基于变量符号关联分析的程序状态优化方法

郭曦[1]，王盼[2]

（1. 华中农业大学信息学院，湖北 武汉 430070；2. 武汉电力职业技术学院电力工程系，湖北 武汉 430079）

摘 要：程序分析是主要的程序属性分析方法，在变量依赖关系、路径覆盖率、测试用例约简等方面有广泛的应用，并取得了大量研究成果。目前，程序分析主要以符号执行工具为核心，但是普遍存在路径条件逻辑表达式难以准确生成和约束求解器性能不够高的问题，从而影响程序分析的效果。以提高路径分析精度为目标，首先分析不同执行路径对应的路径条件，并提取公共的符号表达式以提高符号关联分析的精度，然后逆向生成依赖条件逻辑表达式集合，使用依赖关联分析算法以提高路径分析的精度。实验结果表明，所提方法相对于传统的路径分析方法，有更准确的时间复杂度和更高的路径分析精度。

关键词：程序分析；符号执行；约束求解器；符号分析

中图分类号：TP311

文献标识码：A

doi: 10.11959/j.issn.1000-436x.2018094

Program state optimal method based on variable symbolic relation analysis

GUO Xi[1], WANG Pan[2]

1. College of Informatics, Huazhong Agriculture University, Wuhan 430070, China
2. Department of Power Engineering, Wuhan Electric Power Technical College, Wuhan 430079, China

Abstract: Program analysis is the prime method to program property analysis, which is widely used in the domain of parameter dependent relation, path coverage and test case generation, and a lot of progress has been made. Current program analysis is based on the method of symbolic execution, but symbolic execution is usually tackled with the problems of logic expression generation of path condition and low efficiency of constrain solver, which will affect the results of program analysis. Aiming at enhancing the path analysis efficiency, the path conditions of different paths were collected, the common symbolic expression was extracted and the efficiency of symbolic analysis was enhanced, then the logic expression set was generated, the dependent relation algorithm was used to enhance the efficiency of symbolic analysis. Experimental results demonstrate that the proposed method has the advantages of accurate time complexity and better analysis efficiency compare to traditional program analysis method.

Key words: program analysis, symbolic execution, constrain solver, symbolic analysis

1 引言

对程序的属性和行为进行精确分析是研究人员的主要目标，然而随着程序规模的增大，往往导致程序状态空间也在不断增大，从而严重影响程序分析的效果。符号执行作为广泛使用的程序分析方法，在路径覆盖、分支覆盖、测试用例生成等方面发挥着重要的作用，但是由于其长期受制于路径条件提取精度以及约束求解器性能，极大限制了它的分析效果。

为了提高路径分析的精度，研究人员采用多种

收稿日期：2017-07-17；**修回日期**：2018-04-02

通信作者：郭曦，xguo@mail.hzau.edu.cn

基金项目：国家自然科学基金资助项目（No.61502194）；中央高校基本科研业务费专项基金资助项目（No.2662018JC028）

Foundation Items: The National Natural Science Foundation of China (No.61502194),The Central University Basic Business Expenses Special Funding for Scientific Research Projects (No.2662018JC028)

分析方法从不同角度进行研究，其中，状态优化合并是常用的分析方法，它通过在程序中增加临时变量[1]来区分不同路径对应的符号输入，但是这种分析方法会增加约束求解器的计算开销，并且可能导致约束求解器停止工作而产生误报或漏报。目前，约束求解器能够较好地求解线性逻辑表达式，但是对程序中常见的复杂变量类型和数据结构则难以有较好的分析精度。另外，约束求解器本身的调用开销较大，为了使其能够较长时间工作，往往会删除一些难以求解的逻辑结构，从而导致分析的结果不够完备。

本文主要对程序中的符号变量状态进行研究，通过提取不同路径中的公共逻辑表达式，构建新的符号表示方法，另外，对于逻辑表达式产生过程中存在的条件合并而导致的条件丢失问题，采用改进的路径重构算法，从而提高路径条件求解的效率。本文的主要贡献如下。

1) 把符号变量和路径条件进行合并分析，从而对符号的表示形式进行优化。

2) 对符号执行过程中基本块之间存在的关联关系进行分析，并对路径条件进行优化重构，从而减少路径条件的丢失。

3) 量化分析程序数量的边界，从而可以准确地分析程序的复杂度。

本文首先介绍程序分析的相关研究现状和基本概念，然后详细分析状态合并的不足以及采用的状态合并方法和变量间的关联分析方法，并分析路径执行数量的边界，最后通过实验分析验证本文所提方法的有效性。

2 相关研究进展

程序分析是验证程序性质、提高程序可靠性等需求的必需工作，在程序分析过程中广泛使用的符号执行分析方法以其特有的工作过程，在路径可达分析、测试用例生成等领域有重要的研究价值。其主要工作方式为将变量以符号的形式进行输入，来代替具体的输入值，这种静态的分析方法相对于动态分析方法来说可以获得更为底层的程序逻辑关系，从而可以更好地分析程序的性质。但是程序规模的变化对于符号执行的分析效果有直接的影响，主要体现在程序变量类型的多样化和数据结构的复杂化，这使符号执行在提取路径表达式的过程中难以精确地刻画路径条件，同时由于约束求解器本身性能的限制，其对函数调用、数组下标等复杂数据结构，可能因为难以准确求解而删除部分逻辑表达式，从而影响分析精度或产生误报。

