形状分析符号执行引擎中的状态合并

邓 维 李兆鹏
(中国科学技术大学计算机科学技术学院 合肥 230026)
(中国科学技术大学先进技术研究院中国科大-国创高可信软件工程中心 合肥 230027)

摘 要 符号执行技术以其良好的精确度控制和代码覆盖率被广泛应用于静态程序分析和高覆盖率测试用例自动生成。符号执行在分析程序时,以模拟真实的程序执行过程的方式分析程序的数据流和控制流信息,并检查程序可能出现的所有状态,得到程序的分析结果。高精确度和高覆盖率要求对程序状态描述具体而完备,这会导致符号执行过程中常见的状态爆炸问题。首先提出在不同的执行路径上对具体内存状态进行合并的算法,然后对内存模型进行适度的抽象,扩大状态合并算法的适用范围,最后讨论状态合并所带来的实际效果,并提出了状态合并的优化解决方案。所提出的算法在符号执行引擎ShapeChecker上实现,并取得了良好的实验结果。

关键词 符号执行,状态合并,求解代价,内存模型,状态抽象
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.02.034

State Merging for Symbolic Execution Engine with Shape Analysis

DENG Wei LI Zhao-peng
(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
(USTC-Sinovate High Confidence Software Engineering Center, Institude of Advanced Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract Symbolic execution is widely used in static code analysis and automatic test generation for its well controlled precision and code coverage. When applied to analyze a program, symbolic execution traversals all possible states by simulating the execution of the program to analyze the data-flow and control-flow information and get results. High precision and coverage request detailed and complete description of program states, which will lead to the path-explosion problem in almost all implementations of symbolic execution. State merging is an effective way to solve the path-explosion problem. We firstly proposed an algorithm for the merging of states from different paths, then abstracted the states in a proper way to expand the application scope of the algorithm. Finally, we discussed the actual effect of state merging and put forward an optimization scheme. The whole algorithm is deployed in ShapeChecker, which is our symbolic execution tool, and experiments show good results in performance.

Keywords Symbolic execution, State merging, Query cost, Memory model, State abstraction

1 引言

提高软件的可靠性一直是软件开发所追求的目标之一。程序动态测试、程序静态分析和程序验证是目前确保软件质量的主要方法。程序验证作为保证程序正确性的最严格的手段,通过形式化方法对程序的各种性质给出了严格的数学证明,从而保证了程序的可靠性。然而目前程序验证还未实现自动化证明,需要大量的人工证明,因此未能在工业界得到广泛的应用。程序动态测试提供了程序运行时的检查,其精确性和覆盖率依赖于给定的测试集,而且运行时的检查成本和风险较高。程序静态分析是对程序的源代码进行分析,相比于程序验证,静态分析能自动分析源码中的相关信息;相比于程序动态测试,静态分析不用实际运行程序,从而可以以较低的成本尽早地发现程序中的缺陷。

符号执行[1]是一种常用的程序静态分析技术。符号执行采用一定的执行策略,以带约束的符号值代替程序变量运行时的具体值,精确地模拟程序的执行过程,从而达到分析时的高覆盖率,得到精确的分析结果。符号执行技术通过遍历程序中所有的可达路径来得到高覆盖率的分析结果,这使得采用该技术的分析工具具有随着程序中分支路径的增加所需执行路径呈指数级增加的特性,即符号执行中的路径爆炸问题,这严重制约了该类分析工具的可伸缩性(Scalability)。

到稿日期:2016-01-05 返修日期:2016-04-10 本文受国家自然科学基金项目(61229201,61170018,91318301)资助。
邓 维(1990-),男,硕士生,主要研究方向为程序分析,E-mail:csdgwei@mail.ustc.edu.cn;李兆鹏(1978-),男,副研究员,主要研究方向为程序设计语言、程序验证与程序分析。

随着计算机技术的广泛应用，现有的符号执行分析技术如何在计算机软件规模越来越大、结构也越来越复杂的情况下仍然能有效而准确地发现代码中的缺陷，保持低误报率、低漏报率以及良好的性能和可伸缩性，成为了开发符号执行分析工具中的一大挑战。

目前实验室研究团队已经实现了一个带形状分析的符号执行分析工具 ShapeChecker。为了解决循环和递归的状态遍历问题，该工具引入了循环和递归推断，归纳出了循环的循环不变式和递归函数的函数摘要，从而保证了工具在分析循环和递归时的精确度[2]；对执行过程中内存状态的特点进行分析，能够分析出符号内存中出现的链表、二叉树等数据结构，从而提高了特定数据结构程序的分析精度[3]；此外，工具引入了函数语义模型，对每个分析过的函数建立函数行为规范，减少了对同一个函数的分析次数，实现了可组合分析(Compositional Analysis)。路径爆炸问题同样是影响 ShapeChecker 分析性能和可伸缩性的一大因素，本文主要讨论路径爆炸问题对符号执行分析工具可扩放性和分析性能的影响，并尝试通过路径合并来提高符号执行工具的可扩放性及分析性能。本文的主要工作如下：

- 在使用符号执行技术的程序分析工具中，针对其内存模型，设计并实现了对不同路径上内存状态进行合并的算法和规则。
- 根据符号执行技术的特点，设计了两类合并点，并改进符号执行算法，使得调度器对到达合并点的状态进行合并。
- 根据对合并点的状态进行合并前后的代价估计，启发式地作出是否执行路径合并的决策，以追求符号执行分析工具整体性能最优的目标。

