软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):2980−2994 [doi: 10.13328/j.cnki.jos.006609]

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

基于锁耦合遍历算法的文件系统终止性验证*

邹 沫, 谢昊彤, 魏卓然, 陈海波

(上海交通大学 软件学院, 上海 200240)

通信作者: 陈海波, E-mail: haibochen@sjtu.edu.cn

摘 要: 并发文件系统由于复杂的实现, 容易产生死锁、无限循环等终止性漏洞, 已有的文件系统证明工作都忽视了终止性的证明. 证明了一个并发文件系统 AtomFS 的终止性, 保证了每个文件系统接口在公平调度的条件下都能返回. 证明 AtomFS 接口的终止性要说明当其遇到阻碍时, 阻碍源头(其他线程)终将产生进展, 促使当前线程阻碍的消除, 证明的核心在于说明锁耦合(lock coupling)遍历算法的终止性. 然而还存在着两点挑战: (1) 文件系统的树形结构允许阻碍的线程分布在多条路径上, 全局地识别出多个阻碍源头使证明失去了局部性; (2) rename 接口由于需要遍历两条路径, 会带来"跨路径阻碍"现象, 多个 rename 可能相互跨路径阻碍成环, 导致无法找到阻碍源头. 提出了两个核心的技术点来应对这些挑战: (1) 使用局部思想仅确定单个阻碍源头; (2) 使用偏序法解决跨路径阻碍成环问题. 成功地构建了一个终止性证明框架 CRL-T, 并验证了 AtomFS 的终止性, 所有的证明都在 Coq 中实现.

关键词: 并发文件系统; 终止性; 形式化验证; Coq
中图法分类号: TP311

中文引用格式: 邹沫, 谢昊彤, 魏卓然, 陈海波. 基于锁耦合遍历算法的文件系统终止性验证. 软件学报, 2022, 33(8): 2980–2994. http://www.jos.org.cn/1000-9825/6609.htm

英文引用格式: Zou M, Xie HT, Wei ZR, Chen HB. Verification of Termination for File System Based on Lock Coupling Traversal. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 2980–2994 (in Chinese). http://www.jos.org.cn/1000-9825/6609.htm

Verification of Termination for File System Based on Lock Coupling Traversal

ZOU Mo, XIE Hao-Tong, WEI Zhuo-Ran, CHEN Hai-Bo

(School of Software, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: Termination bugs such as deadlocks and infinite loops are common in concurrent file systems due to complex implementations. Existing efforts on file system verification have ignored the termination property. Based on a verified concurrent file system, AtomFS, this paper presents the verification of its termination property, which ensures that every method call will always return under fair scheduling. Proving a method's termination requires to show that when the method is blocked, the source thread of the obstruction will make progress. The core lies in showing the termination of the lock coupling traversal. However, two major challenges applying the idea are as following. (1) The file system is in the shape of a tree and allows threads that block others to diverge on its traversal. As a result, multiple sources of obstruction globally might be found, which leads to the loss of locality in proof, (2) The rename operation needs to traverse on two paths and could bring obstruction across the path. It not only leads to more difficulty in source location, but also could cause the failure in finding the source of obstruction when two renames block each other. This study handles these challenges through two key techniques: (1) Only recognizing each local blocking chain for source location; (2) Determining partial orders of obstruction among threads. A framework called CRL-T have been successfully built for termination verification and apply it to verify the termination of AtomFS. All the proofs are mechanized in Coq.

Key words: concurrent file system; termination; formal verification; Coq

* 基金项目: 国家杰出青年科学基金(61925206)
本文由"形式化方法与应用"专题特约编辑陈立前副教授、孙猛教授推荐.
收稿时间: 2021-09-08; 修改时间: 2021-10-14, 2022-01-10; 采用时间: 2022-01-27; jos 在线出版时间: 2022-02-28

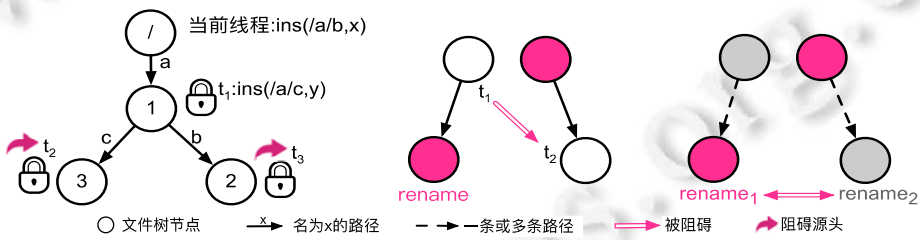
应用依赖文件系统进行数据存储, 然而即使是仔细设计与实现的文件系统都可能包含漏洞, 其中, 死锁、无限循环等终止性漏洞会导致拒绝服务(denial of service)[1]. 形式化方法是目前已知的唯一一种能够保证系统没有编程错误的方法[2,3], 最近的许多工作[4−7]展示了形式化验证文件系统的可行性, 其中, AtomFS[7]是首个被成功验证的并发文件系统. 然而, 这些工作都没有验证文件系统的终止性.

本文验证了 AtomFS 的终止性, 保证了任何一个 AtomFS 的接口在公平调度的条件下都能返回. 证明终止性对于使用 AtomFS 的应用有两点好处: 第一, 其完善了暴露给应用的接口, 保证接口能够确定返回; 第二, 使上层应用能够基于 AtomFS 的终止性验证自身的终止性.

验证 AtomFS 接口的终止性需要说明: 当接口遇到阻碍时, 阻碍终将消除. 为了避免循环论证[8], 需要识别阻碍的源头, 并证明阻碍的源头一定会执行特定操作, 特定操作促使某个衡量进展性的指标减小. 随着特定操作不断发生, 指标不断减小但又不能无限减小, 最终促使阻碍的消除.

然而, AtomFS 的终止性证明具有两点挑战.

- 首先, 树式遍历下存在多个阻碍源头, 使证明失去局部性. AtomFS 的路径遍历实现基于锁耦合算法(lock coupling)实现, 锁耦合算法会先获取下一个节点的锁, 再释放前一个节点的锁. 直观的想法是: 参照锁耦合链表(lock coupling list)的证明[8], 说明比当前线程先开始遍历的操作都会执行结束, 从而当前线程能够无阻碍地完成剩余操作. 然而文件系统操作遍历在文件树上, 路径分叉使线程以树的形式扩散开, 导致阻碍源头分布在多条路径上. 如图 1(a)所示: 当前线程要获取节点 1 的锁, 被 t_1 阻碍, t_1 又被 t_2 阻碍. 除了 t_2 这一阻碍源头, 当前线程要遍历的路径上还将被 t_3 阻碍. 确定所有的阻碍源头并设计相应的进展性指标, 会使证明依赖许多全局信息, 失去局部性(详见第 2.3 节);
- 其次, rename 接口需要遍历两条路径, 如图 1(b)所示. 假如其在一条路径上阻碍了线程 t_1, 另一条路径上被 t_2 阻碍, t_2 对 rename 的阻碍将跨路径进一步阻碍 t_1, 我们称这种现象为跨路径阻碍. 跨路径阻碍一方面使得阻碍源头更加难以确定; 另一方面, 如图 1(c)所示, 多个 rename 相互跨路径阻碍可能导致阻碍成环. 在这种场景下, 无法找到一个阻碍源头, 操作间形成了死锁, 给证明带来了挑战.



(a) 树式遍历存在多个阻碍源头 (b) 跨路径阻碍 (c) 跨路径阻碍成环

图 1 验证 AtomFS 的挑战

针对以上挑战, 我们首先提出了局部的思想来确定阻碍源头, 即仅关注当前线程要拿的下一把锁. 若当前线程要拿的锁被线程 t_1 占有, 则当前线程被 t_1 阻碍; 同理, t_1 可能正在获取下一把锁并被 t_2 阻碍, 使其无法放前一把锁; 线程之间相互阻碍形成了一条阻碍链, 顺着阻碍链找到一个不被阻碍的线程, 此线程即为阻碍源头. 相比于全局的思想, 局部思想下只需要确定一个阻碍源头, 使证明局部性更好.

