软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(9):4141−4159 [doi: 10.13328/j.cnki.jos.007130]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

并发对象强可线性化性质的检测和验证*

王 超[1], 贾巧雯[2,3], 吕 毅[2,3], 吴 鹏[2,3]

[1](西南大学 计算机与信息科学学院 软件研究与创新中心, 重庆 400715)

[2](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

[3](中国科学院大学, 北京 100049)

通信作者: 吕毅, E-mail: lvyi@ios.ac.cn

摘 要: 可线性化被公认为并发对象正确性标准, 但其已被证明不能作为含有随机语句的并发对象的正确性标准. 为此, Golab 等人提出了强可线性化概念, 它在可线性化的定义上增加了前缀保持性质, 对并发对象具有更强的约束性. 关于强可线性化的研究集中在使用特定的基本对象构造满足强可线性化性质的并发对象的可行性. 对常见的并发对象的强可线性化性质的检测和验证方面的研究较为少见. 从并发对象的验证算法和证否方法两个方面研究了强可线性化性质. 首先, 细化强可线性化性质, 将它细分为固定生效点和单纯帮助两类, 并证明固定生效点是已有的固定可线性化点概念的扩展. 其次, 提出两种强可线性化的验证算法, 其中一种基于固定可线性化点, 另一种基于固定生效点. 最后, 给出一个构造性的证明并发对象违背强可线性化的证明方法, 依据该方法证明了 Herlihy&Wing 队列、一种单读单写寄存器实现和一种并发快照实现违反强可线性化性质.

关键词: 并发对象; 正确性标准; 可线性化; 强可线性化

中图法分类号: TP301

中文引用格式: 王超, 贾巧雯, 吕毅, 吴鹏. 并发对象强可线性化性质的检测和验证. 软件学报, 2024, 35(9): 4141–4159. http://www.jos.org.cn/1000-9825/7130.htm

英文引用格式: Wang C, Jia QW, Lü Y, Wu P. Strong Linearizability Checking and Determination for Concurrent Objects. Ruan Jian Xue Bao/Journal of Software, 2024, 35(9): 4141–4159 (in Chinese). http://www.jos.org.cn/1000-9825/7130.htm

Strong Linearizability Checking and Determination for Concurrent Objects

WANG Chao[1], JIA Qiao-Wen[2,3], LÜ Yi[2,3], WU Peng[2,3]

[1](Centre for Research and Innovation in Software Engineering, College of Computer and Information Science, Southwest University, Chongqing 400715, China)

[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Linearizability is universally accepted as a correctness criterion for concurrent objects. However, it has been shown that linearizability cannot be adopted as a correctness criterion for concurrent objects with random sentences. Thus, Golab *et al*. proposed the concept of strong linearizability, which adds prefix preservation properties based on the linearizability definition and has more constraints for concurrent objects. The research on strong linearizability focuses on the feasibility of generating strongly linearizable objects with certain basic objects, while only a few studies are about checking and verification of strong linearizability. This study investigates strong linearizability from two aspects including the verification algorithm and approach for proving non-strong linearizability of concurrent objects. First, it divides strong linearizability into fixed effective points and pure help and proves that the notion of fixed effective points

* 基金项目: 国家自然科学基金 (62002298, 62072443, 62372386); 国家重点研发计划 (2022YFA1005100, 2022YFA1005101, 2022YFA1005104); 重庆市自然科学基金面上项目 (CSTB2022NSCQ-MSX0437)
本文由“形式化方法与应用”专题特约编辑曹钦翔副教授、宋富研究员、詹乃军研究员推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-13; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-05-17

is an extension of that of fixed linearizability points. Then, two verification algorithms for strong linearizability are put forward. One algorithm is based on checking the fixed linearizability points, and the other is based on the fixed effective points. Finally, an approach is provided for proving that the concurrent objects violate strong linearizability, and it helps verify that the Herlihy&Wing queue, a single-reader single-write register, and a snapshot object violate strong linearizability.

Key words: concurrent object; correctness criterion; linearizability; strong linearizability

1 引 言

可线性化[1]是并发对象实际的正确性标准. 如果一个并发对象是可线性化的, 则其每个方法在执行中都等效于在调用和返回之间的某个时间点原子地执行. 可线性化等价于观察精化[2], 因此可线性化的并发对象与原子实现的并发对象在观察行为上等价. 然而, 在含有随机语句的程序中, 这一结论不再成立. Golab 等人[3]证明了在使用可线性化的并发对象来代替对应原子实现的并发对象后, 在含有随机语句的并发程序中不能够保持结果概率分布相同. 为了能够在有着随机语句的环境中确保并发对象与原子实现对象的等效, Golab 等人[3]提出了强可线性化的概念. 强可线性化定义在一组并发执行上, 这组并发执行上存在一个将并发执行映射到合法顺序历史的强可线性化函数, 且要求此函数满足前缀保持的性质.

可线性化性质得到了研究者们的广泛研究[4–6], 很多并发对象的可线性化性质得到了验证. 然而, 针对并发对象的强可线性化验证的研究相对而言很少. 据我们所知, 只有如下几个并发对象被明确证明违背了强可线性化: Golab 等人[3]通过证明并发对象和对应原子实现在含有随机语句的并发程序中结果概率分布不同的方式, 间接证明了 Herlihy&Wing 队列[1], Vidyasankar[7]给出单读单写寄存器的实现和文献 [8] 中的并发快照对象违背强可线性化. Hwang 等人[9]证明了 Micheal&Scott 队列[10]和 Harris 链表[11]对象是非强可线性化的, 并且他们的证明直接给出了违背强可线性化的执行.

在验证算法开发方面, 与可线性化性质仅需要考虑方法的调用和返回动作的序列不同, 强可线性化的前缀保持性质是基于执行的, 需要考虑执行中的每一个动作, 这就增加了验证的难度. 在使用反例进行证否并发对象满足强可线性化性质方面, Hwang 等人[9]的工作给了我们很好的启发, 但其方法缺乏系统性.

本文以验证并发对象的强可线性化性质为目的, 开展了一系列的工作. 与可线性化相比, 强可线性化是一个相对较新的概念, 强可线性化与可线性化点以及帮助机制的关系尚不明确. 本文把强可线性化这个概念细分为了固定生效点和单纯帮助两个子概念. 固定生效点这个概念建模了每个方法其生效点都由本方法语句完成的情况. 与之类似的是一个名为固定可线性化点的已知概念. 固定可线性化点把每个方法的生效点固定地指定为方法的原语命令, 这个指定方式偏向“语法”. 而固定生效点只要求每一条执行中方法的生效点是某个方法内部的原语命令, 这个指定方式基于执行, 偏向“语义”. 显然固定可线性化点在概念上真包含于固定生效点. 本文证明了固定生效点蕴含了强可线性化. 作为示例, 本文提出了一个名为双线队列的并发对象, 这个对象有着固定生效点, 但没有固定可线性化点. 因此固定可线性化点在概念上真包含于固定生效点. 此外, 本文还提出了单纯帮助这个性质, 用来建模一些方法的关键步骤由其他方法帮助完成的强可线性化对象. 通过单纯帮助的概念, 我们给出了强可线性化的一种等价刻画, 即一个并发对象是强可线性化的当且仅当其满足固定生效点或满足单纯帮助性质. 本文以碰撞栈[12]为例, 证明了满足单纯帮助性质的并发对象的存在. 通过这些工作, 本文不仅把强可线性化这个性质做了清晰的划分, 还明确了下面的验证算法的适用范围. 强可线性化、固定生效点, 单纯帮助和固定可线性化点这 4 个概念的关系如图 1 所示.

我们开发了两个强可线性化的验证算法. 在第 1 个算法中, 我们通过验证并发对象满足固定可线性化点来验证强可线性化. 算法枚举可能的固定可线性化点并检测按照这个顺序发生的操作序列是否符合顺序规约. 在第 2 个算法中, 我们通过验证并发对象满足固定生效点来验证强可线性化. 为了处理强可线性化定义中的前缀保持性质, 本文将前缀闭合的执行集合组织为一棵树, 并对树递归地检测.

最后, 本文把 Hwang 等人[9]通过具体执行证明并发对象违背强可线性化的方法概括为二分叉区分性执行方法, 并使用这一方法重新证明了 Herlihy&Wing 队列、文献 [7] 中的单读单写寄存器和文献 [8] 中的并发快照对象

违背强可线性化. 针对这 3 个并发对象, 本文直接给出了违背强可线性化的执行. 相比 Golab 等人[3]的工作, 本文给出了一种更加直接的构造性证明方式, 无需考察概率分布不同.

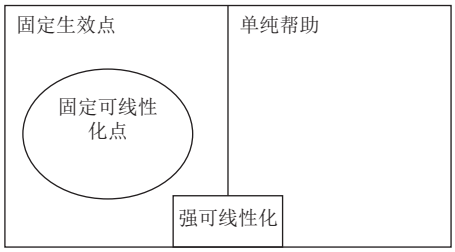


图 1 强可线性化相关概念之间的关系

本文的主要贡献概括如下.

(1) 提出了固定生效点和单纯帮助两个概念, 证明了强可线性化等价于固定生效点和单纯帮助的并集, 证明了固定可线性化点是固定生效点的子概念, 并通过示例说明存在满足单纯帮助但不满足固定生效点的并发对象.

(2) 提出了基于固定可线性化点和基于固定生效点的强可线性化验证算法.

(3) 提出二分叉区分性执行方法并应用这个方法证明了 3 个并发对象违背强可线性化.

