精确的程序静态分析

张 健

(中国科学院软件研究所计算机科学国家重点实验室 北京 100190)

摘 要 程序的静态分析是程序语言和编译领域的一个重要研究方向,已经被研究了很多年.近年来,它也引起形式方法和软件工程领域的重视,被用于程序测试和正确性验证.文中从程序的语法特征、所关心的数据类型和程序性质等方面比较了一些静态分析技术.着重描述基于路径的分析方法,特别是符号执行技术,讨论了程序路径可行性分析问题及其分类、复杂度.针对程序分析精度的一种量化指标,说明了其计算方法.

关键词 静态分析;程序路径;符号执行;数据覆盖

中图法分类号 TP311

Sharp Static Analysis of Programs

ZHANG Jian

(*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190)

Abstract Static program analysis has been studied for many years. It is an important topic in the research of programming languages and compilers. Recently, it has also attracted researchers from other areas such as formal methods and software engineering, and some ideas have been used for program verification and testing. This paper compares various static analysis techniques, according to the syntactic features, data types and correctness properties targeted by the techniques. The paper focuses on a class of analysis techniques that are based on program paths, especially symbolic execution; discusses the path feasibility problem, its difficulty and its subproblems. A method for computing a data coverage measure of paths is also described.

Keywords static analysis; program paths; symbolic execution; data coverage

1 引 言

正确性是程序最重要的属性之一.长期以来,如何保证程序的正确性、尽可能发现程序中各种潜在的错误,一直是计算机科学界关注的一个重要问题[1].这一问题也受到国际大型软件公司的高度重视.

虽然很多学者做了很大的努力,但对于我们常见的绝大多数程序而言,实现完全的正确性验证目前还是很困难的.一种经典的做法是,程序员先写出适当的断言(包括循环不变式),再利用自动化或半自动化的推理工具(定理证明器或者证明检查器)进行验证.另一类形式验证方法是模型检测,其自动化程度很高.但是它只能验证可归结为有限状态的程序.这里所说的"状态"是指程序变量的一种取值.

在工业界,人们通常是利用各种测试手段来发现软件中的错误,提高软件质量.通过运行软件,观察或比较其执行结果,判断软件中是否有错.虽然目前有很多工具能帮助用户统计测试过程中的各种信息(如语句覆盖率),但是测试用例的设计(特别是测试数据的自动生成)还是一个难题.

收稿日期:2008-07-16.**张 健**,男,1969 年生,研究员,博士生导师,研究领域为自动推理、约束求解、程序分析与软件测试. E-mail: zj@ios.ac.cn.

与动态的测试方法不同,我们也可以不运行软件,而直接分析程序代码,发现其中一些错误.这类方法称为静态分析.它是程序语言和编译、软件工程领域的一个重要研究方向.一个比较有名的早期开发的静态分析工具就是 lint,它主要检查 C 语言程序中的各种错误(如没有初始化的变量).近几年,静态分析技术也逐渐受到操作系统、信息安全等领域学者的高度重视.

本文第 2 节简要地介绍几种静态分析及验证方法,指出应以数据类型的处理作为比较不同方法的重要基础;第 3 节介绍路径可行性判定问题及解决此问题的两类方法,特别是符号执行的方法;第 4 节接着论述这种高精度的分析方法在程序分析、验证和测试中所起的作用;在第 5 节,我们描述程序分析精度的一种度量,并通过例子说明其计算方法;第 6 节是结束语.

本文采用C语言的文法来描述算法和程序例子.

2 静态分析与抽象

虽然程序分析技术已经被研究了很长时间,但因为要处理大程序,采用了很多近似的粗略的分析方法.比如,一类被称为流不敏感的(flow-insensitive)方法是,不考虑语句执行的先后次序.这类方法可得到一些保守的结果(比如变量 x 的值可能具有某性质).即使是如此粗糙的分析方法,对指针作别名分析也是 NP 难的[2].相对于流不敏感方法而言,也有流敏感的分析方法.