本文第 2 节介绍符号执行工具 ShapeChecker 的相关细节和其面临的挑战；第 3 节介绍主要的工作内容，即应对符号执行的路径爆炸问题所提出的路径合并操作及相关内容；第 4 节介绍路径合并模块所带来的实际效果以及对路径合并的优化；第 5 节给出实验结果；最后给出相关工作和总结。

2 符号执行工具 ShapeChecker 概述

ShapeChecker 的框架如图 1 所示。ShapeChecker 的输入为 LLVM 中间代码，这是因为对 C 语言源代码进行符号执行模拟较为困难：C 语言源代码的控制结构和表达式构成比较复杂，全局变量、静态变量、局部变量以及指针、指针参数等在源代码级别较难分析出其依赖或绑定关系；某些语言特性的语义在执行器中难以表示。经过 Clang 前端编译后的 LLVM 中间代码[5]大大简化了程序的符号执行分析过程。此外，LLVM 具有静态单赋值特性，便于推断定值引用关系，LLVM 工具链可以对 LLVM 中间代码中的信息进行提取，进而得到可以简化分析的结果。

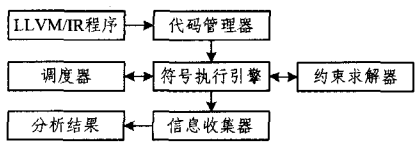

图 1 ShapeChecker 符号执行工具框架

图 1 中，执行引擎作为符号执行工具的核心模块，其作用类似于一个程序解释器，对每一个由调度器传入的程序状态进行模拟执行，即根据当前程序状态语句的语义对程序状态进行更新，这个过程同时会根据具体的分析策略对更新前后的程序状态进行检查，从而得到一系列有用的分析结果；在执行器检查和更新操作的过程中会调用约束求解器[4]，其根据给定的约束对查询的问题给出是否满足的判定，约束求解器的查询时间开销占据了工具整体时间开销的很大比例；在执行器完成对程序状态的更新之后，将会把更新后的程序状态返回给调度器，并由调度器决定下一次执行的程序状态，这个过程即可让调度器实现宽度优先、广度优先、带优先级的路径选择和随机路径选择等调度策略。调度器维护了一组程序状态，在符号执行过程的任一时刻，执行器从调度器中取出一个状态，执行该状态后进行调度器更新，直到调度器可调度状态为空，符号执行结束。

采用符号执行技术对程序进行静态分析的关键点在于对程序的动态运行进行真实模拟，因此在静态分析工具的误报率、漏报率、分析精度和可伸缩性四大评价指标下，面临着一些挑战。符号执行通过状态遍历的方式，试图找出程序运行时所有状态空间中可能出现的错误来实现低误报率和低漏报率，在程序运行时状态空间很大的情况下，这势必会降低工具的分析性能；在程序控制结构中出现很多条件分支语句的情况下，分析工具对所有分支语句的遍历会使状态数量随着分支数量呈指数级增长，从而造成状态爆炸的问题；对于一些无法判定是否终止的循环和递归，符号执行无法完全覆盖程序运行时的状态空间，而归纳出循环不变式与递归函数规范需要大量的求解器开销；复杂结构语义的建模会直接影响分析结果的精度，同时会通过求解器开销影响分析性能。

3 程序状态和状态合并

符号执行过程中存在状态爆炸问题，直观上，状态合并是一种缓解状态爆炸的技术。状态合并方法的设计同样与内存状态的表示紧密相关。本节介绍 ShapeChecker 符号执行引擎中的内存模型和针对其内存模型进行的状态合并算法的设计。

3.1 程序状态和路径爆炸问题

对多条执行路径的处理是导致状态爆炸的根本原因：在执行 branch 语句时，如果跳转条件被满足，则会产生两条不同的执行路径；这对应在符号执行中会将当前状态分裂成为两个状态以对应这两条执行路径，将跳转成立与否的条件分别加入到两个状态的路径约束中，并以调度器的调度策略分别对这两个状态进行符号执行。执行器对产生多条执行路径的 switch 语句的处理方式与 branch 语句相似，在 case 较多的情况下会分裂出更多的程序状态。

如图 2 所示的代码片段，符号执行处理完第 1—5 行的条件语句后，状态将分裂为两个，而在条件语句中没有处理变量 y 和 z；若保持两个状态独立执行，则第 6 行和第 7 行的指令

将会执行相同的语句两次,这显然产生了分析冗余。若在条件语句结束后加入状态合并且把两个状态合并为一个,则可以减少这种分析冗余。

```
1. if (cond(x)) {
2.    true-statement-without(y,z);
3. } else {
4.    false-statement-without(y,z);
5. }
6. do-something-with(y);
7. do-something-with(z);
8. do-something-with(x).
```

图 2　状态合并代码示例

符号执行过程中的状态类需要包含执行到某个程序点时的堆栈信息和程序计数器。此外,状态中的符号值变量是由其路径条件约束的,因此还需要记录从程序执行开始到此程序点的路径约束(Path Constraints)。ShapeChecker 在执行过程中对内存进行分析,并用抽象谓词记录分析得到的有用信息,这一部分信息称为抽象空间信息。ShapeChecker 的程序状态由以上 4 部分组成,表示成四元组(c, pc, cs, as),其含义如下。

