

# GC-MCR:有向图约束指导的并发缺陷检测方法<sup>\*</sup>



李硕川<sup>1</sup>, 王赞<sup>1</sup>, 马明旭<sup>1</sup>, 陈翔<sup>2</sup>, 赵英全<sup>1</sup>, 王海弛<sup>1</sup>, 王昊宇<sup>1</sup>

<sup>1</sup>(天津大学 智能与计算学部,天津 300350)

<sup>2</sup>(南通大学 信息科学技术学院,江苏 南通 226019)

通讯作者: 王赞, E-mail: wangzan@tju.edu.cn

**摘要:** 约束求解应用到程序分析的多个领域,在并发程序分析方面也得到了深入的应用.并发程序随着多核处理器的快速发展而得到广泛使用,然而并发缺陷对并发程序的安全性和可靠性造成了严重的影响,因此针对并发缺陷的检测尤为重要.并发程序线程运行的不确定性导致的线程交织爆炸问题给并发缺陷的检测带来了一定挑战.已有并发缺陷检测算法通过约减无效线程交织,以降低在并发程序状态空间内的探索开销.比如最大因果模型算法把并发程序状态空间的探索问题转换成约束求解问题,然而其在约束构建过程中会产生大量冗余和冲突的约束,大幅度增加了约束求解的时间以及约束求解器的调用次数,并降低了并发程序状态空间的探索效率.针对上述问题,论文提出了一种有向图约束指导的并发缺陷检测方法 GC-MCR(Directed Graph Constraint-guided Maximal Causality Reduction).该方法旨在通过使用有向图对约束进行过滤和约减,从而提高约束求解速度,并进一步提高并发程序状态空间的探索效率.实验结果表明,GC-MCR方法构建的有向图可以有效优化约束的表达式,从而提高约束求解器的求解速度并减少求解器的调用次数.与现有的J-MCR方法相比,GC-MCR的并发程序缺陷检测效率可以取得显著提升且不会降低并发缺陷的检测能力,在现有研究方法广泛使用的38组并发测试程序上的测试时间可以平均减少34.01%.

**关键词:** 并发程序;最大因果约减;约束求解;有向图;冲突约束过滤

**中图法分类号:** TP311

中文引用格式: 李硕川,王赞,马明旭,陈翔,赵英全,王海弛,王昊宇.GC-MCR:有向图约束指导的并发缺陷检测方法.软件学报.  
<http://www.jos.org.cn/1000-9825/6865.htm>

英文引用格式: Li SC, Wang Z, Ma MX, Chen X, Zhao YQ, Wang HC, Wang HY. GC-MCR: Directed-graph constraint guided concurrent bug detection method.Ruan Jian Xue Bao/Journal of Software (in Chinese).<http://www.jos.org.cn/1000-9825/6865.htm>

## GC-MCR: Directed-graph constraint guided concurrent bug detection method

LI Shuo-Chuan<sup>1</sup>, WANG Zan<sup>1</sup>, MA Ming-Xu<sup>1</sup>, CHEN Xiang<sup>2</sup>, ZHAO Ying-Quan<sup>1</sup>, WANG Hai-Chi<sup>1</sup>,  
WANG Hao-Yu<sup>1</sup>

<sup>1</sup>(College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

<sup>2</sup>(School of Computer Science and Technology, Nantong University, Nantong 226019, China)

**Abstract:** Constraint solving has been applied to many domains of program analysis, and deeply applied in concurrent program analysis. Concurrent programs are specific domain software that has been widely used with the rapid development of multi-core processors. However, concurrent defects threaten the security and robustness of concurrent programs, thus it is of great importance to test concurrent programs. Due to the non-deterministic thread scheduling, one of the key challenges for concurrent program testing is how to reduce the thread interleaving space with exponential growth. The state-of-the-art approaches (i.e., J-MCR) tackle the challenge through constraint solving. However, we found that there exist conflicts and redundancies inside constraints (i.e., the conflict of constraint clauses makes constraints unsatisfiable), solving those unsatisfiable constraints results in low efficiency. Thus, we propose a directed graph constraint-guided method called GC-MCR (Directed Graph

\* 基金项目: 国家自然科学基金(61872263); 天津市智能制造专项资金项目(20201180)

收稿时间: 2022-09-04; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

Constraint-guided Maximal Causality Reduction). By removing conflictive constraints and simplify redundant constraints, we reduce the times of constraint solving thereby further improving the efficiency. Comparing with state-of-the-art approach J-MCR, GC-MCR reduces the times of constraint solving by 19.36% on average and reduces the testing time on average by 34.01% on 38 concurrent programs.

**Key words:** concurrent program; maximum causality reduction; constraint solving; directed graph; conflict constraint filtering

目前,约束求解作为符号执行的重要组成部分,已在软件验证、程序分析等领域得到了广泛应用,其在并发程序分析领域的贡献尤为突出.并发程序以计算速度快且资源利用率高为特点,在当今多核架构广泛应用的硬并发时代,其应用越发普遍.从微观角度看,并发程序的线程调度(即某一时刻每个线程是否运行)是由 CPU 决定的,而不是由用户决定的<sup>[1]</sup>.因此,在程序默认配置下,线程的执行顺序具有一定的不确定性.这种不确定性导致开发者难以预料程序的状态,使得程序内部易出现并发缺陷,可能会造成严重事故.例如 2012 年,Facebook 上市当天,纳斯达克的交易程序出现并发缺陷(即由于大量的撤单导致程序一直重新运行并进入死循环,出现了数据竞争问题),导致 20 分钟无法开盘,给 Facebook 造成了巨大损失.故而在实际项目开发中,需要对并发程序进行充分的测试,以避免并发缺陷可能造成的不良影响.因此,针对并发软件的自动缺陷检测技术研究具有重大意义,其研究成果有助于快速有效的检测出各类并发缺陷,以保证并发软件的质量和可靠性.

近些年来,针对并发程序的测试在软件工程领域得到了广泛的关注,并发缺陷检测的相关技术逐渐成为研究热点.目前国内外已经发表多篇与并发缺陷检测相关的综述论文<sup>[1,2,3]</sup>,现有研究从不同角度对并发缺陷检测进行了深入探索并取得丰富的研究成果,在此基础上也开发了一些成熟的并发检测工具.目前常见的并发缺陷可以分为四种:死锁<sup>[4]</sup>、数据竞争<sup>[5]</sup>、原子性违背<sup>[6]</sup>和顺序违背<sup>[7]</sup>.为检测这四种并发缺陷,研究人员提出了动态分析<sup>[8]</sup>、静态分析<sup>[9]</sup>以及动静结合分析<sup>[10]</sup>等三类方法.其中,动态分析通过动态执行被测程序,旨在触发程序异常,且触发的异常问题都是真实存在的.静态分析基于缺陷模式的规则,提取和对比分析代码中的模式以检测缺陷<sup>[11]</sup>.符号执行作为静态分析的主要方式,通过对符号化的程序路径约束进行约束求解,探索程序状态空间.动静结合分析则充分利用动态分析和静态分析的优点,即动态分析可以提升静态分析可靠性,静态分析可以降低动态分析的验证负载<sup>[2,3]</sup>.

并发程序的测试和验证面临的主要挑战是线程交织(Thread Interleaving)空间爆炸问题,即可能的线程交织数量随着线程数量和程序执行时间的增加而呈指数级增长,使并发缺陷难以被检测.解决线程交织爆炸的关键是识别冗余的线程交织<sup>[12]</sup>.为解决这一问题,研究人员提出了偏序约减方法(Partial Order Reduction,简称 POR)<sup>[13,14,15]</sup>,即通过检查程序各个状态和行为间的独立性以减小整体状态空间规模.首先,POR 会将所有线程交织划分为不同的 Mazurkiewicz 轨迹<sup>[16]</sup>,即所有线程交织被分成的不同等价类.然后,POR 从每个 Mazurkiewicz 轨迹中选取一个线程交织进行探索.由于 Mazurkiewicz 轨迹基于 happens-before 关系<sup>[17]</sup>,且 happens-before 关系依赖于所有锁的释放和获取以及冲突的写事件和读事件,POR 在识别冗余线程的交织方面存在局限性,即仅符合 happens-before 关系的线程交织中大部分线程交织存在冗余<sup>[18]</sup>.针对这一问题,Huang 等人提出了最大因果约减算法(Maximal Causality Reduction,简称 MCR)<sup>[18]</sup>,MCR 基于 Mazurkiewicz 轨迹制定了新标准:最大因果关系(Maximal Causality Relation),即根据读和写事件的值划分 Mazurkiewicz 轨迹中的最大可能等效线程交织集合.MCR 将探索并发程序状态空间的问题转换成约束求解问题,稳定高效地检测并发缺陷.然而异常庞大的并发程序状态空间会导致 MCR 为事件添加的约束极其严格且复杂,复杂的约束构建易产生大量冗余甚至存在冲突的约束,这些约束会导致约束求解耗时较长,据实验统计 MCR 在约束求解上的耗时约占 MCR 运行总时间的 89.21%,而冲突的约束在求解过程中甚至存在不可解的问题并消耗了大量时间,大幅度降低并发缺陷的检测效率.

针对上述问题,本文以最大因果约减算法中的约束为研究对象,通过过滤冲突约束并约减冗余约束,以提高并发缺陷检测的效率,最终设计了一种有向图约束指导的并发缺陷检测方法 GC-MCR(Directed Graph Constraint-guided Maximal Causality Reduction).GC-MCR 首先根据程序基本约束关系(即 happens-before 约束、锁互斥约束等)来构建有向图;然后对基于最大因果约减算法<sup>[18]</sup>生成的读写约束进行解析与拆分,即将读写约束构建成读写约束树,根据程序基本约束关系的有向图,获得读写约束树中每个节点的值,该值表示以当前节点

为根节点的子树是否与程序基本约束冲突.最后根据读写约束树中每个节点的值,判断是否过滤或约减约束.GC-MCR 通过对冲突约束进行过滤,并约减冗余约束的表达式,有效降低单次约束的求解时间甚至减少约束求解器的调用次数,因此能够保证并发缺陷的检测能力,并提高并发缺陷效率.

本文选择在已有研究工作中广泛使用的 38 组并发程序作为实验对象.这 38 组并发程序包含的缺陷类型覆盖了死锁、数据竞争、原子性违背、顺序违背等.实验结果表明,GC-MCR 方法无论是在约束求解器的调用次数以及单次调用时间方面还是从探索并发程序状态空间的效率方面,都显著优于现有方法,与 J-MCR 相比在总的调度数量没有损失的情况下,即并发缺陷的检测能力没有任何损失,程序的约束求解次数和总调用时间可平均减少 34.01%.本文的主要贡献总结如下:

- 基于有向图设计并发约束的表示方法,从有向图结构上识别出潜在的不可解并发约束,优化冗余并发约束,提高并发约束表示准确性.
- 提出一种新的有向图约束指导的并发缺陷检测方法 GC-MCR.GC-MCR 基于本文提出的有向图表示方法构建完整的并发程序约束,并对其进行过滤和约减,可以有效减少约束求解成本,从而提高探索并发程序状态空间的效率.
- 实现并开源了 GC-MCR<sup>1</sup>,并在 38 个并发测试程序上进行了验证,实验结果表明,相比于现有研究方法,GC-MCR 可以更快的找出并发程序中的缺陷,其平均性能可提升 34.01%.

