第34卷 第6期
2011年6月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 34 No. 6
June 2011

提高路径敏感缺陷检测方法的效率及精度研究

赵云山[1)] 宫云战[1)] 刘 莉[1)] 肖 庆[1)] 杨朝红[1),2)]

[1)](北京邮电大学网络与交换技术国家重点实验室 北京 100876)

[2)](装甲兵工程学院信息工程系 北京 100072)

摘 要 路径敏感的缺陷检测方法其缺陷状态会关联当前控制流节点的所有数据流信息,由于其中包含与缺陷检测无关的数据流,因此会导致分析效率下降.为了避免全路径敏感分析时的路径爆炸问题,一般会在控制流汇合节点进行缺陷状态合并,而这种粗糙的合并策略带来的精度损失会引起误报.针对上述问题,文中提出一种基于缺陷的程序切片方法,该方法基于缺陷特征和路径条件建立切片准则,根据控制流节点上的数据流信息与切片准则的包含关系进行程序切片,得到的切片程序在缺陷检测时切片掉了缺陷无关节点且与源程序完全等价,以提高缺陷检测效率.为了进一步减少路径敏感分析方法的误报,提出一种基于切片的缺陷状态合并策略,根据控制流分支节点的路径条件,对缺陷状态添加状态属性,从而有选择地对控制流汇合节点进行状态合并,减少精度损失.文中所述方法已在缺陷检测系统(DTSGCC)中实现.对大量Linux中GCC开源工程的测试结果表明,文中提出的方法可以提高路径敏感缺陷检测方法的效率,并减少误报.

关键词 静态分析;缺陷检测;路径敏感;误报;程序切片;上下文敏感分析; 域敏感分析

中图法分类号 TP311 **DOI号**: 10.3724/SP.J.1016.2011.01100

Improving the Efficiency and Accuracy of Path-Sensitive Defect Detecting

ZHAO Yun-Shan[1)] GONG Yun-Zhan[1)] LIU Li[1)] XIAO Qing[1)] YANG Zhao-Hong[1),2)]

[1)](*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing* 100876)

[2)](*Department of Information Engineering, Academy of Armored Force Engineering, Beijing* 100072)

Abstract While detecting defects with path-sensitivity, the defect state contains all data flow information of the current control flow vex, which might lower the efficiency by the defect irrespective data flow information. Further, in order to avoid the path explosion while full-path-sensitive analysis, the defect states encountering the control flow confluent nodes might be simply merged. The preliminary state-merging strategy might lead to an accuracy loss which could induce false positives. To address the above issues, this paper proposes a new program slicing algorithm based on defect patterns. The slice criteria include defect feature and path condition, and the source program is sliced by the inclusion relation between the CFG dataflow information and the slice criteria. The sliced program not only slices the defect irrespective codes, but also is totally equivalent to the original program, which improves the efficiency. In order to further reduce the false positives of path-sensitive analysis, this paper presents a refined state-merging strategy to diminish the accuracy loss, which selectively merges the defect states by adding path condition as state attribute. The authors have implemented the technique in DTSGCC (Defect Testing System for GCC), a software defect detecting tool for GCC projects in Linux. DTSGCC is applied to validate plenty of GCC open source projects. Experimental results suggest that applying the tech-

收稿日期:2010-10-22;最终修改稿收到日期:2011-04-26.本课题得到国家"八六三"高技术研究发展计划项目基金(2009AA012404)、国家自然科学基金项目"航天嵌入式软件缺陷检测方法研究、系统研发及应用"(91018002,2010)资助.**赵云山**,男,1984年生,博士研究生,主要研究方向为软件静态测试、自动化测试工具. E-mail: zilong831026@163.com. **宫云战**,男,1962年生,教授,博士生导师,主要研究领域为软件测试、软件工程. **刘 莉**,女,1988年生,硕士研究生,主要研究方向为静态测试. **肖 庆**,男,1979年生,博士研究生,主要研究方向为软件测试、程序分析. **杨朝红**,男,1976年生,副教授,主要研究方向为软件测试、软件工程.

nique to path-sensitive defect detecting analysis improves the efficiency, at the same time reduce potential false positives.

Keywords static analysis; defect detecting; path-sensitive; false positive; program slicing; context-sensitive analysis; field-sensitive analysis

1 引 言

软件测试是提高软件质量的重要手段，根据是否运行被测试程序，软件测试可以分为动态测试和静态测试.基于软件缺陷的静态分析方法可以针对小概率缺陷实施有效测试，受到了学术界和工业界的广泛关注.静态分析的效率是影响其能否应用于大型软件缺陷检测的关键，它与分析过程中的计算复杂度密切相关.由于静态分析需要抽象出完整的程序语义信息，该抽象语义信息往往是精确程序语义的“保守”近似，从而导致其计算量要远大于程序精确语义所表示的计算量，因此减少保守性分析时的计算量可以提高分析效率.根据 Rice 定理[1]，静态分析针对程序的任何非平凡属性(例如：是否存在运行时错误)，不可能做到既是可靠的(sound)又是完备的(complete)，导致其计算结果可能会出现误报(false positive)和漏报(false negative).大量的误报会使人对分析工具失去信心，而漏报会造成程序具有较高质量的假象，因此提高精度是完善静态分析功能的又一挑战.

路径敏感的缺陷检测方法从控制流图头节点依次进行状态迭代，每个缺陷状态都会关联当前控制流节点的所有数据流信息，在缺陷状态迭代计算时，与缺陷无关的数据流信息会在控制流上进行传递和计算，这种无关计算势必降低缺陷检测的效率[2-4].文献[5-6]通过增加控制流节点构造新的路径，或重构控制流图以消除不可达路径的方法，实现了路径敏感分析，这是一种典型的以效率换精度的方法，限制了其在大型软件缺陷检测中的应用.

基于数据流分析的路径敏感检测方法考虑分支间的组合关系，可以记录控制流图上的不同路径信息，从而有效减少静态分析时的误报.精确的路径敏感分析方法会记录程序中的所有路径信息，在控制流分支较多或存在循环时会导致路径爆炸，从而无法进行分析.因此，实用的路径敏感分析方法往往会采用一些折衷策略，有可能导致精度损失：(1)不同路径上的数据流信息在控制流汇合处合并；(2)数据流在不可达路径上进行传递.文献[4]采用迭代求精策略，每次迭代分析的结果都会更新状态合并准则，这种可调整的合并准则会减少由于关键路径合并而带来的精度损失，但迭代求精要进行重复计算，且有可能面临迭代不终止的情况.文献[3]采用变量的抽象取值来表示状态条件，在控制流汇合节点通过合并相同状态中的状态条件来避免路径爆炸，但该方法的状态合并策略没有区分在哪些汇合节点可以进行安全的状态合并，导致与缺陷相关的路径信息丢失从而引起误报.

基于上述对路径敏感缺陷检测方法的分析，可以得到以下结论：(1)控制流图节点数决定了状态迭代计算的次数；(2)状态条件中关联数据流信息的数量决定了沿控制流进行状态迭代时的复杂度；(3)状态合并策略影响路径敏感检测方法的精度.本文主要关注点是如何优化这 3 个方面，以提高路径敏感分析方法的效率和精度.

借鉴需求驱动[7]程序分析方法的思想，本文将程序切片技术[8]应用于缺陷检测，提出一种基于缺陷的程序切片方法.该方法基于缺陷特征和路径条件建立切片准则，根据控制流节点上的数据流信息与切片准则的包含关系进行程序切片，得到的切片程序在缺陷检测时不仅切片掉了缺陷无关节点，从而减少了数据流迭代时的计算量，而且与源程序完全等价从而保证了静态分析的保守性.为了进一步减少误报，提出一种基于切片的缺陷状态合并策略，根据控制流分支节点的路径条件，对缺陷状态添加状态属性，从而有选择地对控制流汇合节点进行状态合并，以提高检测精度.为验证方法的有效性，在缺陷检测系统 DTSGCC(Defect Testing System for GCC)中实现了上述方法，对 Linux 中 10 个开源项目的缺陷检测结果表明，本文方法在检测代码量较大的程序时，有效提高了路径敏感缺陷检测方法的效率，同时减少了误报，为提高软件质量及安全性提供了有效可行的方法及工具.