• l:程序计数器,在符号执行引擎中记录了程序点当前的指令和下一条指令。

• pc:路径约束,由从程序开始执行到当前程序点的路径上的所有条件语句组成。

• cs:具体内存空间(Concrete Space),根据符号执行引擎对内存状态的描述,表示当前程序点的堆栈信息。

• as:抽象内存空间(Abstract Space),一些内存单元具有统一的性质,抽象谓词对具有这些内存单元的性质进行描述,从而构成了抽象内存空间。

具体内存空间使用式(1)和式(2)的二元组关系表示对内存状态的描述,记录程序中的每个对象的地址和值。

$$Variable \rightarrow Address \cup value \tag{1}$$

$$Address \rightarrow Address \cup value \tag{2}$$

具体内存空间能够精确地描述内存空间,但不支持符号化大小的内存空间,而且无法利用内存空间对象可能存在的一些性质[6],如单链表、双链表及二叉树等递归数据结构。抽象内存空间由对内存空间描述的断言语言组成[3],如果具体内存空间中出现了满足抽象谓词定义的内存单元,这部分内存单元将会被抽象成相应的抽象谓词。

3.2　状态合并

ShapeChecker 分析过程是上下文敏感的。对于两个可以合并的状态,其所在的程序点是相同的,因此状态合并算法主要由对路径约束的合并、具体内存空间的合并和抽象内存空间的合并 3 部分组成,其整体结构如图 3 所示。其输入的前两个参数为待合并状态(es', es''),若合并成功则返回 true,第三个参数为合并后的状态 es;否则返回 false,表示合并失败,且第三个参数无效。

```
1. Procedure merge_state(es', es'', &es)
2. begin
3.   vset'=var_set(es')
4.   vset''=var_set(es'')
5.   if vset' != vset''
6.     return false
7.   else
8.     vset=vset'
9.   end if
10.  init_state(es, vset)
11.  merge_path_constraints(es', es'', es)
12.  foreach v in vset
13.    merge_concrete_space(v, es', es'', es)
14.  end foreach
15.  if apply_abstract_space_merge_rule(es', es'', es)
16.    return true
17.  else
18.    return false
19.  end if
20. end
```

图 3　状态合并算法

3.2.1　路径约束和具体内存的合并

对于两个待合并的程序状态 $es_1=(l, pc_1, cs_1, as_1)$ 和 $es_2=(l, pc_2, cs_2, as_2)$,其路径约束 pc_1 和 pc_2 分别记录了从程序初始状态执行到这两个状态所需要满足的条件(在具体实现中路径约束表示为一个集合,集合的元素表示单个约束,所有元素的合取表示了路径约束);若两个状态能够合并,则合并后从程序初始状态执行到这个合并的状态所需要满足的路径约束为 pc_1 和 pc_2 的交集,即

$$pc=pc_1 \cap pc_2 \tag{3}$$

pc_1 和 pc_2 除其公共部分的余集即为表示两个状态的特征,且从逻辑上而言,这两部分是互斥的:

$$pc_i'=pc_i-pc, i=1,2 \tag{4}$$

在精确的符号执行分析过程中,具体内存空间的合并不允许有信息的丢失。若一个变量在两个状态的具体内存空间中有不同的值,则合并后的状态中该变量的取值应该能表示在不同的条件下变量在合并前的两个状态中分别的取值。结合路径约束合并中的信息,引入 ite(if-then-else)表达式:ite($cond$, $expr_1$, $expr_2$),其语义为:如果 $cond$ 满足,则整个表达式取值为 $expr_1$,否则取值为 $expr_2$。对于 es_1 和 es_2 中的同一个变量 x,其值若在两个状态中相等,则在合并后状态中的值保持不变,否则根据路径约束合并中的信息构建 ite 表达式,具体的如图 4 所示。

```
if eval(es1,x)=eval(es2,x)
  x'=eval(es1,x)
else
  x'=ite(pc1',eval(es1,x),eval(es2,x))
end if
```

图 4　具体内存空间变量的合并

图 4 中 pc_1' 由(4)式计算得到，由于 pc_1' 和 pc_2' 互斥，这样的合并是不会丢失信息的。合并操作可能使变量变成更为复杂的符号表达式，从而需要更多的求解器开销，这一部分的取舍将在第 4 节中详细讨论。

3.2.2 抽象内存空间的合并

抽象空间的合并规则建立在 ShapeChecker 本身的抽象规则上[7]，主要是对抽象空间中的形状谓词进行合并。ShapeChecker 中，形状谓词主要包括描述单链表表段的 Lseg、描述双链表表段的 Dlseg 和描述二叉树的 Tree。

由于篇幅限制，此处以 Lseg 为例来说明抽象规则所维持的形状谓词的同构关系，所有的形状谓词抽象和合并规则请见文献[7]。图 5 为 Lseg 的形式化定义：p 和 q 分别为表段的头、尾节点指针，s 表示表段所占用的内存空间；表段由 Lseg 归纳定义，其含义为一个表段要么为空表段，要么由一个已存在的表段和一个单节点连接而成。

$$\text{lseg}(ty^*\ p,\ ty^*\ q,\ s)\triangleq(p==q\wedge s==\text{empty})\vee(\text{valid}(p)\wedge \text{PtrSet}(ty)==\{next\}\wedge \text{lseg}([\![\text{getelementptr p next}]\!],\ q,\ s')\wedge s==\{p\}\cup s'\wedge\{p\}\cap s'==\text{empty})$$