类似地,静态分析方法还可区分为路径敏感的(path-sensitive)和路径不敏感的.另外,有些方法只注重分析单个过程,被称为过程内分析方法;而针对多个过程的则是跨过程(inter-procedural)分析方法.

从某种意义上讲,上面的分类方法都是依据控制流的.我们认为,还可以从另外一个角度来进行划分,即以能精确处理什么样的数据类型来划分.

虽然很多方法和工具声称能分析比较大的 C 程序,但规模(程序行数)只是复杂性的一个方面.多数工具都不可避免地采用抽象的手段,忽略掉一些看起来无关的细节.

在 SLAM 项目[3]中,引入了"布尔程序"(Boolean program)的概念.这种程序的所有变量都是布尔类型的.SLAM 自动地将普通的 C 程序抽象为布尔程序.这种做法显然会丢掉很多重要信息,但是对设备驱动程序这类应用来说,取得了不错的效果.

数组是程序中常用到的一种语法结构.但它也给静态分析带来了一定的麻烦:如果下标表达式含有变量,很难在程序执行前知道它究竟是指哪个元素.对此,往往需将数组表达式简化.一种做法是,只看数组名,而不管下标.这样的分析方法把 $a[i+j]$ 和 $a[0]$ 当成一个对象;但是把 $a[0]$ 和 $b[1]$ 还是看成不同的对象.

Lev-Ami 等人[4]描述了如何用静态分析的方法来"验证"一些能处理链表结构的程序(比如用链表实现的插入排序算法).但实际上,他们的方法中只记录了程序中一部分信息(就是链表各元素之间的关系),而忽略了其它信息(如每个结构中的数值分量).其出发点应该是,只关心程序的部分性质(如元素之间的序关系).但如果程序不小心把数值变量的值改变了,他们的分析方法发现不了这种错误.

还有很多其它方法也只关心程序的一些特性,如"所有打开的文件要被关闭",等等.从数据类型的角度看,很多分析方法都是针对单变量、甚至是布尔变量.

3 路径可行性及判定方法

本节讨论如何分析具有丰富数据类型的程序,主要是面向路径的分析方法.

我们知道,从程序流图可以得到很多条程序路径.每条路径的起点是程序的入口.如果程序中有循环,可能会有无穷多条路径.

我们可以将路径表示为一个序列.为简便起见,假定序列中的每个元素是条件表达式(用@表示)或者赋值语句.输入语句(scanf)也可看成是一种赋值语句.当然,在序列之前,我们需要加上变量声明.

例 1. 下面是一条简单的程序路径 P_0:

```
int i, j;
@(i>5);
j=i+1;
@(j<4);
```

它表示,程序中有两个整型变量 i 和 j.在路径入口处,i 的值大于 5;后来执行了一条赋值语句;在出口处,又有一个条件表达式(j 的值小于 4).

3.1 路径可行性判定问题

给定一条路径,判断是否有变量的初始取值,使得程序沿该路径执行.这就是路径可行性判定问题.如果能找到合适的变量初始值,使路径被执行,则称该路径为可行的(feasible);否则就称它为不可行

(infeasible).

很显然，上面给出的路径 P_0 是不可行的. 如果 i 的初始值大于 5，那么执行了赋值语句之后，j 的值应该大于 6，而不可能小于 4. 如果 i 的初始值小于或等于 5，程序也不会沿着这条路径执行. 所以，该路径是不可行的.

在最一般情况下，路径可行性判定问题是不可解的. 如果不对程序变量和条件表达式加任何限制，路径可行性判定问题至少包含非线性整数方程求解作为特例.

