

# 中国科学技术大学

# 硕士学位论文



## 面向分布式细粒度一致性模型 的编程语言设计与实现

作者姓名：王佳玮

学科专业：计算机软件与理论

导师姓名：李诚 特任研究员，冯新宇 教授

完成时间：二〇二〇年六月二十九日



University of Science and Technology of China  
A dissertation for master's degree



**A Language for Fine-Grained  
Consistency in Distributed Systems:  
Design and Implementation**

Author: Wang Jiawei

Speciality: Computer Software and Theory

Supervisors: Prof. Xinyu Feng, Prof. Cheng Li

Finished time: June 29, 2020



## 中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：\_\_\_\_\_

签字日期：\_\_\_\_\_

## 中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

公开  保密（\_\_\_\_年）

作者签名：\_\_\_\_\_

导师签名：\_\_\_\_\_

签字日期：\_\_\_\_\_

签字日期：\_\_\_\_\_



## 摘要

数据备份系统是互联网服务的基本组成单元之一，是提供高可用服务的重要保障。随着服务规模的不断扩大，为了能够快速响应不同地区用户的请求，许多互联网服务提供商将数据副本放置到靠近用户的数据中心。然而，在跨地域的数据中心之间进行备份强同步会造成用户请求时延的陡然上升。为解决这一冲突，近年来，一些弱一致性模型相继被提出并被使用在上述场景中。虽然弱一致性模型具有性能优势，但却不能时刻保证上层应用的正确性。因此，结合真实场景中应用的客观需求，研究人们开始关注混合一致性模型的研究和应用。混合一致性模型可以允许互联网服务的设计者在一致性、系统性能、服务正确性等方面做出权衡，同时也为设计者们带来了编程的负担和挑战。这是因为较弱的一致性模型提供的语义与强一致性的语义不同，编程人员需要确保在强同步缺省的情况下，服务并行执行仍然能够满足应用的语义需求（例如满足状态收敛性或满足特定不变量）。

为了解决以上问题，首先，本文提出了一个细粒度一致性模型和相应的编程语言，并给出了相应的操作语义。新的语言兼顾了实用性和表达能力，可使编程人员表达出其想要的一致性语义，从而避免（1）因为遗漏重要的一致性约束而导致的错误；和（2）由于引入不必要的一致性约束所产生的程序效率损失。其次，本文给出了细粒度一致性模型的相关性质，便于编程人员对其有更深刻的理解。而后，本文给出用于验证程序满足收敛性和特定不变量的程序逻辑，并给出了程序逻辑的可靠性证明。相关证明的关键引理已在定理证明工具 Coq 中实现，共计约 3100 行。最后，本文给出了基于上述一致性模型和语言的分布式备份系统设计和实现。系统使用 Java 语言实现，共计约 5300 行代码。除此之外，为保证系统正确性，我们使用了 TLA+ 模型检查工具对其进行死锁和活性检查，共计约 1500 行代码。最后，我们利用该语言实现 3 个应用程序，并在不同工作负载下进行对比实验。实验结果表明了语言的通用性和系统的可扩展性。

**关键词：**分布式系统；一致性模型；细粒度一致性；程序逻辑；收敛性；不变量

## ABSTRACT

Modern large-scale Internet services rely on distributed data systems that maintain multiple copies of data. As the scale of the Internet continues to expand, to quickly respond user's request in different regions, Many Internet service providers replicate data across multiple geographically dispersed data centers and provide a weak consistency guarantee to reduce user-perceived latency. Although the weak consistency model is sufficient to meet the needs of certain application scenarios. But it still has limitations. Some applications may require stronger consistency. In this case, the distributed system needs to support multiple consistency at the same time to provide different consistency guarantees for data with different consistency requirements. Since the semantics provided by the weak consistency model and strong consistency model are different, Programmers need to determine whether their program can meet their semantic requirements by running under different consistency models, such as invariant violation or state convergence. This brings new challenges to distributed system programming.

To solve the above problems, firstly, this paper proposes a fine-grained consistency model and the corresponding programming language. The new language takes into account both practicality and expressive ability, allowing programmers to express the consistency semantics they expect and avoiding efficiency loss. Secondly, this paper presents some properties of the fine-grained consistency model for programmers to have a deeper understanding. Also, by using different synchronization sets and dependency sets in operations, we can make the server meet one of the existing data-centric consistency. Thirdly, this paper presents program logics for verifying the convergence and invariant preservation of a given program. We also give the soundness proof of these logics. The key lemma of the proof has been implemented in the proof assistant Coq, with a total of 3100 lines of proof scripts. Finally, this paper presents the design and implementation of a fine-grained consistency distributed system based on the above consistency model and programming language. The system was checked in the TLA + model checking tool, with a total of 1500 lines of code. The system is implemented in Java and has a total of approximately 5,300 lines of code. The experimental results under different workloads show the reliability and scalability of the system.

**Key Words:** Distributed System; Consistency Model; Fine-Grained Consistency; Program Logic; Convergence; Invariant

## 目 录

第 1 章 绪论 ······	1
1.1 研究动机 ······	1
1.2 国内外相关工作 ······	3
1.2.1 一致性模型 ······	3
1.2.2 基于混合一致性模型的数据备份系统 ······	3
1.2.3 弱内存模型和并发程序逻辑 ······	3
1.2.4 面向分布式混合一致性模型的编程语言 ······	4
1.3 主要贡献 ······	4
1.3.1 细粒度一致性模型及语言设计 ······	4
1.3.2 与一致性相关的性质及证明 ······	5
1.3.3 与程序语义相关的性质及证明 ······	5
1.3.4 细粒度一致性模型的系统实现 ······	5
1.4 论文组织结构 ······	5
第 2 章 背景介绍 ······	7
2.1 系统模型 ······	7
2.1.1 运行流程和历史轨迹 ······	7
2.1.2 历史轨迹示例 ······	8
2.2 一个启发性的实例 ······	9
2.2.1 同步和依赖 ······	10
2.2.2 明确操作的副作用 ······	11
2.3 本章小结 ······	12
第 3 章 细粒度一致性语言设计 ······	13
3.1 语法的定义 ······	13
3.2 语言的应用实例 ······	13
3.2.1 银行系统 ······	13
3.2.2 add-win 集合 ······	15
3.2.3 拍卖系统 ······	17
3.3 操作语义的定义 ······	18
3.3.1 客户端状态转换规则 ······	21
3.3.2 服务器端状态转换规则 ······	22

## 目 录

---

3.4 语言的性质 ······	24
3.4.1 操作同步和操作依赖···	26
3.4.2 仲裁顺序 ······	26
3.4.3 同步关系、依赖关系和程序顺序···	26
3.4.4 一致性 ······	27
3.4.5 示例：银行系统同时取钱 ······	27
3.5 ······	28
3.5.1 因果一致性 ······	28
3.5.2 基于变量的因果一致性 ······	29
3.5.3 顺序一致性 ······	29
3.5.4 基于变量的顺序一致性 ······	29
3.6 本章小结 ······	30
<b>第 4 章 程序逻辑与验证 ······</b>	<b>31</b>
4.1 验证程序满足收敛性的程序逻辑 ······	31
4.1.1 收敛性和可交换性 ······	31
4.1.2 稳定的存储状态 ······	33
4.1.3 独一的请求 ······	34
4.1.4 示例 1：验证拍卖系统的收敛性 ······	35
4.1.5 示例 2：验证 add-win 集合的收敛性 ······	36
4.2 验证程序满足特定不变量的程序逻辑 ······	36
4.2.1 不变量 ······	36
4.2.2 独立执行 ······	38
4.2.3 独立接受 ······	39
4.2.4 示例：银行系统账户余额不小于 0 的条件 ······	39
4.3 使用 Coq 验证程序逻辑的正确性 ······	41
4.3.1 Coq 简介 ······	41
4.3.2 使用 Coq 验证收敛性和不变量程序逻辑的可靠性 ······	42
4.4 本章小结 ······	43
<b>第 5 章 细粒度一致性协议与系统的设计和实现 ······</b>	<b>44</b>
5.1 服务器端系统设计的几个关键问题 ······	44
5.1.1 如何模拟全局消息缓冲区? ······	44
5.1.2 如何决定全局历史记录和仲裁顺序? ······	45
5.1.3 如何进行垃圾回收? ······	46
5.2 服务器端系统运行流程 ······	48

## 目 录

---

5.3 使用 TLA+ 验证服务器端实现的正确性 ······	51
5.3.1 TLA+ 简介 ······	51
5.3.2 使用 PlusCal 算法语言对服务器端建模 ······	53
5.3.3 使用 TLC 工具检查服务器的死锁和活性 ······	53
5.4 其他实现细节 ······	54
5.5 本章小节 ······	55
<b>第 6 章 实验与分析 ······</b>	<b>56</b>
6.1 实验环境 ······	56
6.1.1 站点配置 ······	56
6.1.2 工作负载 ······	56
6.1.3 性能评估指标 ······	56
6.2 实验结果和结果分析 ······	57
6.2.1 三个应用实例运行时的吞吐量和延迟 ······	57
6.2.2 不同的一致性保证对延迟的影响 ······	58
6.2.3 公平性对延迟的影响 ······	59
6.3 本章小结 ······	60
<b>第 7 章 结论 ······</b>	<b>61</b>
7.1 本文工作总结 ······	61
7.2 进一步工作 ······	61
<b>参考文献 ······</b>	<b>62</b>
<b>附录 A 定理 3.1 的证明 ······</b>	<b>68</b>
A.1 引理 A.1 的证明 ······	68
A.2 引理 A.2 的证明 ······	70
A.3 其他引理的证明 ······	75
A.4 命题 3.2 的证明 ······	81
<b>附录 B 命题 3.3、3.4、3.5 和 3.6 的证明 ······</b>	<b>82</b>
<b>附录 C 定理 4.1 的证明 ······</b>	<b>85</b>
<b>附录 D 定理 4.2 的证明 ······</b>	<b>89</b>
<b>附录 E 系统模型 ······</b>	<b>96</b>
<b>致谢 ······</b>	<b>106</b>
<b>在读期间发表的学术论文与取得的研究成果 ······</b>	<b>107</b>



## 插 图 清 单

2.1 系统模型、请求状态转换和运行轨迹示例 ······	8
2.2 银行系统代码和相应若干运行轨迹 ······	9
3.1 程序语言语法的定义 ······	14
3.2 语言的应用实例 ······	15
3.3 add-win 集合的若干历史轨迹图 ······	16
3.4 拍卖系统的若干历史轨迹图 ······	18
3.5 操作语义规则 (一) ······	19
3.6 操作语义规则 (二) ······	20
3.7 定义操作语义规则的一些辅助定义 ······	21
3.8 客户端和服务器端的状态模型 ······	22
3.9 语言性质中的一些辅助定义 ······	24
3.10 定义不同一致性的一些辅助定义 ······	28
4.1 收敛性程序逻辑的一些辅助定义 ······	32
4.2 证明库满足收敛性的逻辑规则 ······	33
4.3 稳定的和唯一的断言含义示意图 ······	33
4.4 证明库满足不变量的逻辑规则 ······	37
4.5 验证不变量的程序逻辑的一些辅助定义 ······	37
4.6 不变量各个逻辑规则所对应的情况 ······	38
4.7 通过等价变换将乱序执行的轨迹转换为串行执行的轨迹 ······	40
4.8 在 Coq 中定义自然数 ······	41
4.9 在 Coq 中定义自然数前驱 ······	42
4.10 在 Coq 中定义自然数相加 ······	42
4.11 在 Coq 中证明 0 小于等于 2 ······	42
4.12 在 Coq 中给出程序逻辑的可靠性定理 ······	43
5.1 对称性事件对仲裁顺序的影响 ······	45
5.2 不恰当的请求回收对系统产生的影响 ······	47
5.3 服务器端系统运行中每个请求的生命周期 ······	49
5.4 服务器端系统运行中的状态转换和消息传递 ······	50
5.5 PlusCal 语言代码实例 ······	52
6.1 系统运行各个实例时的吞吐量和延迟 ······	57

---

## 插 图 清 单

6.2 系统运行不同一致性保证下的各个实例时的延迟	58
6.3 系统运行不同实例时各个站点的延迟	59
A.1 证明一致性的一些辅助定义	69
C.1 证明收敛性程序逻辑可靠性的一些辅助定义	86
D.1 证明不变量程序逻辑可靠性的一些辅助定义	90

## 表 格 清 单

3.1 操作之间的关系和一致性的定义	25
5.1 服务器间通信的消息类型和各类型消息的作用	49
5.2 TLC 模型检查器在不同参数下的检测结果和运行时长	54
6.1 应用实例运行时每个操作所占比例	57



# 第1章 绪论

## 1.1 研究动机

数据备份系统是互联网服务的基本组成单元之一，是提供高可用服务的重要保障。传统的数据备份系统为应用程序提供强一致的存储服务，保证应用程序始终可以看到单一的、最新的数据值<sup>[1-2]</sup>。随着服务规模的不断扩大，为了能够快速响应不同地区用户的请求，互联网服务提供商一般会维护多个数据副本并将其分布于不同的地理位置，同时用户请求将被转发到附近或负载最小的副本服务器。远距离通讯会有较高的延迟，跨数据中心通信的成本比数据中心内的通信成本大两到三个数量级，这导致强一致性数据备份系统在跨数据中心的场景下同步成本过于昂贵<sup>[3]</sup>。

为降低同步成本，目前跨数据中心的备份系统大多提供了较弱的一致性保证，如因果一致性、最终一致性等<sup>[4-12]</sup>。在这种情况下，靠近用户的副本服务器可以直接处理用户的请求，而不需要实时与其他副本服务器同步。这些操作是异步执行的，因此系统能够获得更好的性能和可用性。在这样的备份系统中，客户端可以被返回陈旧数据。当执行读取操作的副本尚未接收到某些副本的写入操作时，读取操作返回的数据是过去某个时间点的值，不一定是最新值；写入操作则被允许在多个副本上同时进行，最后再进行合并。这种技术受到延迟敏感服务的青睐，例如即时通讯、社交网络和在线购物等，因此广泛应用于跨数据中心的备份系统中。

虽然较弱的一致性模型能够满足某些应用场景的需求，但仍然存在一定的局限性。某些应用中数据可能需要更强的一致性，如在银行系统中，对于同一个账户的取钱操作之间需要同步来保证账户余额不小于所有取钱的数额。此时需要数据备份系统支持多级别的一致性，为不同需求的应用分配不同一致性的数据备份系统。由于应用的一致性是由语义最强的操作所决定的，为其分配较强一致性的数据备份系统时，所有操作的语义都必须提升到更强的一致性，从而为只需要弱一致性的操作引入不必要的同步和延迟。除此之外，由于较弱的一致性模型提供的语义与强一致性的语义不同，编程人员需要确定在不同的一致性模型下运行的应用程序能够满足其语义需求，例如保证不变量和不产生状态分歧。这为跨数据中心备份系统的编程带来了新的挑战<sup>[13-17]</sup>。

近年来，基于混合一致性模型的数据备份系统不断涌现<sup>[18-20]</sup>。这些系统允许操作在弱一致性或强一致性下运行，能够在单一系统中支持多种一致性。编程人员在编写混合一致性模型数据备份系统的应用程序时，需要定义操作集合中每两个操作之间的相互关系。此后，系统会根据此关系来决定操作之间是否需要

同步。基于混合一致性模型的数据备份系统在系统性能和语义需求之间实现了很好的平衡。一方面，如果应用中需要同步的操作执行频率很低，则系统的平均响应速度会和弱一致性系统相接近，这种低通信成本的优势会在跨数据中心的场景中更为突出。另一方面，基于混合一致性模型的数据备份系统也能够在基本不影响弱一致性操作性能的情况下，为应用中需要强一致性的操作提供相应的语义需求。

虽然基于混合一致性模型的数据备份系统能够在保证语义的情况下，在特定的应用中比传统的数据备份系统性能更高，但仍不能解决弱一致性模型为编程人员带来的挑战。因此，在目前混合一致性模型数据备份系统中，操作之间的相互关系只有因果关系和强同步关系两种。目前已有工具能够为编程人员根据不变量等语义要求自动生成操作之间的相互关系<sup>[21]</sup>，但仍不能为编程人员提供清晰的语义。

除此之外，在现有的基于混合一致性模型的数据备份系统中，操作之间的两种关系（因果关系和强同步关系）并不能够完全满足用户的需求。一方面，在现有的基于混合一致性模型的数据备份系统中，最弱的一致性保证为因果一致性。因此，在某些不需要因果相关的应用中，这种一致性语义依然很强；另一方面，某些应用可能会有更细粒度的语义（如介于因果一致性语义和最终一致性语义之间），而这些应用在现有的基于混合一致性模型的数据备份系统中会按照更强一致性的语义去执行，故仍会有冗余的同步行为，降低了系统的性能和可用性。

总之，现有的基于混合一致性模型的数据备份系统仍然存在以下几点问题：(1) 编程接口复杂，没有抽象的编程模型，不能为编程人员提供清晰的语义；(2) 编程接口的复杂制约着能提供的接口种类，目前系统中最弱的一致性保证为因果一致性，不支持更弱的最终一致性；(3) 目前的系统不支持更细粒度的语义，降低了系统的表达能力、性能和可用性。

因此，为了解决以上几点问题，我们需要：(1) 给出新的一致性模型及其相应的编程语言和操作语义，新的模型能够支持更细粒度的语义，并且能够支持最终一致性；(2) 通过形式化的方法保证使用此语言编写的应用能够满足与一致性有关的一些性质（如操作的顺序保证等）；(3) 通过验证的方法保证使用此语言编写的应用能够满足与程序语义相关的性质（如不变量，系统状态收敛等）；(4) 给出上述细粒度一致性模型的一种可行的实现，并将其应用在数据备份系统中，提高现有系统的性能。

## 1.2 国内外相关工作

### 1.2.1 一致性模型

Doug Terry 通过使用棒球比赛期间不同的参与者（记分员，裁判员，体育记者等）对于当前分数的不同需求进行类比，展示了应用中不同终端对一致性的不同需求<sup>[22]</sup>。Andrea Cerone 等人提出了一个用于统一的用声明的形式为事务指定各种一致性模型的框架<sup>[23]</sup>。Sebastian Burckhardt 等人提出了一种基于最终一致事务的一致性模型<sup>[24]</sup>。在这个模型中，事务按照两个操作的某些关系进行排序。李诚等人提出了 RedBlue 和 PoR 等混合一致性模型<sup>[18-19]</sup>，Alexey Gotsman 等人提出了相似的混合一致性模型<sup>[25]</sup>。这些模型最弱为因果一致性，不支持更弱一致性的语义。因此，我们需要一个新的致性模型，其能够支持更细粒度的语义并且能够支持最终一致性。

### 1.2.2 基于混合一致性模型的数据备份系统

在过去的几十年中，许多的支持多级别的一致性的数据备份系统被提出，致力于减少并发操作之间的同步，以提高跨数据中心的备份系统的性能<sup>[2,26-37]</sup>。但是，它们只允许程序从他们支持的有限数量的一致性级别中进行选择，例如强一致性，因果一致性或最终的一致性。李诚等人实现了支持混合一致性模型的多副本备份系统，在系统性能和语义需求方面做了很好的平衡<sup>[18]</sup>。但目前的系统并不支持更细粒度的语义，从而降低了系统的表达能力和可用性。

### 1.2.3 弱内存模型和并发程序逻辑

现代的共享内存的多处理器和编程语言提供了弱内存模型，在实际应用中允许程序员根据使用内存栅栏等语句加强一致性<sup>[38-40]</sup>。一些论文使用模型检查器和抽象解释器验证了在弱内存模型下运行的应用的正确性<sup>[41-49]</sup>。虽然在弱内存模型上运行的应用程序的程序逻辑和弱一致性的分布式系统上运行的应用程序之间有很多的相似之处，但还是有一些区别。具体体现在（1）弱内存模型是多个缓存单元和单一内存的存储结构，最终缓存单元的数据会同步到内存中。而多副本备份系统中所有的存储单元具有同等地位，所以可能会出现状态不一致的情况；（2）弱内存模型上的语句大部分由读写变量的语句组成的，这些语句之间不具有交换性，而多副本备份系统中被广播的事件是由操作实例组成的，其中的操作大部分具有交换性，这会为语言带来一些新的性质。

### 1.2.4 面向分布式混合一致性模型的编程语言

Christopher Meiklejohn 等人提出基于 CRDT<sup>[50]</sup> 的编程模型 Lasp<sup>[51]</sup>。Sivararamakrishnan 等人给出了一个最终一致性语义下数据存储的声明式编程的编程模型<sup>[52]</sup>，能够指定细粒度的应用程序级一致性属性。但其编程接口仍然需要声明操作数量平方量级的关系。Nosheen Zaza, Alessandro Margara, Matthew Milano 等人分别给出了混合一致性模型下的编程语言<sup>[20,53-54]</sup>，但其并不能表达细粒度的语义。除此之外，他们将一致性语义与对象进行绑定，即声明一个对象的同时需要给出其一致性语义，之后在这个变量上的所有操作均按照此一致性语义执行。大量的应用<sup>[22]</sup> 表明，将对象绑定到固定的一致性语义并不能完全满足应用的语义需求。因此，我们需要新的编程语言支持对同一变量不同一致性级别的操作，从而能够提供更加丰富的语义支持。

## 1.3 主要贡献

### 1.3.1 细粒度一致性模型及语言设计

目前的基于混合一致性模型的数据备份系统中，需要编程人员定义操作集合中每两个操作之间的相互关系。在操作较多的复杂应用中，这种平方级数量的定义给编程人员带来极大的挑战。我们观察到大多数的应用开发中均使用面向对象的软件开发方法，在这些应用程序中对象与作用在对象上面的操作是一对多的关系。所以，我们将原来的定义操作集合中的相互关系，转化为对象之间的关系。由于对象状态是由其他操作所决定，我们可以通过操作中对象状态的要求间接的定义此操作与其他操作之间的关系。因此，我们在不牺牲一致性语义的表达能力的前提下，大大简化了编程的接口。我们在原有的操作单一数据副本的语言中增加若干个原语，新的语言支持包括最终一致性在内的更细粒度的语义。我们的一致性模型足够通用，能够表示强一致性、最终一致性、因果一致性、RedBlue 一致性、PoR 一致性等多种一致性的语义，同时编程接口又足够简洁。

### 1.3.2 与一致性相关的性质及证明

在描述系统满足的一致性语义时，通常使用事件之间的关系来刻画（事件指操作的实例）<sup>[55-56]</sup>。如强一致性要求事件在所有副本上都是按照相同的顺序执行的，RedBlue 一致性要求所有标记为 ‘red’ 的操作在所有节点按照相同的顺序执行，而对标记为 ‘blue’ 操作的序关系没有特别要求。类似的，我们通过收集在抽象机器上运行事件的历史记录，通过形式化的方法验证这些历史记录满足与序关系相关的性质，从而能够保证使用此语言编写的应用在运行时的事件满足

一定的执行顺序关系。

### 1.3.3 与程序语义相关的性质及证明

除了保证应用在运行时事件需要满足一定的执行顺序关系外，我们还关注一些与特定应用语义相关的一些性质。如银行系统中，所有副本上的账户的余额在任何时刻不应该为负数，而且系统的状态应该为收敛的（即系统停止运行后，所有副本的值应该相同）。我们给出用于推理运行在细粒度一致性模型下的程序的方法，使得我们能够通过形式化验证的方法保证使用此语言编写的应用能够满足与应用语义相关的性质。

### 1.3.4 细粒度一致性的系统实现

在基于混合一致性模型的数据备份系统中，使用了令牌和 Lamport 时钟<sup>[57]</sup>分别保证了与强一致性和因果一致性相关的语义。在我们的细粒度一致性模型中，最弱的一致性被弱化为最终一致性，所以我们需要替换掉保证因果一致性的 Lamport 时钟。在我们的实现中，我们通过操作的历史记录来确定操作之间的序关系，不同的序关系代表了不同的一致性保证。为了保证系统实现的正确性，我们使用 TLA+ 模型检测工具对系统进行建模以及死锁和活性检查。

## 1.4 论文组织结构

- 第二章将给出系统模型的相关介绍，并以一个简单的例子引出一致性模型和语言的设计动机。
- 第三章将给面向分布式细粒度一致性模型编程语言的语法和操作语义、语言的性质以及用此语言可得到的多种一致性保证。
- 第四章将给出用于证明程序收敛性和满足特定不变量的程序逻辑。
- 第五章将给出系统设计思路和实现细节。
- 第六章将给出评估系统性能的实验结果和结果分析。

## 第2章 背景介绍

### 2.1 系统模型

在工业界中，互联网应用程序通常采用多层体系结构<sup>[58-59]</sup>。因此，如图 2.1a 所示，我们假定整个系统分为三层，即客户端，逻辑服务器以及多副本数据库。

**客户端应用程序** 客户端应用程序代表终端用户，使用逻辑服务器为用户提供的函数库来发出请求（即函数调用），该请求将被发送到服务器，通过服务器后端的数据库来执行对数据的操作。当服务器将请求结果返回时，客户端将此结果展示给用户。客户端可以是互联网浏览器或其他终端。

**逻辑服务器** 作为客户端和数据库之间的接口，服务器根据用户请求所调用的函数，通过与数据库进行交互和执行事务的方式来做出逻辑决策并执行相应计算。服务器将放置在多个站点上并与客户端保持密切联系。逻辑服务器包括两部分，即协调系统和函数库。协调系统用于确保客户端用户发出读取或写入数据库请求时，不同客户端提交的请求之间满足特定的顺序。函数库是由若干函数（或操作）组成的集合，为用户提供函数调用接口。

**多副本数据库** 终端用户的数据会存储在数据库中，并从数据库中进行检索。为了确保容错能力、高可用性和高性能，数据通常会在若干个跨地理位置的数据中心中进行备份。多副本数据库需要额外的协调来保证副本之间数据的一致性。

#### 2.1.1 运行流程和历史轨迹

我们假设每个终端用户将通过其客户端应用程序连接到附近或负载最小的逻辑服务器，即此用户请求的原始站点。此外，我们遵循标准的执行请求和广播请求的方法<sup>[60-62]</sup>，即每个请求首先由客户端向服务器端的某个站点（即原始站点）发起函数调用，此后，根据此请求的类型（读请求或写请求），原始站点将对其做不同处理。对于读请求，服务器将返回它的请求结果；对于写请求，原始站点首先根据某个特定的条件（具体将在后文中介绍）决定是否接受此请求，如果请求被接受，那么请求的结果（即调用成功）将会反馈给客户端，此请求的副作用（即对数据库的更改）将被广播到所有站点，并最终在所有站点上被执行。在此之后，此请求完成。如果请求被拒绝，它的副作用将不会在任何站点上被执行，客户端将会收到调用失败的反馈。图 2.1b 给出了对于给定的某个请求在逻辑服务器端的不同状态以及状态之间的转换方式。

在服务器运行时，服务器中的每个请求在产生以及在上述这些状态之间转换的时刻被称为事件。对于一个请求，根据上述运行流程的描述，其将会在原始站点上产生一个调用事件；根据请求的类型以及处理方式，其将会在原始站点上

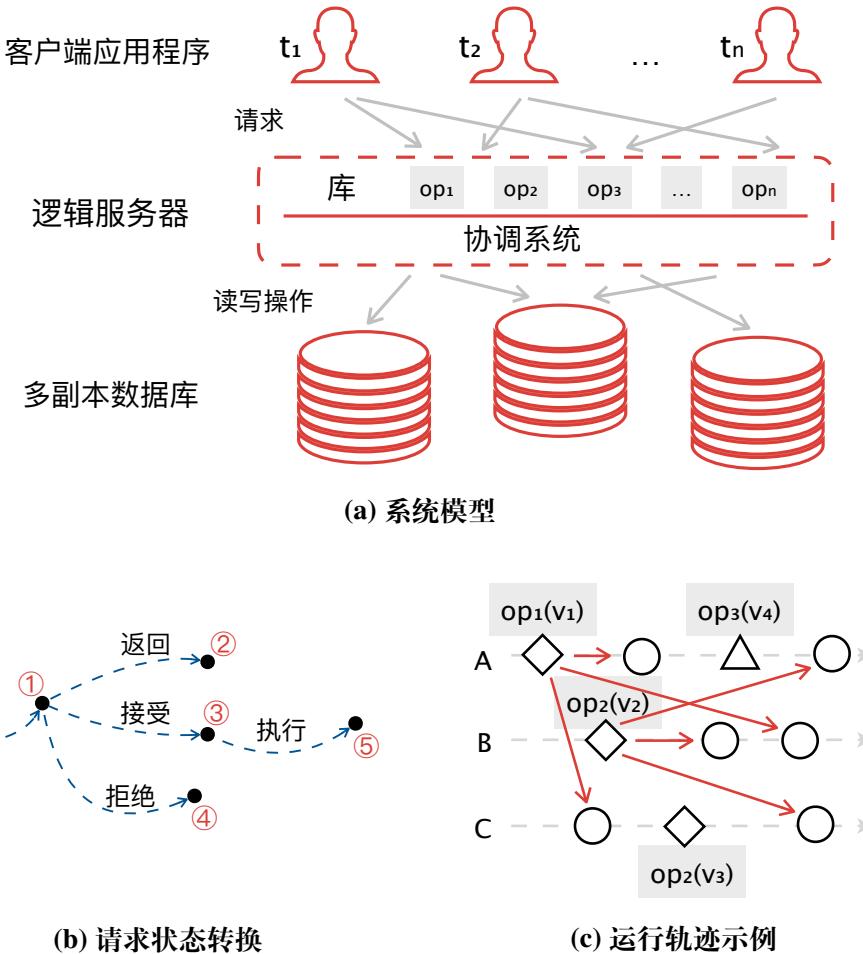


图 2.1 系统模型、请求状态转换和运行轨迹示例

产生返回事件、接受事件和拒绝事件中的一个；如果其被接受，那么这个请求将在所有站点上都产生一个执行事件。对于一个站点，它将会有若干个先后发生的事件。事件和事件在每个站点发生的时间点组成了历史轨迹。

### 2.1.2 历史轨迹示例

图 2.1c 展示了三个站点  $A$ ,  $B$  和  $C$  在执行 4 个请求时的历史轨迹图（坐标轴的方向即为时间顺序）。其中客户端应用程序在站点  $A$  上调用了函数  $op_1$ , 参数为  $v_1$ 。该请求是一个写请求，其被逻辑服务器接受后，副作用在  $A$  站点执行并被广播到站点  $B$  和  $C$  上执行。我们使用  $\diamond$  表示其在原始站点上的决策阶段，使用  $\circ$  表示其副作用在所有站点上的执行阶段。此后，该客户端应用程序又调用了  $op_3$  函数，该请求是一个读请求（使用  $\triangle$  表示），读请求不会被广播到其他站点上。另一个客户端应用程序在站点  $B$  上调用了函数  $op_2$ , 其参数为  $v_2$ ，该请求也被逻辑服务器所接受。其副作用被广播到站点  $A$  和  $C$  上执行。如果我们对消

---

```

1 withdraw(n):
2   if(@acct >= n):
3     @acct = @acct - n;
4
5 deposit(n):
6   @acct = @acct + n;

```

---



---

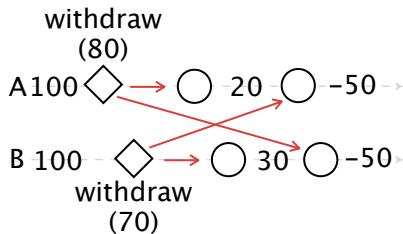
```

7 cal_interest():
8   delta = @acct * 0.05;
9   @acct = @acct + delta;
10
11 read_balance():
12   return @acct;

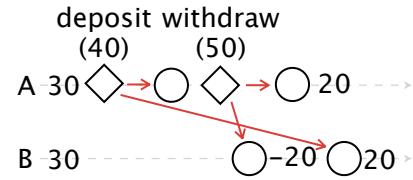
```

---

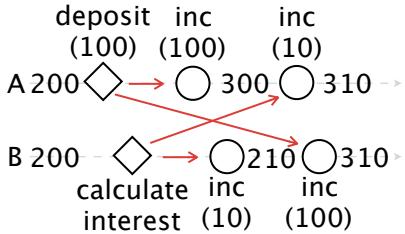
(a) 银行系统代码



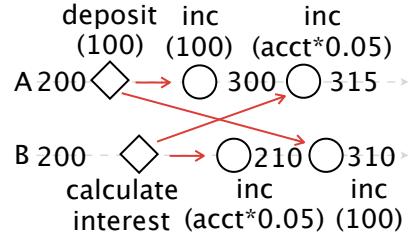
(b) 违背不变量



(c) 违背不变量



(d) 系统收敛



(e) 系统不收敛

图 2.2 银行系统代码和相应若干运行轨迹

息传递的顺序没有要求，仅仅保证最终送达，那么这两个请求的副作用将会以任意顺序在所有站点上被执行。第三个逻辑服务器上收到了另外一个调用函数  $op_2$  的请求，其参数为  $v_3$ 。此写请求被拒绝，因此其副作用不会被广播到其他站点上，也不会在原始站点被执行。

## 2.2 一个启发性的实例

在上小节给出的系统流程中，我们并没有对服务器端的协调系统做介绍，即默认所有到达站点的请求可以立即被处理，所有副作用在广播到一个站点时可以立即被执行。在本小节中，我们将一个简单的银行系统作为上述系统模型中逻辑服务器函数库的实例，来探究如果没有协调系统将会出现什么问题。从而进一步引出如何通过细粒度一致性语言中的原语来控制各个请求之间被处理和被执

行的顺序。

### 2.2.1 同步和依赖

图 2.2a 给出了一个银行系统的代码，该银行系统具有一个名为 `read_balance` 的读操作和三个名为 `withdraw`、`deposit` 和 `cal_interest` 的写操作。这四个操作分别用于查询余额、存款、取款以及计算利息。在这里，我们使用前缀 `@` 表示它是一个共享对象而不是一个局部变量。在本例中，余额 `@acct` 是一个共享对象。图 2.2b - 2.2e 给出了此函数库在两个站点 **A** 和 **B** 上被调用的若干历史轨迹图。

如图 2.2b 所示，最初站点 **A** 和站点 **B** 上账户的余额同为 100。接下来有两个取钱操作 `withdraw` 被调用，取钱的数额分别为 80 和 70。如果他们没有看到彼此的副作用，那么它们都可以被接受（即在执行代码第 2 行 `if` 条件判断成立），这导致在余额只有 100 时取钱的数额为 150，使得二者的副作用在另一个站点上被执行时，账户的余额最终变为负数。这会违背函数库的设计者在设计之初设定下的语义规范，即其希望银行账户的余额不能小于 0（这种只与状态有关的断言我们称之为不变量）。此时，若要保证余额不为负数，我们首先需要对这两个请求进行仲裁（即决定二者发生的先后顺序，具体方式将在后文中介绍），而后被裁决为后发生的请求将会等另一个取钱操作的副作用在其原始站点上执行完毕后再被处理，此时该请求由于账户余额不足（即 `if` 条件判断失败）而被拒绝。

另一个历史轨迹图如图 2.2c 所示。两个站点的初始余额同为 30。在 **A** 站点上，取钱请求 `withdraw(50)` 发生在存钱请求 `deposit(40)` 的副作用执行完成之后，此时 **A** 站点上余额已经变为 70，故取钱请求可以被接受。此后，**A** 站点上这两个请求的副作用被广播到 **B** 站点上。如果这些副作用以乱序的方式广播到站点 **B**，则站点 **B** 可能在某特定时间内余额变为负数，同样会违背不变量。相同的因果关系导致语义被违背的问题也出现在“Lost-Ring”问题<sup>[63]</sup> 中。

上面的示例表明，我们需要某种协调以确保请求被接受或其副作用被执行时，某些相关的副作用已经在它们之前被执行。为此，我们分别通过原语 `sync O` 和 `depend O` 对二者的顺序提供一定的保证。其中 **O** 是某些共享对象组成的集合。具有原语 `sync O` 的请求 `id` 在被处理之前，会记录所有站点上在其之前的对 **O** 中共享对象进行写入操作的请求作为其同步请求集合，并且等待集合中所有被接受请求的副作用在 `id` 的原始站点上执行完毕后，再被处理。具有原语 `depend O` 的请求 `id` 如果被接受，在被广播前会记录原始站点上在其之前的对 **O** 中共享对象进行写入操作的请求作为其依赖请求集合，经过广播后发送到所有站点，在这些站点上等待集合中所有请求的副作用在此站点上执行完毕后，再

被执行。

对于图 2.2d 所示的两个取钱请求，如果 withdraw 请求中包含 sync @acct 原语，其中一个 withdraw 的请求将会等待另外一个 withdraw 的请求的副作用执行完之后再被处理，此时后被处理的请求第 2 行的 if 判定失败，因此图中所示的历史轨迹将不会发生。同样的，对于图 2.2c 所示取钱再存钱的请求，如果 withdraw 请求中包含 depend @acct 原语，withdraw 的请求的副作用广播到 *B* 站点后，将会等待 deposit 的请求的副作用在 *B* 站点执行完之后再执行，因此 deposit 和 withdraw 两个请求的副作用的执行顺序将不会发生乱序，从而不会出现在某特定时间余额为负数的情况。

