z 软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007138]

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

基于交互式定理证明的并发程序验证工作综述 *

王中烨[1], 吴姝姝[1], 曹钦翔[1]

[1](上海交通大学，电子信息与电气工程学院，上海 200240)

通讯作者: 王中烨, E-mail: wangzhongye1110@sjtu.edu.cn

摘 要: 并发程序与并发系统可以拥有非常高的执行效率和相对串行系统较快的响应速度，在现实中有着非常广泛的应用。但是并发程序与并发系统往往难以保证其实现的正确性，实际应用程序运行中的错误会带来严重的后果。同时，并发程序执行时的不确定性会给其正确性验证带来巨大的困难。在形式化验证方法中，人们可以通过交互式定理证明器严格地对并发程序进行验证。本文对在交互式定理证明中可用于描述并发程序正确性的验证目标进行总结，它们包括霍尔三元组、可线性化、上下文精化和逻辑原子性。交互式定理证明方法中常用程序逻辑对程序进行验证，本文分析了基于并发分离逻辑、依赖保证逻辑、关系霍尔逻辑等理论研究的系列成果与相应形式化方案，并对使用了这些方法的程序验证工具和程序验证成果进行了总结。

关键词: 并发程序验证;可线性化;上下文精化;程序逻辑;关系霍尔逻辑

中图法分类号: TP311

中文引用格式: 王中烨,吴姝姝,曹钦翔. 基于交互式定理证明的并发程序验证工作综述.软件学报. http://www.jos.org.cn/1000-9825/7138.htm

英文引用格式: Wang ZY, Wu SS, Cao QX. A Survey of Interactive Theorem Proving Based Concurrent Program Verifications. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7138.htm

A Survey of Interactive Theorem Proving Based Concurrent Program Verifications

WANG Zhong-Ye[1], WU Shu-Shu[1], CAO Qin-Xiang[1]

[1](School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: Concurrent programs and concurrent systems are usually highly efficient and are widely used in practice. However, concurrent programs and systems are often prone to error, which could bring fatal consequences in real world applications. Moreover, the non-determinism brought by concurrency is a major difficulty in the verification of concurrent programs. But with formal verification, people could use interactive theorem provers to rigorously prove the correctness of a concurrent program. We present several correctness criteria for concurrent programs, which can be verified using interactive theorem proof techniques. They include Hoare triple, linearizability, contextual refinement, and logical atomicity. Researchers usually use program logics to verify programs in an interactive theorem prover. We summarize the usage of concurrent separation logic, rely-guarantee based logic, and relational Hoare logic in concurrent program verifications. We also surveyed existing foundational verification tools and verification results using these techniques.

Key words: concurrency verification; linearizability; contextual refinement; program logic; relational Hoare logic

1 引言

伴随着计算机技术的不断进步，人们对于高效且安全的程序与系统的需求也在不断增长。并发程序与并发系统通过同时交互执行多个程序与组件，充分利用各个程序执行的等待时间和发挥多核硬件系统的计算能力，可以显著提升程序与系统的运行速度和多任务情况下对用户输入的响应速度。因此，并发技术在航天航

* 收稿时间: 2023-09-11; 修改时间: 2023-10-30, 2023-12-13; 采用时间: 2023-12-20; jos 在线出版时间: 2024-01-05

空、交通、医疗和金融等多个领域都有着广泛的应用。但是与其高效率相对的，并发程序非常容易发出错误，且其安全性与正确性的验证是非常困难的，系统中一个微小的错误都可能带来不可估量的后果。

并发技术的特性决定了其必然出现潜在的漏洞导致的错误。一个并发程序或并发系统中，通常会有多个组件同时交互运行，不同组件都可以对该系统的状态产生影响。不同组件也会使用共享的内存空间进行通信和合作，如果不同组件间没法对共享的状态达成共识从而有序地访问，会导致某些组件无法正确完成其功能。比如当两个程序并发地访问同一块内存，且未通过临界区等手段保证访问的互斥，就会产生数据竞争（Data Race）。在大多数体系结构中，这类数据竞争会导致两个程序的执行结果是未定义的，进而导致程序的错误。

为了描述什么样的程序是安全的或正确的，进而在现实中避免使用错误的程序，人们会使用多种正确性指标来刻画并发程序的正确性，如功能正确性（Functional Correctness）、信息流安全性（Information Security）、进展性质（Progress）和终止性（Termination）等。功能正确性要求程序能产生正确的结果，正确地实现某种功能。本文关注的程序功能正确性特指该程序执行前后的程序状态变化需要满足的性质，如使用霍尔三元组的前后条件来约束前后程序状态。信息流安全性一般指程序遭到攻击时能保证信息与数据不会被非法地修改、泄露、使用。进展性质和终止性要求程序不会在无期限地延迟其执行，并最终会被执行且终止。

在并发环境中，串行环境下的正确性指标不足以充分刻画一个程序或系统的正确性。即使一个程序在顺序执行的情况下满足这些正确性要求，在有其他并发线程影响其执行的时候，其表现可能不再与开发者所预期的相一致。为了更好地描述并发程序的正确性，研究者们提出使用顺序一致性[1]（Sequential Consistency）、可线性化[35]（Linearizability）、可序列化[2]（Serializability）等作为其正确性指标。Liang 等人[3]对由 Herlihy 和 Shavit[4]提出的并发程序进展性质进行总结，形式化地定义了无等待（Wait-Free）、无锁（Lock-Free）、无死锁（Deadlock-Freedom）、无饥饿（Starvation-Freedom）、部分无死锁（Partial-Deadlock-Freedom）、部分无饥饿（Partial-Starvation-Freedom）等进展性质。对于一个并发系统，一般还可以要求其是故障安全的（Crash-Safe），即当系统在某次运行中因故障而停机了，可以通过恢复程序将系统恢复到故障前的状态继续正常运行。

但是并发技术的特性也决定了其验证是相对困难的。由于并发程序与系统中各个组件交互执行的顺序是不确定的，人们无法直观地判断实际的执行情况。往往并发程序在某次执行中出现的错误在另一次执行中难以被复现，因为两次执行中不同组件的交互顺序是不同的。这使得基于测试的方法难以保证可靠性，容易遗漏可能出错的情况。相比而言，形式化的验证方法使用数学对程序和系统进行精确的建模和严格的验证，可以保证经验证程序的正确性。同时，由于组件间的相互影响，一般难以对单独一个组件的正确性进行验证，这也导致了在规模较大的系统中，无法使用简单的验证方法进行模块化的验证。如何使验证方法支持模块化验证，具有较好的可规模化能力，也是形式化方法研究需要考虑的问题。

1.1 常用的形式化验证方法

（1） 模型检测（Model Checking）

模型检测通过将验证对象抽象成一个形式化的模型从而判断其是否满足规约。在并发程序验证中，其模型就是该程序所有可能的执行，验证过程就是判断每一种执行是否都满足其正确性要求，该验证过程一般会交由 SMT 求解器这类工具进行可满足性判定。但是并发程序与顺序执行程序最大的区别在于其不确定性。两个并发程序交替执行时，每一步都可以由两个程序中的任意一个进行，从而使得其可能的执行顺序远不止一种。这会导致在遍历并发程序所有可能的执行时搜索空间爆炸，使得传统的模型检测在验证并发程序时效率低下，需要使用改进的模型检测方法。

限界模型检测[5]（Bounded Model Checking）通过限制程序执行情况的搜索空间，如限制循环或递归的深度，在牺牲一定准确性的情况下提升模型检测的效率。Inverso 等人[6]借助这种思路，通过限制轮询调度的轮数来减少并发执行可能产生的执行情况，从而在将多线程程序转换为非确定性的顺序执行程序的同时控制两者的相似程度。其使用的高性能 BMC 工具作为顺序验证问题的后端。

动态偏序规约[7]（Dynamic Partial-Order Reduction）是一种并发程序模型检测的搜索剪枝策略。其使用程

序执行时指令间的依赖关系作为偏序来对所有的执行情况进行等价类划分，对同一类执行仅进行一次模型检测，从而在保证正确性的同时大幅提升检测效率。Alglave 等人[8]结合限界模型检测和动态偏序规约构建了一个并发程序的模型检测工具，并在多种内存模型下对 Linux 内核、Apache HTTP 服务器等的代码进行了验证。

无状态模型检测[9]（Stateless Model Checking）通过特殊的调度程序遍历所有并发执行调度情况，从而对并发程序所有可能的执行进行搜索。许多工作[10][11][12]使用无状态模型检测在不同内存模型下对并发程序进行验证。如 Abdulla 等人[10]使用时序痕迹（Chronological Trace）描述并发程序的执行。这种方式适用于 TSO 和 PSO 两种内存模型，且其本身就保证了最优的动态偏序规约剪枝，使得无状态模型检测有较高的效率。

（2） 交互式定理证明（Interactive Theorem Proving）

另一种形式化的验证方法是交互式定理证明，即在 Coq[13]、Isabelle/HOL[14]等交互式定理证明器中证明程序正确性。证明者一般使用程序逻辑，如霍尔逻辑[15]，以及一定的证明自动化来进行证明。为了支持并发程序验证，研究者们先后在霍尔逻辑的基础上设计了基于依赖保证的逻辑[16][17]和并发分离逻辑[18]。通过将程序逻辑与逻辑关系[19]（Logical Relation）结合得到的关系霍尔逻辑（Relational Hoare Logic）使得证明者可以对更多的并发程序正确性条件进行验证。本文对有关理论和工作在后续章节详细介绍。

（3） 自动化定理证明（Automated Theorem Proving）

在交互式定理证明基于程序逻辑的验证方法的基础上，使用 SMT 求解器等方式实现自动化的定理证明。一般自动化定理证明器被实现为证明框架检查器（Proof Outline Checker）或者附带证明的程序的验证器（Proof-Carrying-Code/Annotation Verifier）。用户可以提供证明中的一些关键节点需要满足的结论和证明的提示，或者以附带证明的程序给出关键程序语句需要满足的前后条件标记（Annotation），自动化定理证明器会验证能否通过用户提供的思路证明得到最终结论并自动搜索使该证明框架成立的证明。自动化定理证明器一般会在给定的程序逻辑提供的证明规则中搜索和构造证明。

Starling[20]、Caper[21]是基于 Z3 SMT 求解器[22]的证明框架检查器。Starling 使用 Views 并发程序验证逻辑框架[23]作为其证明使用的逻辑，可以证明简单的并发程序正确性，但由于其程序逻辑表达力的匮乏，无法证明更细粒度的并发程序。Caper 使用 CAP 程序逻辑[24]实现并发程序的证明，其通过回溯（Backtracking）、诱因推导（Abduction）等方式来优化证明的搜索。Diaframe 框架[25]是基于 Iris[26][27]程序逻辑的证明框架检查器，其在 Iris 原有的自动化证明基础上加入了基于证明目标指引的证明搜索算法，可以更高效的利用用户的证明提示和实现并发程序的自动化证明。Voila[28]是使用 Viper 验证器[29]的基于 TADA[30]程序逻辑的证明框架检查器，其对逻辑原子性的证明有着较好的支持。

1.2 论文结构

本文在第 2 节介绍通过交互式定理证明验证程序正确性时的验证对象设定，即需要对一个待验证程序的哪些方面进行形式化。第 3 节介绍交互式定理证明中并发程序的正确性验证目标。第 4 节介绍多种用以验证并发程序的正确性目标的程序逻辑。本文在第 5 节对基于这些程序逻辑的实际验证工具进行总结，并在第 6 节对通过交互式定理证明得到验证的并发系统和并发算法进行总结。第 7 节对本文以及仍待交互式定理证明方法解决的并发程序验证问题进行总结。

2 并发程序验证研究的分类

本节明确几个并发程序验证中需要考虑的基本问题。这些问题界定了不同验证工作间的分类。

程序语言：首先，每一段程序都是在一种选定的程序语言中实现的。为了验证一段程序，需要首先明确对何种程序语言进行形式化。一般来说，可以直接对实际使用的高级程序语言进行形式化，从而直接对实际的程序进行验证，如 VST[31]是对 C 语言程序的验证工具、RustBelt 是对 Rust 语言程序的验证工作；也可以对样例语言（Toy Language）进行形式化，从而在理论层面对一些算法进行实现和验证，为将来对实际程序的验证打下理论基础，如 Iris[27]中对 Iris-Lambda 语言建立验证框架并推广至现实中的高级程序语言。另一种方式是直接对汇编代码进行形式化，对汇编程序进行验证。但是汇编语言的易读性通常不如高级程序语言，

不利于使用交互式定理证明器进行证明。如果需要保证程序在汇编代码层面的正确性，可以通过分别证明高级程序语言下的程序正确性和编译器的正确性，结合两者得到汇编代码的正确性。

并发指令：在并发程序验证工作中对程序语言进行形式化时，除了需要对顺序执行程序中常见的指令和程序的组合语句形式化之外，还需要明确对哪些并发情况下特有的语句进行形式化。主要包括线程的创建,可以分为固定的并发组合（Parallel Composition）和动态的线程创建（Fork-Join）；同步语句，如临界区指令、锁的有关操作、条件变量、信号量等；一般的原子指令，如 CompareAndSwap（CAS）、FetchAndInc（FAI）等。这些指令有些由硬件直接支持，也有通过软件专门实现的并发库。

程序语义与内存模型：其次，在对程序语言进行形式化的时候，不但需要形式化该语言的语法，还需要形式化其语义。而为了对语义形式化定义，需要明确验证对象所处的体系结构是怎样的，因为这会影响程序状态（也就是内存）的定义以及程序和内存交互的方式。例如在满足顺序一致性的内存模型中和在弱内存模型中，一段程序的行为可以是截然不同的。一般会考虑内存模型为顺序一致性（Sequential Consistency）、TSO（Total Store Order）、C11 弱内存模型等。弱内存模型上的程序验证有着较大的难度，本文考虑的验证方式是较成熟的，多为在满足顺序一致性的内存模型中的验证，但也存在一些工作对弱内存模型进行形式化验证[32]。

验证目标：最后，需要明确对何种验证目标进行形式化和验证。一种验证目标规定了程序和系统的正确性定义。不同的验证目标会使用不同的规约（Specification）描述验证对象的正确性，并需要不同的验证方法进行论证。对于不同的验证对象，选用一个合适的验证目标不仅可以清晰地表达用户对于验证对象的要求，而且有助于选用合适的验证方法与验证工具进行高效的验证。

总而言之，在验证一个并发程序或者并发系统前，需要明确实现它的程序语言是什么，需要支持的并发操作有哪些，其基于的体系结构是什么，以及需要验证的目标是什么。只有在明确了这些设定之后，才能在交互式定理证明器中对并发验证理论进行形式化。本文讨论的验证目标与验证方式可适用于绝大部分程序语言，讨论多种不同的并发操作的验证方法，且主要考虑在顺序一致性内存模型下的验证。

3 并发程序的功能正确性验证目标

本文中的功能正确性特指一段程序执行前后程序状态变化需要满足的性质。在大多数情况下，可以使用霍尔三元组的前后条件来描述这种性质。在并发环境下，功能正确性不能仅仅考虑本线程对程序状态产生的修改，还需要考虑到其他线程对程序状态产生的影响以及进一步对本线程程序执行产生的影响。换言之，一个合理的对于并发程序而言的功能正确性需要明确该程序所处的并发上下文。并发上下文是指并发的环境线程以及对它们行为的约束，如可以直接约束不存在环境线程，并发仅在主程序内部通过创建线程产生；可以

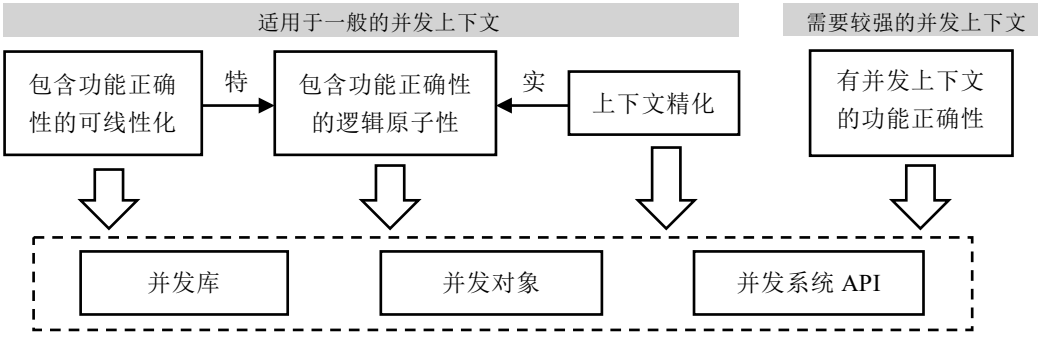

图 1：并发程序的功能正确性验证目标

约束程序执行所依赖的内存不会被其他线程使用；可以约束环境线程的行为和本线程的行为在一定范围内，且两种行为互相没有冲突等。本文称这种性质为有并发上下文的功能正确性。该类功能正确性通常会要求一个明确的且性质较强的并发上下文，否则，由于缺乏对并发上下文的约束，环境线程可以对本线程运行进行过度的干扰，进而使霍尔三元组无法精确的描述一个程序的功能。为了能在较为一般的并发上下文中描述一

个程序的正确性，学者还提出了以下三种针对并发程序的验证目标：包含功能正确性的可线性化（Linearizability），上下文精化（Contextual Refinement），和包含功能正确性的逻辑原子性（Logical Atomicity）。这四种功能正确性目标的关系如图 1 所示。本节对这些验证目标的定义进行详细阐述。

3.1 有并发上下文的功能正确性

有并发上下文的功能正确性指在对环境线程的行为进行特定约束的情况下，验证目标程序的执行前后程序状态的变化需要满足的性质。一般来说，研究者会使用霍尔三元组[15]（Hoare Triple）来描述这种性质。本节首先从并发上下文的一种特殊情况出发，即不存在与目标程序并发执行的线程（单线程串行执行），来介绍基于霍尔三元组定义的功能正确性。

一个霍尔三元组$\{P\}\,c\,\{Q\}$包括：描述程序运行前程序状态所满足的性质的前条件P，该三元组验证的程序c，以及描述程序运行结束后程序的返回值和此时程序状态满足的性质的后条件Q。其含义是在一个线程中，如果程序运行前，其程序状态满足前条件P，那么程序c的运行将不会出错，并且其运行终止时的程序状态将会满足后条件Q。这样的性质一般称为一段程序的部分正确性（Partial Correctness），相应的，完全正确性（Total Correctness）会在部分正确性的基础上要求程序运行终止。大多数工作只考虑部分正确性，但也有研究[30][33]将完全正确性与进展性质的研究进行结合，但是现在尚没有在定理证明器中形式化的成果。证明者一般会通过霍尔逻辑[15]证明一个霍尔三元组是成立的。一般会称证明得到的霍尔三元组为程序逻辑断言（Judgement）。

