本地优先软件的行为类型

ROLAND KUHN，Actyx AG，GermayHERNÁN MELGRATTI，布宜诺斯艾利斯大学与 Conicet，阿根廷EMILIO TUOSTO，格兰萨索科学研究所，意大利

点对点系统是分布式计算中最具韧性的形式，但设计用于协调这些系统的健壮协议并非易事。这使得难以规约和推理此类系统的全局行为。本文提出群协议，以全局视角规约此类系统。群协议会投影到机器上，也就是对等节点的局部规约。

我们的研究受到行为类型的启发，但存在一个关键区别：对等节点通过事件通知机制进行通信，而非通过点对点消息传递。我们的目标是遵循本地优先软件 的原则，使网络设备在保留完全自主性的同时，能够协作完成共同任务：每台参与的设备始终能够在局部取得进展，不受其他设备或网络连接不可用的影响。这种无协调的方式会导致计算过程中可能出现不一致性。我们的主要研究成果表明，在满足群协议的适当良构条件时，一致性最终能够得到恢复，并且符合规范的机器的局部可观察行为最终将与全局规约相匹配。

我们提出的模型在现有工业平台的基础上进行了扩展，并为工具支持奠定了基础（本文对此进行了概述，配套工件论文则进行了完整描述），因此我们认为，这项工作是迈向局部优先和点对点软件系统推理的可行一步。

附加关键词和短语：分布式协调、局部优先软件、行为类型、发布–订阅、异步通信

arXiv:2305.04848v1 [cs.DC] 8 May 2023

1 引言

像点对点网络这样的完全去中心化系统在设计和分析方面出了名地困难。其中一个主要挑战是协调各个组件，使组合后的系统呈现出预期的行为。由于此类系统中的所有决策都是基于可用的局部知识在本地作出的，因此主要问题在于明确规定信息传递给谁、何时传递，以及如何对其进行解释，即协议设计。我们通过以下场景来说明这一点。

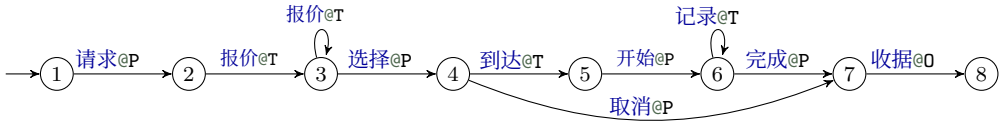
示例 *1.1*（我们的贯穿示例）。出租车车队使用点对点网络来组织城镇内的行程。主要目标是为请求行程的乘客提供一些愿意执行所需运输的出租车报价；乘客可以根据价格和预计到达时间选择一份报价——随后追踪接客、行程和到达情况——或者取消行程。行程结束时，会计办公室会为此次行程或取消提供收据。

不同对等节点（按角色分类）之间预期交互的结构，可以用下面的示意图非正式地表示，其中涉及乘客角色 P、出租车角色 T，以及

本研究部分由欧盟 H2020 RISE 计划通过玛丽·斯克沃多夫斯卡-居里资助协议（编号 778233）资助。部分工作由 MIUR 项目 PRIN 2017FTXR7S*IT MATTERS*（可信智能系统的方法与工具）资助。machine-runner 和 machine-check 部分由欧盟（TaRDIS，101093006）资助。作者还感谢匿名审稿人提出的有用且富有见地的意见，并感谢 Daniela Marottoli 在本项目初期提供的帮助。作者地址：Roland Kuhn，roland@actyx.io，Actyx AG，Germay；Hernán Melgratti，hmelgra@dc.uba.ar，布宜诺斯艾利斯大学与 Conicet，阿根廷；Emilio Tuosto，格兰萨索科学研究所，意大利。

2023年. 2475-1421/2023/1-ART1
$15.00https://doi.org/

作为 O 的会计办公室角色。



当乘客需要行程时（状态 1 在上图中），他们通过执行 请求 来开启拍卖。随后，任何有容量的出租车都可以提交 报价。在此描述中，我们不对扮演每种角色的实例数量做任何假设；事实上，我们预期会有许多出租车扮演角色 T，因此也会有许多报价。乘客在收到至少一份报价后，使用 选择来选出获胜者，从而结束拍卖。

第二阶段从状态 4 开始，此时会出现竞争：出租车要么先调用 到达，行程随之开始；要么乘客先失去耐心，并使用 取消 命令退出。无论哪种情况，办公室都会创建一张收据，但我们必须解决行程是否发生的争议。 ⋄

针对此类问题的经典解决方案是使用中央数据库，为每位参与者提供其有权查看的相应数据的最新视图。这种解决方案能够避免冲突、维护不变量，并通过确保只有一个事实来源来引导整个流程——也就是说，到达 或 取消 命令中总有一个会先发生，而另一个则会被拒绝。众所周知（参见 CAP [17] 和 FLP [14] 结果），如果系统模型包含网络分区，这种解决方案就会遭遇可用性问题，因为数据库要么是本地化的，可能无法访问；要么是分布式的，可能需要拒绝请求以维持一致性。随着无冲突复制数据类型（*CRDT*） [33] 的出现，另一种解决方案逐渐成为可能：CRDT 不再通过协调来避免冲突，而是提供一种在构造上就能确保无冲突的数据模型。CRDT 通过要求数据结构构成一个连接半格来实现这一点，即存在一个合并函数，能够接收两个以不同方式演化的状态，并计算出一个新状态，该状态代表对任一输入执行的所有操作的总和——至少在可能的范围内是如此。将这一方法应用于示例1.1 时，通常会通过在状态 4 中系统性地优先选择其中一方来实现，例如“取消操作始终优先”（参见添加优先集合 CRDT）。这说明，CRDT 并不适合捕获并公平解决我们示例中的此类冲突。尽管如此，对无协调系统日益增长的研究关注促使人们提出了本地优先软件 [26] 这一概念，在这种软件中，分布式系统的所有参与者都能对自己的数据保持完全的自主性和控制权。除了关注自主权和所有权之外，本地优先原则还可用于构建完全可用且具有最大弹性的软件 [27]，使每位参与者都能够独立做出在全局范围内有效的决策。

本文提出一种处理本地优先软件的新方法：拥抱冲突、识别冲突，并最终调和冲突，以达成最终共识 [39]。我们的计算模型基于 Actyx [1]，开发的中间件构建，该中间件为事件日志提供可靠、持久的发布–订阅机制，并为所有事件提供无需协调的全序。在我们的模型中，分布式系统由一组参与者构成，称为机器；它们可能表现出不一致的行为，并通过广播和响应事件进行交互。事件由命令的执行在本地生成，添加到本地日志 中，然后传播给其他参与者，并合并到接收方的本地日志中。我们假设事件以异步方式传播，并且不存在传统的协调机制（例如共识或中心节点）：参与的机器完全依据在系统中传播的事件彼此协作。我们将这类系统称为群体。

群中实现示例场景的每台机器1.1 都扮演一个角色，即订阅一组指定的事件类型，并应用分配给它的逻辑来解释这些事件。根据它的

局部状态（由其本地日志计算得出），任何机器都可以决定执行一条命令。例如，扮演角色 P 的机器可以执行 请求，从而生成包含请求详细信息的新事件。这些事件会被追加到该机器的本地日志中，然后异步传播给订阅了此类事件的其他机器，例如对应出租车的机器。收到此类事件后，出租车会更新其局部状态，并决定是否为该行程提供报价。在这种情况下，它会执行报价，生成描述该报价的事件，将其追加到本地日志中，并传播到系统的其余部分。示例 1.1 中描述的交互可以通过这种方式继续进行，直至完成。

需要指出的是，我们的计算模型并不排斥执行冲突命令。事实上，乘客可能会取消行程，而出租车可能同时确认其到达（示例1.1中的状态 4）。在这种情况下，两台机器都会生成各自的事件，并将其传播到系统中。收到此类事件时，每台机器都负责检测冲突并妥善解决。解决方式是利用事件之间的全序——将其解释为（逻辑）时间的体现——来消除冲突：选择之后最早发出的事件决定采取哪个分支（对应于未选分支的事件会被忽略）。请注意，我们并不假设知道事件日志何时完整，也就是说，机器无法检测某个事件是否为选择之后可能出现的最早事件。一旦某个选择及其之前的事件都在群中的所有机器上完成复制，所有机器便会对采取哪个分支达成一致；我们假设事件最终会对所有机器可用。然而，在此类事件尚未到达所有相关机器之前，计算得出的本地状态可能会暂时出现分歧。

事件仅由订阅者处理这一事实，使选择的解决过程变得微妙。假设会计办公室错误地订阅了取消事件，却没有订阅到达事件。在存在冲突选择的情况下，即使其他所有角色都明白行程已经发生，会计办公室也可能错误地得出行程已取消的结论。我们依靠一种类型纪律来排除选择解决过程中的此类不一致性。我们采用自顶向下的方法，使用群协议，即类似于示例1.1中图示的抽象——与全局类型 [20, 21]类似，它们从全局视角形式化描述预期协议。投影操作可以自动生成每个角色的局部规范，并将其形式化定义为机器（参见第2.3节）。我们的类型纪律确立了充分条件——群协议的良构性——以保证类型良构系统在信息充分传播给参与者后，能够一致地解决冲突选择。这一点，以及群协议完全抽象掉扮演某个角色的实例数量这一事实，构成了我们方法的显著特点。

论文主要贡献与结构概述 我们针对从真实中间件提炼出的形式化操作模型，开发了一套适用于本地优先软件的行为类型纪律。我们的理论也已在一个工具包中实现，该工具包能够对群体进行类型检查并支持其模拟。具体而言：

(1) 我们提出了一种基于复制事件日志以驱动本地状态机的分布式计算操作模型（第 2 节）。所提出的模型不假设稳定性，而是结合推测计算与一种类似于 时间扭曲机器 [23]的“回退”机制：通过沿正确路径回溯并重新执行来解决冲突。(2) 我们定义了一种新颖的行为类型方法（第4节），其中群协议根据特定角色的参与者所执行操作向异构群注入的信息来进行规定。群协议具有轻量级语法和简单的操作语义；它们无死锁且通信安全（第 7.4 节），同时表达能力足以对复杂协议进行建模。

(3) 我们为群协议定义了良构条件（定义 6.7），并定义投影操作以推导局部机器规约（第 5节）。这些确保了局部观察与全局规约行为之间的最终一致性（定理 7.14 和推论 7.15），鉴于不存在任何基础设施协调、不同角色之间的事件订阅并不一致，以及能够使用任意正数的副本来实现角色，这并非易事。(4) 我们将该方法应用于 TypeScript 语言和 Actyx 中间件，形成运行时库和用于检查协议良构性与符合性的工具，以及一个随机模拟工具，用于探索我们模型所允许的可能执行。

机器和日志传输的语义见第 2节，随后第 3节概述了配套工具。我们在第 4 节介绍群协议，随后在第 5节介绍它们在机器上的局部投影。接着，我们在第 6 节提出良构条件，并在第 7 节利用这些条件定义最终正确性的概念。第 8 节讨论相关工作，而第 9 节给出总结性说明。补充材料和证明收录于单独的附录中。

2 非对称复制状态机

我们的模型依赖于三个要素：机器、事件发出与消费，以及日志传输。机器的行为由有限状态自动机捕获，详见第 2.1 节。在第 2.2 节中，我们展示机器如何提供 命令，这些命令在执行时会发出事件；同时还介绍机器如何消费存储在其 本地日志中的（类型化）事件。此类事件会立即存储在发出机器的本地日志中，随后按照第 2.3 节所述，以异步方式传输到其他机器。

2.1 从 TypeScript 到自动机

我们通过提供一个用于编写端点代码的新库，将 Actyx 中间件实现的计算模型形式化（并加以阐释）。与 Actyx SDK 一样，我们使用 TypeScript 语言。我们的 API 着重于以简洁而结构良好的方式表达有限状态机，以解释分布式计算的当前状态。我们从乘客的角度，考察出租车示例中请求与竞拍部分的实现，以此进行说明；相应的 TypeScript 代码见清单 1。

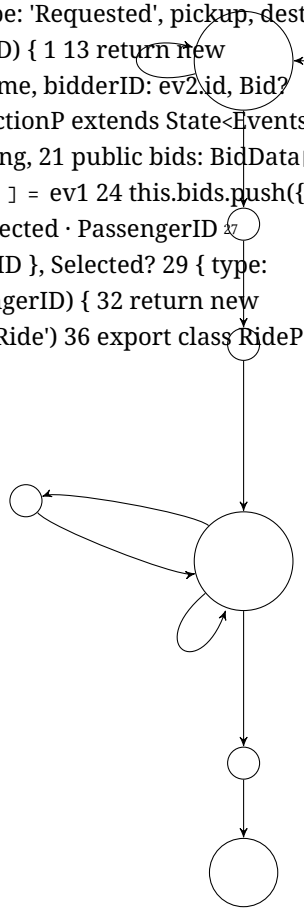
就本节而言，只需知道机器的每个状态都由一个 TypeScript 类表示，其中包含用于命令调用的方法，方法名前缀为“exec”；还包含用于计算下一状态的事件处理方法（方法名前缀为“on”）。事件处理方法参数的类型以及命令方法的返回类型都很重要。其他所有细节，包括运行时求值，将在第3节中介绍。

代码清单 1 还展示了与该代码片段对应的有限状态自动机，该自动机由`machine-check`构建工具推断得出（参见第 3.2节）：

- 自动机的状态 InitialP、AuctionP 和 RideP 分别对应代码片段中同名的类；
- 自动机的状态 1 和 2 对应于穿插在作为参数传递给事件处理方法 onRequested 的事件之间的隐式状态（同样，状态 3 对应于 onBid，状态 4 对应于 onSelected）；
- 命令方法对应于自动机中的自环，标签由命令名称和结果事件日志类型组成，形式为 `c`/`1`；
- 事件处理程序对应于转移或转移序列，其中每个转移都标记有以下形式的事件类型 `t`?.

```
1 // 其他事件的写法类似；"type" 属性与类型名称匹配（由工具检查） 2 type Requested = { type:
'Requested'; pickup: string; dest: string } 3 type Events = Requested |Bid | BidderID | Selected |...
4 5 /** 角色 P 的初始状态 */ 6 @proto('taxiRide') // 装饰器将推断出的协议注入运行时 InitialP 请求 /
Requested 7 export class InitialP extends State<Events> { 8 constructor(public id: string) { super() }
9 execRequest(pickup: string, dest: string) { 10 return this.events({ type: 'Requested', pickup, dest })
Requested? 11 } 12 onRequested(ev: Requested, ev1: Bid, ev2: BidderID) { 1 13 return new
AuctionP(this.id, ev.pickup, ev.dest, [{ 14 price: ev1.price, time: ev1.time, bidderID: ev2.id, Bid?
15 }]) 16 } 2 17 } 18 @proto('taxiRide') BidderID? 19 export class AuctionP extends State<Events>{
20 constructor(public id: string, public pickup: string, public dest: string, 21 public bids: BidData[])
{ super() } 3 22 onBid(ev1: Bid, ev2: BidderID) { 23 const [price, time ] = ev1 24 this.bids.push({
price, time, bidderID: ev2.id }) AuctionP 25 return this 26 } Select / Selected · PassengerID 27
execSelect(taxiId: string) { 28 return this.events({ type: 'Selected', taxiID }, Selected? 29 { type:
'PassengerID', id: this.id }) 30 } 4 31 onSelected(ev: Selected, id: PassengerID) { 32 return new
RideP(this.id, ev.taxiID) PassengerID? 33 } 34 } RideP 35 @proto('taxiRide') 36 export class RideP
extends State<Events>{ ... }
```

清单 1. TypeScript 中状态机的定义



上文概述的对应关系是我们对群进行形式化的基础，也是库 `machine-runner`的核心，该库在第 3 节中介绍。

请注意，在自动机中，我们对载荷进行抽象，只考虑事件的类型。我们也忽略不涉及事件发出/消费的内部计算（例如，状态 AuctionP 的构造函数参数计算并不重要）。

2.2 命令执行与事件消费

第 2.1 节讨论的机器的自动机表示，使我们能够在定义机器行为时减少技术细节。（为方便审稿人，本节引入的构造的形式化定义见附录 A.1。）

下文中，我们固定一组事件类型，以 t 表示；以及一组事件，以 e 表示。当事件 $\vdash e : \mathsf{t}$ 时，我们写作 e 属于类型 t，并假设这一关系是可判定的。每个事件都可以被识别，并与发出它的机器关联：令 $source(e)$ 表示生成 e 的机器的身份。事件日志是事件序列，记作 $e_1 \cdot e_2 \cdot \ldots$。

为了启用命令，带有日志 l 的机器 M 被隐式地置于状态$\delta(\mathsf{M}, \mathsf{l})$。M 的确定性确保了这一状态是唯一的。我们通过转移函数计算 $\delta(\mathsf{M}, \mathsf{l})$，方法是简单地调整有限状态自动机的标准转移函数。从 M 的初始状态开始，我们以归纳方式移除最旧的事件（记为 e），并将其与当前状态（记为 q）的出向转移进行比对：如果 q 存在一个标签为 t? 的转移，且 e 的类型为 t，我们就从该转移的目标状态开始处理日志的其余部分；否则丢弃事件 e，并从状态 q处理日志的其余部分。

机器可以被视为代理（无论是算法还是人类）的代理程序，用于处理本地日志中的信息、得出结论，并做出可能导致调用已启用命令的决策。例如，一旦处于 AuctionP状态，2.1 节中的机器便允许乘客执行命令选择，从而触发类似如下事件日志的生成：已选中 · 乘客 ID 。这一事件序列会被添加到本地事件日志中，使机器先通过消费事件 已选中 移动到状态 RideP，然后通过消费事件 乘客 ID移动到状态 2。这就是为什么推断出的机器状态图将命令记录为自环，而只有事件消费才可能引发状态变化。下文中我们只考虑确定性机器，即从每个状态出发，不存在两个标有相同事件类型的不同出向转移。

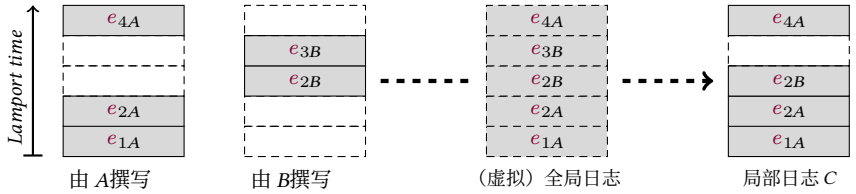
需要注意的是，只有在完全处理本地日志后所达到的状态中启用了下一个命令，才能调用它。此外，发出的事件会追加到其来源机器的本地日志中。这确保了因果关系保持，因为新事件的顺序位于该机器此前已知的所有事件之后。

请注意，当乘客的机器处于状态 2 时，若不消耗一个类型为 AuctionP 的事件，它就无法到达状态 AuctionP 。也就是说，至少有一台出租车的机器必须为该行程出价。由于这些机器的行为各不相同，我们称它们为 非对称。

2.3 群体与日志传输

计算模型的最后一部分，是事件日志如何在群中的机器之间传播。一旦我们建立了将事件从一台机器发送到另一台机器的机制，上一节的规则便描述了这会如何影响接收方：本地日志中包含更多事件，从而计算出新的当前状态，进而可能改变可用命令的集合。