我们的贡献主要有：(1)将程序切片技术应用于缺陷检测，根据缺陷特征和路径条件建立切片准则进行程序切片；(2)采用基于切片的缺陷状态合

并策略，有选择地对控制流汇合节点进行状态合并；(3)设计实现了针对 Linux 系统中 GCC 程序的路径敏感缺陷检测工具，在检测代码量较大的程序时，与同类工具相比效率更高.

本文第 2 节首先介绍缺陷检测的一些基本概念，包括控制流图、缺陷模式、缺陷状态自动机等，然后以资源泄露缺陷模式为例介绍了路径敏感缺陷检测方法的原理，并给出我们的研究动机；第 3 节对提高分析效率和精度的方法进行详细描述，并通过一个具体的例子展示如何进行程序切片及状态合并；第 4 节介绍缺陷检测工具 DTS 的框架及实验环境，并对 10 个开源程序的检测结果进行分析；第 5 节给出相关工作比较；第 6 节总结全文并展望未来工作.

2 研究背景与动机

本节首先对缺陷检测的基本概念作简要介绍，然后使用相同状态合并的路径敏感分析方法，通过对一个具体的例子进行分析，给出我们的研究动机.

2.1 研究背景

根据是否考虑程序语句的执行次序，静态分析可以分为流敏感分析(flow sensitive)和非流敏感分析(flow insensitive). 程序的控制流图(Control Flow Graph, CFG)即是对语句执行次序的抽象，是具有单一的、固定的入口节点和出口节点的有向图.

定义 1. 控制流图. 程序的控制流图可以表示为一个有向图 $G=(N,E,n_0,n_f)$，其中：N 代表节点的集合，每个节点 $n_i \in N$ 反映程序中的顺序执行语句 *CommonStmt*、条件判断(循环)语句 *SelectionStmt* 等，与节点 n_i 关联的程序语句块表示为 $Stmt(n_i)$；$E \in N \times N$ 代表有向边的集合，反映程序中语句间的控制流关系，e_h 和 e_t 分别表示有向边 e 的头节点和尾节点；n_0 为函数的唯一入口节点，n_f 为函数的唯一退出节点.

静态缺陷检测方法关键是对缺陷模式进行定义和检测，能处理的缺陷模式种类越多则分析检测能力越强.

定义 2. 缺陷模式. 指程序中经常发生的缺陷(BUG)所呈现出的语法或语义特征.

缺陷模式是对程序属性的一种描述，如果违反该属性则造成一个缺陷. 例如，申请的资源在使用完后必须释放，否则造成资源泄漏缺陷(Resource Leak, RL)；数组下标的使用必须在其数组声明大小范围以内，否则会造成数组越界缺陷(Out Of Boundary, OOB)；指针在解引用之前必须确保其指向非空，否则会造成空指针引用缺陷(Null Pointer Dereference, NPD).

状态机是对程序语义的一种常用和易于理解的抽象表示，缺陷模式可以用缺陷模式状态机来表示.

定义 3. 缺陷模式状态机[①]. 用于描述缺陷模式的有限状态机(Finite State Machine, FSM)，包括状态集合 D、状态迁移集合 T 及迁移条件集合 *Conditions*，其中 $D=\{\$start, \$error\} \cup D_{other}$，$T: D \times Conditions \rightarrow D$. $\$start$ 和 $\$error$ 分别表示起始状态和错误状态，D_{other} 表示其它中间状态的集合.

本文用 XML 文件对不同种类的缺陷模式状态机进行形式化描述，在缺陷检测时根据待测程序的语法、语义特征决定是否创建某类型的状态机实例，并将产生的状态机实例集合置于待测函数控制流入口处；然后在遍历控制流图过程中，根据当前控制流节点的数据流信息更新状态机实例的缺陷状态同时判定是否会发生状态迁移，以检测某种类型的缺陷. 缺陷模式状态机可以用有向图直观地表示，例如 RL 和 NPD 缺陷模式分别可用状态机描述，如图 1 所示.

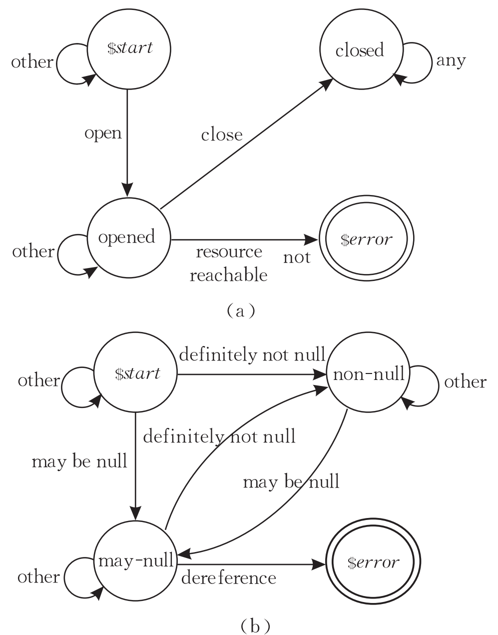


图 1 资源泄漏模式和空指针引用模式的缺陷状态机

2.2 研究动机

路径敏感的缺陷检测方法，从控制流图头节点依次进行状态迭代，每个状态都关联当前控制流节

① 在不引起混淆的情况下，缺陷模式状态机在下文中也简称为缺陷状态机或状态机、缺陷自动机或自动机.

点的所有变量取值信息，称之为状态条件（state condition）．我们用变量的抽象取值[9]来表示状态条件，在数据流迭代过程中不断更新状态条件，就会导致缺陷状态发生迁移，一旦发现状态迁移到 $*error* 就表示程序中存在该类型的缺陷．例如图 2 中代码片段(a)，对其中存在的 RL 缺陷进行检测，采用相同状态合并的路径敏感分析方法，其状态迁移序列如表 1 所示．

```
     void fun(int i, int j){
L0:  f=null;
L1:  if (dump)
L2:      flag=1;
     else
L3:      flag=0;
P1:  if(i>j)    i++; //noise codes
P2:  if(i>10)    j++; //noise codes
L4:  if (dump)
L5:      f=open(…);
L6:  if (flag==1)
L7:      close(f);
L8:  ……//noise codes
     }
```

(a)

```
     void fun(int i, int j){
L0:  f=null;
L1:  if (dump)
L2:      flag=1;
     else
L3:      flag=0;
     //noise codes sliced
     //noise codes sliced
L4:  if (dump)
L5:      f=open(…);
L6:  if (flag==1)
L7:      close(f);
L8:  //noise codes sliced
     }
```

(b)

```
if(flag){
    f=open();
L1: b=0;
L2: a=b;
}
if(a==0){
    close(f);
}
```

(c)

图 2　示例代码片段

表 1　图 2 中代码片段(a)的缺陷状态迁移序列

语句	状态条件(State Condition)	备注
L1	$*start*:	初始状态为 $*start*
L2	$*start*:{*f*[null],*dump*[true],*flag*[1,1]}	根据真分支中数据流，更新状态条件
L3	$*start*:{*f*[null],*dump*[false],*flag*[0,0]}	根据假分支中数据流，更新状态条件
P1	$*start*:{*f*[null],*dump*[true_or_false],*flag*[0,1],*i*[inf],*j*[inf]}	控制流汇合节点的缺陷状态合并策略：相同状态中的状态条件进行合并
P2	$*start*:{*f*[null],*dump*[true_or_false],*flag*[0,1],*i*[inf],*j*[inf]}	在状态条件中添加 *i* 和 *j*，其区间未知
L4	$*start*:{*f*[null],*dump*[true_or_false],*flag*[0,1],*i*[inf],*j*[inf]}	
L5	$*open*:{*f*[notnull],*dump*[true],*flag*[0,1],*i*[inf],*j*[inf]}	真分支中的状态条件更新导致 $*start* 迁移到 $*open*；假分支中隐含的缺陷状态：$*start*:{*f*[null],*dump*[false],*flag*[0,1],*i*[inf],*j*[inf]}
L6	$*start*:{*f*[null],*dump*[false],*flag*[0,1],*i*[inf],*j*[inf]} $*open*:{*f*[notnull],*dump*[true],*flag*[0,1],*i*[inf],*j*[inf]}	缺陷状态合并策略：不同状态不进行合并
L7	$*error*:{*f*[null],*dump*[false],*flag*[1,1],*i*[inf],*j*[inf]} $*close*:{*f*[notnull],*dump*[true],*flag*[1,1],*i*[inf],*j*[inf]}	真分支中状态条件更新导致 $*start* 迁移到 $*error*，$*open* 迁移到 $*close*； 假分支为空语句，不会发生状态迁移；
L8		发现 $*error* 状态，迭代终止