图 5 Lseg 谓词的定义

Lseg 谓词的部分抽象规则如下：

$$\frac{\begin{array}{c}p,q:typ^*\ \ typ=type\{\overline{typ_i'^{\,i}}\}\ PtrSet(typ)=\{next\}\\ P\Leftrightarrow[\![getelementptr\ p\ idx]\!]==data\\ (0\leqslant idx<i\wedge idx\neq next)\wedge P'\end{array}}{\begin{array}{l}P\wedge[\![getelementptr\ p\ next]\!]==q\rightsquigarrow\\ P'[q/[\![getelementptr\ p\ next]\!]][data/[\![getelementptr\ p\ idx]\!]]\wedge lseg(p,q,\{p\})\end{array}}\quad\text{(SingleNodeToLseg)}$$

$$\frac{P\Rightarrow(\{p\}\cap\{q'\})==empty}{P\wedge lseg(p,q',\{p\})\wedge lseg(q',r,\{q'\})\rightsquigarrow P\wedge lseg(p,r,\{p,q'\})}\quad\text{(JoinTwoLseg)}$$

$$\frac{s\ \text{is}\ fresh\ P\Leftrightarrow fresh(p)\wedge fresh(q)\wedge P'}{P\wedge\wedge lseg(p,r,\{p,q\})\rightsquigarrow P'\wedge lseg(p,r,s)\wedge fresh(s)}\quad\text{(SimplifyLesg)}$$

SingleNodeToLseg 规则指从程序状态中识别具有指向自身的指针的节点，将其抽象为单节点的表段，然后根据 Lseg 的定义和 JoinTwoLseg 规则，将含有隐式存在的断点(Cut Point)从表段上消除，并利用 SimplifyLseg 规则将表段所占有的内存空间抽象为符号化空间。最终得到的表段是一系列由显式断点分开的合取谓词。在这样的抽象规则下，保持表段中断点相对位置相同的同构是合理的，因此给出单链表表段的合并规则：

$$\frac{}{\text{true},\text{true},\varepsilon\rightsquigarrow\text{true},\varepsilon',\phi,\phi}\quad\text{(Basic Rule)}$$

$$\frac{AS',AS'',ext(ext(\varepsilon,t',t'',t),s',s'',s)\rightsquigarrow AS,\varepsilon',\delta_0,\delta_1}{\begin{array}{l}lesg(h',t',s')\wedge AS',lseg(h'',t'',s'')\wedge AS'',\varepsilon\rightsquigarrow lseg(h,t,s)\wedge AS,\varepsilon',\delta_0,\delta_1\\ (\text{when}(h',h'',h)\in(\varepsilon\cup EQ)\wedge comb_\varepsilon(t',t'')=t\wedge comb_\varepsilon(s',s'')=s))\end{array}}\quad\text{(Lseg-Match Rule)}$$

$$\frac{AS',AS'',ext(ext(\varepsilon,t',h,t),s',0,s)\rightsquigarrow AS,\varepsilon',\delta_0,\delta_1}{\begin{array}{l}lseg(h',t',s')\wedge AS',AS'',\varepsilon\rightsquigarrow lseg(h,t,s)\wedge AS,\varepsilon',\delta_0\cup(h',t'),\delta_1\\ (\text{when}(h',h'',h)\in(\varepsilon\cup EQ)\wedge comb_\varepsilon(t',h)=t\wedge comb_\varepsilon(s',0)=s)\end{array}}\quad\text{(Lseg-PE-Left Rule)}$$

Lseg-Match 规则表示当待合并的两个状态在各自去除一个 Lseg 谓词之后能够合并成一个状态的前提下，若各自去掉的 Lseg 谓词满足 Lseg-Match 的前提条件，则可以将这两个状态按照 Lseg-Match 规则进行合并，得到合并后的状态；Lseg-PE-Left 规则表示待合并状态中左边的状态出现可能为空的 Lseg 谓词时合并的规则。对两个待合并状态不断应用相应的合并规则，直到可以匹配 Basic 规则，则合并成功；否则，若无法再应用任何规则且无法匹配 Basic 规则，则合并失败。

上述规则中，ε 表示在状态合并过程中产生的一系列替换关系，一个替换关系 (e',e'',e) 表示在合并前两个状态中的变量 e' 和 e'' 在合并后的状态中被替换成了 e，这个替换关系在合并过程中可以保证单链表表段显式断点相对位置的同构，并且在解析合并后谓词时可以根据路径约束还原成合并前的变量，减轻约束求解器的负担；EQ 显式变量的自等价替换关系为：$\{(e,e,e)\mid e \text{ is not a primed var}\}$；$comb_\varepsilon$ 函数提供在合并过程中寻找替换后变量的功能，其定义如下：

$$comb_\varepsilon(e',e'')=\begin{cases}e, & \text{if } (e',e'',e)\in\varepsilon\\ e', & \text{if } e'=e''\\ x \text{ for some } x\notin FV(\varepsilon,e',e''), & \text{otherwise}\end{cases}\quad(5)$$