## 2.2.2 明确操作的副作用

在给定如上的协调之后，哪一部分应该被作为副作用来进行广播也影响着语义的正确性。如图 2.2d 和图 2.2e 所示，根据银行系统代码中计算利息的语句（第 8 行）是否为操作 cal\_interest 副作用的一部分，会有不同的执行结果。在图 2.2d 给出的轨迹中，我们认为计算利息的语句不是操作 cal\_interest 副作用的一部分。在原始站点 *B* 中，我们先按照百分之五的利率计算出利息为 10，此时被广播的副作用仅为对账户余额的加 10 操作（第 9 行），在调用 deposit 和 cal\_interest 的请求的副作用分别在站点 *A* 和站点 *B* 上被执行后，两个站点上账户的余额均为 310。

而在图 2.2e 中，我们认为计算利息的语句是 cal\_interest 副作用的一部分。在站点 *B* 中，按照百分之五的利率计算出利息仍为 10，但在站点 *A* 中，由于 cal\_interest 的副作用在 deposit 的副作用之后执行（此时账户余额为 300），那么通过计算得到的利息则变为了 15。最终 *A* 站点上账户的余额为 315，*B* 站点上账户的余额为 310。这使得多个站点上的数据进入了不一致的状态，违背了系统的收敛性。

同样的情况也会出现在 withdraw 请求中。如果我们将代码第 2 行 if 条件分支判断语句作为其副作用的一部分，那么在如图 2.2d 所示的请求情况下，两个请求都被接受，但在其副作用广播到另一个站点时，if 判定失败使得两个副作用均未被执行，最终导致 *A* 站点余额为 20、*B* 站点余额为 30 的情况，这同样违背了系统的收敛性。当系统的收敛性被违背时，我们不知道用户余额的真实数额，因此违背了收敛性的系统将出现状态分裂的情况，不能再继续运行。

由此可见，我们需要程序员明确说明源码中哪一部分应为操作的副作用。因此，我们在语言中添加了原语 let ... do ...。其中，在原语 let 后面的语句将在原始站点上执行（对应于图 2.1c 中的 ◊）；而在原语 do 之后的语句将作为副作用被广播到所有站点上执行（对应于图 2.1c 中的 ○）。除此之外，由于写请求在可以

被处理时，首先需要判断其是否被接受，我们将原语 **guard**  $B$  添加到语言中（ $B$  为布尔表达式），作为断言来显式的判断该操作调用是被接受还是被拒绝。

### 2.3 本章小结

在本章中，我们首先给出了包含客户端、逻辑服务器和多副本数据库的三层系统模型。而后，我们以图 2.1c 所示的历史轨迹为例，介绍了系统的运行流程。最后，我们通过银行系统不同的历史轨迹图，总结出了四个新的原语，分别为 **sync**、**depend**、**guard** 和 **let...do**，以使其能够在运行时满足预期的行为。在下一章中我们将对新的语言进行进一步的介绍。

## 第3章 细粒度一致性语言设计

本章主要给出面向分布式细粒度一致性的编程语言的语法、语义、性质以及应用实例。其中语法的定义将在第3.1节中给出；第3.2节中将给出此语法应用在若干个分布式系统的实例。形式化的操作语义将在第3.3节中给出；最后，我们给出了语言的性质以及得到的不同的一致性的保证，分别将在第3.4节和第3.5节中呈现。

### 3.1 语法的定义

图3.1a给出了客户端应用程序语言的语法。客户端程序 $\mathbb{P}$ 由若干个并行的命令 $C$ 组成。为了简化我们的模型，我们将一个命令绑定到一个服务器站点，他们的标识符为 $\iota$ 。应用程序的命令包括可以读取和写入局部变量 $x$ 的标准指令，以及命令 $x := \mathbf{call} f v$ ，该命令用于以参数 $v$ 调用服务器上名为 $f$ 的操作，并将服务器的返回值存入变量 $x$ 中。在此我们假设参数只有一个。

图3.1b给出了服务器函数库语言的语法。函数库用 $\Sigma$ 表示，其将函数的名称 $f$ 映射到相应的操作代码 $\tau$ ，该操作可以是 **input**  $x$  **sync**  $O$  **guard**  $B$  **depend**  $O$  **let**  $x' := E$  **do**  $C$  形式的写操作，或是 **input**  $x$  **sync**  $O$  **return**  $E$  形式的读操作。为了简化我们的语言，我们假设**let**命令中只有一个表达式。其中，服务器端的命令 $C$ 为标准指令，其可以读取和写入局部变量 $x$ 和共享对象 $o$ 并在其上做相应运算。

### 3.2 语言的应用实例

本节将给出用上述语言进行服务器函数库设计的若干实例，包括银行系统、add-win集合、以及拍卖系统，如图3.2所示。这三个实例将在以下三个小节中进行具体介绍。

#### 3.2.1 银行系统

图3.2a给出了银行系统的代码。正如第2.2节所介绍的那样，withdraw操作使用对共享变量`accts`同步的`sync`原语来获得其他请求对`@accts`的更新，并使用`guard`原语来确保用户`cid`的账户中有足够的余额。由于取钱操作与作用于共享变量`accts`上的更新之间存在因果关系（例如，取钱成功是由于在其之前存入了一定数额的钱），我们在`withdraw`中加入了对共享变量`accts`依赖的`depend`原语。`withdraw`的副作用是账户余额

---

$(Var) \quad x ::= \dots$        $(Val) \quad v ::= \dots$   
 $(SId) \quad i ::= \dots$        $(OpName) \quad f ::= \dots$   
 $(AExpr) \quad E ::= x \mid v \mid E + E \mid \dots$   
 $(ABexp) \quad B ::= true \mid false \mid E = E \mid \neg B \mid \dots$   
 $(ACmd) \quad C ::= \text{skip} \mid x := E \mid C; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \mid x := \text{call } f \ v$   
 $(Prog) \quad P \in SId \rightarrow ACmd$

(a) 客户端应用程序语法

$(Obj) \quad o ::= \dots$   
 $(ObSet) \quad O \in \wp(Obj)$   
 $(SExpr) \quad E ::= x \mid o \mid v \mid E + E \mid \dots$   
 $(SBexp) \quad B ::= true \mid false \mid E = E \mid \neg B \mid \dots$   
 $(SCmd) \quad C ::= \text{skip} \mid x := E \mid o := E \mid C; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C$   
 $(Op) \quad \tau ::= \text{input } x \text{ sync } O \text{ guard } B \text{ depend } O \text{ let } x' := E \text{ do } C \mid \text{input } x \text{ sync } O \text{ return } E$   
 $(Library) \quad \Sigma \in OpName \rightarrow Op$

(b) 服务器库函数语法

图 3.1 程序语言语法的定义

的自减操作。而对于 deposit 操作，其被接受和被执行均不受其他操作的影响（存钱操作必能成功，且无因果关联），因此不需要同步或依赖共享变量 accts。它的副作用为账户余额的自增操作。cal\_interest 操作会首先获取其他操作对 @accts 共享变量的更新，它的副作用为账户余额的自增操作，增量为使用原始站点上的余额计算得到的利息。注意此处（第 15 行）对共享变量 accts 的同步并不是必须的。这取决于在服务器运行阶段，对同时刻发生的一个调用 cal\_interest 的请求和其他更新共享变量 accts 的请求之间做仲裁时，函数库设计者是否希望 cal\_interest 尽可能的被裁决为后发生的事件。如果删除此处的同步原语，那么在服务器的运行中，逻辑上同时发生的计算利息操作 cal\_interest 和存钱操作 deposit 会以随机的顺序进行排序，即新存入的钱可能被用于计算利息，也可能不会被用于计算。如果保留此处的同步，那么这两个请求中计算利息操作 cal\_interest 一定会发生在存钱操作 deposit 之后，即新存入的钱一定会被用于计算利息。最后一个读操作 read\_balance 用于查询给定帐户 cid 的余额。

---

```

1 withdraw:
2   input cid, n
3   sync @accts
4   guard @accts.get(cid) ≥ n
5   depend @accts
6   do @accts.put(cid,
7     @accts.get(cid) - n)
8
9 deposit:
10  input cid, n
11  do @accts.put(cid,
12    @accts.get(cid) + n)
13 cal_interest:
14  input cid
15  sync @accts
16  let n = 0.05 *
17    @accts.get(cid)
18  do @accts.put(cid,
19    @accts.get(cid) + n)
20
21 read_balance:
22  input cid
23  return @accts.get(cid)
24

```

---

(a) 银行系统

---

```

1 add:
2   input e, id
3   do @set.add(<e, id>)
4
5
6 remove:
7   input e
8   depend @set
9   let es = @set.find_all(<e, _>)
10  do @set.delete_all(es)

```

---

(b) add-win set

---

```

1 register_user:
2   input cname, cinfo
3   sync @clients
4   guard cname ∉ @clients.keys()
5   do @clients.put(cname, cinfo)
6
7 close_auction:
8   sync @bids
9   let wid = highest(@bids)
10  do @item.status = closed
11    @item.winner = wid
12 place_bid:
13  input cname, price
14  sync @item
15  guard @item.status ≠ closed &&
16    cid ∈ @clients.keys()
17  depend @clients
18  do if cname ∉ @bids.keys() ||
19    cname ∈ @bids.keys() &&
20    @bids.get(cname) < price:
21      @bids.put(cname, price)
22

```

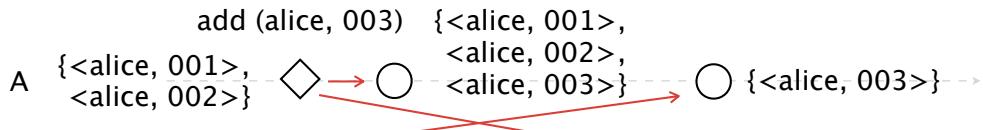
---

(c) 拍卖系统

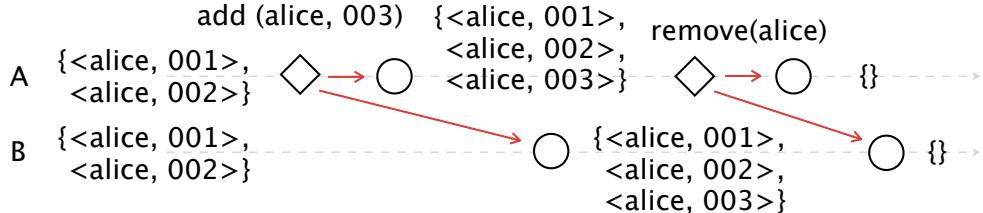
图 3.2 语言的应用实例

### 3.2.2 add-win 集合

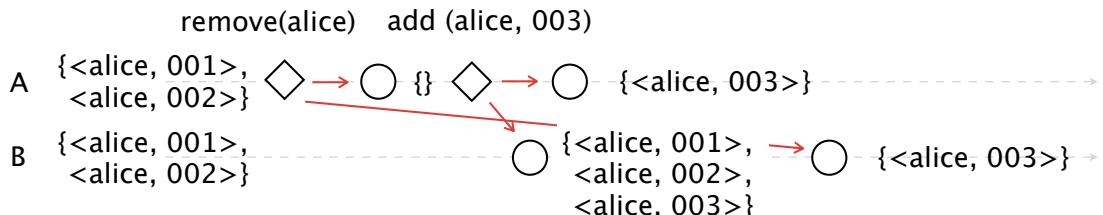
图 3.2b 给出了 add-win 集合的代码。这是一种无冲突复制数据类型 (conflict-free replicated data type)<sup>[50,64]</sup>。相比于传统的集合，add-win 集合能保证在分布式条件下，如果对集合的同一个元素同时进行添加和删除操作，添加操作始终是成功的，即元素会留在集合内。与之相反，无冲突复制数据类型中还存在 remove-win 集合，即对集合的同一个元素同时进行添加和删除操作时删除操作始终是成



(a) 不同站点上同时添加和删除某个元素



(b) 在同一个站点上先添加再删除某个元素



(c) 在同一个站点上先删除再添加某个元素

图 3.3 add-win 集合的若干历史轨迹图

功的。在实现时，add-win 集合需要对添加同一个元素的不同操作分别赋予一个唯一标识符，再通过此标识符来判断如何进行删除操作。

如图 3.2b 所示，在 add 操作中输入参数包括标识符 id 和数据 e，其中标识符的唯一性由外部系统保证。其副作用是将数据 e 和标识符 id 组成的元组放入集合类型的共享变量 @set 中。由于其被接受和被执行均不受其他操作的影响，此处不需要同步或依赖 @set。在 remove 操作中，es 用于表示在原始站点上所有包含数据 e 的元组，这些数据分别由不同的添加操作所放入。它的副作用是删除共享变量 @set 中所有在 es 集合中的元素。由于 es 集合的内容与之前在原始站点上执行过的对共享变量 @set 进行更新的操作有因果关系，我们在此处添加对 @set 进行依赖的 depend 原语。

图 3.3 给出了 add-win 集合的若干历史轨迹图。如图 3.3a 所示，在初始时刻站点 A 和 B 上存在着具有标识符 001 和 002 的数据 alice。此后，两站点上分别同时产生了一个添加和删除 alice 元素的请求，其中添加请求的标识符为

003，两个请求均被接受。*A* 站点上执行完添加请求的副作用后，集合中共有 3 个 `alice` 数据；在 *B* 站点上，删除请求的副作用会删除在原始站点上存在的所有包含 `alice` 数据的元素（即具有标识符 001 和 002 的元素），副作用在 *B* 站点执行完毕后，*B* 站点集合为空集。最后，二者的副作用分别广播到另一个站点上。在 *B* 站点上，执行添加请求的副作用后集合中具有标识符为 003 的 `alice` 数据；在 *A* 站点上，执行删除请求的副作用后，只删除了标识符 001 和 002 的元素，故集合最终也只含有具有标识符为 003 的 `alice` 数据。最终系统收敛，并且同时执行添加和删除某一元素的请求后，此元素仍在集合内，这表明了 add-win 集合语义的正确性。

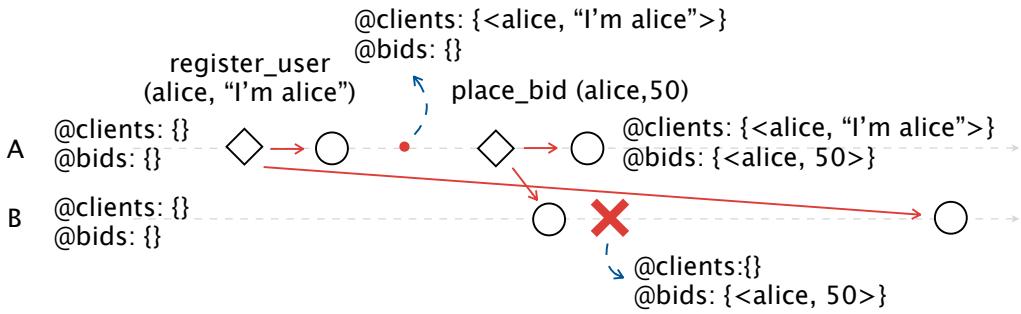
在图 3.3b 中给出了在站点 *A* 上先添加再删除 `alice` 数据的历史轨迹图。由于在 `delete` 操作中我们对共享变量 `@set` 进行了依赖，添加和删除操作的副作用广播到站点 *B* 时没有发生乱序现象，保证了系统的收敛性。类似的，在图 3.3c 中给出了在站点 *A* 上先删除再添加 `alice` 数据的历史轨迹图。此时，删除操作并不会删除标识符为 003 的元素，因此当添加和删除操作的副作用广播到站点 *B* 时，即使发生了乱序现象，`alice` 数据最终仍会留在集合内。

### 3.2.3 拍卖系统

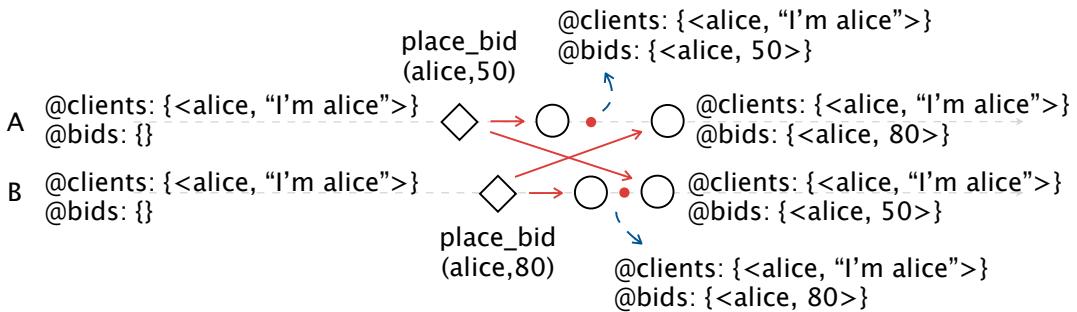
图 3.2c 给出了拍卖系统的代码。拍卖系统共有三个操作，即用于用户注册的 `register_user`、用于竞拍的 `place_bid` 和用于结束竞拍的 `close_auction`。在 `register_user` 操作中，用户名 `cname` 和客户信息 `cinfo` 分别作为键和值放入键-值对集合 `@clients` 中。类似于邮箱系统注册，此处用户名将作为用户的唯一标识符，因此 `register_user` 将同步所有对 `@clients` 共享变量进行写入的操作，并在 `guard` 中确保该用户名尚未被注册。

`place_bid` 操作由用户 `name` 以价格 `price` 发起竞价。它首先同步所有对 `@item` 共享变量进行写入的操作并在 `guard` 中确保拍卖尚未被关闭。此外，它还需要确保客户 `cname` 已经被注册来防止。类似于 `cal_interest` 操作中可以对是否同步 `@accts` 共享变量进行选择的可能，此处我们也可以选择对 `@clients` 变量进行同步或不对其进行同步。如果对 `@clients` 变量进行同步，那么如果在站点 *A* 和 *B* 上同时发生了某个用户的注册请求以及此用户的竞拍请求，由于此处同步的存在，则竞拍请求一定会等待注册请求的副作用执行完毕后再被处理，此时如果拍卖未结束，则拍卖一定会成功；如果没有对 `@clients` 变量进行同步，则没有上述保证（即此用户可能会拍卖成功，也可能因为用户尚未注册而拍卖失败）。

虽然此处是否对 `@clients` 变量进行同步是可以选择的，但由于



(a) 在 place\_bid 操作中不依赖 clients 共享变量导致出现未注册的竞拍



(b) 在 place\_bid 操作中将 if 判断放入 guard 原语中导致系统收敛性被违背

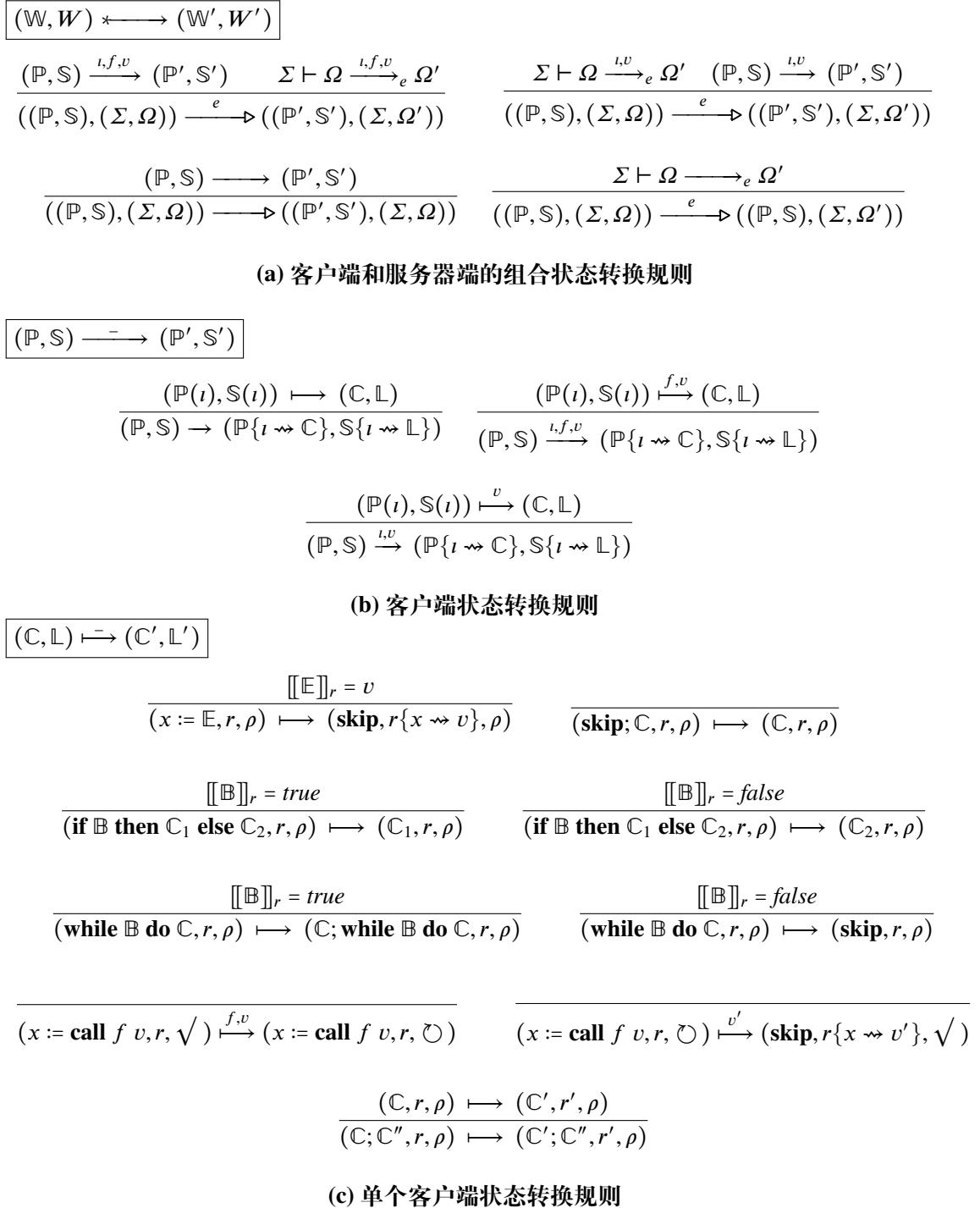
图 3.4 拍卖系统的若干历史轨迹图

place\_bid 与 register\_user 之间存在因果关系（如果没有注册用户，则无法进行拍卖），故此处一定需要 depend 原语，否则会导致在某时刻出现未注册的用户拍卖成功的行为，如图 3.4a 所示。place\_bid 的副作用会再次判断此竞价信息是否有效，其条件为此用户尚未竞拍过物品或此次竞拍的价格是此用户多次竞拍中的最高竞价。如果此条件满足，则将此竞价的信息写入共享变量 @bids 中。注意此处我们需要将判断条件放入 do 原语而不是 guard 原语中。否则同一用户的两个不同竞拍将会导致系统出现收敛性的违背，如图 3.4b 所示。

close\_auction 操作用于关闭拍卖。它同步所有对 @bids 共享变量进行写入的请求以获取所有对物品的竞价，并通过确定性函数 highest 计算出最高出价者。它的副作用是记录最高出价者并结束拍卖。

### 3.3 操作语义的定义

在先前的介绍中，我们已经对各个原语的语义有了一定的了解，本节将给出形式化的操作语义。如图 3.5 和图 3.6 所示，操作语义规则由四个部分组成，分别为客户端和服务器端的组合状态转换、客户端状态转换、服务器端状态转换以



$\Sigma \vdash \Omega \xrightarrow[-]{} \Omega'$	
	(INVOKED)
$\tau = \Sigma(f) \quad \text{id} \notin \text{dom}(M) \quad M' = M\{\text{id} \rightsquigarrow (\iota, \tau, v, \text{dom}(M))\} \quad e = (\iota, \text{id}, \text{op } \tau)$	$\Sigma \vdash (M, S) \xrightarrow[\iota, f, v]{e} (M', S)$
	(RETURN)
$M(\text{id}) = (\iota, \text{input } x \text{ sync } O_s \text{ return } E, v, h_t) \quad h_t \cap (\text{wr}(O_s)_M \cup \text{on}(\iota)_M) \subseteq h_l$	
$S(\iota) = (h_l, s) \quad M' = M\{\text{id} \rightsquigarrow \nabla\} \quad v' = [\![E]\!]_{s, \{x \rightsquigarrow v\}} \quad e = (\iota, \text{id}, \text{rtn})$	$\Sigma \vdash (M, S) \xrightarrow[\iota, v']{e} (M', S)$
	(ACCEPT)
$M(\text{id}) = (\iota, \text{input } x \text{ sync } O_s \text{ guard } B \text{ depend } O_d \text{ let } x' := E \text{ do } C, v, h_t)$	
$h_t \cap (\text{wr}(O_s)_M \cup \text{on}(\iota)_M) \subseteq h_l \quad S(\iota) = (h_l, s) \quad r = \{x \rightsquigarrow v\} \quad [\![B]\!]_{s, r} = \text{true}$	$e = (\iota, \text{id}, \text{acpt } (s, v))$
$M' = M\{\text{id} \rightsquigarrow (\iota, r\{x' \rightsquigarrow [\![E]\!]_{s, r}\}, C, h_l \cap \text{wr}(O_d)_M)\}$	$\Sigma \vdash (M, S) \xrightarrow[\iota, 1]{e} (M', S)$
	(REJECT)
$M(\text{id}) = (\iota, \text{input } x \text{ sync } O_s \text{ guard } B \text{ depend } O_d \text{ let } x' := E \text{ do } C, v, h_t)$	
$h_t \cap (\text{wr}(O_s)_M \cup \text{on}(\iota)_M) \subseteq h_l \quad S(\iota) = (h_l, s) \quad r = \{x \rightsquigarrow v\}$	$e = (\iota, \text{id}, \text{rej})$
$[\![B]\!]_{s, r} = \text{false} \quad M' = M\{\text{id} \rightsquigarrow \nabla\}$	$\Sigma \vdash (M, S) \xrightarrow[\iota, 0]{e} (M', S)$
$M(\text{id}) = (\_, r, C, h_l) \quad S(\iota) = (h_l, s) \quad \text{id} \notin h_l \quad h_t \subseteq h_l$	
$(C, s, r) \mapsto^* (\text{skip}, s', \_) \quad S' = S\{\iota \rightsquigarrow (h_l \cup \{\text{id}\}, s')\}$	$e = (\iota, \text{id}, \text{exe})$
$\Sigma \vdash (M, S) \xrightarrow[e]{} (M, S')$	(EXECUTE)

(a) 服务器端状态转换规则 (一)

$(C, s, r) \mapsto (C', s', r')$	
	(INVOKED)
$\frac{[\![E]\!]_{s, r} = v}{(x := E, s, r) \mapsto (\text{skip}, s, r\{x \rightsquigarrow v\})}$	$\frac{[\![E]\!]_{s, r} = v}{(o := E, s, r) \mapsto (\text{skip}, s\{o \rightsquigarrow v\}, r)}$
$\frac{(C, s, r) \mapsto (C', s', r)}{(\text{skip}; C, s, r) \mapsto (C, s, r)}$	$\frac{(C, s, r) \mapsto (C', s', r)}{(C; C'', s, r) \mapsto (C', C'', s', r)}$
$\frac{[\![B]\!]_{s, r} = \text{true}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, s, r) \mapsto (C_1, s, r)}$	$\frac{[\![B]\!]_{s, r} = \text{false}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, s, r) \mapsto (C_2, s, r)}$

(b) 服务器端状态转换规则 (二)

图 3.6 操作语义规则 (二)

定义事件的轨迹  $\Delta$ , 如图 3.8c 所示。客户端状态转换规则和服务器端状态转换规则将在第 3.3.1 小节和第 3.3.2 小节中分别进行详细的介绍。

$$\text{on}(\iota)_M \stackrel{\text{def}}{=} \{\text{id} \mid M(\text{id}).SId = \iota\}$$

$$\text{wr}(O)_M \stackrel{\text{def}}{=} \{\text{id} \mid (\text{wset}(M(\text{id})) \cap O) \neq \emptyset\}$$

$$\text{wset}(m) \stackrel{\text{def}}{=} \begin{cases} \text{wset}(\tau) & \text{if } m = (\tau, \_, \_) \\ \text{wset}(C) & \text{if } m = (\_, \_, C, \_) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{wset}(\tau) \stackrel{\text{def}}{=} \text{wset}(C) \text{ , 其中 } \tau = \dots \text{ do } C$$

$$\text{rset}(\tau) \stackrel{\text{def}}{=} \begin{cases} \text{rset}(B) \cup \text{rset}(E) \cup \text{rset}(C) & \text{, 其中 } \tau = \dots \text{ guard } B \dots \text{ let } x' := E \text{ do } C \\ \text{rset}(E) & \text{, 其中 } \tau = \dots \text{ return } E \end{cases}$$

$$\text{wset}(C) \stackrel{\text{def}}{=} \begin{cases} \{o\} & \text{if } C = (o := E) \\ \text{wset}(C_1) \cup \text{wset}(C_2) & \text{if } C = (C_1 ; C_2) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{rset}(C) \stackrel{\text{def}}{=} \begin{cases} \text{rset}(E) & \text{if } C = (o := E) \vee C = (x := E) \\ \text{rset}(C_1) \cup \text{rset}(C_2) & \text{if } C = (C_1 ; C_2) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{rset}(B) \stackrel{\text{def}}{=} \begin{cases} \text{rset}(B') & \text{if } B = (!B') \\ \text{rset}(E_1) \cup \text{rset}(E_2) & \text{if } B = (E_1 = E_2) \\ \emptyset & \text{if } B = (\text{true}) \vee B = (\text{false}) \\ \dots & \dots \end{cases}$$

$$\text{rset}(E) \stackrel{\text{def}}{=} \begin{cases} \{o\} & \text{if } E = (o) \\ \text{rset}(E_1) \cup \text{rset}(E_2) & \text{if } E = (E_1 + E_2) \\ \emptyset & \text{if } E = (x) \vee E = (v) \\ \dots & \dots \end{cases}$$

图 3.7 定义操作语义规则的一些辅助定义

### 3.3.1 客户端状态转换规则

图 3.8a 中定义了客户端状态  $\mathbb{S}$ , 它标识符  $\iota$  映射到局部状态  $\mathbb{L}$ 。局部状态  $\mathbb{L}$  由寄存器  $r$  和运行状态  $\rho$  组成。其中寄存器  $r$  是一个由局部变量名称  $x$  到值  $v$  的一个映射, 用于存储局部变量; 运行状态  $\rho$  则用来指示客户端程序是否被服务器端所阻塞。

$$\begin{array}{lll}
 (UWorld) \mathbb{W} ::= (\mathbb{P}, \mathbb{S}) & (ALstate) \mathbb{L} ::= (r, \rho) & (Status) \rho ::= \checkmark \mid \circlearrowleft \\
 (AState) \mathbb{S} \in SId \rightarrow ALstate & (Reg) r \in Var \multimap Val
 \end{array}$$

(a) 客户端状态模型

$$\begin{array}{lll}
 (SWorld) \mathbb{W} ::= (\Sigma, \Omega) & (Msg) m ::= (\iota, \tau, v, h) \mid (\iota, r, C, h) \mid \nabla \\
 (Server) \Omega ::= (M, S) & (State) S \in SId \rightarrow LState \\
 (MsgBuf) M \in TId \multimap Msg & (LState) L ::= (h, s) \\
 (TId) \text{id} ::= \dots & (History) h \in \wp(TId) & (Store) s \in Obj \multimap Val
 \end{array}$$

(b) 服务器端状态模型

$$\begin{array}{lll}
 (Event) e ::= (\iota, \text{id}, \pi) & (Trace) \Delta ::= e :: \Delta \mid \perp & (Ctx) \kappa ::= (s, v) \\
 (Flag) \pi ::= \mathbf{op} \tau \mid \mathbf{acpt} \kappa \mid \mathbf{exe} \mid \mathbf{rej} \mid \mathbf{rtn}
 \end{array}$$

(c) 事件和事件轨迹

图 3.8 客户端和服务器端的状态模型

图 3.5b 给出了客户端的状态转换规则，客户端的单步状态由其中任何一个客户端的单步状态转换完成。单步的状态转换包括三种，分别为内部状态转换、请求调用和请求返回。图 3.5c 中，我们给出了客户端的单步状态转换规则。其中，第七个和第八个规则为请求调用和请求返回，其余的规则为内部状态转换。在请求调用规则中，客户端函数调用指令  $x := \mathbf{call} f v$  通过设置当前客户端的运行状态  $\rho$  为  $\circlearrowleft$  来阻塞当前客户端，请求返回规则中则通过重置运行状态  $\rho$  为  $\checkmark$  来完成此次函数调用。

### 3.3.2 服务器端状态转换规则

服务器  $\Omega$  在图 3.8b 中被定义，其包含消息缓存区  $M$  和服务器状态  $S$ 。其中， $M$  将请求的标识符  $\text{id}$  映射到消息  $m$ ； $S$  将服务器站点标识符  $\iota$  映射到服务器局部状态  $L$ 。服务器局部状态由局部历史  $h$  和存储  $s$  组成。其中， $h$  是请求标识符  $\text{id}$  组成的集合； $s$  则是一个由共享对象名称  $o$  到值  $v$  的一个映射，用于存储共享对象。消息  $m$  包含三类，第一种包含请求刚到达时的信息，即请求的标识符  $\text{id}$ 、请求的操作  $\tau$ 、请求的参数  $v$  以及请求到达时刻的全局历史  $h$ ；第二种包含请求要被执行时的信息，即请求的标识符  $\text{id}$ 、计算完 **let** 语句后的寄存器状态  $r$ 、将要执行的 **do** 命令  $C$  以及执行时的依赖请求集合  $h$ ；第三种为处理完成

后的标记，如果一个读请求被处理或一个写请求被拒绝，它将被最终标记为  $\nabla$ 。

图 3.6a 和图 3.6b 中给出了服务器端的单步状态转换规则。在图 3.6a 中给出的五个规则分别与图 2.1c 中的请求的状态转换图相对应。第一个规则 **Invoke** 给出了当服务器端函数库被调用时的状态转换。此调用可被认为是一个由服务器标识符  $i$ ，函数名称  $f$  和参数  $v$  组成的三元组。服务器端将为此操作调用生成一个唯一标识符  $id$ ，并将其与一条消息一同放入消息缓存区。该消息包含服务器标识符  $i$ 、操作  $\tau$ 、参数  $v$  和当前服务器的全局历史记录  $dom(M)$ 。

第二条 **RETURN** 规则，第三条 **ACCEPT** 规则和第四条 **REJECT** 规则会从消息缓存区  $M$  中选择一个带有标识符  $id$  的请求进行处理。其被处理的前提条件是  $h_t \cap (\text{wr}(O_s)_M \cup \text{on}(i)_M) \subseteq h_l$ ，即要求请求  $id$  在调用阶段时的历史  $h_t$  中，满足以下要求之一的请求都必须在当前站点的本地历史记录中。

- 对  $O_s$  集合中任意一个或多个共享变量进行更新；
- 在  $id$  请求的原始站点  $i$  上并在  $id$  之前。

其中第一个要求即为 **sync** 原语所带来的请求之间的顺序限制；第二个要求则是为了保证每个站点的执行满足程序顺序，即所有在  $i$  站点上请求被处理的顺序和这些请求在  $i$  站点上到达的顺序保持一致。我们将在 3.4 节给出形式化的描述。

如果  $id$  请求可以被处理，如 **RETURN** 规则所示，当请求  $id$  是一个读请求时，我们将会计算该请求的 **return** 表达式的值，并将它存到变量  $v'$  中返回给客户端，其消息缓存区中的消息将被标记为  $\nabla$ 。如果请求  $id$  是一个写请求，**guard** 的后面布尔表达式的值为真或者为假，分别对应于 **ACCEPT** 规则和 **REJECT** 规则。如果请求  $id$  被接受，那么其消息缓存区的消息将更新为一个新的四元组，其中的元素依次为这个请求的原始站点标号、执行完 **let** 之后的寄存器状态、**do** 原语指令以及一个由当前站点中所有写了  $O_s$  集合中对象的请求所组成的历史。如果请求  $id$  被拒绝，那么其消息缓存区中的消息将被标记为  $\nabla$ 。

最后一条规则 **EXECUTE** 是执行规则，其将 **ACCEPT** 规则放入的四元组取出，请求  $id$  可以被执行条件是当前站点的历史记录  $h_l$  中要包含四元组中的历史记录  $h_t$  的所有请求 ( $h_t \subseteq h_l$ )。此规则将在本地执行  $C$  中包含的命令，并更新本地存储  $s$ 。本地执行的状态转换规则如图 3.6b 所示。

以上五个服务器转换规则每个都会生成一个相应的事件  $e$ ，该事件在图 3.8c 中定义。事件由站点标识符  $i$ ，请求标识符  $id$  和用于指示事件类型的标志  $\pi$  组成。事件轨迹  $\Delta$  是由事件组成的序列。在事件中，接受事件 **acpt** 记录了请求在被接受时的上下文  $\kappa$ 。 $\kappa$  由请求在被接受时原始站点的存储  $s$  和请求的参数  $v$  组成。请求的上下文将唯一确定一个请求的副作用。

