软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(7):3116−3133 [doi: 10.13328/j.cnki.jos.006860]

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

面向未解释程序的合作验证方法*

杜一德[1], 洪伟疆[1,2], 陈振邦[1], 王 戟[1,2]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](高性能计算国家重点实验室(国防科技大学), 湖南 长沙 410073)

通信作者: 陈振邦, E-mail: zbchen@nudt.edu.cn

摘 要: 未解释程序的验证问题通常是不可判定的, 但是最近有研究发现, 存在一类满足 coherence 性质的未解释程序, 其验证问题是可判定的, 并且计算复杂度为 PSPACE 完全的. 在此结果的基础上, 一个针对一般未解释程序验证的基于路径抽象的反例抽象精化(counterexample-guided abstraction refinement, CEGAR)框架被提出, 并展现了良好的验证效率. 即使如此, 对未解释程序的验证工作依然需要多次迭代, 特别是利用该方法在针对多个程序验证时, 不同的程序之间的验证过程是彼此独立的, 存在验证开销巨大的问题. 发现被验证的程序之间较为相似时, 不可行路径的抽象模型可以在不同的程序之间复用. 因此, 提出了一个合作验证的框架, 收集在验证过程中不可行路径的抽象模型, 并在对新程序进行验证时, 用已保存的抽象模型对程序进行精化, 提前删减一些已验证的程序路径, 从而提高验证效率. 此外, 通过对验证过程中的状态信息进行精简, 对现有的基于状态等价的路径抽象方法进行优化, 以进一步提升其泛化能力. 对合作验证的框架以及路径抽象的优化方法进行了实现, 并在两个具有代表性的程序集上分别取得了 2.70×和 1.49×的加速.

关键词: 合作验证; 未解释程序; 反例抽象精化; 路径抽象; 复用

中图法分类号: TP311

中文引用格式: 杜一德, 洪伟疆, 陈振邦, 王戟. 面向未解释程序的合作验证方法. 软件学报, 2023, 34(7): 3116–3133. http://www.jos.org.cn/1000-9825/6860.htm

英文引用格式: Du YD, Hong WJ, Chen ZB, Wang J. Collaborative Verification Method of Uninterpreted Programs. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3116−3133 (in Chinese). http://www.jos.org.cn/1000-9825/6860.htm

Collaborative Verification Method of Uninterpreted Programs

DU Yi-De[1], HONG Wei-Jiang[1,2], CHEN Zhen-Bang[1], WANG Ji[1,2]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](State Key Laboratory of High Performance Computing (National University of Defense Technology), Changsha 410073, China)

Abstract: The verification of an uninterpreted program is undecidable in general. Recently, a decidable fragment (called coherent) of uninterpreted programs is discovered and the verification of coherent uninterpreted programs is PSPACE complete. Based on the results of coherent uninterpreted programs, a trace abstraction-based verification method in CEGAR (counterexample-guided abstraction refinement) style is proposed for general uninterpreted programs, and is very effective. Although that, the verification of uninterpreted programs sometimes needs many refinements. Especially when verify multiple programs with this method, the verifications of different programs are independent of each other and has high complexity. However, it is observed that those abstract models of infeasible counter-example traces are reusable and can benefit from each other's verification when the programs to be verified are similar. In this work, a collaborative verification framework is proposed that accumulates the abstract models of infeasible traces during the programs' verification. When a new program is to be verified, the program abstraction is refined with the accumulated abstract model first to wipe

* 基金项目: 国家自然科学基金(62172429, 62032024)

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

off those infeasible traces to improve the verification efficiency. Besides, an optimized congruence-based trace abstraction method is also proposed that compacting the states during the verification to enlarge the scope of the abstractions of the infeasible traces. The collaborative verification framework and the optimized trace abstraction method have been implemented, achieving on average 2.70× and 1.49× speedups on two representative benchmarks.

Key words: collaborative verification; uninterpreted program; counterexample-guided abstraction refinement; trace abstraction; reuse

未解释程序[1]是指这样一类程序, 程序中的函数仅代表一个符号而没有具体的函数定义, 该函数只满足最一般的性质, 即相同的输入会得到相同的输出. 例如, 有未解释程序: $x:=y;x:=f(x);y:=f(y);assert(x=y)$, 该程序中的函数 f 只有函数符号, 而没有具体定义. 该程序在初始时满足 x 与 y 等价, 经过相同的函数 f 作用之后, 程序仍然满足 x 与 y 等价, 即程序的断言成立. 当对一个程序 P 进行验证时, 可以将 P 中的函数作未解释化处理, 用相应的未解释程序 P_u 作为程序 P 的上近似程序, 并对 P_u 进行验证. 通常情况下, 对 P_u 进行验证有着更低的计算开销. 即使如此, 未解释程序的验证问题依然是不可判定的[1].

最近, 一个可判定的未解释的程序类被发现[1], 即满足被称为 coherence 性质的未解释程序的验证问题是可判定的, 并且验证的复杂度是 PSPACE-完全的. 在上述结论的基础上, 一种对一般未解释程序进行验证的、基于路径抽象的 CEGAR (counterexample-guided abstraction refinement)方法被提出来[2]. 在该方法中, 首先对被验证的程序 P 进行抽象, 得到一个有限状态自动机(finite state automata, FSA), 记为 A_P, 其接受的路径包含程序 P 中所有违反断言的路径. 之后, 如果 A_P 接受的语言 $L(A_P)$非空, 则从中抽取一条路径 t, 并检查路径 t 的可行性: 如果 t 是可行的, 则一条真反例路径被找到, 程序 P 不满足断言; 若 t 不可行, 则对 t 抽象得到一个抽象模型, 记为 A_t, A_t 接受的路径都是不可行的路径, 并且与路径 t 有着相同的不可行原因. 之后, 利用 A_t 对 A_P 进行精化, 将 A_t 中接受的路径从 A_P 中精化掉得到 $A_{P'}$, 如果此时 $L(A_{P'})$为空, 则验证结束, 程序 P 为正确的; 否则, 重复上述的步骤对程序的抽象模型进行精化, 直到验证程序为正确, 或者找到一条错误路径证明程序错误, 或者超时.

另外, 上述方法[2]针对未解释程序提出了一种更有效的基于状态等价的路径抽象方法. 该路径抽象方法抽象出了路径不可行的核心原因, 其泛化能力要优于传统的基于插值的路径抽象方法[3], 因此具有更高的验证效率. 即使如此, 未解释程序的验证问题依然有着很高的复杂度, 需要对程序进行多次迭代精化来完成验证过程. 如何提高路径抽象方法的泛化能力, 进一步提高验证方法的效率, 是件具有挑战性的工作.

上述方法在针对批量未解释程序验证时, 程序之间的验证是彼此独立的. 经观察发现, 在一些场景下, 对批量程序进行验证时, 程序之间具有一些相似性, 例如对不同软件版本的回归验证以及对模块化开发的程序集的验证. 在对这些程序进行验证时, 可以利用程序之间的相似性, 收集已验证程序的验证结果, 在对新程序进行验证时复用这些验证结果, 避免对程序的重复验证, 以此提升对批量程序的验证效率. 另外一个观察是, 在对程序进行 CEGAR 验证的过程中, 不可行路径的抽象模型对程序的验证效率有着重要影响, 该抽象模型的泛化能力越好, 能精化掉的不可行路径就越多, 这对程序验证效率的提高起着直接的作用. 而现有的基于状态等价的路径抽象方法[2]对不可行路径进行泛化时, 没有根据路径违反 coherence 性质的不同原因进行分别处理, 有较大的改进空间.

基于第 1 个观察, 本文提出了面向未解释程序的合作验证框架, 该框架的主要想法是, 在不同的程序之间复用程序验证过程中不可行路径的抽象模型. 即收集程序验证过程中不可行路径的抽象模型, 在对新程序验证之前, 首先利用收集的抽象模型对程序进行精化, 删减掉程序中已被验证的不可行的路径, 从而避免重复验证, 提升验证效率. 此外, 根据第 2 个观察, 本文对现有的路径抽象方法[2]进行了改进, 根据路径违反 coherence 性质的不同原因对路径信息修复, 并精简路径的状态迁移中的信息, 从而提升路径抽象方法的泛化能力. 本文对上述合作验证框架以及优化的路径抽象算法进行了实现, 并根据两种常见的场景设计了程序集. 在该程序集上的实验结果表明, 合作验证的方法提升了对批量未解释程序验证的效率.

本文的主要贡献包括以下 3 点.

(1) 提出了面向未解释程序的合作验证的框架, 该框架收集并复用验证过程中的不可行路径的抽象模

型, 提升了对批量未解释程序的验证效率.

(2) 对现有的基于状态等价的路径抽象方法[2]进行了优化, 进一步提升了不可行路径的抽象模型的泛化能力, 从而提升基于路径抽象的 CEGAR 验证方法的验证效率.

(3) 对上述合作验证的框架和路径抽象算法的优化进行了实现, 得到一个针对未解释程序的验证工具, 并在两个具有代表性的程序集上进行了实验, 分别取得了 2.70×和 1.49×的加速, 表明了合作验证方法的有效性.

本文第 1 节介绍相关的背景知识, 包括未解释程序及其验证问题以及基于路径抽象的 CEGAR 方法. 第 2 节阐述研究动机以及面向未解释程序的合作验证框架. 第 3 节给出本文对现有路径抽象方法的改进方法. 第 4 节进行相关的实验并给出实验结果. 第 5 节介绍相关工作. 第 6 节对全文进行总结, 并对未来工作进行展望.

1 相关背景

1.1 未解释程序

(1) 未解释程序的语法

用Σ代表一个有穷集合, 其中, $C\subseteq\Sigma$表示常量的集合, $F\subseteq\Sigma$表示函数的集合, $V\subseteq\Sigma$表示变量的集合. 未解释程序的语法定义如图 1 所示, 其中, $c\in C$ 表示一个常量, $f\in F$ 表示一个函数, $x,y\in V$ 表示变量, $\bar{z}$ 为变量或常量的元组.