2 相关工作

关于强可线性化的研究主要集中在理论方面. 一个研究方向是研究通过特定的基本对象实现强可线性化对象的可能性. 例如, Helmi 等人[13]证明了无法只使用多读单写寄存器实现强可线性化且 non-blocking 的多读寄存器、最大寄存器、快照和计数器. Denysyuk 等人[14]证明了无法只使用多写者寄存器实现可用在 3 个及以上进程的确定性的强可线性化且无等待 (wait-free)[15]的快照、计数器和最大寄存器对象. Attiya 等人[16]证明了对于多写者寄存器、最大寄存器、快照和计数器, 不存在强可线性化且容错的实现. 另一个研究方向是研究强可线性化与其他正确性标准的关系. Rady[17]证明了强可线性化等价于前向模拟. Attiya 等人[18]提出了强观察等价概念, 说明使用观察精化的方法并不会影响并发对象的 hyperproperties[19], 并讨论了强观察等价与强可线性化之间的关系. Vale 等人使用博弈语义给出了强可线性化的另一种定义[20].

Hwang 等人[9]通过给出违背强可线性化的执行的方式证明了 Micheal&Scott 队列和 Harris 链表违反强可线性化. 本文使用二分叉区分性执行方法证明了 Herlihy&Wing 队列, Vidyasankar[7]给出的单读单写寄存器的实现和文献 [8] 中的并发快照对象违背强可线性化, 本文找到的执行违背强可线性化的原因和他们的不同. 以 Herlihy&Wing 队列的违背强可线性化的执行为例, 这些执行对强可线性化的违背的原因是: 由于入队操作可以在当前或未来完成, 在某个时间点队列内部的元素可以处于不同的序; 他们的执行对强可线性化的违背的原因是: 由于出队操作可以在当前或未来完成, 不同执行中同一个出队操作的返回值会不同. 由于二分叉区分性执行方式是对 Hwang 等人[9]的方法的概括, 因此 Hwang 等人[9]给出的强可线性化违背执行也属于二分叉区分性执行. 所以, 二分叉区分性执行是一种更为广泛的方法.

3 并发对象和强可线性化

在本节我们介绍并发对象和强可线性化的概念.

3.1 并发对象

并发系统包含并发运行的客户程序, 客户程序之间通过调用并发对象来完成通信. 一个并发对象提供若干方法以访问该对象. 为了简化起见, 本文假定每个方法有着一个参数并返回一个值. 并发对象和客户程序都可以有私有的内存.

我们首先介绍原语命令的概念. 给定有穷内存集合 X, 有穷方法名集合 M 和有穷的数据域 D, 一个原语命令

有着如下的形式:

$$\tau_i \mid read_i(x,a) \mid write_i(x,a) \mid cas\_suc_i(x,a,b) \mid cas\_fail_i(x,a,b) \mid inv(m,a) \mid res(m,a),$$

其中, $a,b \in D$, $x \in X$, $m \in M$, 并且下标 i 取自然数. 每个原语命令对应并发对象的实现代码中的一个程序步骤, 下标 i 用于区分在实现代码中不同位置的相同命令. $inv(m, a)$ 表示调用命令, 而 $res(m, a)$ 表示返回命令. 为了以标号迁移系统的方式定义语义, 本文在原语命令中包含了命令涉及的值. τ 是对应内部动作的原语命令. 一个 *cas* (compare-and-swap) 命令原子地执行一个读和一个 (可能的) 写动作. 一个成功执行的 *cas* 命令被写为 $cas\_suc((x,a,b)$, 它仅在 x 的值为 a 时执行, 并将 x 的值修改为 b. 一个失败的 *cas* 命令被写为 $cas\_fail(x,a,b)$, 它仅在 x 的值不为 a 时执行, 它的执行不会修改任何内存单元的值.

一个并发对象 O 可以使用五元组 $O = (X_O, M_O, D_O, Q_O, \rightarrow_O)$ 表示, 其中,

- X_O、 M_O 和 D_O 分别是并发对象 O 上的有穷内存集合、有穷方法名集合和有穷数据域.
- $Q_O = \cup_{m \in M_O} Q_m$, 其中 Q_m 是每个方法 $m \in M_O$ 的程序状态集合.
- $\rightarrow_O = \cup_{m \in M_O} \rightarrow_m$ 是每个方法 $m \in M_O$ 中迁移关系的并集.

令 $PCom_O$ 是在 X_O、 M_O 和 D_O 之上的原语命令 (非调用和返回命令) 的集合. 那么, 对于每一个方法 m, 有 $\rightarrow_m \subseteq Q_m \times PCom_O \times Q_m$. 同时, 对于每个数据 $a \in D_O$, 存在一个初始状态 $is_{(m,a)}$ 和一个最终状态 $fs_{(m,a)}$, 并且 $\rightarrow_m$ 中不存在到达 $is_{(m,a)}$ 的迁移或者从 $fs_{(m,a)}$ 出发的迁移. 其中 $is_{(m,a)}$ 表示一个并发对象开始调用方法, 使用参数 a 来调用 m; 而 $fs_{(m,a)}$ 表示方法 m 在结束执行, 然后执行返回步骤并且返回值为 a.

一个最一般客户程序 (most general client) 是一个特殊的客户程序, 它不断以非确定的方式调用任意的方法并使用任意的参数, 用来展示并发对象的所有可能行为. 一个最一般客户程序 C 是一个五元组 $(\varnothing, \mathcal{M}_C, \mathcal{D}_C, \{in_{clt}, in_{lib}\}, \rightarrow_{mgc})$. 这里 $\rightarrow_{mgc} = \{(in_{clt}, inv(m,a), in_{lib}), (in_{lib}, res(m,a), in_{clt}) \mid m \in M_c, a \in D_c\}$ 是迁移关系. in_{clt} 代表当前没有并发对象正在执行, 而 in_{lib} 代表当前有并发对象正在执行.

本文考虑这样的并发系统: 系统包含 n 个进程, 每个进程运行一个最一般客户程序 C_i, 其中 $1 \leqslant i \leqslant n$. 这些最一般客户程序 C_i 都与同一个并发对象 O 交互. 并发系统的操作语义定义为一个标号迁移系统 $[\![O,n]\!]$. 一个标号迁移系统 LTS (labelled transition system) 是一个四元组 $(Q, \Sigma, \rightarrow, q_0)$, 其中 Q 是状态的集合, Σ 是迁移标号的字母表, $\rightarrow \in Q \times \Sigma \times Q$ 是迁移关系而 q_0 是初始状态. LTS 的一条路径 $q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} q_k$ 记录了从初始状态出发到某个状态 q_k 为止的多步迁移, 而一条迹 $\alpha_1 \ldots \alpha_k$ 是一条路径上的迁移标号的连接. $[\![O,n]\!]$ 的详细定义可见电子附录 A (https://github.com/achaocmt/JOS2023/blob/main/JournalOfSoftware-material-submit.doc). 本文称标号迁移系统 $[\![O,n]\!]$ 的一条迹为一个执行.

3.2 强可线性化

并发对象的行为往往表示为其调用动作和返回动作的序列. 一条历史是一个方法调用动作和方法返回动作的有穷序列. 给定一个序列 s, 本文用 $s(i)$ 表示 s 上的第 i 个元素. 我们称历史 h 上的一个调用动作匹配了一个返回动作, 如果在 h 上这个调用动作出现在这个返回动作之前, 它们运行在相同的进程上且使用相同的方法. 我们称一条历史是顺序的, 如果其开始于一个调用动作, 每个调用动作之后是其匹配的返回动作, 而每个返回动作 (除了最后一个动作以外) 之后是一个调用动作. 历史 h 上进程 i 的子历史 $h{\uparrow_i}$ 是 h 在进程 i 的所有动作上的投影. 我们称两条历史 h 和 h'是等价的, 如果对于每个进程 i, $h{\uparrow_i} = h'{\uparrow_i}$ 都成立. 我们称执行中的一个调用动作是待结的如果在这个调用动作之后没有匹配的返回动作. 令 $complete(h)$ 为 h 的包含配对的方法调用动作和方法返回动作的最大子序列.

一个操作 $o = m(a) \Rightarrow b$ 包含了对方法 m 的调用, 参数 a 和返回值 b. 一条历史中的一个操作 o 包括了一对匹配的调用动作 $inv(o) = inv(pid,m,a)$ 和返回动作 $res(o) = res(pid,m,b)$, 这里的 pid 代表动作发生的进程号. 一个顺序规约是一个前缀闭合的操作序列的集合. 给定顺序规约 $Spec$, 令 $history(Spec)$ 为 $Spec$ 的顺序历史的集合, 即 $\{inv(i_1,m_1,a_1) \cdot res(i_1,m_1,b_1) \ldots inv(i_k,m_k,a_k) \cdot res(i_k,m_k,b_k) \mid (m_1(a_1) \Rightarrow b_1) \ldots (m_k(a_k) \Rightarrow b_k) \in Spec,\ i_1,\ldots,i_k \in N\}$. 一条历史 h 引入了一个操作间的 happen-before 序 $<_h$. 我们称历史 h 上 $o_1 <_h o_2$, 如果操作 o_1 的返回动作 $res(o_1)$ 发生在操

作 o_2 的调用动作 $inv(o_2)$ 之前. 给定对象 O, 令 $exec(O,n)$ 为 $[\![O,n]\!]$ 所有有穷执行的集合, 令 $history(O,n)$ 为 $[\![O,n]\!]$ 的所有历史的集合, 即 $history(O,n)$ 是经由将 $exec(O,n)$ 的每条有穷执行投影到调用和返回动作上得到的序列的集合. 一个操作是待结的如果其调用动作是待结的.

给定一条历史 h 和一个顺序规约 $Spec$, 我们称序列 $s \in history(Spec)$ 为 h 对于 $Spec$ 的线性化序列 (linearization), 如果可以通过向 h 末尾添加若干返回动作的方式得到序列 h', 使得: (1) $complete(h')$ 和 s 等价, 且 (2) 历史 h 上的 happen-before 序包含在历史 s 上的 happen-before 序中, 即任取操作 o_1 和 o_2, 如果 $o_1 <_h o_2$, 则 $o_1 <_s o_2$. 我们称一个将序列映射为序列的函数 f 是前缀保持的 (prefix preserving), 如果任取序列 s 和 s', 如果 s 是 s'的前缀且 $f(s)$ 和 $f(s')$ 都有定义, 则 $f(s)$ 是 $f(s')$ 的前缀. 给定一个集合 S, 令 $close(S)$ 为 S 的前缀闭包. 给定一条执行 e, 令 $history(e)$ 为将 e 投影到调用和返回动作的结果. 基于以上这些术语, 本文给出文献 [3] 中强可线性化的定义.

