉仪的目的是为了能够更好地区分探测信号中的噪声，因为一个真正的引力波信号应该可以被三个干涉仪同时检测到。每个激光干涉仪都有一个臂长为 4 公里或 2 公里的 L 型真空管，通过管内激光束的传播可以精确地测量出臂中所悬挂镜面由于引力波传递而造成的相对距离变化，其灵敏度可达到一颗质子直径的千分之一。

LIGO 数据网格[27]为海量引力波数据的获取与分析提供了一个基础平台。LIGO 数据网格的服务包主要构建在 iVDGL⑤和 OSG⑥项目中所提供的虚拟数据工具集（Virtual Data Toolkit）上，其中的 LIGO 数据复制器模块（LDR）基于 Globus 的 Replica Location Service（RLS）和 GridFTP 为从各个干涉仪产生的数据提供了有效的复制与数据发现服务（Hanford 干涉仪的数据产生速率为平均每秒 8 兆字节；Livingston 干涉仪的数据产生速率为平均每秒 4 兆字节）。而 LIGO 数据网格的客户包则包含了 *LSCdataFind* 和 *LSCsegFind* 这两个工具为 LIGO 数据网格的科研工作者们提供按不同数据条件（如：限定所需数据来源于哪个干涉仪，存储在哪一个帧文件中等）和数据质量信息（如：数据产生时激光干涉仪的运行状态，是否存在异常运行现象等）获取引力波探测数据的能力。

⑤ International Virtual Data Grid Laboratory; 见: *http://www.ivdgl.org*

⑥ The Open Science Grid consortium; 见: *http://www.opensciencegrid.org/*

在 LIGO 数据网格中，引力波探测数据分析的服务流可由其 Glue 构建环境（Grid/LSC User Environment）[27]中提供的 Python 脚本自动构建，并产生相应的 DAGMan 输入脚本或其对应的 XML 语法（DAX）以实现该服务流在 Condor 或 Pegasus[47]引擎上的自动运行。然而对引力波探测数据的分析是一个涉及多任务和海量数据的复杂过程。由于通过 Glue 脚本的服务流生成结果规模庞大，且引力波探测数据分析的实际运行所占用的 CPU 时间可长达几周到几个月，因此通过本文的 GridPiAnalyzer 确保 LIGO 引力波探测数据分析的应用在服务流结构和业务逻辑实现上的有效性和正确性是一项重要而有意义的任务。

图 6.7 在该应用中，GridPiAnalyzer 主要负责自动接收由 Glue 所生成的 Condor DAGMan 脚本并依照 LIGO 用户所给定的 LTL 业务逻辑公式完成其自动验证和 XML 结果返回。有关该应用的三个详细服务流实例 SF1~SF3、其 8 条待验证业务逻辑和完整的验证性能与结果已在第 4、5 章中进行了介绍。

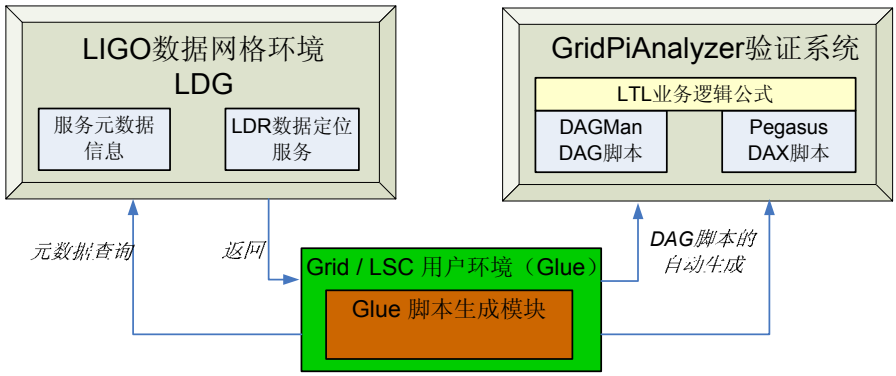


图 6.7 GridPiAnalyzer 在 LIGO 引力波探测数据分析中的应用

6.6.2　基于BPEL4WS的银行开户法律法规验证

随着当今社会全球化脚步的加快和市场经济的日益发展，越来越多的企业开始接受和利用各类信息技术规范和理念（如：COBIT⑦、ITIL⑧等）来实现其业务设计、操作、监控、部署的数字化和自动化。然而伴随着以上信息技术的发展和运用，同样增长的是对企业经营生产活动在各项政策法规上的严格化与标准化。典型的例子包括中国人民银行 2003 年出台的《金融机构反洗钱规定》、欧洲反洗钱法规定、美国爱国者法案、新巴塞尔协议等等。特别是在银行、公共卫生等高度规范化的业务领域，确保企业和机构业务符合相关法规约束和国

⑦ Control Objectives for Information and Related Technology (COBIT), Version 4.0; 见: *http://www.itgi.org*
⑧ IT Infrastructure Library (ITIL), Office of Government Commerce (OGC); 见: *http://www.itil.co.uk*

际标准已经成为一项必要而迫切的任务。一方面，企业试图通过验证其业务的合法性展示其业务的规范和成熟；另一方面，面对各种日新月异的国际标准和相关法规，企业也希望能检验新规章和法规对其现有业务可能造成的影响。

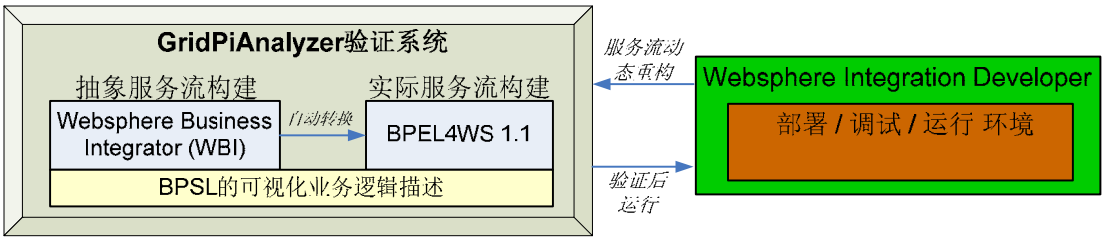


图 6.8 GridPiAnalyzer 在 REALM 框架中的应用

因此，GridPiAnalyzer 除了在科研领域的应用外，在此背景下还作为 REALM 框架[29]的一部分，完成了对某国内银行个人开户服务流针对《金融机构反洗钱规定》[30]的法律法规验证。REALM 是为企业进行内外部规章（Regulation）建模、管理和验证的一套元模型和系统框架。在银行个人开户的解决方案实例中，它包含了对开户服务流进行抽象建模的 IBM Webshpere Business Integrator™（WBI）（其模型可与 UML2.0 活动图部分兼容）、从 WBI™ 到 BPEL4WS 1.1 的自动转换[166]、现有服务及遗留功能的选取、在 IBM Websphere Integration Developer™ 上的部署、调试与运行 4 步骤。而如图 6.8 所示，GridPiAnalyzer 在其中负责实现对基于 WBI™ 和 BPEL4WS 的个人开户抽象/实际服务流的业务逻辑验证，其业务逻辑则进一步采用了本章中业务性质规范语言（BPSL）的可视化描述（见 6.5 小节）。GridPiAnalyzer 在该应用的验证目标主要包括中国人民银行《金融机构反洗钱规定》[30]（2003 年版）的第 11 条和第 13 条：

- “第十一条：金融机构为个人客户开立存款账户、办理结算的，应当要求其出示本人身份证件，进行核对，并登记其身份证件上的姓名和号码。代理他人在金融机构开立个人存款账户的，金融机构应当要求其出示被代理人和代理人的身份证件，进行核对，并登记被代理人和代理人的身份证件上的姓名和号码。”
- “第十三条：金融机构为客户提供金融服务时，发现大额交易的，应当按照有关规定向中国人民银行或者国家外汇管理局报告。”

在该应用的已有验证经验中发现，通过 GridPiAnalyzer 可以在验证时间阀值 15 分钟内完成对规模为 10^6 可达状态的 BPEL4WS 开户服务流的完整

验证。由于该应用涉及涉密信息的授权，因此关于该背景的介绍和应用结果可以参见我们在文献[34]中的工作。

6.7　小结

本章将本文中基于状态π演算的网格服务流形式化验证技术的各研究结果予以系统化和集成，介绍了其自动化验证原型系统 GridPiAnalyzer 的详细实现细节，包括其系统框架与功能模块、状态π演算模型的实现、模型验证引擎的支持和可视化业务逻辑描述方法。GridPiAnalyzer 已在 LIGO 项目的引力波探测数据分析和基于 BPEL4WS 的银行开户系统反洗钱法验证中进行了实际的案例应用。为了支持更多可能的网格服务流验证案例，目前 GridPiAnalyzer 提供了对 BPEL4WS 1.1 规范、Condor DAGMan 脚本和 IBM 的 Websphere Business Integrator™（WBI）的支持。此外，在 GridPiAnalyzer 的应用过程中，我们也协助了 NuSMV2（2.2.5 版本）这一目前 GridPiAnalyzer 所支持的主流开源模型验证引擎在 COI 结构初始化时的一个 Bug 修正。这在其 2.3.1 版本的发布（见：*http://nusmv.irst.itc.it/announce-NuSMV-2.3.1.txt*）中可以找到对我们的致谢。