我们目标的关键在于：同一机器的不同副本在消费相同事件时，必须达到相同状态。根据上述定义的状态转移函数，通常只有当局部日志中的事件顺序一致时，才能实现这一点。我们通过假设事件之间存在全序来解决这一问题，且机器之间无需协调（其可实现性将在第 3.4 节中说明）。这一全序的一项关键性质是必须保持因果性：新事件在局部日志中排在其前驱事件之后，因为它是基于整个局部日志所产生的效果。请注意，相对于本地尚未知晓的事件，其顺序可以任意确定，但必须定义明确。为此，我们引入全局日志这一概念性概念。图 1 展示了日志如何传播：每个事件最初都位于发出它的机器的局部日志中，同时，按照全序，它也在虚拟全局日志中占据一个位置。日志传输被建模为一台机器扩大其复制的全局日志子集；在实践中，事件会从一个局部日志发送到另一个局部日志，但选择源和目标的具体算法与我们的理论无关。



事件的下标指定 Lamport 时间戳以及生成这些事件的机器身份

图 1. 按源切分的事件：每个事件最初都发生在发出它的机器上。日志的传播方式使得接收方（例如机器 C）持有每个源切片的一个前缀，这构成了全局日志的部分视图。接收方的本地日志与全局日志的顺序一致。最终，每个事件都会到达所有机器，填补可能存在的暂时性空隙。

由于采用了无协调全序，传入事件可能会被插入局部日志的中间位置，从而改变在计算当前状态时对其后所有事件的解释。例如，乘客选择出租车的同时，另一辆出租车提交了新的出价。如果选择事件排在出价事件之前，那么之后检查日志时可能会发现，乘客选择的并非最优选项——但该选择仍然有效，出价则被忽略。在基于中央数据库的系统中，这相当于“出价”事务因排在“选择”事务之后而被拒绝。

另一方面，如果 已选中 事件的顺序位于 出价 和 竞价者 ID 事件之间，那么日志复制完成后，该事件就会被忽略。在这种情况下，有两种合理的处理方式：若要尊重乘客的意愿，就需要再次执行选择这一补偿操作；或者将新的出价纳入决策，这可能会导致不同的结果。

鉴于 Actyx 复制事件的方式，我们定义如下关系：

定义 *2.1*（子日志关系）。日志 $l = e_1 \cdots e_n$ 按照如下方式在其元素上诱导出一个全序 $<_l$：$e_i <_l e_j \Leftarrow\Rightarrow i < j$。日志上的子日志关系 $\sqsubseteq$ 要求存在一个保持顺序且向下完备的态射，从 l_1 映射到 l_2。形式化地，$l_1 \sqsubseteq l_2$ 若

(1) l_1 中的所有事件都按相同顺序出现在 l_2 ($l_1 \subseteq l_2$) 中（$<_{l_1} \subseteq <_{l_2}$）；并且 (2) l_1 的按源划分是 l_2 对应划分的前缀，即对于来自给定 *source* 的所有 $e_1 \in l_1$、$e_2 \in l_2$，若 $e_2 <_{l_2} e_1$，则 $e_2 \in l_1$。

示例 2.2（子日志）。设 $l = a_0 \cdot b_1 \cdot d_1 \cdot c_2 \cdot e_1$ 是一个日志，其中下标不同的事件由不同的机器发出（即 $source(a_0) \neq source(b_1)$）。我们有 $b_1 \cdot c_2 \sqsubseteq l$，但没有 $b_1 \cdot a_0$（由于顺序颠倒），也没有 $d_1 \cdot c_2$（因为 d_1 的存在要求 b_1 也必须存在）。 ◇

有了这一概念后，我们通过几个步骤来形式化日志传输。首先，我们对两个日志的合并进行建模，使结果包含两个日志中事件集合的并集，以及 $<_l$ 关系；请注意，不可能发生冲突，因为某个事件只有在先前执行过合并操作后，才可能同时出现在两个日志中。其次，我们通过指出第一步满足的结果集合由交错算子给出，来表达全局顺序是任意的[1] $\bowtie$：

$$l_1 \bowtie l_2 = \{ l \mid l \subseteq l_1 \cup l_2 \quad \text{and} \quad l_1 \sqsubseteq l \quad \text{and} \quad l_2 \sqsubseteq l \} \tag{1}$$

第三，我们通过要求每个新事件同时添加到虚拟全局日志中，来确定一种非确定性的选择，从而选出一种可能的顺序；虚拟全局日志是复制的数据源。

[1]我们的 $\bowtie$ 算子是同步交错算子的一种特例，用 [36], 的说法来说，它也称为同步组合 [15]和混合积 [10]。

2.4 形式化

一个群（大小为 n）是一个二元组 (S, l)，其中 S 将索引 $1 \le i \le n$ 映射到机器及其本地日志，即 $S(i) = (M_i, \mathsf{l}_i)$，而 l 是全局日志。为了方便起见，我们将 $(M_1, \mathsf{l}_1) \mid \ldots \mid (M_n, \mathsf{l}_n) \mid \mathsf{l}$ 记为群 (S, l)，其中对于所有 $1 \le i \le n$，$S(i) = (M_i, \mathsf{l}_i)$。当 $Đ_{1 \le i \le n} \mathsf{l}_i = \mathsf{l}$ 且 $\mathsf{l}_i \sqsubseteq \mathsf{l}$ 对于 $1 \le i \le n$ 成立时，群 $(M_1, \mathsf{l}_1) \mid \ldots \mid (M_n, \mathsf{l}_n) \mid \mathsf{l}$ 是一致的；换句话说，机器 M_i 的本地日志 l_i 由 S 中实际发出的事件组成，并且按照这些事件产生的顺序排列。

日志的类型是其事件类型的序列。下文中，我们令 $\mathtt{l}$ 遍历 日志类型，即有限的事件类型序列。相应地，当 $\vdash l : \mathtt{l}$ 时，表示 l 具有类型 $\mathtt{l}$。注意，这一类型关系的可判定性直接源于我们的假设：事件的类型判定是可判定的。

下面的推理规则 局部 和 传播规则形式化描述了群体的行为。与其他行为类型系统的一个显著区别是，我们不会随着计算的进展而演化机器类型，而是演化局部日志和全局日志。回顾一下，机器 M 带有日志 l 的机器隐式处于状态 $\delta(M, \mathsf{l})$。因此，当状态 $\delta(M, \mathsf{l})$ 启用命令 $\mathtt{c}/\mathtt{l}$ 且 l' 具有类型 $\mathtt{l}$ 时，我们可以定义关系 $(M, \mathsf{l})\ \mathtt{c}/\mathtt{l} \longrightarrow (M, \mathsf{l} \cdot \mathsf{l}')$。

$$\frac{S(i) = (M, l_i) \qquad (M, l_i) \xrightarrow{\mathtt{c}/\mathtt{l}} (M, l_i') \qquad source(l_i' \setminus l_i) = \{i\} \qquad l' \in l \bowtie l_i'}{(S, l) \xrightarrow{\mathtt{c}/\mathtt{l}} (S[i \mapsto (M, l_i')], l')}\ [\textsc{Local}]$$

$$\frac{S(i) = (M, l_i) \qquad l_i \sqsubseteq l_i' \sqsubseteq l \qquad l_i \subset l_i'}{(S, l) \xrightarrow{\tau} (S[i \mapsto (M, l_i')], l)}\ [\textsc{Prop}]$$

规则 局部 描述了在群的第 i台机器上启用的命令的调用(S, l)，假设机器按其索引标识；所有新生成的事件都源自第 i台机器，并使用上文定义的 $\bowtie$运算符以非确定性方式合并到全局日志中。最后，通过将第 i台机器的本地日志更新为 l_i（由命令 $\mathtt{c}$ 的执行产生）来更新群。

规则 传播规则 定义了机器之间的事件日志传播。其思路是非确定性地选择一台本地日志严格包含于全局日志的机器，识别一个更大的子日志 $l_i' \sqsubseteq l$，并通过将 l_i' 指定为该机器的新本地日志，将事件传输到该机器。

规则 局部 和 传播规则 分别形式化了第 2.2 节所述的命令执行效果，以及第 2.3节所示的非确定性日志传输机制。请注意，我们的形式化作用于通过追加新生成的事件而增长的日志。这些特性在实现局部优先原则方面发挥着重要作用，并使我们能够以形式化方式表示[2] 第 1节所讨论的冲突。

下文中，$\Longrightarrow$ 表示一个可能为空的归约步骤序列，$\mathtt{c}$ 遍历命令；对于群协议，我们令

$$\Longrightarrow = \longrightarrow^{\star} \qquad \text{where} \qquad \longrightarrow = \xrightarrow{\tau} \cup \bigcup_{\mathtt{c},\mathtt{l}} \xrightarrow{\mathtt{c}/\mathtt{l}}$$

机器的归约步骤序列同样如此。以下引理对于一致的群体成立。

引理 2.3（相干性保持）。若$(S,\ l)$是一个相干群体，且$(S,\ l) \xrightarrow{\alpha} (S',\ l')$则$(S',\ l')$是相干的。

[2]附录A.3 通过贯穿示例展示了我们的模型中的细微之处。

引理 2.4（最终一致性）。如果 $(S, l) = (M_1, l_1) \mid \ldots \mid (M_n, l_n) \mid l$ 是一个相干群体，那么 $(S, l) \Longrightarrow (M_1, l) \mid \ldots \mid (M_n, l) \mid l$。

注意，规则 [局部] 和 [传播规则] 并不能自动保证机器之间以及群中所有信息之和（即全局日志）的一致性。在我们的贯穿示例中，某台出租车机器可能一直停留在其初始状态，而群中的其他部分却在跟踪乘客运输请求的完成情况，这是因为它暂时没有应用传播规则。本文其余部分旨在解决这一问题。

2.5 计算模型的讨论

与会话类型及其局部投影会针对每个会话重新实例化类似，我们的模型总是从空日志开始。贯穿示例对完成单次出租车行程的交互进行建模，并未描述例如每辆出租车在司机一班工作期间参与的一系列行程。

我们的计算模型类似于成熟的状态机复制(状态机复制) [32]；然而，它在以下方面有很大不同：

机器并非完全相同：我们并不一定认为群中的所有机器都是完全相同的副本；事实上，我们期望它们有所不同，以便在协调交互中承担不同的角色，如示例1.1所示。

传播效果而非命令：状态机复制中的日志由命令序列构成，每条命令都需要在每个副本上执行，并产生完全相同的输出（确定性）。而我们构建的是事件日志，用于传递命令调用在特定机器上产生的效果。这意味着我们不要求命令执行具有可复现性。例如，示例 1.1 中的两个不同竞标者执行命令 出价时，可能会产生完全不同的报价。因此，每条命令只在发起机器上执行一次，其产生的效果会传播到其他机器，并被它们视为有效结果接受。换句话说，我们的场景是可信组件之间的协作交互。

命令的执行：在我们的模型中，命令为群体运行所处的环境提供了一个接口。机器中的一条命令可能由环境非确定性地选定，并在机器的当前状态下处于启用状态时执行。命令的执行会触发事件，使机器能够转移到另一个状态。

机器可以空闲： 与会话类型约束不同，能够启用命令的机器并非必须执行其中某个命令。要使群取得进展，只需最终调用一个已启用的命令。

局部日志之间的一致性： 虽然 状态机复制 可确保两个站点的事件日志中，一个始终是另一个的前缀，但正如第 2.2 (参见图 1）所详述的，这在我们的系统中并不成立。根据引理 2.4，我们的模型只能提供最终一致性。

稳定性与推测计算： 与 状态机复制不同，后者只会针对事件日志中的稳定部分推进本地状态；我们的模型会在收到其他机器发来的子日志后，立即据此计算出新的状态。这会产生一些中间状态，而如第 2.2 (参见图 1）所示，之后收到的子日志可能会“合并到过去”，从而使这些中间状态失效。

3 工具支持

我们的理论研究还配有一套软件工具，用于将群体实现为由经过类型检查的 TypeScript 机器组成的组合 [30]，并基于 Actyx 中间件运行这些机器 [1]。图中展示了这一生态系统：2。

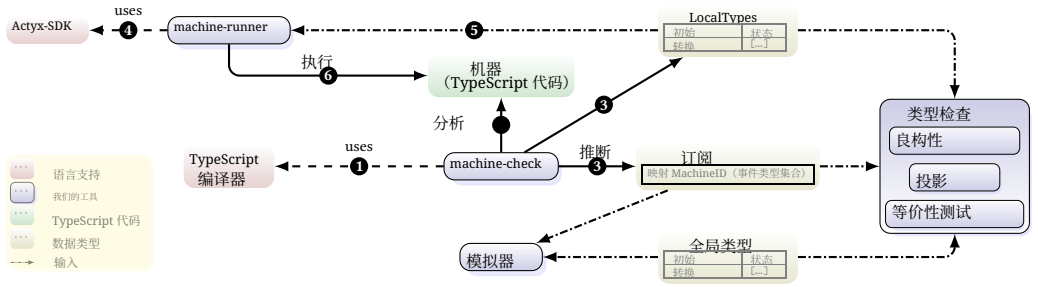


图 2. 工具生态系统

3.1 编译后机器的执行

machine-runner库使用 Actyx SDK（参见图2 4 来驱动以 TypeScript 编写的机器；更准确地说，它会运用5 6

1，我们以第 2 行的Requested事件类型为例。使用名为type的属性来存放单例类型的字符串（此处为：'Requested'），是 TypeScript 中表达可辨识联合（或称带标签联合）的常见方式，如第 *3* 行所示。

每个机器状态都由一个类表示，该类继承自 machine-runner 库提供的 State 基类（在 JavaScript 术语中也称为 原型）。它既用作机器状态的标记，也用于携带类型参数，以将所有发出的事件约束为同一种类型：继承而来的 this.events 函数（例如第 28 行所用）是一种实用工具，可帮助 TypeScript 正确捕获元组类型 [已选中，乘客 ID]（而不是其他情况下推断出的数组类型 Events[]），并断言每种事件类型都符合 Events 类型。

使用乘客的机器的程序会首先通过构造初始状态来开始，例如“`new InitialP('myID')`”。结合一组合适的 Actyx 事件标签（例如`'taxiRide:12345'`，用于表示这次特定的出租车行程协议会话）以及状态变更回调（见下文），然后将此初始状态传递给库的`runMachine`函数。这将使用 Actyx SDK 为带有这些标签的事件设置订阅，Actyx 会先传递所有已在本地知晓的历史事件，然后切换到实时模式。

<escape>每当接收到一个事件时，都会将其安排由相应的事件处理方法进行消费；为此，会在状态所属类底层的 JavaScript 对象中，动态查找与该事件类型对应的处理方法。如果该方法只接受一个参数，则会立即通过调用该方法来消费该事件，并返回机器的下一个状态。否则，该事件将进入队列，等待接收后续所需的事件类型，依此类推，直到所需的事件序列完整，且该方法能够使用全部参数进行调用。在整个过程中，只要传入的事件类型没有匹配的处理程序，或者不属于参数列表中下一个所需的类型，该事件就会被丢弃，具体说明见第 2.2 节。</escape>

每当机器状态发生变化（即 $\delta(M, I)$ 计算出一个新值时），新状态就会传递给应用程序先前传给`runMachine`的函数——这种机制在 TypeScript 中称为 回调 （请注意，这种语言采用带有可变绑定的命令式风格）。它可以更新用户界面，或触发算法来计算响应。状态的命令方法可用于构造已启用命令的适当事件载荷，然后通过 SDK 函数将其存储在 Actyx 中，并通过

事件订阅（现在带有元数据）应用于当前状态，并最终触发对回调的再次调用。

3.2 在运行时强制执行类型检查

熟悉 TypeScript 的读者可能已经注意到，我们在这里略过了一个难点：TypeScript 类型在运行时会被完全擦除，这意味着`machine-runner`代码无法通过事件类型找到事件处理方法，也无法得知该方法接受多少个参数以及这些参数的类型。因此，`machine-check`构建工具的首要职责是分析 TypeScript 代码，并确认所有事件类型都以可在运行时识别的方式声明在其`type`属性上——这样一来，只需在该属性的值前加上`'on'`，便可构造出处理方法的名称。第二项职责是提取所有事件处理程序的函数签名，检查每个处理程序的名称是否与其第一个参数类型的名称相对应，然后为每个状态构建映射，将第一个事件类型映射到后续事件列表（该列表可能为空）。通过装饰`runMachine`用户编写的状态类，便可向其提供这些信息：@protodecorator 会在 JavaScript 虚拟机加载类定义时对其进行转换，并覆盖从 State 原型继承的 reactions 方法。为此，proto 函数的实现需要访问 machine-check 的提取结果。具体做法是导入由 machine-check 生成的源模块，该模块以 JSON 格式包含所有协议信息5 [38] [37]。

尽管前文提到的`machine-check`职责对`machine-runner`的运行至关重要，但该构建工具更有趣的功能在于推断局部类型和订阅（图 中的箭头2），以及启动对群协议的类型检查流程。为此，我们将 TypeScript 编译器用作 一个库，以获取 用户程序的、经过完整类型标注的 AST 表示。由于我们只关注机器，因此入口点是`State`的子类；这些子类通过以“Ini- tial state for role”开头的文档注释标记为初始状态，如清单1第 5 行所示。此注释还有第二个作用，即命名该机器希望扮演的角色（本例中 P 代表乘客）。第 6 行的`@proto`装饰器除了执行上文所述的运行时职责外，其参数还包含提供角色名称上下文的群协议名称——我们将在后续章节详细讨论这两个概念；`machine-check`要求在名称对应的文件中以 JSON 格式查找群协议的定义。最后，`machine-check`通过检查（箭头 ⑤）状态类的方法名称和签名，整理出命令和事件处理方法列表；然后以相同方式递归处理事件处理程序的返回类型。在事件处理程序参数列表中出现的任何事件类型，都会自动添加到机器的订阅集合中（投影所需，详见第5节）。

3.3 类型检查、模拟及更多

根据上述分析，`machine-check`为用户程序中的每个机器定义汇集了以下内容：群协议、角色名称、订阅、状态和转换。随后，这样的元组会再次以 JSON 格式传递给类型检查工具——我们的第三项成果，该工具使用 Haskell 编写。此工具首先检查所提供的群协议和订阅是否格式正确（依据第6节所述规则），然后计算给定角色的投影，最后检查推断出的机器类型是否与投影结果等价（状态名称无关紧要）。

此外，我们还提供了一个用 Haskell 编写的辅助工具，用于模拟该模型的形式操作语义。对于给定的协议和订阅，该工具会计算

投影，并根据这些投影模拟由机器组成的群体的执行。它支持穷举生成和随机生成不超过给定长度的轨迹。该工具已用于验证有关贯穿示例的论断和结果。

上述工具的详细信息已收录在随本文一同提交的工件中。随附的论文还通过一个示例项目，演示如何利用推断出的机器类型通用地渲染机器 UI。除了显示计算的当前状态外，UI 还允许用户通过调用已启用的命令与机器交互；命令参数则通过自动生成的 HTML 表单收集。

3.4 Actyx 的显著特性

Actyx 中间件为带标签的事件日志提供可靠、持久的发布–订阅机制，并为所有事件提供一种无需协调的全序，正如我们在第 2.3 节中形式化阐述的那样。Actyx 是一个点对点系统，由运行于 网络设备 上的节点构成。事件顺序通过结合 Lamport 时钟 [28] 来实现，该时钟提供保持因果性的逻辑时间戳（每发出一个事件，时间戳还会额外递增），并结合基于 ED25519 密钥对公钥的唯一节点标识符。每个节点可以承载多个机器；这些机器的事件日志通过使用不同标签加以区分。每个事件都带有一组标签，而事件日志则通过与这些标签对应的布尔组合进行查询，从而对相应事件集合执行并集或交集操作。由于 Actyx 节点仅是部署单元，我们的模型只涉及机器——节点不会带来语义上的差异。