根据表 1 中缺陷状态迁移序列，在 L7 处缺陷状态自动机迁移到 $*error*，这是一个明显的误报．原因在于：L3 之后的控制流汇合处，将真假分支中两个 $*start* 状态进行了合并，导致 flag 与 dump 的关联关系丢失．另外还可以观察到，P1 之后的状态条件中都关联了 *i* 和 *j* 的抽象取值，此类数据流信息对 RL 缺陷检测的结果不会产生任何影响，只会增加状态迭代时的计算量．

通过对上述示例程序的分析，本文的研究动机为：(1) 消除与缺陷检测无关的冗余代码，可以减少控制流图节点数，从而减少状态迭代的次数；(2) 减少状态条件中与缺陷检测无关的变量，可以降低数据流传递和计算时的复杂度；(3) 对控制流汇合节点的缺陷状态合并策略进行优化，可以减少路径合并导致的精度损失．根据研究动机中的三方面需求，我们借鉴需求驱动分析技术的思想，提出了本文提高缺陷检测效率和精度的方法．

3　提高检测效率和精度的方法

需求驱动的静态分析技术[7,10]从程序分析的需求出发，抽取程序语义信息，在进行分析之前降低程序的复杂度，大幅减少了分析过程中的计算量，不仅可以快速准确地复现程序缺陷[11]，而且可以对大型程序中某些特定故障进行检测．借鉴文献[7,10]的思想，本文采用程序切片技术实现了需求驱动的缺陷检测方法，根据不同的缺陷检测目的计算程序的不同切片，从而缩小了缺陷检测的范围，以提高缺陷检测效率．由于程序切片考虑程序中存在的各种依赖关系(数据依赖和控制依赖)，而且任何一个程序

可以与一组程序切片的并集等价，检测每个切片实际就是测试了整个程序，因此基于缺陷的程序切片方法满足了静态分析方法的保守性.

3.1 节首先根据缺陷特征及路径条件建立切片准则，3.2 节应用切片准则对控制流图进行程序切片，然后 3.3 节根据控制流分支节点的路径条件，对缺陷状态添加状态属性，实现优化的状态合并策略，最后在 3.4 节根据上述方法给出一个完整的实例分析流程.

3.1 基于缺陷特征及路径条件生成的切片准则

程序切片是一种分析和理解程序的技术，通过对源程序中的每个兴趣点分别计算切片来达到对程序的分析和理解. 程序切片的原理和方法由 Weiser 于 1979 年在其博士论文中首次提出[8]，Weiser 认为程序切片与人们在调试程序时所做的智力抽象是相对应的，他定义程序 P 的切片 S 是一个可执行的程序，这个切片程序在某个功能属性上与 P 完全等效. 根据程序分析和理解时不同的兴趣点，可以定义相应的切片准则(slicing criteria)，根据不同的切片准则可以“按需”地对源程序进行功能切片. 正是基于这种“简化问题、缩小目标范围”的原则，程序切片技术成为提高静态分析效率的有效途径之一.

根据路径敏感缺陷检测方法的特点，切片准则可以从缺陷特征及路径条件两个角度获得.

定义 4. 缺陷特征(defect feature). 给定缺陷模式 fsm，它可以检测某一程序属性 $feature$ 是否违反了程序语法或语义规则，该 $feature$ 对应的程序变量即为缺陷模式 fsm 的缺陷特征，记为 $Df(fsm)$.

定义 5. 节点关联变量(vex related variable). 对于控制流图 $G=(N,E,n_0,n_f)$，$\forall n\in N$，$\exists Var\subseteq Stmt(n)$，其中的 Var 即为节点 n 的关联变量，记为 $RelVar(n)$.

定义 6. 路径条件(path condition). 指在控制流迭代过程中会产生路径分支的数据流值. CFG 中只有条件分支及循环节点可以产生新的路径，因此路径条件只能在 $SelectionStmt$ 语句中产生，定义条件分支头节点 n 的路径条件 $Pc(n)=\{var\,|\,var\in RelVar(n)\wedge Stmt(n)\in SelectionStmt\}$，它包含控制流图中条件分支及循环头节点关联的所有变量.

缺陷特征可以理解为缺陷检测是否与程序中某个变量相关，例如 NPD 模式必然与一个指针变量相关，RL 模式必然与一个资源句柄相关；路径条件则是程序中可能会产生新路径的条件分支语句所关联的变量，例如 if-else、switch、while 等语句块中关联的条件变量.

通过对程序控制流的进一步观察发现，并非所有的路径条件都影响缺陷检测结果，如图 2 程序片段(a)中 P1、P2 处的条件分支不会对 RL 的检测结果产生任何影响，此类路径条件不应作为切片准则. 另外，变量间的数据依赖关系也会影响切片准则的生成：考虑图 2 程序片段(c)，L2 处赋值语句使得数据流迭代计算时 a 的数据流值依赖于 b，同时也为 $a=0$，$flag=$true 数据流值建立了关联关系；由于 b 并非路径条件，如果将 L1 处语句切片掉会导致上述变量间的关联关系丢失，有可能导致数据流迭代的断流.

本文方法通过提前进行区间运算，避免了赋值语句形成的数据依赖关系对程序切片的影响. 我们采用数值区间来表示变量的抽象取值，在程序切片前已经通过区间运算获取了 a 的区间信息，即使 L1 处切片掉也不会对缺陷检测产生影响. 基于上述分析，本文切片准则定义如下：

$$SCSet=Df(fsm)\cup\{Pc(n_i)\,|\,Stmt(n_i)\in SelectionStmt\wedge Df(fsm)\in RelVar(n_i)\}.$$

算法 1. 切片准则生成算法.

$BranchList$：程序中所有的条件分支节点.

$IsPc(n)$：标志条件分支节点 n 所在的条件语句块 $Stmt(n)$ 中是否包含切片准则，以标志后续的切片算法中是否可以将整个分支语句块切片掉.

$GetBranchVexList(n)$：获取当前条件分支 n 所在分支的所有语句块.

$SkipBranchVexList(n)$：如果条件分支 n 所在的条件语句块 $Stmt(n)$ 中不包含切片准则，则可以跳过当前分支语句块，继续在分支后续节点中遍历查找路径条件.

```
输入：控制流图 G 和缺陷模式 fsm
输出：切片准则 SCSet
1. 获取缺陷特征
Var=Df(fsm);
SCSet.add(Var);
2. 计算路径条件
 2.1. 查询控制流中所有条件分支头节点，生成条件
      分支列表
 for each n∈G
   if(OutDegree(n)>1) then
     BranchList.add(n);
 end
 2.2. 遍历条件分支列表，查询路径条件
 Reverse(BranchList);
 for each n∈BranchList
   IsPc(n)=false;
```

```
    for each n′∈GetBranchVexList(n)
      if (OutDegree(n′)<2) then
        if(Var∈RelVar(n′)) then
          SCSet.add(Pc(n));
          IsPc(n)=true;
          break;
      else if (IsPc(n′)) then
        SCSet.add(Pc(n));
        IsPc(n)=true;
        break;
      else
        SkipBranchVexList(n′);
    end
  end
```

算法1的主要开销是步2中路径条件的查询，假设CFG节点数为N，分支节点数为Q，每个分支语句块的节点数为$P(P\ll N)$：步2.1对CFG节点进行遍历，将所有条件分支节点加入到*BranchList*列表中，复杂度为$O(N)$；步2.2首先将条件分支列表*BranchList*逆序，然后依次遍历每个条件分支语句块*GetBranchVexList*()中是否包含缺陷相关变量*Var*，一旦当前节点关联变量包含*Var*，则将当前路径条件$Pc(n)$加入到*SCSet*，同时设置$IsPc(n)=$ true标志当前条件语句块中包含切片准则；如果存在嵌套分支结构，通过判断内层分支语句块的$IsPc(n')$，决定是否可以跳过内层分支语句块的分析*SkipBranchVexList*()，复杂度为$O(Q\times P)$. 因此，算法1的复杂度为$O(N)+O(Q\times P)$.