为了提高程序分析的性能，目前主要采用状态合并[2]的分析方法来处理不同路径对应的逻辑表达式，并采用多种分析策略来研究合并点处的逻辑符号[2-5]。文献[2]使用传统的辅助变量来分析程序在合并点的状态，并采用函数摘要的分析方法产生测试集。文献[3]使用状态合并和符号执行的分析方法研究程序的性质，但是需要对程序进行预处理，以消除函数调用和语句跳转，才能进行状态合并操作。文献[4]使用删除冗余路径的不可达分析方法对程序状态进行处理。这几种分析方法的共同特点是都使用了临时符号变量，从而进一步影响约束求解器的分析效果。文献[5]采用不引入临时变量的分析方法，但是该方法的使用范围只能局限于具有相同数据结构的程序。对于程序分析过程中常见的冗余路径和不可达路径，目前的分析方法主要通过比较当前状态和历史状态，或将状态空间划分为规模较小的子空间进行单独求解。常见的分析工具（如JPF[6]）通过状态匹配的分析方法来减少状态空间，文献[7]则采用读写集的分析方法来优化状态匹配条件。函数摘要[8-9]可以对程序的函数调用接口进行分析，作为一种静态分析方法，它可以与其他程序优化方法进行协同工作，如可达分析、指向分析等，但是这些方法存在分析精度较低等问题。文献[10]将变量符号转换为守卫条件表达式，从而分析程序的可达性，但是该方法没有对路径条件顺序进行优化分析，存在漏报的情况。文献[11]依据程序的输出结果对程序的执行路径进行分类，但是该方法存在路径条件未用逻辑符号表达式的形式表示的问题，故约束求解器在处理过程中容易终止分析。对于程序上下文执行语义中存在的恶意代码，文献[12]统计缺陷的种类和与之相关的路径集合，从而生成能够发掘恶意代码的测试用例集。文献[13]使用抽象精化的形式化方法对符号变量进行分析，产生可以同时分析多个目标状态的序列。文献[14]采用条件约束、最短路径等分析方法，推迟程序变量的实例化操作从而减少路径状态的扩大。

由于变量符号状态在合并操作过程中普遍存在临时变量，从而加重求解器的性能开销，同时约束求解器在求解过程中对于路径条件表达式的处理性能不够强大而导致分析精度下降，对于这些问

题，本文在符号执行的分析基础上，将目标符号和对应的约束条件作为整体进行协同表示，从而不引入临时变量。在约束求解器的工作阶段，采用流敏感的上下文分析方法，提取对目标变量有重定义操作的语句集合，然后逆向分析到程序的入口，对于该过程收集到的逻辑表达式，本文提出路径条件重构的分析方法，减少因求解器删除逻辑表达式而产生的误报和漏报。本文方法相对于传统的程序分析方法，可以避免临时变量的引入，同时可以提高约束求解器的分析性能。

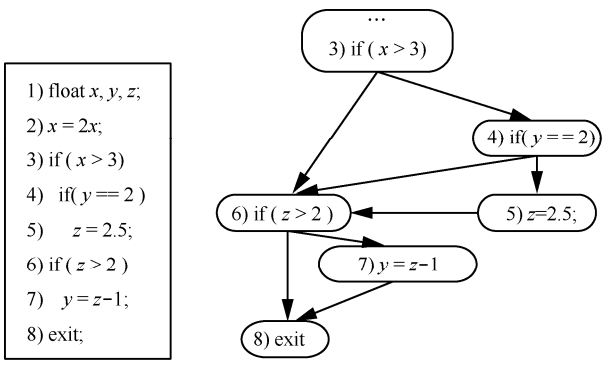
3　路径条件和关联分析

符号执行作为主要的程序分析工具，已经在可达分析、路径覆盖、分支覆盖等方向有了广泛的应用。其核心思想是将变量符号作为程序的输入，符号化地执行程序，从而验证程序的性质。

定义 1　路径条件。记程序 P 和一组对应的输入 t，t 对应的程序路径记为 π，路径条件 φ 为 π 中所有分支语句对应的判定条件组成的逻辑表达式。

符号执行在工作过程中，会沿着当前执行路径记录分支语句对应的判定条件，将其作为当前输入对应的路径条件。后续新的程序输入如果能够满足该路径条件对应的逻辑表达式，则该新的输入也会驱使程序运行到相同的语句位置。路径条件 φ 可以表示为 $\varphi=\varphi_1\vee\varphi_2\vee\varphi_3\vee\varphi_4\vee\varphi_5$，其中，$\varphi_i$ 为一个分支语句对应的逻辑表达式。