## 1 背景知识

本文主要目标是通过有向图表示的方法,构建完整的并发程序约束并对其过滤和约减,减少约束求解成本,提高探索并发程序状态空间的效率.与本文方法相关的理论和方法包括作为 GC-MCR 的基础模型—最大因果模型(Maximal Causal Model,简称 MCM)<sup>[5,19]</sup>、在最大因果模型的基础上提出的最大因果约减算法<sup>[18]</sup>以及约束的构建与求解的工具 Z3<sup>[20]</sup>.

### 1.1 最大因果模型

最大因果模型<sup>[5,19]</sup>根据给定的程序执行轨迹构建出最大且合理的因果模型,其包含所有能生成原始轨迹的并发程序所生成的所有轨迹,即假如并发程序  $P$  能够生成原始轨迹,则  $P$  生成的所有轨迹都包含在最大因果模型中.因此最大因果模型可以发现在庞大的状态空间中尚未被执行到的线程交织.在 MCM 中,事件(Event)是线程在并发对象上执行的原子操作,其中事件的属性包括:访问一个并发对象的线程 Thread、对该并发对象进行的操作类型 OP、该事件访问的并发对象 Target、对并发对象执行的操作值 Data.对于两个事件  $s_1$  和  $s_2$ ,当且仅当  $s_1$  和  $s_2$  所对应的全部属性都相等,称事件  $s_1$  和  $s_2$  相等.MCM 事件的类型和对应描述如下所示.

- begin(t):线程 t 中的第一个事件.
- end(t):线程 t 中最后一个事件.
- read(t,x,v):线程 t 读到 x 的值为 v.
- write(t,x,v):线程 t 写入变量 x 的值为 v.
- lock(t,l):线程 t 获取锁 l.
- unlock(t,l):线程 t 释放锁 l.
- fork(t,t'):线程 t 创建新线程 t'.
- join(t,t'):阻塞线程 t 直到 t'运行结束.

在 MCM 中,程序的一条执行轨迹(Execution trace)可以被抽象为一组事件的序列.并且规定执行轨迹需要满足顺序一致性,即读写一致性、锁互斥和先发生于原则(happens-before).其中,读写一致性需要保证读(Read)的值是执行轨迹在该读事件之前,且距离其最近,且与其访问同一个共享内存地址的写(Write)的值.锁互斥需要

<sup>1</sup> <https://github.com/WingedVampires/GC-MCR.git>

保证每个释放(Release)锁事件之前都有同一个线程且同一个锁变量的获得(Acquire)锁事件与之匹配,且对每一个 Acquire-Release 锁对都不可与其他访问同一锁变量的锁对产生交织.happens-before 原则需要保证程序按照代码的顺序执行;对 volatile 修饰的变量进行的写操作要先发生于对其进行的读操作;对同一个锁的解锁操作要先发生于对其的加锁操作;如果操作 A 先发生于操作 B,操作 B 先发生于操作 C,那么 A 应该先发生于操作 C;在线程 A 中调用 B.start()之前的所有操作先发生于线程 B 的所有操作;线程 B.join()会先执行线程 B,即线程 B 中的所有操作先发生于在线程 A 中的 join 后面的所有操作.

在 MCM 中,feasible( $\tau$ )是由能够产生执行轨迹  $\tau$  的所有由程序的执行产生的顺序一致性轨迹构成最小的集合,feasible( $\tau$ )也是最大因果模型的最终结果.feasible( $\tau$ )需要满足前缀闭合性(Prefix Closedness)和局部确定性(Local Determinism).前缀闭合性假设事件是独立的、不可分割的,且是根据程序的执行按顺序生成的,因此 feasible( $\tau$ )必须是前缀闭合的.局部确定性假设并发操作的执行是由同一线程中的之前发生的事件确定的,并且可以在它们之后的任一时刻发生.

## 1.2 最大因果约减算法

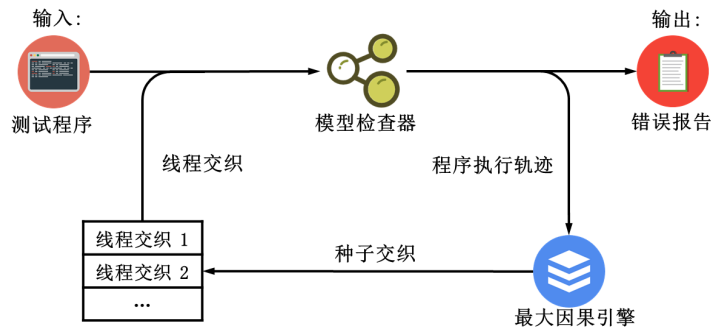


图 1 最大因果约减算法执行流程

最大因果约减算法通过构建并发程序的最大因果模型,将探索程序状态空间问题转换为约束求解问题.最大因果约减算法如图 1 所示,先输入一个被测程序,通过模型检查器(Model Checker)按照指定的线程交织得到程序的执行轨迹(Trace).在模型检查器中的调度器通过控制线程的调度探索指定线程交织.然后通过最大因果引擎(Maximal Causality Engine)生成新的种子线程交织,并将其添加到被测程序的状态空间中,使得该交织之后可以被模型检查器再次使用,并通过运行获取新的种子交织.对于一个线程交织  $s$ ,通过前文介绍的最大因果模型,可以获得一组唯一且规模最大的可行性交织集合,记作  $\text{MaxCausal}(s)$ .且每个 MCR 程序运行产生的新的种子交织都是  $\text{MaxCausal}(s)$  中的可行交织.

在并发程序的运行过程中,不同线程的交错执行会产生一系列线程交织.而每个被触发的线程交织都会使程序到达某一种状态(State).这里规定当程序执行之前产生新交织时,如果从同一个变量上读到和之前不同的值从而使得程序进入了一个新的状态,那么算法需要将这一状态加入到被测程序的状态空间中.当被测程序的状态空间已满,即所有的种子交织都已经被探索过,且没有新的种子交织生成时,即可以认为被测程序的状态空间已经被完全覆盖了.这里 MCR 提出了一个新的约束 Race 约束,即假设  $\text{COP}(x,y)$  表示一个全局变量在不同线程中的读写对,那么对于程序中每一个存在的 COP 关系的全局变量,需要对这些变量的每一个读事件 R 进行约束,即所有 happens-before 于 R 的读事件读到与之前一样的值,且 R 应当读到一个与之前完全不一样的值.由于每次程序读到新值都可能导致程序到达一个与之前不同的状态,因此通过在程序的约束求解的过程中不断地读到新值,可以获得一棵程序的状态树,而这棵树就可以覆盖程序中所有可能的程序状态.

导致并发程序的状态空间庞大的原因分为两方面.一方面由于线程的组合可以是多种多样的,且其中大部分都是可行的,以及线程的调度是不确定的.另一方面并发程序的状态空间存在大量等价的线程交织,除线程的调度顺序不同,读取到的数据是一致的,即这些等价的线程是不能够使程序到达一个新的不同的状态.大量等价

线程交织使其他算法执行时间相对较长,而最大因果约减算法虽然也会重复的执行被测程序,通过干预线程调度,执行所有调度并检查当前调度的可行性,产生新的种子交织,然而该算法仅当读到新值时才会识别程序到达了一个新的状态,并将当前线程调度加入到当前程序的状态空间中,从而能够最大化约减状态空间。

### 1.3 约束的构建与求解

本文将最大因果约减算法中的约束分为程序基本约束和读写约束。程序基本约束是程序中对 happens-before 原则的约束体现,其中包括程序顺序约束、线程约束、锁约束等。读写约束则是约束不同线程对同一共享变量按照指定读写要求进行读写,其中读写要求是程序中所有线程的读事件中有且只有一个读事件 R 读到新的值,而发生在 R 前的读事件需要读到旧的值。如图 2 程序所示, T1 和 T2 是两个线程,则图中程序基本约束是  $x1 < x2$ 、 $x3 < x4$ 、 $x4 < x5$ 、 $x5 < x6$ ,即同一线程中程序按照代码的顺序执行;假如第 4 行 x 应读到值 1,则读写约束为  $x3 < x1$ 、 $x1 < x4$ ,即对 x 的赋值为 1 的操作需要在第四行前执行,且在第三行对 x 赋值为 0 的操作后执行。程序基本约束和读写约束是 and 关系,即并列关系。只有两种约束共同求解出的调度序列才是一个可执行的调度序列。

T1	T2
1: x = 1	3: x = 0
2: y = 2	4: if (x > 0) {
	5: y—
	6: x = 7
	}

图 2 约束构建示例程序

本文使用 Z3<sup>[20]</sup>对构建的约束文件进行求解,Z3 是由微软开发的开源约束求解器,是用来解决一阶逻辑组合的判定性问题的一种通用求解器。在给定部分约束条件的情况下,Z3 能够找到一组满足这些条件的解。使用 Z3 进行求解需要编写 Z3 脚本,Z3 脚本包含一段特定的命令序列,这一段命令序列将会被求解器使用特定算法解析通过,最终求解器会返回公式命题的求解结果以及来自 Z3 的理论求解器的赋值。通过使用 Z3 对 GC-MCR 方法得到的新约束进行求解,然后根据求解结果获得新的调度序列,探索并发程序状态空间。

## 2 GC-MCR 方法

### 2.1 研究动机

为了充分探索并发程序的状态空间,MCR<sup>[18]</sup>对已有的执行轨迹进行分析,根据不同事件间的约束关系生成满足 MCM 和 Race 的约束,并利用约束求解器进行求解。最终根据求解结果对并发程序进行进一步的探索,即若约束可解,则按照求解得到的线程交织控制程序调度;反之,则表示该线程交织无法发生,故跳过当前约束并求解其他约束。尽管 MCR 可以通过较少的调度次数找到并发程序中潜在的并发缺陷。然而,已有的研究工作<sup>[21]</sup>以及本工作对 MCR 方法的重现与深入分析表明, MCR 在构建程序约束时粒度较粗,生成大量的冗余或不可解约束,造成约束求解次数过多,额外消耗大量求解计算资源。假如能够在求解前过滤不可解约束或约减冗余约束,即可节省大量计算资源。因此通过在求解前过滤不可解约束或约减冗余约束,可节省大量求解成本,从而提升并发缺陷的检测效率。为此,本文分别在 hashcode、numberaxis 和 dekker 三个程序状态空间大小不同的测试程序上进行实验验证。实验结果显示,MCR 在上述测试程序中分别进行约束求解 32、242 和 492 次。然而对这些约束进一步的分析发现分别有 21、67 和 150 个约束可通过约束分析进行过滤而无需使用约束求解器进行约束求解,其余约束也能够进行不同程度的约减。基于上述发现,本工作对上述约束进行过滤和约减,最终发现程序执行时间分别减少 51.78%、28.29%和 58.35%,并发程序状态空间的探索效率显著提高。本文后续在其它测试程序

进行实验,最终得到相同结论.为阐述本文所提方法的基本原理,后续方法介绍将以 dekker 程序为例.

