第38卷 第18期
Vol.38 No.18

计 算 机 工 程
Computer Engineering

2012年9月
September 2012

·软件技术与数据库·

文章编号：1000—3428(2012)18—0030—04 文献标识码：A 中图分类号：TP12

并发程序原子图挖掘技术

朱一清

(上海交通大学软件学院，上海 200240)

摘 要：针对当前并发程序的不确定性和复杂性，以及程序原子性质获取困难的问题，提出一种并发程序原子性质提取方法。将并发程序中的同步区域转化为与并发操作相关的并发操作图后，采用频繁子图挖掘算法自动提取程序中的原子图，使其能刻画并发程序的原子性质，包括并发操作以及操作之间的控制依赖关系。实验结果证明，该方法能以较低的误测率有效提取并发程序的原子性质。

关键词：并发程序；原子图；原子性质；多线程；频繁子图挖掘

Atom Graph Mining Technology in Concurrent Program

ZHU Yi-qing

(School of Software, Shanghai Jiaotong University, Shanghai 200240, China)

【Abstract】Because of the complexity and uncertainty of parallel behaviors, currently it is difficult to get atom rules from concurrent programs. Aiming at the problem, this paper presents a new atom property extraction method in concurrent program. A technique to automatically extract atom property from programs forms graph structure by using a frequent subgraph mining algorithm. It targets at inferring atomicity of both low level read-write operations as well as high level functional operations on multiple variables with rich structure information. Experimental result shows that this method manages to find atom property with a relatively high accuracy.

【Key words】concurrent program; atom graph; atom property; multi-threading; frequency subgraph mining

DOI: 10.3969/j.issn.1000-3428.2012.18.008

1 概述

随着多核时代的到来，越来越多的软件引入并发机制，提高程序的效率。一项针对 Dacapo 软件库的调查显示，约 2/3 的软件中存在并发行为。然而，由于并发程序线程切换的不确定性，因此如何理解并且维护并发程序长期以来一直困扰着程序员。其中一个显著问题是提取程序的原子性质。程序的一簇操作具有原子性质是指这些操作在执行时不应该因线程切换而被打断或插入其他操作。许多并发错误都是由于程序原子性质的违背所造成的，因此，获取程序的原子性质对于并发程序来说非常重要。然而，大多数原子性质与程序的语义有关，它们不能仅通过传统的程序分析方法如控制流分析、数据流分析等，从程序中直接获得。

为了解决这个问题，可以通过构建类型系统，即使用特定的语言，让程序员在书写程序时为原子性质进行一定的注解，随后基于 Lipton 的降级理论[1]化简及整理这些注解，从而得到程序的原子性质。然而，这些注解工作对于程序员来说比较困难且非常耗时，因此并不实用。

本文提出一种新的方法来自动提取并发程序的原子性质。使用原子图来表示该原子性质。原子图以图的结构来代表程序中具有原子性质的操作集合。原子图中的节点代表针对共享变量的并发操作，边代表并发操作之间的控制依赖关系。此外，每一个原子图还包含一个代表程序中锁使用情况的节点。使用频繁子图挖掘算法，从并发程序中自动挖掘出其所包含的原子图。本文工作基于程序员在大多数情况下都是正确的这个假设，因此，如果某些并发操作经常出现在一起，则认为这些并发操作具有原子性质，必须被一起执行。在实验阶段，将其应用于 10 个大型的并发程序，包括 tomcat 以及 h2database。

2 相关研究

近年来，针对并发程序的原子性质研究是一个热点课题。许多工作都使用统计学习的方法来探究该性质。

文献[2]从程序中自动采集程序员对程序的 MUST/MAY 观点，使用统计方法来提取程序规则。文献[3]将一种频繁封闭偏序挖掘算法应用于程序路径中，以此推测程序的 API 使用模式。文献[4]使用频繁子图挖掘模式，针对面向对象语言挖掘出对象使用模式。然而，这些工作都非针对并发程序。而且由于并发程序的复杂性，这些工作中使用的统计方法也不能直接应用于本文中。