第 7 章　结束语

网格技术的产生为解决跨组织、跨地域的大规模资源共享与协作提供了一个革新性的方法。一方面，随着对网格相关使能技术（如安全管理、可靠数据传输、事务处理、服务容器等）和规范的研究，使其在天文、生物信息、教育、银行等科研和工业领域拥有越来越广泛的应用和重要的地位。而另一方面，随着网格应用规模的扩大和领域的延伸，如何通过严格的形式化验证方法保障网格服务流在设计与实现过程中的正确性成为了一个亟待解决的重要问题。在网格技术的研究领域，特别是针对复杂网格服务协作流程，目前对该问题的系统化研究与应用成果具有重要的实用意义。该问题的求解涉及系统科学、计算机科学、形式化方法、工业工程等多个交叉研究领域，拥有明确的应用和理论价值，因此有必要进行深入研究。

本文通过结合现有网格服务流规范的自身特点，从对应形式化工具的扩展和分析、网格服务流和常用模式的形式化语义、形式化验证理论的分析与应用、验证技术的性能改进和系统化集成这五个递进层次展开了对网格服务流形式化验证问题的系统研究。部分研究成果已在 LIGO 项目的引力波探测数据分析和银行开户的反洗钱法验证中得到应用验证。

本论文的主要研究成果和创新包括：

1) 状态π演算的提出与分析：

针对 Web 服务资源规范（WSRF）扩充传统 Web 服务对有状态资源的抽象与管理能力，结合形式化方法研究领域中状态/事件混合的思想，本文首先提出了一种称为状态π演算的形式化理论工具，并分析了状态π演算基于状态操作与关联的扩展操作语义、状态互模拟关系及其性质。状态π演算的优点在于它在传统事件（行为）驱动的π演算基础上扩充了其对系统状态生命周期的管理能力，实现了对系统历史事件信息进行管理和灵活业务抽象的能力，从而有效简化了对网格服务流和相应业务逻辑性质的形式化描述。

2) 基于状态π演算的网格服务流形式化语义补充与完善

基于本文的状态π演算，进一步对网格中的服务执行、服务选择以及实

际 BPEL4WS 规范、DAGMan 规范和网格服务流中的并发与管道模式提供了严格的形式化语义。这不仅为网格服务流从单一服务模型、到服务流协作、到相关模式建立了完整的状态π演算语义，也为目前仍缺乏形式化语义支持的网格服务流模式提供了严格的形式化基础，同时更验证了状态π演算自身在网格服务流的形式化建模工作中具有充分的表达能力。

3) 网格服务流基于状态π演算的系统验证方法研究与应用

从网格服务流的结构验证、规范语义约束验证、业务逻辑验证及其一致性检验四个不同方面和静态/动态两个层面对其正确性保障问题进行研究，并实现了网格服务流基于状态π演算的完整形式化验证方法。其中，通过实现从状态π演算语义到对应有穷状态标号迁移系统的自动转换，提出了状态π演算的强/弱状态断言方法，从而实现了网格服务流的结构验证与规范语义约束验证；同时，通过对模型验证技术的支持和灵活复用实现了对网格服务流的业务逻辑验证、状态π演算语义的动态重构，以及冗余和冲突业务逻辑的检验。在 LIGO 数据网格的应用实例中发现，本文基于状态π演算的网格服务流形式化验证方法实现了状态π演算中进程演化和状态操作的同步推演及其模型验证支持，并有着减少重复验证开销和验证方法独立于特定的模型验证引擎的优点。

4) 基于验证分解和错误过程模式的进一步验证性能改进

从网格服务流形式化验证方法的性能改进工作出发，分别研究了基于网格服务流验证分解和错误过程模式这两种思路的验证性能改进方法，从而使本文工作能在实际复杂网格服务流中得到更有效的应用。网格服务流的验证分解首先实现了对服务流基于标准域和并发枝的分解，并分析了各标准域和并发枝间存在的严格顺序和并发关系。在此基础上，基于模块化验证的思想提出了与服务流分解相对应的验证分解策略，从而实现了通过对局部标准域上的验证结果逐一逆向递增推导整个服务流验证结果的完整方法。另一方面，基于错误过程模式的搜索向导方法则根据经典的工作流模式总结了其对应的反模式，用以描述网格服务流中可能存在的常见违例行为与结构，并通过 IEEE 标准的性质描述语言（PSL）给出了它们的形式化语义。此外，通过错误过程模式只需在单一路径上进行非回溯搜索的特点，进一步将搜索向导（Guided Search）结合入网格服务流的形式化验证中。根据 LIGO 数据网格引力波探测数据分析应用中 3 个不同复杂度的实际服务

流实例和数值分析结果发现，这两种方法在提升网格服务流验证效率、降低最大内存占用有明显的效果。

以上研究成果在本文的系统化原型 GridPiAnalyzer 得到了具体的结合与实现，从而为网格服务流的正确性保障提供了自动化和可视化的工具支持。本文部分研究成果已在 LIGO 项目的引力波探测数据分析和银行开系统的反洗钱法验证中得到了实际的应用案例。通过对整体理论研究、数值计算和实际应用的结果表明，本文提出的基于状态π演算的网格服务流形式化验证方法为网格服务流的正确性保障提供了可行的思路，是一种行之有效的方法。

展望下一阶段的研究工作，可以围绕以下几方面展开：

1) 面向过程模型和基于语义的 AI（人工智能）规划是服务组合的两种重要方法。基于本文研究结果，一个后续研究方向是研究其对语义和本体信息的结合和推理，并推广应用于对语义有较高需求的网格应用中；
2) 研究如何从业务逻辑驱动的模型切片和验证反例信息的利用等其它途径来实现对网格服务流形式化验证的性能改进，并将其在实际网格服务流中进行有效性检验和应用；
3) 如何将本文研究结果在形式化验证层次上从抽象和具体的服务流模型规范延伸至更底层的实现代码验证，从而在纵向上形成对不同层次间模型特点和验证性质上关联的分析，也是下一步的重要研究工作之一。

参考文献

[1] Foster I, Kesselman C and Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. International Journal of High Performance Computing Applications, 2001, 15 (3): 200-222

[2] Foster I and Kesselman C. 网格计算 (金海, 袁平鹏, 等., 译者). 北京: 电子工业出版社, 2004. 1-54

[3] Stevens R, Woodward P, DeFanti T, et al. From the I-WAY to the national technology grid. Communications of the ACM, 1997, 40 (11): 50-60

[4] Catlett C E. TeraGrid: A foundation for US cyberinfrastructure. Network and Parallel Computing, LNCS, 2005, 3779: 1

[5] Coles J. The evolving grid deployment and operations model within EGEE, LCG and GridPP. in: H. Stockinger, R. Buyya, et al. (eds). Proceedings of the First International Conference on e-Science and Grid Computing. California: IEEE Computer Society, 2005. 90-97

[6] Miura K. Overview of Japanese science Grid project NAREGI. Progress in Informatics, 2006 (3): 67-75

[7] Jin H. ChinaGrid: Making grid computing a reality. Digital Libraries: International Collaboration and Cross-Fertilization, LNCS, 2004, 3334: 13-24

[8] Depei Q. CNGrid: A test-bed for grid technologies in China. in: B. Xu, U. Ramachandran, et al. (eds). 10th IEEE International Workshop on Future Trends of Distributed Computing Systems. New Zealand: IEEE Computer Society, 2004. 135-139

[9] 怀进鹏, 胡春明, 李建欣, 等. CROWN:面向服务的网格中间件系统与信任管理. 中国科学 E 辑, 2006, 36 (10): 1127-1155

[10] 徐志伟, 冯百明, 李伟. 网格计算技术. 北京: 电子工业出版社, 2004. 25-162

[11] Deak O. Grid service execution for JOpera: [Master Thesis]. Zurich: Swiss Federal Institue of Technology, 2005

[12] Czajkowski K, Ferguson D F, Foster I, et al. The WS-Resource Framework [EB/OL]. [2004-3-5]. http://www.globus.org/wsrf/specs/ws-wsrf.pdf

[13] Foster I, Frey J, Graham S, et al. Modeling stateful resources with Web services [EB/OL]. [2004-3-5]. http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf

[14] Wasson G, Humphrey M, Wasson G, et al. State and events for Web services:a comparison of five WS-resource framework and WS-Notification implementations. in: A. Grimshaw, M. Parashar, et al. (eds). 14th IEEE International Symposium on High Performance Distributed Computing. Research Triangle Park: IEEE Computer Society, 2005. 24-27

[15] Foster I. GT4 Primer [EB/OL]. [2005-5-8]. http://www.globus.org/toolkit/docs/4.0/key/

[16] Sotomayor B. The Globus Toolkit 4 Programmer's Tutorial [EB/OL]. [2005-11-26].

http://gdp.globus.org/gt4-tutorial/

[17] Li M and Baker M. 网格计算核心技术 (王相林, 张善卿, 等., 译者). 北京: 清华大学出版社, 2006. 165-261

[18] Foster I, Czajkowski K, Ferguson D F, et al. Modeling and managing state in distributed systems: The role of OGSI and WSRF. Proceedings of the IEEE, 2005, 93 (3): 604-612

[19] Czajkowski K, Ferguson D, Foster I, et al. From Open Grid Services Infrastructure to WSResource Framework: Refactoring & Evolution [EB/OL]. [2004-3-5]. http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf

[20] 吴澄, 李伯虎. 从计算机集成制造到现代集成制造--兼谈中国 CIMS 系统论的特点. 计算机集成制造--CIMS, 1998, 4 (5): 1-6

[21] 吴澄. 现代集成制造系统的理论基础--一类复杂性问题及其求解. 计算机集成制造--CIMS, 2001, 7 (3): 1-7

[22] Yu J and Buyya R. A taxonomy of workflow management systems for grid computing. (TR). Grid Computing and Distributed Systems Laboratory, University of Melbourne. Australia: March 10, 2005, p.1-31. 2005

[23] Cesare Pautasso G A. Parallel computing patterns for grid workflows. in: E. Deelman (eds). Workshop on Workflows in Support of Large-Scale Science. Paris: IEEE Computer Society, 2006.

