软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(7):3064−3079 [doi: 10.13328/j.cnki.jos.006856]

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

基于约束依赖图的并发程序模型检测工具*

苏 杰[1,2], 杨祖超[1,2], 田 聪[1,2], 段振华[1,2]

[1](西安电子科技大学 计算机理论与技术研究所, 陕西 西安 710071)

[2](综合业务网理论及关键技术国家重点实验室(西安电子科技大学), 陕西 西安 710071)

通信作者: 田聪, E-mail: ctian@mail.xidian.edu.cn

摘 要: 模型检测是一种基于状态空间搜索的自动化验证方法, 可以有效地提升程序的质量. 然而, 由于并发程序中线程调度的不确定性以及数据同步的复杂性, 对该类程序验证时存在更为严重的状态空间爆炸问题. 目前, 大多采用基于独立性分析的偏序约简技术缩小并发程序探索空间. 针对粗糙的独立性分析会显著增加需探索的等价类路径问题, 开发了一款可细化线程迁移依赖性分析的并发程序模型检测工具CDG4CPV. 首先, 构造了待验证可达性性质对应的规约自动机; 随后, 根据线程迁移边的类型和共享变量访问信息构建约束依赖图; 最后, 利用约束依赖图剪裁控制流图在展开过程中的独立可执行分支. 在SV-COMP 2022竞赛的并发程序数据集上进行了对比实验, 并对工具的效率进行比较分析. 实验结果表明, 该工具可以有效地提升并发程序模型检测的效率. 特别是, 与基于BDD的程序分析算法相比, 该工具可使探索状态数目平均减少91.38%, 使时间和空间开销分别平均降低86.25%和69.80%.

关键词: 约束依赖图; 偏序约简; 并发程序; 模型检测; 工具

中图法分类号: TP311

中文引用格式: 苏杰, 杨祖超, 田聪, 段振华. 基于约束依赖图的并发程序模型检测工具. 软件学报, 2023, 34(7): 3064–3079. http://www.jos.org.cn/1000-9825/6856.htm

英文引用格式: Su J, Yang ZC, Tian C, Duan ZH. Model Checking Tool for Concurrent Program Based on Constrained Dependency Graph. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3064–3079 (in Chinese). http://www.jos.org.cn/1000-9825/6856.htm

Model Checking Tool for Concurrent Program Based on Constrained Dependency Graph

SU Jie[1,2], YANG Zu-Chao[1,2], TIAN Cong[1,2], DUAN Zhen-Hua[1,2]

[1](Institute of Computing Theory and Technology, Xidian University, Xi'an 710071, China)

[2](State Key Laboratory of Integrated Service Networks (Xidian University), Xi'an 710071, China)

Abstract: Model checking is an automatic verification approach based on the state-space exploration strategy, which can effectively improve the quality of a program. However, due to the non-deterministic of thread scheduling and the complexity of data synchronization, the state-space explosion problem in concurrent program verification is more serious. At present, the independence analysis based partial-order reduction techniques have been widely applied to reduce the exploration space of concurrent program verification tasks. In the face of the problem that imprecise independence analysis will significantly increase the number of equivalent trace classes to be explored, a concurrent program model checking tool CDG4CPV that can refine the dependencies between thread transitions has been developed. Firstly, specification automata are constructed corresponding to the reachability property. Then, a constrained dependency graph is constructed according to the types of transition edges of threads and the access information of shared variables. Finally, the constrained dependency graph is utilized to prune the independent and enabled branches when unwinding the control-flow graph. The experiments have been carried out on the concurrency track of SV-COMP 2022 benchmark suite, and the efficiency of the proposed tool is

* 基金项目: 国家自然科学基金(62192730, 62192734, 61732013, 62172322); 科技创新 2030——"新一代人工智能"重大项目(2018AAA0103202)

本文由"形式化方法与应用"专题特约编辑董云卫教授、刘关俊教授、毛晓光教授推荐.

收稿时间: 2022-09-04; 修改时间: 2022-10-08; 采用时间: 2022-12-05; jos在线出版时间: 2022-12-30

also compared and analyzed. Empirical results show that the proposed tool can effectively improve the efficiency of model checking for concurrent programs. Specially, compared with the BDD-based program analysis algorithm, the proposed tool reduces the number of explored states by 91.38%, and the time and memory overheads are reduced by 86.25% and 69.80%, respectively.
Key words: constrained dependency graph; partial-order reduction; concurrent program; model checking; tool

为了提升硬件计算资源的利用率, 并行化技术被广泛运用于大数据、云计算、高性能计算等应用场景, 以提升系统整体的吞吐量. 然而, 并发程序的编写相较于串行程序更为困难, 需正确处理共享变量(shared variable)的可见性、程序执行的有序性和操作的原子性等问题. 并且, 随着程序规模的不断扩大, 多个并发任务间的交替执行空间急剧增长, 线程同步逻辑愈发复杂, 致使线程死锁[1]、数据竞争[2]和原子性违背[3]等并发缺陷更加难以被发现. 因此, 如何高效地检测出这些并发缺陷, 在计算机基础研究领域具有非常重要的意义.

作为一种验证系统是否满足性质规约的形式化方法, 模型检测技术[4]通过自动化遍历程序状态空间, 并分析该空间中是否存在违反给定性质的状态以检测程序设计缺陷. 然而, 随着程序规模的不断扩大, 程序状态空间呈指数增长, 因此, 模型检测技术常需面对状态空间爆炸的难题. 为了缓和因显式枚举状态空间造成的程序验证规模受限问题[5], Burch 等人[6]提出了一种基于符号化表示的模型检测算法. 该算法利用二元决策图(binary decision diagram, BDD)[7]表示状态的集合, 并在状态空间遍历的单个步骤中同时考虑大量状态, 因此, 该算法可用于较大规模程序的验证. MacMillan[8]进一步充分利用 SAT 求解器生成反驳的能力,计算不可满足子句的 Craig 插值以排除不可行路径, 使符号模型检测效率得到显著提升. 此外, 为了降低并发程序调度约束公式的复杂度, Yin 等人[9]通过构建反例路径的事件序图(event order graph), 并基于该图计算较短但有效的细化约束, 避免了精确调度约束编码带来的大量时空开销. 在我们的前期工作[10]中, 提出了一种基于条件插值的并发程序模型检测方法, 该方法可以有效地缩小并发程序验证时需探索的状态空间. 具体而言, 条件插值用于限制每个状态的抽象可达域, 以此剪裁与抽象可达域冲突(conflict)的冗余条件分支. 条件插值也进一步地用于缩短插值路径的长度, 使插值路径公式的可满足性判定和条件插值计算的时间开销大幅度降低.

对于多个线程可交替执行的并发程序, 上述基于模型检测的方法通常存在更为严重的状态空间爆炸问题. 为了进一步缩小探索空间, 目前多采用基于独立性分析的偏序约简(partial-order reduction, POR)技术[11,12]对路径空间进行等价类划分, 并探索每个路径等价类中至少一条路径, 以保证程序行为分析的完备性. Wang 等人[13]提出了守卫独立迁移(guarded independent transition)的概念, 即这类迁移仅在特定执行路径中表现出与其他迁移相互独立的特性, 并且基于该概念编码的约束可容易地整合到 SMT/SAT 约束求解过程之中, 实现并发程序冗余分支的自动化剪裁. 其后续工作[14]通过维护线程迁移的依赖链(dependency chain), 并限制所有探索路径满足准单调计算(quasi-monotonic computation)准则, 使算法在不遗漏任何路径等价类的同时保证探索的最优性, 即任意两条探索路径均不属于相同的路径等价类. 动态偏序约简(dynamic partial-order reduction, DPOR)[15]通过动态地追踪线程间的交互以确定探索回溯点, 可以有效地提升分析精度并降低并发程序验证的时空开销. Abdulla 等人[16]提出了首个基于 source set 和 wakeup tree 的最优 DPOR 算法. 该算法所使用的 source set 有效地避免了休眠集阻塞(sleep set blocking)现象的发生, 减少了对冗余路径的探索. 此外, Chalupa 等人[17]通过在顺序一致性语义(sequential consistency semantics)下引入路径的观测等价, 使路径空间划分相较于基于 Mazurkiewicz 等价划分[18]更为粗略, 进而降低了所需探索的路径等价类和并发程序验证时空开销.

基于偏序约简的模型检测算法的效率在很大程度上取决于独立性分析的精确程度. 然而, 为了保证算法的正确性和完备性, 大多数偏序约简算法往往采用更为保守的独立性估计策略, 致使路径等价类过于细化, 所需探索等价类路径显著增多. 针对上述问题, 本文开发了一款可细化线程迁移依赖性的并发程序模型检测工具 CDG4CPV: 首先, 该工具构造了待验证可达性性质对应的规约自动机, 以实现对可达性性质的检测; 随后,根据线程迁移边的语句类型以及共享变量访问信息计算不同边之间相互依赖的约束条件, 构建并发程序的约束依赖图, 实现对独立性分析的进一步细化; 最后, 在展开控制流图时, 利用约束依赖图动态地计算每个状态处线程迁移的独立性, 并剪裁冗余的独立可执行分支. 为了评估所开发工具对并发程序的验证效率, 本文在软件验证竞赛 SV-COMP 2022 的并发程序数据集上与其他工具和偏序约简算法进行了实验对比. 实验

结果表明，所提工具可以有效地提升并发程序模型检测的效率. 相比于基于 BDD 的程序分析算法，所提工具可使探索状态数目平均减少 91.38%，时间和空间开销分别平均降低 86.25%和 69.80%.

本文第 1 节介绍并发程序控制流图和偏序约简等相关基础概念. 第 2 节详细描述工具 CDG4CPV 的整体框架，包括规约自动机的构造、约束依赖图的构造以及约束依赖图辅助的 BDD 程序分析过程. 第 3 节展示工具在 SV-COMP 2022 并发程序数据集上的实验结果. 第 4 节对全文工作进行总结并展望未来的研究方向.