其次, 我们提出了偏序法来解决多个 rename 跨路径阻碍成环的问题. 跨路径阻碍成环是由于多个 rename 之间阻碍关系无法形成偏序, 我们的证明思路受到了 AtomFS 中 rename 实现的启发, rename 会先锁住要遍历的两条路径(源路径和目标路径)的公共祖先, 只有在完成两条路径的遍历后才放锁, 这保证了其他线程(包括其他 rename)不会出现在两条路径中. 即要么在公共祖先之前或两条路径之后, 从而可以确定 rename 与其他线程阻碍关系的偏序, 找到一个阻碍源头, 避免死锁(详见第 2.4 节).

本文做出了如下贡献:

(1) 针对 AtomFS 树式遍历中存在多个阻碍源头的问题, 提出了局部的思想来仅确定一个阻碍源头;

(2) 针对 AtomFS 中 rename 可能带来的跨路径阻碍成环问题, 提出了偏序法寻找阻碍源头;
(3) 在 Coq 中搭建了 CRL-T 框架, 用于证明程序的终止性;
(4) 在 CRL-T 框架中成功证明了 AtomFS——首个证明了终止性的并发文件系统. 目前我们完成了 ins 接口的证明工作, 并将继续完善剩余接口的证明.

1 相关工作

在系统的终止性漏洞调研方面, 许多工作[1,9−12]调研了系统中的终止性漏洞, 其中: 文献[1]对 157 个文件系统的 CVE 进行了研究, 发现其中 7%的漏洞与终止性相关, 这些漏洞会导致系统拒绝响应; 文献[9]对系统的并发漏洞进行了调研, 发现其中 30%的并发漏洞都来源于死锁; JUXTA[10]对文件系统中的语义漏洞进行了研究, 其中, 死锁相关的语义漏洞平均潜伏期达到了 7 年. 这些工作说明, 终止性漏洞危害高、占比大、难以消除.

许多工作[13−17]在检测终止性漏洞方面取得了进展, 其中, Looper[15]在系统运行时执行动态分析, 检测无限循环导致的终止性漏洞; Gadara[13]通过静态分析为程序的并发行为进行形式化建模, 使用外部控制逻辑控制程序并发行为, 从而避免死锁. 这些工作在实际应用中可以解决特定的问题, 然而这些工作无法保证终止性漏洞的消除, 本身引入了运行开销, 并且目前仍然没有一种统一的方式来检测终止性漏洞.

许多工作对文件系统[4−7]进行了形式化验证, 其中, FSCQ 项目[4,18]通过在霍尔逻辑上扩展了崩溃条件来验证文件系统的崩溃安全性; AtomFS[7]是一个证明了原子性的并发文件系统, 其通过帮助机制来验证 rename 和其他操作间的路径依赖现象. 然而, 这些工作都没有对文件系统的终止性性质进行验证.

在系统的终止性验证方面, IronFleet[19]工作结合了 TLA 风格的自动验证与基于霍尔逻辑的定理证明, 验证了分布式系统的终止性, 然而他们的终止性证明在协议层完成, 不考虑阻碍等问题, 使得他们的证明方法无法直接用于证明文件系统. CertiKOS[20]项目基于 CCAL(certified concurrent abstraction layers)[21]验证了 MCS[22]锁的一个复杂实现的终止性, 其核心想法是, 通过分层来简化证明的复杂性. 然而他们没有一个程序逻辑来指导终止性证明, 不清楚如何用于证明文件系统的终止性. VSync[23]使用 AMC(await model checking)实现了锁的终止性的自动化验证, 然而 VSync 对锁的证明基于给定的应用程序, 不清楚如何将其扩展并证明如文件系统的大型系统. 综上所述, 这些工作都无法直接用于证明文件系统的终止性.

在终止性验证理论方面, 程序逻辑 LiLi(Linearizability and Liveness)[8,24]支持无死锁性、无饥饿性的验证, 并提出了相应的程序逻辑. 验证了锁耦合链表(lock coupling list)的正确性, 然而纸上证明无法带来机器证明同样的高可靠性, 并且锁耦合链表的验证思路无法用于 AtomFS 文件系统的验证(详见第 2.3 节). TaDA-Live[25]在确定性事件基础上进行了分层, 然而其要求分层是静态的. 但在文件系统中, rename 可能改变文件系统结构, 使得分层可能动态变化, 导致其理论无法用于 AtomFS 的证明. 本文工作基于 LiLi 中的验证理论, 同时考虑了文件系统的实际问题, 如树式遍历带来多个阻碍源头的问题, 这使我们成功验证了 AtomFS 的终止性.

2 AtomFS 终止性证明总览

对于串行文件系统而言, 说明终止性不需要考虑线程间的干扰; 而并发文件系统中, 线程间通常需要细粒度锁的同步, 会产生相互干扰. 这种干扰在 AtomFS 中体现为阻碍的形式, 因此证明 AtomFS 的终止性, 需要说明每一个线程遇到的阻碍终将消除. 本节首先介绍已有的终止性证明理论, 然后给出 AtomFS 终止性的直观解释, 并阐述用全局思路证明 AtomFS 终止性遇到的挑战以及我们的解决办法.

2.1 终止性证明理论背景

程序终止性问题一直以来都是研究的热点[8,24,26−29]. 对于并发程序, 特别地, 对于像文件系统这一类基于锁同步的程序, 一个线程可能被另一个线程阻碍, 使前者的终止性依赖于后者(如 t_2 线程因为一直不放锁阻碍 t_1 线程的拿锁操作, 使得 t_1 的终止性依赖于 t_2). 因而, 证明基于锁的并发程序的终止性, 需要在公平调度的假

设下, 对于每个线程证明其环境(即其他线程)一定会完成特定事件, 使当前线程的阻碍解除(如 t_2 一定会放锁来解除 t_1 的阻碍), 阻碍解除之后, 再按照串行程序的终止性证明逻辑说明程序的终止性.

LiLi[8]中提出了确定性事件(definite action)来刻画环境一定会完成的事. 具体地, 确定性事件的形式为 $P \rightsquigarrow Q$, 其含义为: 一旦 P 成立在系统状态上, 随着程序执行, P 会一直保持成立直到 Q 成立并且 Q 终将成立.

为了衡量确定性事件的发生对阻碍解除的贡献, 需要定义一个进展性指标, 每当确定性事件发生, 进展性指标就会减小, 并且进展性指标需要是良序的(well-founded order), 即不能无限减小.

LiLi 中, 提出了确定性进展来形式化阻碍解除的过程, 如图 2 所示: 假如当前语句遇到阻碍, 我们需要说明一定有确定性事件发生(阻碍条件), 使进展性指标减小(指标减小条件), 并且指标不会增大(指标不增条件). 因此, 随着有限个确定性事件的发生, 进展性指标减小为最小值, 标志着阻碍的解除(详见第 5.1 节).

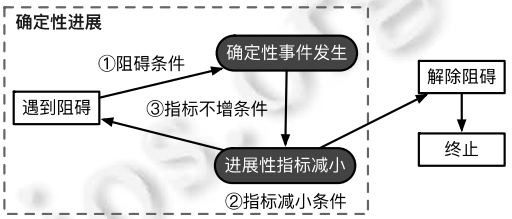


图 2 程序终止性证明逻辑

基于 LiLi 中的理论, 证明程序终止性的逻辑包括: (1) 证明确定性进展条件; (2) 在无阻碍的条件下, 按照一般串行程序终止性的证明逻辑, 说明当前语句能够终止.