实验发现 MCR 生成的约束中,部分约束存在明显冲突,且能够通过简单有效的方式处理这些冲突.例如,图 3 和图 4 分别为 dekker 程序在一次约束求解中出现的一个程序基本约束和两个读写约束.其中每个 assert 语句由 assert 关键词和一个子约束构成,每个子约束使用前缀表达式表示,其中最内层的一组括号中的内容是一个约束单元,每个约束单元可使用 and/or 进行组合并构成一个子约束.例如图 4(b)中( $\text{or} (> x1 x9) (< x3 x6)$ )是一个子约束,其中( $> x1 x9$ )是一个约束单元,实际意义为事件  $x1$  在事件  $x9$  之后发生;图 3 中( $> x1 x2$ )既是一个约束单元,又是一个子约束.每个 assert 语句之间是 and 关系,即当且仅当每个约束都为真,才能表示整体约束是满足的.

```
(assert (> x1 x2)) (assert (< x5 x6)) (assert (< x7 x8))
(assert (> x3 x5)) (assert (> x6 x8)) (assert (< x4 x6))
(assert (> x3 x7)) (assert (< x6 x2)) (assert (> x9 x4))
(assert (< x4 x2)) (assert (> x3 x6))
```

图 3 程序基本约束示例

```
(assert (and (or (or (< x8 x7) (< x9 x4)) (and (> x1 x8)
(> x3 x5)))) (and (or (> x1 x4) (> x1 x6)) (< x3 x6))))
```

(a) 冲突约束

```
(assert (or (> x1 x9) (< x3 x6)))
```

(b) 冗余约束

图 4 读写约束示例

**冲突约束.** 图 4 中读写约束 a 中的  $x8 < x7$ 、 $x4 > x9$ 、 $x3 < x6$  分别与图 3 中程序基本约束中的  $x8 > x7$ 、 $x9 > x4$ 、 $x3 > x6$  冲突,而其他约束与程序基本约束一致,通过逻辑运算判断其最终结果为 false,与程序基本约束冲突,即满足程序基本约束的条件下读写约束不成立.故定义这类与程序基本约束冲突的读写约束为冲突约束.冲突约束即不可解约束.并发测试程序 dekker 中存在 150 个冲突,在使用约束求解器求解前识别并过滤这类冲突约束,则能够节省大量计算资源.

**冗余约束.** 进一步分析发现,读写约束 b 中的  $x3 < x6$  与程序基本约束中的  $x3 > x6$  冲突,而另一部分  $x1 > x9$  与程序基本约束不冲突,且在满足程序基本约束的条件下读写约束 b 成立.故定义这类部分子约束与程序基本约束冲突,且约束整体与基本约束不冲突的读写约束为冗余约束.冗余约束能够根据程序基本约束进行约减,图 4 中的冗余约束就可以被约减为( $> x1 x9$ ).而对于约减后的约束,约束求解器可以更快的进行求解,节省程序在约束求解步骤中的求解计算资源.而且对约束进行过滤与约减产生的时间开销远小于求解器求解原约束的时间开销.

基于上述分析,本文提出了一种有向图约束指导的并发缺陷检测方法 GC-MCR,首先把约束分为读写约束和程序基本约束,然后先将程序基本约束构建成为有向图,再将读写约束构建成为前缀树,最后根据有向图判断读写约束中是否存在冲突约束和冗余约束.若存在冲突约束,则过滤掉当前约束;若存在冗余约束,则将其约减后,再次调用约束求解器进行求解,从而提高并发缺陷检测效率.

## 2.2 方法流程

### 1. 整体框架:输入输出及整体流程简介.

GC-MCR 方法的输入是并发测试程序,输出是程序的并发缺陷错误报告.首先,模型检查器执行并发测试程序,获取程序的初始执行轨迹.最大因果引擎分析处理执行轨迹,生成满足 MCM 和 Race 约束的约束.然后,GC-MCR 方法将约束分为读写约束和程序基本约束,程序基本约束被分析并重构成为有向图,读写约束被分析、拆分并重组成为读写约束树.随后,约束过滤器根据有向图计算读写约束树中每个节点的值,并根据读写约束树根节点的值判断:是否跳过当前求解或约减当前约束.假如进行约束求解,求解得到的线程交织会存储到线程

交织集合中.最后,模型检查器通过依次执行线程交织集合中的线程交织,获取程序执行轨迹并检测程序是否存在并发缺陷.假如检测到并发缺陷则输出错误报告,反之则重复上述操作.GC-MCR 方法的具体工作流程如图 5 所示.

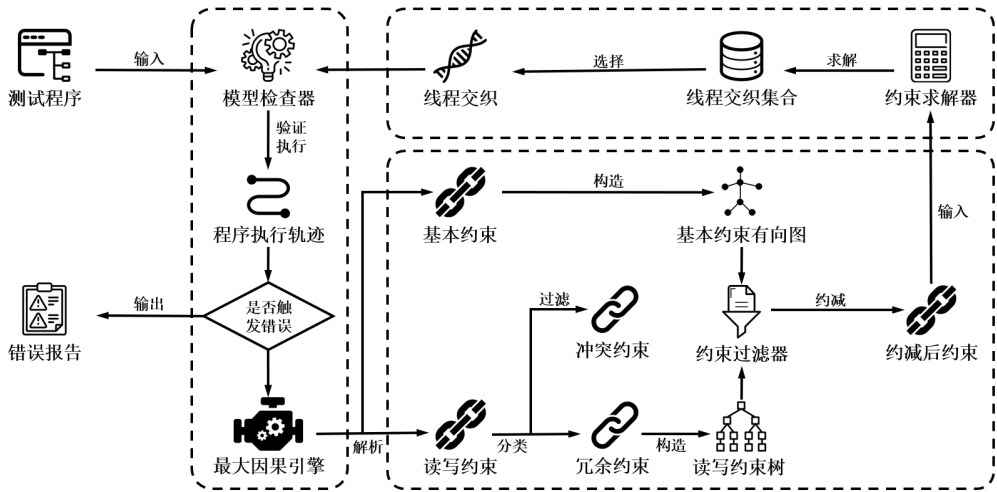


图 5 基于有向图构建约束的并发缺陷检测方法流程图

## 2. 基于有向图的程序基本约束重构.

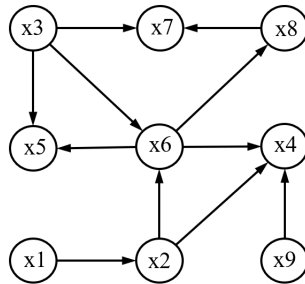


图 6 基本约束的有向图

图 3 中约束为 GC-MCR 生成的程序基本约束,程序基本约束都满足 happens-before 原则.根据顺序一致性模型的定义,程序中的约束必须遵循 happens-before 原则,因此不应该存在违背程序基本约束的约束.图 6 是本方法生成的有向图,表示程序中各个调度点在满足 happens-before 原则下的顺序关系.

由于程序基本约束中的顺序不等式存在传递性关系,且不是所有调度点都存在这些顺序关系.因此通过数轴上的大小表示难于归纳出所有调度点的顺序关系,而为了清晰表明调度点的顺序关系,本方法将这些顺序关系转化为有向图来表示.具体实现时使用邻接表构建有向图,并定义新的节点类,其中包含当前节点对应的调度点和在有向图中直接指向该调度点的节点的集合.在图 3 中,若存在约束  $x_1 > x_2$ ,则在有向图中构建  $x_1$  指向  $x_2$  的边,不断重复该过程,直至最终构建出该有向图.如算法 1 所示,首先分析程序基本约束中每对调度点间的关系,然后在图中找到对应节点,最后将节点间指向关系存储在邻接表中.通过上述步骤,使用有向图表示程序基本约束关系,仅通过遍历该有向图,即可获得程序中线程调度点间的关系,用于后续处理读写约束.

### 算法 1 基于有向图的程序基本约束重构

输入:  $\Phi_{base}$ :程序基本约束;

输出:  $G$ :由邻接表构成的有向图

---

```

1   $G \leftarrow \emptyset$ ;
2  for  $\varphi_i$  in  $\Phi_{base}$  do
3     $option, target, next \leftarrow analysisConstraint(\varphi_i)$ ;
4     $targetNode, nextNode \leftarrow findNode(target, next, G)$ ;
5     $targetNode.largeNodes.add(nextNode)$ ;
6  end for
7  return  $G$ 

```

---

### 3. 读写约束的解析与拆分

读写约束的生成是 GC-MCR 的核心步骤,该步骤确保所有共享变量的每一个读事件 R,需要满足两个条件:读事件 R 要读到一个之前没有读到过的值;所有发生在读事件 R 之前的读事件要读到和原来一样的值.这样保证只有一个读事件读到新值,其余读事件仍读到原值.例如,对于变量 x,若需要读取新值,则让其中一个具有新的写值的写 x 事件排在该读 x 事件之前,若需要保持读取到原值,则应当保证新的 x 写事件在该读事件之后.因此,GC-MCR 能够通过控制读写事件,改变调度点的顺序排布.

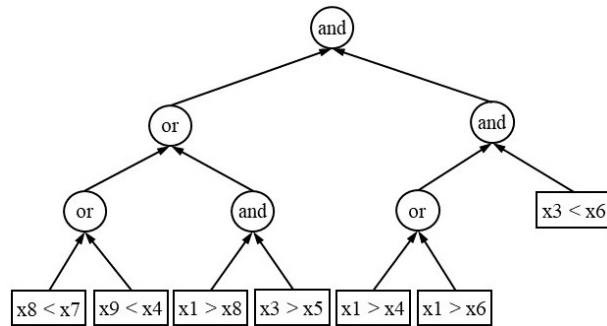


图 7 读写约束树

然而令一个读事件读到新值有多种限制和可能情况,其中既包含保证新的 x 的写入值的约束,又包含保证其他读事件的值保持不变的约束,因此读写约束颇为复杂.如图 4 所示,读写约束 a 是由多个 or 和 and 组合表示的约束.现有方法将这类读写约束和先前生成的程序基本约束传入约束求解器中进行求解,获得调度点的顺序关系后再解析成新的线程调度序列,进行线程定向调度,最终探索程序整个状态空间.然而,现有方法忽视了读写约束和程序基本约束之间的关系,其中包含明显冲突的约束和冗余的约束,冲突约束不需要调用约束求解器进行求解,冗余约束可以约减后进行求解.通过过滤冲突约束和约减冗余约束,能够有效减少单次求解时间甚至减少约束求解器的调用次数,显著提高并发缺陷检测效率.