符号执行树如图 1 所示。符号执行在工作过程中需要记录并更新变量的符号状态，主要包括变量对应的符号以及对应的路径条件 φ。对于图 1(a)中的程序，符号执行将 x_0、y_0、z_0 作为程序的输入。表 1 列举了图 1(a)中的程序对应的所有状态。



(a) 程序片段　　(b) 符号执行树

图 1　程序片段及对应的符号执行树

在程序当前执行路径中，新加入的路径分支条件会更新当前的符号状态，同时也会更新变量的符号值。为了发掘新的执行路径，常用的方法是将某一个路径条件对应的逻辑表达式进行取反，从而获得不同的路径条件逻辑值，以驱使程序沿着另外的分支运行。由于程序中变量类型的多样化和复杂化，表 1 中 φ_5 所对应的示例程序中的语句包含浮点类型的变量，对于该类型的变量，目前的约束求解器的分析效果难以满足程序性质研究的需要，故一般的约束求解器在工作过程中，会采取近似值或删除对应的逻辑条件表达式来确保尽可能长的求解过程，这样的结果必然会影响程序分析的精度，类似的情况还包括函数调用和数组下标等情形。为了应对这种问题，本文将输入变量的符号和对应的路径条件进行协同处理，从而能够减少因逻辑表达式删除而产生的约束求解器中断或误报等情形。

表 1　　**符号执行状态集合**

路径 π	当前路径条件 φ	x	y	z
1,2,3,6,8	$\varphi_1=2x_0\leqslant 3\wedge z_0\leqslant 2$	$2x_0$	y_0	z_0
1,2,3,6,7,8	$\varphi_2=2x_0\leqslant 3\wedge z_0>2$	$2x_0$	z_0-1	z_0
1,2,3,4,6,8	$\varphi_3=2x_0>3\wedge y_0\neq 2\wedge z_0\leqslant 2$	$2x_0$	y_0	z_0
1,2,3,4,6,7,8	$\varphi_4=2x_0>3\wedge y_0\neq 2\wedge z_0>2$	$2x_0$	z_0-1	z_0
1,2,3,4,5,6,7,8	$\varphi_5=2x_0>3\wedge y_0=2$	$2x_0$	1.5	2.5

本文采用的分析策略包含后向分析方法，由于后向分析通常以程序的出口语句或指定语句为起点，逆向分析到程序的入口语句，在该过程中可以提取与出口语句或指定语句中符号变量相关的逻辑表达式集合。

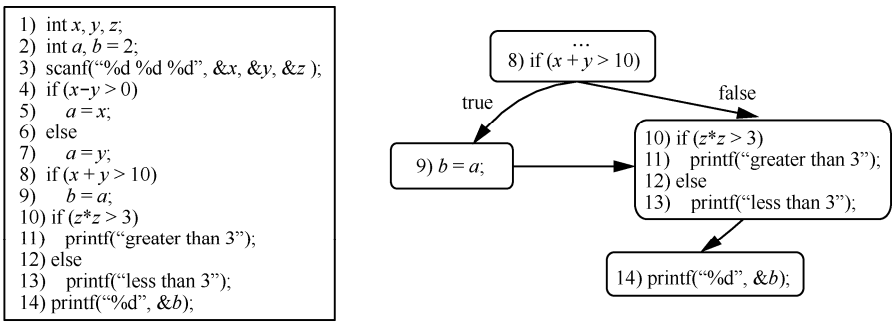
由于程序中的条件嵌套语句可以转化为不带嵌套的形式，为了描述方便，采用图 2(a)所示的不带条件嵌套的程序进行说明。图 2(a)所示程序的目标语句中变量 b 在不同路径条件 φ 的作用下有不同的符号值。

1) $(x-y>0)\wedge(x+y>10)\mapsto(b==x)$。

2) $\neg(x-y>0)\wedge(x+y>10)\mapsto(b==y)$。

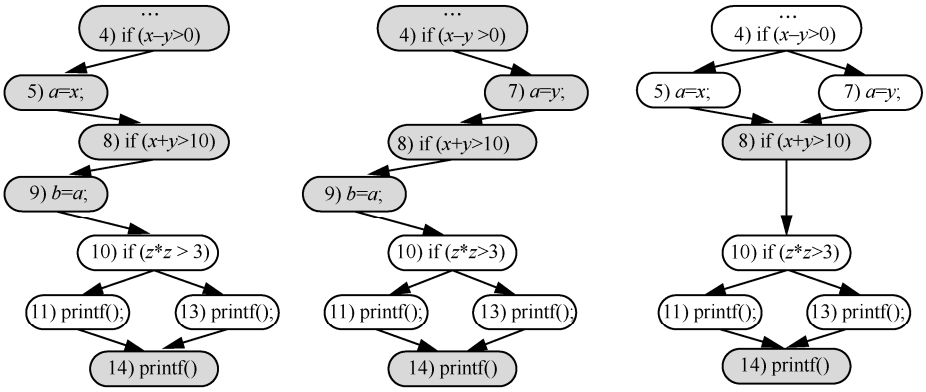
3) $\neg(x+y>10)\mapsto(b==2)$。

这种符号化的表示方法在一定程度上可以简化程序的可达路径集合，对于图 2(a)中程序的目标语句 14)中的变量 b，与 b 相关的符号表达式可以表示该程序所有的 8 条可达路径，如图 3 所示。根据程序的输出符号，可以将可达路径进行划分，这样就可以进一步分析符号变量在不同可达路径之间