终止性证明中, 核心难点是定义正确的确定性事件. 为了避免循环论证, 确定性事件必须确定发生. 例如, t_1 的代码为 lock *A*; unlock *A*, 线程 t_2 的代码为 lock *A*; lock *B*; unlock *A*. 假如 t_2 先拿到 *A* 的锁, 使 t_1 在 lock *A* 语句遇到阻碍, 直观想法是: 将确定性事件定义为 t_2 拿到 *A* 的锁后终将放掉 *A* 的锁, 那么必须保证 t_2 拿到 *A* 的锁之后能无阻碍地执行放锁操作. 如果 t_2 可能因为 lock *B* 语句受到阻碍, 这便不是一个正确的确定性事件定义. 为了保证确定性事件的确定性, 我们要识别出线程遇阻时的阻碍源头, 并将源头能够推进阻碍解除的操作作为确定性事件.

2.2 AtomFS终止性的直观解释

表 1 展示了 AtomFS ins 接口的伪代码, 我们以 *ins* 为代表, 介绍 AtomFS 的锁耦合(lock coupling)[30]路径遍历算法. 具体地, 在已获取 *cur* 节点锁的情况下, 它会先获取路径上下一个节点的锁(第 13 行), 然后再释放手中 *cur* 节点的锁(第 14 行). *ins* 先以这样的方式遍历 *path* 路径(第 4 行), 随后完成插入操作(第 5 行、第 6 行).

表 1 AtomFS *ins* 接口的伪代码

```
1  //error and corner case handling omitted
2  def ins(path,name)
3      lock(root);
4      cur=locate(root,path);
5      node=init(·);
6      insert(cur,name,node);
7      unlock(cur);
8      return success;
9  def locate(cur,path)
10     i=0;
11     while (path[i]) {
12         next=find(cur,path[i]);
13         lock(next);
14         unlock(cur);
15         cur=next;
16         i=i+1;
17     }
18     return cur;
```

通过观察, 我们发现了 AtomFS 拿锁行为的两个特点: 有序性和不可旁路性.

有序性是指 AtomFS 的所有接口(包括文件操作, 如 *read*, *write*)都会从文件树的根开始, 沿着参数指定的路径遍历, 并且 AtomFS 中不考虑硬链接和软链接, 这保证了一个接口内部的拿锁行为是按照树根到树梢的顺序; 不可旁路性是指锁耦合算法下, 每一个正在路径遍历的线程拿着至少一把锁, 这使得同一条路径上后面的线程无法超越前面的线程.

AtomFS 中这种拿锁的特点使我们得到了两个深入的发现.

- 第一, 先遍历的线程一定能够无阻碍地执行结束. 根据有序性, 每个操作都要获取根节点的锁, 优先获取根节点锁的线程为先遍历的线程. 根据不可旁路性, 先遍历的线程将要遍历的节点不可能被别的线程占有, 因为后遍历的线程无法超过先遍历的线程. 这保证了先遍历的线程可以无阻碍地执行, 并且该线程要遍历的节点数是有限的, 因此先遍历的线程具有终止性;
- 第二, 后遍历的线程只需要等待先遍历的线程完成. 当先遍历的线程完成操作返回, 总会有一个新的线程成为最先的线程. 由于先遍历的线程数是有限的, 当前线程最终会成为最先的线程, 从而能够保证执行结束.

这两点发现从全局的视角看待一个线程从遇阻到阻碍解除的过程, 给出了每个线程为什么能够终止的直观解释. LiLi 中使用这种全局思路去证明锁耦合链表的终止性: 对于一个线程 t, 其将最接近链表末尾的线程识别为阻碍源头, 将确定性事件定义为阻碍源头的线程需要完成的操作, 进展性指标通过比线程 t 先开始遍历的线程的完成情况来刻画(完成度越高, 则指标越小, 不存在先开始遍历的线程时, 指标为最小值), 并说明随着确定性事件发生, 这个指标会不断减小, 直到当前线程成为最接近链表末尾的线程, 便可以无阻碍地完成自己的操作.

2.3 全局思路证明AtomFS终止性的挑战

然而, 将这种全局思路应用于 AtomFS 终止性的证明时, 遇到了两点挑战.

(1) AtomFS 的树形结构下存在多个阻碍源头, 使证明失去局部性.

类比锁耦合链表, 我们首先尝试将路径末尾的线程识别为阻碍源头, 然而文件树中可能存在多个阻碍源头. 考虑图 1 中的例子, *ins*(/*a*/*b*,*x*)将要遍历 1 号、2 号节点构成的路径, 记为路径 1-2, 并发现 2 号节点此时被线程 t_3 锁住, 将 t_3 确定为一个阻碍源头. 然而文件树中不止一条遍历路径, 线程 t_1 此时拿着 1 号节点的锁,它按照路径 1-3 遍历, 这时 t_1 被末尾的 t_2 阻碍, t_2 间接地通过 t_1 阻碍了当前线程, 因此我们需要将 t_2 也识别为阻碍源头.

识别出多个阻碍源头依赖于许多全局信息. 图 1 中, 要识别出 *ins* 操作的阻碍源头, 需要: (1) 根据路径"/*a*/*b*"计算出它将遍历的 1 号、2 号节点; (2) 找出锁住 1 号、2 号节点的线程(t_1 和 t_3), 并获取它们的遍历信息; (3) 对这些线程(t_1 和 t_3)重复步骤(1)–步骤(3)的过程, 直到找到可以无阻碍完成遍历的线程作为阻碍源头. 上述计算过程中依赖的全局信息, 包括文件树的多条路径上节点的内容以及分布在这些路径上的线程的遍历情况.

这些全局信息被用于定义确定性事件以及进展性指标, 这意味着当任何一个全局状态发生变化时, 需要证明它对于当前线程进展性的影响. 例如图 1 中, 我们需要说明阻碍源头 t_3 的行为如何给当前线程带来进展. 然而 t_3 释放 2 号节点的锁对于当前线程获取下一把锁并不是必要的, 对全局信息不必要的依赖, 使证明失去了局部性.

(2) rename 带来跨路径阻碍现象, 使阻碍可能成环, 无法找到阻碍源头.

文件系统中除 rename 外的接口仅按照一条路径进行遍历, 线程间相互阻碍形成的链也在一条路径上, 阻碍链上不被阻碍的线程即为阻碍源头. 然而 rename 接口需要遍历两条路径——源路径和目标路径, rename 可能与别的线程在这两条路径上分别形成阻碍关系, 使得一条路径上的线程通过 rename 间接阻碍到另一条路径上的线程. 这种跨路径阻碍现象, 使我们在确定阻碍源头时需要特别考虑 rename 的遍历情况. 并且如果多个 rename 操作相互阻碍, 如图 1(c)中 $rename_1$ 一条路径的遍历被 $rename_2$ 阻碍, $rename_2$ 一条路径的遍历被 $rename_1$ 阻碍, 无法找到一个阻碍源头, 导致终止性证明失败.

2.4 解决办法

针对上述两点挑战, 我们首先提出使用局部思想来仅识别一个阻碍源头, 使证明具有局部性; 接着提出偏序法来解决 rename 的跨路径阻碍成环问题.

(1) 使用局部思想来仅识别一个阻碍源头, 使证明具有局部性.

在全局思想中, 一个线程无阻碍的条件被定义为能够无阻碍地完成剩余的所有操作. 然而这样一个无阻碍的定义过强了, 我们可以仅关注遍历的下一步能否执行, 即下一把锁能否拿到. 如果能够顺利执行, 我们便认为其局部处于无阻碍的状态; 假如下一把锁不能拿到, 我们仅关注下一把锁上形成的阻碍如何解除. 如图 3(a), 假设当前线程要拿的下一把锁是 1 号节点, 其被线程 t_1 占据, t_1 下一把要拿的锁被 t_2 占据, t_2 的下一把锁能够拿到, 那么便认为 t_2 是一个局部的阻碍源头, 将 t_2 放掉 2 号节点的锁定义为确定性事件. 随着 t_2 放掉 2 号节点的锁, 使得 t_1 变成了新的阻碍源头, t_1 放掉 1 号节点的锁成为新的确定性事件, 当它发生后, 在 1 号节点上形成的阻碍便得到了解除.