$\langle stmt\rangle ::= \textbf{skip}\mid x:=c\mid x:=y\mid x:=f(\bar{z})$
$\mid assume(\langle cond\rangle)\mid assert(\langle cond\rangle)$
$\mid \langle stmt\rangle;\langle stmt\rangle$
$\mid \textbf{if}\ (\langle cond\rangle)\ \textbf{then}\ \langle stmt\rangle\ \textbf{else}\ \langle stmt\rangle$
$\mid \textbf{while}\ (\langle cond\rangle)\ \langle stmt\rangle$
$\langle cond\rangle ::= x=y\mid \langle cond\rangle\wedge\langle cond\rangle\mid\neg\langle cond\rangle$

图 1 未解释程序的语法

在上述定义中, ***skip*** 代表该语句什么都不做, $x:=c$, $x:=y$ 与 $x:=f(\bar{z})$表示不同的赋值语句. *assume* 语句表示程序约束, 可作为程序中的前提条件; *assert* 语句表示断言, 表示程序需要验证的属性. 而当一条路径到达 *assume*(*assert*)语句时, 如果当前路径不满足⟨*cond*⟩, *assume*(⟨*cond*⟩) (*assert*(⟨*cond*⟩))会使路径终止; 如果路径满足⟨*cond*⟩, 该 *assume*(*assert*)语句就相当于 ***skip*** 语句. 上述定义中有 3 种控制结构, 分别为顺序结构、**if-then-else** 分支结构、**while** 循环结构. 关于条件语句⟨*cond*⟩的定义, 原子的条件语句为 $x=y$, 在此基础上定义了条件语句的合取和取非运算, 通过这些运算可以得到组合的条件语句.

(2) 未解释程序的语义

未解释程序的语义定义在一个给定的数据模型 $M=(U,I)$上, 其中, U 代表全部元素的集合, I 给出了Σ中符号的一个解释. 例如, I 将 C 中的常量映射到 U 中的元素上, 将 F 中的函数映射到 U 中的关系上. 因此, 未解释程序的语义可以定义为在这样的数据模型上的一个状态迁移图, 其中每个状态定义为一个映射 $S:V\to U$,该映射将程序中的变量映射到数据模型中的元素. 在初始状态时, 该映射为空, 记为∅, 并且引入断言失败的状态, 记为 *Fail*. 每个状态可通过程序语句更新到新的状态, 详细的状态迁移规则在 Hong 等人[2]的工作中定义.

(3) 未解释程序的验证问题

未解释程序的验证问题即检查程序中每个断言语句 *assert*(⟨*cond*⟩)的可满足性, 这要求条件语句⟨*cond*⟩在程序执行过程中, 在任意的数据模型上都成立, 也就是¬⟨*cond*⟩不被满足. 故程序 P 的验证问题可以归约为一个可达性问题, 定义如下(其中, "$\twoheadrightarrow^{P}$"表示经过程序 P 的执行):

$$\forall M\cdot\langle M,\varnothing\rangle\twoheadrightarrow^{P}\langle M,Fail\rangle.$$

其中, 元组⟨M,S⟩表示当前数据模型为 M, 当前程序状态为 S. 在上式中, ∅表示初始状态, *Fail* 表示失败状态. 因此, 该公式表示在程序的执行过程中, 在任何数据模型 M 上, *Fail* 状态都是不可达的. 一般来说, 未解释程序的可达性问题是不可判定的. 最近有工作[1]发现, 满足称为 coherence 性质的未解释程序的验证问题是可判

定的, 并且计算复杂度为 PSPACE-完全的.

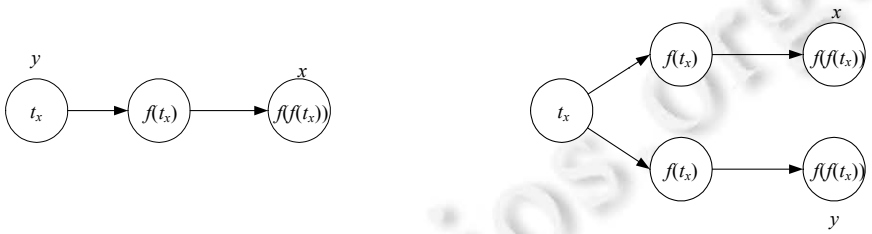
(4) 满足 coherence 性质的程序

在实际的程序执行过程中, 程序变量都保存有一个值. 而未解释程序可定义在任意数据模型上, 因此, 在未解释程序执行中的每一个程序点处, 假定其中的变量都保存有一个项, 而程序语句可看作对项的修改. 例如, 有程序变量 y, 其保存的当前项为 t_y, 语句 $x:=f(y)$执行后, 函数 f 以项 t_y 为输入计算得到项 $f(t_y)$, 并赋值给变量 x. 因此, 从程序变量的初始项出发, 经过每条程序语句可计算得到执行完当前语句之后的程序点的项的集合, 并且可推断项之间的等价关系、不等价关系以及函数关系. 在程序执行过程中, 项是由程序变量来保存的, 故程序执行到某一程序点处的程序状态可用三元组(E,D,F)的形式表示, 其中, E 代表程序变量间的等价关系, D 代表程序变量间的不等价关系, F 代表程序变量间的函数关系. 则程序的初始状态可记为$(ID,\varnothing,\varnothing)$, 其中, ID 代表变量与其自身等价的关系. 在程序的执行过程中, 由于项及项之间的关系不断被程序语句修改, 因此代表程序状态的三元组也不断被修改. 在这个过程中, 称两个程序状态是等价的当且仅当两个程序状态的 E, D, F 都是等价的.

在程序的执行过程中, 所有的项都是由初始项计算得来的, 因此, 若存在两个项, 例如项 v 和项 u, 并且项 v 是通过项 u 计算得来的话, 则称项 v 是项 u 的超项, 例如 $v=f(u)$. 在这个计算过程中, coherence 性质需满足两点要求, 即 memoizing 和 early assumes, 这两条性质的具体要求如下.

- Memoizing[1]: 若一个项 t 被计算过, 当该项被再次计算到时, 必须有一个变量 x 当前的项赋值为 t, 即重复计算的项不能被丢弃.
- Early assumes[1]: 当路径中有形如 $assume(x=y)$的语句出现时, x 或 y 的超项必须赋值给了某一变量. 即由语句 $assume(x=y)$得到新的等价关系并计算等价关系的闭包时, 该语句出现的位置要足够早, 此时 x 或 y 的超项还未被丢弃.

例如, 有路径 t_1 为 $x:=y;x:=f(x);x:=f(x);y:=f(y);y:=f(y);assume(x!=y)$, 该路径违反了 memoizing 的要求. t_1 初始执行语句 $x:=y$ 之后, x 与 y 保存的项等价, 记为 t_x. 之后, 程序语句在该项的基础上进行计算. 图 2(a)为继续执行语句 $x:=f(x);x:=f(x)$后, 计算得到的项之间的关系. 可以看到, 项 $f(t_x)$没有被变量保存. 而由于三元组状态数量有限性的要求, 其中不保存类似 $x:=f(f(y))$等超过一层的函数关系, 因此, x 与 y 之间的函数关系丢失. 在后续执行语句 $y:=f(y);y:=f(y)$时, 项 $f(t_x)$被重新计算, 并且 $f(t_x)$没有被变量保存, 这违反了 memoizing 的要求, 因此产生了图 2(b)下方分支中项之间的关系. 由于函数关系 $x:=f(f(y))$的丢失, 执行语句 $y:=f(y);y:=f(y)$后, 无法得到变量 x 和 y 保存的项是一样的, 导致 x 与 y 的等价关系丢失, 使得三元组状态中的信息不再完备.



(a) 执行 $x:=y;x:=f(x);x:=f(x)$后项之间的关系　　(b) 执行 $x:=y;x:=f(x);x:=f(x);y:=f(y);y:=f(y)$后项之间的关系

图 2　路径 t_1 执行时, 项之间的关系

再例如, 有路径 t_2 为 $x:=f(t);y:=f(k);x:=f(x);y:=f(y);assume(t=k);assume(x!=y)$, 该路径违反了 early-assumes 的要求. t_2 在执行语句 $assume(t=k)$之前, 项之间的关系如图 3 所示. 在这之后, t_2 执行语句 $assume(t=k)$, 项 t_t 和项 t_k 等价, 而此时项 t_t 和项 t_k 的超项 $f(t_t)$与 $f(t_k)$没有被相应变量保存, 这违反了 early-assumes 的要求. 语句 $assume(t=k)$出现的位置太晚, 导致项 t_t 和项 t_k 的超项没有被变量保存, 从而由项 t_t 和项 t_k 等价计算等价闭包时, 丢失了 x 与 y 的等价关系, 使得三元组状态中的信息不再完备.

如果一条路径满足上述两个条件, 则称该路径是满足 coherence 性质的. 若一个程序 P 的所有路径都是满

足 coherence 性质的, 则称程序 P 是满足 coherence 性质的. 在这种情况下, P 中可推断出的等价关系闭包是完备的, 这是保证验证完备性的关键. 这时, 假设程序 P 的语句集合为Σ, 以该集合为字母表定义 FSA A_P 与 FSA A_U, 其中, A_P 为程序 P 的一个上近似, 接受 P 中所有违反断言的路径; A_U 接受所有满足 coherence 性质并且可行的路径. 则程序 P 的验证问题就可转化为判断 $L(A_P)\cap L(A_U)$是否为空的问题[1].