---

#### 算法 2 读写约束的解析与拆分

---

输入:  $\varphi_{rw}$ : 程序的一条读写约束,  $G$ : 程序基本约束的有向图;

输出:  $T$ : 以前缀表达式表示的读写约束树, 树中的每个节点包含当前节点的布尔值以及从叶子节点到当前节点的约;

```

1   $T \leftarrow \emptyset$ ;
2   $list \leftarrow split(\varphi_{rw})$ ;
3  for  $s$  in  $list$  do
4    if  $s \in \{<, >\}$  then
5       $target, next \leftarrow analysisConstraint(s)$ ;
6       $isConflict \leftarrow isConflictWithOrderMap(target, next, G)$ ;
7       $expression \leftarrow getConstraintExpression(s, target, next, isConflict)$ ;
8       $T.add(Pair(isConflict, expression))$ ;
9    else
10      $T.add(Pair(s, s))$ ;
11   end if
12 end for

```

---



---

 13 `return T;`


---

因此 GC-MCR 对读写约束进行解析,图 7 是根据图 4 中的读写约束 a 构造的前缀树,其中非叶子节点皆是 and/or 的逻辑运算符,叶子节点则是调度点的顺序关系.后续用于遍历程序基本约束有向图进行冲突分析.具体算法见算法 2.例如,其中一个叶子节点中  $x_3 > x_9$ ,查看在图 6 中是否存在  $x_9 > x_3$ ,即是否存在一条由  $x_9$  到  $x_3$  的路径(行 6),若存在,则设该叶子节点为 false 表示该子表达式中存在冲突,反之该叶子节点为 true 表示该子表达式中不存在冲突.然后对每个叶子节点执行该操作,将全部叶子节点的结果替换为 true/false,同时每个叶子节点还应存储当前节点的约束表达式(行 7-行 8),后续用于约束表达式约减.最终经过冲突分析后,叶子节点应全部变为 true/false,此时,整棵树为一个由 and/or 和 true/false 组成的运算树.最终只需将该运算树进行运算即可.若结果为 false,则表示该读写约束与程序基本约束冲突,直接跳过此次对约束求解器的调用即可.反之,则约减当前约束.被过滤约束没有调用约束求解器,这部分时间被过滤算法优化,从而提高并发缺陷检测效率.

#### 4. 约束表达式约减.

对冗余约束的约减,GC-MCR 将约减其中冲突的子表达式,最后对约减后的约束表达式及程序基本约束的表达式调用约束求解器进行求解.

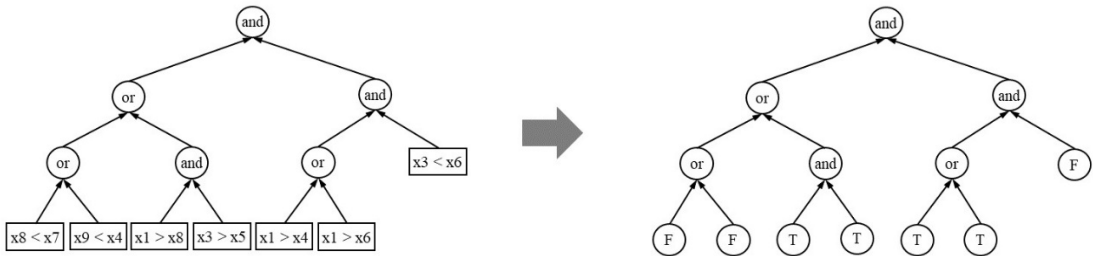
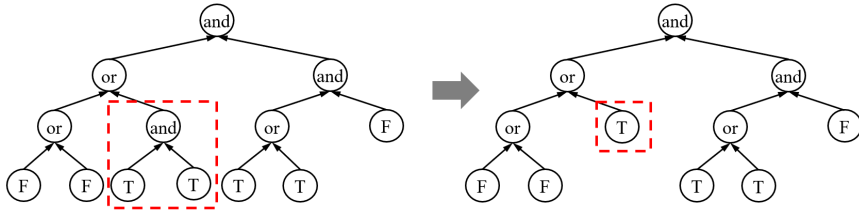


图 8 前缀树替换的示例

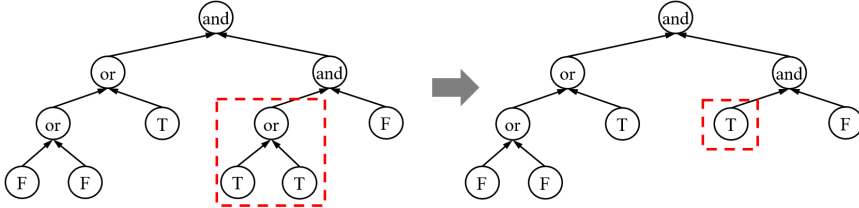
本文中,读写约束树的每个节点既包含以当前节点为根节点的子树是否满足程序基本约束的值,又包含子树的约束表达式,称约束表达式前缀树中的叶子节点中的约束表达式为子表达式.首先将子表达式与程序基本约束冲突的叶子节点的值置为 false,其余叶子节点的值置为 true.如图 8 所示,赋值后的前缀树的叶子节点为 true/false,在图中使用 T 表示 true,F 表示 false.然后根据父节点的操作符及其两个子节点的值,考虑以下五种情况:

- 父节点为 and,两个子节点皆为 true,如图 9a 所示,将父节点的值置为 true,并将父节点的约束表达式置为 and 关系的两个子节点的约束表达式.
- 父节点为 or,两个子节点皆为 true,如图 9b 所示,将父节点的值置为 true,并将父节点的约束表达式置为 or 关系的两个子节点的约束表达式.
- 父节点为 or,两个子节点皆为 false,如图 9c 所示,将父节点的值置为 false,并将父节点的约束表达式置为空.
- 父节点为 and,两个子节点中至少一个为 false,如图 9d 所示,将父节点的值置为 false,并将父节点的约束表达式置为空.
- 父节点为 or,两个子节点中仅一个为 true,如图 9e 所示,将父节点的值置为 true,并将父节点的约束表达式置为值为 true 子节点的约束表达式.

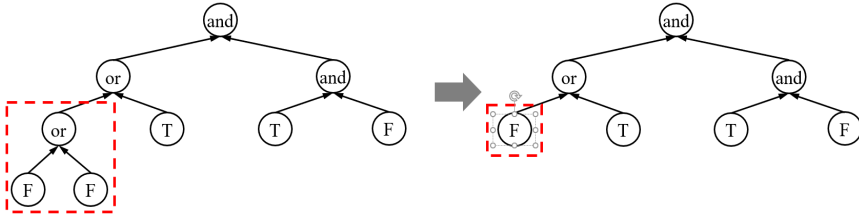
重复上述操作直至读写约束树中仅存在根节点.此时,根据根节点的值判断是否使用约束求解器进行约束求解:若值为 false 则不需要,反之使用根节点的约束表达式,即最简的读写约束表达式,结合程序基本约束的表达式使用约束求解器进行求解,在当前例子中最终根节点的值 false,因此可以直接跳过约束求解步骤.执行 GC-MCR 方法的后续操作,检测并发程序中的并发缺陷.



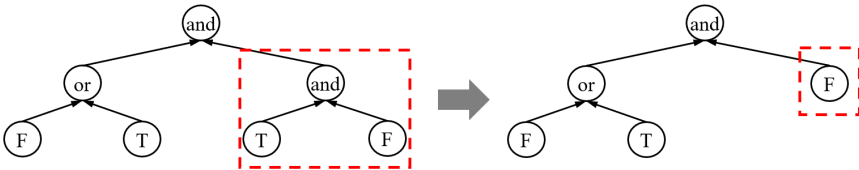
(a) 父节点为 and,两个子节点皆为 true



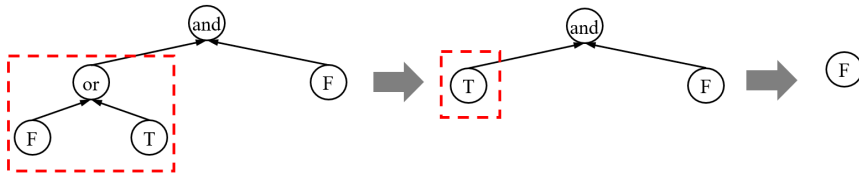
(b) 父节点为 or,两个子节点皆为 true



(c) 父节点为 or,两个子节点皆为 false



(d) 父节点为 and,两个子节点中至少一个为 false



(e) 父节点为 or,两个子节点中仅一个为 true

图9 前缀树简化的示例

### 3 实验设计与结果分析

#### 3.1 研究问题

为了验证 GC-MCR 方法的有效性,本文从程序的约束求解次数、程序运行时间和程序的调度次数的角度设计了 3 个研究问题.

**RQ1: GC-MCR 方法是否有效减少程序约束求解的次数,减少约束求解器的计算资源消耗?**

由于调用约束求解器进行约束求解相比于程序调度更加耗时,如果可以减少程序的约束求解次数,降低约束求解器的计算资源消耗,对并发缺陷检测有着重要意义.现有方法<sup>[18]</sup>在约束进行约束求解前没有对约束进行任何处理,使得许多可以提前被过滤的冲突约束被约束求解器求解,增加约束求解的次数,降低探索并发程序状态空间的效率.因此本研究问题中,本文希望分析:与现有方法相比,GC-MCR 方法能否有效减少程序调用约束求解器进行约束求解的次数,降低约束求解器的计算资源消耗.

### RQ2: GC-MCR 方法能否有效减少并发缺陷检测的时间,提高探索并发程序状态空间的效率?

程序检测到并发缺陷所需时间是评估并发缺陷检测效率的一个重要指标.通过统计程序约束求解的次数以及程序的运行时间,还可以分析两者之间的关系,即是否可以通过减少约束求解次数以降低并发缺陷检测所消耗的时间,以及影响并发缺陷检测的因素.因此在本研究问题中,本文希望从多种角度来分析:与现有方法相比,GC-MCR 方法能否有效减少并发缺陷检测的时间,提高并发程序状态空间的探索效率.

### RQ3:GC-MCR 方法是否可以保证现有方法的并发缺陷检测能力?

现有的研究方法在并发缺陷检测上已取得一定的效果,即可以使用最少的调度次数准确找到并发程序中的缺陷.而并发程序的调度次数同时也是衡量并发缺陷检测能力的一个主要指标.如果 GC-MCR 方法的调度次数与现有方法的调度次数相等甚至更少,且发现相同缺陷,则并发缺陷检测能力没有下降.因此,在本研究问题中,本文希望验证:与现有方法相比,GC-MCR 方法能否拥有相同甚至更优的并发缺陷检测能力.

## 3.2 实验配置

**测试对象.** 本文通过从现有研究<sup>[18,22,23]</sup>广泛使用的并发数据集中收集一组包含 38 个含有并发缺陷的测试程序作为测试对象.表 1 显示了这些类的详细信息.其中 ID 列表示测试程序的实验序号.测试程序的列表示测试程序的名称.代码行数的列表示测试程序中的代码行数(使用 Statistic 工具统计).线程数量的列表示测试程序中线程的数量.变量数量的列表示测试程序中定义的变量数量,其中包含来自它们父类的变量.注意变量的统计不区分变量的访问权限(例如,公共或私有),因为所有的实例变量都是正确共享的.方法数量的列表示测试程序中公共方法的数量,其中包括那些来自父类的方法.最后,并发缺陷类型的列表示测试程序中的并发缺陷的类型.特别的,为评估 GC-MCR 的有效性和可扩展性,本文在真实世界程序 weblech 上对 GC-MCR 进行了进一步验证,weblech 是一个网站下载工具,包含一个启动服务器、执行客户机请求和终止的测试驱动程序.相比与其他测试程序,weblech 包含更多的程序事件以及更大程序状态探索空间.

