软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(2):455−472 [doi: 10.13328/j.cnki.jos.006160]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

SOTIMiner: 一种基于集合运算的时序不变式挖掘方法*

孙德权[1,2], 周竞文[1,2], 周海芳[1]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](复杂系统软件工程湖南省重点实验室(国防科技大学), 湖南 长沙 410073)

通信作者: 周竞文, E-mail: jwzhou@nudt.edu.cn

摘 要: 时序不变式反映了事件间的时序逻辑关系, 被广泛应用于异常检测、系统行为理解、模型推理等技术. 在实际使用中, 一般通过分析软件系统的日志数据挖掘时序不变式. 相比全序日志, 偏序日志可为挖掘算法提供更为准确的数据来源. 但是, 现有的基于偏序日志的时序不变式挖掘方法存在效率较低等问题. 为此, 以系统执行路径为数据来源, 提出了一种基于集合运算的时序不变式挖掘方法 SOTIMiner, 并研究了改进方案. 相比现有方法, 该方法不需要反向遍历日志数据, 从而具有较高效率. 实验显示. 该方法在保证挖掘相同结果的基础上, 效率平均是 Synoptic 挖掘工具的 3.23 倍.

关键词: 系统执行路径; 时序不变式; 集合运算; SOTIMiner

中图法分类号: TP311

中文引用格式: 孙德权, 周竞文, 周海芳. SOTIMiner: 一种基于集合运算的时序不变式挖掘方法. 软件学报, 2022, 33(2): 455–472. http://www.jos.org.cn/1000-9825/6160.htm

英文引用格式: Sun DQ, Zhou JW, Zhou HF. SOTIMiner: Mining Method of Temporal Invariants Based on Set Operations. Ruan Jian Xue Bao/Journal of Software, 2022, 33(2): 455–472 (in Chinese). http://www.jos.org.cn/1000-9825/6160.htm

SOTIMiner: Mining Method of Temporal Invariants Based on Set Operations

SUN De-Quan[1,2], ZHOU Jing-Wen[1,2], ZHOU Hai-Fang[1]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](Laboratory of Software Engineering for Complex Systems (National University of Defense Technology), Changsha 410073, China)

Abstract: The temporal invariants reflect the temporal logic relationship between events and have been widely used in anomaly detection, system behavior understanding, model reasoning, and other techniques. Generally, mining temporal invariants through analyzing the log data of software system is in actual use. Compared with totally ordered log, partially ordered log can provide a more accurate data source for mining algorithm. However, the existing temporal invariants mining methods based on partially ordered log have some problems such as its low efficiency. For this reason, this study uses the system execution path as the data source and proposes a temporal invariants mining method SOTIMiner based on set operations and studies an improved scheme. Compared with existing methods, it does not need to traverse the log data in reverse and for the reason that it has a higher efficiency. Experiments show that the method's average efficiency is 3.23 times of the Synoptic mining tool on the basis of guaranteeing the same result.

Key words: system execution trace; temporal invariants; set operations; SOTIMiner

时序不变式(temporal invariant)[1]反映了软件系统中事件间的时序逻辑关系, 是一种比较常用的反映系统行为的不变式, 可以从多种来源获得, 如通过分析软件开发文档、源代码或系统日志等. 在实际应用中, 分析源代码时通常执行效率较低、难度较大且只能反映程序本身的问题, 不能反映与外部环境之间的关系. 而在对软件系统产生的日志数据进行分析时, 则可以通过事件之间的时序关系推理出系统应该满足的不变式. 研

* 基金项目: 国家自然科学基金(61702530, 61690203); 国家重点研发计划(2018YFB0204301, 2017YFB1001802)
收稿时间: 2020-01-21; 修改时间: 2020-06-19, 2020-07-31; 采用时间: 2020-09-15

究表明: 日志信息对系统调试有很大的帮助, 对于缩短故障调试时间的加速比为 2.2[2].

根据日志类型, 时序不变式的挖掘方法可以分为基于全序日志和基于偏序日志的方法.

- 在全序日志中, 任意两个事件都有确定的时序关系, 例如传统日志属于全序日志, 它通过时间戳来判断事件间的时序关系, 即: 如果事件 A 的时间戳小于事件 B 的时间戳, 则认为事件 A 发生在事件 B 之前, 基于事件间的这种时序关系可以挖掘时序不变式. 但是, 由于软件行为的复杂性, 事件之间并不总是存在时序关系. 例如: 对于并发程序中的两个并发事件 A 和 B, 即使 A 发生在 B 之前, 也不应该认为它们之间存在时序关系. 因此, 全序日志会为本不存在时序关系的事件强加上时序关系, 基于这种数据挖掘出的时序不变式往往也是不准确的;
- 而偏序日志能够有效地解决全序日志存在的问题. 偏序日志不是利用时间戳判断事件间的时序关系, 而是显式地记录事件间的时序关系. 例如: 系统执行路径(system execution trace)[3]可以看作是一种偏序日志, 它利用事件间的相关性反映事件间的时序关系, 如果事件 A 和事件 B 相关, 则认为 A 和 B 存在时序关系, 否则不存在. 因此, 偏序日志可为时序不变式挖掘方法提供更为准确的数据来源, 得到更为准确的时序不变式. 但是, 现有的基于偏序日志的挖掘方法存在挖掘效率偏低的问题, 不能高效地从大量日志中挖掘出时序不变式. 针对该问题, 本文提出了一种基于集合运算的挖掘方法 SOTIMiner, 相比传统方法, 该方法不需要反向遍历日志数据, 从而具有较高的效率.

本文第 1 节介绍系统执行路径以及向量时钟的表示方法, 对涉及到的常用时序不变式和挖掘时序不变式的基本过程给出形式化定义, 并介绍先前工作的主要思路. 第 2 节给出一种基于集合运算的挖掘方法 SOTIMiner, 并在此基础上提出了改进方案. 第 3 节将 SOTIMiner 与先前的工具 Synoptic[4]进行对比实验, 对方法的功能、效率等进行验证. 第 4 节介绍相关工作并与本文方法进行比较. 第 5 节总结全文并对未来工作进行展望.