δ_0,δ_1 记录了对状态中可能出现的空表段的合并，δ_0,δ_1 则在相应的 PE 规则中更新，记录这样的合并操作。

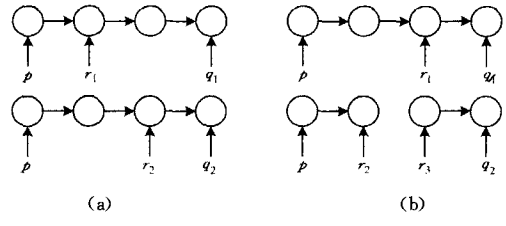

图 6 Lseg 合并示例

图 6 中上下两个部分分别表示两个程序状态中的单链表表段的形状信息。对于图 6(a)，经抽象后两个状态中的形状谓词为 $lseg(p,r_1,s_1)\wedge lseg(r_1,q_1,s_2)$ 和 $lseg(p,r_2,s_2)\wedge lseg(r_2,q_2,s_4)$，经过两次 Lseg-Match 规则后，其变为 $lseg(p,r,s_5)\wedge lseg(r,q,s_6)$。其中第二次应用 Lseg-Match 规则时，$r_1$ 和 r_2 已经在第一次应用 *Lseg-Match* 规则时扩充到三元关系 ε 中，并被替换成 r，因此能够合并。对于图 6(b)，经抽象后的形状谓词为 $lseg(p,r_3,s_1)\wedge lseg(r_1,q_1,s_2)$ 和 $lseg(p,r_2,s_3)\wedge lseg(r_3,q_2,s_4)$，在对第一个 Lseg 谓词应用 Lseg-Match 规则之后，因为 ε 以及自等价关系集 EQ 中没有包含 r_1 和 r_3 相等的关系，所以无法第二次应用 Lseg-Match 规则，合并失败。图 6(b)中的形状本身不是同构的，合并后并不能解析出两个状态中的形状特征。

3.3 符号执行算法的改进和合并点

引入状态合并后,符号执行算法的框架如图 7 所示,*w_list* 表示待执行的状态集合,初始为 *main* 函数开始的空状态,其路径约束为 *true*。

```
1. Procedure symbolic_execution(w_list)
2. begin
3.   merge_set:=∅
4.   while w_list != ∅ or merge_set != ∅
5.     s:=select_state(w_list)
6.     if s = NULL
7.       merged_set:=merge(merge_set)
8.       w_list:=w_list ∪ merged_set
9.       merge_set:=∅
10.      continue
11.    end if
12.    update_set:=execute(s)
13.    foreach s' in update_set
14.      if location(s') ∈ Merge_Point
15.          update_set:=update_set/{s'}
16.          merge_set:=merge_set∪{s'}
17.      end if
18.    end foreach
19.      update(w_list, update_set, s)
20.    end while
21. end
```

图 7　加入合并操作后的符号执行算法

上述算法中,*merge_set* 表示达到合并点,需要调用合并函数的状态集合。算法第 5 行从 *w_list* 中选取一个状态 *s*,此处的 select_state 函数依赖于具体的执行策略。若 *w_list* 为空,会导致 *s* 为 NULL,从而触发第 7－10 行的代码,对 *merge_set* 中的状态进行合并,并将合并后的所有状态返回到 *w_list* 中;若 *w_list* 不为空,则会直接执行第 12 行的语句,*update_set* 表示执行 *s* 后产生的状态集合。第 13－18 行表示对于执行后的状态集中的所有状态,若其处于合并点,则将其放入 *merge_set* 中;若不处于合并点(Merge_Point),则将其放入 *w_list* 中。第 19 行表示根据 *update_set* 的当前状态 *s* 更新 *w_list*。

有效的路径合并点应当使模拟执行器在模拟两条不同的路径之后在第一个汇合点进行合并[9],这些汇合点称为第一类合并点。第一类合并点可以归纳为条件语句结束点和循环语句起始点,在 llvm 中间表示的 CFG(Control Flow Graph)上所有入度大于 1 的节点构成了第一类合并点。其次,工具为了实现可组合分析,引入了霍尔风格[11]的函数行为规范(Function Behavior)和函数摘要(Function Summary)[3]。为了协同工具的可组合分析,有必要在函数调用点和函数返回点对状态进行合并,从而引入了第二类合并点。在函数调用点进行状态合并可以减少函数调用点的状态数量,从而减少应用函数行为规范而产生的程序状态的数量;在函数返回点进行状态合并可以减少构建函数行为规范的数量。

4 状态合并的优化

第 3 节提到状态合并会引入更多、更复杂的符号值,这会使得合并后约束求解器的求解开销增大;本节讨论状态合并决策的优化,用于解决引入状态合并后分析性能反而降低的问题。

4.1 查询热值

符号执行引擎的时间开销主要由状态遍历和约束求解器调用两部分组成。符号值相对于常值而言,其取值是由路径约束决定的,而常值则表示与路径约束无关的值。当符号执行引擎执行到某一条语句需要对语句中变量的值进行确定时,若该变量的值是符号值,就需要调用约束求解器。而在约束求解器的性能确定的情况下,这个过程主要取决于待求解的变量符号表达式及其约束表达式的复杂程度。