3.2 基于缺陷的程序切片方法

程序切片算法比较经典的有Weiser基于数据流方程的算法、Ottenstein K J和Ottenstein L M[12]以及Horwitz[13]的基于程序依赖图的可达性算法，以及基于系统依赖图的上下文敏感算法等. 本文采用Weiser基于数据流方程的算法，按如下基本规则进行切片：遍历CFG节点，观察当前节点数据流集合中是否包含算法1切片准则中的相关变量，以确定该节点是否应该被切片，并根据每个节点的切片标志重构控制流图.

由于切片准则*SCSet*中可能包含全局变量，而全局变量可以当作函数的“隐式”参数，因此对于程序中存在函数调用的语句节点，在切片时需要特殊处理：如果切片准则中包含全局变量，且被调用函数的函数摘要中存在对该全局变量的前置约束①，则该函数调用语句不应被切片，否则会造成缺陷检测的漏报；对于其它类型的函数调用语句，可以统一按上述规则进行切片，即根据函数调用点的实参与切片准则的包含关系决定是否进行切片.

对于C语言中较为常见的复杂数据类型成员引用，如结构体成员、数组元素等，由于本文采用的是非域敏感分析，从而切片准则中的相关变量是整个的结构体或数组变量(例如int a[]，struct s)，而非具体的某个成员(例如$a[i]$，s.field)，因此包含此类变量的程序语句可以统一按基本规则进行切片. 本文切片算法的局限性在于C语言中的指针变量：指针有可能对切片准则中的变量生成复杂或难以发现的别名关系，进一步地会对本文的切片算法产生干扰；为了减少指针操作对本文切片精度的影响，我们已实现了别名分析，如何提高别名分析的精度也是我们未来工作的一个重要方面. 总之，本文切片算法的根本原则是“保守切片”，即对无法确定是否应该被切片掉的语句予以保留，以保证缺陷检测过程中与缺陷相关的程序语义的完备性.

算法2. 程序切片算法.

sliceProcCall：为false时表示对函数调用语句不进行切片，默认为true.

GetScope(*var*)：获取程序中变量*var*的作用域(分为*SourceFileScope*，*MethodScope*，*LocalScope*等).

SliceBranchVexList(n)：如果条件分支节点n所在语句块*Stmt*(n)不包含切片准则，则将n所在的整个分支语句块切片掉，以提高切片效率.

ExistProcCall(n)：查询节点n所在的程序语句*Stmt*(n)中是否包含函数调用.

Slice(n)：将n从控制流图中切片掉，将其所有前驱与后继节点相关联.

输入：控制流图G，切片准则*SCSet*

输出：切片后的控制流图G'

1. 查询切片准则中是否包含全局变量，以标志函数调用语句是否应该被切片掉

```
sliceProcCall=true;
for each var∈SCSet
  if(GetScope(var)==SourceFileScope)
    sliceProcCall=false;
    break;
end
```

2. 进行程序切片

for each $n\in G\wedge n\neq n_0\wedge n\neq n_f$

① 本文采用函数摘要技术实现函数间分析：根据函数调用关系，为被调用函数生成前置约束信息，标志其是否存在与全局变量相关的缺陷信息；在函数调用点，将函数摘要根据上下文实例化，不仅可以得到上下文敏感的数据流信息以提高静态分析的精度，并且根据此前置摘要信息可以提高切片精度. 由于本文讨论重点并非函数间分析技术，因此算法中并没有对函数摘要相关内容进行描述.

```
  if(RelVar(n)∩SCSet==null) then
    if(!IsPc(n)) then
      SliceBranchVexList(n);
    else if(!ExistProcCall(n)∨sliceProcCall)
      Slice(n);
end
```

算法2首先检测切片准则中是否存在文件作用域的全局变量,并设置 *sliceProcCall* 标志是否需要对函数调用语句进行切片;然后遍历CFG,计算每个节点关联变量与切片准则 *SCSet* 的交集,如果交集为空,再根据函数调用语句的切片规则,决定当前节点是否应该被切片掉 *Slice*().为了进一步提高算法效率,如果被切片的节点为条件分支节点,则可以利用算法1中设置的条件语句块是否包含切片准则标志 *IsPc*(),确定能否切片掉整个分支语句块 *SliceBranchVexList*().上述切片算法中,切片准则 *SCSet* 中变量数为常量,因此步1中for循环的复杂度为$O(1)$;在控制流图创建过程中可以为每个节点n添加是否存在函数调用语句的附加属性,因此 *ExistProcCall*(n)的复杂度为$O(1)$;*Slice*(n)将n的所有前驱与后继相关联,复杂度取决于n的出度与入度之和,在实际程序中为常值;*SliceBranchVexList*(n)直接将分支语句块的前驱及后继相关联,复杂度取决于从分支入口节点遍历至汇合节点处的节点数,即分支语句块的节点数P,因此复杂度为$O(P)$.整个算法的复杂度取决于for循环中CFG节点数N,因此算法2的复杂度为$O(N\times P)$.

应用程序切片技术后,控制流图节点数及状态条件中变量数会相应地减少,从而分别影响缺陷状态迭代的次数及每次迭代时的计算量,因此可以通过切片效果对效率提升情况进行粗略估计.切片效果η可以理解为切片掉节点数与原节点数的比值,η越接近1则表明切片效果越好.假定源程序中控制流节点总数为ϕ,切片程序中节点总数为φ,则切片效果$\eta=\frac{\phi-\varphi}{\phi}$可以粗略反映由于节点数、变量数减少而引发的效率提升情况.当然,实际的缺陷检测还受到其它因素的影响,如状态条件的计算和传递、状态迁移条件是否成立的判断等,因此实际的效率提升值会小于η.

3.3 一种基于切片的缺陷状态合并策略

为进一步提高检测精度,本文提出一种基于切片的缺陷状态合并策略,根据控制流分支节点的路径条件,在缺陷状态上添加状态属性(state attribute),从而可以根据状态属性来决定在控制流汇合节点是否需要进行状态合并.该合并策略有选择地在汇合节点进行状态合并,保存了与缺陷特征(defect feature)有关的路径信息从而减少误报.

定义7. 状态属性(state attribute).给定缺陷状态S及当前CFG节点n,如果 *Stmt*(n)∈ *SelectionStmt*,则S的状态属性 *Attr*(S)= *RelVar*(n).

算法3. 一种基于切片的缺陷状态合并算法.

S_T、S_F:真假分支中的缺陷状态.

StateCompute(S):根据当前节点的数据流信息,更新S的状态条件,并判断状态迁移情况.

MergeState(S_T,S_F):合并真假分支汇聚处的缺陷状态,同时将相同的状态条件进行合并.

IsSameTypeState(S_T,S_F):判断分支汇聚处真假分支的缺陷状态S_T、S_F是否为相同类型的状态.