AVIO[5]通过动态地执行程序，并不断地更新变量访问

基金项目：国家自然科学基金资助面上项目(60673120)

作者简介：朱一清(1987－)，男，硕士研究生，主研方向：并发程序分析

收稿日期：2011-12-12 **修回日期：**2012-02-07 **E-mail：**alex.yqzhu@gmail.com

集(access-interleaving set)来提取并发程序中的访问不变量(access-interleaving invariant)，以此作为具有原子性的代码集。MUVI[6]使用频繁项集挖掘算法 FPclose，从并发程序中自动挖掘出经常一起被访问的变量，并以此作为原子集。

PRETEXT[7]使用机器学习算法 sk-strings 从程序的动态执行轨迹中寻找变量状态的迁移不变量(typestate property)。由于该迁移不变量是由代码块的原子性所保证的，因此识别出的变量状态的迁移不变量可以作为程序的原子性。然而，这些工作都存在一定缺陷，它们或是只针对一部分操作，如直接读写操作，而略去了函数操作，或是只针对单个共享变量，或是只针对多个共享变量的无序操作，而略去了控制依赖信息。

相比以往工作，本文在提取并发程序原子性质方面做出以下贡献：

(1)提出并发操作图，能够全面地刻画并发程序同步区域中的并发操作行为。

(2)通过频繁子图挖掘算法从并发操作图中挖掘出原子图，结构化地表示并发程序的原子性质。

(3)基于以上思想，依赖 Soot 框架实现 EAGraph 工具，并通过实验验证了本文工作的有效性。

3 并发程序原子图挖掘

原子性质挖掘器的工作(EAGraph)流程如图 1 所示，分为 2 个主要步骤，预处理以及原子图挖掘，其唯一的输入为程序本身，除此之外无需其他信息。

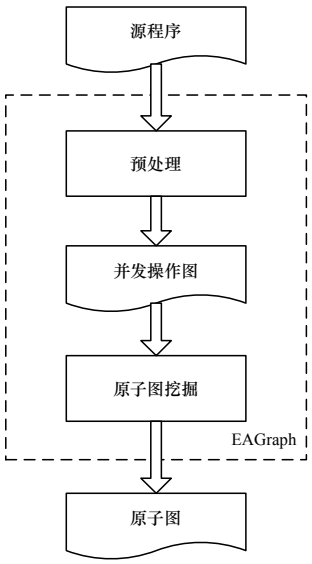

图 1 EAGraph 流程

3.1 预处理

本文希望能够自动地识别程序员在程序开发中所设计的原子性质，并将其以原子图的形式呈现。由于程序员保持程序原子性质的最佳方法是将它们放在程序的同步代码块中，因此 EAGraph 将从程序的同步区域中提取原子图(Java 语言中的同步区域包括同步块以及同步函数)。然而，同步区域中会有许多操作非并发操作，程序员将这些操作放置在同步区域中只是随意为之，并不具有目的性。而这些操作的存在会影响原子图挖掘的效率及准确性。此外，同步区域中的程序在字面上并不具有结构化的信息，给挖掘工作带来了一定的难处。因此，EAGraph 首先将对所有的同步区域进行预处理，将其转换为适合下一步骤的并发操作图。

定义 1(并发操作图 *SRG*) 对于任意一个同步区域，其相应的并发操作图 *SRG* 是一个有向标记图，满足以下性质：

(1)并发操作图中每一个节点代表该同步区域中针对共享变量的一个操作，每个节点的标记为一个二元组(操作名称，操作对象)，操作名称为操作共享变量的函数名称，操作对象为该操作所涉及的共享变量。