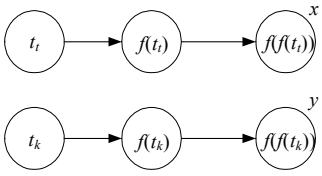


图 3 路径 t_2 在执行语句 *assume*(t=k)之前, 项之间的关系

由于路径是有限长的, 因此, 所有的路径都可转化为满足 coherence 性质的路径[1]. 若有一条路径不满足 coherence 性质, 即路径在执行过程中违反了 memoizing 或 early assumes 的要求, 导致某些项的信息丢失, 可在项丢失的位置添加程序中未出现过的变量作为辅助变量, 对这些项进行保存, 从而使路径满足 coherence 的性质, 这些辅助变量被称作 *ghost* 变量[1]. 例如, 在路径 t_1 中添加 *ghost* 变量之后, 路径变为 x:=y;x:=$f(x)$;g_0:=x;x:=$f(x)$;y:=$f(y)$;y:=$f(y)$;*assume*(x!=y), 在第 1 次计算得到项 $f(t_x)$时, 由 *ghost* 变量 g_0 对该项进行保存. 因此, 在执行语句 y:=$f(y)$之前, 状态中的函数关系 x:=$f(g_0)$和 g_0:=$f(y)$间接表示了 x 与 y 之间的函数关系 x=$f(f(y))$. 故在语句 y:=$f(y)$; y:=$f(y)$执行后, x 与 y 之间的等价关系被计算了出来, 保证了信息的完备性. 同样地, 在 t_2 中添加 *ghost* 变量后, 路径变为 x:=$f(t)$;y:=$f(k)$;g_0:=x;x:=$f(x)$;g_1:=y;y:=$f(y)$;*assume*(t=k);*assume*(x!=y). 此时, 路径在执行语句 *assume*(t=k)时, 项 $f(t_t)$和项 $f(t_k)$分别被 g_0 和 g_1 保存, 函数关系 x:=$f(g_0)$和 g_0:=$f(t)$间接表示了函数关系 x:=$f(f(t))$, 函数关系 y:=$f(g_1)$和 g_1:=$f(k)$间接表示了函数关系 y:=$f(f(k))$, 因此, 在由项 t_t 和项 t_k 等价来计算等价闭包时, x 与 y 的等价关系是准确的, 保证了信息的完备性.

1.2 基于状态等价实现路径抽象的CEGAR方法

尽管 coherent 的未解释程序的验证问题是可判定的, 但一般未解释程序的验证问题仍然是不可判定的. 最近, 有方法基于 coherent 程序可判定的结果以及 CEGAR 的验证框架, 提出了对一般未解释程序的基于路径抽象的 CEGAR 框架[2], 其过程与传统的 CEGAR 过程相似. 该方法首先对程序中满足 coherence 性质的部分直接验证[1], 然后对不满足 coherence 性质的部分采用基于路径抽象的 CEGAR 的方法进行验证.

在对程序利用 CEGAR 验证的过程中, 不可行路径的抽象方法对程序验证的效率有重要的影响, 因此,该方法[2]针对未解释程序提出了基于状态等价的路径抽象方法. 该方法利用上述三元组(E,D,F)的形式表示程序状态, 这种表示方式可以抽象出路径不可行的核心原因, 即程序变量间等价关系与不等价关系的冲突. 据此, 该方法首先将不可行的路径转换为满足 coherence 性质的路径, 以保证计算得到的状态迁移中信息的完备性; 之后, 该方法根据该状态迁移中重复出现的状态序列泛化出循环, 得到接受不可行原因相同的更多路径.这种基于状态等价的路径抽象方法抽象出了路径不可行的核心原因, 故该路径抽象方法的泛化能力要优于传统的基于插值的路径抽象方法[3].

2 面向未解释程序的合作验证框架

2.1 研究动机

表 1 中给出了 3 个相似的程序, 这 3 个程序是以两个代码块为基础, 通过不同的控制结构组合而得到的. 这 3 个程序实现的功能不同, 但程序之间却有着相似之处. 通过观察可知, 程序 P_1 进入真分支的路径和程序 P_0 中进入循环一次的路径是相似的, 程序 P_2 真分支中的路径和程序 P_0 的路径是相似的, 并且程序 P_2 进入假分支的路径和程序 P_1 进入假分支的路径是相同的. 这说明在对 P_0 验证过程中的不可行路径的抽象模型, 可以对 P_1 和 P_2 的抽象模型进行精化; 同样地, 对 P_1 验证过程中的不可行路径的抽象模型, 也可以对 P_2 的抽象模

型进行精化.

表 1 3 个相似的程序

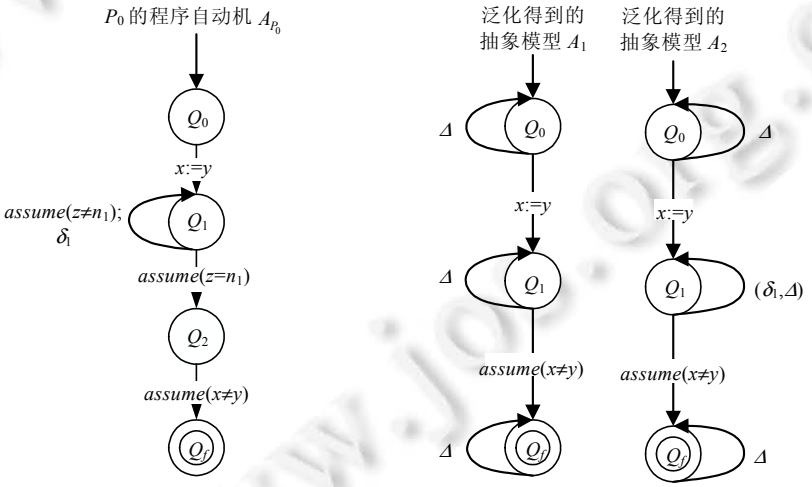
	//程序 P_0		//程序 P_1		//程序 P_2
1	x:=y;	1	x:=y;	1	x:=y;
2		2		2	**if** (z≠n_1){
3		3		3	**while** (z=n_2) {
4	**while** (z≠n_1) {	4	**if** (z≠n_1) {	4	x:=*decr*(x);
5	x:=*decr*(x);	5	x:=*decr*(x);	5	x:=*incr*(x);
6	x:=*incr*(x);	6	x:=*incr*(x);	6	y:=*decr*(y);
7	y:=*decr*(y);	7	y:=*decr*(y);	7	y:=*incr*(y);
8	y:=*incr*(y);	8	y:=*incr*(y);	8	z:=*next*(z);
9	z:=*next*(z);	9	} **else** {	9	}
10	}	10	x:=*incr*(x);	10	} **else** {
11		11	y:=*incr*(y);	11	x:=*incr*(x);
12		12	}	12	y:=*incr*(y);
13	*assert*(x=y);	13		13	}
14		14	*assert*(x=y);	14	*assert*(x=y);

在上述场景下, 如果对这 3 个程序采用单独验证的方式进行验证, 分别需要 2 次、2 次和 5 次 CEGAR 迭代精化来完成验证. 而单独验证的方式没有利用到程序之间的相似性, 从而对一些路径进行了重复验证. 因此, 本文考虑保留程序验证过程中的不可行路径的抽象模型, 后续对新程序进行验证时, 首先利用保存的抽象模型对新程序进行精化, 避免重复验证, 从而提高验证效率.

下面以表 1 中给出的 3 个未解释程序作为例子, 采用保存并复用验证过程中不可行路径抽象模型的方式, 对这 3 个程序进行验证. 为简洁起见, 引入了下列符号.

- δ_1: 表示基本块: x:=*decr*(x);x:=*incr*(x);y:=*decr*(y);y:=*incr*(y).
- δ_2: 表示基本块: x:=*incr*(x);y:=*incr*(y).
- Δ: 表示语句集合: {z:=*next*(z),*assume*(z=n_1),*assume*(z≠n_1),*assume*(z=n_2),*assume*(z≠n_2)}, 即不相关的语句集合, 这些语句对最后断言的成立与否没有影响.

在对这 3 个程序进行验证时, 变量 A_C 被引入, 用来收集验证过程中不可行路径的抽象模型, A_C 初始化为空. 首先对程序 P_0 进行验证, 验证开始时, 对 P_0 抽象得到其对应的抽象模型 A_{P_0}, 如图 4(a)所示.



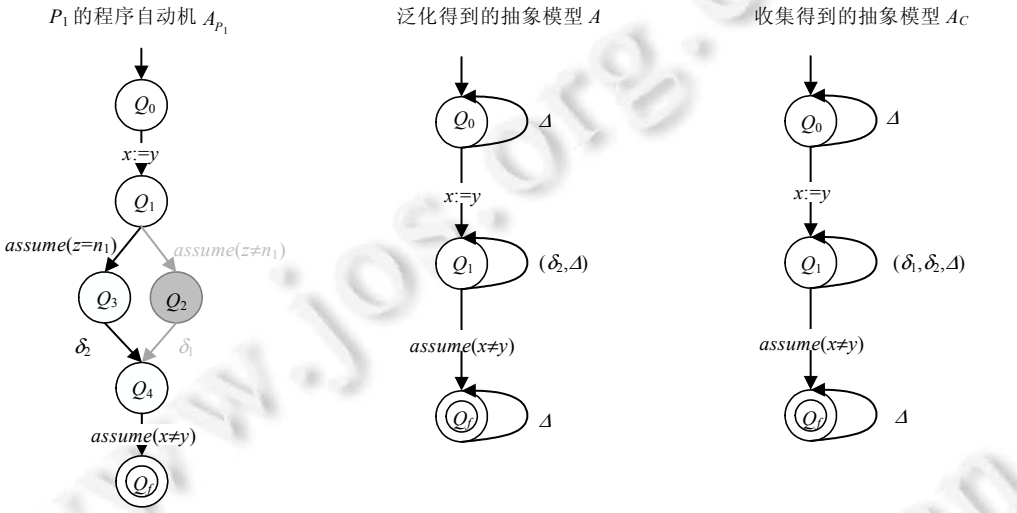
(a) P_0 的程序自动机 (b) P_0 的反例路径泛化得到的抽象模型

图 4 P_0 的验证过程及中间结果

在对 A_{P_0} 利用 CEGAR 进行验证之前, 先利用 A_C 对 A_{P_0} 进行精化. 由于 A_C 初始为空, 因此经过精化后的 A_{P_0} 仍为其本身. 接着, 利用 CEGAR 的方法继续对 A_{P_0} 进行验证, 验证过程中会得到两条路径, 分别为不进

入循环的路径以及进入循环一次的路径. 这两条路径均为不可行路径, 对其泛化后, 得到如图 4(b)中所示的 A_1, A_2 两个抽象模型. 这时, $L(A_{P_0}) \subseteq L(A_1 \cup A_2)$, 即由 A_1, A_2 对 A_{P_0} 进行精化之后, A_{P_0} 接受语言为空, 证明程序 P_0 中没有违反断言的可行路径, 故程序 P_0 为正确的.