$e_A \leqslant_{\Delta} e_B$	$\stackrel{\text{def}}{=} \Delta = \dots :: e_B :: \dots :: e_A :: \dots$
$\text{events}(\iota)_{\Delta}$	$\stackrel{\text{def}}{=} \{e \mid e \in \Delta \wedge e = (\iota, \_, \_)\}$
$\text{events}(\text{id}, \pi)_{\Delta}$	$\stackrel{\text{def}}{=} \{e \mid e \in \Delta \wedge e = (\_, \text{id}, \pi)\}$
$\text{his\_of}(\iota)_{\Omega}$	$\stackrel{\text{def}}{=} h, \text{ 其中 } \Omega = (\_, S), S(\iota) = (h, \_)$
$\text{store\_of}(\iota)_{\Omega}$	$\stackrel{\text{def}}{=} s, \text{ 其中 } \Omega = (\_, S), S(\iota) = (\_, s)$
$\text{sync}(\tau_A, \tau_B)$	$\stackrel{\text{def}}{=} \tau_A.O_w \cap \tau_B.O_s \neq \emptyset$
$\text{dep}(\tau_A, \tau_B)$	$\stackrel{\text{def}}{=} \tau_A.O_w \cap \tau_B.O_d \neq \emptyset$
$\text{sync}(\text{id}_A, \text{id}_B)_{\Delta}$	$\stackrel{\text{def}}{=} \text{sync}(\text{op}(\text{id}_A)_{\Delta}, \text{op}(\text{id}_B)_{\Delta})$
$\text{dep}(\text{id}_A, \text{id}_B)_{\Delta}$	$\stackrel{\text{def}}{=} \text{dep}(\text{op}(\text{id}_A)_{\Delta}, \text{op}(\text{id}_B)_{\Delta})$
$\tau.O_s$	$\stackrel{\text{def}}{=} O, \text{ 其中 } \tau = \dots \text{ sync } O \dots$
$\tau.O_d$	$\stackrel{\text{def}}{=} O, \text{ 其中 } \tau = \dots \text{ dep } O \dots$
$\tau.O_w$	$\stackrel{\text{def}}{=} \text{wset}(C), \text{ 其中 } \tau = \dots \text{ do } C \dots$
$\text{op}(\text{id})_{\Delta}$	$\stackrel{\text{def}}{=} \tau, \text{ 其中 } (\_, \text{id}, \text{op } \tau) \in \Delta$
$\text{site\_of}(\text{id})_{\Delta}$	$\stackrel{\text{def}}{=} \iota, \text{ 其中 } (\iota, \text{id}, \text{op } \_) \in \Delta$
$\text{idle}(\Omega)$	$\stackrel{\text{def}}{=} \forall \iota, \text{his\_of}(\iota)_{\Omega} = \{\text{id} \mid M(\text{id}) \neq \nabla\}$
$[\Omega]$	$\stackrel{\text{def}}{=} \exists s, \forall \iota, \text{store\_of}(\iota)_{\Omega} = s$
$\text{init}(\Omega)$	$\stackrel{\text{def}}{=} M = \{\} \wedge [\Omega] \wedge \forall \iota, \text{his\_of}(\iota)_{\Omega} = \{\}$
$e_A \xrightarrow[\iota]{\text{so}} e_B$	$\stackrel{\text{def}}{=} e_A \leqslant_{\Delta} e_B \wedge e_A \in \text{events}(\iota)_{\Delta} \wedge e_B \in \text{events}(\iota)_{\Delta}$
$\Sigma \vdash \Omega \succ^{\Delta} \Omega'$	$\stackrel{\text{def}}{=} \exists \mathbb{P} \mathbb{P}' \mathbb{S} \mathbb{S}', \text{init}(\Omega) \wedge ((\mathbb{P}, \mathbb{S}), (\Sigma, \Omega)) \xrightarrow{\Delta}^* ((\mathbb{P}', \mathbb{S}'), (\Sigma, \Omega'))$
$\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$	$\stackrel{\text{def}}{=} \Sigma \vdash \Omega \succ^{\Delta} \Omega' \wedge \text{idle}(\Omega')$

图 3.9 语言性质中的一些辅助定义

### 3.4 语言的性质

通过上一节对操作语义的介绍，我们知道每个请求可被分为两个阶段，即在原始站点上被接受和在所有站点上（副作用）被执行。编程人员主要关心在请求的原始站点上满足的某些属性（如 **guard** 中的断言  $B$ ）在其副作用被执行时仍可以被满足。由于副作用是唯一对站点存储进行修改的操作，这些属性是否被满足取决于此请求被接受或被执行时，在这两个站点（在此我们只考虑被执行的站点中的其中一个）上局部历史的差别。例如，在图 2.2d 中，`withdraw` 请求在

表 3.1 操作之间的关系和一致性的定义

前提			结论	
静态关系 +	动态关系	= 关系		
操作同步 ( $\text{sync}(\_, \_)$ )	仲裁顺序 (定义 3.3)	同步关系 (定义 3.4)	可见性 (定义 3.1)	一致性 (定义 3.7)
—	—	程序顺序 (定义 3.6)		
操作依赖 ( $\text{dep}(\_, \_)$ )	可见性 (定义 3.1)	依赖关系 (定义 3.5)	执行顺序 (定义 3.2)	

站点  $A$  上被接受时，余额不小于 0 这个属性被满足，而在站点  $B$  执行时不满足，是由于这两个事件在站点  $A$  和  $B$  上局部历史的差别（即在  $B$  站点上被执行时少看到一个存钱 40 的副作用）。

进一步的，由于在单个站点上事件的顺序是严格全序的，这种历史的关系可以等价为两个请求之间的顺序关系（例如，在图 2.2d 所示的例子中，表现为 `withdraw` 和 `deposit` 两个请求之间的关系），我们将请求被接受时和其原始站点局部历史中的请求之间的关系定义为可见性，将请求副作用被执行时和执行副作用的站点局部历史中的请求之间的关系定义为执行顺序，分别如定义 3.1 和定义 3.2 所示。

**定义 3.1 (可见性)** 一个请求  $\text{id}_A$  对另一个请求  $\text{id}_B$  在轨迹  $\Delta$  下是可见的（用  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$  表示），当且仅当存在  $\iota$ 、 $e_A$  和  $e_B$  使得  $e_A \in \text{events}(\text{id}_A, \text{exe})_{\Delta}$ 、 $e_B \in \text{events}(\text{id}_B, \text{acpt})_{\Delta}$  并且  $e_A \xrightarrow[\iota]{\text{so}}_{\Delta} e_B$ 。

定义 3.1 表示，一个请求  $\text{id}_A$  对另一个请求  $\text{id}_B$  是可见的，当且仅当  $\text{id}_B$  在原始站点被处理前看到了  $\text{id}_A$  的副作用。即  $\text{id}_A$  的副作用被执行这个事件和  $\text{id}_B$  被接受的事件在  $\text{id}_B$  的原始站点  $\iota$  上满足站点顺序  $(e_A \xrightarrow[\iota]{\text{so}}_{\Delta} e_B)$ 。两个事件  $e_A$  和  $e_B$  之间的站点顺序的定义见图 3.9。

**定义 3.2 (执行顺序)** 两个请求  $\text{id}_A$  和  $\text{id}_B$  在轨迹  $\Delta$  下具有执行顺序（用  $\text{id}_A \xrightarrow{\text{eo}}_{\Delta} \text{id}_B$  表示）当且仅当对于所有的  $\iota$ ，存在  $e_A$  和  $e_B$  使得  $e_A \in \text{events}(\text{id}_A, \text{exe})_{\Delta}$ 、 $e_B \in \text{events}(\text{id}_B, \text{exe})_{\Delta}$  和  $e_A \xrightarrow[\iota]{\text{so}}_{\Delta} e_B$ 。

定义 3.2 给出了表示两个请求  $\text{id}_A$  和  $\text{id}_B$  之间的执行顺序，即二者的副作用在所有站点上的执行事件都具有此顺序。

在定义了可见性和执行顺序之后，我们需要进一步确定如何才能得到编程人员关心的这两种顺序。这两种顺序关系都由一个静态关系（在代码层面上操作之间的关系）和一个动态关系（在系统运行时调用这些操作的请求之间的关系）作为前提，如表 3.1 所示，接下来我们将逐个解释这些关系。

### 3.4.1 操作同步和操作依赖

如之前介绍的那样，我们在操作中使用原语 **sync** 和 **depend** 来保证一个操作与其他操作之间能够保证上述可见性关系和执行顺序。具体来说，具有同步集合  $O_s$ （原语 **sync** 之后的共享变量集合）和依赖集合  $O_d$ （原语 **depend** 之后的共享变量集合）的操作将会同步所有写入  $O_s$  的操作，并依赖于写入  $O_d$  的操作。两个操作  $\tau_A$  和  $\tau_B$  之间的这种同步和依赖关系分别被定义为  $\text{sync}(\tau_A, \tau_B)$  和  $\text{dep}(\tau_A, \tau_B)$ ，如图 3.9 中所示。

### 3.4.2 仲裁顺序

**定义 3.3 (仲裁顺序)** 两个请求  $\text{id}_A$  和  $\text{id}_B$  在轨迹  $\Delta$  下具有仲裁顺序（用  $\text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B$  表示）当且仅当存在  $e_A$  和  $e_B$  使得  $e_A \in \text{events}(\text{id}_A, \text{op})_{\Delta}$ 、 $e_B \in \text{events}(\text{id}_B, \text{op})_{\Delta}$  和  $e_A \leq_{\Delta} e_B$ 。

定义 3.3 给出了服务器上对两个请求之间裁决的仲裁顺序，即当两个请求在逻辑时间上同时发生时，服务器对二者做出的顺序判断。两个请求  $\text{id}_A$  和  $\text{id}_B$  之间有仲裁顺序即为全局视角下请求  $\text{id}_A$  和  $\text{id}_B$  发生的事件的顺序关系（由全局事件轨迹  $\Delta$  所记录）。

### 3.4.3 同步关系、依赖关系和程序顺序

**定义 3.4 (同步关系)** 两个请求  $\text{id}_A$  和  $\text{id}_B$  具有同步关系（用  $\text{id}_A \xrightarrow{\text{sync}}_{\Delta} \text{id}_B$  表示）当且仅当  $\text{sync}(\text{id}_A, \text{id}_B)_{\Delta}$  以及  $\text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B$ 。

定义 3.4 给出两个请求之间的同步关系。即一个请求  $\text{id}_B$  同步另一个请求  $\text{id}_A$  当且仅当它们具有静态的操作同步关系  $\text{sync}(\text{id}_A, \text{id}_B)_{\Delta}$  和动态的仲裁顺序关系  $\text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B$ 。

**定义 3.5 (依赖关系)** 两个请求  $\text{id}_A$  和  $\text{id}_B$  在轨迹  $\Delta$  下具有依赖关系（用  $\text{id}_A \xrightarrow{\text{dep}}_{\Delta} \text{id}_B$  表示），当且仅当  $\text{dep}(\text{id}_A, \text{id}_B)_{\Delta}$  并且  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。

定义 3.4 给出两个请求之间的依赖关系。即，一个请求  $\text{id}_B$  依赖另一个请求  $\text{id}_A$  当且仅当他们有静态的操作依赖关系  $\text{dep}(\text{id}_A, \text{id}_B)_{\Delta}$  和动态的可见性关系  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。

**定义 3.6 (程序顺序)** 两个请求  $\text{id}_A$  和  $\text{id}_B$  在轨迹  $\Delta$  下具有程序顺序（用  $\text{id}_A \xrightarrow{\text{po}}_{\Delta} \text{id}_B$  表示）当且仅当  $\text{site\_of}(\text{id}_A)_{\Delta} = \text{site\_of}(\text{id}_B)_{\Delta}$  并且  $\text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B$ 。

定义 3.6 给出了两个请求之间的程序顺序，这个顺序是由每个客户端的请求到来的顺序决定的。即，两个请求  $\text{id}_A$  和  $\text{id}_B$  具有程序顺序当且仅当  $\text{id}_A$  和

$\text{id}_B$  在同一站点上被调用，而且  $\text{id}_A$  的调用要早于  $\text{id}_B$ 。

### 3.4.4 一致性

**定义 3.7(一致性)** 一个轨迹  $\Delta$  满足一致性当且仅当以下几点都成立：

1. 对于所有的  $\text{id}_A$  和  $\text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{po}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。
2. 对于所有的  $\text{id}_A$  和  $\text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{sync}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。
3. 对于所有的  $\text{id}_A$  和  $\text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{dep}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{eo}}_{\Delta} \text{id}_B$ 。

使用上述定义，我们可以在轨迹上定义一致性，如定义 3.7 所示。这意味着通过请求之间的同步关系、依赖关系和程序顺序，我们可以分别保证它们之间的可见性关系和执行顺序关系。

**定理 3.1** 对于所有的库  $\Sigma$ , 轨迹  $\Delta$ , 以及服务器  $\Omega \Omega'$ , 如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ , 那么  $\Delta$  满足一致性。

定理 3.1 则说明了我们之前定义的操作语义满足此一致性，即对于任何函数库  $\Sigma$ , 如果服务器从任何一个初始的(见定义 `initial`)状态  $\Omega$  运行到任何一个闲置(见定义 `idle`)的状态，并生成了事件轨迹  $\Delta$ ，则  $\Delta$  将满足定义 3.7 中定义的一致性。其中 `initial` 和 `idle` 在图 3.9 中被定义，`idle` 状态意味着服务器所有的请求均被处理(即被返回，被拒绝或被接受后在所有服务器上被执行。对应于图 2.1b 中的状态②、④或⑤)。一致性定理通过对系统历史轨迹的断言来描述系统运行时的行为，通过此定理可以进一步加深编程人员对语言中各个原语语义的理解。定理 3.1 的证明见附录 A。

### 3.4.5 示例：银行系统同时取钱

这里我们将用上述性质说明在用图 3.2a 中的银行系统作为库函数时，为什么不会产生如图 2.2d 所示的两个取钱操作同时请求后余额为负数的情况。

在代码的静态分析中，操作 `withdraw` 中包含 `sync @acct` 原语，并且其亦对共享变量 `@acct` 做了修改，故可知：

$$\forall \Delta, \text{sync}(\text{withdraw}, \text{withdraw})_{\Delta}.$$

而后，通过定理 3.1，我们可以得到如下命题：

**命题 3.2** 如果轨迹  $\Delta$  满足一致性，那么对于所有的  $\text{id}_A$  和  $\text{id}_B$ , 如果  $\text{sync}(\text{id}_A, \text{id}_B)_{\Delta}$  并且  $\text{sync}(\text{id}_B, \text{id}_A)_{\Delta}$ , 那么  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$  或者  $\text{id}_B \xrightarrow{\text{vo}}_{\Delta} \text{id}_A$ 。

命题 3.2 表示，如果两个请求之间存在双向同步，则其中一个请求在被接受时会看到另一个请求的副作用。命题 3.2 的证明见附录 A.4。

在银行系统的例子中，我们可知两个 `withdraw` 请求之间存在双向同步，

$$\begin{aligned}\Delta.acpts &\stackrel{\text{def}}{=} \{\text{id} \mid (\_, \text{id}, \text{acpt } \_) \in \Delta\} & \Sigma.ops &\stackrel{\text{def}}{=} \text{codom}(\Sigma) \\ \Delta.wr(o) &\stackrel{\text{def}}{=} \{\text{id} \mid o \in \text{wset}(\text{op}(\text{id})_\Delta)\} & \mathbb{U}_\Sigma &\stackrel{\text{def}}{=} \{o \mid o \in \text{wset}(\tau) \cup \text{rset}(\tau) \wedge \tau \in \Sigma.ops\}\end{aligned}$$

图 3.10 定义不同一致性的一些辅助定义

那么其中一个取钱请求在被接受时会看到另一个取钱操作对余额的更新，在此基础上再做是否取钱的判断，则不会发生余额为负数的情况。

## 3.5

接下来我们介绍通过在操作中使用不同的同步集和依赖集，我们可以将服务器的一致性特例化为现有的以数据为中心的一致性之一，这体现了细粒度一致性模型的表达能力。定义 3.8、3.9、3.10 和 3.11 分别给出了因果一致性 (causal consistency)<sup>[57]</sup>、基于变量的因果一致性 (per-object causal consistency)<sup>[55]</sup>、顺序一致性 (sequential consistency)<sup>[13]</sup> 和基于变量的顺序一致性 (per-object sequential consistency)<sup>[65]</sup> 的定义。在接下来的几个小节中，我们将给出为了达到这些一致性，操作的同步集合和依赖集合需要满足什么条件。

### 3.5.1 因果一致性

**定义 3.8** 一个库  $\Sigma$  满足因果一致性，当且仅当对于所有的轨迹  $\Delta$  和服务器  $\Omega \Omega'$ ，如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ ，那么对于所有的  $\text{id}_A$  和  $\text{id}_B$ ，如果  $\text{id}_A \xrightarrow{\text{vo}} \Delta \text{id}_B$ ，那么  $\text{id}_A \xrightarrow{\text{eo}} \Delta \text{id}_B$ 。

定义 3.8 给出了因果一致性的定义，即某个函数库  $\Sigma$  是满足因果一致性的，当且仅当运行得到的历史轨迹  $\Delta$  中，对于其中任何两个请求  $\text{id}_A$  和  $\text{id}_B$ ，如果  $\text{id}_B$  在被接受时看到了  $\text{id}_A$  的副作用，则  $\text{id}_B$  的副作用在每个服务器上执行时，都会看到  $\text{id}_A$  的副作用。

**命题 3.3** 对于任何的库  $\Sigma$ ，如果对于库中的所有操作  $\tau$ （即  $\tau \in \Sigma.ops$ ），都有  $\tau.O_d = \mathbb{U}_\Sigma$ ，那么  $\Sigma$  将满足因果一致性。

为此，如命题 3.3 中所示，我们要求所有操作都依赖于函数库中的所有共享对象。命题 3.3 的证明见附录 B。

### 3.5.2 基于变量的因果一致性

**定义 3.9** 一个库  $\Sigma$  满足基于变量的因果一致性，当且仅当对于所有的轨迹  $\Delta$  和服务器  $\Omega \Omega'$ ，如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ ，那么对于所有的  $\text{id}_A \text{id}_B$ ，如果

$\text{id}_A \xrightarrow{\text{vo}} \text{id}_B$  并且  $\text{wset}(\text{op}(\text{id}_A)_A) \cap \text{rset}(\text{op}(\text{id}_B)_A) \neq \emptyset$ , 那么  $\text{id}_A \xrightarrow{\text{eo}} \text{id}_B$ 。

定义 3.9 给出了基于变量的因果一致性的定义, 即某个函数库  $\Sigma$  是满足基于变量的因果一致性的, 当且仅当运行得到的历史轨迹  $\Delta$  中, 对于其中任何两个请求  $\text{id}_A$  和  $\text{id}_B$ , 如果  $\text{id}_B$  在被接受时看到了  $\text{id}_A$  的副作用并且  $\text{id}_B$  读取了  $\text{id}_A$  写入的变量, 则  $\text{id}_B$  的副作用在每个服务器上执行时, 都会看到  $\text{id}_A$  的副作用。

**命题 3.4** 对于任何的库  $\Sigma$ , 如果对于库中的所有操作  $\tau$  (即  $\tau \in \Sigma.\text{ops}$ ), 都有  $\tau.O_d = \text{rset}(\tau)$ , 那么  $\Sigma$  将满足基于变量的因果一致性。

为此, 如命题 3.4 中所示, 我们要求所有操作都依赖于此操作的读集。命题 3.4 的证明见附录 B。

### 3.5.3 顺序一致性

**定义 3.10** 一个库  $\Sigma$  满足顺序一致性, 当且仅当对于所有的轨迹  $\Delta$  和服务器  $\Omega \Omega'$ , if  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ , 那么将会在  $\Delta.acpts$  集合上存在一个全序关系  $R$ , 使得以下条件均成立:

1. 对于所有的  $\text{id}_A \text{id}_B$ , 如果有  $R(\text{id}_A, \text{id}_B)$ , 那么将会有  $\text{id}_A \xrightarrow{\text{vo}} \text{id}_B$  和  $\text{id}_A \xrightarrow{\text{eo}} \text{id}_B$ .
2. 如果  $\text{id}_A \xrightarrow{\text{po}} \text{id}_B$ , 那么将会有  $R(\text{id}_A, \text{id}_B)$ .

定义 3.10 给出了顺序一致性的定义, 即某个函数库  $\Sigma$  是满足因果一致性的, 当且仅当运行得到的历史轨迹  $\Delta$  中, 所有被接受的请求在所有站点上均按照某个相同的顺序被执行, 同时此顺序不违背程序顺序。

**命题 3.5** 对于所有的库  $\Sigma$ , 如果对于所有库中的操作  $\tau$  (即  $\tau \in \Sigma.\text{ops}$ ), 都有  $\tau.O_s = \tau.O_d = \cup_{\Sigma}$ , 那么  $\Sigma$  将满足顺序一致性。

为此, 如命题 3.5 中所示, 我们要求所有操作都同步并且依赖于函数库中的所有共享对象。命题 3.5 的证明见附录 B。

### 3.5.4 基于变量的顺序一致性

**定义 3.11** 一个库  $\Sigma$  满足基于变量的顺序一致性, 当且仅当对于所有的轨迹  $\Delta$  和服务器  $\Omega \Omega'$ , 如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ , 那么对于所有的对象  $o$ , 在  $\Delta.acpts \cap \Delta.wr(o)$  集合上都存在着一个全序  $R$ , 使得以下条件均成立:

1. 对于所有的  $\text{id}_A \text{id}_B$ , 如果  $R(\text{id}_A, \text{id}_B)$ , 那么  $\text{id}_A \xrightarrow{\text{vo}} \text{id}_B$  并且  $\text{id}_A \xrightarrow{\text{eo}} \text{id}_B$ .
2. 如果  $\text{id}_A \xrightarrow{\text{po}} \text{id}_B$ , 那么  $R(\text{id}_A, \text{id}_B)$ .

定义 3.11 给出了基于变量的顺序一致性的定义，即某个函数库  $\Sigma$  是满足基于变量的顺序一致性的，当且仅当运行得到的历史轨迹  $\Delta$  中，所有被接受并修改了同一共享变量的请求在所有站点上均按照某个相同的顺序被执行，同时此顺序不违背程序顺序。

**命题 3.6** 对于所有的库  $\Sigma$ ，如果对于库的所有操作  $\tau$ （即  $\Sigma \in \Sigma.ops$ ），都有  $\tau.O_s = \tau.O_d = \text{wset}(\tau)$ ，那么  $\Sigma$  将满足基于变量的顺序一致性。

为此，如命题 3.6 中所示，我们要求所有操作都同步并且依赖于此操作的写集。命题 3.6 的证明见附录 B。

### 3.6 本章小结

在本章中，我们首先给出了细粒度一致性模型编程语言的语法，并以银行系统、add-win 集合和拍卖系统为例，展示如何使用此语言对函数库进行设计。此后，我们给出了编程语言的形式化语义并进一步给出了语言的性质。最后，我们展示了如何通过配置语言中的同步集合和依赖集合来保证多种常见的一致性语义。

## 第4章 程序逻辑与验证

为了给编程人员提供更强的可靠性保证，在本章中我们将给出用于验证程序收敛性和满足特定不变量的程序逻辑，并给出相应的可靠性证明。

### 4.1 验证程序满足收敛性的程序逻辑

在本节中，我们将提出一种程序逻辑来验证库的收敛性。

#### 4.1.1 收敛性和可交换性

在第 2.2 节中我们已经给出过收敛性的非形式化描述，并通过图 2.2d 和图 2.2e 的历史轨迹分别展示了满足和不满足收敛性的系统。接下来我们将给出收敛性的形式化定义，如定义 4.1 所示。

**定义 4.1 (收敛性)**  $\models [\Sigma]$  成立当且仅当对于所有的  $\Delta \Omega \Omega'$ ，如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$ ，那么  $[\Omega']$ 。

定义 4.1 表示，如果库  $\Sigma$  是收敛的，当且仅当对于任何调用它的用户程序  $\mathbb{P}$ ，如果系统从一个初始的 `initial` 状态  $\Omega$  运行到任何一个闲置的 `idle` 状态  $\Omega'$ ，则  $\Omega'$  中所有的存储均相同 ( $[\Omega']$  的定义见图 3.9)。

为了证明这一点，一个直观的想法是要求执行过程中的任何两个副作用都是可交换的，即意味着它们两个的执行顺序不影响最终存储的结果。即

$$\begin{aligned} & \forall \text{id id}', (\exists \kappa \kappa', ((\_, \text{id}, \text{acpt } \kappa') \in \Delta \wedge \\ & (\_, \text{id}', \text{acpt } \kappa') \in \Delta) \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa'))). \end{aligned}$$

其中， $\kappa$  和  $\kappa'$  分别为请求 `id` 和 `id'` 在被接受时的上下文， $\tau(\kappa)$  和  $\tau'(\kappa')$  分别为两个请求的副作用，交换性 (`commute`) 的定义则在定义 4.2 中给出。

**定义 4.2 (交换性)**  $\text{commute}(\tau_A(\kappa_A), \tau_B(\kappa_B))$  成立，当且仅当对于所有的  $s s'$ ，存在  $s_1$  使得  $\tau_A(\kappa_A) \vdash s \leftrightarrow s_1$  并且  $\tau_B(\kappa_B) \vdash s_1 \leftrightarrow s'$  成立当且仅当存在  $s_2$  使得  $\tau_B(\kappa_B) \vdash s \leftrightarrow s_2$  并且  $\tau_A(\kappa_A) \vdash s_2 \leftrightarrow s'$  成立。

定义 4.2 表示两个副作用  $\tau_A(\kappa_A)$  和  $\tau_B(\kappa_B)$  满足交换性，当且仅当对于所有的存储  $s$  和  $s'$ ，如果在  $s$  上先执行  $\tau_A$  的副作用再执行  $\tau_B$  的副作用后存储状态变为  $s'$ ，当且仅当在  $s$  上先执行  $\tau_B$  的副作用再执行  $\tau_A$  的副作用后存储状态变为  $s'$ 。定义 4.2 中的  $\tau(\kappa) \vdash s \leftrightarrow s'$  表示在存储  $s$  上执行  $\tau$  在上下文  $\kappa$  的环境下生成的副作用后，新的存储为  $s'$ ，形式化定义在图 4.1 中给出。

在所有站点上，无论以怎样的顺序执行这些副作用，通过所有副作用均可交

$$\begin{aligned}
 [B] &\stackrel{\text{def}}{=} \{s \mid \llbracket B \rrbracket_s = \text{true}\} & \Sigma.\text{trans} &\stackrel{\text{def}}{=} \{\tau \mid \tau \in \Sigma.\text{ops} \wedge \neg \text{is\_query}(\tau)\} \\
 \kappa \in [B] &\stackrel{\text{def}}{=} s \in [B], \text{ 其中 } \kappa = (s, \_) & \text{is\_query}(\tau) &\stackrel{\text{def}}{=} \tau = \dots \text{return} \dots \\
 \tau(s, v) \vdash s' \hookrightarrow s'' &\stackrel{\text{def}}{=} (\llbracket B \rrbracket_{s,r} = \text{true} \Rightarrow (s', r\{x' \rightsquigarrow \llbracket E \rrbracket_{s,r}\}) \xrightarrow{C} (s'', \_)) \vee \\
 &\quad (\llbracket B \rrbracket_{s,\{x \rightsquigarrow v\}} = \text{false} \Rightarrow (s' = s'')) \\
 , \text{ 其中 } r &= \{x \rightsquigarrow v\}, \tau = \text{input } x \text{ sync } O \text{ guard } B \text{ depend } O \text{ let } x' := E \text{ do } C \\
 (s, r) \xrightarrow{C} (s', r') &\stackrel{\text{def}}{=} \begin{cases} s' = s \wedge r' = r & \text{if } C = \text{skip} \\ s' = s\{o \rightsquigarrow \llbracket E \rrbracket_{s,r}\} \wedge r' = r & \text{if } C = o := E \\ s' = s \wedge r' = r\{x \rightsquigarrow \llbracket E \rrbracket_{s,r}\} & \text{if } C = x := E \\ \exists s'' r'', (s, r) \xrightarrow{C_1} (s'', r'') \wedge \\ \quad (s'', r'') \xrightarrow{C_2} (s', r') & \text{if } C = C_1; C_2 \\ \llbracket B \rrbracket_{s,r} = \text{true} \Rightarrow (s, r) \xrightarrow{C_1} (s', r') \vee \\ \quad \llbracket B \rrbracket_{s,r} = \text{false} \Rightarrow (s, r) \xrightarrow{C_2} (s', r') & \text{if } C = \text{if } B \text{ then } C_1 \text{ else } C_2 \\ \llbracket B \rrbracket_{s,r} = \text{false} \Rightarrow (s, r) \xrightarrow{\text{skip}} (s', r') \vee \\ \quad \llbracket B \rrbracket_{s,r} = \text{true} \Rightarrow (s, r) \xrightarrow{C'; C} (s', r') & \text{if } C = \text{while } B \text{ do } C' \end{cases} \\
 \tau(B) \vdash B' \hookrightarrow B'' &\stackrel{\text{def}}{=} \forall \kappa s s', \kappa \in [B] \wedge s \in [B'] \wedge \tau(\kappa) \vdash s \hookrightarrow s' \Rightarrow s' \in [B'']
 \end{aligned}$$

图 4.1 收敛性程序逻辑的一些辅助定义

换这个条件，对于任何的轨迹，我们总能将副作用两两交换，使得所有节点上这些副作用执行顺序均相同（即满足收敛性）。

但是，上述前提条件比较强。考虑到我们在上一章节中对一致性的介绍，我们发现有些请求之间满足定义 3.2 中给出的执行顺序。这意味着每个站点上将以相同顺序执行这些请求的副作用。考虑一个系统按照定义 3.10 所示的顺序一致性运行的极端情况，即系统上的所有请求的副作用均按照相同的顺序在所有站点上被执行。此时，即使任何请求之间都不满足可交换性，系统仍可收敛。通过此分析，我们可将上述要求弱化，即副作用均按照相同的顺序在所有站点上被执行的请求之间不需要满足可交换性。因此，我们只需要证明对于所有的  $\text{id}$  和  $\text{id}'$ ，即

$$\begin{aligned}
 \text{id} \xrightarrow{\text{eo}} \Delta \text{id}' \vee \text{id}' \xrightarrow{\text{eo}} \Delta \text{id} \vee (\exists \kappa \kappa', ((\_, \text{id}, \text{acpt } \kappa') \in \Delta \wedge \\
 (\_, \text{id}', \text{acpt } \kappa') \in \Delta) \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa'))).
 \end{aligned}$$

我们使用  $\mathcal{C}(\text{id}, \text{id}')$  表示  $\text{id}$  和  $\text{id}'$  之间的上述关系，以便后文中使用。除此之外，我们还使用  $\text{id}^\tau$  来表示调用了操作  $\tau$  的请求。

**定理 4.1 (可靠性)** 如果  $\vdash [\Sigma]$ ，那么  $\vDash [\Sigma]$ 。

图 4.2 中给出了用于证明收敛性的逻辑规则。定理 4.1 则说明了其可靠性

$$\begin{array}{c}
 \frac{\forall \tau \tau' \in \Sigma.trans, \vdash \tau, \tau' : [\Sigma]}{\vdash [\Sigma]} \text{ (CV-TOP)} \quad \frac{\vdash_{vis} \tau, \tau' : [\Sigma] \quad \vdash_{par} \tau, \tau' : [\Sigma]}{\vdash \tau, \tau' : [\Sigma]} \text{ (CV-SPLIT)} \\
 \\ 
 \frac{\forall \kappa \kappa', \text{commute}(\tau(\kappa), \tau'(\kappa'))}{\vdash \tau, \tau' : [\Sigma]} \text{ (CV-COM)} \quad \frac{\text{sync}(\tau, \tau')}{\vdash_{par} \tau, \tau' : [\Sigma]} \text{ (CV-SYNC)} \quad \frac{\text{dep}(\tau, \tau')}{\vdash_{vis} \tau, \tau' : [\Sigma]} \text{ (CV-DEP)} \\
 \\ 
 \frac{\forall \kappa \kappa', \text{exclusive}(\tau, \kappa, B)_{\Sigma} \wedge (\kappa' \notin [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))}{\vdash_{par} \tau, \tau' : [\Sigma]} \text{ (CV-EXC)} \\
 \\ 
 \frac{\forall \kappa \kappa', \text{stable}(\tau, \kappa, B)_{\Sigma} \wedge (\kappa' \in [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))}{\vdash_{vis} \tau, \tau' : [\Sigma]} \text{ (CV-STA)}
 \end{array}$$

图 4.2 证明库满足收敛性的逻辑规则

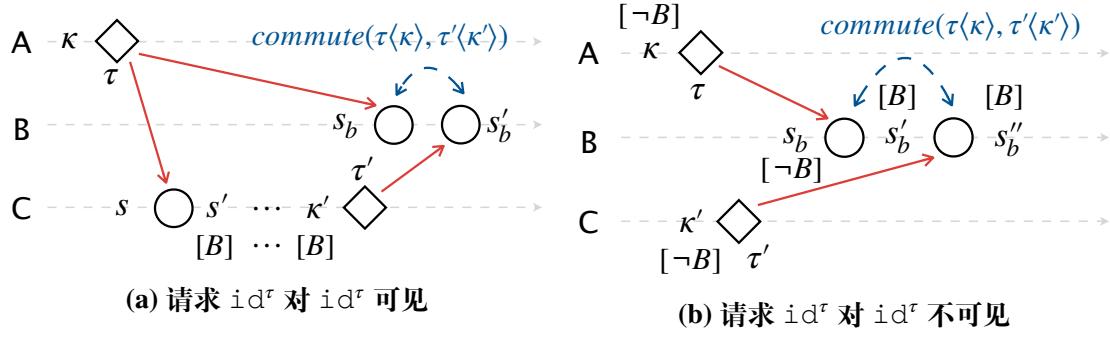


图 4.3 稳定的和唯一的断言含义示意图

(soundness)。定理 4.1 的证明见附录 C。作为证明规则的第一条，规则 CV-TOP 表示，为了保证库  $\Sigma$  是收敛的，我们需要对  $\Sigma$  中的任何两个操作  $\tau$  和  $\tau'$  进行两两检查来确保他们对应的请求  $\text{id}^{\tau}$  和  $\text{id}^{\tau'}$  之间满足  $\mathcal{C}(\text{id}^{\tau}, \text{id}^{\tau'})$ 。如图 4.3 所示，我们将在不同的情况下分析  $\mathcal{C}(\text{id}^{\tau}, \text{id}^{\tau'})$  是否被满足。

由于  $\mathcal{C}(\text{id}^{\tau}, \text{id}^{\tau'})$  具有对称性（即  $\mathcal{C}(\text{id}^{\tau}, \text{id}^{\tau'}) \Leftrightarrow \mathcal{C}(\text{id}^{\tau'}, \text{id}^{\tau})$ ），同时请求之间的仲裁关系为严格全序（即  $\text{id} \xrightarrow{\text{ao}} \text{id}' \vee \text{id}' \xrightarrow{\text{ao}} \text{id}$ ），我们只需要考虑其中一种情况（如在请求  $\text{id}^{\tau}$  先到达、请求  $\text{id}^{\tau'}$  后到达）即可，即  $\text{id}^{\tau} \xrightarrow{\text{ao}} \text{id}^{\tau'} \Rightarrow (\text{id}^{\tau} \xrightarrow{\text{eo}} \text{id}^{\tau'} \vee \text{id}^{\tau'} \xrightarrow{\text{eo}} \text{id}^{\tau} \vee (\exists \kappa \kappa', ((\_, \text{id}^{\tau}, \text{acpt} \kappa') \in \Delta \wedge (\_, \text{id}^{\tau'}, \text{acpt} \kappa') \in \Delta) \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa'))))$ 。

在此条件下，我们将根据两个操作的可见性进行分情况讨论，两种情况分别为  $\text{id}^{\tau} \xrightarrow{\text{vo}} \text{id}^{\tau'}$ 、 $\neg \text{id}^{\tau} \xrightarrow{\text{vo}} \text{id}^{\tau'}$ ，如图 4.3 所示。

### 4.1.2 稳定的存储状态

我们首先考虑图 4.3(a) 中的情况，即请求  $\text{id}^{\tau}$  的副作用对  $\text{id}^{\tau'}$  是可见的。对于这种情况，我们有两个相应的规则，即规则 CV-DEP 和规则 CV-STA。规则 CV-DEP

表示，如果操作  $\tau'$  依赖于操作  $\tau$ （即  $\text{dep}(\tau, \tau')$ ），由定理 3.1 我们可以得到  $\text{id}^\tau \xrightarrow{\text{eo}} \Delta \text{id}^{\tau'}$ ，因此我们不需要其他额外条件便可得到  $\mathcal{C}(\text{id}^\tau, \text{id}^{\tau'})$ 。

如果操作  $\tau'$  不依赖于操作  $\tau$ （即  $\neg\text{dep}(\tau, \tau')$ ），由于副作用  $\tau(\kappa)$  和  $\tau'(\kappa')$  在所有站点上执行顺序可能会不相同，我们需要保证  $\text{commute}(\tau(\kappa), \tau'(\kappa'))$  成立。如规则 cv-STA 所示，如果存在一个断言  $B$  使得其能够被执行完副作用  $\tau(\kappa)$  之后的存储  $s'$  所满足，并且任何其他请求的副作用都不能使得  $s'$  满足  $\neg B$ ，那么我们知道在  $\text{id}^\tau \xrightarrow{\text{vo}} \Delta \text{id}^{\tau'}$  条件下，请求  $\text{id}^{\tau'}$  的上下文  $\kappa'$  将会满足  $B$ 。因此，我们只需要保证  $\text{commute}(\tau(\kappa), \tau'(\kappa'))$  在条件  $\kappa' \in [B]$  下被满足即可。我们将上述  $\tau, \kappa$  和  $B$  的关系定义为稳定的（用  $\text{stable}(\tau, \kappa, B)_\Sigma$  表示），如定义 4.3 所示。如果这样的断言  $B$  并不存在，我们则应用规则 cv-COM。