子日志关系（定义 2.1）源于 Actyx 对本地日志的刻画方式。事件会写入每个节点的单写者、仅追加日志切片。本地日志通过各切片在局部可用的长度来刻画。与标记单个事件是否存在相比，这种表示方式简洁得多，有利于定期广播以促进可靠复制。

实时事件订阅有两种模式：subscribeSDK 方法会产生一个流，在发出所有过往事件后，每当收到或在局部发出一个新事件时，都会立即发出该事件。subscribeMonotonic SDK 方法执行相同的操作，但只会持续到某个新事件需要排在先前已发出的事件之前为止（在 Actyx 中，这称为时间穿越 ）。machine-runner 使用后一种函数，并在发生时间穿越时从初始状态重新开始计算；请注意，此时已发出的事件日志中的历史部分将包含导致时间穿越的事件。

4 群协议

一个群协议（下文简称协议）描述了由一群机器实现的预期总体事件日志结构；按照会话类型的术语，它对应于一种全局类型，而我们的机器所扮演的角色与局部类型类似。该协议既捕获了整体通信结构，也涵盖了使用机器实现该结构所需的相关细节。与行为类型的惯例一致，群协议依赖于一个理想化的环境，其中所有通信都可靠无误且瞬时完成。下一节将通过投影操作，说明如何借助机器实现该协议。

为简化递归类型的技术处理，我们将满足余归纳文法的一个群协议定义为一个正则项[3]

$$\mathsf{G} \overset{\text{co}}{::=} \sum_{i \in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i$$

[3]如果一个项由有限个不同的子项构成，那么它就是正则的。此条件确保余归纳文法生成的语言可以用有限方式表示，既可以使用所谓的“μ 记法” [31]，也可以表示为有限方程组的解[8]。此外，这些结构与有限状态自动机之间存在对应关系；感兴趣的读者可参阅附录A.4，或参阅 [8]以了解全面论述。

其中，R、R_1, R_2,等表示角色。当 I为空时，我们记作 $G = 0$；对于共同的后续过程，即对所有 $i \in I$均有 $G_i = G$ 时，我们记作 $(\text{Í}_{i\in I} c_i@R_i\langle l_i\rangle i).G$。直观而言，协议通过某个角色 R_i 调用命令 c_i而向前推进：将一个类型为 l_i 的非空事件序列追加到全局日志中，并继续执行协议 G_i。正如第 2.4 节末尾所述，群协议中指定的选择，其决议并非由参与该选择的角色实例共同协调（即不存在唯一的选择者）。事实上，参与选择的不同角色的实例可能同时启用命令，同一角色的不同实例也可能启用不同的命令（请记住，每台机器都维护独立的本地日志，且事件复制是异步的）。这与迄今为止大多数其他行为类型系统形成对比；那些系统不允许出现此类竞争条件。

定义 4.1（确定性）。如果事件类型 $l_i[0]$ 两两不同，且所有 G_i均为日志确定的，则群协议 $G = \text{Í}_{i\in I} c_i@R_i\langle l_i\rangle i \,.\, G_i$是日志确定的。如果元组 (c_i, R_i) 两两不同，且所有 G_i 均为命令确定的，则 G 是命令确定的。如果 G 同时满足日志确定性和命令确定性，则 G 是确定的。

此后我们只考虑确定性群协议。请注意，由于正则性约束，群协议的确定性显然是可判定的。

示例 *4.2*（出租车服务）。我们可以将示例中的协议形式化1.1 如下：

$$G = \text{请求@Ph}\langle\text{已请求}i\rangle.\ \text{报价@Th}\langle\text{出价}\cdot\text{竞价者 IDi}\rangle.\ G_{auction} \quad G_{auction} = \text{报价@Th}\langle\text{出价}\cdot\text{竞价者 IDi}\rangle.\ G_{auction} + \text{选择@Ph}\langle\text{已选中}\cdot\text{乘客 IDi}\rangle.\ G_{choose} \quad G_{choose} = \text{到达@Th}\langle\text{已到达}i\rangle.\ \text{开始@Ph}\langle\text{已开始}i\rangle.\ G_{ride} + \text{取消@Ph}\langle\text{已取消}i\rangle.\ \text{收据@Oh}\langle\text{收据}i\rangle.\ 0 \quad G_{ride} = \text{记录@Th}\langle\text{路径}i\rangle.\ G_{ride} + \text{完成@Ph}\langle\text{已完成}\cdot\text{评分}i\rangle.\ \text{收据@Oh}\langle\text{收据}i\rangle.\ 0$$

根据命令和角色描述的结构，可直接从示例 1.1 推导得出，其中事件日志类型根据更具体的要求进行了补充。事件类型 竞价者 ID表示身份识别信息，说明并非所有角色都关心所有事件：可以合理地假设，办公室不需要了解所有竞标者，只需知道哪辆出租车已 选中 即可。很容易验证，协议 G 既是日志确定的，也是命令确定的。

◇

仿照对机器的表述，我们通过生成和处理事件日志，为群协议赋予操作语义。状态计算的主要区别在于，群协议生成和使用的是日志，而非事件，如上例中的 报价、选择和完成命令所示。

对于群协议 $G = \text{Í}_{i\in I} c_i@R_i\langle l_i\rangle i \,.\, G_i$，若存在 $j \in I$，使得 $c = c_j$、$l = l_j$ 且 $G' = G_j$，则记作 $G \xrightarrow{c\,/\,l} G'$。群协议 G 处理日志 l 后到达的状态，使用机器上转移函数的扩展版本 δ 计算，具体如下：

$$\delta(G, \epsilon) = G \qquad \delta(G, l) = \begin{cases} \delta(G', l'') & \text{if } G \xrightarrow{c\,/\,l} G' \text{ and } \vdash l' : l \text{ and } l = l' \cdot l'' \\ \bot & \text{otherwise.} \end{cases}$$

与机器的定义类似，转移函数 $\delta(G, l)$ 返回处理完整日志 l 后群协议 G的后续状态。我们强调，δ是群协议上的偏函数；事实上，当日志 l无法根据 G 生成时，它没有定义（这不同于 δ 在机器上的定义，后者是全函数，因为它只会丢弃无法识别的

事件）。另请注意，δ在日志确定性群协议上定义良好，因为某个分支中的日志不可能是选择中其余分支日志的前缀。

群协议的操作语义由一个带标记的转移系统定义，该系统由以下归纳规则给出：

$$\frac{\delta(\mathsf{G}, l) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} \mathsf{G}' \qquad \vdash l' : \mathtt{l} \qquad l' \text{ log of fresh events}}{(\mathsf{G}, l) \xrightarrow{\mathtt{c}\,/\,\mathtt{l}} (\mathsf{G}, l \cdot l')}\,[\text{G-Cmd}]$$

一种群协议 G 及日志 l 支持命令 c，调用该命令后，日志会追加新的事件 l'，类型为 l 之后，可能允许调用另一条命令。通过包含节点 ID 和逻辑时间戳等信息，可以保证 l' 中事件的新鲜性。

示例 *4.3*（理想化出租车服务）。考虑示例 4.2中的协议，从$(\mathsf{G}\ \epsilon)$开始。调用 请求 命令后，我们的日志包含 请求，并到达状态 $\mathsf{G}_{\text{Auction}}$。两次出价后，乘客做出选择，于是我们到达

$$\delta(\mathsf{G}, \mathit{requested} \cdot \mathit{bid}_A \cdot \mathit{bidderid}_A \cdot \mathit{bid}_B \cdot \mathit{bidderid}_B \cdot \mathit{selected} \cdot \mathit{passengerid}) = \mathsf{G}_{\text{choose}}$$

与 $\mathsf{G}_{\text{choose}}$ 提供两种选择：乘客发起 取消 或出租车 到达。 ◇

5 投影

为定义我们的投影操作，引入一种与迄今所使用的基于自动机的表示等价的机器文本表示会很方便。令 κ 表示一个将命令映射到非空日志类型的有限函数；我们允许将 κ 视为集合（函数 κ的图），例如将 $\mathtt{c}\,/\,\mathtt{l} \in \kappa$写作 $\kappa(\mathtt{c}) = \mathtt{l}$，或者将 $\{\mathtt{c}_1\,/\,\mathtt{l}1, \ldots, \mathtt{c}_h\,/\,\mathtt{l}_h\}$写作函数$\kappa$，该函数将 $\mathtt{c}_i$映射到 $\mathtt{l}_i$，对每个 $i \in \{1, \ldots h\}$均如此。$\mathtt{t}_i$遍历事件类型。

与群协议类似，我们的机器的文本表示是一个正则项[4]，其服从以下余归纳文法：

$$\mathtt{M} \overset{\text{co}}{::=} \kappa{\cdot}[\mathtt{t}_1?\,\mathtt{M}_1 \ \&\ \cdots\ \&\ \mathtt{t}_n?\,\mathtt{M}_n] \tag{2}$$

当 $n = 0$ 时，我们将 $\kappa{\cdot}[\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ 简写为 $\kappa{\cdot}\,0$；当 κ 是空映射时，则将其简写为 $\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n$。我们还用 $\&_{1 \le i \le n}\mathtt{l}_i?\,\mathtt{M}_i$ 代替 $\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n$。

转而关注投影操作时，我们首先注意到，推动协议向前发展的职责分散在各参与者之间：群协议中的每个转移都标记有一个角色，该角色可以通过调用命令触发该转移。每台机器都扮演一个角色，其机器规约通过投影操作 $\mathsf{G}\downarrow_{\mathtt{R}}$获得。请注意，多台机器可以实现同一个角色。

一种简单直接的定义方式是 $\mathsf{G}\downarrow_{\mathtt{R}}$，使得 G 中的每个转移都会在机器中生成一系列事件转移，并在角色匹配时，在源状态上调用命令。

$$\Big(\sum_{i\in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i\rangle \,.\, \mathsf{G}_i\Big)\downarrow_{\mathtt{R}} = \kappa{\cdot}\big[\&_{i\in I}\,\mathtt{l}_i?\,\mathsf{G}_i\downarrow_{\mathtt{R}}\big] \qquad \text{where} \qquad \kappa = \{(\mathtt{c}_i\ /\ \mathtt{l}_i) \;\big|\; \mathtt{R}_i = \mathtt{R} \text{ and } i \in I\}$$

尽管简单，但除最简单的情况外，这种投影方案都会生成不必要地庞大的机器。更重要的是，出于安全性和效率方面的考虑，要求每台机器处理所有事件并不可取。我们非常希望能够将某些信息对特定角色保密（例如示例 4.2中的 乘客 ID），而且，如果每个角色只处理足以正确启用和禁用命令调用的信息，效率会最高。因此，我们定义了一种更合适的构造方式。

[4]这些正则项与有限状态自动机之间的对应关系，恰好导出了我们迄今采用的以有限状态自动机来表示机器的形式。

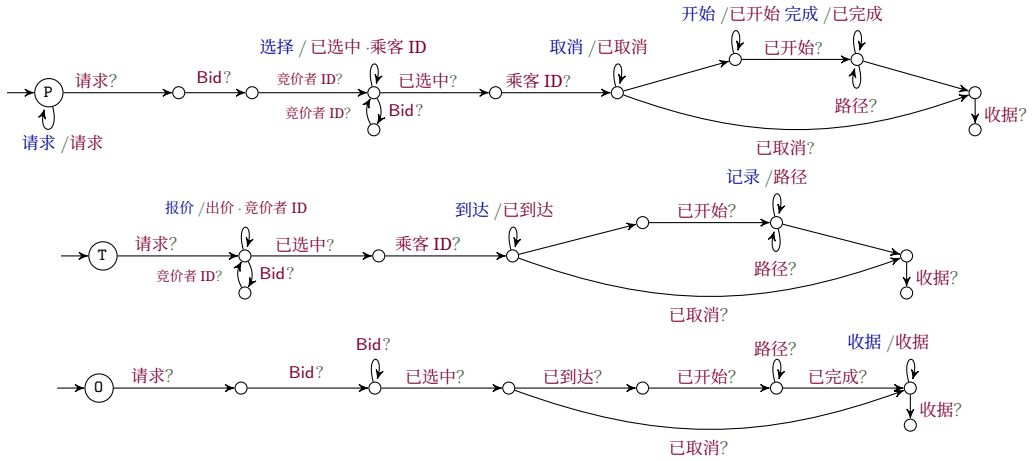


图 3. Pr ojections of the 贯穿示例 as automata (see AppendixA.3 for the textual presentation)

我们的投影基于这样一个概念：机器是否应处理某种类型的事件。从形式上说，投影操作由一个订阅参数化，即一个映射 σ它为每个角色分配一组该角色会响应的事件类型。给定一组日志类型 E，令筛选$(_,\ E)$为一个转换日志类型的函数，它仅保留 E 中的事件类型，同时保持其相对顺序。直观而言，订阅对应于发布–订阅框架中的主题；在这种框架中，各个进程会声明自己希望接收哪些类型的消息。

定义 *5.1*（投影）。给定一个群协议G 和一个订阅 σ，*G*G在角色 R 相对于 σ的投影，记作$G\downarrow_{\mathtt{R}}^{\sigma}$，定义如下：

$$\left(\sum_{i\in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i\rangle\ .\ G_i\right)\downarrow_{\mathtt{R}}^{\sigma} = \{\mathtt{c}_i\ /\ \mathtt{l}_i \mid \mathtt{R}_i = \mathtt{R} \text{ and } i\in I\}\cdot\left[\&_{j\in J} \mathit{filter}\left(\mathtt{l}_j, \sigma\left(\mathtt{R}\right)\right)\ ?\ \left(G_j\downarrow_{\mathtt{R}}^{\sigma}\right)\right]$$

其中 $J = \{i\in I \mid \text{筛选}\left(\mathtt{l}_i, \sigma\left(\mathtt{R}\right)\right) \neq \epsilon\}$.

请注意，当某个角色 R 未订阅选择该分支的命令所发出的任何事件类型时，我们会省略该分支的投影。我们选择对形式体系进行这种简化，是因为我们的良构条件（参见第 6 节）确保，如果某个角色参与了后续过程，那么它就会订阅该分支中的第一个事件。此外，请注意，这种剪枝仅适用于孤立的分支；通过其他路径可达的后续状态仍然属于该投影。

例5.2. 设 $\sigma = \{\mathtt{P}\ \mapsto E、\mathtt{T}\ \mapsto E、\mathtt{O}\ \mapsto E\setminus\{\mathtt{BidderID}, \mathtt{PassengerID}\}\}$，其中 E 是示例 4.2 中定义的协议 G 的所有事件类型的集合。G 在不同角色上的投影 $G\downarrow_{\mathtt{P}}^{\sigma}$=P、$G\downarrow_{\mathtt{T}}^{\sigma}$= T 和 $G\downarrow_{\mathtt{O}}^{\sigma}$=O 如图 3 所示。⋄

6 良构性

现在，我们将重点关注群协议的良构性条件。与行为类型领域的惯例一致，我们为全局规范建立充分条件，以确保投影具备相关属性，例如不会发生死锁或锁饥饿，并且不存在孤立消息。值得注意的是，我们关注的属性与标准情境中常见的属性大不相同（相关讨论见第7.4 节）。相反，我们旨在确保最终能够达成共识，即使某些参与者因对全局日志的了解不完整而做出不一致的选择。

只要最终恢复一致性（这会在信息在群内充分传播后实现），就可以容忍暂时的偏离。例如，在我们的贯穿示例中，即使乘客已经做出选择，只要选择事件尚未送达，出租车仍可能继续对该乘客的拍卖出价。一旦事件分别传播到做出偏离的出租车和乘客，这种暂时的不一致就会被识别并解决。

要实现具备这一属性的群体并非易事。本节其余部分将通过几个示例，说明我们所处情境中会出现的问题。针对每个问题，我们都会找出群协议必须满足的充分条件，以排除该问题（附录 A.3、A.5 和 A.6 基于贯穿示例，展示了现实场景中的协调问题）。这些条件最终汇聚为我们对良构性的定义（参见定义 6.7）。

6.1 关于因果性和传播

我们要探讨的第一个问题与命令被调用后如何禁用有关。在我们的设定中，这归结为对角色注册到事件类型的方式进行精细调整。例如，如果命令$\mathtt{c}$只有在另一个命令（例如$\mathtt{c}'$）执行后才应启用，那么执行$\mathtt{c}'$的角色就应订阅某种事件类型，该事件类型会在执行$\mathtt{c}$后发出。另一个例子是，如果执行某命令的角色未察觉由此产生的任何事件，那么该命令可能会一直处于启用状态。附录 A.7 中的示例A.5 针对我们的贯穿示例展示了这类问题。

另一类问题源于这样一个事实：事件会在群内异步传播，并且如示例A.8所示，多事件的发出并不保证会以一次原子传输的方式抵达所有其他机器。这种异常可以通过一个运行时系统加以排除：该系统从不将 [传播规则] 应用于单个命令所发出事件日志的严格子集；换言之，它将每次命令调用产生的日志视为一个原子单元。我们选择不限制运行时系统在网络站点之间传播事件的方式，因为我们认为，实现应当能够自由地以不同方式优化其策略（例如针对延迟、带宽、效率或一致性）。

我们将群协议活跃角色$\mathtt{G}$ 定义为：能够在$\mathtt{G}$ 顶层选择中选择某个分支的角色；并且，给定一个订阅 σ，将角色$\mathtt{G}$定义为：能够调用命令或订阅$\mathtt{G}$中的事件的角色。形式化地，

$$\mathit{active}\Big(\sum_{i\in I}\mathtt{c}_i@\mathtt{R}_i\langle\mathtt{l}_i\rangle\,.\,\mathtt{G}_i\Big)=\{\mathtt{R}_i\;\big|\;i\in I\}$$

$$\mathit{roles}\Big(\sum_{i\in I}\mathtt{c}_i@\mathtt{R}_i\langle\mathtt{l}_i\rangle\,.\,\mathtt{G}_i\,,\sigma\Big)=\bigcup_{i\in I}\left(\{\mathtt{R}\in\operatorname{dom}\sigma\;\big|\;\mathtt{R}_i=\mathtt{R}\;\vee\;\mathtt{l}_i\cap\sigma(\mathtt{R})\neq\emptyset\}\;\cup\;\mathit{roles}\,(\mathtt{G}_i,\sigma)\right)$$

（注意，后者是一个余归纳定义）。借助这一记法，我们定义以下足以避免上述问题的条件。

定义 *6.1*（因果一致性）。群协议$\sum_{i\in I}\mathtt{c}_i@\mathtt{R}i\langle\mathtt{l}_i\rangle$。 $\mathtt{G}_i$满足在订阅中因果一致 σ 当且仅当对于所有 $i\in I$

(1) $\mathtt{l}_i\cap\sigma(\mathtt{R}_i)\neq\emptyset$，且 (2) 若 $\mathtt{R}\in$在 $\mathtt{G}_i$ 中活跃，则 $\mathtt{l}_i\cap\sigma(\mathtt{R})\neq\emptyset$，并且对于所有 $\mathtt{R}'\in$角色 $(\mathtt{G}_i,\sigma)$，$\mathtt{l}_i\cap\sigma(\mathtt{R}')\subseteq\mathtt{l}_i\cap\sigma(\mathtt{R})$