在对 P_0 进行验证之后, A_C 收集了对 P_0 验证过程中不可行路径的抽象模型, 即 A_C=A_C∪A_1∪A_2. 接着, 继续对程序 P_1 进行验证, 图 5(a)展示了程序 P_1 的抽象模型 FSA A_{P_1}, 同样地, 在对 A_{P_1} 用 CEGAR 的方式验证之前,首先利用 A_C 对 A_{P_1} 进行精化, 即 $A_{P_1} = A_{P_1} \setminus A_C$. 此时 A_C 非空, 对 A_{P_1} 精化之后, A_{P_1} 中的在程序 P_0 中已验证过的路径被精化掉, 即程序中真分支的路径(图 5(a)中所示灰色的分支). 之后, CEGAR 的过程证明 P_1 的假分支的路径也是不可行的, 并且该路径经过泛化之后得到如图 5(b)中所示的抽象模型 A. 在利用 A 对 A_{P_1} 剩余的部分精化之后, $L(A_{P_1})$ 为空, 证明程序 P_1 是正确的. 因此, 利用 A_C 复用验证结果的方式, 仅通过一次 CEGAR 的迭代, 就证明了程序 P_1 是正确的. 验证结束之后, A_C 更新得到 A_C=A_C∪A, 如图 5(c)中所示.



(a) P_1 的程序自动机 (b) P_1 的反例路径泛化得到的抽象模型 (c) 收集的不可行路径的抽象模型

图 5 P_1 的验证过程及中间结果

接着在对程序 P_2 进行验证时, 这时 $L(A_{P_2}) \subseteq L(A_C)$, 即 A_{P_2} 中接受的路径都是在对程序 P_0 和 P_1 验证过程中已经验证的不可行的路径, 故可以证明程序 P_2 是正确的. 因此, 在对 P_2 进行验证时, 由于复用了对程序 P_0 和 P_1 验证过程中收集的抽象模型, 程序 P_2 的正确性被直接证明, 而没有通过 CEGAR 的迭代验证, 避免了对程序的重复验证.

在对上述 3 个程序进行验证时, 单独验证的方式分别需要 2 次、2 次、5 次迭代, 而利用变量 A_C 对验证过程中的不可行的抽象模型的收集与复用之后, 分别只用了 2 次、1 次、0 次迭代就完成了验证. 据此, 本文提出了针对批量未解释程序的合作验证的框架, 该框架保存并复用对程序验证过程中的不可行路径的抽象模型, 避免了重复验证, 提升了对批量程序验证的效率.

2.2 合作验证框架及算法

图 6 展示了面向未解释程序合作验证的框架, 图中右侧虚线框为对单个程序反例抽象精化的框架, 在此基础上, 框架引入变量 A_C 来收集不可行路径的抽象模型. 对程序集 S 中的程序进行验证时, 选取程序集中的一个程序 P 进行验证.

- 首先, 对 P 进行抽象得到抽象模型 FSA A_P. 在对 A_P 进行 CEGAR 验证前, 先利用 A_C 对 A_P 进行精化, 即 A_P=A_P\A_C, 精化掉 A_P 中已验证为不可行的路径.

- 之后, 对 A_P 剩余的部分进行 CEGAR 的迭代验证, 在这个过程中, 如果 $L(A_P)=\varnothing$ 成立, 则证明程序 P 为正确的; 否则, 从 $L(A_P)$ 中抽取一条路径 t, 并检查 t 的可行性:
  - 如果 t 是可行的, 则一条真反例路径被找到, 程序 P 为错误的, 路径 t 可作为程序违反断言的一个证明;
  - 如果 t 是不可行的, 则需要对 t 进行抽象得到自动机 A_{n+1}, 其中, A_{n+1} 接受的路径均为不可行路径, 并且不可行原因与 t 相同, 之后, 利用 A_{n+1} 对 A_P 进行精化.

上述迭代过程重复, 直至 A_P 接受的语言为空, 证明程序正确, 或者找到一条可行路径证明程序错误.

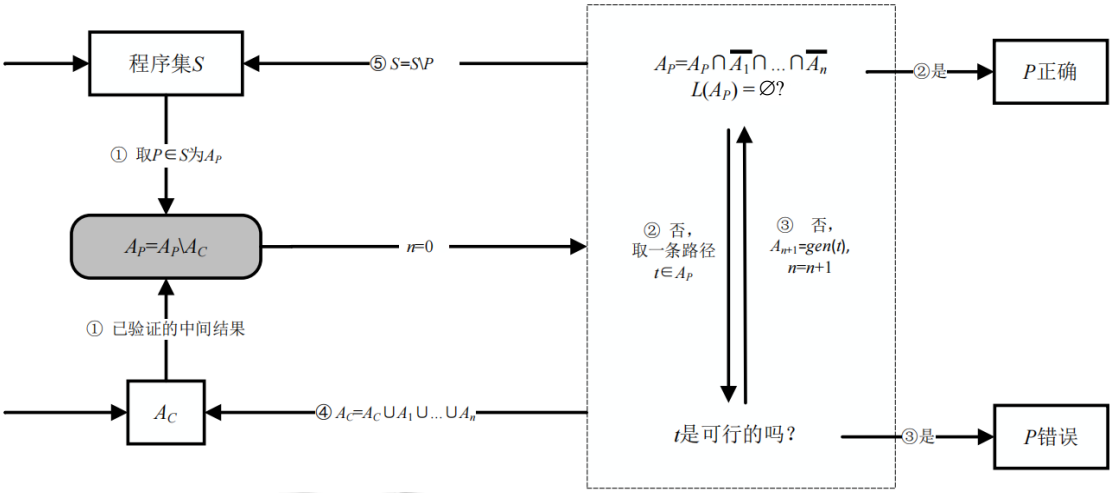


图 6 合作验证框架

当对程序 P 的验证完成之后, 对其验证过程中不可行路径的抽象模型被收集到 A_C 中. 在对新程序进行验证时, 框架首先利用 A_C 对程序的抽象模型进行精化, 再利用 CEGAR 方法对程序剩余的部分进行验证. 因此, 在对批量未解释程序的验证过程中, 合作验证的框架利用了被验证的程序之间的相似性, 避免了重复验证, 提升了验证效率.

算法 1 中给出了合作验证框架的算法细节. 算法的输入为一组待验证的未解释程序集 S, 输出结果为这组未解释程序集中每个程序的验证结果, 正确或者错误. 算法引入变量 A_C 收集验证过程中不可行路径的抽象模型, 且 A_C 初始化为空(第 1 行). 待验证的程序集 S 中的每一个程序 P(第 2 行), $FSA(P)$ 代表对程序 P 进行抽象得到其上近似的模型 A_P. 算法首先利用 A_C 对程序的抽象模型 A_P 进行精化(第 3 行); 之后, 对 A_P 非空的部分通过 CEGAR 的方式进行验证(第 4–11 行), 同时, 变量 A_C 收集验证过程中不可行路径的抽象模型(第 11 行). 在 CEGAR 的过程中, 对不可行路径的抽象方法直接影响着验证方法的效率(第 9 行的 Generalize 方法)和变量 A_C 收集不可行路径抽象模型的效率. 本文对现有的路径抽象方法进行了改进, 相关的细节在第 4 节中介绍.

算法 1. 面向未解释程序的合作验证方法.

输入: 待验证的未解释程序集 S.

输出: 验证结果 R:$S\rightarrow${正确,错误}.

```
1  A_C:=∅;
2  while S≠∅∧P∈S do
3     A_P:=FSA(P)\A_C;   //利用 A_C 复用之前程序的验证结果
4     while L(A_P)≠∅∧t∈L(A_P) do   //反例抽象精化过程
5        (r,T):=Feasible(t);   //算法 2 的路径可行性检查算法, T 表示根据路径 t 计算的状态迁移
6        if r is True then
```

```
7          break;
8       else
9          A_t:=Generalize(t,T);          //算法 3 的路径泛化算法
10         A_P:=A_P\A_t;
11         A_C:=A_C∪A_t;
12      S:=S\{P};
13      if L(A_P)≠∅ then
14         R[P]:=错误;
15      else
16         R[P]:=正确;
17 return R
```

在对程序 P 的 CEGAR 过程完成后, 算法检查 $L(A_P)$是否为空: 如果非空, 说明一条错误路径在 CEGAR 的过程中被发现, 程序 P 错误(第 14 行); 如果为空, 说明 A_P 中所有路径都是不可行的, 即证明程序 P 是正确的(第 16 行). 如果程序集 S 非空的话(第 2 行), 接着继续验证下一个程序, 直至 S 为空, 即所有的程序的验证任务完成, 得到了所有程序的验证结果.

3 细粒度的路径抽象方法

在对程序进行 CEGAR 验证的过程中, 不可行路径的路径抽象方法对程序验证的效率有直接的影响. 路径抽象方法的泛化能力越好, 一次迭代从程序的抽象模型中精化掉的不可行路径越多, 对程序验证需要的迭代次数就越少.

已有的基于状态等价的路径抽象方法[2]以三元组表示程序状态, 该方式可有效抽象出路径不可行的核心原因, 因此, 比基于插值的方法[3]具有更好的泛化能力. 该方法首先通过添加 *ghost* 变量的方式使路径满足 coherence 性质, 保证计算得到的路径状态中信息的完备性; 然后, 根据其中重复出现的状态序列泛化出循环. 但这种方式没有考虑路径违反 coherence 性质的不同原因(即 memoizing 或 early assumes), 引入了过多的 *ghost* 变量, 从而忽略掉了一些可视为等价的状态, 因此在泛化能力上还有较大的提升空间.

针对以上问题, 本文提出了更加细粒度的路径抽象方法, 根据路径违反 coherence 性质的不同原因来修复路径信息, 使路径满足 coherence 性质, 从而保证路径状态中的信息是可靠并且完备的; 并且, 本文对状态中的信息进行了精简, 删除了其中的冗余信息, 从而得到了更多等价的状态. 之后, 检查路径的可行性: 若路径不可行, 则通过合并等价状态的方式对路径进行泛化, 得到不可行原因相同的更多路径. 该方法减少了 *ghost* 变量的引入, 并且精简了状态中的信息, 对路径不可行的原因进行了进一步抽象, 从而泛化出不可行原因相同的更多路径.