**定义 4.3** (稳定的)  $\text{stable}(\tau(\kappa), B)_\Sigma$  成立当且仅当以下条件均成立：

1.  $\forall s s', \tau(\kappa) \vdash s \hookrightarrow s' \Rightarrow s' \in [B]$ 。
2.  $\forall \tau' \in \Sigma.\text{trans}, \tau'(\text{true}) \vdash B \hookrightarrow B$ 。

定义 4.3 表示，库  $\Sigma$  中操作  $\tau$  的副作用  $\tau(\kappa)$  在断言  $B$  下是稳定的（用  $\text{stable}(\tau(\kappa), B)_\Sigma$  表示），当且仅当：(1) 执行完副作用  $\tau(\kappa)$  后，新的存储  $s'$  满足  $B$ ，(2)  $s'$  在执行完其他副作用后，依然满足  $B$ 。

这个断言用来解决那些有单调增长 (grow-only) 的共享对象的情况。例如，在图 3.2c 所示的拍卖系统中，共享变量 `@clients` 的定义域是单调增长的。这意味着如果有一个副作用将用户名 `cname` 被放入其中，那么此副作用在断言 `cname ∈ dom(@clients)` 下将是稳定的。

### 4.1.3 独一的请求

对于图 4.3(b) 中  $\text{id}^{\tau'}$  没有见到  $\text{id}^\tau$  副作用的情况，根据操作  $\tau'$  是否同步操作  $\tau$ ，我们有两个相应的规则，即规则 cv-EXC 和 cv-SYNC。规则 cv-SYNC 表示，如果  $\tau'$  同步于操作  $\tau$ （即  $\text{sync}(\tau, \tau')$ ），由于我们有条件  $\text{id}^\tau \xrightarrow{\text{ao}} \Delta \text{id}^{\tau'}$ ，根据定理 3.1 便可以得到  $\text{id}^\tau \xrightarrow{\text{vo}} \Delta \text{id}^{\tau'}$ ，即  $\text{id}^\tau$  的副作用对  $\text{id}^{\tau'}$  可见，说明图 4.3(b) 所示的情况不会发生。

如果操作  $\tau'$  不同步于操作  $\tau$ （即  $\neg\text{sync}(\tau, \tau')$ ），由于副作用  $\tau(\kappa)$  和  $\tau'(\kappa')$  在所有站点上执行顺序可能会不相同，我们需要保证  $\text{commute}(\tau(\kappa), \tau'(\kappa'))$  成立。如规则 cv-EXC 所示，如果存在一个断言  $B$  使得其能且只能被执行完副作用  $\tau(\kappa)$  之后的存储  $s'$  所满足，那么我们知道如果  $\neg\text{id}^\tau \xrightarrow{\text{vo}} \Delta \text{id}^{\tau'}$ ，请求  $\text{id}^{\tau'}$  的上下文  $\kappa'$  将不会满足  $B$ 。因此，我们只需要保证  $\text{commute}(\tau(\kappa), \tau'(\kappa'))$  在条件  $\kappa' \in [\neg B]$  下被满足即可。我们将上述  $\tau, \kappa$  和  $B$  的关系定义为唯一的（用  $\text{exclusive}(\tau, \kappa, B)_\Sigma$  表示），如定义 4.4 所示。如果这样的断言  $B$  并不存在，我们

则应用规则 cv-COM。

**定义 4.4**  $\text{exclusive}(\tau(\kappa), B)_{\Sigma}$  成立当且仅当以下条件均成立：

1.  $\kappa \in [\neg B] \wedge \text{stable}(\tau, \kappa, B)_{\Sigma}$ 。
2.  $\forall \tau' \kappa', (\tau' \neq \tau \vee \kappa' \neq \kappa) \Rightarrow \tau'(\kappa') \vdash \neg B \leftrightarrow \neg B$ 。

定义 4.4 表示，库  $\Sigma$  中操作  $\tau$  的副作用  $\tau(\kappa)$  在断言  $B$  下是唯一的（用  $\text{exclusive}(\tau(\kappa), B)_{\Sigma}$  表示）当且仅当副作用  $\tau(\kappa)$  可以将存储从满足  $\neg B$  的状态转换到满足  $B$  的状态，而其他副作用只能维持这个状态，即一直保持满足  $B$  的状态或一直保持满足  $\neg B$  的状态。为了实现这一点，第一个条件要求上下文  $\kappa$  满足  $\neg B$ ，并且在  $B$  上是稳定的（如果没有此要求，那么可能在  $s$  上再一次产生满足  $B$  的状态，从而使得副作用  $\tau(\kappa)$  可以再次被生成）。第二个条件要求其他副作用不能在满足  $\neg B$  的存储状态下生成一个能使其满足  $B$  的副作用。此时，如果对于同样的调用  $\tau$  的请求，如果上下文不满足  $\neg B$ （可能是系统初始时已经不满足，也可能是因为其他请求的副作用将存储从  $\neg B$  转换成  $B$  状态），根据第二条规则，此时  $\tau$  的副作用将不能使得存储从  $\neg B$  的状态转换成  $B$  状态，这就保证了其他请求的唯一性。

这个断言用于共享对象的增长只能被单个副作用所触发的场景。例如，在图 3.2b 所示的 add-win 集合中，由于每个 add 操作的操作有一个唯一标识符 id，因此在添加元素 e 时，此副作用在断言  $\langle e, id \rangle \in @sets$  下将是唯一的。

#### 4.1.4 示例 1：验证拍卖系统的收敛性

在这里我们只考虑证明拍卖系统收敛性过程中的一个最令人关注的情况，即在证明当  $\tau = \tau' = \text{register\_user}$  时，我们有  $\vdash \tau, \tau' : [\Sigma]$ 。

在注册用户时，如果两个不同的用户选择了相同的用户名同时进行注册操作，由于用户名不可重复，故操作  $\text{register\_user}$  和其自身不具有可交换性，即

$$\forall cname cinfo cinfo', cinfo \neq cinfo' \Rightarrow \neg \text{commute}(\text{register\_user}, (\_, \langle cname, cinfo \rangle), \text{register\_user}, (\_, \langle cname, cinfo' \rangle)),$$

因此，我们不能直接应用规则 cv-COM。但我们知道如果第二个请求可以看到第一个请求的副作用，那么第一个请求的用户名  $cname$  已经被注册。我们先证明

$$\begin{aligned} & \forall cname s, s \in [cname \notin @clients.keys()] \Rightarrow \\ & \quad \text{stable}(\text{register\_user}, (s, \langle cname, \_ \rangle), cname \in @clients.keys())_{\Sigma}. \end{aligned}$$

此后，如果第二个操作产生的副作用上下文满足  $cname \in @clients.keys()$ ，那么这个副作用将会和第一个操作的副作用可交换，即

$$\forall \kappa' cname, \kappa' \in [cname \in @clients.keys()] \Rightarrow$$

$\text{commute}(\text{register\_user}, (\_, \langle cname, \_ \rangle), \text{register\_user}, \kappa')$ 。

通过规则 CV-STA、CV-SYNC 和 CV-SPLIT，即可得  $\vdash \text{register\_user}, \text{register\_user} : [\Sigma]$ 。

#### 4.1.5 示例 2：验证 add-win 集合的收敛性

在这里我们只考虑证明拍卖系统收敛性过程中的一个最令人关注的情况，即在证明当  $\tau = \text{add}$  和  $\tau' = \text{remove}$  时，我们有  $\vdash \tau, \tau' : [\Sigma]$ 。我们用集合  $\text{used\_id}$  来代表所有在执行过程中使用过的标识符。

如果  $\text{add}$  操作和  $\text{remove}$  操作添加或者删除同一个元素，那么他们之间将不具有可交换性，即

$$\forall id\ e\ s, \langle id, e \rangle \in s(@set) \Rightarrow \neg \text{commute}(\text{add}, (\_, \langle id, e \rangle), \text{remove}, (s, e)).$$

因此，我们不能直接应用规则 CV-COM。

但我们知道，如果  $\text{remove}$  操作没有看到  $\text{add}$  操作的副作用，那么它将不会删除  $\text{add}$  操作所添加的元素。我们先证明

$$\forall id\ s, s \in [id \notin \text{used\_id}] \Rightarrow \text{exclusive}(\text{add}, (s, \langle id, \_ \rangle), id \in \text{used\_id})_\Sigma.$$

此后，如果  $id \notin \text{used\_id}$ ，那么带有此标识符的数据将不会在  $\text{remove}$  操作要移除的数据集合中，因此  $\text{add}$  and  $\text{remove}$  的副作用将会可交换，即

$$\forall \kappa' id, \kappa' \notin [id \in \text{used\_id}] \Rightarrow \text{commute}(\text{add}, (\_, \langle id, \_ \rangle), \text{remove}, \kappa').$$

通过规则 CV-EXC，可知  $\vdash_{\text{par}} \text{add}, \text{remove} : [\Sigma]$ 。

由于  $\text{dep}(\text{add}, \text{remove})$ ，通过规则 CV-DEP 我们知道  $\vdash_{\text{vis}} \text{add}, \text{remove} : [\Sigma]$ 。

因此，通过规则 CV-SPLIT，即可得  $\vdash \text{add}, \text{remove} : [\Sigma]$ 。

## 4.2 验证程序满足特定不变量的程序逻辑

在本节中，我们将提出一种程序逻辑来验证库满足特定的不变量。

### 4.2.1 不变量

在第 2.2 节中我们已经给出过不变量的定义，并通过图 2.2b 和图 2.2c 分别展示了银行系统中满足和不满足余额不为负数不变量的历史轨迹。接下来我们将给出系统满足特定不变量的形式化定义，如定义 4.1 所示。

**定义 4.5**  $\vdash \Sigma : I$  成立，当且仅当对于所有的库  $\Delta$  和系统  $\Omega \Omega'$ ，如果  $\Omega \models I$  并且  $\Sigma \vdash \Omega \xrightarrow{\Delta} \Omega'$ ，那么  $\Omega' \models I$ 。

如定义 4.5 所示，一个库  $\Sigma$  满足不变量  $I$ ，当且仅当服务器的初始状态满足  $I$ ，接下来无论客户端调用什么操作，服务器始终满足  $I$ 。其中，不变量  $I$  是作用在存储上的断言； $\Omega \models I$  表示系统  $\Omega$  满足不变量，具体见图 D.1 中的定义。

$$\begin{array}{c}
 \frac{\vdash [\Sigma] \quad \forall \tau \in \Sigma.trans, \vdash \tau : I \quad \forall \tau \tau' \in \Sigma.trans, \vdash \tau, \tau' : I}{\vdash \Sigma : I} \text{ (INV-TOP)} \\
 \\ 
 \frac{\mathbf{safe}(\tau')_I}{\vdash \tau, \tau' : I} \text{ (INV-SAFE)} \quad \frac{\forall s' v, s \in [I] \wedge \tau(s, v) \vdash s \hookrightarrow s' \Rightarrow s' \in [I]}{\vdash \tau : I} \text{ (INV-SEQ)} \\
 \\ 
 \frac{\vdash_{vis} \tau, \tau' : I \quad \vdash_{par} \tau, \tau' : I}{\vdash \tau, \tau' : I} \text{ (INV-SPLIT)} \quad \frac{\mathbf{vo\_independent}(\tau, \tau')_I}{\vdash_{par} \tau, \tau' : I} \text{ (INV-VO)} \\
 \\ 
 \frac{\mathbf{eo\_independent}(\tau, \tau')_I}{\vdash_{vis} \tau, \tau' : I} \text{ (INV-EO)} \quad \frac{\mathbf{sync}(\tau, \tau')}{\vdash_{par} \tau, \tau' : I} \text{ (INV-SYNC)} \quad \frac{\mathbf{dep}(\tau, \tau')}{\vdash_{vis} \tau, \tau' : I} \text{ (INV-DEP)}
 \end{array}$$

图 4.4 证明库满足不变量的逻辑规则

$$\begin{array}{ll}
 (Inv) \quad I \in BExp & \mathbf{safe}(\tau)_I \stackrel{\text{def}}{=} \tau \langle I \rangle \vdash I \hookrightarrow I \\
 \Omega \models I \stackrel{\text{def}}{=} \forall \iota, \mathbf{store\_of}(\iota)_\Omega \in [I] & \\
 \tau_A \langle s_A \rangle \leqslant_I \tau_B \langle s_B \rangle \stackrel{\text{def}}{=} \forall s' v_A, \tau_A \langle s_A, v_A \rangle \vdash s \hookrightarrow s' \Rightarrow \exists v_B, \tau_B \langle s_B, v_B \rangle \vdash s \hookrightarrow s'
 \end{array}$$

图 4.5 验证不变量的程序逻辑的一些辅助定义

**定理 4.2** (可靠性) 如果  $\vdash \Sigma : I$ , 那么  $\models \Sigma : I$ 。

在图 4.4 中给出了用于证明库满足不变量的逻辑规则。定理 4.2 则说明了其可靠性 (soundness)。定理 4.2 的证明见附录 D。作为证明规则的第一条, 规则 INV-TOP 有 3 个前提。第一个前提要求库满足收敛性。第二个前提要求在一个最基本的情况下, 即所有请求都按照相同的顺序来被接受和被执行 (例如图 4.7(f) 所示的历史轨迹) 时, 库满足给定的不变量。在这种情况下, 每个请求都可被认为是原子的, 多个服务器可以等价为单个服务器的串行执行。如规则所示, 我们要求库中的每个操作  $\tau$  都满足  $\vdash \tau : I$ , 即对于所有的存储  $s$ 、 $s'$  和值  $v$ , 如果  $s$  满足不变量, 并且操作  $\tau$  在上下文  $\langle s, v \rangle$  下生成的副作用, 在存储  $s$  上执行后, 新的存储  $s'$  仍然满足不变量  $I$ 。

第三个前提则用于处理站点副作用在执行时乱序的情况 (即乱序执行)。如规则所示, 我们要求库中的每两个操作  $\tau$   $\tau'$  之间都满足  $\vdash \tau, \tau' : I$ 。如图 4.6(1) 所示, 请求  $\text{id}^{\tau'}$  在站点  $A$  上被接受, 它的副作用被广播到站点  $B$  上被执行。我们考虑站点  $B$  在执行完此副作用后, 新的存储是否满足不变量。我们在第 3.4 节中提到过, 请求  $\text{id}^{\tau'}$  在接受时和副作用被执行时的事件的历史可能会不同 (在图 4.6(1) 中表示为  $h_A \neq h_B$ )。这会导致此请求站点  $\iota_A$  上被接受的时刻所满足的性质, 不一定在其副作用在  $\iota_B$  站点上被执行时所满足。

然而, 如果我们能够保证  $h_A \neq h_B$  情况下的执行能够等价于  $h_A = h_B$  (如图

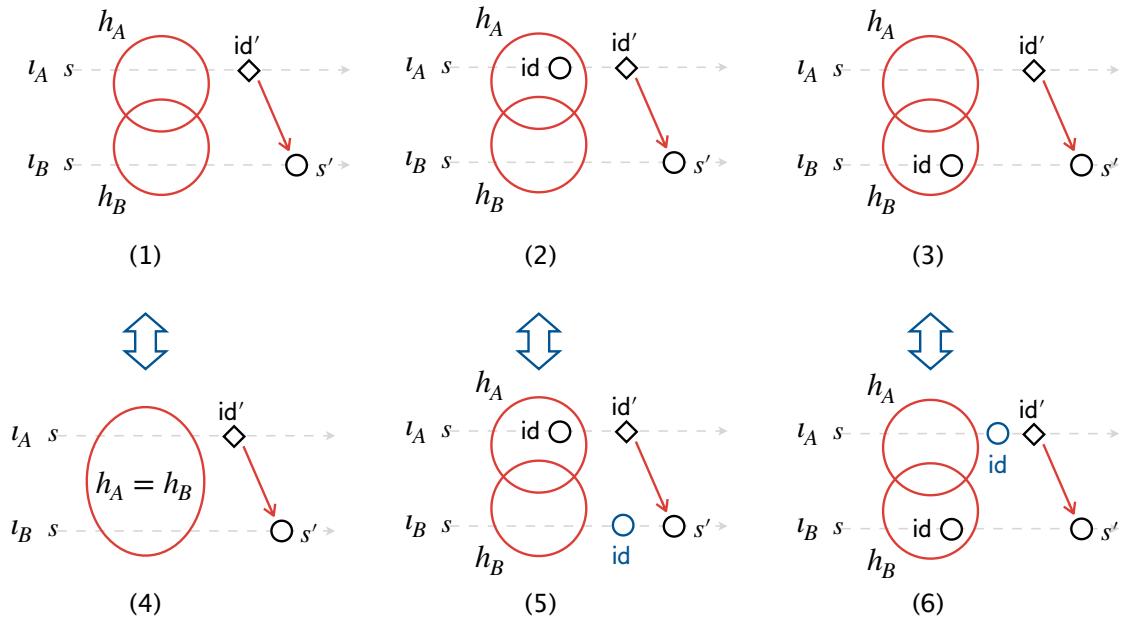


图 4.6 不变量各个逻辑规则所对应的情况

4.6(4) 所示), 即对于所有乱序执行, 都存在着一个串行执行, 使得当串行执行满足不变量时, 乱序执行都能够满足不变量), 那么通过规则 INV-SEQ 可知, 所有的乱序执行都满足不变量。

规则 INV-TOP 的第三个前提给出了保证所有如图 4.6(1) 所示的执行都能够找到如图 4.6(4) 所示的等价执行的条件。为了保证这一点, 规则 INV-SAFE 表示, 如果  $\tau'$  是安全的 (即这个操作在任何上下文中产生的副作用在任何满足不变量的存储下执行后, 新的存储都满足不变量, 用 safe 表示), 那么我们不再需要其它额外的条件。否则, 我们必须在如下所示的两个例子中, 保证  $\tau'$  副作用的执行不会造成不变量的违背:

- 如图 4.6(2) 所示, 请求  $\text{id}^\tau$  的副作用在历史  $h_A$  中而不在历史  $h_B$  中。
- 如图 4.6(3) 所示, 请求  $\text{id}^\tau$  的副作用在历史  $h_B$  中而不在历史  $h_A$  中。

规则 INV-SPLIT 中的两个条件分别对应于上述两种情况。在接下来的介绍中, 我们使用  $e^\tau$  来表示  $\tau$  的副作用, 使用  $I(e, I)$  来表示在执行完副作用  $e$  后, 新的存储满足不变量  $I$ 。

## 4.2.2 独立执行

对于图 4.6(2) 中的情形, 取决于操作  $\tau'$  是否依赖于操作  $\tau$ , 我们有两条规则, 即规则 INV-DEP 和规则 INV-EO。规则 INV-DEP 表示如果操作  $\tau'$  依赖于操作  $\tau$ , 那么我们不再需要其他额外前提。原因是, 如果我们有  $\text{id} \xrightarrow{\text{vo}}_A \text{id}'$ , 根据定理 3.1, 我们知道  $\text{id} \xrightarrow{\text{eo}}_A \text{id}'$ , 这表示这种情况不会发生。

否则,  $e^{\tau'}$  可能会因为没有看到操作  $\tau$  的副作用而违背不变量。为了判断  $\tau$

的副作用是否会对  $e^\tau$  产生影响，如规则 INV-EO 所示，我们只需要给出如下断言：如果操作  $\tau'$  的所有副作用在看到了  $\tau$  的副作用时可以被安全执行（即执行后的存储满足不变量），都能够在看不到  $\tau$  的副作用时被安全的执行，那么我们知道  $\tau$  的副作用对  $\tau'$  没有影响，即  $\tau'$  的副作用在不变量  $I$  下相对于  $\tau$  是独立执行的（用  $\text{eo\_independent}(\tau, \tau')_I$  表示），如定义 4.6 所示。此时图 4.6(2) 的情形便可以等价为图 4.6(4) 所示的情形。

**定义 4.6 (独立执行)**  $\text{eo\_independent}(\tau_A, \tau_B)_I$  成立当且仅当对于所有的  $\kappa_A s_A s'_A$ ，如果  $\tau_A(\kappa_A) \vdash s_A \hookrightarrow s'_A$ ，那么对于所有的  $\kappa_B s_B s'_B$ ，如果  $\tau_B(\kappa_B) \vdash s'_A \hookrightarrow s'_B \Rightarrow s'_B \in [I]$ ，那么  $\tau_B(\kappa_B) \vdash s_A \hookrightarrow s_B \Rightarrow s_B \in [I]$ 。

### 4.2.3 独立接受

对于图 4.6(3) 中的情形，取决于操作  $\tau'$  是否同步操作  $\tau$ ，我们有两条规则，即规则 INV-VO 和规则 INV-SYNC。规则 INV-SYNC 表示如果操作  $\tau'$  同步操作  $\tau$ ，那么我们不再需要其他额外前提。具体原因可见附录 D 中的证明。

否则， $e^{\tau'}$  可能会因为  $\tau'$  在接受时没有看到操作  $\tau$  的副作用而违背不变量。为了判断  $\tau$  的副作用是否会对  $\tau$  生成的副作用产生影响，如规则 INV-VO 所示，我们只需要给出如下断言：如果操作  $\tau'$  看到了  $\tau$  的副作用时所生成的所有副作用，都能够在看不到  $\tau$  的副作用时被生成，那么我们知道  $\tau$  的副作用对  $\tau'$  生成怎样的副作用没有影响，即  $\tau'$  不变量  $I$  下相对于  $\tau$  是独立接受的（用  $\text{vo\_independent}(\tau, \tau')_I$  表示），如定义 4.7 所示。此时图 4.6(3) 的情形便可以等价为图 4.6(6) 所示的情形。

**定义 4.7 (独立接受)**  $\text{vo\_independent}(\tau_A, \tau_B)_I$  成立当且仅当对于所有的  $\kappa_A s_A s'_A$ ，如果  $\tau_A(\kappa_A) \vdash s_A \hookrightarrow s'_A$ ，那么  $\tau_B(s_A) \leq_I \tau_B(s'_A)$ 。

通过独立接受和独立执行给出的两种等价方式，我们便可以逐步的将一个乱序执行的轨迹等价成为一个串行执行的历史轨迹。图 4.7 给出了一个包含三个请求的乱序历史轨迹通过变换等价为一个串行历史轨迹的例子，由于 INV-SEQ 规则保证了串行的执行满足不变量，我们可以证明乱序的历史轨迹满足不变量。

### 4.2.4 示例：银行系统账户余额不小于 0 的条件

#### (1) 对共享变量 @acct 同步的必要性

首先，我们证明为了使得银行系统账户余额不小于 0，在 withdraw 请求中对账户 @acct 的同步是必要的。根据证明不变量的逻辑规则，在此我们只考虑证明过程中的一个最令人关注的情况，即：

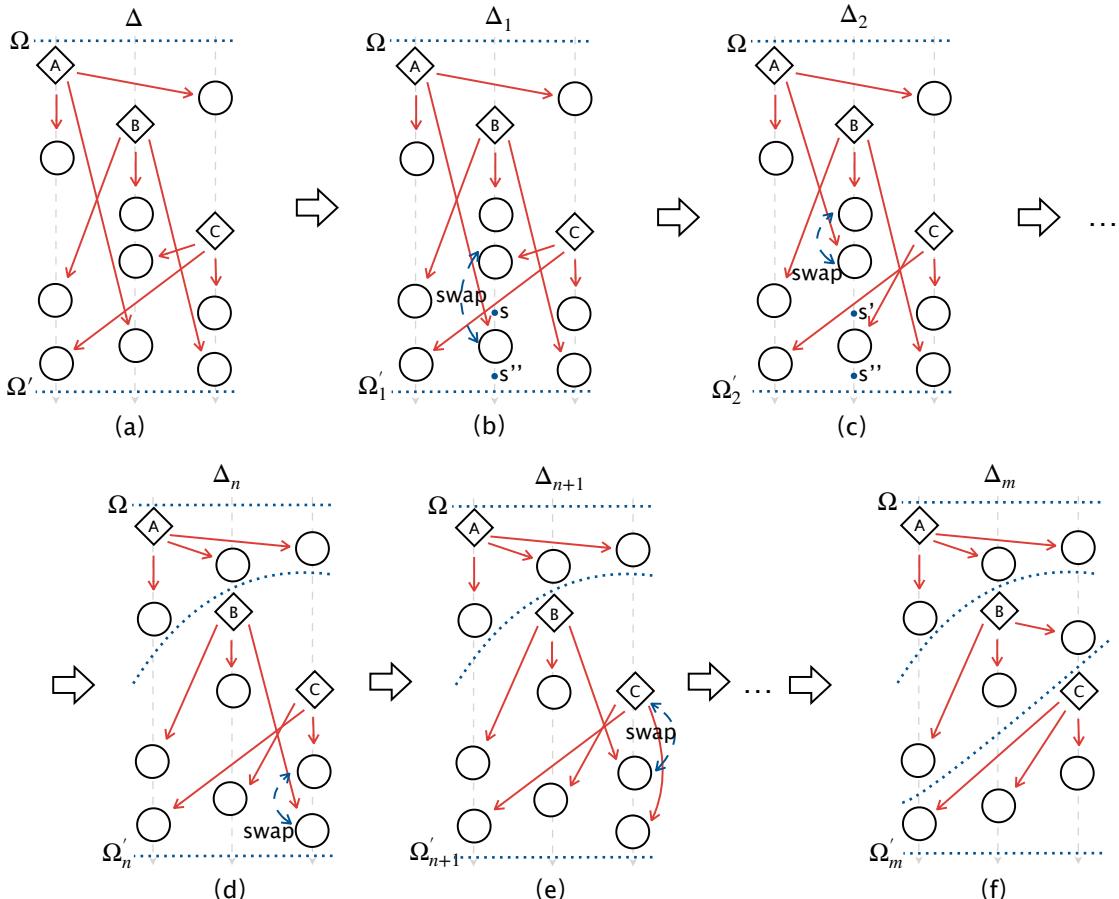


图 4.7 通过等价变换将乱序执行的轨迹转换为串行执行的轨迹

 $\neg \text{vo\_independent}(\text{withdraw}, \text{withdraw})_I,$ 

其中不变量  $I$  的定义为：

$$I \stackrel{\text{def}}{=} \forall cid \in \text{dom}(@accts), @accts.get(cid) \geq 0.$$

为了证明这个目标，通过定义 4.7，我们只需要找到操作 `withdraw` 的一个副作用，使得其不能够在看到另一个 `withdraw` 操作副作用的时候被产生，但是能够在看不到这个 `withdraw` 操作副作用时产生。即

$$\exists \kappa_A s_A s'_A, \kappa_A \in [I] \wedge s_A \in [I] \wedge s'_A \in [I] \wedge$$

$$\text{withdraw}(\kappa_A) \vdash s_A \hookrightarrow s'_A \wedge \neg \text{withdraw}(s_A) \leq_I \text{withdraw}(s'_A).$$

令  $\kappa_A = (s_A, k)$ ,  $s_A \in [\text{@accts.get}(cid) = k]$ ,  $s'_A \in [\text{@accts.get}(cid) = 0]$ , 其中  $k > 0$ .

我们只需要证明：

$$\exists s s', s \in [I] \wedge s' \in [I] \wedge \text{withdraw}(s_A) \vdash s \hookrightarrow s' \wedge \neg \text{withdraw}(s'_A) \vdash s \hookrightarrow s'.$$

令  $s \in [\text{@accts.get}(cid) = k]$ ,  $s' \in [\text{@accts.get}(cid) = 0]$ , 我们知道这个声明成立。

## (2) 对共享变量 `@acct` 依赖的必要性

除此之外，我们还需要证明为了使得银行系统账户余额不小于 0，在 `withdraw` 请求中对账户 `@acct` 的依赖也是必要的。根据证明不变量的逻辑规则，只考虑证明过程中的一个最令人关注的情况，即：

---

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.
```

---

图 4.8 在 Coq 中定义自然数

$\neg \text{eo\_independent}(\text{deposit}, \text{withdraw})_I$ 。

为了证明这个目标，通过定义 4.6，我们只需要找到一个存储  $s$  和两个副作用  $\text{deposit}(\kappa_A)$ 、 $\text{withdraw}(\kappa_B)$ ，使得一个具有存储  $s$  的节点可在分别执行副作用  $\text{deposit}(\kappa_A)$  和副作用  $\text{withdraw}(\kappa_B)$  后不违背不变量，但在仅仅执行副作用  $\text{withdraw}(\kappa_B)$  后却违背不变量。即证明：

$$\begin{aligned} \exists \kappa_A \kappa_B s_A s_B s'_A s'_B, \kappa_A \in [I] \wedge \kappa_B \in [I] \wedge s_A \in [I] \wedge s'_A \in [I] \wedge \\ \text{deposit}(\kappa_A) \vdash s_A \hookrightarrow s'_A \wedge \\ (\text{withdraw}(\kappa_B) \vdash s'_A \hookrightarrow s'_B \Rightarrow s'_B \in [I]) \wedge \\ \text{withdraw}(\kappa_B) \vdash s_A \hookrightarrow s_B \wedge s_B \notin [I]. \end{aligned}$$

令

$$\begin{aligned} \kappa_A = (s_A, k) \wedge \kappa_B = (s'_A, k) \wedge s_A \in [\text{@accts.get}(cid) = 0] \wedge \\ s'_A \in [\text{@accts.get}(cid) = k] \wedge s_B \in [\text{@accts.get}(cid) = -k] \wedge \\ s'_B \in [\text{@accts.get}(cid) = 0], \end{aligned}$$

其中  $k \geq 0$ ，我们知道此声明成立。

## 4.3 使用 Coq 验证程序逻辑的正确性

### 4.3.1 Coq 简介

Coq 是法国国家科学研究中心等机构开发的交互式定理证明辅助工具。它提供了一套标准语言，允许用户将数学表达式、程序语言、算法等的定义形式化，并基于这些定义构造形式化证明。Coq 在定理证明中得到了广泛的应用。

#### (1) 归纳定义

在 Coq 中，“Inductive” 用于归纳和定义类型。归纳定义通过名称、类型、规则来定义归纳类。每个规则都有一个唯一的名称，称为“构造函数”。代码 4.8 是一个自然数的定义，其中  $O$  是一个自然数， $S$  是一个构造函数，它的输入类型是自然数，返回类型也是自然数。即如果  $n$  是自然数，那么  $S n$  也是自然数。

#### (2) 定义非递归函数和递归函数

在 Coq 中，“Definition” 和 “Fixpoint” 分别用于定义非递归函数和递归函数。代码 4.9 显示了用 “Definition” 定义的自然数前驱函数。在 Coq 中，递归函数的定义需要显示或隐式的提供一个在结构上递减的递归参数来保证函数的终止性。

---

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
```

---

图 4.9 在 Coq 中定义自然数前驱

---

```
Fixpoint plus (n: nat) (m: nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
end.
```

---

图 4.10 在 Coq 中定义自然数相加

---

```
Theorem le_0_2:le 0 2.
Proof.
  (* le 0 2 *)
  apply le_S.

  (* le 0 1 *)
  apply le_S.

  (* le 0 0 *)
  apply le_n.
Qed.
```

---

图 4.11 在 Coq 中证明 0 小于等于 2

代码 4.10 给出了由“Fixpoint”定义的自然数加法函数。其中 n 为递归参数。

### (3) 证明对象和证明脚本

证明对象 (proof object) 是 Coq 的核心组成部分。当 Coq 运行证明脚本 (proof script) 时，它将逐步构造证明项 (proof term)。证明项目的类型是待证明的命题 (proposition)。“Proof” 和 “Qed” 之间的证明策略 (proof tactic) 给出了构造证明项的过程。代码 4.11 给出了 0 小于等于 2 (即  $\text{le } 0 \ 2$ ) 的一个证明。

## 4.3.2 使用 Coq 验证收敛性和不变量程序逻辑的可靠性

我们使用定理证明工具 Coq 验证以上两个程序逻辑的可靠性。图 4.12 给出了验证收敛性和不变量程序逻辑可靠性的 Coq 代码。我们在 Coq 中对上述定理进行证明，证明共包含约 3100 行 Coq 脚本（通过 cloc 软件测量<sup>[66]</sup>）、34 个引

---

```
Theorem Convergence_Soundness:  
  forall (lib : Library),  
  |- |_ lib _| -> |= |_ lib _| .  
  
Theorem Invariant_Soundness:  
  forall (lib : Library) (inv : Inv),  
  |- lib :- inv -> |= lib :- inv.
```

---

图 4.12 在 Coq 中给出程序逻辑的可靠性定理

理、以及 94 个定义和变量。附录 C 和附录 D 中给出了证明的一些主要引理的简要证明。

#### 4.4 本章小结

在本章中，我们首先给出了用于证明系统收敛性的程序逻辑。在相应的逻辑规则中通过定义稳定的存储状态和唯一的请求来弱化所有请求均可交换的条件。其次，我们给出了用于证明系统满足特定不变量的程序逻辑。在逻辑规则中，独立接受和独立执行的断言保证了乱序执行的轨迹可以等价为一个串行执行的轨迹。最后，我们将上述两个程序逻辑可靠性证明的部分关键引理在定理证明工具 Coq 中进行验证。

## 第 5 章 细粒度一致性协议与系统的设计和实现

回顾第 3 章中，我们给出了客户端和服务器端语言的语法、抽象的状态模型，以及用此语言写出的库实例是如何与抽象机器进行交互的（即状态转换）。由于客户端使用的语言和状态模型较为简单，在本章我们主要介绍服务器端状态模型 ( $\Omega$ )、服务器端库实例 ( $\Sigma$ ) 是如何在一个真实的分布式系统上实现的，以及图 3.6 所示的服务器端状态转换规则是如何在此基础上进行进一步细化的。服务器端状态模型的实现在第 5.1 小节中给出。第 5.2 节会对以单个请求为例，对服务器端细化后的状态转换规则进行介绍。除此之外，为了保证服务器端逻辑的正确性，我们使用模型检查工具 TLA+<sup>[67]</sup> 对服务器端建模并进行死锁和活性检查，这部分内容将在第 5.3 节介绍。最后，我们在第 5.4 节给出服务器端库实例的具体实现方式以及整个系统实现层面的一些细节。

### 5.1 服务器端系统设计的几个关键问题

我们已经在第 3.3 节中给出了抽象机上的操作语义。在实现的过程中，系统需要在分布式环境下运行，其与抽象的状态机有一定差距。因此抽象机器中定义的一些全局状态，如全局消息缓冲区 ( $M$ ) 和仲裁顺序（包含在全局历史记录  $h$  中）等，在真实机器中需要进行进一步实现。我们将在第 5.1.1 小节和第 5.1.2 小节中分别讲述上述两个全局状态是怎样通过在每个机器上的本地状态和特定的消息传递协议进行实现的。除此之外，在真实机器上并不能像抽象机器那样有无限的存储空间，可将所有请求和消息存储在变量中。但同步集合和依赖集合使得请求之间具有相关性，我们不能简单的直接删除他们。因此，我们需要一种垃圾回收机制。这个回收机制将在第 5.1.3 小节中给出。

#### 5.1.1 如何模拟全局消息缓冲区？

在图 3.6b 给出的服务器端状态转换规则中，机器模型中存在一个抽象的全局消息缓冲区  $M$ 。此缓冲区用于放置每个请求  $id$  所具有的消息，而后再根据消息的类型进行相应状态转换。在真实分布式系统中，我们不存在这样的全局变量。因此，我们通过每个服务器之间的通信来模拟全局消息缓冲区。我们在每一个服务器站点上均设置一个类似于消息缓冲区的结构，并设定一个消息线程专门用于接收其他站点的消息和发送消息给其他站点以同步各个缓冲区之间的内容。在后面的介绍中我们将看到，各个站点缓冲区之间同步的内容包括请求的状态、请求的副作用、同步集合和依赖集合等等。