条件（1）要求，执行某一命令$\mathtt{c}_i$的角色应观察到相应发出的事件中的某些事件$\mathtt{l}_i$。这一简单机制确保，只有在群协议预期的情况下，才能重复调用命令。条件（2）确保后续命令调用能充分追踪因果性。第一部分确保紧接着的命令必须等待启用转移发生，而第二部分则

保证后续命令生成的事件，排在前一命令产生的所有事件之后；这些事件须由某个角色在协议的后续演进过程中观察到。

示例 *6.2*。 贯穿示例的协议 G 在示例 4.2 中对于订阅 σ 在示例 5.2 中是因果一致的。事实上，这些命令生成的日志以两两不同的事件类型开头。因此，对于角色 P 和 T（它们会观察每个事件），这些条件显然成立。对于 O，我们注意到它们只执行收据命令；该命令应在 已取消 或 已完成之后执行，而这两个事件也会被 O 观察到。 ⋄

6.2 关于分布式选择

我们研究的下一个异常源于这样一个事实：我们的模型允许多个角色在没有协调的情况下同时处于活跃状态——对于本地优先协作所要求的完美可用性而言，这一属性至关重要。据我们所知，在所有全局类型系统中，这种行为都会被排除。我们应对不可避免地出现的冲突的策略是：允许机器做出不一致的局部决策，但在相应事件传播到所有相关参与者后，再对这些决策进行协调；示例A.10 在附录 A.5 中说明了示例中的会计办公室如何做出与其他参与者所采取决策不一致的选择。我们通过要求 确定性来解决这一问题。

定义 *6.3*（确定性）。群协议$\mathsf{G} = \sum_{i \in I} \mathsf{c}_i@\mathsf{R}_i\langle 1_i \rangle . \mathsf{G}_i$是带有下标$\sigma$的确定协议，当且仅当它是因果一致的，并且$\mathsf{R} \in$个角色$(\mathsf{G}_i, \sigma)$缠含$1_i[0] \in \sigma(\mathsf{R})$对所有 $i \in I$。

这种确定性定义源于我们的确定性规则定义4.1：我们根据日志中发出的第一个事件类型来标识一个分支。请注意，参与某个分支但未参与另一个分支的角色，可能会调用随后被作废的命令，而该角色却无法识别这种情况；我们将在未来的工作中探讨对此类错误进行补偿的机制，这可能需要加强上述规则。

我们注意到，定义6.1 中的确定性条件（2 ）可以放宽（允许角色跳过第一个事件与某个商定的最后一个事件之间发生的部分事件），但这会使构造和证明更加复杂。

示例 *6.4*。 群协议 G 在示例 4.2 对于订阅而言是确定的 σ 在示例5.2。 ⋄

6.3 关于干扰

冲突中失败方发出的事件应予以忽略，以便每台机器最终都能就每项选择达成一致。每台机器都必须能够在局部忽略此类事件，以避免下文所述的最后一类问题。

考虑一种出租车行程协议的变体，该变体重新使用完成 事件类型来表示取消。由于取消与已到达 标记行程开始之间存在竞态条件，如果参与者未观察到已到达 事件，取消实际上可能会被误认为行程已立即完成。为避免这种混淆，我们要求群协议的任何分支都使用专用事件类型进行通信，即该事件类型不能由任何其他命令发出。我们用群协议的所有子项（包括间接子项）的集合 *subterms*（G）来表述这一概念，其中群协议为G。请注意，由于我们的群协议是正则的，该集合是有限的。

在下文中，我们分别用 events (G) 和 guards (G) 表示所有事件类型的集合和用于标识分支的集合；形式上，如果 G Í c@Rhl i. G then

$$events(\mathsf{G}) = \bigcup_{i\in I}\left(events(\mathsf{G}_i) \cup \bigcup_j \mathtt{l}_i[j]\right) \qquad \text{and} \qquad guards(\mathsf{G}) = \bigcup_{i\in I}(\{\mathtt{l}_i[0]\} \cup guards(\mathsf{G}_i))$$

(注意，事件(0) = 守卫(0) = ∅ 并且我们以余归纳的方式定义这些集合，也就是说，取最大不动点；请记住，群协议是正则树，因此这些有限集可以在有限步内计算得出。

群协议G 在 事件类型 t 下不变t 未出现在 G 中，即 t ∉事件 (G)；或者（ii）它只作为同一选择的一部分出现，即存在唯一的 $\mathsf{G}' \in subterms(\mathsf{G})$，使得 $\mathsf{G}' \xrightarrow{\mathtt{c}\,/\mathtt{l}}$ 且 t ∈ l。

定义 *6.5*（无混淆性）。群协议G 是无混淆的，当且仅当G 对于守卫(G)中的所有事件类型都不变。

示例 *6.6*。 很容易验证，贯穿示例的协议 G （示例 4.2 ）对所有类型都满足不变性。出现在两个守卫中的唯一类型是 收据；不过，这些出现都关联到同一个子项。因此，该协议是无混淆的。 ⋄

6.4 将各部分拼接起来

由此，我们终于可以给出我们的良构性条件。

定义 *6.7*（良构性）。群协议$\mathsf{G} = \sum^{i\in I}\mathtt{c}_i@\mathtt{R}_i\langle\mathtt{l}_i\rangle.\,\mathsf{G}_i$相对于订阅 σ (σ，简称 -WF）是良构的，当且仅当

(1) G 是因果一致的、确定的且无混淆的；并且 (2) G_i 对于所有
σ 都是 $i \in I$;

N注意，良构性是以余归纳方式定义的，并且在群协议上是可判定的 .

例*6.8*。例6.2、6.4 和6.6 表明，例4.2 中的协议G 相对于订阅 σ （见例5.2）是良构的。⋄

The foll 以下结果确保投影在良构协议中保留确定性。 s.

命题6.9。 设 G 和 σ 分别为一个群协议和一个订阅。若 G 是 σ-*WF*，则 $\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}}$ 对于所有 R都是确定的。

良构群协议保证局部机器在每项选择上都能达成最终共识 [39]，正如我们接下来将展示的那样。然而，异常[5] 会出现在系统层面：

- 机器可能启用了某些命令，而如果模型是同步的，这些命令本应被禁用；这可能会导致发出一些之后需要忽略的事件。
- 事件仅根据其类型被忽略，因此，即使全局日志中的事件已完全传播，机器仍可能处理源自命令异常调用的事件。

W我们指出，上述第一个异常是本地优先架构要求所固有的 ts.

第二个异常可以通过跟踪完整因果关系信息的运行时系统来避免。我们选择不要求完整的因果关系跟踪，因为这会增加对实现的存储、通信和计算要求。我们较弱的因果关系模型支持在必要时部署到能力较弱的硬件上。

[5]Exa例子，对于理解本文并非必需，见附录A.6 以方便审稿人 ce.

7 群协议的正确实现

现在，我们将注意力转向对群协议的正确实现进行形式化刻画。正如前文各节所讨论的，我们偏离了主流（多方）会话类型通常所期望的属性，例如通信安全性、会话保真性和进展（即不存在死锁或其变体）。首先需要指出的是，在我们的模型中不存在通信不匹配，因为每台机器都会简单地忽略意外发生或不需要的事件（回顾第 2.1 节中δ 的定义2.1）。而会话保真性则倡导实现应当按照其类型所描述的方式运行，这通常意味着所有组件的状态始终与协议的全局状态保持一致。与此相反，我们旨在容忍偏差，只要所有机器最终能够就协议执行的状态达成一致即可。在我们的设定中，即使机器暂时出现分歧，执行协议的不同分支，实现仍然可能是正确的；如果允许基于对全局状态的不完整视图独立作出决策，那么这种情况完全在预期之中。因此，正确实现可能执行不同的命令序列，进而生成与相应协议推导出的日志不同的日志。在这种情况下，我们仍然期望机器最终能够就日志的一种解释达成一致，并且该解释与规约的一种可能执行相匹配。

我们通过首先定义执行过程中的相关事件来解决这一问题，即属于有效日志的事件。在此基础上，我们在日志上建立一种等价关系，从而能够刻画群协议的一种实现作为群执行时所能产生的日志。借助这些工具，我们进一步指出，所有正确的实现都会产生有效日志，并且所有群协议执行都有与之对应、能够实现这些执行的群。

7.1 最终保真性

我们首先引入一些机制，以便精确定义群协议的正确实现这一概念。

粗略地说，可以认为 $(\mathtt{S}\ \epsilon)$ 是群协议 G 的忠实实现，前提是它只产生可由 G生成的全局日志。然而，对于我们的场景而言，这一概念过于严格；事实上，我们采用一种较弱的忠实性概念：对于由 $(\mathtt{S}\ \epsilon)$产生的任意全局日志 $(\mathtt{S}\ \epsilon) \Longrightarrow (\mathtt{S}, l)$，即$l$，都存在一个相关的日志 l'，而 G 允许该日志，即 $(\mathtt{G}\ \epsilon) \Longrightarrow (\mathtt{G}, l')$。我们暂且不形式化定义日志之间预期的关系，而是在下面的示例中阐述一些直观理解。

示例 *7.1.* 考虑示例 4.2中的群协议 G，以及由三个出租车组成的群 $(\mathtt{P}\ \epsilon) \mid (\mathtt{T}\ \epsilon) \mid (\mathtt{T}\ \epsilon) \mid (\mathtt{O}\ \epsilon) \mid (\mathtt{T}\ \epsilon) \mid \epsilon$，它们分别被命名为 A、B 和 C。该群能够生成[6] 全局日志

$$l_{\text{auc}} = \text{请求} \cdot \text{出价}_B \cdot \text{竞价者 ID}_B \cdot \text{出价}_A \cdot \text{竞价者 ID}_A \cdot \text{已选中} \cdot \text{出价}_C \cdot \text{竞价者 ID}_C \cdot \text{乘客 ID}$$

相比之下，G 无法生成这样的日志；事实上，生成前缀 $l_1 = \text{requested} \cdot \text{出价}_B \cdot \text{竞价者id}_B \cdot \text{出价}_A \cdot \text{竞价者id}_A$后，协议后续过程为 $\delta(\mathtt{G}, l_{\text{auc}}) = \mathtt{G}_{\text{Bid}}$；因此，日志 l_1 只能通过追加出价$_C \cdot$ 竞价者id$_C$或 selected $\cdot$ 乘客id 来增长。在第二种情况下，我们得到日志 $l_2 = \text{requested} \cdot \text{出价}_B \cdot \text{竞价者}_B \cdot \text{出价}_A \cdot \text{竞价者}_A \cdot \text{selected} \cdot \text{乘客id}$。值得注意的是，所有机器在处理 l_{auc}时都会丢弃事件出价$_C$和竞价者id$_C$，也就是说，它们的行为就像是在处理 l_2。事实上，$\delta(\mathtt{P}, l_{\text{auc}}) = \delta(\mathtt{P}, l_2)$，$\delta(\mathtt{T}, l_{\text{auc}}) = \delta(\mathtt{T}, l_2)$，以及 $\delta(\mathtt{O}, l_{\text{auc}}) = \delta(\mathtt{O}, l_2)$。$\diamond$

正如前一个示例所强调的，尽管群生成的实际日志与协议生成的日志有所不同，但所有机器都能够在传播完成后，一致地丢弃这些生成错误的事件。换句话说，机器的状态在其中的

[6]有关如何访问此类全局日志的详细信息，请参阅附录 A.3。

群只依赖日志中的部分事件。我们通过一种类型来刻画这一子集，称为有效类型。直观而言，日志的有效类型是子日志的类型，其中包含所有与群中的机器实际相关的事件。给定协议 G 和订阅 σ，我们期望实现只处理某个角色已订阅的事件；因此，我们的有效类型概念是相对于订阅而言的。然而，日志的有效类型并不只是其类型相对于订阅像的投影σ。下面的示例对此进行了说明。

示例 *7.2*（有效类型和投影）。日志的类型l_{auc} 在示例 7.1 是

$$\mathtt{l}_{\text{auc}} = \text{请求} \cdot \text{出价} \cdot \text{出价} \cdot \text{已选中} \cdot \text{出价} \cdot \text{乘客 ID}$$

这与 $\mathtt{l}_2$ 的类型不同，后者为请求 · 出价 · 出价 · 已选中 · 乘客 ID。由于乘客 P 处理 $\mathtt{l}_{\text{auc}}$ 中的所有类型，我们期望定义一个适当的订阅 σ，使得 $\sigma(\mathsf{P}) \supseteq \{$请求, 出价, 已选中, 乘客 ID$\}$。因此，如果我们只保留 $\mathtt{l}_{\text{auc}}$ 中至少有一个角色已订阅的所有类型的子日志，那么得到的恰好是 $\mathtt{l}_{\text{auc}}$，但它并未反映机器实际处理的子日志类型。因此，有效类型也取决于所实现的协议。◇

定义 *7.3*（有效类型）。设 G 为一个群协议，且 σ 为一个订阅。相对于 群日志 l 关于 G 和 σ 的有效类型，记作 $\mathbb{T}_\sigma(l, \mathsf{G}) = \mathbb{T}_\sigma(l, \mathsf{G}\ \epsilon)$，定义如下：

$$\mathbb{T}_\sigma(\epsilon, \mathsf{G}, \mathtt{l}) = \epsilon \tag{3}$$

$$\mathbb{T}_\sigma(e \cdot l, \mathsf{G}, \epsilon) = \mathtt{t} \cdot \mathbb{T}_\sigma(l, \mathsf{G}', \mathtt{l}') \quad \text{if} \quad \vdash e : \mathtt{t}, \mathtt{t} \in \sigma(\mathit{roles}(\mathsf{G}, \sigma)), \mathsf{G} \xrightarrow{\mathsf{c@R}\langle \mathtt{t} \cdot \mathtt{l} \rangle} \mathsf{G}', \text{and } \mathtt{l}' = \mathit{filter}(\mathtt{l}, \sigma(\mathit{active}(\mathsf{G}'))) \tag{4}$$

$$\mathbb{T}_\sigma(e \cdot l, \mathsf{G}, \mathtt{t}.\mathtt{l}) = \mathtt{t} \cdot \mathbb{T}_\sigma(l, \mathsf{G}, \mathtt{l}) \quad \text{if} \quad \vdash e : \mathtt{t} \tag{5}$$

$$\mathbb{T}_\sigma(e \cdot l, \mathsf{G}, \mathtt{l}) = \mathbb{T}_\sigma(l, \mathsf{G}, \mathtt{l}) \quad \text{否则} \tag{6}$$

正如预期，空日志的有效类型就是空日志类型。非空日志的有效类型会记录那些与协议可生成的日志类型相匹配的事件（并丢弃所有顺序错误的事件）。因此，相对于 G，e · l 的有效类型仅当首个事件 e 的类型符合协议预期时，才会记录其类型 t（也就是说，协议中的至少一个角色订阅了该类型）。根据情况 (4)，协议 G 预期某个事件的类型 t 与其某个分支的守卫一致，且至少有一个角色订阅 t。在这种情况下，首先通过消耗类型为 l′ 的事件来处理剩余日志 l 的有效类型（规则 (5)）；这些事件构成该分支生成的剩余类型序列，并由后续过程 G′ 中的活跃角色观察；随后再考虑后续过程 G′。需要指出的是，只有与活跃角色相关的事件类型才会反映在有效类型中（更多详情见第 7.2.1 节）。注意，未被观察到的类型会按照 (6) 直接从有效类型中忽略。对于确定性群协议，$\mathbb{T}_\sigma$(l , G) 是定义良好的函数，因为日志确定性（定义 4.1）确保 G 至多有一个分支能够匹配事件 e 的事件类型 t。

示例 *7.4.* 设 $\mathsf{G}_{\text{auction}}$ 和 $\mathsf{G}_{\text{choose}}$ 为示例 4.2 中定义的 $\mathsf{G}_{\text{bid}} = \text{报价@T}\langle\text{出价} \cdot \text{竞价者 ID}\rangle$。$\mathsf{G}_{\text{auction}}$。我们有

$$\mathsf{G}_{\text{bid}} \xrightarrow{\mathsf{Offer@T}\langle \mathsf{Bid} \cdot \mathsf{BidderID} \rangle} \mathsf{G}_{\text{Auction}} \tag{7}$$

$$\mathsf{G}_{\text{auction}} \xrightarrow{\mathsf{Select@T}\langle \mathsf{Selected} \cdot \mathsf{PassengerID} \rangle} \mathsf{G}_{\text{choose}} \tag{8}$$

让我们计算日志的有效类型 $l = \text{出价}_A \cdot \text{竞价者 ID}_A \cdot \text{评分} \cdot l'$，在 G_{bid} 使用一个订阅 σ，其中 P、T 和 O 订阅了除 评分之外的所有事件。我们有

$$\begin{aligned}\mathbb{T}_\sigma(l, \mathsf{G}_{\text{bid}}) &= \mathsf{Bid} \cdot \mathbb{T}_\sigma(\mathit{bidderid}_A \cdot \mathit{rating} \cdot l', \mathsf{G}_{\text{auction}}, \mathsf{BidderID}) \\ &= \mathsf{Bid} \cdot \mathsf{BidderID} \cdot \mathbb{T}_\sigma(\mathit{rating} \cdot l', \mathsf{G}_{\text{auction}}, \epsilon) \\ &= \mathsf{Bid} \cdot \mathsf{BidderID} \cdot \mathbb{T}_\sigma(l', \mathsf{G}_{\text{auction}}, \epsilon)\end{aligned}$$

其中，第一个等式根据（4）成立，因为 $\mathsf{Bid} \in \sigma(\mathsf{T})$，并根据（7）成立；第二个等式根据（5）成立，因为 $\vdash$ 竞价者 ID_a：竞价者 ID；第三个等式根据（6）成立，因为根据假设，评分 $\notin \sigma(\mathsf{P}) \cup \sigma(\mathsf{T}) \cup \sigma(\mathsf{O})$。如果 $\mathsf{l}' = \epsilon$，则根据（3），$\mathsf{T}_\sigma(\mathsf{l}, \mathsf{G}_{\text{bid}}) =$ 出价 · 竞价者 ID。

反之，假设 $l' = \text{已选中} \cdot \text{乘客 ID} \cdot \text{出价}_B \cdot \text{竞价者 ID}_B$。那么，与上面的前两个等式类似（使用 (8)），我们有

$$\mathbb{T}_\sigma(l', \mathsf{G}_{\text{auction}}, \epsilon) = \mathsf{Selected} \cdot \mathsf{PassengerID} \cdot \mathbb{T}_\sigma(\mathit{bid}_B \cdot \mathit{bidderid}_B, \mathsf{G}_{\text{choose}}, \epsilon)$$

并且，根据（5），$\mathbb{T}_\sigma(\text{出价}_B \cdot \text{竞价者 ID}_B, \mathsf{G}_{\text{choose}}, \epsilon) = \mathbb{T}_\sigma(\epsilon, \mathsf{G}_{\text{choose}}, \epsilon) = \epsilon$。 ◇

我们需要的实现日志与规约日志之间的关系，是由它们的有效类型相等所导出的等价关系。

定义 7.5（日志等价）。对于群协议 G 和订阅 σ，如果日志 l 和 l′ 具有相同的有效类型，则称它们等价，记作 $\mathsf{l} \equiv_{\mathsf{G},\sigma} \mathsf{l}'$；这里的有效类型是相对于 G 和 σ 而言的，即 $\mathsf{T}_\sigma(\mathsf{l}, \mathsf{G}) = \mathsf{T}_\sigma(\mathsf{l}', \mathsf{G})$。