路径合并模块在减少状态数量的同时,使得合并后的状态在其后的执行过程中变得更难解析:合并会将变量由常值变为符号值,或者由简单的符号值变为复杂的符号值,每次需要用到该变量的值的信息时,均需要更多的求解器时间开销,如果这种时间开销过多,则会降低合并所带来的收益。

为此,提出查询热值。程序点 l 处的变量 v 称为查询热值,它是指两个状态合并后,从该程序点到符号执行结束,该变量引起的查询时间开销大于两个状态不合并而独立执行到符号执行结束的时间开销。而对变量 v 进行合并后,引起的求解器时间开销增加主要是来源于两方面:1)变量 v 经过状态合并后变成了形式更为复杂的 ite 符号值,这使得求解器解析时间增加;2)经过状态合并,变量 v 由不相等的常值变成了 ite 符号值,使得求解器调用次数增多。由查询热值的定义得到其数学表示为:

$$qhv(l)=\{v|(cost_{all}(l)-cost_{fst}(l,v))+(\alpha+1)*cost_{fst}(l,v)+cos_{snd}(l,v)>cost_{all}(l)+cost_{all}(l)\} \quad (6)$$

其中,$qhv(l)$ 表示在 l 处查询热值的集合;$cost_{all}(l)$ 表示不使用状态合并的情况下单个状态在 l 处往下执行所需要的查询次数;$cost_{fst}(l,v)$ 表示在 l 处若执行了合并操作,变量 v 由于原因 1)增加的查询次数,α 表示为了求解形式更为复杂的 ite 表达式而导致的约束求解器求解时间增加的增量,这个值大于 1;$cost_{snd}(l,v)$ 表示变量 v 由于原因 2)增加的约束求解器调用次数。

4.2 求解代价驱动的优化

由查询热值的定义可知,若在待合并的状态中将查询热值变为更复杂的符号值,则会使得状态合并的收益被约束求解器的开销增大所覆盖。因此,在遇到含有查询热值的状态时,不应该将其合并。由于约束求解器的求解时间无法静态确知,本节通过对求解代价的估计计算出查询热值。

首先引入辅助计算函数 $cost(l,query)$,其含义为从程序点 l 往后执行一共所需要的查询次数,定义如下:

$$cost(l,query)=\begin{cases}\beta*cost(l',query)+\beta*cost(l'',query)+query(l,e), & \text{if } instr(l)=brance(e,l',l'')\\ cost(succ(l),query)+query(l,e), & \text{if } instr(l)=assert(e) \text{ or } instr(l)=mem\_acc(e)\\ 0, & \text{if } l \text{ is the last instruction}\\ cost(succ(l),query), & \text{otherwise}\end{cases} \quad (7)$$

其中，*query* 代表了一个以(l,e)为参数的函数，表示了在 l 处对表达式 e 进行解析时是否需要调用约束求解器，如果需要则返回 1，如果不需要则返回 0。约束求解器主要在条件语句判断、断言检查和内存读写时调用，因此在子式中分别使用 $query(l,e)$ 表示这样的需求。由于分支条件不能静态确定，因此引入了假设“条件跳转语句的每个分支都以概率 $\beta(0<\beta<1)$ 可达”。整个函数以递归的方式，可以计算到程序指令为终止时的查询次数。

记 $sym(s,v)$ 表示变量 v 在状态 s 中为符号值，并记 $depend(l,v,l',e)$ 表示 l' 处的表达式 e 依赖于 l 处的变量 v 的值，结合 $cost(l,query)$，可以得到 $cost_{fst}(l,v)$，$cost_{snd}(l,v)$ 和 $cost_{all}(l)$ 的计算方式：

$$cost_{all}(l)=cost(l,\lambda(l',e)).\ ite(\exists v:(sym(s_1,v)\vee sym(s_2,v))\wedge depend(l,v,l',e),1,0) \tag{8}$$

$$cost_{fst}(l,v)=cost(l,\lambda(l',e).\ ite(\exists v:(sym(s_1,v)\vee sym(s_2,v))\wedge(s_1[v]!=s_2[v])\wedge(depend(l,v,l',e)),1,0) \tag{9}$$

$$cost_{snd}(l,v)=cost(l,\lambda(l',e).\ ite(\exists v:!(sym(s_1,v)\vee sym(s_2,v))\wedge s_1[v]!=s_2[v])\wedge(depend(l,v,l',e)),1,0) \tag{10}$$

其中，s_1 和 s_2 表示待合并的两个状态。式(8)给出了利用 $cost(l,query)$ 计算 $cost_{all}(l)$ 的具体实现，即若变量 v 在任一状态中为符号值，*query* 保证在 l' 遇到表达式 e 时，若对该表达式的求值依赖于程序点 l 处的变量 v，则返回 1，表示需要一次求解器调用，否则返回 0；同理，式(9)表示变量 v 已经为符号值且在两个状态中不相等，合并将会引入更复杂的 ite 表达式产生的求解次数；式(10)计算变量 v 由具体值因合并而变为符号值后引起的额外的求解器调用次数。

