软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2167−2201 [doi: 10.13328/j.cnki.jos.007580] [CSTR: 32375.14.jos.007580]

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

分布式系统模型检验技术研究进展*

唐瑞泽, 黄 宇, 欧阳凌志, 程 潜, 张宇奇, 马晓星

(计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

通信作者: 黄宇, E-mail: yuhuang@nju.edu.cn

摘 要: 分布式系统作为现代计算基础设施的核心, 其正确性至关重要. 然而, 由于分布式系统所处的计算环境中的高度不确定性以及代码设计与实现的复杂性, 验证分布式系统的正确性始终面临巨大挑战. 分布式系统模型检验 (DMCK) 技术通过代码级的穷尽式状态探索, 能够发现深层缺陷, 在真实系统中确定性重现缺陷并验证修复正确性, 有效应对了分布式系统缺陷“难发现、难诊断、难修复”等典型难题. 系统性梳理了 DMCK 的研究进展, 围绕“状态爆炸”与“人工成本”的权衡, 归纳其发展脉络为 3 个阶段: 第 1 阶段聚焦于使 DMCK 有效的代码级确定性模拟执行与状态空间探索技术; 第 2 阶段通过引入少量人工建模以利用系统语义信息缓解状态爆炸问题; 第 3 阶段致力于增强模型层与代码层的交互能力以进一步提升代码级模型检验效率. 最后, 在总结既有工作的基础上, 探讨了目前 DMCK 的局限和未来可能的发展方向.

关键词: 分布式系统; 模型检验; 正确性; 缺陷发现; 状态空间爆炸; 形式化规约

中图法分类号: TP311

中文引用格式: 唐瑞泽,黄宇,欧阳凌志,程潜,张宇奇,马晓星.分布式系统模型检验技术研究进展.软件学报,2026,37(5):2167–2201. http://www.jos.org.cn/1000-9825/7580.htm

英文引用格式: Tang RZ, Huang Y, Ouyang LZ, Cheng Q, Zhang YQ, Ma XX. Research Progress on Distributed System Model Checking Technologies. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2167–2201 (in Chinese). http://www.jos.org.cn/1000-9825/7580.htm

Research Progress on Distributed System Model Checking Technologies

TANG Rui-Ze, HUANG Yu, OUYANG Ling-Zhi, CHENG Qian, ZHANG Yu-Qi, MA Xiao-Xing

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Distributed systems serve as the core of modern computing infrastructure, making their correctness essential. However, the high nondeterminism in the computing environment of distributed systems, combined with the complexity of code design and implementation, makes the correctness verification of distributed systems a significant challenge. Distributed system model checking (DMCK) enables the discovery of deep bugs, deterministic reproduction of bugs in real systems, and repair correctness verification by exhaustive code-level state exploration, thereby addressing the typical problems of distributed systems, such as difficult discovery, diagnosis, and repair. This study provides a systematic summary of the research progress in DMCK. Centering around the trade-off between “state explosion” and “manual effort”, it categorizes the development of DMCK into three stages. The first stage focuses on deterministic simulation execution and state space exploration technologies that make DMCK effective. The second stage introduces a small amount of artificial modeling to leverage system semantics for alleviating state explosion, and the third stage aims to enhance the interaction between the model layer and code layer to improve code-level model checking efficiency. Finally, based on the summary of existing work, this study discusses the current limitations of DMCK and promising development directions in the future.

Key words: distributed system; model checking; correctness; bug detection; state space explosion; formal specification

---

* 基金项目: 国家杰出青年科学基金 (62025202); 国家自然科学基金 (62372222); CCF-华为胡杨林基金形式化专项 (CCF-Huawei FM202505)
收稿时间: 2025-04-18; 修改时间: 2025-06-16; 采用时间: 2025-12-02; jos 在线出版时间: 2026-01-28
CNKI 网络首发时间: 2026-01-29

1 引 言

随着互联网、大数据和云计算的快速发展, 分布式系统已在诸多领域获得大规模的部署和应用[1−3]. 分布式系统由一组通过网络通信的计算节点组成, 共同完成数据存储、处理和计算等任务, 并向外提供服务. 这些系统旨在实现高性能处理、负载均衡、高可用性和容错性等需求, 是支撑现代计算领域的基础设施. 例如, 分布式数据库[2,4]、分布式协调服务[5,6]以及大数据处理平台[1,7]等, 广泛应用于电子商务、云计算和大数据处理等领域.

然而, 正确实现分布式系统面临诸多挑战, 这体现在分布式系统所处的复杂计算环境和分布式系统自身复杂的代码实现中. 计算环境中存在高度的不确定性, 它主要由网络和外设交互的异步性、并发性和易错性引起, 例如, 网络消息延迟或乱序到达、异步磁盘读写、时钟超时、高度并发的客户端请求、节点宕机和网络分区故障等. 为了应对计算环境带来的不确定性, 并向上层应用提供可理解、易编程的抽象, 分布式系统通常需要精巧的协议设计 (例如 Paxos[8]、Raft[9]和 Zab[6,10]) 和复杂的系统实现. 这种复杂性使得系统更容易引入细微但致命的缺陷, 而这些缺陷往往只在由计算环境的不确定性触发的极端调度条件下才会显现, 因此在测试环境中难以发现、定位与修复. 这类依赖特定调度、触发路径极深的问题被称为深层缺陷[11]. 尽管在测试阶段难以暴露, 深层缺陷却可能在分布式系统大规模部署和长时间运行中被触发.

分布式系统中的深层缺陷将导致数据丢失、数据不一致和服务不可用等严重后果. 由于分布式系统支撑着众多重要系统的运行, 任何一个细微的缺陷可能带来巨大的损失. 这些年来, 即便是成熟的工业级系统也时常出错, 并造成巨大损失. 例如, 2019 年, 美国 Google 公司由于存储服务负载均衡存在缺陷, 导致配置变更时引发级联故障, 影响了包括北美和南美在内的多个地区, 造成谷歌服务中断超过 3 h[12]. 此外, 阿里云在 2022–2023 年期间发生了两次重大故障, 分别导致香港地区服务中断超过 15 h、全球核心业务宕机超过 3 h, 进而影响了阿里系多个核心产品 (如淘宝、钉钉等) 的正常使用[13,14].

提升分布式系统正确性的方法主要包括测试和形式化验证. 两者都能发现系统中的缺陷, 但关键的区别在于是否能提供系统正确性保证. 决定是否采用测试或形式化验证的关键因素在于技术的使用成本. 测试技术成本较低, 实用性较强, 能够发现大部分浅层缺陷. 然而, 由于分布式系统的复杂性和不确定性, 依赖测试人员和开发者编写的单元测试和集成测试只能覆盖少量的执行路径. 自动化程度更高的随机测试或压力测试 (如 Jepsen[15]、Chaos[16]和 Fuzzing[17]) 能探索更多的系统执行路径, 但它们无法提供状态空间的覆盖保证, 也难以在后续测试中确定性地重现已经发现的缺陷, 这使得分布式系统中的深层缺陷的诊断和修复变得异常困难.

形式化验证技术相较于测试成本更高, 它采用数学的方法对系统的正确性进行证明, 有效解决了测试中缺陷发现难、复现难和修复难等问题, 主要包括模型检验 (model checking)[18]和定理证明 (theorem proving)[19]两种方式. 定理证明要求使用者利用程序逻辑 (如霍尔逻辑) 进行数学证明, 并借助工具 (如 Coq[20]、Isabelle/HOL[21]) 自动化或半自动化地检查数学命题是否成立. 尽管这种方法提供全面的正确性保证, 但它的使用门槛较高, 且学习曲线陡峭, 限制了它在工业界的广泛应用.

模型检验通过自动化地对系统模型进行状态空间的系统性穷尽式探索, 确保所有可能的状态都得到验证, 其穷尽式的验证方式特别适合应对分布式系统中由不确定性引发的“缺陷难发现、难诊断、难修复”的挑战. 不过, 由于状态空间往往呈指数级增长, 模型检验仍面临严重的状态爆炸 (state explosion) 问题. 近年来, 研究者提出了对称约减、偏序约减等多种优化技术, 配合工具链的不断成熟和计算能力的提升, 有效缓解了状态爆炸问题, 显著提升了模型检验的实用性, 其适用范围也在持续拓展, 尤其在工业场景中正逐步落地应用[22−26].

1.1 分布式模型检验技术概述

模型检验具备高度自动化、严格正确性保障等优点, 但传统模型检验技术主要针对系统的抽象模型而非系统的代码实现展开验证, 面临系统模型与真实代码之间的语义鸿沟 (semantic gap) 问题, 验证结果受到转译错误 (transcription error) 的制约, 仅针对模型的验证结果通常无法准确反映真实系统的情况. 分布式系统模型检验

(distributed system model checking, DMCK) 是在软件模型检验 (software model checking)[27]基础上发展而来的一个子方向, 该术语最早由 SAMC[11]提出, 专注于验证分布式系统的真实代码. 它结合了两个关键要素: 代码级模型检验和面向分布式系统的针对性适配.

首先, 代码级模型检验在系统的真实代码上进行状态空间探索, 而非仅对系统的抽象模型进行验证. 这一方式解决了传统模型检验中模型与代码之间存在的语义鸿沟问题, 能够更有效地发现实际系统中的缺陷, 且验证结果直接反映到代码. 然而, 它仍面临如何在代码层控制分布式系统固有的不确定性, 以系统性覆盖所有可能执行路径的挑战.

其次, 分布式系统特有的不确定性进一步加剧了代码级模型检验中的状态爆炸问题, 主要体现在两个维度: 1) 系统实现的复杂性: 为满足容错需求, 分布式系统通常采用复杂的协议设计 (如 Paxos[8]、Raft[9]和 Zab[6,10]), 并引入多线程、磁盘异步读写、超时等实现机制, 导致状态空间极其庞大, 潜在缺陷更难被覆盖; 2) 计算环境的不确定性: 分布式系统运行在充满异步、并发和错误事件的环境中, 例如消息延迟与乱序、节点宕机、时钟漂移、用户请求等都可能引发大量不确定行为, 进一步加剧了状态空间爆炸问题.

因此, DMCK 在带来面向分布式系统的代码级模型检验能力的同时, 面临着如何控制环境与系统行为的不确定性、如何缓解状态爆炸等关键挑战. 本文聚焦于真实分布式系统代码的模型检验技术, 梳理了 DMCK 从早期原型工具到具备工业实用性的演进脉络. 在这一过程中, 研究者不断应对前述挑战, 解决了如何控制不确定性和如何有效高效进行状态探索的问题. 通过引入系统语义信息, 降低状态爆炸程度. 通过代码层与模型层的互动, 进一步提升了分布式系统模型检验的效率. 这些技术在保证代码级验证能力的同时, 有效提升了 DMCK 的效率与实用性.

1.2 相关综述工作与本综述贡献

就我们前期调研的文献来看, 目前尚无专门针对 DMCK 技术的综述, 但是国内外有一些与之相关的优秀综述. 国内研究者对形式化方法的研究进展进行了非常详尽的综述[28], 涵盖了形式化方法各个方面的研究现状, 但该综述更关注于形式化验证技术本身, 而较少关注于分布式系统代码级模型检验技术. Godefroid 等人[29]认为模型检验提供的保证是有限的, 在实际使用中是一种发现软件缺陷的“超级测试”, 因此也将软件模型检验称为系统性测试 (systematic testing). 该综述主要梳理了并发系统和串行系统的相关研究, 在并发系统方面主要针对多线程并发, 而对分布式系统的探讨较少. 国内研究者对分布式系统的动态测试技术进行了详尽的综述, 从不同类型系统缺陷的角度介绍了典型的分布式系统动态测试工具[30]. 其中部分 DMCK 研究被归入鲁棒性缺陷测试工具. 然而, 该综述主要关注测试技术, 对 DMCK 相关工作的讨论较少. 国内研究者对分布式共识协议的形式化验证进行了系统性综述[31], 涵盖模型检验与定理证明等方法, 但主要聚焦于协议设计层, 未涉及系统实现层的技术进展. 国内研究者对模型检验中的状态爆炸问题进行了系统性综述[32], 其中总结的多种缓解技术已被广泛应用于 DMCK 相关工作中. 此外, 国内研究者还有专门针对交互式定理证明的并发程序验证工作的综述[33], 详细整理了并发程序形式化验证中定理证明方向的研究进展, 但未系统性梳理模型检验方向的进展.

我们观察到, DMCK 的发展受到传统模型检验技术演进的影响, 其核心挑战在于模型检验的效率与模型检验的人工成本之间的权衡. 这一权衡推动了 DMCK 技术的演进. 围绕上述基本权衡, 本文梳理了 DMCK 的 3 个发展阶段. 第 1 阶段的 DMCK 研究解决了无需人工建模即可直接检验分布式系统实现的可行性问题. 第 2 阶段为缓解状态爆炸问题, 逐步引入被验证系统的语义知识. 近年来, 面向分布式系统协议与设计的传统模型检验技术, 在实用性与易用性方面有了长足的进步[34,35]. 第 3 阶段 DMCK 开始寻求与传统模型检验进行有机互动, 以进一步提升代码级模型检验的效率. 上述发展过程形成了本文的主线.

本文第 2 节介绍综述的框架. 第 3 节描述传统 DMCK 技术, 这一阶段的核心在于实现代码层模型检验的核心使能技术与状态爆炸缓解技术. 第 4 节介绍语义感知 DMCK 技术, 该阶段引入少量人工建模以利用系统语义信息进一步缓解状态爆炸问题. 第 5 节讨论模型层与代码层互动的 DMCK 技术, 具体介绍分布式系统形式化建模技术, 以及模型与代码之间的一致性比对技术. 第 6 节探讨 DMCK 派生出的新型测试和验证技术. 最后, 第 7 节对综述进行总结并展望未来研究方向.

2 分布式系统模型检验技术综述框架

传统模型检验技术通过穷尽式搜索系统的状态空间, 具备两大核心优势: 1) 严格的正确性保障: 能够对系统的所有可能状态进行穷举验证; 2) 高度自动化: 用户只需提供系统的模型和正确性规约, 即可通过“push-button”式的方式自动完成验证. 然而, 该方法存在显著不足. 首先, 模型与实际系统之间存在语义鸿沟, 模型的正确性并不意味着系统实现正确. 其次, 状态爆炸问题严重, 使得其正确性保障通常需在一定限制条件下才能成立. 这些问题成为模型检验技术在软件系统中实际应用的主要障碍.

分布式系统模型检验技术 DMCK 是针对分布式系统的代码级模型检验, 对代码所有可能执行路径对应的状态空间进行系统性、穷尽式探索. DMCK 从根本上解决了抽象模型与具体实现之间的语义鸿沟问题, 避免了模型检验中的转译错误 (transcription error), 并保留了传统模型检验技术的高度自动化优势. 然而, 由于代码实现包含复杂具体细节 (例如多线程调度、磁盘异步读写、超时等机制) 以及分布式系统环境高度不确定性 (例如消息延迟与乱序、节点宕机、用户请求), DMCK 面临比传统模型检验更严重的状态空间爆炸问题. DMCK 的状态空间爆炸不仅体现在状态数量的指数级增长, 还体现在探索速度的显著降低. 由于 DMCK 需要实际执行代码, 其状态空间探索的开销远高于传统模型检验在抽象模型上的状态搜索. 各种 DMCK 状态空间探索优化技术被提出来缓解状态爆炸问题.

早期研究为了让 DMCK 具备可行性, 研究的技术主要包括控制分布式系统中的不确定性的确定性模拟执行技术 (deterministic simulation)、系统性探索状态空间的搜索算法和通用的状态空间约减算法. 通过这些技术, DMCK 工具去除了传统模型检验中的建模成本, 验证结果直接反映到代码中. 这些技术不仅奠定了 DMCK 的基本能力, 也为后续研究奠定了基础.

由于 DMCK 技术面临的状态爆炸问题更为严重, 所以通用的优化技术迅速遇到瓶颈. 此时的突破口在于利用特定系统的特定语义知识, 例如系统事件的独立性和对称性, 针对性地、垂直性地进行约减状态空间. 语义感知的优化对于效率的提升, 是以验证技术的通用性与自动化程度为代价的. 但是针对分布式系统正确性保障的迫切需求, 面对极其严重的状态爆炸问题, 适当牺牲通用性与自动化程度并引入一定人工建模成本是一种有效、高效的权衡.

传统模型检验技术在 DMCK 发展过程中也在进步, 尤其是轻量级建模语言和验证工具链的成熟, 使得复杂的分布式系统设计及优化得到较低成本验证和较高正确性保障, 这些进步推动了模型检验技术在工业界的广泛应用[22−26]. 这一背景下, 模型层与代码层互动的模型检验技术被提出, 该技术通过对二者状态空间的比对, 建立模型与代码的映射, 使得模型检验结果可反映到代码, 大幅提升了 DMCK 的状态探索效率、有效性和性价比. 从 DMCK 的发展脉络角度来看, 与模型层互动是语义感知优化的更深和更广的应用方式.

本文根据 DMCK 的发展脉络, 将 DMCK 的发展分为 3 个阶段, 如图 1 所示. 这些阶段的发展体现了平衡 DMCK 的自动化程度和状态空间探索效率之间的矛盾的过程, 逐步增加了人工成本但提升了性价比. 下面我们将遵循这一脉络, 分别介绍具体技术.

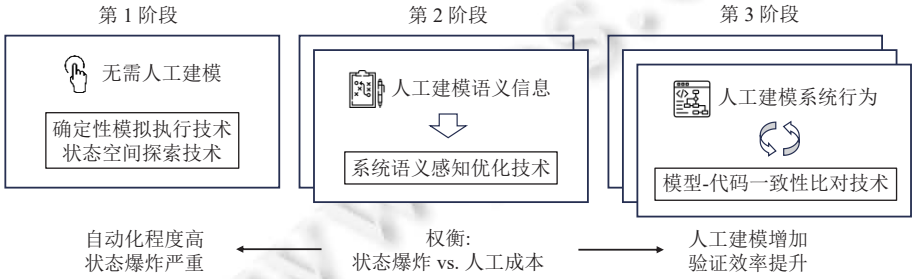


图 1 DMCK 技术发展阶段划分示意图

3 传统分布式系统模型检验

尽管传统模型检验技术在分布式协议验证方面取得了显著成功[36,37], 但应用于分布式系统实现时, 面临着形

式化建模成本和模型与真实系统偏差等问题. 这促使研究者探索直接以代码作为模型的验证方法, 从而避免建模成本和模型偏差. 本节主要介绍 DMCK 发展的第 1 阶段, 该阶段的研究重点在于使 DMCK 具备可行性, 即无需人工建模即可进行模型检验, 且具有有效性和通用性, 我们将其归类为传统分布式系统模型检验技术.

传统模型检验技术通常基于抽象形式模型, 通过特定的检验器进行分析, 从形式化语言设计到支撑工具链实现, 全流程支持对状态空间的可控且高效的穷尽式探索. 然而, 当模型检验技术应用到真实代码时, 面临两大关键挑战: 1) 分布式系统中存在来自节点间通信与节点内部执行环境的高度不确定性, 而主流编程语言与运行时环境并未为其提供可控穷尽式探索的支持. 为了系统性地枚举所有可能的执行路径, DMCK 需解决如何对系统所面临的各种不确定性进行确定性模拟的问题; 2) 分布式系统中复杂的代码实现与环境不确定性共同加剧了状态空间爆炸问题, DMCK 需解决如何高效探索状态空间并执行属性验证的问题. 后续各节将围绕上述挑战, 分别介绍确定性模拟执行、状态空间探索与属性验证等关键技术. 图 2 展示了传统 DMCK 框架的整体架构, 分为被控节点与 DMCK 后端引擎两部分, 图中展示了各关键组件及其交互关系. 其中, 被控节点内的图标分别表示线程、时钟和网络消息包, 由不确定性截获层进行控制.

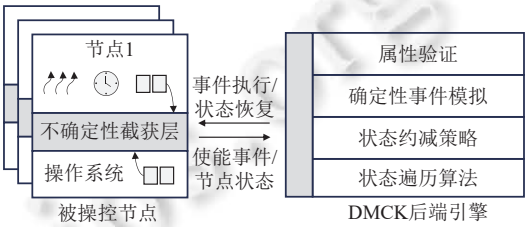


图 2 传统 DMCK 整体架构图

3.1 确定性模拟执行技术

保障分布式系统正确性的根本挑战在于系统行为的高度不确定性, 主要源于网络通信和外设交互的异步性、并发性以及运行环境的易错性. 这些因素导致相同输入可能产生大量不同的执行路径, 使得深层缺陷难以触发与发现. DMCK 技术之所以适用于分布式系统, 正是因为其核心原理“系统性穷尽探索”能够有效应对这种不确定性, 通过对执行过程中的不确定性事件进行控制, 在探索算法的调度下全面覆盖可能的路径.