**对比方法.** 最大因果约减算法的 Java 版本的实现,即将探索并发程序状态空间的问题转换成约束求解问题,以检测并发缺陷.本文提出的 GC-MCR 方法不仅将探索并发程序状态空间的问题转换成约束求解问题,还在此基础上提出有向图表示方法,构建完整的并发程序约束并对其进行过滤和约减,降低方法在约束求解上的时间开销.

论文中所有相关实验的运行环境为具有一个四核 CPU Intel(R) Core(TM) CPU i7-7500U @ 2.70GHz 2.90GHz 和 8 G 内存,运行 Windows 10 企业版(64 位)的机器.每次程序执行的超时设置为 90 分钟.为了尽量减少随机性因素的影响,每个实验重复 10 次执行,最后取平均值作为实验结果

表 1 GC-MCR 测试对象集合描述

ID	测试程序	代码行数	线程数量	变量数量	方法数量	并发缺陷类型
1	account	377	10	1	19	Atomicity
2	airline	182	5	4	15	DataRace
3	alarmclock	372	9	19	21	DataRace
4	allocation	347	2	2	14	Atomicity
5	array	104	2	20	2	DataRace

6	bakery	126	2	6	5	Atomicity
7	boundedbuffer	159	3	1	2	Deadlock
8	bubblesort	160	4	4	7	DataRace
9	checkfield	65	2	2	4	DataRace
10	clean	145	12	10	9	DataRace
11	consistency	59	3	2	2	Atomicity
12	counter	57	2	2	2	DataRace
13	critical	117	2	1	2	Atomicity
14	cyclic	72	4	1	3	DataRace
15	datarace	16	2	10	2	DataRace
16	deadlock	64	2	3	4	Deadlock
17	dekker	118	2	6	2	Atomicity
18	dirkaccount	167	2	7	10	Deadlock
19	fileappender	410	2	7	33	Atomicity
20	hashcode	2461	2	1	3	Atomicity
21	lambport	160	2	9	2	Atomicity
22	mergesort	384	3	11	13	DataRace
23	numberaxis	1637	2	43	110	Atomicity
24	peruserpooldatasource	682	2	35	65	DataRace
25	peterson	85	2	5	2	Atomicity
26	pingpong	294	5	9	13	DataRace
27	pool	243	2	4	2	DataRace
28	rax	95	2	2	2	Deadlock
29	readerswriters	608	3	12	18	DataRace
30	replicatedcasestudies	1427	4	1	2	Deadlock
31	rvexample	75	2	3	2	DataRace
32	sharedobject	67	2	1	2	DataRace
33	sharedpooldatasource	516	2	30	51	Atomicity
34	simple	65	2	2	2	Atomicity
35	store	79	2	1	4	Atomicity
36	transfer	91	2	3	7	Deadlock
37	waitnotify	99	4	1	1	DataRace
38	weblech	35K	3	49	90	DataRace

### 3.3 结果分析

#### RQ1: GC-MCR 方法是否有效减少程序约束求解的次数,减少约束求解器的计算资源消耗?

为了回答 RQ1,本文系统地将 GC-MCR 方法应用于每个测试程序,并统计检测出第一个并发缺陷时所需要的约束求解次数、约束求解消耗时间,检查两种方法之间是否存在差异.为了与基准作对比,本文选择将 GC-MCR 与 J-MCR 方法进行了对比,实验结果如表 2 和表 3 所示.其中 GC-MCR 约束求解次数和 J-MCR 约束求解次数分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中第一个并发缺陷所需的约束求解次数;减少的约束求解次数表示相比于 J-MCR,通过使用 GC-MCR 方法可以减少的约束求解次数;减少的约束百分比表示减少的约束求解次数占 J-MCR 约束求解次数的百分比;GC-MCR 约束求解耗时和 J-MCR 约束求解耗时分别表示使用 GC-MCR 方法和 J-MCR 方法检测出测试程序第一个并发缺陷时,约束求解器进行求解消耗的总时间;减少的求解时间表示相比于 J-MCR,通过使用 GC-MCR 方法可以减少的使用约束求解器进行求解消耗的总时间;减少的求解时间百分比表示减少的求解时间占 J-MCR 约束求解耗时的百分比;GC-MCR 约束过滤耗时表示使用 GC-MCR 方法找到测试程序中第一个并发缺陷所需的对约束进行过滤和约减消耗的总时间;过滤求解时间百分比表示 GC-MCR 约束过滤耗时占 J-MCR 约束求解耗时的百分比.

根据表 2 和表 3 中的结果,本文得到如下发现.首先,与 J-MCR 相比,GC-MCR 方法在大部分测试程序上可以进行更少次数的约束求解,使用 GC-MCR 可以平均减少 20%左右的约束求解,最高能够减少 71.43%的约束求解.并且对于在 J-MCR 中约束求解次数少的测试程序,其在使用 GC-MCR 后的过滤的约束求解次数占比越多.其次,由于 GC-MCR 方法减少了大部分测试程序的约束求解次数,而且对少部分没有减少求解次数的测试程序进行了约束表达式约减,所以 GC-MCR 的约束求解耗时相比 J-MCR 更低.与 J-MCR 相比,使用 GC-MCR 可以平均减少 37%左右的用于约束求解的计算资源,最多能够减少 71.46%的用于约束求解的计算资源.最后,GC-MCR 方法对约束进行过滤和约减所需的时间远少于使用 J-MCR 减少的约束求解时间,其最高仅占 J-MCR 约束求解耗时的 8%.因此可以得出由于基于有向图的约束约减和约束过滤时间开销远小于求解器求解带有冲突约束的时间开销,通过对约束表达式进行提前过滤并约减,确实能够有效减少程序调用约束求解器进行约束求解的

次数,降低程序在约束求解部分的耗时,减少约束求解器的计算资源消耗。

通过进一步分析发现,在 J-MCR 算法中约束求解次数多的测试程序,其按特点被分为两种:线程多的测试程序和线程相对较少,但每个线程中对同一变量的读写事件很多的测试程序。前者由于其线程较多,线程的交织数量会很多,因此需要进行大量的约束求解。然而在使用了 GC-MCR 方法后,GC-MCR 会在固定程间的执行先后顺序后去判断读写约束是否冲突,固定线程执行顺序可以更加充分地判断冲突,过滤掉更多约束,使 GC-MCR 检测并发缺陷的效果更好。后者由于线程数量较少,对同一变量的读写事件很多,因此即使固定了线程的执行顺序,很多读写事件的顺序组合依旧成立,故使用 GC-MCR 方法不一定过滤掉很多约束。因此导致过滤约束的效果很不稳定。未来可以进行更深一步的研究去解决这个问题。

表 2 GC-MCR 与 J-MCR 的约束求解次数对比

ID	测试程序	GC-MCR 约束 求解次数/次	J-MCR 约束 求解次数/次	减少的约束 求解次数/次	减少的约束 百分比/%
1	account	12	12	- 0	- 0.00
2	airline	2303	2530	↓ 227	↑ 8.97
3	alarmclock	24526	26799	↓ 2273	↑ 8.48
4	allocation	30	36	↓ 6	↑ 16.67
5	array	399	437	↓ 38	↑ 8.70
6	bakery	75	93	↓ 18	↑ 19.35
7	boundedbuffer	16	16	- 0	- 0.00
8	bubblesort	272	286	↓ 14	↑ 4.90
9	checkfield	4	14	↓ 10	↑ 71.43
10	clean	25	25	- 0	- 0.00
11	consistency	8	12	↓ 4	↑ 33.33
12	counter	66	67	↓ 1	↑ 1.49
13	critical	2	3	↓ 1	↑ 33.33
14	cyclic	100	100	- 0	- 0.00
15	datarace	5	15	↓ 10	↑ 66.67
16	deadlock	1	1	- 0	- 0.00
17	dekker	342	492	↓ 150	↑ 30.49
18	dirkaccount	425	456	↓ 31	↑ 6.80
19	fileappender	172	175	↓ 3	↑ 1.71
20	hashcode	11	32	↓ 21	↑ 65.63
21	lampport	372	515	↓ 143	↑ 27.77
22	mergesort	317	546	↓ 229	↑ 41.94
23	numberaxis	175	242	↓ 67	↑ 27.69
24	peruserpooldatasource	7	8	↓ 1	↑ 12.50
25	peterson	28	37	↓ 9	↑ 24.32
26	pingpong	28	28	- 0	- 0.00
27	pool	9	9	- 0	- 0.00
28	rax	31	31	- 0	- 0.00
29	readerswriters	1324	1339	↓ 15	↑ 1.12
30	replicatedcasesstudies	104	104	- 0	- 0.00
31	rvexample	76	92	↓ 16	↑ 17.39
32	sharedobject	3	8	↓ 5	↑ 62.50
33	sharedpooldatasource	4	5	↓ 1	↑ 20.00
34	simple	12	18	↓ 6	↑ 33.33
35	store	23	28	↓ 5	↑ 17.86
36	transfer	2	2	- 0	- 0.00
37	waitnotify	1159	1456	↓ 297	↑ 20.40
38	weblech	1124	2297	↓ 1173	↑ 51.07

表 3 GC-MCR 与 J-MCR 的约束求解时间对比

ID	测试程序	GC-MCR 约束 求解耗时/ms	J-MCR 约束 求解耗时/ms	减少的求解 时间/ms	减少的求解 时间百分比/%	GC-MCR 约束 过滤耗时/ms	过滤求解 时间百分 比/%
1	account	794	1029	↓ 235	↑ 22.84	15	1.46
2	airline	304448	360345	↓ 55897	↑ 15.51	246	0.07

3	alarmclock	3658195	3880957	↓ 222762	↑ 5.74	3206	0.08
4	allocation	2278	2646	↓ 368	↑ 13.91	60	2.27
5	array	56899	98283	↓ 41384	↑ 42.11	272	0.28
6	bakery	8860	21655	↓ 12795	↑ 59.09	95	0.44
7	boundedbuffer	1011	1373	↓ 362	↑ 26.37	38	2.77
8	bubblesort	48876	62142	↓ 13266	↑ 21.35	443	0.71
9	checkfield	289	757	↓ 468	↑ 61.82	21	2.77
10	clean	7236	11049	↓ 3813	↑ 34.51	64	0.58
11	consistency	544	896	↓ 352	↑ 39.29	29	3.24
12	counter	28457	33786	↓ 5329	↑ 15.77	82	0.24
13	critical	172	232	↓ 60	↑ 25.86	12	5.17
14	cyclic	23573	27273	↓ 3700	↑ 13.57	70	0.26
15	datarace	353	1158	↓ 805	↑ 69.52	31	2.68
16	deadlock	93	117	↓ 24	↑ 20.51	7	5.98
17	dekker	77291	154573	↓ 77282	↑ 50.00	70	0.05
18	dirkaccount	101272	216110	↓ 114838	↑ 53.14	1098	0.51
19	fileappender	21218	29786	↓ 8568	↑ 28.77	220	0.74
20	hashcode	970	2374	↓ 1404	↑ 59.14	36	1.52
21	lambert	22329	38688	↓ 16359	↑ 42.28	86	0.22
22	mergesort	245428	347597	↓ 102169	↑ 29.39	11404	3.28
23	numberaxis	188382	265010	↓ 76628	↑ 28.92	3593	1.36
24	peruserpooldatasource	1040	3644	↓ 2604	↑ 71.46	17	0.47
25	peterson	10905	17111	↓ 6206	↑ 36.27	34	0.20
26	pingpong	8918	13409	↓ 4491	↑ 33.49	28	0.21
27	pool	514	622	↓ 108	↑ 17.36	14	2.25
28	rax	11302	16625	↓ 5323	↑ 32.02	58	0.35
29	readerswriters	220750	322420	↓ 101670	↑ 31.53	2823	0.88
30	replicatedcasestudies	16219	29711	↓ 13492	↑ 45.41	133	0.45
31	rvexample	4512	7548	↓ 3036	↑ 40.22	30	0.40
32	sharedobject	224	562	↓ 338	↑ 60.14	15	2.67
33	sharedpooldatasource	477	1244	↓ 767	↑ 61.66	19	1.53
34	simple	689	1580	↓ 891	↑ 56.39	22	1.39
35	store	2165	3037	↓ 872	↑ 28.71	46	1.51
36	transfer	153	194	↓ 41	↑ 21.13	16	8.25
37	waitnotify	108400	179769	↓ 71369	↑ 39.70	142	0.08
38	weblech	207126	406561	↓ 199435	↑ 49.05	497	0.12