由于无法静态地确定 $sym(s,v)$ 函数，因此引入假设“所有变量中，其值为符号值的变量所占比例为一个系数 $\gamma(0<\gamma<1)$”，则 $cost_{all}(l)$ 的具体实现可以简化如下：

$$cost_{all}(l)=\gamma*cost(l,\lambda(l',e).1) \tag{11}$$

为了减少决策过程的开销，对于 $cost_{fst}(l,v)$ 和 $cost_{snd}(l,v)$，以平均变量的额外查询次数估计具体变量的额外查询次数。假设在合并点待合并的两个状态中只有一个变量的值不相同，也即只需要对这一个变量进行合并，此时计算出的 $cost_{fst}(l,v)$ 和 $cost_{snd}(l,v)$ 为平均变量的额外查询次数，可记为 $cost_{fst}(l)$ 和 $cost_{snd}(l)$，具有性质：

$$cost_{fst}(l)=cost_{snd}(l)=cost(l,\lambda(l',e).\ ite(depend(l,v,l',e),1,0)) \tag{12}$$

引入假设“每个具体变量的额外查询次数为平均变量的额外查询次数乘以一个系数 δ”之后，得到 $cost_{fst}(l,v)$ 和 $cost_{snd}(l,v)$ 的计算方式：

$$cost_{fst}(l,v)=cost_{snd}(l,v)=\delta*cost(l,\lambda(l',e).\ ite(depend(l,v,l',e),1,0)) \tag{13}$$

将上述假设和其产生的效果化简，可以得到查询热值的计算方式：

$$v\in qhv(l)\Leftrightarrow_{def}(1+\alpha)*\delta*cost(l,\lambda(l',e).\ ite(depend(l,v,l',e),1,0))>\gamma*cost(l,\lambda(l',e).1) \tag{14}$$

其中，α 是 γ 进行代价估计时最重要的参数，若其取值为 0，则对因合并引入的复杂的 ite 表达式进行解析而导致的时间开销增量可以忽略不计，此时查询热值集为空集，表示不开启优化；若取值趋向于无穷大，则解析因合并引入的 ite 表达式导致的时间开销增量趋向于无穷大，此时优化模块倾向于把所有需要查询的变量当作查询热值，从而使得状态合并的条件更为严苛。计算出集合 $qhv(l)$ 后，优化策略则可简单地概括为：若待合并状态中的某个变量是查询热值，且该变量在合并前两个状态中取值不相等，则两个状态不能合并，即不为查询热值集中的变量引入更复杂的 ite 表达式。

5 实验结果

本文在符号执行引擎 ShapeChecker 中实现了以上所述的状态合并算法，在工具中引入了状态合并算法后，某些程序分析的时间明显缩短，而对于另外一些控制流结构，状态合并算法的优化效果并不明显。本文选取了 GNU Coreutils 中的 35 个程序作为测试用例，对比了工具加入状态合并优化方式前、后的工具性能，部分结果如表 1 所列，测试的硬件条件为 Intel i7-3770，内存空间上限设为 2GB。

表 1 GNU Coreutils 部分实验结果

Coreutils 程序	程序规模/IR 行数	状态合并前分析时间/s	状态合并后分析时间/s
tail. c	6937	369.232	157.586
tac. c	9144	7979.154	3840.463
nl. c	3127	246.385	124.657
fold. c	2031	166.597	105.058
fmt. c	3425	3762.241	3596.158
test_who. c	5759	818.291	1154.738

实验结果可以分为 3 类，在本文所给出的抽象规则和合并规则下，对于第一类实验结果，例如 tail. c，tac. c，nl. c 等程序，状态合并具有良好的优化效果，得到这类实验结果的测试程序共有 19 个；对于第二类结果，例如 fmt. c 程序的优化效果并不明显，得到这类实验结果的测试程序共有 15 个；第三类较为极端的实验结果出现了时间开销更大的情况(如 test_who. c)。

这些实验现象在理论分析中可以得到解释：

(1)第一类测试用例优化效果较好，其控制结构出现了较多的条件分支，而且这些条件分支中的判定条件大多无后效性，因此对于每一次的状态合并，都能将工具的分析时间按照一定的比例减少，最终得到较好的优化效果。

(2)第二类优化效果不明显，其原因主要有两个：1)其控制结构出现的条件分支判定条件有很强的后效性，因此对于每一次的状态合并，将会增加很大的求解器开销，这种开销抵消了状态合并的优化效果；2)其条件分支中的内存状态结构相差较大，因此基于已有的抽象和合并规则不能对这些状态进行合并，所以状态合并模块并未对这些测试用例起到效果。

(3)第三类优化效果不佳的原因是对于后效性很强的条件分支语句，状态合并将其不同状态合并后严重地增加了求解器的开销，导致工具整体性能的下降。而引入合并代价估计模块主要也是为了减少这种情况的出现，使得状态合并模块至少不会使工具性能降低。此类情况在所有测试过程中仅出现了一次。

对于第三类优化效果不佳的情况，引入了状态合并的优化。在实验中，对 α 的取值进行控制，而设定 β 值为 1，即所有分支认为可达，γ 值为 0.05，σ 为 1，得到如表 2 所列的实验结果。

表 2　test_who.c 状态合并优化实验结果

α 值	0	1	10	不合并
分析时间/s	1162.943	831.677	824.337	818.291

由第 4 节的分析可知，表中 α 值为 0 时表示任何情况下均可以执行合并，此时相当于不加入优化模块；而当 α 值增大时，合并条件逐渐变得更严格，使得合并的次数变少。α 值取 1 时的分析时间已经接近不引入状态合并的分析时间。在开启合并代价估计后，工具能较好地识别条件分支语句的后效性。在合并将会导致更大的时间开销的情况下，状态合并模块将不会对状态进行合并，从而使得待分析的程序不适宜使用状态合并时，自动避免对其状态进行合并。