支撑 DMCK 的首要使能技术是确定性模拟执行 (deterministic simulation). 该技术通过精确控制消息传递、线程调度与错误注入等关键事件, 将不确定行为转化为可预测、可控的执行过程, 从而支持状态空间的系统性遍历. 具体而言, 确定性模拟执行需支持事件的拦截与调度、状态的保存与恢复, 以便探索过程可在任意状态下回溯与分叉. 借助这一机制, DMCK 能在可控环境中系统地重现复杂路径, 发现隐藏在边界调度下的深层缺陷.

3.1.1 不确定的环境与确定性的受控执行

为实现这一目标, 首先需要明确分布式系统中的不确定性来源, 并通过特定的劫持技术加以操控. 分布式系统运行于开放、动态、难控的环境, 不确定性主要来自节点间分布式环境的不确定性和节点内部环境的不确定性[38,39]. 前者主要包括消息的异步到达、错误事件、客户端请求等, 后者主要包括线程调度、资源访问、随机数等. 由于分布式系统往往具有明显的事件驱动特性, 这些环境的不确定性会触发系统中特定事件的执行. 参考 DeMeter[40], 我们将导致节点间分布式环境不确定性的事件称为全局事件 (global event), 而导致节点内部环境不确定性的事件称为局部事件 (local event). 由于分布式系统及其核心协议强调节点间分布式环境中的协同交互, 大多数 DMCK 研究更关注全局事件的受控执行.

全局事件指影响节点间分布式环境的不确定性事件. 在分布式系统中, 这些事件的处理通常涉及系统实现的核心分布式协议设计, 决定了系统的协同机制和一致性保障. 主要包括以下几类.

- 消息事件: 由于网络延迟, 服务器节点可能在任意时刻接收消息, 而消息的处理可能进一步触发新的消息发送, 增加系统执行的不确定性.

● 错误事件: 包括节点宕机、重启、网络分区、消息丢失、重复或乱序等. 错误事件是分布式系统的固有特性, 影响节点间的网络状态, 并可能进一步改变整个系统的执行路径, 尤其是在边缘情况下.

● 客户端请求/工作负载: 客户端可以在任意时刻发送请求至服务器, 影响系统状态.

在上述全局事件中, 超时机制在多个事件的判定与触发中发挥关键作用. 分布式系统常通过超时检测网络异常或节点故障 (如网络分区、节点宕机), 进而触发重试、错误处理等操作, 例如定期发送保活消息并在超时未收到响应时认定节点失联. 此外, 由于系统时钟漂移 (clock drift)[41]和定时器调度的不确定性, 超时事件的触发时刻本身具有不确定性, 进一步加剧了代码级状态空间的复杂性.

局部事件指影响节点内部执行环境的不确定性, 通常涉及线程调度与资源访问, 并取决于系统的代码实现方式. 主要包括以下几类.

● 线程调度: 在多线程并发执行时, 线程可能在任意执行点被抢占或交替运行, 不同的调度顺序可能改变共享数据结构的访问顺序, 从而影响系统行为. 例如, 持久化线程与消息处理线程的调度顺序不同, 可能导致节点状态的差异.

● 资源访问: 某些资源访问相关的系统调用可能受环境因素影响, 尤其是磁盘 I/O 操作, 其执行结果可能成功或失败, 从而引入不确定性.

● 随机数: 系统中若通过随机数控制执行逻辑 (如随机化超时), 将引入不确定性. 随机数的生成依赖操作系统提供的种子, 因此受环境影响.

此外, 未定义行为和弱内存模型等节点内部环境的不确定性通常不属于 DMCK 的关注范围, 研究中常通过简化假设加以规避, 例如默认不存在未定义行为, 并采用强一致性内存模型.

为了实现这些事件的确定性模拟执行, 系统需要能够从一个状态精确地转移到另一个状态, 并具备回溯能力, 以便探索不同的执行路径. 根据状态保存和恢复的方式, 现有技术主要分为两类: 基于状态的方法通过存储和恢复内存信息来回溯系统状态, 而基于事件的方法则依赖于事件序列的重新执行来恢复先前状态.

不同的研究工作由于研究目标不同, 在技术选型、技术特性、全局事件与局部事件的支持能力、实现方式以及被测系统上各有差异. 表 1 对比了实现确定性模拟执行技术的典型工具, 其中, “技术特性”主要考察以下 3 个方面: 易扩展性 (extensibility), 即是否能在不修改 DMCK 工具自身的情况下, 轻松引入新的事件类型; 可适用性 (adoptability), 即是否能广泛适用于不同类型的分布式系统; 透明性 (transparency), 即 DMCK 工具是否能在无需大幅修改被测系统的前提下, 直接验证新的目标系统.

表 1 确定性模拟执行技术典型工具的支持能力、实现技术及应用范围

工具	发表会议及年份	技术实现分类	技术特性	全局事件支持能力	局部事件支持能力	事件操控方式	被测系统
CMC	OSDI 2002[42], NSDI 2004[43]	基于状态	易扩展、可适用	支持所有类型	部分支持 (仅资源访问)	手动适配被测系统到CMC运行时环境	Linux等系统实现的AODV、TCP协议
Mace/MaceMC	PLDI 2007[44], NSDI 2007[45]	基于状态/事件混合	透明、易扩展	支持所有类型	部分支持 (仅线程调度)	自动编译到Mace语言运行时环境	Mace语言编写的程序: Chord、RandTree等
McErlang[46]	ICFP 2007[47]	基于状态	易扩展	支持所有类型	未支持	自动翻译到McErlang运行时接口	Erlang语言 (子集) 实现的分布式协议: Chord、Leader Election
Chess	OSDI 2008[48]	基于事件	透明、可适用	部分支持 (仅支持消息事件)	支持所有类型 (主要面向线程调度)	透明截获操作系统 (Windows) 调度相关API	主要面向多线程程序: PLINQ、CDS, 也可检查分布式系统: Dryad
MoDist	NSDI 2009[49]	基于事件	透明、可适用	支持所有类型	支持所有类型	透明截获应用程序与操作系统 (Windows) 的交互, 错误注入	Berkeley DB、MPS、PacificA
CrystalBall	NSDI 2009[50]	基于状态	透明、易扩展	支持所有类型	部分支持 (仅线程调度)	自动编译到CrystalBall运行时环境 (基于Mace)	Mace语言编写的程序: RandTree、BulletPrime、Paxos和Chord

表 1 确定性模拟执行技术典型工具的支持能力、实现技术及应用范围 (续)

工具	发表会议及年份	技术实现分类	技术特性	全局事件支持能力	局部事件支持能力	事件操控方式	被测系统
Basset[51]	ASE 2009[52]	基于状态	透明、易扩展	部分支持 (消息事件和客户端操作)	部分支持 (仅支持线程调度)	在Basset运行时 (基于JPF) 运行Java程序	基于Scala语言或Actor-Foundry库实现的演员 (actor) 编程模型程序
ISP[53]	PPoPP 2009[54]	基于事件	透明、可适用	部分支持 (仅支持消息事件)	未支持	透明截获了MPI编程接口	使用MPI接口通信的程序: ParMETIS、MADRE
dBug[55]	SSV 2010[56]	基于事件	易扩展、可适用	支持所有类型	部分支持 (仅资源访问和线程调度)	为被测系统手动插桩/适配dBug接口	C++语言实现的分布式系统: PVFS、FAWN-KV
DeMeter	SOSP 2011[40]	基于事件	取决于后端	支持所有类型	支持所有类型	依赖于后端 (MoDist或Mace)	包括MoDist和Mace论文中的被测系统
LMC	NSDI 2011[57]	基于状态	透明、易扩展	支持所有类型	部分支持 (仅线程调度)	自动编译到LMC运行时环境 (基于MaceMC和CrystalBall)	Mace语言编写的Paxos和1Paoxs
MP-Basset[58]	DSN 2011[59]	基于状态	透明、易扩展	支持所有类型	部分支持 (仅线程调度)	在MP-Basset运行时 (基于Basset) 运行MP语言编写的程序	MP语言编写的程序/协议: Paxos、Multicast等
DBSS[58]	SRDS 2013[60]	基于状态	透明、易扩展	支持所有类型	部分支持 (仅线程调度)	基于MP-Basset运行时	MP语言编写的程序/协议: Paxos、Zab等
net-iocache	ASE 2013[61], TSE 2014[62],	基于状态	透明	部分支持 (消息事件和客户端操作)	部分支持 (仅线程调度)	运行于net-iocache扩展JPF运行时	Java语言编写的client/server架构的程序: HTTP server等
Concuerror[63]	ICST 2013[64]	基于事件	透明、可适用	部分支持 (仅支持消息事件)	部分支持 (仅线程调度)	自动对Erlang代码插桩	主要面向Erlang编写的演员 (actor) 模型程序
SAMC[65]	OSDI 2014[11]	基于事件	易扩展、可适用	支持所有类型	未支持	基于AspectJ手动插桩Java代码, 错误注入	ZooKeeper、Hadoop、Cassandra
FlyMC[66]	EuroSys 2019[67]	基于事件	易扩展、可适用	支持所有类型	未支持	手动插桩 (Java, C++), 错误注入	ZooKeeper、Hadoop、Cassandra、Spark、LogCabin、Kudu
DSLabs[68]	EuroSys 2019[69]	基于状态	透明、易扩展	支持所有类型	未支持 (被测系统是单线程的)	自动编译到DSLabs运行时	学生基于DSLabs接口实现的分布式系统/协议: 键值存储、Paxos等
SandTable[70]	EuroSys 2024[71]	基于事件	透明、可适用	支持所有类型	未支持	透明截获应用程序与操作系统 (POSIX) 的交互, 错误注入	实现了关键分布式协议的系统: ZooKeeper、Xraft-KV、RedisRaft等
Remix[72]	EuroSys 2025[73]	基于事件	易扩展、可适用	支持所有类型	支持所有类型	基于AspectJ手动插桩Java代码, 错误注入	实现了关键分布式协议的系统: ZooKeeper

3.1.2 基于状态的确定性模拟执行技术

基于状态的确定性模拟执行技术遵循传统模型检验的思路, 将分布式系统建模为状态机, 其中状态和事件是核心概念. 每个事件触发状态转移, 系统完整地记录当前状态, 并据此判断可调度的下一个事件. 调度器通过状态比对识别已探索的状态, 避免重复执行. 在状态回溯时, 直接恢复先前存储的状态. CMC[42,43]、Mace[44,45]、基于Java PathFinder (JPF)[74]的Basset[52]等一系列典型研究均采用此方法, 并提供运行时环境来执行事件及管理状态的保存与恢复. 由于分布式系统普遍采用事件驱动机制, 这些工作通常通过一套接口执行事件触发的代码, 但适配代码到接口的方式有所不同, 主要分为两类.

● 手动移植现有系统的事件处理函数至运行时[42,43]: 这种方式透明性较差, 适配新系统需要大量修改被测代码, 但适应性较强.

● 设计框架接口, 使符合框架规范的程序可直接运行于运行时[45,47,52]: 这种方式适用性较差, 难以直接应用于已有系统, 但透明性较强.

状态的保存与恢复主要涉及网络状态与进程状态.

● 网络状态: 包含暂存但尚未送达的网络消息, 这些消息需要缓存, 以便调度器按照不同顺序调度, 从而触发消息处理事件和模拟网络故障事件 (如消息丢失、乱序、重复、网络分区等).

● 进程状态: 可采用低层次的内存字节流方式编码保存, 或高层次的变量键值对方式编码保存. 对于手动移植代码的方式, 完整变量值难以直接获取, 通常通过操作系统底层提取堆、栈、静态变量等内存字节流进行存储. 对于框架化编写的代码, 由于变量受框架管理, 可直接存储变量键值对. 此外, 操控进程状态可用于模拟节点故障、重启等错误事件.

CMC[42]观察到分布式/网络系统的核心协议实现 (如 AODV 路由协议) 通常采用事件驱动风格, 因此直接提取原始系统中的核心协议事件执行代码, 并适配到 CMC 的虚拟运行时环境. 该环境提供最小化的虚拟操作系统接口, 如时间管理、网络收发等系统调用, 以支持移植代码的执行. CMC 将网络消息状态与所有节点的内存字节流状态整合为一个整体, 视为整个系统的状态, 并在事件执行时同步恢复网络与节点内存状态. CMC 发现了包括 Linux 在内的重要系统中 AODV 实现的多个缺陷, 并发现了 AODV 协议的深层设计问题, 首次展现了模型检验在复杂分布式协议代码级验证上的有效性. CMC 后续工作[43]进一步用于 Linux TCP 网络协议的验证, 过程中发现手动提取和适配复杂代码容易引入难以调试的假阳性缺陷. 最终, 研究者选择直接在 CMC 虚拟执行环境中运行修改版的用户态 Linux 内核, 并成功发现了 4 个缺陷. CMC 在真实复杂分布式代码验证中展现了有效性, 但该方法存在两个主要挑战: 1) 手动适配过程工作量大且易出错; 2) 状态保存和恢复的精确控制困难, 细微的内存比特变化 (如堆区动态分配顺序) 可能导致逻辑等价状态被误判为不同状态. 为减少代码剥离原始环境带来的问题, CMC 后续采用直接运行 Linux 的方式, 但适配 Linux 仍需大量工作量. 此外, CMC 通过状态标准化 (state canonicalization) 合并等价状态, 但该过程仍依赖大量人工干预.

JPF[74]是 NASA 开发的 Java 语言模型检验工具, 提供一个可控制不确定性和支持状态回溯的 Java 虚拟机. JPF 主要操控线程调度等局部事件, 不支持 DMCK 所需的全局事件 (如网络收发和错误事件). 它维护了线程调用栈、静态与动态对象的状态信息, 从而支持状态比对和去重. JPF 的虚拟机工作方式类似于 CMC 提供的虚拟运行时环境, 由于 Java 程序本身运行于虚拟机, JPF 避免了 CMC 在适配新系统时面临的大量人工适配和状态标准化问题. 扩展 JPF 以支持 (部分或全部) 全局事件的工具包括如下 3 种.

● Basset[52]: 提供了一个框架, 支持基于 Scala 或 ActorFoundry Java 库的 Actor 计算模型, 使得 Actor 之间的消息交互可被确定性操控. Basset 主要增加了对网络消息调度的控制, 但仍不支持错误事件的操控.

● MP-Basset[59]: 在 Basset 基础上扩展, 增强了对基于消息传递 (message passing) 的容错分布式协议 (如 Paxos) 的检验能力, 支持节点失效和网络错误模拟. 该研究更侧重高效状态探索方法, 错误模拟简化为不调度特定节点. 其后续研究 LPOR[75]和 DBSS[60]进一步优化了状态探索策略.

● net-iocache[61]: 提供了一个 JPF 插件, 使得 JPF 能够处理单线程管理多个非阻塞网络连接的情况. 该工具通过缓存网络通信历史, 使 JPF 在状态回溯时能够重用历史数据. 后续研究[62]进一步扩展了其适用范围. 然而, net-iocache 及后续工作不支持分布式系统中的典型错误场景.

Mace 语言框架[44]源自 Macedon 原型[76], 扩展了 C++ 以提供分布式系统编程的语言级支持. Mace 采用状态-事件-转移 (state-event-transition) 模型, 将分布式系统建模为状态机, 并提供统一框架支持代码事件、网络事件和错误事件的操控. 开发者通过 Mace 框架定义状态变量和状态转移, 每个事件被原子执行 (相当于控制了节点内线程调度), 内置模型检验工具在事件执行后记录节点变量和网络消息队列的完整状态, 以支持状态探索. CrystalBall[50]基于 Mace, 主要用于深度在线调试 (deep online debugging) 和执行操控 (execution steering), 可预测并规避可能导致不一致的执行路径. 由于 CrystalBall 针对已部署的运行中分布式系统, 其面临无法高效获取全局一致快照的问题. 为此, CrystalBall 采用局部快照方案, 每个节点仅采集与其直接通信的邻居节点状态, 并通过独立线程执行模型检验, 向运行时反馈需要规避的路径. 采集快照时, CrystalBall 使用 fork 技术复制节点的虚拟内存. CrystalBall

的后续研究 LMC[57]主要聚焦于优化状态探索, 将网络状态与节点状态分离, 使网络历史消息在全局共享且不被删除, 仅依赖节点状态进行状态等价判定. 这种方式减少了一定深度内的状态数量 (但在更深层次可能导致状态空间急剧膨胀), 加快了在线模型检验在限定时间内的深度探索.

McErlang[47]是一个针对 Erlang 程序的模型检验工具, 专门用于分布式系统协议的验证. 它提供了一套运行时 API, 替换了标准 Erlang 运行时系统中与分布式、并发和通信相关的部分. McErlang 自动将用户提供的 Erlang 协议代码翻译为使用该运行时 API 的形式, 并结合用户提供的环境代码进行模型检验, 后者可用于定义节点故障、网络错误等事件. 在此过程中, McErlang 通过模拟进程执行来实现程序的确定性模拟执行.

DSLabs[69]主要用于分布式系统教学, 提供测试、调试和模型检验框架, 与 Mace 类似. DSLabs 通过 Java 接口定义节点变量状态及原子事件 (主要为消息处理和超时处理), 使用者专注于协议实现 (如 Paxos、主备系统), 而框架负责操控网络、时钟、随机数和错误等不确定性因素并触发相关事件代码执行. DSLabs 采用基于状态的模型检验方法, 将协议级状态存储并加入探索队列, 并提供可视化调试器, 便于教学和分析.

从以上基于状态的确定性模拟执行技术的工作可以看出, 这些工作原理上与传统模型检验技术相似, 其优势在于能够直接运行真实代码, 从而发现代码中的实际缺陷. 然而, 其劣势也同样明显: 一方面, 这些技术通常需要对现有系统进行大量手工修改以进行移植, 或者限制使用特定语言或框架编写新系统; 另一方面, 基于状态的确定性模拟执行技术通常将分布式系统模拟为单进程运行[42−44,52,59,69], 与真实分布式环境存在差异, 而对多个节点的状态进行一致性快照既低效又难以精确实现[50,57]. 这些问题限制了该类技术在学术界的进一步发展和应用, 且尚未在工业界得到广泛应用.

3.1.3 基于事件的确定性模拟执行技术

基于事件的确定性模拟执行技术由 VeriSoft 首创, 该方法识别到基于状态的确定性模拟执行在管理复杂内存状态时存在挑战[77], 尤其是对使用任意编程语言编写的程序, 例如 CMC 涉及复杂的状态标准化. 尽管 JPF 和 Mace 通过运行时直接管理状态, 在一定程度上规避了状态管理问题, 但其适用范围仅限于特定语言和框架.

与基于状态的确定性模拟执行技术的状态回溯方法不同, 基于事件的技术通过在相同初始状态下确定性重放相同的事件序列来恢复系统状态, 从而避免了直接管理分布式系统全局一致内存状态的复杂性和开销. 基于事件的确定性模拟执行技术的优势影响了后续工作的技术选型, 例如 Chess[48]以及 MaceMC, 即使其所依赖的 Mace 框架支持基于状态的确定性模拟执行. 许多后续 DMCK 工具采用了基于事件的确定性模拟执行[11,40,49,56,64,67,71,73].

在技术实现上, 基于事件的确定性模拟执行主要依赖截获/插桩技术, 使得被测系统中与特定事件相关的代码确定性模拟执行. 根据截获位置在操作系统底层还是应用程序上层, 可将其分为两类: 通用型技术和精准型技术.

- 通用型技术[48,49,71]: 截获被测系统与操作系统或运行时环境交互的接口, 使得被测系统代码无需修改或仅需少量修改, 对被测系统具有较好的透明性. 然而, 由于事件的执行受环境间接控制, 可能存在操控精度问题, 并且新增事件支持可能需要修改 DMCK.

通用型技术的核心特点是无需修改被测系统代码即可实现确定性模拟执行, 从而更易适配新系统, 并使被测系统的运行环境更接近真实情况. 程序的执行受操作系统或运行时环境的影响, 例如超时、消息接收、磁盘读写、线程调度等, 在底层均由操作系统控制. 因此, 通用型技术通过直接截获操作系统或运行时接口来操控节点内的不确定性, 而无需修改被测系统代码. 同时, 在分布式环境中, 与错误相关的不确定性事件 (如节点故障和网络异常) 可通过错误注入进行模拟. 典型采用该技术的工作包括 MoDist[49]和 SandTable[71].