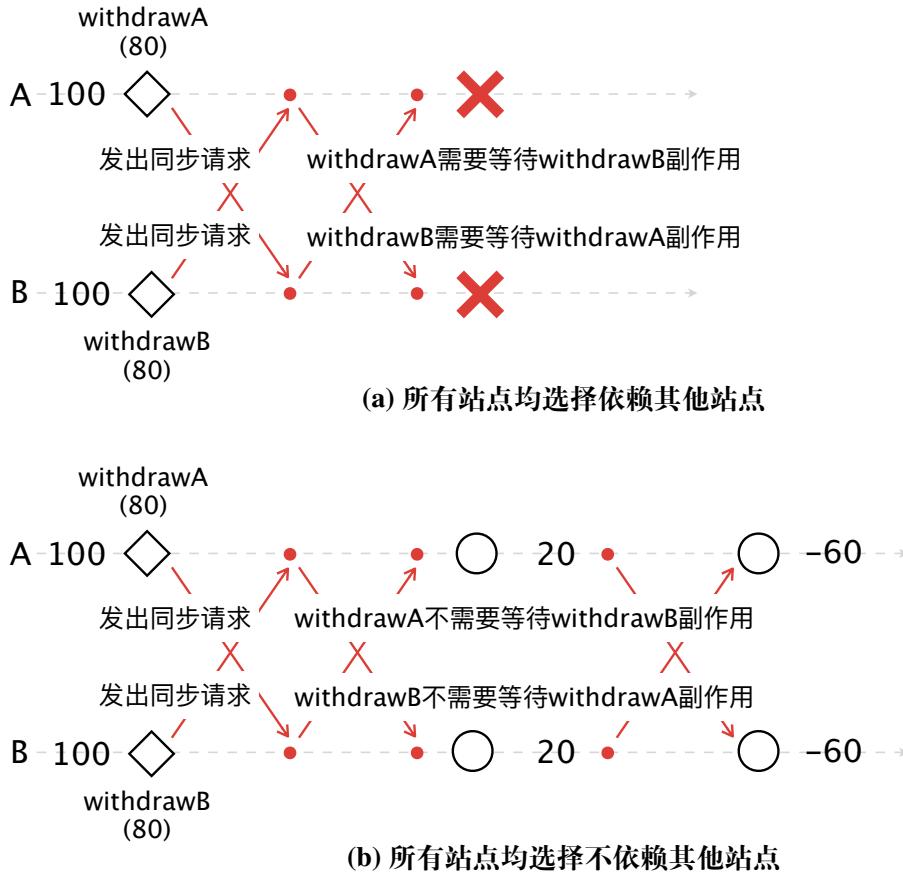


图 5.1 对称性事件对仲裁顺序的影响

### 5.1.2 如何决定全局历史记录和仲裁顺序？

在服务器端状态转换规则中，机器模型中存在一个抽象的全局仲裁顺序，这个顺序是通过每个请求从客户端到达服务器端时，放入消息缓冲区中的带有当前时刻历史记录的消息来实现的（如规则 INVOKE 所示）。因此，此仲裁顺序就是全局视角下所有请求到达客户端的次序。在真实分布式系统中，我们不存在这样的一个全局顺序关系。因此，我们通过每个站点之间的消息通信来使得他们对所有需要做全局仲裁顺序的请求之间产生顺序的共识即可。

如果通过每个站点之间的消息通信的方式来裁决请求之间的顺序，不必要的顺序裁决必然会影响请求的处理时长。因此，我们只做必要的顺序裁决，具体体现为：在某个站点上，如果一个请求对应的操作是一个写操作，并且 sync  $O_s$  原语中的  $O_s$  集合不为空，那么此站点需要与所有站点进行通信，来确定哪些写了  $O_s$  集合的请求发生在此操作之前（即确定 ACCEPT 等规则中的  $h_t \cap \text{wr}(O_s)_M$ ）。

这里还有一个值得注意的问题，就是需要打破上述确定仲裁顺序方法的对称性。否则，在银行系统同时取钱的例子中，如果 withdraw 函数需要同步 @acct 变量，并且两个 withdraw 请求同时在 A、B 两个站点上发生，那么

在接下来的确定仲裁顺序的过程中，如果二者的消息传递时间间隔恰好相同（即历史轨迹对称）那么由于对称性，只可能发生这两个请求同时认为自己发生在对方之后或者这两个请求同时认为自己发生在对方之前这两种情况，如图 5.2 所示。第一种情况会造成系统的死锁，第二种情况会使得 **sync** 原语失效，二者同时取钱成功导致钱数变为负数，都不是我们希望看到的。我们给出了一个非常简单的解决方案，即上述情况发生时，我们认为从更小标识符站点发生的请求永远在更大标识符站点发生的请求之前，即这些请求的仲裁顺序与站点的标号顺序一致。在此我们暂不考虑公平性（fairness）的问题。

### 5.1.3 如何进行垃圾回收？

在服务器端状态转换规则以及上两小节的介绍中，我们发现每个站点的历史记录都是只增的。由于我们需要不断计算并确定请求之间的仲裁顺序，在系统运行一段时间后，过大的历史记录会使得这个计算过程非常缓慢。因此，我们需要一个垃圾回收机制，将每个站点上所有处理过的节点（即图 2.1b 所示的标号为 2, 4, 5 的请求状态）进行回收。其中，由于读请求只在原始站点上被处理，不涉及到站点之间的通信，故我们可以将被处理完的读（图 2.1b 中的 ②）请求安全回收。但对于写请求，即处理完成的被接受（图 2.1b 中的 ⑤）的和被拒绝（图 2.1b 中的 ④）的请求我们不能轻易的直接回收它们。

对于被拒绝的请求，我们考虑一个最简单的情况。如图 5.2a 所示的银行系统同时取钱的例子中，**A** 站点和 **B** 站点产生了两个 `withdraw` 请求（记为 `withdrawA` 和 `withdrawB`），**B** 站点在执行 `withdrawB` 请求时，会因为其同步集合不为空，而向 **A** 站点发起同步请求。按照第 5.1.2 小节的介绍，我们认为站点 **A** 和站点 **B** 上两个 `withdraw` 请求的顺序与站点的标号顺序一致，故 `withdrawB` 请求同步到了 `withdrawA` 请求，它需要等 `withdrawA` 请求的副作用在 **B** 站点上执行过后，才可以被处理。而此时，如果 `withdrawA` 请求 **guard** 断言失败（即请求被拒绝），如果直接将此请求回收，则会使 **B** 站点发生死锁。

一个直接的解决方案是，**B** 站点向 **A** 站点发起同步请求后，会等待 `withdrawA` 请求 **guard** 断言的结果，如果断言成功则需要同步 `withdrawA` 请求，反之则不需要同步。但这种解决方案会带来效率上的问题。如果 `withdrawA` 请求迟迟不能被处理，那么 **B** 站点将会被阻塞。

为此，我们给出的解决方案如下：我们首先假设 `withdrawA` 请求的 **guard** 断言永远会成功，即 `withdrawB` 请求会等待 `withdrawA` 请求的副作用。在此之后，如果 `withdrawA` 请求 **guard** 断言失败，则其在回收这个请求前，会向所有的站点发送消息，通知其取消同步此请求，**B** 站点在收到此请求后，则会

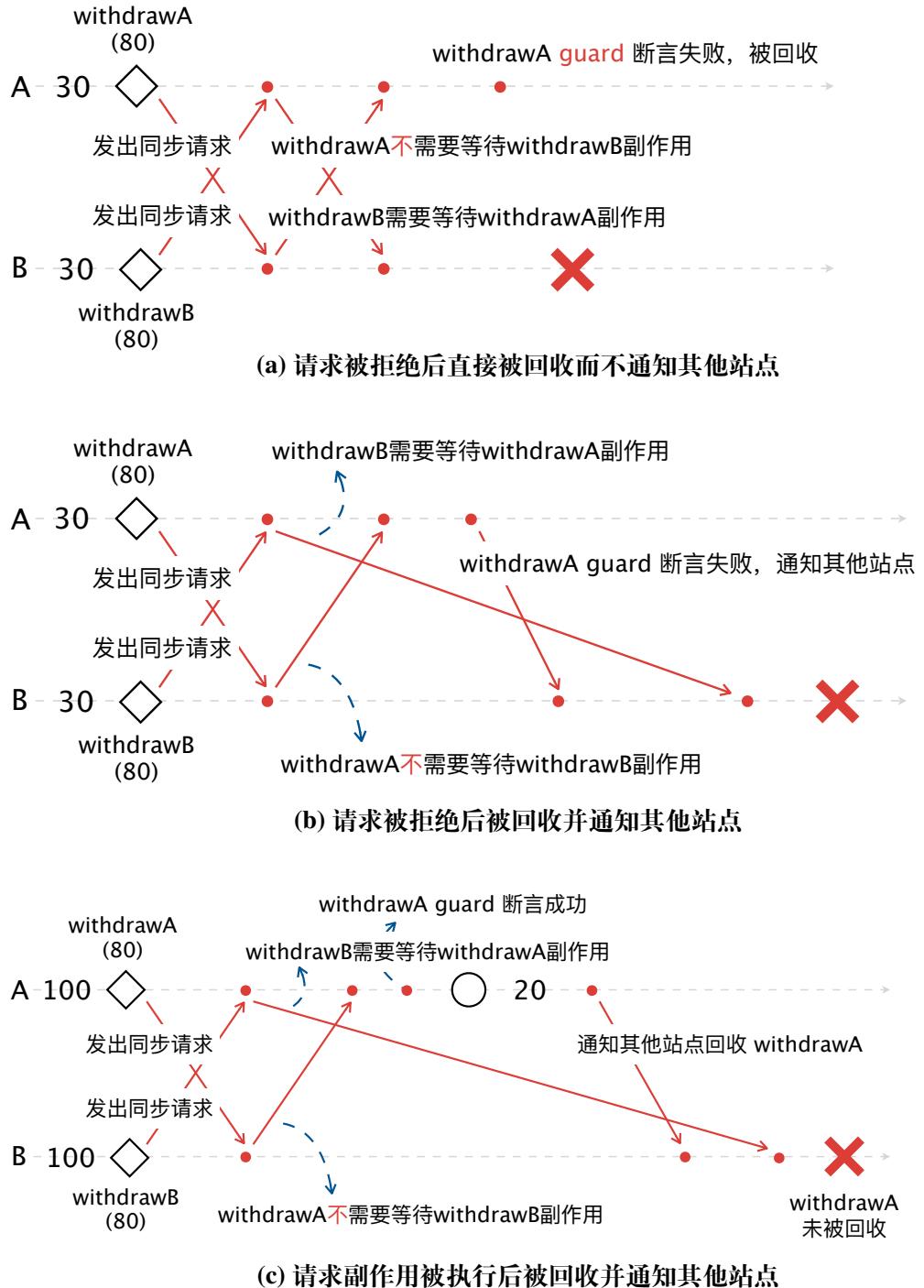


图 5.2 不恰当的请求回收对系统产生的影响

将其从 withdrawB 所等待请求的集合中删除。

虽然上述方案会解决 B 站点被阻塞的问题，但其仍会发生死锁。由于我们并没有要求各个站点之间消息的传递满足先进先出的原则，故站点之间消息可能发生乱序现象。如图 5.2b 所示，在 B 站点向 A 站点发出请求同步后，由于我们假设 withdrawA 请求的 **guard** 断言会成功，故 A 站点会将其 withdrawA

请求的标识符回复给  $B$  站点，要求其 withdrawB 请求等待 withdrawA 请求的副作用。在此之后，withdrawA 请求的 **guard** 断言失败， $A$  站点再向  $B$  站点发出删除请求的消息。如果这两个从  $A$  站点到  $B$  站点的消息乱序，那么 withdrawA 将不会从 withdrawB 的同步集合中删除，从而在  $B$  站点上发生死锁。

对于被接受的请求的回收处理上，消息传递的乱序也会产生影响。如图 5.2c 所示，如果 withdrawA 请求的 **guard** 断言成功，它将会把此请求的副作用广播到  $B$  站点，而后  $A$  站点会执行完此请求的副作用并将其回收，而后通知  $B$  站点也回收掉此请求。如果这两个从  $A$  站点到  $B$  站点的消息乱序，那么  $B$  站点上 withdrawA 请求将不会被回收，从而导致漏回收请求的情况发生。

为了在保证系统运行效率的同时解决上述问题，我们首先令每一个站点上的请求标识符由站点标号和本地的计数器标号组成。如  $A$  站点的请求标识符为  $A_1, A_2, \dots$ 。而后，我们在每个站点上额外维护一个向量时钟 (Vector Clock)<sup>[57]</sup>，即一个从站点标识符到整数的映射。里面记录了所有站点在此站点上请求的回收情况，如  $A$  站点回收到了标号为 3 的请求， $B$  站点回收到了标号为 4 的请求等等。在回收请求时，如果向量时钟中  $A$  站点的值为  $n$ ，那么标识符为  $A_{n+1}$  的请求可被回收，此时向量时钟中  $A$  站点的值会被更新为  $n+1$ 。此后，在计算同步集合和依赖集合时，所有  $A$  站点的标号值小于等于  $n+1$  的请求将会被忽略。

从上述讨论中我们发现，服务器端系统设计较为复杂，非常容易发生死锁情况，故我们在系统设计完毕后，会对整个系统用 TLA+ 工具进行验证。服务器端系统运行的整体流程将在下一小节中介绍，TLA+ 验证将会在第 5.3 节中给出。

## 5.2 服务器端系统运行流程

服务器端的每一个服务器均由三个线程组成。这三个线程共享一个 请求池。请求池是一个请求标识符到请求对象的映射。请求对象包含当前请求的状态、请求所调用的操作的名称、请求的参数等等。第一个线程（前端线程）用于处理客户端发来的请求。在接收到用户请求后，它将初始化此请求的标识符以及请求对象，并将它们放入请求池中；在用户请求处理完毕后，会将结果返回给用户。第二个线程（请求线程）用于处理请求池中的请求。它轮流查询请求池中的每个请求，并进行状态转换。状态转换主要分为两种情况，即无条件状态转换和有条件状态转换。无条件状态转换会根据请求目前的状态进行不同的操作（如给其他站点发送消息、执行副作用等等），而后进行状态转换。有条件状态转换则根据请求目前的状态判断是否触发状态转换（如是否收到了所有站点的请求、此站点是否达到执行某副作用的条件等等）。如果判定成立，则进行状态转换。第三个线

表 5.1 服务器间通信的消息类型和各类型消息的作用

消息类型	通信方式	作用
ASK_SYNC	广播	向其他站点查询需要同步的请求标识符
RTN_SYNC	单播	向指定站点回复需要同步的请求标识符
BCT_DO	广播	向其他站点广播需要执行的副作用
BCT_NOT_DO	广播	向其他站点发送取消执行某副作用
DO_RTN	单播	向某个请求的原始站点回复已经执行完该请求的副作用
DEL_OP	广播	通知其他站点删除某个请求



图 5.3 服务器端系统运行中每个请求的生命周期

程（消息线程）用于处理其他服务器传入此服务器的消息。它将根据消息类型，以及消息对应的请求标识符，在请求池对相应请求的某些变量进行修改。

每一个请求的状态包括 8 种，请求之间的状态转换图如图 5.3 所示。每种状态代表的含义以及状态之间的转换条件将在下文中进行讲述。消息的类型包括 6 种，消息通信的方式包括两种，分别为广播和单播，其中广播会发送给除自身站点外的所有站点，单播则会发送给带有指定标识符的站点。每个消息的消息传递类型和作用如表 5.1 所示。

接下来，我们将以图 5.4 为例，具体介绍服务器端是怎样处理客户端发来的请求的。图中共有  $A, B, C$  三个站点，在  $B$  站点上新产生了客户端请求（由前端线程放入请求池中）。我们把每一个请求分为两个阶段，第一阶段为处理和其他服务器的同步请求，第二阶段为执行、广播、回收等等。每个阶段根据通信消息的数量，又可分为慢速路径和快速路径。因此，对阶段和路径进行组合，一个请求可能有六种不同的处理方式。

当一个请求被前端线程放入请求池中时，其初始状态标记为  $B\_SYNC$ 。此后，它将检查当前请求是否存在需要同步的对象（即此请求调用的操作中的原语  $sync O_s$  中的  $O_s$  是否为空）。如果需要同步，如图 5.4(1) 所示，其向所有其他所有站点发送类型为  $ASK\_SYNC$  的消息，消息中带有同步对象集合  $O_s$ ，发送完成后，此请求的状态被更新为  $L\_SYNC$ 。此状态用于表示此请求在等待其他站点的回复。如果不需要同步，如图 5.4(2) 所示，则直接把状态更新为  $B\_PROC$ 。此状态用于表示同步已经完成，等待进一步处理。

站点  $A$  和站点  $C$  在收到  $ASK\_SYNC$  类型的消息后，将会根据消息中的同步对象集合  $O_s$  以及此消息发送方站点的标识符，计算出需要同步的请求标识符的集合（计算方式已在第 5.1.2 小节中介绍），通过  $RTN\_STNC$  类型的消息返回

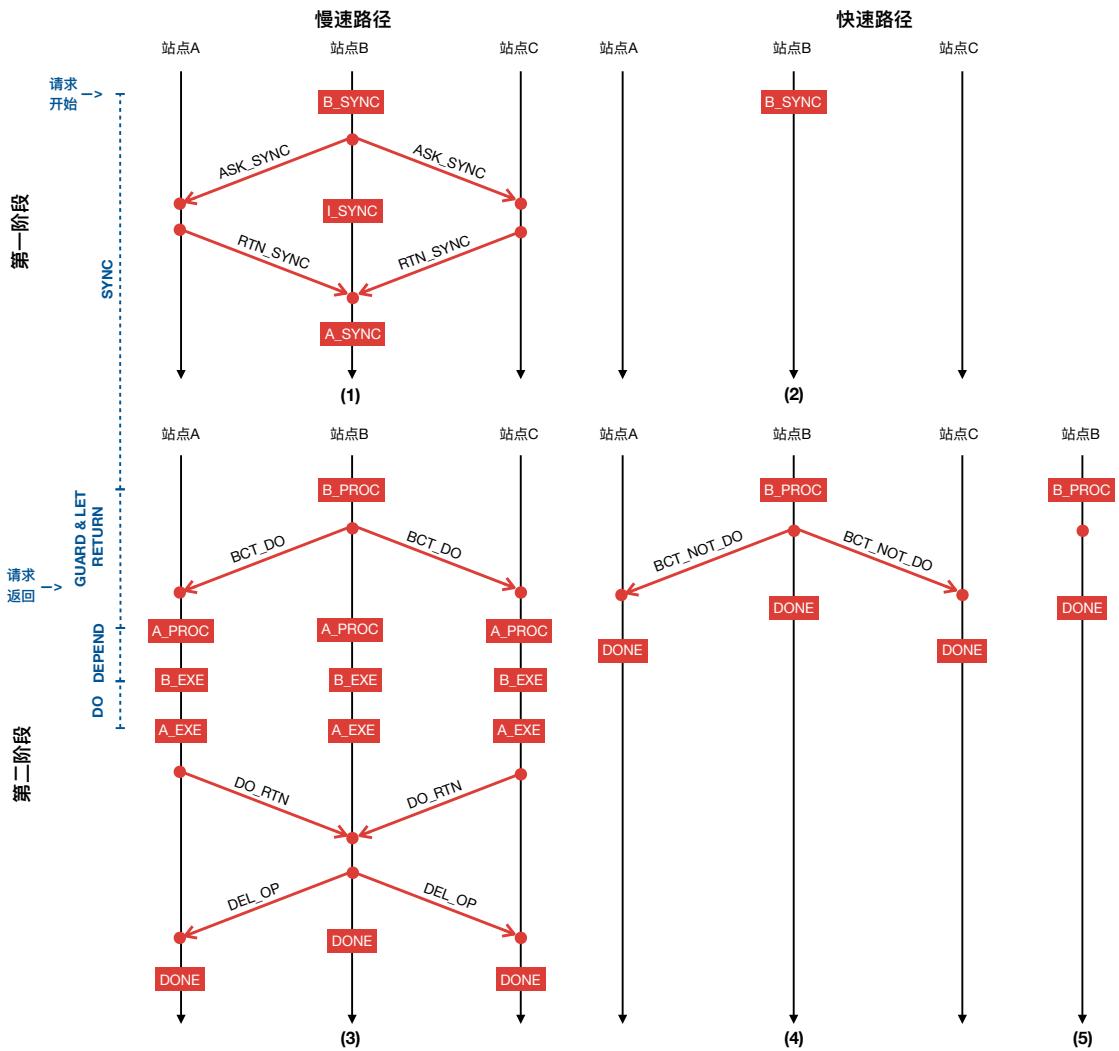


图 5.4 服务器端系统运行中的状态转换和消息传递

给站点 **B**。在站点 **B** 收到所有其他节点所回复的消息后，此请求的状态更新为 **A\_SYNC**。此状态表示同步请求全部返回，将等待同步过来的所有请求的副作用在此节点上执行完毕。在这些副作用执行完毕后，请求的状态更新为 **B\_PROC**。此时，第一阶段执行完毕。

第二阶段中，请求的初始状态为 **B\_PROC**。如果此请求是一个读请求，如图 5.4(5) 所示，它将计算需要返回的值，放入相应位置，前端线程会将此结果返回给客户端。之后，此请求的状态会被标记为 **DONE**。此状态用于表示这个请求已经处理完毕，等待回收。读请求处理的第二阶段不需要原始站点和其他站点进行通信。

如果此请求是一个写请求，那么将会首先判断 **guard** 原语中的断言是否成立。如果断言不成立，如图 5.4(4) 所示，则首先将写请求的结果放入相应位置

(由前端线程会将此结果返回给客户端), 再通过 BCT\_NOT\_DO 类型的消息通知所有节点此操作已经被撤销 (具体原因已经在第 5.1.2 小节中介绍), 最后将此请求的状态会标记为 DONE。

如果断言成立, 如图 5.4(3) 所示, 则首先将写请求的结果放入相应位置 (由前端线程会将此结果返回给客户端), 再执行 let 原语计算出此请求的副作用, 并通过此请求调用的操作中的原语 **depend**  $O_d$  中的对象集合  $O_d$  来计算需要依赖的请求标识符的集合, 而后通过 BCT\_DO 类型的消息将副作用和依赖集合广播到其他所有站点, 最后将此请求的状态会标记为 A\_PROC。此状态用于表示需要等待依赖集合中的所有请求的副作用在此节点上执行完毕。

我们先介绍此请求在原始节点上的处理过程, 再介绍在其他站点的处理过程。在站点  $B$  上, 如果依赖集合中的所有请求的副作用执行完毕, 此请求的状态将更新为 B\_EXE。而后, 副作用会被执行, 请求的状态将更新为 A\_EXE。接下来, 当原始节点接收到所有其他站点传来的用于表示执行副作用完成的消息 (DO\_RTN 类型的消息) 后, 再向所有节点广播删除此请求的消息 (原因已在第 5.1.2 小节中解释), 最后将此请求的状态会标记为 DONE, 表示其已经可以被回收。

在站点  $A$  和站点  $C$  上, 当收到 BCT\_TO 类型的消息后, 消息线程会将请求的标识符以及请求对象放入请求池中, 并将消息的类型初始化为 A\_PROC。同样的, 在站点  $A$  和站点  $C$  上同样需要等待依赖集合中的所有请求的副作用执行完毕后, 才能执行此请求的副作用。副作用执行完毕后, 请求的状态将更新为 A\_EXE。此后, 在收到请求的原始站点传来的 DEL\_OP 类型的消息后, 此请求的状态最终被标记为 DONE。

在原始站点和其他站点上, 被标记为 DONE 的请求最后会按照第 5.1.3 小节中介绍的方式被回收。在所有节点上此请求被回收后, 该请求的生命周期结束。更详细的流程介绍请参考附录 E 中使用 PlusCal 算法语言书写的伪代码。

## 5.3 使用 TLA+ 验证服务器端实现的正确性

### 5.3.1 TLA+ 简介

TLA+ (TLA 是“行为的时间逻辑”的首字母缩写) 是 Leslie Lamport 开发的一种正式规范语言用于对代码级的软件进行建模的语言。它用于设计, 建模, 记录和验证程序, 尤其是并发系统和分布式系统。它有一个 IDE (集成开发环境), 用于编写模型并运行工具来检查它们。

在编写 TLA+ 程序时, 通常使用 PlusCal 语言进行编写。PlusCal 是一种用于编写算法的语言, 可用于编写顺序算法和并发算法, 具有描述并发性和不确定性

---

```

1 ----- MODULE transfer -----
2 EXTENDS Naturals, TLC
3
4 (* --algorithm transfer
5 variables alice_account = 10, bob_account = 10, money = 5;
6
7 begin
8 A: alice_account := alice_account - money;
9 B: bob_account := bob_account + money;
10 C: assert alice_account >= 0;
11
12 end algorithm *)
13 ====

```

---

图 5.5 PlusCal 语言代码实例

的能力。它基于 TLA+ 规范语言，并且 PlusCal 算法语言可以自动转换为 TLA+ 规范，可以使用 TLC 模型检查器进行检查并进行形式化推理。图 5.5 给出了一个用 PlusCal 语言编写的转账模型（取自<sup>[68]</sup>），即 Alice 向 Bob 发起了一个金额为 5 的转账。其中，A、B 和 C 为标签（label），每两个标签中的代码的执行具有原子性，我们可以认为在模型检查过程中，每一个标签代码状态会执行一步（在 PlusCal 中被称为行为）。命令 `assert` 用于判断其后的断言是否被满足。如果在模型检查过程中断言不成立，则模型检查失败。

除了 `assert` 外，PlusCal 还支持其他与多线程和分布式系统相关的断言用于检测系统死锁（deadlock）和活性（liveness）。例如，我们使用  $\Box F$  来表示断言  $F$  是始终成立的。即  $\Box F$  在行为  $\langle s_1, s_2, \dots \rangle$  下为真，当且仅当对所有满足  $i > 0$  的  $i$ ,  $F$  在行为  $\langle s_i, s_{i+1}, \dots \rangle$  下为真。我们还使用  $\Diamond F$  来表示断言  $F$  最终会成立。即  $\Diamond F$  在行为  $\langle s_1, s_2, \dots \rangle$  下为真，当且仅当存在所有满足  $i > 0$  的  $i$ ,  $F$  在行为  $\langle s_i, s_{i+1}, \dots \rangle$  下为真。

在使用 PlusCal 进行建模和性质描述后，我们将其放入 TLC 中进行检查。TLC 会根据其模型检测算法，检查我们的模型对于特定的参数在运行过程中是否会违背这些性质。除了布尔值的检测结果，TLC 还返回一个称为碰撞概率的数值。由于在模型检测过程中，TLC 使用哈希函数将每个检测的状态映射到固定值  $h$ （用 64 位整数表示）。如果不同的状态映射到了同一个  $h$ ，那么将发生哈希碰撞，使得状态被覆盖，检测发生遗漏。上述碰撞概率即为在检测过程中发生哈希碰撞的几率，一般用于表示检测结果的可信度。

### 5.3.2 使用 PlusCal 算法语言对服务器端建模

本小节主要介绍如何使用 PlusCal 算法语言对服务器进行建模。为了模拟服务器端和客户端通信、服务器之间通信、以及单个服务器的不同线程之间执行的不确定性，我们使用 PlusCal 中的 **either ... or** 命令来进行模拟（如附录 E 中第 9-14 行和第 27-32 行所示）。除此之外，我们使用 **choose** 语句来从消息缓冲区和请求池中随机选择消息和请求（如附录 E 中第 40 行和第 63 行所示）。为了在保证模型检查可能性的情况下尽可能的接近服务器真实的运行场景，我们还做了以下几点改变：

- 定义了服务器端的最大服务器数量 MAX\_SERVER 以及系统运行时的最大请求数量 MAX\_OP\_INS，来保证模型检查的可能性；
- 在模型中，库是随机生成的，库中操作的数量为 MAX\_OP，每个操作的同步集合、依赖集合和写集合为集合  $\{1, \dots, \text{MAX\_OBJECT}\}$  子集中的其中一个，其中写集合不为空集。
- 每个请求在判断断言是否成立时，进行随机选择。即有 50% 的概率成功，50% 的概率失败。
- 我们假设库中的每个请求的副作用一定会终止。

### 5.3.3 使用 TLC 工具检查服务器的死锁和活性

本小节将介绍如何使用 TLC 工具检查服务器的死锁和活性。一个算法发生死锁是指算法没有终止，但没有办法进行下一步。多进程算法发生死锁是指没有进程可以进行下一步，但是某些进程尚未终止。由于 TLC 工具自动支持对模型死锁的检查，我们只需要给出活性的断言即可。活性断言是对死锁检查的进一步加强，它不仅要求算法能够在不终止的情况下能够运行下一步，而且要求算法能够在有限步骤内终止。例如 `while (n--) {}` 程序能够通过死锁检测，但由于变量 `n` 小于 0 时不终止而无法通过活性检测。如附录 E 中第 324 - 325 行所示。活性断言要求系统最终满足 IDLE 的状态。其中 IDLE 的定义与上文中一致。

我们在单个主机上进行模型检查实验，机器包含 40 个型号为 Intel Xeon E5-2630 的处理器和 264G 内存。机器运行的 Linux 内核版本为 4.15.0, PlusCal 翻译软件的版本为 1.9, TLC 模型检查器的版本为 2.14。在运行检查时，TLC 模型检查器的线程数为机器处理器的个数。

表 5.2 给出了 TLC 模型检查器在不同参数下的检测结果和检测时长。由于网络中的消息传递顺序、每个站点上消息的处理顺序和处理结果的不确定性。在 3 个站点上运行 3 个请求就可能出现 40 多亿种不同的行为。这体现了系统中请求处理和通信算法的复杂性。因此，对服务器进行建模和模型检测来避免非预期的行为是十分必要的。

表 5.2 TLC 模型检查器在不同参数下的检测结果和运行时长

参数				状态数量 (去重复后)	运行时长 (秒)	碰撞概率
MAX_SERVER	MAX_OP	MAX_OBJECT	MAX_OP_INS			
1	1	1	1	74	4	1.3E-16
1	10	10	10	5,694	4	2.6E-13
1	10	10	100	555,213	17	1.7E-7
2	10	10	1	3824	4	1.1E-12
2	10	10	3	565,600	21	3.6E-8
2	10	10	10	218,606,437	7,201	0.0034
3	10	10	1	367,811	25	1.8E-8
3	10	10	3	4,160,022,338	132,613	0.025

## 5.4 其他实现细节

我们使用 Java 语言对系统进行实现，总计代码行数约 5300 行。其中数据库的实现为支持键值存储的内存数据库，键的类型为字符串，值的类型为 Java 对象（Object）。请求池、向量时钟均使用 `java.util.concurrent` 包中的 `ConcurrentHashMap` 实现，以便多个线程同时进行读写操作。

### (1) 通信的实现

系统中客户端和服务器、服务器之间使用开源的基于 JMS (Java Message Servie) 规范的消息中间件 ActiveMQ 进行通信<sup>[69]</sup>。为了提升通信性能，我们使用了以下几个配置选项：

- 关闭了 Broker 组件的持久化存储 (`setPersistent(false)`)；
- MessageProducer 组件使用异步方式进行消息发送 (`setUseAsyncSend(true)`)；
- 关闭 MessageProducer 组件发送消息时的拷贝 (`setCopyMessageOnSend(false)`)；
- MessageListener 组件使用异步方式接受消息 (`setMessageListener(...)`)。

### (2) 函数库的接口

库编写者通过配置文件指定库的位置，库的类型为 `Map <String, Operation>`，其中 `Operation` 为 Java 中的接口（Interface）。库编写者在编写库中的每个操作时，需要实现此接口。`Operation` 接口包含以下几个函数：

- 获取操作名称的函数 `getOpName`；
- 获取操作同步结合、依赖集合和写集合的函数 `getSyncSet`、`getDepSet` 和 `getWriteSet`；
- 原语 `guard`、`let`、`do` 或 `return`。

除此之外，库编写者还需给出初始化数据库的方法。

## 5.5 本章小节

本章给出了细粒度一致性模型分布式系统的设计和实现。首先，我们在第 3.3 节的操作语义的基础上做了进一步细化，给出了在分布式系统环境下的消息通信、顺序仲裁和垃圾回收的方法。其次，我们给出了服务器端系统的运行流程，具体介绍了服务器中请求的状态转换以及消息传递的方式。再次，我们使用 PlusCal 算法语言对服务器端建模，并是用 TLC 工具对服务器进行死锁和活性检测。最后，我们给出了实现层面的一些细节。

## 第 6 章 实验与分析

在本章中，我们将使用第 3.2 节中所给出的三个语言的应用实例来评估系统的性能，这些实例用第 5.4 节中给出的库接口进行实现，并运行在第 5 章介绍的系统上。除此之外，我们还在保证程序语义的前提下依据第 3.5 小节介绍的不同一致性保证的命题修改同步集合和依赖集合，来使得程序在不同的一致性模型下运行，并与我们的混合一致性模型进行性能对比。最后，我们还将探究系统的公平性对每个站点延迟的影响。

### 6.1 实验环境

#### 6.1.1 站点配置

我们在亚马逊云服务平台<sup>[70]</sup>美国西部（US West-Oregon）和欧洲西部（EU West-London）两个数据中心的 EC2 m5.xlarge 虚拟机（VM）上进行实验。两个站点之间的平均往返延迟为 144 ms。每个虚拟机有 4 个虚拟处理器（vCPU），16GB 运行内存，8GB 的磁盘空间以及最高 10 千兆位的网络带宽。虚拟机运行 Ubuntu Server 16.04 LTS 操作系统和 Java 8 软件。

#### 6.1.2 工作负载

所有的应用实例在运行时均采用混合工作负载，包括 85% 的读请求和 15% 的写请求。应用实例每个操作所占的具体比例如表 6.1 所示。在运行时，用户的请求均匀的分布在每个数据中心，并向最近的站点发出请求。每个实验的测量时间为 70 秒，其中前 10 秒的数据由于系统刚刚启动，数据不具有代表性而被舍弃。每个实验重复 3 次并使用其平均值作为最终结果。

#### 6.1.3 性能评估指标

性能评估指标包括吞吐量和延迟两种。其中，吞吐量是指所有服务器站点每秒能够处理的用户请求的总量。延迟是指用户可感知延迟，即用户在发起请求时的时间与用户收到请求结果的时间之差。注意，对于写请求，请求结果可以在 **guard** 原语断言的判断结束后立即返回（请求成功或失败），而不需要等待此请求的副作用被执行之后再返回。

表 6.1 应用实例运行时每个操作所占比例

实例名称	操作名称和所占比例 (%)				
银行系统	deposit	5	withdraw	5	cal_interest 5 read_balance 85
add-win 集合	add	10	remove	5	check_exist 85
拍卖系统	register_user	10	place_bid	5	query_bid 85 close_auction 0*

\* 由于执行 `close_auction` 请求后拍卖将结束，故我们只在最后一个请求中调用。

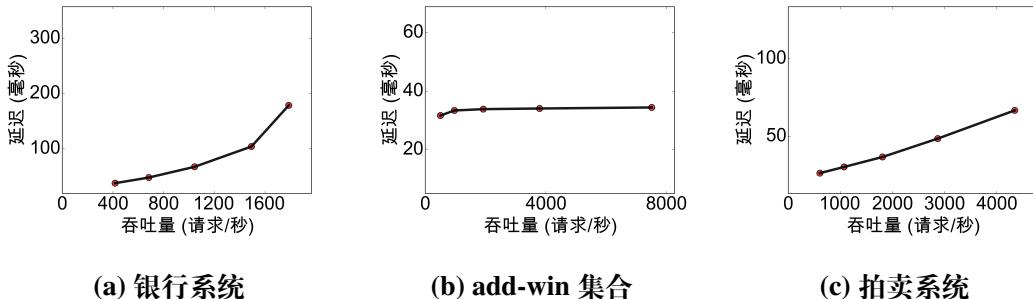


图 6.1 系统运行各个实例时的吞吐量和延迟

## 6.2 实验结果和结果分析

### 6.2.1 三个应用实例运行时的吞吐量和延迟

图 6.1 给出了三个工作负载的吞吐量-延迟图。每个图上的五个数据点分别代表每个数据中心有 1、2、4、8 和 16 台虚拟机时的吞吐量和延迟。从图中我们可以发现，对于银行系统工作负载，在系统有 32 个站点时系统过载，吞吐量相比 16 个站点增幅不大但延迟大幅度提高。对于 add-win 集合工作负载，则并没有因为系统规模的增加而对吞吐量或延迟造成较大影响。对于拍卖系统，随着系统规模的增加，对吞吐量影响较小，而对延迟影响较大。

上述实验结果体现了系统优秀的可扩展性。对于 add-win 集合应用实例，由于每个操作的同步集合均为空且只有 `remove` 操作的依赖集合不为空，吞吐量的增加对于延迟几乎没有影响。对于银行系统和拍卖系统应用实例，随着吞吐量的增大，每个请求依赖于其他请求数量会大大增加，从而造成了较高延迟。相比拍卖系统，银行系统在同等站点规模下吞吐量更低、延迟更高、不容易使得系统过载。这种现象的原因是在拍卖系统中，三个写请求会同步不同的共享对象，降低了每个请求依赖于其他请求的数量，使得每个请求等待的时间减少。