3.1 可行性检查

本节介绍路径的可行性检查算法, 该算法的基本思路是: 首先, 将输入的路径 t 转化为满足 coherence 性质的路径 t', 保证根据路径中语句序列计算得到的状态序列中的信息是可靠并且完备的; 之后检查这些状态, 如果计算得到的状态序列中有不一致状态(即状态中的等价关系 E 和不等价关系 D 之间产生冲突, 也就是 E 和 D 的交集非空, 记作 $Inconsistent(S)$), 则对应的路径为不可行的路径.

在使路径满足 coherence 性质时, 与已有的直接在路径中添加 *ghost* 变量的方式相比[2], 本文的方法根据路径违反 coherence 性质的不同原因, 即违反 memoizing, 或者违反 early assumes 来对路径信息进行修复. 之后,根据修复的状态迁移序列检查路径的可行性, 如果路径是不可行的, 则利用在路径可行性检查过程中计算的状态迁移序列对路径进行泛化(后文算法 3).

本文的路径可行性算法首先通过向路径 t 中添加 *ghost* 变量, 以此来修复路径违反 memoizing 要求的情况. 路径中添加的 *ghost* 变量用来保存被丢弃的项, 保证路径状态中信息的完备, 但此方式同时也带来了冗余的信

息. 在第 1.1 节的添加 *ghost* 变量使路径 t_1 满足 memoizing 要求的介绍中, 可以看到, *ghost* 变量作为辅助变量保存被丢弃的中间项, 使得这些项被重复计算到时, 推理所得的关系闭包是完备的. 在这之后, 如果 *ghost* 变量所保存的项不会再被计算到, 此时, 这些 *ghost* 变量相关的信息就是冗余的. 例如, 在通过添加 g_0 保存中间项 $f(t_x)$, 使路径 t_1 满足 memoizing 的要求之后, g_0 作为辅助变量的作用已经完成, 在这之后, 项 $f(t_x)$不会再被计算到, 此时, 这些 *ghost* 变量相关的信息就是冗余的, 例如函数关系 $x:=f(g_0)$和 $y:=f(g_0)$. 因此, 本文在通过添加 *ghost* 变量的方式修复路径违反 memoizing 的情况之后, 删除了状态迁移中冗余的定义.

之后, 算法通过提取违反 early assumes 语句的信息, 并补充到足够提前的位置的方式来修复路径违反 early assumes 要求的情况. 例如第 1.1 节中的路径 t_2 在执行语句 *assume*(t=k)时违反了 early assumes 要求, 即由于该语句位置出现太晚, 导致项 t_t 和项 t_y 的超项被丢弃, 则由 t_t 和 t_k 的等价关系来计算等价关系闭包时, 项 t_t 和项 t_k 的超项之间的等价关系丢失, 导致路径状态中信息的不完备. 在这种情况下, 算法提取到语句 *assume*(t=k)中 t 和 k 的等价关系, 并将这一等价关系补充到足够靠前, 即 t_t 和 t_k 未出现超项被丢弃情况的状态中. 由于提前得到 t 和 k 的等价关系, 在之后的状态迁移的计算过程中, 项 t_t 和项 t_k 的超项之间的等价关系可被计算得到, 保证了状态中信息的完备性.

算法 2 介绍了本文的路径可行性检查算法, 算法首先对路径 t 进行修复, 使其满足 memoizing 的要求(第 1 行). 之后的 *EarlyAssume*(t)函数会检查路径违反 early assumes 要求的情况, 该函数会返回一个由二元对组成的集合ε_t(第 2 行), 集合中的元素为(st,loc), 其中, st 代表路径违反 early assumes 要求的语句, loc 代表可将语句 st 提前以修复违反 early assumes 要求的位置. 在后续计算路径的状态迁移时, 可利用该集合中的元素补充路径由于违反 early assumes 而丢失的信息. 在对路径的状态迁移进行计算时, 初始状态中的信息为变量自身的等价关系(记作 *ID*)(第 3 行). 从初始状态出发, 对于每一条语句 st_i, 该算法根据状态迁移定义[2]来更新状态(第 5 行), 其中, 算法利用ε_t来修复路径违反 early assumes 要求的情况. 例如, 若ε_t中存在元素(*assume*(e),loc), 则当算法根据位置 loc 处的语句 st_i 更新状态后, 关系 e 会被用来继续更新该状态, 从而提前获取相应 assume 语句中的关系. 在根据程序语句更新得到对应的状态 S_i 之后, 算法检查并删除 S_i 中由于添加 *ghost* 变量而带来的冗余的定义(第 6 行). 如前所述, 当一个 *ghost* 变量 g 没有与其他变量的等价关系也没有被函数关系所定义时, F 中由 g 定义的函数关系是多余的, 因为辅助变量 g 保存的中间项不会再被计算到. 因此, 本文按公式(1)所示的方式定义了 *Compact*(S_i), 其中, S_i=(E,D,F), G 是 *ghost* 变量的集合, #E[g]表示 g 的等价类中元素的数量.

$$(E,D,F\backslash F') \mid F'=\{(g,x)\in F \mid g\in G\wedge \#E[g]=1\wedge\forall(i,o)\in F\cdot o\neq g\} \tag{1}$$

算法 2. 路径可行性检查.

输入: 一条反例路径 $t=\langle st_1,\ldots,st_n\rangle$.

输出: t 的可行性以及状态迁移 T.

```
1  t′:=Memoizing(t);
2  E_t:=EarlyAssume(t′);
3  S_0:=(ID,∅,∅)
4  for each 1≤i≤len(t′) do   //由初始状态出发, 根据每一条语句更新状态以及状态间的迁移
5     S_i:=Trans(S_{i−1},st_i,ε_t)
6     S_i:=Compact(S_i)
7     T:=T∪{(S_{i−1},st_i,S_i)}
8     if Inconsistent(S_i) then
9        return (False,T)
10 return (True,T)
```

在对路径的可行性检查的过程中, 算法同时收集了状态之间的迁移(第 7 行). 并且在计算状态迁移的过程中, 检查状态是否为一致的(第 8 行): 若有不一致的状态, 说明该路径是一条不可行路径, 不必再计算之后的状态, 算法终止并返回该路径的可行性结果以及迁移(第 9 行); 若路径中没有不一致状态, 则说明这条路径是

可行的, 返回该路径的可行性结果以及迁移(第 10 行). 其中, 算法中的 for 循环(第 4–9 行)对路径进行了一次遍历, 因此该算法的时间空间复杂度均为 O(n), n 为程序的行数.

3.2 路径泛化

当算法 2 中路径可行性检查发现路径 t 不可行之后, 需要对路径 t 进行泛化. 算法 3 介绍了本文基于可行性检查计算得到的状态序列进行路径抽象的算法. 由于三元组表示程序状态的方式抽象出了路径不可行的核心原因, 故该算法的主要想法是获取目标路径对应的状态迁移, 该状态迁移中的信息要求是完备的; 之后, 算法根据该状态迁移序列合并等价状态, 从而得到不可行原因相同的更多路径.

在对目标路径进行可行性检查时, 为了保证状态信息的完备性, 路径可行性检查算法在路径中添加了 *ghost* 变量来保存丢弃的中间项的信息. 从第 1.1 节中可以看到, 在路径中添加 *ghost* 变量虽然不会影响路径的执行结果, 但修改之后的路径与原始路径是不同的. 因此, 根据修改后的路径计算得到的状态序列与原始路径并不是对应的. 例如, 在第 1.1 节的路径 t_1 中有语句序列片段 $x:=f(x);x:=f(x)$, 把该语句序列片段对应的状态序列记为 $q_1\to q_2\to q_3$, 在通过添加 *ghost* 变量修复 t_1 违反 memoizing 的情况之后, 该语句序列片段变为 $x:=f(x);g_0:=x;x:=f(x)$, 此时, 该片段对应的状态迁移为 $q_1\to q_2\to q_{ghost}\to q_3$, 该状态迁移中状态的信息是完备的. 但新计算得到的状态序列相比原始路径的状态序列多出一个, 这时需要删减掉其中多余的状态, 从而得到原始路径对应的信息完备的状态序列.

带 *ghost* 变量的语句的执行使路径对应的状态迁移多出一个状态, 而该语句的执行效果是将要丢弃的项赋值给 *ghost* 变量进行保存. 本文注意到, 带 *ghost* 变量的语句执行前后的两个状态差别在于: 更新之后的状态中, 要丢弃的项被 *ghost* 变量保存了起来. 因此, 对应到原始路径上, 为了保证路径信息的完备性, 应该保留被带 *ghost* 变量的语句更新之后的状态, 删除更新之前的状态. 在上述的例子中, 状态 q_{ghost} 相比状态 q_2 利用 *ghost* 变量保存丢失项的信息. 故原始语句片段 $x:=f(x);x:=f(x)$对应的信息完备的状态序列为 $q_1\to q_{ghost}\to q_3$. 删除多余状态之后, 路径抽象算法合并状态序列中等价的状态, 从而泛化出循环, 得到不可行原因相同的更多路径.

算法 3 详细描述了本文的路径抽象方法, 算法首先将由所有非 *ghost* 语句计算得到的状态迁移加入 FSA A_t 的迁移中去(第 2 行); 之后, 删除由于添加 *ghost* 语句而多出的状态(第 3 行、第 4 行); 最后, 算法检查所有的状态并将等价状态合并(第 5–7 行). 在泛化过程中, 算法还删除了 A_t 中不可达的状态, 并在不一致状态上添加所有语句的自圈以及在所有状态上添加不相关语句的自圈, 从而提升了路径抽象方法的泛化能力. 算法中的 for 循环对等价状态进行合并(第 5–7 行)时, 可用 map 保存已查询过的状态, 因此, 该路径泛化方法的时间空间复杂度均为 O(n), n 为程序的行数.

算法 3. 路径泛化算法.

输入: 一条不可行的路径 t 和其迁移: $S_0\to_{st_1} S_1\to\ldots\to_{st_m} S_m$, 其中, $t=\langle st_1,\ldots,st_n\rangle$, S_m 是不一致的, 并且 $m\leqslant n$.

输出: FSA A_t 接受与 t 不可行原因相同的路径.