霍尔三元组中前后条件P和Q一般会在断言逻辑（Assertion Language）中形式化。不同的断言逻辑会包含不同的断言表达式从而可以描述不同的程序状态的特性，因而霍尔三元组依然是一个非常一般化的规约形式。在验证下文所涉及的大部分验证目标时，都会使用霍尔三元组进行证明并将其作为中间结果，但是它们将会加入专门的机制用来更准确地描述一个程序在并发的系统中的行为和功能。

本节以一个简单的例子来说明霍尔三元组的使用。假设有一个指针变量 p，其指向的地址上保存了一个队列的头指针，并且完整的队列可以用列表l表示。本节用断言$\mathrm{p}\mapsto l$描述该性质。同时假设存在一个顺序执行的函数 enq(x)，其接受一个函数参数 x，且 x 的类型与队列l中的元素类型相同。该函数的功能是将一个新的以 x 作为值的元素加入指针 p 所指的队列。在一个顺序执行，即没有并发的环境中执行时，可以使用霍尔三元组$\{\mathrm{p}\mapsto l\}\mathrm{enq(x)}\{\mathrm{p}\mapsto(l :: \mathrm{x})\}$直观地定义这样的行为。其前条件描述了函数执行前，指针 p 所指的队列为l，同时后条件描述了函数执行后，指针所指的队列变为了$l :: \mathrm{x}$，即队列末尾增加了值为 x 的元素。该霍尔三元组直观且正确地表达了一个队列入队函数 enq(x)的功能。

霍尔三元组同样可以用于描述完整的并发程序的正确性。比如以下霍尔三元组说明了一有并发执行的程序的正确性。该程序会通过 fork 指令创建一个新的线程 t 来执行 enq(x)[1]，同时主线程会执行 enq(y)并通过 join 指令等待子线程 t 运行完成进而实现同步。该霍尔三元组表达了该程序的功能正确性：该程序运行没有出错，且运行结束后，队列的末尾新增了 x 和 y 两个元素，但是存在两种不同的入队顺序。

$$\{\mathrm{p}\mapsto l\}\ t = \mathrm{fork(enq(x))}\ ;;\ \mathrm{enq(y)}\ ;;\ \mathrm{join}(t)\ \{\mathrm{p}\mapsto(l :: \mathrm{x} :: \mathrm{y}) \vee \mathrm{p}\mapsto(l :: \mathrm{y} :: \mathrm{x})\}$$

在该例子中，霍尔三元组$\{P\}\,c\,\{Q\}$所描述的程序c的并发上下文为空，即在它开始运行时，系统中仅存在运行程序c自身的一个线程，且运行结束时所有产生的线程均已同步，不存在仍在运行的其他线程。这是一个过于严格的并发上下文约束。一个并发系统中通常存在多个线程并发执行。当一个程序在并发系统中开始执行时，一般存在并行执行的其他线程，且其他线程可以对共享内存进行修改从而影响当前程序的执行。这种情况下，对于 enq 函数，上文使用的霍尔三元组$\{\mathrm{p}\mapsto l\}\mathrm{enq(x)}\{\mathrm{p}\mapsto(l :: \mathrm{x})\}$将无法正确表达该函数的功能。因为在一个线程运行 enq(x)时，可能存在其他线程也在运行入队或出队函数对同一个指针指向的队列操作。此时，当该线程执行 enq(x)结束，指针 p 所指的队列不一定仅有 x 被入队了，而且 x 之前的队列也不一定依然是l。

为了能在该情况下使用霍尔三元组对程序的功能正确性进行描述，需要引入对并发上下文的约束。在并

[1] 为了保证并发执行的正确性，这里的两个 enq 函数与上文顺序执行的函数 enq 有着不同的实现。这里的 enq 函数需要使用锁（Lock）实现线程间同步，或者使用类似 Herlihy-Wing 队列[35]这类通过原子指令实现无锁的线程同步。

发分离逻辑[18]中（4.2 节），可以通过约束当前程序使用的内存为该线程独占的或要求所有线程对于共享内存操作均满足一定的不变量来限制并发上下文。在依赖保证逻辑[16][17]中（4.3 节），可以通过显式地要求所有环境线程的行为均满足给定的依赖关系（Rely）来约束并发上下文。有了这种对并发上下文的限制，可以在一定程度上使原本仅支持串行程序正确性描述的霍尔三元组能对程序在并发环境下的功能正确性进行描述。对于 enq 函数，可以要求环境线程不持有访问指针 p 所指内存的权限，或通过依赖关系要求环境线程不对指针 p 所指队列进行修改，这样就可以使用霍尔三元组$\{p\mapsto l\}\mathrm{enq}(x)\{p\mapsto(l :: x)\}$来准确描述该函数的功能。

这依然是非常强的对于并发上下文的约束，它们均杜绝了其他线程并发地使用当前程序的可能。此时可以进一步弱化并发上下文的约束，使所有线程都可以访问该队列但是需要使用不变量来保护共享内存使得所有访问该队列的线程均维护该队列的合法性，或者在依赖关系中允许环境线程对该队列进行操作的同时要求本地线程的霍尔三元组前后条件关于该依赖关系是稳定的（stable）。随之而来的结果是，可以证明得到的霍尔三元组也会被弱化为三元组$\{\exists l.\, p\mapsto l\}\mathrm{enq}(x)\{\exists l.\, p\mapsto l\}$，其前后条件均只描述了“指针 p 所指的内存是一个队列”这一事实，而非当前程序实际对这个队列进行了何种操作。因为在并发分离逻辑中，所有对于共享内存（如此处的队列）的访问均要通过不变量实现，因此本地线程的三元组前后条件将不能直接持有共享内存的权限，也就无法直接描述共享内存发生了什么具体变化 [2]。而在依赖保证逻辑中，稳定性要求使得前后条件不能过强，在依赖关系足够一般化的时候，前后条件将仅能描述共享内存所需要满足的不变量。

这类更一般化的并发上下文约束是非常常见的。如在并发系统或者并发对象的验证中，相对常用的并发上下文约束是各线程能进行的函数调用是该系统和对象所提供的函数接口，大多数情况下不对各线程调用的函数范围和调用顺序进行进一步限制。为了能在这种并发上下文约束下准确地表达一个程序的功能正确性，往往会用到下文所阐述的其他正确性目标。

3.2 包含功能正确性的可线性化

可线性化[35]（Linearizability）是描述并发对象功能正确性的非常重要的准则。它的思路是直接使用所有正确的执行记录（Trace）的集合作为一个对象的线性化规约[34]（Linearization Specification）。因此它能非常准确地表达一个程序在并发环境下的正确行为，很好地解决上文霍尔三元组无法处理的情况。本节首先以上文中的队列为例介绍何为可线性化，再形式化地给出可线性化的定义和该队列在可线性化意义下的规约。

可线性化的讨论对象一般是并发对象。一个并发对象由多个线程共享其内容，其本身的数据是存在于共

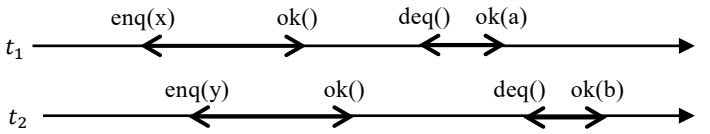

图 2：队列的一次执行的历史记录

享内存中的。一个并发对象会提供若干方法作为用户通过各个线程操作其内容的唯一手段，只要保证这些方法的实现在并发调用的情况下满足可线性化要求就能保证该并发对象的正确性。在可线性化理论中，一个并发对象的并发上下文是由任意多个线程组成的并发环境，且一般对该环境唯一的要求是仅能通过调用该并发对象的方法来对其内容进行修改。

以队列为例，其提供了两个方法：入队函数$\mathrm{enq}: \mathrm{Val} \to 1$，其包含一个类型为 Val 的参数，其返回值为空，以单元类型 **1**（Unit Type）作为其返回类型；出队函数$\mathrm{deq}: \mathrm{Val}$，其不接受任何参数，其返回值类型为 Val。这里使用 enq(x)和 deq()表示一个方法调用开始事件，使用$\mathrm{Ok}(v)$表示一个方法的返回事件，其中v为返回值。图 2 为该队列对象的一次运行的历史记录（History），即由多个事件组成的列表，其中时间轴方向从左到右。

[2] 可以通过 $t_1:\mathrm{enq}(x)\cdot t_2:\mathrm{enq}(y)\cdot t_1:\mathrm{ok}\cdot t_2:\mathrm{ok}\cdot t_1:\mathrm{deq}()\cdot t_1:\mathrm{ok}(a)\cdot t_2:\mathrm{deq}()\cdot t_2:\mathrm{ok}(b)$ 与式通常使得到的功能正确性更接近于其他三种并发程序的功能正确性目标。本文将这种方法视作证明其他三种目标的方法之一。

该历史记录由以下列表定义。

线程t_1依次调用 enq(x)和 deq()并依次返回，线程t_2依次调用 enq(y)和 deq()并依次返回。两个线程的入队函数调用在时间上有重合，是并发执行的。这里使用两端带箭头的区间表示每个调用从开始到返回之间的时间区间。该次执行中，两个线程的函数调用在时间上没有重叠，有固定的先后执行顺序。假设在执行这段程序前的队列为空，那么两次出队操作的返回值 a 和 b 必然分别是 x 和 y 中先/后入队的那个元素。这样一个性质是该队列对象的正确性的必要条件，可线性化理论可以很好的表达出这样一个约束。

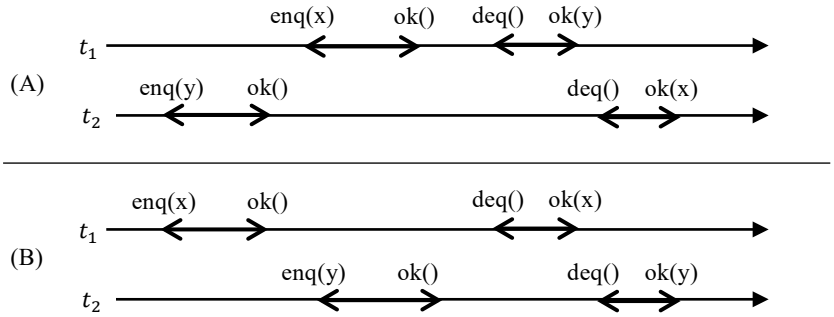

图 3：对列的一次执行的线性化结果

首先，注意到图 2 中的两个入队操作 enq(x)和 enq(y)在时间上是重叠的，如果该队列对象的实现没法正确地处理两个对共享内存中的队列数据的并发操作，那么就会发生数据竞争（Data Race），并且执行结束后队列中的数据将不再是一个合法的队列，上述性质必不能满足。因此，任何一个合理的正确性准则都应该要求程序的实现避免该情况。排除该情况之后，便只剩下先完成执行 enq(x)和先完成执行 enq(y)两种情况了。对于一个正确的队列实现，图 2 中执行应该等效于图 3 中的两种执行顺序之一，即两次出队操作的结果根据 x 和 y 的入队顺序依次返回 x 和 y。其分别对应的历史记录列表如下。

$$t_2{:}\mathrm{enq}(y)\cdot t_2{:}\mathrm{ok}\cdot t_1{:}\mathrm{enq}(x)\cdot t_1{:}\mathrm{ok}\cdot t_1{:}\mathrm{deq}()\cdot t_1{:}\mathrm{ok}(y)\cdot t_2{:}\mathrm{deq}()\cdot t_2{:}\mathrm{ok}(x)$$
$$t_1{:}\mathrm{enq}(x)\cdot t_1{:}\mathrm{ok}\cdot t_2{:}\mathrm{enq}(y)\cdot t_2{:}\mathrm{ok}\cdot t_1{:}\mathrm{deq}()\cdot t_1{:}\mathrm{ok}(x)\cdot t_2{:}\mathrm{deq}()\cdot t_2{:}\mathrm{ok}(y)$$

该队列对象的规约需要包括所有类似的入队与出队顺序一致的历史记录，同时排除有错误的（如数据竞争）的历史记录。这样一个规约便可以准确地表达一个并发对象应有的正确行为，即其功能正确性。可线性化理论对于并发对象的正确性要求便是：任何一种合法的执行顺序都可以等效为规约中的某一个历史。

3.2.1 Herlihy-Wing 可线性化

可线性化理论最初由 Herlihy 和 Wing[35]提出，这依然是现在最常用的可线性化定义。首先，定义一个事件为由线程 ID、事件名称、事件的参数组成的三元组，记作$t{:}\,e(\vec{v})$。其中，事件名称可以区分为调用事件（如 enq）和返回事件（如 Ok）。定义一个历史记录为由事件组成的列表，记作h。定义一个合法的历史记录为尊重每个线程的顺序一致性（Sequential Consistency）的历史记录，即把该列表投影到仅包含任意一个线程的事件的序列上，其中的调用事件和对应的返回事件交替出现，且第一个事件为调用事件。如果一个历史记录中，所有调用事件之后的第一个事件为其对应的返回事件，那么称该历史为顺序历史（Sequential History），这样调用后立即返回的事件被称为原子（Atomic）事件。

Herlihy 和 Wing 主要通过偏序关系来定义可线性化。对于一个历史记录h,定义一个关于其中事件的偏序关系$<_h$。规定$t_1{:}\,e_1 <_h t_2{:}\,e_2$当且仅当$e_1$在$e_2$之前发生且$e_1$为返回事件，$e_2$为调用事件。换言之，偏序关系$<_h$记录了任意两个非并发执行的事件间的顺序,因为$e_1$代表的函数调用在 e_2代表的函数调用开始前就已经结束了。历史记录h可以线性化到历史记录s，记作$h \rightsquigarrow s$，当且仅当：（1）可以通过在h结尾加上若干个返回事件，然后移除所有没有对应返回事件的调用事件，得到历史h'，（2）h'投影到任意线程上的事件序列和s投影到任意线程上的事件序列是相同的，（3）s的偏序关系包含了h的偏序关系，即$<_h \subseteq <_s$。Herlihy-Wing 可线性化的定义中还额外要求了s是顺序历史，因为这样能把一个并发的历史记录h变换到有相同效果的顺序历史记录，方

便对其进行描述与约束。对于一个并发对象的实现，可以使用由顺序历史组成的集合S作为其线性化规约，该对象是可线性化的，当且仅当任何该实现可以产生的历史记录都可以线性化到S中的某一个历史。

案例 1：以上文的队列为例，其规约S为满足以下性质的最大的集合。一个历史h满足这个规约当且仅当（1）它是一个空历史，（2）或者它是一个满足规约的历史h'在结尾加上了一个原子的 enq 事件，（3）或者它是一个满足规约的历史h'在结尾加上了一个原子的 deq 事件，并且返回值v是历史h'所代表的队列的头元素。

$$h \in S \iff \begin{pmatrix} h = \epsilon \vee (h = h' \cdot \mathrm{enq}(x) \cdot \mathrm{ok} \wedge h' \in S) \vee \\ (h = h' \cdot \mathrm{deq}() \cdot \mathrm{ok}(v) \wedge h' \in S \wedge \mathrm{queue}(h') = v :: l) \end{pmatrix}$$

可线性化一个非常重要的性质是局部性[35]（Locality）：一个完整的并发系统通常由多个并发对象组成，如果把整个系统当作一个并发对象，那么该系统是可线性化的，当且仅当其中的每个对象都是可线性化的。这个性质使得验证者可以将系统分解成多个模块，且分别验证多个模块的可线性化后可以重新组合得到整个系统的可线性化。换言之，在仅包括一个并发对象自身的事件的环境下验证其可线性化性质，就能保证其在并发系统中的正确性。这样一种将同一层级的多个对象分别验证并重新组合的横向可组合性，是可线性化相较顺序一致性[1]所具有的优势之一。

可以从两个角度来理解 Herlihy-Wing 可线性化：首先，它要求一个程序的并发执行可以等效成该程序的串行原子执行，这保证了程序在并发意义上的正确性，即并发的环境线程不会对其执行造成不良影响；同时，它要求一个程序的并发执行的可线性化结果符合给定的线性化规约，这保证了程序的功能正确性，即每个函数调用会在正确的时刻（历史记录中特定的位置）返回正确的结果（正确的返回事件参数）。在一些特定的语境下，人们用“可线性化”一词仅仅指代前一性质，而非两个性质的整体。本文用“包含功能正确性的可线性化”作为一种目标的名称，以强调本文使用可线性化性质作为功能正确性目标，并以“可线性化”作为简称。在下文介绍弱化可线性化的原子性要求得到更一般的可线性化指标时，其的功能正确性属性显得尤为重要。

3.2.2 集合可线性化与区间可线性化

Herlihy-Wing 可线性化可以处理所有规约仅包含顺序历史的并发对象。对于这类对象，有一个非常重要的概念就是可线性化点（Linearization Point）。许多可线性化的并发对象的方法会在执行某一个原子指令的时候，完成该方法需要对共享内存产生的影响。该过程在这一瞬间生效，不会被其他线程影响。这样的指令所在的位置就称为可线性化点。在方法执行过程中，不论是在可线性化点之前还是之后的执行都不会对共享内存产生实质的影响。由于可线性化点对应的是原子指令，任何两个方法的可线性化点在任何并发执行中都不会发生重叠。因此可以直接按照一次执行中的可线性化点的顺序对该次执行进行线性化，从而证明这种具有可线性化点的对象是可线性化的。

但是，并不是所有 Herlihy-Wing 可线性化的并发对象的函数调用都具有固定的可线性化点[36]，有许多并发对象[37][38][39]的实现会需要一个函数调用的可线性化点在其他线程中被其他函数调用完成。更进一步的，并不是所有并发对象都有一个原子的可线性化点。比如 java.util.concurrent 库中的 exchanger 对象，其作用是在两个并行的线程间交换数据。该对象有一个 exchange(v)方法，调用该方法后如果有另一个线程也在调用 exchange，那么就交换两个调用的参数并返回，否则返回空值表示交换失败。可以看出，这样一个方法的线性化结果不能简单的将每个调用当作原子的事件来处理，因为形如$\mathrm{exchange}(x) \cdot \mathrm{ok}(y) \cdot \mathrm{exchange}(y) \cdot \mathrm{ok}(x)$的历史记录显然是不满足上述 exchanger 对象的要求的，在 exchange(x)调用过程中，并没有任何 exchange 方法在并行执行，该次交换不应该成功。Neiger[40]和 Hemed[41]提出使用集合可线性化（Set-Linearizability）对这种对象的可线性化条件进行形式化，即线性化结果中的一个事件可以是若干个原子事件组成的集合。这样一个集合里的事件被理解为并发执行的事件，而非相互分离的原子事件。