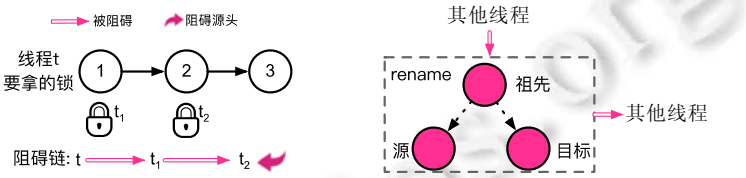
通过说明每个线程的下一把锁都能够成功获取, 并且每个 AtomFS 接口要遍历的节点数是有限的, 我们便能够证明 AtomFS 接口的终止性. 相比于全局的思想, 局部思想依赖的状态更少了, 并且只需要确定一个阻碍源头, 使我们的证明具有更好的局部性.

(2) 使用偏序法解决跨路径阻碍成环问题.

要证明跨路径阻碍不会成环, rename 实现本身要保证不会形成死锁. 我们的证明思路受到了 AtomFS 的 rename 实现的启发, rename 会先遍历源路径和目标路径公共的部分, 拿着这两条路径公共祖先的锁, 再分别去获取源目录和目标目录的锁, 获取了这两把锁之后, 再放掉公共祖先的锁.

如图 3(b)所示: 这种拿锁策略保证了祖先到源目录和目标目录之间的路径上没有其他线程, 使祖先与源目录和目标目录形成了一个整体. 在公共祖先的锁没有释放时, rename 对其他操作的阻碍是由于公共祖先的拿锁造成的, 同时, rename 受到的阻碍是在路径遍历过程中遇到的, 因此这个阶段与 rename 产生阻碍关系的线程要么在公共祖先之前或是在两条路径之后, 使 rename 与其他线程的阻碍关系是偏序的. 当公共祖先的锁释放之后, 这个性质依然成立. 因此, 多个 rename 之间不会出现相互阻碍成环的问题.

同时, 考虑 rename 之后, 在寻找阻碍源头时, 不能仅根据下一把锁能否拿到判定这个线程是否是无阻碍状态, 因为 rename 中公共祖先的锁需要在完成两段路径遍历之后再释放, 即使 rename 当前仍然能够顺着路径遍历, 但其将来可能被别的线程所阻碍, 不保证能够释放公共祖先的锁. 因此, 被 rename 公共祖先的锁阻碍的情况下, 需要根据其能否顺利完成两条路径遍历来定义其是否是无阻碍状态.



(a) 局部思想识别阻碍源头 (b) 偏序法确定 rename 与其他线程阻碍关系

图 3 我们的解决办法

3 CRL-T 框架概述

在 Coq 中搭建了 CRL-T (concurrent relational logic with termination)框架来支持对并发程序终止性的验证, CRL-T 框架基于两个已有的工作, 下面我们对这些工作进行简要的介绍.

- 首先, CRL-T 继承了已有 Coq 框架 CRL-H[7]对并发 C 程序验证的支持. CRL-H 框架中对 C 语言的一个子集进行了建模[31,32], CRL-H 使用 LRG (local rely guarantee)[33]进行并发验证, *R*(rely)和 *G* (guarantee)分别用来描述环境和当前线程在共享状态上的状态转移, 共享状态的界限通过 *I* (invariant)来刻画. CRL-H 框架要求定义抽象状态和抽象操作, 并证明抽象操作和具体操作之间的精化关系[34]来说明功能正确性, CRL-H 还支持了帮助机制以此来证明原子性[35], 其被成功用于证明了 AtomFS 的原子性.在 CRL-T 中, 我们专注于证明程序的终止性, 因此 AtomFS 的高层操作被默认为空语句(skip), 同时去掉了对于帮助机制的支持; 可以保留 CRL-H 中的功能来继续支持功能正确性和原子性的证明;

- 其次, CRL-T 中融合了 LiLi[8]中对终止性证明的支持. LiLi 中提出了一套统一的程序逻辑来同时支持对无死锁性和无饥饿性验证. 其中, 无死锁性同时允许有限的延迟和阻碍, 无饥饿性仅允许有限的阻碍. AtomFS 中仅存在阻碍, 因此在 CRL-T 中我们支持了证明无饥饿性(即终止性)部分的逻辑. 具体地, LiLi 中引入了令牌(token)来证明一般情况下(即无阻碍时)循环的终止性, 并要求每进一次循环消耗掉一个令牌; 有阻碍时, 提出了确定性事件和确定性进展来证明终止性(见第 2.1 节). CRL-T 在 Coq 中实现了这些概念和相应的推理规则.

使用 CRL-T 进行证明的流程如图 4 所示, 其中, 白色方框为用户需要提供的源码, 包括规约、实现和证明; 深色框中的推理规则和导入 Coq 建模的 C 代码过程由 CRL-T 框架提供. LiLi 中证明了推理规则蕴涵终止性敏感的上下文精化, 终止性在 LiLi 中采用事件序列(event trace)的方式形式化. 简单来说, 其收集了程序在任意上下文调用下、所有可能的交互执行情况作为事件序列集合, 并要求在这些事件序列集合中每一个方法调用事件都有对应的返回事件, 以此来刻画程序的终止性. 并证明终止性敏感的上下文精化能够推导出终止性,说明了程序逻辑的可靠性. 证明后的 C 代码通过编译执行. 我们使用 CRL-T 证明了 AtomFS 的终止性, 并将在后续章节分别介绍如何定义 AtomFS 的规约与证明 AtomFS 的终止性.

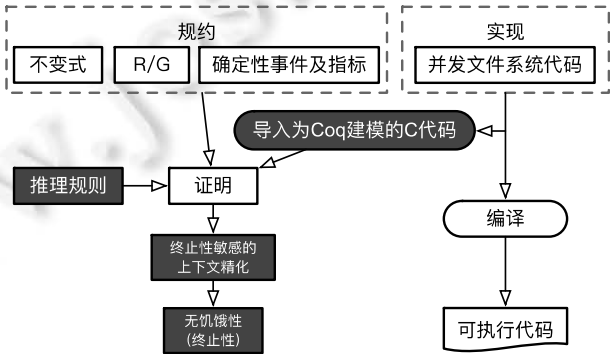


图 4 将 CRL-T 用于并发文件系统证明的流程

4 定义 AtomFS 的终止性规约

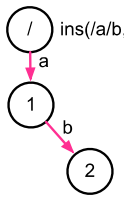
本节首先介绍辅助状态的设计, 辅助状态使我们更方便地定义确定性事件, 接着介绍进展性指标的设计, 最后描述其他规约的定义.

4.1 辅助状态设计

辅助状态[36]是一种常用的验证技术, 引入辅助状态的目的是为了更方便地证明. 辅助状态独立于底层状态, 而仅存在于抽象模型中, 它们不会影响到系统的行为, 可以在程序执行的同时原子地更新辅助状态.

我们引入了各个线程的遍历情况这一辅助状态来帮助 AtomFS 终止性的证明. 遍历状态 *travstat* 记录了线程当前将要遍历的节点的情况. 一般情况下其为 Some *inum*, 意为接下来要去执行 *lockinum* 语句; 或者是 *None*, 意为接下来不会或者是无法去执行拿锁操作. 对于 *rename* 拿着公共祖先锁的情况, 其需要遍历两条路径上的节点, 因此 *travstat* 需要记录更多信息, 包括一条路径上当前所处的节点位置 *curinum*、已经遍历的路径 *havetrav* 和将要遍历的路径 *totrav*, 它们形成的三元组(*curinum*,*havetrav*,*totrav*)记录了一条路径上的遍历情况 *pathstat*; 要记录两条路径的遍历情况, 则 *travstat* 为(*pathstat*$_1$,*pathstat*$_2$).