MoDist[49]是首个通用型 DMCK, 仅需提供配置文件即可验证未经修改的分布式系统. 它在 Windows 操作系统与被测应用程序之间插入了一个轻量级截获层, 以截获应用对 WinAPI 的调用, 并具备较全面的截获能力, 支持对网络、时间、文件、内存、线程及错误注入等全局和局部事件的操控. 针对 Windows 复杂的异步网络 API 带来的挑战, MoDist 采用代理线程机制, 将网络操作从目标线程中分离, 使所有网络交互均由模型检验后端控制. MoDist 模拟了分布式系统可能遇到的多种错误事件, 包括节点故障、网络异常及 WinAPI 调用失败等. 此外, MoDist 还针对确定性模拟执行的效率进行了优化, 例如通过推进虚拟时钟触发超时事件以加快时间推进, 并利用静态分析确定超时事件的触发值. 尽管如此, MoDist 仍然存在一些局限性, 例如它依赖于 Windows API 截获和控

制机制, 无法直接用于 Linux 或其他操作系统, 同时状态爆炸问题也限制了其错误注入的次数.

SandTable[71]主要面向实现复杂分布式协议的系统, 如 ZooKeeper 和 RedisRaft, 其确定性模拟执行技术支持对给定事件序列的重放. 其核心技术与 MoDist 类似, 但面向 POSIX API, 采用 LD_PRELOAD 动态链接库截获技术, 主要截获 C 标准库封装的系统调用, 在操作系统内核与被测系统之间插入轻量级截获层. 在分布式环境的操控方面, SandTable 采用 TPROXY (transparent proxy) 代理技术, 使网络消息在节点无感知的情况下被转发至确定性模拟执行引擎, 并由该引擎决定消息的送达时机. 此外, SandTable 还通过错误注入模拟了错误事件, 并实现了虚拟时钟机制, 加速节点感知的时间. 然而, SandTable 主要关注触发协议级逻辑缺陷的全局事件, 并且由于其操控底层操作系统的方式难以精准识别用户级线程调度, 因此不支持对局部事件进行有效操控; 另一方面, SandTable 的状态探索能力转移到了模型层, 因此无法“push-button”式直接使用.

Chess[48]专注于多线程并发程序的不确定性操控 (即局部事件), 其确定性模拟执行技术与 MoDist 类似, 通过透明截获 WinAPI 来操控线程调度. 然而, Chess 主要针对单机多线程程序的并发错误检测, 不支持分布式系统确定性模拟执行所需的全局事件截获. 不过, Chess 的实验包含了一个复杂分布式系统 Dryad, 表明其技术在分布式系统模型检验中的潜力.

从以上工作可以看出, 通用型技术已经具备了应用于工业级分布式系统的能力, 并且具有快速适配新系统的优势. 然而, Chess 和 MoDist 是专有工具, 未对外开源, 而 SandTable 由于面向特定场景, 无法“push-button”式直接使用. 同时, 由于底层系统调用截获和新增事件支持的复杂性, 基于该技术的相关研究尚不多见. 尽管如此, 该技术仍然展示了广阔的应用前景和研究潜力, 尤其在被截获的系统调用数量足够多的情况下, 适配新系统和增加新事件将变得更加容易.

● 精准型技术[11,45,56,64,67,73]: 直接插桩被测系统本身, 使事件代码能够精准确定性模拟执行, 具有较强的可扩展性. 然而, 在适配新系统时, 往往需要对被测系统进行一定程度的手工插桩.

精准型技术通过直接在源代码中插桩, 实现对事件调度的精确、细粒度控制, 确保系统的确定性模拟执行. 相比通用型方法, 它具有更高的灵活性和精准度, 但通常需要对被测系统进行一定的手工插桩. 典型代表性工作包括 SAMC[11]、FlyMC[67]、dBug[56]和 Remix[73].

插桩技术的主要优点包括: 1) 原理直接: 在源代码层面添加特定的截获逻辑, 使代码执行受确定性模拟执行器控制, 并按需阻塞或调度; 2) 操控精准: 插桩可精确作用于特定代码位置, 而非依赖外部环境间接影响, 从而提供更细粒度的控制; 3) 广泛适用: 插桩技术适用于多种编程语言, 如 Java[11,67]、C++[56,67]和 Erlang[64]等. 然而, 插桩技术也存在一定局限性: 1) 依赖源代码: 需要访问被测系统的源代码, 无法直接操控仅提供可执行文件的系统; 2) 适配成本: 不同系统需进行特定适配, 增加使用和维护成本; 3) 潜在副作用: 不当的插桩可能影响原始代码逻辑, 导致误报.

SAMC[11]、FlyMC[67]和 Remix[73]利用 Java 成熟的插桩库 AspectJ, 对 ZooKeeper、Cassandra、Hadoop 等工业级开源分布式系统进行插桩操控. 这些工具的状态恢复机制均通过从初始状态重新执行事件序列的方式来实现. SAMC 和 FlyMC 主要关注全局事件探索, 特别是在多错误注入场景下的状态空间约减; FlyMC 进一步优化了状态探索效率. Remix 在支持全局事件的同时, 还进一步支持多线程并发、磁盘 I/O 等局部事件, 并允许不同粒度的事件控制.

dBug[56]对 DMCK 技术原理和实现进行了详细描述. 它依赖于手动使用其提供的 C/C++ 库运行时 API 对不确定性事件进行插桩, 插桩的细节程度决定了控制不确定性事件的范围, 理论上支持全局事件和局部事件的控制. 被测系统运行于 dBug 客户端, 截获事件后发送至 dBug 服务器端并等待调度. dBug 使用虚拟机运行被测系统, 虽然虚拟机提供了快照功能, 但快照和恢复操作耗时且消耗空间, 同时虚拟机的内存状态难以用于状态去重, 因此 dBug 仅对初始状态进行保存和恢复, 非初始状态则采用基于事件的回溯技术.

MaceMC[45]是基于 Mace 框架[44]开发的模型检验工具, 专注于发现活性缺陷 (liveness bug). Mace 框架提供了编程规范和运行时环境, Mace 代码通过使用框架提供的接口, 实现了类似于自动化精准插桩的功能. 然而, 该框架并不支持直接应用于现有的系统. Mace 框架支持基于状态的确定性模拟执行, MaceMC 利用这一特性, 通过状态的哈希比对避免遍历重复状态. 然而, MaceMC 并不存储所有状态, 而是采用基于事件的状态回溯方法, 从而减少了实

现复杂度和内存消耗, 但回溯过程会增加一定的 CPU 开销. 为提高回溯效率, MaceMC 可选择性地存储部分状态.

Concuerror[64]是一个针对 Erlang 并发程序的模型检验工具. 它利用了 Erlang 语言的 actor 模型及其统一的进程通信机制, 使得 Concuerror 的自动插桩工具能够直接对 Erlang 源代码进行语法变换, 从而对不确定事件自动插入插桩代码. 然而, 尽管 Erlang 的 actor 模型和消息传递特性常用于构建分布式系统, Concuerror 主要用于检测并发错误, 关注多线程执行的验证, 不支持涉及节点故障、网络异常等全局错误事件的检验.

从以上工作可以看出, 精准型插桩技术因其原理直接、操控精准以及广泛适用等优点, 得到了广泛的研究. 然而, 其主要劣势在于被测系统需要一定程度的手工插桩. 尽管如此, 在工业界, 由于大多数企业自主开发系统并拥有全面的系统知识, 插桩的成本相对可控[11].

3.1.4 确定性模拟执行作为关键技术的相关研究

近年来, 确定性模拟执行技术在分布式系统测试和重放调试研究中得到了广泛应用[78−83]. 在工业界和开源社区, 受到 FoundationDB[84]启发的确定性模拟执行测试技术得到了广泛应用[85−88]. 这些技术在分布式系统的测试与调试中发挥了重要作用. 尽管这些研究通常不涉及状态空间的系统性探索, 因此不属于严格意义上的 DMCK, 确定性模拟执行技术作为 DMCK 的关键使能技术, 经过适当扩展和演变, 可以转化为 DMCK. 例如, MoDist[49]继承了其前身 D3S[89]的分布式系统调试能力, 而 MaceMC 及其调试器 MDB 也依赖 MaceMC 的模拟器来实现确定性模拟执行[45]. 鉴于此, 这些工作具有重要的研究价值. 接下来, 我们将分别从分布式系统确定性重放调试作为关键技术的研究工作和 FoundationDB 启发的工业界应用两个角度展开讨论.

Liblog[83]是一个分布式系统事件记录和确定性重放库, 基于 LD_PRELOAD 技术截获 C 标准库的系统调用封装函数. Liblog 采用中心化的 logger 进程记录事件日志, 不同节点截获的信息 (如网络接收、节点超时等) 会被发送到 logger. 重放时, Liblog 通过恢复最近的内存快照, 并按照事件日志的顺序执行, 以确保执行过程的可复现性. Friday[79]在 Liblog 的基础上, 提供了分布式断点和全局状态分析机制. 它能够协调多个 GDB 实例, 在指定节点上设置断点, 并检查全局属性, 从而增强分布式系统的调试能力. WiDS[78]基于 Macedon 框架, 在单个进程中模拟分布式系统. 其核心方法是通过随机执行来发现缺陷, 并提供了缺陷确定性重放的调试功能, 使开发者能够精准复现并分析问题. Morpheus[80]是一个针对 Erlang 分布式系统的并发测试工具, 通过程序重写技术拦截通信原语, 采用偏序采样 (partial order sampling, POS) 方法进行随机测试, 同时提供确定性执行能力. 当发现错误时, Morpheus 能够记录并重放精确的执行轨迹, 确保错误的可复现性. 该工具在 RabbitMQ 等系统中发现了 11 个先前未知的协议错误, 展现了其有效性. DEMi (distributed execution minimizer)[81]通过确定性调度优化分布式系统的缺陷执行路径, 减少 fuzzing 发现的缺陷中不必要的执行步骤, 从而提升可分析性和缺陷重现能力. 该工具通过插桩 Akka 框架的消息和时钟, 实现受控的确定性执行. Mocket[82]利用形式化模型的状态空间生成测试用例, 并通过 Java ASM 插桩技术对分布式系统进行确定性执行, 以精确控制测试用例的执行过程. 测试过程中, Mocket 通过对比执行状态与模型状态是否一致来检测系统缺陷.

FoundationDB[84]是工业界最早在分布式数据库测试中引入确定性模拟执行 (deterministic simulation) 的系统. 由于当时 C++ 语法尚未提供对 actor 模型和协程 (如 async/await) 的语言级支持, FoundationDB 开发了 Flow 语言, 在 C++ 之上引入 actor 模型, 并通过对网络、磁盘、时间、随机数的模拟使得测试环境完全可控. 然而, 这也导致其确定性模拟引擎与 FoundationDB 紧密耦合, 难以复用于其他项目. FoundationDB 以极高的可靠性著称, 其支撑着苹果 iCloud 业务的数十亿级数据库. FoundationDB 的联合创始人 Will Wilson 随后创办了 Antithesis 公司[85], 专注于利用确定性模拟执行技术进行分布式系统测试. Antithesis 构建了一个确定性模拟的虚拟机环境, 使得分布式系统中 (或计算机系统中) 的几乎所有不确定性因素均可控[90]. 该公司获得了多轮风险投资 (当前共融资 7 700 万美元), 反映了确定性模拟执行技术在工业界的广泛关注和应用潜力. 在 FoundationDB 的启发下, 多款工业级产品构建了各自的确定性模拟执行框架. 例如, RisingWave 开发者开发了 MadSim[87]框架, sled 数据库构建了其专属的确定性模拟执行框架[86], tokio 项目也集成了类似的测试方法[88].

3.2 状态空间探索技术

本节将介绍状态空间探索技术的原理及其优化方法. 状态空间探索从初始状态出发, 系统性穷举可调度事件

的执行顺序, 通过确定性模拟执行技术来执行调度的事件. 为应对状态爆炸问题, 研究者们提出了多种优化技术, 包括状态空间约减技术和探索效率优化. 接下来, 我们将分别讨论这些技术的实现和优化.

3.2.1 状态空间搜索方法

状态空间搜索方法可分为有状态搜索 (stateful search)[91]和无状态搜索 (stateless search)[77]. 根据定义, 有状态搜索的下一次搜索依赖于历史状态, 即之前探索的路径; 而无状态搜索不记录历史状态, 每次搜索的决策独立于之前的探索过程.

第 3.1 节中提到确定性模拟执行技术分为基于状态和基于事件两种方法. 在状态空间搜索上, 基于事件的方法由于缺乏状态保存能力, 只能采用无状态搜索; 而基于状态的方法具备完整的状态保存能力, 通常采用有状态搜索, 也可采用无状态搜索.

有状态搜索 (stateful search): 有状态搜索的原理与传统模型检验搜索算法类似, 算法从一个初始状态开始, 搜索并探索整个状态空间, 算法伪代码如图 3 所示[77]. 算法维护两个集合: 已探索状态集 (*Explored*) 和未探索状态集 (*Unexplored*). 初始状态 s_0 加入未探索集合中 (第 2 行). 然后, 算法循环从未探索状态集取出一个状态. 如果该状态不在已探索集合中, 则对其进行探索并将其加入已探索集合 (第 3–13 行). 对被探索的状态的所有可执行/使能 (*enabled*) 转移进行执行并产生新的后继状态, 并将这些后继状态加入未探索集合中 (第 7–11 行). 循环此过程直到未探索状态集为空.

```
1   Initialize:  Unexplored is empty;  Explored is empty;
2                add s0 to Unexplored;
3   Loop: while  Unexplored ≠ ∅ do {
4                take  s out of  Unexplored;
5                if s is NOT already in  Explored then {
6                   enter  s in Explored;
7                   T = enabled(s);
8                   for all  t in T do {
9                      s′ = succ(s) after  t;
10                     add  s′ to  Unexplored;
11                  }
12               }
13          }
```

图 3 有状态搜索算法 (引自 VeriSoft[77])

下面解释 DMCK 中有状态搜索的具体实现细节, 不同实现方式可能影响探索效率和缺陷发现速度.

● 状态取出顺序影响缺陷发现的路径和速度 (第 4 行). 深度优先搜索 (未探索状态集为 LIFO 栈) 优先探索较深状态, 可更快触及长路径缺陷. 广度优先搜索 (FIFO 队列) 发现的缺陷触发路径最短, 便于分析, 但当缺陷位于较深层时, 探索时间可能更长. 随机策略可提高状态多样性, 加快代码覆盖率. 启发式策略需结合领域知识, 优先探索高价值状态, 可提升特定缺陷发现速度, 并可与其他搜索方式结合[42].

● 状态比对用于状态去重 (第 5 行). 最基本的方法是计算状态哈希进行比对, 更高级的方法是在计算哈希前进行状态标准化, 例如利用节点角色对称性将等价状态归一化[74], 以进一步减少冗余状态.

● 状态表示方式影响状态空间规模. 最直接的方法是记录系统内存信息[42], 更精简的方法是仅存储协议关键变量状态[44,69], 可减少状态数量和内存占用, 但可能误判不同状态为相同, 导致遗漏.

● 可执行/使能转移 (enabled transitions) 由确定性模拟执行器报告 (第 7 行). 例如, 可触发的超时、可送达的消息和可注入的错误等. 由于不确定性, 多个动作可能同时使能, 确定性模拟执行器通常使用 Toss/Choose[42,45,77]函数, 在指定范围内依次返回不同值 (类似 fork 效果), 确保探索所有路径.

无状态搜索 (stateless search): 无状态搜索按定义不记录历史状态, 每次探索的决策独立于此前的执行路径. 这种策略虽实现简单, 但会导致等价状态无法区分而被多次重复遍历, 使状态/路径空间急剧增长. 此外, 无状态搜索回溯时需从初始状态重新执行事件序列来生成状态, 增加了 CPU 开销. 因此, 纯粹的无状态搜索算法在实践中效

率极低, 基本不可行.

为解决上述问题, VeriSoft 提出了高效的无状态搜索算法 (efficient stateless search)[77], 引入偏序约减 (partial-order reduction)[92]等技术来减少对等价执行路径的重复探索. 尽管该算法仍在形式上不保存状态, 其搜索决策实际上利用了历史信息来约减状态空间. 由于 VeriSoft 是最早面向真实程序进行系统性状态空间探索的工具, 首次系统化提出了无状态搜索的概念, 其定义和实现方式影响深远. 本文后续提及的“无状态搜索”均指这种结合偏序约减的高效实现方式, 而非原始意义上的完全无历史信息的搜索算法.

VeriSoft 核心思想是, 若多个动作无依赖关系 (如不同节点上的独立操作), 则不同执行顺序最终到达的状态等价. 例如, 若网络缓存中有发送至节点 A 和 B 的消息, 则 A 先接收或 B 先接收不会影响二者都接收消息后的最终状态. 偏序约减通过选择性搜索 (selective search) 减少冗余探索, 即在每个状态仅搜索部分使能动作, 而非所有可能的动作.

VeriSoft 算法 (图 4) 基于深度优先搜索, 并结合 sleep-set[93]和 persistent-set[92]剪枝优化. *Stack* 结构存储从初始状态到当前状态的转移动作 (第 9 行和第 12 行), 用于支持撤销 (Undo) 操作 (第 13 行), 即通过重放 *Stack* 中的转移动作重构先前状态. 每个状态 *s* 关联 *Persistent* 集合和 *Sleep* 集合: *Persistent* 集合 (由 Persistent_Set 函数返回) 包含被使能的部分动作, 未包含的动作与其独立, 因此仅探索 *Persistent* 集合可避免重复探索等价路径 (Mazurkiewicz trace)[94]. *Sleep* 集合记录已探索的独立事件组合, 作为深度优先搜索的参数 (第 11 行), 在下一次探索时排除 (第 6 行), 进一步减少等价路径的重复搜索.

```
1   Initialize:  Stack is empty;
2   Search() {
3     DFS(∅);
4   }
5   DFS(set: Sleep) {
6     T = Persistent_Set () \ Sleep;
7     while T ≠ ∅ do {
8       take t out of T;
9       push(t) onto Stack;
10      Execute(t);
11      DFS({t′ ∈ Sleep | t′ and t are independent});
12      pop t from Stack;
13      Undo(t);
14      Sleep = Sleep ∪ {t};
15    }
16  }
```

图 4 无状态搜索算法 (引自 VeriSoft[77])

无状态搜索算法的优势在于避免存储状态带来的问题. 相比广度优先或随机搜索, 深度优先搜索减少了无状态搜索中高开销的回溯次数, 并结合 *Stack* 结构, 仅存储必要的路径信息以在回溯时重构状态, 从而降低事件存储成本. 其缺点是无法检测状态空间中的环, 遇到环时可能导致无限递归, 可通过限制搜索深度来规避. 此外, 由于不支持环检测, 该算法无法直接验证标准的活性 (liveness) 属性, 但可通过检查有限事件序列来近似检测特定活性属性, 例如, 若系统未终止于某个期望状态, 则可判定为死锁.

3.2.2 状态空间约减方法

状态空间探索面临状态爆炸问题, 导致在有限时间内无法穷尽所有状态, 从而影响 DMCK 的有效性. 因此, 所有相关研究均提出了相应的应对方案. 传统 DMCK 研究主要集中在适用于分布式系统的通用状态空间约减技术上. 由于状态空间约减的核心在于: 约减后的探索路径是否仍能覆盖所有可能导致属性违反 (即系统缺陷) 的执行路径, 这一能力通常被称为约减算法的 soundness[95], 是模型检验中确保验证有效性的基础. 为便于后文讨论, 本文将满足 soundness 要求的方法称为“可靠型约减方法”, 而不满足 soundness 要求的方法称为“不可靠型约减方法”.

可靠型 (sound) 约减方法: 可靠型约减方法包括状态去重、对称约减、偏序约减和动态接口约减, 这些方法相

互独立且正交. 对于有状态搜索, 相关工作[42,43,47,69,74]采用基于状态的确定性模拟执行技术, 通过计算状态哈希并比对来识别和约减重复状态. 有状态搜索的约减技术通常利用状态的对称性[43,74], 将理论上等价的状态归并为一类, 并通过状态标准化机制确保等价状态的哈希值相同. 当探索到一个状态时, 如果其等价状态已被探索, 则跳过该状态, 减少需要探索的状态数量[96]. 例如, 在分布式系统中, 如果多个节点具有对等身份且执行相同代码, 交换节点的身份 ID 不会影响系统行为, 这时可以将交换身份 ID 的调度产生的状态视为对称的.