案例 2：在集合可线性化理论中，exchanger 对象的规约为满足以下性质的最大的集合。一个历史h满足这个规约当且仅当（1）它是一个空历史，（2）或者它是一个满足规约的历史h'在结尾加上了一个原子的 exchange

$$h \in S \iff \begin{pmatrix} h = \epsilon \vee (h = h' \cdot \mathrm{exchange}(x) \cdot \mathrm{ok}(\perp) \wedge h' \in S) \vee \\ (h = h' \cdot \{\mathrm{exchange}(x) \cdot \mathrm{ok}(y), \mathrm{exchange}(y) \cdot \mathrm{ok}(x)\} \wedge h' \in S) \end{pmatrix}$$

事件并且返回结果为空值，（3）或者它是一个满足规约的历史h'在结尾加上了一个原子事件的集合，其中包括了一组互相交换成功的 exchange 事件。

Hemed[41]将该理论应用到了 Hendler[38]提出的后退消除栈（Elimination back-off stack）的验证上，其中当两个线程并发的进行入栈和出栈的操作时，程序将会通过上述 exchanger 的机制将入栈的值直接传递给出栈函数作为返回值，从而高效地避免了并发操作的同步带来的延迟。这种机制也被称为线程间帮助（Helping）或者不定可线性化点[36]（Non-fixed Linearization Point），因为其中一方的可线性化点是和另一方的可线性化点一起执行的，可以理解为是后者帮助前者完成了线性化，或者前者的可线性化点在其他函数中被执行。

更进一步的，Castañeda 等人[42][43]提出使用区间可线性化（Interval-Linearizability）来规定更一般化的并发对象的可线性化性质。区间可线性化可以理解为集合可线性化的一个推广，线性化结果中的每个集合不一定需要包含完整的原子事件，一个函数的调用和对应的返回可以在不同的集合中出现，期间可以有若干个原子事件发生。假设一个函数的调用在可线性化结果中某个位置出现了，但是其返回事件并未与该调用在同一个集合中出现，这一般意味着该调用之后的其他事件会依赖于“这个调用开始执行了但并未结束”这一事实，同时该调用在未来返回的时候，其结果会依赖于在这段区间内发生的其他事件。

不难发现，此时的规约已经完全移除了 Herlihy-Wing 可线性化中“线性化结果s是顺序历史”这个要求，Oliveira Vale 等人[44]由此出发，将规约定义为合法的历史记录的集合。该规约不要求线性化结果是顺序历史记录，因此可以非常直白地表达集合可线性化对象和区间可线性化对象的线性化规约。此时，一个程序的可线性化性质被定义为：所有可能的执行都可以通过线性化的方式等效为规约中所定义的某种执行。研究者可以通过定义这个规约允许的执行情况来约束一个程序的正确行为，即功能正确性。该工作使用重写系统（Rewrite System）和博弈语义[45][46]（Game Semantics）来定义可线性化的概念。定义历史记录间的重写关系$h \rightsquigarrow h'$为包含以下三种重写方式的最大的关系：

$t_1:\mathrm{inv}_1 \cdot t_2:\mathrm{inv}_2 \rightsquigarrow t_2:\mathrm{inv}_2 \cdot t_1:\mathrm{inv}_1$，$t_1:\mathrm{res}_1 \cdot t_2:\mathrm{res}_2 \rightsquigarrow t_2:\mathrm{res}_2 \cdot t_1:\mathrm{res}_1$，$t_1:\mathrm{inv} \cdot t_2:\mathrm{res} \rightsquigarrow t_2:\mathrm{res} \cdot t_1:\mathrm{inv}$

即可以交换两个并发的调用事件（inv）或者两个并发的返回事件（res），同时可以将一个返回事件提前到与其并发的调用事件之前。将任意一个实际的历史记录通过这种重写方式得到规约中的历史记录就是可线性化的过程。类似的重写系统也在 Goubault 等人[47]的工作中出现，且在顺序规约的情况下与 Herlihy-Wing 可线性化等价。Oliveira Vale 等人[44]进一步通过博弈语义对其可线性化定义进行改进，并由此得到了更便于证明且更具有可组合性的可线性化理论。

3.3 上下文精化

上下文精化（Contextual Refinement），又称可见行为精化[48][49]（Observational Refinement），是另外一种

```
data: pointer p_i
inc_i() {
   b = false;
   while(!b)
   { v = *p_i; b = CAS(p_i, v, v+1); }
}
```

（a）计数器细粒度实现

```
data: pointer p_s, lock l
inc_s() {
   acquire(l);
   *p_s = *p_s + 1;
   release(l);
}
```

（b）计数器粗粒度规格

图 4：计数器的实现与规格

用来描述并发程序正确性的行为。其通过使用粗粒度的简单且易理解的程序作为规约，来描述细粒度的复杂但高效的程序的功能正确性。粗粒度程序所实现的功能便是细粒度程序需要正确实现的功能。

案例 3：图 4 中，左侧（a）是一个计数器的细粒度实现，其数据包括一个指向整型计数器的指针p_i，在调用inc_i来增加计数器数值的时候，通过 compare and swap（CAS）判断当前的更新是否与当前计数器的值相符（即更新后计数器值确实是当前值加一），并原子地完成这个更新操作。右侧（b）是这个计数器一个粗粒度的实

现代码，用作该计数器的规约。其数据包括计数器指针p_s和一个锁变量 l。在调用inc_s时，它会首先请求获得这把锁，从而当其获得锁之后，将不会再有其他线程并发地对指针p_s的值进行操作，当前线程也就可以安全地将计数器增加，并在操作完成后释放锁从而使其他线程可以继续使用该计数器。

右侧（b）的代码有明确的临界区区分和临界区出入的控制，其正确性是显然的。相较而言，左侧（a）代码没有明确的临界区且不使用锁来进行线程间的协调，其正确性不显然，需要使用规约来说明其功能性与正确性并进行验证。当以下条件成立时，函数inc_s便是inc_i的一个直观的规约：任何对inc_i的调用都可以替换成对inc_s的调用并且替换后程序仍然包含原程序所有的语义表现。一般称inc_i是inc_s的一个上下文精化。如果该性质成立，那么inc_i的功能就可以由inc_s描述和代替，又由于inc_s是显然正确的，可知inc_i的实现也是正确的。这便是使用上下文精化来定义并发程序正确性的思路。

接着，本节形式化地定义上下文精化。假设存在一个具体的程序实现O，和一个抽象的粗粒度的规约$\mathbb{O}$。对于任何一个用到了该程序的顶层（Top-Level）应用A，又称为该程序的上下文（Context），定义$[\![A[O]]\!]$为A在使用程序O执行时会产生的所有可能的可见行为，并相似地定义使用$\mathbb{O}$时的可见行为。称O是$\mathbb{O}$的一个上下文精化，记作$O \sqsubseteq \mathbb{O}$，当且仅当对于任意的应用A，使用O时的可见行为是使用规约$\mathbb{O}$时的可见行为的子集，即$\forall A.[\![A[O]]\!] \sqsubseteq [\![A[\mathbb{O}]]\!]$。对不同的系统，针对不同的目标进行描述的时候，可见行为计算函数$[\![-]\!]$的定义会有不同。一般可以将可见行为定义为执行过程的可见事件的历史记录[36]或者完整程序的指称[50]，即前后程序状态的二元关系。在一些工作中[81][72]，上下文精化也被直接定义为具体程序O和规约$\mathbb{O}$这两个程序的语义上的逻辑关系[51]（Logical Relation）。在上下文精化理论中，一个程序的并发上下文即所有可能的顶层应用A，一般会通过限制用来定义A的程序语言的表达力来约束该并发上下文。

由于上下文精化中具体程序O和规约$\mathbb{O}$有语义上的包含关系，所以对于规约$\mathbb{O}$证明得的性质一般对于具体程序O也能成立。同时规约$\mathbb{O}$一般比具体程序O更简单更易理解，在证明关于规约$\mathbb{O}$的性质的时候会相对简单。相较可线性化的横向可组合性，上下文精化可以实现纵向可组合性[52]，即在证明完整系统$A[O]$正确性的时候，可以首先证明其中更小的模块O可以精化$\mathbb{O}$，再证明简化后的系统$A[\mathbb{O}]$的正确性。一般而言，上下文精化中的上下文A是顶层应用，即$A[O]$组成了完整系统。更进一步的，如果将其语境推广到更一般的上下文，如A可以是另一个调用了O的并发对象，便能反复通过上下文精化[44]将一个复杂的系统分解成多层并发对象，并对每层使用相对精简的上下文精化证明，从而一步一步地将复杂的系统简化并保证每一步证明的模块化。

如果仅考虑类似图 4（b）中整个函数调用都是由临界区保护起来的程序作为规约，那么不难发现，这样的程序组成的并发对象所能产生的历史记录都是顺序历史，其中的事件都是原子的，这与 Herlihy-Wing 可线性化中的规约是一致的。这也就引出了可线性化与上下文精化间的关系。Filipović 等人[50]证明了对于仅产生顺序历史的规约$\mathbb{O}$和一个具体程序O，O可以线性化到$\mathbb{O}$所能产生的历史记录组成的规约，当且仅当，O是$\mathbb{O}$的一个上下文精化。在 Oliveira Vale 等人[44]的工作中，该结论被推广到了不要求规约仅包括顺序历史的设定下，并进一步利用上下文精化来改进可线性化证明的纵向可组合性。

在一些上下文精化的工作中[33][53]，为了能够对进展性质进行描述与证明，会引入终止保持（Termination Preservation）的概念。即如果规约$\mathbb{O}$运行终止，则具体程序O运行同样会终止。这样一种性质在编译器验证中有非常重要的作用，同时也是并发程序验证中对具体程序O的终止性一个很好的刻画。

一些工作中[54][33]也会将模拟关系（Simulation Relation）作为并发程序功能正确性要求。模拟关系是与上下文精化相似的概念，其显示的要求目标程序和规约程序的行为有一致性，且不引入上下文的概念。用以验证模拟关系的程序逻辑以及验证方式与上下文精化的类似，本文不对其定义和验证进行详细展开。

3.4 包含功能正确性的逻辑原子性

逻辑原子性（Logical Atomicity）是基于可线性化点的一种对并发程序的正确性要求。一种对于 Herlihy-Wing 可线性化的直观理解是“一个对象是可线性化的，当且仅当它的所有方法都是在一瞬间生效的”[55]。在上文对于可线性化点的讨论中，如果一个操作的执行可以线性化到一个不与任何在其他线程中执行的操作有重叠的原子操作上，这意味着在这样一种线性化后的执行中，该操作不会对其他线程的执行有影响，

同时其他线程的执行也不会影响该操作的执行。在这种情况下，一个并发执行的程序就和顺序执行的程序没有区别。对于这样的操作，一个简单的霍尔三元组就足以用前后条件来说明该程序的功能正确性，因为 3.1 节中遇到的问题不会在这样一种等效于原子执行的情况下发生。

与可线性化类似，在一些特定语境下，人们更关心一个程序具有逻辑原子性这个性质，而非该程序逻辑原子地执行了什么功能。本文使用“包含功能正确性的逻辑原子性”来强调本文关注使用原子霍尔三元组$\langle P\rangle c\langle Q\rangle$这种规约来同时描述程序的逻辑原子性以及其可线性化点前后的程序状态分别满足对应的前后条件，即功能正确性。有许多工作[30][56]使用逻辑原子性来描述程序的功能正确性。

案例 4：图 4 中的程序inc_i满足原子霍尔三元组：$\langle p \mapsto n\rangle\ \mathrm{inc}_i\ \langle p \mapsto n+1\rangle$。其在 CAS(p, v, v+1)原子指令成功时完成共享内存状态的更新，在执行其他指令以及 CAS 指令失败时不对共享内存产生影响，具有逻辑原子性。

逻辑原子性也有着较大的局限性。首先，它只能描述 Herlihy-Wing 可线性化的程序的正确性，在其他不具有固定的可线性化点的情况下（集合可线性化和区间可线性化），没法很好的发挥作用。同时，基于霍尔三元组的逻辑原子性规约只能约束单个程序的正确性，无法作为完整系统的正确性规约。因此，它也常用作一种证明的中间结果和证明思路，用来辅助其他证明目标的证明。

4 基于程序逻辑的验证方法

并发程序验证研究中的一大主题是并发程序逻辑研究与其在交互式定理证明器中的形式化，目前这些程序逻辑主要包括并发分离逻辑、基于依赖保障的逻辑、关系霍尔逻辑等，它们可以用于验证前文第 3 节所提到的验证目标。这些程序逻辑之间的关系如图 5 所示。

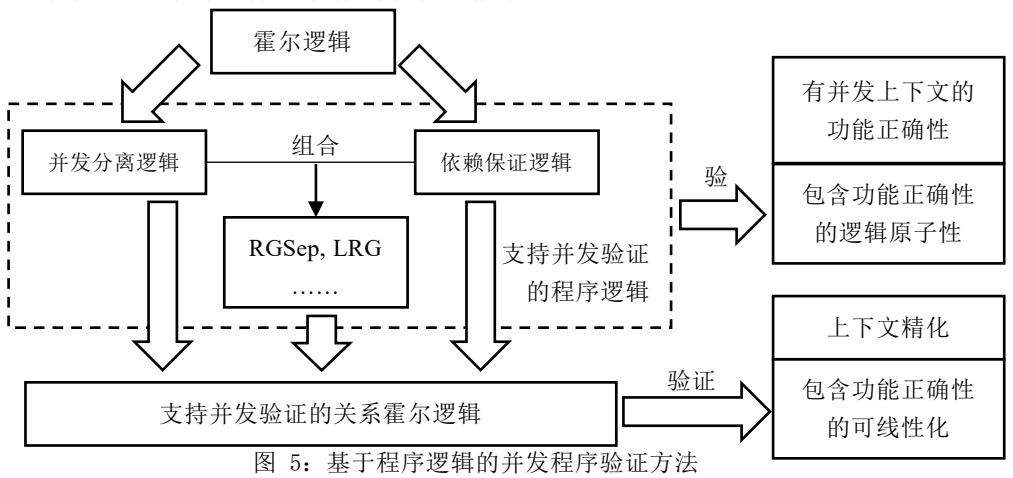

图 5：基于程序逻辑的并发程序验证方法

程序逻辑一般指霍尔逻辑[15]（4.1 节），其使用一套公理化的证明规则对一个程序证明其霍尔三元组，并通过保证证明规则的可靠性（Soundness）来保证证明结果的正确性。但是基础的霍尔逻辑只能支持顺序执行程序的验证，需要在此基础上引入额外的机制才能支持并发程序验证。目前学术界研究中，支持并发程序验证的霍尔逻辑扩展主要有两类：并发分离逻辑[57]（4.3 节）和基于依赖-保证的霍尔逻辑[16]（4.2 节）。也可以通过将两种方法组合使用来更好地对并发程序进行验证。虽然并发分离逻辑和依赖-保证可以对并发程序证明其所满足的霍尔三元组，但是正如 3.1 节中描述的问题，一般的霍尔三元组无法准确地描述并发程序和并发对象的正确性。一种常用的解决方案是在霍尔逻辑的基础上引入关系霍尔逻辑（4.4 节）。结合关系霍尔逻辑和上述两种对并发程序的验证方法就可以实现对于并发程序的可线性化以及上下文精化的验证。

4.1 霍尔逻辑

本节简要地回顾霍尔逻辑[15]（Hoare Logic）有关的基础知识。展示了一个简单的指令式语言和用来验证该语言下的程序的霍尔逻辑。该语言包括了空指令 skip，赋值指令$x = e$，顺序执行指令$c_1;;c_2$，选择语句（if

语句），和循环语句（while 循环）。在该语言下，程序状态$s \in \text{state} \triangleq \text{var_name} \to \text{val}$被定义为变量名到变量数值的映射。为了描述程序状态，定义断言$P, Q \in \text{assertion} \triangleq \text{state} \to \text{Prop}$为关于程序状态的谓词 [3]，且断言逻辑中包含基础的一阶逻辑的逻辑运算符。霍尔逻辑使用图 6 中这样的证明规则来构造一个霍尔三元组的证明树，称可以被构造出来的霍尔三元组为可证的霍尔三元组。

$$c \in \text{command} := \texttt{skip} \mid x = e \mid c_1 \,;; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}(b)\ c$$

$$\text{HOARE-SKIP}\quad \vdash \{P\}\texttt{skip}\{P\} \qquad \text{HOARE-ASSIGN}\quad \vdash \{P[x \mapsto e]\}x = e\{P\}$$

$$\text{HOARE-SEQ}\quad \frac{\vdash \{P\}c_1\{Q'\} \qquad \vdash \{Q'\}c_2\{Q\}}{\vdash \{P\}c_1 \,;; c_2\{Q\}}$$

$$\text{HOARE-IF}\quad \frac{\vdash \{P \wedge [\![b]\!] = \texttt{true}\}c_1\{Q\} \qquad \vdash \{P \wedge [\![b]\!] = \texttt{false}\}c_2\{Q\}}{\vdash \{P\}\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\{Q\}}$$

$$\text{HOARE-WHILE}\quad \frac{\vdash \{I \wedge [\![b]\!] = \texttt{true}\}c\{I\}}{\vdash \{I\}\texttt{while}(b)\ c\{I \wedge [\![b]\!] = \texttt{false}\}}$$

$$\text{HOARE-CONSEQ}\quad \frac{\vdash \{P'\}c\{Q'\} \qquad P \vdash P' \qquad Q' \vdash Q}{\vdash \{P\}c\{Q\}}$$

图 6：顺序执行程序的霍尔逻辑证明规则

在基于交互式定理证明的并发程序研究中，除了需要对程序逻辑进行形式化，还需要对程序逻辑的可靠性在交互式定理证明器中进行形式化验证。程序逻辑的可靠性是指所有可证的霍尔三元组都是有效的（Valid），即其语义确实满足该霍尔三元组。霍尔三元组的有效性（Validity）有多种形式化方式，比如可以使用大步语义（Big-step semantics）进行如下定义：一个霍尔三元组$\{P\}c\{Q\}$是有效的，当且仅当，对于任意满足前条件P的程序状态s_1，程序c从s_1开始的执行不会出错，且任意满足大步语义关系$(c, s_1) \Downarrow s_2$的程序状态s_2满足后条件Q。Cook[58]证明了图 6 的霍尔逻辑在这种有效性的定义下是可靠的（Sound），即任何可证的霍尔三元组都是有效的，且该逻辑是完备（Complete）的，即任何有效的霍尔三元组都是可证的。本文重点讨论各种程序逻辑的特点和如何使用这些程序逻辑进行并发程序验证，不把重心放在解释各种霍尔三元组有效性的形式化上。