[24] Fu X, Bultan T and Su J W. Realizability of conversation protocols with message contents. in: B. Werner (eds). IEEE International Conference on Web Services. California: IEEE Computer Society, 2004. 96-103

[25] 李景霞, 侯紫峰. Web 服务组合综述. 计算机应用研究, 2005, 22 (12): 4-7

[26] Antonioletti M. 'Workflow' issues in data access and integration: An OGSA-DAI/DAIS perspective. in: D. Berry and S. Parastatidis (eds). e-Science Workflow Services Workshops. Edinburgh: National e-Science Center, 2003.

[27] Brown. D A, Brady. P R, Alexander D, et al. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. in: I. J. Taylor., D. Ewa, et al. (eds). Workflows for e-Science. Heidelberg: Springer-Verlag, 2006. 41-61

[28] Maglio P P, Srinivasan S, Kreulen J T, et al. Service systems, service scientists, SSME, and innovation. Communications of the ACM, 2006, 49 (7): 81-85

[29] Giblin C, Liu A Y, Muller S, et al. Regulations Expressed As Logical Models (REALM). in: M.-F. Moens and P. Spyns (eds). 18th Annual Conference on Legal Knowledge and Information Systems. Amsterdam: IOS Press, 2005. 37-48

[30] 中国人民银行. 金融机构反洗钱规定 [EB/OL]. [2003-1-3]. http://www.pbc.gov.cn/detail.asp?col=1510&ID=17

[31] BCBS. The Revised Basel Capital Framework (Basel II) [EB/OL]. [2004-6-26]. http://www.federalreserve.gov/boarddocs/press/bcreg/2004/20040626/attachment.pdf

[32] Statutory Instrument No. 3075. The Money Laundering Regulations 2003 [EB/OL]. [2003-11-28]. http://www.opsi.gov.uk/si/si2003/20033075.htm

[33] Electronic Privacy Information Center. USA Patriot Act of 2001, PL 107-56, HR 3162 RDS

[EB/OL]. [2001-10-24]. http://www.epic.org/privacy/terrorism/hr3162.html

[34] (All first authors) Xu K, Samuel M and Liu Y. A Static Compliance Checking Framework for Business Process Models. IBM Systems Journal, 2007, 46 (2): 1-27

[35] Xu K, Wang Y X and Wu C. Formal verification technique for grid service chain model and its application. Science in China (Series F), 2006, 50 (1): 1-20

[36] Xu K, Liu L C and Wu C. A Three Layered Method for Business Process Discovery and its Application. Computers in Industry, 2007, 58(3): 265-278

[37] Xu K, Wang Y X and Wu C. Ensuring secure and robust grid applications - From a formal method point of view. Advances in Grid and Pervasive Computing, LNCS, 2006, 3947: 537-546

[38] Xu K, Wang Y X and Wu C. Aspect oriented region analysis for efficient equipment grid application reasoning. in: L. O'Conner (eds). 5th International Conference on Grid and Cooperative Computing. California: IEEE Computer Society, 2006. 28-31

[39] Wang Y X, Wu C and Xu K. Study on pi-calculus based equipment grid service chain model. Network and Parallel Computing, LNCS, 2005, 3779: 40-47

[40] Xu K, Liu Y and Wu C. Guided reasoning of complex E-business processes with business bug patterns. in: S. Kawada (eds). International Conference on E-Business Engineering. California: IEEE Computer Society, 2006. 195-202

[41] Xu K, Liu Y, Zhu J, et al. Pi Calculus based Bi-transformation of State-driven Model and Flow-driven Model. International Journal of Business Process Integration and Management, 2007, 1(4): 292-306

[42] Xu K, Wang Y X and Wu C. Service provenance based abstraction of grid application knowledge. in: L. O'Conner (eds). International Conference on Semantic and Knowledge Grid. California: IEEE Computer Society, 2006. 50-53

[43] Xu K, Liu Y and Wu C. BPSL Modeler - Visual notation language for intuitive business property reasoning. in: R. Bruni and D. Varro (eds). Electronic Notes in Theoretical Computer Science. Austria: Elsevier, 2006. 205-214

[44] 许可, 刘连臣, 吴澄. 时间π演算及其弱时间互模拟分析. 计算机集成制造系统, 2006, 12 (4): 511-516

[45] Abramovici A, Althouse W E, Drever R W P, et al. LIGO - The Laser Interferometer Gravitational-Wave Observatory. Science, 1992, 256 (5055): 325-333

[46] Barish B C and Weiss R. LIGO and the detection of gravitational waves. Physics Today, 1999, 52 (10): 44-50

[47] Blythe J, Deelman E and Gil Y. Automatically composed workflows for grid environments. IEEE Intelligent Systems, 2004, 19 (4): 16-23

[48] Deelman E, Blythe J, Gil Y, et al. Mapping abstract complex workflows onto grid environments. Journal of Grid Computing, 2003, 1 (1): 9-23

[49] Snyder K. deconstruction: LIGO analysis. Symmetry, 2005, 2 (9): 32-33

[50] Brown D A. Using the INSPIRAL program to search for gravitational waves from low-mass binary inspiral. Classical and Quantum Gravity, 2005, 22 (18): S1097-S1107

[51] 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 F 辑, 2006: 已接收

[52] Li H C and Yang Y. Dynamic checking of temporal constraints for concurrent workflows. Electronic Commerce Research and Applications, 2005, 4 (2): 124-142

[53] Chen J J and Yang Y. Key Research Issues in Grid Workflow Verification and Validation. in: R. Buyya and T. Ma (eds). 4th Australasian Symposium on Grid Computing and e-Research. Australia: Australian Grid Forum, 2006.

[54] Chaki S, Clarke E M, Ouaknine J, et al. State/Event-Based software model checking. Integrated Formal Methods, LNCS, 2004, 2999: 128-147

[55] Hansen H, Virtanen H and Valmari A. Merging state-based and action-based verification. in: J. Lilius, F. Balarin, et al. (eds). 3rd International Conference on Application of Concurrency to System Design. Portugal: IEEE Computer Society, 2003. 150-156

[56] Chechik M and Paun D O. Events in property patterns Theoretical and Practical Aspects of SPIN Model Checking, LNCS, 1999, 1680: 154 - 167

[57] Clarke E M, Jr Grumberg O and Peled D A. Model Checking. Cambridge, Mass: MIT Press, 1999. 1-231

[58] Amin K, Von Laszewski G, Hategan M, et al. GridAnt: A client-controllable grid workflow system. in: R. H. Sprague (eds). 37th Annual Hawaii International Conference on System Sciences. Hawaii: IEEE Computer Society, 2004. 3293-3301

[59] Andrews T, Curbera F, Dholakia H, et al. Business Process Execution Language for Web Services Version 1.1 [EB/OL]. [2003-5-5]. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[60] Alexandre Alves, Arkin A, Askary S, et al. Web Services Business Process Execution Language Version 2.0 (Public Review Draft) [EB/OL]. [2006-8-23]. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

[61] Yu J and Buyya R. A novel architecture for realizing grid workflow using tuple spaces. in: B. Gropp, D. Reed, et al. (eds). 5th IEEE/ACM International Workshop on Grid Computing. Pittsburgh: IEEE Computer Society, 2004. 119-128

[62] Taylor I, Shields M, Wang I, et al. Visual grid workflow in Triana. Journal of Grid Computing, 2005, 3 (3-4): 153-169

[63] Pautasso C and Alonso G. The JOpera visual composition language. Journal of Visual Languages and Computing, 2005, 16 (1-2): 119-152

[64] Deelman E, Blythe J, Gil Y, et al. Pegasus and the pulsar search: From metadata to execution on the grid. Parallel Processing and Applied Mathematics, LNCS, 2004, 3019: 821-830

[65] Fahringer T, Jun Q and Hainzer S. Specification of Grid workflow applications with AGWL: An abstract Grid workflow language. in: D. W. Walker and C. Kesselman (eds). IEEE International Symposium on Cluster Computing and the Grid. Cardiff: Institute of Electrical and Electronics Engineers Computer Society, 2005. 676-685

[66] Hunt C S, Ferner C S and Brown J L. JXPL: An XML-based scripting language for workflow execution in a grid environment. in: Y. Levy (eds). IEEE SouthestCon. Ft. Lauderdale:

Institute of Electrical and Electronics Engineers Inc, 2005. 345-350

[67] Krishnan S, Wagstrom P and Laszewski G v. GSFL: A workflow framework for Grid services [EB/OL]. [2002-7-19]. http://www-unix.globus.org/cog/papers/gsfl-paper.pdf

[68] Long Y, Lam H and Su S Y W. Adaptive grid service flow management: Framework and model. in: B. Werner (eds). IEEE International Conference on Web Services. California: IEEE Computer Society, 2004. 558-565

[69] Pllana S, Fahringer T, Testori J, et al. Towards an UML based graphical representation of grid workflow applications. Grid Computing, LNCS, 2004, 3165: 149-158

[70] Hoheisel A and Der U. An XML-based framework for loosely coupled applications on Grid environments. Computational Science, LNCS, 2003, 2657: 245-254

[71] Slomiski A. On using BPEL extensibility to implement OGSI and WSRF Grid workflows. Concurrency and Computation-Practice & Experience, 2006, 18 (10): 1229-1241

[72] Wang Y, Hu C and Huai J. A new grid workflow description language. in: C. K. Chang and L. J. Zhang (eds). IEEE International Conference on Services Computing. Orlando: IEEE Computer Society, 2005. 257-258

[73] Chao K-M, Younas M, Griffiths N, et al. Analysis of grid service composition with BPEL4WS. in: L. Barolli (eds). 18th International Conference on Advanced Information Networking and Applications. Washington: IEEE Computer Society, 2004.