那么，正确实现的概念可以简单地表述如下。

定义 7.6（最终保真性）。若群 $(\mathsf{S}\ \epsilon)$ 对于群协议 G 和订阅 σ 最终忠实，则对于任意 $(\mathsf{S}\ \epsilon) \Longrightarrow (\mathsf{S}, \mathsf{l})$，都存在 l′，使得 $(\mathsf{G}\ \epsilon) \Longrightarrow (\mathsf{G}, \mathsf{l}')$，且 $\mathsf{l} \equiv_{\mathsf{G},\sigma} \mathsf{l}'$。

7.2 通过投影实现正确性

我们的投影操作为从良构群协议中获得正确实现提供了一种有效的方法，我们将这些实现称为实现。

定义 7.7（实现）。设 G 为群协议，σ 为订阅。相对于 σ，G 的（规模为 n 的）实现（简称为 (σ, G)-实现）是一个规模为 n 的群 $(\mathsf{S}\ \epsilon)$，满足对于每个 $1 \le i \le n$，都存在一个角色 $\mathsf{R} \in$ 属于 $\mathsf{roles}(\mathsf{G}\ \sigma)$，使得 $\mathsf{S}(i) = (\mathsf{G} \downarrow^{\sigma}_{\mathsf{R}}, \epsilon)$。若对于所有 $\mathsf{R} \in$ 属于 $\mathsf{roles}(\mathsf{G}\ \sigma)$，都存在 $1 \le i \le n$ 使得 $\mathsf{S}(i) = (\mathsf{G} \downarrow^{\sigma}_{\mathsf{R}}, \epsilon)$，则称实现 S 是完整的；不完整的实现称为部分实现。

值得注意的是，实现中的机器数量与相应群协议中的角色数量无关。事实上，定义 7.7 仅要求群中的每台机器都扮演群协议中的某个角色。具体而言，我们可以有多个组件实现同一个角色（即该角色被复制），也可以存在没有对应机器的角色，即部分实现。

示例 *7.8*（实现）。示例4.2中定义的贯穿示例的群协议，通常会由一台扮演乘客 $\mathsf{G} \downarrow^{\sigma}_{\mathsf{P}}$ 角色的机器、若干台运行机器 $\mathsf{G} \downarrow^{\sigma}_{\mathsf{T}}$ 的出租车，以及至少一间运行 $\mathsf{G} \downarrow^{\sigma}_{\mathsf{O}}$ 的会计办公室来实现。没有会计办公室的实现则属于部分实现，在这种情况下，没有机器能够生成收据事件。◇

本节余下部分将说明，如果实现（无论完整还是部分）是通过投影良构群协议得到的，那么它们最终忠实。

7.2.1 投影与有效类型。我们首先确立单个投影的行为与相应协议行为之间的对应关系。特别是，我们将证明，有效类型能够准确抽象日志中与某个角色相关的信息。具体而言，下一个结果指出，只有在处理日志之后，投影才会启用某个命令l；而协议只有在生成等价日志 l' 后，才会启用同一命令。

引理 *7.9.* 若 *G* 是一个 σ-*WF* 群协议，且 $(\delta(G \downarrow_{\mathrm{R}}^{\sigma}, l))\downarrow_{\mathrm{c}}/l$，则存在 $l' \equiv_{\mathrm{G},\sigma} l$，使得 $(G\ \epsilon) \Longrightarrow (G, l')$ 且 $\delta(G, l')\ \mathrm{c@R}\ \mathrm{hl}\ i \longrightarrow G'$。

人们可能会认为，对于一台机器来说，等价的日志是不可区分的，即 $l \equiv_{\mathrm{G},\sigma} l'$意味着 $\delta(\mathrm{G} \downarrow_{\mathrm{R}}^{\sigma}, l) = \delta(\mathrm{G} \downarrow_{\mathrm{R}}^{\sigma}, l')$。然而，如果日志不完整，情况可能就并非如此；所谓不完整，是指日志未包含同一命令生成的所有事件。

示例 7.10（日志等价并不意味着状态不可区分）。考虑群协议 $\mathrm{G} = \mathrm{c@Rha} \cdot \mathrm{bi}\,.\,\mathrm{d@Shci}\,.\,0$，其中 $\sigma = \{\mathrm{R} \mapsto \{\mathrm{a}, \mathrm{b}\},\ \mathrm{S} \mapsto \{\mathrm{a}, \mathrm{c}\}\}$。如果 $\vdash \mathrm{a} : \mathrm{a}$ 且 $\vdash \mathrm{b} : \mathrm{b}$，那么 $\mathrm{T}_{\sigma}(\mathrm{a}, \mathrm{G}) = \mathrm{a}$，且 $\mathrm{T}_{\sigma}(\mathrm{a} \cdot \mathrm{b}, \mathrm{G}) = \mathrm{a}$；事实上，第一个等式根据定义成立，而第二个等式成立是因为 $\mathrm{b} \notin \sigma$(活跃(d@Shci . 0′)) $= \sigma(\mathrm{S}) = \{\mathrm{a}, \mathrm{c}\}$。因此，$\mathrm{a} \equiv_{\mathrm{G},\sigma} \mathrm{a} \cdot \mathrm{b}$。现在，考虑 G 关于 R 的投影，并以 σ 为依据，即 $\mathrm{M}_{\mathrm{R}} = \mathrm{G} \downarrow_{\mathrm{R}}^{\sigma} = \mathrm{a?\ b?\ (d@Shci}\ .\ 0 \downarrow_{\mathrm{R}}^{\sigma}) = \mathrm{a?\ b?\ 0}$。显然$\delta(\mathrm{M}_{\mathrm{R}}, \mathrm{a} \cdot \mathrm{b}) \neq \delta(\mathrm{M}_{\mathrm{R}}, \mathrm{a})$。⋄

关于示例 7.10 有两点值得考虑。一方面，虽然 $a \cdot b$ 包含执行命令 c所产生的所有事件，但仅由事件 a 构成的日志并非如此。由于我们假设所有事件最终都会传播，因此下一节的技术论述将忽略不完整的（全局）日志。另一方面，人们可能会对有效类型不收集活跃角色未观察到的事件信息这一点感到疑惑。这对于说明实现可能插入事件这一事实至关重要。例如，实现实际上可能生成类型为 $\mathrm{a} \cdot \mathrm{c} \cdot \mathrm{b}$ 的日志，因为实现角色 S 的机器可能会在处理类型为 d 的命令时，一旦处理了类型为 a的事件便立即执行；因此，在合并后的日志中，所生成的事件可能先于类型为 b 的事件。如果我们的日志等价概念足够精细，能够区分类型为 $\mathrm{a} \cdot \mathrm{c} \cdot \mathrm{b}$ 与 $\mathrm{a} \cdot \mathrm{b} \cdot \mathrm{c}$ 的日志，那么我们就会排除上述行为的实现；但这并非我们所希望的，因为这种交互并未违反协议。

7.2.2 协议所允许的日志的刻画。现在，我们将刻画良构群协议的实现能够生成的日志。为此，我们必须考虑忠实实现协议的机器可能产生的重排和伪事件。直观地说，我们可以认为，实现生成的日志对应于协议多个执行的组合，这些执行可能共享一个公共前缀。再次考虑示例 4.2 中的群协议 G。如示例 7.1 中所述，我们预期实现能够生成该示例中的日志 $\mathrm{l}_{\mathrm{auc}}$。请注意，这种日志可以通过合并 G 的多个不同归约来生成，例如 $(\mathrm{G}\ \epsilon) \Longrightarrow (\mathrm{G}, \mathrm{requested} \cdot \mathrm{bid}_B \cdot \mathrm{bidderid}_B \cdot \mathrm{bid}_A \cdot \mathrm{bidder}_A \cdot \mathrm{selected} \cdot \mathrm{passengerid})$ 和 $(\mathrm{G}\ \epsilon) \Longrightarrow (\mathrm{G}, \mathrm{requested} \cdot \mathrm{bid}_B \cdot \mathrm{bidder}_B \cdot \mathrm{bid}_C \cdot \mathrm{bidderid}_C)$。请注意，这些归约共享事件（对应于计算发生分歧的情形）。直观地说，两个运行可以组合，当且仅当它们产生互不相交的日志，或者共享来自某个公共执行前缀的事件（如前例所示）。形式上，若存在日志 $\mathrm{l}, \mathrm{l}'_1 \cap \mathrm{l}'_2 = \emptyset$，使得 $\mathrm{l}_i = \mathrm{l} \cdot \mathrm{l}'_i$ 且 $(\mathrm{G}\ \epsilon) \Longrightarrow (\mathrm{G}, \mathrm{l}) \Longrightarrow (\mathrm{G}, \mathrm{l} \cdot \mathrm{l}'_i)$，则两个运行 $(\mathrm{G}\ \epsilon) \Longrightarrow (\mathrm{G}, \mathrm{l}_i)$ with $i = 1, 2$ 是一致的。通过要求两两一致，可将一致性的概念推广到运行集合 $\{(\mathrm{G}\ \epsilon) \Longrightarrow (\mathrm{G}, \mathrm{l}_i)\}_{1 \le i \le k}$。我们用 $\mathrm{l}\,{}_i^{\,j}$ 表示归约 i中第 j 步产生的事件序列，即 $(\mathrm{G}\ \epsilon) \Longrightarrow^{j-1} (\mathrm{G}, \mathrm{l}'_i)\ \mathrm{c}\ /\ \mathrm{l} \longrightarrow (\mathrm{G}, \mathrm{l}'_i \cdot \mathrm{l}\,{}_i^{\,j}) \Longrightarrow (\mathrm{G}, \mathrm{l}_i)$。

定义 7.11（可容许日志）。若存在一致运行集合 $\{(G\ \epsilon) \Longrightarrow (G, l_i)\}_{1\le i\le k}$ 以及日志 $l' \in (\bowtie_{1\le i\le k} l_i)$，使得 $l \equiv_{G,\sigma} l'$，$l = \text{Đ}_{1\le i\le k} l_i$，并且对于所有 $1 \le i \le k$ 以及 $l \overset{j}{i} \sqsubseteq l$，所有由归约 i 中第 j 步产生的事件 $l \overset{j}{i}$，则日志 l 是 σ-WF 协议 G 的可容许日志。

值得注意的是，可容许日志并不只是那些能够通过合并若干日志 l_i 得到的日志；也可能出现 l 可容许但 $l \notin (\bowtie_{1\le i\le k} l_i)$ 的情况。事实上，这一概念的限制更弱，并将那些不会改变日志有效类型的事件可能重排纳入考量。考虑示例 7.10 中的协议及其由两台机器组成的完全实现。如前所述，该实现可能生成类型为 $a \cdot c \cdot b$ 的日志。只运行一次协议，即固定 $k = 1$ 并取 $(G\ \epsilon)\ c / a \cdot b \longrightarrow (G, a \cdot b)\ d / c \longrightarrow (G, l\ 1)$，其中 $l_1 = a \cdot b \cdot c$，我们便可得出结论：某个实现会生成 $a \cdot c \cdot b$。注意，$\bowtie_{1\le i\le 1} l_i = \{l_1\} = \{a \cdot b \cdot c\}$。因此，$l' \in \bowtie_{i\le 1\le 1} l_i$ 当且仅当 $l' = a \cdot b \cdot c$ 且 $T_\sigma(l', G) = a \cdot c$。于是，日志 $a \cdot c \cdot b$ 与之等价（即有效类型相同），而且其元素相同，并保留了由同一命令生成的事件之间的相对顺序（即 a 先于 b）。因此，我们得出结论，该协议允许日志 $a \cdot c \cdot b$。相反，最后一项条件 $l \overset{j}{i} \sqsubseteq l$ 要求保留由同一命令生成的事件之间的相对顺序，因此排除了 $b \cdot a \cdot c$ 这样的日志。

以下结果确保，良构协议的任何可容许日志都等价于通过顺序执行该协议所获得的日志。换言之，尽管日志可能包含由相互冲突的决策产生的事件，但其有效类型对应于顺序运行。

引理 7.12。若 l 是一个 σ-*WF* 群协议 G 的可容许日志，则存在一个日志 l' 使得 $(G\ \epsilon) \Longrightarrow (G,\ l')$ 并且 $l \equiv_{G,\ \sigma} l'$。

此外，如果我们将由在全局日志的部分视图上启用的命令执行所生成的事件添加到可容许日志中，就会得到一个可容许日志。这一性质对于下一节的主要结果（定理7.14）至关重要。

引理 *7.13*。设 $l\ 1$ 和 $l_2 \subseteq l\ 1$ 是 σ-*WF* 群协议 *G* 的可容许日志。如果 *(G, l 2) c@R hl i* $\longrightarrow$ *(G, l 2 · l 3)*，且 $l \in l_1 \bowtie (l_2 \cdot l_3)$，那么 l 对 *G* 是可容许的。

7.2.3 实现是忠实的。现在我们来说明主要结果：良构协议的实现只会生成可容许日志。

定理 7.14。设 $(S\ \epsilon)$ 是 σ-*WF* 群协议 G 的一个实现。若 $(S\ \epsilon) \Longrightarrow (S',\ l)$ 则 l 对 G 是可容许的。

由于每个可容许日志都等价于协议生成的日志（引理7.12），我们可以得出结论：良构群协议的任何实现最终都是忠实的（即正确的）。请注意，这意味着良构群协议的所有实现最终都会就其选择中采取哪个分支达成共识 [39]（具体而言，经过一个 $\xrightarrow{\tau}$ 步骤后，所有机器在其状态计算中都会采取同一分支）。

推论 7.15。每个大小为 n 的 σ-*WF* 群协议 G 的实现相对于 G 和 σ 都是最终忠实的。

上述结果与实现每个角色的副本数量无关；对于部分实现（即某些角色缺失时）也同样成立。

7.3 实现完整性

与常见的会话类型系统不同，完全实现（即每个角色都得到实现的实现）的行为相对于协议是完备的，也就是说，协议的每次归约都可以由该实现模拟。这源于这样一个事实：我们模型中的非确定性来自外部命令的执行，而非对内部（通常是确定的）选择进行抽象。首先，我们注意到，按照协议顺序生成的日志会将机器驱动至相应状态。

引理 7.16。 如果 σ-*WF* 群协议，且 $(\mathsf{G}\ \epsilon) \Longrightarrow (\mathsf{G},\ \ l)$ 那么 $\delta(\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}},\ \ l) = \delta(\mathsf{G},\ \ l)\downarrow^{\sigma}_{\mathtt{R}}$ 对于所有 $\mathtt{R} \in 角色(\mathsf{G}\ \sigma)$。

此外，命题 7.17 指出，完全实现能够生成由协议生成的日志（该结果通过使用先前的结果，并在机器执行命令后立即将所有事件传播到所有副本而得出）。

命题 7.17。 设 $(\mathtt{S}\ \epsilon)$ 是大小为 n 的 σ-*WF* 群协议 G的完全实现。如果$(\mathsf{G}\ \epsilon) \Longrightarrow (\mathsf{G},\ \ l)$ 那么存在一个群 $\mathtt{S}'$ 使得 $(\mathtt{S}\ \epsilon) \Longrightarrow (\mathtt{S}',\ \ l)$。

7.4 死锁、活性和安全性

根据我们对群协议的余归纳定义 $\mathsf{G} = \sum_{i\in I} \mathtt{c}_i@\mathtt{R}_i\langle \mathtt{l}_i\rangle.\,\mathsf{G}_i$ 显然，每个作为 G' 子项的协议状态 G 都可以通过适当的命令调用序列到达。因此，完全实现——经过充分的事件传播后，我们便可以调用任何已启用的命令——能够到达与 G' 对应的局部机器状态，如命题 7.17所述。一般而言，活性在我们的理论中并无保证，因为该理论允许状态循环被无限次访问。我们猜想，采用适当的公平性概念这一常见方法或许能够保证活性。为我们的系统找到此类条件仍是一个开放问题。

假设事件最终会传播（即 $\xrightarrow{\tau}$最终会耗尽），并且每台有已启用命令的机器最终都会选择调用其中一个命令，那么只要群协议允许（即有效全局日志所诱导的群协议状态启用了命令），进展显然就会持续发生。换句话说，我们的群体是无死锁的。

我们采取了不同于会话类型约束的方法，刻意允许孤立消息——我们的事件处理语义确保接收方会丢弃意外消息。因此，结合我们的良构条件，我们确保每台机器只处理其根据全局协议忠实取得进展所需的消息；只要事件仍在传播，这些消息就由其部分视图详细规定。

8 相关工作

人们普遍认为，分布式协调问题的解决方案高度依赖于所采用的计算模型 [2, 16, 22]。我们的方案建立在本地优先协作 [26, 27]的原则之上。这一架构的关键在于群中每个参与节点的自主性。自主性使每个节点能够独立于网络连接、其他节点的可用性或通信延迟而取得进展。我们的目标模型具有区别于其他行为类型系统的鲜明特征。在我们的情境中，分布式异构组件根据给定协议中规定的角色（例如贯穿示例中的乘客和出租车），通过异步发出和消费事件进行交互。更确切地说，事件是组件以非确定方式执行命令所产生的副作用；每个组件都会在本地记录事件，并将其异步传播到群中。至关重要的是，我们不对通信的相对速度作任何假设，而只

要求日志最终就事件顺序达成一致 [4]。这种宽松的设置允许存在不一致：组件可能做出不一致的决策，从而损害协议的执行，并表现出在强一致性模型中被排除的行为。我们的方法属于与数据复制相关的方法，而这类方法一向十分复杂。事实上，标准技术必须在可用性、一致性和分区容错性之间进行权衡[17]。人们已经提出了多种技术，例如无冲突复制数据类型 [34], 云类型 [5], 一致性契约 [35], 不变量 [3, 19, 25], 线性一致性 [41], 以及适用于 GSP 等应用的操作模型 [6, 18]。我们方法的一个独特方面是，利用行为类型来约束数据复制，以最终实现一致性。我们关注每个节点仅依据本地信息做出决策的能力所带来的后果。

我们的方案受到开创性研究中提出的编排框架的启发，该研究聚焦于多方会话类型 [20, 21]。然而，我们的执行模型具有独特之处，且目标属性也有所不同，因此必须彻底改变全局类型的一些成熟概念的定义，例如投影和良构性。我们方法的主要创新在于：只要最终能够达成一致，组件就可以沿着分布式选择所提供的多个（可能彼此不一致的）分支进行推测性执行。直观而言，当本地日志“整合”时，也就是相关事件已传播到所有相关组件时，系统会忽略除一条执行路径之外的所有执行路径。据我们所知，大多数行为类型系统的良构条件都禁止存在多个选择器[22]。文献 [24] 对这一条件略作放宽，但其中的条件仍会排除示例1.1中的协议。值得注意的是，在所追求的性质方面，我们偏离了行为类型的研究路径。我们的目标是保证全局规约的投影能够产生群体的实现，并使其最终对分布式执行达成一致视图，即使期间存在暂时的不一致。这不同于旨在实现无（死）锁或某种进展性质的行为类型系统（参见 [22] 和 [9] ，其中近期讨论了二元情形）。

其次，我们的行为规范完全抽象了执行协议中每个角色的实例数量。多方会话类型通常并非如此。参数化多方会话类型已在 [11, 12, 42] 中被研究，最近也在 [7, 24]中得到研究。这些提案旨在捕捉这样一个事实：协议中的角色彼此“连接”以形成一种拓扑，该拓扑可以被泛化（例如，通过其大小参数化环形拓扑）。因此，这些行为类型系统需要显式处理协议的参数。相反，我们的规约完全不考虑此类参数。据我们所知，多方会话类型一直专注于点对点的消息传递通信模型，即使是在处理高度动态的场景时也是如此，例如涉及机器人协调的场景 [29]。