4.2 并发分离逻辑

并发分离逻辑[18]是基于分离逻辑[59][60]的一种用于并发程序功能正确性验证的程序逻辑。并发分离逻辑将程序状态（内存）分割成多个分离的部分，每个部分各供一个线程独占，并将共享的内存通过不变量进行保护。任何线程都可以随意使用其独占的内存，但是访问共享内存时需要维护共享内存使其满足不变量。

为了展示并发分离逻辑，本节在图 6 中的程序语言的基础上加入并行组合（Parallel Composition）语句，$c_1 \| c_2$，来表示并发执行程序c_1和c_2，并在它们都结束后完成该并行组合语句的执行从而继续执行后续程序；加入命名临界区语句，$\text{with}\ r\ \text{do}\ c$，来表示程序$c$由以资源$r$命名的临界区保护，对于任意资源$r$，同一时刻只能有一个线程进入以$r$命名的临界区。并发分离逻辑中的一种常用的霍尔三元组的形式为$\Gamma \vdash \{P\}c\{Q\}$，其中$\Gamma$为资源名称和对应的不变量的列表。图 7 中为并发分离逻辑中最为核心的几条证明规则。

$$\text{HOARE-PAR}\quad \frac{\Gamma \vdash \{P_1\}c_1\{Q_1\} \qquad \Gamma \vdash \{P_2\}c_2\{Q_2\}}{\Gamma \vdash \{P_1 * P_2\}c_1 \| c_2\{Q_1 * Q_2\}}$$

$$\text{REGION}\quad \frac{\Gamma \vdash \{P * I\}c\{Q * I\}}{\Gamma, r : I \vdash \{P\}\text{with}\ r\ \text{do}\ c\{Q\}}$$

$$\text{FRAME}\quad \frac{\Gamma \vdash \{P\}c\{Q\} \qquad c\ \text{不使用}\ R\ \text{中的变量和内存}}{\Gamma \vdash \{P * R\}c\{Q * R\}}$$

图 7：并发分离逻辑有关证明规则

（1） Frame 规则

分离逻辑[59][60]（Separation Logic）的目标是在霍尔逻辑的基础上支持可寻址内存的验证，其难点在于同一个内存地址可以被多个指针所指向，当一个指针指向的数据被修改时，有关其他指向该地址的指针的程序

[3] 一个霍尔三元组的前条件一般都是关于前程序状态的谓词，但是后条件就可以有很多选择。本节选用描述后程序状态的谓词以求表达的简洁。有时候为了描述程序（特别是函数式程序）的返回值，后条件会是关于返回值和后程序状态的二元谓词。更进一步，有时候为了更好的表达前后程序状态的关系，后条件也可以是关于返回值、前程序状态、和后程序状态的三元谓词。

断言可能不再成立，从而使证明难以进行。分离逻辑的解决方法是在程序断言中引入分离合取，$P_1 * P_2$，来表示程序断言P_1和P_2各自描述了一块符合对应断言的内存，且各自的内存地址是没有重叠的。如$p_1 \mapsto 1 * p_2 \mapsto 2$表达了两个指针$p_1$和$p_2$各自指向的值为 1 和 2，且$p_1$和$p_2$指向不同的地址。这样就避免了不同的断言之间有共享的地址，在使用一部分断言和其对应的内存进行证明的时候，程序对内存的影响就不会改变其他断言的成立性。Frame 规则是分离逻辑用来支持模块化证明非常重要的规则。其结论中的前后条件分别都分离合取了同一个断言R，如果程序c不会对R所描述的那部分程序状态与内存进行修改，那么证明c的过程中可以不使用断言R，因而其前提是一个前后条件为P和Q的霍尔三元组。这样一种将不需要用到的内存的断言从霍尔三元组中暂时移除的方式，既使得模块化的证明更为简洁，又保证了可以证明一个程序不会访问禁止其访问的内存。

（2）Hoare-par 规则

基于分离逻辑这样一种限制一个程序在证明中可以使用的内存的方式，并发分离逻辑使用 Hoare-par 规则限制一个线程可以使用的内存从而避免线程间不必要的干扰。当证明者使用 Hoare-par 规则证明两个程序的并行执行的时候，他们需要将前条件分成两块分离的部分，分别用程序断言P_1和P_2描述，分别对两个程序证明只使用P_1和P_2中对应的一个断言作为前条件的霍尔三元组。两个程序的后条件Q_1和Q_2依然需要是分离的，并且可以组成$Q_1 * Q_2$作为整个程序的后条件。如此一来，不同线程间便在互相独立的内存空间中运行，互不干扰，因而可以如证明顺序程序一般使用霍尔逻辑进行验证。

但是现实中的并发程序并不会完全在相互分离的内存中运行，它们往往会通过共享内存或者信道进行通信或合作。图 7 中的并发分离逻辑中使用一系列的不变量Γ来保护这部分共享内存，Hoare-par 规则要求两个线程的执行都能维护这些不变量从而实现线程间有序的合作。

（3）Region 规则

Region 规则规定了每个线程如何使用并维护Γ中保护的共享内存。它要求所有线程仅能通过临界区访问对应该临界区资源r的内存，并且该资源的不变量I存在不变量列表中（$\Gamma, r: I$）。一般称不变量I或者资源r保护了对应的共享内存，当该共享内存在临界区外满足不变量I。由于其他所有线程都会维护不变量，即不变量I在进入临界区前成立，且进入临界区后不会再有其他线程并发地访问资源r所保护的共享内存，临界区内程序可以直接将I作为前条件，并使用其中的内存来完成程序执行。但是该程序完成执行后，需要保证后条件中仍然能重新构造出一块满足不变量I内存，因为在临界区结束的时候，只有保证共享内存依然满足不变量，才能使其他线程也能访问共享内存并在进入临界区时能使用不变量I来描述共享内存。注意到，在证明临界区内的程序的时候$r: I$从不变量列表中移除，从而避免重复进入临界区导致获得两个描述同一片共享内存的不变量I。

资源r的创建可以通过以下证明规则实现。语句 new resource r in c限制了资源r只能在程序c中实现。在证明内层语句c的时候，可以选择内存中满足不变量I的一部分作为程序c中并发的语句使用的共享内存，并将其作为不变量加入Γ。在程序c执行结束后，资源r的生命域结束时，共享内存便可以重新加入到当前线程的内

$$\frac{\Gamma, r: I \vdash \{P\} c \{Q\}}{\Gamma \vdash \{I * P\} \text{new resource } r \text{ in } c \{I * Q\}}$$

存中并可以知道其依然满足不变量I。

Brookes[6]证明了包含图 7 中的证明规则以及其他辅助证明的规则组成的并发分离逻辑的可靠性。其使用的有效性是基于迹语义（Trace-based semantics）的指称语义进行定义的。

案例 5：考虑对霍尔三元组$\{p \mapsto 0\}$ new resource r in {with r do *p+=1 ‖ with r do *p+=2} $\{p \mapsto 3\}$的验证。该程序使用资源r控制两个线程并发地对指针p所指的值进行累加。需要证明前条件为p所指的值为 0，后条件为p所指的值为 3。其证明过程可以用以下使用程序断言注释的形式表示。为了完成这个证明，需要首先在两个线程中加入辅助代码 a+=1 和 b+=2。在创建资源的时候，需要引入两个辅助的自然数类型变量a和b，分别表示两个线程对指针 p 已经增加的数值，并将$p \mapsto a + b$作为不变量。在开始执行前，两个线程均未对 p 作任何

$$
\begin{array}{l}
\{p \mapsto a + b * a = 0 * b = 0\} \\
\quad \text{new resource } r \text{ in} \\
\quad r : (p \mapsto a + b) \vdash \{a = 0 * b = 0\} \\
\left\{ \begin{array}{l|l}
\{a = 0\} & \{b = 0\} \\
\text{with } r \text{ do} & \text{with } r \text{ do} \\
\quad \{p \mapsto a + b * a = 0\} & \quad \{p \mapsto a + b * b = 0\} \\
\quad \text{*p += 1;} & \quad \text{*p += 2;} \\
\quad \{p \mapsto a + b + 1 * a = 0\} & \quad \{p \mapsto a + b + 2 * b = 0\} \\
\quad \text{a += 1;} & \quad \text{b += 2;} \\
\quad \{p \mapsto a + b * a = 1\} & \quad \{p \mapsto a + b * b = 2\} \\
\{a = 1\} & \{b = 2\}
\end{array} \right\} \\
\{p \mapsto a + b * a = 1 * b = 2\}
\end{array}
$$

修改，因此a和b均为 0，不变量成立。在进行并发执行的时候，分别将$a = 0$和$b = 0$作为两个线程独占的程序状态分配给它们。两个程序在各自进入临界区时通过 region 规则将r保护的不变量加入到当前线程的程序状态中，并在有了$p \mapsto a + b$该断言作为临界区内程序的前条件之后，可以安全地对于指针p所指值进行累加。累加完成后，通过执行辅助代码使得不变量得到满足。并发执行结束后，资源r的生命域结束，不变量$p \mapsto a + b$重新加入到当前线程程序状态中作为后条件一部分。结合辅助变量的值，便可以推出整个程序的后条件为$p \mapsto 3$。

4.2.1 辅助变量与辅助代码

上述案例的证明中引入了辅助变量和辅助代码。辅助变量一般被形式化为逻辑变量，辅助代码则是为了直观地表现出对逻辑变量的更新，并不会对实际的程序状态进行任何操作。不难看出，移除辅助变量和辅助代码后，程序的行为将保持不变。所以上述证明适用于原程序。

辅助变量最早在 Owicki 和 Gries 的并发程序验证方法[62]中被使用，并且成为了大部分并发程序验证方法（包括并发分离逻辑和基于依赖保证的逻辑）中不可或缺的一个工具。在早期的并发分离逻辑中[59][60] [61]，程序逻辑本身是不直接对辅助变量进行操作的，它们和程序变量一样需要通过实际的代码进行修改。因而许多工作中都会在被验证的程序中加入辅助代码，从而可以显式地操作辅助变量。辅助代码一般需要和实际的代码绑定于同一个原子块中进行原子执行，从而避免辅助程序状态与实际程序状态间的关系被破坏。如上述案例中，如果 a 和 b 的累加不在临界区中执行，不变量I，即指针 p 与变量 a 和 b 之间的关系就无法维持。

后续的并发分离逻辑框架，如 CAP[24]、iCAP[63]、Iris[27]等，在程序断言中直接加入了进行辅助变量更新

$$
\frac{\Delta \vdash P \Rrightarrow P' \qquad \Delta, \Gamma \vdash \{P'\} c \{Q'\} \qquad \Delta \vdash Q' \Rrightarrow Q}{\Delta, \Gamma \vdash \{P\} c \{Q\}}
$$

的操作，称为视角变换（View Shift）。简单来说，程序断言Q是P的一个视角变换，记作$P \Rrightarrow Q$，当且仅当P和Q对实际程序状态的描述是一致的，即满足P的实际程序状态必然满足Q，但是允许证明者在满足一定条件的情况下修改辅助变量。这样一种证明方式一般通过支持视角变换的 consequence 规则实现。比如以下证明规则是 CAP[24]中利用视角变换的主要方式。证明者可以在证明中任意位置，对程序断言中的辅助变量进行更新。但是由于这些辅助变量一般是全局变量，对于其他线程也是可见的，因此这些更新需要记录在Δ中并通过其他证明机制，如依赖保证（见 4.3 节），来证明线程间的有序合作。

4.2.2 并发分离逻辑与其他并发程序特性

本小节进一步讨论如何使用并发分离逻辑验证更多样化的并发程序特性，如：基于 fork-join 的动态线程创建、互斥锁、原子操作等。

（1）动态线程创建

动态线程创建一般指通过 fork 语句创建一个新的线程执行指定程序。该程序执行与父线程是并发的，并且可以通过 join 语句进行同步，需要同步的线程完成执行后，join 语句才会完成执行。类似的动态线程创建在 pthread[64]和 OpenMP[65]中都有实现。基于 fork-join 的动态线程创建与并发组合语句最大的区别在于：（a）其并发执行的跨度不确定。由 fork 指令创建的线程与其他线程同步的时刻是不确定的，join 语句可以动态地出现在其他线程中，并通过控制流语句（如选择语句）来动态决定是否执行同步。而并行组合语句本身就确

定了两个线程 join 的位置，即并行组合语句的结尾。（b）理论上，更一般的动态线程创建的同步并不一定发生在固定的两个线程中。同样由于 join 语句可以是动态的，执行 join 语句的线程可以是环境中的任何一个线程。但是并行组合语句已经明确的是将其结构中并发执行的两个程序进行同步。

为了解决以上难题，Gostman[66]提出使用线程句柄（Thread Handle）对“存在一个正在执行的动态创建的线程”这样一个事实进行描述，并将其定义为一个分离逻辑断言，记作 Thread(t, Q)。其使用以下 fork 规则

$$\text{FORK}\ \frac{\Gamma \vdash \{P\}c\{Q\}}{\Gamma \vdash \{P * R\}x = \text{fork}(c)\{\text{Thread}(x, Q) * R\}} \qquad \text{JOIN}\ \Gamma \vdash \{[\![e]\!] = t \wedge \text{Thread}(t, Q) * R\}\text{join}(e)\{Q * R\}$$

和 join 规则对动态线程创建进行验证。在 fork 规则中，如果动态创建的线程所执行的程序c可以满足以P为前条件的霍尔三元组，那么在创建这个线程的时候需要准备一块与其他部分R分离的且满足P的内存，并在完成创建后从父线程所有的内存中移除这一部分以保证子线程独占该部分，从而实现线程之间独占内存间的分离。对于共享内存的处理方式与 Hoare-par 规则类似，子线程依然可以使用且需要维护Γ中的不变量。在父线程完成创建之后，变量x将会保存子线程的线程 ID，其后条件会增加一个线程句柄 Thread(x, Q)用来表示通过 join 语句将变量x对应的线程进行同步的权限，并且可以告知进行同步的线程有关该子线程运行结束时程序状态的性质，即后条件Q。与之相对应的，在 join 规则中，如果当前线程持有这样一个线程句柄，其就可以执行 join 语句，并且在完成子线程的同步后，其所占有的满足Q的内存将归还给当前线程以供后续使用。线程句柄的使用使得对于动态的线程同步的证明变得可能，只要持有线程句柄就可以有当前线程决定何时同步，并且线程句柄可以通过不变量等方式转移至实际执行同步语句的线程，由其进行同步。线程句柄作为一个程序断言，是关于程序状态的一个谓词。但是只包含局部变量和内存的程序状态是不足够的，需要将线程池也作为程序状态的一部分。一个可行的方案是将从线程 ID 到对应线程后条件的部分映射作为程序状态的一部分。

在 Iris 程序逻辑框架[27]中，它们同样支持使用 fork 语句进行动态线程创建。但是不同的是，Iris 中的子线程在运行终止后由系统进行回收，并不通过任何 join 语句进行同步。因而，其 fork 规则的后条件中不需要保留线程句柄，只需证明在将子线程依赖的内存从父线程中移除后，主线程依然能正常执行至终止即可。

（2）锁

锁是实现临界区的一种非常重要的手段。一般会通过 acquire 指令获取一把锁，并通过 release 指令释放一把锁。一个正确的锁的实现会保证在任何时刻最多只有一个线程持有这个锁，从而使得一组配对的 acquire

$$\text{MAKE-LOCK}\ \vdash \{\text{emp}\}\text{make\_lock}(l)\{\text{isLock}_1(l, I) * \text{locked}(l)\} \qquad \text{DEL-LOCK}\ \vdash \{\text{isLock}_1(l, I) * \text{locked}(l)\}\text{del\_lock}(l)\{\text{emp}\}$$

$$\text{ACQ-LOCK}\ \vdash \{\text{isLock}_\pi(l, I)\}\text{acquire}(l)\{\text{isLock}_\pi(l, I) * \text{locked}(l) * I\}$$

$$\text{REL-LOCK}\ \vdash \{\text{isLock}_\pi(l, I) * \text{locked}(l) * I\}\text{release}(l)\{\text{isLock}_\pi(l, I)\}$$

指令和 release 指令之间的区域成为临界区。上文中的以资源命名的临界区就可以拆分为这样一组对应的锁的操作。但是相较而言，锁的获取和释放是动态的，这一点与动态线程创建类似。同时，一个锁是可以被动态地创建与销毁的，这也对其验证带来一定的难度。Gostman[66]提出使用类似于线程句柄的方式和使用分数权限（Fractional Permission）对锁有关的操作进行验证。类似的方法也在许多工作中[24][63][27]用作锁的具体实现的规约。其程序逻辑有关锁的证明规则如下所示。其中的程序断言使用到了 isLock 句柄和 locked 句柄。$\text{isLock}_\pi(l, I)$表示变量l存储了一个锁，同时该锁保护的共享内存满足不变量I。其中的下标π是一个大于 0 小于等于 1 的有理数，表示该断言的权限大小。根据 make-lock 规则，当锁刚刚通过 make_lock 指令创建出来的时候，isLock 句柄会拥有完整的 1 权限。该程序断言满足性质：$\text{isLock}_{\pi_1+\pi_2}(l, I) \Leftrightarrow \text{isLock}_{\pi_1}(l, I) * \text{isLock}_{\pi_2}(l, I)$，即可以将一个$\pi_1 + \pi_2$权限的句柄拆分成两个各拥有$\pi_1$和$\pi_2$权限的句柄，并将它们分别分配到不同线程中，以供这些线程对锁进行操作。只有一个线程的程序状态中拥有 isLock 句柄，它才可以对该锁进行操作。 在创建锁的同时，后条件会新增一个 locked(l)句柄用来表示变量l存储的锁处于锁定的状态。在 acq-lock 规则中，

如果一个线程拥有一部分$\text{isLock}_\pi(l,I)$句柄，那么它就可以获取该锁，且后条件会增加 locked(l)句柄和该锁保护的共享内存并获知其满足不变量I。由于 locked(l)句柄有性质：$\text{locked}(l) * \text{locked}(l) \Rightarrow \text{False}$，所以无法在已经获取锁的情况下重复获取。在成功获取锁之后，由于不会有其他线程成功获取同一个锁，当前线程可以随意修改该锁保护的内存。但正如 rel-lock 规则的前条件所表达的，在释放锁的时候需要使共享内存重新满足不变量，从而使其他线程依然可以通过不变量使用锁和访问其保护的共享内存。最后，当一个线程中还原出具有完整权限的$\text{isLock}_1(l,I)$句柄，并且该线程已经获得了该锁，其可以通过 del-lock 规则将该锁占据的空间释放。