1 相关概念

本节结合一个生活中的例子, 介绍方法涉及到的相关概念, 包括系统执行路径、向量时钟、时序不变式、挖掘时序不变式的方法以及先前工作的主要思路.

1.1 示 例

在某就诊预约系统中, 某医生每天的预约名额是固定的, 假设现在还剩余 1 个预约名额, 此时有病人 1 和病人 2 两人先后在网上查询该医生是否可预约, 发现状态为可预约, 然后分别进行预约, 最终病人 1 预约成功而病人 2 预约失败, 病人 2 再查询时发现该医生的状态为不可预约.

- 图 1(a)展示了整个过程, 共包含 10 个事件, 编号为①–⑩;
- 图 1(b)将图 1(a)中显示的事件过程恢复为系统执行路径的请求树形式, 是本文方法的分析对象;
- 图 1(c)给出了各事件对应的向量时钟, 通过比较向量时钟, 可以判断两个事件间的时序关系, 本文认为时序关系是通过向量时钟记录的;
- 图 1(d)给出了从该过程中分析出的若干时序不变式. 例如: 在分析得到的一些结果中, 医生没有预约名额后(不考虑病人取消预约等情况), 其他病人则不能够成功预约该医生. 即预约失败后从不发生预约医生的事件(预约失败↛病人 1 预约、预约失败↛病人 2 预约). 而另一些结果如两名病人的查询和预约在图中是并行发生的事件, 即病人的查询和预约不存在确切的先后顺序, 每个病人都可以选择先查询或预约.

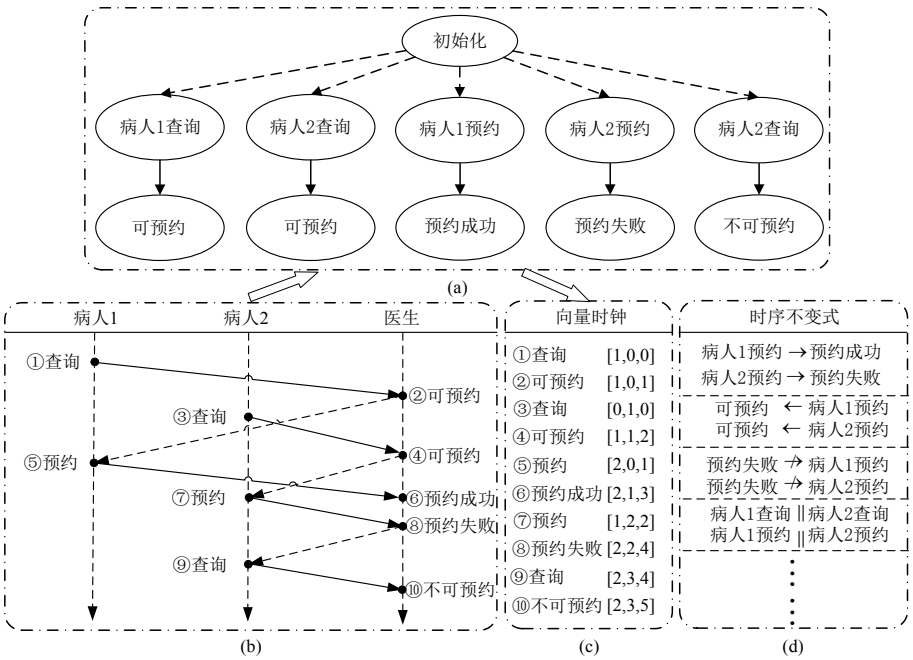


图 1　时序不变式应用示例

1.2 系统执行路径

系统执行路径[3]记录了用户请求在系统中的执行过程, 如在图 1(b)中, 路径 1 记录了病人 1 的预约过程, 先在客户端查询医生状态, 服务端查询后显示可预约, 客户端再进行预约, 服务端处理预约信息并返回预约成功.

系统执行路径由事件和事件之间的关联两类信息组成. 事件对应系统在执行用户请求过程中进行的操作(如函数调用等), 如图 1(b)中的查询、预约都是事件. 除记录操作名字之外, 事件中还记录了这些操作的详细信息, 如执行时间、所在节点地址等. 而关联信息反映了事件与事件之间的依赖、时序等关系(如函数间的调用、节点间的通信等), 如在图 1(b)中, 事件①和事件②之间的边表示这两个事件是直接相关的.

在分布式系统中, 系统发生的事件主要包括节点内部发生的事件和节点间的通信事件, 节点内部发生的事件可用线性关系表示, 而通信事件的加入将可能导致分支的产生. 利用事件之间已知的相关信息, 可将路径中的事件组织成线性、树状或图状, 从而直观地反映请求执行过程. 例如, 图 1(b)采用树状结构(请求树)表示路径, 一条路径对应一棵树, 树中节点对应事件, 边表示相关信息. 新分支的产生可以根据两事件具有的相关信息决定. 如果有关联的两事件不是日志记录的顺序事件, 则在系统执行路径中体现为事件后的新分支. 此外, 线状图主要用于全序日志的表达, 而树状作为图状的特例, 当仅求解两事件间的不变式时所表达的内容没有差别. 这种表达方式可用于描述并行行为.

捕获系统执行路径的技术称为系统行为跟踪. 在系统行为跟踪过程中, 事件的记录方法和内容与传统日志类似, 其关键问题是如何获取事件间的相关信息. 根据相关信息获取方法的不同, 系统行为跟踪技术可以分为基于白盒插桩、基于黑盒插桩和基于黑盒推理的跟踪[1]: 基于白盒插桩的跟踪技术通过对目标系统的源代码进行插装捕获精确的路径; 基于黑盒插桩的跟踪技术不修改目标系统的源代码, 而是对中间件、虚拟机等运行环境进行插装获取路径; 基于黑盒推理的跟踪技术是从传统日志等数据中推理出近似的路径. 这 3 类方法都能获取系统执行路径, 在实际中的应用都较为广泛.