[74] Vardi M Y. Branching vs. Linear time: Final showdown. Tools and Algorithms for the Construction and Analysis of Systems, LNCS, 2001, 2031: 1-22

[75] Alur R, Henzinger T A and Kupferman O. Alternating-time temporal logic. Compositionality: The Significant Difference, LNCS, 1997, 1536: 23-60

[76] Lamport L. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 1994, 16 (3): 872-923

[77] Hornos M J and Capel M I. On-the-fly model checking from interval logic specifications. ACM SIGPLAN Notices, 2002, 37 (12): 108 - 119

[78] Bimbo D, L R A and E V. Visual specification of branching time temporal logic. in: V. Harrslev (eds). 11th IEEE Symp. on Visual Languages. Darmstadt: IEEE Computer Society, 1995. 61-68

[79] Brambilla M, Deutsch A, Sui L, et al. The role of visual tools in a Web application design and verification framework: A visual notation for LTL formulae. Web Engineering, LNCS, 2005, 3579: 557-568

[80] Ferrari G L, Gnesi S, Montanari U, et al. A model-checking verification environment for mobile processes. ACM Transactions on Software Engineering and Methodology, 2003, 12 (4): 440-473

[81] Chen T, Han T and Lu J. A modal logic for Pi-calculus and model checking algorithm. Electronic Notes in Theoretical Computer Science, 2005, 123: 19-33

[82] Geist D. The PSL/Sugar specification language a language for all seasons. Journal on Data Semantics, LNCS, 2003, 2800: 3-3

[83] Clarke E M, McMillan K L, Campos S, et al. Symbolic model checking. Computer Aided

Verification, LNCS, 1996, 1102: 419

[84] Clarke E, Biere A, Raimi R, et al. Bounded model checking using satisfiability solving. Formal Methods in System Design, 2001, 19 (1): 7-34

[85] Grumberg O and Long D E. Model checking and modular verification. ACM Transactions on Programming Languages and Systems, 1994, 16 (3): 843-871

[86] Yorav K and Grumberg O. Syntax-directed model checking of sequential programs. Journal of Logic and Algebraic Programming, 2002, 52-53: 129-162

[87] Jonsson B and Tsay Y-K. Assumption/Guarantee specifications in Linear-Time temporal logic. Theoretical Computer Science, 1996, 167: 47-72

[88] Loiseaux C, Graf S, Sifakis J, et al. Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design, 1995, 6 (1): 11-44

[89] Bloem R, Ravi K and Somenzi F. Symbolic guided search for CTL model checking. in: IEEE (eds). 37th Design Automation Conference. Los Angeles: ACM, 2000. 29-34

[90] Peranandam P M, Weiss R J, Ruf J, et al. Dynamic guiding of bounded property checking. in: L. Fournier (eds). IEEE International High Level Design Validation and Test Workshop. Los Alamitos: IEEE Computer Society, 2004. 15-18

[91] Emerson E A and Halpern J Y. Sometimes and not never revisited: On branching versus linear time. Journal of the ACM, 1986, 33 (1): 151-178

[92] Clarke E M and Draghicescu I A. Expressibility results for linear-time and branching-time logics. Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS, 1988, 354: 428-437

[93] Vardi M Y. Sometimes and not never re-revisited: on branching vs. linear time. Concurrency Theory, LNCS, 1998, 1466: 1-17

[94] Wang W L, Hidvegi Z, Bailey A D, et al. E-process design and assurance using model checking. Computer, 2000, 33 (10): 48-+

[95] Anderson B B, Hansen J V, Lowry P B, et al. Model checking for design and assurance of e-Business processes. Decision Support Systems, 2005, 39 (3): 333-344

[96] OASIS WSBPEL TC No. 42. Need for formalization [EB/OL]. [2003-7-31]. http://www.oasis-open.org/archives/wsbpel/200307/msg00177.html

[97] Baeten J C M. A brief history of process algebra. Theoretical Computer Science, 2005, 335 (2-3): 131-146

[98] Sangiorgi D and Walker D. The Pi-calculus : a theory of mobile processes. Cambridge: Cambridge University Press, 2001. 1-310

[99] Wombacher A, Fankhauser P and Neuhold E J. Transforming BPEL into annotated deterministic finite state automata for service discovery. in: B. Werner (eds). IEEE International Conference on Web Services. California: IEEE Computer Society, 2004. 316-323

[100] Fu X, Bultan T and Su J W. WSAT: A tool for formal analysis of web services. Computer Aided Verification, LNCS, 2004, 3114: 510-514

[101] Farahbod R, Glasser U and Vajihollahi M. Specification and validation of the business process

execution language for web services. Abstract State Machines: Advances in Theory and Practice, LNCS, 2004, 3052: 78-94

[102] Berardi D, De Rosa F, De Santis L, et al. Finite state automata as conceptual model for e-Services. Journal of Integrated Design and Process Science, 2004, 8 (2): 105-121

[103] Cimatti A, Clarke E, Giunchiglia E, et al. NuSMV2: an open source tool for symbolic model checking. Computer Aided Verification, LNCS, 2002, 2404: 359-364

[104] Ben-David S, Eisner C, Geist D, et al. Model checking at IBM. Formal Methods in System Design, 2003, 22 (2): 101-108

[105] Foster H, Uchitel S, Magee J, et al. LTSA-WS: a tool for model-based verification of web service compositions and choreography. in: L. J. Osterweil, H. D. Rombach, et al. (eds). 28th International Conference on Software Engineering. Shanghai: ACM, 2006. 771-774

[106] Tang Y, Chen L, He K-T, et al. SRN:an extended Petri-net-base workflow model for web service composition. in: B. Werner (eds). IEEE International Conference on Web Services. California: IEEE Computer Society, 2004. 591-599

[107] Yang Y, Tan Q, Xiao Y, et al. Exploiting hierarchical CP-Nets to increase the reliability of Web services workflow. in: (eds). International Symposium on Applications on Internet. California: IEEE Computer Society, 2006. 116 - 122

[108] Ouyang C, Verbeek E, van der Aalst W M P, et al. WofBPEL: A tool for automated analysis of BPEL processes. Service-Oriented Computing, LNCS, 2005, 3826: 484-489

[109] Lohmann N, Massuthe P, Stahl C, et al. Analyzing interacting BPEL processes. Business Process Management, LNCS, 2006, 4102: 17-32

[110] Puhlmann F and Weske M. Using the Pi-calculus for formalizing workflow patterns. Business Process Management, LNCS, 2005, 3649: 153-168

[111] Decker G, Puhlmann F and Weske M. Formalizing service interactions. Business Process Management, LNCS, 2006, 4102: 414-419

[112] Salaun G, Bordeaux L and Schaerf M. Describing and reasoning on Web services using process algebra. in: B. Werner (eds). IEEE International Conference on Web Services, 2004. 43-50

[113] Camara J, Canal C and Cubo J. Issues in the formalization of Web service orchestrations. in: S. Becker, C. Canal, et al. (eds). 2nd International Workshop on Coordination and Application Techniques for Software Entities. Scotland: Springer-Verlag, 2005. 17-24

[114] Liu F, Zhang L, Shi Y, et al. Formal analysis of compatibility of Web services via CCS. in: M. Paprzycki (eds). International Conference on Next Generation Web Services Practices. California: IEEE Computer Society, 2005. 143-148

[115] Yinxing W, Shensheng Z and Farong Z. On formalizing and verifying web services. High Technology Letters, 2005, 11 (1): 47-50

[116] Gao C, Liu R, Song Y, et al. A model checking tool embedded into services composition environment. in: L. O'Conner (eds). 5th International Conference on Grid and Cooperative Computing. California: IEEE Computer Society, 2006. 355-362

[117] 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

[118] Victor B and Moller F. The mobility workbench - A tool for the Pi-calculus. Computer Aided Verification, LNCS, 1994, 818: 428-440

[119] Zsolt N and Vaidy S. Characterizing grids: Attributes, definitions, and formalisms. Journal of Grid Computing, 2003, 1: 9-23

[120] Nemeth Z and Sunderam V. A formal framework for defining grid systems. in: IEEE (eds). 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID2002. Berlin: IEEE Computer Society, 2002. 202-211

[121] 张鹏, 杜玉越, 左风朝. 网格体系的 Petri 网模拟与分析. 系统仿真学报, 2005, 17 (z1): 223-228

[122] Nemeth Z, Perez C and Priol T. Workflow enactment based on a chemical metaphor. in: B. K. Aichernig and B. Beckert (eds). 3rd IEEE International Conference on Software Engineering and Formal Methods. California: IEEE Computer Society, 2005. 127-136

[123] Groth P and Moreau M L L. Formalising a protocol for recording provenance in Grids. in: S. Cox (eds). UK OST e-Science second All Hands Meeting. Nottingham: EPSRC, 2004. 147-154

[124] 卢暾, 张望, 李志蜀, 等. 基于 I/O 自动机的网格服务组合的形式化. 华南理工大学学报（自然科学版）, 2005, 33 (11): 55-60

[125] van der Aalst W M P. Pi Calculus versus Petri Nets: Let us eat humble Pie rather than further inflate the Pi hype. BPTrends, 2005, 3 (5): 1-11

[126] Smith H. Business process management—the third wave: business process modeling language (bpml) and its pi-calculus foundations. Information and Software Technology, 2003, 45: 1065-1069