1 $A_t:=(\{\langle st_1,\ldots,st_n\rangle\},\{S_0,\ldots,S_m\},T,S_0,S_m)$

2 $T:=\{(S_i,st_{i+1},S_{i+1})|0\leqslant i\leqslant m-1\wedge st_{i+1}$ 不是一个 *ghost* 语句$\}$

3 $R:=\{(S_i,st,S_j)\in T|st_{j+1}$ 是一个 *ghost* 语句$\}$

4 $T:=(T\cup\{(S_i,st_{i+1},S_k)|(S_i,st_{i+1},S_j)\in R\wedge k=\arg\min_{k>j}(S_k,st_{k+1},S_{k+1})\in T\})\setminus R$

5 **for** $S_i,S_j\in\{S_s,S_t|(S_s,st,S_t)\in T\}\wedge i\neq j$ **do**

6 **if** $S_i=S_j$ **then**

7 $T:=T\cup\{(S_i,\textbf{\textit{skip}},S_j)\}$

8 return A_t

本文的路径抽象方法相对 Hong 等人[2]的基于状态等价的路径抽象方法在两方面做了改进: 一方面是在

路径可行性检查的过程中, 本文通过不添加 *ghost* 变量的方式修复路径违反 early assumes 要求的情况(在算法 2 的第 5 行, 利用ε_t对状态更新), 这种方式减少了修复路径违反 coherence 性质时所需添加的 *ghost* 变量的数量; 另一方面, 本文删除了状态中有关 *ghost* 变量的冗余定义(定义 2 的 $Compact(S_i)$函数). 这两种优化得到了更多的等价状态, 从而提升了路径抽象方法的泛化能力, 因此减少了程序的 CEGAR 过程的精化次数, 提升了验证效率, 在后文第 4.3 节中的实验, 也说明的本文泛化方法的有效性.

4 实验及分析

本文用 Ocaml 对合作验证的框架以及优化的路径抽象方法进行了实现, 并在程序集上对以下几个方面进行了评估.

(1) 有效性: 当给出一组相似的未解释程序时, 合作验证的方法与单独验证的方法相比, 有效性如何.

(2) 稳定性: 合作验证方法在程序的不同验证顺序、正确程序在程序集中所占的比例等因素的影响下, 是否依然有效.

本文所有的实验均在一台 8 核、32 GB 内存的计算机上进行的, 操作系统为 Ubuntu18.04, 每个实验均采用运行 3 次取平均值的方式消除实验误差.

由于合作验证的方法的应用场景是对批量相似的程序的验证, 考虑到没有满足这种特性的标准程序集, 本文以从 SV-COMP[4]提取的程序作为原始程序, 设计生成相似的程序集. 本文考虑了两种代表性的场景: 一是对多版本程序的回归验证, 另一个是对模块化开发的程序集的验证. 这两种场景下的程序之间具有一定的相似性, 已验证程序的中间结果对后续程序的验证具有复用意义. 以下是本文模拟这两种场景设计的相似程序集.

(1) 以回归验证为背景(记为 Benchmark-I)

在软件开发版本迭代的过程中, 开发者通常只改变软件的一部分, 例如漏洞修复、增加功能、优化代码, 而程序的主体框架和功能没有较大的改变, 此时, 两个版本的程序之间具有较高的相似性. 在这样的场景下, 对前一个版本的验证结果, 可以用于对新版本的验证上, 避免重复验证. 为了模拟软件版本迭代, 本文随机抽取 N 个程序作为初始程序, 并在初始程序的基础上进行变异来得到新的程序. 本文设计了 4 种变异方式: 添加新语句、删除当前语句、变量替换、条件取反, 并采用累加变异的方式, 即在变异得到的新程序的基础上继续变异. 累加变异模拟了实际开发中软件版本的迭代过程, 但也存在着变异次数过多会导致新程序与原始程序完全不同的情况. 本文需要根据原始程序的长度, 控制变异的次数 M, 确保变异得到的新程序与原始程序之间的相似度. 因此, 对每个初始程序, 本文对其进行累加变异 M 次获得 M+1 个程序. 在实际实验中, 本文将 N 和 M 分别设置为 10 和 9, 故总共得到 10 组程序集, 每组包含 10 个程序.

(2) 以模块化程序开发为背景(记为 Benchmark-II)

在模块化的程序开发中, 软件是由若干个模块组合而成的, 而这些模块可以作为基本模块支持不同需求的软件开发过程, 例如调用第三方的接口进行开发的过程, 可以对调用的接口的种类及顺序进行调整获得不同功能的程序. 这些程序之间虽然功能各不相同, 但组成结构相似, 彼此之间具有较高的相似性. 在这种场景下, 对批量组合的程序进行验证时, 已验证的程序的泛化结果也可以用在后续待验证的程序上. 为了模拟模块化开发程序的场景, 本文随机选择 10 个程序作为模块程序, 然后随机对这些模块进行组合获得新的程序. 组合的方式包括顺序组合, 或者通过 if-then-else 分支来进行组合. 例如, 对于模块程序 C_1 和 C_2, 可以通过 P= if (⟨*cond*⟩) then C_1 else C_2 组合构造新程序. 在组合的过程中, 每次随机选取 2–4 个模块进行组合, 模块间随机选择顺序结构或者分支结构进行组合. 在实际的实验中, 本文选取不同的模块程序进行组合, 一共得到 6 组程序集, 每组有 1 000 个程序, 其中, 正确程序和错误程序的数量都是 500 个.

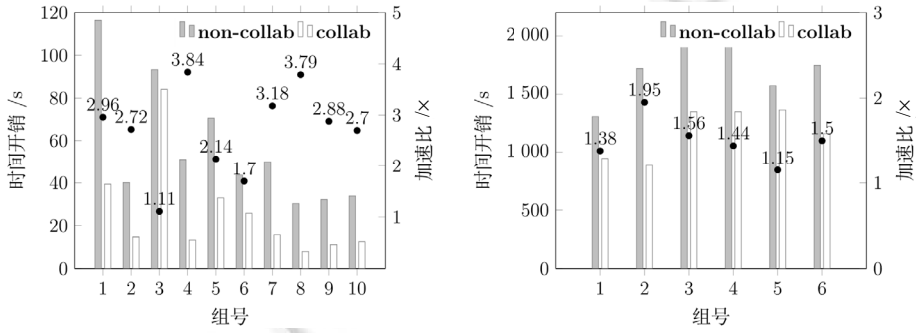
4.1 合作验证方法效率分析

为了评估合作验证方法的有效性, 本节将合作验证的方法(简记为 collab)与单独验证每个程序的方法(记为 non-collab)在验证时间上进行了比较, 在上述的程序集 Benchmark-I 和 Benchmark-II 上分别运行这两种方

法, 每种方法运行 3 次取平均值. 图 7(a)和(b)分别展示了这两个程序集的实验结果, 图中灰色条带代表 non-collab 方法的结果, 白色条带代表 collab 方法的结果, X 轴表示程序集的组索引, 左侧 Y 轴表示验证时间, 右侧 Y 轴表示加速比. 计算加速比的公式如下:

$$speedup=\begin{cases}\dfrac{T(non\text{-}collab)}{T(collab)}, & T(non\text{-}collab)>T(collab)\\ 0, & T(non\text{-}collab)=T(collab)\\ -\dfrac{T(collab)}{T(non\text{-}collab)}, & T(non\text{-}collab)<T(collab)\end{cases} \tag{2}$$

如图 7 中所示, collab 方法在两个程序集上表现都更好. 其中, collab 方法在 Benchmark-I 上取得了 2.70× (1.11×–3.84×)的平均加速, 在 Benchmark-II 取得了 1.49× (1.15×–1.95×)的平均加速. 故可得出结论: 在对相似的批量未解释程序进行验证时, 合作验证的方法相比对程序单独进行验证的方式, 可以提高验证效率.



(a) Benchmark-I 的实验结果 (b) Benchmark-II 的实验结果

图 7 不同方式得到的程序集上的时间开销和加速比

为了进一步探究 collab 方法的有效性, 本节详细分析了对 Benchmark-II (每组包含 1 000 个程序)中的程序验证期间的时间开销变化趋势. 图 8 展示了对这 6 组程序进行验证的结果, 其中, X 轴表示已经验证的程序的数量, Y 轴表示验证的时间, 阴影部分表示这 6 组程序在不同方法下的时间开销的范围, 虚线表示这 6 组程序在 non-collab 方法下时间开销的均值, 实线表示这 6 组程序在 collab 方法下时间开销的均值.

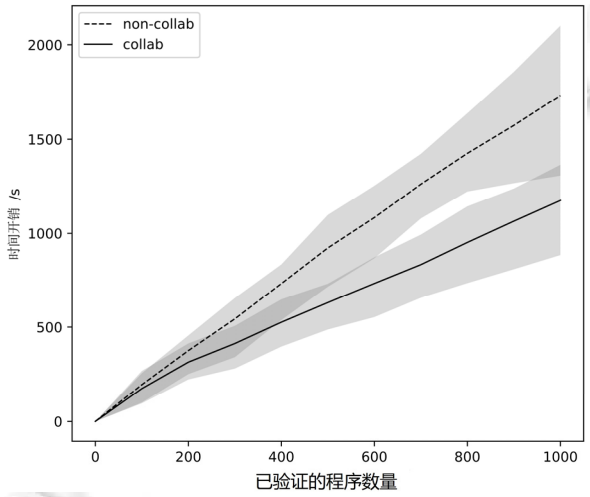


图 8 Benchmark-II 中程序验证时间开销的变化趋势

通过观察, 有以下几个结论.

(1) 在验证刚开始的阶段, collab 方法的性能表现与 non-collab 方法区别不大, 在某些情况下, non-collab 方法的效果甚至更好. 这主要由于在验证刚开始的阶段, 收集不可行路径的抽象模型的开销占比较大, 而收集的抽象模型还比较少, 导致对新程序进行验证时复用之前的验证结果带来的收益较少, 因此在验证刚开始的阶段, collab 方法效果不明显甚至会较差.

(2) 随着收集的抽象模型越来越多, 验证新的程序时可以复用的结果也更多, 复用避免了重复验证, 使得验证任务减轻, 带来的收益比收集不可行路径的抽象模型的开销更大. 因此, collab 方法的效果随着验证任务规模的增大而变得越来越明显.

故在对批量相似的未解释程序进行验证时, 合作验证的方法与程序单独验证的方式相比, 可以提高验证效率, 当验证的程序数目较多时, 效果更加明显.