与传统日志数据相比, 执行路径显式地记录了事件间的相关信息, 可为分析和理解系统行为、检测和诊断系统异常等提供更为丰富的信息. 因此, 系统执行路径已成为当前的研究热点, 越来越多的公司也开始在

系统中增加跟踪机制, 以记录系统执行路径, 从而帮助更好地维护系统, 如 Google 的 Dapper[5]、Twitter 的 Zipkin[6]等. 在这一背景下, 本文选用系统执行路径作为数据来源, 以挖掘系统中的时序不变式.

1.3 向量时钟

在系统执行路径中, 相关信息可采用不同的方式加以表示, 其中一种较为典型的方法是向量时钟[7]. 与时间戳相比, 向量时钟主要有两点不同.

1) 并不是任意两个向量时钟都可以比较大小: 若能比较大小, 则较小者发生在前; 若不能比较大小, 则不存在先后关系. 该特点使向量时钟适合于记录偏序信息;
2) 通过事件触发向量时钟的更新, 可以解决不同机器之间存在时钟偏移的问题, 因此适合于记录多节点系统(如分布式系统等)中的信息.

下面以分布式系统为例, 对向量时钟机制进行介绍.

在一个具有 n 个节点(节点编号为 0~n−1)的分布式系统中, 每个节点都维护一个向量时钟, 每个节点上的向量时钟都是一个 n 维的数组 $T=[t_0,t_1,\ldots,t_{n-1}]$, 各节点上向量时钟的初始值均为[0,0,…,0]. 当某节点发生一个事件后, 系统会更新本节点上的向量时钟, 更新后的向量时钟会被记录到事件信息中, 因此, 每个事件都对应一个向量时钟, 该向量时钟就是对应事件发生的逻辑时间.

在对节点的向量时钟进行更新时, 相应的方法如下.

(1) 当节点 n_i 上发生一个事件时, 对 n_i 上的向量时钟进行更新, 方法是将向量时钟的第 i 个分量加 1, 即将$[t_0,t_1,\ldots,t_i,\ldots,t_{n-1}]$更新为$[t_0,t_1,\ldots,t_i+1,\ldots,t_{n-1}]$;
(2) 当节点 n_i 的事件触发节点 n_j 的事件(如通信或远程过程调用等)时, n_i 将本节点的向量时钟发送至 n_j, n_j 首先按照情况 1 更新本地时钟, 再保留两个向量时钟对应维度的较大分量, 即:

$$\forall k, T_j[k]=\max(T_i[k],T_j[k]).$$

根据上述过程, 可将每个事件与唯一的向量时钟值进行关联, 进而利用向量时钟可对事件的时序关系加以判断: 对于两个事件 e_i 和 e_j, 其向量时钟分别为 T_i 和 T_j: 如果$\forall k$, $T_i[k]\leqslant T_j[k]$, 则这两个事件存在偏序关系 $e_i\prec e_j$; 否则, 不认为存在这样的偏序关系. 例如, 两个向量时钟值[1,2]和[2,1]就不能被这样顺序排序, 具体实例将在第 2 节给出. 需要说明的是: 在利用向量时钟挖掘时序不变式时, 为了确保不同节点的两个事件的偏序关系是可排序的, 对于最新发生事件的向量时钟, 其分量不应小于先前发生事件的向量时钟的分量. 另外, 并不是所有跟踪工具都采用向量时钟的方式记录事件间的时序关系, 但其他形式的时序关系可以转换成向量时钟的形式. 因此, 本文方法假设路径采用向量时钟表示相关性, 并不影响方法应用于其他形式的路径数据.

1.4 时序不变式

先前的工作表明, 从路径日志中挖掘时序不变式对发现系统中存在的问题有不错的效果. 多数挖掘时序不变式的工作都建立在传统日志的基础上, 从路径日志中挖掘的工作较少. 从路径日志中挖掘得到的时序不变式种类可以归结为 5 种. 这些不变式均反映了两个事件之间的关系, 而非描述多个事件之间的复杂关系, 这能够较好地记录系统所应该满足的行为特征. 在时序不变式中, 反映两个事件之间的关系实际上是反映两个事件类型之间的关系. 路径日志中两个事件类型不同被定义为除与主机上进行相同操作的两个事件外, 其余均认为是不同事件类型. 例如: 主机 1 和主机 2 均发生了获取文件信息的事件, 即调用相同函数操作, 但由于发生在不同的主机上, 这两个事件将被归为不同种类. 为方便简洁, 本文后续对事件间关系判断的说明, 均表示对事件类型间关系的判断. 下面给出 Synoptic 中各个时序不变式的获得过程.

在一阶逻辑中, 量词包含全称量词$\forall$和存在量词$\exists$, 基本的逻辑连接词包含且$\wedge$或$\vee$非$\neg$. 对于日志记录的事件集合中的任意两个事件实例 e_i 和 e_j, 可能出现的量词前缀和逻辑表达式分布如下.

选取量词前缀时, 考虑到路径中每两个事件类型之间的不变关系, 而$\forall e_i\forall e_j$与$\forall e_j\forall e_i$表达的含义相同, 仅保留$\forall e_i\forall e_j$. 当存在量词$\exists$作为先决条件时, 所产生的逻辑式子的含义仅指存在某种情况, 不能够代表不变的

情形, 需要舍去. 因此, 在表 1 中, 最终用到的量词前缀包括$\forall e_i\forall e_j$、$\forall e_i\exists e_j$、$\forall e_j\exists e_i$这 3 种.