## RQ2: GC-MCR 方法能否有效减少并发缺陷检测的时间,提高探索并发程序状态空间的效率?

为了回答 RQ2,本文系统地将 GC-MCR 方法应用于每个测试程序,统计当发现第一个并发缺陷时所需要的时间,检查两种方法之间是否存在差异.本文选择将 GC-MCR 与 J-MCR 方法进行了对比,实验结果如表 4 所示.其中 GC-MCR 平均执行时间和 J-MCR 平均执行时间分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中第一个并发缺陷所需的平均时间;程序执行相差时间表示被测程序使用 J-MCR 和使用 GC-MCR 所需的时间的差值;性能提升表示程序执行相差时间占 J-MCR 平均执行时间的百分比.

根据表 4 中的内容,本文观察到以下结果.首先,相比于 J-MCR 方法,使用 GC-MCR 方法过滤并简化约束后可以有效减少并发缺陷检测的时间,总调用时间平均减少 34.01%,即程序的总体性能平均提升 34.01%.其次,再根据表 2,可以发现过滤约束百分与性能提升的百分比不成正比.再次,对于那些 J-MCR 平均执行时间越短的测试程序,即使用 J-MCR 时约束求解次数越多的测试程序,使用 GC-MCR 方法过滤并约减约束后其性能提升越高.最后,对于那些没有约束被过滤的测试程序,使用 GC-MCR 方法约减其约束同样可以减少程序执行时间,提升性能.特别的,在真实世界应用程序 weblech 中,本文在 RQ1 和 RQ2 得到的结论依旧有效,且进一步验证了 GC-MCR 方法的有效性,,表明 GC-MCR 能够检测状态空间庞大的并发程序的并发缺陷,并可应用于真实世界应用软件.

GC-MCR 平均执行时间由过滤和约减约束耗时、约束求解耗时和程序调度耗时三部分组成.进一步分析发现,根据表 3,过滤和约减约束的时间开销远小于求解器求解带有冲突约束的时间开销,故一旦存在可过滤的约束或可约减的约束,即可减少约束求解部分的时间开销(约束求解部分包括约束过滤和约减).继而通过

GC-MCR 总体的并发缺陷检测时间降低,发现约束求解部分减少的时间开销远多于程序调度的耗时开销的变化量,即缺陷检测时间减少主要来自约束求解部分减少的时间开销与过滤和程序调度的耗时开销变化之差.该结果出现的原因仍是程序约束求解的时间开销远大于程序调度的时间开销,GC-MCR 仅对约束进行过滤和约减,未改变程序的调度,致使两者的变化量存在与之前相同的数量关系.

表 4 GC-MCR 与 J-MCR 的平均执行时间对比

ID	测试程序	GC-MCR 平均 执行时间/ms	J-MCR 平均 执行时间/ms	程序执行 相差时间/ms	性能提升/%
1	account	929	1170	↓ 241	↑ 20.60
2	airline	308183	363946	↓ 55763	↑ 15.32
3	alarmclock	3751189	3976459	↓ 225270	↑ 5.67
4	allocation	2445	2813	↓ 368	↑ 13.08
5	array	57881	98859	↓ 40978	↑ 41.45
6	bakery	9104	21856	↓ 12752	↑ 58.35
7	boundedbuffer	1143	1509	↓ 366	↑ 24.25
8	bubblesort	49650	62486	↓ 12836	↑ 20.54
9	checkfield	400	811	↓ 411	↑ 50.68
10	clean	7380	11248	↓ 3868	↑ 34.39
11	consistency	699	982	↓ 283	↑ 28.82
12	counter	28581	33977	↓ 5396	↑ 15.88
13	critical	271	320	↓ 49	↑ 15.31
14	cyclic	23361	27442	↓ 4081	↑ 14.87
15	datarace	595	1641	↓ 1046	↑ 63.74
16	deadlock	172	243	↓ 71	↑ 29.22
17	dekker	78075	155317	↓ 77242	↑ 49.73
18	dirkaccount	102830	218181	↓ 115351	↑ 52.87
19	fileappender	21954	30468	↓ 8514	↑ 27.94
20	hashcode	1233	2557	↓ 1324	↑ 51.78
21	lambport	22791	39056	↓ 16265	↑ 41.65
22	mergesort	284283	380757	↓ 96474	↑ 25.34
23	numberaxis	194446	271151	↓ 76705	↑ 28.29
24	peruserpooldatasource	1560	3980	↓ 2420	↑ 60.80
25	peterson	11271	17240	↓ 5969	↑ 34.62
26	pingpong	9048	13640	↓ 4592	↑ 33.67
27	pool	720	850	↓ 130	↑ 15.29
28	rax	11426	16830	↓ 5404	↑ 32.11
29	readerswriters	224668	327420	↓ 102752	↑ 31.38
30	replicatedcasesstudies	16667	30071	↓ 13404	↑ 44.57
31	rvexample	4840	7724	↓ 2884	↑ 37.34
32	sharedobject	330	650	↓ 320	↑ 49.23
33	sharedpooldatasource	920	1590	↓ 670	↑ 42.14
34	simple	821	1669	↓ 848	↑ 50.81
35	store	2310	3129	↓ 819	↑ 26.17
36	transfer	270	330	↓ 60	↑ 18.18
37	waitnotify	110780	182196	↓ 71416	↑ 39.20
38	weblech	9296869	17532083	↓ 8235214	↑ 46.97

### RQ3:GC-MCR 方法是否可以保证现有方法的并发缺陷检测能力?

为了回答 RQ3,本文系统地将 GC-MCR 方法应用于每个测试程序,并统计其发现第一个并发缺陷的类型,检查两种方法之间是否存在差异,实验结果如表 5 所示.其中 GC-MCR 调度次数和 J-MCR 调度次数分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中的并发缺陷所需的调度次数;GC-MCR 的并发缺陷和 J-MCR 的并发缺陷分别表示使用 GC-MCR 方法和 J-MCR 方法找到的测试程序中的并发缺陷.

根据表 5 中的内容,本文观察到以下结果.首先,GC-MCR 能够正确检测出测试程序中的并发缺陷,GC-MCR 和 J-MCR 检测到的并发缺陷一致,且在大部分测试程序上以相同的线程调度触发了并发错误,即 GC-MCR 方法与 J-MCR 相比并发缺陷的检测能力没有任何损失.当程序状态空间较小时随机性对程序并发空间的探索影响

较小,即与 J-MCR 相比 GC-MCR 可以通过对并发程序执行相同次数检测出发缺陷.其次,GC-MCR 在 airline, alarmclock 以及 weblech 三个测试程序上相比于 J-MCR 可以通过更少的线程调度触发并发错误表明,随着程序状态空间庞大时随机性对程序并发空间的探索影响增大,冗余的约束导致现有方法探索冗余的线程调度,从而降低了检测效率.而 GC-MCR 可以通过比 J-MCR 更少次数的程序执行检测出发缺陷,有效过滤冲突以及约减冗余约束.即通过 GC-MCR 过滤掉的冲突约束都是无效约束,对于冗余约束地约减也是等效且合理的.因此约束过滤和约束约减后,测试框架的完备性没有降低,没有漏掉任何线程交织序列,说明 GC-MCR 方法在提升了性能的同时没有漏掉任何测试线程序列,既提升了并发约束检测的效率,又保证了测试框架的完备性.

通过进一步分析发现,GC-MCR 方法使用有向图对约束进行过滤以及约减,对于约束求解步骤中的约束部分,GC-MCR 要么选择过滤当前约束,要么选择对当前约束进行约减,得到等价的约束表达式,其中没有任何操作对约束的求解结果进行修改,故尽管程序执行存在随机性,GC-MCR 仍可以保证调度方案与 J-MCR 一致或更少,故 GC-MCR 保证了其完备性,GC-MCR 方法可以保证现有方法的并发缺陷检测能力.

表 5 GC-MCR 与 J-MCR 的有效调度次数对比

ID	测试程序	调度次数		并发缺陷	
		GC-MCR	J-MCR	GC-MCR	J-MCR
1	account	3	3	Atomicity	Atomicity
2	airline	434	445	DataRace	DataRace
3	alarmclock	888	915	DataRace	DataRace
4	allocation	4	4	Atomicity	Atomicity
5	array	22	22	DataRace	DataRace
6	bakery	15	15	Atomicity	Atomicity
7	boundedbuffer	1	1	Deadlock	Deadlock
8	bubblesort	10	10	DataRace	DataRace
9	checkfield	5	5	DataRace	DataRace
10	clean	2	2	DataRace	DataRace
11	consistency	4	4	Atomicity	Atomicity
12	counter	2	2	DataRace	DataRace
13	critical	2	2	Atomicity	Atomicity
14	cyclic	8	8	DataRace	DataRace
15	datarace	3	3	DataRace	DataRace
16	deadlock	2	2	Deadlock	Deadlock
17	dekker	145	145	Atomicity	Atomicity
18	dirkaccount	2	2	Deadlock	Deadlock
19	fileappender	3	3	Atomicity	Atomicity
20	hashcode	5	5	Atomicity	Atomicity
21	lampport	75	75	Atomicity	Atomicity
22	mergesort	2	2	DataRace	DataRace
23	numberaxis	17	17	Atomicity	Atomicity
24	peruserpooldatasource	3	3	DataRace	DataRace
25	peterson	5	5	Atomicity	Atomicity
26	pingpong	4	4	DataRace	DataRace
27	pool	2	2	DataRace	DataRace
28	rax	2	2	Deadlock	Deadlock
29	readerswriters	10	10	DataRace	DataRace
30	replicatedcasestudies	1	1	Deadlock	Deadlock
31	rvexample	21	21	DataRace	DataRace
32	sharedobject	3	3	DataRace	DataRace
33	sharedpooldatasource	1	1	Atomicity	Atomicity
34	simple	6	6	Atomicity	Atomicity
35	store	2	2	Atomicity	Atomicity
36	transfer	2	2	Deadlock	Deadlock
37	waitnotify	560	560	DataRace	DataRace
38	weblech	1096	2024	DataRace	DataRace

#### 4 讨论

本文提出了基于有向图的并发约束表示方法,并基于此设计并实现构建约束的并发缺陷检测方法 GC-MCR.现有的研究方法基于最大因果模型构建并发程序约束,然而在并发程序的表示上尚没有系统的研究,



本文提出的基于有向图的表示方法从结构上优化了并发程序约束的表示.一方面,借助有向图表示的优势在构建过程中即可识别出不可解的约束,减少了约束求解的成本;另一方面,精减了构建的并发程序约束,对于相同的线程交织,GC-MCR 可以构建出更准确更精减的并发程序约束,继而提升了约束求解速度.因此在有限的测试资源下,约束求解的成本降低,约束求解部分节省的资源用于探索并发程序状态空间,故而并发缺陷的检测效率提升.实验结果也表明,相比于现有方法,GC-MCR 可以更快的找出并发程序中的缺陷.然而通过分析实验结果发现,GC-MCR 在检测效率上的提升并不稳定,在平均执行时间的提升从 5%到 64%不等.同时,在调度次数上的优势并不明显,接下来本章将从程序状态空间、因果模型及并发程序三个角度进行讨论.