[127] Lumpe M, Achermann F and Nierstrasz O. Foundations of component based systems. Cambridge: Cambridge University Press, 2000. 69-90

[128] Nierstrasz O and Meijler T D. Requirements for a composition language. Object-Based Models and Languages for Concurrent Systems, LNCS, 1995, 924: 147-161

[129] Pahl C. A formal composition and interaction model for a Web component platform. Electronic Notes in Theoretical Computer Science, 2002, 66 (4): 1-15

[130] Lumpe M. A Pi-calculus based approach for software composition. Switzerland: Institute of Computer Science and Applied Mathematics, University of Bern, 1999

[131] Vitus S W L and Julian P. Analyzing equivalences of UML statechart diagrams by structural congruence and open bisimulations. in: J. Hosking and P. Cox (eds). IEEE Symposium on Human Centric Computing Languages and Environments. New Zealand: IEEE Computer Society, 2003. 137-144

[132] Yang D and Zhang S S. Using pi-calculus to formalize UML activity diagram for business process modeling. in: P. Alexander (eds). 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems. California: IEEE Computer Society, 2003. 47-54

[133] Puhlmann F and Weske M. Investigations on soundness regarding lazy activities. Business

Process Management, LNCS, 2006, 4102: 145-160

[134] Gorla D. On the relative expressive power of calculi for mobility. Technical Report in Dip. di Informatica, Univ. di Roma "La Sapienza", 2006: 1-31

[135] Dam M. Model checking mobile processes. Information and Computation, 1996, 129 (1): 35-51

[136] Bouali A, Gnesi S and Larosa S. JACK: Just another concurrency kit: The integration project. Bulletin of the European Association for Theoretical Computer Science, 1994, 54: 207-224

[137] Foster I. Globus toolkit version 4: Software for service-oriented systems. Network and Parallel Computing, LNCS, 2006, 3779: 2-13

[138] 郭小群, 郝克刚. Web 服务的 Pi 演算描述. 计算机科学, 2006, 33 (3): 261-262

[139] Bolognesi T. A conceptual framework for state-based and event-based formal behavioural specification languages. in: B. Steffen, P. Bellini, et al. (eds). 9th International Conference on Engineering of Complex Computer Systems. Florence: IEEE Computer Society, 2004. 107-116

[140] Abadi M and Lamport L. Composing specifications. ACM Transactions on Programming Languages and Systems, 1993, 15 (1): 73-132

[141] Nicola R D and Vaandrager F. Three logics for branching bisimulation. Journal of the ACM, 1995, 42 (2): 458-487

[142] Giannakopoulou D and Magee J. Fluent model checking for event-based systems. in: L. Barroca (eds). 11th ACM SIGSOFT Symposium on Foundations of Software Engineering. Helsinki: ACM, 2003. 257-266

[143] Taguchi K, Dong J S and Ciobanu G. Relating pi-calculus to Object-Z. in: B. Steffen, P. Bellini, et al. (eds). 9th International Conference on Engineering of Complex Computer Systems. Florence: IEEE Computer Society, 2004. 97-106

[144] Milner R. Communicating and Mobile Systems: the Pi Calculus. Cambridge: Cambridge University Press, 1999. 16-97

[145] Kacsuk P, Dozsa G, Kovacs J, et al. P-GRADE: A grid programming environment. Journal of Grid Computing, 2003, 1 (2): 171-197

[146] 穆黎森. CGSP 网格中工作流系统的研究：复合资源管理: [硕士论文]. 北京: 清华大学, 2006

[147] Montanari U and Pistore M. Checking bisimilarity for finitary Pi-calculus. Concurrency Theory, LNCS, 1995, 962: 42-56

[148] Chen J and Yang Y. An activity completion duration based checkpoint selection strategy for dynamic verification of fixed-time constraints in grid workflow systems. Lecture Notes in Informatics, 2005, 69: 296-310

[149] Chen J and Yang Y. Temporal dependency for dynamic verification of fixed-date constraints in grid workflow systems. Web Technologies Research and Development, LNCS, 2005, 3399: 820-831

[150] Ioannidis Y E and Sellis T K. Supporting inconsistent rules in database systems. Journal of Intelligent Information Systems, 1992, 1 (3-4): 243-270

[151] Quinlan J R. Induction of decision trees. Machine Learning, 1986, 1: 81-106

[152] Chomicki J, Lobo J and Naqvi S. Conflict resolution using logic programming. IEEE Transactions on Knowledge and Data Engineering, 2003, 15 (1): 244-249

[153] Lindgren T. Methods for rule conflict resolution. Machine Learning, LNCS, 2004, 3201: 262-273

[154] Giannakopoulou D and Lerda F. From states to transitions: Improving translation of LTL formulae to Buchi automata. Formal Techniques for Networked and Distributed Sytems, LNCS, 2002, 2529: 308-326

[155] Ho Y-C and Pepyne D L. Simple explanation of the No Free Lunch Theorem of Optimization. in: M. Dymkov, I. Gaishun, et al. (eds). EEE Conference on Decision and Control. Orlando: Institute of Electrical and Electronics Engineers Inc., 2001. 4409-4414

[156] Liu R and Kumar A. An analysis and taxonomy of unstructured workflows. Business Process Management, LNCS, 2005, 3649: 268-284

[157] Vardi M Y. On the complexity of modular model checking. in: (eds). San Diego, CA, USA: IEEE, Piscataway, NJ, USA, 1995. 101-111

[158] 董威, 王戟, 齐治昌. UML 模型中并发对象的组合验证. 计算机科学, 2005, 32 (7): 231-234

[159] Een N and Sorensson N. An extensible SAT-solver. Theory and Applications of Satisfiability Testing, LNCS, 2003, 2919: 502-518

[160] Yang C H and Dill D L. Validation with guided search of the state space. in: IEEE (eds). 35th Design Automation Conference. San Francisco: ACM, 1998. 599-604

[161] Seppi K, Jones M and Lamborn P. Guided model checking with a bayesian meta-heuristic. Fundamenta Informaticae, 2006, 70 (1-2): 111-126

[162] Ravi K and Somenzi F. Hints to accelerate symbolic traversal. Correct Hardware Design and Verification Methods, LNCS, 1999, 1703: 250-264

[163] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, et al. Workflow patterns. Distributed and Parallel Databases, 2003, 14: 5-51

[164] Havey M. Essential Business Process Modeling. USA: O'Reilly, 2005. 1-332

[165] Hulse R A and Taylor J H. Discovery of a pulsar in a binary system. Journal of Astrophys, 1975, 195: L51

[166] Mantell K. From UML to BPEL. IBM DeveloperWorks, 2005: 1-10

[167] Heinis T, Pautasso C, Alonso G. Mirroring resources or mapping requests: implementing WS-RF for grid workflows. in: IEEE (eds). 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06). Florence: IEEE Computer Society, 2006. 497-504

致　谢

衷心感谢我的导师吴澄教授对本人的悉心指导，他高深的学术造诣和严谨的治学态度激励着我不断努力进取。吴老师从各方面对我的极大关怀和帮助令我永铭于心，在此谨表示衷心的感谢！

感谢信息科学与技术国家实验室曹军威老师在 LIGO 数据网格项目上的合作与指导。感谢 CIMS 研究中心宋士吉老师、刘连臣老师、王跃宣老师以及全体同窗的热情帮助和支持！感谢 IBM 中国研究中心的刘英博士以及 Zurich 研究中心的 Samuel Muller 博士对本文工作的支持。

感谢我的母亲对我的一贯支持和关爱！

本课题承蒙国家自然科学基金和国家教育部 211 工程项目资助，特此致谢。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____日　期：_____

附录 A　LIGO 引力波探测数据分析实例 SF1~SF3 的完整验证性能及其比较

- SF：待验证的网格服务流及其（松弛后）分解的标准域；
- Prop：待验证的业务逻辑公式；
- RA：（松弛后）标准域的分解时间；
- SLTS：SF 对应的状态标号迁移系统和状态断言检查时间；
- SMCA’：符号模型验证算法 SMCA 的可达状态计算时间；
- SMCA：符号模型验证算法 SMCA 对 Prop 的验证总时间；
- COI：SMCA+COI 对 Prop 的验证总时间；
- Dynamic：SMCA+Dynamic 对 Prop 的验证总时间；
- BMC（k）：BMC 对 Prop 的验证总时间，k 为其步长设置；

其中，对验证时间的阀值（超过 10 分钟）和内存占用阀值（小于 10Mb）在表中分别以>10m 和<10Mb 表示。

验证的硬件环境为：Pentium 4 1.73G CPU，2.0G DDR2 RAM，40G 5400-RPM 硬盘；软件环境为：Windows XP SP2，J2SDK1.4.2 + MinGW，Ecplise 开发平台。

表 A.1　完整的 SF1 及其松弛后标准域的验证时间性能比较（单位：ms）

SF	RA	Prop	SLTS	SMCA’	SMCA	COI	Dynamic	BMC(10)	结果
R11		p11	125	47	78	93	203	2750	TRUE
		p12	157	63	110	109	203	4125	TRUE
		p2	109	47	63	78	125	1625	TRUE
		p3	125	62	109	141	250	2109	TRUE
		p41	109	62	62	78	109	1859	TRUE
		p42	110	63	63	78	110	1858	TRUE
		p43	109	47	62	79	109	2078	TRUE
		p44	109	47	62	94	125	2609	TRUE