(2)并发操作图中的每一条边代表其所连接的 2 点所代表的操作之间具有控制依赖关系，边没有标记。

(3)每一个并发操作图具有一个代表该同步区域锁信息的节点，其标记为(“lock”，锁集)，锁集为此同步区域所持有的锁变量。该节点是一个孤立点。

定义 2(并发操作图集合 *SRGset*) *SRGset* 是程序中所有同步区域所对应的并发操作图的集合。

预处理步骤如下：

Step1 获得函数的控制流图。

Step2 截取同步区域的控制流图。

Step3 添加传递闭包。

Step4 剔除非并发操作。

Step5 转换为并发操作图。

用一个实例来说明预处理流程：

```
1. Public class Example {
2.       Integer globalInt;
3.       String globalStr;
4.       boolean local;
5.       public void test() {
6.             print(local);
7.             Integer temp;
8.             synchronized(globalInt) {
9.                   temp = globalInt;
10.                  if (local) {
11.                        globalStr = “a”;
12.                        record(temp);
13.                  }else{
14.                        globalStr = “b”;
15.                  }
16.            }
17.      }
18. }
```

在类 Example 中，globalInt 和 globalStr 为共享变量；local 为局部变量；test 函数中含有一个同步区域，其持有的锁变量为 globalInt，在该共享变量中，与共享变量相关的操作有 4 个，分别为第 10 行、第 12 行、第 13 行、第 15 行。

首先，EAGraph 获得 test 函数的控制流图，该控制流图由 Soot 的 UnitGraph 表示，UnitGraph 中的节点对应函数中的语句，边对应控制依赖关系。接下来，EAGraph 删除在同步区域以外的节点及边，得到 test 函数中同步区域所对应的控制流图。然后，EAGraph 将剔除该控制流图中与共享变量无关的节点，为了保证在删除节点后控制依赖结构不被破坏，EAGraph 先针对该控制流图添加一个传递闭包，即在任意 2 个有控制依赖关系的节点间均增加一条边。

然后，EAGraph 使用 Soot 中的静态分析方法 ThreadLocalObjects Analysis 和 SideEffectAnalysis 来分析出哪些节点对应的语句中涉及到共享变量，即识别出并发操作。在分析完毕后，EAGraph 将删除控制依赖图中的非并发操作节点，以及与这些节点相连的边。最后，EAGraph 将基于以上得到的针对同步区域的，仅包含并发操作的控制流图，构建出并发操作图。EAGraph 根据该图中每一个节点所代表的并发操作，为其赋予标记(操作名称，操作对象)。同时，EAGraph 将为该图增加一个节点，以代表锁的信息。

图 2 为最终所得的并发操作图。

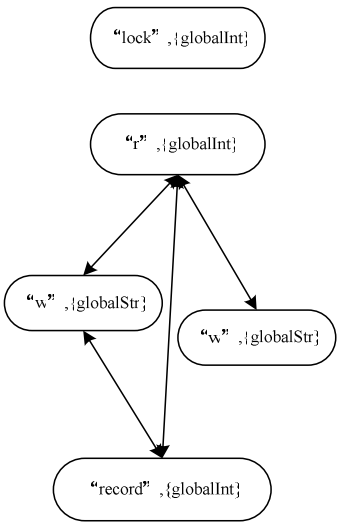

图 2 并发操作图

综上所述，预处理流程通过以上 5 个步骤，将每一个同步区域转化为一个并发操作图。这些并发操作图将作为下一步骤的输入，用以提取出并发程序中的原子图。

3.2 原子图挖掘

正确的规则通常会重复出现，在程序中也不例外。由于需要去推测出程序中与原子性相关的规律，因此从预处理中得到的并发操作图入手，在其上进行频繁子图的挖掘，以期找出频繁出现的子图模式，并以此作为程序原子性的一个表征。

该步骤的输入为程序中所有同步区域所对应的并发操作图，其输出为该程序中的原子图，每个原子图代表程序中具有原子性质的一簇操作。

定义 3(导出子图) 对于标记图 $G(V,E)$ 和 $G'(V',E')$，G 被称为 G' 的导出子图，如果 $V \subseteq V'$，则 $\forall e=(v_1,v_2)\in E'$，如果 $v_1 \in V$ 及 $v_2 \in V$，则 $e \in E$。