（3） 一般的原子操作

并发分离逻辑同样可以对更一般的原子操作，如 FetchAndAdd（FAA）和 CompareAndSwap（CAS），组成的程序进行验证。它们可以理解为临界区的特例，即临界区中仅包含一个指令，因而同样可以使用不变量来进行验证。比如可以用以下证明规则对原子指令c进行验证。其中$\boxed{I}$表示存在一个满足不变量I的共享内存

$$\frac{\vdash \{I * P\}c\{I * Q\} \qquad c \text{ 是原子指令}}{\vdash \{\boxed{I} * P\}c\{\boxed{I} * Q\}}$$

且该内存现在为未被使用的状态，该断言与锁的不变量句柄类似。该原子指令证明规则允许在执行c的时候打开不变量，将其加入到当前线程的前条件以供证明使用，且需要保证c执行完成后不变量依然能被满足，从而可以关闭不变量重新得到$\boxed{I}$句柄。

4.3 基于依赖-保证的霍尔逻辑

依赖-保证[16][17]（Rely-Guarantee）方法将其他线程对于共享内存的影响形式化为当前线程的依赖（Rely），记作$\mathcal{R}$，并将当前线程对于共享内存的影响形式化为它的保证（Guarantee），记作$\mathcal{G}$。依赖-保证方法使用这两个额外的条件描述并发环境下各线程对共享内存的操作，从而可以在证明单个线程的程序的时候知晓环境（其他线程）会对其执行产生的影响，进而实现对于并发程序的验证。依赖和保证被形式化地定义为关于程序状态的二元关系，用来记录环境和当前线程会对共享内存进行的原子操作。

案例 6：图 4（a）中的inc_i函数的保证$\mathcal{G}$定义如下。该函数中的 CAS 指令会原子地将指针 p 所指的内存空间

$$(s_1,s_2)\in\mathcal{G} \iff \exists n,l.\begin{pmatrix} s_1.\text{var}[p]=l \wedge s_1.\text{mem}[l]=n \wedge \\ s_2.\text{var}=s_1.\text{var} \wedge s_2.\text{mem}=s_1.\text{mem}[l\mapsto(n+1)]\end{pmatrix}$$

存储的值加一。由于其他指令都是对局部变量的操作和对共享内存的只读操作，所以不需要在保证中记录。如果并发环境中其他线程都只能调用 inc_i 函数，那么对于该线程来说，环境的依赖$\mathcal{R}$与此处$\mathcal{G}$的定义是相同的。

在依赖-保证证明框架下霍尔三元组的形式为$\mathcal{R},\mathcal{G}\vdash\{P\}c\{Q\}$，表示在环境行为被依赖$\mathcal{R}$限制的情况下，程序$c$满足前后条件$P$和$Q$，且其对共享内存的影响被保证$\mathcal{G}$所约束。与 4.2 节相同，本节考虑图 6 中的程序语

$$\text{HOARE-PAR}\ \frac{\mathcal{R}\cup\mathcal{G}_2,\mathcal{G}_1\vdash\{P_1\}c_1\{Q_1\} \quad \mathcal{R}\cup\mathcal{G}_1,\mathcal{G}_2\vdash\{P_2\}c_2\{Q_2\}}{\mathcal{R},\mathcal{G}_1\cup\mathcal{G}_2\vdash\{P_1\wedge P_2\}c_1\|c_2\{Q_1\wedge Q_2\}} \qquad \text{ATOMIC}\ \frac{\text{stable}(\mathcal{R},P) \quad \text{stable}(\mathcal{R},Q) \quad \text{atomic}(c) \quad \vdash\{P\}c\{Q\} \quad \forall s_1\in P,s_2\in Q.(s_1,s_2)\in\mathcal{G}}{\mathcal{R},\mathcal{G}\vdash\{P\}c\{Q\}}$$

图 8：依赖保证有关证明规则

言加上并发组合语句和原子指令，同时加入图 8 中的证明规则：Hoare-par 规则和 Atomic 规则。

在 Hoare-par 规则中，为了证明并发执行$c_1\|c_2$在被$\mathcal{R}$限制的环境下满足的霍尔三元组，需要分别证明两个程序的霍尔三元组。以c_1为例，它与c_2并行执行，因而c_2相对于c_1是一个环境线程，加上相对于$c_1\|c_2$的环境线程就构成了相对于c_1的完整的环境。如果c_2可以给出其对共享内存操作的保证为$\mathcal{G}_2$，则可以知道对于c_1而言，其完整的环境行为是被$\mathcal{R}\cup\mathcal{G}_2$所约束的。因此可以在假设$c_1$对环境的依赖是$\mathcal{R}\cup\mathcal{G}_2$的情况下证明其需要满足的霍尔三元组。为了保证证明的自洽，c_2的霍尔三元组需要保证它对共享内存的影响确实满足保证$\mathcal{G}_2$。如

果从c_2的证明出发，把c_1当作环境线程也是一样的道理。通过这种方式，Hoare-par 使得在证明某个线程的时候可以把环境线程对于共享内存的影响记录下来，以供证明的时候使用。这是依赖-保证实现对并发程序的证明的关键。该证明规则中前后条件的理解相对简单，完整的并发执行的前条件$P_1 \wedge P_2$需要可以推出两个线程各自的前条件P_1和P_2，两个线程结束运行后共享的程序状态会同时满足Q_1和Q_2，因此后条件为$Q_1 \wedge Q_2$。

另一条 Atomic 规则则明确了如何使用依赖-保证证明一个原子指令c的霍尔三元组。首先，其霍尔三元组的前后条件需要在环境的影响下是稳定的，即 stable$(\mathcal{R}, P)$和 stable$(\mathcal{R}, Q)$需要成立。其定义如下：

$$\text{stable}(\mathcal{R}, P) \Leftrightarrow \forall s_1 \in P, s_2.\,(s_1, s_2) \in \mathcal{R} \Rightarrow s_2 \in P$$

即任何满足该断言的程序状态s_1，在受到环境线程的影响后变成了状态s_2，新的程序状态依然满足该断言。该条件反映了并发执行过程中的不确定性：在一个原子指令执行前后，无法确定是否有其他线程对共享内存进行了修改，因而需要保证前后条件在环境的影响下是不变的。其次，需要证明一个关于原子指令的简单霍尔三元组$\vdash \{P\}c\{Q\}$。该霍尔三元组是 4.1 节中的顺序执行霍尔逻辑下的三元组，它直接的反应了该原子操作对程序状态的改变。在有些工作中[67]，该霍尔三元组是通过顺序执行霍尔逻辑证明得到的，同时这个程序c也可以是一个由临界区包围的一个原子程序块；在有些工作中[44]，该霍尔三元组是直接由语义得到的，其类似于三元组有效性的定义，但直接参与到逻辑的证明中。最后，由这个霍尔三元组前后条件规定的状态更新需要在该程序的保证$\mathcal{G}$中进行记录，从而可以作为其他线程中的依赖来帮助证明。

案例 7：本节证明霍尔三元组$\text{id}, \mathcal{G}_1 \cup \mathcal{G}_2 \vdash \{p \mapsto 0\}\langle$*p+=1$\rangle \parallel \langle$*p+=2$\rangle\{p \mapsto 3\}$来展示基于依赖保证的逻辑。该程序与 4.2 节中的案例非常相似，但是这里使用原子块$\langle - \rangle$表示两个对指针所指值得操作为原子操作。由于该霍尔三元组中的程序就是完整程序，没有其他环境线程的干扰，所以其依赖为id，表示不会改变共享内存。$\mathcal{G}_1$和$\mathcal{G}_2$分别为左右两个线程的保证，其定义分别为

$$(s_1, s_2) \in \mathcal{G}_1 \Leftrightarrow \exists n.\, s_1 \vDash p \mapsto n \wedge s_1 \vDash p \mapsto n+1,\ \ (s_1, s_2) \in \mathcal{G}_2 \Leftrightarrow \exists n.\, s_1 \vDash p \mapsto n \wedge s_1 \vDash p \mapsto n+2$$

由并发组合规则，如下构造两个线程前后条件并证明以下霍尔三元组来证明该程序。其中$\mathcal{G}_2$和$\mathcal{G}_1$分别作为两个线程的依赖。为了保证前后条件是稳定的，前后条件需要同时考虑对方线程是否完成执行的两种情况。

$$\mathcal{G}_2, \mathcal{G}_1 \vdash \{p \mapsto 0 \vee p \mapsto 2\}\langle\text{*p+=1}\rangle\{p \mapsto 1 \vee p \mapsto 3\},\ \ \mathcal{G}_1, \mathcal{G}_2 \vdash \{p \mapsto 0 \vee p \mapsto 1\}\langle\text{*p+=2}\rangle\{p \mapsto 2 \vee p \mapsto 3\}$$

4.3.1 依赖保证与其他并发程序特性

由于依赖保证直接对线程间干扰进行记录，所有原子指令对共享内存的影响都可以统一地表达为依赖保证中的状态转移关系，一般不需要额外的机制来辅助证明。类似于锁的同步指令一般也为原子指令，可以用相同的方式处理。在对较大的系统进行依赖保证的逻辑证明时，一般会使用一个稳定的程序断言作为整个并发系统的不变量，并在各线程执行过程中任何时刻都得到满足。但是依赖保证无法很好的支持动态线程创建，因为其依赖和保证相对一个线程而言是静态的。在通过 fork 创建新线程之后，父线程需要额外考虑子线程的干扰并将其作为依赖的一部分，但是在 fork 执行前不应该将其作为依赖的一部分，否则这部分不存在的对共享内存的干扰会使得证明难以得到一个条件足够强的结论。Dodds 等人[68]在依赖保证的基础上提出了否定保证（Deny-Guarantee），通过将依赖和保证两个关系内嵌至程序状态中，使得程序断言本身就能描述当前线程的依赖与保证。其使用分离逻辑对不同状态转移关系进行拆分和权限调整实现在必要的位置将新的子线程的保证（父线程的依赖）进行分离。他们同样使用线程句柄[66]实现线程创建与 join 同步时的程序状态转移。

4.3.2 局部依赖保证

从 Hoare-par 规则可以看出，一般在使用依赖-保证证明完整的程序时，会将所有环境线程的保证记录在当前线程的依赖中，这会导致依赖这个二元关系包含许多共享状态的转移关系。即使当前线程仅对共享内存中很小一部分进行证明，也需要在依赖保证中将完整的共享内存上的所有转移关系记录下来。局部依赖保证[67]（Local Rely Guarantee）结合了并发分离逻辑的思想，实现了类似 Frame 规则的机制，将证明中不会用到的共享内存的有关依赖和保证进行隐藏。其霍尔三元组的形式为$\mathcal{R}, \mathcal{G}, I \vdash \{P\}c\{Q\}$，在依赖保证原有霍尔三元组的基础上加入了不变量$I$对$\mathcal{R}, \mathcal{G}$所描述的共享内存的范围进行约束。定义$I$约束$\mathcal{R}$的范围，记作$I \rhd \mathcal{R}$，当且仅当，（1）$\mathcal{R}$中的任何转移关系$(s_1, s_2)$的前后状态都能满足$I$，即$s_1 \in I$，$s_2 \in I$，（2）$\mathcal{R}$包含了$I$上的恒等转移

关系，即对任意$s \in I$，$(s,s) \in \mathcal{R}$,（3）I是精确的。

在证明过程中，需要维护$I \rhd \mathcal{R}$和$I \rhd \mathcal{G}$一直成立，这样才能在合适的时刻将证明中不会用到的共享内存部分同时从三者中移除。为了将依赖保证中这部分共享内存有关的转移关系移除，定义$\mathcal{R}_1 * \mathcal{R}_2$为两个在相互

$$(s,s') \in \mathcal{R}_1 * \mathcal{R}_2 \iff \exists s_1, s_2, s_1', s_2'. \begin{pmatrix} s = s_1 \uplus s_2 \wedge s' = s_1' \uplus s_2' \wedge \\ (s_1, s_1') \in \mathcal{R}_1 \wedge (s_2, s_2') \in \mathcal{R}_2 \end{pmatrix}$$

分离的程序状态上的转移关系$\mathcal{R}_1$和$\mathcal{R}_2$的分离合取，即该转移关系同时改变两块分离的内存，且分别按照$\mathcal{R}_1$和$\mathcal{R}_2$规定的转移关系它们进行修改，且结果依然是两块分离的内存。

在以上定义的基础上，本节假设程序状态的所有部分都是线程间共享的，并给出简化后的局部依赖保证的 Frame 规则。其中$\mathcal{R}'$、$\mathcal{G}'$、I'和R分别是约束同一块内存的依赖、保证、不变量和断言，并由$I \rhd \mathcal{R}$、$I \rhd \mathcal{G}$和$R \Rightarrow I$这三个前提对这一约束进行保证。同时要求R在$\mathcal{R}'$的影响下是稳定的，这样才能保证结论中前后条件在完整的环境$\mathcal{R} * \mathcal{R}'$的影响下是稳定的。实际在证明程序$c$的时候不需要用到这部分内存，可以仅证明在剩余的共享内存中该程序的霍尔三元组是可证的，即$\mathcal{R}, \mathcal{G}, I \vdash \{P\}c\{Q\}$。

4.4 关系霍尔逻辑

虽然并发分离逻辑和依赖保证已经具有对并发程序证明霍尔三元组的能力，但是仍无法系统化地支持可

$$\frac{\mathcal{R}, \mathcal{G}, I \vdash \{P\}c\{Q\} \qquad \text{stable}(\mathcal{R}', R) \qquad I \rhd \mathcal{R}' \qquad I \rhd \mathcal{G}' \qquad R \Rightarrow I}{\mathcal{R} * \mathcal{R}', \mathcal{G} * \mathcal{G}', I * I' \vdash \{P * R\}c\{Q * R\}} \text{FRAME}$$

线性化和上下文精化的验证。关系霍尔逻辑（Relational Hoare Logic）是一种在霍尔逻辑中对两个对象间的关系进行验证的方法。它可以建立一个程序的历史记录和该程序的线性化规约间的可线性化关系，也可以建立一个程序和另一个作为其规约的程序间的上下文精化关系。通过在并发分离逻辑或者基于依赖保证的逻辑中实现关系霍尔逻辑，便可以支持并发程序的功能正确性验证。

关系霍尔逻辑是建立在逻辑关系[51][19]（Logical Relation）之上的。在许多工作中[69][70][71]，逻辑关系被用来证明程序之间的精化（Refinement）关系。逻辑关系一般会在两个类型中的元素之间建立关系。比如对于两个类型为$A \to B$的函数，可以这样定义它们之间的逻辑关系：对于任何类型A中的元素，将“有联系的”元素作为两个函数的输入，那么各自的输出需要是类型B中的“有联系的”元素。这里“有联系”这一性质同样可以定义为在类型A和B中的一个逻辑关系，比如相等关系。后续的一系列工作[72][73][81]提出了关系霍尔逻辑，通过程序逻辑的方式对两个程序间的逻辑关系进行刻画，并实现了并发程序证明的支持。

在关系霍尔逻辑中，程序断言不再是仅关于一个程序状态的谓词，而是关于具体（Concrete）程序状态和抽象（Abstract）程序状态的二元关系，称为关系断言（Relational Assertion）。以上下文精化为例，前者是具体实现（Implementation）的程序所运行的程序状态，后者是程序规约（Specification）所运行的程序状态。使用关系断言的霍尔三元组，可以方便地表达并证明具体程序状态和抽象程序状态间的逻辑关系。比如，3.3 节中图 4 的案例中，可以用$\exists n. p_i \mapsto n * p_s \mapsto n$表达两个程序状态中指针$\mathrm{p}_i$和$\mathrm{p}_s$所指的值需要是一样的。简而言之，关系霍尔逻辑就是程序断言为关于两种状态的二元关系的霍尔逻辑。视情况而定，两种程序状态可以有不同的形式化定义，在证明并发程序时，也可以使用并发分离逻辑（4.2 节）和依赖保证（4.3 节）与关系霍尔逻辑进行结合，从而支持并发程序的关系证明。

4.4.1 关系霍尔逻辑与上下文精化

在使用关系霍尔逻辑证明上下文精化的时候，关系断言所描述的具体程序状态包括实际程序使用的物理内存和并发证明可能需要的辅助程序状态，抽象程序状态包括规约程序使用的程序状态和规约程序本身。比如对于 3.3 中图 4 的案例，可以写出以下在顺序执行环境中的一个使用分离逻辑的霍尔三元组 [4]。

---

[4] 在并发的情况下，需要将该霍尔三元组中的$\exists n. p_i \mapsto n * p_s \mapsto n$作为不变量从前后条件中移除。

$$\{\exists n.\, p_i \mapsto n * p_s \mapsto n * [\mathrm{inc}_s]\}\ \mathrm{inc}_i()\ \{\exists n.\, p_i \mapsto n * p_s \mapsto n * [\mathrm{end}]\}$$

其前后条件均描述了物理内存中的指针p_i和抽象状态中指针p_s的值，前条件中$[\mathrm{inc}_s]$表示需要执行的规约程序为函数inc_s，后条件中[end]表示需要执行的规约程序已经完成。该霍尔三元组的有效性定义需要蕴含前条件中的规约程序和实际程序间的上下文精化关系，即对于任意的c_s，c_i，P，Q，有$\{P * [c_s]\}c_i\{Q * [\mathrm{end}]\} \Rightarrow c_i \sqsubseteq c_s$。这样便使霍尔三元组建立起了两个程序间的逻辑关系，即上下文精化关系。

在一些工作中[81][73][72]，霍尔三元组所描述的程序和程序断言中的规约程序会有一个线程 ID 参数，并且通过约束这两个 ID 有一对一的关系从而保证规约程序是霍尔三元组所描述的程序的规约。同时，它们的程序语言采用了函数式语言，所有程序表达式执行结果都是一个数值，需要对实际程序和规约程序的执行结果也建立逻辑关系。其霍尔三元组形式是$\{P * i' \hookrightarrow c_s\}i{:}\ c_i\{v_i.\, Q * i' \hookrightarrow v_s * \varphi(v_i, v_s)\}$。它表示实际程序$c_i$在线程$i$中执行，规约程序$c_s$在抽象线程池中的线程$i'$上执行，执行结束的结果为$v_i$和$v_s$且满足$\varphi$所规定的逻辑关系。

这样的关系霍尔三元组的证明中，一般会区别对待实际程序和规约程序。实际程序是直接表达在霍尔三元组中的，一般通过针对单独指令的规则（如赋值语句规则、锁有关证明规则、原子指令证明规则等）和组合指令的规则（如循环规则、并发组合规则等）进行结构化的证明。而规约程序则会通过支持规约程序执行的 consequence 规则进行执行。一般会使用如下的 abs-conseq 规则中的$\Rrightarrow_s$关系表达在关系断言中的规约程序

$$\text{ABS-CONSEQ}\ \frac{P \Rrightarrow_s P' \qquad \vdash \{P'\}c\{Q'\} \qquad Q' \Rrightarrow_s Q}{\vdash \{P\}c\{Q\}} \qquad \text{ABS-STEP}\ \frac{\vdash_s \{P\}c_s\{Q\}}{P * [c_s ;; c_s'] \Rrightarrow_s Q * [c_s']}$$