表 1　量词前缀

	$\forall e_i$	$\exists e_j$
$\forall e_i$	$\forall e_i\forall e_j(\Leftrightarrow\forall e_j\forall e_i)$	$\forall e_i\exists e_j$, $\exists e_j\forall e_i$
$\exists e_i$	$\exists e_i\forall e_j$, $\forall e_j\exists e_i$	$\exists e_i\exists e_j(\Leftrightarrow\exists e_j\exists e_i)$

选取逻辑连接词时, 首先分析$\prec$和$\not\prec$的含义, 事件 e_i 与事件 e_j 不具备偏序关系 $e_i\not\prec e_j\Leftrightarrow\neg(e_i\prec e_j)$在路径中表示可能事件 e_j 先于事件 e_i 发生, 即 $e_j\prec e_i$, 也可能事件 e_i 与事件 e_j 在路径中的向量时钟不可比较, 即不能顺序排序. 换言之, $e_i\not\prec e_j$ 所包含的信息多于 $e_j\prec e_i$. 接下来, 按照以下步骤进行逻辑表达式的删减.

① 除去永真式、永假式等没有实际意义的表达式.

由于 $e_i\not\prec e_j$与 $e_i\prec e_j$是互补的, 因此$e_i\not\prec e_j\vee e_i\prec e_j$、$e_j\not\prec e_i\vee e_j\prec e_i$、$e_j\not\prec e_i\vee e_i\not\prec e_j$是永真式, 而$e_i\prec e_j\wedge e_i\not\prec e_j$、$e_i\prec e_j\wedge e_j\prec e_i$、$e_j\prec e_i\wedge e_j\not\prec e_i$是永假式, 对于不变式的生成没有意义.

② 除去含义相同的表达式.

由于 $e_i\not\prec e_j$ 包含 $e_j\prec e_i$, 同理, $e_j\not\prec e_i$ 包含 $e_i\prec e_j$, 在剩余的表达式中, 由于包含关系导致含义相同的表达式有$e_j\not\prec e_i(e_j\not\prec e_i\vee e_i\prec e_j)$、$e_i\not\prec e_j(e_j\prec e_i\vee e_i\not\prec e_j)$、$e_i\prec e_j(e_i\prec e_j\wedge e_j\not\prec e_i)$、$e_j\prec e_i(e_i\not\prec e_j\wedge e_j\prec e_i)$, 括号内的表达式将被舍弃.

③ 除去在交换事件 e_i 与事件 e_j 时含义相同的表达式.

如$\forall e_i\forall e_j(e_i\prec e_j)$与$\forall e_i\forall e_j(e_j\prec e_i)$, 即交换了事件, 但不变式反映出的事件发生意义是一致的, 均为一个事件总在另一个事件之后发生. 这种在对称性下含义相同的表达式有 $e_j\prec e_i$ 与 $e_j\not\prec e_i$.

因此, 在表 2 中, 最终用到的逻辑表达式包括 $e_i\prec e_j$、$e_i\not\prec e_j$、$e_i\not\prec e_j\wedge e_j\not\prec e_i$、$e_j\prec e_i\vee e_i\prec e_j$ 这 4 种.

表 2　逻辑表达式

$\vee$ \ $\wedge$	$e_i\prec e_j$	$e_i\not\prec e_j$	$e_j\prec e_i$	$e_j\not\prec e_i$
$e_i\prec e_j$	$e_i\prec e_j$	$e_i\prec e_j\wedge e_i\not\prec e_j$	$e_i\prec e_j\wedge e_j\prec e_i$	$e_i\prec e_j\wedge e_j\not\prec e_i$
$e_i\not\prec e_j$	$e_i\not\prec e_j\vee e_i\prec e_j$	$e_i\not\prec e_j$	$e_i\not\prec e_j\vee e_j\prec e_i$	$e_i\not\prec e_j\wedge e_j\not\prec e_i$
$e_j\prec e_i$	$e_j\prec e_i\vee e_i\prec e_j$	$e_j\prec e_i\vee e_i\not\prec e_j$	$e_j\prec e_i$	$e_j\prec e_i\wedge e_j\not\prec e_i$
$e_j\not\prec e_i$	$e_j\not\prec e_i\vee e_i\prec e_j$	$e_j\not\prec e_i\vee e_i\not\prec e_j$	$e_j\not\prec e_i\vee e_j\prec e_i$	$e_j\not\prec e_i$

通过表 1 和表 2 得到的量词和逻辑表达式可以获得 12 种不变式组合, 但最终具有实际意义的不变式只包含表 3 给出的 5 种. 通常情况下, 越严格的量词条件对不变式的生成越有意义, 但也要根据实际情况决定.

表 3　时序不变式

表达式 \ 量词	$\forall e_i\forall e_j$	$\forall e_i\exists e_j$	$\forall e_j\exists e_i$
$e_i\prec e_j$		$e_i\rightarrow e_j$	$e_i\leftarrow e_j$
$e_i\not\prec e_j$	$e_i\nrightarrow e_j$		
$e_i\not\prec e_j\wedge e_j\not\prec e_i$	$e_i\|\|e_j$		
$e_j\prec e_i\vee e_i\prec e_j$	$e_i\|\!\dagger e_j$		

对于 $e_i\prec e_j$、$\forall e_i\forall e_j$ 相比$\forall e_i\exists e_j$ 和$\forall e_j\exists e_i$ 更为严格, 局限性更大, 要求涉及的两个事件必须同时出现在一条路径中, 这违背了实际情况. 如: 在区分节点与节点的不同后, 同一类事件也被划分为多种加编号的不同事件类型, 这些事件是无法保证一定在所有路径中同时出现的, 因此$\forall e_i\forall e_j(e_i\prec e_j)$被舍弃. 而$\forall e_i\exists e_j(e_i\prec e_j)$作为 $e_i\rightarrow e_j$, $\forall e_j\forall e_i(e_i\prec e_j)$作为 $e_i\leftarrow e_j$ 被保留(对于以存在量词为前提的事件, 相对于另一个事件类型只需要路径中存在某个事件实例满足不变式的约束即可, 同时存在不满足不变式的约束或向量时钟无法比较的情况也没有关

系). 如考虑一条执行路径 login→fail→login→success, 该路径描述了用户登录某个页面失败随后成功的过程. 在这条路径中, 对任意 login 事件, 不是每次都跟随着 fail 事件, 因此 login→fail 不成立; 而对任意 fail 事件, 用户继续做了 login 的尝试, 因此 fail→login 成立.