图 5 展示了 *travstat* 是如何随着程序执行同步更新. 需要注意的是, 在 *lock* 1 语句之后, *travstat* 设置为 *None* 的含义是: 当前线程不会立马去执行下一个拿锁操作, 而是等到 *unlock root* 之后才要去获取下一把锁. 因此, 我们在 *unlock root* 之后再设置下一个拿锁目标.



语句(执行后)	travstat
	Some root
lock root	Some 1
lock 1	None
unlock root	Some 2
lock 2	None
unlock 1	None

图 5　遍历状态随着代码执行的变化

遍历状态的引入，是为了更方便地从局部思路定义确定性事件，我们将在下一节详细说明.

4.2 局部思路定义确定性事件

确定性事件为环境需要完成的、帮助当前线程解除阻碍的事件，在 AtomFS 中即为放锁事件. 在定义确定性事件时，我们需要明确 3 点：无阻碍状态的定义、哪个线程放锁和放哪把锁.

- 首先，确定性事件发生在一个无阻碍的线程上，要确定线程是否处于无阻碍状态，我们需要这个线程的遍历信息. 然而各个线程的遍历信息在参数对应的内存中，是各自的局部信息，不应该被其他线程可见. 解决这个问题便用到了第 4.1 节中提出的遍历状态这一辅助信息. 通过查看各个线程的遍历状态，我们能够知道它接下来将如何遍历，通过查看文件树，能知道它要遍历的节点是否正被别的线程占据，从而确定这个线程是否处于无阻碍状态. 特别地，对于 rename 而言，当其拿着公共祖先的锁时，需要遍历两条路径才会放掉公共祖先的锁，因此其无阻碍状态需要根据这两条路径是否能顺利遍历来定义;
- 其次，要回答哪个线程放锁，需要明确当前线程的阻碍源头. 按照局部思路，我们需要：① 找出当前线程要拿的下一把锁被谁占据；② 如果占据着锁的线程是无阻碍状态，那么便为阻碍源头；③ 如果其也被阻碍，需要重复上述过程，直到找到所形成的阻碍链的阻碍源头. 例如在图 3(a)中，t_1 接下来要遍历的节点 2 被 t_2 占据，而 t_2 要遍历的节点 3 是无锁状态，因此可以确定 t_2 为阻碍源头. 另外，遍历状态为 None，也意味着线程处于无阻碍状态;
- 最后，确定性事件中放的锁需要能帮助解除阻碍. 为了表达这一点，我们需要将阻碍链提取出来，获取阻碍链最后的那个节点，并约束确定性事件放的锁为这个节点的锁. 例如图 3(a)中形成的阻碍链为 t_1-t_2，阻碍链最后的为 2 号节点，因此确定性事件应当为 t_2 释放 2 号节点的锁.

综上所述，我们对确定性事件的形式化定义见表 2.

表 2　确定性事件在 Coq 中的定义

```
(*detailed definition of Inv, locked, blockingchain, notblocked and unlocked are omitted*)
Definition dp inum t:=
  ∃lfs aux tsrc, Inv lfs aux∧
                 locked inum t lfs∧
                 blockingchain tsrc t inum lfs aux∧
                 notblocked t lfs aux.

Definition dq inum:=
  ∃lfs aux tsrc, Inv lfs aux∧
                 unlocked inum lfs.

Definition D t:=∀inum, dp inum t⇝dq inum.
```

这是线程 *t* 释放 *inum* 节点这把锁的确定性事件，其中，*dp inum t* 和 *dq inum* 分别为确定性事件的前置和后置. 我们在前置中约束当前系统状态包含底层状态 *lfs* 和辅助状态 *aux* 以及他们应遵循的不变式(由 *Inv* 条件表达)，并要求 *inum* 当前被线程 *t* 锁住(由 *locked* 条件表达)；同时，线程 *t* 阻碍了线程 *tsrc* 并形成了一条链，这条链尾的节点正是 *inum*(由 *blockingchain* 条件表达)；同时，线程 *t* 当前没有被阻碍(由 *notblocked* 条件表达). 后置中则简单地要求系统状态仍遵循不变式，并且 *inum* 节点的锁被释放(由 *unlocked* 条件表达).

4.3 最远距离链定义进展性指标

确定性事件的完成逐步推动系统的进展，需要定义相应的指标来衡量进展性，这个指标需要是良序的(well-founded order). 首先尝试使用阻碍链的长度作为进展性指标，例如在图 3 中，阻碍链由 t_1 和 t_2 构成，记作 t_1-t_2，其长度为 2；当 t_2 放掉 2 号节点锁的确定性事件发生后，这时阻碍链变为了 t_1，长度减为 1. 以此推广开来，当阻碍链末尾的线程放锁之后，阻碍链的长度理应减少 1.

然而在文件系统中存在某些边界情况，使得我们不能简单使用阻碍链的长度作为进展性指标. 如图 6 所示：在 t_1 线程释放 *b* 点的锁之前，对于 *rename* 而言，剩余阻碍链是 t_1，长度为 1；然而 t_1 放锁之后，*rename* 的阻碍并没有解除，这时它会继续遍历目标路径，并被 t_2 线程阻碍，此时剩余阻碍链是 t_2-t_3，阻碍链长度反而增加了，称为阻碍链改变现象.

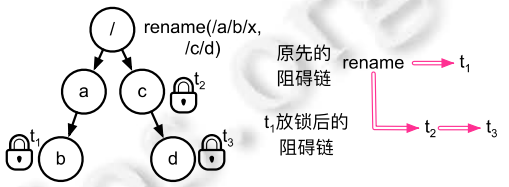


图 6　阻碍链改变现象

出现阻碍链改变现象的原因在于，*rename* 的阻碍条件需要根据能否遍历完两条路径来决定. 这时，阻碍链的长度并不完全代表进展性，我们还需要考虑这个线程当前能遍历的最远距离，假如最远距离变远了，意味着其受到的阻碍相对更小了，即使阻碍链可能变长.

我们提出将阻碍链中各个线程的最远遍历距离提取出来形成的最远距离链作为进展性指标，其可以表达为 $dis_1(t_1)$-$dis_2(t_2)$-…-$dis_n(t_n)$，其中，$dis_i(t_i)$意为最远遍历 dis_i 距离后会被 t_i 所阻碍. 对于图 3 中的例子，最远距离链为 $0(t_1)$-$0(t_2)$，意为当前线程最远遍历距离 0 就会被 t_1 阻碍，t_1 最远遍历距离 0 被 t_2 阻碍，t_2 则没有被阻碍. 要比较进展性指标的大小关系，我们只需要对 dis_1-dis_2-…-dis_n 按照字典序进行大小比较. 比较规则如图 7 所示. 简单来说，链越短，最大前进距离越长，则指标越小. 为了保证指标为良序(不存在无限递减的链条)，我们使用参数中允许的最大路径长度(max)为最远遍历距离的约束(由 *bound_elem* 条件表达). 对于 *rename* 而言，其最远遍历距离以公共祖先为起始点计算，按照先源路径后目标路径的方式累加. 现在图 6 中原先的指标为 $2(t_1)$，当 t_1 放锁后，指标变为 $3(t_2)$-$0(t_3)$，符合了确定性事件的发生使指标变小的要求.

$$\text{(Distance) } dis \in nat \qquad \text{(Metric) } m \in nil \mid dis::m$$

$$\frac{}{nil<dis::m} \qquad \frac{dis_1 > dis_2}{dis_1::m_1<dis_2::m_2} \qquad \frac{m<m'}{dis::m<dis::m'}$$

$$m_1 <_{max} m_2 \triangleq m_1 < m_2 \wedge (bound\_elem\ m_1\ m_2\ max)$$

图 7　指标良序定义

4.4 其他规约

CRL-T 框架中除了确定性事件、进展性指标之外，还需要定义 *R*/*G* 和 *I* 作为规约. 要定义一个线程的 *G*，需要找出这个线程可能进行的所有共享状态转换. 在文件系统中，我们总结出了以下状态转换，分别是拿锁(Lock)、放锁(Unlock)、插入一个节点(Ins)、删除一个节点(Del)、写文件内容(Write)、分配辅助状态(AllocD)以及局部状态变化(ID). 线程 *t* 的 *R* 则定义为其他线程的 *G* 的并集.