如果对程序变量和路径中的表达式加以一定的限制，我们可以有自动化的算法判定路径可行性. 事实上，我们曾实现了一个针对 C 语言子集的路径可行性判定工具，称为 PAT[5]. 它能处理的数据类型不仅包括整型、布尔型，也包括数组. 但是，路径中不能有非线性的数值表达式. 给定上面的路径 P_0，PAT 会报告 infeasible. 但如果将最后一个条件中的 4 改为 9，那么 PAT 就会报告 feasible，并且给出一个合适的初始值(如 $i=6$).

后来我们对 PAT 加以扩展，使之能处理指针和结构类型[6].

3.2　精确的判定方法

PAT 所用的方法主要是基于符号执行和约束求解. 它是一种精确的判定方法. 也就是说，只要有足够的时间和空间资源，该方法总能判断出给定的路径是否可行，并且给出的答案肯定是正确的(而不是近似的).

所谓符号执行(symbolic execution)[7] 就是用符号(如 a)作为值赋给变量，对程序路径模拟执行. 相对而言，在普通的执行过程中，变量得到的值是具体的(比如 3). 如果变量 i 的初始值是符号 a，那么经过赋值语句 $j=i+1$ 之后，j 的值就变成 $a+1$.

通过符号执行可以得到一组关于变量初始值的约束——称为路径条件(path condition). 给定的程序路径是可行的，当且仅当路径条件可被满足. 例如，对前面给的路径 P_0，假定变量 i 的初始值是 a，那么通过符号执行，可以依次得到如下两个约束：

$$(a>5);\ (a+1<4).$$

它们构成了路径条件. 这显然是不可满足的. 如果将 4 改为 9，那么路径条件就可以被满足. 比如，a 取值为 6 就使得下面两个约束都成立：

$$(a>5);\ (a+1<9).$$

如何判断路径条件是否可满足？这就要用到约束求解算法(或者说“判定过程”). 详细情况可以见文献[5].

具体的判断算法及复杂度取决于路径中表达式的形式. (1) 如果分析的是布尔程序，路径中只有布尔变量和布尔逻辑运算符，那么得到的路径条件应该是布尔逻辑表达式. 判断它是否可满足的问题，是 NP-难的. (2) 如果我们分析的是数值计算程序，并且得到的路径条件是一组线性不等式，那么我们可以用线性规划的方法来判断路径条件是否可满足. (3) 自动工具 PAT 采用了布尔逻辑可满足性算法和线性规划相结合的方法，来判断路径条件的可满足性. 因为 PAT 可接受的路径中，既有布尔逻辑运算，又有线性的数值运算.

3.3　近似的判定方法

虽然符号执行的方法可以准确地判断路径的可行性，但目前其算法和实现技术还不太成熟，代价比较高. 所以，人们也提出了一些计算代价比较低的近似方法. 比如，软件工程领域早期出现过的一种判断方法[8] 是，如果一条路径中有很多不同的谓词(条件表达式)，那么这条路径不可行的概率就比较大. 从约束的观点看，这样的路径所对应的路径条件要满足很强的约束，很可能达不到要求.

最近，Ngo 和 Tan[9] 观察到常见程序中不可行路径的一些特征，提出了一些启发式判断方法. 在实际应用中，效果不错.

4　路径可行性判定与程序分析、验证、测试

路径可行性判断问题本来是软件测试中的重要问题. 在测试技术研究的早期，人们往往根据程序的控制流图来构造测试用例(从图中选取一定的路径)，但发现这样得到的很多路径是不可行的. 由于路径可行性判断问题本身固有的难度，在这一问题上一直没有大的进展. 如果能很好地解决该问题，对软件测试自动化(特别是测试数据自动生成)的重要性是不言而喻的.

除了测试以外，我们还可以在一定程度上“反证”程序的正确性. 方法是，将正确性性质取反，加在程序之后作为断言. 如果能找到输入数据，使得断言成立，就说明程序有错.

前面我们提到，很多静态分析方法的精度不高. 其好处是，效率很高，能处理大程序；但其问题是，分析结果的可信度差，可能会有很多的误报和漏报. 所谓“误报”，就是分析工具报告的错误实际上不存在；

所谓“漏报”,就是程序中某些错误没有被分析工具发现.利用高精度的路径分析工具,可在一定程度上避免上述问题.