定义 1 (**强可线性化**[3]). 执行集合 E 是强可线性化到顺序规约 $Spec$ 的, 如果存在一个函数将 $close(E)$ 中的执行映射到集合 $history(Spec)$, 并满足:

(1) 对于任何执行 $e \in close(E)$, $f(e)$ 是 $history(e)$ 对于 $Spec$ 的线性化序列.

(2) 函数 f 是前缀保持的.

这样的函数 f 被称为强可线性化函数. 如果执行集合 $exec(O,n)$ 是强可线性化到顺序规约 $Spec$ 的, 则称对象 O 在 n 进程下强可线性化.

除了正确性之外, 并发对象的另一个重要性质是活性, 代表性的活性性质包括无等待 (wait-free)、无锁 (lock-free) 和无障碍 (obstruction-free) 等[15]. 在后面章节会涉及无障碍的活性性质.

4 强可线性化的分类

强可线性化的定义仅要求存在强可线性化函数, 对并发对象本身的特征缺乏刻划. 在本节从并发对象本身的行为特性出发对强可线性化进行刻划. 提出了固定生效点和单纯帮助两个概念, 并证明强可线性化等价于这两个概念的并集. 已有的固定可线性化点在概念上包含于固定生效点. 进一步, 通过构造双线队列对象证明了固定可线性化点在概念上真包含于固定生效点. 以碰撞栈[12]为例证明了固定生效点在概念上真包含于强可线性化.

4.1 固定生效点与固定可线性化点

固定生效点这个概念力图描述强可线性化并发对象中那些关键操作都由方法自己完成的并发对象. 给定序列 s, 令 $s[i,j]$ 为序列从 $s(i)$ 到 $s(j)$ 的子序列.

对于一个并发对象 O, 我们称一个将执行映射到真假值的函数 f 是固定点函数, 如果:

(1) 对 O 的每一条执行 e 和进程 i, 对 $e{\uparrow_i}$ 的每一条从状态 $is_{(m,a)}$ 出发到状态 $fs_{(m,b)}$ 为止的子序列 p, 在 p 上存在且只存在一个原语命令被函数 f 映射到 true, 而 p 中的其他命令将被 f 映射到 false. 在 $e{\uparrow_i}$ 的最后一个调用动作无匹配返回时, 对从这个 $is_{(m,a)}$ 出发执行到最后一个进程 i 的动作为止的子序列 p, 在 p 上存在最多一个原语命令被函数 f 映射到 true, 而 p 中的其他命令将被 f 映射到 false.

(2) 对每一对执行 e_1 和 e_2, 如果 $e_1[0,i]$ 是 e_2 的前缀, 则 e_2 被 f 映射为真的原语命令是 $e_1[0,i]$ 被 f 映射为真的原语命令的超集.

给定并发对象 O 的固定点函数 f, 我们称函数 g 是其恢复函数, 如果满足如下条件: 对于 O 的每一条执行 e 和进程 i, 对 $e{\uparrow_i}$ 的每一条从状态 $is_{(m,a)}$ 出发到状态 $fs_{(m,b)}$ 为止的子序列 p, 设在 p 上满足 $f(c) = \text{true}$ 的原语命令 c 对应的事件为 $p[i]$, 则:

(1) 对任意 $0 \leqslant j < i$, $g(p[0,j]) = \odot$. 这里 $\odot$ 是一个特殊符号.

(2) 对任意 $i \leqslant j < |p|$, $g(p[0,j]) = m(a) \Rightarrow b$. 这里 $|p|$ 为 p 的长度.

给定并发对象 O 的固定点函数 f 和恢复函数 g, 以及 $[\![O,n]\!]$ 上的一条执行 e, 令 $fep(e)$ 为按照如下方式构造的序列.

(1) 从 e 中删除所有被函数 f 映射为 false 的事件, 得到序列 $e_1 \ldots e_k$.

(2) 对任意 $1 \leqslant i \leqslant k$, 令 l_i 为如下方式得到的序列: 先将 e 投影到 e_i 所在进程的动作上, 然后取从 e_i 之前第 1 个调用直到 e_i 的子序列.

(3) $fep(e) = g(l_1)\ldots g(l_k)$.

基于这些概念, 本文给出固定生效点的定义.

定义 2 (固定生效点). 如果并发对象有着固定点函数和恢复函数, 使得对于 $[\![O,n]\!]$ 的每一条执行 e, $fep(e)$ 都是 $history(e)$ 的可线性化序列, 则称并发对象 O 上有着固定生效点 (fixed effective point).

固定生效点以一种前缀保持的方式 (见固定点函数定义的条件 2) 描述了那些生效点在自身语句内的并发对象. 有一个相似 (但事实上不同, 见第 4.3 节的定理 3) 的已有概念称为固定可线性化点. 固定可线性化点在每个方法内选择一个原语命令, 使得方法在执行这条原语命令时“生效”. 或者说, 对并发对象的每一条执行 e, 令 s 为将 e 投影到这些被选择的原语命令得到的序列, 则将 s 中这些原语命令替换为对应操作的调用和返回动作所得到的序列就是 e 的可线性化序列. 对这些原语命令的选择并不与具体的执行绑定. 显然, 固定可线性化点是固定生效点的子概念, 因为不难证明固定可线性化点概念中选择原语命令的函数本身也是一个固定点函数, 如下面引理所述. 由于证明非常显然, 这里不予赘述.

引理 1. 如果一个并发对象有固定可线性化点, 则其有固定生效点.

下面定理说明了固定生效点蕴含了强可线性化. 这是符合固定生效点的定义直觉的.

定理 1. 如果并发对象 O 存在固定生效点, 那么 O 是强可线性化的.

证明: 我们通过证明 fep 函数可以作为强可线性化函数的方式证明此引理.

由定义可知, fep 是从一条执行到一个操作序列的函数. 因为 O 有固定生效点, 所以对 $[\![O,n]\!]$ 的任何一条执行 e, $fep(e)$ 是 $history(e)$ 的线性化序列.

给定 O 上的两条执行 e_1 和 e_2, 假定 e_1 是 e_2 的前缀. 固定点函数 f 是定义在原语命令上的, 显然 $f(e_1)$ 是 $f(e_2)$ 的前缀. 假设 $e_1 = \alpha_1 \cdot \ldots \cdot \alpha_k$, $e_2 = \alpha_1 \cdot \ldots \cdot \alpha_{k+u}$. 根据 g 和 fep 的定义, 不难看出 $fep(e_1)$ 是 $fep(e_2)$ 的前缀, 或者说 fep 是前缀保持的. 证毕.

4.2 单纯帮助

由定理 1 可知, 固定生效点是强可线性化的一个充分条件. 那么, 固定生效点是否是强可线性化的一个必要条件呢? 在本节我们会看到该命题并不成立, 因为我们证明了碰撞栈[12]是强可线性化的, 但不满足固定生效点.

对于那些满足强可线性化却不满足固定生效点的并发对象, 本文关注这样的子类: 某个方法的生效点不在自己方法内, 而在其他方法中. 为此我们定义单纯帮助这个概念.

定义 2 (单纯帮助). 我们称一个并发对象是单纯帮助的, 如果其满足强可线性化, 并且对其任何一个强可线性化函数 f, 一定存在某个执行 e 和进程 i 的动作 α, 使得 $f(e\cdot\alpha) = f(e)\cdot s$, 且 s 中包含了不属于进程 i 的操作.

文献 [4] 指出, 并发对象的可线性化点位置不固定往往来自两个原因, 一个原因是帮助现象, 另一个原因是未来决定的可线性化点现象 (future dependence linearization point). 本文意图使用单纯帮助性质来刻画只有帮助没有未来决定的现象.

我们下面介绍碰撞栈, 并证明其是强可线性化的, 但没有固定生效点. 因此, 碰撞栈是单纯帮助的一个例子. 碰撞栈维持着一个由节点 *Node* 组成的链表, 以及两个数组 *opInfos* 和 *collision*. 每个 *Node* 类型的对象存储值 *val* 和指向 *Node* 的指针 *next*. 每个 *opInfo* 类型的对象存储一个 *Op* 类型的值和一个指向 *Node* 的指针, 这里 *Op* 是枚举类型, 包含 *NONE*、*PUSH* 和 *POP* 这 3 个值. *opInfo* 类型的对象存储了一个待执行的操作. 令 *Top* 为栈顶指针, *opInfos* 为 *opInfo* 类型的值的数组, *collision* 为值为进程 *id* 的数组.

直观上, 碰撞栈的入栈操作 (出栈操作) 会反复执行这样的循环: 先尝试向 *Top* 指向的栈中放入元素 (取出元素); 在不成功时尝试去遇到一个并发的出栈操作 (入栈操作), 并且这一对入栈和出栈操作同时生效并抵消彼此, 这种碰撞现象发生在 *opInfos* 数组中, 不会改动由 *Top* 指向的栈. 碰撞栈的伪代码如下, 这里的 *n*:*ptr_to Node* 表示 *n* 是一个 *ptr_to_Node* 类型的变量, *getPosition*() 返回一个随机的下标, *delay* 进行随机的延时. 我们用粗体标识关键字.