对于 $e_i \not\prec e_j$、$e_i \not\prec e_j \wedge e_j \not\prec e_i$ 和 $e_j \prec e_i \vee e_i \prec e_j$, 同样, $\forall e_i \forall e_j$ 相比 $\forall e_i \exists e_j$ 和 $\forall e_j \exists e_i$ 更为严格, 但这 3 种表达式的意义关注的是在路径中两个事件同时出现的情形, 所以后两种量词前提所宽限的范围将使得不变式不够精确. 需要注意的是: 即使某个事件单独出现在某条路径中, 也不会影响两个事件同时出现形成的不变式关系. 仍考虑 login→fail→login→success 这条路径, 另有一条路径 login→success. 对每个 success 事件, 其后均没有 login 事件发生, 因此 success$\nrightarrow$login 成立. 路径包含的有关||与|不变式, 在给出 5 种不变式的定义和性质之后描述.

两个事件之间的时序不变式以如下形式出现, 即可被判定为真.

(1) $e_i \rightarrow e_j$, 表示事件 e_j 总是在事件 e_i 之后发生, 即 $\forall e_i \exists e_j (e_i \prec e_j)$;

(2) $e_i \nrightarrow e_j$, 表示事件 e_j 从不在事件 e_i 之后发生, 即 $\forall e_i \forall e_j (e_i \not\prec e_j)$;

(3) $e_i \leftarrow e_j$, 表示事件 e_i 总是在事件 e_j 之前发生, 即 $\forall e_j \exists e_i (e_i \prec e_j)$;

(4) $e_i \| e_j$, 表示事件 e_i 总是与事件 e_j 并行发生, 即 $\forall e_i \forall e_j (e_i \not\prec e_j \vee e_j \not\prec e_i)$;

(5) $e_i \nparallel e_j$, 表示事件 e_i 从不与事件 e_j 并行发生, 即 $\forall e_i \forall e_j (e_i \prec e_j \vee e_j \prec e_i)$.

关于以上时序不变式, 还存在以下几条性质.

① 并行发生意味着两个事件不存在确切的先后顺序, 不并行发生则意味着两个事件存在确切的先后顺序;

② $\nrightarrow$与$\nleftarrow$逻辑上是等价的, 均描述了事件 e_j 从不在事件 e_i 之后发生, 因此不考虑$\nleftarrow$;

③ 若 $e_i \nrightarrow e_j$ 与 $e_i \| e_j$ 均成立, 即事件 e_i 和事件 e_j 总是并行发生, 没有固定的先后顺序, 因此需舍弃 $e_i \nrightarrow e_j$, 因为||比$\nrightarrow$更严格;

④ 如果事件 e_i 和事件 e_j 存在固定的先后顺序, 或者 e_i 总先于 e_j, 或者 e_i 总跟随 e_j, 满足 $e_i \rightarrow e_j$ 或者 $e_j \leftarrow e_i$, 这时也将同时满足 $e_i \nparallel e_j$, 但 $e_i \nparallel e_j$ 包含的含义范围更广, 条件较弱, 需要舍弃.

反映在时序不变式的定义中, 性质③描述了如果 $\forall e_i \forall e_j (e_i \not\prec e_j)$ 与 $\forall e_i \forall e_j (e_i \not\prec e_j \wedge e_j \not\prec e_i)$ 同时成立, 将两式作与运算, 依然仅得到||不变式. 在一条路径中, 两个事件的向量时钟始终不可比较(由向量时钟的定义决定)时, ||不变式成立.性质④描述了如果 $\forall e_i \exists e_j (e_i \prec e_j)$ 与 $\forall e_i \forall e_j (e_i \prec e_j \vee e_j \prec e_i)$ 同时成立, 将两式作与运算, 得到 $\forall e_i \forall e_j (e_i \prec e_j)$, 此时表达 e_j 总跟随 e_i 更加严格. 如在单一的 login→success 路径中, 时序不变式 login→success 和 login|success 都是成立的, 此时将仅保留 login→success.

1.5 挖掘时序不变式

挖掘时序不变式的过程是利用日志总结系统运行过程记录的事件之间的时序相关性的过程, 可以概括为 $M[P(L): \vec{T}] \rightarrow I(e_i, e_j)$. 其中, M 表示挖掘算法, P 表示日志解析, L 表示日志, $\vec{T}$ 表示向量时钟, I 表示时序不变式,e 表示事件.

日志(log)是多条系统执行路径的集合, 而每条路径则是一组事件的有序序列, 每个事件实例(event)通常包含时间戳、所在位置(如 IP 地址或主机名)、事件描述字段(如动作信息、异常值)等. 某些日志也可能包含事件相关的其他信息, 如日志类型、路径 ID、事件 ID 等. 从日志中挖掘时序不变式(invariant)就需要从上述事件信息中分析得出事件之间的时序关系, 其过程为: 首先对日志解析(parse), 得到多条路径所包含的关键的事件信息, 由于日志具有特异性, 需要利用不同的正则表达式分析得出事件名称以及所在位置; 然后, 利用事件发生的位置、路径记录的部分事件的相关信息计算得出向量时钟, 通过比较向量时钟得出事件之间的偏序关系; 最后, 利用时序不变式的挖掘方法处理这些偏序关系得出最终的结果.