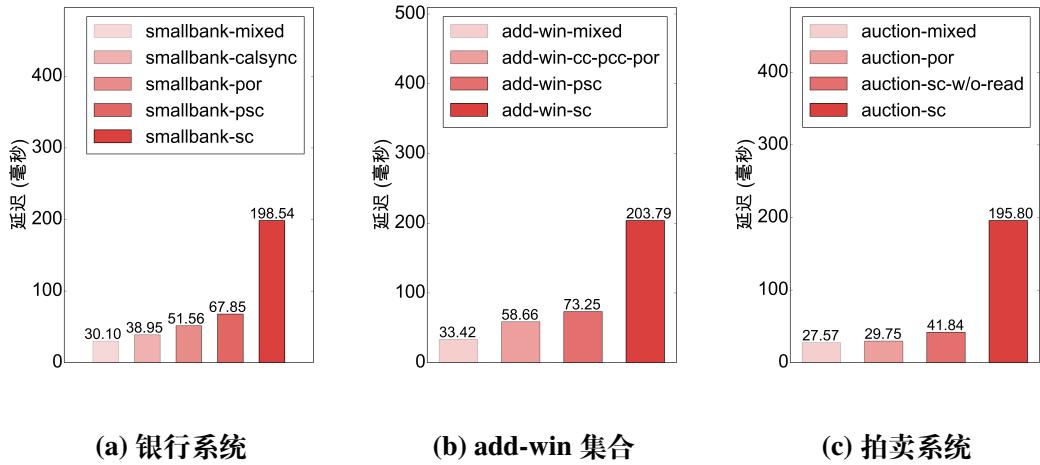


图 6.2 系统运行不同一致性保证下的各个实例时的延迟

### 6.2.2 不同的一致性保证对延迟的影响

图 6.2 给出了在系统共 4 个站点时运行不同一致性保证下的各个实例的延迟。在图 6.2a 银行系统的延迟图中, `smallbank-mixed`、`smallbank-calsync` 分别代表细粒度一致性模型。二者的区别在于使用 `calculate_interest` 操作计算利息时, 是否对账户余额进行同步。`smallbank-por`、`smallbank-psc`、`smallbank-sc` 分别代表 PoR 一致性和基于变量的顺序一致性和顺序一致性模型。PoR 一致性模型要求系统最弱的一致性保证为因果一致性, `smallbank-psc`、`smallbank-sc` 的区别在于 `read_balance` 操作中同步集合是否为空。在图 6.2b `add-win` 集合的延迟图中, `add-win-mixed` 代表细粒度一致性模型, `add-win-cc-pcc-por` 代表因果一致性、基于变量的因果一致性和 PoR 一致性模型, 其中后两种均等价于因果一致性。`add-win-psc`、`add-win-sc` 分别代表基于变量的顺序一致性和顺序一致性模型, 二者的区别在于查询操作中同步集合是否为空。在图 6.2c 拍卖系统的延迟图中, `auction-mixed` 和 `auction-por` 分别代表细粒度一致性模型和 PoR 一致性模型, `auction-sc-w/o-read` 和 `auction-sc` 代表顺序一致性模型, 二者的区别在于 `auction-sc-w/o-read` 模型中对竞拍结果查询操作的同步集合为空。

图 6.2 的实验结果表明了细粒度一致性模型对用户可感知延迟度带来了显著性的影响。与顺序一致性相比, 混合一致性在三个实例中将延迟降低为原来的 15% 左右。对于银行系统实例, 如图 6.2a 所示, 使用 `calculate_interest` 操作计算利息时首先对账户余额进行同步会有一定的延迟增大, 但却提供了一定的语义保证。库的编写者可以在二者之间做出抉择, 这体现了细粒度一致性模型的灵活性。对于 `add-win` 集合实例, 目前相关工作中的算法实现均要满足因果一致性的保证。我们将此保证进一步弱化, 并给出了弱化后的混合一致性模型和

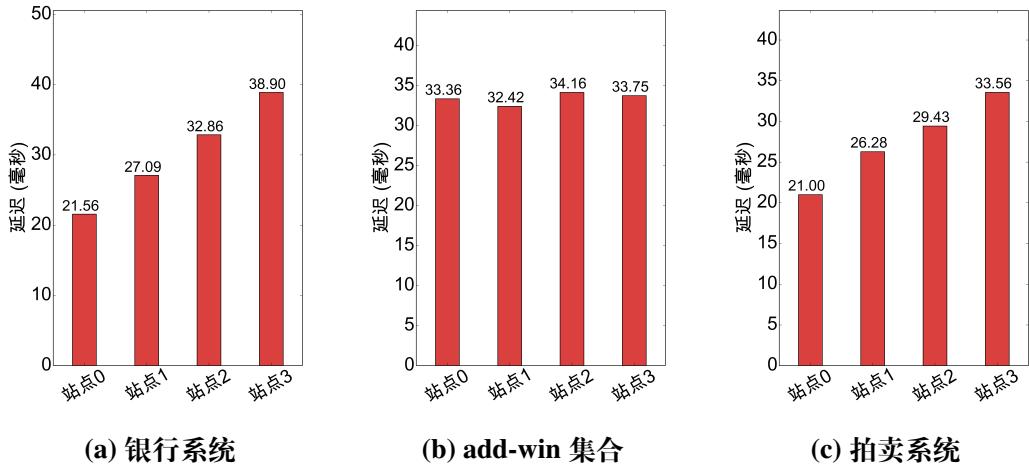


图 6.3 系统运行不同实例时各个站点的延迟

因果一致性保证下的延迟对比。图 6.2b 中的结果表明，细粒度一致性模型会将延迟降低为原来的 57%。对于拍卖系统实例，如图 6.2c 所示，相比于 PoR 一致性模型，混合一致性模型带来的性能提升并非显著。具体原因是因为拍卖系统的一致性保证要求较高，并且其可用粗粒度一致性模型进行完整语义表达。

除此之外，我们还观察到对于所有的应用实例的工作负载在顺序一致性的保证下的延迟均相同，这也符合我们的预期。在此种一致性保证下，延迟为图 5.4(1) 所示的第一阶段慢速路径时长，即为每个所有站点之间的一次往返时间（至少为 144 ms）加上每个站点对这些消息的平均处理时常。

### 6.2.3 公平性对延迟的影响

回顾在第 5.1.2 小节中，对于仲裁顺序我们在分析各种情况下可能出现的状况后，最终采用了以站点标号顺序作为请求之间的仲裁顺序的方案。这种方案没有考虑公平性问题，即标号越小的请求，其依赖于其他请求的数量会越小，从而能够更快的被处理完。本节中我们将通过实验数据分析系统公平性问题。在系统中有四个站点时，如图 6.3 所示，三个应用实例中银行系统和拍卖系统的延迟图表明其会受到系统公平性影响，而 add-win 集合则因为其所有操作的依赖集合均为空而不受影响。我们可以使用更复杂的协议来保证系统的公平性。例如，我们可使用动态优先级，每个站点经过一定时间后优先级将发生变化。进一步的，我们也使用 ZooKeeper<sup>[71]</sup> 等分布式应用程序协调服务，或使用 Paxos<sup>[72]</sup>、Raft<sup>[73]</sup> 等分布式共识算法的方式来保证系统的公平性。改进仲裁方法使得不同站点请求的公平性得到保证将作为我们的进一步工作。

### 6.3 本章小结

在本章中，我们通过第 3.2 节中所给出的银行系统、add-win 集合和拍卖系统三个实例来评估系统在不同站点数量和不同工作负载下的延迟和吞吐量。除此之外，我们还通过配置三个实例每个操作的同步集合和依赖集合来使得程序在因果一致性、顺序一致性、PoR 一致性等一致性保证下运行，并与混合一致性模型进行性能对比。最后，我们还探究了系统的公平性对每个站点延迟的影响并给出了可行的改进方法。

## 第 7 章 结 论

### 7.1 本文工作总结

本文在分布式一致性模型的理论和应用方面做了如下工作：

- 本文提出了一个细粒度一致性模型和相应的编程语言，并给出了相应的操作语义。新的语言兼顾了实用性和表达能力，可使编程人员表达出其想要的一致性语义，从而避免不必要的保证所产生的程序效率损失。
- 本文给出了细粒度一致性模型的相关性质，便于编程人员对其有更深刻的理解。除此之外，编程人员还可以通过配置语言中的同步集合和依赖集合来保证多种常见的一致性语义，这体现了模型的通用性。
- 本文给出了用于验证程序满足收敛性和特定不变量的程序逻辑，并给出了程序逻辑的可靠性证明。相关证明的关键引理已在定理证明工具 Coq 中实现，共计约 3100 行证明脚本。
- 本文给出了基于上述一致性模型和语言的系统设计和实现。系统使用 Java 语言实现，共计约 5300 行代码。除此之外，为保证系统正确性，我们使用了 TLA+ 模型检查工具对其进行死锁和活性检查，共计约 1500 行代码。在不同工作负载下的实验结果表明了系统的高可靠性和可扩展性。

### 7.2 进一步工作

进一步的工作内容可以总结为以下几点：

- 为新的语言设计一个解析器 在本文中，库的设计者需要实现 Operation 接口的相关函数来编写操作。在将来，我们考虑为语言设计一个更友好的编程接口，使得如图 3.2 所示的代码可以转换成抽象语法树并与先有的编程接口衔接。
- 完善程序逻辑相关证明 在本文中我们使用定理证明工具 Coq 对程序逻辑证明的关键引理进行了证明。在将来，我们将对这些证明进行进一步完善。
- 对系统进行进一步优化 在本文中我们给出了细粒度一致性分布式系统的实现。在将来，会对系统进行进一步优化。例如，给请求增加优先级、请求处理的方法由轮询改为通知、以及解决系统公平性问题等等。

## 参 考 文 献

- [1] CALDER B, WANG J, OGUS A, et al. Windows azure storage: a highly available cloud storage service with strong consistency[C]//Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP). ACM, 2011: 143-157.
- [2] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally-distributed database [C]//Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 2012: 261-264.
- [3] ABADI D. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story[C]//IEEE Computer. IEEE Computer Society Press, 2012: 54(2), 37-42.
- [4] BALAKRISHNAN M, MALKHI D, WOBBER T, et al. Tango: Distributed data structures over a shared log[C]//Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). ACM, 2013: 325-340.
- [5] DU J, IORGULESCU C, ROY A, et al. Gentlerain: Cheap and scalable causal consistency with physical clocks[C]//Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC). ACM, 2014: 4:1-4:13.
- [6] DECANDIA G, HASTORUN D, JAMPANI M, et al. Dynamo: Amazon's highly available key-value store[C]//Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP). 2007: 205-220.
- [7] SIVASUBRAMANIAN S. Amazon dynamodb: A seamlessly scalable non-relational database service[C]//Proceedings of the 2012 International Conference on Management of Data (SIGMOD). ACM, 2012: 729-730.
- [8] CROOKS N, PU Y, ESTRADA N, et al. Tardis: A branch-and-merge approach to weak consistency[C]//Proceedings of the 2016 International Conference on Management of Data (SIGMOD). ACM, 2016: 1615-1628.
- [9] BALEGAS V, DUARTE S, FERREIRA C, et al. Putting consistency back into eventual consistency[C]//Proceedings of the 10th European Conference on Computer Systems (EuroSys). ACM, 2015: 6:1-6:16.
- [10] BURCKHARDT S, LEIJEN D, FÄHNDRICH M, et al. Eventually consistent transactions [C]//Proceedings of the 21st European Symposium on Programming (ESOP). Springer, 2012: 67-86.
- [11] XIE C, SU C, KAPRITSOS M, et al. Salt: Combining ACID and BASE in a distributed database[C]//Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 2014: 495-509.

- [12] XIE C, SU C, LITTLEY C, et al. High-performance acid via modular concurrency control[C]// Proceedings of the 25th Symposium on Operating Systems Principles (SOSP). ACM, 2015: 279-294.
- [13] LAMPORT L. How to make a multiprocessor computer that correctly executes multiprocess programs[C]//Communications of the ACM. IEEE Computer Society, 1979: 28(9), 690-691.
- [14] BURCKHARDT S, GOTSMAN A, YANG H. Understanding eventual consistency[C]// Microsoft Research Technical Report (MSR-TR). Microsoft, 2013: 39.
- [15] BURCKHARDT S, et al. Principles of eventual consistency[C]//Foundations and Trends® in Programming Languages. Now Publishers Inc., 2014: 1-150.
- [16] BRUTSCHY L, DIMITROV D, MüLLER P, et al. Serializability for eventual consistency: Criterion, analysis, and applications[C]//Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). ACM, 2017: 458-472.
- [17] LLOYD W, FREEDMAN M J, KAMINSKY M, et al. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops[C]//Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP). ACM, 2011: 401-416.
- [18] LI C, PORTO D, CLEMENT A, et al. Making geo-replicated systems fast as possible, consistent when necessary[C]//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI). ACM, 2012: 265-278.
- [19] LI C, PREGUIÇA N, RODRIGUES R. Fine-grained consistency for geo-replicated systems [C]//2018 USENIX Annual Technical Conference (USENIX ATC). USENIX Association, 2018: 359-372.
- [20] MILANO M, MYERS A C. Mixt: A language for mixing consistency in geodistributed transactions[C]//Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, 2018: 226-241.
- [21] LI C, LEITÃO J, CLEMENT A, et al. Automating the choice of consistency levels in replicated systems[C]//2014 USENIX Annual Technical Conference (USENIX ATC). USENIX Association, 2014: 281-292.
- [22] TERRY D. Replicated data consistency explained through baseball[C]//Communications of the ACM. ACM, 2013: 56(12), 82-89.
- [23] CERONE A, BERNARDI G, GOTSMAN A. A framework for transactional consistency models with atomic visibility[C]//Proceedings of the 26th International Conference on Concurrency Theory (CONCUR). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015: 58-71.
- [24] BURCKHARDT S. Consistency in distributed systems[C]//Software Engineering: International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures. Springer, 2015: 84-120.

- [25] GOTSMAN A, YANG H, FERREIRA C, et al. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems[C]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2016: 371-384.
- [26] BAILIS P, FEKETE A, FRANKLIN M J, et al. Coordination avoidance in database systems [C]//Proceedings of the 40th International Conference on Very Large Data Bases (VLDB). VLDB Endowment, 2014: 185-196.
- [27] GUERRAOUI R, PAVLOVIC M, SEREDINSCHI D A. Incremental consistency guarantees for replicated objects[C]//Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 2016: 169-184.
- [28] KRASKA T, HENTSCHEL M, ALONSO G, et al. Consistency rationing in the cloud: Pay only when it matters[C]//Proceedings of the 35th International Conference on Very Large Data Bases (VLDB). VLDB Endowments, 2009: 253-264.
- [29] BAILIS P, FEKETE A, HELLERSTEIN J M, et al. Scalable atomic visibility with ramp transactions[C]//Proceedings of the 2014 International Conference on Management of Data (SIGMOD). ACM, 2014: 27-38.
- [30] LIU J, MAGRINO T, ARDEN O, et al. Warranties for faster strong consistency[C]// Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI). USENIX Association, 2014: 503-517.
- [31] LLOYD W, FREEDMAN M J, KAMINSKY M, et al. Stronger semantics for low-latency geo-replicated storage[C]//Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI). USENIX, 2013: 313-328.
- [32] PLUGGE E, HAWKINS T, MEMBREY P. The definitive guide to mongodb: The nosql database for cloud and desktop computing[M]. Apress, 2010.
- [33] SHARMA Y, AJOUX P, ANG P, et al. Wormhole: Reliable pub-sub to support geo-replicated internet services[C]//Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI). USENIX Association, 2015: 351-366.
- [34] ZHANG I, SHARMA N K, SZEKERES A, et al. Building consistent transactions with inconsistent replication[C]//Proceedings of the 25th Symposium on Operating Systems Principles (SOSP). ACM, 2015: 263-278.
- [35] SOVRAN Y, POWER R, AGUILERA M K, et al. Transactional storage for geo-replicated systems[C]//Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP). ACM, 2011: 385-400.
- [36] SHUDO K, YAGUCHI T. Causal consistency for distributed data stores and applications as they are[C]//IEEE 40th Annual Computer Software and Applications Conference (COMP-

- SAC). 2016: 602-607.
- [37] TERRY D B, PRABHAKARAN V, KOTLA R, et al. Consistency-based service level agreements for cloud storage[C]//Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). ACM, 2013: 309-324.
- [38] ABDULLA P A, ATIG M F, NGO T P. The best of both worlds: Trading efficiency and optimality in fence insertion for tso[C]//Proceedings of the 24th European Symposium on Programming (ESOP). Springer, 2015: 308-332.
- [39] KUNG H T, ROBINSON J T. On optimistic methods for concurrency control[C]//ACM Transactions on Database Systems (TODS). ACM, 1981: 6(2), 213-226.
- [40] LAHAV O, VAFFEADIS V, KANG J, et al. Repairing sequential consistency in c/c++11[C]// Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, 2017: 618-632.
- [41] ALGLAVE J, KROENING D, NIMAL V, et al. Software verification for weak memory via program transformation[C]//Proceedings of the 22rd European Symposium on Programming (ESOP). Springer, 2013: 512-532.
- [42] BATTY M, OWENS S, SARKAR S, et al. Mathematizing c++ concurrency[C]//Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2011: 55-66.
- [43] FU M, LI Y, FENG X, et al. Reasoning about optimistic concurrency using a program logic for history[C]//Proceedings of the 21st International Conference on Concurrency Theory (CONCUR). Springer, 2010: 388-402.
- [44] GOTSMAN A, RINETZKY N, YANG H. Verifying concurrent memory reclamation algorithms with grace[C]//Proceedings of the 22rd European Symposium on Programming (ESOP). Springer, 2013: 249-269.
- [45] KANG J, HUR C K, LAHAV O, et al. A promising semantics for relaxed-memory concurrency[C]//Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). ACM, 2017: 175-189.
- [46] RAAD A, LAHAV O, VAFFEADIS V. On parallel snapshot isolation and release/acquire consistency[C]//Proceedings of the 27th European Symposium on Programming (ESOP). Springer, 2018: 940-967.
- [47] LAHAV O, GIANNARAKIS N, VAFFEADIS V. Taming release-acquire consistency[C]// Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2016: 649-662.
- [48] LAHAV O, VAFFEADIS V. Owicky-gries reasoning for weak memory models[C]//Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP).

- Springer-Verlag, 2015: 311-323.
- [49] TURON A, VAFEIADIS V, DREYER D. Gps: Navigating weak memory with ghosts, protocols, and separation[C]//Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA). ACM, 2014: 691-707.
- [50] SHAPIRO M, PREGUIÇA N, BAQUERO C, et al. Conflict-free replicated data types[C]// Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). Springer, 2011: 386-400.
- [51] MEIKLEJOHN C, VAN ROY P. Lasp: a language for distributed, eventually consistent computations with crdts[C]//Proceedings of the 1st Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC). ACM, 2015: 7.
- [52] SIVARAMAKRISHNAN K, KAKI G, JAGANNATHAN S. Declarative programming over eventually consistent data stores[C]//Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, 2015: 413-424.
- [53] ZAZA N, NYSTROM N. Data-centric consistency policies: A programming model for distributed applications with tunable consistency[C]//Proceedings of the 1st Workshop on Programming Models and Languages for Distributed Computing (PMLDC). ACM, 2016: 3:1-3:4.
- [54] MARGARA A, SALVANESCHI G. Consistency types for safe and efficient distributed programming[C]//Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs (FTFJP). ACM, 2017: 1-2.
- [55] BURCKHARDT S, GOTSMAN A, YANG H, et al. Replicated data types: Specification, verification, optimality[C]//Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2014: 271-284.
- [56] DONGOL B, JAGADEESAN R, RIELY J. Transactions in relaxed memory architectures[C]// Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2017: 18:1-18:29.
- [57] LAMPORT L. Time, clocks, and the ordering of events in a distributed system[C]// Communications of the ACM. ACM, 1978: 21(7), 558-565.
- [58] ORACLE. Overview of Multitier Architecture[EB/OL]. 2019. [https://docs.oracle.com/cd/B28359\\_01/server.111/b28318/dist\\_pro.htm#CNCPT702](https://docs.oracle.com/cd/B28359_01/server.111/b28318/dist_pro.htm#CNCPT702).
- [59] STACKIFY. What is N-Tier Architecture? How It Works, Examples, Tutorials, and More [EB/OL]. 2017. <https://stackify.com/n-tier-architecture/>.
- [60] LI C, PORTO D, CLEMENT A, et al. Making geo-replicated systems fast as possible, consistent when necessary[C]//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI). ACM, 2012: 265-278.

- [61] LI C, PREGUIÇA N, RODRIGUES R. Fine-grained consistency for geo-replicated systems [C]//2018 USENIX Annual Technical Conference (USENIX ATC). USENIX Association, 2018: 359-372.
- [62] GOTSMAN A, YANG H, FERREIRA C, et al. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems[C]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2016: 371-384.
- [63] LESANI M, BELL C J, CHLIPALA A. Chapar: certified causally consistent distributed key-value stores[C]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, 2016: 357-370.
- [64] SHAPIRO M, PREGUIÇA N, BAQUERO C, et al. A comprehensive study of convergent and commutative replicated data types[J]. 2011.
- [65] LU H, VEERARAGHAVAN K, AJOUX P, et al. Existential consistency: measuring and understanding consistency at facebook[C]//Proceedings of the 25th Symposium on Operating Systems Principles (SOSP). ACM, 2015: 295-310.
- [66] Count Lines of Code[EB/OL]. <https://github.com/AlDanial/cloc>.
- [67] LAMPORT L. Specifying systems: the tla+ language and tools for hardware and software engineers[M]. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [68] Learn TLA+[EB/OL]. <https://www.learntla.com/introduction/example/>.
- [69] ActiveMQ[EB/OL]. <https://activemq.apache.org>.
- [70] Amazon Web Service[EB/OL]. <https://aws.amazon.com/>.
- [71] HUNT P, KONAR M, JUNQUEIRA F P, et al. Zookeeper: Wait-free coordination for internet-scale systems.[C]//USENIX Annual Technical Conference (USENIX ATC): volume 8. 2010.
- [72] LAMPORT L, et al. Paxos made simple[J]. ACM Sigact News, 2001, 32(4):18-25.
- [73] ONGARO D, OUSTERHOUT J. In search of an understandable consensus algorithm[C]// USENIX Annual Technical Conference (USENIX ATC). 2014: 305-319.

## 附录 A 定理 3.1 的证明

为了证明定理 3.1，我们首先给出定义 A.1-A.3 和引理 A.1。

**定义 A.1 (广播顺序)** 两个事件  $e_A$  和  $e_B$  在轨迹  $\Delta$  下有广播顺序 (用  $e_A \xrightarrow{\text{bo}}_{\Delta} e_B$  表示) 当且仅当存在  $\text{id}$  使得  $e_A \in \text{events}(\text{id}, \text{acpt})_{\Delta}$  并且  $e_B \in \text{events}(\text{id}, \text{exe})_{\Delta}$ 。

**定义 A.2 (发生顺序)** 一个事件  $e_A$  在轨迹  $\Delta$  下发生在另一个事件  $e_B$  之前 (用  $e_A \xrightarrow{\text{hb}}_{\Delta} e_B$  表示) 当且仅当以下条件其中之一成立:

1.  $e_A \xrightarrow{\text{so}}_{\Delta} e_B$ 。
2.  $e_A \xrightarrow{\text{bo}}_{\Delta} e_B$ 。
3. 存在  $e_C$  使得  $e_A \xrightarrow{\text{hb}}_{\Delta} e_C$  和  $e_C \xrightarrow{\text{hb}}_{\Delta} e_B$ 。

**定义 A.3 (结构良好的轨迹)** 一个轨迹  $\Delta$  是结构良好的，当且仅当以下条件全部成立:

1. 每一个  $\Delta$  中的元素是唯一的。
2. 对于所有的  $\text{id} \in \Delta.ids$ , 存在唯一的  $e$  使得  $e \in \text{events}(\text{id}, \text{rej})_{\Delta} \cup \text{events}(\text{id}, \text{rtn})_{\Delta} \cup \text{events}(\text{id}, \text{acpt})_{\Delta}$ 。
3. 对于所有的  $\text{id} \in \Delta.ids$ , 如果存在事件  $e$  使得  $e \in \text{events}(\text{id}, \text{rej})_{\Delta} \cup \text{events}(\text{id}, \text{rtn})_{\Delta}$ , 那么  $\text{events}(\text{id}, \text{exe})_{\Delta} = \emptyset$ 。
4. 对于所有的  $\text{id} \in \Delta.ids$ , 如果存在事件  $e$  使得  $e \in \text{events}(\text{id}, \text{acpt})_{\Delta}$ , 那么对于所有的  $\iota$ , 存在唯一的事件  $e'$  使得  $e' \in \text{events}(\iota)_{\Delta} \cap \text{events}(\text{id}, \text{exe})_{\Delta}$ 。
5. 在  $\Delta$  中, 不存在一个事件  $e$  使得  $e \xrightarrow{\text{hb}}_{\Delta} e$ 。

**引理 A.1** 对于所有的库  $\Sigma$ , 轨迹  $\Delta$  和 `idle` 状态的系统  $\Omega$   $\Omega'$ , 如果  $\Sigma \vdash \Omega \longrightarrow_{\Delta}^* \Omega'$ , 那么轨迹  $\Delta$  是结构良好的。

接下来, 我们用他们来证明下述引理 A.2。

**引理 A.2** 对于所有的库  $\Sigma$ , 轨迹  $\Delta$  和 `idle` 状态的系统  $\Omega$   $\Omega'$ , 如果  $\Sigma \vdash \Omega \longrightarrow_{\Delta}^* \Omega'$ , 那么轨迹  $\Delta$  满足一致性。

通过引理 A.2, 我们知道定理 3.1 成立。 □

### A.1 引理 A.1 的证明

为了证明引理 A.1, 我们只需要证明以下 (1)(2)(3)(4)(5) 点即可。

- (1) 每一个  $\Delta$  中的元素是唯一的。

$$\Delta.ids \stackrel{\text{def}}{=} \{\text{id} \mid (\_, \text{id}, \_) \in \Delta\} \quad L.his \stackrel{\text{def}}{=} h, \text{ 其中 } (h, \_) = L$$

$$\Sigma \vdash \Omega \longrightarrow_{\Delta}^{*} \Omega' \stackrel{\text{def}}{=} \exists \mathbb{S} \mathbb{S}' \mathbb{P} \mathbb{P}', ((\mathbb{P}, \mathbb{S}), (\Sigma, \Omega)) \longrightarrow_{\Delta}^{*} ((\mathbb{P}', \mathbb{S}'), (\Sigma, \Omega'))$$

图 A.1 证明一致性的一些辅助定义

假设结论不成立，即

$$\exists \iota \pi \pi', \exists \text{id} \in \Delta.ids, (\text{id}, \iota, \pi) \in \Delta \wedge (\text{id}, \iota, \pi') \in \Delta \wedge \pi = \pi'。$$

通过引理 A.14 我们知道

$$\text{id} \in \text{dom}(M') \setminus \text{dom}(M)。$$

我们对  $\pi$  进行分情况讨论。

- $\pi = \text{rej} \vee \text{rtn} \vee \text{acpt}_{\_}$ 。

这个假设与引理 A.7 冲突。

- $\pi = \text{op}_{\_}$ 。

这个假设与引理 A.8 冲突。

- $\pi = \text{exe}$ 。

这个假设与引理 A.9 冲突。

综上，假设不成立，结论成立。即每一个  $\Delta$  中的元素是唯一的。

(2) 对于所有的  $\text{id} \in \Delta.ids$ , 存在唯一的  $e$  使得  $e \in \text{events}(\text{id}, \text{rej})_{\Delta} \cup \text{events}(\text{id}, \text{rtn})_{\Delta} \cup \text{events}(\text{id}, \text{acpt})_{\Delta}$ 。

令

$$(M, \_) = \Omega \wedge (M', \_) = \Omega'。$$

通过引理 A.14 我们知道

$$\Delta.ids = \text{dom}(M') \setminus \text{dom}(M)。$$

对于所有的  $\text{id} \in \Delta.ids$ , 通过  $\text{idle}$  的定义, 我们可以对  $M'(\text{id})$  进行分情况讨论:

- 存在  $\iota r C h$  使得  $M'(\text{id}) = (\iota, r, C, h)$ 。

通过引理 A.10, 我们知道

$$\exists! e \text{ that } e \in \text{events}(\text{id}, \text{acpt})_{\Delta}。$$

- $M'(\text{id}) = \nabla$ 。

通过引理 A.7, 我们知道

$$\exists! e, e \in \text{events}(\text{id}, \text{rej})_{\Delta} \vee e \in \text{events}(\text{id}, \text{rtn})_{\Delta}。$$

因此声明成立。

(3) 对于所有的  $\text{id} \in \Delta.ids$ , 如果存在  $e$  使得  $e \in \text{events}(\text{id}, \text{rej})_{\Delta} \cup \text{events}(\text{id}, \text{rtn})_{\Delta}$ , 那么  $\text{events}(\text{id}, \text{exe}) = \emptyset$ 。

通过引理 A.14 我们知道

$$\Delta.ids = \text{dom}(M') \setminus \text{dom}(M)。$$

通过引理 A.7 我们知道

$$M'(\text{id}) = \nabla。$$

假设

$$\exists e, e \in \text{events}(\text{id}, \text{exe})_{\Delta}。$$

通过引理 A.12, 我们知道

$$\exists! e', e' \in \text{events}(\text{id}, \text{acpt})_{\Delta}。$$

通过引理 A.10, 我们知道

$$\exists \iota r C h, M'(\text{id}) = (\iota, r, C, h),$$

这与

$$M'(\text{id}) = \nabla$$

冲突。因此声明成立。

(4) 对于所有的  $\text{id} \in \Delta.ids$ , 如果存在  $e$  使得  $e \in \text{events}(\text{id}, \text{acpt})_{\Delta}$ , 那么对于所有的  $\iota, \exists! e$  使得  $e \in \text{events}(\iota)_{\Delta} \cap \text{events}(\text{id}, \text{exe})_{\Delta}$ 。

通过引理 A.10, 我们知道

$$\forall \iota, \exists r C h, M'(\text{id}) = (\iota, r, C, h)。$$

通过定义 **idle**, 我们知道

$$\text{id} \in S'(\iota).his。$$

因为

$$\text{id} \notin \text{dom}(M),$$

通过定义 **idle**, 可得

$$\text{id} \notin S(\iota).his。$$

通过引理 A.9, 我们知道声明成立。

(5) 在  $\Delta$  中, 不存在一个事件  $e$  使得  $e \xrightarrow{\text{hb}}_{\Delta} e$ 。

通过引理 A.15 可证。

证明结束。 □

## A.2 引理 A.2 的证明

为了证明这个断言, 我们只需要证明下述 (1)(2)(3) 均成立。

(1) 对于所有的  $\text{id}_A \text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{po}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。

通过定义 3.6, 我们知道

$$\text{acpt\_on}(\text{id}_A)_{\Delta} = \text{acpt\_on}(\text{id}_B)_{\Delta} \text{ 和 } \text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B。$$

通过引理 A.1, 可得

$\exists! e_A e_B, e_A \in \text{events}(\text{id}_A, \text{op})_\Delta \wedge e_B \in \text{events}(\text{id}_B, \text{op})_\Delta \wedge e_A \lessdot_\Delta e_B$ 。

通过定义  $\text{acpt\_on}$ , 可得

$\exists! e'_A e'_B, e_A \in \text{events}(\text{id}_A, \text{acpt\_})_\Delta \wedge e_B \in \text{events}(\text{id}_B, \text{acpt\_})_\Delta$ 。

假设

$$e'_A = (\iota, \text{id}_A, \text{acpt\_}) \wedge e'_B = (\iota, \text{id}_B, \text{acpt\_})$$

通过引理 A.1, 我们知道

$$(\iota, \text{id}_A, \text{exe}) \in \Delta$$

为了证明这个断言, 通过定义 3.1, 我们只需要证明

$$(\iota, \text{id}_A, \text{exe}) \xrightarrow[\iota]{\text{so}} e'_B$$

我们知道

$$\exists \Omega \Omega', \Sigma \vdash \Omega \longrightarrow^*_\Delta \Omega'$$

那么我们知道

$$\begin{aligned} & \exists M_1 M_2 S_1 S_2 \Delta_1 \Delta_2, \Sigma \vdash (M, S) \longrightarrow^*_{\Delta_1} (M_1, S_1) \wedge \\ & \Sigma \vdash (M_1, S_1) \xrightarrow{e'_B} (M_2, S_2) \wedge \Sigma \vdash (M_2, S_2) \longrightarrow^*_{\Delta_2} (M', S') \end{aligned}$$

因此我们只需要证明

$$(\iota, \text{id}_A, \text{exe}) \in \Delta_1$$

通过定义  $\_ \vdash \_ \Rightarrow \_$ , 我们知道

$$h_t \cap \text{on}(\iota)_{M_1} \subseteq \text{his\_of}(\iota)_{(M_1, S_1)} \wedge M_1 = (\iota, \text{id}_B, \_, h_t)$$

因为我们有

$$e_A \lessdot_\Delta e_B$$

可得

$$\text{id}_A \in h_t$$

通过

$$\text{id}_A \in \text{on}(\iota)_{M_1}$$

可得

$$\text{id}_A \in \text{his\_of}(\iota)_{(M_1, S_1)}$$

通过引理 A.5, 我们知道

$$\text{id}_A \in \text{his\_of}(\iota)_{(M', S')}$$

通过引理 A.8, 可得

$$\text{id}_A \in \text{his\_of}(\iota)_{(M', S')} \setminus \text{his\_of}(\iota)_{(M, S)}$$

因此可得

$$\text{id}_A \in \text{his\_of}(\iota)_{(M_1, S_1)} \setminus \text{his\_of}(\iota)_{(M, S)}$$

通过引理 A.8, 可得

$$(\iota, \text{id}_A, \text{exe}) \in \Delta_1$$

证明结束。

(2) 对于所有的  $\text{id}_A \text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{sync}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。

通过定义 3.4, 可得

$$\text{sync}(\text{id}_A, \text{id}_B)_{\Delta} \wedge \text{id}_A \xrightarrow{\text{ao}}_{\Delta} \text{id}_B.$$

通过定义 3.3 和引理 A.1, 我们知道

$$\exists! e_A e_B, e_A \in \text{events}(\text{id}_A, \text{op})_{\Delta} \wedge e_B \in \text{events}(\text{id}_B, \text{op})_{\Delta} \wedge e_A \leq_{\Delta} e_B.$$

因此

$$\exists \iota_A \iota_B \tau_A \tau_B, e_A = (\iota_A, \text{id}_A, \text{op} \tau_A) \wedge e_B = (\iota_B, \text{id}_B, \text{op} \tau_B).$$

通过引理 A.1, 我们知道

$$\exists! e'_B, e'_B \in \text{events}(\text{id}_B, \text{acpt})_{\Delta}.$$

因此

$$\exists \iota', e_B = (\iota', \text{id}_B, \text{acpt}).$$

通过引理 A.1, 我们知道

$$\exists! e'_A, e'_A \in \text{events}(\iota') \cap \text{events}(\text{id}_A, \text{exe})_{\Delta}.$$

因此

$$e'_A = (\iota', \text{id}_A, \text{exe})_{\Delta}.$$

为了证明这个断言, 通过定义 3.1,

我们只需要证明

$$e'_A \leq_{\Delta} e'_B.$$

通过定义 sync, 可得

$$\tau_B.O_s \cap \tau_A.O_w \neq \emptyset.$$

我们知道

$$\exists \Omega \Omega', \Sigma \vdash \Omega \longrightarrow_{\Delta}^{*} \Omega'.$$

令

$$(M, S) = \Omega \wedge (M', S') = \Omega'.$$

我们知道

$$\begin{aligned} \exists M_1 M_2 S_1 S_2 \Delta_1 \Delta_2, \Sigma \vdash (M, S) \longrightarrow_{\Delta_1}^{*} (M_1, S_1) \wedge \Sigma \vdash (M_1, S_1) \xrightarrow{e'_B} (M_2, S_2) \\ \wedge \Sigma \vdash (M_2, S_2) \longrightarrow_{\Delta_2}^{*} (M', S'). \end{aligned}$$

通过引理 A.13, 我们知道

$$e_B \leq_{\Delta} e'_B \wedge e_B \in \Delta_1.$$

通过引理 A.8, 我们知道

$$\text{id}_B \in \text{dom}(M_1).$$

假设

$$\exists v h, M_1(\text{id}_B) = (\iota_B, \tau_B, v, h).$$

因为

$$e_A \lessdot_{\Delta} e_B,$$

我们知道

$$\text{id}_A \in h.$$

因为

$$\tau_B.O_s \cap \tau_A.O_w \neq \emptyset,$$

通过定义 wr, 可得

$$\text{id}_A \in \text{wr}(\tau_B.O_s)_{M_1}.$$

因此

$$\text{id}_A \in h \cap \text{wr}(\tau_B.O_w)_{M_1}.$$

通过规则 ACPT, 我们知道

$$\text{id}_A \in S_1(\iota').his.$$

假设

$$\text{id}_A \in S(\iota').his.$$

通过引理 A.5 和规则 EXE, 我们知道

$$\nexists e, e \in \text{events}(\text{id}_A, \text{exe})_{\Delta},$$

此声明与假设矛盾。

因此

$$\text{id}_A \notin S(\iota').his.$$

因为

$$\text{id}_A \notin S(\iota').his \wedge \text{id}_A \in S_1(\iota').his,$$

通过引理 A.9 我们知道

$$\exists! e, e \in \text{events}(\iota')_{\Delta_1} \cap \text{events}(\text{id}_A, \text{exe})_{\Delta_1}.$$

因此

$$(\iota', \text{id}_A, \text{exe}) \in \Delta_1.$$

因此

$$e'_A \in \Delta_1.$$

因此

$$e'_A \lessdot_{\Delta} e'_B.$$

因此

$$\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B.$$

(3) 对于所有的  $\text{id}_A \text{id}_B$ , 如果  $\text{id}_A \xrightarrow{\text{dep}}_{\Delta} \text{id}_B$ , 那么  $\text{id}_A \xrightarrow{\text{eo}}_{\Delta} \text{id}_B$ 。

通过定义 3.5, 可得

$$\text{dep}(\text{id}_A, \text{id}_B)_{\Delta} \text{ 和 } \text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B.$$

通过定义 3.1 和引理 A.1, 我们知道

$\exists! e_A e_B, e_A \in \text{events}(\text{id}_A, \text{exe})_\Delta \wedge e_B \in \text{events}(\text{id}_B, \text{acpt})_\Delta \wedge e_A \lessdot_\Delta e_B$ 。

因此

$$\exists \iota \tau_A \tau_B, e_A = (\iota, \text{id}_A, \text{exe}) \wedge e_B = (\iota, \text{id}_B, \text{acpt})。$$