伪代码 1. 碰撞栈.

```
1. method push(v)
输入: 待入栈元素 v;
输出: 无.
2. n:ptr_to Node := new Node();
3. n→val := v; //构造包含待入栈数据的节点
4. info:opInfo := (PUSH, n); //记录操作类型和信息
5. loop
6.    if tryPush(n) then exit //尝试通过修改 Top 指针入栈
7.    if tryElimination(&info) then exit //尝试通过碰撞入栈
8. endloop
9. return;

10. method tryPush(n:n:ptr_to Node)
输入: 包含待入栈元素的节点;
输出: 尝试通过修改 Top 指针入栈成功与否的结果.
11. ss:ptr_to Node := Top;
12. n→next := ss;
13. return cas(&Top, ss, n); //通过修改 Top 入栈

14. method pop()
输入: 无;
输出: 出栈元素, 或在栈为空时出栈 empty.
15. info:opInfo := (POP, null); //记录操作类型
16. loop
17.    if tryPop(info.node) then exit //尝试通过修改 Top 指针出栈
18.    if tryEliminate(&info) then exit //尝试通过碰撞出栈
19. endloop
20. if info.node = null then return empty
21. else
22.    v:Val := info.node→val;
23.    return v;
24. fi

25. method tryPop(n)
输入: 存放出栈结果的节点;
输出: 尝试通过修改 Top 指针出栈成功与否的结果.
26. ss:ptr_to Node := Top;
27. if ss = null then //如果通过读 Top 可以判断栈为空
28.    n := null;
29.    return true;
```

```
30. else
31.     n := ss;
32.     ssn:ptr_to Node := ss→next;
33.     return cas(&Top, ss, ssn); //通过修改 Top 指针出栈

34. method tryElimination(pinfo:ptr_to opInfo)
输入: 待碰撞的操作类型及数据;
输出: 碰撞是否成功.
35. opInfos[mypid] := *pinfo; //在 opInfos 中记录当前待碰撞操作的信息
36. pos := getPosition();
37. repeat him := collision[pos] until cas(collision[pos], him, mypid) //选择待碰撞的进程
38. qinfo := opInfos[him]; //读取待碰撞的操作的信息
39. if qinfo.op+pinfo→op = PUSH + POP then
40.     if cas(&opInfos[myid], *pinfo, (NONE, pinfo→node)) then
41.         if cas(&opInfos[him], qinfo, (NONE, pinfo→node)) then //碰撞成功
42.             pinfo→node := qinfo.node;
43.             return true;
44.         else return false;
45.         fi
46.     else //自身是被碰撞的节点
47.         pinfo→node := opInfos[myid].node;
48.         return true;
49.     fi
50. fi
51. delay;
52. if !cas(&opInfos[myid], *pinfo, (NONE, pinfo→node)) then //自身是被碰撞的节点
53.     pinfo→node := opInfos[myid].node;
54.     return true;
55. else return false;
56. fi
```

下面的引理说明碰撞栈是强可线性化的. 这里 *Stack* 是栈的顺序规约, 每个 *Stack* 中的序列是一个 LTS $(Q,\Sigma,\rightarrow,q_0)$ 的迹, 每个 Q 中的状态是某个有穷字母表中元素的有穷序列, 初始状态 q_0 是空串, 迁移关系包含如下 3 种迁移: $s \xrightarrow{push(a)} s\cdot a$ 、 $s\cdot a \xrightarrow{pop()\Rightarrow a} s$ 和 $\varepsilon \xrightarrow{pop()\Rightarrow \text{empty}} \varepsilon$, 这里 ε 代表空串.

引理 2. 碰撞栈强可线性化到栈的顺序规约 *Stack*.

证明: 我们通过给出强可线性化函数的方式证明此引理.

我们以如下方式构造强可线性化函数 *f*: 给定执行 e 和动作 α, 那么:

- 如果 α 是进程 i 在执行 $push(a)$ 的过程中在 *tryPush*(a) 中被成功执行的 *cas* 语句 (第 13 行), 则 $f(e\cdot\alpha)=f(e)\cdot push(i,a)$. 这里我们向操作中额外写入进程 *id* 以处理多个进程可能执行同一个操作的情况.
- 如果 α 是进程 i 在执行 $pop()\Rightarrow a$ 的过程中在 *tryPop*() 中被成功执行的 *cas* 语句 (第 33 行), 则 $f(e\cdot\alpha)=f(e)\cdot(pop(i)\Rightarrow a)$.
- 如果 α 是进程 i 在执行 $push(a)$ 的过程中在 *tryElimination*() 中被成功执行的 *cas*(&*opInfos*[*him*],*qinfo*,

($NONE, pinfo \to node$)) 语句 (第 41 行). 显然 *tryElimination*() 中选择的 *him* 值对应某个进程 j 的 *pop*() 操作. 则 $f(e\cdot\alpha)=f(e)\cdot push(i,a)\cdot(pop(j)\Rightarrow a)$. 从伪代码显然可知 α 发生在这一对 *push* 和 *pop* 的调用和返回之间.

如果 α 是进程 i 在执行 *pop*() 的过程中在 *tryElimination*() 中被成功执行的第 41 行语句. 显然 *tryElimination*() 中选择的 *him* 值对应某个进程 j 的 *push*(a) 操作. 则 $f(e\cdot\alpha)=f(e)\cdot push(j,a)\cdot\ (pop(i)\Rightarrow a)$. 从伪代码显然可知 α 发生在这一对 *push* 和 *pop* 的调用和返回之间.

● 对其他情况, $f(e\cdot\alpha)=f(e)$.

显然, f 是前缀保持的.

$f(e)$ 是 *history*(e) 的可线性化序列 (详细证明过程见电子附录 B https://github.com/achaocmt/JOS2023/blob/main/JournalOfSoftware-material-submit.doc). 证毕.

引理 2 说明碰撞栈是强可线性化的, 下面的引理说明碰撞栈不满足固定生效点. 因此, 碰撞栈满足单纯帮助.

引理 3. 碰撞栈不满足固定生效点.

证明: 反证, 假定碰撞栈满足固定生效点, 并令 f 为其固定点函数.

令执行 e 为如下执行:

● 并发调用 *push*(a), *push*(b) 和 *pop*().

● *push*(b) 执行 *tryPush*() 直到第 1 次 *cas* 语句 (第 13 行) 之前, *pop*() 执行 *tryPop*() 直到第 1 次 *cas* 语句 (第 33 行) 之前. 然后, *push*(a) 执行直至完成.

● *push*(b) 执行 *tryPush*() 失败, 开始执行 *tryElimination*() 直至执行完 *cas*(&*opInfos*[*myid*], $*pinfo$, (*NONE*, $pinfo \to node$)) 语句 (第 40 行), 且 *him* 为 *pop*() 所在的进程. *pop*() 执行 *tryPop*() 失败, 开始执行 *tryElimination*() 直至执行第 40 行, 且 him 为 *push*(b) 所在的进程.

在执行 e 上, *Top* 指向的栈存储 a. *push*(b) 和 *pop*() 都正在尝试碰撞且碰撞的对象是彼此.

我们分析 $f(e)$ 的各个可能取值.

● 如果 $f(e)$ 中包含 *pop*() 且 *pop*() 返回 b, 则我们可在后续执行中调用 *push*(c) 和另外两个 *pop*(), 为了区分, 分别称其为 *pop*1() 和 *pop*2(). *push*(c) 调用 *tryPush*(), *pop*1() 和 *pop*2() 调用 *tryPop*(), 且 *tryPush*() 成功而 *pop*1() 和 *pop*2() 调用的 *tryPop*() 失败. 之后 *push*(c) 执行直到返回, *pop*1() 执行碰撞失败, 再次执行 *tryPop*() 并最终返回 c. *pop*2() 执行 *tryElimination*() 且尝试的碰撞对象为 *push*(b) (即 *him* 设置为 *push*(b) 所在的进程) 并和 *push*(b) 碰撞成功 (即 *pop*2() 执行第 41 行语句成功), 之后 *pop*2() 返回 b. 然后 e 中的 *pop*() 碰撞失败并继续执行直至返回 empty. 因此 e 中的 *pop*() 返回 empty, 导致矛盾.

如果 $f(e)$ 中包含 *pop*() 且 *pop*() 返回 empty, 则我们可以在后续执行中让 e 中的 *pop*() 和 *push*(b) 碰撞成功, 之后该 *pop*() 返回 b, 导致矛盾.

因此 $f(e)$ 中没有 *pop*().

● 如果 $f(e)$ 中包含 *push*(b). 我们可在后续执行中执行如下动作: 首先调用一个新的 *pop*() 并执行直至返回 a, 为了区分不同的 *pop*() 我们称这个操作为 *pop*3(). 然后调用一个新的 *pop*() 直至返回 empty, 我们称这个操作为 *pop*4(). 然后调用 *push*(c) 并执行成功. 然后并发地调用两个 *pop*(), 称为 *pop*5() 和 *pop*6(). *pop*5() 和 *pop*6() 各自调用 *tryPop*() 且并发地使用 *cas* 语句尝试修改 *Top*, *pop*6() 成功并返回 c, *pop*5() 失败并开始执行 *tryElimination*(). *pop*5() 在 *tryElimination*() 中设置 *him* 为 *push*(b) 所在的进程并和 *push*(b) 碰撞成功, 进而返回 b. 之后, e 中的 *pop*() 返回 empty, e 中的 *push*(b) 返回. 由于 *pop*3() 和 *pop*4() 有着 happen-before 关系, 且 *pop*4() 和 *pop*5() 以及 *pop*6() 有着 happen-before 关系, 我们可以确定在 $f(e)$ 之后栈先被清空然后 b 才被出栈, 所以 $f(e)$ 中不应该包含 *push*(b), 而已知 $f(e)$ 中包含 *push*(b), 导致矛盾. 因此 $f(e)$ 中不含有 *push*(b).

● 因此, $f(e)=push(a)$.