1 基础概念

1.1 控制流图与抽象可达树

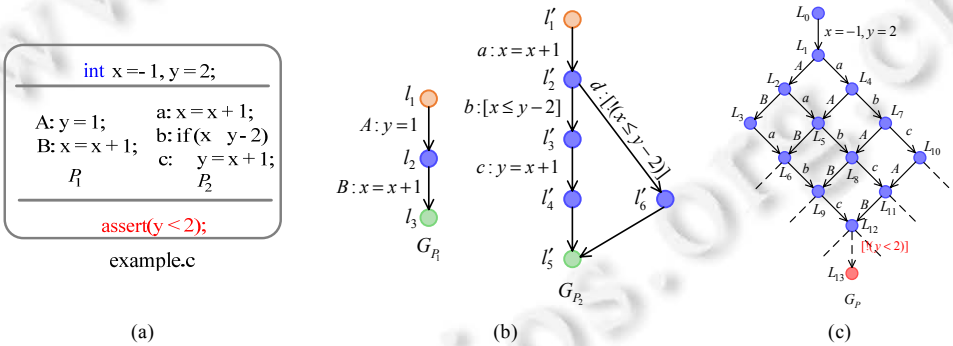
一个并发程序 P 由 N 个线程 P_i ($1\leqslant i\leqslant N$)组成. 每个线程 P_i 代表了操作系统运算调度的最小单位，其整体结构和执行过程可用线程控制流图 G_{P_i} 描述.

通过将这 N 个子控制流图并行合并，可得到程序 P 的控制流图 G_P.

定义 1(并发程序控制流图)[10]. 对于由 $N(N\geqslant 1)$个线程组成的并发程序 P，其每个线程的线程控制流图为四元组 $G_{P_i}=(L_{P_i},T_{P_i},l_{0P_i},f_{P_i})$ 表示的有向图. 而该程序的并发程序控制流图 $G_P=(L_P,T_P,l_{0P},f_P)$为这 N 个线程控制流图 G_{P_i} 的并行合并，其中:

- $L_P\subseteq L_{P_1}\times\ldots\times L_{P_N}$ 为程序 P 的位置集合，其中，L_{P_i} 表示线程 P_i 所处的位置.
- $T_P\subseteq L_P\times\Sigma\times L_P$ 表示程序 P 的迁移关系，其中，$\Sigma=\bigcup\Sigma_i$ 为所有线程 P 中语句 $\sigma\in\Sigma_i$ 的集合. 对于任意迁移 $t\in T_P$，仅有一个线程 P_i 可从其前驱位置迁移到对应的后继位置，而其他线程 $P_j(i\neq j)$的对应位置保持不变.
- $l_{0P}\subseteq l_{0P_1}\times\ldots\times l_{0P_N}$ 为程序 P 的初始位置集合，其中，l_{0P_i} 表示线程 P_i 的初始位置.
- $f_P\subseteq f_{P_1}\times\ldots\times f_{P_N}$ 为程序 P 执行结束的位置集合，其中，f_{P_i} 表示线程 P_i 执行结束位置集合.

以图 1(a)所示的并发程序 example.c 为例. 该程序初始化共享变量 x 和 y 并创建两个子线程 P_1 和 P_2，当所有子线程均执行完毕后，检查共享变量 y 是否小于 2，以判断该并发程序是否到达错误位置. 若 $y\geqslant 2$，则表明程序中存在可使共享变量 y 最终赋值大于等于 2 的路径，此时，我们称该并发程序是不安全的.



(a) (b) (c)

图 1 示例并发程序及其控制流图

该示例并发程序中，两个子线程 P_1 和 P_2 对应的线程控制流图 G_{P_1} 和 G_{P_2} 如图 1(b)所示，其中，每个圆圈代表线程的位置，位置之间的单向箭头表示线程迁移. 这些线程控制流图通过并行合并可进一步得到图 1(c)所示的并发程序控制流图 G_P，其中，每个程序位置 L_i 唯一地表示一种线程位置组合(例如位置 L_2 表示线程 P_1 位于 l_2 处且线程 P_2 位于 l'_1 处). 当程序位置 L_{13} 可达时，意味着该并发程序不安全. 在并发程序实际的模型检测过程中，由于程序位置的排列组合空间往往较大，且并非所有位置均可真实到达，因此，控制流图 G_P 并不需要显示地构造，仅需构造每个线程的控制流图 G_{P_i} .

基于特定的分析(如基于谓词抽象的可达性分析[10,19]以及基于 BDD 的程序分析[20])展开(unwind)并发程序

控制流图 G_P, 可以得到一棵带标签的抽象可达树(abstract reachability tree, ART). 本文中采用基于 BDD 的程序分析, 并以 on-the-fly 的方式[21]探索并发程序的可达状态空间.

定义 2(抽象可达树). 程序 P 的抽象可达树 $A=(S,E)$为状态集 S 和边集 E 构成的二元组, 用以表示程序可达状态空间的一部分, 其中,

- $s=(l^s,c^s,b^s)\in S\subseteq L_P\times\mathcal{C}\times\mathcal{B}$为程序状态, 其中, L_P 为程序位置; $\mathcal{C}$为当前状态处的函数调用栈信息; $\mathcal{B}$为由 BDD 编码的一阶公式$\varphi\in\Phi$, 用以表示满足公式φ的可达域(reachable region). 给定状态$s,s'\in S$, 若$l^s=l^{s'}$, $c^s=c^{s'}$, $b^s\to b^{s'}$且 s'已被探索, 则称 s 被 s'覆盖(cover). 此时, 状态 s 的所有后继状态不必继续探索.
- $e=(s,t,s')\in E\subseteq S\times T_P\times S$ 为程序状态迁移. 后继状态 s'可达当且仅当存在线程迁移 $t=(l^s,\sigma,l^{s'})\in T_P$, 使得 $b^s\wedge\mathcal{B}_t\nrightarrow$false($\mathcal{B}_t$ 为线程迁移 t 中语句σ对应一阶公式的 BDD 编码), 我们用 $s\xrightarrow{t}s'$ 表示该状态迁移, 此时, 可达后继状态 s'的可达域 $b^{s'}$更新规则如下[20]:

$$b^{s'}=\begin{cases}b^s\wedge\mathcal{B}_{cond}, & \text{if } \sigma:cond\\ (\exists v:b^s)\wedge\mathcal{B}_{v=exp}, & \text{if } \sigma:v:=exp\end{cases},$$

其中, σ.$cond$ 和σ.v:=exp 分别为条件判断语句和赋值语句, $\mathcal{B}_{cond}$ 和$\mathcal{B}_{v=exp}$ 分别为语句σ.$cond$ 和σ.v:=exp 对应的一阶公式的 BDD 编码; 当σ为赋值语句时, $\exists v:b^s$ 将变量 v 从 b^s 中移除, $\mathcal{B}_{v=exp}$ 重新对变量 v 的可达域进行编码. 若迁移 t 在直接前驱状态 s 处的执行条件是满足的, 则称 t 在状态 s 处是可执行的(enabled).

图 2 为示例并发程序 example.c 的控制流图通过 BDD 程序分析展开后得到的 ART, 其中, 每个圆圈和方框结点均代表程序的可达状态. 为使可视化的抽象可达树结构更简洁, 图 2 中并未画出方框结点的后继可达状态. 抽象可达树中的一条路径 $\pi:s_0\xrightarrow{t_0\ldots t_{n-1}}s_n$ 为从初始状态 s_0 出发的一个状态迁移序列. 若并发程序的所有路径中均不包含错误状态, 则称该程序是安全的.

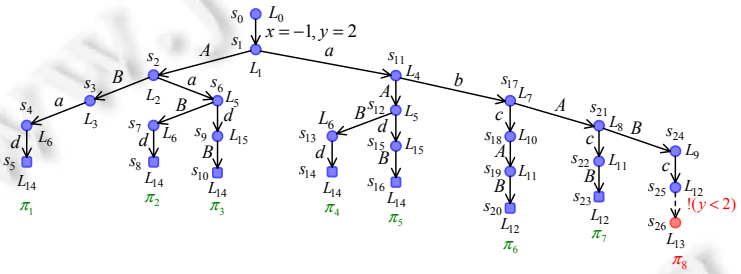


图 2 通过 BDD 程序分析展开控制流图构造得到的抽象可达树

1.2 偏序约简

由于并发程序中线程可交替执行, 上述展开并发程序控制流图以探索抽象可达树的方法会产生许多冗余路径, 造成严重的状态空间爆炸问题. 偏序约简技术通过分析探索路径是否等价, 以减少冗余路径的探索. 对于抽象可达树中的两条路径π_1 和π_2, 若π_1 可通过交换π_2 中相邻且独立的线程迁移得到, 则称路径π_1 与路径π_2 等价(记为$\pi_1\sim\pi_2$). 其中, 独立的定义如下所示.

定义 3(独立)[12]. 对于并发程序 P 中的两个线程迁移 $t_1,t_2\in T_P$, 若这两个迁移分属不同线程, 且对于任意状态 $s\in S$ 均满足下面两个性质时, 线程迁移 t_1 和 t_2 是相互独立的(记为 $t_1\|t_2$).

- 若 t_1 在状态 s 处是可执行的, 且存在状态迁移 $s\xrightarrow{t_1}s'$, 当且仅当线程迁移 t_2 在状态 s 处可执行时, 线程迁移 t_2 在状态 s'处也是可执行的(即两个相互独立的线程迁移不会影响对方的执行).
- 若线程迁移 t_1 和 t_2 在状态 s 处是可执行的, 则存在唯一的一个状态 $s''\in S$, 使得 $s\xrightarrow{t_1t_2}s''$ 且 $s\xrightarrow{t_2t_1}s''$ (即两个相互独立的线程迁移是可交换的).