定义 4(子同构) 对于标记图 G 和 G'，G 对于 G' 子同构，如果 G' 中含有导出子图 G''，则使得 G 与 G'' 标记同构。

定义 5(频繁子图) 对于标记图 G 以及标记图集 $D=\{G_1,G_2,\cdots,G_n\}$，D 中对应于 G 的支持图集为 $D_G=\{G_i \mid G$对于G_i子同构，$G_i \in D\}$。G 在 D 中频率为 $freq(D_G)=|D_G|$。D 对 G 的支持度为 $sup(D_G)=freq(D_G)/|D|$。G 是 D 在最小支持度 min_sup 下的频繁子图，$sup(D_G)\geqslant min\_sup$。

定义 6(原子图 AG) 对应于最小支持度 min_sup 的原子图 AG 是一个标记图，满足：

(1)存在 $G \in SRGset$，使得 AG 是 G 的导出子图。

(2) AG 是 $SRGset$ 在最小支持度 min_sup 下的频繁子图。

为了能够根据给定的最小支持度 min_sup 从 $SRGset$ 找出所有 AG，EAGraph 使用频繁子图挖掘算法 Patt-Explorer[4]来从 $SRGset$ 中挖掘 AG，PattExplorer 是一个点增长算法，基于频繁子图的反单调性。

定义 7(频繁子图的反单调性) 频繁子图的反单调性是指图 G 的某一子图是频繁的，仅当该子图的所有子图是频繁的。

虽然近年来有许多频繁子图挖掘算法被提出[8]，但 PattExplorer 是最适合本文的算法。这是因为 EAGraph 对同步区域的控制依赖图添加了传递闭包，所以所得的并发操作图常常为稠密图，而 PattExplorer 算法针对稠密图极其有效。同时，由于并发操作图中有环的出现，对该算法进行了一定的改进，使其能够解决有向带环图的挖掘。由于篇幅的限制，本文不介绍该算法的细节。当挖掘结束后，EAGraph 将获得一簇频繁出现的并发操作图的子图，即该并发程序的原子图。最后，通过手动检测，从中得到正确的原子图，这些原子图能够代表程序员在程序开发中所设计的原子性质。

4 EAGraph 应用与评估

基于 Soot 框架实现了 EAGraph 工具，并将其应用于 10 个大型的并发软件系统以评估其有效性。实验在一台 16 核、2.27 GHz 主频、33 GB 内存的服务器上进行，使用的操作系统为 Linux2.6.32。

4.1 实验程序介绍

如表 1 所示，使用 10 个大型的实验程序，其中 6 个来自于 dacapo 代码库。#SR 为每个实验程序中同步区域的数量，#Aver-RW 和#Aver-FC 分别为每个同步区域中针对共享变量的平均直接读写次数以及函数操作次数。RW:FC 为两者数量之比，从中可以发现，对于大多数并发程序，两者均在程序的并发操作中占有一定比例。因此，EAGraph 对这 2 种操作行为均进行分析是非常合理的，略去其中任何一种都会影响准确性。

表 1　实验程序信息

实验程序	#SR	#ARW	#AFC	RW:FC
batik	58	2.8	2.3	1:0.8
corba	353	3.7	2.6	1:0.7
derby	124	0.5	3.1	1:5.8
h2database	145	5.0	3.2	1:0.6
jigsaw	213	2.2	1.9	1:0.9
luindex	165	0.7	2.7	1:4.0
lusearch	50	1.9	1.3	1:0.7
openjms	269	1.8	1.2	1:0.6
tomcat	401	4.2	2.6	1:0.6
xalan	43	2.2	3.6	1:1.6

4.2　预处理评估

预处理步骤的目的是为了剔除每一个同步区域中与共享变量无关的操作。预处理可以增加工作的精度，同时提高后续原子图挖掘的效率。图 3 显示了预处理前后同步区域中操作数量的变化。CFGsize 为预处理前同步区域的控制流图中节点的平均个数，SRGsize 为预处理后并发操作图中节点的平均个数。可以发现，预处理将同步区域的规模平均缩减至原来的 1/3。