令 α_1 为 e 中的 *pop*() 成功执行的第 41 行语句. 下面证明碰撞栈不满足固定生效点. 我们采取反证法, 假定碰撞栈满足固定生效点. 则我们在 $e\cdot\alpha_1$ 的后续执行 $e\cdot\alpha_1\cdot l$ 中单独执行 *pop*() 所在进程直至返回 b. 已知 $f(e)=push(a)$, 且在 $\alpha_1\cdot l$ 中我们未执行 *push*(b) 的语句, 但由 *deq*() 返回 b 知, $f(e\cdot\alpha_1\cdot l)$ 中一定包含 *push*(b). 这与固定生效点的定

义矛盾. 因此, 碰撞栈不满足固定生效点. 证毕.

下面定理说明, 固定生效点概念和单纯帮助概念构成了强可线性化概念.

定理 2. 一个并发对象是强可线性化的, 当且仅当其满足固定生效点或单纯帮助.

证明: 首先证明固定生效点或单纯帮助蕴含强可线性化. 这直接来自定理 1 和单纯帮助的定义.

然后我们证明强可线性化蕴含固定生效点或单纯帮助. 给定一个强可线性化的并发对象 O. 如果对 O 的每一个强可线性化函数 f, 存在执行 e 和进程 i 的动作 α, 使得 $f(e\cdot\alpha)=f(e)\cdot s$, 且 s 中包含了不属于进程 i 的操作, 则 O 满足单纯帮助.

否则, 一定存在一个强可线性化函数, 使得对于每个执行 e 和进程 i 的动作 α, $f(e\cdot\alpha)$ 相比 $f(e)$ 不会多出其他进程的操作. 由于本进程上之前操作一定在其返回动作之前生效, 因此 $f(e\cdot\alpha)$ 相比 $f(e)$ 最多增加一个本进程当前操作. 因此基于 O 的强可线性化函数 f 我们可以构造这样的固定点函数 f': 给定执行 e 和动作 α, 如果 $f(e\cdot\alpha)=f(e)\cdot(m(a)\Rightarrow b)$, 则 f'将 $e\cdot\alpha$ 中的 α 映射为 true, 将 α 所在的操作的其他动作映射为 false. 可以很容易证明这个函数 f'是固定点函数. 证毕.

4.3 关于固定可线性化点和固定生效点的进一步讨论

学术界通常用可线性化点这个说法指代方法原子生效的时刻. 固定可线性化点要求在“语法”层次直接将方法的可线性化点固定在原语命令上, 而固定生效点则定义在更加“语义”的层次上, 其在每条路径上为每个方法指定生效点. 这两者有着微妙的不同. 例如, 是否存在两条不同的执行 e_1 和 e_2, 以及进程 i 和下标 i_1,i_2,i_3,i_4, 使得 $e_1\uparrow_i[i_1,i_2]=e_2\uparrow_i[i_3,i_4]$, 但对 $e_1\uparrow_i[i_1,i_2]$ 和 $e_2\uparrow_i[i_3,i_4]$ 选择的生效点却不同.

在本节, 我们考察固定可线性化点和固定生效点这两者的关系, 并给出一个这样的例子.

本文提出一种名为双线队列的并发对象. 直观上, 双线队列同时使用两个强可线性化的队列对象实例 O_1 和 O_2 作为内部存储, 每个 *enq*(a) 操作会先后把元素 a 入队到 O_1 和 O_2 中. 然而在 *deq*() 的时候只会固定地使用 O_1 或 O_2 之一来出队. 为此, 本文需要一个额外的 *trigger*() 方法, 它的参数取自{1, 2}, 用来指定出队时使用 O_1 还是 O_2. 本文假定在对应的顺序规约中, 需要先执行 *trigger*() 方法来决定 *deq*() 时使用 O_1 还是 O_2, 然后才可以入队和出队, 第 1 个生效的 *trigger*() 方法之外的其他 *trigger*() 方法不起实际作用.

一种很显然的强可线性化队列的实现方式为, 为一个可线性化的队列实现添加一把全局锁, 每次执行入队操作时需先获取锁, 然后执行原本的入队操作, 再释放锁. 对出队操作以类似方式改进. 令 O_1 和 O_2 分别是这样的强可线性化队列的两个实例.

双线队列的伪代码如下. 变量 *choice* 存储对使用哪个队列的选择结果, 其初始值为 0.

伪代码 2. 双线栈.

```
1. method enq(v).
输入: 待入队的值;
输出: 无.
2. O1 → enq(v); // O1 入队
3. O2 → enq(v); // O2 入队
4. return;

5. method deq()
输入: 无;
输出: 出队的值.
6. wait until choince ≠ 0;
7. if choice=1 then result := O1 → deq() //数据存储在 O1 中, 因此执行 O1 的出队操作
8. if choice=2 then result := O2 → deq() //数据存储在 O2 中, 因此执行 O2 的出队操作
```

9. **return** *result*;

10. **method** *trigger*(*v*)
输入: 选择的队列编号;
输出: 无.
11. **if** $v \in \{1,2\}$ **then** *cas*(&*choice*, 0, *v*)
12. **return**;

下面引理说明双线队列是强可线性化的. 这里 *DQueue* 是双线队列的顺序规约, 每个 *DQueue* 中的序列是一个 LTS $(Q,\Sigma,\rightarrow,q_0)$ 的迹, $Q \subseteq \Sigma_1^* \times \{0,1,2\}$ 是状态的集合, 这里的 Σ_1 是一个有穷字母表, 初始状态 $q_0=(\varepsilon,0)$, 迁移关系包含如下几种迁移:

- $v \in \{1,2\}$ 时, 有迁移 $(\varepsilon,0) \xrightarrow{trigger(v)} (\varepsilon,v)$.
- $(s,x) \xrightarrow{trigger(v)} (s,x)$. 这里 $x \in \{1,2\}$, s 是一个序列.
- $v \in \{1,2\}$ 时, 有 $(s,v) \xrightarrow{enq(a)} (a \cdot s,v)$、 $(s \cdot \mathrm{a},v) \xrightarrow{deq() \Rightarrow a} (s,v)$ 和 $(\varepsilon,v) \xrightarrow{deq() \Rightarrow \mathrm{empty}} (\varepsilon,v)$.

引理 4. 双线队列强可线性化到 *DQueue*.

证明: 我们通过构造强可线性化函数的方式证明此引理.

我们以如下方式构造强可线性化函数 *f*: 给定执行 *e* 和动作 α,

- 如果 α 是第 1 个也是唯一一个更改 *choice* 的 *trigger*() 方法中的 *cas* 语句 (第 11 行), 且由进程 *i* 执行, 则 $f(e \cdot \alpha) = trigger(i,v)$, *v* 是 *trigger*() 方法的参数.

如果 α 是其他 *trigger*() 方法中的 *cas* 语句 (第 11 行), 且由进程 *i* 执行, 则 $f(e \cdot \alpha) = f(e) \cdot trigger(i,v)$, *v* 是 *trigger*() 方法的参数.

- 如果 *choice* 的值为 1, α 是进程 *i* 的 $O_1 \rightarrow enq(v)$ 中成功执行的 *unlock*() 方法的最后一条指令, 则 $f(e \cdot \alpha) = f(e) \cdot enq(i,v)$. 如果 *choice* 的值为 2, α 是进程 *i* 的 $O_2 \rightarrow enq(v)$ 中成功执行的 *unlock*() 方法的最后一条指令, 则 $f(e \cdot \alpha) = f(e) \cdot enq(i,v)$.

- 如果 *choice* 的值为 1, α 是进程 *i* 的 $O_1 \rightarrow deq()$ 中成功执行的 *unlock*() 方法的最后一条指令, 则 $f(e \cdot \alpha) = f(e) \cdot (deq(i) \Rightarrow v)$, 这里 *v* 是 *deq* 的返回值. 如果 *choice* 的值为 2, α 是进程 *i* 的 $O_2 \rightarrow deq()$ 中成功执行的 *unlock*() 方法的最后一条指令, 则 $f(e \cdot \alpha) = f(e) \cdot (deq(i) \Rightarrow v)$.

- 对其他情况, $f(e \cdot \alpha) = f(e)$.

显然, *f* 是前缀保持的. $f(e)$ 确实是 $history(e)$ 的可线性化序列 (证明过程见电子附录 C https://github.com/achaocmt/JOS2023/blob/main/JournalOfSoftware-material-submit.doc). 证毕.

根据引理 4 的证明, 显然双线队列满足固定生效点. 下面引理说明了双线队列并不满足固定可线性化点.

引理 5. 双线队列不满足固定可线性化点.

证明: 反证, 假定双线队列满足固定可线性化点.

令执行 e_1 为如下执行: 调用 *trigger*(1) 直至执行完成. 然后进程 1 调用 *enq*(*a*), 进程 2 调用 *enq*(*b*). *enq*(*a*) 执行完毕 $O_1 \rightarrow enq(a)$, 然后, *enq*(*b*) 执行完毕 $O_1 \rightarrow enq(b)$ 和 $O_2 \rightarrow enq(b)$, 之后, *enq*(*a*) 执行完毕 $O_2 \rightarrow enq(a)$. 然后 *enq*(*a*) 和 *enq*(*b*) 分别返回, 再串行地调用两个 *deq*(), 先后返回 *a* 和 *b*.

令执行 e_2 为如下执行: 调用 *trigger*(2) 直至执行完成. 然后进程 1 调用 *enq*(*a*), 进程 2 调用 *enq*(*b*). *enq*(*a*) 执行完毕 $O_1 \rightarrow enq(a)$, 然后, *enq*(*b*) 执行完毕 $O_1 \rightarrow enq(b)$ 和 $O_2 \rightarrow enq(b)$, 之后, *enq*(*a*) 执行完毕 $O_2 \rightarrow enq(a)$. 然后 *enq*(*a*) 和 *enq*(*b*) 分别返回, 再串行地调用两个 *deq*(), 先后返回 *b* 和 *a*.