的执行。$P \Rrightarrow_s P'$不但表达了关于具体程序状态的程序断言的蕴含关系，还可以通过类似 abs-step 规则的方式对抽象程序状态和规约程序进行推导。这里的 abs-step 规则表达了一个断言（包括了程序状态断言P和规约程序$c_s;;c_s'$）可以通过规约程序的执行得到另一个断言（包括了程序状态断言Q和规约程序c_s'），仅当被执行了的规约程序c_s将程序状态从满足P的状态更新为满足Q的状态，且其仅更新了抽象程序状态，即满足规约程序霍尔三元组$\vdash_s \{P\}c_s\{Q\}$。这样一种形式的关系霍尔三元组的证明方式在许多工作中都有使用：（a）在 CaReSL 逻辑[81][74]中，定义了一个规约程序的重写（rewrite）系统作为$\Rrightarrow_s$关系。规约程序在执行类似循环和选择这类组合语句时不会对程序产生影响，不改变程序状态有关断言，可以直接对断言中的规约程序部分进行重写。如果需要执行一条会对程序状态产生影响的指令，则需要证明满足对应前后条件的规约程序霍尔三元组。由于组合语句可以全部通过重写完成执行，只需要定义各种原子的规约指令的霍尔三元组需要满足的前后条件即可。（b）在其他大部分的工作中，abs-step 规则中的前提可以不使用规约程序霍尔三元组，而是用语义的形式描述执行的规约程序会对程序状态产生怎么的改变从而定义$\Rrightarrow_s$关系。如 Xu 等人[52]的并发操作系统验证框架中和 Liang 等人[36][33]的并发程序验证框架中，均直接使用规约程序的语义定义了$\Rrightarrow_s$关系。在 ReLoC[72][73]的并发关系逻辑框架中，程序断言的蕴含关系就结合了规约程序的语义，直接将$\Rrightarrow_s$关系定义为了断言蕴含关系的一部分。（c）在有些工作中[36][33]，规约程序的执行是和具体程序的执行绑定的。即不通过类似 abs-conseq 规则进行抽象程序状态更新和规约程序执行，而是直接在证明具体原子指令时将对应的规约程序一同完成执行。

不难发现，抽象程序状态和作为程序状态一部分的规约程序和 4.2.1 节中提到的辅助程序状态非常类似。通过 abs-conseq 规则进行抽象程序状态更新和规约程序执行的过程也与辅助程序状态的更新如出一辙。事实上，抽象程序状态和规约程序可以理解为是一种特殊的辅助程序状态，其定义方式完全相同。但是不同的是一般的辅助程序状态不应该在最终霍尔三元组的有效性中出现，因为这些状态是假想的，仅用来辅助并发程序的证明，真正需要的是原程序在物理程序状态中运行的正确性。相对的，在关系霍尔逻辑中，验证者真正需要的就是具体程序和规约程序之间的关系以及物理程序状态和抽象程序状态之间的关系，用于定义它们的这部分辅助程序状态是必须在霍尔三元组有效性定义中出现的。

案例 8：本小节以 3.3 中图 4 的程序为例来展示关系霍尔逻辑的使用。这里使用支持关系霍尔逻辑的并发分离逻辑证明inc_i和inc_s间的精化关系，即程序断言中包含抽象程序状态和规约程序作为辅助程序状态。

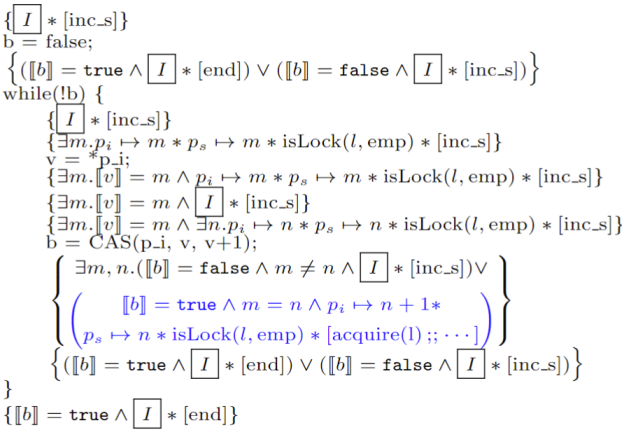

图 9：计数器的上下文精化证明

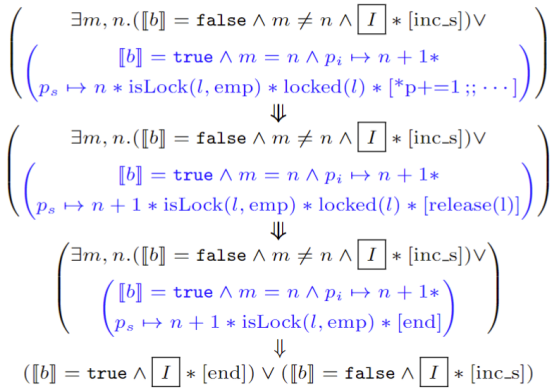

图 10：抽象程序的符号执行

假设在创建计数器时，指针p_i和p_s所指向的值均为 0，并且规约程序在创建锁l之后立刻释放锁，可以定义不变量为$I \triangleq \exists n.\, p_i \mapsto n * p_s \mapsto n * \mathrm{isLock}(l, \mathrm{emp})$以供后续的原子操作使用。除了约束两个指针所指的值是相同的，该不变量还包括了锁l的不变量以便后续规约程序对锁的操作可以正常进行。图 9 展示了计数器的上下文精化的证明，其中的 while 循环的不变量为$(\mathrm{eval}(b) = \mathrm{true} \wedge \boxed{I} * [\mathrm{end}]) \vee (\mathrm{eval}(b) = \mathrm{false} \wedge \boxed{I} * [\mathrm{inc}_s])$，即如果表达式$b$的值为真的时候，则规约程序全部完成执行，否则规约程序仍保持原样。指针p_i的求值语句和 CAS 语句均为原子指令，根据原子指令证明规则，可以将不变量打开并加入前条件用以证明。指针p_i的求值语句执行后，可以知道 v 的值为某个自然数m，同时共享内存未被修改，可以重新满足不变量。CAS 语句执行后，需要分情况讨论。如果 CAS 返回值为假，说明共享内存未被修改，不变量依然满足，此时循环不变量的第二个分支成立。如果 CAS 返回值为真，说明p_i的值累加成功，此时需要使用 abs-conseq 规则对规约程序也进行执行从而使不变量再次得到满足。蓝色部分的断言为规约程序的证明过程：（1）首先，不变量中提供了$\mathrm{isLock}(l, \mathrm{emp})$句柄，使得锁的获取指令可以安全执行。由于实际程序中 CAS 这个原子指令已经保证了线程间不会有数据竞争，所以不再需要规约程序的锁保护任何数据，锁的不变量可以为 emp。事实上，由于这里的 CAS 指令是一个可线性化点，规约程序中所有的指令都可以理解为在此处原子地完成了执行。规约程序的锁仅仅是用来约束整个规约程序是一个原子块。（2）然后，规约程序需要对指针p_s进行修改。由于原子指令 CAS 打开的不变量包含了$p_s \mapsto n$，所以可以安全执行该指令并将其值修改为$n+1$。（3）最后，释放规约程序中的锁，使得不变量重新得到满足，并结束原子指令 CAS 关闭不变量。此时，循环不变量重新得到满足，完成循环的证明。（4）退出循环后，由循环证明规则可知，后条件为循环不变量中第一个分支，即所有规约程序完成了执行。结合关系霍尔三元组的有效性定义和关系霍尔逻辑的可靠性，可知程序inc_i是inc_s的上下文精化。

4.4.2 关系霍尔逻辑与可线性化

许多通过程序逻辑证明可线性化的工作[75][36][30][76]都是从可线性化点逻辑原子性的角度出发的。它们通过程序逻辑证明一个并发对象的所有方法都可以作为一个原子的规约指令的上下文精化，或证明一个并发程序对共享内存的实际效果等效于其中某个原子指令的效果，从而证明该程序具有可线性化点或是逻辑原子的。由以原子指令作为规约程序的上下文精化与 Herlihy-Wing 可线性化之间的关系，可以直接得出并发对象的可线性化。但是这种方式有一个较显著的问题：这种方法实际上是在证明上下文精化，且会限制规约程序为原子的。这使得证明集合可线性化时，不定可线性化点[36]和线程间帮助[55]的证明会遇到一定的困难。并且它从根本上断绝了证明一个并发对象具有区间可线性化性质的可能，因为此时一个方法的实际生效过程是一个区间，在不同时刻对共享内存产生的效果是不同的，无法使用一个可线性化点进行表达。

除了证明逻辑原子性，关系霍尔逻辑也可以直接从程序执行历史记录的角度进行论证从而证明可线性化。

通过这种方法证明得到的关系霍尔三元组的有效性可以直接得出一个并发对象与其线性化规约间的可线性化关系，而不再需要通过上下文精化来间接地证明可线性化。该方法思路在 Herlihy-Wing 提出可线性化的工作[35]中就有体现，在 Hemed 的集合可线性化工作[41]中就有使用。而后 Khyzha 等人[76][77]的工作中给出了该逻辑的形式化，并通过基于偏序和无环有向图的历史记录定义对并发对象进行验证。其工作可以支持依赖于未来执行的可线性化点的证明，实现类似于先知变量[78]（Prophecy Variable）的效果。近期，Oliveira Vale 等人[44]通过博弈语义[45][46]（Game Semantics）将可线性化推广得到了可以支持垂直组合（Vertical Composition）的可线性化并且支持所有可能的可线性化规约（包括集合可线性化和区间可线性化），并基于 Khyzha 等人[76][77]的程序逻辑给出了一套相对完善的且支持所有可线性化规约证明的程序逻辑。

在 Oliveira Vale 等人[44]的程序逻辑中，抽象程序状态被定义为当前并发对象的执行历史记录的可能的可线性化结果（Possibility），记作ρ，具体程序状态 [5]定义为局部变量映射和当前并发对象的实际执行历史记录[44]，分别记作Δ和s。前者仅包括了验证的并发对象的调用事件和返回事件，后者还包括了验证的并发对象所使用的底层对象的行为，其中包括一些用于线程间同步的事件，在一个正确的并发对象实现中，这些事件决定了当前并发对象的行为满足其可线性化规约。在证明过程中，需要始终维护两个不变量：对于满足断言具体历史记录s和抽象历史记录ρ，s可以线性化到ρ，且ρ可线性化到可线性化规约（即去掉未返回事件后属于规约给出的线性化结果的集合）。同时，证明规则会保证每个程序断言都包含了所有可能的具体历史记录s，

$$\frac{\begin{array}{ll}\mathrm{stable}(\mathcal{R},P) & \mathrm{stable}(\mathcal{R},Q)\\ c\ \text{是原子指令} & \mathcal{G}\vDash_t\{P\}c\{Q\}\end{array}}{\mathcal{R},\mathcal{G}\vdash_t\{P\}c\{Q\}}\qquad \mathcal{G}\vDash_t\{P\}c\{Q\}\iff\begin{pmatrix}\forall(\Delta,s,\rho)\in P.\,\forall(\Delta',s')\in[\![c]\!]_t(\Delta,s).\,\exists\rho'.\rho\dashrightarrow\rho'\\ \wedge(\Delta',s',\rho')\in Q\wedge((\Delta,s,\rho),(\Delta',s',\rho'))\in\mathcal{G}\end{pmatrix}$$

这样便可以证明该并发对象所有可能的执行都可以线性化到其规约。

其逻辑的核心证明规则是如上原子指令规则。该规则与一般依赖保证的原子指令规则相似，但其中的关于原子指令的语义霍尔三元组$\mathcal{G}\vDash_t\{P\}c\{Q\}$是证明可线性化的关键。该三元组考虑任意满足前条件的程序状态(Δ,s,ρ)，以及任何由指令c的语义定义的可到达的后置程序状态(Δ',s')。这里$[\![c]\!]_t(\Delta,s)$定义了从具体程序状态(Δ,s)出发执行c能得到的所有后置程序状态的集合，其中s'是通过在s末尾添加对应指令c的调用事件和返回事件得到的。该三元组要求证明者给出某个新的线性化历史记录ρ'来组成完整的后置程序状态，并证明后置程序状态(Δ',s',ρ')满足后条件Q，且这样一种从前置状态到后置状态的转移需记录在保证$\mathcal{G}$中以供其他线程的证明参考。最后，证明者需要证明该线性化历史记录ρ'是有原来的ρ增加若干返回事件后重写得到的，即$\rho\dashrightarrow\rho'\iff\exists t\in R^*.\rho\cdot t\rightsquigarrow\rho'$。由于$\rho$可以通过具体历史记录线性化得到的，经过重写得到的$\rho'$依然可以通过具体历史记录线性化得到。同时，该重写过程可能会通过增加返回事件，从而对某些正在执行的操作完成可线性化，对应了 abs-conseq 规则中的$\Rrightarrow_s$这样一个对规约代码的执行步骤。

案例 9：依然以 3.3 中图 4 的inc_i程序为例来展示该逻辑是如何证明可线性化的。这里再引入一个 read 函数，其定义为 read() { v = *p; return v; }。这里证明该并非对象满足如下性质定义的规约S。

$$\rho\in S\iff\begin{pmatrix}\rho=\epsilon\vee(\exists t',\rho'.\rho=\rho'\cdot t{:}\mathrm{inc}\cdot t{:}\mathrm{ok})\\ \vee(\exists t',\rho',n.\rho=\rho'\cdot t{:}\mathrm{read}\cdot t{:}n\wedge\mathrm{cnt}(\rho')=n)\end{pmatrix}$$

其中 cnt(ρ)为ρ代表的执行历史记录在当前时刻计数器的数值，其定义为当前所有有对应返回事件的 inc 事件。对于这样一个计数器对象，如定义其不变量为$I\triangleq\exists n.\,\mathrm{value}(s|_{\mathrm{cas}})=n\wedge\mathrm{cnt}(\rho)=n$。其含义是$\mathrm{p}_i$所指的值与当前计数器的值相等。其中 value($s|_{cas}$)是一个计算一个可以使用 CAS 指令的内存地址的值。这里p_i同样被当作一个并发对象，其有关的 CAS 事件和读取事件也同样被记录在具体历史记录中。也就是说，CAS 语句的

$$[\![b=\mathrm{CAS}(p,n,m)]\!]_t(\Delta,s)\triangleq\left\{(\Delta',s')\middle|\begin{pmatrix}\mathrm{value}(s|_{\mathrm{cas}})=n\wedge\Delta'=\Delta[b/\mathtt{true}]\\ \wedge s'=s\cdot t{:}\mathrm{CAS}(p,n,m)\cdot t{:}\mathtt{true}\end{pmatrix}\vee\begin{pmatrix}\mathrm{value}(s|_{\mathrm{cas}})\neq n\wedge\Delta'=\Delta[b/\mathtt{false}]\\ \wedge s'=s\cdot t{:}\mathrm{CAS}(p,n,m)\cdot t{:}\mathtt{false}\end{pmatrix}\right\}$$

[5] 视情况也可以定义为一般的实际程序状态[77]（即局部变量的部分映射和物理内存）。一个并发对象的实际执行历史记录也可以确定任意时刻的实际程序状态。

语义$[\![b = \mathrm{CAS}(p_i, n, m)]\!]$可以如下定义。其包含两种情况，如果当前$s$中计算得到$\mathrm{p}_i$所指的值为 n，则在 s 结尾加上 CAS 调用事件和结果为 true 的返回事件；否则，结尾加上 CAS 调用事件和结果为 false 的返回事件。

图 11 为计数器对象的 inc 函数的可线性化证明。其证明思路与上下文精化类似，即在循环不变量中区分 b 为真和假两种情况，前者完成可线性化，后者不对线性化结果ρ作任何操作。两种证明方式分别都在执行 CAS 指令的时候进行规约程序的执行或可线性化，不同之处在于如何进行这一操作。

该证明分情况讨论 CAS 的执行结果，即分别证明图 12 和图 13 中两个霍尔三元组后再通过析取规则得到图 11 中 CAS 语句的证明。图 12 前条件中p_i所指值不等于 m，CAS 返回值为 fasle，因此不对ρ进行修改。

$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{inc}\}$
b = false;
$$\left\{ \begin{array}{c} I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{inc} \wedge \\ \left(\begin{array}{c} ([\![b]\!] = \texttt{true} \wedge \mathrm{last}(\rho|_t) = \mathrm{ok}) \\ \vee([\![b]\!] = \texttt{false} \wedge \mathrm{last}(\rho|_t) = \mathrm{inc}) \end{array} \right) \end{array} \right\}$$
while(!b) {
$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{inc}\}$
v = *p_i;
$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{inc} \wedge \exists m.[\![v]\!] = m\}$
b = CAS(p_i, v, v+1);
$$\left\{ \begin{array}{c} I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{inc} \wedge \\ \left(\begin{array}{c} ([\![b]\!] = \texttt{true} \wedge \mathrm{last}(\rho|_t) = \mathrm{ok}) \\ \vee([\![b]\!] = \texttt{false} \wedge \mathrm{last}(\rho|_t) = \mathrm{inc}) \end{array} \right) \end{array} \right\}$$
}
$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{inc} \wedge \mathrm{last}(\rho|_t) = \mathrm{ok}\}$
return ok;
$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{ok}\}$

图 11：计数器对象的可线性化证明

$$\left\{ \begin{array}{c} I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{inc} \wedge \\ \exists m.[\![v]\!] = m \wedge \mathrm{value}(s|_{\mathrm{cas}}) \neq m \end{array} \right\}$$
b = CAS(p_i, v, v+1);
$$\left\{ I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{inc} \wedge [\![b]\!] = \texttt{false} \right\}$$