结束语　符号执行技术最初用于硬件的检测，取得了很好的效果，这是因为相比于软件，硬件结构一般较为简单，状态数也较少，因此很难暴露出符号执行的遍历状态的缺点。随着符号执行技术越来越多地应用于软件的测试和分析，符号执行状态爆炸的问题成为了所有采用符号执行技术的静态分析工具的性能瓶颈。

V. Kuznetsov 等提出了动态状态合并策略[8]，本文的查询热值和合并代价估计借鉴了他们的思路，并在 llvm 中间代码上实现了数据依赖关系分析，使得合并代价估计更为准确。

KLEE[10]在分析过程中为每个对象分配内存空间，从而可以直接调用库函数，较好地解决了外部函数调用的问题。但其由于简单的符号执行方式，如遇到循环体和递归函数时，会一直执行到循环条件不成立和递归函数非递归路径，以及其简单的内存分配方式，使得 KLEE 在分析大型程序时很容易出现不终止和内存耗尽的情况。

Clang[12]所带的静态分析工具 scan-build[13]所述的内存模型不支持符号化大小的内存空间，在处理动态内存分配的程序时则很容易产生误报和漏报。同时，scan-build 对循环的处理仅仅是展开几次进行分析，且其后端所使用的约束求解器的能力较弱，遇到较复杂的约束求解问题时只能快速返回，这使得 scan-build 的时间性能较好，但对大多数情况下的循环分析是错误的。

Coverity[14]通过限定分析路径数目上限的方法来缓解状态爆炸的问题，Fority 通过设置时间上限和内存空间上限的方法来缓解状态爆炸的问题。这两种方式均是以牺牲状态遍历完备性作为代价，因此会造成较为严重的漏报问题，这在实际分析过程中需要权衡折中。

目前 ShapeChecker 已经能对一定规模的程序在可以接受的时间内给出较好的分析结果，但仍然有很大的优化空间，如支持的抽象数据结构目前主要是单链表、二叉树和双链表，如何支持更为复杂的数据结构(如 hash 表、嵌套数据结构等)分析仍是接下来研究的重点。增强分析工具求解能力[15]的同时，降低分析工具的负担，从程序本身控制流和数据流结构上删除对分析结果没有影响的代码，对程序进行切片[16]，这是从另外一个角度提升工具整体性能的方式。这些都是我们后续工作的方向。

参 考 文 献

[1] KING J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.

[2] LIANG J B, LI Z P, ZHU L, et al. Symbolic Execution Engine with Shape Analysis[J]. Computer Science, 2016, 43(3): 193-198. (in Chinese)
梁家彪，李兆鹏，朱玲，等. 支持形状分析的符号执行引擎的设计与实现[J]. 计算机科学，2016，43(3)：193-198.

[3] ZHU L, LI Z P, LIANG J B, et al. A Specification Language for Precise Shape Analysis of C Program[J]. Journal of Chinese Mini-Micro Computer Systems, 2016, 37(4): 653-658. (in Chinese)
朱玲，李兆鹏，梁家彪，等. C 程序精确形状分析中的规范语言设计[J]. 小型微型计算机系统，2016，37(4)：653-658.

[4] SOOS M. SMT Competition'14 and STP[OL]. http://www.msoos.org/2014/06/smt-competition14-and-stp.

[5] LLVM I R. LLVM Language Reference Manual[OL]. http://llvm.org/docs/LangRef.html.

[6] BAUDIN P, CUOQ P, FILLARTE J C, et al. ACSL: ACSL/ISO C Specification Language(Version 1.7)[Z]. 2009-2013.

[7] ShapeChecker [OL]. http://kyhcs.ustcsz.edu.cn/~shapechecker.

[8] KUZNETSOV V, KINDER J, BUCUR S, et al. Effiecient State Merging in Symbolic Execution[J]. ACM Sigplan Conference on Programming language Design & Implemetation, 2012, 46(6): 193-204.

[9] ARONS T, ELSTER E, OZER S, et al. Efficient Symbolic Simulation of Low Level Software[C]// Design, Automation and Test in Europe. 2008: 825-830.

[10] CADAR C, DUNBAR D, ENGLER D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[C]// Usenix Symposium on Operating Systems Design & Implementation. 2008: 208-224.

[11] HOARE C A R. An Axiomatic Basis for Computer Programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[12] Clang Static Analyzer[OL]. http://clang-analyzer.llvm.org.

[13] XU Z X, KREMENEK T, ZHANG J. A Memory Model for Static Analysis of C Programs[C]// Proceeding of the 4th International Conference on Leveraging Applications of Formal Methods, Verification and Validation. 2010.

[14] Coverity[OL]. https://coverity.com.

[15] PALIKAREVA H, CADAR C. Multi-solver Support in Symbolic Execution[M]// Computer Aided Verification: Lecture Notes in Computer Science, 2013: 53-68

[16] CIFUENTES C, SCHOLZ B. Parfait: Designing a Scalable Bug Checker [C] // IEEE International Working Conference on Source Code Analysis & Manipulation. 2008: 4-11.