在 e_1 和 e_2 执行的过程中, *enq*(*a*) 和 *enq*(*b*) 的所有指令完全一样, 但从后续 *deq*() 的结果可知, 在 e_1 和 e_2 中入队元素的顺序不同. 因此, 不管如何为 *enq*(*a*) 和 *enq*(*b*) 指定固定可线性化点, 都一定会导致矛盾. 证毕.

由引理 5 可知, 固定可线性化点在概念上真包含于固定生效点, 正如如下定理所述. 这个定理由引理 5 以及双

线队列满足固定生效点可以直接证明.

定理 3. 存在一个强可线性化并发对象, 其满足固定生效点, 但不满足固定可线性化点.

4.4 关于恢复函数的讨论

恢复函数是非常必要的. 在证明强可线性化的过程中, 需要在操作 *op* 的生效点获取 *op* 的调用参数和返回值. 如果操作 *op* 中含有返回动作, 那么可以通过这个动作来获得其返回值. 否则, 需要一种机制来获取 *op* 的返回值, 这就需要恢复函数对计算 *op* 的返回值的过程进行建模. 恢复函数被定义为从一段程序步骤中恢复返回值, 而不是从单纯的一条指令恢复返回值.

下面将举例说明以这样的方式定义恢复函数的必要性.

构造并发对象扫描栈, 其伪代码如下所示. 这个并发对象的内部实现类似于栈的实现机制, 共享变量包含 *A* 和 *Top*, 其中 *A* 是 *Node* 类型的链表, 用来存储栈中内容, 而 *Top* 表示当前的栈顶. 与栈的实现不同的是, 扫描栈仅包含一个方法 *pushAndScan(a)*, 其中 *a* 是被入栈的数值, 该方法执行入栈并返回更新后栈的内容序列. 每个 *Node* 类型的对象存储值 *data* 和指向 *Node* 类型的指针 *next*. 变量 *curResult* 是一个存储值的序列.

伪代码 3. 扫描栈.

```
1. method pushAndScan(v)
输入: 待入栈的值
输出: 更新后栈的内容
2. n:Node := new Node(v);
3. while true do
4.    oldTop:ptr_to Node := Top;
5.    n.next := oldTop;
6.    curN:ptr_to Node := oldTop;
7.    curResult := curN.data concatenates v; //开始读取栈内容
8.    while curN.next != null do
9.        curResult := curN.next.data concatenates curResult;
10.       curN := curN.next;
11.   end while
12.   if cas(&Top, oldTop, n) then //尝试入栈元素
13.       break;
14.   fi
15. end while
16. return curResult;
```

扫描栈的顺序规约的定义如下: 每个抽象状态 l 是一个有穷字母表的有穷序列, 代表栈内容. 迁移关系是 $l \xrightarrow{pushAndScan(a)\Rightarrow a\cdot l} a\cdot l$. 在 *pushAndScan*() 中成功执行的 *cas* 语句 (第 12 行) 是该方法的生效点. 由于生效点是固定的, 所以扫描栈是强可线性化的.

但是, 在生效点处的原语命令 *cas*(&*Top*, *oldTop*, *n*) 中, 仅包含待修改结点的地址 *Top*、*Top* 的旧有内容 *oldTop* 和新节点 *n*, 无法看到当前栈中所有的内容. 所以需使用内部 while 循环中对栈内容的读取 (第 7–11 行) 才能有效地计算出返回值.

5 强可线性化验证和检测算法

检测并发对象是否满足可线性化性质的算法和工具已有相当多的研究[21–23], 但是目前尚缺乏并发对象是否满

足强可线性化的验证算法. 本节给出两个自动化地检测和验证强可线性化的算法.

5.1 通过固定可线性化点检测和验证强可线性化

固定可线性化点为每个方法固定了原子地生效的时间点, 因此其天然有着前缀保持的性质. 基于引理 1 和定理 1, 固定可线性化点蕴含了强可线性化. 本小节提出一个检测并验证并发对象是否满足强可线性化的方法, 算法的思想为检测是否存在固定可线性化点.

方法 *checkFLP*(*e*, *Spec*, *flp*) 的伪代码在下面给出, 它用来检测给定固定可线性化点的选取方案在一条执行上是否有效. 其参数是一条执行 *e*, 顺序规约 *Spec*, 以及一个为每个方法分配固定可线性化点的方案 *flp*. 可以认为 *flp* 是原语命令的集合. 为了保证在每个固定可线性化点处可以得到其操作的返回值, 这里我们假定执行 *e* 中只包含完整的历史, 即执行 *e* 中每个调用动作都有匹配的返回动作. *checkFLP*(*e*, *Spec*, *flp*) 返回一个 *bool* 值, 表示依照 *flp* 指定的操作序列是否符合 *Spec*.

算法 1. *checkFLP*.

1. **method** *checkFLP*(*e*, *Spec*, *flp*)
输入: 固定可线性化点方案 *flp*, 执行 *e* 和规约顺序 *Spec*;
输出: 检测依据固定可线性化点方案 *flp*, 执行 *e* 是否强可线性化到顺序规约 *Spec*.
2. let $\alpha_1 \cdot \ldots \cdot \alpha_k$ be the projection of *e* into *flp*;
3. let $m_i(a_i) \Rightarrow b_i$ be the operation of method name, argument and return value of α_i ; //从 *e* 和 *S* 构造操作序列. 通过查询执行 *e* 为每个 *flp* 中的原语命令得到其调用参数和返回值
4. **return** $(m_1(a_1) \Rightarrow b_1) \rightarrow (m_k(a_k) \Rightarrow b_k) \in Spec$; //检测操作序列是否属于 *Spec*

方法 *hasFLP*(*E*, *Spec*, *O*) 检测有穷执行集合 *E* 是否符合固定可线性化点. 参数 *Spec* 是顺序规约, *O* 是并发对象. *CandidateFLP*(*O*) 是一个集合, 其每个元素 $S \in CandidateFLP(O)$ 是一个原语命令的集合, 并且对并发对象 *O* 的每个方法 *m* 指定原语命令作为可线性化点. *hasFLP*(*E*, *Spec*, *Q*) 遍历每个可能的固定可线性化点选择方案, 并调用 *hasFLP* 对每一条执行进行检测. 如果其中一个方法对所有执行成立, 则返回 true. 否则, 返回 false.

算法 2. *hasFLP*.

1. **method** *hasFLP*(*E*, *Spec*, *O*).
输入: 有穷执行集合 *E*, 顺序规约 *Spec*, 并发对象 *O*;
输出: 是否存在 *O* 的固定可线性化点方案, 使得有穷执行集合 *E* 强可线性化到顺序规约 *Spec*.
2. *result*:*bool* := false;
3. begin: **for each** $S \in CandidateFLP(O)$ //检测每一个可能方案
4. **for each** $e \in E$
5. **if** !*checkFLP*(*e*, *Spec*, *S*) **continue** begin
6. **end for**
7. **return** true;
8. **end for**
9. **return** false;

通过上述方法可以实现对并发对象是否满足强可线性化性质的自动化检测. 算法的正确性来自引理 1 和定理 1.

如果数据域和内存大小都是有穷的, 且每个进程只能执行限界数目的调用和返回, 则该算法成为一个验证并发对象是否由于固定可线性化点而满足强可线性化的模型检测方法.

5.2 通过固定生效点验证强可线性化

固定生效点不同于固定可线性化点, 后者被固定在方法中的原语命令上, 而前者则可能随着执行的不同发生在方法的不同原语命令上. 之前的双线队列的例子反映了这一点. 而且, 固定生效点要求某种意义上的前缀保持(见其定义中的条件 2). 这使得固定生效点的验证相比固定可线性化点的验证更加困难. 本节提出一个验证并发对象强可线性化的算法, 算法的思想为检测是否存在固定生效点.

给定两条执行 e 和 e', 我们称关系 $e \prec e'$ 成立, 如果 e 是 e'的前缀. 给定一个前缀闭合的有穷执行集合 E, 本文根据 $\prec$ 关系构造一棵树 T=(N, R), 其中 N 是树节点的集合, R 是树中边的集合. 树的每个节点是 E 中的一条执行, 树的根节点是空执行. $(e,e') \in R$ 如果存在某个动作 α, 使得$e' = e \cdot \alpha$. 给定这样的一棵树 T=(N, R) 和树的节点 e, 集合 *possibleFEP*(T, e) 表示点 e 处可能的生效点方案集合, 每个 *possibleFEP*(T, e) 中的元素是一个原语命令的集合 S, 使得:

- 在 e 对每个进程的投影上从状态 $is_{(m,a)}$ 出发到状态 $fs_{(m,b)}$ 路径的子序列, S 包含其中一个原语命令.
- 对 e 中每个待结的操作, 如果存在节点 e 的 (未必是立即) 子孙节点 e'和 e'', 使得 e'和 e''中这个待结操作返回不同值, 则 S 不包含 e 中这个待结的操作的任何原语命令.

第 2 条要求保证了 *possibleFEP*(T, e) 中不会存在明显不可能作为固定生效点的原语命令. 在后续的研究中, 我们可以添加更多基于特定规约和库的特性所开发的对固定生效点的过滤方案.

集合 *possibleFEPForChild*(T, e, S) 中的每个元素是一个函数 f, 这个函数为 e 的每个直接子节点 e'分配一个 *possibleFEP*(T, e') 中的生效点方案 $f(e')$, 且这个生效点方案需和 S(节点 e 的生效点方案) 一致, 即要求 $S \subseteq f(e')$.