1.6 现有挖掘方法

Beschastnikh 设计的 Synoptic 工具[4]与本文的研究最相近, 后续的大部分工作也都是围绕 Synoptic 进行的,

如 Perfume[8]、SpecForge[9]、InvariMint[10]、Texada[11]、CrowdSpec[12]等, 都是在 Synoptic 的基础上做一些改进来挖掘更易于接受的时序不变式结果. 在 Synoptic 工具中, 作者定义了第 1.4 节中的 5 种时序不变式, 分别用基于传递闭包和统计量的算法挖掘出了不变式. 由于生成的后两种不变式数量较多, 随后的工作主要处理前 3 种不变式, 如: Perfume[8]扩展了资源使用等性能约束的时间属性; SpecForge[9]增加了两事件之间没有其他事件的约束, 形成了另外 3 种更紧凑的不变式; InvariMint[10]从 k-Tails[13]算法出发, 为算法生成的状态机提供一种描述性的实现; Texada[11]为不变式增加支持度和置信度阈值来挖掘近似的时序不变式; CrowdSpec[12]基于 SpecForge, 通过增加人工处理提高自动挖掘得到的不变式的准确性.

在这些工作中, 常见的挖掘不变式的方式是利用人工生成的日志数据或某些具有特定形式的数据作为数据来源, 将用户自定义的正则表达式应用到日志数据中分离出算法可接受的输入形式, 即日志中有价值的数据. 然后使用不同的挖掘算法(如基于 k-Tails[13]算法衍生出的改进方法)处理这些数据, 分析出事件之间的时序关系, 再利用线性时序逻辑 LTL 或 Dwyer[14]提出的属性规约模式等方法描述得到的时序不变式, 也可以进一步使用 FSM 或有向图等方式直观地描述结果. 以上的流程在得出不变式结果后, 应用它们进行系统调试的主要方式是研究不变式是否符合预期的结果, 为调试提供更简洁、直观的判断依据.

本文主要选用 Synoptic 工具作为对比方法, 尽管前 3 种不变式对于时序属性的研究十分关键, 但同样不能忽视后两种不变式. 因为在分布式系统中, 其日志记录的事件在调用过程中都离不开对并行关系与不并行关系的研究. 这两种不变式反映的不是简单的在某一路径或时间范围内两个事件的先后顺序, 而是能够一定程度地描述在一条执行路径中事件调用的分支情况, 从而更好地了解事件的运行过程. 其他比较相近的工作如 Tempsy[15]基于模型的方法研究 12 种时序属性, 涵盖了事件的存在性和发生的优先顺序, 并加入了时间距离用于更细粒度地表述事件之间的时间长度, 进而反映事件的时序相关信息. 然而这些时序属性同样依赖于分析传统日志, 对于并行发生的事件, 使用传统时间戳的距离会导致其大部分时序属性失去意义.

2 SOTIMiner

在借鉴并系统分析了 Synoptic 工具中的 5 种时序不变式的表达方式后, 本节提出了一种基于集合运算的时序不变式挖掘方法 SOTIMiner. 首先, 结合一个例子介绍 SOTIMiner 算法的思想; 然后给出 SOTIMiner 算法的具体过程, 对算法的时间复杂度及效率进行分析; 最后提出能够使 SOTIMiner 算法与用户操作相结合的改进方案.

2.1 SOTIMiner算法描述

从日志中寻找时序不变式的方法有很多, 但大多数都离不开统计的方式. 利用矩阵或者数组等方法对日志中事件之间的前后关系进行判断和出现的次数进行累加, 都是常用的处理手段. 但是由于需要对跟随关系和先于关系进行统计和计算, 通常都需要正向遍历和反向遍历两次遍历系统执行路径, 将路径中的每个事件作为一个单独的元素, 计算出路径中与该事件相关的所有事件的统计结果, 并在最后将统计结果与基准结果进行条件判断才能得到正确的不变式. 本文希望能够实时计算出第 1.4 节所给出的 5 种时序不变式, 而无需两次遍历执行路径和复杂的条件判断, 因此采用了集合求解角度出发的计算方法. SOTIMiner 算法的基本思想是: 通过对事件的前驱或后继集合整体而不是对每个事件元素进行运算, 力求使用最少的步骤得到所需要的结果. 这里, 结合一个具体的例子来阐述该方法的计算流程.

假设系统中存在两条执行路径, 即包含两棵请求树. 这两棵请求树的形状如图 2 所示. 两条执行路径共涉及到 2 个主机节点(n_1,n_2), 4 种事件类型(A,B,C,D), 其中, 事件 A、C、D 发生在主机 n_1 上, 事件 B 发生在主机 n_2 上. 为了在一个简洁的形式下包含尽可能多的情形, 这两条路径仅作为举例, 并不包含过多的实际意义.

SOTIMiner 算法首先根据这两棵请求树来计算每个事件的向量时钟, 每两个事件比较向量时钟后得到事件间的偏序关系. 例如: 在请求树(b)中, 首先, 主机 n_1 上发生 A 事件, 向量时钟在 n_1 索引值加 1, 得到[1,0]; 随后, 同样在主机 n_1 上发出远程过程调用请求, 发生 C 事件, 向量时钟更新为[2,0]; 主机 n_2 收到该请求, 发生 B 事件, 向量时钟在 n_2 索引值加 1, 本地向量时钟更新为[0,1], 由于与 n_1 发生交互需根据规则合并数组并取较大

分量值, 实际向量时钟更改为[2,1]; 主机 n_1 发生 D 事件, 由于与 n_2 未发生交互, 不进行合并数组操作, 只在 n_1 索引值加 1, 得到[3,0].

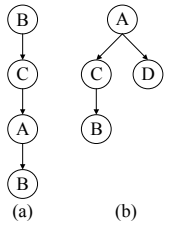


图 2 两棵请求树

获得每个事件的向量时钟后, 还需获得每个事件的前驱和后继集合. 对于某个事件 e, 其前驱和后继特指与本身存在偏序关系的事件. 如前驱事件集合包括请求树中事件的向量时钟小于事件 e 向量时钟的事件集合, 而不是只能通过有向边到达事件 e 的事件集合. 这时, 事件 e 的前驱集合 P_e 为与事件 e 间偏序关系成 $P_e \prec e$, 同理得到事件 e 的后继集合 S_e, 与事件 e 间偏序关系成 $S_e \succ e$. 例如: 在请求树(a)中, 事件 B 出现了两次, 先出现的 B 事件的后继集合为{A,B,C}, 后出现的 B 事件的后继集合为空集∅; 而在请求树(b)中, 事件 B 的向量时钟为[2,1], 事件 D 的向量时钟为[3,0], 不能够顺序排序, 因此 B 事件的后继集合为空集∅. 得到事件的前驱集合和后继集合后, 即可利用公式和规则对不变式求解. 算法 1 展示了 SOTIMiner 方法求解时序不变式的伪代码.