(a) 程序片段　　(b) 关联关系分析

图 2　隐式依赖分析



(a) b==x　　(b) b==y　　(c) b==2

图 3　基于依赖条件的路径划分

的关联关系。

对于程序的执行路径中的 2 个节点 b_1 和 b_2，设目标语句中的变量符号为 v，若 b_2 与 b_1 之间是直接依赖关系，则存在如下可能：b_1 对 v 进行了赋值运算等操作；b_2 中存在使用 v 的语句。这种依赖关系是程序分析过程中较普遍的关系，但是在实际分析过程中，还存在间接依赖关系，下面给出其定义。

定义 2　间接依赖。设 s 为程序 P 中的目标语句，P 存在一条可达路径 π，s'为 π 中的一条语句，φ 为其对应的路径条件，若 s 与 s'存在间接依赖关系，则有如下条件成立。

1) s 和 φ 有相同的路径条件。

2) s 中的某个变量 v 在当前执行路径 π 中不存在赋值操作，但该变量在 s 与 s'之间的其他路径中被重新赋值。

依据定义 2，图 2(b)中语句 8)和语句 14)之间存在间接依赖[12]关系，即语句 14)中的变量 b 在当前可达路径[8,10,11,12,13,14]里无赋值操作，而在另一条可达路径[8,9,10,11,12,13,14]里存在赋值操作。

由于依赖条件分析同时具备目标语句与程序输入的符号信息，故可以根据目标语句的符号值将优化后的符号执行树的路径划分为若干组。图 2(a)所示程序经路径划分后的结果如图 3 所示，程序所有的 8 条执行路径被划分为 3 组，每组具有不同的符号输出表达式。

在路径条件分析过程中，通过修改最新加入的路径条件表达式的逻辑值来发掘新的路径集合，间接依赖分析可以把程序的依赖条件和输入符号与目标语句中的变量进行关联分析，若存在可解的路径条件，则可以从路径集合中删除该路径条件。对于图 2(a)中的程序，表 2 为程序输入 x=6, y=2, z=2 时的路径分析过程。

表 2　当程序输入为 x=6, y=2, z=2 时的路径分析过程

步骤	输入	路径	依赖条件
1)	{x=6, y=2, z=2}	[$p_{1,t}$, $p_{2,f}$, $p_{3,t}$]	¬(x+y>10)
2)	{x=6, y=5, z=2}	[$p_{1,t}$, $p_{2,t}$, $p_{3,t}$]	(x−y>0)∧(x+y>10)
3)	{x=6, y=2, z=2}	[$p_{1,t}$, $p_{2,f}$, $p_{3,t}$]	¬(x+y>10)
4)	{x=2, y=6, z=2}	[$p_{1,f}$, $p_{2,f}$, $p_{3,t}$]	¬(x+y>10)

注意到表 2 中的间接依赖分析，路径列中缺乏

路径条件$\neg(x-y>0)\wedge(x+y>10)$对应的程序执行路径。若该路径表达式可解，则当前的路径可达。但是目前的约束求解器在工作过程中为了保持工作的持续性，可能会删除或合并某些逻辑表达式，这将会产生误报或漏报现象。这种自行删除的逻辑表达式往往蕴含丰富的逻辑关系，故第 4 节将详细分析路径条件优化的方法。

4　符号条件重构算法

在第 3 节的程序分析过程中，常见的约束求解器对于线性逻辑表达式有较好的分析效果，但是对于较为复杂的数据结构，为了减少约束求解器的调用次数，求解器会删除难以分析的表达式，而删掉的表达式可能包含关键的路径信息，导致路径分析的结果不够精确。另外，在路径发掘过程中，新增的路径条件可能导致约束表达式隐式删除，原因是在依赖条件分析过程中产生的变量符号与已有的变量符号一致，则这种新增的符号会被删除，而被删除的符号与新增的路径有关联关系，这样会影响求解器的分析精度。本文采用改进的路径表达式分析方法，提取不同路径在目标语句处的符号表达式，这样可以不引入辅助变量；对于已有的逻辑表达式集合，本文使用优化算法进行处理，从而减少因逻辑表达式的删除而带来精度上的损失。

4.1　符号签名

采用符号执行的程序分析方法在收集变量逻辑表达式的过程中会保存变量的符号表达式，新执行路径的生成会导致目标语句中的变量符号有多个表达式，故通常使用辅助变量来进行分析。图 1(a)语句 6)中的变量 z 在不同路径中对应的符号表达式为$(z_0, 2.5)$，故常规的符号分析方法会引入临时变量 z'来进行分析，在该处状态合并过程中，变量 z 对应的路径条件为$(2x_0\leqslant 3\wedge z'=z_0)\vee(2x_0>3\wedge y_0\neq 2\wedge z'=z_0)\vee(2x_0>3\wedge y_0=2\wedge z'=2.5)$。