有状态搜索还可以与 (静态) 偏序约减 (partial order reduction, POR) 技术结合, 进一步约减执行路径[52,59,74]. 然而, 动态偏序约减 (dynamic partial order reduction, DPOR) 技术[97] (后文将讨论) 与有状态搜索的结合较为复杂, 因为在 DPOR 的状态探索或回溯过程中, 已遍历的状态被跳过, 从而导致状态探索不完整. 尽管一些新算法[98,99]成功结合了有状态搜索和 DPOR, 并解决了可靠性问题, 但这些算法可能会导致更高的内存消耗和更多的时间开销[59], IoTCheck[99]的实验结果也表明了这一点.

对于无状态搜索, 目前仅有个别研究使用基于状态的确定性模拟执行技术[45]来支持状态去重, 而状态回溯依赖于无状态搜索的事件回放. 相比之下, 更常见的无状态搜索方法采用基于事件的确定性模拟执行技术, 不存储状态信息, 无法区分已访问状态, 容易导致状态空间迅速膨胀.

尽管无状态搜索算法结合了偏序约减技术, 该算法仍然面临探索较大的状态空间挑战. 计算 *Persistent* 集合通常需要进行静态分析, 例如分析每个进程可能在哪些通信对象上执行哪些操作. 然而, 要精确计算出一个合适的 *Persistent* 集合仍然非常困难, 导致状态约减效果不佳[97].

为了解决静态分析的不准确性并最大程度减少等价执行路径的探索, Flanagan 等人[97]提出了动态偏序约减 (DPOR) 技术. DPOR 通过动态检测下一转移与当前执行路径中的转移的独立性, 在必要时为当前路径添加回溯点, 只探索那些与当前路径不等价的执行路径. 具体而言, DPOR 基于深度优先搜索, 在探索路径中的每个状态 s 时, 为每个进程的下一转移维护一个回溯集合 (backtracking set). 当遇到一个转移时, 算法会追溯到当前路径中最后一个与该转移有依赖关系的转移 (记作 S_i), 并利用“happens-before”关系来判断该依赖是否影响进程 p 的未来行为. 如果有影响, 则将可能导致不同结果的转移添加到 S_i 执行前状态的回溯集合中. 当深度优先搜索返回至 S_i 执行前的状态时, 回溯集合中的转移会被探索, 确保所有加入回溯集合的路径都得到探索. DPOR 算法通过数学证明了, 已探索状态的回溯集合实际上是该状态的一个 *Persistent* 集合, 从而验证了 DPOR 算法的正确性.

DPOR 算法易于实现且能大幅减少状态空间, 为 DMCK 研究的开展提供了基础, 并成为后续基于无状态搜索的研究所需实现的标准技术[11,49,56,67]. 后续研究提出了分布式 DPOR 技术, 通过将不同路径的探索分布到多个节点上, 大幅提升了 DPOR 的可验证规模[100]. 此外, DPOR 并不保证探索路径数量的最小化, 为此, 后续研究提出了最优 DPOR (optimal DPOR)[101], 使等价的执行路径仅探索一次. 然而, 这些更先进的 DPOR 技术尚未有 DMCK 研究工作进行集成.

尽管对称约减技术与偏序约减技术是正交的, 但在无状态搜索中无法直接结合. Godefroid[102]通过将传统对称约减基于的状态等价性转换为执行路径等价性, 在 VeriSoft 的偏序约减基础上实现了对称约减. Spore[103]将对称约减与 DPOR 结合, 进一步减少了无状态搜索的状态空间. 然而, 这种结合方式尚未在 DMCK 场景下适配和应用.

尽管状态去重、对称约减和偏序约减技术大幅减少了状态空间, 分布式系统的状态空间仍然十分庞大, 难以完全穷尽. 这主要源于系统节点之间的消息通信与节点内部进程或线程间的交互逻辑的复杂组合. DeMeter[40]提出了动态接口约减 (dynamic interface reduction, DIR) 技术, 该技术利用分布式系统良好定义的接口, 即组件通过消息进行通信. 其核心思想是动态发现系统组件之间的接口, 将不同组件分开检查 (局部状态空间探索), 并通过接口行为组合它们, 以避免昂贵且不必要的全局状态空间探索, 从而在保持可靠性的前提下, 实现指数级的状态空间压缩 (可达 10^5 倍). DeMeter 结合全局探索与局部探索两种模式. 全局探索阶段首先获取仅包含消息收发的执行路径, 并将消息投射到相应组件. 局部探索阶段对各组件进行单独检查, 例如通过重放即将达到的消息, 并探索组件内部的不同执行顺序 (如改变检查点线程与消息接收线程的调度). 如果局部探索过程中某组件发送了全局探索未发现的新消息, 该消息将被汇报给全局探索, 与已有的消息执行路径组合, 重复执行此过程, 直至全局探索不再产生新消息执行路径, 且局部探索完成已有消息的探索.

从以上研究可以看出, 可靠型通用分布式系统状态空间约减技术已被广泛研究, 许多工作的核心贡献即在于提出或优化状态约减算法. 然而, 尽管这些约减技术在理论上是正交的, 它们的结合并不容易. 目前尚未有 DMCK 研究同时集成所有这些约减算法.

不可靠型 (unsound) 约减方法: 不可靠约减方法通常利用领域知识或启发式策略, 针对特定场景进行优化, 例如启发式丢弃状态、存储状态时忽略部分变量、限制探索深度等.

CMC[42]采用启发式算法标记某些状态为更有价值的状态, 使其优先被探索, 而价值较低的状态可能被丢弃. 该启发式算法倾向于搜索明显偏离常规的状态, 其核心思想是: 如果状态的比特位数突然增加, 或变量取值变得罕见, 则认为该状态更有价值. 此外, CMC 允许在状态标准化时剔除用户认为不重要的内存信息, 使得某些状态被视为等价.

MoDist[49]结合多种策略来有效发现缺陷, 例如使用随机策略发现浅层缺陷, 并结合随机探索与 DPOR 提高路径多样性. 由于随机策略的引入, 部分状态可能无法被探索. DeMeter[40]采用快照机制, 对感兴趣的状态进行存储, 并将这些状态作为初始状态进行探索, 同时丢弃不感兴趣的状态, 导致约减方法不可靠.

MaceMC[45]旨在发现活性 (liveness) 属性被违反的缺陷, 采用三阶段状态探索方法来检查活性属性. 其中, 前两个阶段使用有界深度优先搜索 (bounded depth-first search, BDFS) 探索状态空间到一定深度, 并利用随机游走策略深入探索可能导致活性属性违反的外围状态. 对于超过上万深度的随机游走仍无法满足活性属性的路径, MaceMC 启发式地标记其为违反路径. 这种限定深度和随机游走策略使得 MaceMC 的约减方法不可靠.

CrystalBall[50]主要针对已部署分布式系统的在线模型检验, 旨在预测和预防系统进入不一致状态. 由于分布式环境难以获取全局一致性快照, CrystalBall 仅对节点及其通信的邻居节点进行局部快照和模型检验, 采用迭代加深的深度优先搜索算法, 并根据设定的时间约束停止搜索. CrystalBall 属于不可靠型约减, 因为其模型检验仅针对特定时机进行空间和时间上的限定, 尽管能聚焦最关心的状态, 但无法捕捉整个分布式系统的全局状态变化可能带来的缺陷. 此外, 搜索时间受限可能导致未能探索到导致缺陷的深度.

CrystalBall 的后续工作 LMC[57]也面向在线模型检验, 提出了与 DeMeter 类似的观察, 即全局状态由所有节点状态和网络状态组成, 并且二者是可以分离的. 不同的是, LMC 观察到网络状态的频繁变化会导致大量不同的全局状态, 使得在线模型检验难以在短时间内达到有意义的深度. LMC 进一步发现网络状态并不用于属性检查, 因此提出“局部模型检验 (local model checking, LMC)” 技术, 该技术使用共享的网络状态, 包含网络中所有历史消息 (即发送到网络的消息不会被删除), 仅根据节点状态区分状态是否相同, 并将共享网络状态与节点状态合成全局状态进行检验. 这种方法可能导致系统进入实际不存在的状态, 因此 LMC 在检测到属性违反后, 会对执行路径进行检查, 确保报告的反例真实存在. 由于该方法面向已部署系统的在线检验, 相较于传统探索方式, 它能在数秒内探索到一定深度 (如 20–30 层状态), 当网络消息过多导致状态爆炸时, LMC 关注的是系统运行中的最新状态, 而不会处理所有可能的状态.

从以上不可靠型约减方法的研究可以看出, 传统 DMCK 工具大多采用不可靠型约减来加速缺陷发现. 这反映出状态爆炸问题的严峻性, 使得通用型可靠状态约减技术已接近瓶颈, 促使这些工具在牺牲可靠性的前提下换取更高效的缺陷发现能力.

3.2.3 状态探索效率优化

与状态空间约减正交的缓解状态爆炸问题的方式是优化探索速度. 探索速度越快, 有限时间内能够探索的状态或路径数量就越多. 一些工作在单个模型检验进程内模拟分布式系统节点和环境[42,45,52]. 尽管这种方式通常状态探索速度较快, 但它仅适用于使用特定框架编写的程序, 或者需要较高的适配成本. 越来越多的工作则在更真实环境中启动多个分布式系统节点[11,40,49,67,71,73], 这些工作通常基于无状态搜索技术. SandTable[71]总结了 3 大原因, 导致传统 DMCK 状态探索速度较慢: 1) 无状态搜索回溯时重复执行事件序列 (例如反复重新初始化); 2) 代码事件执行速度较慢 (例如文件 I/O); 3) 模型检验代码带来的额外开销 (例如事件调度同步开销). 相关工作的优化策略通常有一定共性, 主要通过虚拟时钟、缓存部分状态 (尤其是初始状态) 和减少事件执行间等待时间等手段来提高

探索效率.

在探索一条执行路径时, 首先需要初始化集群环境, 这一过程涉及诸多耗时操作, 如清理磁盘和启动节点进程. 例如, FlyMC 的集群初始化平均耗时 18 s. 无状态搜索的状态回溯会导致重复执行事件序列, 而初始化集群是所有执行路径的首个事件, 因此消耗大量时间. 为优化此过程, FlyMC 和 dBug 采用初始状态快照, 大幅减少了重复初始化的开销, FlyMC 将平均初始化时间从 18 s 降至 2 s[104].

代码执行过程中存在多种影响速度的因素, 如超时等待、磁盘写入和虚拟机运行时开销. 其中, 与超时相关的逻辑通常可通过虚拟时钟加速[49,67,71], 避免真实等待. MoDist 采用虚拟时钟, 实现了 22–159 倍的加速. 然而, 其他影响执行速度的因素通常难以进一步优化.

模型检验器本身也可能引入额外开销, 如插桩和调度开销. 插桩开销因确定性模拟执行技术的不同而有所变化, 例如 MoDist 的插桩开销占代码执行时间的 18%–57%, 通常难以进一步优化. 调度开销则与实现方式相关. 例如, FlyMC 在执行 54 个事件时耗时 18 s, 其中约 14 s 用于调度器在事件执行前的等待, 以防止并发问题. FlyMC 观察到并发问题较为罕见, 因此采用历史记录追踪缓存状态-事件转移, 缓存命中时无需等待, 将事件执行时间缩短至 4 s[104]. 调度开销还受状态探索算法影响. MP-Basset 实现了有状态和无状态两种搜索算法, 有状态搜索在大规模状态空间下优于无状态搜索; 而在较小状态空间下, 无状态搜索更具优势[59].

现有工作均在一定程度上优化了状态探索速度, 但由于代码执行时间已成为主要开销, 进一步优化已接近瓶颈. 我们总结了对代码直接进行模型检验的典型工具在单条执行路径上的平均执行时间: 底层操作系统劫持的 MoDist 约为 2 s; 基于代码插桩的 dBug 需 8–38 s, SAMC 接近 40 s, FlyMC 在 SAMC 基础上优化至 6 s.

3.3 属性验证

模型检验通过检查安全性 (safety) 和活性 (liveness) 属性来判定系统是否满足预期要求, 若属性不满足, 则表明系统存在缺陷. 安全性检查确保系统在整个状态空间内始终满足特定条件, 即“坏事不会发生”. 活性检查则要求系统最终达到预期状态, 即“好事最终会发生”. 目前, 所有传统 DMCK 均支持安全性检查, 而仅有少数支持活性检查 (如 MaceMC 和 MoDist)[45,49].

3.3.1 安全性验证

通用安全性约束: DMCK 直接运行被测系统代码, 因此需满足一些基本的通用安全性约束, 例如无内存泄漏、无堆栈数据损坏、无节点异常崩溃等. DMCK 可检测节点在无错误注入的情况下是否崩溃, 若发生崩溃, 则表明系统存在缺陷[49,71]. 此外, 由于代码真实动态运行, 其他与内存相关的安全性问题通常可借助外部运行时检测工具, 例如 Valgrind[105].

领域特定安全性: 分布式系统的安全性通常依赖于具体应用场景, 违反这些属性可能导致严重后果. 例如, 共识算法 (Raft、Paxos 等) 实现的分布式数据库需确保数据一致性、不可丢失、仅有一个有效主节点、提交编号单调递增等. 这些领域安全性检查通常由开发者手动编码. 部分安全性可通过单个节点的本地数据检查, 例如提交编号的单调递增性, 可直接在代码中插入断言 (assertion) 实现. 工业级系统通常在开发阶段已广泛使用断言. 而涉及全局一致性的安全性 (如集群仅有一个有效主节点) 则需获取分布式系统的全局快照. 全局状态快照的获取依赖于确定性模拟执行技术, 该技术可确保所有节点状态一致. 在基于事件的确定性模拟执行技术中, 系统需提供额外的状态观测能力. 例如, MoDist 采用 D3S[89]的 state exposer 技术, 通过透明劫持节点进程的特定函数或系统调用来收集状态信息, 并由全局检查器验证系统行为是否符合安全性要求.

3.3.2 活性验证

活性属性要求系统最终达到某个期望状态, 而不仅是检查某些状态是否满足条件. 理论上, 活性验证需覆盖所有可能 (包括无穷) 的执行路径或状态. 传统模型检验通常采用强连通分量 (strongly connected component, SCC) 分析, 并基于公平性假设 (fairness assumption) 排除“不现实”的执行路径, 其核心思想是: 如果系统进入一个 SCC 无限循环且不满足活性属性, 则认为活性属性被违反.

然而, 在 DMCK 中, SCC 技术难以适用: 1) 有状态搜索在面对大规模分布式系统状态空间时, 难以高效使用

SCC 进行活性验证; 2) 无状态搜索从原理上无法支持 SCC 技术. 因此, 大部分 DMCK 主要关注安全性验证, 但仍有一些工作通过启发式方法和安全性属性来近似判定活性属性[45,49].

MaceMC[45]是首个主要面向活性验证的 DMCK, 认为活性与安全性同样重要. MaceMC 采用启发式方法, 检测系统是否进入无法恢复的死状态, 而非传统活性验证要求的无穷执行路径分析. MaceMC 采用三阶段探索方法: 1) 有界深度优先搜索: 在有限深度 (约 25–30) 内穷尽所有执行路径, 识别外围状态中可能不满足活性属性的中间状态; 2) 随机游走: 从中间状态出发, 进行深度达上万步的随机探索, 若仍未满足活性属性, 则标记为可能的活性违反; 3) 关键转移分析: 识别导致系统进入死状态的关键转移, 以帮助开发者定位根本原因. 虽然 MaceMC 的方法不能保证所有发现的活性违反都是实际缺陷, 但其报告的缺陷未出现误报, 表明方法有效.

MoDist 采用启发式方法结合安全性属性检查来近似验证活性: 假设在无错误注入的情况下, 系统应当持续推进工作. 在所有被测系统中, MoDist 均发现了活性违反缺陷, 这些缺陷被确认为协议设计层面的严重问题. 值得一提的是, MoDist 在全局一致快照中仅发现了通用安全性问题及系统自带断言的违反, 而未发现额外的领域安全性违反. 这一结果表明被测系统的成熟度, 同时支持了 MaceMC 关于活性属性同样重要的观察.

4 系统语义感知的分布式系统模型检验

状态爆炸问题是制约模型检验可扩展性的核心挑战, 在 DMCK 中尤为突出, 其根本原因在于分布式系统设计与实现的高度复杂性及运行环境中的高度不确定性, 导致状态空间迅速增长. 为缓解状态爆炸, 传统 DMCK 通常基于通用策略开发了状态空间约减技术, 如状态去重、对称约减、动态偏序约减和动态接口约减等. 然而, 这些通用约减策略缺乏对系统语义的深入挖掘与融合, 其约减能力已接近瓶颈. 为提升缺陷发现能力, 部分研究转向不可靠约减方法[40,42,45,49,50,57], 虽可加速发现缺陷, 但难以提供全面的正确性保障.

为应对上述瓶颈, 研究工作开始探索如何融合系统语义信息以提升状态约减能力. SAMC[11]指出, 传统 DMCK 未系统性利用事件间的独立性与对称性等语义特性, 导致动态偏序约减与对称约减的效果受限, 重复探索了大量等价路径. 尤其在分布式系统中, 节点崩溃与消息传递等事件之间往往存在独立性与对称性, 但传统方法对此缺乏建模, 从而造成不必要的状态穷举. SAMC 首次提出系统语义感知的分布式系统模型检验技术, 聚焦分布式容错系统中普遍存在的语义特性, 开发者仅需提供少量语义建模, 即可进行高效的状态约减.

这一阶段的 DMCK 工作关注如何通过少量人工建模或标注, 显式表达系统中的对称关系、事件独立性与垂直领域特性, 以辅助状态约减与状态划分. 实践表明, 系统语义感知的 DMCK 能显著约减状态空间, 即使不依赖不可靠的约减策略, 仍具备高效发现深层缺陷的能力, 从而显著提升模型检验结果的有效性.

此外, 随着 DevOps (Development 与 Operations 融合的软件开发流程) 模式的普及, 开发者通常也承担测试职责, 对系统语义具备深入理解. 这一趋势为语义建模的推广提供了现实基础: 语义信息的构建成本较低, 具备良好的实用性与扩展性. 后续研究也表明, 更系统化地挖掘与利用语义信息具有推动 DMCK 向下一阶段演进的潜力.

4.1 系统语义感知模型检验框架

我们深入分析了系统语义感知 DMCK 的相关工作, 并从 DMCK 的演化视角出发, 将其框架划分为 3 层 (如图 5 所示). 其中, 系统语义感知 DMCK 位于中间层, 研究重点在于系统性引入语义信息以增强状态约减能力.

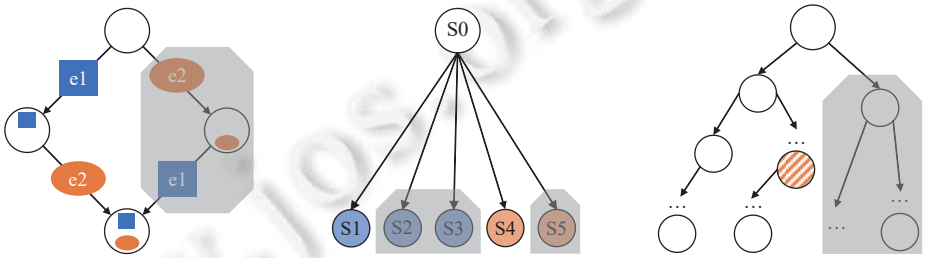
图 5 系统语义感知的模型检验框架

框架从下至上, 底层为传统 DMCK 所研究的通用状态探索与约减机制, 包括搜索算法、动态偏序约减、对称性约减, 以及面向不同局部组件的状态划分策略等, 构成上层利用语义策略进行状态探索的基础支撑. 中间层关注分布式系统中的共性语义特征, 如消息传递与节点崩溃事件之间的独立性与对称性, 也包括面向容错等垂直领域的通用语义结构. 此阶段开始系统性分析分布式容错逻辑, 并通过白盒方式收集系统内部语义信息, 结合人工建模加以利用. 最上层则代表后续研究对系统关键模块的深入建模, 通过形式化模型和正确性规约等手段表达更细粒度的系统特定语义, 进一步推动 DMCK 向新阶段演进.

本节重点关注中间层的语义规则设计. 此类规则可通过手工建模或结合人工标注的静态分析生成, 并反馈给底层机制加以利用: 与独立性相关的规则由动态偏序约减机制驱动, 对称性规则由对称约减机制驱动, 垂直领域相关规则则结合专门的领域化约减策略, 聚焦关键行为或故障模式.

4.2 系统语义感知优化技术

本节从独立性、对称性和垂直领域约减这 3 个方面展开讨论. 独立性与对称性约减主要利用分布式系统中消息与崩溃的语义信息去除等价状态或路径, 而垂直领域约减针对特定分布式系统的领域特性聚焦特定状态空间, 如图 6 所示, 灰色框表示被约减的状态或路径.