算法 1. *ExecutionPathBased*(L,P_e,S_e,U'_e).

输入: 路径日志, 事件的关系集合: 前驱集合 P_e, 后继集合 S_e, 在同一棵请求树发生的事件集合(共生集合) U'_e;

输出: 挖掘得到的 5 种时序不变式集合.

1. 计算事件的全集 U;
2. **for each** $e \in L$ **do**
3. $\tilde{P}_e \leftarrow \bigcap_{j=1}^{n_j} P_e$, $S_{\rightarrow} \leftarrow \bigcap_{j=1}^{n_j} S_e$, $S_{e\cup} \leftarrow \bigcup_{j=1}^{n_j} S_e$, $U'_{e\cup} \leftarrow \bigcup_{j=1}^{n_j} U'_e (n_j = \sum_{i=1}^{n} n_i)$
4. 计算向量时钟不可比较的事件集合 $T_e \leftarrow \bigcup_{j=1}^{n_j} (U'_e - P_e - S_e)(n_j = \sum_{i=1}^{n} n_i)$
5. **end for**
6. **for each** $e \in U$ **do**
7. $S_{e\not\rightarrow} \leftarrow U - S_{e\cup}$
8. reorganize $\tilde{P}_e$ to $P_{e\leftarrow}$
9. **end for**
10. **for each** $e \in U$ **do**
11. **for each** $e' \in S_{e\not\rightarrow}$ **do**
12. **if** $e \in S_{e'\not\rightarrow} \cap U'_{e'\cup}$ **then**
13. $S_{e\|} \leftarrow e'$
14. **end if**
15. $S_{e\not\rightarrow} \leftarrow S_{e\not\rightarrow} - S_{e\|}$
16. $S_{e\|} \leftarrow U'_{e\cup} - T_e - S_{e\rightarrow} - P_{e\leftarrow}$
17. **end for**
18. **for each** $e' \in S_{e\|}$ **do**
19. **if** $e \notin S_{e'\|}$ **then**
20. $S_{e\|} \leftarrow S_{e\|} - e'$

21. **end if**

22. **end for**

23. **end for**

24. **return** $S_{e\rightarrow}$, $P_{e\leftarrow}$, $S_{e\nrightarrow}$, $S_{e||}$, $S_{e|\!|}$

算法(第 1 行–第 5 行)首先为运算准备好与事件 e 相关联的集合, 包括事件的全集、在执行路径中与事件 e 同时出现过的所有事件的集合(共生集合的并集)、所有与事件 e 的向量时钟不能顺序排序的事件集合 T_e、事件 e 的前驱和后继的交集等集合. 利用以下 5 个步骤(第 6–24 行)得到请求树所能得出的所有时序不变式.

① 求解某事件 e 后总存在的事件的集合(第 2 行).

将所有请求树中有关事件 e 的后继集合取交集, 表示每次事件 e 发生时, 都会存在于后继集合中的事件, 即 $\bigcap_{j=1}^{n_j} S_e(n_j=\sum_{i=1}^{n} n_i)$ (这里, n_i 表示第 i 棵请求树中该事件出现的次数, 下同). 例如: 计算事件 B 后总存在的事件, 则需用对应请求树中 3 次出现的事件 B(请求树(a)中 2 次, 请求树(b)中 1 次)的后继集合取交集, 即 $\{A,B,C\}\cap\varnothing\cap\varnothing=\varnothing$, 表示事件 B 后不固定发生某个事件.

② 求解某事件 e 后从不发生的事件的集合.

将所有请求树中有关事件 e 的后继集合取并集的补集(第 7 行), 表示从全部事件中除去在该事件后发生过的所有事件集合, 即 $U-\bigcup_{j=1}^{n_j} S_e(n_j=\sum_{i=1}^{n} n_i)$. 之后, 还需要在结果中舍弃两事件为总并行事件的结果(第 15 行), 因为||比$\nrightarrow$更严格(见第 1 节的说明④). 例如: 在两棵请求树中, 事件 D 后均未发生任何事件, 上式的结果为 $\{A,B,C\}$, 但是由于事件 B 与事件 D 之间的向量时钟无法比较, 不存在固定的先后顺序, 为总并行事件, 因此需在结果中除去事件 B, 则最终事件 D 后从不发生的事件的集合为 $\{A,C\}$.

③ 求解某事件 e 总先于的事件的集合.

首先将所有请求树中有关事件 e 的前驱集合取交集, 表示每次事件 e 发生时, 都会存在于前驱集合中的事件, 即 $\bigcap_{j=1}^{n_j} P_e(n_j=\sum_{i=1}^{n} n_i)$, 随后将结果整合(第 8 行). 例如: 根据事件 D 的前驱集合 $\{A,C\}$ 得到 $D(\leftarrow)$:$\{A,C\}$, 再调换顺序, 结果为 $A\leftarrow D$ 和 $C\leftarrow D$, 表示事件 A 总先于 D, 事件 C 也总先于 D.

④ 求解与某事件 e 总并行的事件的集合(第 12–14 行).