图 12：CAS 执行失败的证明

$$\left\{ \begin{array}{c} \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho'|_t) = \mathrm{inc} \wedge \\ \exists m.[\![v]\!] = m \wedge \mathrm{value}(s|_{\mathrm{cas}}) = \mathrm{cnt}(\rho') = m \end{array} \right\}$$
b = CAS(p_i, v, v+1);
$$\left\{ \begin{array}{c} \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho'|_t) = \mathrm{inc} \wedge \\ \exists m.[\![v]\!] = m \wedge \mathrm{value}(s|_{\mathrm{cas}}) = m+1 \wedge \mathrm{cnt}(\rho') = m \end{array} \right\}$$
$$\left\{ \begin{array}{c} \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho'|_t) = \mathrm{inc} \wedge \\ \exists m.[\![v]\!] = m \wedge \mathrm{value}(s|_{\mathrm{cas}}) = m+1 \wedge \\ \rho' \cdot t{:}\mathrm{ok} \rightsquigarrow \rho \wedge \mathrm{cnt}(\rho) = m+1 \end{array} \right\}$$
$$\left\{ \begin{array}{c} I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{inc} \wedge \\ \mathrm{last}(\rho|_t) = \mathrm{ok} \wedge [\![b]\!] = \texttt{true} \end{array} \right\}$$

图 13：CAS 执行成功的证明

图 13 中前条件p_i所指的值等于 m，此时返回值为 true 且p_i所指的值被修改为 m+1，不变量不再满足。此时可知该 inc 函数的效果已生效，可以通过在原来的线性化历史记录ρ'结尾加上该此调用的返回事件，并通过重写将新增的事件移至 inc 的调用事件后方，并将所有未返回的调用事件移至该组 inc 事件之后，因为它们必定将晚于该 inc 事件生效。这里蓝色的程序断言是不稳定的，需要将其弱化到一个稳定的后条件并与前种情况的后条件结合作为 CAS 指令完整的后条件。

该证明中线程t使用的保证$\mathcal{G}_t$如下定义。线程t所使用的依赖$\mathcal{R}_t$为其他所有线程的保证的并集。

$$((\Delta, s, \rho), (\Delta', s', \rho')) \in \mathcal{G}_t \iff \left(\begin{array}{c} (\mathrm{value}(s'|_{\mathrm{cas}}) = \mathrm{value}(s|_{\mathrm{cas}}) \wedge \mathrm{cnt}(\rho') = \mathrm{cnt}(\rho)) \vee \\ (\mathrm{value}(s'|_{\mathrm{cas}}) = \mathrm{value}(s|_{\mathrm{cas}}) + 1 \wedge \mathrm{cnt}(\rho') = \mathrm{cnt}(\rho) + 1) \end{array} \right)$$

$$\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = \mathrm{op}\}\mathrm{op}()\{I \wedge \mathrm{last}(s|_{\mathrm{cnt},t}) = \mathrm{last}(\rho|_t) = r\}$$

对于这样一个并发对象，需要证明其所有操作满足图 11 中的霍尔三元组，也就是如下霍尔三元组。其前条件表达的是同时在具体历史记录里和线性化结果中加上该操作的调用事件。后条件表达的是具体历史记录中和线性化结果中，当前操作都已经返回，且返回值是相同的。这样就保证了证明者对ρ进行的线性化操作与实际程序的运行是一致的，从而保证实际的执行是可线性化的。

4.5 小结

本节分析了几种用于验证并发程序的功能正确性的程序逻辑。在霍尔逻辑的基础上，可以分别使用并发分离逻辑或者基于依赖保证的逻辑或者二者的结合对并发程序证明其霍尔三元组。其证明得到的结果有着较强的并发上下文约束。进一步的，可以通过关系霍尔逻辑将上下文精化关系和可线性化性质以霍尔三元组的形式进行表达，并通过关系霍尔逻辑的可靠性由霍尔三元组推导出证明对象的这两种功能正确性。

表 1 列出了部分现有的基于程序逻辑的对并发程序的验证工作，并对其验证目标和验证方法进行总结。表中使用“功能正确性”表示“有并发上下文的功能正确性”。部分工作包含较有意义的理论工作，但并未在交互式定理证明器中实现，在表中标记为未经机器验证。部分工作通过证明逻辑原子性来证明可线性化，并未直接证明可线性化，将其验证目标归类为逻辑原子性。部分上下文精化验证工作支持终止保持性质的证明，或者直接支持证明进展性质，在验证目标一栏中标出。

表 1：基于程序逻辑对并发程序的功能正确性验证工作

文献	验证目标	验证方法	是否进行机器验证
RGSep[79]	功能正确性+逻辑原子性	并发分离逻辑+依赖保证	是
Deny Guarantee[68]	功能正确性	并发分离逻辑+依赖保证	否
RGSim[54]	上下文精化	依赖保证+关系霍尔逻辑	是
RGSimT[33]	上下文精化+终止保持	依赖保证+关系霍尔逻辑	否
LRG[67]	功能正确性	并发分离逻辑+依赖保证	否
LRG with nonfixed LP[36]	上下文精化	并发分离逻辑+依赖保证+关系霍尔逻辑	否
μC[52]	上下文精化	并发分离逻辑+关系霍尔逻辑	是
CAP[24], iCAP[63]	功能正确性	并发分离逻辑	否
FCSL[80]	功能正确性	并发分离逻辑	是
TaDA[30]	功能正确性+逻辑原子性	并发分离逻辑	否
CaReSL[81][74]	上下文精化	并发分离逻辑+关系霍尔逻辑	是
Iris[26][27]	功能正确性	并发分离逻辑	是
ReLoC[72][73][81]	上下文精化+逻辑原子性	并发分离逻辑+关系霍尔逻辑	是
Simuliris[53]	上下文精化+终止保持	并发分离逻辑+关系霍尔逻辑	是
VST[31][94]	功能正确性	并发分离逻辑	是
CSimpl[82]	上下文精化	依赖保证	是
LiLi[3]	上下文精化+进展性质	依赖保证+关系霍尔逻辑	否
A Generic Logic for Proving Linearizability[76][77]	可线性化	并发分离逻辑+依赖保证+关系霍尔逻辑	否
Compositional Linearizability[44]	可线性化	依赖保证+关系霍尔逻辑	否

5 基于程序逻辑的并发程序验证工具

基于并发分离逻辑和依赖保证逻辑，研究者们开发出了许多支持并发程序验证的验证工具。这些工具在交互式定理证明器中得到实现，保证其自身的证明是可靠的。本节将对这些工具进行简单介绍。

（1） Iris

Iris[26][27]是一个高阶并发分离逻辑框架，其在 Coq 中实现形式化，用以证明带并发上下文的功能正确性。与一般的并发分离逻辑不同，其分离逻辑程序断言允许一定程度上的不变量自指，同时其霍尔三元组以最弱前条件的形式定义且其最弱前条件本身就定义为一种程序断言。这种形式化方式也成为非直谓程序断言[83]

（Impredicative Assertion），这种允许自指的定义使得对于高阶函数的证明非常方便，但同样也给其形式化带来困难。Iris 使用 Step-Index 定义其非直谓程序断言。Iris 中分离逻辑的程序状态允许证明者进行自定义，可以设定程序证明中用到的辅助程序状态的类型以及允许的程序状态转移关系，从而通过视角变换（View Shift）实现辅助程序状态更新。这使得其逻辑框架在证明并发程序的时候具有灵活性，不需要完全改变框架就能支持不同并发程序的证明。Iris 不仅在 Coq 中实现了其逻辑的可靠性，还在 Coq 中建立了其专用的证明工具 Iris-Proof-Mode（IPM）。IPM 提供了证明者一套非常实用的证明指令和一定的证明自动化来简化证明过程，同时改进了 Coq 的证明视窗（Proof view）对 Iris 中分离逻辑的证明进行适配，使证明者可以非常清楚地分辨和使用证明中出现的关于内存的分离逻辑断言、可重复使用的持久性（Persistent）的断言和与程序状态无关的一般前提。这一系列工作使得 Iris 有着极强的可用性。

为了支持并发程序的上下文精化，研究者们在 Iris 的基础上建立了 ReLoC[72][73][81]。ReLoC 是一个基于并发分离逻辑的关系霍尔逻辑框架。它将抽象程序状态和规约程序定义为 Iris 中的辅助程序状态，从而可以使用 Iris 的程序断言直接描述它们，证明者可以使用 Iris 中的视角变换对具体程序和规约程序进行异步的符号执行。ReLoC 的可靠性保证了其关系霍尔逻辑三元组可以推出相应的上下文精化关系。

一系列的工具在 Iris 和 ReLoC 的基础上被开发出来对现实中的程序和系统进行验证。

RustBelt[84]使用 Iris 逻辑框架实现了对于 Rust[85]程序的证明工具，证明目标为带并发上下文的功能正确性。其核心是支持 Rust 的基于权限的类型系统的验证，且可以支持 Rust 中并发程序的证明。后续工作分别尝试使 RustBelt 对弱内存模型上的 Rust 程序进行验证[32]，和将 RustBelt 与 RustHorn[86]结合从而更好地对 Rust 程序的功能正确性进行验证[87]。

Perennial[88]是基于 Iris 的对于 Go 语言[89]程序的验证工具。其通过证明一个系统中所有的函数都是其原子规约的上下文精化，即具有逻辑原子性，对该系统的正确性进行证明。同时 Perennial 支持故障安全性的验证，即恢复程序可以正确地将系统从故障中回复到正常状态。为了支持故障安全性的证明，他们的程序逻辑也结合故障霍尔逻辑[90]（Crash Hoare Logic）对并发程序的故障与恢复进行验证。研究者使用 Perennial 证明了日志系统 GoJournal 是正确的和故障安全的[91]。

RefinedC[92]是基于 Iris 的对于 C 程序的验证工具，证明目标为带并发上下文的功能正确性。其通过基于权限的类型对于 C 中的并发程序进行验证，并提供了许多自动化证明手段。

（2） VST

Verified Software Toolchain[31]（VST）是在 Coq 中实现的基于分离逻辑的 C 语言程序的验证工具。VST 可以验证 C 程序的安全性与带并发上下文的功能正确性，并且支持依赖锁的不变量的并发程序证明。其验证结果可以直接与 CompCert[93]的 C 语言编译器正确性结合，得到从源程序的证明到汇编代码的端到端的正确性证明。Mansky[94]进一步将 Iris 中丰富的辅助程序状态和不变量的设计加入到了 VST 中，并支持类似 IPM 的证明方式。其工作使得 VST 可以证明无锁的更细粒度的 C 语言并发程序的功能正确性。

（3） CSimpl

CSimpl[82]是基于依赖保证的程序逻辑框架，可以验证 CSimpl 语言编写的并发程序的带并发上下文的功能正确性。其程序语言前身为 seL4 操作系统验证[95]中的 Simpl[95]。CSimpl 语言支持大部分现实中程序语言的特性，且存在将 C 程序翻译到 Simpl 和 CSimpl 程序的工具（如 CParser[96]）。 CSimpl 在 Isabelle/HOL[14]中得到形式化。Sim2[97]的工作通过证明 CSimpl 语言下的具体程序与规约程序间的模拟关系，使对于规约程序证明得的霍尔三元组同样对于具体程序成立，以此实现证明的可组合性。其模拟关系的证明并非通过程序逻辑而是直接构造得到的。基于 Sim2 框架，研究者们使用 CSimpl 语言为一个 Arinc-653 标准下的通信服务定义了规约代码以及证明了该规约代码的不变量，并通过模拟关系证明了该服务的具体实现也满足该不变量。

PiCore[98]是一个基于依赖保证的并发响应式系统的验证框架。该框架在 Isabelle/HOL 中形式化，并将 IMP[99]和 CSimpl[82]两个依赖保证逻辑在其框架下实例化，并用于验证现实操作系统 Zephyr RTOS 中的内存管

理系统且发现了 Zephyr RTOS 的 C 代码中的三个问题[100]。

（4） μC

μC[52]是一个在 Coq 中形式化的，基于并发分离逻辑的操作系统验证框架，其证明目标为上下文精化。其为了支持操作系统中多层级的中断处理，使用了多层级的并发分离逻辑不变量设计，使得其程序逻辑能更好地描述和验证内核的行为。其程序逻辑通过证明底层具体代码和高层规约代码之间的上下文精化关系来保证操作系统中所有模块都具有功能正确性。该框架还拓展了 C 语言机器模型，将汇编代码封装为特殊的 C 语言指令，克服了操作系统使用 C 语言内嵌汇编实现时所面临的挑战。该工作使用其验证框架验证了基于优先级调度的抢占式的实时操作系统内核 μC/OS-II[101]的正确性。

6 基于程序逻辑的并发程序验证成果

6.1 经验证的并发系统

Perennial 团队使用其 Perennial 验证框架验证了一个并发的邮件服务器 Mailboat[88]和一个日志系统 GoJournal[91]。两者均为使用 Go 语言编写的并发系统，且为了应对系统的故障，其均设计了恢复程序，可以将系统从故障中恢复到故障前的状态。Perennial 框架是 Iris 框架与故障霍尔逻辑[90]的结合，可以很好地证明在存在故障的环境中一个系统的逻辑原子性。同时其使用的逻辑原子性规约也非常适合将 GoJournal 这样一个大型的系统拆解成多个内部模块，并对每个模块的接口的逻辑原子性进行分别验证。

Feng 等人[100]使用基于依赖保证的 PiCore 框架[98]对 Zephyr RTOS 中的伙伴内存管理模块进行验证。Zephyr RTOS 中的伙伴内存分配将较大的块划分为较小的块，允许高效地分配和释放不同大小的块，同时支持较小块的立即自动组合。并且该内存分配的执行是抢占式的，这意味着当没有可用于内存请求的块时，内存分配可能会重新调度。该工作对代码中出现的所有内存初始化、分配和释放语句给出了细粒度的形式化规约。为了处理抢占，该工作将内存分配分解为一组事件，并通过状态机归纳定义了 Zephyr 的抢占执行。最终，该工作使用不变量约束内存数据结构的一致性和表达内存分配的功能正确性。

Zou 等人[102]使用基于依赖保证的并发关系逻辑对细粒度并发文件系统 AtomFS 进行了形式化验证。其工作通过对该文件系统的逻辑原子性进行验证，从而证明系统的可线性化性质。该文件系统的一个验证难点是，部分操作的可线性化点是在其他并发的操作中进行的，即不定的可线性化点[36]。该工作在其程序逻辑中引入了线程间帮助机制，通过辅助程序状态将一个线程中需要借助外部线性化点的操作进行记录，从而证明在其他线程中的某个操作可以帮助当前线程完成线性化点的执行。

Xu 等人[52]使用并发关系逻辑框架μC 验证了基于优先级调度的抢占式的实时操作系统内核 μC/OS-II[101]的功能正确性。该内核提供包括任务管理、信箱、内存管理、互斥锁、消息队列、信号量管理等基本的系统服务。该操作系统使用了多层级的中断，且 C 语言代码中内嵌了汇编代码。这些都给操作系统验证带来了困难，但是该工作很好地支持了这些特性的证明。同时，该工作还证明了操作系统中的调度程序不会出现优先级反转[103]（Priority Inversion），确保高优先级任务不会被低优先级任务无限期地阻塞，保证了系统的公平与高效。

6.2 经验证的并发算法与并发对象

许多高效的并发算法与并发对象也在许多工作中得到了验证。不同于传统的基于锁实现同步的算法，无锁的细粒度的并发算法一般具有更高的效率，同时也有着更高的验证难度。这些细粒度的并发算法与对象是众多并发验证工作挑战的对象，在试图验证它们的工作中，许多有效的验证机制和手段得以面世。本小节对常在理论工作中出现的作为样例的算法和对象以及其验证工作进行总结，以供未来的有关理论工作进行参考。

大部分无锁的细粒度并发程序通过开放式并发[104]（Optimistic Concurrency）实现线程间的有序合作。在这种并发编程方式下，一般会通过一个 CAS 指令重复执行当前函数需要执行的可线性化点上的指令，如果失败，即存在其他线程并发地进行同一处内存的修改且修改成功，那么就重新执行该 CAS 指令。这种方式相较

基于临界区的同步，使得所有线程都可以有所进展，从而降低了临界区中无关指令和调度带来的时间开销

队列（Queue）是一种线性的数据存储结构，其提供入队和出队两种操作，并且要求入队和出队顺序满足先进先出（FIFO）原则。在并发的情况下，通过无锁的方式实现 FIFO 要求具有很大的挑战。MS 队列[105]是典型的使用开放式并发的算法。其通过头尾两个指针控制队列的头尾，并且在入队和出队时分别通过一个循环反复尝试对尾部或头部指针使用 CAS 指令操作队列元素。该算法特殊之处在于一部分队列元素如果已经出队但内存尚未被回收，那么一个滞后的入队操作仍然会通过访问这部分元素找到真正的队列尾部。Liang 等人[36]使用了基于依赖保证的逻辑证明了 MS 队列的逻辑原子性。

并发队列可以使用时间戳（Time-stamp）控制出入队顺序。比如 Herlihy-Wing 队列[35]通过一个原子的计数器（如 FAI 指令）对每个入队元素分配时间戳，每次仅允许时间戳最小的元素出队，且出队时通过开放式并发尝试对该元素进行操作。其特点是允许两个并发的入队元素有相同的时间戳，两者的出栈顺序是任意的，但是出栈时的顺序决定了该次执行的线性化结果中两者的先后顺序。在 Khyzha 等人的偏序线性化理论的工作中[76]，他们使用了依赖保证逻辑证明了时间戳队列的可线性化性质。Vindum 等人[39]使用了 ReLoC 框架证明了一个更细粒度的队列是粗粒度队列的上下文精化。该队列在 Facebook 的 Folly 库[106]中使用。

栈（Stack）同样是一种线性的数据存储结构，其提供入栈和出栈两种操作，并且要求入栈和出栈顺序满足先进后出（FILO）原则。Treiber 栈[107]同样是一个非常典型的开放式并发算法。其入栈与出栈操作会反复尝试使用 CAS 指令对栈顶指针进行修改，直到当前操作成功修改了栈顶指针。Turon 等人[74]使用 CaReSL 框架对该算法进行了验证。Treiber 栈的一个衍生算法是后退消除（Backoff-Elimination）栈[38]。当一个入栈或出栈操作尝试 CAS 栈顶指针失败后，这意味着很可能存在其他线程在与其竞争，如果贸然再次进行 CAS 操作可能还会与其他线程发生冲突导致 CAS 失败。一个做法是通过 Backoff，即等待一段时间再进行执行从而试图避免再次冲突。而后退消除栈的做法就是在这段等待的时间内尝试与其他等待线程进行交换，如果一组入栈与出栈操作匹配了，则不经过底层数据结构直接完成两个操作。这种方式也被称为线程间帮助机制。Hemed 等人[41]使用集合可线性化描述这个交换操作所依赖的交换对象（Exchanger），并证明了以此实现的栈是 Herlihy-Wing 可线性化的。Frumin 等人[108]使用 ReLoC 证明了一个简化的后退消除栈。