9 最后总结

这里提出的行为类型相当非传统。我们偏离了在点对点消息传递通信之上实现通信协议这一默认选择；这种做法十分普遍（例如参见 [13, 22, 40]）。在我们的设定中，组件通过共享日志进行交互，而该日志以分布式方式构建，无需任何额外的协调机制。更准确地说，每个组件都维护着对全局日志的本地视图，该视图可能不完整且彼此不一致。组件仅凭这一视图，就可以执行一个会立即影响其本地状态的操作；随后，这些影响会异步传播到系统的其余部分。这意味着组件可能执行全局无效的操作（只要这些操作在本地有效即可），但我们要求协议中的每个组件都能够识别这些操作，并最终表现得仿佛只执行了有效操作。从技术上说，这意味着我们放弃了会话保真性等既有性质，转而关注系统在没有专门协调的情况下最终达成一致的能力；不过，我们确实指出，我们的类型纪律能够保证无死锁性。

我们的目标应用本质上涉及组件集合，而其数量在静态时未知：组件可能动态加入或离开交互的执行。按照行为类型中的常见做法，我们用组件可能扮演的角色来描述协议。与大多数行为类型不同，我们的行为类型不受每个角色对应的实例数量限制。我们假设，任何群协议中的任何角色都可以根据需要复制任意多个实例。这一选择也影响了对选择的解释。在标准方法中，每个选择都分配给由单个组件实现的单一角色，该组件负责协调决策。当一个角色的实现可以被复制时，这种做法就会产生问题；如果副本的状态还可能彼此不一致，问题就更加严重：不同组件可能会作出不同决定。因此，我们的群协议中的选择旨在由可能实现不同角色的组件分布式地解决。我们的解决方案基于推测计算：不同的选择可以并发进行，直到组件能够通过检查各自的本地状态，就依据所有事件的全序所选中的分支达成一致（例如，可以通过 Lamport 时间戳和唯一节点标识符来实现该全序）。

本文提出的计算模型对现有基础设施进行了抽象描述 [1], , 其中机器实际上是用主流编程语言（例如 `TypeScript`）编写的程序。本文介绍的机器在此意义上扮演局部类型的角色，用于描述各组件的预期行为。

通常很难确定合适的订阅。我们推测，可以利用群协议本身推断出合适的“最小”注册信息，并建议设计人员采用，以确保良构性。随后可以手动细化此类订阅，但须保持良构性。此外，我们可以设想由程序员仅指定需要传输的相关信息，再自动推断出协调选择所需的事件（这与会话类型中标签的通信方式颇为相似）。

关于推测性执行，一个基本假设是，执行无效操作所产生的影响可以被丢弃。换言之，无效操作不会造成后果。在某些情况下，这可能无法接受。群协议可用于系统地识别此类情况——例如，注意机器何时忽略了相应事件——并在应用层面采取有原则的处理方式。我们计划研究适用于投影的扩展，以自动注入执行修正操作所需的行为。或者，我们也将探索配备消毒器的监控方法，由其负责进行补偿。

我们只部分讨论了模型中的故障：虽然我们确实对暂时无法接收（这会阻止事件传播转移 [传播规则]）或无法运行（即阻止命令调用 [局部]）进行了建模，但没有对永久无法发送进行建模。发生停止故障时，某台机器可能会传播选择中的第一个事件，却无法继续传播命令调用所产生的所有预期后续事件，从而导致系统陷入停滞。一种解决方法是，只有当本地日志中包含所有指定事件时，才执行外部选择，这样群便能永久丢弃由故障机器作出的选择；在机器语义的定义中，可以通过接收日志而非单个事件来表达这一点。

第2.3节中提到的另一项值得进一步研究的扩展是：如果选择中的第一个事件不仅决定要采用哪个分支，还决定从哪个源机器消费该选择中其余的事件，那么这样做是合理的，也能实现一种针对每个选择的非干扰概念。这将进一步强化上述概述的故障处理机制，确保始终丢弃来自故障机器的输入。刻画此类方案所能提供的精确保证，将是一个值得进一步研究的课题。

参考文献

[1] Actyx AG。2020–2022。Actyx 开发者网站。https://developer.actyx.com，访问日期：2022/07/06。[2] Davide Ancona、Viviana Bono、Mario Bravetti、Joana Campos、Giuseppe Castagna、Pierre-Malo Deniélou、Simon J Gay、Nils Gesbert、Elena Giachino、Raymond Hu 等。2016。编程语言中的行为类型。《编程语言前沿与趋势》3, 2-3 (2016), 95–230。[3] Valter Balegas、Sérgio Duarte、Carla Ferreira、Rodrigo Rodrigues 和 Nuno M. Preguiça。2018。IPA：面向弱一致性复制数据库的保持不变应用。《VLDB 会议录》12, 4 (2018), 404–418。https://doi.org/10 14778 3297753 3297760 . / . Sebastian Burckhardt。.《最终一致性原理》。《编程语言前沿与趋势》1, 1–2（2014年10月），1–150。2014 https://doi.org/10 1561 2500000011 . / Sebastian Burckhardt、Manuel Fähndrich、Daan Leijen 和 Benjamin P. Wood。[5] 2012。面向最终一致性的云类型。载于《ECOOP 2012——面向对象编程》，James Noble（编）。Springer，柏林，海德堡，283–307。[6] Sebastian Burckhardt、Daan Leijen、Jonathan Protzenko 和 Manuel Fähndrich。2015。全局序列协议：复制共享状态的健壮抽象。载于第29届欧洲面向对象编程会议（ECOOP 2015）（《莱布尼茨国际信息学会议论文集》（LIPIcs），第37卷），John Tang Boyland（编）。Schloss Dagstuhl– Leibniz-Zentrum fuer Informatik，德国 Dagstuhl，568 590 10 4230 2015 568 – 。[7] 2019 https://doi.org/ 。/LIPIcs.ECOOP. 。David Castro-Perez、Raymond Hu、Sung-Shik Jongmans、Nicholas Ng 和 Nobuko Yoshida。 。在 Go 中使用角色参数化会话类型进行分布式编程：面向动态实例化通信结构的静态类型端点 API。《ACM 程序设计语言会议录》3, 2019 29 1 29 30 10 1145 3290342 POPL（ ），： – ：。https://doi.org/ 。/ Bruno Courcelle。 。《无限树的基本性质》。《理论计算机科学》（ ），25 1983 –169。[8] 1983 95 https://doi.org/10 1016 0304 3975 83 90059 2 . / - ( ) - Ornela Dardha 和 Jorge A. Pérez。 。死锁自由类型系统的比较。《逻辑、代数与程序设计方法杂志》[9] 2022 124 2022 100717 10 1016 2021 100717 ( )， 。https://doi.org/ . /j.jlamp. 。Robert De Simone。 。《无限语言与混合积》。《理论计算机科学》31, 1-2 (1984), 83–100。[10] 1984 [11] Pierre-Malo Deniélou 和 Nobuko Yoshida。2011。动态多角色会话类型。载于 POPL，Thomas Ball 和 Mooly Sagiv（编）。ACM，435 446 10 1145 1926385 1926435 – 。https://doi.org/ 。/ 。Pierre-Malo Deniélou、Nobuko Yoshida、Andi Bejleri 和 Raymond Hu。2012。参数化多方会话类型。《逻辑方法与计算机科学》[12] 8, 4 2012 10 2168 8 4 6 2012 ( )。https://doi.org/ 。/LMCS- (:) Mariangiola Dezani-Ciancaglini 和 Ugo de’Liguoro。 。会话与会话类型：概述。载于 WS-FM（LNCS，第6194卷）。Springer，1–28。[14] Michael J. Fischer、Nancy A. Lynch 和 Michael S. Paterson。1985。一个进程发生故障时分布式共识的不可能性。《ACM学报》[13] 2009 32, 2 1985 374 382 10 1145 3149 214121 (apr)， – 。https://doi.org/ . / . Vijay K Garg 和 MT Ragunath。 。并发正则表达式及其与佩特里网的关系。《理论计算机科学》96, 2 (1992), 285–304。[16] Simon Gay 和 Antonio Ravara（编）。2009。《行为类型：从理论到工具》。River。[17] Seth Gilbert 和 Nancy Lynch。2002。布鲁尔猜想与一致性、可用性及分区容忍的 Web 服务的可行性。《SIGACT 新闻》[15] 1992 33, 2 2002 51 59 10 1145 564585 564601 (June)， – 。https://doi.org/ . / . Alexey Gotsman 和 Sebastian Burckhardt。[18] 。具有全局操作排序的一致性模型及其组合。载于第31届分布式计算国际研讨会（DISC 2017）（《莱布尼茨国际信息学会议论文集》（LIPIcs），第91卷），Andréa W. Richa（编）。Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik，德国 Dagstuhl，2017 23 1 23 16 10 4230 2017 23 : – : 。https://doi.org/ 。/LIPIcs.DISC. 。Alexey Gotsman、Hongseok Yang、Carla Ferreira、Mahsa Najafzadeh 和 Marc Shapiro。2016。“因为我足够强大”：分布式系统中一致性选择的推理。载于 POPL 2016。371–384。[20] Kohei Honda、Nobuko Yoshida 和 Marco Carbone。2008。多方异步会话类型。载于第35届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集。273–284。[21] Kohei Honda、Nobuko Yoshida 和 Marco Carbone。2016。多方异步会话类型。《ACM学报》（JACM）63, 1 (2016), 1–67。[22] Hans Hüttel、Ivan Lanese、Vasco T. Vasconcelos、Luís Caires、Marco Carbone、Pierre-Malo Deniélou、Dimitris Mostrous、Luca Padovani、António Ravara、Emilio Tuosto、Hugo Torres Vieira 和 Gianluigi Zavattaro。2016。《会话类型与行为契约基础》。《ACM计算调查》49, 1 (2016), 3:1–3:36。[23] David R Jefferson。1985。虚拟时间。《ACM编程语言与系统汇刊》（TOPLAS）7, 3 (1985), 404–425。[24] Sung-Shik Jongmans 和 Nobuko Yoshida。2020。探索类型级互模拟，以实现表达力更强的多方会话类型。载于《编程语言与系统》，Peter Müller（编）。Springer International Publishing，Cham，251–279。

[25] Gowtham Kaki、Kapil Earanky、KC Sivaramakrishnan 和 Suresh Jagannathan。2018。通过有界并发验证实现安全复制。《ACM 编程语言汇刊》2，OOPSLA（2018），1–27。[26] Martin Kleppmann、Adam Wiggins、Peter van Hardenberg 和 Mark McGranaghan。2019。本地优先软件：即使有云端，数据仍归你所有。载于《2019 年 ACM SIGPLAN 编程与软件新思想、新范式与反思国际研讨会论文集》（希腊雅典）（Onward! 2019）。计算机协会，纽约，纽约州，美国，

154 178 10 1145 3359591 3359737 – 。https://doi.org/ 。/ 。

[27] Roland Kuhn。 。2021。本地优先协作：边缘自主性，由密码技术保障，100% 可用。https://www.infoq.com/articles/local-first-cooperation/，访问于 2021/06/20。[28] Leslie Lamport。2019。分布式系统中的时间、时钟与事件排序。计算机协会，纽约，纽约州，美国，

179 196 10 1145 3335772 3335934 – 。https://doi.org/ 。/ 。

[29] Rupak Majumdar、Marcus Pirron、Nobuko Yoshida 和 Damien Zufferey。2019。 。用于机器人交互的运动会话类型（前沿新想法论文）。载于第 33 届欧洲面向对象编程会议（ECOOP 2019）。Schloss Dagstuhl-Leibniz 信息学中心。[30] Microsoft。2012-2023。TypeScript：带类型语法的 JavaScript。https://www.typescriptlang.org/，访问于 2023/02/02。[31] Benjamin C. Pierce。2002。《类型与编程语言》。MIT 出版社。[32] Fred B Schneider。1990。使用状态机方法实现容错服务：教程。《ACM 计算调查》（CSUR）22，4（1990），299–319。[33] Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。2011。无冲突复制数据类型。载于《分布式系统的稳定性、安全性与保障》。Springer，386–400。[34] Marc Shapiro、Nuno M. Preguiça、Carlos Baquero 和 Marek Zawirski。2011。无冲突复制数据类型。载于 SSS *2011* 386 400 10 1007 978 3 642 24550 3_29 。– 。https://doi.org/ 。/ - - - - KC Sivaramakrishnan、[35] Gowtham Kaki 和 Suresh Jagannathan。2015。在最终一致的数据存储上进行声明式编程。载于 PLDI 2015。ACM，413–424。[36] Martin Sulzmann 和 Peter Thiemann。2019。正则交错表达式的导数与偏导数。《计算机系统科学杂志》104（2019），323–341。[37] Ecma TC39。2017。ECMA-404。https://www.ecma-international.org/publications-and-standards/standards/ecma-404/，访问于 2023/02/13，替代标准 RFC8259。[38] Ecma TC39。2022。装饰器。https://github.com/tc39/proposal-decorators，访问于 2023/02/02。[39] Lewis Tseng。2019。最终共识：在存储与区块链中的应用：（扩展摘要）。载于 2019 年第 57 届 Allerton 通信、控制与计算年会（Allerton）。840–846。https://doi.org/

10 1109 2019 8919675 。/ALLERTON。 。Vasco T. Vasconcelos。 。会话类型基础。《信息与计算》217（2012），52–70。https://doi.org/ 10 1016 2012 05 002 。/j.ic. 。 。Chao Wang、[40] Constantin Enea、Suha Orhun Mutluergil 和 Gustavo Petri。2019。面向复制的线性一致性。载于第 40 届 ACM SIGPLAN 编程语言设计与实现会议论文集。980–993。[41] Nobuko Yoshida、Pierre-Malo Deniélou、Andi Bejleri 和 Raymond Hu。2010。参数化多方会话类型。载于《软件科学与计算结构基础：第 13 届国际会议 FOSSACS 2010，作为软件理论与实践联合欧洲会议 ETAPS 2010 的一部分于 2010 年 3 月 20—28 日在塞浦路斯帕福斯举行。论文集》（计算机科学讲义，第 6014 卷），C.-H. Luke Ong（编）。Springer，128–145。https://doi.org/10.1007/978-3-642-12032-9_10

A 补充材料

A.1 机器语法

全文中，我们固定一个事件集合 E（变量为 e、e_1,e2、.....），并将日志 l 定义为一个不重复事件组成的序列；我们用 L 表示日志集合。我们假设存在一个可判定类型关系 $\vdash e : t$，它为 e 唯一指定类型 t，其中 T 为事件类型集合。因此，可以通过指定日志类型 $l \in T^\star$（即事件类型序列）为日志赋予类型。我们将序列视为其元素所构成的集合；例如，当 e 出现在 l 中时，我们写作 $e \in l$。对于自然数 n，我们令 $\underline{n}$ 表示集合 $\{1, \ldots, n\}$（注意 $\underline{0} = \emptyset$）。下文中我们只考虑确定性机器。

定义 *A.1*（确定性）。机器 $M = \kappa\cdot[t_1?\, M_1 \,\&\, \cdots \,\&\, t_n?\, M_n]$ 是确定的，如果不同分支之间的事件类型两两不同，即 $t_j \neq t_k$ 对所有 $j \neq k \in \underline{n}$，并且 M_i 对所有 $i \in n$ 都是确定的。确定性机器的类记作 $\mathcal{M}$。

我们将把机器指定为有限方程组 $M = \ldots$，其中右侧出现的 M 均受前缀守卫。满足此类方程的正则树保证唯一，并对应一个有限自动机 [8]。我们的文本表示法等价于正文中使用的基于自动机的表示法[7]；我们有时会利用这一点，因为它既有助于可视化，也有助于推理机器性质的可判定性和复杂性。我们有时也会将 M 称为有限状态自动机的状态。

机器的行为由其对执行的部分视图决定，即其本地日志。因此，操作语义基于日志处理。我们令 守卫(M) 为机器 M 在其初始状态下处理的事件类型集合：

$$guards(M) = \{t_1, \ldots, t_n\} \quad \text{for} \quad M = \kappa\cdot[t_1?\, M_1 \;\&\; \cdots \;\&\; t_n?\, M_n]$$

机器的当前状态 $\delta(M, l)$ 是指机器 M 相对于日志 l 的状态，由一个映射 δ 给出，该映射返回机器 M' 在消耗 l 中的事件后所达到的状态。形式化地，

$$\delta(M, \epsilon) = M \tag{9}$$

$$\delta(M, e \cdot l) = \begin{cases} \delta(M, l) & \text{if } \vdash e : t \text{ and } t \notin guards(M) \\ \delta(M_j, l) & \text{if } \vdash e : t_j \text{ and } M = \kappa\cdot\left[t_1?\, M_1 \;\&\; \ldots \;\&\; t_j?\, M_j \;\&\; \ldots \;\&\; t_n?\, M_n\right] \end{cases} \tag{10}$$

这是一个简单的观察：δ 对于确定性机器而言，这是一个定义良好的函数。我们的状态转移函数有一个特点：当事件与当前状态中的任何守卫都不匹配时，它会忽略这些事件。

机器的能力 M 执行命令 c 并发出类型为 l 的非空事件序列，记作 $(M)\downarrow_{c\,/\,l}$；如果 c 在 M 处已启用，则称 $(M)\downarrow_{c\,/\,l}$ 成立。对于机器 $M = \kappa\cdot[\ldots]$，当且仅当 $\kappa(c) = l$ 成立时，情况便是如此。命令调用（在我们的模型中属于一种外部刺激）遵循以下规则：

$$\frac{\delta(M, l) = M' \qquad (M')\downarrow_{c\,/\,l} \qquad \vdash l' : l \qquad l' \in \mathcal{L} \text{ fresh}}{(M, l) \xrightarrow{c\,/\,l} (M, l \cdot l')}\,[\textsc{Cmd}] \tag{11}$$

一个值得注意的细节是，发出的事件都是全新的，即不同于此前、同时或之后发出的任何其他事件。实际中，这可以通过包含起源时间戳来实现，例如将逻辑时钟或物理时钟时间戳与唯一的网络节点标识符配对。另请注意，这些全新的事件会被追加到本地日志中，排序在当前已知的所有事件之后

[7]就我们的结果而言，如何从正则树获得有限状态自动机并不重要；感兴趣的读者可参阅附录 A.4 了解详细信息。

于此位置。应用 [命令] 规则后机器的状态为 $\delta(\mathtt{M}, l\cdot l')$，这表明本地日志中的更改会在下一个命令可被启用之前处理。这使得我们能够控制命令的调用频率。

引理 A.2。 如果 $(\mathtt{M}, l) \xrightarrow{\mathtt{c}/\mathtt{l}} (\mathtt{M}, l')$ 那么存在 $l'' \in \mathcal{L}$ 使得 $\vdash l'' : \mathtt{l}$ 并且 $l' = l\cdot l''$。

证明。由规则 [命令] 可直接得出，注意到日志扩展了新的 eve nts. □

A.2 群的形式化

现在我们介绍用于规范机器之间以及网络节点之间交互的模型。如上所述，机器通过读取局部日志或向其中添加新事件来操作自己的局部日志。更改会异步传播到其他机器的局部日志。因此，每台机器当前可用的日志可能只包含系统迄今生成的事件集合的部分视图。