否则, 线程迁移 t_1 和 t_2 是相互依赖的(记为 $t_1\nparallel t_2$).

由于上述独立性判断需在所有状态处检查两个性质是否成立, 该方法在实际的并发程序验证过程中难以

直接运用，因此，该判断依据主要用于独立性的语义检查[12]. 目前，大部分偏序约简技术常采用更为保守但易于检查的方法粗略估计线程迁移是否相互依赖. 例如，当迁移 $t_1,t_2\in T_P$ 满足下面任意一个条件时，两个迁移相互依赖.

- 线程迁移 t_1 和 t_2 属于相同的线程.
- 其中一个线程迁移写访问的共享变量被另一个线程迁移读或写访问.

由上述粗略依赖性估计条件易知：图 1(a)的示例程序 example.c 中，仅线程 P_1 的迁移 A:y=1 与线程 P_2 的迁移 a:x=x+1 因均不访问相同共享变量而独立，即 $A||a$. 基于该粗略依赖性分析展开并发程序控制流图构造的抽象可达树如图 3 所示. 在该树中，由于路径 π_4 和 π_5 分别可通过交换路径 π_2 和 π_3 中相邻的独立迁移 A:y=1 和 a:x=x+1 得到(即 π_2~π_4 且 π_3~π_5)，因此，路径 π_4 和 π_5 均视为冗余路径而不必探索.

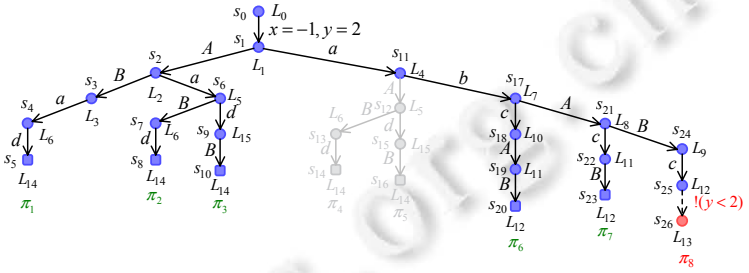


图 3　基于粗略依赖性分析展开并发程序控制流图构造的抽象可达树

进一步分析图 3 可知，虽然使用粗略独立性分析可剪裁部分冗余路径，但在该抽象可达树中，仍存在大量等价路径. 例如，状态 s_6 处的 BDD 表示为 $b^{s_6}:x=0\wedge y=1$，其后继线程迁移 B:x=x+1 和 d:!(x≤y−2)在该状态处满足独立性定义的两个性质，即线程迁移 B 和 d 在状态 s_6 处的执行不会相互影响且可交换，则有 π_2~π_3. 因此，精确的迁移依赖性分析是实现高效并发程序模型检测的关键，下一节将介绍可细化线程迁移依赖性分析的并发程序模型检测工具 CDG4CPV.

2　原型工具

本文在开源程序验证工具 CPAchecker[22]的基础上，设计并实现了可细化线程迁移依赖性分析的并发程序模型检测工具 CDG4CPV，该工具采用 Java 语言开发，其整体框架如图 4 所示.

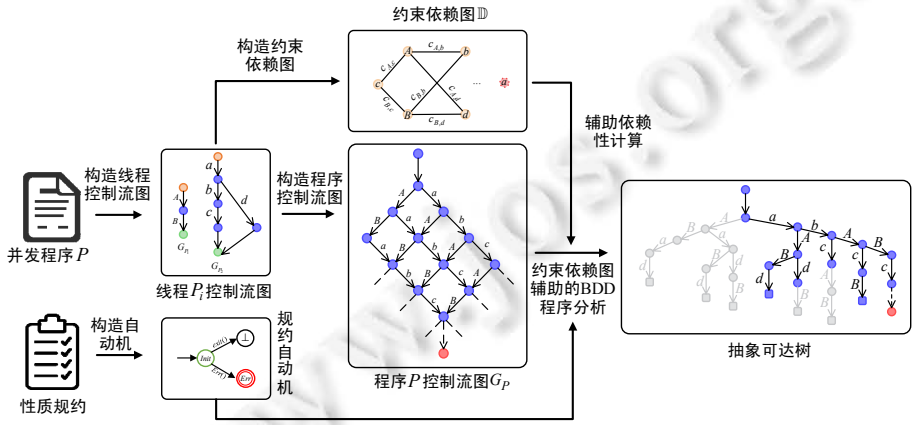


图 4　CDG4CPV 的框架

在该工具中，线程控制流图的构造可调用相关分析模块和方法实现. 下面我们主要对规约自动机的构造、约束依赖图的构造和约束依赖图辅助的 BDD 程序分析进行介绍.

2.1 规约自动机构造

CPAchecker 可对内存安全、可达性和数据溢出等多种性质进行验证, 其底层使用与 BLAST 查询语言[23]类似的规约自动机语言描述验证性质对应的规约自动机.

本文实现的工具主要对可达性性质进行验证, 此性质可使用 SV-COMP 2022 竞赛中目标函数不可达规约“*CHECK*(*init*(*main*(·)),*LTL*(*G*!*call*(*Err*(·))))”描述(https://sv-comp.sosy-lab.org/2022/rules.php).

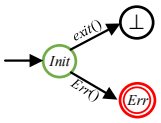
其中: 定义 *init*(*main*(·))通过调用 main 函数给出程序的初始状态; 线性时序逻辑定义 *LTL*(*G*!*call*(*Err*(·)))表示目标函数 *Err* 在程序的任何有限执行路径上均不会被调用, 即目标函数不可达. 对于此规约, CPAchecker 提供了图 5(a)所示的规约自动机语言实现, 通过在待验证并发程序的特定位置处插入目标函数 *Err*(·), 即可实现对目标函数 *Err*(·)是否被调用的可达性检测. 该自动机包含 *Init*, *ERROR* 和 *STOP*(即“⊥”)这 3 个状态.

- *Init* 为规约自动机初始状态.
- *ERROR* 为错误状态. 迁移到错误状态即表明可达树中存在违反性质的路径.
- *STOP* 为终止状态. 意味着不必计算从该状态开始的后继迁移.

若当前线程迁移 $t=(l,\sigma,l')\in T_P$ 中, 程序语句σ包含的函数调用与函数 *Err*(·)相匹配, 则规约自动机从 *Init* 状态迁移到 *ERROR* 状态; 若σ的函数调用与函数 *exit*(·)匹配, 则规约自动机从 *Init* 状态迁移到 *STOP* 状态; 否则, 该自动机状态维持在 *Init* 状态. 本文所提工具将图 5(a)中的规约自动机作为文件输入, 自动地构造成可视化表示为如图 5(b)所示等价的规约自动机, 该自动机将参与随后的并发程序验证过程中, 目标位置的可达性检测过程.

```
CONTROL AUTOMATON SVCOMP
INITIAL STATE Init;
STATE USEFIRST Init:
  MATCH {Err($?)}→ERROR("error location is reached!");
  MATCH {exit($?)}→STOP;
END AUTOMATON
```

(a)



(b)

图 5 可达性性质规约自动机及其图示

通过在示例程序 example.c 中断言为假的分支后方插入 *Err*(·)函数, 即可将对共享变量 y 的最终赋值是否满足 $y\geqslant 2$ 的检测转换成对目标函数的可达性检测.

2.2 约束依赖图构造

为了减少在展开并发程序控制流图时对等价路径的冗余探索, 需构造约束依赖图以实现对独立性分析的细化.

定义 4(约束依赖图). 给定并发程序 P, 其约束依赖图 $\mathbb{D}=(\mathbb{T},\mathbb{C})$是一个由结点集合 $\mathbb{T}\subseteq T_P$和结点约束依赖关系 $\mathbb{C}\subseteq\mathbb{T}\times\Phi\times\mathbb{T}$构成的无向图. 其中, Φ为可满足的(satisfiable)一阶公式的集合, 用于表示线程迁移 $t,t'\in\mathbb{T}$相互依赖时应满足的约束 $\varphi\in\Phi$. 为了表示方便, 我们使用 $c_{t,t'}=(t,\varphi,t')\in\mathbb{C}$表示该约束公式$\varphi$.

约束依赖图构造的核心在于约束依赖关系的建立, 该关系可通过遍历并发程序控制流图 G_P 中不同线程的线程迁移结点对 $(t,t')\in\mathbb{T}\times\mathbb{T}$ $(t\in T_{P_i},t'\in T_{P_j},i\neq j)$ 计算得到. 图 6 为结点对依赖约束的计算算法. 该算法首先获取线程迁移 t 和 t'中均被访问且其中一个线程迁移写访问的共享变量集合 G, 若不存在该类共享变量, 则表明 $t\|t'$; 否则, t 和 t'的依赖性需进一步分析. 当线程迁移 t 或 t'中存在取值不确定的变量时, 二者间的依赖约束难以有效生成. 因此, 为了保证后续程序分析的完备性, 我们保守地认为 t 和 t'相互依赖. 在其他情况中, 根据线程迁移 t 和 t'的类型, 分别调用不同的算法生成依赖约束. 本文主要关注包含条件判断语句的线程迁移(用集合$\mathcal{A}_{su}\subseteq T_P$表示)和使用线性整型算数(linear integer arithmetic)表达式的赋值语句对应的线程迁移(用集合$\mathcal{A}_{sg}\subseteq T_P$表示)这两种类型.