**GC-MCR 性能提升受并发程序状态空间影响.**并发程序的状态空间随着并发程序内的线程数量及读写操作数量的增长而呈指数级增长.同时,随着并发程序状态空间的增长,找到并发缺陷前的约束求解次数及程序调度次数也随之增长.从实验结果提升的百分比中可以发现,受程序状态空间的影响,GC-MCR 在约束求解次数及平均时间上的提升并不稳定.一方面,基于现有的最大因果模型,程序的线程数量读写操作越多其构建的有效并发约束越多.从表 2 的实验结果中可以看出,减少的约束求解次数随着程序状态空间的增长而增长,然而约束百分比的提升却因为其庞大的有效并发约束空间的增长而减少.另一方面,本文基于有向图的构建约束方法相比于现有方法,需要对约束进行重新表示并分析.因此随着程序状态空间的增长,有向图表示方法的成本也随之增长.从表 4 的结果中可以看出,由于约束求解过于耗时,有向图表示方法的成本依然低于现有方法约束求解的成本.

**调度次数受最大因果模型的限制.**本文提出基于有向图对并发程序进行表示,并基于此提出并发程序约束的构建方法.已有研究表明,最大因果模型可以最大化的减少探索冗余的并发程序状态空间.因此本文约束构建的核心模型同样基于最大因果模型.从表 5 的实验结果可以看出,GC-MCR 的有效调度序列与现有方法相比优势并不明显,其采用相同的最大因果模型为主要原因.GC-MCR 的优势在于更有效的生成符合最大因果模型的并发约束,减少约束求解的成本,继而更快地找到并发缺陷.

**各项评测数据与并发测试程序的关系.**本文用于评测 GC-MCR 和 J-MCR 并发缺陷检测能力的评测指标分别为约束求解次数、平均执行时间、约束过滤耗时、调度次数.由于本文基于最大因果模型,约束求解次数、约束过滤耗时和调度次数都受限制于最大因果模型,因此三者都与并发测试程序中的线程数量和不同线程中对变量的读写次数相关.而平均执行时间不但与约束求解次数、约束过滤耗时和调度次数相关联,而且硬件的配置及执行速度也会影响到平均执行时间.由于平均执行时间和调度次数能够直观且全面地评价方法的并发缺陷检测能力,本文将调度次数和平均执行时间作为评测的评测指标.

#### 4.1 有效性影响因素分析

本小节主要对可能会影响到本文实验结果有效性的因素进行分析,具体分析如下.

**内部有效性分析.**考虑本文方法的实现较复杂,因此可能影响到本文实验结果有效性的内部因素主要为本文方法的代码实现是否正确.为了减少代码实现的误差给本文实验结果带来的影响,本文使用到的相关技术均使用成熟的第三方框架,例如,ASM<sup>[24]</sup>,Z3<sup>[20]</sup>等.同时,本文参考了已有研究成果的代码实现,例如,MCR<sup>[18]</sup>,CovCon<sup>[25]</sup>等,且编码过程采用 peer review 的方式,最大程度上保证了代码实现的正确性.

**外部有效性分析.**影响本文实验结果的结论是否具有一般性的因素主要包含三个因素.第一个因素是本文使用的数据集是否具有代表性,为了确保本文研究结论的有效性,本文采用现有并发缺陷检测研究中常用的 27 个并发数据集,该数据集涵盖了并发缺陷常见的错误类型.另一方面,由于本文依赖的约束求解器 Z3 是 CPU 密集型工具,因此第二个影响因素为实验平台是否一致,为了减少实验平台对本文实验结果带来的影响,本文所有实验均在同一实验平台上进行,同时,所有实验均运行多次以消除可能的随机性对实验结果造成的影响.第三个影响因素为本文使用的对比方法是否具有代表性.本文选择在 Java 程序上验证 GC-MCR 的有效性.相比之下,其他并发测试方法,例如基于饱和覆盖的主动测试方法 Maple<sup>[26]</sup>以及 IDAT<sup>[27]</sup>,以及基于 CBMC 实现的多个并发测试方法<sup>[28,29,30]</sup>均是在 C 语言上对其方法的有效性进行了验证.SeqCheck<sup>[23]</sup>通过对程序分支信息进行建模,通过预测事件序列的可行性来检测 Java 程序中的并发缺陷,然而该方法并未公开源代码.考虑到上述工作在

Java 程序上进行复现的成本较高,且无法保障复现程序的正确性.同时,在最新研究工作的 J-MCR 已经验证了其方法的先进性,具有一定的代表性,故本工作主要与 J-MCR 进行对比.

**结论有效性分析.**影响本文结论有效性的因素主要为本文在实验分析过程中使用的评价指标是否合理.针对此问题,本文采用现有并发缺陷检测研究中两个常用的评判并发缺陷检测能力的评价指标,检测到第一个并发缺陷所用的时间及检测到第一个并发缺陷时并发程序的调度次数.同时,为了衡量 GC-MCR 是否能有效的减少约束求解的次数,本文引入约束求解次数评价指标,进一步确保本文结论的有效性.

## 5 相关工作

### 5.1 并发程序覆盖度量

并发程序覆盖标准可用于量化测试套件的丰富程度(例如,程序是否经过充分测试)或提供有关测试用例生成的实用指南(例如,作为程序模糊引擎中使用的目标函数).例如,HaPSet<sup>[31]</sup>是收集程序中的排序约束,并通过分析约束来指导程序的测试.CovCon<sup>[25]</sup>方法是通过分析记录的执行来提取常见执行的方法对,生成有可能覆盖那些未覆盖方法对的测试用例.TSA<sup>[33]</sup>旨在通过生成线程调度来覆盖未覆盖的覆盖需求(Coverage Requirement),从而实现对于并发程序的高同步覆盖.ConSuite<sup>[32]</sup>方法是通过静态分析线程交织集并且检查程序执行记录以检查特定线程交织是否被覆盖,然后使用遗传算法来生成可以覆盖更多线程交织的测试.AutoConTest<sup>[33]</sup>方法则是考虑调用上下文信息,动态以及迭代地计算出覆盖需求,根据顺序覆盖生成顺序测试,将顺序测试组装成并发测试.

Yang 等人<sup>[35]</sup>提出了一种基于全双路径覆盖的定义使用对覆盖(Definition-use Coverage).Krena 等人<sup>[36]</sup>提出了一种基于现有动态或静态分析方法(例如 Eraser<sup>[37]</sup>和 GoldiLocks<sup>[38]</sup>)为测试并发软件推导出新覆盖率指标的方法.他们扩展了多个现有的并发测试覆盖率指标,例如 ConcurPairs、定义使用覆盖(Definition-use Coverage)和同步对覆盖(Synchronisation Pair Coverage).MAP-Coverage<sup>[22]</sup>方法使用内存访问模式(Memory Access Pattern)抽象测试执行并测量覆盖了哪些内存访问模式,然后生成可以覆盖更多内存访问模式的测试用例.MAP-Coverage 比线程交织覆盖更抽象,并且比方法对覆盖更与 bug 相关.

### 5.2 模型检测和约束求解

现有并发缺陷检测研究中,模型检测技术<sup>[39,40,41]</sup>常被用于穷举探索线程调度空间,以此来检测程序中潜在的并发缺陷.例如,CHES<sup>[42]</sup>以上下文有界的方式动态探索目标程序的不同线程调度.Shacham 等人<sup>[40]</sup>基于模型检查器为 lockset 算法<sup>[43,44,45]</sup>报告的竞争构建测试用例.然而,面对程序路径和调度空间的指数级增长,模型检查技术难以扩展到大型多线程程序.故而,Huang 等人提出 MCR<sup>[18]</sup>方法,将探索并发程序状态空间的问题转换成约束求解问题,从而减少对并发程序冗余状态空间的探索,提升并发缺陷的检测效率.同样,STORM 方法<sup>[46]</sup>将约束求解与验证条件生成结合,通过构建 BoogiePL 程序的全局存储的映射之间的约束,从而检测 Windows 设备驱动程序中潜在的缺陷.

约束求解主要依赖 SMT 求解器或者 SAT 求解器,目前致力于提升求解器能力的工作<sup>[47,48,49]</sup>层出不穷,但是约束难解问题仍旧是亟待解决的问题.为此,Wang 等人<sup>[47]</sup>提出了通过深度强化学习来利用神经网络的决策能力加速约束求解.与上述工作不同,本文通过提出有向图表示方法,将并发程序约束中的程序基本约束构建有向图,并根据有向图对并发程序约束进行过滤和约减,提高约束求解过程的速度,降低约束求解的计算资源消耗.

### 5.3 并发程序缺陷检测

并发程序缺陷检测通常研究三个问题,即如何提高检测效率、如何提高检测的有效性以及如何减少误报.并发缺陷检测的经典方法有 happens-before 分析方法<sup>[50,51]</sup>和 lockset 算法<sup>[43,44,45]</sup>,它们被广泛用于检测并发程序的错误.例如,RaceChecker<sup>[50]</sup>是一个数据竞争检测器,它在报告要验证的潜在竞争之前使用 happens-before 关系来修剪不可行的竞争.Eraser<sup>[36]</sup>提出了一种 lockset 算法,通过监视每个共享内存引用和锁定行为来检测基于锁的并发程序中的错误.lockset 的改进方法<sup>[43,44,45,52]</sup>都对 lockset 算法进行了改进,以减少开销和误报.lockset 与

happens-before 的组合算法<sup>[38,53,54,55,56]</sup>将 lockset 算法与 happens-before 分析方法相结合.然而,由于静态分析方法的局限性,与动态并发缺陷检测方法相比,lockset 算法和 happens-before 分析方法经常会出现误报.