在 AtomFS 的证明中，共享状态具体包括底层状态和辅助状态，需要用 *I* 刻画出共享状态的分布并约束共享状态. 这些约束需要时刻成立在共享状态上，并在证明中起到关键作用. 我们将在下一节具体介绍不变式的内容.

5 证明 AtomFS 的终止性

CRL-T 框架提供了推理规则来帮助我们证明 AtomFS 的实现与规约相一致, 首先介绍框架的推理规则, 然后讨论在证明中起到关键作用的不变式, 接着描述如何证明确定性进展以及其他证明.

5.1 推理规则

CRL-T 提供了推理规则来帮助使用者证明终止性. 推理规则分为两类, 分别是并发对象规则和 C 语句规则. 其中, 并发对象规则给出了一个并发对象符合终止性需要满足的条件, 主要包括这个并发对象的每个接口都能按照 C 语句规则完成推理, 以及对 *R*, *G*, *I*, *D* 等规约的约束. 这里主要讨论对 *D* 的良构性(well-formedness)约束, 具体地, 假如线程 *t* 的确定性事件被触发了(即确定性事件的前置成立), 记作 $Enabled(D_t)$, 需要证明: ① 环境执行一步 $Enabled(D_t)$依然成立; ② 线程 *t* 自己执行一步, 要么 $Enabled(D_t)$成立, 要么这一步完成了确定性事件. 这两个证明义务保证了确定性事件 D_t 一旦被触发, 就一定会由线程 *t* 自身完成.

C 语句规则的逻辑判断的形式可以简单表示为 $D,R,G,I\vdash_t\{P\}S\{Q\}$, 其中, *D* 是确定性事件, *R*/*G* 为依赖/保证条件, *I* 是不变式, *P*/*Q* 为前置/后置条件, *S* 是要证明的语句, *t* 为当前证明的线程号. CRL-T 为每一种 C 语言构造都提供了推理规则, 包括赋值规则、if 规则、while 规则、函数规则和锁规则等等. 与终止性证明相关的规则是锁规则和 while 规则, 下面以锁规则为例来介绍推理规则.

CRL-T 中将锁作为抽象原语, 其中, 锁的抽象基于满足部分无饥饿性[24]的锁, 如 ticketlock, MCSlock 等. 在强公平调度的条件下, 这些锁的抽象保证了只要我们证明拿锁的线程终将放锁, 便可说明锁的终止性. 一个简化的锁规则如图 8 所示, *B* 定义了线程 *t* 下一把要拿的锁 *inum* 没有被其他线程占有(拿锁条件). 需要注意的是: 假如当前线程的确定性事件被触发(即 $Enabled(D_t)$成立), 由于确定性事件的完成不能被阻碍, 因此需要证明这时 *B* 是成立的(无阻碍条件).

$$\frac{\begin{array}{c}(\text{拿锁条件})B\triangleq \text{inum.lock=0}\quad (\text{锁串行规则})\vdash_t\{p\wedge B\}\ \text{lock(n)}\ \{q\}\\ (\text{无阻碍条件})p\wedge Enabled(D_t)\Rightarrow B\\ (\text{确定性进展})p\Rightarrow DefProg\qquad \text{others omitted}\end{array}}{D,R,G,I\vdash_t\{p\}\ \text{lock(inum)}\ \{q\}}\ (\text{lock rule})$$

图 8 简化的锁规则

在 *B* 成立的时候, 我们只需要按照锁串行规则得到 *lock* (*inum*)语句的后置 *q*; 当 *B* 不成立时, 我们需要通过确定性进展说明 *B* 终将成立. 如图 2 所示, 确定性进展条件 *DefProg* 的成立需要以下 3 个条件同时成立.

① 阻碍条件: 当前线程要么无阻碍(*B* 成立), 要么存在其他线程的确定性事件被触发;

② 指标减小条件: 当其他线程的确定性事件发生时, 将促使当前线程的进展性指标减小;

③ 指标不增条件: 假如当前线程一直处于阻碍状态, 环境执行不会使进展性指标增加.

由于一旦当前线程遇到阻碍, 根据条件①, 我们知道有其他线程的确定性事件被触发; 根据并发对象规则中确定性事件一旦触发必定完成和条件②, 能得到当前线程的进展性指标减小; 并且由于条件③, 能知道进展性指标会不断减小, 并且由于指标的良序性, 其不会无限减小, 因此 *B* 终将成立.

在锁规则中, 省略了一些与 *R*, *G*, *I* 相关的条件, 我们将在第 5.4 节中介绍这些条件的证明.

5.2 不变式及其在证明中的作用

在介绍确定性进展的证明之前, 首先介绍我们提取的系统不变式, 这些不变式不仅作为规约刻画了系统中的共享状态[37], 还描述了共享状态的良构性, 在证明中起到关键的作用[38]. 表 3 中列举了主要的用于刻画系统状态良构性的不变式, 在这里, 选取最重要以及有趣的两个不变式进行说明.

(1) *Blockingchain_exists* 不变式描述的是对任意一个线程, 其要么处于无阻碍状态, 要么处于阻碍状态并且存在一条从这个线程开始的阻碍链. 其中, 阻碍条件的定义如第 4.2 节所述, 阻碍链的存在则

说明了阻碍的无环性, 阻碍链的最后那个线程需要处于无阻碍状态, 因而是阻碍源头. 在说明系统进展性时, 这个不变式起到了最关键的作用, 我们将在第 5.3 节描述证明确定性进展时, 如何用到这个不变式;

(2) *Rename_wellformed* 不变式描述的是对于 *rename* 而言, 当其拿着公共祖先的锁, 遍历两条路径时, 公共祖先分别到源和目标的路径上没有其他线程, 即这两条路径上的节点要么被 *rename* 锁住, 要么是无锁状态. 这个不变式保证了公共祖先与源和目标形成了一个整体, *rename* 和其他线程的阻碍关系是偏序的, 从而保证了多个 *rename* 不会出现相互阻碍的现象, 保证了阻碍源头的存在性. 在 *Blockingchain_exists* 不变式的证明中, 当讨论到与 *rename* 相关的情况时, *Rename_wellformed* 起到了重要作用.

表 3 AtomFS 中主要的不变式

不变式名称	刻画的状态	不变式含义
Lfs_wellformed	文件系统	文件系统具有树的性质, 如每个节点从根可达
Aux_wellformed	辅助状态	辅助状态记录各个线程的遍历状态, 对应一块精确的区域
Blockingchain_exists	文件系统和辅助状态	任何一个线程要么无阻碍, 要么存在一条从其开始的阻碍链
Rename_wellformed	文件系统和辅助状态	*Rename* 的公共祖先到源和目标之间没有其他线程

5.3 证明确定性进展

确定性进展的证明分为两处, 分别是在 while 规则和锁规则中. 如表 1 第 11 行所示: while 语句存在于 *locate* 函数中, 其循环的对象是 *path* 参数. 由于 *path* 的长度固定, 使得这个 while 循环是有界的. 我们可以轻松地证明其确定性进展的成立.

对于锁规则, 证明确定性进展需要说明其 3 个条件的成立.