ComputeDependencyConstraint(t,t').
输入: 约束依赖图结点对$(t,t')\in \mathbb{T}\times\mathbb{T}$.
输出: t 和 t'相互依赖时需满足的约束φ.

```
1:  G=(Read(t)∩Write(t′))∪(Write(t)∩Read(t′))∪(Write(t)∩Write(t′));
2:  if G≠∅ then   //t 和 t′访问共同的共享变量且存在潜在的依赖关系
3:    if ¬NondetVars(t) and ¬NondetVars(t′) then
4:      if t,t′∈A_sg then
5:        return ComputeAsgPairConstraint(t,t′);
6:      elseif t∈A_su and t′∈A_sg then
7:        return ComputeAsuAsgConstraint(t,t′);
8:      else t∈A_sg and t′∈A_su then
9:        return ComputeAsuAsgConstraint(t′,t);
10:     endif
11:   Else
12:     return ⊤;   //t 或 t′中存在赋随机值的变量, 保守认为 t 和 t′相互依赖
13:   endif
14: else
15:   return ⊥;
16: endif
```

图 6　结点对的依赖约束计算算法

- 当 $t,t'\in\mathcal{A}_{sg}$ 时

在该情况下, 仅需破坏定义 3 中独立线程迁移的可交换性, 即可保证线程迁移 t,t'相互依赖. 图 7 为该情况下依赖约束的计算算法, 其核心思想是, 计算线程迁移 t:v_1:=exp_1 和 t':v_2:=exp_2 在不同执行顺序下(即 $t{\cdot}t'$和 $t'{\cdot}t$)对应变量最终赋值不相同时的条件. 当 v_1 和 v_2 为同一共享变量时, t 和 t'在布尔表达式"exp_1=exp_2"为永真式时相互独立; 否则, 二者的依赖约束为φ.$exp_1[exp_2/v_2]\neq exp_2[exp_1/v_1]$. 当 v_1 和 v_2 为不同变量时, 在执行顺序 $t{\cdot}t'$下, 变量 v_1 的最终赋值表达式为 exp_1, 而变量 v_2 的最终赋值为 $v_2'\coloneqq exp_2[exp_1/v_1]$(其中, $exp_2[exp_1/v_1]$表示将表达式 exp_2 中的变量 v_1 替换成 exp_1); 在执行顺序 $t'{\cdot}t$ 下, 变量 v_1 的最终赋值为 $v_1'\coloneqq exp_1[exp_2/v_2]$, 而变量 v_2 的最终赋值表达式为 exp_2. 通过使 v_1 和 v_2 中任意一个对应变量最终赋值不相等, 即可得到线程迁移 t 和 t'相互依赖时的约束.

ComputeAsgPairConstraint(t,t').
输入: 约束依赖图结点对$(t,t')\in \mathbb{T}\times\mathbb{T}$, $t,t'\in\mathcal{A}_{sg}$.
输出: t 和 t'相互依赖时需满足的约束φ.

```
1:  t:v1:=exp1, t′:v2:=exp2;
2:  if v1 和 v2 为同一共享变量 then
3:    if CanSimplifyToTrue(exp1=exp2) then
4:      return ⊥;
5:    else
6:      exp1[exp2/v2]≠exp2[exp1/v1]
7:  else
8:    v2′ := exp2[exp1/v1];   //t·t′顺序
9:    v1′ := exp1[exp2/v2];   //t′·t 顺序
10:   φ1:=⊥, φ2:=⊥;
11:   if ¬CanSimplifyToTrue(exp1 = v1′) then
12:     φ1:=exp1≠exp1[exp2/v2];
13:   endif
14:   if ¬CanSimplifyToTrue(exp2 = v2′) then
15:     φ2:=exp2≠exp2[exp1/v1];
16:   endif
17:   return φ1∨φ2;
18: endif
```

图 7　赋值结点对的依赖约束计算算法

- 当 $t\in\mathcal{A}_{su}$, $t'\in\mathcal{A}_{sg}$ 时

在该情况下, 仅需破坏定义 3 中独立线程迁移不会影响对方执行的性质, 即可保证线程迁移 t,t'相互依赖. 图 8 为该情况下依赖约束的计算算法, 其核心思想是, 计算线程迁移 t':v:=exp 影响迁移 t:$cond$ 中条件判断语句 $cond$ 真值的条件, 该条件可通过将 $cond$ 中的共享变量 v 替换为表达式 exp 并取非得到.

ComputeAsuAsgConstraint(t,t'):
输入: 约束依赖图结点对(t,t')$\in\mathbb{T}\times\mathbb{T}$, $t\in\mathcal{A}_{su}$,$t'\in\mathcal{A}_{sg}$.
输出: t 和 t'相互依赖时需满足的约束φ.

```
1:  t:cond, t′:v:=exp;
2:  if CanSimplifyToTrue(cond[exp/v]) then
3:      return ⊥;
4:  else
5:      return ¬cond[exp/v];
6:  endif
```

图 8　条件判断和赋值结点对的依赖约束计算算法

- 当 $t\in\mathcal{A}_{sg}$, $t'\in\mathcal{A}_{su}$ 时, 同上

当 t,$t'\in\mathcal{A}_{su}$ 时, 由于二者均读访问程序变量, 此时有 $t\|t'$. 此外, 当计算出的约束为⊥时, 由于该约束是不可满足的, 因此, 约束依赖图$\mathbb{D}$中的 t 和 t'之间不存在无向边. 由于约束依赖图是无向图, 任意不同线程的线程迁移对仅需计算一次依赖约束, 因此, 对于创建 n 个线程、每个线程含有 m 个线程迁移的并发程序, 构建其相应约束依赖图的时间复杂度为 $O(m^2n^2)$.

图 9 为示例程序 example.c 的约束依赖图, 其中, 每个圆圈代表一个线程迁移结点, 结点连接边上的公式对应这两个结点相互依赖时应满足的约束条件. 值得注意的是, 线程 P_2 的迁移 a:x=x+1 虽然与线程 P_1 的迁移 B:x=x+1 写访问相同, 共享变量 x, 但约束依赖图中, 线程迁移 a:x=x+1 不与任何其他线程的迁移相连接. 这是由于这两个线程迁移在任意状态处可执行时, 均会向共享变量 x 写入相同的值. 即不会破坏定义 3 中的两个性质.

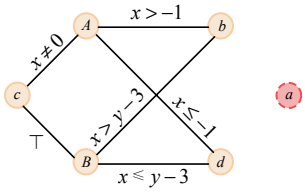


图 9　示例程序约束依赖图

定理 1(约束依赖). 在抽象可达树 A=(S,E)的状态 $s\in S$ 处, 两个均不含有取值不确定变量的线程迁移 t,$t'\in\mathbb{T}$ 在该状态处存在约束依赖(记为 $t\not\|_s t'$), 当且仅当 $c_{t,t'}\in\mathbb{C}$且 $s\models c_{t,t'}$; 否则, 当 $c_{t,t'}\notin\mathbb{C}$或 $s\not\models c_{t,t'}$时, t 和 t'在状态 s 处独立(记为 $t\|_s t'$).

证明: 设两个均不含有取值不确定变量的线程迁移 t,$t'\in\mathbb{T}$在状态 $s\in S$ 处有 $t\not\|_s t'$, 即 t 和 t'在状态 s 处依赖; 同时, 根据约束 $c_{t,t'}$的构造方式可知: 这两个线程迁移中, 至少有一个写访问相同的全局变量. 因此, 约束依赖图$\mathbb{D}$中存在 t 和 t'的无向边(即 $c_{t,t'}\in\mathbb{C}$). 假设 $s\not\models c_{t,t'}$, 则针对 t,$t'\in\mathcal{A}_{sg}$和 $t\in\mathcal{A}_{su}$, $t'\in\mathcal{A}_{sg}$ 这两种情况的讨论如下.

- 当 t,$t'\in\mathcal{A}_{sg}$ 时, 由 $s\not\models c_{t,t'}$易知: 在状态 s 处, 执行顺序 $t\cdot t'$和 $t'\cdot t$ 使得图 7 中变量 $v_1=v_1'$且 v_2=v'成立, 即这两种执行顺序会到达相同状态. 由于 t 和 t'在状态 s 处不会影响对方的执行, 则有 t 和 t'在状态 s 处独立. 该结论与最初假设 t 和 t'在状态 s 处依赖相违背, 因此有 $s\models c_{t,t'}$.
- 当 $t\in\mathcal{A}_{su}$, $t'\in\mathcal{A}_{sg}$ 时, 由 $s\not\models c_{t,t'}$易知, 图 8 中 $cond[exp/v]$的真值为真, 使得状态 s 处存在执行顺序 $t\cdot t'$, 即在状态 s 处, 线程迁移 t 的执行不会影响线程迁移 t'的执行. 由于仅有线程迁移 t'写访问变量, 执行顺

序 $t{\cdot}t'$和 $t'{\cdot}t$ 均会到达相同状态, 则有 t 和 t'在状态 s 处独立. 该结论与最初假设 t 和 t'在状态 s 处依赖相违背, 因此有 $s\models c_{t,t'}$.

上述两种情况均与假设 $s\not\models c_{t,t'}$相违背, 因此该假设不成立, 我们有 $s\models c_{t,t'}$. 使用类似方法可证明: 两个均不含有取值不确定变量的线程迁移 $t,t'\in\mathbb{T}$在状态 $s\in S$ 处有 $t\|_s t'$, 则 $c_{t,t'}\notin\mathbb{C}$或 $s\not\models c_{t,t'}$成立.

需注意的是, 当线程迁移 t 或 t'中含有取值不确定变量时(即通过随机数生成函数赋值的变量), 为保证验证结果的正确性, 所提方法对二者的依赖性作保守估计($c_{t,t'}:\top$), 即仍然保守地判定 t 和 t'在给定状态 s 处依赖.

通过查询所构造的约束依赖图$\mathbb{D}$, 抽象可达树中每个状态 $s\in S$ 处的直接可执行后继线程迁移 $t\in T_{P_i}$, $t'\in T_{P_j}(i\neq j)$之间是否存在约束依赖, 可通过判断状态 s 是否建模相应约束 $c_{t,t'}$来动态且精确地计算, 其计算结果可辅助后续 BDD 程序分析过程中冗余等价路径的剪裁. 需注意的是, 在判断状态 s 是否建模约束 $c_{t,t'}$时, 公式 $c_{t,t'}$会被编码成与状态 s 中可达域 b^s 相同类型的符号表示形式(即 $c_{t,t'}$的 BDD 编码 $\mathcal{B}_{c_{t,t'}}$).