通过引理 A.1, 我们知道

$$\forall \iota', \exists! e'_A e'_B, e'_A \in \text{events}(\iota')_\Delta \cap \text{events}(\text{id}_A, \text{exe})_\Delta \wedge e'_B \in \text{events}(\iota')_\Delta \cap \text{events}(\text{id}_B, \text{exe})_\Delta。$$

因此

$$e'_A = (\iota', \text{id}_A, \text{exe}) \wedge e'_B = (\iota', \text{id}_B, \text{exe})。$$

为了证明这个断言, 通过定义 3.2,

我们只需要证明

$$e'_A \lessdot_\Delta e'_B。$$

我们知道

$$\exists \Omega \Omega', \Sigma \vdash \Omega \longrightarrow^*_\Delta \Omega'.$$

令

$$(M, S) = \Omega \wedge (M', S') = \Omega'.$$

通过引理 A.12, 可得

$$e_B \lessdot_\Delta e'_B。$$

因此我们有

$$\begin{aligned} & \exists M_1 \cdots M_4 S_1 \cdots S_4 \Delta_1 \cdots \Delta_3, \Sigma \vdash (M, S) \longrightarrow^*_{\Delta_1} (M_1, S_1) \wedge \\ & \Sigma \vdash (M_1, S_1) \xrightarrow{e_B} (M_2, S_2) \wedge \Sigma \vdash (M_2, S_2) \longrightarrow^*_{\Delta_2} (M_3, S_3) \wedge \\ & \Sigma \vdash (M_3, S_3) \xrightarrow{e'_B} (M_4, S_4) \wedge \Sigma \vdash (M_4, S_4) \longrightarrow^*_{\Delta_3} (M', S') \end{aligned}$$

因为  $e_A \lessdot_\Delta e_B$ , 我们知道

$$e_A \in \Delta_1。$$

通过引理 A.9, 可得

$$\text{id}_A \in S_1(\iota).his。$$

通过引理 A.14, 我们知道

$$\text{id}_A \in M_1。$$

假设

$$M_1(\text{id}_B) = (\_, \tau_B, \_, \_)$$

通过定义 dep, 可得

$$\text{id}_A \in \text{wr}(\text{id}_B.O_d)_{M_1}。$$

因为

$$\text{id}_A \in S_1(\iota).his,$$

我们知道

$$\text{id}_A \in S_1(\iota).his \cap \text{wr}(\text{id}_B.O_d)_{M_1}。$$

因此通过规则 **ACPT**, 我们知道

$$\exists h, M_2(\text{id}_B) = (\_, \_, \_, h) \wedge \text{id}_A \in h.$$

通过引理 A.6, 我们知道

$$M_2(\text{id}_B) = M_3(\text{id}_B).$$

因此, 通过规则 **EXE**, 可得

$$\text{id}_A \in S_3(\iota').\text{his}.$$

通过引理 A.14 和定义 **idle**, 我们知道

$$\text{id}_A \notin S(\iota').\text{his}.$$

令

$$\Delta_4 = \Delta_1 ++ [e_B] ++ \Delta_2.$$

通过引理 A.9, 我们知道

$$\exists! e, e \in \text{events}(\iota')_{\Delta_4} \cap \text{events}(\text{id}_A, \text{exe})_{\Delta_4}.$$

因此

$$(\iota', \text{id}_A, \text{exe}) \in \Delta_4.$$

因此

$$e'_A \in \Delta_4.$$

因此

$$e'_A \leqslant_{\Delta} e'_B.$$

因此

$$\text{id}_A \xrightarrow{\text{eo}}_{\Delta} \text{id}_B.$$

证明结束。 □

### A.3 其他引理的证明

**引理 A.3** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $(M, S)$  ( $M', S'$ ), 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^{*} (M', S')$ , 那么对于所有的  $\text{id}$ , 如果  $M(\text{id}) = \nabla$ , 那么  $M'(\text{id}) = \nabla$ 。

**证明** 因为

$$\text{id} \in \text{dom}(M') \wedge M'(\text{id}) = \nabla,$$

通过在  $\Delta$  上进行归纳我们知道

$$M'(\text{id}) = \nabla.$$

证明结束。 □

**引理 A.4** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $(M, S)$  ( $M', S'$ ), 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^{*} (M', S')$ , 那么  $\text{dom}(M) \subseteq \text{dom}(M')$ 。

**证明** 通过在  $\Delta$  上进行归纳, 我们只需要证明

$$\forall e (M, S) (M', S'), \Sigma \vdash (M, S) \xrightarrow{e} (M', S') \Rightarrow \text{dom}(M) \subseteq \text{dom}(M')。$$

为了证明这个声明, 我们对  $e$  进行分情况讨论:

- $e = (\_, \_, \mathbf{rtn}) \vee (\_, \_, \mathbf{rej}) \vee (\_, \_, \mathbf{exe}) \vee (\_, \_, \mathbf{acpt} \_)$ 。

通过规则 RTN, REJ 和 EXE, 可得

$$\text{dom}(M) = \text{dom}(M')。$$

- $e = (\_, \_, \mathbf{op} \_)$ , 那么通过规则 OP, 我们知道

$$\text{dom}(M) \subseteq \text{dom}(M')。$$

因此  $\text{dom}(M) \subseteq \text{dom}(M')$ 。  $\square$

**引理 A.5** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $\Omega \Omega'$ , 如果  $\Sigma \vdash \Omega \longrightarrow_{\Delta}^{*} \Omega'$ , 那么对于所有的  $\iota$ , 我们有  $\text{his\_of}(\iota)_{\Omega} \subseteq \text{his\_of}(\iota)_{\Omega'}$ 。

**证明** 与引理 A.4 证明相似。  $\square$

**引理 A.6** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $(M, S) (M', S')$ , 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^{*} (M', S')$ , 那么对于所有的  $\text{id}$ , 如果存在  $\iota r C h$ , 使得  $M(\text{id}) = (\iota, r, C, h)$ , 那么  $M'(\text{id}) = (\iota, r, C, h)$ 。

**证明** 与引理 A.3 证明相似。  $\square$

**引理 A.7** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $(M, S) (M', S')$ , 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^{*} (M', S')$ , 那么对于所有的  $\text{id}$ ,  $\text{id} \in \text{dom}(M') \setminus \text{dom}(M)$ ,  $M'(\text{id}) = \nabla \Leftrightarrow \exists! e$  使得  $e \in \text{events}(\text{id}, \mathbf{rej})_{\Delta}$  或  $e \in \text{events}(\text{id}, \mathbf{rtn} \_)_{\Delta}$ 。

**证明**

$\Leftarrow$ : 我们知道

$$\exists! e, e \in \text{events}(\text{id}, \mathbf{rej})_{\Delta} \vee e \in \text{events}(\text{id}, \mathbf{rtn})_{\Delta}。$$

因此

$$\begin{aligned} & \exists M'' M''' S'' S''' \Delta' \Delta'', \Sigma \vdash (M, S) \longrightarrow_{\Delta'}^{*} (M'', S'') \wedge \\ & \Sigma \vdash (M'', S'') \xrightarrow{e} (M''', S''') \wedge \Sigma \vdash (M''', S''') \longrightarrow_{\Delta''}^{*} (M', S')。 \end{aligned}$$

通过规则 RTN 和 REJ, 可得

$$M'''(\text{id}) = \nabla。$$

通过引理 A.3 我们知道

$$M'(\text{id}) = \nabla。$$

$\Rightarrow$ : 通过规则 REJ 和 RTN 我们知道

$$\exists e, e \in \text{events}(\text{id}, \text{rej})_{\Delta} \vee e \in \text{events}(\text{id}, \text{rtn})_{\Delta}.$$

那么可得

$$\begin{aligned} & \exists M'' M''' S'' S''' \Delta' \Delta'', \Sigma \vdash (M, S) \xrightarrow{\Delta'}^* (M'', S'') \wedge \\ & \Sigma \vdash (M'', S'') \xrightarrow{e} (M''', S''') \wedge \Sigma \vdash (M''', S''') \xrightarrow{\Delta''}^* (M', S') \wedge M''(\text{id}) \neq \nabla \wedge \\ & M'''(\text{id}) = \nabla. \end{aligned}$$

为了证明这个断言，

我们只需要证明下述 1、2 两点均成立：

1.  $\nexists e, e \in \text{events}(\text{id}, \text{rej})_{\Delta'} \cup \text{events}(\text{id}, \text{rtn})_{\Delta'}.$

假设

$$\exists e', e' \in \text{events}(\text{id}, \text{rej})_{\Delta'} \cup \text{events}(\text{id}, \text{rtn})_{\Delta'}$$

我们知道

$$\begin{aligned} & \exists M_A M_B S_A S_B \Delta_A \Delta_B, \Sigma \vdash (M, S) \xrightarrow{\Delta_A}^* (M_A, S_A) \wedge \\ & \Sigma \vdash (M_A, S_A) \xrightarrow{e'} (M_B, S_B) \wedge \Sigma \vdash (M_B, S_B) \xrightarrow{\Delta_B}^* (M'', S''). \end{aligned}$$

通过规则 RTN 和 REJ, 可得

$$M_B(\text{id}) = \nabla.$$

那么通过引理 A.3 我们知道

$$M''(\text{id}) = \nabla,$$

这与

$$M''(\text{id}) \neq \nabla$$

冲突。因此声明成立。

2.  $\nexists e, e \in \text{events}(\text{id}, \text{rej})_{\Delta''} \cup \text{events}(\text{id}, \text{rtn})_{\Delta''}.$

因为

$$\text{id} \in \text{dom}(M'') \wedge M''(\text{id}) = \nabla,$$

通过在  $\Delta''$  上进行归纳，我们知道声明成立。

证明结束。 □

**引理 A.8** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和 idle 系统  $(M, S)$  ( $M', S'$ ), 如果  $\Sigma \vdash (M, S) \xrightarrow{\Delta}^* (M', S')$ , 那么对于所有的  $\text{id}, \text{id} \in \text{dom}(M') \setminus \text{dom}(M) \Leftrightarrow \exists! e$  使得  $e \in \text{events}(\text{id}, \text{op})_{\Delta}$ 。

**证明** 通过应用引理 A.6 可证，与引理 A.7 证明相似。 □

**引理 A.9** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$ , idle 的系统  $\Omega$  和系统  $\Omega'$ , 如果  $\Sigma \vdash \Omega \xrightarrow{\Delta}^* \Omega'$  那么对于所有的  $i, \text{id}, \text{id} \in \text{his\_of}(i)'_{\Omega} \setminus \text{his\_of}(i)_{\Omega} \Leftrightarrow \exists! e$  使得  $e \in \text{events}(i)_{\Delta} \cap \text{events}(\text{id}, \text{exe})_{\Delta}$ 。

**证明** 与引理 A.8 证明相似。 □

**引理 A.10** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$  和系统  $(M, S)$  ( $M', S'$ ), 如果  $\Sigma \vdash (M, S) \rightarrow_{\Delta}^* (M', S')$ , 那么对于所有的  $\text{id}, \text{id} \in \text{dom}(M') \setminus \text{dom}(M)$ , 存在  $\iota r C h$ , 使得  $M'(\text{id}) = (\iota, r, C, h) \Leftrightarrow \exists! e$  使得  $e \in \text{events}(\text{id}, \text{acpt})_{\Delta}$ 。

**证明** 通过应用引理 A.4 可证, 与证明引理 A.7 相似。  $\square$

**引理 A.11** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$ , `idle` 的系统  $(M, S)$  和系统  $(M', S')$ , 如果  $\Sigma \vdash (M, S) \rightarrow_{\Delta}^* (M', S')$  和  $\exists e$  使得  $e \in \text{events}(\text{id}, \text{exe})_{\Delta}$ , 那么  $\text{id} \notin \text{dom}(M)$ 。

**证明** 假设

$$\text{id} \in \text{dom}(M).$$

我们对  $M(\text{id})$  进行分情况讨论:

- $M(\text{id}) \neq \nabla$ 。

通过定义 `idle`, 我们知道

$$\forall \iota, \text{id} \in S(\iota).his.$$

通过引理 A.5 和规则 `EXE`, 我们知道

$$\nexists e, e \in \text{events}(\text{id}, \text{exe})_{\Delta},$$

此声明与假设矛盾。

- $M(\text{id}) = \nabla$ 。

通过引理 A.3 和规则 `EXE`, 我们知道

$$\nexists e, e \in \text{events}(\text{id}, \text{exe})_{\Delta},$$

此声明与假设矛盾。

因此  $\text{id} \notin \text{dom}(M)$ 。  $\square$

**引理 A.12** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$ , `idle system`  $(M, S)$  和系统  $(M', S')$ , 如果  $\Sigma \vdash (M, S) \rightarrow_{\Delta}^* (M', S')$  和  $\exists e$  使得  $e \in \text{events}(\text{id}, \text{exe})_{\Delta}$ , 那么  $\exists! e'$  使得  $e' \in \text{events}(\text{id}, \text{acpt})_{\Delta}$  和  $e' \ll_{\Delta} e$ 。

**证明** 因为

$$e \in \text{events}(\text{id}, \text{exe})_{\Delta},$$

我们知道

$$\begin{aligned} & \exists M_A M_B S_A S_B \Delta_A \Delta_B, \Sigma \vdash (M, S) \rightarrow_{\Delta_A}^* (M_A, S_A) \wedge \\ & \Sigma \vdash (M_A, S_A) \xrightarrow{e} (M_B, S_B) \wedge \Sigma \vdash (M_B, S_B) \rightarrow_{\Delta_B}^* (M', S'). \end{aligned}$$

通过规则 `EXE`, 我们知道

$$\exists \iota r C h, M_B(\text{id}) = (\iota, r, C, h).$$

通过引理 A.11, 我们知道

$$\text{id} \notin \text{dom}(M)。$$

因此

$$\text{id} \in \text{dom}(M_B) \setminus \text{dom}(M)。$$

通过引理 A.10, 我们知道

$$\exists! e', e' \in \text{events}(\text{id}, \text{acpt})_{\Delta_A} \wedge e' \lessdot_{\Delta} e。$$

通过  $\Delta_B$  在上进行归纳以及引理 A.6, 我们知道

$$\nexists e', e' \in \text{events}(\text{id}, \text{acpt})_{\Delta_B}。$$

因此

$$\exists! e', e' \in \text{events}(\text{id}, \text{acpt})_{\Delta} \wedge e' \lessdot_{\Delta} e。$$

证明结束。  $\square$

**引理 A.13** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$ , `idle` 的系统  $(M, S)$  和系统  $(M', S')$ , 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^* (M', S')$  和  $\exists e$  使得  $e \in \text{events}(\text{id}, \text{acpt})_{\Delta}$ , 那么  $\exists! e'$  使得  $e' \in \text{events}(\text{id}, \text{op\_})_{\Delta}$  和  $e' \lessdot_{\Delta} e$ 。

**证明** 与引理 A.12 证明相似。  $\square$

**引理 A.14** 对于任何的库  $\Sigma$ , 轨迹  $\Delta$ , `idle` 的系统  $(M, S)$  和系统  $(M', S')$ , 如果  $\Sigma \vdash (M, S) \longrightarrow_{\Delta}^* (M', S')$ , 那么  $\Delta.ids = \text{dom}(M') \setminus \text{dom}(M)$ 。

**证明** 为了证明这个断言, 我们只需要证明下述 1、2 两点均成立:

1.  $\Delta.ids \subseteq (\text{dom}(M') \setminus \text{dom}(M))$ 。

因此

$$\forall \text{id}, \text{id} \in \Delta.ids \Rightarrow \text{id} \notin \text{dom}(M) \wedge \text{id} \in \text{dom}(M')。$$

通过规则 `OP`, 我们知道

$$\text{id} \in \text{dom}(M')。$$

为了证明

$$\text{id} \notin \text{dom}(M),$$

假设

$$\text{id} \in \text{dom}(M),$$

我们对  $M(\text{id})$  进行分情况讨论:

- $\exists \iota r C h, M(\text{id}) = (\iota, r, C, h)$ 。

通过定义 `idle`, 我们知道

$$\forall \iota, \text{id} \in S(\iota).his.$$

通过引理 A.6 和引理 A.5 我们知道

$$\nexists e, (\_, \text{id}, \_) \in \Delta.ids,$$

此声明与假设矛盾。

- $M(\text{id}) = \nabla$ 。

通过引理 A.3 我们知道

$$\nexists e, (\_, \text{id}, \_) \in \Delta.ids,$$

此声明与假设矛盾。

因此声明成立。

$$2. \Delta.ids \supseteq (dom(M') \setminus dom(M))。$$

因此

$$\forall \text{id}, (\text{id} \notin dom(M) \wedge \text{id} \in dom(M')) \Rightarrow \text{id} \in \Delta.ids.$$

通过规则 OP 和引理 A.4, 我们知道

$$\exists \iota \tau, (\iota, \text{id}, \text{op } \tau) \in \Delta.$$

因此

$$\text{id} \in \Delta.ids.$$

因此声明成立。

证明结束。  $\square$

**引理 A.15** 对于所有的  $\Omega \Delta \Sigma$ , 如果  $\text{idle}(\Omega)$  和  $\Sigma \vdash \Omega \xrightarrow{\Delta} \_$ , 那么不存在一个事件  $e \in \Delta$  使得  $e \xrightarrow{\text{hb}} e$ 。

**证明** 通过  $\Delta$  在上进行归纳。

**归纳基础:**  $\Delta = \perp$ 。

易知成立。

**归纳假设:**  $\Delta = e : \Delta'$ 。

通过归纳假设, 我们知道

$$\nexists e' \in \Delta', e' \xrightarrow{\text{hb}}_{\Delta'} e'.$$

假设

$$\exists \text{id} \iota \pi, e = (\text{id}, \iota, \pi).$$

通过定义  $\_ \xrightarrow{\text{so}} \_$ , 因为

$$\forall e' \in \Delta', e' \lessdot_{\Delta} e,$$

可得

$$\nexists e' \in \Delta', e \xrightarrow[\iota]{\text{so}}_{\Delta'} e'.$$

通过定义 A.2, 我们只需要证明

$$\nexists e' \in \Delta', e \xrightarrow{\text{bo}}_{\Delta'} e'.$$

我们对进行分情况讨论  $\pi$ :

- $\pi = \mathbf{rej} \vee \mathbf{rtn} \vee \mathbf{op\_} \vee \mathbf{exe}$ 。

通过定义 A.1, 我们知道这个声明成立。

- $\pi = \mathbf{acpt\_}$ 。

通过定义 A.1, 假设

$$\exists e_B, e_B \in \text{events}(\text{id}, \mathbf{exe})_{\Delta'}.$$

通过引理 A.12 我们知道

$$\exists! e', e' \in \text{events}(\text{id}, \mathbf{acpt})_{\Delta'}.$$

因为

$$e = e' \wedge e \in \Delta \wedge e' \in \Delta,$$

通过引理 A.12, 可得矛盾。

因此声明成立。

□

## A.4 命题 3.2 的证明

我们知道

$$\text{id}_A \xrightarrow{\text{ao}} \Delta \text{id}_B \vee \text{id}_B \xrightarrow{\text{ao}} \Delta \text{id}_A.$$

因为我们有

$$\text{sync}(\text{id}_A, \text{id}_B) \wedge \text{sync}(\text{id}_B, \text{id}_A)$$

通过定义 3.4, 可得

$$\text{id}_A \xrightarrow{\text{sync}} \Delta \text{id}_B \vee \text{id}_B \xrightarrow{\text{sync}} \Delta \text{id}_A.$$

通过定义 3.7, 我们知道

$$\text{id}_A \xrightarrow{\text{vo}} \Delta \text{id}_B \vee \text{id}_B \xrightarrow{\text{vo}} \Delta \text{id}_A.$$

证明结束。

## 附录 B 命题 3.3、3.4、3.5 和 3.6 的证明

命题 3.3 的证明。

通过定理 3.1, 我们知道

$\Delta$  满足一致性。

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_d = \mathbb{U}_\Sigma.$$

我们知道

$$\forall \tau \tau', \text{dep}(\tau, \tau').$$

因此, 我们有

$$\forall \text{id id}', \text{dep}(\text{id}, \text{id}').$$

因此, 如果

$$\text{id}_A \xrightarrow{\text{vo}}_\Delta \text{id}_B,$$

通过定义 3.5, 我们知道

$$\text{id}_A \xrightarrow{\text{dep}}_\Delta \text{id}_B,$$

通过定义 3.7, 可得

$$\text{id}_A \xrightarrow{\text{eo}}_\Delta \text{id}_B.$$

证明结束。

命题 3.4 的证明。

通过定理 3.1, 我们知道

$\Delta$  满足一致性。

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_d = \text{rset}(\tau).$$

因此, 我们知道

$$\text{op}(\text{id}_A)_\Delta.O_w \cap \text{op}(\text{id}_B)_\Delta.O_d \neq \emptyset$$

因此

$$\text{dep}(\text{id}_A, \text{id}'_B).$$

由于我们有

$$\text{id}_A \xrightarrow{\text{vo}}_\Delta \text{id}_B,$$

通过定义 3.5, 我们知道

$$\text{id}_A \xrightarrow{\text{dep}}_\Delta \text{id}_B,$$

通过定义 3.7, 可得

$$\text{id}_A \xrightarrow{\text{eo}}_\Delta \text{id}_B.$$

证明结束。

命题 3.5 的证明。

通过定理 3.1, 我们知道

$\Delta$  满足一致性。

令  $R$  满足

$$(id_A \xrightarrow{\text{ao}} id_B \wedge id_A \in \Delta.acpts \wedge id_B \in \Delta.acpts) \Rightarrow R(id_A, id_B)。$$

那么我们知道

$R$  在  $\Delta.acpts$  上是一个全序。

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_s = \mathbb{U}_\Sigma。$$

我们知道

$$\text{sync}(id_A, id_B)。$$

通过定义 3.4, 可得

$$id_A \xrightarrow{\text{sync}}_\Delta id_B。$$

通过定义 3.7, 我们知道

$$id_A \xrightarrow{\text{vo}}_\Delta id_B。$$

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_d = \mathbb{U}_\Sigma。$$

我们知道

$$\text{dep}(id_A, id_B)。$$

通过定义 3.5, 可得

$$id_A \xrightarrow{\text{dep}}_\Delta id_B。$$

通过定义 3.7, 我们知道

$$id_A \xrightarrow{\text{eo}}_\Delta id_B。$$

证明结束。

命题 3.6 的证明。

通过定理 3.1, 我们知道

$\Delta$  满足一致性。

令  $R$  满足

$$(id_A \xrightarrow{\text{ao}} id_B \wedge o \in \text{wset}(\text{op}(id_A)_\Delta) \wedge o \in \text{wset}(\text{op}(id_B)_\Delta)) \Rightarrow R(id_A, id_B)。$$

那么我们知道

$R$  在  $\Delta.acpts \cap \Delta.wr(o)$  上是一个全序。

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_s = \text{wset}(\tau)。$$

我们知道

$$o \in \text{op}(\text{id}_A)_\Delta.O_w \wedge o \in \text{op}(\text{id}_A)_\Delta.O_s。$$

因此

$$\text{sync}(\text{id}_A, \text{id}_B)。$$

通过定义 3.4, 可得

$$\text{id}_A \xrightarrow{\text{sync}}_\Delta \text{id}_B。$$

通过定义 3.7, 我们知道

$$\text{id}_A \xrightarrow{\text{vo}}_\Delta \text{id}_B。$$

由于我们有

$$\forall \tau \in \Sigma.ops, \tau.O_d = \text{wset}(\tau)。$$

我们知道

$$o \in \text{op}(\text{id}_A)_\Delta.O_w \wedge o \in \text{op}(\text{id}_A)_\Delta.O_d。$$

因此

$$\text{dep}(\text{id}_A, \text{id}_B)。$$

通过定义 3.5, 可得

$$\text{id}_A \xrightarrow{\text{dep}}_\Delta \text{id}_B。$$

通过定义 3.7, 我们知道

$$\text{id}_A \xrightarrow{\text{eo}}_\Delta \text{id}_B。$$

证明结束。

## 附录 C 定理 4.1 的证明

为了证明这个声明, 通过引理 C.3, 我们只需要证明

$$\forall \text{id id}', (\text{id} \xrightarrow{\text{eo}} \Delta \text{id}' \vee \text{id}' \xrightarrow{\text{eo}} \Delta \text{id} \vee (\exists \kappa \kappa', ((\_, \text{id}, \text{acpt } \kappa') \in \Delta \wedge (\_, \text{id}', \text{acpt } \kappa') \in \Delta) \Rightarrow \text{commute}(\tau, \kappa, \tau', \kappa'))).$$

我们进行分情况讨论:

- $\text{id} \xrightarrow{\text{vo}} \Delta \text{id}'$ 。

通过规则 cv-TOP, 我们有一下几种情况:

1.  $\text{dep}(\text{op}(\text{id})_\Delta, \text{op}(\text{id}')_\Delta)$ 。

通过定理 3.1, 我们知道

$$\text{id} \xrightarrow{\text{eo}} \Delta \text{id}'.$$

2.  $\forall \kappa \kappa', \exists B, \text{exclusive } (\tau(\kappa), B)_\Sigma \wedge (\kappa' \notin [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))$ 。

通过引理 C.2, 我们知道

$$\kappa' \notin [B].$$

因此, 我们有

$$\text{commute}(\tau(\kappa), \tau'(\kappa')).$$

3.  $\forall \kappa \kappa', \nexists B, \text{exclusive } (\tau(\kappa), B)_\Sigma \wedge (\kappa' \notin [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))$ 。

通过规则 cv-COM 可证。

- $\text{id}' \xrightarrow{\text{vo}} \Delta \text{id}$ 。

与证明  $\text{id} \xrightarrow{\text{vo}} \Delta \text{id}'$  类似。

- $\neg \text{id} \xrightarrow{\text{vo}} \Delta \text{id}' \vee \neg \text{id}' \xrightarrow{\text{vo}} \Delta \text{id}$  类似。

通过命题 3.2 我们有以下两种情况:

1.  $\neg \text{sync}(\text{op}(\text{id})_\Delta, \text{op}(\text{id}')_\Delta)$ .

通过规则 cv-TOP, 我们有

- (a)  $\forall \kappa \kappa', \exists B, \text{stable } (\tau(\kappa), B)_\Sigma \wedge (\kappa' \in [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))$ 。

通过引理 C.1, 我们知道

$$\kappa' \in [B],$$

因此, 我们有

$$\text{commute}(\tau(\kappa), \tau'(\kappa')).$$

- (b)  $\forall \kappa \kappa', \nexists B, \text{stable } (\tau(\kappa), B)_\Sigma \wedge (\kappa' \in [B] \Rightarrow \text{commute}(\tau(\kappa), \tau'(\kappa')))$ 。

$$\begin{aligned}
 \mathbf{se}(\text{id})_{\Delta} &\stackrel{\text{def}}{=} \tau\langle\kappa\rangle, \text{ 其中 } \tau = \mathbf{op}(\text{id})_{\Delta} \wedge (\_, \text{id}, \mathbf{acpt} \kappa) \in \Delta \\
 \mathbf{eo\_se}(\Delta) &\stackrel{\text{def}}{=} \mathbf{se}(\text{id}_0)_{\Delta}, \dots, \mathbf{se}(\text{id}_i)_{\Delta}, \dots, \mathbf{se}(\text{id}_j)_{\Delta}, \dots, \mathbf{se}(\text{id}_n)_{\Delta} \\
 &\quad , \text{ 其中 } \{\text{id}_0, \dots, \text{id}_i, \dots, \text{id}_j, \dots, \text{id}_n\} = \Delta.ids \wedge (\text{id}_i \xrightarrow{\text{eo}}_{\Delta} \text{id}_j \Rightarrow i < j) \\
 \mathbf{site\_se}(\iota)_{\Delta} &\stackrel{\text{def}}{=} \mathbf{se}(\text{id}_0)_{\Delta}, \dots, \mathbf{se}(\text{id}_i)_{\Delta}, \dots, \mathbf{se}(\text{id}_j)_{\Delta}, \dots, \mathbf{se}(\text{id}_n)_{\Delta} \\
 &\quad , \text{ 其中 } \{\text{id}_0, \dots, \text{id}_i, \dots, \text{id}_j, \dots, \text{id}_n\} = \Delta.ids \wedge i < j \Leftrightarrow (\iota, \text{id}_i, \mathbf{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta} (\iota, \text{id}_j, \mathbf{exe}) \\
 \tau_0\langle\kappa_0\rangle \cdots \tau_n\langle\kappa_n\rangle &\equiv \tau'_0\langle\kappa'_0\rangle \cdots \tau'_n\langle\kappa'_n\rangle \stackrel{\text{def}}{=} \forall s s', \exists s_1 \cdots s_n, \tau_0\langle\kappa_0\rangle \vdash s \hookrightarrow s_1 \wedge \cdots \wedge \tau_n\langle\kappa_n\rangle \vdash s_n \hookrightarrow s' \Leftrightarrow \\
 &\quad \exists s'_1 \cdots s'_n, \tau'_0\langle\kappa'_0\rangle \vdash s \hookrightarrow s'_1 \wedge \cdots \wedge \tau'_n\langle\kappa'_n\rangle \vdash s'_n \hookrightarrow s'
 \end{aligned}$$

图 C.1 证明收敛性程序逻辑可靠性的一些辅助定义

通过规则 cv-COM 可证。

2.  $\neg \mathbf{sync}(\mathbf{op}(\text{id}')_{\Delta}, \mathbf{op}(\text{id})_{\Delta})$ 。

与证明  $\neg \mathbf{sync}(\mathbf{op}(\text{id})_{\Delta}, \mathbf{op}(\text{id}')_{\Delta})$  类似。

证明结束。  $\square$

**引理 C.1** 如果  $\mathbf{stable}(\tau\langle\kappa\rangle, B)_{\Sigma}$ , 那么对于所有满足  $\Sigma \vdash \Omega \vdash^{\Delta} \_$  的  $\Omega \Delta$ , 如果  $\exists \text{id}$  使得  $\mathbf{op}(\text{id}) = \tau, (\iota, \text{id}, \mathbf{acpt} \kappa) \in \Delta$ , 那么  $\forall \text{id}' \kappa'$ , 如果  $\text{id}' \in \Delta.ids$ ,  $\text{id} \xrightarrow{\text{vo}}_{\Delta} \text{id}'$ , 并且  $(\_, \text{id}', \mathbf{acpt} \kappa') \in \Delta$ , 那么  $\kappa' \in [B]$ 。

**证明** 通过定义 4.3 可证。  $\square$

**引理 C.2** 如果  $\mathbf{exclusive}(\tau\langle\kappa\rangle, B)_{\Sigma}$ , 那么对于所有满足  $\Sigma \vdash \Omega \vdash^{\Delta} \_$  的  $\Omega \Delta$ , 如果  $\exists \text{id}$  使得  $\mathbf{op}(\text{id}) = \tau, (\iota, \text{id}, \mathbf{acpt} \kappa) \in \Delta$ , 那么  $\forall \text{id}' \kappa'$ , 如果  $\text{id}' \in \Delta.ids$ ,  $\neg \text{id} \xrightarrow{\text{vo}}_{\Delta} \text{id}'$ , 并且  $(\_, \text{id}', \mathbf{acpt} \kappa') \in \Delta$ , 那么  $\kappa' \in [\neg B]$ 。

**证明** 通过定义 4.4 可证。  $\square$

**引理 C.3** 对于所有的  $\Sigma \Omega \Omega' \Delta$ , 如果  $\Sigma \vdash \Omega \vdash^{\Delta} \Omega'$  并且  $\forall \text{id} \text{id}' \in \Delta.ids, (\text{id} \xrightarrow{\text{eo}}_{\Delta} \text{id}' \vee \text{id}' \xrightarrow{\text{eo}}_{\Delta} \text{id} \vee (\exists \kappa \kappa', ((\_, \text{id}, \mathbf{acpt} \kappa) \in \Delta \wedge (\_, \text{id}', \mathbf{acpt} \kappa') \in \Delta) \Rightarrow \mathbf{commute}(\tau\langle\kappa\rangle, \tau'\langle\kappa'\rangle)))$ , 那么  $[\Omega']$ 。

**证明** 为了证明此引理, 我们证明

$$\forall \iota, \mathbf{site\_se}(\iota)_{\Delta} \equiv \mathbf{eo\_se}(\Delta).$$

因此

$$\begin{aligned}
 \mathbf{se}(\text{id}_0)_{\Delta}, \dots, \mathbf{se}(\text{id}_i)_{\Delta}, \dots, \mathbf{se}(\text{id}_j)_{\Delta}, \dots, \mathbf{se}(\text{id}_n)_{\Delta} &\equiv \\
 \mathbf{se}(\text{id}'_0)_{\Delta}, \dots, \mathbf{se}(\text{id}'_i)_{\Delta}, \dots, \mathbf{se}(\text{id}'_j)_{\Delta}, \dots, \mathbf{se}(\text{id}'_n)_{\Delta},
 \end{aligned}$$

其中

$$\begin{aligned} \{\text{id}_0, \dots, \text{id}_i, \dots, \text{id}_j, \dots, \text{id}_n\} &= \Delta.ids \wedge (\text{id}_i \xrightarrow{\text{eo}} \Delta \text{id}_j \Rightarrow i < j) \wedge \\ \{\text{id}'_0, \dots, \text{id}'_i, \dots, \text{id}'_j, \dots, \text{id}'_n\} &= \Delta.ids \wedge i < j \Leftrightarrow (\iota, \text{id}'_i, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'_j, \text{exe}) \end{aligned}$$

我们在 n 上做归纳。

**归纳基础:** n = 0。

易知成立。

**归纳假设:** 如果

$$\begin{aligned} \exists \text{id}''_0, \dots, \text{id}''_i, \dots, \text{id}''_j, \dots, \text{id}''_{k-1} \in \\ \Delta.ids, \text{se}(\text{id}''_0)_\Delta, \dots, \text{se}(\text{id}''_i)_\Delta, \dots, \text{se}(\text{id}''_j)_\Delta, \dots, \text{se}(\text{id}''_{k-1})_\Delta \equiv \\ \text{se}(\text{id}'_0)_\Delta, \dots, \text{se}(\text{id}'_i)_\Delta, \dots, \text{se}(\text{id}'_j)_\Delta, \dots, \text{se}(\text{id}'_{k-1})_\Delta, \text{其中} \\ (i < j \Leftrightarrow (\iota, \text{id}'_i, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'_j, \text{exe})) \wedge (\text{id}''_i \xrightarrow{\text{eo}} \Delta \text{id}''_j \Rightarrow i < j)。 \end{aligned}$$

那么

$$\begin{aligned} \exists \text{id}'''_0, \dots, \text{id}'''_i, \dots, \text{id}'''_j, \dots, \text{id}'''_k \in \\ \Delta.ids, \text{se}(\text{id}'''_0)_\Delta, \dots, \text{se}(\text{id}'''_i)_\Delta, \dots, \text{se}(\text{id}'''_j)_\Delta, \dots, \text{se}(\text{id}'''_k)_\Delta \equiv \\ \text{se}(\text{id}'_0)_\Delta, \dots, \text{se}(\text{id}'_i)_\Delta, \dots, \text{se}(\text{id}'_j)_\Delta, \dots, \text{se}(\text{id}'_k)_\Delta, \text{其中} \\ (i < j \Leftrightarrow (\iota, \text{id}'_i, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'_j, \text{exe})) \wedge (\text{id}'''_i \xrightarrow{\text{eo}} \Delta \text{id}'''_j \Rightarrow i < j)。 \end{aligned}$$

我们知道

$$\begin{aligned} \text{se}(\text{id}''_0)_\Delta, \dots, \text{se}(\text{id}''_i)_\Delta, \dots, \text{se}(\text{id}''_j)_\Delta, \dots, \text{se}(\text{id}''_{k-1})_\Delta, \text{se}(\text{id}'_k)_\Delta \equiv \\ \text{se}(\text{id}'_0)_\Delta, \dots, \text{se}(\text{id}'_i)_\Delta, \dots, \text{se}(\text{id}'_j)_\Delta, \dots, \text{se}(\text{id}'_k)_\Delta \end{aligned}$$

假设

$$\text{id}''_i \xrightarrow{\text{eo}} \text{id}'_{k-1} \wedge \text{id}'_{k-1} \xrightarrow{\text{eo}} \text{id}''_{i+1},$$

令

$$\begin{aligned} \text{id}'''_0, \dots, \text{id}'''_i, \dots, \text{id}'''_j, \dots, \text{id}'''_k &= \text{id}''_0, \dots, \text{id}''_i, \text{id}'_k, \text{id}''_{i+1}, \dots, \text{id}''_j, \dots, \text{id}''_{k-1}, \text{其中} \\ \text{id}'''_i \xrightarrow{\text{eo}} \Delta \text{id}'''_j &\Rightarrow i < j。 \end{aligned}$$

我们知道

$\text{se}(\text{id}'_k)$  与  $\{\text{id}''_{i+1}, \dots, \text{id}''_j, \dots, \text{id}''_{k-1}\}$  中的所有副作用可交换。

否则, 假设

$$\neg \text{commute}(\text{se}(\text{id}'_k), \text{se}(\text{id}^*)).$$

我们知道

$$(\iota, \text{id}^*, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'_k, \text{exe}),$$