输入:分支节点n及缺陷状态S

输出:汇合节点n'处的缺陷状态S'

1. 遍历条件分支语句的真假分支,进行缺陷状态迭代

```
   Var=Pc(n);
   Attr(S_T)=Attr(S_F)=Var;
   StateCompute(S_T);
   StateCompute(S_F);
```

2. 在条件分支语句的汇合节点,按规则合并真假分支中的缺陷状态

```
   if(!IsSameTypeState(S_T,S_F)) then
     S'=S_T∪S_F;
   else if(Attr(S_T)⊂SCSet∨Attr(S_F)⊂SCSet) then
     S'=S_T∪S_F;
   else
     S'=MergeState(S_T,S_F).
```

上述算法中,步1首先计算分支节点n的路径条件 *Pc*(n),为初始状态S添加状态属性,然后根据真假分支中不同的数据流值进行状态迭代 *StateCompute*()并更新S的状态条件;状态迭代的复杂度取决于分支语句块的节点数P,因此复杂度为$O(P)$;步2首先通过 *IsSameTypeState*(S_T,S_F)判断汇合节点是否为相同状态,状态合并前再进一步检查状态属性 *Attr*(S)是否包含于切片准则,以确定是否需要状态合并 *MergeState*(S_T,S_F);状态合并实际上是对相同缺陷状态的状态条件进行合并,复杂度取决于执行路径上相关变量的数量,在实际程序中这是一个常值,因此算法3的复杂度为$O(P)$.

通过前文对算法1~3复杂度的描述,本文算法复杂度主要依赖于控制流图节点数N、条件分支数

量 Q 及每个条件分支语句块包含的节点数 P，实际程序中 $Q \ll P \ll N < \lambda$，λ 为常量，因此本文算法复杂度较低.另外，算法 1 生成了后续算法 2、3 所用到的大部分数据，且该部分数据可以作为控制流节点的静态属性进行传递，从而进一步减少了算法 2、3 的计算量.

3.4　实例分析

对于图 2 中程序片段(a)，应用前文描述的程序切片方法和缺陷状态合并策略对 RL 模式进行检测，其简要的分析流程如下：

1. 得到 RL 缺陷模式特征：$Df(\text{RL})=\{f\}$；

2. 应用算法 1，得到 L4、L6 处路径条件 $Pc(\text{L4})=\{dump\}$，$Pc(\text{L6})=\{flag\}$，因此切片准则 $SCSet(\text{RL})=\{f, dump, flag\}$；

3. 应用算法 2 将程序片段(a)进行切片，即可切片掉 P1、P2 处与 RL 分析无关的代码，得到程序片段(b)；

4. 应用算法 3，(b)片段的状态迭代序列如表 2.由于切片掉了 P1、P2 处的冗余语句，因此状态条件中与缺陷检测无关的变量 i 和 j 被切片掉；在 L4 处由于 $Attr(Start)=\{dump\} \subset SCSet(RL)$，因此 L2 分支产生的两个 $\$start$ 状态在控制流汇合节点 L4 不进行状态合并；在 L6 处，$\$start$ 状态保留了 $flag$ 与 $dump$ 的关联关系，不会执行真分支中的 $Close$ 操作，因此不会产生误报.

表 2　应用本文方法后图 2 中程序(b)的状态迁移序列

语句	包含状态属性的状态条件	备注
L1	$\$start$：{f[null]}	$SCSet=\{f, flag, dump\}$ $Attr(start)=Pc(\text{L1})=\{dump\}$
L2	$\$start(dump)$：{f[null], dump[true], flag[1,1]}	
L3	$\$start(dump)$：{f[null], dump[false], flag[0,0]}	
L4	$\$start$：{f[null], dump[true], flag[1,1]} $\$start$：{f[null], dump[false], flag[0,0]}	$Attr(start) \subset SCSet$：因此不进行状态合并
L5	$\$open$：{f[notnull], dump[true], flag[1,1]}	$\$start$ 在真分支中迁移为 $\$open$ 状态
L6	$\$start(dump)$：{f[null], dump[false], flag[0,0]} $\$open$：{f[notnull], dump[true], flag[1,1]}	不同状态，因此不进行状态合并
L7	$\$close$：{f[notnull], dump[true], flag[1,1]}	$\$open$ 在真分支中迁移为 $\$close$ 状态
L8	$\$start(dump)$：{f[null], dump[false], flag[0,0]} $\$close$：{f[notnull], dump[true], flag[1,1]}	假分支中的 $\$start$ 与真分支中的 $\$close$

4　实验结果及分析

本节首先介绍缺陷检测系统(Defect Testing System, DTS)测试框架及缺陷检测的整体分析流程，给出我们 DTSGCC 的实验环境设置，然后对实验数据进行分析，分别以检测时间和误报率作为衡量效率和精度提升情况的指标.

4.1　实验环境

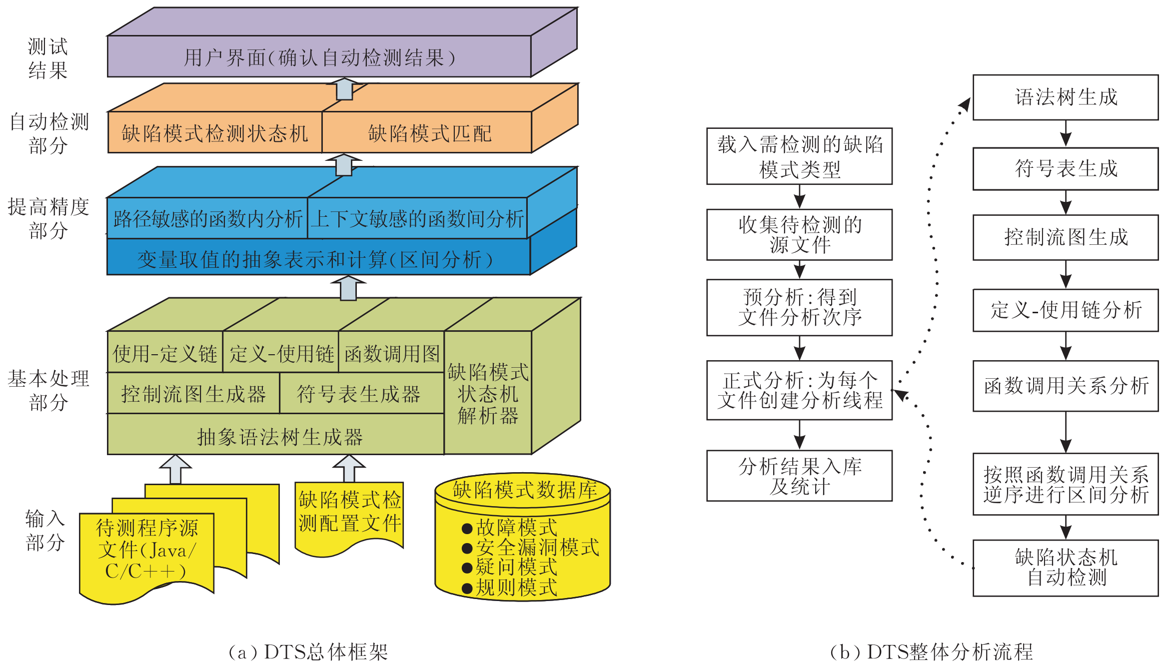
DTS[3,14] 采用状态机来统一描述缺陷模式，目前能检测的缺陷模式包括 200 多种，且可以根据用户需求进行快速模式定制.将缺陷模式按其造成的影响分为四类：故障模式、安全漏洞模式、疑问代码模式、规则模式，本文讨论的 RL 和 NPD 模式属于故障模式.DTS 总体框架及整体的缺陷检测流程如图 3：

(1) 对外接口.输入部分收集待测程序源文件，并从缺陷模式配置文件中读取待检测的缺陷模式；测试结果模块负责将缺陷检测结果写入缺陷库中，并提供 UI 界面以方便人工缺陷确认；

(2) 基本处理模块是整个静态缺陷检测的基础.首先通过语法分析器(CParser)生成待测代码的抽象语法树(Abstract Syntax Tree, AST)，然后遍历 AST 生成符号表(Symbol Table)，其中包含了源程序中各词法单元的类型、作用域等信息，进一步生成源程序中各函数的控制流图，并基于控制流图进行定义-使用链分析；

(3) 缺陷模式自动检测模块按照函数调用关系的拓扑逆序，依次对每个函数进行缺陷自动机状态迭代.首先调用缺陷模式状态机解析器加载待检测的缺陷模式，然后根据缺陷模式的创建条件遍历函数语法树，将创建的自动机实例集合置于函数控制流入口节点，最后根据每个控制流节点数据流分析的结果更新所有缺陷状态的状态条件同时进行状态迁移判断，直至缺陷状态迁移为 Error 表示发现某类型的缺陷，或者迁移到 End 状态表示未发现缺陷并自动销毁状态机实例；

(4) 提高精度模块.区间分析的结果直接影响整个缺陷检测的精度，因此 DTS 分别实现了基于函数摘要的上下文敏感的函数间分析，及基于缺陷状态合并的路径敏感的函数内分析，本文的研究点即为如何使后者效率及精度更好.