(a) 事件独立性约减 (b) 状态对称性约减 (c) 垂直领域约减

图 6 独立性、对称性和垂直领域约减示意图

4.2.1 独立性约减

动态偏序约减依赖事件的独立性, 以避免对独立事件进行无意义的重排序. 对系统事件独立性的信息越充分, 可约减的等价执行路径就越多. 传统 DMCK 未利用系统语义信息, 通常仅判断不同节点间的消息独立性. 例如, 在同一状态下, 若多个消息发送至不同节点, 因各节点对不同消息的处理互不影响, 系统执行结果不会因接收顺序变化而发生改变, 因此这类消息被视为彼此独立.

为更全面地识别分布式系统中的独立事件, 尤其是多条消息发送至同一节点的独立性, 以及崩溃事件与消息事件的独立性, SAMC[11]提出了两种基于系统语义的独立性约减策略. 首先, 本地消息独立性 (local-message independence, LMI) 关注同一节点接收的多条消息是否独立. SAMC 根据消息对本地 (节点) 状态的影响方式, 并结合分布式系统中常见的设计模式, 总结出 4 类消息处理语义, 并将其封装为条件语句: 1) 消息被丢弃 (predicate discard, pd), 即消息不会影响本地状态, 直接丢弃, 例如许多分布式协议 (如 Raft, Paxos) 会丢弃旧版本号或过时的投票信息; 2) 变量自增 (predicate increment, pi), 即消息接收处理对某个变量进行自增操作, 例如选举投票计数的更新; 3) 变量赋值为常量 (predicate constant, pc), 即消息到达后直接赋值为固定值, 例如节点接收到领导者消息后角色变为跟随者; 4) 本地状态修改 (predicate modify, pm), 即消息会导致本地状态发生变化, 例如接收到客户端请求后更新日志内容. 基于这些语义, SAMC 设定了以下规则来判定消息是否独立: 1) 若任意一条消息满足 pd, 则两条消息独立; 2) 若两条消息均满足 pi 或 pc, 则它们独立; 3) 若其中一条消息为 pm, 且无 pd 消息, 则两条消息不独立. 用户只需对语义进行简单建模即可利用这些规则, 例如 m.vote < localState.myVote 这一条件可用于判断 pd, 表示当消息 m 的投票值小于本地投票值 localState.myVote 时直接丢弃该消息.

其次, 崩溃-消息独立性 (crash-message independence, CMI) 关注崩溃事件与消息的独立性. 传统 DMCK 并未建模节点崩溃事件的语义, 因此需与所有事件重排序, 加剧状态爆炸, 导致许多 DMCK 仅能注入极少量崩溃事件

并依赖不可靠约减策略加速缺陷发现. SAMC 识别到崩溃事件可能产生两类影响: 1) 全局影响 (predicate global impact, pg), 即若节点 X 的崩溃会导致节点 N 向其他节点发送消息, 则 X 的崩溃对 N 产生全局影响; 2) 局部影响 (predicate local impact, pl), 即若节点 X 的崩溃不会导致节点 N 进一步发送消息, 则 X 的崩溃仅对 N 产生局部影响. 基于此, SAMC 定义崩溃事件与消息的独立性: 若崩溃事件会产生全局影响 (pg), 则需与所有消息重排序, 存在依赖关系; 若仅产生局部影响 (pl), 则与其他消息独立, 无需重排序. CMI 特别适用于基于 quorum 机制的容错分布式系统. 用户可通过 #follower >= majority 建模 pl, 反转条件则可用于建模 pg.

FlyMC[67]将这些独立性策略进一步推广为更广义的事件独立性, 包括同一节点上更新不同变量的消息之间的独立性, 以及与崩溃事件相关的独立性. 在分布式协议中, 多个协议或模块可能同时运行, 导致同一节点可能接收并处理属于不同协议的消息. 例如, ZooKeeper 的 Zab 协议[6,10]同时包含原子广播协议和选主协议, 相关消息仅修改各自协议中的变量, 因此彼此独立. 为支持此类优化, FlyMC 设计了一套机制, 允许开发者标注不相互影响的消息, 并利用静态分析自动转换为系统语义信息模型.

具体而言, FlyMC 通过静态分析维护 readSet 和 updateSet, 用于判断消息是否独立. 若两条消息处理的 readSet 和 updateSet 变量无交集, 则它们独立. 一个特殊规则是, 若两条消息的 updateSet 变量完全相同, 且所有变量的变更方式均为 +1 或−1, 则它们仍然独立. 这种模式在分布式协议中常见, 例如 ack++ 统计确认数. 对于崩溃事件, FlyMC 将其对其他节点的影响抽象为 updateSet 变量的变更. 例如, 在集群中, 若一个跟随者节点崩溃, 领导者维护的 liveNodes 变量减少 1 (liveNodes– –), 则该跟随者崩溃事件的 updateSet 包含领导者 liveNodes 变量.

在 SAMC 和 FlyMC 之前, 一些研究已开始利用系统语义信息进行状态约减[59,75], 然而, 这些工作针对的是编程语言实现的分布式协议, 而非 ZooKeeper 等工业级分布式系统. 例如, MP-Basset[59]主要针对基于消息通信的容错分布式协议 (如 Paxos) 进行模型检验优化. 这类协议的典型特征是接收多条消息、修改状态并向网络发送消息, 而一次状态更新通常需要多个消息共同触发. 例如, 在 Paxos 共识协议中, 提案者需等待大多数 (quorum) 接受者的响应才能进行状态转移.

基于这一观察, MP-Basset 提出了 quorum transition 概念, 在单次状态转移中同时处理多个消息, 从而跳过中间状态, 避免传统方法因逐条处理消息而导致的状态爆炸. 为支持此机制, MP-Basset 在消息传递模型中引入 quorum transition, 并设计了 MP 语言用于建模此类语义. MP 语言提供 @message 注解标记消息处理的状态转移函数, 并使用 @guard 注解定义执行前需满足的条件 (如收到足够数量的消息). 此外, MP-Basset 采用 transition refinement (转移精化) 技术调整状态转移粒度, quorum transition 是 transition refinement 的特例, 使偏序约减技术能够利用精化后的独立事件, 从而优化状态空间. 实验表明, MP-Basset 最大减少了 92% 的内存开销和 85% 的时间消耗. 本质上, 该方法利用的独立性是 SAMC 和 FlyMC 所描述的同一节点内消息独立性的特例.

MP-Basset 的后续研究进一步提出了 LPOR (local partial-order reduction) 框架[75], 基于静态偏序约减优化领域相关等价状态空间. LPOR 允许开发者将系统语义编码为 Can-Enable 关系和 Dependency 关系, 以捕捉消息传递系统中的事件依赖性. 实验表明, LPOR 在验证复杂协议时可实现高达 94% 的状态空间约减. 然而, 该方法主要聚焦于理论研究, 在真实分布式系统环境中的模型检验应用仍然存在挑战.

4.2.2 对称性约减

对称性约减技术在传统的有状态搜索中被广泛应用[96,106]. 分布式系统具有大量对称性, 例如节点角色的对称性. 然而, 在 SAMC[11]之前, 较少有 DMCK 利用对称性约减. 这主要是因为 DMCK 早期研究更侧重于提高工具的有效性, 而能够验证真实分布式系统的 DMCK 采用无状态设计. 传统对称性约减通常依赖于有状态搜索中的状态标准化, 以消除重复状态, 而在无状态搜索中直接应用较为困难.

SAMC 定义了崩溃恢复对称性 (crash recovery symmetry, CRS) 和重启同步对称性 (reboot synchronization symmetry, RSS), 两者原理相似, 这里以崩溃恢复对称性为例. DMCK 在原理上需要遍历所有可能的消息组合, 而崩溃事件是由 DMCK 主动注入的, 因此可以减少不必要的注入. 例如, 在一个包含 4 个节点的分布式系统中, n1–n3 均为跟随者 (follower), n4 为领导者 (leader). 如果 n1–n3 中任意一个节点崩溃, 系统的恢复行为相同, 则仅需注入一次崩溃事件.

SAMC 通过恢复抽象全局状态 (recovery abstract global state, RAGS) 来实现这一优化, 其中, 抽象状态表示只包含部分变量的状态子集. 首先, 如果两个恢复动作产生相同的消息, 并以相同方式改变恢复抽象局部状态 (recovery abstract local state, RALS), 则它们被视为对称的. 随后, SAMC 分析恢复相关代码, 并计算不同节点在该崩溃事件下的恢复代码如何影响 RALS 及可能发送的网络消息. 然后, 所有 RALS 经过排序后合并, 形成 RAGS. 最终, 在注入崩溃事件前, 若某个恢复抽象全局状态已出现过, 则该崩溃事件不再被注入. 本质上, SAMC 的崩溃恢复对称性处理方式与有状态搜索中的状态标准化 (state canonicalization) 类似或部分实现了这一过程. 通过对恢复抽象局部状态的排序和合并, 消除了节点 ID 的影响, 相当于在等价错误注入中选取一个代表, 并将该抽象状态用于对称性约减中的状态去重. 用户需建模各节点的 RALS 变量及其在崩溃恢复事件中的更新方式, 以使用该约减策略.

FlyMC 进一步泛化了对称性, 涵盖通信对称性和状态对称性. 分布式系统中的节点通常具有相同角色 (例如跟随者), 且其状态转移仅取决于接收消息的内容, 而与发送或接收节点的 ID 无关. FlyMC 通过记录抽象状态-事件转移历史来识别对称性. 其中, 抽象状态去除了与节点 ID 相关的信息, 并对各节点的状态进行排序后合并. 后续状态空间探索时, 若遇到相同的抽象状态-事件转移, 则表明该执行路径已被探索, 从而避免冗余计算. 本质上, FlyMC 的方法与 SAMC 类似, 都借鉴了有状态搜索中的对称性处理方式, 以减少等价状态遍历.

MP-Basset[59]的后续工作 DBSS (decomposition-based stateful search)[60]采用状态分解 (state decomposition) 方法, 将协议状态划分为相关变量 (relevant) 和辅助变量 (auxiliary). 其中, 相关变量影响协议行为, 而辅助变量不会影响协议行为. 在状态去重过程中, 仅考虑相关变量. 在计算状态哈希前, DBSS 通过状态标准化选择对称状态中的代表性状态进行存储和去重. 该方法要求被忽略的辅助变量不会影响未来状态转移, 而这取决于系统本身及待检查的属性, 因此需要具有系统领域知识或语义信息的专家进行分解. DBSS 相比传统方法约减了高达 39% 的状态空间, 但该工作主要侧重于算法的可靠性 (soundness) 证明, 尚未在大规模分布式系统的模型检验中广泛应用.

4.2.3 垂直领域约减

对于完整的分布式系统状态空间, 模型检验难以穷尽所有路径. 垂直领域约减聚焦于特定系统行为或故障模式 (如错误注入、容错逻辑等), 结合领域特定的语义信息, 剪裁与目标属性无关的状态空间, 从而显著提升验证效率. 这类语义信息通常来源于对垂直领域的专门建模过程, 本质上将领域知识直接编码进模型中, 使状态空间探索更具针对性. 例如, SAMC、FlyMC 和 MP-Basset 利用容错逻辑、错误注入和协议级语义, 在人工建模中主动排除了与验证目标无关的状态空间, 主要包括某些局部事件引起的线程调度交织, 从而聚焦于垂直领域相关的关键行为.

此外, 一些研究借鉴模型检验的基本原理, 聚焦于特定垂直领域, 通过引入领域特定的语义信息, 仅探索与该领域相关的状态, 或仅检测特定属性的违反, 从而有效剪裁与验证目标无关的状态空间, 提升缺陷发现效率或正确性保障能力. 例如, 错误注入已成为分布式系统中一个典型的垂直领域研究方向. 尽管此类工作通常不属于严格意义上的 DMCK (例如不支持确定性模拟执行), 但在垂直领域约减下依然具有重要的实践价值. 下面将简要介绍几项具有代表性的研究.

FATE/DESTINI[107]借鉴模型检验的穷举搜索思想, 系统性探索故障序列 (FATE) 并使用 Datalog 规约 (DESTINI) 检查正确性. FATE 通过 failure ID 抽象错误场景, 结合静态信息 (源文件、调用)、动态信息 (调用栈、节点 ID) 以及领域相关信息 (消息类型等). 由于暴力枚举带来的状态爆炸问题, FATE 利用领域信息优先探索更可能增加覆盖率的故障序列来优化搜索. DESTINI 采用 Datalog 语言, 解决了如何精确定义期望行为以辅助恢复代码测试的问题. FATE/DESTINI 在 HDFS 等工业系统中发现了 16 个新缺陷, 并复现了 51 个已知缺陷. 不同于 FATE 的正向搜索策略, Molly[108]采用 LDFI (lineage-driven fault injection), 从正确执行结果反向推导关键故障点, 逐步加深注入层级, 以减少错误注入次数, 并保持类似于模型检验的覆盖完整度. LDFI 基于 Dedalus 语言高效提取数据谱系, 结合 SAT 求解器快速识别潜在错误组合, 避免冗余探索. Modulo[109]关注分布式系统状态收敛性, 提出基于模型的测试方法, 发现收敛失败缺陷 (convergence failure bug, CFB). 与 DMCK 逐步操控每个不确定性事件的方式不同, Modulo 通过构建发散-收敛模型减少状态空间. 其方法由抽象执行模型 (AEM) 和具体执行模型 (CEM)

组成, AEM 描述触发发散和收敛的条件, 例如 ZooKeeper 依赖 quorum 机制接受写操作并可能导致发散; CEM 则将 AEM 生成的事件映射到具体操作, 从而探索边缘情况. 通过这一方式, Modulo 避免了冗余探索, 高效发现分布式系统的收敛性问题.

5 模型层与代码层互动的分布式系统模型检验

系统语义感知的 DMCK 通过人工建模增强了语义信息利用, 但状态爆炸问题依然严峻, 不仅由于状态空间庞大, 还因探索速度受限. 传统模型检验在状态空间探索方面具备优势, 但人工建模成本高, 且可能与实现代码不一致. 尽管早期研究尝试从代码中抽取模型[110–113], 但尚无工作能直接适用于复杂的分布式系统.

在分布式系统, 尤其是共识协议的实现中, 传统测试难以有效验证设计与实现的正确性. 近年来, 手写核心协议的模型并进行模型检验逐渐成为业界的最佳实践. 例如, Amazon[22]、Azure[23]、MongoDB[24]和 DeepSeek[26]等公司采用 TLA+/PlusCal[34,35,114]、P[115]等形式化语言, 验证关键分布式系统的核心模块. TLA+的广泛应用, 既源于其发明者 Leslie Lamport 在分布式系统领域的深厚影响, 也得益于 TLA+规约语言的轻量设计以及 Amazon 成功经验的推动[22]. 鉴于 TLA+对分布式系统验证的深远影响, 本节沿用 TLA+中对规约 (specification) 的广义定义, 即规约不仅描述系统的正确性属性, 还包括系统行为与环境建模. 此后出现的面向分布式系统的模型检验语言, 如 P 语言[115], 也在一定程度上借鉴 TLA+的设计理念. 然而, 即便形式化模型已广泛存在, 模型与代码之间仍可能存在语义鸿沟 (semantic gap), 导致行为不一致. 如何实现模型层与代码层有效互动, 既充分发挥模型层模型检验的优势, 又能在实现层实现代码级的模型检验 (DMCK), 成为高性价比的技术方向, 并催生了多项相关研究.

本节将探讨模型层与代码层互动的 DMCK (简称互动式 DMCK) 关键技术, 涵盖规约的内容、定位与语言选择, 解决模型层与代码层之间语义鸿沟的两种方法, 以及最终在代码层实现互动式执行的机制. 图 7 展示了其整体框架, 核心思想是通过形式化规约与一致性比对技术, 建立模型与代码之间的有机互动, 并将状态空间探索任务交由传统模型检验工具完成.

图 7 互动式 DMCK 框架图

5.1 系统建模与规约技术

编写分布式系统规约已成为行业最佳实践, 但如何构建高质量的规约仍面临多个关键问题: 规约应包含哪些内容? 如何确定规约的层次与粒度? 业界主要采用哪些语言? 本节围绕这些问题, 探讨相关研究的解决方案.

5.1.1 规约的内容

本文沿用 TLA+ 对规约 (specification) 的定义, 将其划分为系统行为、环境和正确性属性这 3 部分, 一个概要的 Zab 协议的 TLA+ 规约如图 8 所示, 分别体现了消息处理的系统行为、节点宕机的环境行为, 以及仅存在一个合法主节点的正确性属性.

系统行为规约描述系统的核心行为, 例如分布式协议 (Paxos[8]、Raft[9]、Zab[6,10]、CRDT[116]等)、业务逻辑和容错机制, 具体内容因系统而异. 环境规约则刻画分布式环境的高度不确定性, 包括网络消息分发、用户负载、节点故障和网络错误等因素. 正确性属性规约用于定义系统的正确性要求, 通常分为安全性 (safety) 和活性 (liveness), 其详细内容在第 3.3 节已有讨论.

```
(* 模型配置常量 *)
CONSTANTS Servers, MaxCrashes, MaxEpoch
CONSTANTS LOOKING, NOTIFICATION_MSG, FOLLOWERINFO_MSG

(* 变量 *)
VARIABLES role, messages, epochLeader

(* 系统行为 *)
HandleMsg == \E m \in messages:
  CASE m.mType = NOTIFICATION_MSG -> HandleNotMsg(m.dst, m)
    [] m.mType = FOLLOWERINFO_MSG -> HandleFInfoMsg(m.dst, m)
    [] OTHER -> Assert(FALSE, "Error: unknown msg")

(* 环境行为 *)
NodeCrash == \E s \in Server: Crash(s)

(* 正确性属性 *)
LeadershipInv ==
  \A epoch \in 1..MaxEpoch: Cardinality(epochLeader[epoch]) <= 1

(* 初始状态和状态转移 *)
Init ==
  /\ role = [s \in Server |-> LOOKING]
  /\ epochLeader \in [1..MaxEpoch -> SUBSET Server]
  /\ messages = [s \in Servers |-> [d \in Servers \ {s} |-> <<>>]]

Next == \/ HandleMsg \/ HandleTimeout \/ ClientRequest
        \/ NodeCrash \/ NodeStart
```

图 8 TLA+ 规约内容概要

5.1.2 规约的定位

规约作为对目标系统的抽象, 其系统行为规约可根据定位和使用场景分为高层次规约和低层次规约. 高层次规约侧重于抽象描述系统的核心协议和正确性属性, 通常用于验证分布式协议设计的正确性, 而低层次规约更贴近实现, 关注具体代码层面的特定逻辑和实现方式, 以支持一致性检查和代码级验证.

(1) 高层次规约

在分布式系统的设计阶段, 关键协议的正确性验证至关重要. 高层次规约用形式化方式精确描述核心协议逻辑和正确性属性, 避免非正式文档的模糊性和歧义, 同时省略非核心模块和无关细节, 保持高可读性和简洁性. 这类规约通常在开发初期用于验证设计意图, 发现潜在问题, 并为后续实现提供无歧义的“超级文档”.

共识协议的验证实践: Paxos[8]和 Raft[9]等重要分布式协议采用了模型检验验证设计的正确性. Paxos 由 Lamport[36] 提出, 并使用 TLA+ 进行验证, 这种方式比手写数学验证可靠[117], 比定理证明轻量, 后续许多共识协议的设计 (如 Raft) 也沿用了这一方式[37,118]. MongoDB 使用 TLA+ 验证其基于拉取的 Raft 变体[24], 阿里巴巴 PolarDB[119]、腾讯 PaxosStore[120]、Kafka[121]、TiKV[25]、Microsoft CCF (confidential consortium framework)[122]等系统亦采用 TLA+ 进行验证其改进后的核心协议. 开源项目如 etcd[5]和 ZooKeeper[6,10], 社区也提供了 TLA+ 规约以检验其实现和改进的 Raft[123]和 Zab 协议[73,124].

近年来, 越来越多的工业界公司开始在工业级系统的开发与验证中引入模型检验方法, 并取得了显著成效. 本文梳理了若干具有代表性的实践案例如下.

● Amazon. 2014 年, AWS 团队分享了 TLA+ 在复杂系统设计中的成功经验[22]. 他们发现, TLA+ 规约 (通常仅数百行) 能够发现其他方法难以发现的深层缺陷, 即使未发现缺陷, 也能显著增强信心并得以支持激进优化. TLA+ 在 Amazon 内部推广后, 成为复杂系统设计流程的一部分. 此外, AWS 还开发了轻量级形式化方法, 利用 Rust 语言直接编写模型[125], 以提升规约编写效率, 并借此避免了 16 个缺陷进入产品.