定义 A.3（群体）。一个大小为 n 的群体是一个对 $(\mathtt{S}, \mathtt{l}) \in (\mathtt{M}\times\mathtt{L})^{\underline{n}}\times\mathtt{L}$，其中 $\mathtt{S}(i) = (\mathtt{M}_i, \mathtt{l}_i)$ 描述第 i 台机器。每个事件都通过映射 $source : \mathtt{E} \to \mathtt{N}$ 分配一个源机器，这意味着事件 e 是机器 $\mathtt{S}(source(\mathtt{e}))$ 上一次命令调用的结果。

请注意，同一台机器 $\mathtt{M} \in \mathtt{M}$ 可以有多个副本，也就是说，它可以作为群体中多个参与者的机器；每台参与的机器都有唯一身份，并跟踪各自独立的本地日志（直观地说，参与者的身份就是它在群体 S 的定义域中的索引）。为方便起见，我们用 $(\mathtt{M}_1, \mathtt{l}_1) \mid \ldots \mid (\mathtt{M}_n, \mathtt{l}_n) \mid \mathtt{l}$ 表示群体 $(\mathtt{S}, \mathtt{l})$，其中对于所有 $i \in n$，$\mathtt{S}(i) = (\mathtt{M}_i, \mathtt{l}_i)$。我们称 l 为 S 的全局日志，称 $\mathtt{l}_i$ 为机器 $\mathtt{M}_i$ 的本地日志。

定义 A.4（一致性）。若群体 $(\mathtt{S}, l) = (\mathtt{M}_1, l_1) \mid \ldots \mid (\mathtt{M}_n, l_n) \mid l$ 满足一致性，当且仅当 $l_i \sqsubseteq l$ 对于所有 $i \in n$ 并且 $l \Đ _{i\in\underline{n}} l_i$。

此定义确立了局部视图与全局视图之间的一致性要求：每个本地日志都是全局日志的子日志，而全局日志本身不包含超出各本地日志范围的事件。下文中，我们假设群体均满足一致性。向下完备性这一条件使实际实现中的本地日志能够得到更简洁的刻画，因为子日志可由全局日志上下文中每个源的最新事件集合唯一确定。换言之，子日志可以表示为一组事件标识（例如源和时间戳），其基数至多为群体的大小；Actyx 实现利用这一点为状态快照添加标记，以加快后续的状态计算。向下完备性对于定义 6.1 中给出的因果一致性规则也至关重要。

请注意，根据相干性，l_i 中的所有元素都已在 l 中。因此，合并操作会将新生成的事件追加到全局日志 l 中；这些事件可以在 l_i 中最后一个元素出现之后的任意位置交错插入 l 中。

示例 A.5（事件生成）。假设 $(\mathtt{M}_1, \mathtt{b})\ \mathtt{c}/\mathtt{l} \longrightarrow (\mathtt{M}_1, \mathtt{b}\cdot\mathtt{d}\cdot\mathtt{e})$ 成立。那么，群 $(\mathtt{S}, \mathtt{l}) = (\mathtt{M}_1, \mathtt{b}) \mid \ldots \mid \mathtt{a}\cdot\mathtt{b}\cdot\mathtt{c}$（其中 a 和 c 来自其他某个源）可以演化为

$$(\mathtt{S}, l) \xrightarrow{\mathtt{c}/\mathtt{l}} (\mathtt{M}_1, b\cdot d\cdot e) \mid \ldots \mid l' \quad \text{with} \quad l' \in \{a\cdot b\cdot c\cdot d\cdot e,\ a\cdot b\cdot d\cdot c\cdot e,\ a\cdot b\cdot d\cdot e\cdot c\}$$

我们使用非确定性来建模这样一个事实：对于生成的事件与那些局部未知的事件之间的相对顺序，没有任何保证。◇

规则 [传播规则] 确立了我们的事件传播策略。本地日志可以通过全局日志中的事件进行扩展 l，前提是扩展后仍然是 l 的子日志；此约束确保来自同一命令调用的事件按顺序交付。

示例 *A.6*（事件传播）。考虑相干群体

$$(\mathtt{M}_0, a_0) \mid (\mathtt{M}_1, b_1 \cdot d_1 \cdot e_1) \mid (\mathtt{M}_2, c_2) \mid a_0 \cdot b_1 \cdot d_1 \cdot c_2 \cdot e_1$$

其中下标表示每个事件的源。根据规则 [传播规则], 我们可以将a传播到机器 2，从而得到$a \cdot c$——请注意，此日志可能意味着导致c的命令一开始就不应被启用！我们也可以将b或整个全局日志传播到机器 2，但如果不同时传播d，就无法传播b，如示例2.2所示。⋄

A.3 一个完整的示例

我们通过贯穿示例中的一个复杂场景来阐释框架的要点。请考虑贯穿示例中的以下机器集合：

passenger: $\mathtt{P} = \{\mathsf{Request} \,/\, \mathsf{Requested}\}\cdot[\mathsf{Requested}?\, \mathsf{Bid}?\, \mathsf{BidderID}?\, \mathtt{P}_1]$
$\mathtt{P}_1 = \{\mathsf{Select} \,/\, \mathsf{Selected} \cdot \mathsf{PassengerID}\}\cdot[\mathsf{Bid}?\, \mathsf{BidderID}?\, \mathtt{P}_1 \;\&\; \mathsf{Selected}?\, \mathsf{PassengerID}?\, \mathtt{P}_2]$
$\mathtt{P}_2 = \{\mathsf{Cancel} \,/\, \mathsf{Cancelled}\}\cdot[\mathsf{Cancelled}?\, \mathsf{Receipt}?\, 0 \;\&\; \mathsf{Arrived}?\, \mathtt{P}_3]$
$\mathtt{P}_3 = \{\mathsf{Start} \,/\, \mathsf{Started}\}\cdot[\mathsf{Started}?\, \mathtt{P}_4]$
$\mathtt{P}_4 = \{\mathsf{Finish} \,/\, \mathsf{Finished}\}\cdot[\mathsf{Path}?\, \mathtt{P}_4 \;\&\; \mathsf{Finished}?\, \mathsf{Receipt}?\, 0]$

taxi: $\mathtt{T} = \mathsf{Requested}?\, \mathtt{T}_1$
$\mathtt{T}_1 = \{\mathsf{Offer} \,/\, \mathsf{Bid}\}\cdot[\mathsf{Bid}?\, \mathsf{BidderId}?\, \mathtt{T}_1 \;\&\; \mathsf{Selected}?\, \mathsf{PassengerID}?\, \mathtt{T}_2]$
$\mathtt{T}_2 = \{\mathsf{Arrive} \,/\, \mathsf{Arrived}\}\cdot[\mathsf{Arrived}?\, \mathsf{Started}?\, \mathtt{T}_3 \;\&\; \mathsf{Cancelled}?\, \mathsf{Receipt}?\, 0]$
$\mathtt{T}_3 = \{\mathsf{record} \,/\, \mathsf{Path}\}\cdot[\mathsf{Path}?\, \mathtt{T}_3 \;\&\; \mathsf{Finished}?\, \mathsf{Receipt}?\, 0]$

办公室：$\mathtt{O} = \text{请求}?\ \text{出价}?\ \mathtt{O}_1$
$\mathtt{O}_1 = \text{出价}?\ \ \mathtt{O}_1 \,\&\, \text{已选中}?\ \ \mathtt{O}_2\mathtt{O}_2 = \text{已到}$
达？ 已开始？ $\mathtt{O}_3$ & 已取消？ $\mathtt{O}_4\mathtt{O}_3$ = 路
径？ $\mathtt{O}_3$ & 已完成？ $\mathtt{O}_4\mathtt{O}_4 = \{\text{收据} \,/\, \text{收据}\}\cdot$
[收据？ 0]

机器作为自动机的图形化表示如示例 5.2 所示。乘客（与出租车一样）会处理所有事件类型，而办公室会跳过那些对于遵循协议整体进展并非必需的事件——尤其是，办公室无法看到乘客 ID 事件，而出租车正是利用该事件来寻找、识别并接载乘客。

在下文中，我们将事件命名为与其类型完全相同的名称，并在需要进一步区分时添加索引。例如，$\vdash$ 收据$_1$: 收据。

从一个群（P ϵ）|（T ϵ）|（T ϵ）|（O ϵ）|ϵ开始——其中有两辆出租车，分别命名为 A 和 B——唯一可能的归约是调用请求命令，得到群（P, requested）|（T ϵ）|（T ϵ）|（O ϵ）|requested。现在乘客处于状态 $\mathtt{P}_1$，可以立即选择一辆出租车，但我们的乘客决定等待出价。请求事件传播到两辆出租车后，它们可以并发调用报价命令，例如得到群

$$(\mathtt{P}, \mathit{requested}) \mid (\mathtt{T}, \mathit{requested} \cdot \mathit{bid}_A \cdot \mathit{bidderid}_A) \mid (\mathtt{T}, \mathit{requested} \cdot \mathit{bid}_B \cdot \mathit{bidderid}_B) \mid (\mathtt{0}, \epsilon) \mid \mathit{requested} \cdot \mathit{bid}_B \cdot \mathit{bidder}_B \cdot \mathit{bid}_A \cdot \mathit{bidder}_A \tag{12}$$

注意，（12）中的传播并不完整：事件 出价$_A$ 和 竞价者 ID$_A$（分别是 出价$_B$ 和竞价者 ID$_B$））仅存在于全局日志和出租车 A（分别为 B）的本地日志中；尚未传播

到出租车 B（分别为 A）。另外，请注意，这些事件在全局日志中的顺序是“任意的”；事实上，也可能是相反的顺序。

此时，第三辆出租车 C 登场了。我们通过向群中添加一台新机器来模拟这种动态行为；从概念上说，这台机器从一开始就已存在，但迄今为止，它既不是任何 [局部] 规则的目标，也不是任何 [传播规则] 的目标；这不会影响在规模较小的群上执行的归约，因为没有规则要求我们检查或修改群中的所有参与者。

随着请求和出价在群中传播，可以到达以下状态：

$$(\mathtt{P}, requested \cdot bid_B \cdot bidder_B) \mid (\mathtt{T}, requested \cdot bid_B \cdot bidder_B \cdot bid_A \cdot bidder_A) \mid (\mathtt{T}, requested \cdot bid_B \cdot bidder_B \cdot bid_A \cdot bidder_A) \mid (\mathtt{O}, \epsilon) \mid (\mathtt{T}, requested \cdot bid_A) \mid requested \cdot bid_B \cdot bidder_B \cdot bid_A \cdot bidder_A \quad (13)$$

如果此时乘客选择出租车 B，而出租车 C 同时出价，那么我们可能会得到如下全局日志 $l_{\text{auc}} = requested \cdot bid_B \cdot bidder_B \cdot bid_A \cdot bidder_A \cdot selected \cdot bid_C \cdot bidderid_C \cdot passengerid$ 以及群

$$(\mathtt{P}, \text{已请求} \cdot \text{出价}_B \cdot \text{出价者}_B \cdot \text{已选中} \cdot \text{乘客 ID}) \mid (\mathtt{T}, \text{请求}\ sted \cdot \text{出价}_B \cdot \text{竞标者}_B \cdot \text{出价}_A \cdot \text{竞标者}_A) \mid (\mathtt{T}, \text{请求} \cdot \text{出价}_B \cdot \text{竞标者}_B \cdot \text{出价}_A \cdot \text{竞标者}_A) \mid (\mathtt{O}, \epsilon) \mid (\mathtt{T}, requested \cdot bid_A \cdot bidder_A \cdot bid_C \cdot bidder_C) \mid l_{\text{auc}} \quad (14)$$

此时，乘客认为只有一次出价，随后进行了选择；出租车则认为有两次出价（但并非同一次出价），且没有进行选择，而办公室尚未掌握任何信息。一旦所有事件都传播完毕，乘客便意识到实际上还有第二次出价，所有人都知道拍卖已经结束，而 C 出租车则发现自己的出价因提交过晚而被忽略。

所有机器分别处于以下状态 $\mathtt{P}_2, \mathtt{T}_2$, 和 $\mathtt{O}_1$ 我们发现自己正处于一个十字路口：行程要么发生，要么被取消。由于乘客和出租车都可能并发行动，群可能进入不一致的状态。假设并发调用 到达 和 取消，随后将 已取消 事件传播到办公室，群处于状态

$$(\mathtt{P}, l\ \text{auc} \cdot \text{已取消}) \mid (\mathtt{T}, l_{\text{auc}}) \mid (\mathtt{T}, l\ \text{auc} \cdot \text{已到达}) \mid (\mathtt{O}, l\ \text{auc} \cdot \text{已取消}) \mid (\mathtt{T}, l_{\text{auc}}) \mid l\ \text{auc} \cdot \text{已到达} \cdot \text{已取消} \quad (15)$$

出租车 A 和 C 仍在等待作出选择，出租车 B 正在等待乘客调用 开始 命令，乘客正在等待办公室调用 收据 命令，而办公室很可能会这么做。办公室生成 收据_1 事件后，全局日志必然会变成 $l_{\text{冲突}} = l_{\text{auc}} \cdot \text{已到达} \cdot \text{已取消} \cdot \text{收据}_1$。办公室将其所有本地日志传播给其他每台机器后，除出租车 B 外的所有机器都会认为协议已结束，且没有行程，只有出租车 B 知道存在冲突：

$$(\mathtt{P}, l_{\text{auc}} \cdot cancelled \cdot receipt_1) \mid (\mathtt{T}, l_{\text{auc}} \cdot arrived \cdot cancelled \cdot receipt_1) \mid (\mathtt{T}, l_{\text{auc}} \cdot cancelled \cdot receipt_1) \mid (\mathtt{O}, l_{\text{auc}} \cdot cancelled \cdot receipt_1) \mid (\mathtt{T}, l_{\text{auc}} \cdot cancelled \cdot receipt_1) \mid l_{\text{confused}} \quad (16)$$

但实际上，一旦出租车 B 成功将 已到达 事件（例如）传送给乘客，该机器就会根据新的本地日志重新计算其状态 $l_{\text{auc}} \cdot \text{已到达} \cdot \text{已取消} \cdot \text{收据}_1$，最终进入状态 $\mathtt{P}_3$。因此，现在可以通过 开始 命令来开始行程。如果乘客不再希望乘坐出租车出行，他们仍需遵循协议，开始并完成一次可能为空的行程，以便办公室为其生成收据（逾期取消可能会产生费用）。

A.4 从正则树到有限 LTS

正则树可以表示为有限标记转换系统。我们展示如何将机器语法中的项（参见（2））转换为论文中使用的图形表示法。

机器也可以等价地表示为有限标记转移系统，其中 κ 变为自环，而 $\mathtt{t}_i?\,\mathtt{M}_i$ 变为从 M 到 $\mathtt{M}_i$的转移。更准确地说，我们可以为机器 $\mathtt{M} = \kappa\cdot[\mathtt{t}_1?\,\mathtt{M}_1 \,\&\, \cdots \,\&\, \mathtt{t}_n?\,\mathtt{M}_n]$ 关联一个有限状态自动机，具体如下：自动机的状态是 M 的子项；在状态 M 中，对于每个 $\mathtt{c}/\mathtt{l} \in \kappa$，都有一条标记为 $\mathtt{c}/\mathtt{l}$ 的指向 M 的自环转移；对于每个 $i \in n$，都有一条标记为 $\mathtt{t}_i?\,\mathtt{M}_i$、指向状态 $\mathtt{M}_i$的转移。每个子机器 $\mathtt{M}_i$也同样如此。直观而言，分支 $\mathtt{t}_i?\,\mathtt{M}_i$描述了一种转移：机器从日志中消费一个类型为 $\mathtt{t}_i$（称为守卫）的事件，并继续处于由 $\mathtt{M}_i$描述的状态。请注意，由于 M 具有正则性，这种构造会产生一个有限状态自动机。

同样，可以根据第 4 节给出的群协议语法中的项构造一个有限状态自动机，方法如下。对于每个 $i \in n$，状态 $\mathtt{G} = \Íi\in I\mathtt{c}_i@\mathtt{R}_i\mathtt{hl}_i\mathtt{i}\,.\,\mathtt{G}_i$ 都有一条转移指向状态 $\mathtt{G}_i$，该转移标记为 $\mathtt{c}_i@\mathtt{R}_i\mathtt{hl}_i\mathtt{i}$，表示角色 R 的参与者执行命令 c，并产生一个非空的 l 类型事件序列。请注意，这种构造会得到从给定状态出发、标记不同角色的多条转移的自动机；这对应于第 4 节讨论的非局部决策。

A.5 确保性质

以下示例与第 6.1 节中的问题有关。

示例 A.7（因果性）。考虑对贯穿示例开头的群协议作如下变体：$\mathtt{G} = \mathtt{Request@Ph}\,\mathtt{Requested}i\,.\,\mathtt{Offer@Th}\,\mathtt{Bid}i\,.\,\mathtt{G}'$。此类型的语义是：首先必须发生 Request，之后才发生 Offer，然后协议继续。如果$\sigma(\mathtt{P}) = \emptyset$，那么命令 Request 将始终保持启用状态；因此，为了确保行为忠实，我们必须要求 $\mathtt{requested} \in \sigma(\mathtt{P})$。同样，我们需要要求 $\mathtt{Bid} \in \sigma($T)，否则出租车机器将永远无法到达最终状态，并会不断发出事件。但 T 也必须对 Requested 作出响应，否则 Offer 可能与 Request 并发调用，合并日志可能变成 $\mathtt{bid}\cdot\mathtt{requested}$，从而违反预期语义。⋄

示例 *A.8*（传播）。考虑群协议

$$\mathtt{G} = \mathtt{Select@P}\langle\mathtt{Selected}\cdot\mathtt{PassengerID}\rangle\,.\,\mathtt{Arrive@T}\langle\mathtt{Arrived}\rangle\,.\,\mathtt{Receipt@O}\langle\mathtt{Receipt}\rangle$$

和订阅

$$\sigma : \begin{cases} \mathtt{P} \mapsto \{\mathtt{Selected}, \mathtt{Arrived}\} \\ \mathtt{T} \mapsto \{\mathtt{Selected}, \mathtt{Arrived}\} \\ \mathtt{O} \mapsto \{\mathtt{Selected}, \mathtt{PassengerID}, \mathtt{Arrived}, \mathtt{Receipt}\} \end{cases}$$

我们得到以下投影：

$$\mathtt{M}_\mathtt{P} = \mathtt{G}\downarrow^\sigma_\mathtt{P} = \{\text{选择} \,/\, \text{已选中}\cdot\text{乘客 ID}\}\cdot[\text{已选中}?\ \text{已到达}?\ \ 0]\mathtt{M}_\mathtt{T} = \mathtt{G}\downarrow^\sigma_\mathtt{T} = \text{已选中}?\ \ \{\text{到达} \,/\, \text{已到达}\}\cdot[\text{已到达}?\ \ 0]\mathtt{M}_\mathtt{O} = \mathtt{G}\downarrow^\sigma_\mathtt{O} = \text{已选中}?\ \text{乘客 ID}?\ \text{已到达}?\ \ \{\text{收据} \,/\, \text{收据}\}\cdot[\text{收据}?\ \ 0]$$

从空日志开始，唯一可能的归约是调用 选择，生成日志已选中 · 乘客 ID。如果现在只让 已选中 传播到 $\mathtt{M}_\mathtt{T}$ 我们可以调用 到达 来生成事件 已到达。由于事件 乘客 ID 在 $\mathtt{M}_\mathtt{T}$ 未知，日志合并操作可能会将 已到达 排在 乘客 ID之前。此时 $\mathtt{M}_\mathtt{O}$ 会陷入阻塞：其命令 收据 将