4.2 合作验证方法稳定性分析

在对批量未解释程序验证时, 有很多因素会对合作验证方法的效率造成影响, 例如程序集中正确程序的占比和程序集中程序的不同验证顺序. 探究正确程序在程序集中的占比, 是因为验证程序为正确和错误时情况有所不同: 验证程序为错误时, 只需找到一条错误路径就可证明程序错误, 验证过程中止; 验证程序正确需要验证程序所有的路径都是正确的. 探究程序的不同验证顺序是因为验证顺序不同, 验证过程中合作验证方法收集的抽象模型会有不同, 每个程序复用的之前程序的验证结果也会有所不同, 从而影响合作验证方法的效率. 因此, 本文探究了不同正确程序占比和不同程序验证顺序对合作验证方法的影响. 下面介绍相应的实验设置与实验结果.

(1) 正确程序占比的影响

为了探究正确程序在程序集中不同占比对合作验证方法的效率的影响, 本节以 Benchmark-II 中的程序集为基础, 构造了不同正确程序占比的程序集, 分别为 0%, 20%, 40%, 60%, 80%和 100%, 每组包含程序 500 个. 之后, 在该程序集上对比了 collab 的方法和 non-collab 的方法的运行时间, 每种方法运行 3 次求平均值. 图 9 展示了实验的结果, 其中, X轴代表正确程序的占比, 左侧 Y轴代表验证的时间开销, 右侧 Y轴代表加速比. 可以看到, collab 方法在 40%, 60%, 80%和 100%等正确程序占比下取得了更好的性能, 并且正确程序的占比越高, 合作验证的方法效果就越好. 在全部为正确程序的程序集中, collab 方法取得了 2.25×的加速.

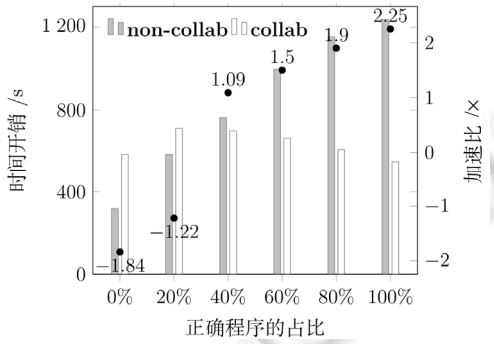


图 9 不同正确程序占比的程序集验证的时间开销和加速比

以下分析说明了 collab 方法在 0%和 20%的占比下表现不佳的原因: 当验证程序为不正确时, 找到一条错误路径就证明程序是错误的, 没必要将全部路径验证完毕. 而 collab 方法在对新程序进行验证时, 会利用在 CEGAR 过程中保存的不可行路径的抽象模型来对新程序进行精化, 这会导致 collab 方法相比 non-collab 方法验证了更多的路径. 当错误路径在较少次迭代就能找到时, collab 方法的时间开销就会大于 non-collab 方法的时间开销. 在上述的程序集中, 错误程序的错误路径需要较少次迭代就可以找到, 因此在错误程序占比较高的程序集上, collab 方法表现不佳; 但在大多数情况下, collab 方法都比 non-collab 方法取得了更好的效果.

(2) 程序不同验证顺序的影响

为了探究程序集中程序的验证顺序对合作验证方法的影响, 本节以 Benchmark-II 中的程序集为基础, 打乱其顺序, 得到了 6 组、每组 10 种不同顺序的程序集. 之后, 在该程序集上对比 collab 方法和 non-collab 方法的运行时间. 与图 8 类似, 图 10 展示了每组 10 种不同顺序的程序集验证时间的平均结果, 阴影部分代表了这 10 种验证顺序的时间开销的范围, 实线代表 collab 方法的均值, 虚线代表 non-collab 方法的均值. 可以看到, 不同的验证顺序确实对 collab 方法的效率产生了影响. 因为在程序的不同验证顺序下, 保存的不可行路径的抽象模型是不同的, 从而同一个程序在不同的验证顺序下, 通过保存的抽象模型精化掉的部分是不一样的. 但 collab 的时间开销在不同的顺序下基本保持稳定, 并且在所有的程序集上 collab 方法的验证效率是优于 non-collab 方法的.

因此, 合作验证方法在大部分的正确程序占比程序集上和不同验证顺序的程序集上都保持了有效性, 说明了合作验证方法在不同正确程序占比和不同验证顺序下的稳定性.

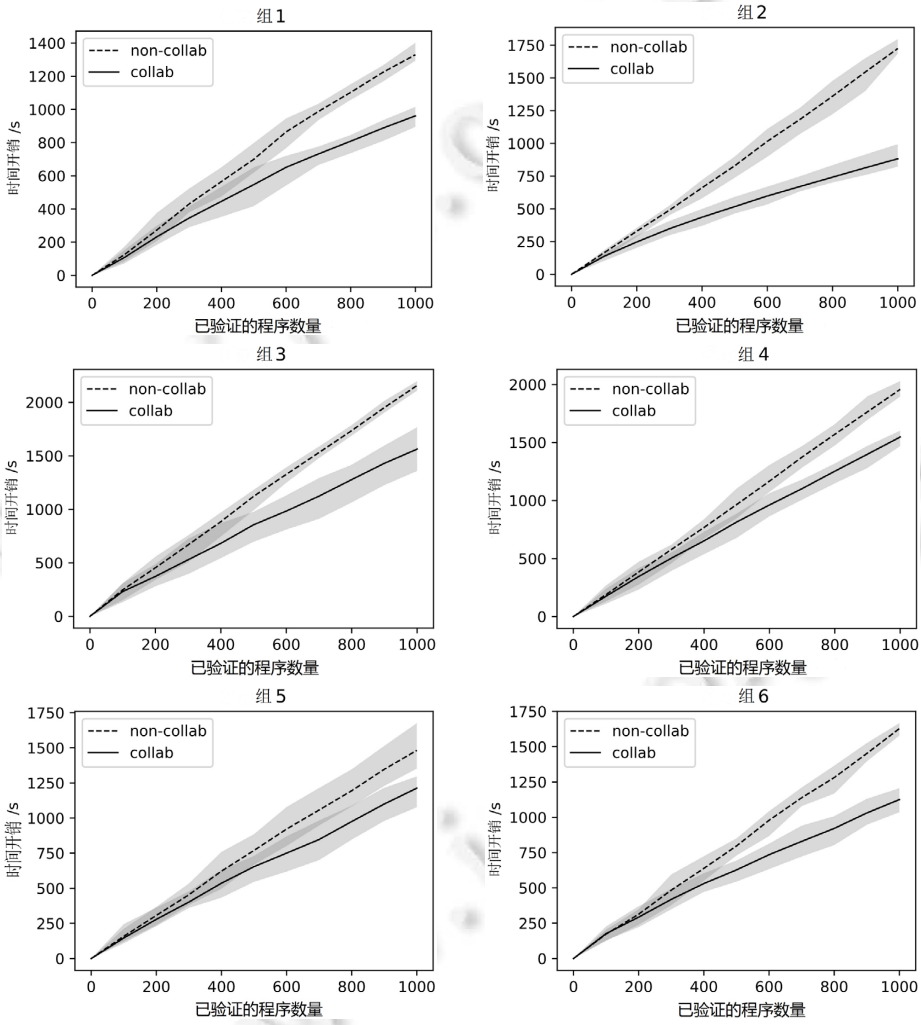


图 10 Benchmark-II 中程序的不同验证顺序下的时间开销变化趋势

4.3 泛化方法有效性分析

在第 3 节中, 本文对基于状态等价的路径泛化方法[2]进行了改进. 为了评估改进的泛化方法的有效性, 本

节选择没有复用中间验证结果的工具版本(记为 non-collab)和 Hong 等人[2]提出的路径抽象方法(简记为 pure)进行了对比, 测试的程序集采用 Hong 等人[2]工作的实验数据集, 其中, 总共 50 个未解释程序, 每种方法运行 3 次取平均值. 表 2 展示了评估路径抽象方法效率的实验结果, 表中列出了每个被验证程序的行数以及每种工具的实验结果, 包括程序的正确性、验证时间以及迭代次数, 其中, TO 代表超时, 灰色格子表示对应的工具表现更好. 可以看到, 在总共 50 个程序中, 本文的方法完成了所有的验证任务(100%), 而 pure 完成了 46 个验证任务(92%). 在时间开销方面, 本文的方法在所有的程序中都比 pure 方法好得多. 在对程序进行验证时的迭代次数方面, 本文的方法在所有的正确程序上的迭代次数都少于 pure 方法的迭代次数, 总体上, 本文的方法减少了 19.8%的迭代次数. 在一些不正确的程序上, 本文的方法需要更多的迭代次数, 这是因为两种方法在实现上有所不同导致的.