表 A.1（续）完整的 SF1 及其松弛后标准域的验证时间性能比较（单位：ms）

R12		p11	281	156	203	235	1391	3625	TRUE
		p12	359	203	344	547	1594	8094	TRUE
		p2	296	188	265	328	1469	3328	TRUE
		p3	297	172	265	547	1734	3422	TRUE
		p41	312	172	235	313	1531	3265	TRUE
		p42	344	203	281	406	1562	3922	TRUE
		p43	318	188	256	375	1593	3406	TRUE
		p44	318	188	250	359	1578	3359	TRUE
R13		p11	750	593	703	672	3703	7875	TRUE
		p12	750	593	813	953	3906	24781	TRUE
		p2	750	593	672	719	3656	7453	TRUE
		p3	750	593	687	859	3843	20375	TRUE
		p41	750	593	797	1000	4000	29281	TRUE
		p42	750	593	828	1031	4125	12250	TRUE
		p43	750	593	781	1000	3922	17500	TRUE
		p44	750	593	781	1016	3937	20328	TRUE
SF1	369	p11	2625	1908	2659	3625	13297	66281	TRUE
		p12	2625	1908	3453	6156	15047	52656	TRUE
		p2	2625	1908	2485	3313	13406	46219	TRUE
		p3	2625	1908	4437	7938	15954	184391	TRUE
		p41	2625	1908	3031	4922	13859	56156	TRUE
		p42	2625	1908	2937	5485	14141	62313	TRUE
		p43	2625	1908	2937	5485	14141	89968	TRUE
		p44	2625	1908	2812	5094	14109	55140	TRUE

表 A.2　完整的 SF1 及其松弛后标准域的验证内存占用性能比较（单位：Mb）

SF	Prop	States	SMCA'	SMCA	COI	Dynamic	BMC(10)
R11	p11	61 ($2^{5.9307}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.420
	p12	82 ($2^{6.3576}$)	<10Mb	<10Mb	<10Mb	<10Mb	13.318
	p2	54 ($2^{5.7549}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.091
	p3	58 ($2^{5.8580}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.388
	p41	52 ($2^{5.7004}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.220
	p42	52 ($2^{5.7004}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.220
	p43	50 ($2^{5.6439}$)	<10Mb	<10Mb	<10Mb	<10Mb	12.313
	p44	50 ($2^{5.6439}$)	<10Mb	<10Mb	<10Mb	<10Mb	13.519
R12	p11	177 ($2^{7.4676}$)	<10Mb	11.250	11.240	10.516	18.856
	p12	333 ($2^{8.3794}$)	11.887	13.548	13.508	12.160	21.888
	p2	255 ($2^{7.9944}$)	<10Mb	11.704	11.628	10.728	20.432
	p3	216 ($2^{7.7549}$)	<10Mb	12.604	13.024	11.500	19.280
	p41	216 ($2^{7.7549}$)	<10Mb	11.784	12.168	10.788	19.316
	p42	294 ($2^{8.1997}$)	11.139	12.484	13.112	11.056	20.882
	p43	255 ($2^{7.9944}$)	<10Mb	12.076	12.300	10.932	20.856
	p44	255 ($2^{7.9944}$)	<10Mb	11.982	12.023	10.782	20.388
R13	p11	395 ($2^{8.6257}$)	16.740	17.620	17.482	17.188	34.80
	p12	395 ($2^{8.6257}$)	16.740	18.452	17.864	17.186	36.724
	p2	395 ($2^{8.6257}$)	16.740	18.772	17.384	16.416	34.780
	p3	395 ($2^{8.6257}$)	16.740	19.064	19.168	18.104	34.951
	p41	395 ($2^{8.6257}$)	16.740	19.372	19.024	17.384	35.036
	p42	395 ($2^{8.6257}$)	16.740	20.904	20.880	19.068	34.846
	p43	395 ($2^{8.6257}$)	16.740	19.188	19.140	17.746	41.376
	p44	395 ($2^{8.6257}$)	16.740	19.137	19.158	17.876	41.771

表 A.2（续）完整的 SF1 及其松弛后标准域的验证内存占用性能比较（单位：Mb）

SF1	p11	1019 ($2^{9.9929}$)	30.592	38.772	35.512	35.044	72.313
	p12	1019 ($2^{9.9929}$)	30.592	38.868	35.821	37.118	71.285
	p2	1019 ($2^{9.9929}$)	30.592	37.122	35.416	34.320	72.268
	p3	1019 ($2^{9.9929}$)	30.592	38.804	35.240	37.428	74.421
	p41	1019 ($2^{9.9929}$)	30.592	38.552	35.204	36.125	73.156
	p42	1019 ($2^{9.9929}$)	30.592	38.736	35.580	37.099	72.054
	p43	1019 ($2^{9.9929}$)	30.592	37.960	35.240	35.508	72.543
	p44	1019 ($2^{9.9929}$)	30.592	37.920	35.288	35.600	72.291

表 A.3　完整的 SF2 及其松弛后标准域的验证时间性能比较（单位：ms）

SF	RA	Prop	SLTS	SMCA’	SMCA	COI	Dynamic	BMC(10)	结果
R21		p1	34187	84453	102860	161188	>10m	574500	TRUE
		p2	30928	55469	56859	56016	>10m	496219	TRUE
		p3	31782	56485	84218	494234	>10m	563328	TRUE
		p41	31719	56187	67625	167016	>10m	546891	TRUE
		p42	32625	63031	67719	217147	>10m	541797	TRUE
		p43	31795	56199	61359	189328	>10m	513985	TRUE
R22		p1	1359	1000	1328	1532	6328	15062	TRUE
		p2	1359	1000	1063	1047	5953	9500	TRUE
		p3	1359	1000	1250	1406	6281	11625	TRUE
		p41	1359	1000	1203	1563	6313	27593	TRUE
		p42	1359	1000	1297	1547	6297	10062	TRUE
		p43	1359	1000	1281	1594	6344	13938	TRUE
SF2	1795	p1	90188	259875	306610	405500	>10m	>10m	TRUE
		p2	90188	259875	265671	264328	>10m	>10m	TRUE
		p3	90188	259875	366281	>10m	>10m	>10m	TRUE
		p41	90188	259875	278734	440219	>10m	>10m	TRUE
		p42	90188	259875	275406	485828	>10m	>10m	TRUE
		p43	90188	259875	268625	500157	>10m	>10m	TRUE

表 A.4　完整的 SF2 及其松弛后标准域的验证内存占用性能比较（单位：Mb）

SF	Prop	States	SMCA’	SMCA	COI	Dynamic	BMC(10)
R21	p1	7797 ($2^{12.9287}$)	183.129	229.608	196.304	171.396	351.334
	p2	7484 ($2^{12.8696}$)	185.394	219.199	185.112	168.868	356.719
	p3	7500 ($2^{12.8727}$)	185.991	222.472	191.600	169.931	336.912
	p41	7500 ($2^{12.8727}$)	186.225	222.960	191.352	169.211	356.683
	p42	7580 ($2^{12.8880}$)	186.225	225.566	193.096	170.770	359.557
	p43	7500 ($2^{12.8727}$)	185.886	222.570	190.304	170.008	352.288
R22	p1	1043 ($2^{10.0265}$)	21.756	26.672	25.520	22.830	40.234
	p2	1043 ($2^{10.0265}$)	21.756	24.152	21.940	18.483	36.684
	p3	1043 ($2^{10.0265}$)	21.756	24.512	24.400	21.692	39.154
	p41	1043 ($2^{10.0265}$)	21.756	25.484	25.126	22.056	41.372
	p42	1043 ($2^{10.0265}$)	21.756	26.689	27.032	22.978	37.661
	p43	1043 ($2^{10.0265}$)	21.756	25.572	25.508	22.516	39.452
SF2	p1	23096 ($2^{14.4954}$)	334.304	403.309	342.692	280.192	504.329
	p2	23096 ($2^{14.4954}$)	334.304	342.102	282.693	276.133	505.171
	p3	23096 ($2^{14.4954}$)	334.304	400.928	344.840	287.530	498.799
	p41	23096 ($2^{14.4954}$)	334.304	400.372	344.670	291.519	510.288
	p42	23096 ($2^{14.4954}$)	334.304	399.876	343.920	285.294	502.561
	p43	23096 ($2^{14.4954}$)	334.304	400.468	342.912	283.787	504.201

表 A.5　完整的 SF3 及其松弛后标准域的验证时间性能比较（单位：ms）

SF	RA	Prop	SLTS	SMCA’	SMCA	COI	Dynamic	BMC(5)	结果
R31		p11	29703	99984	106125	144500	>10m	337388	TRUE
		p12	29797	99435	104226	143183	>10m	331600	TRUE
		p2	28250	98438	99265	98515	>10m	300641	TRUE
		p3	29078	99538	112547	171047	>10m	307750	TRUE
		p41	29078	99886	103982	166138	>10m	298980	TRUE
		p42	30268	99713	102094	159969	>10m	315719	TRUE
		p43	29726	99366	102967	160663	>10m	309836	TRUE
		p44	30543	101478	105123	159887	>10m	312645	TRUE
R32		p11	18797	57719	59319	58360	121562	179984	TRUE
		p12	18797	57719	57949	58651	120927	178390	TRUE
		p2	18797	57719	58812	58175	120390	176906	TRUE
		p3	18797	57719	61135	68344	148578	176891	TRUE
		p41	18797	57719	61131	69890	147578	178844	TRUE
		p42	18797	57719	61694	69797	147106	179516	TRUE
		p43	18797	57719	60287	69813	147563	181109	TRUE
		p44	18797	57719	61284	70123	147206	180396	TRUE
SF3	2213	p11	222172	501672	511640	548250	>10m	>10m	TRUE
		p12	222172	501672	510837	551732	>10m	>10m	TRUE
		p2	222172	501672	504469	497828	>10m	>10m	TRUE
		p3	222172	501672	594984	>10m	>10m	>10m	TRUE
		p41	222172	501672	511141	>10m	>10m	>10m	TRUE
		p42	222172	501672	510323	>10m	>10m	>10m	TRUE
		p43	222172	501672	509981	>10m	>10m	>10m	TRUE
		p44	222172	501672	512393	>10m	>10m	>10m	TRUE