- 首先, 对于阻碍条件, 需要对当前线程要拿的锁分情况讨论: (1) 这个锁没有被别的线程获取, 这时当前线程无阻碍; (2) 这把锁被占有, 并且不是被当前线程自己占有(否则, 能根据前提推出矛盾). 我们需要找出阻碍源头, 并说明其确定性事件被触发. 直观想法是: 定义一个函数, 这个函数接受系统状态作为参数, 其可以顺着阻碍链查找返回阻碍源头. 然而, 这样一个函数难以定义成严格递减的形式, 因为无法仅根据系统状态说明这个阻碍链的无环性. 解决办法是增加对系统状态的良构性约束, 这个约束要说明系统中任何一条阻碍链都是无环的, 并且每条阻碍链的阻碍源头都是存在的. 这一约束体现为 *blockingchain_exists* 不变式, 其始终成立在系统状态上. 应用这一不变式, 我们可以得到阻碍源头, 并证明其确定性事件被触发;
- 第二, 对于指标减小条件, 假设原有指标是 m:$dis_1(t_1)$-$dis_2(t_2)$-…-$dis_n(t_n)$, 由于阻碍源头 t_n 放锁, 对 $t_{(n-1)}$ 分情况讨论: 假如其仍处于阻碍状态, 其最远遍历距离变大了, 根据图 7 所示规则, 新的指标 $m'<m$; 假如其处于无阻碍状态, 则最远距离链变短了, 同样有 $m'<m$. 由此说明了确定性事件的发生使指标减小;
- 第三, 对于指标不增条件, 由于当前线程一直处于阻碍状态下, 并通过 *blockingchain_exists* 条件得知, 存在一条从当前线程开始的阻碍链, 我们对环境执行的一步进行分类讨论来说明指标不增条件的成立.
  - (1) 假如执行的线程不在阻碍链的线程中, 其无法影响到阻碍链涉及到的相关状态, 因此阻碍链不变, 进展性指标也保持不变;
  - (2) 假如执行的线程在阻碍链中, 除去不改变共享状态的 ID 操作, 可能执行其他操作的只有阻碍链中的 *rename* 线程或是阻碍源头, 因为其他线程都处于阻碍状态, 对其分情况讨论: (i) 阻碍链中 *rename* 线程的执行并不改变其最远遍历距离, 因此阻碍链不变; (ii) 阻碍源头发生的操作如果是确定性事件, 则阻碍链变短, 否则阻碍链不变.

综上所述, 确定性进展条件成立.

5.4 其他证明

CRL-T 中引入的新的与终止性相关的证明义务还包括并发对象规则中对 D 的良构性的证明. 如第 5.1 节所述, D 的良构性要求确定性事件 D 一旦被触发, 就一定会由线程自身完成, 这实质上要求了 D 的前置条件需要具有稳定性. AtomFS 中, D 定义在阻碍源头上, 由于所有操作拿锁的有序性和不可旁路性, 这保证了阻碍源头接下来要遍历的状态无法被环境所修改, 只能由线程自身去修改, 并完成确定性事件, 因此保证了 D 的良构性.

除了上述证明义务, 在 CRL-T 中, 还需要完成与并发相关的证明义务, 具体的包括以下 3 个方面. (1) 证明 G 描述了每一个共享状态的变化; (2) 证明所有断言在 R 下具有稳定性; (3) 证明系统不变式始终保持成立. 值得一提的是, 我们需要证明与阻碍链相关的不变式 *blockingchain_exists* 始终成立, 这需要对环境执行一步的情况分类讨论, 这与证明确定性进展的指标不增条件的讨论类似, 只不过还需要考虑当前线程处于无阻碍状态下时, 环境执行一步后的情况. 具体的证明可以参考我们的 Coq 代码.

6 实现、测试及讨论

本工作使用 Coq 构建了 CRL-T 框架, 同时在新框架中完成了对 AtomFS 终止性的证明. 本节首先介绍代码实现情况和测试情况, 并讨论工作的可信基与局限.

6.1 实现和测试

CRL-T 的代码实现基于 CRL-H 框架, 新增的部分包括确定性事件相关的定义、锁相关的推理规则, 并需要根据新的状态模型修正已有的定理.

在 AtomFS 的规约上, 我们复用了已有的文件系统的建模, 其余部分都无法复用, 包括 R/G 和 I 的定义、D 以及进展性指标函数(接受系统状态返回线程 t 的指标)的定义. 在进行代码证明前, 应用 CRL-T 的理论对 AtomFS 的终止性进行了分析, 使我们在 Coq 证明前获得了 AtomFS 终止性的理论解释.

证明的代码量见表 4, 目前完成了 *ins* 接口的代码证明, 我们的证明/代码比例高于原有 AtomFS 的比例(约 100/1), 主要因为新增了进展性相关的证明(约 3 千行). 其中, 路径遍历函数 *locate*(见表 1)复用于其余接口(*del*, *rename*, *open*, *read*, *write*)中, 因此在这些接口中无需重复 *locate* 函数的终止性证明. 临界区的代码受到锁的保护, 不会与环境相互干扰, 我们仅证明了其串行规约, 还将完成完整的证明作为将来的工作.

表 4 AtomFS 的规约、实现及证明行数

组成	代码行数
R/G	510
I	660
D 及指标函数	65
ins 接口非临界区部分代码行数	31
ins 接口非临界区部分证明	6 500

AtomFS 是一个用户态的内存文件系统, 能够运行许多实际的软件, 包括 Vim 和 GCC 等. 我们使用一些实际的应用对 AtomFS 进行了测试, 包括克隆 git 项目 xv6、对 xv6 进行编译、拷贝 qemu 的源码并对源码进行字符串查找. 我们还使用了 Filebench 中两个常用的工作负荷(workload)Fileserver 和 Webproxy 进行了测试. AtomFS 通过了这些测试, 测试性能结果见文献[7]. 在终止性上, 在这些测试中, AtomFS 都能及时返回.

6.2 讨 论

与其他形式化证明工作相同, 我们的工作中存在一些可信基: 首先, 依赖于 Coq 系统本身的可靠性;其次, 系统调用通过 VFS 和 FUSE 内核模块分发给 AtomFS, 为保证整个系统调用的终止性, 可以基于 AtomFS 接口的终止性进一步验证 FUSE 和 VFS 的终止性; 并且, AtomFS 的终止性依赖于调度器保证公平调度条件.

同时, 工作还存在一些局限: 首先, AtomFS 不支持崩溃一致性, 如何结合崩溃一致性和终止性的证明, 是一个有意义的研究方向; 其次, CRL-T 框架的终止性理论不具有完备性, 暂时仅支持定义单层的确定性事件,

我们可以用其证明 AtomFS 这类基于锁耦合遍历算法的文件系统, 对于更复杂的文件系统(如 ext4)需要能够更加模块化的定义确定性事件来降低证明的负担, 如何支持通用文件系统的证明是一个有挑战的方向;最后, CRL-T 仅支持手动证明, 为了提高框架的可扩展性和自动化程度, 可以参考已有工作[39−41], 支持如验证条件生成器(verification condition generator)等自动化证明机制.

7 总 结

本工作提出了 CRL-T 框架, 其能够支持并发系统终止性的验证. 使用 CRL-T 成功地验证了 AtomFS 文件系统的终止性, 通过局部思想和偏序法解决了验证过程中遇到的挑战. 据我们所知, 这是首个验证了终止性的并发文件系统. 经验表明: 验证系统终止性的证明负担是可控的, 并希望我们的证明思想能够复用于其他系统的终止性证明中, 所有的代码将在 https://ipads.se.sjtu.edu.cn/pub/projects/atomfs 开源.

References:

[1] Cai M, Huang H, Huang J. Understanding security vulnerabilities in file systems. In: Proc. of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems. 2019. 8–15.

[2] Klein G, Elphinstone K, Heiser G, *et al*. seL4: Formal verification of an OS kernel. In: Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles. 2009. 207–220.

[3] Wang J, Zhan NJ, Feng XY, *et al*. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[4] Chen H, Ziegler D, Chajed T, *et al*. Using Crash Hoare logic for certifying the FSCQ file system. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015. 18−37.

[5] Sigurbjarnarson H, Bornholt J, Torlak E, *et al*. Push-button verification of file systems via crash refinement. In: Proc. of the 12th {USENIX} Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016. 1−16.

[6] Amani S, Hixon A, Chen Z, *et al*. Cogent: Verifying high-assurance file system implementations. ACM SIGARCH Computer Architecture News, 2016, 44(2): 175−188.