在文献[10-13]中,给出了一些例子,说明符号执行等方法在程序测试、分析和验证中起的作用.

5 程序正确性的一种度量

我们知道,在程序测试技术中有一些测试覆盖标准(如语句覆盖率、分支覆盖率).

在文献[14]中,我们给出了程序路径的一种度量:对一条路径 P,定义 $\delta(P)$为其路径条件的解空间的大小(或者说其解空间的“体积”).如果所有变量都是整数类型,我们可将 $\delta(P)$定义为满足路径条件的整数点的个数(假定每个变量都在有限范围内取值).

通常我们只能检查有限条程序路径:$P_1, P_2, \cdots, P_m$. 即使对每条路径,我们可以自动地验证程序的正确性,一般也不能说明整个程序是完全正确的.但是我们可以用如下表达式的值,在一定程度上表示程序被验证的程度:$\delta(P_1)+\delta(P_2)+\cdots+\delta(P_m)$.其直观意义是,对于输入空间中的多少个点,我们已经验证了程序的正确性?

对于有界整型变量构成的输入空间,如果路径条件是线性不等式,我们可以利用 azove(Another Zero One Vertex Enumeration tool)[15]这样的工具对解的个数进行统计,从而计算出 δ 的值.如果路径条件中既有布尔变量,又有数值不等式,那就需要采用搜索过程.

例 2. 在文献[14]中,我们给了一个简单的程序作为例子,它计算两个正整数 m 和 n 的最大公约数.从中选出 3 条路径,分别对应如下 3 种情形:

(1) $m=n$;

(2) $m=2n$;

(3) $2m=n$.

如果限制 m 和 n 的取值范围都是[1..100],那我们可以算出上述 3 条路径的 δ 值分别是 100,50,50.对第 1 种情形,只需给出一些不等式:$1\leq m\leq 100$, $1\leq n\leq 100$, $m\leq n$, $n\leq m$.其它情形类似.

因为输入空间有 10000 个点,我们检查的这 3 条路径只对应了其中一小部分.但从语句覆盖率的测试标准来看,这几条路径达到了 100% 的覆盖率.再增加几条路径,语句覆盖率还是一样的.

由此可以看出,我们的度量 δ 从另外一个侧面反应了程序被检查、验证的程度.

6 结 语

目前常用的程序设计语言往往具有多种语言结构和特征,使得语言的表达能力很丰富.它带来的一个问题是,对程序进行深度的分析变得相当的困难.因此,在过去几十年,人们设计了各种近似的分析方法,对程序进行一定的抽象,只处理其中一部分信息.这些方法在实践中取得了一定的成功,即使有的分析方法相当粗糙.它们的好处是,分析算法的效率较高,因而能处理较大规模的程序.最近,国内学者也利用相关技术发现了公开源码程序中的一些错误[16,17].

随着人们对软件正确性、安全性要求的提高,随着计算机硬件性能的快速提升、相关算法(如 SAT 求解算法)及其实现技术的日益成熟,最近越来越多的人开始关注高精度的程序分析技术.它们能给出更加准确的分析结果,减少误报率.当然,在这一方向还有不少困难需要克服.

本文着重讨论了基于符号执行和约束求解的路径敏感的分析方法.我们认为,这种方法可以通过多种形式与其它技术相结合,以提高程序分析和测试的自动化程度及准确度.本文描述了一些具体的结合方式.

当然,对符号执行这类方法,要解决的问题也有很多,例如,如何处理程序中的各种数据类型和语法成分(数组、指针、过程调用等等),得到路径条件?如何简洁地表示路径条件?如何高效地求出它的解(或判断它无解)?如何计算路径条件的解空间的“体积”?如何从大量路径中选择合适的路径?如何采取一定的搜索策略尽快地找到错误?