基于时间戳的算法同样适用于并发栈。Dodds[37]结合了时间戳与后退消除技术，实现了一个并发栈，其使用时间戳决定不同元素的出栈顺序。与 Herlihy-Wing 队列类似，其允许两个并发的入栈元素有相同的时间戳，并由实际出栈顺序决定线性化结果，这是其证明的一大难点。目前尚无工作使用程序逻辑证明其功能正确性。一个可能的方向是通过基于偏序的可线性化证明方式对其进行验证[76]。

7 总结与未来展望

本文对交互式定理证明工作中常用于描述并发程序正确性的证明目标进行了归纳，它们主要包括带并发上下文的功能正确性、可线性化、上下文精化以及逻辑原子性。其中带并发上下文的功能正确性需要在较强的并发上下文约束下才能精确描述一个程序的功能正确性，而其他三种证明目标可以在较一般的并发上下文约束下精确描述程序的功能正确性。可线性化与上下文精化间有着较强的联系，同时逻辑原子性是可线性化的一种特例且可以通过上下文精化进行表达。并发分离逻辑和基于依赖保证的逻辑可以验证并发程序的霍尔三元组，将其与关系霍尔逻辑相结合，可以进一步对可线性化和上下文精化进行验证。现实中，有许多证明工具以及并发系统和并发算法的证明工作使用了这些证明方法。

目前的验证工作中，有不少在考虑并发的同时考虑了系统的故障正确性。现实中，即使软件本身的安全性得到了验证，硬件层面的故障仍然是无法避免的。因此软件的实现能否使系统从故障中恢复到正常状态也是一个非常重要目标。对于并发程序的故障正确性的验证工作并不像并发程序正确性一样充分，仍然有很多潜在的研究方向，例如如何对并发程序的持久可线性化（Durable Linearizability）进行模块化的验证等。

同时，并发程序验证中的大多数研究工作仅考虑了满足顺序一致性的内存模型。但现实中很多体系结构使用的是 TSO、C11 等弱内存模型。在弱内存模型中，程序中有着先后顺序的两个指令在内存中的实际生效

时间可以不保持原本的顺序，而大部分程序逻辑都依赖于顺序执行规则对程序按代码顺序进行验证，所以无法通过本文所涵盖的程序逻辑直接对这类程序进行验证。如何使用程序逻辑对弱内存模型上的并发程序进行验证是一个仍待解决的问题。

许多验证工作都实现了程序和证明的模块化，但是在证明完成后往往需要将不同程序模块（库与库、库与用户程序）相结合得到完整的程序。不同模块可能会使用不同的语言，甚至是不同的证明目标。如何将不同语言下的模块重新组合，并将各自可能不同的正确性证明组合得到完整程序的正确性是非常重要的问题。一些实际使用的并发代码库还需要考虑底层体系结构带来的弱内存一致性，使得组合验证问题更加复杂。

除此之外，交互式定理证明一个较大的问题是如何使证明更加的自动化。对一个现实中使用高级语言的程序进行验证，一行代码往往需要数倍的验证指令才能证明。这对交互式定理证明方法的使用有非常大的阻碍，是需要形式化理论验证者们解决的一大关键问题。

References:

[1] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs[J]. IEEE transactions on computers, 1979, 100(9): 690-691.

[2] Bernstein P A, Hadzilacos V, Goodman N. Concurrency control and recovery in database systems[M]. Reading: Addison-wesley, 1987.

[3] Liang H, Feng X. Progress of Concurrent Objects[J]. Foundations and Trends® in Programming Languages, 2020, 5(4): 282-414.

[4] Herlihy M, Shavit N. On the nature of progress[C]//International Conference On Principles Of Distributed Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 313-328.

[5] Biere A, Cimatti A, Clarke E M, et al. Bounded model checking[J]. Handbook of satisfiability, 2009, 185(99): 457-481.

[6] Inverso O, Tomasco E, Fischer B, et al. Bounded model checking of multi-threaded C programs via lazy sequentialization[C]//Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. Springer International Publishing, 2014: 585-602.

[7] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software[J]. ACM Sigplan Notices, 2005, 40(1): 110-121.

[8] Alglave J, Kroening D, Tautschnig M. Partial orders for efficient bounded model checking of concurrent software[C]//Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25. Springer Berlin Heidelberg, 2013: 141-157.

[9] Godefroid P. Model checking for programming languages using VeriSoft[C]//Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1997: 174-186.

[10] Abdulla P A, Aronis S, Atig M F, et al. Stateless model checking for TSO and PSO[J]. Acta Informatica, 2017, 54: 789-818.

[11] Huang J. Stateless model checking concurrent programs with maximal causality reduction[J]. ACM SIGPLAN Notices, 2015, 50(6): 165-174.

[12] Kokologiannakis M, Lahav O, Sagonas K, et al. Effective stateless model checking for C/C++ concurrency[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-32.

[13] The Coq Proof Assistant. https://coq.inria.fr/

[14] Isabelle/HOL: a proof assistant for higher-order logic[M]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[15] Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[16] Stark E W. A proof technique for rely/guarantee properties[C]//Foundations of Software Technology and Theoretical Computer Science: Fifth Conference, New Delhi, India December 16–18, 1985 Proceedings 5. Springer Berlin Heidelberg, 1985: 369-391.

[17] Jones C B. Specification and design of (parallel) programs[C]//9th IFIP World Computer Congress (Information Processing 83). Newcastle University, 1983.

[18] O'hearn P W. Resources, concurrency, and local reasoning[J]. Theoretical computer science, 2007, 375(1): 271-307.

[19] Tait W W. Intensional interpretations of functionals of finite type I[J]. The journal of symbolic logic, 1967, 32(2): 198-212.

[20] Windsor M, Dodds M, Simner B, et al. Starling: Lightweight concurrency verification with views[C]//Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30. Springer International Publishing, 2017: 544-569.

[21] Dinsdale-Young T, da Rocha Pinto P, Andersen K J, et al. Caper: automatic verification for fine-grained concurrency[C]//Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26. Springer Berlin Heidelberg, 2017: 420-447.

[22] De Moura L, Bjørner N. Z3: An efficient SMT solver[C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008: 337-340.

[23] Dinsdale-Young T, Birkedal L, Gardner P, et al. Views: compositional reasoning for concurrent programs[C]//Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. 2013: 287-300.

[24] Dinsdale-Young T, Dodds M, Gardner P, et al. Concurrent abstract predicates[C]//ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24. Springer Berlin Heidelberg, 2010: 504-528.

[25] Mulder I, Krebbers R, Geuvers H. Diaframe: automated verification of fine-grained concurrent programs in Iris[C]//Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2022: 809-824.

[26] Jung R, Swasey D, Sieczkowski F, et al. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning[J]. ACM SIGPLAN Notices, 2015, 50(1): 637-650.

[27] Jung R, Krebbers R, Jourdan J H, et al. Iris from the ground up: A modular foundation for higher-order concurrent separation logic[J]. Journal of Functional Programming, 2018, 28: e20.

[28] Wolf F A, Schwerhoff M, Müller P. Concise outlines for a complex logic: a proof outline checker for TaDA[J]. Formal Methods in System Design, 2023: 1-27.

[29] Müller P, Schwerhoff M, Summers A J. Viper: A verification infrastructure for permission-based reasoning[C]//Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17. Springer Berlin Heidelberg, 2016: 41-62.

[30] da Rocha Pinto P, Dinsdale-Young T, Gardner P. TaDA: A logic for time and data abstraction[C]//ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28. Springer Berlin Heidelberg, 2014: 207-231.

[31] Appel A W. Verified Software Toolchain: (Invited Talk)[C]//European Symposium on Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 1-17.

[32] Dang H H, Jourdan J H, Kaiser J O, et al. RustBelt meets relaxed memory[J]. Proceedings of the ACM on Programming Languages, 2019, 4(POPL): 1-29.

[33] Liang H, Feng X, Shao Z. Compositional verification of termination-preserving refinement of concurrent programs[C]//Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). 2014: 1-10.

[34] Lamport L. On Interprocess Communication: Part II: Algorithms[J]. Distributed Computing, 1986, 1: 86-101.

[35] Herlihy M P, Wing J M. Linearizability: A correctness condition for concurrent objects[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1990, 12(3): 463-492.

[36] Liang H, Feng X. Modular verification of linearizability with non-fixed linearization points[C]//Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2013: 459-470.

[37] Dodds M, Haas A, Kirsch C M. A scalable, correct time-stamped stack[J]. ACM SIGPLAN Notices, 2015, 50(1): 233-246.

[38] Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm[C]//Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. 2004: 206-215.

[39] Vindum S F, Frumin D, Birkedal L. Mechanized verification of a fine-grained concurrent queue from meta's folly library[C]//Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. 2022: 100-115.

[40] Neiger G. Set-linearizability[C]//Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing. 1994: 396.

[41] Hemed N, Rinetzky N, Vafeiadis V. Modular verification of concurrency-aware linearizability[C]//Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings 29. Springer Berlin Heidelberg, 2015: 371-387.

[42] Castañeda A, Rajsbaum S, Raynal M. Specifying concurrent problems: beyond linearizability and up to tasks[C]//Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings 29. Springer Berlin Heidelberg, 2015: 420-435.

[43] Castañeda A, Rajsbaum S, Raynal M. Unifying concurrent objects and distributed tasks: Interval-linearizability[J]. Journal of the ACM (JACM), 2018, 65(6): 1-42.

[44] Oliveira Vale A, Shao Z, Chen Y. A Compositional Theory of Linearizability[J]. Proceedings of the ACM on Programming Languages, 2023, 7(POPL): 1089-1120.

[45] Lafont Y, Streicher T. Games semantics for linear logic[C]//Proceedings 1991 Sixth Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society, 1991: 43, 44, 45, 46, 47, 48, 49, 50-43, 44, 45, 46, 47, 48, 49, 50.

[46] Hyland J M E, Ong C H L. On full abstraction for PCF: I, II, and III[J]. Information and computation, 2000, 163(2): 285-408.

[47] Goubault É, Ledent J, Mimram S. Concurrent specifications beyond linearizability[C]//22nd International Conference on Principles of Distributed Systems (OPODIS 2018). 2018.

[48] Hoare C A R, Jifeng H, Sanders J W. Prespecification in data refinement[J]. Information Processing Letters, 1987, 25(2): 71-76.

[49] He J, Hoare C A R, Sanders J W. Data refinement refined resume[C]//ESOP 86: European Symposium on Programming Saarbrücken, Federal Republic of Germany March 17–19, 1986 Proceedings 1. Springer Berlin Heidelberg, 1986: 187-196.

[50] Filipović I, O'Hearn P, Rinetzky N, et al. Abstraction for concurrent objects[J]. Theoretical Computer Science, 2010, 411(51-52): 4379-4398.

[51] Plotkin G. Lambda-definability and logical relations[M]. Edinburgh University, 1973.

[52] Xu F, Fu M, Feng X, et al. A practical verification framework for preemptive OS kernels[C]//International Conference on Computer Aided Verification. Cham: Springer International Publishing, 2016: 59-79.

[53] Gäher L, Sammler M, Spies S, et al. Simuliris: a separation logic framework for verifying concurrent program optimizations[J]. Proceedings of the ACM on Programming Languages, 2022, 6(POPL): 1-31.

[54] Liang H, Feng X, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations[C]//Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2012: 455-468.

[55] Herlihy M, Shavit N, Luchangco V, et al. The art of multiprocessor programming[M]. Newnes, 2020: 54-55.

[56] Svendsen K, Birkedal L, Parkinson M. Modular reasoning about separation of concurrent data structures[C]//Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. Springer Berlin Heidelberg, 2013: 169-188.

[57] O'Hearn P W, Yang H, Reynolds J C. Separation and information hiding[C]//Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2004: 268-280.

[58] Cook S A. Soundness and completeness of an axiom system for program verification[J]. SIAM Journal on Computing, 1978, 7(1): 70-90.

[59] O'Hearn P, Reynolds J, Yang H. Local reasoning about programs that alter data structures[C]//Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15. Springer Berlin Heidelberg, 2001: 1-19.

[60] Reynolds J C. Separation logic: A logic for shared mutable data structures[C]//Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2002: 55-74.

[61] Brookes S. A semantics for concurrent separation logic[J]. Theoretical Computer Science, 2007, 375(1-3): 227-270.

[62] Owicki S, Gries D. Verifying properties of parallel programs: An axiomatic approach[J]. Communications of the ACM, 1976, 19(5): 279-285.

[63] Svendsen K, Birkedal L. Impredicative concurrent abstract predicates[C]//Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23. Springer Berlin Heidelberg, 2014: 149-168.

[64] Alfieri R A. An Efficient Kernel-Based Implementation of POSIX Threads[C]//USENIX Summer. 1994: 59-72.

[65] The OpenMP API specifcation for parallel programming.: https:// www.openmp.org/ (2020).

[66] Gotsman A, Berdine J, Cook B, et al. Local reasoning for storable locks and threads[C]//Programming Languages and Systems: 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007. Proceedings 5. Springer Berlin Heidelberg, 2007: 19-37.

[67] Feng X. Local rely-guarantee reasoning[C]//Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2009: 315-327.

[68] Dodds M, Feng X, Parkinson M, et al. Deny-guarantee reasoning[C]//Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 18. Springer Berlin Heidelberg, 2009: 363-377.

[69] Dreyer D, Neis G, Rossberg A, et al. A relational modal logic for higher-order stateful ADTs[C]//Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2010: 185-198.

[70] Ahmed A, Dreyer D, Rossberg A. State-dependent representation independence[J]. ACM SIGPLAN Notices, 2009, 44(1): 340-353.

[71] Pitts A M, Stark I D B. Operational reasoning for functions with local state[J]. Higher order operational techniques in semantics, 1998: 227-273.

[72] Frumin D, Krebbers R, Birkedal L. ReLoC: A mechanised relational logic for fine-grained concurrency[C]//Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. 2018: 442-451.

[73] Frumin D, Krebbers R, Birkedal L. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity[J]. Logical Methods in Computer Science, 2021, 17.

[74] Turon A, Dreyer D, Birkedal L. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency[C]//Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. 2013: 377-390.

[75] Mulder I, Krebbers R. Proof automation for linearizability in separation logic[J]. Proceedings of the ACM on Programming Languages, 2023, 7(OOPSLA1): 462-491.

[76] Khyzha A, Dodds M, Gotsman A, et al. Proving linearizability using partial orders[C]//Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26. Springer Berlin Heidelberg, 2017: 639-667.

[77] Khyzha A, Gotsman A, Parkinson M. A generic logic for proving linearizability[C]//FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21. Springer International Publishing, 2016: 426-443.

[78] Jung R, Lepigre R, Parthasarathy G, et al. The future is ours: prophecy variables in separation logic[J]. Proceedings of the ACM on Programming Languages, 2019, 4(POPL): 1-32.

[79] Vafeiadis V, Parkinson M. A marriage of rely/guarantee and separation logic[C]//CONCUR 2007–Concurrency Theory: 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007. Proceedings 18. Springer Berlin Heidelberg, 2007: 256-271.

[80] Sergey I, Nanevski A, Banerjee A. Mechanized verification of fine-grained concurrent programs[C]//Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2015: 77-87.

[81] Turon A J, Thamsborg J, Ahmed A, et al. Logical relations for fine-grained concurrency[J]. Acm Sigplan Notices, 2013, 48(1): 343-356.

[82] Sanán D, Zhao Y, Hou Z, et al. Csimpl: A rely-guarantee-based framework for verifying concurrent programs[C]//Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23. Springer Berlin Heidelberg, 2017: 481-498.

[83] Ni Z, Shao Z. Certified assembly programming with embedded code pointers[C]//Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2006: 320-333.

[84] Jung R, Jourdan J H, Krebbers R, et al. RustBelt: Securing the foundations of the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-34.

[85] Matsakis N D, Klock F S. The rust language[J]. ACM SIGAda Ada Letters, 2014, 34(3): 103-104.

[86] Matsushita Y, Tsukada T, Kobayashi N. RustHorn: CHC-based verification for Rust programs[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2021, 43(4): 1-54.

[87] Matsushita Y, Denis X, Jourdan J H, et al. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code[C]//Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2022: 841-856.

[88] Chajed T, Tassarotti J, Kaashoek M F, et al. Verifying concurrent, crash-safe systems with Perennial[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 243-258.

[89] Donovan A A A, Kernighan B W. The Go programming language[M]. Addison-Wesley Professional, 2015.

[90] Chen H, Ziegler D, Chajed T, et al. Using Crash Hoare logic for certifying the FSCQ file system[C]//Proceedings of the 25th Symposium on Operating Systems Principles. 2015: 18-37.

[91] Chajed T, Tassarotti J, Theng M, et al. GoJournal: a verified, concurrent, crash-safe journaling system[C]//15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 2021: 423-439.

[92] Sammler M, Lepigre R, Krebbers R, et al. RefinedC: Automating the foundational verification of C code with refined ownership types[C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 158-174.

[93] Leroy X, Blazy S, Kästner D, et al. CompCert-a formally verified optimizing compiler[C]//ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. 2016.

[94] Mansky W. Bringing Iris into the Verified Software Toolchain[J]. arXiv preprint arXiv:2207.06574, 2022.

[95] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 207-220.

[96] Tuch H, Klein G, Norrish M. Types, bytes, and separation logic[C]//Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2007: 97-108.

[97] Sanan D, Zhao Y, Lin S W, et al. CSim 2: Compositional Top-down Verification of Concurrent Systems Using Rely-Guarantee[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2021, 43(1): 1-46.

[98] Zhao Y, Sanán D, Zhang F, et al. A parametric rely-guarantee reasoning framework for concurrent reactive systems[C]//International Symposium on Formal Methods. Cham: Springer International Publishing, 2019: 161-178.

[99] Nieto L P. The rely-guarantee method in Isabelle/HOL[C]//European Symposium on Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003: 348-362.

[100] Zhao Y, Sanán D. Rely-guarantee reasoning about concurrent memory management in zephyr RTOS[C]//Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31. Springer International Publishing, 2019: 515-533.

[101] The real-time kernel: µC/OS-II. http://micrium.com/rtos/ucosii/overview

[102] Zou M, Ding H, Du D, et al. Using concurrent relational logic with helpers for verifying the AtomFS file system[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 259-274.

[103] Babaoğlu Ö, Marzullo K, Schneider F B. A formalization of priority inversion[J]. Real-Time Systems, 1993, 5(4): 285-303.

[104] Turon A. Understanding and expressing scalable concurrency[D]. Northeastern University, 2013.

[105] Michael M M, Scott M L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms[C]//Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. 1996: 267-275.

[106] Folly: Facebook Open-source Library. https://github.com/facebook/folly

[107] Treiber R K. Systems programming: Coping with parallelism[M]. New York: International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

FRUMIN D A N, KREBBERS R, BIRKEDAL L. RELOC RELOADED: TECHNICAL APPENDIX[J]. 2020.