[7] Zou M, Ding H, Du D, *et al*. Using concurrent relational logic with helpers for verifying the AtomFS file system. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. 2019. 259−274.

[8] Liang H, Feng X. A program logic for concurrent objects under fair scheduling. In: Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2016. 385−399.

[9] Lu L, Arpaci-Dusseau AC, Arpaci-Dusseau RH, *et al*. A study of Linux file system evolution. In: Proc. of the 11th USENIX Conf. on File and Storage Technologies (FAST 2013). 2013. 31−44.

[10] Min C, Kashyap S, Lee B, *et al*. Cross-checking semantic correctness: The case of finding file system bugs. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015. 361−377.

[11] Lu S, Park S, Seo E, *et al*. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. 2008. 329−339.

[12] Su XH, Yu Z, Wang TT, *et al*. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015, 395(11): 93−111 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02215]

[13] Wang Y, Kelly T, Kudlur M, *et al*. Gadara: Dynamic deadlock avoidance for multithreaded programs. In: Proc. of the OSDI. 2008, 8: 281−294.

[14] Jula H, Tralamazza DM, Zamfir C, *et al*. Deadlock immunity: Enabling systems to defend against deadlocks. In: Proc. of the OSDI. 2008. 295−308.

[15] Burnim J, Jalbert N, Stergiou C, *et al*. Looper: Lightweight detection of infinite loops at runtime. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2009. 161−169.

[16] Carbin M, Misailovic S, Kling M, *et al*. Detecting and escaping infinite loops with Jolt. In: Proc. of the European Conf. on Object-oriented Programming. Berlin, Heidelberg: Springer, 2011. 609−633.

[17] Lu FM, Zheng JJ, Bao YX, *et al*. Deadlock detection of multithreaded programs based on lock-augmented segmentation graph. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1682−1700 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]

[18] Chen H, Chajed T, Konradi A, *et al*. Verifying a high-performance crash-safe file system using a tree specification. In: Proc. of the 26th Symp. on Operating Systems Principles. 2017. 270−286.

[19] Hawblitzel C, Howell J, Kapritsos M, *et al*. IronFleet: Proving practical distributed systems correct. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015. 1−17.

[20] Gu R, Shao Z, Kim J, *et al*. Certified concurrent abstraction layers. ACM SIGPLAN Notices, 2018, 53(4): 646−661.

[21] Gu R, Shao Z, Chen H, *et al*. Certikos: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th {USENIX} Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016. 653−669.

[22] Kim J, Sjöberg V, Gu R, *et al*. Safety and liveness of MCS lock—Layer by layer. In: Proc. of the Asian Symp. on Programming Languages and Systems. Cham: Springer, 2017. 273−297.

[23] Oberhauser J, Chehab RLL, Behrens D, *et al*. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. 2021. 530−545.

[24] Liang H, Feng X. Progress of concurrent objects with partial methods. Proc. of the ACM on Programming Languages, 2017, 2(POPL): 1−31.

[25] D'Osualdo E, Farzan A, Gardner P, *et al*. TaDA live: Compositional reasoning for termination of fine-grained concurrent programs. arXiv preprint arXiv: 1901.05750, 2019.

[26] Liang H, Feng X, Shao Z. Compositional verification of termination-preserving refinement of concurrent programs. In: Proc. of the Joint Meeting of the 23rd EACSL Annual Conf. on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symp. on Logic in Computer Science (LICS). 2014. 1−10.

[27] Liang H, Hoffmann J, Feng X, *et al*. Characterizing progress properties of concurrent objects via contextual refinements. In: Proc. of the Int'l Conf. on Concurrency Theory. Berlin, Heidelberg: Springer, 2013. 227−241.

[28] Leroy X. A formally verified compiler back-end. Journal of Automated Reasoning, 2009, 43(4): 363−446.

[29] Hoffmann J, Marmar M, Shao Z. Quantitative reasoning for proving lock-freedom. In: Proc. of the 28th Annual ACM/IEEE Symp. on Logic in Computer Science. IEEE, 2013. 124−133.

[30] Herlihy M, Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, 2020.

[31] Ma JB, Fu M, Feng XY. Formal verification of the message queue communication mechanism in μC/OS-II. Journal of Chinese Computer Systems, 2016, 37(6): 1179−1184 (in Chinese with English abstract).

[32] Xu FW. Design and implementation of a verification framework for preemptive OS kernels [Ph.D. Thesis]. Hefei: University of Science and Technology of China, 2016 (in Chinese with English abstract).

[33] Feng X. Local rely-guarantee reasoning. In: Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2009. 315−327.

[34] Liang HJ. Refinement verification of concurrent programs and its applications [Ph.D. Thesis]. Hefei: University of Science and Technology of China, 2014 (in Chinese with English abstract).

[35] Cao HX, Feng XY. Linearizability of a lock-free concurrent skiplist. Journal of Chinese Computer Systems, 2015, 36(6): 1158−1164 (in Chinese with English abstract).

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

[37] Ma D, Fu M, Qiao L, *et al*. Formal specification and verification of complex kernel data structures. Journal of Chinese Computer Systems, 2019, 40(2): 359−366 (in Chinese with English abstract).

[38] Gu HB, Fu M, Qiao L, *et al*. Formalization and verification of several global properties of SpaceOS. Journal of Chinese Computer Systems, 2019, 40(1): 141−148 (in Chinese with English abstract).

[39] Zhang HR, Fu M. Design and implementation of Coq tactics based on Z3. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 819−826 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5196.htm [doi: 10.13328/j.cnki.jos.005196]

[40] Schirmer N. Verification of sequential imperative programs in Isabelle/HOL [Ph.D. Thesis]. München: Technische Universität München, 2006.

[41] Sanán D, Zhao Y, Hou Z, *et al*. Csimpl: A rely-guarantee-based framework for verifying concurrent programs. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2017. 481−498.

附中文参考文献:

[3] 王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌. 软件学报, 2019, 30(1): 33−61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[12] 苏小红, 禹振, 王甜甜, 等. 并发缺陷暴露、检测与规避研究综述. 计算机学报, 2015, 395(11): 93−111. [doi: 10.11897/SP.J.1016.2015.02215]

[17] 鲁法明, 郑佳静, 包云霞, 等. 基于锁增广分段图的多线程程序死锁检测. 软件学报, 2021, 32(6): 1682−1700. http://www.jos.org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]

[31] 马杰波, 付明, 冯新宇. μC/OS-II 中消息队列通信机制的形式化验证. 小型微型计算机系统, 2016, 37(6): 1179−1184.

[32] 许峰唯. 抢占式操作系统内核验证框架的设计和实现 [博士学位论文]. 合肥: 中国科学技术大学, 2016.

[34] 梁红瑾. 并发程序精化验证及其应用 [博士学位论文]. 合肥: 中国科学技术大学, 2014.

[35] 曹红星, 冯新宇. 一种无锁并发跳表算法的可线性化证明. 小型微型计算机系统, 2015, 36(6): 1158−1164.

[37] 马顶, 付明, 乔磊, 等. 复杂内核数据结构的形式化描述和验证. 小型微型计算机系统, 2019, 40(2): 359−366.

[38] 顾海博, 付明, 乔磊, 等. SpaceOS 中若干全局性质的形式化描述和验证. 小型微型计算机系统, 2019, 40(1): 141−148.

[39] 张恒若, 付明. 基于 Z3 的 Coq 自动证明策略的设计和实现. 软件学报, 2017, 28(4): 819−826. http://www.jos.org.cn/1000-9825/5196.htm [doi: 10.13328/j.cnki.jos.005196]

邹沫(1994－), 男, 博士生, 主要研究领域为操作系统, 形式化验证.

谢昊彤(1999－), 男, 主要研究领域为操作系统, 形式化验证.

魏卓然(1998－), 男, 主要研究领域为操作系统, 形式化验证.

陈海波(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统.