表 A.6　完整的 SF3 及其松弛后标准域的验证内存占用性能比较（单位：Mb）

SF	Prop	States	SMCA'	SMCA	COI	Dynamic	BMC(5)
R31	p11	9805 ($2^{13.2593}$)	233.632	226.924	232.840	236.205	289.086
	p12	9805 ($2^{13.2593}$)	232.875	226.487	232.304	232.781	288.906
	p2	9801 ($2^{13.2587}$)	230.550	225.897	223.559	235.908	279.197
	p3	9805 ($2^{13.2593}$)	271.933	226.584	232.748	236.532	285.911
	p41	9805 ($2^{13.2593}$)	269.583	225.988	231.776	231.440	281.383
	p42	9805 ($2^{13.2593}$)	270.531	226.276	232.504	231.769	279.888
	p43	9805 ($2^{13.2593}$)	269.731	227.267	232.990	230.824	280.909
	p44	9805 ($2^{13.2593}$)	269.731	227.267	231.579	231.776	280.336
R32	p11	8251 ($2^{13.0104}$)	186.894	182.213	183.173	189.600	241.420
	p12	8251 ($2^{13.0104}$)	187.211	182.213	183.516	190.133	240.313
	p2	8251 ($2^{13.0104}$)	186.099	182.213	166.774	158.786	235.076
	p3	8251 ($2^{13.0104}$)	215.884	182.213	186.870	191.640	230.918
	p41	8251 ($2^{13.0104}$)	215.600	182.213	186.887	191.608	233.987
	p42	8251 ($2^{13.0104}$)	218.144	182.213	186.800	192.388	231.888
	p43	8251 ($2^{13.0104}$)	218.736	182.213	186.188	192.591	241.432
	p44	8251 ($2^{13.0104}$)	214.996	182.213	187.332	192.772	239.585
SF3	p11	43073 ($2^{15.3945}$)	445.570	409.970	415.136	339.988	502.955
	p12	43073 ($2^{15.3945}$)	439.776	409.970	414.877	341.672	498.763
	p2	43073 ($2^{15.3945}$)	415.348	409.970	351.604	349.320	499.699
	p3	43073 ($2^{15.3945}$)	488.560	409.970	415.740	347.661	511.074
	p41	43073 ($2^{15.3945}$)	487.896	409.970	415.628	345.546	505.181
	p42	43073 ($2^{15.3945}$)	488.327	409.970	415.869	345.787	503.96
	p43	43073 ($2^{15.3945}$)	487.896	409.970	414.966	345.411	501.700
	p44	43073 ($2^{15.3945}$)	487.896	409.970	416.477	346.215	505.255

附录 B　SF1~SF3 待验证业务逻辑的对应 LTL 公式表

业务逻辑	所属服务流	描述
p11	SF1 / SF3	G (*TmpltBank _ H*1.*Exit* → ((F *TrigBank _ H*1.*Exit*) ∧ (F *Inspiral _ H*1.*Exit*)))
p12	SF1 / SF3	G (*TmpltBank _ H*2.*Exit* → ((F *TrigBank _ H*2.*Exit*) ∧ (F *Inspiral _ H*2.*Exit*)))
p1	SF2	G (*TmpltBank _ L*1.*Exit* → ((F *TrigBank _ H*1.*Exit*) ∧ (F *Inspiral _ H*1.*Exit*)))
p2	SF1 / SF2	G ((*InitData _ H*1*H*2.*Exit*) → ((¬ *Inspiral _ H*2.*Exit* U *sInca _ L*1*H*1.*Exit*) ∧ (¬ *Inspiral _ H*2.*Exit* U *thInca _ L*1*H*1.*Exit*)))
	SF3	G ((*InitData _ H*1*H*2.*Exit*) → (¬ *Inspiral _ H*2.*Exit* U *thInca _ L*1*H*1*H*2.*Exit*))
p3	SF1 / SF2	F *thIncaII _ L*1*H*1.*Exit* ∧ ((F *sInca _ L*1*H*1.*Active* ∧ (¬*thIncaII _ L*1*H*1.*Exit* U *sInca _ L*1*H*1.*Active*)) ∨ (F *thInca _ L*1*H*1.*Active* ∧ (¬*thIncaII _ L*1*H*1.*Exit* U *thInca _ L*1*H*1.*Active*)))
	SF3	F *thIncaII _ L*1*H*1*H*2.*Exit* ∧ ((F *thInca _ L*1*H*1*H*2.*Active* ∧ (¬*thIncaII _ L*1*H*1*H*2.*Exit* U *thInca _ L*1*H*1*H*2.*Active*)) ∨ (F *sInca _ L*1.*Active* ∧ F *sInca _ H*1.*Active* ∧ (¬*thIncaII _ L*1*H*1*H*2.*Exit* U *sInca _ L*1.*Active*) ∧ (¬*thIncaII _ L*1*H*1*H*2.*Exit* U *sInca _ H*1.*Active*)))
p41	SF1 / SF2	G (*Inspiral _ H*1.*Exit* → (F *thIncaII _ L*1*H*1.*Exit*))
	SF3	G (*Inspiral _ H*1.*Exit* → (F *thIncaII _ L*1*H*1*H*2.*Exit*))
p42	SF1 / SF2	G (*Inspiral _ H*2.*Exit* → (F *thIncaII _ L*1*H*1.*Exit*))
	SF3	G (*Inspiral _ H*2.*Exit* → (F *thIncaII _ L*1*H*1*H*2.*Exit*))
p43	SF1	G (*TmpltBank _ H*1.*Exit* → (F *thIncaII _ L*1*H*1.*Exit*))
	SF2	G (*TmpltBank _ L*1.*Exit* → (F *thIncaII _ L*1*H*1.*Exit*))
	SF3	G (*TmpltBank _ H*1.*Exit* → (F *thIncaII _ L*1*H*1*H*2.*Exit*))
p44	SF1	G (*TmpltBank _ H*2.*Exit* → (F *thIncaII _ L*1*H*1.*Exit*))
	SF3	G (*TmpltBank _ H*2.*Exit* → (F *thIncaII _ L*1*H*1*H*2.*Exit*))

附录 C 基于标准域分析的验证策略分解定理证明

证明：要证明性质 5.3 中的$<\varphi>M<\psi>$可以实现推理 5–3，关键要证明$\varphi=\varphi_1 \text{ U } \varphi_2$和$\Psi=\text{G } \varphi_1$的两种情况，因为此处 LTL-X 语义中 W（Weak Until）、R（Release）和 F（Finally）操作均以 U（Until）和 G（Globally）操作为基础而定义，即：有$\varphi_1 \text{ W } \varphi_2 = \text{G } \varphi_1 \vee (\varphi_1 \text{ U } \varphi_2)$、$\varphi_1 \text{ R } \varphi_2 = \text{G } \varphi_2 \vee (\varphi_2 \text{ U } (\varphi_2 \wedge \varphi_1))$和$\text{F } \varphi = TRUE \text{ U } \varphi$。LTL-X 的语义可以定义在一个模型的路径上，也可以定义在一个模型的状态上。在性质 5.3 中，记 $M, \pi \vDash \varphi$表示一 LTL-X 公式φ在一个状态标号迁移系统 M 的一条路经π（一无限状态序列，可通过对序列中的终止状态进行自循环形成）上被满足；记 $M, \sigma \vDash \varphi$表示一个 LTL-X 公式φ在一个状态标号迁移系统 M 的一状态σ上被满足，当对于该标号迁移系统中所有以σ为初始状态的路径，有 $M, \pi_i \vDash \varphi$成立；此外，记π^k表示以路径π中第 k 个状态为初始的子路径，则：