定义 3　符号签名。设程序 P 对应的输入集合为 T，P 在每个输入作用下对应的路径条件集合为 $\Phi(\Phi=\varphi_1\vee\varphi_2\vee\cdots\vee\varphi_n)$，且在状态合并点处变量 t 的符号值集合为 $V(V=v_1\vee v_2\vee\cdots\vee v_i)$，对于任意一个 v_k，其对应的路径条件为 $\Phi_{vk}=\{\varphi_x\vee\varphi_y\vee\cdots\vee\varphi_z|1<x,y,z<n, x\neq y\neq z\}$，其中，$\varphi_x$、$\varphi_y$、$\varphi_z$ 为 3 条不同的执行路径对应的路径条件，此时变量 t 的符号签名[12]为$\{(\varphi, v_k)|\ \varphi\in\Phi_{vk}\}$。

在符号签名定义的基础上，可以在程序某个待分析的语句位置将程序的输入变量映射到其对应的值签名，将该过程的结果定义为“符号签名视图”。

定义 4　符号签名视图。对于一个程序 P，其输入变量集合为 T，程序中某个位置的变量$t(t\in T)$的值签名为 vs_t，则该处程序的符号签名视图为$\{t\mapsto vs_t\,|\,t\in T\}$。

符号签名视图可以直观地展示出该位置变量所有的路径条件和符号值。依据该定义，图 1(a)所示的程序对应的符号签名视图为$x\mapsto\{(\text{true},2x_0)\}$，$y\mapsto\{(\varphi_1\vee\varphi_3, y_0),(\varphi_2\vee\varphi_4, z_0-1),(\varphi_5, 1.5)\}$，$z\mapsto\{(\varphi_1\vee\varphi_2\vee\varphi_3\vee\varphi_4, z_0),(\varphi_5, 2.5)\}$。此种路径条件的逻辑表示形式中没有引入其他的临时变量且语义上保持不变，从而有利于减少约束求解器的开销。故符号签名的主要操作过程为：对于符号签名 vs 中的 2 个不同元素(φ_1,v_1)和(φ_2,v_2)，若 $v_1=v_2$，符号签名最后的结果为$(\varphi_1\vee\varphi_2,v_1)$，即等价于 $vs\backslash\{(\varphi_1,v_1),(\varphi_2,v_2)\}\cup\{\varphi_1\vee\varphi_2,v_1\}$，若 φ 为空，则可以将该符号签名删除。其优点在于可以共享不同执行路径中的路径条件表达式，减少约束求解器的调用次数。

4.2　依赖条件重构

在第 2 节中，图 2(a)所示的程序对应的 8 条可达路径中，存在$\neg(x-y>0)\wedge(x+y>10)$的路径条件，但在表 2 所示的路径分析中，并没有发掘该路径，这样会导致程序分析的不完备。导致这种路径条件丢失的原因是表 2 中步骤 3)在步骤 2)的基础上加入 $x+y>10$ 并对其逻辑结果取反，得到步骤 3)中的依赖条件$(x-y>0)\wedge\neg(x+y>10)$，满足该条件的输入设为 $x=6$, $y=2$, $z=2$，而该组输入对应的依赖条件是$\neg(x+y>10)$，这样导致步骤 3)中的条件 $x-y>0$ 在表 2 结果中丢失，本文采用依赖条件重构的算法来处理这种条件丢失的问题，该算法通过对依赖条件处理的顺序进行优化，从而保留关键的路径逻辑表达式，依赖条件重构的算法[13]如算法 1 所示。

算法 1　依赖条件重构算法

输入　程序 P，测试集 t，目标语句 C

输出　重构后的依赖条件

1) 栈 *Stack* 为空;
2) Procedure *Execute*(t, n)
3) 计算关于 C 的依赖条件;
4) 将 $\varphi_1\wedge\varphi_2\wedge\cdots\wedge\varphi_m$ 赋值给 *dc*;
5) 将 *Reorder*(*dc*) 赋值给 *dc'*;
6) $dc'=\varphi_1'\wedge\varphi_2'\wedge\cdots\wedge\varphi_m'$;
7) for all 从 n+1 到 m 中的变量 i

8)　　$k=\varphi_1' \wedge \varphi_2' \wedge \cdots \wedge \neg\varphi_i'$;
9)　　将函数值(k,j)压入栈 *Stack* 中;
10)　end for
11) return
12) end Procedure
13)
14) Procedure *Reorder*(*seq*)
15)　if *seq* 的长度为 0
16)　return *seq*;
17)　end if
18)　将 $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$ 赋值给 *seq*;
19)　将 seq_1 和 seq_2 的值都赋值为 true;
20)　for all 从 1 到 k_1 中的变量 i
21)　　if $br(\varphi_i)$是 $br(\varphi_k)$的子集
22)　　将 $seq_1 \wedge \varphi_i$ 的值赋给 seq_1;
23)　　else
24)　　将 $seq_1 \wedge \varphi_i$ 的值赋给 seq_2;
25)　　end if
26)　end for
27)　return *Reorder*(seq_1)$\wedge$*Reorder*(seq_2)$\wedge\varphi_k$;
28) end Procedure