定义 5(孤立迁移). 在约束依赖图$\mathbb{D}$=($\mathbb{T}$,$\mathbb{C}$)中, 若不存在与线程迁移结点 $t\in T_{P_i}$ 相连的其他线程对应线程迁移结点 $t'\in T_{P_j}(i\neq j)$, 即 $c_{t,t'}\notin\mathbb{C}$, 则称线程迁移 t 为孤立迁移, 并用符号$\mathbb{I}\subseteq T_P$表示该类线程迁移.

我们注意到, 约束依赖图中可能存在不与其他线程迁移结点相互连接的孤立结点(例如图 9 中的线程迁移结点 a:x=x+1), 我们称这类线程迁移为孤立迁移. 由约束依赖图的构造特点可知, 孤立迁移 $t\in\mathbb{I}\cap T_{P_i}$ 与其他线程的线程迁移 $t'\in T_{P_j}(i\neq j)$ 在任意状态 s 处均相独立(即 $t\|_s t'$总成立), 线程迁移 t 和 t'的不同执行顺序不会产生属于不同等价类的探索路径, 因此可以得到以下定理:

定理 2(孤立等价). 对于并发程序 P, 我们使用 $t'_j\in T_{P_j}$ 表示线程 P_j 第 i 次执行的线程迁移. 对于该程序对应抽象可达树中的两个状态 $s_m,s_n\in S$, 若线程迁移 $t^0_q\in\mathbb{I}$ 在状态 s_m 的可执行后继线程迁移集合 $\{t^0_1,...,t^0_k,t^0_q\}$ 中, 且 $t^0_1\in T_{P_1},...,t^0_k\in T_{P_k}$, 则对于状态 s_m 处沿着线程迁移 $t'\in\{t^0_1,...,t^0_k\}$ 出发的任意子路径 $\pi_1: s_m\xrightarrow{t'...t^0_q...}s_n$, 均存在状态 s_m 处沿着孤立迁移 t^0_q 出发且与π_1 等价的子路径 $\pi_2: s_m\xrightarrow{t^0_q t'...}s_n$(即$\pi_1\sim\pi_2$). 其中, 箭头上的点表示一系列线程迁移.

该定理的证明已在我们前期工作中给出, 此定理表明: 在抽象可达树中的任意状态 s 处, 若存在可执行的孤立迁移后继 $t\in\mathbb{I}\cap T_{P_i}$, 则可以优先探索线程迁移 t 而剪裁状态 s 处其他线程的可执行线程迁移 $t'\in T_{P_j}(i\neq j)$, 即延迟了线程迁移 t'的探索. 这种孤立迁移优先化探索策略仍然可以保证路径等价类探索的完备性, 即不会遗漏对任何路径等价类的探索. 在探索并发程序抽象可达树的过程中, 可以在每个状态处, 利用约束依赖图查询后继可执行分支中是否含有孤立迁移, 以此避免对冗余分支的探索. 例如, 在后文图 11 中的状态 s_1 处, 本文所提工具将优先探索线程 P_2 的孤立迁移 a:x=x+1, 并剪裁线程 P 的可执行迁移分支 A:y=1.

2.3 约束依赖图辅助的BDD程序分析

在探索并发程序 P 的抽象可达树时, 工具 CDG4CPV 维护可达状态集合 $R\subseteq S$ 与待处理状态集合 $W\subseteq S$, 并在分析开始之前, 将初始状态 s_0=(L_0,−,true)放入集合 W 中, 以计算其后续可达状态. 自初始状态 s_0 开始, 基于约束依赖图辅助的 BDD 程序分析以深度优先搜索(depth-first search)的方式探索可达的状态, 在探索的过程中, 利用约束依赖图更新每个后继状态的休眠集合, 以避免对冗余等价路径的探索. 该过程循环进行, 直至没有新的状态产生或到达包含错误位置的状态为止. 其中, 休眠集合用于保存每个状态处不需要探索的直接后继线程迁移.

定义 6(休眠集合). 若 $t\in T_{P_i}$ 为状态 s 处可执行的线程迁移, 且有 $s\xrightarrow{t}s'$, 则对于状态 s 处其他可执行线程迁移 $t'\in T_{P_j}$, 后继状态 s'的休眠集合 $sleep(s')\subseteq T_P\setminus T_{P_i}$ 为

$$sleep(s')=\begin{cases}\{t'\mid t'\in T_{P_j}\wedge j<i\wedge t'\|_s t\}, & t\in T_{P_i}\setminus\mathbb{I}\\ sleep(s), & t\in\mathbb{I}\end{cases}.$$

另外, $sleep(s_0)=\varnothing$.

图 10 为整个程序分析的过程, 主要包括错误位置检测与可达状态集计算、孤立线程迁移优先探索、休眠集合更新和依赖线程迁移后继探索这 4 个步骤.

- 第 1 步, 错误位置检测与可达状态集计算. 工具从集合 W 中取出一个还未探索后继的状态 s. 当该状态到达错误位置时, 工具停止分析, 并通过回溯抽象可达树构建不满足可达性性质的反例路径; 否则, 工具基于 BDD 的程序分析计算出状态 s 的可达状态集合:

$$suc=\{s'|\exists t=(l,\sigma,l')\in T_P.l=l^s\wedge l'=l^{s'},b^s\wedge\mathcal{B}_t\nrightarrow\text{false})\in T_P\}\subseteq S.$$

- 第 2 步, 孤立线程迁移优先探索. 此时, 在线程迁移集合 $T_s=\{t\mid s'\in suc,s\xrightarrow{t}s'\}$ 中, 可能存在孤立线程迁移 $t\in\mathbb{I}\cap T_{P_i}$. 由约束依赖图的定义可知, 孤立线程迁移结点 t 与其他线程的任意线程迁移结点 $t'\in T_{P_j}(i\neq j)$ 互不相连(即 $t\|t'$), 线程迁移 t 可以在状态 s 处与任意其他线程迁移 $t'\in T_s\backslash\{t\}$在交换执行顺序后, 仍会到达相同的后继状态 $s''\in S$. 因此, 在状态 s 处, 工具首先利用构造的约束依赖图$\mathbb{D}$查询 T_s 中是否存在孤立线程迁移: 若存在, 则任选其中一个孤立线程迁移 $t\in\mathbb{I}$(此时有 $s\xrightarrow{t}s'$, $s'\in suc$); 随后, 将状态 s'放入可达状态集合 R 和待处理状态集合 W 中, 并更新后继状态 s'的休眠集合为其直接前驱状态的休眠集合; 最后, 返回第 1 步继续探索其他待探索状态. 在该步骤中, 返回到第 1 步的操作使工具无须探索除孤立线程迁移 t 之外的其他可执行线程迁移, 极大地减少了对冗余路径等价类的探索.
- 第 3 步, 休眠集合更新. 由于状态 s 处可能存在可执行的独立线程迁移对 $(t,t')\in T_{P_i}\times T_{P_j}(i\neq j)$ (即 $t\|_s t'$), 为避免探索 t 和 t'的两个不同执行顺序 $t\cdot t'$和 $t'\cdot t$, 需在其中一种执行顺序中标记不需要探索的线程迁移. 为此, 可查询约束依赖图$\mathbb{D}$中是否存在线程迁移对(t,t')的依赖约束 $c_{t,t'}$: 若存在, 则可通过定义 6 计算这些依赖约束是否在状态 s 处不成立(即判断线程迁移 t 和 t'在状态 s 处是否独立), 以此获得状态 s 处的所有可达后继状态 $s'\in suc$ 的休眠集合. 在该定义中, $j<i$ 保证了当线程迁移 $t\in T_{P_i}$ 和 $t'\in T_{P_j}$ 在状态 s 处独立时(即约束依赖图$\mathbb{D}$中不存在约束 $c_{t,t'}$或状态 s 不能够建模约束 $c_{t,t'}$), 工具将线程迁移 t'放入后继状态 s'的休眠集合中, 从而仅会探索执行顺序 $t'\cdot t$.
- 第 4 步, 依赖线程迁移后继探索. 若线程迁移 t 不在状态 s 的休眠集合中, 则对于每个未被覆盖的后继状态 $s'\in suc(s\xrightarrow{t}s')$, 将其放入可达状态集合 R 和待处理状态集合 W 中, 并返回第 1 步继续探索未处理的状态.

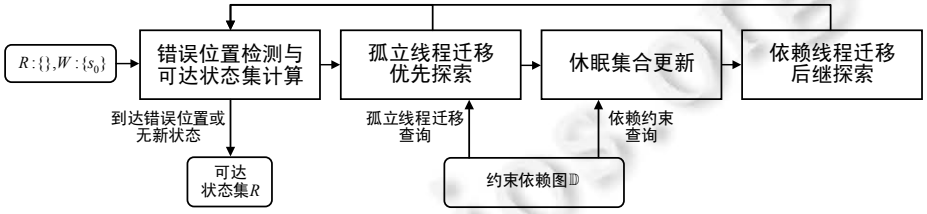


图 10 约束依赖图辅助的 BDD 程序分析过程

对于第 2 步和第 4 步中探索的状态迁移 $s\xrightarrow{t}s'$, 若 $t=(l^s,\sigma,l^{s'})$中程序语句σ的函数调用与函数 $Err(\cdot)$相匹配, 则规约自动机从初始状态 *Init* 迁移到 *ERROR* 状态, 此时, 工具标记后继状态 s'为错误状态(此时的 $l^{s'}$为错误位置); 否则, 规约自动机仍维持在初始状态 *Init* 处. 值得注意的是, 本文所开发的工具仅在无新状态产生或找到错误状态时停止: 当工具未能发现任何错误状态时, 所构建的抽象可达树即为可达性性质的证明; 而当发现错误状态时, 工具可根据抽象可达树构造一条反例路径.