误报意味着与缺陷无关的线程交互被视为错误.静态检测技术不执行程序,而是通过分析源代码来检测缺陷.因此检测器无法正确确定 happens-before 信息,这可能会导致误报.MCR<sup>[18]</sup>方法将探索并发程序状态空间的问题转换成约束求解问题,根据每次求解结果控制程序的调度,能够使用最少的程序执行次数探索整个程序的状态空间.ConRacer<sup>[57]</sup>方法使用控制流分析来构建调用图,使用上下文敏感的别名分析查找别名对象,最后使用 happens-before 分析以减少误报和漏报.Maple<sup>[26]</sup>提出了一种基于饱和覆盖的主动测试方法,Yue 等人<sup>[27]</sup>基于 Maple 提出了一种多线程程序的测试用例多样性度量.Alglave 等人<sup>[29]</sup>提出了一种整数差分逻辑编码,借助有界模型检查,而对部署的并发系统代码进行验证.He 等人<sup>[28]</sup>提出了一种应用于 SC 的多线程程序验证的排序一致性理论,实现了增量式一致性检验、最小冲突子句生成和专业化理论传播.Yin 等人<sup>[30]</sup>根据事件顺序图(Event Order Graph, EOG)设计了基于 EOG 的反例验证和优化生成算法,能够获得较小但有效的优化约束,基于此提出了一种基于调度约束的多线程 C 程序验证抽象细化方法.SeqCheck<sup>[23]</sup>方法对程序分支进行建模,并对事件序列的可行性进行预测.其首先计算事件集以确定序列的可行性,然后通过构造图以重新排序集合中的事件,最后计算图中顺序的闭包.

本文的工作与以上的工作都有一定的区别,本文不仅将探索并发程序状态空间的问题转换成约束求解问题,还在此基础上提出了有向图表示方法,构建完整的并发程序约束并对其进行过滤和约减,减少了约束求解成本,继而提高探索并发程序状态空间的效率.

## 6 总结与展望

本文基于有向图构建约束,加速最大因果约减算法的约束求解,在调用约束求解器前对约束进行操作,将读写约束进行拆分与解析,并将其和程序基本约束进行冲突检测,过滤冲突约束并约减冗余约束,减少单次求解耗时,甚至减少约束求解器的调用,减少约束求解器的计算资源消耗,继而提高探索并发程序中所有可能的状态的效率.本文在 38 组并发测试程序上对 GC-MCR 方法进行了实证研究.实验表明,GC-MCR 方法在保证了调度的完备性和 MCR 方法的并发缺陷检测能力的情况下,减少了单次约束求解的耗时和对约束求解器的调用,降低了约束求解器的计算资源消耗,提高了探索被测程序状态空间的效率,显著优于现有的方法.

在未来的研究工作中,可以添加一些可行性松弛,通过添加分支事件的可行性松弛相关算法,松弛分支事件的具体可行性约束要求,在不影响准确性的情况下进一步提高并发缺陷检测的效率.

## References:

- [1] Jiang YY, Xu C, Ma XX, Lü J. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: A survey. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):747–763. (in Chinese)
- [2] Bo LL, Jiang SJ, Zhang UM, Wang XY, Yu Q. Research progress on techniques for concurrency bug detection. Computer Science, 46(5):13–20, 1 2019. (in Chinese)
- [3] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal Of Computers, (11):2215–2233, 1 2015. (in Chinese)
- [4] Putchala MK, Bryant AR. Synchron-ITS: An Interactive Tutoring System to Teach Process Synchronization and Shared Memory Concepts in an Operating Systems Course. In: 2016 International Conference on Collaboration Technologies and Systems, CTS 2016, Orlando, FL, USA, October 31 - November 4, 2016, 2016: 180–187.
- [5] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, 2014: 337–348.
- [6] Park S, Vuduc RW, Harrold MJ. Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, 2010: 245–254.

- [7] Lu GZ, Xu L, Yang YB, Xu BW. Predictive analysis for race detection in software-defined networks. *Sci. China Inf. Sci.*, 2019, 62 (6): 62101:1–62101:20.
- [8] Cai Y, Lu Q. Dynamic Testing for Deadlocks via Constraints. *IEEE Trans. Software Eng.*, 2016, 42 (9): 825–842.
- [9] Engler DR, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003*, Bolton Landing, NY, USA, October 19-22, 2003, 2003: 237–252.
- [10] Letko Z, Vojnar T, Krena B. AtomRace: data race and atomicity violation detector and healer. In: *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), PADTAD 2008, Seattle, Washington, USA, July 20-21, 2008, 2008: 7.
- [11] Huang J, Zhang C, Dolby J. CLAP: recording local executions to reproduce concurrency failures. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, Seattle, WA, USA, June 16-19, 2013, 2013: 141–152.
- [12] Lu S, Tucek J, Qin F, Zhou YY. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. *IEEE Micro*, 2007, 27 (1): 26–35.
- [13] Clarke EM, Grumberg O, Minea M, Peled D. State Space Reduction Using Partial Order Techniques. In: *Int. J. Softw. Tools Technol. Transf.*, 1999, 2 (3): 279–287.
- [14] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, Long Beach, California, USA, January 12-14, 2005, 2005: 110–121.
- [15] Godefroid P. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.
- [16] Mazurkiewicz A. Trace theory. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Berlin, Heidelberg, 1987: 278–324.
- [17] Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 1978, 21 (7): 558–565.
- [18] Huang J. Stateless model checking concurrent programs with maximal causality reduction. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, USA, June 15-17, 2015, 2015: 165–174.
- [19] Serbanuta TF, Chen F, Rosu G. Maximal Causal Models for Sequentially Consistent Systems. In: *Runtime Verification, Third International Conference, RV 2012*, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, 2012: 136–150.
- [20] de Moura LM, Bjørner N. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*, Budapest, Hungary, March 29-April 6, 2008. *Proceedings*, 2008: 337–340.
- [21] Zhao YQ, Wang Z, Liu S, Sun J, Chen JJ and Chen X. Achieving High MAP-Coverage through Pattern Constraint Reduction. In: *IEEE Transactions on Software Engineering*. Los Alamitos, CA, USA, jan 2022: 1–1.
- [22] Wang Z, Zhao YQ, Liu S, Sun J, Chen X, Lin HR. MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes. In: *34<sup>th</sup> IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, San Diego, CA, USA, November 11-15, 2019, 2019: 722–734.
- [23] Cai Y, Yun H, Wang JQ, Qiao L, Palsberg J. Sound and efficient concurrency bug prediction. In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, August 23-28, 2021, 2021: 255–267.
- [24] ASM bytecode analysis framework. <http://asm.ow2.org/>.
- [25] Choudhary A, Lu S, Pradel M. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, Buenos Aires, Argentina, May 20-28, 2017, 2017: 266–277.
- [26] Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, New York, NY, USA, 2012: 485–502.

- [27] Yue H, Wu P, Chen T-Y, Lv Y. Input-Driven Active Testing of Multi-threaded Programs. In: 2015 Asia-Pacific Software Engineering Conference (APSEC), 2015: 246–253.
- [28] He F, Sun ZH, Fan HY. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, 2021: 1264–1279.
- [29] Alglave J, Kroening D, Tautschnig M. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, 2013: 141–157.
- [30] Yin L, Dong W, Liu WW, Wang J. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Trans. Software Eng.*, 2020, 46 (5): 549–565.
- [31] Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, 2011: 221–230.
- [32] Hong S, Ahn J, Park S, Kim M, Harrold MJ. Testing concurrent programs to achieve high synchronization coverage. In: International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, 2012: 210–220.
- [33] Steenbuck S, Fraser G. Generating Unit Tests for Concurrent Classes. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, 2013: 144–153.
- [34] Terragni V, Cheung S-C. Coverage-driven test code generation for concurrent classes. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, 2016: 1121–1132.
- [35] Yang CD, Souter AL, Pollock LL. All-du-path Coverage for Parallel Programs. In: Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1998, Clearwater Beach, Florida, USA, March 2-5, 1998, 1998: 153–162.
- [36] Krena B, Letko Z, Vojnar T. Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software. In: Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers, 2011: 177–192.
- [37] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 1997, 15 (4): 391–411.
- [38] Choi JD, Lee K, Loginov A, O’Callahan R, Sarkar V, Sridharan M. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002, 2002: 258–269. <https://doi.org/10.1145/512529.512560>.
- [39] Musuvathi M, Qadeer S, Ball T, Basler G. Finding and Reproducing Heisenbugs in Concurrent Programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings, 2008: 267–280.
- [40] Shacham O, Sagiv M, Schuster A. Scaling model checking of dataraces using dynamic informatio. *J. Parallel Distributed Comput.*, 2007, 67 (5): 536–550.
- [41] Khurshid S, Pasareanu CS, Visser W. Test input generation with Java PathFinder: then and now (invited talk abstract). In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, 2018: 1–2.
- [42] Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, 2007: 446–455.
- [43] Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. In Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003, 2003: 237–252.
- [44] Nishiyama H. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In: Proceedings of the 3rd Virtual Machine Research and Technology Symposium, May 6-7, 2004, San Jose, CA, USA, 2004:127–138.
- [45] Elmas T, Qadeer S, Tasiran S. Precise race detection and efficient model checking using Locksets. MSR-TR-2005-118.

- [46] Lahiri SK, Qadeer S, Rakamaric Z. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, 2009: 509–524.
- [47] Wang F, Rompf T. From Gameplay to Symbolic Reasoning: Learning SAT Solver Heuristics in the Style of Alpha(Go) Zero. CoRR, 2018, abs/1802.05340.
- [48] Bello I, Pham H, Le QV, Norouzi M, Bengio S. Neural Combinatorial Optimization with Reinforcement Learning. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings, 2017.
- [49] Xu L, Hoos H, Leyton-Brown K. Predicting Satisfiability at the Phase Transition. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada, 2012.
- [50] Lu K, Wu Z, Wang XP, Chen C, Zhou X. RaceChecker: Efficient Identification of Harmful Data Races. In 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015, 2015: 78–85.
- [51] Perkovic D, Keleher PJ. Online Data-Race Detection via Coherency Guarantees. In: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996, 1996: 47–57.
- [52] von Praun C, Gross TR. Object Race Detection. In: Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001, 2001: 70–82.
- [53] Yu Y, Rodeheffer T, Chen W. RaceTrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005, 2005: 221–234.
- [54] von Praun C, Gross TR. Static conflict analysis for multi-threaded object-oriented programs. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003, 2003: 115–128.
- [55] Elmas T, Qadeer S, Tasiran S. Goldilocks: a race and transaction-aware java runtime. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, 2007: 245–255.
- [56] Pozniansky E, Schuster A. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. In: *Concurr. Comput. Pract. Exp.*, 2007, 19 (3): 327–340.
- [57] Zhang Y, Liu H, Qiao L. Context-Sensitive Data Race Detection for Concurrent Programs. In: *IEEE Access*, vol. 9, pp. 2021.20861-20867.

#### 附中文参考文献:

- [1] 蒋炎岩,许畅,马晓星,吕建.获取访存依赖:并发程序动态分析基础技术综述.软件学报,28(4):745 - 763, 1 2017.
- [2] 薄莉莉,姜淑娟,张艳梅,王兴亚,于巧.并发缺陷检测技术研究进展.计算机科学, 46(5):13 - 20,1 2019.
- [3] 苏小红,禹振,王甜甜,马培军.并发缺陷暴露、检测与规避研究综述.计算机学报, (11):2215 - 2233, 1 2015.