我们相信,在不久的将来,高度精确的、高效率的程序分析技术会给软件工程师和算法设计人员带来更多的便利.

致 谢 感谢严俊、刘生提供的建议和信息!

参 考 文 献

[1] Hoare C A R. The verifying compiler: A grand challenge for computing research. Journal of the ACM, 2003, 50(1): 63-69

[2] Horwitz S. Precise flow-insensitive may-alias analysis is NP-hard. ACM Transactions on Programming Languages and Systems, 1997, 19(1): 1-6

[3] Ball T, Rajamani S K. The SLAM project: Debugging system software via static analysis//Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL 2002). Portland, OR, USA, 2002: 1-3

[4] Lev-Ami T et al. Putting static analysis to work for verification: A case study//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000). Portland, OR, USA, 2000: 26-38

[5] Zhang J, Wang X. A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge Engineering, 2001, 11(2): 139-156

[6] Zhang J. Symbolic execution of program paths involving pointer and structure variables//Proceedings of the QSIC. Braunschweig, Germany, 2004: 87-92

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

[8] Yates D F, Malevris N. Reducing the effects of infeasible paths in branch testing. ACM SIGSOFT Software Engineering Notes, 1989, 14(8): 48-54

[9] Ngo M N, Tan H B K. Heuristics-based infeasible path detection for dynamic test data generation. Information & Software Technology, 2008, 50(7-8): 641-655

[10] Zhang J et al. Path-oriented test data generation using symbolic execution and constraint solving techniques//Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004). Beijing, China, 2004: 242-250

[11] Zhang J. Constraint solving and symbolic execution//Proceedings of the VSTTE. Zurich, Switzerland, 2005: 539-544

[12] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C//Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE). Lisbon, Portugal, 2005: 263-272

[13] Xu Z, Zhang J. A test data generation tool for unit testing of C programs//Proceedings of the QSIC. Beijing, China, 2006: 107-116

[14] Zhang J. Quantitative analysis of symbolic execution (Extended Abstract)//Proceedings of the COMPSAC (Fast Abstracts). Hong Kong, China, 2004: 184-185

[15] Behle M, Eisenbrand F. 0/1 vertex and facet enumeration with BDDs//Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07). New Orleans, LA, USA, 2007: 158-165

[16] Wang Li, Yang Xue-Jun, Wang Ji, Luo Yu. Automatedly checking function execution context of kernel programs in operation systems. Journal of Software, 2007, 18(4): 1056-1067(in Chinese)
(汪黎,杨学军,王戟,罗宇.操作系统内核程序函数执行上下文的自动检验.软件学报, 2007, 18(4): 1056-1067)

[17] Xu Z, Zhang Jian. Path and context sensitive inter-procedural memory leak detection//Proceedings of the QSIC2008. Oxford, UK, 2008: 412-420.

ZHANG Jian, born in 1969, professor, Ph. D. supervisor. His research interests include automated reasoning, constraint solving, program analysis and software testing.

Background

Program correctness has been a central issue in computer science for many decades, and it has also become a serious concern for software engineers. To ensure the correctness of programs, many formal verification methods have been proposed, but they are often difficult to use in practice. In industry, engineers typically rely on testing to find bugs. In contrast to the above two methods, static analysis techniques try to find specific types of bugs in a program, automatically and efficiently, without running the program. Tools based on such techniques have been quite helpful to programmers. However, to deal with big programs, most of the techniques try to abstract away some aspects of the program which are not so relevant. This may bring various kinds of false warnings. This paper compares some of these techniques, and argues that we should pay more attention to more advanced, more accurate analysis methods. As an example of such methods, symbolic execution is described in detail. We believe that it is a promising way of analyzing programs accurately. Some research issues are also discussed.

This work is partially supported by the National Natural Science Foundation of China (NSFC) under grant No. 60673044 and No. 60633010, and by the National High-Tech Program (863) under grant No. 2006AA01Z402.