● Microsoft. Azure Cosmos DB 采用 TLA+ 建模接口层[23], 规约仅 390 行, 却帮助解释了一个持续 28 天的高优先级故障的根本原因. 然而, 团队仍担忧规约与代码缺乏互动可能导致遗漏缺陷. Microsoft CCF 采用 TLA+发现

了 6 个缺陷[122]. 此外, Microsoft Research 开发了 P 语言[115], 设计上借鉴了 TLA+.

● MongoDB. 在 2016–2019 年间, MongoDB 团队花费数年排查出 5 个严重缺陷, 使用 TLA+后, 仅用数周建模验证便找出这些问题[126]. 他们的经验表明, 在复杂的分布式协议中, 缺乏形式化模型几乎无法保证正确性. 此后, MongoDB 采用 TLA+验证其改进版 Raft 设计[24]并在内部大量使用 TLA+规约.

● DeepSeek. 采用 P 语言验证 AI 基础设施中的高性能分布式文件系统 3FS[26], 涵盖集群管理、元数据、存储服务和客户端等设计, 特别是针对其改良的 CRPQ 协议及 RDMA 软硬件结合设计进行验证.

这些案例表明, 在高层次设计阶段使用模型检验技术能够发现大量设计缺陷, TLA+ 等工具在分布式系统正确性保障中发挥了重要作用, 并已成为业界最佳实践. 然而, 高层次规约在编写时未考虑与代码的互动, 多个工作均指出规约与实现的不一致可能导致缺陷仍然存在于系统实现中[22,23,127].

(2) 低层次规约

尽管已有不少业界成功实践, 但在多数软件开发过程中, 代码实现前并不常见完整的高层次规约. 此外, 随着需求变化、性能优化或代码缺陷, 代码可能偏离原始设计, 即使高层次规约得到验证, 仍无法确保代码的可靠性. 因此, 尽可能贴近代码的低层次规约被提出, 其核心目标是保证规约与代码的一致性, 并提供交互机制以检测二者的语义鸿沟 (第 5.2 节). 在此基础上, 低层次规约通过检查属性是否违反并通过交互机制来确认代码中的缺陷 (第 5.3 节).

MongoDB 团队对产品级系统 Realm Sync 中的操作转换函数进行了 TLA+建模, 采用直接翻译代码为 TLA+的方式, 使规约不仅反映设计, 也包含代码细节[127]. 这一低层次规约发现了一个栈溢出问题, 令开发者震惊, 因为该缺陷此前未被长时间海量的随机测试发现, 且该系统已是成熟的产品级系统. 在翻译代码过程中 MongoDB 团队也发现了多个代码与规约的分歧并进行了修复, 相关技术将在第 5.2 节介绍.

SandTable[71]作为互动式 DMCK 的典型代表, 观察到语义感知的 DMCK 仍面临状态爆炸问题, 并提出将状态探索转移到规约层面, 以突破 DMCK 代码执行效率的瓶颈. 该方法采用低层次规约, 建模代码中的核心状态转移, 包括全局事件中的消息处理、超时、客户端请求、网络错误和节点故障, 同时抽象掉线程调度、消息序列化等不关注部分. 该方法通过低层次规约验证, 发现了 17 个真实代码缺陷, 另有 6 个代码缺陷在规约与代码交互过程中发现.

Remix[73]也是互动式 DMCK 的代表工作, 发现粗粒度规约可能与代码存在分歧, 导致验证结果不准确, 而细粒度规约则容易引发更严重的状态爆炸问题. 为此, Remix 提出了粗细粒度结合的规约方法, 在保持交互性的同时, 避免丧失发现缺陷的能力. 该方法遵循“由粗到细、逐步细化”的开发原则, 支持增量式迭代开发, 兼具轻量级与实用性. 通过该方法, 研究者在 ZooKeeper 中发现了 6 个深层缺陷. 相比 SandTable, Remix 规约方法在框架性和方法学上更完善, 并覆盖更多不确定性事件, 如线程交互、磁盘操作等, 从而揭示更多难以发现的代码缺陷.

低层次规约强调与代码贴近, 因此在发现真实缺陷方面更具实用价值. 从 MongoDB 启发的从代码转译为规约的方式到交互式 DMCK, 相关研究逐步形成规约框架和方法学. 由于其紧密贴合代码, 这些方法都需要提供特定机制, 以支持规约与代码的互动 (第 5.2 节).

5.1.3 规约的语言

我们根据规约所使用的语言类型, 将规约语言归纳为 3 类: 形式化规约语言、综合式规约语言和通用编程语言. 这 3 类语言在表达方式上逐步贴近代码, 使规约更符合程序员的使用习惯, 降低模型与代码交互的成本, 并提高了实用性.

形式化规约语言: 形式化规约语言主要提供高层抽象视角, 帮助开发者简洁地描述系统的核心特性和行为, 适用于快速验证系统设计的正确性. 尽管这些语言也具备低层次规约的能力, 但由于其与编程语言的设计目标不同, 规约与代码间的同步演化和一致性维护是一个挑战.

Alloy[128]是一种轻量级的形式化建模语言, 基于一阶逻辑和关系代数, 使用 SAT/SMT 求解. 它主要用于描述和分析结构化系统 (如数据结构、访问控制等静态性质), 但缺乏显式的并发机制, 不利于分布式系统建模. 尽管 Alloy 6 引入了时序逻辑支持, 能够描述随时间演变的系统, 但其对分布式系统的支持仍有限. 早期有研究使用

Alloy 建模了分布式协议, 如 Zave[117]建模了 Chord 协议并发现了设计缺陷, 这表明即使是经过手写数学证明验证的协议, 仍可能存在设计问题.

TLA+[34]是一种基于数学的轻量级形式化规格语言, 主要用于并发和分布式系统的描述与验证, 由 Lamport 发明, 核心基于时序逻辑 (temporal logic) 和行动逻辑 (actions logic). TLA+规约使用状态-转移的状态机模型, 支持时序逻辑表达 (如 eventually、always 等), 并可通过精化进行层次化建模. PlusCal[114]是 TLA+ 的类 Pascal/C 风格的子语言, 旨在更接近命令式编程, 通过编写伪代码后转换为 TLA+ 进行验证. TLA+/PlusCal 语言简单易学, 并拥有丰富的配套工具, 如 TLC[35] (用于模型检验) 和 TLAPM[129] (用于定理证明). TLC 工具成熟, 支持复杂的属性规约及高效的状态约减技术, 如状态去重和对称约减等.

Amazon 曾调研 TLA+ 和 Alloy 在工业界应用中的适用性, 在综合考虑了建模能力、学习成本和回报比等因素后, 选择了 TLA+[22,130], Amazon 的成功经验进一步在工业界推广了 TLA+ 的使用.

综合式规约语言: 综合式规约语言结合了规约语言和通用编程语言的特征, 是一种领域特定语言. 它一方面为规约提供高层次的抽象框架, 并配备成熟的验证工具, 另一方面其语言风格接近传统编程语言, 减轻了开发者的学习和开发成本. 基于综合式编程语言开发的规约与底层代码实现更为贴近, 有助于减少开发人员在保持一致性方面的负担. 此外, 综合式规约语言还提供面向目标代码的编译工具链, 能够将规约自动编译为基于通用编程语言的目标代码.

P 语言[115]是一种代表性的综合式规约语言, 主要用于分布式 (或事件驱动) 系统建模. 最初由 Microsoft Research 开发, P 语言为分布式系统提供了基于状态机的高层抽象框架. 它采用异步事件驱动模型, 将系统建模为一组互相通信的状态机, 状态机之间通过异步方式进行交流. 与 TLA+ 等基于数学的规约语言相比, P 语言的使用体验更接近传统编程语言. P 语言将正确性规约、系统行为规约和测试场景配置分开, 符合开发者的习惯, 使得规约开发和维护变得更加可控. P 语言编译器可以自动将规约转译为 C、C#、Java 等语言的代码, 确保代码与规约一致. P 语言已得到 Amazon、Microsoft 和 DeepSeek 等公司使用.

MPCal (modular PlusCal)[131]是另一种综合式规约语言, 是 PlusCal 语言的模块化扩展. MPCal 将系统规约和环境规约模块化, 从而提高了通用模块的复用性. 在规约过程中, 开发者只需分别定义系统行为、环境行为以及系统的环境依赖. MPCal 的编译工具链 PGo 提供了对规约验证和代码生成的支持. PGo 不仅能将 MPCal 规约自动转译为 PlusCal 代码并生成 TLA+ 供后续验证, 还能将 MPCal 规约编译为基于 Go 语言实现的可执行代码, 确保规约与代码的一致性.

通用编程语言: 除了前述的两类规约语言外, 开发者还可以基于通用编程语言进行规约. 使用通用编程语言进行规约的成本较低, 因为开发者无需专门学习特定的规约语言, 这有助于快速编写规约. 此外, 使用与系统代码相同的编程语言编写的规约, 保持了与系统代码的紧密联系, 有利于规约和代码的同步演化. 然而, 在没有适合的验证工具或框架支持的情况下, 基于通用编程语言的规约可能面临验证能力上的挑战.

在工业界, Amazon S3 团队采用 Rust 语言对其分布式键值存储节点 ShardStore 进行规约与验证[125]. 开发团队为确保轻量级和实用性 (对开发者友好), 选择了与系统实现相同的编程语言 Rust 进行规约. 团队为系统编写了参考模型 (reference model), 用于定义系统的预期行为. 该参考模型的代码量约占系统实现的 1%, 高度简洁, 并由开发团队编写和维护, 确保与代码的同步演化.

Stateright[132]是一个面向分布式系统的 Rust 库, 设计上参考了 TLA+. Stateright 基于异步消息通信的 Actor 框架构建规约, 并提供了几种可配置的网络模型, 支持指定节点崩溃次数以及不同类型的正确性性质. 此外, Stateright 还允许开发者自定义状态空间搜索算法, 为模型检验提供了更高的灵活性. FizzBee[133]是一个面向分布式系统的 Python 工具, 同样参考了 TLA+的设计. FizzBee 采用 Python 语法风格, 更注重易用性和实用性, 旨在降低形式化方法的学习和使用门槛, 帮助开发者在数分钟内上手进行规约编写.

5.2 解决模型层与代码层语义鸿沟的方法

交互式 DMCK 技术需要解决模型层与代码层的语义鸿沟问题, 保障二者的一致性, 才能有效地在代码中发现

缺陷. 因此, 一致性比对技术是交互式 DMCK 的重要组成部分. 根据规约的定位, 一致性比对技术可用于发现代码中的两类缺陷. 对于高层次规约, 它反映系统的设计需求, 其行为通常被视为正确的, 因此可以作为正确性判据 (oracle). 通过一致性比对, 代码中任何与高层次规约不一致的行为都可以视为代码的缺陷. 相比之下, 低层次规约紧贴代码实现细节, 本身可能存在问题, 因此需在一致性比对后, 进一步借助属性规约验证是否违反预期属性. 由于模型与代码已完成交互, 任何违反属性的路径也会出现在代码中. 一致性比对技术有两种方式: 自顶向下与自底向上. 自顶向下的一致性比对从模型生成执行轨迹, 检查该轨迹是否能够在代码中成功重现. 自底向上的一致性比对则从代码生成执行轨迹, 并在模型层进行比对检查.

5.2.1 自顶向下的一致性比对方法

自顶向下的一致性比对方法通过在代码中复现模型层的执行轨迹来实现. 其主要流程如下: 首先, 对规约进行模型检验, 生成模型层的所有执行轨迹; 然后, 在目标系统中重放这些轨迹 (重放技术依赖于第 3.1 节中的确定性模拟执行技术), 并检查每一步的动作和状态是否一致. 如果发现不一致, 则需要人工介入分析分歧的原因. 如果使用的是高层次规约, 不一致表明是代码中的实现缺陷, 需要修复代码使其与高层次规约一致; 如果使用的是低层次规约, 不一致则说明规约本身存在转译错误, 需要修复规约. 此过程需持续迭代直到所有生成的执行轨迹都完成比对.

然而, 在实践中, 高层次规约常常面临规约本身质量不高或与代码分歧过大的问题, 难以直接确定分歧是否由代码缺陷引起. 此外, 将规约层的所有执行用例复现到代码中还会面临状态爆炸问题. 因此, 当前的工具通常只针对低层次规约的较小规模模型配置或部分状态空间生成执行轨迹, 通过规约分支覆盖率或在一定时间内未发现不一致的情况来判断规约是否达到高质量.

SandTable[71]是一个典型的交互式 DMCK 技术. 它的动机是现有的 DMCK 技术在代码层直接进行状态空间探索时速度过慢, 因此选择在规约层进行状态空间探索, 以显著加速 DMCK 状态探索速度, 并有效利用规约层的高级状态约减技术 (如 TLC 的对称约减). SandTable 注意到, 现有代码不一定有高层次规约, 或代码实现很可能与现有的高层次规约不一致, 因此选择使用低层次规约, 并手动为系统开发这些规约. 开发方式基于 SandTable 支持的全局事件类型来编写对应的事件动作. 理论上, 一致性比对过程需要对整个状态空间生成代码层的执行轨迹, 但这在实际中并不现实. 因此, SandTable 通过随机模拟测试 (simulation testing) 来生成执行轨迹, 模拟测试的配置通过启发式算法优化, 从而快速覆盖更多的规约分支. 所有比对不一致的情况 (discrepancy) 都视为规约转译错误, 修复后, 若一定时间内未发现不一致, 则认为规约质量已达标. 对模型检验中发现的缺陷可以直接使用确定性模拟执行技术在代码中复现. SandTable 首次提出并建立了交互式 DMCK 的工作流程, 在模型检验过程中发现了 17 个代码设计缺陷, 并在一致性比对中发现了 6 个代码实现缺陷.

SandTable 的后续工作 Remix[73]也是一种交互式 DMCK 技术, 其一致性比对流程与 SandTable 相似. Remix 的核心在于通过粗细粒度的规约组合来平衡状态爆炸问题和规约与代码语义鸿沟问题, 并提出了一种保交互的粗细粒度组合规约方式 (见第 5.1.2 节).

5.2.2 自底向上的一致性比对方法

自底向上的一致性比对方法通过在规约的状态空间中对所有可能的代码执行轨迹进行验证, 从而建立代码到高层次规约的精化 (refinement) 关系[134]. 该方法理论上需要穷举代码层的状态空间 (通过传统 DMCK 技术进行系统性探索), 对每条执行轨迹进行采集, 并将其中涉及的变量状态转换为规约中变量的对应表示, 从而在模型层验证该轨迹是否满足高层规约定义的精化条件.

这一思路最早在文献 [135,136] 中初步体现, 但这些工作的研究重点不在于轨迹生成的穷举性策略. 而且, 穷举代码状态空间的开销极高, 导致其在实际缺陷发现场景中性价比较低, 不如直接采用传统 DMCK 验证属性的方式有效和高效. 因此, 目前尚无工作严格依此思路进行完整验证. 相比之下, 放弃穷举探索、只验证部分运行轨迹是否与规约一致的轻量化派生技术轨迹确认 (trace validation)[137]被广泛采用, 相关内容将在第 6.2 节中进一步讨论.

5.3 互动式的代码层执行

互动式 DMCK 的核心机制在于: 在保证模型和规约准确的前提下, 将状态空间的探索卸载 (offloading) 到模

型层, 代码层则通过确定性模拟执行进行重放, 以支持一致性比对和缺陷确认. 在一致性比对阶段, 模型与代码的互动主要依赖两类技术: 自顶向下 (第 5.2.1 节) 和自底向上 (第 5.2.2 节). 理论上, 这两类技术均可支持高层次和低层次规约的互动, 但在实践中, 由于高层次规约与代码实现差距较大, 自顶向下技术的互动效果往往不理想. 已有工作多采用低层次规约 (如 SandTable 与 Remix). 而在处理高层次规约时, 自底向上技术通常会放宽条件, 转而采用轨迹确认技术 (第 6.2 节).

低层次规约完成一致性比对后, 规约达到较高质量, 探索状态空间的代价得以从代码层卸载到模型层. 模型层可直接借助成熟的规约级模型检验工具, 这些工具通常集成了高效的状态约减技术 (如状态去重、对称约减、偏序约减等), 且避免了直接运行代码的高成本, 实现更快速的缺陷发现. 例如, SandTable 实验表明规约层探索状态空间的速度相较代码层提升了 114–2 989 倍.

在模型层发现缺陷后, 相关技术通常进一步在代码层重放缺陷事件序列, 确保缺陷能准确复现, 以便于理解、调试和修复. 这一过程中, 确定性模拟执行 (第 3.1 节) 技术再次发挥关键作用, 实现了模型与代码之间的精准互动.

6 分布式系统模型检验派生技术

一些方法使用了部分 DMCK 关键思想, 在不同环节进行适当改进后, 形成了一些更适用于特定场景的变体方法. 我们将这类方法统称为 DMCK 的派生技术. 主要包括两类: 一是模型检验驱动的分布式系统测试 (model-checking-driven testing), 二是分布式系统的轨迹确认技术 (trace validation).

6.1 模型检验驱动的分布式系统测试

测试技术因其高效性和实用性, 在分布式系统中被广泛用于缺陷发现. 许多工作采用随机、覆盖率反馈等方式运行系统并注入故障, 如 Jepsen[15]、Chaos[16]、CrashFuzz[17]等. 这些方法无需对分布式系统的执行进行确定性控制, 也不依赖穷尽式的状态空间探索, 因此实现成本较低. 然而, 它们通常面临缺少正确性判据 (oracle) 的问题, 且在发现深层次边缘缺陷、复现缺陷和提供正确性保障方面存在不足.

为了进一步发现深层次复杂缺陷, 必须生成高质量的测试用例, 而这往往是测试技术中的瓶颈. 由于规约技术已成为验证分布式系统核心协议正确性的主流手段, 直接复用模型检验过程中的状态空间信息, 生成高覆盖率、高质量的测试用例, 是一种性价比极高的方案. 由于分布式系统的不确定性, 模型检验驱动的测试方法通常需配合确定性模拟执行技术, 以保证生成的测试用例在真实系统中复现, 这种方式类似于自顶向下的一致性比对技术.

MET (model-checking-driven explorative testing)[138]通过模型检验验证了其设计的新的 CRDT 协议的正确性, 并将模型检验结果用于指导测试. 该工作基于 Redis 实现了设计的 CRDT 协议, MET 手动插桩了 Redis 系统, 使得客户端操作、消息传递、网络错误等事件被确定性模拟操控执行, 从而驱动测试用例的执行. 这种方法不仅在规约层确保设计正确的基础上, 也大幅提升了对实现正确性的信心.

Mocket (model checking guided testing)[82]要求用户提供高层次、已验证的 TLA+ 规约, 使用模型检验工具探索状态空间并保存状态空间图, 再通过图遍历算法生成测试用例. Mocket 利用 ASM 字节码插桩技术支持对分布式系统的确定性模拟执行, 并对执行结果进行一致性检查, 不一致即意味着潜在缺陷. Mocket 应用于 ZooKeeper 和多个 Raft 系统中, 其中 Raft 系统直接使用了 Raft 作者提供的高层次规约. Mocket 成功发现或复现了 9 个难以通过随机测试发现的缺陷. 然而, 已有的或自行编写的高层次规约并不总是高质量的, 不一致可能来自规约自身问题、代码实现与规约设计偏离等. 因此 Mocket 发现不一致后, 仍需人工分析是误报、代码缺陷还是规约错误. 其实验结果表明, 3 类情况均有发生.

6.2 分布式系统轨迹确认技术

自底向上的一致性比对技术虽理论完备 (第 5.2.2 节), 但在实际应用中成本极高而不切实际. 后续研究发展出分布式系统轨迹确认技术 (trace validation), 通过放弃穷举探索或放宽轨迹与状态空间的匹配条件, 大幅提升了其实用性和有效性.

这一方法最早由文献 [135,136] 描述, 类似于构建精化关系 (refinement)[134]. 但这些相关研究的研究重点不在轨迹生成策略上, 通常简单地采用随机等策略来生成执行轨迹. 该技术在分布式系统中的首次应用是在 MongoDB 中, 并被称为 MBTC (model-based trace checking)[127]. MBTC 中的代码执行轨迹通过现有的集成测试和随机测试生成, 而规约则复用了 MongoDB 设计的基于拉取的 Raft 改进版协议 TLA+ 高层次规约. MBTC 的验证过程使用了 TLA+ 传统的精化映射方法, 以验证代码执行轨迹是否符合高层次规约的精化关系.