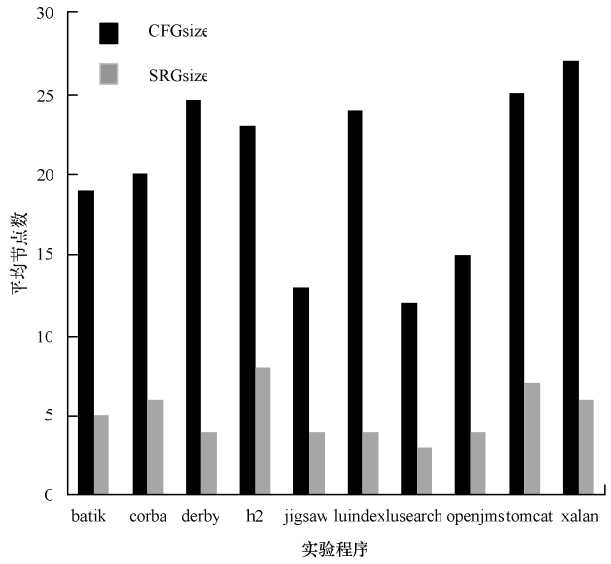

图 3　预处理实验结果

4.3　原子图挖掘评估

原子图挖掘步骤通过频繁子图挖掘算法 PattExplorer，从并发操作图中提取原子图。表 2 为该步骤的实验评估结果。对于同步区域个数小于 100 的实验程序，将定义 5 中的频率设置为 3，对于其余程序，频率设置为 4。在该表中，#AG 是 EAGraph 从每一个实验程序中挖掘出的原子图数量，#AGsize 为原子图的平均节点个数。为了能够验证该步骤的准确性，从挖掘出的原子图中随机挑选一部分，通过查看原程序来手动检测它们是否正确。#realAG 显示了正确的原子图在所检测原子图中所占的数量。通过实验发现，EAGraph 能够较准确地自动提取程序原子图，其平均的误测率仅为 28.5%。

表 2　原子图挖掘的实验结果

实验程序	#AG	#AGsize	#realAG	误测率/(%)
batik	9	3	7/9	22
corba	102	3	7/10	30
derby	43	3	6/10	40
h2database	77	4	6/10	40
jigsaw	22	2	8/10	20
luindex	32	2	7/10	30
lusearch	4	2	4/4	0
openjms	95	3	7/10	30
tomcat	88	3	6/10	40
xalan	15	3	10/15	33

5　结束语

本文详细介绍了如何构建同步区域的并发操作图，以及如何采用图挖掘算法来提取原子图。在实验阶段，基于本文思想实现了 EAGraph 工具，并使用该工具对 10 个大型的并发程序进行了测试。实验结果表明，EAGraph 能够以较低的误测率有效地从并发程序中提取原子性质。

参考文献

[1] Lipton R J. Reduction: A Method of Proving Properties of Parallel Programs[J]. Communications of the ACM, 1975, 18(12): 717-721.

[2] Engler D, Chen Yu, Hallem S, et al. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code[C]// Proceedings of the 18th ACM Symposium on Operating Systems Principles. New York, USA: ACM Press, 2001: 57-72.

[3] Acharya M, Xie Tao, Pei Jian, et al. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications[C]//Proceedings of ESEC-FSE'07. New York, USA: ACM Press, 2007: 25-34.

[4] Nguyen T T, Nguyen H A, Pham N H, et al. Graph-based Mining of Multiple Object Usage Patterns[C]//Proceedings of ESEC-FSE'09. New York, USA: ACM Press, 2009: 383-392.

[5] Lu Shan, Tucek J, Qin Feng, et al. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants[C]//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA: ACM Press, 2006: 37-48.

[6] Lu Shan, Soyeon P, Hu Chongfeng, et al. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs[C]//Proceedings of the 21st ACM Symposium on Operating Systems Principles. [S. l.]: ACM Press, 2007: 103-116.

[7] Joshi P, Sen K. Predictive Typestate Checking of Multithreaded Java Programs[C]//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. Washington D. C., USA: IEEE Computer Society, 2008: 288-296.

[8] 陈振强，徐宝文. 一种并发程序依赖性分析方法[J]. 计算机研究与发展, 2002, 39(2): 27-29.

编辑　陆燕菲