定理 3(正确性). 所有在基于约束依赖图辅助的 BDD 程序分析过程中剪裁的路径均是冗余的.

定理 4(最优性). 对于包含两个线程的并发程序, 基于约束依赖图辅助的 BDD 程序分析过程可以保证剪

裁所有冗余路径.

上述两个定理的证明均在我们前期工作中给出. 定理 3 表明: 所提工具不会缺少对任何路径等价类的探索, 确保了路径等价类探索的完备性. 定理 4 表明: 所提工具对于只含有两个线程的并发程序来说, 路径探索是最优的, 即仅探索每个路径等价类中的一条路径.

3 工具展示

3.1 实例展示

本节通过示例并发程序 example.c 的验证过程展示工具 CDG4CPV 的工作效果.

图 11 为使用约束依赖图展开该程序控制流图得到的抽象可达树, 其中, 每个状态右上角的红色花括号为休眠集合. 当从初始状态 s_0 出发到达状态 s_1 时, 由于该状态不为错误状态且存在可执行的孤立线程迁移 a:x=x+1, 因此工具仅优先探索其直接后继 s_{11}. 在状态 s_{11} 处, 由于线程迁移 A:y=1 和 b:x≤y−2 均不为孤立迁移且 $s_{11} \vDash c_{A,b}(b^{s_{11}}: x=0 \wedge y=2, c_{A,b}: x>-1)$, 这两个线程迁移在该状态处存在约束依赖关系, 因此工具在状态 s_{11} 处需要探索 $A{\cdot}b$ 和 $b{\cdot}A$ 这两种不同的执行顺序. 随后, 当探索到状态 s_{12} 处时, 由于这两个线程迁移在该状态处不存在约束依赖(即 $s_{12} \not\vDash c_{B,d}$, 其中, $b^{s_{12}}: x=0 \wedge y=1$, $c_{B,d}: x \leqslant y-3$), 因此, 工具将线程 P_1 的线程迁移 B:x=x+1 放入后继可达状态 s_{15} 的休眠集合中, 此时仅需要在状态 s_{12} 处完整探索执行顺序 $B{\cdot}d$. 类似地, 当工具探索到状态 s_{17} 处时有 $c \|_{s_{17}} A$, 工具将线程 P_1 的线程迁移 A:y=1 放入其可达后继状态 s_{18} 的休眠集合中, 后续将不会在状态 s_{17} 处探索执行顺序 $c{\cdot}A$. 经过多轮探索, 工具最终到达错误状态 s_{26}, 表明该示例并发程序中存在可以使得共享变量 y 最终赋值 y≥2 的路径. 通过回溯抽象可达树, 可获得一条反例路径:

$$\pi: s_0 \xrightarrow{x=-1,y=2} s_1 \xrightarrow{a} s_{11} \xrightarrow{b} s_{17} \xrightarrow{A} s_{21} \xrightarrow{B} s_{24} \xrightarrow{c} s_{25} \xrightarrow{!(y<2)} s_{26}.$$

在使用更为细化的依赖性分析后, 对于该示例程序有 π_1~π_2~π_3~π_4~π_5 和 π_6~π_7.

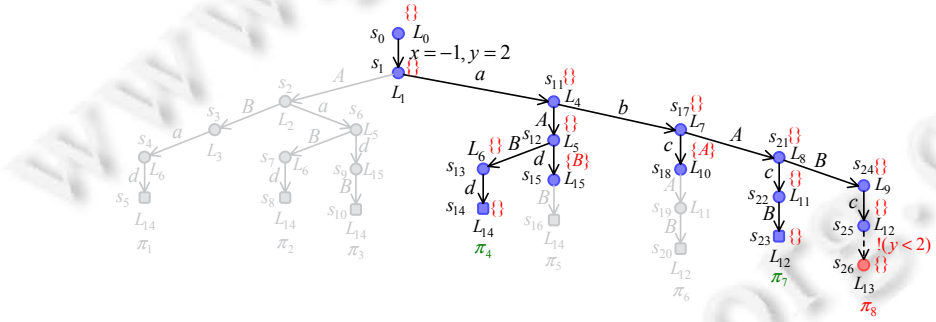


图 11 使用约束依赖图展开并发程序控制流图得到的抽象可达树

3.2 实验与分析

为了展示所开发工具 CDG4CPV 在并发程序上的实际验证效率以及证明工具的可用性, 我们在 SV-COMP 2022 竞赛的并发程序验证数据集(https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks.git, branch:svcomp22)上, 与其他工具和偏序约简算法进行了对比实验. 此外, 为了保证对比实验的公平性, 我们从数据集中 12 个测试子包里选取了 529 个可使后续对比工具和偏序约简方法均得出有效验证结果(即 TRUE(错误位置不可达)、FALSE (错误位置可达)或 UNKNOWN(验证超时或内存资源超出规定上限))的可达性验证任务. 实验所用数据集的详细信息如表 1 的左半部分所示. 所有实验均在一台 2.6 GHz CPU、内存总容量 735 GB 并装有 Ubuntu 18.04 LTS 系统的服务器上完成, 我们限制每个任务的验证时间上限为 15 min, 内存使用上限为 15 GB. 同时, 我们使用标准的实验及对比工具 BenchExec (https://github.com/sosy-lab/benchexec)进行精准且可靠的实验. 整体实验分为两个部分: (1) 与 SV-COMP 2022 竞赛中的工具进行对比; (2) 与其他偏序约简方法进行对比.

1) 与不同工具对比

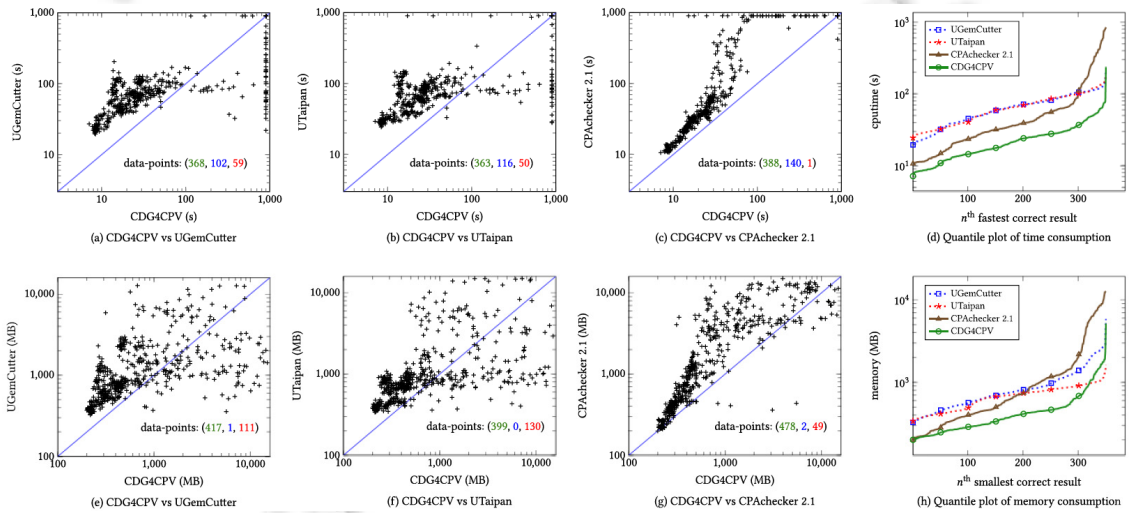
由于本文实现的工具可证明给定并发程序满足可达性性质, 然而 SV-COMP 2022 竞赛中的大部分工具主要用于快速发现程序中的 bug, 因此, 我们选择了该竞赛中不使用限界模型检测技术(bounded model checking, BMC)[24]的 3 个最高得分工具, 即 UGemCutter[25]、UTaipan[26]和 CPAchecker 2.1[22]进行对比实验. 其中, 工具 UGemCutter 将传统的 CEGAR 技术与路径的正交泛化(orthogonal generalization)技术相结合, 以减少冗余证明问题; UTaipan 是一个基于自动机的路径抽象验证工具, 可动态地构造程序的抽象; 工具 CPAchecker 2.1 主要采用朴素的 BDD 程序分析方法验证并发程序, 同时, 使用局部变量访问锁以减少冗余路径的探索. 这些工具均可在 SV-COMP 竞赛的网站(https://sv-comp.sosy-lab.org/2022)下载.

表 1 的右半部分为对比实验中 4 个工具在 SV-COMP 2022 部分并发程序数据集上的实验结果, 其中, #Suc.列为每个工具得出验证结果与任务标签一致的程序数目, #Unk.列为验证超时或内存使用达到上限的程序数目. 从该表可以看出, 工具 UGemCutter 和 UTaipan 可验证出的并发程序数目均多于 CDG4CPV, 而工具 CPAchecker 2.1 可验证出的并发程序数目比 CDG4CPV 少 34 个.

表 1 测试数据集和不同工具实验结果

No.	测试子包名称	文件数量	总行数	UGemCutter		UTaipan		CPAchecker 2.1		CDG4CPV	
				#Suc.	#Unk.	#Suc.	#Unk.	#Suc.	#Unk.	#Suc.	#Unk.
1	goblint-regression	46	33 042	45	1	46	0	44	2	45	1
2	ldv-races	8	14 093	8	0	8	0	8	0	8	0
3	pthread	21	15 299	15	6	9	12	7	14	9	12
4	pthread-atomic	11	8 748	10	1	10	1	6	5	8	3
5	pthread-C-DAC	2	2 717	1	1	1	1	0	2	2	0
6	pthread-complex	3	5 868	0	3	0	3	0	3	0	3
7	pthread-deagle	2	1 532	0	2	1	1	0	2	0	2
8	pthread-driver-races	2	13 730	0	2	0	2	1	1	2	0
9	pthread-ext	5	3 660	3	2	3	2	3	2	3	2
10	pthread-lit	1	751	1	0	1	0	1	0	1	0
11	pthread-wmm	283	243 734	283	0	283	0	260	23	281	2
12	weaver	145	17 748	54	91	42	103	23	122	28	117
总计		529	360 922	**420**	**109**	404	125	353	176	387	142

为了更清晰地对比本文实现的工具 CDG4CPV 和其他 3 个工具在验证时间和内存开销的差异, 我们绘制了如图 12 所示的散点图(包含验证结果为#Unk.的并发程序)和分位数图(为了保证比较的公平性, 分位数图中仅选择了 4 个工具均能得出正确验证结果的 350 个并发程序).