表 2 评估路径抽象方法效率的实验结果

程序名	代码行数	pure			non-collab		
		结果	时间	迭代次数	结果	时间	迭代次数
benchmark0	41	incorrect	0.791	0	incorrect	0.008	1
benchmark1	43	correct	2.555	18	correct	0.027	11
benchmark2	54	correct	221.634	108	correct	1.927	85
benchmark3	49	correct	169.123	90	correct	0.113	65
benchmark4	41	incorrect	0.778	0	incorrect	0.001	0
benchmark5	46	correct	11.597	36	correct	0.309	30
benchmark6	42	correct	3.369	24	correct	0.035	17
benchmark7	52	correct	161.052	90	correct	0.206	60
benchmark8	44	incorrect	0.781	0	incorrect	0.004	1
benchmark9	40	incorrect	0.745	0	incorrect	0.001	0
benchmark10	46	incorrect	0.799	0	incorrect	0.011	3
benchmark11	54	correct	318.596	108	correct	0.153	85
benchmark12	37	incorrect	0.733	0	incorrect	0.006	1
benchmark13	39	incorrect	0.756	0	incorrect	0.002	1
benchmark14	50	TO	TO	N/A	correct	0.205	80
benchmark15	41	incorrect	0.769	0	incorrect	0.002	1
benchmark16	40	correct	3.504	20	correct	0.016	18
benchmark17	47	incorrect	0.781	0	incorrect	0.013	6
benchmark18	46	incorrect	0.814	0	incorrect	0.005	4
benchmark19	49	TO	TO	N/A	correct	0.101	54
benchmark20	37	incorrect	0.747	0	incorrect	0.007	2
benchmark21	39	incorrect	0.744	0	incorrect	0.001	1
benchmark22	48	incorrect	0.791	0	incorrect	0.031	0
benchmark23	54	correct	164.031	81	correct	0.154	67
benchmark24	41	incorrect	0.770	0	incorrect	0.004	1
benchmark25	44	incorrect	0.750	0	incorrect	0.001	0
benchmark26	47	correct	15.541	36	correct	0.483	32
benchmark27	48	incorrect	0.817	0	incorrect	0.001	0
benchmark28	45	incorrect	0.790	0	incorrect	0.013	1
benchmark29	45	correct	59.844	60	correct	0.100	45
benchmark30	57	TO	TO	N/A	correct	0.105	51
benchmark31	39	incorrect	0.750	0	incorrect	0.001	0
benchmark32	41	incorrect	0.778	0	incorrect	0.002	1
benchmark33	46	incorrect	0.774	0	incorrect	0.001	1
benchmark34	55	correct	289.730	108	correct	0.132	78
benchmark35	51	incorrect	0.802	0	incorrect	0.096	23
benchmark36	43	incorrect	0.780	0	incorrect	0.004	1
benchmark37	41	incorrect	0.774	0	incorrect	0.004	2
benchmark38	49	correct	16.094	36	correct	0.058	28
benchmark39	47	correct	13.007	36	correct	0.071	28
benchmark40	42	correct	5.095	24	correct	0.079	16
benchmark41	41	incorrect	0.781	0	incorrect	0.001	1
benchmark42	22	correct	0.730	3	correct	0.001	2
benchmark43	46	incorrect	0.798	0	incorrect	0.001	0
benchmark44	40	correct	3.708	20	correct	0.001	15
benchmark45	53	correct	191.292	90	correct	0.035	67
benchmark46	28	correct	1.060	9	correct	0.102	7
benchmark47	46	correct	11.532	36	correct	0.009	29
benchmark48	53	TO	TO	N/A	correct	0.027	135
benchmark49	44	correct	39.776	50	correct	0.525	32

5 相关工作

在对程序进行验证时, 合作不同的工具可以显著提高验证效率. Bodden 等人[5]提出不同的用户之间进行合作来对大型的软件进行运行时验证. Mitsch 等人[6]给出了一个面向大型混成系统的验证工具集的愿景, 展示了合作验证方法在对大型软件验证时的巨大潜力. 还有许多方法[7−9]联合不同的验证工具, 根据不同验证工具的特性进行合作, 提升了验证的效率与验证断言的能力, 这些方法都取得了良好的效果. 上述的方法都是面向单个程序的验证, 而本文的工作主要针对批量程序的验证. 一个典型的场景例如回归验证, 利用之前版本的验证结果对后续的程序版本进行精化是一个直观的想法. Beyer 等人[10]复用验证过程中的抽象精度, 而对以自动机表示的状态空间型[11,12]、函数摘要[13]、过程摘要和循环摘要[14]、谓词分析中的断言[15]和反例路径的复用[9], 都极大地提高了回归验证的效率. 在 Rothenberg 等人[16]的工作中, 在对程序的基于路径抽象的验证中,复用验证过程中的抽象模型, 避免不同版本间的重复验证.

在未解释程序的验证相关工作方面, 未解释程序的验证问题通常是不可判定的[1]. 而 Mathur 等人提出了一个称为 coherent 的可判定的程序类[1], 后续有工作[17−19]将这一结果推广到了其他类型的程序上. Krogmeier 等人[20]在这个结果的基础上提出了一种综合满足给定规约的 coherent 程序的方法. 在 coherent 的程序可判定的基础上, 针对一般的未解释程序, Hong 等人[2]提出了基于路径抽象的 CEGAR 验证框架, 并提出了基于状态等价的路径抽象方法. 这种方法抽象出了路径不可行的核心原因, 因此, 与基于插值的路径抽象方法[3]相比更加高效. 本文的工作在此方法的基础上进行了改进, 取得了更好的效果.

6 总结与展望

本文提出了一个面向未解释程序的合作验证的框架, 该框架在对批量未解释程序进行验证时, 收集验证过程中不可行路径的抽象模型, 在对新程序验证时, 复用收集的抽象模型, 避免重复验证, 从而提升验证效率. 另外, 在对未解释程序进行 CEGAR 迭代验证时, 不可行路径的路径抽象方法对验证的效率有直接的影响. 本文提出了一种细粒度的路径抽象方法, 根据路径违反 coherent 的不同原因对路径进行修复, 并对路径状态中的信息进行精简, 保留路径不可行的核心原因. 本文对合作验证的框架和路径抽象方法的优化进行了实现, 并在相应的程序集上进行了实验, 在两个具有代表性的程序集上分别取得了 2.70×和 1.49×的加速. 本文还评估了合作验证方法在不同影响因素下的稳定性以及改进的泛化方法的有效性, 取得了良好的实验结果.

合作验证的方法在对批量相似的程序进行验证时会有显著效果, 但由于路径泛化方法仍然存在一定的局限性, 因此在对一些程序进行验证时算法会不终止, 导致路径爆炸问题的产生. 并且对程序相似度的语义还有待形式化定义, 后续会寻找合适的相似度度量方式, 例如基于程序文本、抽象语法树、程序依赖图或者控制流图等程序相似度的度量方式[21,22], 并分析程序之间的相似度对合作验证方法效率的影响. 另外, 本文在未解释程序上实现了合作验证的方法, 下一步会将合作验证的方法推广到实际程序上, 将对实际的相似程序进行合作验证, 并且会在更多的验证工具上考虑合作验证方法的可行性. 另外, 在对改进的泛化方法效率进行评估时, 本文只关注了改进的路径抽象方法在对程序验证效率方面的提升, 后续会详细评估方法在对不可行路径进行抽象时, 对不同违反 coherent 原因的路径的泛化效果的提升, 以及方法在减少 ghost 变量的添加方面的效果.

References:

[1] Mathur U, Madhusudan P, Viswanathan M. Decidable verification of uninterpreted programs. In: Proc. of the ACM on Programming Languages (POPL). 2019. Article 46.

[2] Hong W, Chen Z, Du Y, *et al*. Trace abstraction-based verification for uninterpreted programs. In: Proc. of the Int'l Symp. on Formal Methods. Cham: Springer, 2021. 545−562.

[3] Heizmann M, Hoenicke J, Podelski A. Refinement of trace abstraction. In: Proc. of the Int'l Static Analysis Symp. Berlin, Heidelberg: Springer, 2009. 69−85.

[4] SV-Benchmarks. 2022. https://github.com/sosy-lab/sv-benchmarks

[5] Bodden E, Hendren L, Lam P, *et al*. Collaborative runtime verification with tracematches. Journal of Logic and Computation, 2008, 20(3): 707−723.

[6] Mitsch S, Passmore GO, Platzer A. Collaborative verification-driven engineering of hybrid systems. Mathematics in Computer Science, 2014, 8(1): 71−97.

[7] Christakis M, Müller P, Wüstholz V. Collaborative verification and testing with explicit assumptions. In: Proc. of the Int'l Symp. on Formal Methods. Berlin, Heidelberg: Springer, 2012. 132−146.

[8] Csallner C, Smaragdakis Y. Check '*n*' Crash: Combining static checking and testing. In: Proc. of the 27th Int'l Conf. on Software Engineering. 2005. 422−431.

[9] Beyer D, Wendler P. Reuse of verification results. In: Proc. of the Int'l SPIN Workshop on Model Checking of Software. Berlin, Heidelberg: Springer, 2013. 1−17.

[10] Beyer D, Löwe S, Novikov E, *et al*. Precision reuse for efficient regression verification. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. 2013. 389−399.

[11] Beyer D, Holzer A, Tautschnig M, *et al*. Information reuse for multi-goal reachability analyses. In: Proc. of the European Symp. on Programming. Berlin, Heidelberg: Springer, 2013. 472−491.

[12] Lauterburg S, Sobeih A, Marinov D, *et al*. Incremental state-space exploration for programs with dynamically allocated data. In: Proc. of the 30th ACM/IEEE Int'l Conf. on Software Engineering. IEEE, 2008. 291−300.

[13] Sery O, Fedyukovich G, Sharygina N. Incremental upgrade checking by means of interpolation-based function summaries. In: Proc. of the Formal Methods in Computer-aided Design (FMCAD). IEEE, 2012. 114−121.

[14] He F, Yu Q, Cai L. Efficient summary reuse for software regression verification. IEEE Trans. on Software Engineering, 2020, 48(4): 1417−1431.

[15] Yu Q, He F, Wang BY. Incremental predicate analysis for regression verification. In: Proc. of the ACM on Programming Languages (OOPSLA). 2020. Article 184.

[16] Rothenberg BC, Dietsch D, Heizmann M. Incremental verification using trace abstraction. In: Proc. of the Int'l Static Analysis Symp. Cham: Springer, 2018. 364−382.

[17] Mathur U, Murali A, Krogmeier P, *et al*. Deciding memory safety for single-pass heap-manipulating programs. In: Proc. of the ACM on Programming Languages (POPL). 2019. Article 35.

[18] La Torre S, Parthasarathy M. Reachability in concurrent uninterpreted programs. In: Proc. of the 39th IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science. 2019. Article No.46.

[19] Mathur U, Madhusudan P, Viswanathan M. What's decidable about program verification modulo axioms? In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2020. 158−177.

[20] Krogmeier P, Mathur U, Murali A, *et al*. Decidable synthesis of programs with uninterpreted functions. In: Proc. of the Int'l Conf. on Computer Aided Verification. Cham: Springer, 2020. 634−657.

[21] Ragkhitwetsagul C, Krinke J, Clark D. A comparison of code similarity analysers. Empirical Software Engineering, 2018, 23(4): 2464−2519.

[22] Xu X, Liu C, Feng Q, *et al*. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security. 2017. 363−376.

杜一德(1995－), 男, 硕士, 主要研究领域为程序验证.

陈振邦(1981－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为程序分析, 形式化方法及其应用.

洪伟疆(1995－), 男, 博士生, 主要研究领域为程序验证.

王戟(1969－), 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.