(a) DTS总体框架 (b) DTS整体分析流程

图 3 DTS 总体框架及 DTS 整体分析流程

DTS 除对外接口外,其它模块对用户是透明的,且底层改动不会影响上层处理逻辑,这种松耦合架构使其具有良好的扩展性,例如通过自定义缺陷模式及第三方语法生成器(Compiler Compiler),可以快速生成支持某一语言族程序[①]的缺陷检测工具.

GCC 作为 Linux 系统中开发 C 语言程序的主流编译器,通过对 ANSI C 标准进行语法扩展获得了更高灵活性的同时,其程序语义也更加复杂,导致其出现软件缺陷的机率也更高,且难于检测.本文基于上述 DTS 框架设计实现了路径敏感的静态缺陷检测工具 DTSGCC,不仅可以检测 ANSI C 标准的源程序,而且可以针对 Linux 中 GCC 标准的开源工程进行检测:首先修改工程配置文件中的预处理选项,并调用修改后的配置文件编译原工程,使之保留编译过程中产生的中间文件(后缀名为.i);然后,以此预处理后的中间文件作为 DTSGCC 的输入,即可以按照图 3 中的分析流程进行缺陷检测[②].

本文使用 DTSGCC1.0 采用不同的分析方法对 Linux 中 10 个开源工程进行了缺陷检测对比实验,扫描的缺陷模式包括 RL 和 NPD.这 10 个开源工程中,源代码量最小的 Combine 为 1.6 万行,最大的 Binutils 为 103 万行.实验所使用电脑基本配置为 Intel E2160 1.8GHz CPU、2GB 内存、Windows XP 操作系统.实验过程中使用两种不同的分析方法:

方法 1. 文献[3]提出的相同状态合并的路径敏感方法;

方法 2. 本文提出的基于程序切片的路径敏感方法.

4.2 检测结果及分析

基于以上实验设置,对静态缺陷检测结果进行了人工确认,结果如表 3 所示.其中,文件数只统计后缀为.c 或.h 的源文件,源代码行数为去除了空行后的统计结果.DTSGCC 并不对 C 源文件直接进行检测,而是通过 GCC 编译器对其进行预处理(包括头文件展开、条件编译执行、宏定义替换),对得到的中间文件进行处理,因此实际测试代码量为中间文件的代码量.应用本文程序切片方法后,控制流图节点数的变化可以反映程序切片的效果,从而可以粗略地估计效率提升情况.

统计结果表明,对表 3 中 10 个工程检测其 NPD 和 RL 缺陷,共检测源代码量 154 万行,中间文件代码量 761 万行,方法 1 用时 13.52h,共上报 2228 个检查点(Inspection Point,IP),而本文方法用时 11.73h,IP 数减少为 1896 个,检测效率提高了 13.28%,误报率降低了 5.90%.下文分别对效率、

① 以 ANSI C 为基础语言,通过对其基本语法进行扩展,目前已经衍生了众多“类 C 语言”,如 keil、gcc 等.DTS 正力图实现一个类 C 语言的语法全集,从而可以分析任意类 C 语言的源程序.

② 由于条件编译的存在,如果将待测工程中用到的 Linux 系统库文件移植到 Windows 系统中,然后在 Windows 环境下进行编译得到的中间文件往往是不精确的.本文中间文件生成是在 Linux 环境下得到的,从而可以得到精确的程序执行语义,缺陷检测的结论也更有意义.

精度提升情况及 DTS 缺陷检测工具的有效性进行分析.

(1) 效率提升分析. 3.2 节从理论上分析了应用本文方法后效率提升情况与切片效果的关系. 表 3 统计结果表明，在切片效果 $\eta=41.19\%$时，实际的效率提升为 13.28%，与我们的分析结论是相吻合的.

另外，对 Combine 和 Antiword 等代码量相对较小的工程，采用程序切片方法后对效率影响不大，甚至极端情况下分析时间会略有增长，而对于代码量较大的工程则不会出现类似情况，主要原因是代码量较小时切片准则计算及程序切片所用时间与方法 1 中冗余节点的状态迭代计算时间抵消了.

表 3 对比实验结果 1

工程名称	源文件数	源代码行数	中间文件行数	确认故障数	相同状态合并策略的结果			基于程序切片合并策略的结果			误报减少数
					节点数	分析时间/s	报告检查点	节点数	分析时间/s	报告检查点	
Combine-0.3.4	74	16701	32892	8	6779	153	28	3360	134	21	7
Antiword-0.37	80	24315	126058	28	16742	316	110	9699	265	102	8
Make3.81	60	23284	98810	69	12141	276	134	8072	241	113	21
Readline-6.1	74	21460	109775	30	11687	1552	86	7597	1279	59	27
Wdiff-0.6.3	81	23280	29089	2	1787	69	6	1242	61	3	3
gnome-media-2.14.0	118	27457	1522037	14	12753	7093	92	7161	5892	58	34
UUCP-1.07	251	52595	501219	85	32248	936	283	18883	889	278	5
Bash-4.1	355	96489	802269	223	60426	5910	848	37021	5419	663	185
Openssl-0.9.8	1118	226722	3110791	179	130401	22035	380	74358	18877	399	−19
Binutils-2.20.1	1668	1035045	1284531	113	44959	10341	261	26639	9159	200	61
合计	3879	1547348	7617471	751	329923	48681	2228	194032	42216	1896	332

(2) 精度提升分析. 上述分析表明本文方法可以减少 5.90% 的误报，但对于 Openssl 工程，误报反而出现了增长的情况. 经过人工确认，这种精度"反提升"是由于部分文件分析时内存溢出导致分析失败造成的. 对中间代码进行逐个排查发现，导致内存溢出的原因是预处理后的中间文件过于复杂，另外一些成员数量非常多的 enum、struct 类型也是导致分析失败的另一重要原因. 一个非常典型的分析失败的例子如下：

位于 openssl-0.9.8/crypto/bn/bn_asm.c 源文件中 435 行的宏定义 mul_add_c，在 550 行的函数 bn_mul_comba8 中大量使用，经过预处理后宏定义的展开形式非常复杂①，这种复杂的中间代码有可能导致本文语法分析器创建 AST 时失败，或者造成在数据流分析过程中由于控制流节点众多而导致迭代分析失败，从而造成整个源文件的分析失败；另外，由于 C 工程中的交叉引用关系，使得依赖于此文件的其它文件也分析失败，从而导致分析结果中上报 IP 的数量减少.

而采用切片算法后，由于切片后控制流图节点数减少，原来由于第 2 种原因分析失败的文件将有可能正常分析，因此会上报更多的 IP，也就出现了 IP 数不减反增的现象.