(a) CDG4CPV vs UGemCutter (b) CDG4CPV vs UTaipan (c) CDG4CPV vs CPAchecker 2.1 (d) Quantile plot of time consumption

(e) CDG4CPV vs UGemCutter (f) CDG4CPV vs UTaipan (g) CDG4CPV vs CPAchecker 2.1 (h) Quantile plot of memory consumption

图 12 与不同工具的实验对比结果

在该图中: 图 12(a) –(c)分别为工具 CDG4CPV 与工具 UGemCutter、UTaipan 和 CPAchecker 2.1 在 529 个

并发程序上的验证时间开销散点图; 图 12(d)为在这 4 个工具均能得出正确验证结果的 350 个并发程序上验证时间开销的分位数图; 图 12(e)–(g)分别为工具 CDG4CPV 与工具 UGemCutter、UTaipan 和 CPAchecker 2.1 在 529 个并发程序上的内存开销散点图; 图 12(h)为在这 4 个工具均能得出正确验证结果的 350 个并发程序上内存开销的分位数图. 在每个散点图中, 绿色、蓝色和红色这 3 个数据点(data-points)数据分别表示工具 CDG4CPV 比其他工具在验证时间或内存开销指标上少的、相等的和多的程序数目. 从该图可以看出:

- 工具 CDG4CPV 在大部分并发程序上的验证时间开销更低. 与工具 UGemCutter 和 UTaipan 相比, 本文实现的工具 CDG4CPV 分别在 69.57%和 68.62%的并发程序上的时间开销更低, 而仅分别在 59 个和 50 个并发程序上的验证时间开销更高; 此外, 工具 CDG4CPV 在 73.35%的并发程序上的时间开销低于 CPAchecker 2.1 (在这 388 个时间开销更低的结果中, 存在一个因工具 CDG4CPV 超出内存使用上限的数据点), 而仅在一个验证任务上的时间开销更高. 由分位数图 12(d)中每条曲线的下面积(area under curve)可知, 工具 CDG4CPV 在 350 个并发程序上的验证时间开销更低.
- 工具 CDG4CPV 在大部分并发程序上的内存开销更低. 与工具 UGemCutter、UTaipan 和 CPAchecker 2.1 相比, 本文实现的工具 CDG4CPV 分别在 78.83%, 75.43%和 90.36%的并发程序上内存开销更低, 而分别在 20.98%, 24.57%和 9.26%的并发程序上内存开销更高. 由分位数图 12(h)中每条曲线的下面积可知,工具 CDG4CPV 在 350 个并发程序上的内存开销更低.

此外, 我们发现, 在散点图 12(a)和(b)中存在一些 CDG4CPV 工具验证超时而 UGemCutter 和 UTaipan 工具未超时的并发程序. 出现该现象的主要原因是, 在这些并发程序(主要为 weaver 测试子包中的程序)中, 存在大量对共享变量赋随机值的语句. 在构建约束依赖图时, 我们保守地估计这些语句与其他访问相同共享变量的语句之间存在约束依赖, 以防止在对路径等价类探索时存在遗漏.

2) 与其他偏序约简方法对比

本文实现的工具建立在偏序约简方法基础上, 为比较不同偏序约简方法对并发程序验证效率的影响,我们将本文工具所使用的基于约束依赖图辅助的 BDD 程序分析方法与 PPOR[13]和 MPOR[14]这两种符号(symbolic)偏序约简方法进行了对比. 其中, PPOR 方法基于守卫独立迁移的概念编码约束, 可实现对并发程序冗余分支的剪裁; MPOR 为一种最优的偏序约简方法, 该方法通过维护线程迁移的依赖链并限制所探索路径满足准单调计算准则, 使得任意两条探索路径均不属于相同的路径等价类. 这两个符号偏序约简方法均已在 CPAchecker 上实现. 在本实验中, 我们将这两种方法与基于 BDD 的程序分析方法相结合(即 BDD+PPOR 和 BDD+MPOR), 并将其与工具 CDG4CPV 和基于 BDD 的程序分析(即 BDD)进行实验对比. 实验设置和所使用的并发程序数据集均与前一实验保持一致, 因此, 本实验消除了其他因素对不同偏序约简方法效率的影响.

在该实验中, BDD、BDD+PPOR、BDD+MPOR 和 CDG4CPV 分别验证出 347, 351, 323 和 387 个并发程序. 同样地, 图 13 为 4 种方法在验证时间和内存开销上差异的散点图和分位数图(为了保证比较的公平性, 分位数图中仅选择了 4 种方法均能得出正确验证结果的 305 个并发程序). 在该图中: 图 13(a) –(c)分别为 CDG4CPV 与 BDD、BDD+PPOR 和 BDD+MPOR 在 529 个并发程序上的验证时间开销散点图; 图 13(d)为在这 4 种方法均能得出正确验证结果的 305 个并发程序上验证时间开销的分位数图; 图 13(e) –(g)分别为 CDG4CPV 与 BDD、BDD+PPOR 和 BDD+MPOR 在 529 个并发程序上的内存开销散点图; 图 13(h)为在这 4 种方法均能得出正确验证结果的 305 个并发程序上内存开销的分位数图. 从该图中可以看出: 工具 CDG4CPV 所使用的基于约束依赖图辅助的 BDD 程序分析方法可以使大部分并发程序的验证时间和内存开销更低; 由分位数图 13(d)和(h)可知, CDG4CPV 在 305 个并发程序上的时间和内存开销更低.

为了量化不同偏序方法对并发程序模型检测所探索状态数目、验证时间和内存开销带来的影响, 使性能比较更为清晰, 我们进一步统计了 4 种方法在 305 个并发程序上的实验数据. 表 2 为这 4 种方法的具体实验结果统计, 其中, “#State”列为方法平均探索的状态数目, “#Time”列为平均的状态空间探索时间开销(这里不包括 CPAchecker 的其他准备和后处理等操作的时间开销, 单位: s), “#Mem.”列为平均内存开销(单位: MB), “#Avg.”行为工具在这 305 个并发程序上相应评估指标的平均值. 表中标记为“–”的行表示该测试子包中不含 4

种方法均能得出正确验证结果的并发程序, 黑色加粗的数据表明该行中结果最好的数值. 由表 2 可知:

- 与 BDD 方法相比, CDG4CPV 可使平均探索状态数目减少 91.38%, 平均状态空间探索时间缩短 86.25%, 平均内存开销降低 69.80%.
- 与 BDD+PPOR 和 BDD+MPOR 方法相比, CDG4CPV 可分别使平均状态数减少 89.46%和 85.43%, 平均状态空间探索时间缩短 79.61%和 81.38%, 平均内存开销降低 64.96%和 73.53%.

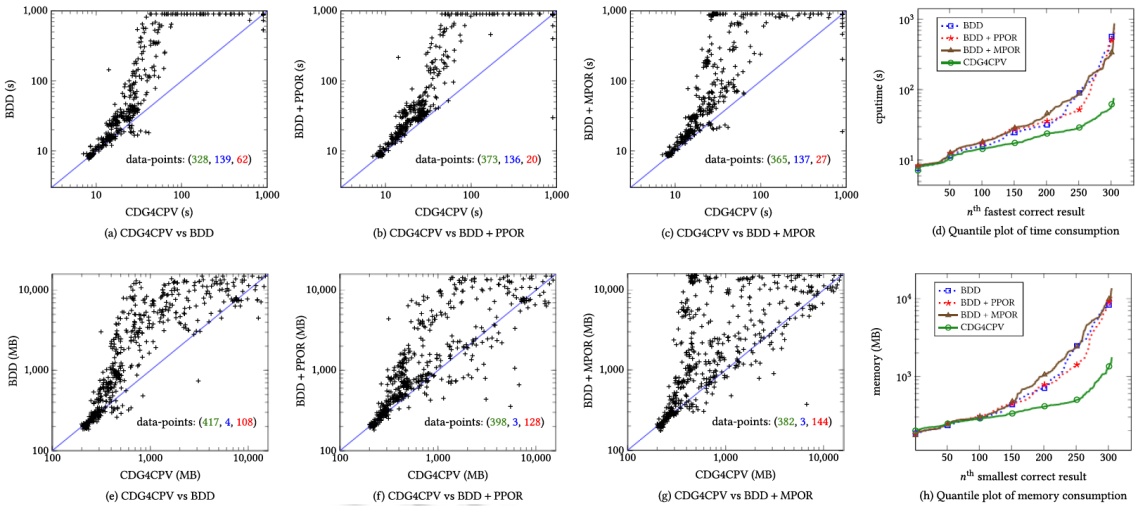


图 13 与其他算法的实验对比结果

表 2 不同方法在 305 个并发程序上的性能对比

No.	BDD			BDD+PPOR		
	#State	#Time	#Mem.	#State	#Time	#Mem.
1	9 564.95	1.89	258.68	10 876.82	2.48	291.27
2	10 412.13	1.77	283.00	10 933.00	1.43	305.13
3	104 556.14	84.41	2 009.14	64 793.43	74.91	1 994.00
4	30 973.63	5.70	458.75	20 506.50	5.50	414.38
5	1 122.00	1.20	258.00	**872.00**	**0.86**	239.00
6	–	–	–	–	–	–
7	–	–	–	–	–	–
8	7 543.50	5.15	1 479.50	9 316.50	4.57	1 364.00
9	958.00	1.25	252.67	1 170.00	1.06	255.33
10	2 962.00	1.30	320.00	3 475.00	1.80	312.00
11	203 716.97	47.22	1 672.65	164 558.60	29.94	1 411.10
12	58 060.86	12.45	755.68	67 355.41	17.39	679.00
#Avg.	148 721.53	35.71	1 317.65	121 590.51	24.08	1 135.84