算法中的 *Reorder*(…)的作用是对依赖条件分析顺序进行重构，它通过加入最新的路径条件从而将当前的路径序列 *seq* 划分为 seq_1 和 seq_2 这 2 个子序列，*seq* 与最新加入的路径条件之间有间接依赖关系，依据 *seq* 中条件的顺序将 *seq* 与最新加入的路径条件进行重构并存储在 seq_1 中，类似地，将 *seq* 与最新加入的路径条件之间没有间接依赖关系的放在 seq_2 中。然后对 seq_1 和 seq_2 分别再次进行该操作，当原路径条件集合为空时，算法停止工作并得到重构后的路径表达式。通过依赖条件重构分析，可以提高路径条件分析的精度。

5　实验及分析

为了验证本文方法的有效性，实验以符号执行为基础并进行状态合并分析，来对比本文方法和传统符号执行方法在状态合并效率以及程序分析精度上的提高。实验步骤为：通过构建程序的控制流图，以流敏感的方式对程序进行分析，同时采用 Z3 工具作为约束求解器。实验流程如图 4 所示。

为了对比本文方法和传统程序分析方法在状态合并等方面的效率，从合并数量、约束求解器调用次数等方面进行分析，实验结果如表 3 所示。其中，符号签名均值为程序出口语句中变量在不同执行路径中符号表达式的数量，其数值可以体现程序不同符号之间的关联关系紧密程度。若路径表达式和符号表达式在不同路径中存在相同前缀，则在新路径发掘过程中，可以利用已有的分析结果而减少计算开销；符号签名共享度用来衡量这种开销的比例，其值为到达目标语句位置的路径数量与符号个数的比值。从表 3 可以看出，本文方法较传统的分析方法有更好的状态合并效果，本文使用的分析方法可以较好地缓解因路径条件删除而带来的分析效率上的降低。

实验的另一个目的是分析路径的复杂度。现有的程序路径复杂度分析方法主要是分析程序输入数量与路径数量之间的关系。圈复杂度分析方法使用基本点测试的方法来计算复杂度，但是该方法的主要问题在于难以处理一条路径中的节点都归属

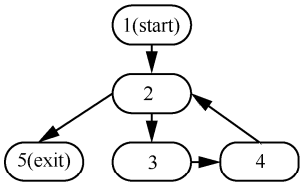
图 4　实验流程

表 3　　**状态合并和符号执行的对比**

测试集	基本特征				状态合并		符号执行		
	规模/行	执行时间/s	符号签名均值/个	符号签名共享度	合并数量/个	传统方法合并率	加速比	传统方法可行路径百分比	约束求解器平均调用次数/个
用例 1	72	21.2	5.5	4.4	209	100.0%	1.2	76.5%	3.8
用例 2	91	13.0	3.6	6.5	181	100.0%	5.2	65.2%	1.3
用例 3	207	29.9	4.7	6.4	224	53.3%	5.2	86.5%	2.6
用例 4	435	37.5	2.5	7.1	324	76.7%	4.8	61.6%	1.7
用例 5	591	31.7	2.6	15.5	280	43.1%	2.0	67.7%	1.8
用例 6	744	57.2	5.7	20.3	363	37.6%	3.3	85.8%	2.1
用例 7	1 329	86.3	5.5	18.6	408	51.3%	2.4	73.6%	1.6

于另一条路径的程序。NPATH 复杂度通过将循环次数限制在零次或一次的前提下进行分析，其分析结果对复杂度的计算有较大干扰。

由于循环语句的存在，导致路径分析和约束求解难以精确衡量程序路径的复杂程度。本文提出的路径复杂度分析方法以程序的执行深度为输入，从而获得程序执行路径的上界。该方法以控制流图（CFG，control flow graph）为研究对象并采用深度优先的搜索策略（DFS），当程序中没有循环结构时，该方法与圈复杂度和 NPATH 方法的分析结果一致；当存在循环结构时，复杂度为关于程序执行深度 n 的表达式。记 $path(n)$为长度不超过 n 的路径集合，同时定义路径统计序列$(a_0,a_1,\cdots,a_i,\cdots)$，其中，$a_i=path(n)$，图 5 所示的程序对应的路径统计序列为(0,0,1,1,1,2,2,2,3,⋯)。



(a) 程序结构

路径长度	路径数量	$path(n)$
00	0	0
10	0	0
21	1	1
30	0	1
40	0	1
51	1	2
60	0	2
70	0	2
81	1	3
90	0	3