然而, MongoDB 团队发现 MBTC 实施效果不佳. 尽管投入了大量的人力和时间, 最终并未达到预期效果. 在此过程中, 有大量的模型与代码不一致的情况, 但这些不一致经过人工分析后并非代码缺陷, 反而需要通过避免产生不一致的执行轨迹、修改规约或强行模拟一致等方式来解决问题. 最终, 5 个测试中只有一个通过了 MBTC 的一致性比对. 开发者意识到, 规约必须与代码更加接近才能让这种技术更具实用性. 尽管 MongoDB 团队随后重新编写了更贴近代码的 Raft 规约, MBTC 仍未成功实施. 这主要因为精化关系验证过于严格, 要求每条轨迹中包含规约中的全部变量, 而实际中很难完整获取所有变量. 同时, TLC 工具当时缺乏对轨迹验证的关键支持.

后续研究通过放宽轨迹与状态空间的匹配条件, 大幅提升了其实用性和有效性. 该方法的核心思想是, 执行轨迹只需提供部分变量状态, 由模型检验工具自动推导缺失变量, 从而构造一个状态空间. 只要该状态空间与规约的状态空间存在交集, 即认为该轨迹与规约一致.

SEFM[137]工作基于 TLA+ 提出了自动推导缺失变量的轨迹确认框架, 框架支持从分布式系统收集日志并在 TLC 工具中验证轨迹与规约的一致性. 该流程首先对被测系统进行随机执行 (例如使用被测系统的端到端测试套件等, 不需要确定性模拟执行技术), 随机执行过程中可随机注入节点故障和网络错误. 然后收集各节点日志, 日志需记录关键事件名与部分变量信息. 随后, 分布式系统各个节点的所有事件按时间戳 (或逻辑时钟) 全局排序, 生成一条串行执行路径. 轨迹确认阶段中, 该路径被输入 TLC 工具, 每个事件和变量赋值被编码到规约的状态转移使能条件中, 从而将状态空间限制在当前轨迹事件允许的状态范围内 (轨迹中记录的变量越多, 则限制的状态空间越精确). 若 TLC 探索的状态抵达轨迹末尾的事件, 则视为轨迹与规约一致; 若提前终止状态探索, 则表明存在不一致.

SEFM 所提出的轨迹确认技术已在微软 CCF 项目[122]和开源 etcd[123]中取得成功. 在 CCF 中, 该技术被称为 Smart Casual Verification, 即介于严格和不严格之间的轻量级验证技术. 开发者为共识模块和客户端一致性模块编写了高层次 TLA+ 规约, 并复用原有测试生成日志用于验证. 尽管日志中不包含全部变量的赋值, TLC 工具依然能根据规约和轨迹使能条件限制有效推导缺失变量的可能状态. 根据我们的观察, 轨迹确认还缓解了规约与代码粒度不一致问题, 可通过对规约的动作合并和对轨迹的事件筛选来对齐二者粒度, 使得高层次和低层次规约均可用于真实代码的比对. CCF 团队通过模型检验和基于 Q-Learning 的模拟测试 (simulation testing) 等方式, 发现了 6 个缺陷. 尽管轨迹确认技术无法自动复现这些缺陷, 但缺陷可通过已有测试或手动复现方式确认.

总体而言, 轨迹确认技术作为互动式 DMCK 的实用化变体, 有效利用了模型层的状态探索能力, 又避免了代码层确定性控制的高成本. 其在 TLA+ 工具链中已逐渐发展成熟, 并因其易用性和粒度对齐的灵活性, 在工业界逐渐获得认可和采用.

7 总结和展望

分布式系统作为现代计算基础设施的核心, 其可靠性与正确性至关重要. 然而, 由于运行环境的不确定性以及代码设计与实现的高度复杂性, 验证其正确性始终是一项极具挑战的任务. 分布式系统模型检验 (DMCK) 通过穷尽式状态探索, 有效应对了不确定性所带来的“缺陷难发现、难诊断、难修复”等难题. 本文梳理了 DMCK 的关键研究进展, 归纳为 3 个阶段, 并在此基础上分析当前存在的局限与未来发展方向.

7.1 分布式系统模型检验工作的总结

分布式系统模型检验技术在早期缓解了建模成本高和模型与代码存在语义鸿沟的问题, 但也带来了严重的状态爆炸挑战. 通用状态约减方法逐渐达到瓶颈, 引入适量人工的系统特定语义优化成为有效补充, 显著缓解了该问题. 随后, 形式化规约的引入以及规约与代码互动技术的推进, 进一步提升了 DMCK 效率与可用性. 在这一演进过

程中, DMCK 技术始终围绕“状态爆炸”与“人工成本”之间的权衡展开, 其发展大致可划分为 3 个阶段.

第 1 阶段聚焦于如何“使 DMCK 有效”. 该阶段的核心突破在于确定性模拟执行与无状态搜索技术以及通用状态约减技术的引入与结合. CMC 通过手动适配支持模拟执行, MaceMC 提供编程框架, MoDist 截获底层系统调用以提升通用性, dBug 手动精准插桩提升操控精度, 这些工作都控制了分布式系统的不确定性. 同时, VeriSoft 与 DPOR 提出的无状态搜索算法和动态偏序约减技术让 DMCK 技术变得实用并成为基础技术. 在状态约减方面, DeMeter、CrystalBall 与 LMC 等工作利用节点状态与系统环境的局部性, 缓解了特定场景下的状态爆炸问题. MoDist 和 FlyMC 等工作优化了代码级状态空间的探索速度. MaceMC 还提出了对活性属性的近似检测策略. 这一阶段 DMCK 工具开始具备应用于真实系统的能力, 但依赖于不可靠的约减策略来加快缺陷发现速度, 存在遗漏缺陷的风险.

第 2 阶段致力于“使 DMCK 高效”. 核心思路是将系统语义信息引入状态空间探索过程, 特别是消息传递、崩溃、同步等事件的对称性与独立性被系统建模并加以利用, 从而实现更大规模的状态约减. 同时, 部分工作利用了模型检验原理还结合了垂直领域知识, 取得了良好效果. 以 SAMC 和 FlyMC 为代表的工作中, 语义感知能力被集成进 DMCK 框架, 在不依赖不可靠约减策略的情况下, 成功发现并复现了多个工业级分布式系统的深层缺陷. 尽管该方法需少量人工标注语义或人工建模, 其性价比已有显著提升.

第 3 阶段聚焦于“增强模型与代码的互动能力”. 该阶段的核心方法是利用形式化规约语言的状态空间探索能力, 并通过将代码执行与规约状态进行一致性比对, 实现代码与模型的互动验证. 尽管这一路径看似“回退”到早期需要手动编写规约的模式, 实则得益于形式化方法自身的成熟与普及, 其整体效能和可用性已大幅提升. 代表性工作包括 SandTable 和 Remix, 以及 DMCK 的派生技术 Smart Casual Verification, 验证了此类方法在实际系统中的可行性与应用价值.

7.2 局限性和未来展望

DMCK 技术的关键局限在于状态爆炸问题、代码操控成本、建模成本以及模型与代码一致性检查等方面: 1) 缺乏开箱即用的代码级 DMCK 工具: MoDist、DeMeter 未开源、SandTable 依赖规约轨迹、dBug、SAMC、FlyMC、Remix 等需手动插桩, 使用门槛偏高. CMC、JPF、Mace 等基于状态的系列工具已逐渐淡出学术界与工业界视野; 2) 状态爆炸问题仍然严重: 虽然已有多种约减与优先探索技术, 但未有工作系统性整合实现, 且实际使用中依然以“发现缺陷”为主, 难以穷尽整个状态空间; 3) 规约开发存在一定成本: 尽管形式化规约+互动验证的路径具备较高性价比, 现有方法如 SandTable 和 Remix 等依然需要用户编写规约, 影响其推广与普及; 4) 规约与代码之间仍存在消除分歧的成本: 尽管互动验证技术已用于缩小差距, 但规约适配、分歧分析仍需人工参与.

上述问题集中反映了当前 DMCK 技术在实用化进程中仍面临多方面挑战, 尤其是在工具可用性、状态空间探索、规约开发与一致性检查等关键环节尚未形成系统性解决方案. 为进一步推动 DMCK 的发展与落地, 研究者已开始从多个角度提出改进方向, 未来的演进可围绕以下几个方面展开.

(1) 构建全能、易用的 DMCK 工具: 在确定性模拟执行方面, 目前尚无工具同时具备良好的易扩展性、可适用性与透明性. 通用型截获技术若能进一步控制底层的不确定性, 有望构建通用的“确定性虚拟机”执行环境. Antithesis 公司提出商业级实现与此方法类似, 尽管状态空间探索尚非全穷尽, 其确定性控制能力已展示出巨大的应用潜力. 未来构建类似的开源平台将具有重要研究价值和产业前景.

(2) 集成多种状态约减与优先探索策略: 当前多数模型检验工具仅采用部分特定的状态空间优化技术, 整体上的状态约减效果与探索效率尚未达到理想水平. 在规约层, TLA+ 等工具已支持多种约减策略, 但如利用事件独立性的偏序约减[139]等仍有待进一步集成 (例如集成到 TLA+ community modules 社区模块[140]). 此外, 优先级探索策略, 如基于 Q-Learning 等机器学习方法的动态调度[122,141], 已开始用于提升缺陷发现效率, 展现出良好潜力. 面向未来, 构建集成多种约减与探索策略的统一工具平台, 将成为该领域的重要研究方向.

(3) 利用大语言模型 (large language model, LLM) 辅助规约开发与语义标注: 规约开发或语义标注是语义感知和互动式 DMCK 推广中的核心瓶颈. 大语言模型为降低人工成本提供了新路径, 已有初步研究探索从代码自动生

成规约的可能性[142]. 尽管生成的规约尚存在语法错误、幻觉及处理长代码片段时准确率下降的问题, 但可预见未来 LLM 生成规约的质量将持续提升. 结合代码-模型互动机制, 还可进一步增强生成规约的正确性.

(4) 面向一致性检查的轻量级形式化方法设计: 规约语言与真实代码之间的差异是导致分歧分析成本高的关键原因. 未来可以通过更贴近代码语义的规约语言设计与工具链建设, 减少手工适配成本. 同时, 在一致性比对中, 引入大语言模型辅助分析、自动修正规约或代码, 也将有助于降低人工成本、提升易用性.

(5) 支持持续集成的增量式 DMCK: 分布式系统在演进过程中, 其代码不断变化, 当前多数 DMCK 工具除上手成本较高外, 往往只在一次性验证场景中使用, 缺乏持续性维护机制, 容易导致规约与代码产生分歧, 进而影响验证结果的可信度. 随着 DevOps 与持续集成 (CI) 流程的普及, 已有研究将互动式 DMCK 与其派生方法集成至 CI 流程中, 推动规约与代码的协同演化[122]. 未来, 结合轻量级形式化方法与自动化工具链的持续集成能力, 将成为 DMCK 工具的重要发展方向.

References

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113. [doi: 10.1145/1327452.1327492]

[2] Corbett JC, Dean J, Epstein M, *et al*. Spanner: Google's globally-distributed database. In: Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation. Hollywood: USENIX Association, 2012. 261–264.

[3] Tanenbaum AS, van Steen M. Distributed Systems: Principles and Paradigms. 2nd ed., Upper Saddle River: Prentice-Hall, Inc., 2006.

[4] Taft R, Sharif I, Matei A, VanBenschoten N, Lewis J, Grieger T, Niemi K, Woods A, Birzin A, Poss R, Bardea P, Ranade A, Darnell B, Gruneir B, Jaffray J, Zhang L, Mattis P. CockroachDB: The resilient geo-distributed SQL database. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 1493–1509. [doi: 10.1145/3318464.3386134]

[5] etcd. 2013. https://etcd.io/

[6] Hunt P, Konar M, Junqueira FP, Reed B. ZooKeeper: Wait-free coordination for Internet-scale systems. In: Proc. of the 2010 USENIX Annual Technical Conf. Boston: USENIX Association, 2010. 145–158.

[7] Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng XR, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I. Apache Spark: A unified engine for big data processing. Communications of the ACM, 2016, 59(11): 56–65. [doi: 10.1145/2934664]

[8] Lamport L. The part-time parliament. ACM Trans. on Computer Systems (TOCS), 1998, 16(2): 133–169. [doi: 10.1145/279227.279229]

[9] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proc. of the 2014 USENIX Annual Technical Conf. Philadelphia: USENIX Association, 2014. 305–320.

[10] Junqueira FP, Reed BC, Serafini M. Zab: High-performance broadcast for primary-backup systems. In: Proc. of the 41st IEEE/IFIP Int'l Conf. on Dependable Systems & Networks. Hong Kong: IEEE, 2011. 245–256. [doi: 10.1109/DSN.2011.5958223]

[11] Leesatapornwongsa T, Hao MZ, Joshi P, Lukman JF, Gunawi HS. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 399–414.

[12] Google Cloud. Google cloud networking incident #19009. 2019. https://status.cloud.google.com/incident/cloud-networking/19009

[13] Alibaba Cloud. Notice on the service interruption in Availability Zone C of the Alibaba Cloud Hong Kong region. 2022 (in Chinese). https://cn.aliyun.com/noticelist/articleid/1061819219.html

[14] Alibaba Cloud. Notice on an abnormality in Alibaba Cloud console services (resolved). 2023 (in Chinese). https://cn.aliyun.com/noticelist/articleid/1064981333.html

[15] Jepsen. Distributed systems safety research. 2013. https://jepsen.io/

[16] Chaos Monkey. 2010. https://netflix.github.io/chaosmonkey/

[17] Gao Y, Dou WS, Wang D, Feng WH, Wei J, Zhong H, Huang T. Coverage guided fault injection for cloud systems. In: Proc. of the 45th Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2211–2223. [doi: 10.1109/ICSE48619.2023.00186]

[18] Graham SL, Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. on Programming Languages and Systems (TOPLAS), 1986, 8(2): 244–263. [doi: 10.1145/5397.5399]

[19] Gordon MJC, Melham TF. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. New York: Cambridge University Press, 1993.

[20] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.

Berlin: Springer, 2004.

[21] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin, Heidelberg: Springer, 2002.

[22] Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M. How Amazon Web services uses formal methods. Communications of the ACM, 2015, 58(4): 66–73. [doi: 10.1145/2699417]

[23] Hackett F, Rowe J, Kuppe MA. Understanding inconsistency in Azure Cosmos DB with TLA+. In: Proc. of the 45th Int'l Conf. on Software Engineering: Software Engineering in Practice. Melbourne: IEEE, 2023. 1–12. [doi: 10.1109/ICSE-SEIP58684.2023.00006]

[24] Zhou SY, Mu S. Fault-tolerant replication with pull-based consensus in MongoDB. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. USENIX Association, 2021. 687–703.

[25] TLA+ specifications for TiDB. 2018. https://github.com/pingcap/tla-plus

[26] P specifications in DeepSeek 3FS. 2025. https://github.com/deepseek-ai/3FS/tree/main/specs

[27] Jhala R, Majumdar R. Software model checking. ACM Computing Surveys (CSUR), 2009, 41(4): 21. [doi: 10.1145/1592434.1592438]

[28] Bu L, Chen LQ, Chen Z, *et al*. Research progress and trend of formal methods. In: China Computer Federation, ed. CCF 2017–2018 China Computer Science and Technology Development Report. Beijing: China Machine Press, 2018 (in Chinese).

[29] Godefroid P, Sen K. Combining model checking and testing. In: Clarke EM, Henzinger TA, Veith H, Bloem R, eds. Handbook of Model Checking. Cham: Springer, 2018. 613–649. [doi: 10.1007/978-3-319-10575-8_19]

[30] Chen YL, Ma FC, Zhou YH, Yan Z, Jiang Y, Sun JG. Survey on dynamic testing technologies for distributed systems. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 2964–3002 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7334.htm [doi: 10.13328/j.cnki.jos.007334]

[31] Ge N, He YK, Zhai SM, Li XZ, Zhang L. Formal verification of consensus protocols: Survey and perspective. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 4989–5007 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6684.htm [doi: 10.13328/j.cnki.jos.006684]

[32] Hou G, Zhou KJ, Yong JW, Ren LT, Wang XL. Survey of state explosion problem in model checking. Computer Science, 2013, 40(6A): 77–86, 111 (in Chinese with English abstract). [doi: 10.3969/j.issn.1002-137X.2013.z1.018]

[33] Wang ZY, Wu SS, Cao QX. Survey on interactive theorem proving based concurrent program verification. Ruan Jian Xue Bao/Journal of Software, 2024, 35(9): 4069–4099 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7138.htm [doi: 10.13328/j.cnki.jos.007138]

[34] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Boston: Addison-Wesley Longman Publishing Co., 2002.

[35] TLC and TLA+ toolbox. TLA+. 2025. https://github.com/tlaplus/tlaplus

[36] Lamport L. TLA+ specifications for Paxos. 2019. https://github.com/tlaplus/Examples/tree/master/specifications/Paxos

[37] Ongaro D. TLA+ specifications for Raft. 2014. https://github.com/ongardie/raft.tla

[38] Lü J, Ma XX, Tao XP, Cao C, Huang Y, Yu P. On environment-driven software model for internetware. SCIENTIA SINICA Technologica, 2008, 38(6): 864–900 (in Chinese). [doi: 10.3321/j.issn:1006-9275.2008.06.005]

[39] Lü J, Ma XX, Tao XP, Huang Y, Yu P, Xu C. Explicit environmental constructs for internetware. SCIENTIA SINICA Informationis, 2013, 43(1): 1–23 (in Chinese with English abstract). [doi: 10.1360/112012-527]

[40] Guo HY, Wu M, Zhou LD, Hu G, Yang JF, Zhang LT. Practical software model checking via dynamic interface reduction. In: Proc. of the 23rd ACM Symp. on Operating Systems Principles. Cascais: ACM, 2011. 265–278. [doi: 10.1145/2043556.2043582]

[41] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558–565. [doi: 10.1145/359545.359563]

[42] Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. In: Proc. of the 5th Symp. on Operating Systems Design and Implementation. Boston: USENIX Association, 2002. 75–88.

[43] Musuvathi M, Engler DR. Model checking large network protocol implementations. In: Proc. of the 1st Conf. on Symp. on Networked Systems Design and Implementation. San Francisco: USENIX Association, 2004. 12.

[44] Killian CE, Anderson JW, Braud R, Jhala R, Vahdat AM. Mace: Language support for building distributed systems. In: Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego: ACM, 2007. 179–188. [doi: 10.1145/1250734.1250755]

[45] Killian C, Anderson JW, Jhala R, Vahdat A. Life, death, and the critical transition: Finding liveness bugs in systems code. In: Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation. Cambridge: USENIX Association, 2007. 18.

[46] Fredlund. McErlang. 2007. https://github.com/fredlund/McErlang

[47] Fredlund LÅ, Svensson H. McErlang: A model checker for a distributed functional programming language. In: Proc. of the 12th ACM

SIGPLAN Int'l Conf. on Functional Programming. Freiburg: ACM, 2007. 125–136. [doi: 10.1145/1291151.1291171]

[48] Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing Heisenbugs in concurrent programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 267–280.

[49] Yang JF, Chen ST, Wu M, Xu ZL, Liu XZ, Lin HX, Yang M, Long F, Zhang LT, Zhou LD. MoDist: Transparent model checking of unmodified distributed systems. In: Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2009. 213–228.

[50] Yabandeh M, Knezevic N, Kostic D, Kuncak V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In: Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2009. 229–244.

[51] Basset: A tool for systematic testing of actor programs. 2009. https://mir.cs.illinois.edu/basset/

[52] Lauterburg S, Dotta M, Marinov D, Agha G. A framework for state-space exploration of Java-based actor programs. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Auckland: IEEE, 2009. 468–479. [doi: 10.1109/ASE.2009.88]

[53] ISP. 2009. https://github.com/cogumbreiro/isp

[54] Vo A, Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM, Thakur R. Formal verification of practical MPI programs. In: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Raleigh: ACM, 2009. 261–270. [doi: 10.1145/1504176.1504214]

[55] Simsa J, Bryant R, Gibson G. dBug: Systematic testing of distributed and multi-threaded systems (software repository). 2010. https://www.cs.cmu.edu/~jsimsa/dbug/

[56] Simsa J, Bryant R, Gibson G. dBug: Systematic evaluation of distributed systems. In: Proc. of the 5th Int'l Workshop on Systems Software Verification. Voncouver: USENIX Association, 2010. 3.

[57] Guerraoui R, Yabandeh M. Model checking a networked system without the network. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2011. 225–238.