对于可以正常分析的文件，对其缺陷检测结果进行人工确认后，发现了引起误报的几类主要原因. 一是由于 DTSGCC 并未对数组、结构体成员建模，即没有实现域敏感分析[15-16]，而实际工程中结构体、数组应用非常广泛，导致此类误报较多，例如 combine 工程 src\index.c 文件会在 500 行上报一个 NPD：

```
487: curr_record_position = chunk[key_position].first_record;
                                  //结构体成员信息无法精确计算
488: curr_record_entry = NULL;
489: while (curr_record_position != 0){
      //循环条件信息不准确，导致循环内区间计算不精确
     ……
494:   if (curr_record_entry == NULL)
                      //静态分析的保守性导致该判断不准确
495:       return EXIT_FAILURE;
     ……
     }
500: curr_record_entry->next_same_key=
                    new_record_position; //NPD 误报
```

由于 487 行将结构体成员赋值给 curr_record_position，在 489 行的循环无法判断循环迭代情况，因此 494 行的条件判断只能保守性计算出 curr_record_entry 的值为空或非空的集合，因此 500 行会出现 NPD 误报. 另一类误报则是由于本文方法采用的上下文敏感分析策略不完善，导致程序中的函数调用节点无法精确计算其对上下文数据流迭代的影响，从而状态迁移的条件无法精确计算，

① 如位于 563 行的宏定义 mul_add_c(a[0], b[0], c1, c2, c3)，其展开形式为各种与运算、移位运算、条件运算、复合语句块的集合. 针对这种非常复杂的中间语句，可以考虑通过程序转换技术(program transformation)对其进行复杂度约减.

导致误报.

第1类误报的消除需要在DTS框架中实现域敏感分析,从而对结构体和数组成员都可以独立分析;第2类误报的消除需要改进现有的上下文敏感分析策略,避免由于函数调用引起状态迭代错误.尽管存在上述不足,实验统计结果表明误报率由66.29%降低到了60.39%,减少了6%左右,说明本文方法对精度提升的效果较好.

(3) DTSGCC的有效性.从表3可以观察到,DTSGCC真正分析的中间文件代码量一般是源代码量的几倍甚至十几倍,其中binutils、openssl、gnome-media等工程更是达到了几百万行代码.在我们的实验环境中,DTSGCC内存占用最多800MB左右,CPU 50%左右,百万行代码量平均可以在1.54h内完成检测.ASTRÉE的统计结果表明,要检测程序的运行时错误(Run Time Error,RTE),万行代码的平均检测时间为数小时①.我们的DTSGCC与Saturn[17]相比,Saturn检测Binutils工程内存泄露缺陷的时间为4h左右[18],而我们检测该工程的RL和NPD故障用时2.87h左右②.上述测试结论验证了DTSGCC在检测代码量较大工程时,与同类静态检测工具相比检测效率更高.

实验1的分析结论表明,应用本文程序切片方法后,控制流图节点数减少40%左右,从而在路径敏感的缺陷状态迭代过程中总的状态迭代次数大幅减少,因此缺陷检测时间减少13%左右.为了验证缺陷无关的数据流信息在状态迭代时对内存占用的影响,我们实现了不同的方法与实验1相同的实验设置进行对比实验.

方法1. 在程序切片的同时消除控制流节点中与缺陷无关的变量.

方法2. 只进行程序切片而不消除无关变量,实验结果如表4所示.

表4 对比实验结果2

	时间/h	内存占用/MB	分析失败文件数
方法1	13.29	275	37
方法2	11.73	347	55

方法1较方法2虽然检测效率降低了13%,但内存占用率降低了20.75%,因此方法1在检测过程中由于内存溢出而分析失败的文件数量也明显减少.方法1检测效率降低的原因是显而易见的:在每次状态迭代过程中,方法1都需要查询当前控制流节点的关联变量集合,通过与切片准则的对比来删除缺陷无关变量;然而对大型程序来说,控制流节点关联的变量集合较大,对其中某个变量的查询和删除操作的开销抵消了程序切片后带来的正面影响.实验2的分析结论表明,在代码量较大的缺陷检测过程中,通过本文方法切片掉缺陷无关变量,可以有效减少内存使用,降低缺陷检测过程中某些复杂文件分析失败的几率.

5 相关工作对比

路径敏感分析方法最早应用于模型检测领域,它可以对建模后的程序进行全路径覆盖测试,每次执行一条路径并检测当前路径上的错误[19-22].SLAM[19]是一个路径敏感的模型检测工具,它基于迭代求精策略,其初始状态是对程序语义的粗略近似,在迭代过程中不断更新状态信息,通过定理证明检测不可达路径、验证程序的时序安全属性,但它采用的谓词抽象理论存在状态数爆炸的隐患.结合谓词抽象和符号执行方法,文献[20-21]将路径上的每个谓词转化为一个布尔变量,将路径上的数据流信息抽象为符号执行状态,通过约束求解理论进行不可达路径判断,并通过符号执行状态间的关系检测程序缺陷,但该类方法中约束求解所能处理的程序语义复杂度是其潜在的瓶颈.模型检测方法虽然能够发现程序深层次的错误,但它需要对软件的运行环境进行建模,这也限制了其在大型软件缺陷检测中的应用,而基于数据流分析框架的静态分析技术可以很好地解决上述问题.

基于数据流迭代的路径敏感分析方法可以记录程序中的不同路径信息,减少控制流汇合节点数据流合并带来的精度损失.由于全路径分析时的路径爆炸会导致分析代价较大,因此实际采用的往往是部分路径敏感分析策略(partial path-sensitive):(1)有选择地对控制流汇合节点进行数据流合并;(2)通过路径可达性分析防止不可达路径上的数据流传递.策略1的典型应用是ESP[2],它是微软ESC研究小组开发的路径敏感的轻量级静态测试工具,通过在模型检测的属性状态上增加程序执行符号状态信息,跟踪属性状态和程序执行符号状态间的关联关系,并将其用于排除不可达路径来提高数据流分析的精度,其不足之处在于它的状态合并策略是

① http://www.astree.ens.fr/
② 由于测试环境不同,缺陷检测的关注点不同,Binutils工程的版本不同,该检测效率对比仅能粗略反映我们工具与同类工具的效率差异.

静态的，无法在关键路径合并处改变合并策略，而本文所述基于切片的状态合并策略可以根据切片准则和数据流分析结果动态生成. Dhurjati 等人[4]对ESP中上述不足加以改进，在迭代求精过程中更新状态合并准则并预测精度损失原因，避免了关键路径的合并而减少精度损失. 文献[3]采用变量的抽象取值来表示状态条件，在控制流汇合节点通过合并相同状态中的状态条件来避免路径爆炸. 本文与文献[3-4]相比，都是有选择地对控制流汇合节点进行状态合并，但文献[3]方法仅根据缺陷状态是否相同来进行状态合并，有可能导致与缺陷相关的路径信息丢失，而本文通过切片准则识别出在哪些节点可以安全地进行状态合并；文献[4]采用的迭代求精策略利用首次缺陷检测的结果，更新状态合并策略然后进行二次迭代计算，实际上会进行两次或多次分析，会导致更大的计算量，且迭代有可能面临不终止的情况，而本文的程序切片操作某种意义上也可以视为一种迭代求精技术，但实施代价较低且可以很快终止.

策略2路径可达性分析可以发现程序语义中不可达的程序状态，在缺陷检测工具、程序验证工具及测试用例生成工具中都有广泛使用. Saturn[17,23]将路径上的每个谓词转化为一个布尔变量，利用SAT求解器进行不可达路径判断，能针对C程序进行路径敏感的缺陷检测，但它采用启发式方法以跟踪程序中的关键谓词，可能忽略程序中某些平凡属性，因此其测试结果不是可靠的. Bodik[24]提出一种低代价的基于检测静态边关联的查找不可达路径方法，用于提高定义使用分析(define-use analysis)的精度. Fischer[25]通过在数据流分析半格中元素上增加谓词信息来提高数据流分析精度，实际上这些谓词将程序路径集合进行了划分，从另一个角度也可以认为是记录了不同的路径信息，该信息可用于不可达路径判断. 文献[26]将控制流图中分支节点的条件特征分为4类：相同/相反的条件、互斥的条件、检测-执行逻辑、循环标记，通过识别程序中这四类可以生成不可达路径的模式来检测不可达路径. Ammons等人[27]使用路径形态特征识别出程序中的“热门路径”(hot paths)，在CFG中增加节点避免此类热门路径上的数据流合并，从而通过传统的数据流分析得到了路径敏感的分析结果，这是一种典型的以效率换精度的方法. SLR[5]通过路径可达性分析得到不可达路径上相关节点，然后重构控制流图使该类节点不再位于同一路径，从而在程序语义不变的前提下去除了不可达路径. 本文方法与文献[5]相比，虽然没有消除不可达路径，但通过程序切片技术实现了基于切片的状态合并策略，从而达到了同样目的.