(b) 路径序列

图 5　路径统计序列示例

路径统计序列可以通过产生函数表示为一个有限多项式的形式：$g(z)=\sum_{i>0}a_i z^i$，进一步可以使用泰勒级数的形式来表示：$g(z)=\dfrac{p(z)}{q(z)}$，其中，$p(z)$和 $q(z)$为有限多项式，此时统计序列中每一个元素可以通过计算 $a_i=\dfrac{g^i(0)}{i!}$ 得出，其中，$g^i(n)$表示 $g(z)$的第 i 次迭代。此时 $g(z)$可以表示为

$$g(z)=\frac{z^2}{1-z+z^3-z^4}=z^2+z^3+z^4+2z^5+2z^6+2z^7+3z^8+\cdots \tag{1}$$

所以可以通过泰勒级数的系数来决定与指定长度 n 相关的路径数量，例如，该处 $g(z)$的系数构成了图 5 中 $path(n)$对应的数值。对于给定的 CFG，可以通过构建转换矩阵 $\boldsymbol{T}$[15]来计算产生函数：如果 CFG 中 2 个节点 v_i 和 v_j 存在连接的边，则 $\boldsymbol{T}_{ij}=1$，否则 $\boldsymbol{T}_{ij}=0$。同时有 $\boldsymbol{T}_{|N|,|N|}=1$，则 $path(n)$的产生函数可以表示为

$$g(z)=(-1)^{|N|+1}\frac{\det((\boldsymbol{I}-z\boldsymbol{T}):|N|,1)}{\det(\boldsymbol{I}-z\boldsymbol{T})} \tag{2}$$

其中，$(\boldsymbol{I}-z\boldsymbol{T}):|N|,1$表示从行列式 $\boldsymbol{I}-z\boldsymbol{T}$ 中删除第 $|N|$ 行和第 1 列，函数 det(⋯)用来计算参数的行列式。图 5 程序对应的矩阵为

$$\boldsymbol{T}=\begin{pmatrix}0&1&0&0&0\\0&0&1&0&1\\0&0&0&1&0\\0&1&0&0&0\\0&0&0&0&1\end{pmatrix}$$

$$(\boldsymbol{I}-z\boldsymbol{T})=\begin{pmatrix}1&-z&0&0&0\\0&1&-z&0&-z\\0&0&1&-z&0\\0&-z&0&1&0\\0&0&0&0&1-z\end{pmatrix}$$

$$(\boldsymbol{I}-z\boldsymbol{T}:5,1)=\begin{pmatrix}-z&0&0&0\\1&-z&0&-z\\0&1&-z&0\\-z&0&1&0\end{pmatrix} \tag{3}$$

将式(3)代入式(2)中，可以得到 $g(z)=\dfrac{z^2}{1-z+z^3-z^4}$，这和式(1)的表示形式一致。对于一个产生函数，$path(n)$由 $q(z)$对应的根表示，设多项式 $q(z)$的最高次数为 d，$q(z)$有 d 个根，设其中不重复的$(r_1,r_2,\cdots,r_D)$的个数为 D，r_i 为第 i 个不重复的根，m_i 为 r_i 的个数，此时，$path(n)$可以表示为

$$path(n)=\sum_{i=1}^{D}\sum_{j=0}^{m_i}c_i n^j\left(\frac{1}{r_i}\right)^n \tag{4}$$

其中，c_i 为多项式中的系数。对于图 5 所示程序，$q(z)=1-z+z^3-z^4$ 有 4 个根，由于根 $r=1$ 有 2 个，故 $r_1=r_2=1$，且 $m_1=m_2=2$，$r_3=-\dfrac{1}{2}-\dfrac{\sqrt{3}}{2}\mathrm{i}$，$r_4=-\dfrac{1}{2}+\dfrac{\sqrt{3}}{2}\mathrm{i}$，$m_3=m_4=1$。

因此有

$$path(n)=c_0+c_1^n+c_2r_3^n+c_3r_4^n$$

$$=\frac{4}{3}+\frac{n}{3}+\frac{1}{6}\left(\frac{\mathrm{i}}{\sqrt{3}}-1\right)\left(-\frac{1}{2}-\frac{\sqrt{3}\mathrm{i}}{2}\right)^n-$$

$$\frac{1}{6}\left(\frac{\mathrm{i}}{\sqrt{3}}+1\right)\left(-\frac{1}{2}+\frac{\sqrt{3}\mathrm{i}}{2}\right)^{n} \tag{5}$$

由于 $q(z)$存在共轭复根 r 和 $\bar{r}$，且 $r^n+\bar{r}^n$ 为实数，故有 $path(n)\leqslant\frac{4}{3}+\frac{n}{3}+\frac{1}{6}\left(\frac{\mathrm{i}}{\sqrt{3}}-1\right)-\frac{1}{6}\left(\frac{\mathrm{i}}{\sqrt{3}}+1\right)=\frac{n+1}{3}$。从而得到了在指定路径长度的情况下，程序执行路径数量的上界。为了分析该方法的复杂度，依据 $path(n)$的上界进行逼近分析。若 $\lim\limits_{x\to\infty}\frac{f(n)}{g(n)}=1$，则 $f(n)=\Theta(g(n))$。