这里要求待验证的前缀闭合的有穷执行集合 E 来自一个无障碍 (obstruction-free)[15]的并发对象. 我们称一个无穷执行 e 满足 *sched*(e), 如果对于 e 上存在进程 i 有着待结操作, 则至少其中一个这样的进程被调用无穷多次. 我们称一个无穷执行 e 满足 *iso*(e), 如果在 e 上从某个动作开始, 所有动作都属于同一个进程. 我们称一个无穷执行 e 满足 *prog-s*(e), 如果对 e 上的每一个调用动作, 在其后面都有一个返回动作 (未必匹配). 我们称一条无穷执行 e 满足无障碍性质, 如果 e 满足 $sched(e) \wedge iso(e) \Rightarrow prog-s(e)$. 我们称一个并发对象 O 在 n 进程下是无障碍的, 如果 $[\![O,n]\!]$ 中的每一条无穷执行都是无障碍的. 由于 O 是无障碍的, 当执行中某个调用动作是待结的时, 可以一直执行此过程直到其返回. 因此不妨假定在树 T 中的每个叶子节点上, 每个被调用的方法都已经返回. 当处理树中某个节点并遇到一个待结操作时, 如果这个操作的生效点已经给出, 我们可以直接从该节点在树中的子节点处获得此待结操作的返回值.

过程 *checkFEP*(E, e, *Spec*, O) 的伪代码在下面给出, 它用来检测给定固定生效点的选取方案 S 在从 e 开始的子树上是否成立.

算法 3. *checkFEP*.

1. **method** *checkFEP*(E, e, *Spec*, S).
输入: 有穷执行集合 E, 有穷执行 e, 顺序规约 *Spec*, 固定生效点选取方案 S;
输出: 固定生效点的选取方案 S 在从 e 开始的子树上是否成立.
2. let T be the tree generated from execution set E;
3. let $\alpha_1 \cdot \ldots \cdot \alpha_k$ be the projection of e into S. To deal with pending operation, we can obtain their return value from its descendent node of the tree;
4. let $m_i(a_i) \Rightarrow b_i$ be the operation of the method name, argument and return value of α_i ; //从 e 和 S 构造操作序列
5. **if** $(m_1(a_1) \Rightarrow b_1)\ldots(m_k(a_k) \Rightarrow b_k) \notin Spec$, **then** return false
6. **for each** $f \in possibleFEPForChild(T,e,S)$ //检测是否存在可行的固定生效点方案使得每一个从 e 开始的执行都是强可线性化的
7. *tmp* := true;
8. **for each** children e' of e
9. **if** !*checkFEP*(E, e', *Spec*, f)

```
10.      then tmp := false;
11.          break;
12.      fi
13.   if tmp = true then return true
14. return false;
```

基于 *checkFEP* 构造方法 *hasFEP*, 其伪代码如下. *hasFEP*(*E*, *Spec*) 用来检测前缀闭合的有穷执行集合 *E* 是否有着固定生效点. 这里 S_0 是一个空集合.

算法 4. *hasFEP*.

1. **method** *hasFEP*(*E*, *Spec*).
输入: 有穷执行集合 *E* 和顺序规约 *Spec*;
输出: 是否存在固定生效点方案使得 *E* 中执行强可线性化到 *Spec*.
2. **return** $checkFEP(E,\varepsilon,Spec,S_0)$;

算法的正确性来自定理 1. 如果数据域和内存大小都是有穷的, 且每个进程只能执行限界数目的调用和返回, 则可以得到前缀闭合的有穷执行集合, 进而该算法成为一个验证并发对象是否由于固定生效点而满足强可线性化的模型检测方法.

6 对违背强可线性化的构造性证明方法及应用

关于并发对象违反强可线性化的证明, 之前的方法是证明在随机环境中并发对象的实现和并发对象的规约的行为具有不同的概率分布. 这是一种间接的证明方式. 本节总结了 Hwang 等人[9]证明并发对象不满足强可线性化的方法, 提出了一种构造性证明方法: 直接利用并发对象的顺序规约, 例如队列的先入先出性质, 构造违反强可线性化性质的执行集合, 从而证明该对象违反强可线性化性质. 之后, 本文应用这个方法证明 3 个并发对象违背强可线性化, 以说明该方法的有效性.

6.1 对违背强可线性化的构造性证明方法

本文提出一种名为二分叉区分性执行的方法来证明一个并发对象不满足强可线性化. 我们称一对执行 (*e*, *e*′) 是顺序规约 *Spec* 的二分叉区分性执行, 如果 $e=e_1\cdot e_2$, $e'=e_1\cdot e_3$, 且不存在将执行映射到 *history*(*Spec*) 的函数 *f*, 使得 $f(e)=f(e_1)\cdot s_1$, $f(e')=f(e_1)\cdot s_2$ 且 $f(e_1)\cdot s_1$ 和 $f(e_1)\cdot s_2$ 都属于 *history*(*Spec*). “二分叉区分性执行”名字中的“二分叉”来自 $e_1\cdot e_2$ 和 $e_1\cdot e_3$ 可以视为相同前缀 e_1 的两个分叉.

下面定理说明, 二分叉区分性执行蕴含了对强可线性化的违背.

定理 4. 如果并发对象 *O* 有着顺序规约 *Spec* 的二分叉区分性执行, 则 *O* 不满足强可线性化.

证明: 反证, 假定 *O* 满足强可线性化. 令 *f* 为 *O* 的强可线性化函数. 令 $e=e_1\cdot e_2$ 和 $e'=e_1\cdot e_3$ 是 *O* 的执行, 且是顺序规约 *Spec* 的二分叉区分性执行.

由于 *f* 是前缀保持的, 显然存在序列 s_1 和 s_2, 使得 $f(e)=f(e_1)\cdot s_1$ 且 $f(e')=f(e_1)\cdot s_2$. 根据强可线性化函数的定义可知, $f(e_1)\cdot s_1\in history(Spec)$ 且 $f(e_1)\cdot s_2\in history(Spec)$. 矛盾. 证毕.

6.2 Herlihy&Wing 队列

我们使用 Herlihy&Wing[1]队列作为例子, 说明如何构造二分叉区分性执行证明其不满足强可线性化. Herlihy&Wing 队列的代码如下所示. 共享变量包括数组 *item*, 队列的头尾指针 *head* 和 *tail*. Herlihy&Wing 队列包含两个方法 *enq* 和 *deq*, 分别执行入队和出队.

enq 方法首先在 *item* 中定位一个下标 *i*, 然后将待入队的值写入 *item*[*i*]. *deq* 方法包括一个循环, 在每轮循环中, 先读取当前 *tail* 值, 然后从 *item*[0] 到 *item*[*tail*] 依次尝试取出元素, 如果成功则返回, 如果都失败则开始下一个

循环. 这里 *tail.getIandIncreasement*() 指令原子地读取 *tail* 的当前值并将 *tail* 的值加 1, 然后返回读到的 *tail* 的值. *item*[*i*].*getAndSet*(null) 原子地将 *item*[*i*] 的值和 null 进行交换, 即原子地读取 *item*[*i*] 的值, 然后将 *item*[*i*] 的值设置为 null 并返回之前读到的值.

伪代码 4. Herlihy&Wing 队列.

```
1. method enq(x).
输入: 待入队元素;
输出: 无.
2. i := tail.getAndIncrement(); //确定插入队列中的下标
3. item[i] := x; //插入元素
4. return;

5. method deq().
输入: 无;
输出: 出队元素. 会一直尝试直到成功.
6. while true do
7.    range := tail;
8.    for i ∈ [0,range] i ∈ (0,range) do //在每一轮循环中从前向后尝试出队元素
9.        value := item[i].getAndSet(null);
10.       if value ≠ null then
11.           return value;
12.       fi
13.   end for
14. end while
```

6.3 Herlihy&Wing 队列的实现违反强可线性化性质证明

Herlihy&Wing 队列是一个经典的并发对象, 得到了广泛研究. Herlihy&Wing 队列满足可线性化性质, 具体证明见文献 [1]. 文献 [3] 以间接的方式证明了 Herlihy&Wing 队列不满足强可线性化. 在本节通过构造 Herlihy&Wing 队列的二分叉区分性执行的方式证明其不满足强可线性化. 下面给出具体证明.

定理 5. Herlihy&Wing 队列的实现违反强可线性化性质.

证明: 我们构造二分叉区分性执行 $e_1 \cdot e_2$ 和 $e_1 \cdot e_3$ 如下. 操作 op_1 执行 *enq*(1), 操作 op_2 执行 *enq*(2), 操作 op_3 执行 *deq*().

执行 e_1 先后执行如下动作:

- 进程 1 调用 op_1 并执行完 *i* := *tail.getAndIncrement*() (第 2 行).
- 进程 2 调用 op_2 并执行完毕.

执行 $e_1 \cdot e_2$ 先后执行如下动作:

- 执行 e_1.
- 进程 1 执行 op_1 直到执行完 *item*[*i*] := *x* (第 3 行).
- 进程 2 调用 op_3 并执行完毕, 此时返回 1.
- op_1 返回.

执行 $e_1 \cdot e_3$ 先后执行如下动作:

- 执行 e_1.

● 进程 2 调用 op_3 并执行完毕, 此时返回 2.

● 进程 1 执行 op_1 直到返回.

下面证明 $e_1 \cdot e_2$ 和 $e_1 \cdot e_3$ 是队列规约的二分叉区分性执行. 使用反证法, 假定存在将执行映射到 *history*(*Queue*) 的函数 f, 使得 $f(e_1 \cdot e_2) = f(e_1) \cdot s_1$, $f(e_1 \cdot e_3) = f(e_1) \cdot s_2$ 且 $f(e_1) \cdot s_1$ 和 $f(e_1) \cdot s_2$ 都属于 *history*(*Queue*). 这里 *Queue* 是队列的顺序规约, 每个 *Queue* 中的序列是一个 LTS $(Q, \Sigma, \rightarrow, q_0)$ 的迹, 每个 Q 中的状态是某个有穷字母表上的有穷串, 初始状态 $q_0 = \varepsilon$, 迁移关系包含如下几种迁移: $s \xrightarrow{enq(a)} a \cdot s$、 $s \cdot a \xrightarrow{deq() \Rightarrow a} s$ 和 $\varepsilon \xrightarrow{deq() \Rightarrow \text{empty}} \varepsilon$.