6　小　结

本文从提高路径敏感缺陷检测方法的效率和精度出发，借鉴需求驱动的程序分析思想，将程序切片技术应用于静态缺陷检测，提出一种基于缺陷的程序切片方法，切片程序在缺陷检测时与源程序完全等价，减少了状态迭代时的计算量. 为了进一步减少误报，根据控制流分支节点的路径条件生成缺陷的状态属性，有选择地对控制流汇合节点进行状态合并，减少了路径合并导致的与缺陷检测相关的精度损失. 实验结果表明，本文方法在检测代码量较大程序时，可以提高13%左右的分析效率，并减少6%左右的误报.

由于GCC语法的复杂性，可能导致切片准则计算不准确，得到的切片程序可能仍然包含冗余语句，虽然对分析效率产生影响，但并不妨碍本方法应用于实际的缺陷检测. 将来的工作主要有两个方面：一方面是研制较为精确的程序切片算法，进一步减少切片不准确对缺陷检测效率的影响；另一方面考虑如何实现域敏感分析方法，减少由于结构体、数组成员引用造成的误报. 如何提高指针别名分析的精度，以减少指针操作对切片精度、分析精度的影响，也是今后的一个研究重点.

参　考　文　献

[1] Rice H. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, 1953, 74(2): 358-366

[2] Das M, Lerner S, Seigle M. ESP: Path-sensitive program verification in polynomial time//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002). Berlin, Germany, 2002: 57-68

[3] Xiao Qing, Gong Yunzhan, Yang Zhaohong, Jin Dahai, Wang Yawen. Path sensitive static defect detecting method. Journal of Software, 2010, 21(2): 209-217(in Chinese)
(肖庆，宫云战，杨朝红，金大海，王雅文. 一种路径敏感的缺陷检测方法. 软件学报，2010，21(2)：209-217)

[4] Dhurjati D, Das M, Yang Y. Path-sensitive dataflow analysis with iterative refinement//Proceedings of the Static Analysis Symposium (SAS 2006). Seoul, Korea, 2006: 425-442

[5] Balakrishnan G, Sankaranarayanan S, Ivan F, Wei O, Gupta A. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement//Proceedings of the Static Analysis Symposium (SAS 2008). Valencia, Spain, 2008: 238-254

[6] Thakur A, Govindarajan R. Comprehensive path-sensitive data-flow analysis//Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization(CGO 2008). Boston, Massachusetts, 2008: 55-63

[7] Duesterwald E, Gupta R, Soffa M. A practical framework for demand-driven interprocedural data flow analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), 1997, 19(6): 992-1030

[8] Weiser M. Program slicing//Proceedings of the 5th International Conference on Software Engineering (ICSE 1981). San Diego, California, 1981: 439-449

[9] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints//Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977). Los Angeles, California, 1977: 238-252

[10] Le W, Soffa M. Refining buffer overflow detection via demand-driven path-sensitive analysis//Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007). San Diego, California, 2007: 63-68

[11] Manevich R, Sridharan M, Adams S, Das M, Yang Z. PSE: Explaining program failures via postmortem static analysis//Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004). California, USA, 2004: 63-72

[12] Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment//Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. 1984: 177-184

[13] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), 1990, 12(1): 26-60

[14] Yang Zhao-Hong, Gong Yun-Zhan, Xiao Qing, Wang Ya-Wen. A defect model based testing system. Journal of Beijing University of Posts and Telecommunications, 2008, 31(5): 1-4(in Chinese)
(杨朝红，宫云战，肖庆，王雅文. 基于软件缺陷模型的测试系统. 北京邮电大学学报，2008，31(5)：1-4)

[15] Pearce D, Kelly P, Hankin C. Efficient field-sensitive pointer analysis of C. ACM Transactions on Programming Languages and Systems (TOPLAS), 2007, 30(1): 4-es

[16] Albert E, Arenas P, Genaim S, Puebla G. Field-sensitive value analysis by field-insensitive analysis//Proceedings of the Formal Methods (FM 2009). Eindhoven, The Netherlands, 2009: 370-386

[17] Xie Y, Aiken A. Saturn: A scalable framework for error detection using boolean satisfiability. ACM Transactions on Programming Languages and Systems(TOPLAS), 2007, 29(3): 16-es

[18] Xie Y, Aiken A. Context- and path-sensitive memory leak detection//Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005). Lisbon, Portugal, 2005: 115-125

[19] Ball T, Rajamani S. Automatically validating temporal safety properties of interfaces//Proceedings of the 8th International SPIN Workshop on Model Checking Software. Toronto, Canada, 2001: 102-122

[20] Henzinger T, Jhala R, Majumdar R, McMillan K L. Abstractions from proofs//Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL 2004). Venice, Italy, 2004: 232-244

[21] Dillig I, Dillig T, Aiken A. Sound, complete and scalable path-sensitive analysis//Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008). Tucson, USA, 2008: 270-280

[22] Henzinger T, Jhala R, Majumdar R, Sutre G. Lazy abstraction//Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL 2002). Portland, USA, 2002: 58-70

[23] Aiken A, Bugrara S, Dillig I, Dillig T, Hackett B, Hawkins P. An overview of the Saturn project//Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007). San Diego, California, 2007: 43-48

[24] Bodik R, Gupta R, Soffa M. Refining data flow information using infeasible paths//Proceedings of the 6th European Software Engineering Conference Held Jointly with 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1997). Zurich, Switzerland, 1997: 361-377

[25] Fischer J, Jhala R, Majumdar R. Joining dataflow with predicates//Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005). Lisbon, Portugal, 2005: 227-236

[26] Ngo M, Tan H. Detecting large number of infeasible paths through recognizing their patterns//Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007). Dubrovnik, Croatia, 2007: 215-224

[27] Ammons G, Larus J. Improving data-flow analysis with path profiles//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1998). Montreal, Canada, 1998: 72-84

ZHAO Yun-Shan, born in 1984, Ph. D. candidate. His research interests include static analysis and automating test.

GONG Yun-Zhan, born in 1962, Ph. D., professor, Ph. D. supervisor. His research interests include software test and software engineering.

LIU Li, born in 1988, M. S.. Her research interests focus on static analysis.

XIAO Qing, born in 1979, Ph. D. candidate. His research interests include software test and program analysis.

YANG Zhao-Hong, born in 1976, associate professor. His research interests include software test and software engineering.

Background

In recent years, program analysis techniques have been used to build tools for partial verification. The programmer provides a description of a temporal safety property written as a finite state machine. An analysis tool then tracks the state of the property FSM through a program. If the error state is never reached, the program obeys the safety property, or there reports an error.

Previous work on partial verification was focused on path-sensitive analysis methods. These methods are accurate enough for verification because they are able to reason about branch correlations, which is usually necessary to control the number of false error reports generated during verification. However, the cost of path-sensitivity has limited the applicability of these methods to large programs.

The authors present a new program slicing algorithm for path-sensitive defect detection. The slice criteria consists of defect feature and path condition, and slice the source program by the inclusion relation between the CFG dataflow information and slice criteria. The sliced program not only slices the defect irrespective codes, but also is totally equivalent to the original program, which improves the efficiency. In order to further reduce the false positives of path-sensitive analysis, the authors present a refined state-merging strategy to diminish the accuracy loss, which selectively merges the defect states by adding path condition into state attribute. The authors have implemented the technique in DTSGCC (Defect Testing System for GCC), a software defect detection tool for GCC projects in Linux. The authors apply DTSGCC to validate plenty of GCC open source projects. The experience suggests that applying the technique to path-sensitive defect detecting analysis improves the efficiency, and at the same time reduce potential false positives.

This project is sponsored by the National High Technology Research and Development Program (863 Program) of China under Grant No. 2009AA012404 and the National Natural Science Foundation of China (A Defect Detection Research, Implementation and Application on Aerospace Software, No. 91018002, 2010).