- 若$\varphi=\varphi_1 \text{ U } \varphi_2$，由于已知$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$可得：$\forall \pi' = \sigma_i, \sigma'_1, \sigma'_2, \ldots \sigma'_p, \sigma'_p, \ldots \in TransSys(M_{i+1};M_{i+2};...;M_n, \mathbb{S}(M_{i+1};...;M_n))$，$\exists k \geq 0$，s.t. $(M_{i+1};M_{i+2};...;M_n), \pi'^k \vDash \varphi_2$且对于所有的 $0 \leq i < k$，$(M_{i+1};M_{i+2};...;M_n), \pi'^i \vDash \varphi_1$。因此对于 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$中任意以$\mathbb{S}(M_i;...;M_n)$为初始的状态序列$\pi=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots \sigma_k$：
  - 由于（1）$\exists i \in N^+$，s.t. $\sigma_i \in \pi$，$\sigma_i \in \mathbb{S}(M_{i+1};...;M_n)$，且$\mathbb{E}(M_i) = \mathbb{S}(M_{i+1};M_{i+2};...;M_n) \neq$Empty；（2）从$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$可知 $Trans(M_{i+1};M_{i+2};...;M_n, \mathbb{S}(M_{i+1};...;M_n))$的可接受执行严格包含在 $Trans(\varphi_1 \text{ U } \varphi_2)$中，因此对于以上任一$\pi'$，总存在对应的$\pi' \in Trans(\varphi_1 \text{ U } \varphi_2) = \sigma^0, \sigma^1, \sigma^2, \ldots$，s.t. $\forall \pi'' = \mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma^1, \sigma^2, \ldots$，由已知 $M_i, \pi' \vDash \varphi_1 \text{ U } \varphi_2$可得对应地$\forall \pi''' = \mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma'_1, \sigma'_2, \ldots \sigma'_p, \sigma'_p \ldots$时，亦有 $M_i, \pi'' \vDash \varphi_1 \text{ U } \varphi_2$成立。所以$\forall \sigma \in \mathbb{S}(M_i;...;M_n)$，$(M_i;...;M_n), \sigma \vDash \varphi_1 \text{ U } \varphi_2$成立。
  - 当 $M_i, \pi^* \vDash \varphi_1 \text{ U } \varphi_2$，因为 $TransSys(M_i, \mathbb{S}(M_i;...;M_n)) \supseteq \mathbb{E}(M_i) = \mathbb{S}(M_{i+1};M_{i+2};...;M_n) \neq$Empty，则在 $TransSys(M_i;...;M_n, \mathbb{S}(M_i;...;M_n))$中（由于$\mathbb{S}(M_i;...;M_n)$本身就是 $M_i;...;M_n$ 的域初始状态集合），$\forall \sigma \in \mathbb{S}(M_i;...;M_n)$，$(M_i;...;M_n), \sigma \vDash \varphi_1 \text{ U } \varphi_2$直接成立；
- 对于$\varphi=\text{G } \varphi$的讨论与$\varphi_1 \text{ U } \varphi_2$类似。若$\varphi=\text{G } \varphi$，由于已知

$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$ 可得：$\forall\pi' = \sigma_i, \sigma'_1, \sigma'_2, \ldots\sigma'_p, \sigma'_p, \ldots\in Trans(M_{i+1};M_{i+2};...;M_n, \mathbb{S}(M_{i+1};...;M_n))$，对于所有的 $i\geq 0$，$(M_{i+1};M_{i+2};...;M_n), \pi' {}^{i}\vDash\varphi$。因此对于 $TransSys(M_i, \mathbb{S}(M_i;...;M_n))$中任意以$\mathbb{S}(M_i;...;M_n)$为初始的状态序列$\pi=\mathbb{S}(M_i;...;M_n), \sigma_1, \sigma_2, \ldots\sigma_k$：

- 由于（1）$\exists i\in N^+$，s.t. $\sigma_i\in\pi$，$\sigma_i\in\mathbb{S}(M_{i+1};...;M_n)$，且$\mathbb{E}(M_i) = \mathbb{S}(M_{i+1};M_{i+2};...;M_n)\neq$Empty；（2）从$<TRUE>M_{i+1};M_{i+2};...;M_n<\varphi>$可知 $Trans(M_{i+1};M_{i+2};...;M_n, \mathbb{S}(M_{i+1};...;M_n))$的可接受执行严格包含在 $Trans(\mathrm{G}\ \varphi)$中，因此对于以上任一$\pi'$，总存在对应的$\pi'\in Trans(\mathrm{G}\ \varphi) = \sigma^0, \sigma^1, \sigma^2, \ldots$, s.t. $\forall\ \pi'' = \mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma^1, \sigma^2, \ldots$，由已知 $M_i, \pi'\vDash \mathrm{G}\ \varphi$可得对应地$\forall\ \pi''' = \mathbb{S}(M_i;...;M_n), \sigma_1, ..., \sigma_i, \sigma'_1, \sigma'_2, \ldots\sigma'_p, \sigma'_p \ldots$时，亦有 $M_i, \pi''\vDash \mathrm{G}\ \varphi$成立。所以$\forall\sigma\in\mathbb{S}(M_i;...;M_n)$，$(M_i;...;M_n), \sigma\vDash \mathrm{G}\ \varphi$成立。
- 当 $M_i, \pi^* \vDash \mathrm{G}\ \varphi$，因为 $TransSys(M_i, \mathbb{S}(M_i;...;M_n)) \supseteq \mathbb{E}(M_i) = \mathbb{S}(M_{i+1};M_{i+2};...;M_n)\neq$Empty，则在 $TransSys(M_i;...;M_n, \mathbb{S}(M_i;...;M_n))$中（由于$\mathbb{S}(M_i;...;M_n)$本身就是 $M_i;...;M_n$ 的域初始状态集合），$\forall\sigma\in\mathbb{S}(M_i;...;M_n)$，$(M_i;...;M_n), \sigma\vDash \mathrm{G}\ \varphi$直接成立；

证毕。 □

在以上性质 5. 3 及其证明中，公式Ψ的规范化定义在以下 LTL 时序逻辑的标准等价关系上：

- 转换：

  $\mathrm{F}\ \varphi_1 = \mathrm{TRUE\ U}\ \varphi_1$　　$\mathrm{G}\ \varphi_1 = \mathrm{FALSE\ R}\ \varphi_1$

  $\varphi_1\ \mathrm{W}\ \varphi_2 = \mathrm{G}\ \varphi_1 \vee (\varphi_1\ \mathrm{U}\ \varphi_2)$　　$\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$
- 取反：

  $\neg(\varphi_1\ \mathrm{R}\ \varphi_2) = \neg\varphi_1\ \mathrm{U}\neg\varphi_2$　　$\neg(\mathrm{F}\ \varphi_1) = \mathrm{G}\neg\varphi_1$　　$\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$
- 分解：

  $\mathrm{G}\ (\varphi_1 \wedge \varphi_2) = \mathrm{G}\ \varphi_1 \wedge \mathrm{G}\ \varphi_2$　　$\mathrm{F}\ (\varphi_1 \vee \varphi_2) = \mathrm{F}\ \varphi_1 \vee \mathrm{F}\ \varphi_2$

个人简历、在学期间发表的学术论文与研究成果

个人简历

1980 年 3 月 28 日出生于上海市宝山区。

1998 年 9 月考入上海交通大学自动化系自动化专业，2002 年 7 月本科毕业并获得工学学士学位。

2002 年 9 月免试进入清华大学自动化系攻读工学博士至今。

在学期间发表的学术论文

[1] **Xu K**[①], Samuel M, Liu Y. A Static Compliance Checking Framework for Business Process Models. IBM Systems Journal, 2007, 46(2): 335-362 （SCI 源刊　139HN　影响因子 1.255）

[2] **Xu K**, Liu L C, Wu C. A Three Layered Method for Business Process Discovery and its Application. Computers in Industry, 2007, 58(3): 265-278 （SCI 源刊　141MN　影响因子 0.935）

[3] **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 F 辑: 信息科学, 2007, 37(4): 467-485. （SCI 源刊　134SF　影响因子 0.386）

Xu K, Wang Y X, Wu C. Formal Verification Technique for Grid Service Chain Model and its Application. Science in China, Series F: Information Sciences, 2007, 50(1): 1-20　（SCI 源刊　134SF　影响因子 0.386）

[4] **Xu K**, Wang Y X, Wu C. Ensuring Secure and Robust Grid Applications – From a Formal Method Point of View. Advances in Grid and Pervasive Computing, Lecture Notes in Computer Science, 2006, 3947: 537-546 （SCI 检索　BEK24: 000237540300053　影响因子 0.402）

[5] **Xu K**, Liu Y, Zhu J, Wu C. Pi Calculus based Bi-transformation of State-driven Model and Flow-driven Model. International Journal of Business Process Integration and Management, 2006, 1(4): 292 - 306

[6] **许可**, 刘连臣, 吴澄. 时间π演算及其弱时间互模拟分析. 计算机集成制造系统–CIMS, 2006, 12(4): 511-516　（EI 检索 06279979883）

① 三人并列为第一作者，按姓氏字母排序。可参见该论文中的声明。

[7] **Xu K**, Liu Y, Wu C. Guided Reasoning of Complex E-Business Processes with Business Bug Patterns. In: International Conference on E-Business Engineering, IEEE Computer Society, 2006: 195-202 （Ei 源刊）

[8] **Xu K**, Wang Y X, Wu C. Aspect Oriented Region Analysis for Efficient Equipment Grid Application Reasoning. In: 5th International Conference on Grid and Cooperative Computing (GCC 2006), IEEE Computer Society, 2006: 28-31 （Ei 源刊）

[9] **Xu K**, Liu Y, Wu C. BPSL Modeler - Visual Notation Language for Intuitive Business Property Reasoning. Graph Transformation and Visual Modelling Techniques, To appear in: Electronic Notes in Theoretical Computer Science (ENTCS), 2006: 205-214

[10] **Xu K**, Wang Y X, Wu C. Service Provenance based Abstraction of Grid Application Knowledge, In: International Conference on Semantic and Knowledge Grid, IEEE Computer Society, 2006

[11] Wang Y X, Wu C, **Xu K**. Study on Pi Calculus Based Equipment Grid Service Chain Model. Network and Parallel Computing, Lecture Notes in Computer Science, 2005, 3779: 40-47 （SCI 检索 BDQ15: 000234857500006 影响因子 0.402）

在学期间的研究成果

在学期间**个人获奖情况**主要包括：

[1] 2006—2007 年度 IBM Ph.D. Fellow（大中华区仅 7 人）

[2] 2006 年度清华大学自动化系学术新秀

[3] 被邀担任 IEEE Transactions on Systems, Man and Cybernetics 审稿人

[4] 2006 年度清华大学北电网络优秀学生奖

在学期间参加的**科研项目**主要包括：

[1] 国家教育部 211 工程项目“全国高校仪器设备与优质资源共享系统”；编号 CERS-219899004

[2] 国家自然科学基金项目“面向可溯源的设备网格服务链模型推理和验证研究”；编号 60604033