其与

$$\text{id}'_k \xrightarrow{\text{eo}} \Delta \text{id}^*$$

矛盾。

因此，我们有

$$\begin{aligned}\text{se}(\text{id}_0'')_\Delta, \dots, \text{se}(\text{id}_i'')_\Delta, \dots, \text{se}(\text{id}_j'')_\Delta, \dots, \text{se}(\text{id}_k'')_\Delta &\equiv \\ \text{se}(\text{id}_0')_\Delta, \dots, \text{se}(\text{id}_i')_\Delta, \dots, \text{se}(\text{id}_j')_\Delta, \dots, \text{se}(\text{id}_k')_\Delta.\end{aligned}$$

证明结束。 □

## 附录 D 定理 4.2 的证明

**定义 D.1** (观察顺序) 请求  $\text{id}_A$  和请求  $\text{id}_B$  在轨迹  $\Delta$  下具有观察顺序 (用  $\text{id}_A \xrightarrow{\text{ob}}_{\Delta} \text{id}_B$  表示) 如果下列条件之一成立:

1.  $\text{id}_A \xrightarrow{\text{vo}}_{\Delta} \text{id}_B$ 。
2.  $\text{sync}(\text{id}_B, \text{id}_A)_{\Delta} \wedge \neg \text{id}_B \xrightarrow{\text{vo}}_{\Delta} \text{id}_A$ 。
3.  $\exists \text{id}_C, \text{id}_A \xrightarrow{\text{ob}}_{\Delta} \text{id}_C \wedge \text{id}_C \xrightarrow{\text{ob}}_{\Delta} \text{id}_B$ 。

**引理 D.1** 如果轨迹  $\Delta$  满足一致性, 那么不存在  $\text{id}$  使得  $\text{id} \xrightarrow{\text{ob}}_{\Delta}^* \text{id}$ 。

为了证明定理 4.2, 我们只需要证明引理 D.2 即可。

**引理 D.2** 如果  $\vdash \Sigma : I$ , 那么对于所有的  $\Delta \Omega$ , 如果  $\Sigma \vdash \Omega \vdash^{\Delta} \_$  并且  $\Omega \vDash I$ , 那么  $I \stackrel{\Delta}{\approx} (\Sigma, \Omega)$ 。

**证明** 为了证明  $I \stackrel{\Delta}{\approx} (\Sigma, \Omega)$ , 通过引理 D.1, 我们只需要证明

**引理 D.3** (归纳基础) 对于所有的  $\text{id}$ , 如果  $\text{id} \in \Delta.ids$  并且  $\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta} \text{id}'$ , 那么  $I \stackrel{\Delta}{\approx} (\Sigma, \Omega, \text{id})$ 。

**引理 D.4** (归纳假设) 对于所有的  $\text{id}$ , 如果  $\text{id} \in \Delta.ids$  并且对于所有的  $\text{id}'$ ,  $\text{id} \xrightarrow{\text{ob}}_{\Delta} \text{id}' \Rightarrow I \stackrel{\Delta}{\approx} (\Sigma, \Omega, \text{id}')$ , 那么  $I \stackrel{\Delta}{\approx} (\Sigma, \Omega, \text{id})$ 。

(1) 引理 D.3 的证明

通过定义  $\_ \approx \_$ , 为了证明

$$I \stackrel{\Delta}{\approx} (\Sigma, \Omega, \text{id}),$$

我们只需要证明

$$\forall \Delta', \Delta \approx \Delta' \Rightarrow I \stackrel{\Delta'}{\bowtie} (\Sigma, \Omega, \text{id}).$$

通过定义  $\_ \bowtie \_$ , 我们只需要证明

$$\forall \iota, I \stackrel{\Delta'}{\bowtie} (\Sigma, \Omega, \text{id}, \iota).$$

通过定义  $\_ \approx \_$ , 我们知道

$$\text{id} \in \Delta'.ids \text{ 并且 } \nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta'} \text{id}'.$$

因此, 我们知道

$$\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta} (\iota, \text{id}, \text{exe})$$

因此, 通过应用定理 D.6 即可证明。

$\text{before\_acpt}(\text{id}, \text{id}')_{\Delta} = \Delta'$	$\stackrel{\text{def}}{=} (\forall i' \neq i, \text{on}(i')_{\Delta'} = \text{on}(i')_{\Delta}) \wedge \text{on}(i)'_{\Delta} = \Delta_1 \text{++ } \Delta_2$ $+ + [(\iota, \text{id}, \text{exe})] + + [(\iota, \text{id}', \text{acpt})] + + \Delta_3$ $, \text{其中 } \text{on}(i)_{\Delta} = \Delta_1 + + [(\iota, \text{id}, \text{exe})] + + \Delta_2 + + [(\iota, \text{id}', \text{acpt})] + + \Delta_3,$ $\text{site\_of}(\text{id})_{\Delta} = i$
$\text{before\_exe}(\text{id}, \text{id}', i)_{\Delta} = \Delta'$	$\stackrel{\text{def}}{=} (\forall i' \neq i, \text{on}(i')_{\Delta'} = \text{on}(i')_{\Delta}) \wedge \text{on}(i)'_{\Delta} = \Delta_1 \text{++ } \Delta_2$ $+ + [(\iota, \text{id}, \text{exe})] + + [(\iota, \text{id}', \text{exe})] + + \Delta_3$ $, \text{其中 } \text{on}(i)_{\Delta} = \Delta_1 + + [(\iota, \text{id}, \text{exe})] + + \Delta_2 + + [(\iota, \text{id}', \text{exe})] + + \Delta_3$
$\text{after\_exe}(\text{id}, \text{id}', i)_{\Delta} = \Delta'$	$\stackrel{\text{def}}{=} (\forall i' \neq i, \text{on}(i')_{\Delta'} = \text{on}(i')_{\Delta}) \wedge \text{on}(i)'_{\Delta} = \Delta_1 \text{++ } \Delta_2$ $+ + [(\iota, \text{id}', \text{exe})] + + [(\iota, \text{id}, \text{exe})] + + \Delta_3$ $, \text{其中 } \text{on}(i)_{\Delta} = \Delta_1 + + [(\iota, \text{id}, \text{exe})] + + \Delta_2 + + [(\iota, \text{id}', \text{exe})] + + \Delta_3$
$\text{acpt\_nextto}(\text{id}, \text{id}')_{\Delta}$	$\stackrel{\text{def}}{=} (\iota, \text{id}, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}', \text{acpt}) \wedge$ $\nexists \text{id}'', (\iota, \text{id}, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'', \text{exe}) \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}', \text{acpt})$ $, \text{其中 } \text{site\_of}(\text{id})_{\Delta} = i$
$\text{exe\_nextto}(\text{id}, \text{id}', i)_{\Delta}$	$\stackrel{\text{def}}{=} (\iota, \text{id}, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}', \text{exe}) \wedge$ $\nexists \text{id}'', (\iota, \text{id}, \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'', \text{exe}) \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}', \text{exe})$
$(\Omega, i) \models B$	$\stackrel{\text{def}}{=} \text{store\_of}(i)_{\Omega} \in [B]$
$\text{prefix}(\Delta) = \Delta'$	$\stackrel{\text{def}}{=} \Delta = \Delta' \text{++ } \underline{\quad}$
$\text{head}(\Delta) = e$	$\stackrel{\text{def}}{=} \Delta = e :: \underline{\quad}$
$\text{exe\_his}(\iota, \text{id})_{\Delta}$	$\stackrel{\text{def}}{=} \{ \text{id}' \mid (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe}) \}$
$\text{acpt\_his}(\text{id})_{\Delta}$	$\stackrel{\text{def}}{=} \{ \text{id}' \mid (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{acpt}) \}, \text{其中 } \text{site\_of}(\text{id})_{\Delta} = i$
$\Delta \approx \Delta'$	$\stackrel{\text{def}}{=} \text{set}(\Delta) = \text{set}(\Delta') \wedge \forall \text{id} \text{id}', \text{id} \xrightarrow[\Delta]{\text{ob}} \text{id}' \Rightarrow \text{id}' \xrightarrow[\Delta]{\text{ob}} \text{id}$
$\text{HOLD}(\Sigma, \Omega, \Delta, i, I)$	$\stackrel{\text{def}}{=} \forall \Omega', \Sigma \vdash \Omega \succ^{\Delta} \Omega' \Rightarrow (\Omega', i) \models I$
$I \overset{\Delta}{\bowtie} (\Sigma, \Omega, \text{id}, i)$	$\stackrel{\text{def}}{=} \forall \Delta', \text{prefix}(\Delta) = \Delta' \wedge \text{head}(\Delta') = (\iota, \text{id}, \text{exe}) \wedge \text{HOLD}(\Sigma, \Omega, \Delta', i, I)$
$I \overset{\Delta}{\bowtie} (\Sigma, \Omega, \text{id})$	$\stackrel{\text{def}}{=} \forall i, I \overset{\Delta}{\bowtie} (\Sigma, \Omega, \text{id}, i)$
$I \overset{\Delta}{\approx} (\Sigma, \Omega, \text{id})$	$\stackrel{\text{def}}{=} \forall \Delta', \Delta \approx \Delta' \Rightarrow I \overset{\Delta'}{\bowtie} (\Sigma, \Omega, \text{id})$
$I \overset{\Delta}{\approx} (\Sigma, \Omega)$	$\stackrel{\text{def}}{=} \forall \text{id} \in \Delta.ids, I \overset{\Delta}{\approx} (\Sigma, \Omega, \text{id})$

图 D.1 证明不变量程序逻辑可靠性的一些辅助定义

(2) 引理 D.4 的证明

通过定义  $\_ \approx \_$ , 为了证明

$$I \overset{\Delta}{\approx} (\Sigma, \Omega, \text{id}),$$

我们只需要证明

$$\forall \Delta', \Delta \approx \Delta' \Rightarrow I \xrightarrow{\Delta'} (\Sigma, \Omega, \text{id})。$$

通过定义  $\_\bowtie\_\$ , 我们只需要证明

$$\forall \iota, I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}, \iota)。$$

我们进行分类讨论:

- $\nexists \text{id}', \text{id} \xrightarrow{\text{ob}} \Delta \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})$ 。

通过应用定理 D.6 可证。

- $\exists \text{id}', \text{id} \xrightarrow{\text{ob}} \Delta \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})$ 。

我们可以找到一个  $\text{id}'$  满足:

$$\nexists \text{id}'', \text{id}' \xrightarrow{\text{ob}} \Delta \text{id}'' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}'', \text{exe}) \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})。$$

否则, 我们知道

$$\text{id} \xrightarrow{\text{ob}} \Delta \text{id}''。$$

因此, 我们可以令  $\text{id}''$  为那个  $\text{id}'$ 。

由于我们有

$$\forall \text{id}', \text{id} \xrightarrow{\text{ob}} \Delta \text{id}' \Rightarrow I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}'),$$

我们知道

$$I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}')。$$

因此, 我们可以应用定理 D.7。

□

**引理 D.5** 对于所有的  $\Sigma \Delta \Omega \text{id} I \iota$ , 如果

1.  $\vdash \Sigma : I$
2.  $\text{exe\_his}(\iota, \text{id})_{\Delta} = \text{acpt\_his}(\text{id})_{\Delta}$

那么  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}, \iota)$ 。

**证明** 由收敛性定理 4.1 可证。□

**引理 D.6** 对于所有的  $\Sigma \Delta \Omega \text{id} \text{id}' I \iota$ , 如果

1.  $\vdash \Sigma : I$
2.  $\nexists \text{id}', \text{id} \xrightarrow{\text{ob}} \Delta \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})$

那么  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}, \iota)$ 。

**证明** 此引理可由  $\text{safe}(\text{op}(\text{id})_{\Delta})$  证明。因此, 我们只需要考虑情况

$$\neg \text{safe}(\text{op}(\text{id})_{\Delta})。$$

令  $h_{diff}(\Delta^*) \stackrel{\text{def}}{=} \text{exe\_his}(\iota, \text{id})_{\Delta^*} \cup \text{acpt\_his}(\text{id})_{\Delta^*} \setminus \text{exe\_his}(\iota, \text{id})_{\Delta^*} \cap \text{acpt\_his}(\text{id})_{\Delta^*}$ , 我们在  $h_{diff}$  集合的大小上进行归纳。

归纳基础:

$$\forall \Delta_0, |h_{diff}(\Delta_0)| = 0 \wedge (\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_0} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_0} (\iota, \text{id}, \text{exe})) \Rightarrow I \xrightarrow{\Delta_0} (\Sigma, \Omega, \text{id}, \iota)。$$

通过应用定理 D.5 可证。

归纳假设:

$$\begin{aligned} \text{如果 } \forall \Delta_{n-1}, |h_{diff}(\Delta_{n-1})| = n - 1 \wedge (\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_{n-1}} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_{n-1}} (\iota, \text{id}, \text{exe})) \Rightarrow I \xrightarrow{\Delta_{n-1}} (\Sigma, \Omega, \text{id}, \iota), \text{ 那么 } \forall \Delta_n, |h_{diff}(\Delta_n)| = n \wedge (\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}' \\ \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_n} (\iota, \text{id}, \text{exe})) \Rightarrow I \xrightarrow{\Delta_n} (\Sigma, \Omega, \text{id}, \iota)。 \end{aligned}$$

为了证明  $I \xrightarrow{\Delta_n} (\Sigma, \Omega, \text{id}, \iota)$ , 我们进行分情况讨论

- $\exists \text{id}'', \text{id}'' \in \text{acpt\_his}(\text{id})_{\Delta_n} \wedge \text{id}'' \notin \text{exe\_his}(\text{id})_{\Delta_n} \wedge \nexists \text{id}''' \in h_{diff}(\Delta_n)$  使得  $(\iota', \text{id}'', \text{exe}) \xrightarrow[\iota']{\text{so}}_{\Delta_n} (\iota', \text{id}''', \text{exe}) \wedge (\iota', \text{id}''', \text{exe}) \xrightarrow[\iota']{\text{so}}_{\Delta_n} (\iota', \text{id}, \text{acpt})$ , 其中  $\iota' = \text{site\_of}(\text{id})_{\Delta_n}$ 。

因此

$$\text{id}'' \xrightarrow{\text{vo}}_{\Delta_n} \text{id} \wedge \neg(\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_n} (\iota, \text{id}, \text{exe})$$

因此, 我们有

$$\neg \text{dep}(\text{op}(\text{id}'')_{\Delta_n}, \text{op}(\text{id})_{\Delta_n})$$

由

$$\vdash \Sigma : I,$$

可得

$$\text{eo\_independent}(\text{op}(\text{id}'')_{\Delta_n}, \text{op}(\text{id})_{\Delta_n})$$

令  $\Delta'_n$  和  $\Delta''_n$  满足

$$\begin{aligned} \Delta'_n &= \text{before\_exe}(\text{id}, \text{id}')_{\Delta_n} \wedge \\ \Delta''_n &= \Delta_n \cup (\iota, \text{id}, \text{exe}) \wedge \text{acpt\_nextto}(\text{id}'', \text{id})_{\Delta''_n}。 \end{aligned}$$

可得

$$\text{exe\_nextto}(\text{id}'', \text{id})_{\Delta''_n}。$$

由于我们知道  $\text{id}''$  与所有满足在

$$\{\text{id}''' \mid (\iota', \text{id}'', \text{exe}) \xrightarrow[\iota']{\text{so}}_{\Delta_n} (\iota', \text{id}''', \text{exe}) \wedge (\iota', \text{id}''', \text{exe}) \xrightarrow[\iota']{\text{so}}_{\Delta_n} (\iota', \text{id}, \text{acpt})\},$$

中的  $\text{id}'''$  可交换。因此

$$I \xrightarrow{\Delta'_n} (\Sigma, \Omega, \text{id}, \iota) \Rightarrow I \xrightarrow{\Delta''_n} (\Sigma, \Omega, \text{id}, \iota)。$$

我们知道

$$|h_{diff}(\Delta''_n)| = n - 1 \wedge (\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_n} (\iota, \text{id}, \text{exe}))。$$

因此, 我们有

$$I \xrightarrow{\Delta''_n} (\Sigma, \Omega, \text{id}, \iota)。$$

通过引理 D.8, 可得

$$I \xrightarrow{\Delta'_n} (\Sigma, \Omega, \text{id}, \iota)。$$

因此, 我们知道

$$I \xrightarrow{\Delta_n} (\Sigma, \Omega, \text{id}, \iota)。$$

- $\exists \text{id}'', \text{id}'' \in \text{exe\_his}(\text{id})_{\Delta_n} \wedge \text{id}'' \notin \text{acpt\_his}(\text{id})_{\Delta_n} \wedge \nexists \text{id}''' \in h_{diff}(\Delta_n)$   
使得  $(\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}''', \text{exe}) \wedge (\iota, \text{id}''', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})$ 。

因此

$$\neg \text{id}'' \xrightarrow{\text{vo}}_{\Delta_n} \text{id} \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})$$

因此, 我们有

$$\neg \text{sync}(\text{op}(\text{id}'')_{\Delta_n}, \text{op}(\text{id})_{\Delta_n})$$

否则, 通过

$$\text{sync}(\text{op}(\text{id}'')_{\Delta_n}, \text{op}(\text{id})_{\Delta_n})$$

和

$$\neg \text{id}'' \xrightarrow{\text{vo}}_{\Delta_n} \text{id},$$

我们有

$$\text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}'',$$

这与

$$\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}} (\iota, \text{id}, \text{exe})。$$

矛盾。由于  $\vdash \Sigma : I$ , 可得

$$\text{vo\_independent}(\text{op}(\text{id}'')_{\Delta_n}, \text{op}(\text{id})_{\Delta_n})$$

令  $\Delta'_n$  和  $\Delta''_n$  满足

$$\begin{aligned} \Delta'_n &= \text{before\_acpt}(\text{id}, \text{id}')_{\Delta_n} \wedge \\ \Delta''_n &= \Delta_n \cup (\iota, \text{id}, \text{exe}) \wedge \text{exe\_nextto}(\text{id}'', \text{id})_{\Delta''_n}。 \end{aligned}$$

可得

$$\text{acpt\_nextto}(\text{id}'', \text{id})_{\Delta''_n}。$$

由于  $\text{id}''$  与所有

$$\begin{aligned} \{\text{id}''' \mid (\iota', \text{id}'', \text{exe}) \xrightarrow[\iota']{\text{so}} (\iota', \text{id}''', \text{exe}) \wedge (\iota', \text{id}''', \text{exe}) \xrightarrow[\iota']{\text{so}} \\ (\iota', \text{id}, \text{acpt})\}, \end{aligned}$$

中的  $\text{id}'''$  可交换。因此

$$I \xrightarrow{\Delta'_n} (\Sigma, \Omega, \text{id}, \iota) \Rightarrow I \xrightarrow{\Delta_n} (\Sigma, \Omega, \text{id}, \iota)。$$

由于

$$\neg \text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}'',$$

我们知道

$$|h_{diff}(\Delta''_n)| = n - 1 \wedge (\nexists \text{id}', \text{id} \xrightarrow{\text{ob}}_{\Delta_n} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta_n} (\iota, \text{id}, \text{exe})).$$

因此，我们有

$$I \xrightarrow{\Delta''_n} (\Sigma, \Omega, \text{id}, \iota).$$

通过引理 D.9, 可得

$$I \xrightarrow{\Delta'_n} (\Sigma, \Omega, \text{id}, \iota).$$

因此，我们知道

$$I \xrightarrow{\Delta_n} (\Sigma, \Omega, \text{id}, \iota).$$

□

**引理 D.7** 对于所有的  $\Sigma \Delta \Omega \text{id} \text{id}' I \iota$ , 如果

1.  $\vdash \Sigma : I$
2.  $\text{id} \xrightarrow{\text{ob}}_{\Delta} \text{id}' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta} (\iota, \text{id}, \text{exe})$
3.  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}')$
4.  $\nexists \text{id}'' \in T, \text{id}' \xrightarrow{\text{ob}}_{\Delta} \text{id}'' \wedge (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta} (\iota, \text{id}'', \text{exe}) \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota'']{\text{so}}_{\Delta} (\iota, \text{id}, \text{exe})$

那么  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}, \iota)$ 。

**证明** 我们知道  $\text{id}'$ 、 $\text{id}$  和所有

$$T \stackrel{\text{def}}{=} \{ \text{id}'' \mid (\iota, \text{id}', \text{exe}) \xrightarrow[\iota]{\text{so}}_{\Delta} (\iota, \text{id}'', \text{exe}) \wedge (\iota, \text{id}'', \text{exe}) \xrightarrow[\iota'']{\text{so}}_{\Delta} (\iota, \text{id}, \text{exe}) \},$$

中的  $\text{id}''$  可交换。否则，通过

$$\vdash \Sigma : I,$$

我们有

$$\text{id}' \xrightarrow{\text{ob}}_{\Delta} \text{id}'',$$

这个与 (4) 矛盾。

令

$$\Delta' = \text{after\_exe}(\text{id}', \text{id}, \iota)_{\Delta},$$

我们知道

$$I \xrightarrow{\Delta'_n} (\Sigma, \Omega, \text{id}', \iota) \Rightarrow I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}, \iota).$$

由于

$$\forall \text{id}'' \in T, \text{id}'' \xrightarrow{\text{ob}}_{\Delta} \text{id}' \Rightarrow \text{id}'' \xrightarrow{\text{ob}}_{\Delta'} \text{id}',$$

可得

$$\Delta \approx \Delta'.$$

通过 (3) 可知

$$I \xrightarrow{\Delta'} (\Sigma, \Omega, \text{id}', \iota).$$

证明结束。 □

**引理 D.8** 对于所有的  $\Sigma \Delta \Delta' \Omega \text{id} \text{id}' I \iota$ , 如果

1.  $\text{eo\_independent}(\text{op}(\text{id})_\Delta, \text{op}(\text{id}')_\Delta)_I$
2.  $\text{exe\_nextto}(\text{id}, \text{id}', \iota)_\Delta$
3.  $\Delta' = \Delta \setminus (\iota, \text{id}, \text{exe})$

那么  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}', \iota) \Rightarrow I \xrightarrow{\Delta'} (\Sigma, \Omega, \text{id}', \iota)$ 。

**证明** 通过定义 4.6 可证。 □

**引理 D.9** 对于所有的  $\Sigma \Delta \Omega \text{id} \text{id}' I \iota$ , 如果

1.  $\text{vo\_independent}(\text{op}(\text{id})_\Delta, \text{op}(\text{id}')_\Delta)_I$
2.  $\text{acpt\_nextto}(\text{id}, \text{id}')_\Delta$
3.  $\Delta' = \Delta \setminus (\text{site\_of}(\text{id}')_\Delta, \text{id}, \text{exe})$

那么  $I \xrightarrow{\Delta} (\Sigma, \Omega, \text{id}', \iota) \Rightarrow I \xrightarrow{\Delta'} (\Sigma, \Omega, \text{id}', \iota)$ 。

**证明** 通过定义 4.7 可证。 □

## 附录 E 系统模型

MODULE Protocol\_in\_PlusCal

```

1 variables
2   network = empty_network;
3   library = init_library;
4
5 fair process ( sid ∈ SERVERS )
6   variables pool = empty_pool; counter = 0; cursor = init_cursor; calls = 0;
7 {
8   while ( true ) {
9     either {
10       call client_step();
11     }
12     or {
13       call server_step();
14     };
15   } ;
16 } ;
17
18 procedure client_step( ) {
19   if ( counter2tid(calls, self) ≤ MAX_OP_INS ) {
20     calls := calls + 1;
21     call newCall(CHOOSE x ∈ OPS : true);
22   } ;
23   return ;
24 }
25
26 procedure server_step( ) {
27   either {
28     call server_op_step();
29   }
30   or {

```

```

31   call server_msg_step();
32 }
33 return;
34 }
35
36 procedure server_msg_step( )
37   variables msg;
38 {
39   if ( Cardinality(network[self]) > 0 ) {
40     msg := CHOOSE x ∈ network[self] : true ;
41     if ( msg.type = ASK_SYNC ) {
42       call replySyncTids(msg.t_id, msg.site_id, msg.objs);
43     } else if ( msg.type = RTN_SYNC ) {
44       call processRtnSync(msg.site_id, msg.t_id, msg.tids);
45     } else if ( msg.type = BCT_DO ) {
46       call processDo(msg.site_id, msg.t_id, msg.op, msg.tids);
47     } else if ( msg.type = BCT_NOT_DO ) {
48       call processNotDo(msg.site_id, msg.t_id);
49     } else if ( msg.type = DO_RTN ) {
50       call processDoRtn(msg.site_id, msg.t_id);
51     } else if ( msg.type = DEL_OP ) {
52       call processDelete(msg.t_id);
53     };
54     network[self] := network[self] \ {msg};
55   };
56   return;
57 }
58
59 procedure server_op_step( )
60   variables tid; b;
61 {
62   if ( Cardinality({x ∈ DOMAIN pool : op_ins_exists(pool[x])}) > 0 ) {
63     tid := CHOOSE x ∈ DOMAIN pool : op_ins_exists(pool[x]);
64     if ( pool[tid].status = B_SYNC ) {
65       if ( Cardinality(pool[tid].op.sync_set) > 0 ) {

```

```

66     call askForSyncTids(tid);
67     pool[tid].status := I_SYNC;
68 } else {
69     pool[tid].status := B_PROC;
70 }
71 } else if ( pool[tid].status = I_SYNC ) {
72     if ( Cardinality(pool[tid].msg_wait) = 0 ) {
73         pool[tid].status := A_SYNC;
74     }
75 } else if ( pool[tid].status = A_SYNC ) {
76     localsReady(tid, b);
77     if ( b = true ) {
78         pool[tid].status := B_PROC;
79     }
80 } else if ( pool[tid].status = B_PROC ) {
81     b := RandomElement(BOOLEAN);
82     if ( b = false ) {
83         pool[tid].status := DONE;
84         call allSitesCancel(tid);
85     } else {
86         if ( Cardinality(pool[tid].op.dep_set) > 0 ) {
87             call calDepTids(tid);
88             pool[tid].status := A_PROC;
89         } else {
90             pool[tid].status := B_EXE;
91         }
92         call allSitesExecute(tid);
93     }
94 } else if ( pool[tid].status = A_PROC ) {
95     localsReady(tid, b);
96     if ( b = true ) {
97         pool[tid].status := B_EXE;
98     }
99 } else if ( pool[tid].status = B_EXE ) {
100    pool[tid].status := A_EXE;

```

```

101    if ( !(pool[tid].site_id = self) ) {
102        call replyDo(tid);
103    }
104 } else if ( pool[tid].status = A_EXE ) {
105     if ( pool[tid].site_id = self ∧ Cardinality(pool[tid].msg_wait) = 0 ) {
106         pool[tid].status := DONE;
107         call broadcastDelete(tid);
108     }
109 } else if ( pool[tid].status = DONE ) {
110     if ( tid2counter(tid) = cursor[pool[tid].site_id] + 1 ) {
111         cursor[pool[tid].site_id] := cursor[pool[tid].site_id] + 1 ;
112         pool[tid] := empty_op_ins;
113     }
114 } else {
115     skip;
116 }
117 }
118 return;
119 }

120

121 define {
122     empty_network ≡ [x ∈ SERVERS ↦ {}]
123     init_cursor ≡ [x ∈ SERVERS ↦ -1]
124     empty_op ≡ [sync_set ↦ {}, dep_set ↦ {}, write_set ↦ {}]
125     empty_op_ins ≡ [site_id ↦ -1,
126     op ↦ empty_op,
127     status ↦ NOT_EXIST,
128     msg_wait ↦ {},
129     tid_wait ↦ {}]
130     empty_pool ≡ [x ∈ OP_INSTANCES ↦ empty_op_ins]
131     init_obj_set ≡ CHOOSE x ∈ SUBSET OBJECTS : Cardinality(x) > 0
132     init_op ≡ [sync_set ↦ RandomElement(SUBSET OBJECTS),
133     dep_set ↦ RandomElement(SUBSET OBJECTS),
134     write_set ↦ CHOOSE x ∈ SUBSET OBJECTS : Cardinality(x) > 0]
135     init_library ≡ [x ∈ OPS ↦ init_op]

```

```

136 op_exists(op)  $\triangleq$  op  $\neq$  empty_op
137 op_ins_exists(op_ins)  $\triangleq$  op_ins.status  $\neq$  NOT_EXIST
138 counter2tid(counter, self)  $\triangleq$  counter * MAX_SERVER + self
139 tid2counter(tid)  $\triangleq$  (tid - 1)  $\div$  MAX_SERVER
140 intersected(s1, s2)  $\triangleq$  Cardinality(s1  $\cap$  s2)  $>$  0
141 }
142
143 macro assignUUID( tid ) {
144     tid := counter * MAX_SERVER + self ;
145     counter := counter + 1 ;
146 }
147
148 macro broadcastMessage( from, msg ) {
149     network := [x  $\in$  DOMAIN network  $\mapsto$ 
150                 if x = from then network[x] else network[x]  $\cup$  {msg}] ;
151 }
152
153 macro replyMessage( to, msg ) {
154     network := [x  $\in$  DOMAIN network  $\mapsto$ 
155                 if x = to then network[x]  $\cup$  {msg} else network[x]] ;
156 }
157
158 macro locallsReady( tid, b ) {
159     b := false  $\notin$  {tid2counter(otherTid)  $\leqslant$  cursor[pool[tid].site_id]  $\vee$ 
160             (op_ins_exists(pool[otherTid])  $\wedge$ 
161              (pool[otherTid].status = A_EXE  $\vee$  pool[otherTid].status = DONE))
162             : otherTid  $\in$  pool[tid].tid_wait} ;
163 }
164
165 procedure processRtnSync( sid, tid, tids ) {
166     pool[tid].tid_wait := pool[tid].tid_wait  $\cup$  tids ;
167     pool[tid].msg_wait := pool[tid].msg_wait \ {sid} ;
168     return ;
169 }
170

```

```

171 procedure replySyncTids( tid, otherSid, objs ) {
172     if ( self < otherSid ) {
173         replyMessage(otherSid,
174             [type  $\mapsto$  RTN_SYNC,
175              t_id  $\mapsto$  tid,
176              site_id  $\mapsto$  self,
177              op  $\mapsto$  empty_op,
178              objs  $\mapsto$  {} ,
179              tids  $\mapsto$  {otherTid  $\in$  DOMAIN pool :
180                  op_ins_exists(pool[otherTid])  $\wedge$ 
181                  pool[otherTid].site_id = self  $\wedge$ 
182                  intersected(pool[otherTid].op.write_set, objs)}]);
183     } else if ( self > otherSid ) {
184         replyMessage(otherSid,
185             [type  $\mapsto$  RTN_SYNC,
186              t_id  $\mapsto$  tid,
187              site_id  $\mapsto$  self,
188              op  $\mapsto$  empty_op,
189              objs  $\mapsto$  {} ,
190              tids  $\mapsto$  {otherTid  $\in$  DOMAIN pool :
191                  op_ins_exists(pool[otherTid])  $\wedge$ 
192                  pool[otherTid].site_id = self  $\wedge$ 
193                  (pool[otherTid].status = A_PROC  $\vee$ 
194                  pool[otherTid].status = B_EXE  $\vee$ 
195                  pool[otherTid].status = A_EXE  $\vee$ 
196                  pool[otherTid].status = DONE)  $\wedge$ 
197                  intersected(pool[otherTid].op.write_set, objs)}]);
198     } else {
199         call processRtnSync(self, tid, {otherTid  $\in$  DOMAIN pool :
200             op_ins_exists(pool[otherTid])  $\wedge$ 
201             pool[otherTid].site_id = self  $\wedge$ 
202             tid2counter(otherTid) < tid2counter(tid)  $\wedge$ 
203             intersected(pool[otherTid].op.write_set, objs)});
204     } ;
205     return ;

```

```

206 }
207
208 procedure askForSyncTids( tid ) {
209   pool[tid].msg_wait := SERVERS ;
210   call replySyncTids(tid, self, pool[tid].op.sync_set) ;
211   broadcastMessage(self,
212     [type ↪ ASK_SYNC,
213      t_id ↪ tid,
214      site_id ↪ self,
215      op ↪ empty_op,
216      objs ↪ pool[tid].op.sync_set,
217      tids ↪ {}]) ;
218   return ;
219 }
220
221 procedure allSitesCancel( tid ) {
222   broadcastMessage(self,
223     [type ↪ BCT_NOT_DO,
224      t_id ↪ tid,
225      site_id ↪ self,
226      op ↪ empty_op,
227      objs ↪ {},
228      tids ↪ {}]) ;
229   return ;
230 }
231
232 procedure calDepTids( tid ) {
233   pool[tid].tid_wait := {otherTid ∈ DOMAIN pool :
234     op_ins_exists(pool[otherTid]) ∧
235     (pool[otherTid].status = A_EXE ∨
236     pool[otherTid].status = DONE) ∧
237     op_exists(pool[otherTid]) ∧
238     intersected(pool[otherTid].op.write_set, pool[tid].op.dep_set)} ;
239   return ;
240 }
```

```

241
242 procedure allSitesExecute( tid ) {
243     pool[tid].msg_wait := SERVERS \ {self} ;
244     broadcastMessage(self,
245                     [type ↪ BCT_DO,
246                      t_id ↪ tid,
247                      site_id ↪ self,
248                      op ↪ pool[tid].op,
249                      objs ↪ {} ,
250                      tids ↪ pool[tid].tid_wait]) ;
251     return ;
252 }
253
254 procedure processDo( otherSid, tid, opName, tids ) {
255     if ( Cardinality(tids) > 0 ) {
256         pool[tid] := [site_id ↪ otherSid,
257                      op ↪ opName,
258                      status ↪ A_PROC,
259                      msg_wait ↪ {} ,
260                      tid_wait ↪ tids] ;
261     } else {
262         pool[tid] := [site_id ↪ otherSid,
263                      op ↪ opName,
264                      status ↪ B_EXE,
265                      msg_wait ↪ {} ,
266                      id_wait ↪ {} ] ;
267     } ;
268     return ;
269 }
270
271 procedure processNotDo( otherSid, tid ) {
272     pool[tid] := [site_id ↪ otherSid,
273                   op ↪ empty_op,
274                   status ↪ DONE,
275                   msg_wait ↪ {} ],

```

```
276         tid_wait ↦ {};
277     return ;
278 }
279
280 procedure processDoRtn( otherSid, tid ) {
281     pool[tid].msg_wait := pool[tid].msg_wait \ {otherSid} ;
282     return ;
283 }
284
285 procedure processDelete( tid ) {
286     pool[tid].status := DONE ;
287     return ;
288 }
289
290 procedure replyDo( tid ) {
291     broadcastMessage(self,
292                     [type ↦ DO_RTN,
293                      t_id ↦ tid,
294                      site_id ↦ self,
295                      op ↦ empty_op,
296                      objs ↦ {},
297                      tids ↦ {}]);
298     return ;
299 }
300
301 procedure broadcastDelete( tid ) {
302     broadcastMessage(self,
303                     [type ↦ DEL_OP,
304                      t_id ↦ tid,
305                      site_id ↦ self,
306                      op ↦ empty_op,
307                      objs ↦ {},
308                      tids ↦ {}]);
309     return ;
310 }
```

```
311
312 procedure newCall( op_id )
313   variables tid ;
314 {
315   assignUUID(tid) ;
316   pool[tid] := [site_id ↠ self,
317                 op ↠ library[op_id],
318                 status ↠ B_SYNC,
319                 msg_wait ↠ {} ,
320                 tid_wait ↠ {} ] ;
321   return ;
322 }
323
324 IDLE == network = empty_network ∧ ∀x ∈ SERVERS : pool[x] = empty_pool
325 LIVENESS ==> IDLE
```

## 致 谢

感谢母校对我的培养。在这七年的时光里，它给了我一个广阔的学习平台和安静的学习环境，让我不断地学习新知，充实自己。感谢冯新宇老师对我的教导。他渊博的专业知识，严肃的科学态度，严谨的治学精神，诲人不倦的高尚师德都对我产生了深远的影响。是他教导我不断突破自我，敢于向难题挑战，使我树立了远大的学术目标并为之不断努力奋斗。感谢李诚老师在研究生阶段对我的指导。他在我的学习与研究中给予了悉心的指导，同时还在精神和生活上给了我以无微不至的关怀。他严谨的治学态度和渊博的学识，朴实无华及平易近人的人格魅力对我影响深远，是我一生都值得学习的榜样。感谢我的女朋友杨行同学在这几年时间里对我的细心关怀和默默支持。感谢实验室的小伙伴们，感谢一路上给予我鼓励和支持的家人、朋友和同学，感谢那些曾经给予我关心和帮助的人，祝大家前程似锦。

## 在读期间发表的学术论文与取得的研究成果

### 已发表论文

1. **Jiawei Wang**, Ming Fu, Lei Qiao, and Xinyu Feng. Formalizing SPARCv8 Instruction Set Architecture in Coq. *Dependable Software Engineering. Theories, Tools, and Applications: Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*. Vol. 10606. Springer, 2017.
2. **Jiawei Wang**, Ming Fu, Lei Qiao, and Xinyu Feng. Formalizing SPARCv8 instruction set architecture in Coq. *Science of Computer Programming* 187 (2020): 102371. (CCF B 类杂志)

### 待发表论文

1. **Jiawei Wang**, Cheng Li, Jingze Huo, Kai Ma, Feng Yan, Xinyu Feng, and Yinlong Xu. AutoGR: Automated Geo-Replication with Fast System Performance and Preserved Application Semantics.