永远不会启用，因为事件 已到达 已被忽略（请回想 $\delta(\mathtt{M}_0,\ 已选中 \cdot 已到达 \cdot 乘客\ ID) = 已到达?\{收据 / 收据\}\cdot[收据?\ \ 0]$）。⋄

例 *A.9.* 请注意，因果性的规则恰好要求订阅例 A.7 中讨论的那些事件。此外，这些规则还要求角色 T 订阅事件类型 乘客 ID 以解决例A.8 中的问题。⋄

以下示例是第 6.2 节中问题的一个实例。

例 *A.10*（发散的局部选择）。考虑贯穿示例中的主要选择（去掉无关的路径跟踪后）：

$$\begin{aligned} G_{choose} = {} & Arrive@T\langle Arrived\rangle . Finish@P\langle Finished\rangle . Receipt@O\langle Receipt\rangle 0 \\ & + Cancel@P\langle Cancelled\rangle . Receipt@O\langle Receipt\rangle 0 \end{aligned}$$

假设 到达 和 取消 并发调用，使得生成的全局日志为已到达 · 已取消 。定义 6.1 的限制仅要求 $\sigma(O) \supseteq \{已完成, 已取消, 收据\}$
在这种情况下，给定上述日志，角色 O 的机器错误地认为已选择下分支，而群协议却继续执行上分支。⋄

示例 A.10 中的问题源于角色 O 不知道某个选择的一个分支，尽管它仍然参与该分支的后续过程。

以下示例是第 6.3 节所述问题的一个实例。

示例 *A.11*（决策含糊）。设 $\sigma = \mathcal{R} \times \{\mathcal{T}\}$ 采用贯穿示例的以下变体：

$$\begin{aligned} G_{choose} = {} & 到达@T\langle 已到达\rangle . 完成@P\langle 已完成 \cdot 评分\rangle . 收据@O\langle 收据\rangle 0 + 取消@P\langle \\ & 已完成\rangle . 收据@O\langle 收据\rangle 0 \end{aligned}$$

如果 到达 和 取消 并发调用，那么生成的日志可能是 已到达 · 已完成，这会阻碍进一步进展：命令完成 将不再可启用，因此无法发出 评分事件。即使没有这种死锁，我们也注意到，上分支会处理一个源自失败下分支的 已完成 事件。⋄

A.6 异常

为便于阅读，我们重述第 6.4 节末尾所述的异常

- 如果模型是同步的，机器中启用的某些命令本应被禁用；这可能导致发出之后需要忽略的事件。
- 事件仅根据其类型被忽略，因此，即使全局日志中的事件已完全传播，机器仍可能处理源自某条命令的异常调用的事件。

贯穿示例恰当地说明了第一点，尤其是附录 A.3。我们用群协议 $G = c@Rhti . c@Rhti . 0$ with $\sigma(R) = \{t\}$ and $M = G \downarrow^{\sigma}_{R} = \{c / t\}\cdot[t?\ \{c / t\}\cdot[t?\ 0]]$ 说明第二点。考虑群 $(M\ \epsilon) \mid (M\ \epsilon) \mid \epsilon$，并设两台机器同时调用命令 c，从而得到群 $(M, t_1) \mid (M, t_2) \mid t_1 \cdot t_2$。将全局日志传播到两台机器后，两者都会到达最终状态，尽管它们本应忽略第二个事件，因为该事件源自对 c 的异常调用。

虽然可以通过禁止同一角色的多个实例来避免上述情况，但类似的异常无法用同样的方式修复。考虑通用订阅 $\sigma = R \times \{T\}$、群协议 $G = c_1@Aha \cdot ci\ .0 + c_2@Bhb \cdot ci\ .0$、投影 $A = G \downarrow^{\sigma}_{A}$ 和 $B = G \downarrow^{\sigma}_{B}$，以及群 $(A\ \epsilon) \mid (B\ \epsilon) \mid \epsilon$。并发调用这两条命令很可能会产生全局日志 $b \cdot$

一个 $\cdot c_1 \cdot c_2$ 与 $source(c_1) = 1$。传播到所有机器后，它们将把 b 解释为，丢弃 a，然后处理 c_1 而不是 c_2—— 后一种方式更为恰当，因为它是伴随 b 并源自 c_2 的。

A.7 第 2、2、第 6 和7 节证明

引理 2.3（相干性保持）。若(S, l) 是一个相干群体，且$(S, l) \xrightarrow{\alpha} (S', l')$，则$(S', l')$ 是相干的。

证明。根据推导$(S, l) \xrightarrow{\alpha} (S', l')$，通过归纳即可得出。 [传播规则] 由前提$l'_i \sqsubseteq l$立即成立，而规则 [局部] 根据 $_ \bowtie _$的定义成立。□

引理 2.4（最终一致性）。若 $(S, l) = (M_1, l_1) \mid \ldots \mid (M_n, l_n) \mid l$ 是一个相干群体，则 $(S, l) \Longrightarrow (M_1, l) \mid \ldots \mid (M_n, l) \mid l$。

证明。 对 n进行归纳，并注意到 $(S, l) \xrightarrow{\tau} (S[i \mapsto (M_i, l)], l)$ 对所有 i均成立，依据规则[传播规则]。 □

命题6.9。 设 G 和 σ 分别为群协议和订阅。若 G 是 σ-*WF*，则 $G \downarrow_R^\sigma$ 对于所有 R都是确定的。

证明。 证明通过余归纳论证，并注意到定义 6.3 确保每个角色都订阅发出的事件中的第一个类型；根据定义，这些类型应当彼此不同6.5。 □

引理 *7.9.* 若 G 是一个 σ-*WF* 群协议，且 $(\delta(G \downarrow_R^\sigma, l)) \downarrow_c / l$，则存在 $l' \equiv_{G,\sigma} l$，使得 $(G\ \epsilon) \Longrightarrow (G, l')$，并且 $\delta(G, l')\ c@R\ hl\ i \longrightarrow G'$。

证明。 假设 $G = \sum_{i \in I} c_i @ R_i \langle 1_i i$ 。 G_i并对 l进行归纳。取$l' = \epsilon$即可直接得到基础情形。对于归纳步骤，我们考虑$l = e \cdot l'$，并分情况讨论。

如果 $\mathbb{T}_\sigma(e, G) = \epsilon$那么 $\vdash e : t$且 $t \notin$ 守卫 $(G \downarrow_R^\sigma)$。因此，$\delta(G \downarrow_R^\sigma, e \cdot l') = \delta(G \downarrow_R^\sigma, l')$，由归纳假设可得该情形成立。

如果 $T_\sigma(e, G) = t$。根据有效类型的定义，存在 $i \in I$，使得 $G\ c_i@R_i\ hl_i\ i \longrightarrow G'' = G_i$，满足筛选 $(l_i, \sigma(R)) = t \cdot l_i'$。那么，存在两种情况。

(i) 如果存在 $l\,1$ 和 $l\,2$，使得 $l' = l\,1 \cdot l_2$，且 $T_\sigma(l_1, G) = l_i'$。那么，$(\delta(G \downarrow_R^\sigma, l)) \downarrow_c / l$ 当且仅当 $(\delta(G'' \downarrow_R^\sigma, l\,2)) \downarrow_c / l$。根据归纳假设，我们得出结论：存在 $l'_2 \equiv_{G,\sigma} l_2$，使得 $(G'', \epsilon) \Longrightarrow (G', l'2)$，且 $\delta(G'', l_2)\ c@R\ hl\ i \longrightarrow G'$。通过取 $l' = l'1 \cdot l'2$ 即可完成证明，其中 $l'1$ 是任意一个类型为 l_i的新鲜日志。(ii) 如果对于所有满足 l_1和 l_2且 $l = l_1 \cdot l_2$ 的情况，都有 $T_\sigma(l\,1, G) = l_1 \neq l_i$。那么，我们得出结论：$T_\sigma(l, G)$ 是 l_i 的一个真前缀；因此，$\delta(G \downarrow_R^\sigma, l) = t_1 ?\ M$，这与假设 $(\delta(G \downarrow_R^\sigma, l)) \downarrow_c / l$ 相矛盾。

□

引理 7.12。 若 l 对于某个 σ-*WF* 群协议 G 是可容许的，则存在一个日志 l' 使得$(G\ \epsilon) \Longrightarrow (G,\ l')$并且 $l \equiv_{G,\ \sigma} l'$。

证明。对 $T_\sigma(l, G) = l$ 的长度进行归纳。基础情形显然成立。对于归纳步骤，考虑 $l = l' \cdot t$。根据 $T_\sigma(l, G)$ 的定义，应存在 $e \in l$，使得 $\vdash e : t$，且 $G\ c@R hl i \longrightarrow G'$ with $t \in l_e$。由于 l 是可容许的，其有效类型与通过合并 G 的若干次执行所得到的日志的有效类型一致，即 $\{(G\ \epsilon) \Longrightarrow$

$(G, m_i)\}_{i\in\underline{k}}$（因此，$l_e$ 的所有相关事件都在该日志中）。因此，存在 l_1，使得 $l = l_1 \cdot$ 筛选 $(l_e, \sigma$(活跃 $(G'))$（$G' = 0$ 的情形类似）。由此，存在 l_1 和 $l\,2$，使得 $l = l\,1 \cdot l_2,$，且 $l_1 = T_\sigma(l\,1, G)$，并且 $T_\sigma(l\,2, G') =$ 筛选 $(l_e, \sigma$(活跃 $(G')))$。需要注意的是，l_1 对 G 是可容许的，因为它可以与 l' $\in(\bowtie_{i\in\underline{k}} l'_i)$ 中的一个日志相匹配，其中 $(G\ \epsilon) \Longrightarrow (G, l'_i) \Longrightarrow (G, l_i)$，并且对于所有 $i \in k$，$l'_i \cap l_1 \neq \emptyset$。根据归纳假设，存在 $(G\ \epsilon) \Longrightarrow (G, l'1)$，且 $l'_1 \equiv_{G,\sigma} l_1$。然后，取 $l' = l'_1 \cdot l'2$ 即可完成证明，其中 $l'2$ 是任意具有适当类型的新鲜日志，即 $\vdash l'2 : l_e$。□

引理 *7.13*。设 $l\,1$ 和 $l_2 \subseteq l\,1$ 是协议 G 的可容许 *WF* 群日志。若 $(G, l\,2)\ c@R\ hl\,i \longrightarrow (G, l\,2 \cdot l\,3)$，且 $l \in l_1 \bowtie (l_2 \cdot l_3)$，则 l 对 G 可容许。

证明。由于 l_1 对 G 可容许，因此存在 $\{(G\ \epsilon) \Longrightarrow (G, m_i)\}_{i\in\underline{k}}$，以及 $l'_1 \in (\bowtie_{i\in\underline{k}} m_i)$，其中 $l'_1 \equiv_{G,\sigma} l_1$。由于 l_2 可容许，根据引理 7.12，有 $(G\ \epsilon) \Longrightarrow (G, l'2)$。因此，取 $m_{k+1} = l'2 \cdot l_3$。于是，$(G\ \epsilon) \Longrightarrow (G, m_{k+1})$。取 $l \in (\bowtie_{i\in\underline{k+1}} m_i)$。假设 $l_3 = e_1 \ldots e_n$。那么，$l = h_0 \cdot e_1 \cdot l'_h$，其中 $l'_h \in e_2 \ldots e_n \bowtie h_1$，且 $l_1 = h_0 \cdot h_1$，$l_2 \subseteq h_0$。于是 $T_\sigma(l, G) = h_0 \cdot T_\sigma(e_1 \cdot l'_h, G_0)$。有两种情况。若 $G_0 \neq \delta(G, l_2)$，即它不同于允许执行 c 的全局图，则 $T_\sigma(e_1 \cdot l'_h, G_0) = T_\sigma(l'_h, G_0)$，也就是说，该事件未反映在类型中。此外，基于守卫不变性，$e_2 \ldots e_3$ 中的其余事件不能作为守卫；因此它们无法改变有效类型。因此，$T_\sigma(l, G) = T_\sigma(l_1, G)$。

若 $G_0 = \delta(G, l_2)$，则 $T_\sigma(l, G) = h_0 \cdot T_\sigma(e_1 \cdot l'_h, G_0) = h_0 \cdot$ 筛选 $(l_e, \sigma$(活跃 $(G'))) \cdot T_\sigma(l''_h, G')$，其中 $l''_h \in e_k \ldots e_n \bowtie h''1$，且 $k > 1$ and $h_1 = h'_1 \cdot h''1$。接下来有两种情况：要么 $T_\sigma(l''_h, G') = \epsilon$，要么 $T_\sigma(l''_h, G') = t_e \cdot l_h'''$。前一种情况显然成立；后一种情况可由以下观察得出：类型为 t_e 的事件应属于 h'_1，因为对于 $k > 0$，没有任何 $e_k \ldots e_n$ 的类型可能是 t_e（t_e 是 G' 的一个守卫，根据不变性，它不能出现在两个转移中）。□

定理 7.14. 设 $(S\ \epsilon)$ 是一个 σ-*WF* 群协议 G。若 $(S\ \epsilon) \Longrightarrow (S',\ l)$ 则 l对 G是可容许的。

证明。 该证明通过对推导长度进行归纳得出。基础情形显然成立。对于归纳情形，取 $(S\ \epsilon) \Longrightarrow (S'',\ l'') \xrightarrow{\alpha} (S',\ l)$。根据归纳假设，$l''$ 对于 G是可容许的。接下来，我们根据最后一步归约所使用的规则进行分类讨论：

- 局部。然后，存在 $i \in \mathrm{dom}\ S$，使得 $S(i) = (G \downarrow^{\sigma}_{R_i}, l_i)$，且 $G \downarrow^{\sigma}_{R} l_i\, c / l \longrightarrow G \downarrow^{\sigma}_{R} l'_i$，并且 $l \in l'' \bowtie l'_i$。根据引理 7.9，l_i是可容许的。根据命令 Cmd，$l'_i = l_i \cdot l''_i$，其中 l''_i是一个新鲜的变量，满足$\vdash l'_i : l$。显然，l'_i是可容许的。然后，根据引理 7.13，$l \in l'' \bowtie l'_i$是可容许的。
- 传播规则。那么，$l = l''$ 其对于 G 是可容许的。

□

引理 7.16。 如果 G 是 σ-良构群协议，并且 $(G\ \epsilon) \Longrightarrow (G, l)$ 那么 $\delta(G \downarrow^{\sigma}_{R}, l) = \delta(G, l) \downarrow^{\sigma}_{R}$对于所有 $R \in$ 角色 $(G\ \sigma)$。

证明。对运行的长度 k 进行归纳。情形 $k = 0$ 显然成立。对于归纳步骤，我们假设 $(G\ \epsilon) \Longrightarrow (G, l')\ c / l \longrightarrow (G, l)$。根据规则全局命令，$\delta(G, l') = G'\ c / l \longrightarrow G''$ 且 $l = l' \cdot l''$，其中 $\vdash l'' : l$。由 $G'\ c / l \longrightarrow G''$，我们得出 $G' = \acute{I}_{i\in I} c_i@R_i hl_i i\ .\ G_i$，并且存在 $j \in I$

使得$\mathtt{c}_j = \mathtt{c}$且$\mathtt{l}_j = \mathtt{l}$并且$\mathsf{G}_j = \mathsf{G}''$。于是，

$$\begin{aligned}\delta(\mathsf{G}, l) &= \delta(\mathsf{G}, l' \cdot l'') && \text{because } l = l' \cdot l' \\ &= \delta(\mathsf{G}', l'') && \text{because } \mathsf{G}' = \delta(\mathsf{G}, l') \\ &= \mathsf{G}'' = \mathsf{G}_j && \text{because } \mathsf{G}' = \sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i \text{ and } \vdash l'' : \mathtt{l}\end{aligned}$$

我们需要证明 $\delta(\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}}, \mathtt{l}' \cdot \mathtt{l}'') = \mathsf{G}''\downarrow^{\sigma}_{\mathtt{R}}$。根据归纳假设，$\delta(\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}}, \mathtt{l}') = \delta(\mathsf{G}, \mathtt{l}')\downarrow^{\sigma}_{\mathtt{R}} = \mathsf{G}'\downarrow^{\sigma}_{\mathtt{R}}$。因此，$\delta(\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}}, \mathtt{l}' \cdot \mathtt{l}'') = \delta(\mathsf{G}'\downarrow^{\sigma}_{\mathtt{R}}, \mathtt{l}'')$。注意，

$$\mathsf{G}'\downarrow^{\sigma}_{\mathtt{R}} = (\sum_{i \in I} \mathtt{c}_i @ \mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i)\downarrow^{\sigma}_{\mathtt{R}} = \kappa\cdot\left[(\mathtt{c}_i @ \mathtt{R}_i\langle \mathtt{l}_i \rangle \,.\, \mathsf{G}_i)\downarrow^{\sigma}_{\mathtt{R}} \;\middle|\; i \in I\right]$$

我们分两种情况讨论，若$\mathtt{R} \notin$角色(G')，则$\mathsf{G}'\downarrow^{\sigma}_{\mathtt{R}} = 0$且$\mathsf{G}_i\downarrow^{\sigma}_{\mathtt{R}} = 0$对所有$i \in I$。因此，

$$\delta(\mathsf{G}\downarrow^{\sigma}_{\mathtt{R}}, l' \cdot l'') = \delta(\mathsf{G}'\downarrow^{\sigma}_{\mathtt{R}}, l'') = 0 = \mathsf{G}_j\downarrow^{\sigma}_{\mathtt{R}} = \mathsf{G}''\downarrow^{\sigma}_{\mathtt{R}} = \delta(\mathsf{G}, l)\downarrow^{\sigma}_{\mathtt{R}}$$

在这种情况下，$\mathtt{R} \in$角色（G'），根据定义 6.5，可得$\mathtt{l}_j[0] \in \sigma(\mathtt{R})$和$(\mathtt{c}_j @ \mathtt{R}_j\langle \mathtt{l}_j \rangle)\downarrow^{\sigma}_{\mathtt{R}} \neq 0$。由于$\vdash \mathtt{l}'' : \mathtt{l}_i$，显然存在$\mathtt{l}_1$和$\mathtt{l}_2$，使得$\mathtt{l}'' = \mathtt{l}_1 \cdot \mathtt{l}_2$，并且

$$\delta(\mathit{filter}(\mathtt{l}_j, \sigma(\mathtt{R}))\ ?.\ (\mathsf{G}_j\downarrow^{\sigma}_{\mathtt{R}}), l'') = \delta(\mathsf{G}_j\downarrow^{\sigma}_{\mathtt{R}}, l_2)$$

也就是说，所有匹配筛选$(\mathtt{l}_j, \sigma(\mathtt{R}))$的事件都会被处理。需要注意的是，$l_2$中的事件类型都不属于$\sigma(\mathtt{R})$；因此，$\delta(\mathsf{G}_j\downarrow^{\sigma}_{\mathtt{R}}, l_2) = \mathsf{G}_j\downarrow^{\sigma}_{\mathtt{R}} = \mathsf{G}''\downarrow^{\sigma}_{\mathtt{R}}$。□

命题 7.17。设$(\mathtt{S}\ \epsilon)$是大小为n的σ-*WF*群协议G的完全实现。若$(\mathsf{G}\ \epsilon) \Longrightarrow (\mathsf{G}, l)$则存在一个群$\mathtt{S}'$，使得$(\mathtt{S}\ \epsilon) \Longrightarrow (\mathtt{S}', l)$。

证明。对归约长度进行归纳。基础情形显然成立。归纳步骤由归纳假设、引理 7.16 以及将所有事件传播到所有副本得出。□