[58] MP-Basset. 2011. https://ssg.lancs.ac.uk/research/tools/mp-basset/

[59] Bokor P, Kinder J, Serafini M, Suri N. Efficient model checking of fault-tolerant distributed protocols. In: Proc. of the 41st IEEE/IFIP Int'l Conf. on Dependable Systems & Networks. Hong Kong: IEEE, 2011. 73–84. [doi: 10.1109/DSN.2011.5958208]

[60] Saissi H, Bokor P, Muftuoglu CA, Suri N, Serafini M. Efficient verification of distributed protocols using stateful model checking. In: Proc. of the 32nd Int'l Symp. on Reliable Distributed Systems. Braga: IEEE, 2013. 133–142. [doi: 10.1109/SRDS.2013.22]

[61] Artho C, Hagiya M, Potter R, Tanabe Y, Weitl F, Yamamoto M. Software model checking for distributed systems with selector-based, non-blocking communication. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. Silicon Valley: IEEE, 2013. 169–179. [doi: 10.1109/ASE.2013.6693077]

[62] Leungwattanakit W, Artho C, Hagiya M, Tanabe Y, Yamamoto M, Takahashi K. Modular software model checking for distributed systems. IEEE Trans. on Software Engineering, 2014, 40(5): 483–501. [doi: 10.1109/TSE.2013.49]

[63] Concuerror. 2013. https://concuerror.com/

[64] Christakis M, Gotovos A, Sagonas K. Systematic testing for detecting concurrency errors in Erlang programs. In: Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation. Luxembourg: IEEE, 2013. 154–163. [doi: 10.1109/ICST.2013.50]

[65] SAMC. 2014. https://ucare.cs.uchicago.edu/projects/samc/

[66] FlyMC. 2019. https://ucare.cs.uchicago.edu/projects/FlyMC/

[67] Lukman JF, Ke H, Stuardo CA, Suminto RO, Kurniawan DH, Simon D, Priambada S, Tian C, Ye F, Leesatapornwongsa T, Gupta A, Lu S, Gunawi HS. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In: Proc. of the 14th EuroSys Conf. Dresden: ACM, 2019. 20. [doi: 10.1145/3302424.3303986]

[68] DSLabs. 2019. https://github.com/emichael/dslabs

[69] Michael E, Woos D, Anderson T, Ernst MD, Tatlock Z. Teaching rigorous distributed systems with efficient model checking. In: Proc. of the 14th EuroSys Conf. 2019. Dresden: ACM, 2019. 32. [doi: 10.1145/3302424.3303947]

[70] SandTable. 2024. https://github.com/tangruize/SandTable

[71] Tang RZ, Sun XD, Huang Y, Wei YY, Ouyang LZ, Ma XX. SandTable: Scalable distributed system model checking with specification-level state exploration. In: Proc. of the 19th European Conf. on Computer Systems. Athens: ACM, 2024. 736–753. [doi: 10.1145/3627703.3650077]

[72] Remix. 2025. https://github.com/Lingzhi-Ouyang/Remix

[73] Ouyang LZ, Sun XD, Tang RZ, Huang Y, Jivrajani M, Ma XX, Xu TY. Multi-grained specifications for distributed system model checking and verification. In: Proc. of the 20th European Conf. on Computer Systems. Rotterdam: ACM, 2025. 379–395. [doi: 10.1145/3689031.3696069]

[74] Visser W, Havelund K, Brat GP, Park S. Model checking programs. In: Proc. of the 15th IEEE Int'l Conf. on Automated Software Engineering. Grenoble: IEEE, 2000. 3–12. [doi: 10.1109/ASE.2000.873645]

[75] Bokor P, Kinder J, Serafini M, Suri N. Supporting domain-specific state space reductions through local partial-order reduction. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lawrence: IEEE, 2011. 113–122. [doi: 10.1109/ASE.2011.6100044]

[76] Rodriguez A, Killian C, Bhat S, Kostić D, Vahdat A. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proc. of the 1st Conf. on Symp. on Networked Systems Design and Implementation. San Francisco: USENIX Association, 2004. 20.

[77] Godefroid P. Model checking for programming languages using VeriSoft. In: Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Paris: ACM, 1997. 174–186. [doi: 10.1145/263699.263717]

[78] Liu XZ, Lin W, Pan AM, Zhang Z. WiDS checker: Combating bugs in distributed systems. In: Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation. Cambridge: USENIX Association, 2007. 19.

[79] Geels D, Altekar G, Maniatis P, Roscoe T, Stoica I. Friday: Global comprehension for distributed replay. In: Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation. Cambridge: USENIX Association, 2007. 21.

[80] Yuan XH, Yang JF. Effective concurrency testing for distributed systems. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 1141–1156. [doi: 10.1145/3373376.3378484]

[81] Scott C, Panda A, Brajkovic V, Necula G, Krishnamurthy A, Shenker S. Minimizing faulty executions of distributed systems. In: Proc. of the 13th USENIX Conf. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2016. 291–309.

[82] Wang D, Dou WS, Gao Y, Wu CN, Wei J, Huang T. Model checking guided testing for distributed systems. In: Proc. of the 18th European Conf. on Computer Systems. Rome: ACM, 2023. 127–143. [doi: 10.1145/3552326.3587442]

[83] Geels D, Altekar G, Shenker S, Stoica I. Replay debugging for distributed applications. In: Proc. of the 2006 USENIX Annual Technical Conf. Boston: USENIX Association, 2006. 27.

[84] Zhou JY, Xu M, Shraer A, Namasivayam B, Miller A, Tschannen E, Atherton S, Beamon AJ, Sears R, Leach J, Rosenthal D, Dong X, Wilson W, Collins B, Scherer D, Grieser A, Liu Y, Moore A, Muppana B, Su XG, Yadav V. FoundationDB: A distributed unbundled transactional key value store. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2653–2666. [doi: 10.1145/3448016.3457559]

[85] Antithesis: Autonomous software testing. 2018. https://antithesis.com/

[86] Sled simulation guide (jepsen-proof engineering). 2020. http://sled.rs/simulation.html

[87] MadSim: Magical deterministic simulator for distributed systems in Rust. 2021. https://github.com/madsim-rs/madsim

[88] tokio-rs/turmoil. Tokio. 2022. https://github.com/tokio-rs/turmoil

[89] Liu XZ, Guo ZY, Wang X, Chen FB, Lian XC, Tang J, Wu M, Kaashoek MF, Zhang Z. D^{3}S: Debugging deployed distributed systems. In: Proc. of the 5th USENIX Symp. on Networked Systems Design and Implementation. San Francisco: USENIX Association, 2008. 423–437.

[90] Pshenichkin A. So you think you want to write a deterministic hypervisor? 2024. https://antithesis.com/blog/deterministic_hypervisor/

[91] Clarke EM Jr, Grumberg O, Peled DA. Model Checking. Cambridge: MIT Press, 1999.

[92] Godefroid P. Partial-order Methods for the Verification of Concurrent Systems. Berlin, Heidelberg: Springer, 1996. [doi: 10.1007/3-540-60761-7]

[93] Godefroid P. Using partial orders to improve automatic verification methods. In: Proc. of the 2nd Int'l Conf. on Computer-aided Verification. New Brunswick: Springer, 1991. 176–185. [doi: 10.1007/BFb0023731]

[94] Mazurkiewicz A. Trace theory. In: Brauer W, Reisig W, Rozenberg G, eds. Petri Nets: Applications and Relationships to Other Models of Concurrency. Berlin, Heidelberg: Springer, 1987. 278–324. [doi: 10.1007/3-540-17906-2_30]

[95] Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BYE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D. In defense of soundiness: A manifesto. Communications of the ACM, 2015, 58(2): 44–46. [doi: 10.1145/2644805]

[96] Emerson EA, Sistla AP. Symmetry and model checking. In: Proc. of the 5th Int'l Conf. on Computer Aided Verification. Elounda: Springer, 1993. 463–478. [doi: 10.1007/3-540-56922-7_38]

[97] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Long Beach: ACM, 2005. 110–121. [doi: 10.1145/1040305.1040315]

[98] Yang Y, Chen XF, Gopalakrishnan G, Kirby RM. Efficient stateful dynamic partial order reduction. In: Proc. of the 15th Int'l Workshop on Model Checking Software. Los Angeles: Springer, 2008. 288–305. [doi: 10.1007/978-3-540-85114-1_20]

[99] Trimananda R, Luo WY, Demsky B, Xu GH. Stateful dynamic partial order reduction for model checking event-driven applications that

do not terminate. In: Proc. of the 23rd Int'l Conf. Verification, Model Checking, and Abstract Interpretation. Philadelphia: Springer, 2022. 400–424. [doi: 10.1007/978-3-030-94583-1_20]

[100] Simsa J, Bryant R, Gibson G, Hickey J. Scalable dynamic partial order reduction. In: Qadeer S, Tasiran S, eds. Runtime Verification. Berlin, Heidelberg: Springer, 2013. 19–34. [doi: 10.1007/978-3-642-35632-2_4]

[101] Abdulla PA, Aronis S, Jonsson B, Sagonas K. Source sets: A foundation for optimal dynamic partial order reduction. Journal of the ACM (JACM), 2017, 64(4): 25. [doi: 10.1145/3073408]

[102] Godefroid P. Exploiting symmetry when model-checking software. In: Wu JP, Chanson ST, Gao Q, eds. Formal Methods for Protocol Engineering and Distributed Systems. Boston: Springer, 1999. 257–275. [doi: 10.1007/978-0-387-35578-8_15]

[103] Kokologiannakis M, Marmanis I, Vafeiadis V. Spore: Combining symmetry and partial order reduction. Proc. of the ACM on Programming Languages, 2024, 8(PLDI): 219. [doi: 10.1145/3656449]

[104] Lukman JF. FlyMC technical report. 2019. https://tinyurl.com/flymc-technicalreport

[105] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego: ACM, 2007. 89–100. [doi: 10.1145/1250734.1250746]

[106] Clarke EM, Emerson EA, Jha S, Sistla AP. Symmetry reductions in model checking. In: Proc. of the 10th Int'l Conf. on Computer Aided Verification. Vancouver: Springer, 1998. 147–158. [doi: 10.1007/BFb0028741]

[107] Gunawi HS, Do T, Joshi P, Alvaro P, Hellerstein JM, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Sen K, Borthakur D. FATE and DESTINI: A framework for cloud recovery testing. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2011. 238–252.

[108] Alvaro P, Rosen J, Hellerstein JM. Lineage-driven fault injection. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 331–346. [doi: 10.1145/2723372.2723711]

[109] Kim BH, Kim T, Lie D. Modulo: Finding convergence failure bugs in distributed systems with divergence resync models. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 383–398.

[110] Methni A, Lemerre M, Ben Hedia B, Haddad S, Barkaoui K. Specifying and verifying concurrent C programs with TLA+. In: Artho C, Ölveczky PC, eds. Formal Techniques for Safety-critical Systems. Cham: Springer, 2015. 206–222. [doi: 10.1007/978-3-319-17581-2_14]

[111] Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Păsăreanu CS, Robby, Zheng HJ. Bandera: Extracting finite-state models from Java source code. In: Proc. of the 22nd Int'l Conf. on Software Engineering. Limerick: ACM, 2000. 439–448. [doi: 10.1145/337180.337234]

[112] Havelund K. Java PathFinder a translator from Java to Promela. In: Dams D, Gerth R, Leue S, Massink M, eds. Theoretical and Practical Aspects of SPIN Model Checking. Berlin: Springer, 1999. 152. [doi: 10.1007/3-540-48234-2_11]

[113] Regan P, Hamilton S. NASA's mission reliable. Computer, 2004, 37(1): 59–68. [doi: 10.1109/MC.2004.1260727]

[114] Lamport L. The PlusCal algorithm language. In: Proc. of the 6th Int'l Colloquium on Theoretical Aspects of Computing. Malaysia: Springer, 2009. 36–60. [doi: 10.1007/978-3-642-03466-4_2]

[115] Desai A, Gupta V, Jackson E, Qadeer S, Rajamani S, Zufferey D. P: Safe asynchronous event-driven programming. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Seattle: ACM, 2013. 321–332. [doi: 10.1145/2491956.2462184]

[116] Shapiro M, Preguiça N, Baquero C, Zawirski M. Conflict-free replicated data types. In: Proc. of the 13th Int'l Symp. on Stabilization, Safety, and Security of Distributed Systems. Grenoble: Springer, 2011. 386–400. [doi: 10.1007/978-3-642-24550-3_29]

[117] Zave P. Using lightweight modeling to understand chord. ACM SIGCOMM Computer Communication Review, 2012, 42(2): 49–57. [doi: 10.1145/2185376.2185383]

[118] Ongaro D. Consensus: Bridging Theory and Practice. Stanford: Stanford University, 2014.

[119] Gu XS, Wei HF, Qiao L, Huang Y. Raft with out-of-order executions. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1748–1778 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6248.htm [doi: 10.13328/j.cnki.jos.006248]

[120] Yi XC, Wei HF, Huang Y, Qiao L, Lü J. TPaxos consensus protocol in PaxosStore: Derivation, specification, and refinement. Ruan Jian Xue Bao/Journal of Software, 2020, 31(8): 2336–2361 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5964.htm [doi: 10.13328/j.cnki.jos.005964]

[121] Vanlightly J. TLA+ specifications for Kafka. 2023. https://github.com/Vanlightly/kafka-tlaplus

[122] Howard H, Kuppe MA, Ashton E, Chamayou A, Crooks N. Smart casual verification of the confidential consortium framework. In: Proc. of the 22nd USENIX Symp. on Networked Systems Design and Implementation. Philadelphia: USENIX Association, 2025. 15.

[123] TLA+ specifications and trace validation for etcd. 2024.https://github.com/etcd-io/raft/tree/main/tla

[124] Ouyang LZ, Huang Y, Huang BY, Ma XX. Leveraging TLA$^+$ specifications to improve the reliability of the ZooKeeper coordination

service. In: Proc. of the 9th Int'l Symp. on Dependable Software Engineering. Theories, Tools, and Applications. Nanjing: Springer, 2023. 189–205. [doi: 10.1007/978-981-99-8664-4_11]

[125] Bornholt J, Joshi R, Astrauskas V, Cully B, Kragl B, Markle S, Sauri K, Schleit D, Slatton G, Tasiran S, van Geffen J, Warfield A. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In: Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles. ACM, 2021. 836–850. [doi: 10.1145/3477132.3483540]

[126] Schultz W. Fixing a MongoDB replication protocol bug with TLA+. 2019. https://youtu.be/x9zSynTfLDE

[127] Davis AJJ, Hirschhorn M, Schvimer J. Extreme modelling in practice. Proc. of the VLDB Endowment, 2020, 13(9): 1346–1358. [doi: 10.14778/3397230.3397233]

[128] Jackson D. Software Abstractions: Logic, Language, and Analysis. Cambridge: The MIT Press, 2012.

[129] The TLA+ proof manager. 2019. https://github.com/tlaplus/tlapm

[130] Newcombe C. Why Amazon chose TLA$^+$. In: Proc. of the 4th Int'l Conf. on Abstract State Machines. Toulouse: Springer, 2014. 25–39. [doi: 10.1007/978-3-662-43652-3_3]

[131] Hackett F, Hosseini S, Costa R, Do M, Beschastnikh I. Compiling distributed system models with PGo. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 159–175. [doi: 10.1145/3575693.3575695]

[132] Nadal J. Stateright/Stateright. 2018. https://github.com/stateright/stateright

[133] FizzBee. FizzBee—Design reliable, scalable distributed systems. 2024. https://fizzbee.io/

[134] Abadi M, Lamport L. The existence of refinement mappings. In: Proc. of the 3rd Annual Symp. on Logic in Computer Science. Edinburgh: IEEE, 1988. 165–175. [doi: 10.1109/LICS.1988.5115]

[135] Pressler R. Conjunction capers: A TLA+ truffle. 2020. https://conf.tlapl.us/2020/04-Ron_Pressler-TLA+_Truffle_Composition_Capers-Slides.pdf

[136] Howard Y, Gruner S, Gravell A, Ferreira C, Augusto JC. Model-based trace-checking. arXiv:1111.2825, 2011.

[137] Cirstea H, Kuppe MA, Loillier B, Merz S. Validating traces of distributed programs against TLA$^+$ specifications. In: Madeira A, Knapp A. Software Engineering and Formal Methods. Cham: Springer, 2025. 126–143. [doi: 10.1007/978-3-031-77382-2_8]

[138] Zhang YQ, Huang Y, Wei HF, Ma XX. Model-checking-driven explorative testing of CRDT designs and implementations. Journal of Software: Evolution and Process, 2024, 36(4): e2555. [doi: 10.1002/smr.2555]

[139] Rosa C, Merz S, Quinson M. A simple model of communication APIs—Application to dynamic partial-order reduction. In: Proc. of the 10th Int'l Workshop on Automated Verification of Critical Systems. Düsseldorf: AVOCS, 2010.

[140] TLA+ community modules. 2019. https://github.com/tlaplus/CommunityModules

[141] Mukherjee S, Deligiannis P, Biswas A, Lal A. Learning-based controlled concurrency testing. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 230. [doi: 10.1145/3428298]

[142] Cao JL, Lu YJ, Li MZN, Ma HY, Li HK, He MD, Wen C, Sun L, Zhang HY, Qin SC, Cheung SC, Tian C. From informal to formal—Incorporating and evaluating LLMs on natural language requirements to verifiable formal proofs. In: Proc. of the 63rd Annual Meeting of the Association for Computational Linguistics. Vienna: ACL, 2025. 26984–27003. [doi: 10.18653/v1/2025.acl-long.1310]

附中文参考文献

[13] 阿里云. 关于阿里云香港 Region 可用区 C 服务中断事件的说明. 2022. https://cn.aliyun.com/noticelist/articleid/1061819219.html

[14] 阿里云. [异常(已恢复)]阿里云云产品控制台服务异常. 2023. https://cn.aliyun.com/noticelist/articleid/1064981333.html

[28] 卜磊, 陈立前, 陈哲, 等. 形式化方法的研究进展与趋势. 见: 中国计算机学会. CCF 2017–2018 中国计算机科学技术发展报告. 北京: 机械工业出版社, 2018.

[30] 陈元亮, 马福辰, 周远航, 颜臻, 姜宇, 孙家广. 分布式系统动态测试技术研究综述. 软件学报, 2025, 36(7): 2964–3002. http://www.jos.org.cn/1000-9825/7334.htm [doi: 10.13328/j.cnki.jos.007334]

[31] 葛宁, 贺俞凯, 翟树茂, 李晓洲, 张莉. 共识协议的形式化验证研究现状与展望. 软件学报, 2023, 34(11): 4989–5007. http://www.jos.org.cn/1000-9825/6684.htm [doi: 10.13328/j.cnki.jos.006684]

[32] 侯刚, 周宽久, 勇嘉伟, 任龙涛, 王小龙. 模型检测中状态爆炸问题研究综述. 计算机科学, 2013, 40(6A): 77–86, 111. [doi: 10.3969/j.issn.1002-137X.2013.z1.018]

[33] 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, 2024, 35(9): 4069–4099. http://www.jos.org.cn/1000-9825/7138.htm [doi: 10.13328/j.cnki.jos.007138]

[38] 吕建, 马晓星, 陶先平, 曹春, 黄宇, 余萍. 面向网构软件的环境驱动模型与支撑技术研究. 中国科学: 技术科学, 2008, 38(6):

864–900. [doi: 10.3321/j.issn:1006-9275.2008.06.005]
[39] 吕建, 马晓星, 陶先平, 黄宇, 余萍, 许畅. 面向网构软件的环境显式化技术. 中国科学: 信息科学, 2013, 43(1): 1–23. [doi: 10.1360/112012-527]
[119] 谷晓松, 魏恒峰, 乔磊, 黄宇. 支持乱序执行的 Raft 协议. 软件学报, 2021, 32(6): 1748–1778. http://www.jos.org.cn/1000-9825/6248.htm [doi: 10.13328/j.cnki.jos.006248]
[120] 易星辰, 魏恒峰, 黄宇, 乔磊, 吕建. PaxosStore 中共识协议 TPaxos 的推导、规约与精化. 软件学报, 2020, 31(8): 2336–2361. http://www.jos.org.cn/1000-9825/5964.htm [doi: 10.13328/j.cnki.jos.005964]

作者简介

唐瑞泽, 博士, 主要研究领域为分布式系统, 形式化方法.
黄宇, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为分布式系统, 系统软件, 软件形式化方法.
欧阳凌志, 博士, 主要研究领域为分布式共识系统, 形式化方法.
程潜, 博士生, CCF 学生会员, 主要研究领域为分布式系统, 大语言模型, 形式化方法.
张宇奇, 博士, 主要研究领域为分布式算法, 分布式系统, 形式化方法.
马晓星, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件自适应技术, 智能软件质量保障技术.