表 2 不同方法在 305 个并发程序上的性能对比(续)

No.	BDD+MPOR			CDG4CPV		
	#State	#Time	#Mem.	#State	#Time	#Mem.
1	1 613.84	1.07	238.02	**397.18**	**0.31**	**214.32**
2	3 717.88	1.36	297.38	**953.75**	**0.50**	**238.13**
3	42 747.14	53.57	2 406.00	**8 461.14**	**5.49**	**563.14**
4	11 966.75	4.87	446.13	**3 041.25**	**1.77**	**313.75**
5	**830.00**	1.24	264.00	1 132.00	1.62	298.00
6	–	–	–	–	–	–
7	–	–	–	–	–	–
8	4 897.50	4.92	1 197.00	**3 173.00**	**1.99**	**861.00**
9	778.33	1.03	274.00	**593.33**	**0.73**	**248.67**
10	1 797.00	1.61	278.00	**708.00**	**0.89**	**276.00**
11	123 749.72	35.20	1 949.14	**16 931.06**	**6.56**	**443.00**
12	20 615.45	9.02	644.32	**11 481.05**	**2.24**	**362.23**
#Avg.	87 959.69	26.37	1 503.50	**12 819.03**	**4.91**	**397.98**

从表 2 中可以发现, 虽然 BDD+MPOR 方法的平均探索状态数目比 BDD+PPOR 方法的少, 但是其平均验证时间开销更高. 出现该异常的主要原因在于: MPOR 方法在并发程序的验证过程中需维护和计算线程迁移的依赖链, 以此保证所探索的路径满足准单调准则(即保证探索方法的最优性), 其使用的准单调计算(quasi-monotonic computation)方法开销相对较高. 平均而言, BDD+MPOR 方法在 305 个并发程序的验证过程中需耗费 2.91 s 来维护依赖链, 当遇到规模较大的并发程序时, MPOR 方法的剪枝收益与准单调计算开销难以得到有效的平衡. 相比之下, 本文工具不仅通过孤立迁移的优先探索策略减少了大部分冗余分支的探索, 而且仅在每个状态处计算可执行后继迁移间是否存在约束依赖. CDG4CPV 在 305 个并发程序的验证过程中平均仅计算 268.36 次约束依赖, 耗时 0.23 s, 因此, 这部分计算的时间开销平均较低.

4 结 论

因线程调度的不确定性和数据同步的复杂性, 并发程序模型检测常面临状态空间爆炸的问题. 如何高效且系统性地遍历程序路径空间, 具有重要研究意义. 针对粗略的线程迁移独立性分析会显著增加冗余路径等价类探索的问题, 本文通过分析线程迁移边间的依赖约束, 构建并发程序的约束依赖图, 实现了对迁移依赖性判断的细化. 并基于开源程序验证工具 CPAchecker, 实现了基于约束依赖图的并发程序模型检测工具 CDG4CPV, 可在展开控制流图时动态地剪裁冗余的独立可执行分支. SV-COMP 2022 并发程序数据集的对比实验展示了 CDG4CPV 在状态空间缩减方面具有一定的优势, 验证所需时间和内存开销相比更低, 表明该工具在并发程序验证方面具有可用性.

未来将进一步优化基于约束依赖图的并发程序模型检测工具 CDG4CPV, 并在此基础上分析设计其他类型迁移边的依赖约束计算方法, 实现对并发程序路径等价类的进一步划分.

References:

[1] Chandy KM, Misra J, Haas LM. Distributed deadlock detection. ACM Trans. on Computer Systems (TOCS), 1983, 1(2): 144−156. [doi: 10.1145/357360.357365]

[2] Kasikci B, Zamfir C, Candea G. Data races vs. data race bugs: Telling the difference with portend. ACM SIGPLAN Notices, 2012, 47(4): 185−198. [doi: 10.1145/2248487.2150997]

[3] Jin G, Song L, Zhang W, *et al*. Automated atomicity-violation fixing. In: Andy G, ed. Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM, 2011. 389−400. [doi: 10.1145/1993316.1993544]

[4] Clarke EM. Model checking. In: Ramesh S, Sivakumar G, eds. Proc. of the Int'l Conf. on Foundations of Software Technology and Theoretical Computer Science. Berlin, Heidelberg: Springer, 1997. 54−56. [doi: 10.1007/BFb0058022]

[5] Clarke EM, Henzinger TA, Veith H, *et al*. Handbook of Model Checking. Cham: Springer, 2018. [doi: 10.1007/978-3-319-10575-8]

[6] Burch JR, Clarke EM, McMillan KL, *et al*. Symbolic model checking: 1020 states and beyond. Information and Computation, 1992, 98(2): 142−170. [doi: 10.1016/0890-5401(92)90017-A]

[7] Akers SB. Binary decision diagrams. IEEE Trans. on Computers, 1978, 27(6): 509−516. [doi: 10.1109/TC.1978.1675141]

[8] McMillan KL. Interpolation and SAT-based model checking. In: Hunt WA, Somenzi F, eds. Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2003. 1−13. [doi: 10.1007/978-3-540-45069-6_1]

[9] Yin L, Dong W, Liu W, *et al*. On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. on Software Engineering, 2018, 46(5): 549−565. [doi: 10.1109/TSE.2018.2864122]

[10] Su J, Tian C, Duan Z. Conditional interpolation: Making concurrent program verification more effective. In: Diomidis S, Georigios G, Marsha C, *et al*., eds. Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2021. 144−154. [doi: 10.1145/3468264.3468602]

[11] Clarke EM, Grumberg O, Minea M, *et al*. State space reduction using partial order techniques. Int'l Journal on Software Tools for Technology Transfer, 1999, 2(3): 279−287. [doi: 10.1007/s100090050035]

[12] Godefroid P. Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem. Berlin, Heidelberg: Springer, 1996. [doi: 10.1007/3-540-60761-7]

[13] Wang C, Yang Z, Kahlon V, *et al*. Peephole partial order reduction. In: Ramakrishnan CR, Rehof J, eds. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2008. 382−396. [doi: 10.1007/978-3-540-78800-3_29]

[14] Kahlon V, Wang C, Gupta A. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: Bouajjani A, Maler O, eds. Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2009. 398−413. [doi: 10.1007/978-3-642-02658-4_31]

[15] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. ACM SIGPLAN Notices, 2005, 40(1): 110−121. [doi: 10.1145/1040305.1040315]

[16] Abdulla P, Aronis S, Jonsson B, *et al*. Optimal dynamic partial order reduction. ACM SIGPLAN Notices, 2014, 49(1): 373−384. [doi: 10.1145/2578855.2535845]

[17] Chalupa M, Chatterjee K, Pavlogiannis A, *et al*. Data-centric dynamic partial order reduction. In: Philip W, ed. Proc. of the ACM on Programming Languages. New York: Association for Computing Machinery, 2017. No.31. [doi: 10.1145/3158119]

[18] Mazurkiewicz A. Trace theory. In: Brauer W, Reisig W, Rozenberg G, eds. Proc. of the Advanced Course on Petri Nets. Berlin, Heidelberg: Springer, 1986. 278−324. [doi: 10.1007/3-540-17906-2_30]

[19] Tian C, Duan Z, Duan Z, *et al*. More effective interpolations in software model checking. In: Grigore R, Massimiliano DP, Tien NN, eds. Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). New York: IEEE, 2017. 183−193. [doi: 10.1109/ASE.2017.8115631]

[20] Beyer D, Stahlbauer A. BDD-based software verification. Int'l Journal on Software Tools for Technology Transfer, 2014, 16(5): 507−518. [doi: 10.1007/s10009-014-0334-1]

[21] Peled D. Combining partial order reductions with on-the-fly model-checking. In: Dill DL, ed. Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 1994. 377−390. [doi: 10.1007/BF00121262]

[22] Beyer D, Keremoglu ME. CPAchecker: A tool for configurable software verification. In: Gopalakrishnan G, Qadeer S, eds. Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2011. 184−190. [doi: 10.1007/978-3-642-22110-1_16]

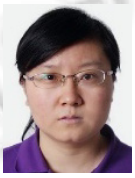
[23] Beyer D, Chlipala AJ, Henzinger TA, *et al*. The Blast query language for software verification. In: Giacobazzi R, ed. Proc. of the Int'l Static Analysis Symp. Berlin, Heidelberg: Springer, 2004. 2−18. [doi: 10.1007/978-3-540-27864-1_2]

[24] Biere A, Cimatti A, Clarke EM, *et al*. Handbook of Satisfiability. Amsterdam: IOS Press, 2009. 457−481.

[25] Klumpp D, Dietsch D, Heizmann M, *et al*. Ultimate GemCutter and the axes of generalization. In: Fisman D, Rosu G, eds. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2022. 479−483. [doi: 10.1007/978-3-030-99527-0_35]

[26] Dietsch D, Heizmann M, Nutz A, *et al*. Ultimate Taipan with symbolic interpretation and fluid abstractions. In: Biere A, Parker D, eds. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2020. 418−422. [doi: 10.1007/978-3-030-45237-7_32]

苏杰(1993－), 男, 博士生, CCF 学生会员, 主要研究领域为并发程序模型检测, 形式化验证.



田聪(1981－), 女, 博士, 教授, CCF 杰出会员, 主要研究领域为软件安全, 形式化方法, 模型检测.

杨祖超(1997－), 男, 博士生, 主要研究领域为软件安全, 形式化方法, 并发程序数据竞争检测.

段振华(1948－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为高可信软件开发及验证.