表 4 对比了圈复杂度、NPATH 复杂度和本文方法在处理分支语句和循环语句时的时间复杂度。由表 4 可以看出，圈复杂度的表示形式单一，NPATH 复杂度在处理嵌套分支条件和嵌套循环时的表示形式相同，故这 2 种表示方法不适合用来分析复杂程序的复杂度，而本文方法对于不同的语句形式有不同的复杂度表示形式，故可以作为程序复杂度的分析依据。其中，K 为条件语句数量，b 为嵌套的层数，n 为程序当前路径长度。

表 4　　复杂度表示形式对比

对比项	顺序分支	嵌套分支	顺序循环	嵌套循环
圈复杂度	K+1	K+1	K+1	K+1
NPATH 复杂度	2^K	K+1	2^K	K+1
本文方法	2^K	K+1	$\Theta(n^K)$	$\Theta(b^n)$

6 结束语

目前的程序分析方法存在路径条件提取不够精确以及约束求解器难以处理复杂的数据结构的情况，本文提出新的路径条件表示方法，通过将路径条件和符号表达式进行协同分析，可以减少临时符号变量的引入，另外本文还提出了依赖条件的重构算法，可以减少因路径条件删除而带来分析结果精度上的损失。实验结果表明，本文方法可以有效地进行变量状态的合并，并且可以提高程序分析的性能。将来的工作可以针对函数调用、数组下标等复杂数据结构进行精确建模，从而使本文方法有更广的使用范围。

参考文献：

[1] KUZNETSOV V, KINDER J, BUCUR S, et al. Efficient state merging in symbolic execution[J]. ACM Sigplan Notices, 2012, 47(6): 193-204.

[2] GODEFROID P. Compositional dynamic test generation[C]//ACM Sigplan-Sigact Symposium on Principles of Programming Languages. 2007: 47-54.

[3] AVGERINOS T, REBERT A, SANG K C, et al. Enhancing symbolic execution with veritesting[C]//International Conference on Software Engineering. 2014: 1083-1094.

[4] GODEFROID P, NORI A V, RAJAMANI S K, et al. Compositional maymust program analysis: unleashing the power of alternation[C]// ACM Sigplan-Sigact Symposium on Principles of Programming Languages. 2010: 43-56.

[5] 王翀, 吕荫润, 陈力, 等. SMT 求解技术的发展及最新应用研究综述[J]. 计算机研究与发展, 2017, 54(7): 1405-1425.
WANG C, LYU Y R, CHEN L, et al. Survey on development of solving methods and state of theart application of satisfiability modulo theories[J]. Journal of Computer Research and Development, 2017, 54(7): 1405-1425.

[6] TORLAK E, BODIK R. A lightweight symbolic virtual machine for solver-aided host languages[J]. ACM SIGPLAN Notices, 2014, 49(6): 530-541.

[7] VISSER W, REANU C S, NEK R. Test input generation for Java containers using state matching[C]//International Symposium on Software Testing and Analysis. 2006: 37-48.

[8] CHAKRABARTI A, GODEFROID P. Software partitioning for effective automated unit testing[C]//ACM & IEEE International Conference on Embedded Software. 2006: 262-271.

[9] SHARIR M, PNUELI A. Two approaches to interprocedural dataflow analysis[M]. Englewood: Prentice-Hall, 1981: 189-234.

[10] REPS T, HORWITZ S, SAGIV M. Precise interprocedural dataflow analysis via graph reachability[C]//ACM Sigplan-Sigact Symposium on Principles of Programming Languages. 1995: 49-61.

[11] PENG T, PADUA D. Gated SSA-based demand driven symbolic analysis for parallelizing compilers[C]//International Conference on Supercomputing. 1995: 414-423.

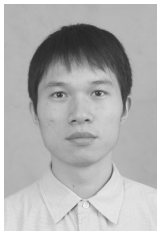
[12] SEN K, NECULA G, GONG L, et al. MultiSE: multi-path symbolic execution using value summaries[C]//The 10th Joint Meeting on Foundations of Software Engineering. 2015: 842-853.

[13] QI D W, NGUYEN H , ROYCHOUDHURY A. Path exploration based on symbolic output[J]//ACM Transactions on Software Engineering and Methodology. 2011, 22(4): 278-288.

[14] 王伟光, 曾庆凯, 孙浩. 面向危险操作的动态符号执行方法[J]. 软件学报, 2016, 27(5): 1230-1245.
WANG W G, ZENG Q K, SUN H. Dynamic symbolic execution method oriented to critical operation[J]. Journal of Software, 2016, 27(5): 1230-1245 .

[15] BANG L, AYDIN A, BULTAN T. Automatically computing path complexity of programs[C]//Joint Meeting on Foundations of Software Engineering. 2015: 61-72.

[作者简介]



郭曦（1983-），男，湖北鄂州人，博士，华中农业大学副教授、硕士生导师，主要研究方向为软件分析与测试、信息安全等。

王盼（1987-），女，河南济源人，博士，武汉电力职业技术学院讲师，主要研究方向为电力电子变换等。