因为执行 e_1 中 *enq*(2) 是完整的, 所以 $f(e_1)$ 必须包含 *enq*(2). 因此, $f(e_1)$ 的可能取值是: *inv*(*enq*, 1).*res*(*enq*). *inv*(*enq*, 2).*res*(*enq*), *inv*(*enq*, 2).*res*(*enq*) 和 *inv*(*enq*, 2).*res*(*enq*).*inv*(*enq*, 1).*res*(*enq*). 为了便于书写, 我们将 f 的值域写成顺序历史对应的操作的序列.

● 如果 $f(e_1) = enq(1) \cdot enq(2)$, 则 $f(e_1 \cdot e_3) = enq(1) \cdot enq(2) \cdot (deq \Rightarrow 2)$ 且 $f(e_1 \cdot e_3)$ 不符合 *Queue*.

● 如果 $f(e_1) = enq(2)$ 或 $f(e_1) = enq(2) \cdot enq(1)$, 则 $f(e_1 \cdot e_2)$ 中 *deq*() 返回 1, 但由于 2 先入队, 所以不符合 *Qeueue*. 矛盾. 因此 $e_1 \cdot e_2$ 和 $e_1 \cdot e_3$ 是队列规约的二分叉区分性执行. 证毕.

采用二分叉区分性执行方法，在本文的电子附录 D 和附录 E 中 (详见 https://github.com/achaocmt/JOS2023/blob/main/JournalOfSoftware-material-submit.doc), 我们分别证明了 Vidyasankar[7]给出的一种单读单写寄存器的实现和文献 [8] 中的并发快照对象不满足强可线性化性质.

6.4 讨 论

二分叉区分性执行往往构造一个不完整的共同前缀, 共同前缀中至少包含一个完整的操作以及一些待结的操作. 通过这些操作的不同排列到达不同的抽象状态. 完整的操作保证对这个前缀任何可能的线性化序列都需要包含特定操作. 一些待结的操作的原子性生效时刻是不固定的, 并且依赖于后续执行. 这一点对应了文献 [4] 中提到的未来决定的可线性化点现象. 这种现象指一个原语命令是否是可线性化点将依赖于未来发生的事件的结果. 例如, *enq* 方法的 *i* := *tail.getAndIncrement*() 语句就是未来决定的, 在执行 $e_1 \cdot e_2$ 中其是 *enq* 方法的可线性化点, 而在执行 $e_1 \cdot e_3$ 中其并不是 *enq* 方法的可线性化点. 一个自然的猜想是, 如果并发对象 O 中某个方法有着未来决定的可线性化点现象, 那么 O 是违反强可线性化的. 在后续的工作中, 我们将尝试证明此猜想.

7 结 论

强可线性化是含有随机语句的并发对象的一种正确性标准, 目前尚缺乏关于强可线性化的验证方面的研究. 本文从验证算法和证否方法两个方面研究了强可线性化的验证问题. 本文证明了强可线性化等价于固定生效点和单纯帮助的并集, 并叙述了固定生效点、单纯帮助和固定可线性化点之间的关系. 本文给出了基于固定可线性化点和基于固定生效点的强可线性化的验证和检测算法. 本文提出了二分叉区分性执行这种证明并发对象违背强可线性化的证明方法, 并通过 3 个并发对象实例证明了该方法的有效性. 本文的工作为后续强可线性化验证工具的开发提供了理论和算法基础.

帮助机制[24]是并发对象设计中的一种实现无等待 (wait-free) 活性性质的方法, 被若干并发对象所采用, 例如文献 [25] 中的通用构造. 探索单纯帮助和帮助机制的关系, 以及开发基于单纯帮助的强可线性化验证方法都是有意义的未来研究方向. 另一个有探索价值的问题是, 二分叉区分性执行这种证明方法针对违背强可线性化的验证而言是否完备, 即是否违背强可线性化的并发对象一定会展示二分叉区分性执行. 我们在后续工作中将对这个问题开展研究.

References:

[1] Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems, 1990, 12(3): 463–492. [doi: 10.1145/78969.78972]

[2] Filipović I, O'Hearn PW, Rinetzky N, Yang H. Abstraction for concurrent objects. Theoretical Computer Science, 2010, 411(51–52): 4379–4398. [doi: 10.1016/j.tcs.2010.09.021]

[3] Golab W, Higham L, Woelfel P. Linearizable implementations do not suffice for randomized distributed computation. In: Proc. of the 43rd Annual ACM Symp. on Theory of Computing. San Jose: ACM, 2011. 373–382. [doi: 10.1145/1993636.1993687]

[4] Liang HJ, Feng XY. Modular verification of linearizability with non-fixed linearization points. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Seattle: ACM, 2013. 459–470. [doi: 10.1145/2491956.2462189]

[5] Bouajjani A, Emmi M, Enea C, Mutluergil SO. Proving linearizability using forward simulations. In: Proc. of the 29th Int'l Conf. on Computer Aided Verification. Heidelberg: Springer, 2017. 542–563. [doi: 10.1007/978-3-319-63390-9_28]

[6] Liang HJ, Feng XY, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Philadelphia: ACM, 2012. 455–468. [doi: 10.1145/2103656.2103711]

[7] Vidyasankar K. Converting Lamport's regular register to atomic register. Information Processing Letters, 1988, 28(6): 287–290. [doi: 10.1016/0020-0190(88)90175-5]

[8] Afek Y, Attiya H, Dolev D, Gafni E, Merritt M, Shavit N. Atomic snapshots of shared memory. Journal of the ACM, 1993, 40(4): 873–890. [doi: 10.1145/153724.153741]

[9] Hwang SM, Woelfel P. Strongly linearizable linked list and queue. In: Proc. of the 25th Int'l Conf. on Principles of Distributed Systems. Strasbourg: OPODIS, 2022. 28:1–28:20. [doi: 10.4230/LIPIcs.OPODIS.2021.28]

[10] Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing. Philadelphia: ACM, 1996. 267–275. [doi: 10.1145/248052.248106]

[11] Harris TL. A pragmatic implementation of non-blocking linked-lists. In: Proc. of the 15th Int'l Conf. on Distributed Computing. Lisbon: Springer, 2001. 300–314. [doi: 10.1007/3-540-45414-4_21]

[12] Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm. In: Proc. of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures. Barcelona: ACM, 2004. 206–215. [doi: 10.1145/1007912.1007944]

[13] Helmi M, Higham L, Woelfel P. Strongly linearizable implementations: Possibilities and impossibilities. In: Proc. of the 2012 ACM Symp. on Principles of Distributed Computing. Madeira: ACM, 2012. 385–394. [doi: 10.1145/2332432.2332508]

[14] Denysyuk O, Woelfel P. Wait-freedom is harder than lock-freedom under strong linearizability. In: Proc. of the 29th Int'l Symp. on Distributed Computing. Tokyo: Springer, 2015. 60–74. [doi: 10.1007/978-3-662-48653-5_5]

[15] Liang HJ, Hoffmann J, Feng XY, Shao Z. Characterizing progress properties of concurrent objects via contextual refinements. In: Proc. of the 24th Int'l Conf. on Concurrency Theory. Buenos Aires: Springer, 2013. 227–241. [doi: 10.1007/978-3-642-40184-8_17]

[16] Attiya H, Enea C, Welch JL. Impossibility of strongly-linearizable message-passing objects via simulation by single-writer registers. In: Proc. of the 35th Int'l Symp. on Distributed Computing. Freiburg, 2021. 7:1–7:18.

[17] Rady AS. Characterizing implementations that preserve properties of concurrent randomized algorithms [MS. Thesis]. Toronto: York University, 2017.

[18] Attiya H, Enea C. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In: Proc. of the 33rd Int'l Symp. on Distributed Computing. Budapest, 2019. 2:1–2:17.

[19] Clarkson MR, Schneider FB. Hyperproperties. Journal of Computer Security, 2010, 18(6): 1157–1210. [doi: 10.3233/JCS-2009-0393]

[20] Vale AO, Shao Z, Chen YX. A compositional theory of linearizability. Proc. of the ACM on Programming Languages, 2023, 7: 38. [doi: 10.1145/3571231]

[21] Emmi M, Enea C. Violat: Generating tests of observational refinement for concurrent objects. In: Proc. of the 31st Int'l Conf. on Computer Aided Verification. New York: Springer, 2019. 534–546. [doi: 10.1007/978-3-030-25543-5_30]

[22] Koval N, Fedorov A, Sokolova M, Tsitelov D, Alistarh D. Lincheck: A practical framework for testing concurrent data structures on JVM. In: Proc. of the 35th Int'l Conf. on Computer Aided Verification. Paris: Springer, 2023. 156–169. [doi: 10.1007/978-3-031-37706-8_8]

[23] Jia QW, Lv Y, Wu P, Zhan BH, Hao JF, Ye H, Wang C. VeriLin: A Linearizability checker for large-scale concurrent objects. In: Proc. of the 17th Int'l Symp. on Theoretical Aspects of Software Engineering. Bristol: Springer, 2023. 202–220. [doi: 10.1007/978-3-031-35257-7_12]

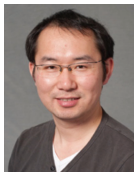
[24] Censor-Hillel K, Petrank E, Timnat S. Help! In: Proc. of the 2015 ACM Symp. on Principles of Distributed Computing. Donostia-San: ACM, 2015. 241–250. [doi: 10.1145/2767386.2767415]

[25] Herlihy M, Shavit N, Luchangco V, Spear M. The Art of Multiprocessor Programming. 2nd ed., San Francisco: Morgan Kaufmann, 2020.

王超(1985－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化方法, 并发数据结构, 分布式数据类型.

吕毅(1972－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为并发理论, 形式化方法.

贾巧雯(1992－), 女, 博士, 主要研究领域为并发系统, 可线性化验证.



吴鹏(1977－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为形式化方法, 并发测试, 机器学习.