在事件 e_1 和 e_2 均发生的请求树中, e_1 存在于 e_2 后从不发生的事件集合中, 且 e_2 存在于 e_1 后从不发生的事件集合中, 即 $e_2\in\{U'-\bigcup_{j=1}^{n_j} S_{e_1}\}\wedge e_1\in\{U'-\bigcup_{j=1}^{n_j} S_{e_2}\}(n_j=\sum_{i=1}^{n} n_i)$ (这里, U'表示在请求树中与某事件共同发生的全集, 下同), 这时将满足两事件总并行的条件. 例如: 在请求树(b)中, 事件 B 与事件 D 均发生, 事件 B 后从不发生的事件集合为 $\{D\}$, 事件 D 后从不发生的事件集合为 $\{A,B,C\}$, 相互包含, 因此 $B||D$ 成立.

⑤ 求解与某事件 e 总不并行的事件的集合.

在请求树中与事件 e 一同发生的事件集合中, 需要去掉所有具有严格时序关系的事件集合以及所有不能比较顺序的事件集合 T_e, 包括事件 e 总先于的集合、总跟随事件 e 的集合(第 16 行)、总先于事件 e 的集合、事件 e 总跟随的集合(见第 1 节的说明⑤), 即 $U'-\bigcap_{j=1}^{n_j} S_e-\bigcap_{j=1}^{n_j} P_e-\tilde{S}_e-\tilde{P}_e(n_j=\sum_{i=1}^{n} n_i)$ (这里, $\tilde{S}_e$、$\tilde{P}_e$ 分别表示 e 总跟随的集合和总先于 e 的集合), 这时反映在请求树中的事件关系是向量时钟是可比较的. 在算法中, $\tilde{S}_e$ 和 $\tilde{P}_e$ 是通过查询与事件 e'相关的包含事件 e 的集合进行移除解决的(第 18–22 行). 例如: 在上述请求树中, 事件 A 要么发生在事件 C 之后($A\succ C$), 要么发生在事件 C 之前($A\prec C$), 因此是从不并行发生的.

2.2 算法分析

以上就完成了对 5 种不变式的求解, SOTIMiner 算法的时间复杂度与系统中事件的数量和事件的类型以及进行的集合操作有关. 在 Python 的内置数据结构中, 集合对于本算法较为实用, 原因是算法运行时处理的对象都是无序的且任意集合所包含的事件数量不会多于 m. 这里, m 为事件类型的数量. 因此, 进行集合运算的算法复杂度为 O(m), 而判断元素是否在集合中的复杂度为 O(1). 算法需要一次遍历所有的事件用于添加事件的前驱集合和后继集合(第 2 行–第 5 行), 这时的开销为 O(n), 这里, n 是事件的数量; 随后进行的集合运算

是建立在事件类型的数量上, 算法将相同的事件类型合并后统计出每种事件应该满足的不变式, 由于步骤③–步骤⑤均涉及在事件类型全集中对某一事件类型的循环判断, 做交换、插入、删除等操作(第 10–23 行), 开销为 $O(m^2)$. 因此在实际应用中, 如果事件类型不多而重复发生次数很多, 则适宜采用该方法.

基于日志的时序不变式挖掘方法普遍采用的是统计的方法, 即对日志中事件之间的时序关系进行判断和计数, 在此基础上分析各类时序不变式. 由于涉及 $e_i \rightarrow e_j$ 和 $e_i \leftarrow e_j$ 两类时序不变式, 这些方法通常都需要正向遍历和反向遍历路径数据, 对任意两事件在请求树中的前驱和后继关系下重复发生的次数进行统计, 才能得到所需的统计量[4]. 而本文采用的方法进行正向遍历时, 在请求树中每遍历到一个节点, 就可以为该节点更新出前驱和后继集合. 此外, 统计的方法在得到统计结果后, 还要根据计算公式对每对事件在请求树中的前驱和后继关系次数进行卷积计算, 而 SOTIMiner 不需要进行这一步操作, 最终的集合即为得到的结果.

2.3 改进方案

如何准确、快速地定向寻求问题的根源, 一直是分析系统软件行为的难点, 无论是日志还是调试信息都很难直接瞄准问题的关键所在, 通常它们只能告诉用户异常属于某一类, 而无法具体而准确地进行定位. 真正的异常或错误需要用户自己调试探索. 如果能够将这些指向不明确的提示信息转换为特定的函数、端口或事件, 则使得用户更加容易发现异常所在, 意味着系统软件更加智能化. 当系统出现故障时, 用户第一时间希望了解到故障位置和故障原因, 这时时序不变式能做的就是比较出故障系统违反了的不变式种类, 确定可疑的异常事件以进行定位, 利用正确的不变式推测出故障原因, 用户再根据以上信息寻求解决方案. 由于偏序日志的自身特性导致时序不变式的数量随着事件种类的增加而以幂级数增加, 大多数时间用户不易从中寻找到自己真正需要的不变式, 为挖掘出这些不变式也会消耗大量的时间. 挖掘用户关心的不变式而不是所有的不变式, 对用户来说是有利的. 如果用户能够选取自己想查看的问题节点或事件, 既能减少挖掘不变式的工作量, 又能节约用户的查询时间. 因此, 提高不变式挖掘的针对性, 对于寻找错误事件根源这类问题的理解、排查以及解决都有较好的实用性.

针对性与效率的提高主要涉及到 3 个方面——特定的节点事件种类、特定的不变式种类以及不变式的挖掘方式: ① 选用特定的节点事件种类是指用户如果预知了故障节点的位置, 挖掘不变式进行排查时则针对该节点处理, 得到未发生或错误发生的事件, 有必要时可以利用该事件回溯, 找到导致故障事件问题的根源事件; ② 选用特定的不变式种类是指用户可结合自身的需要研究选择适用于排查故障的不变式, 这样在用户筛选的过程中也可减轻挖掘不变式的负担, 但不同不变式的挖掘之间会相互关联, 这时需要牺牲一部分空间保存关联的不变式以保持效率的提升; ③ 不变式的挖掘对象主要为全序日志和偏序日志. 对偏序日志处理得到的多数不变式具有相同的含义, 这是由于偏序日志区分了不同主机上的同名事件, 尽管可以挖掘出粒度更细的时序不变式, 将各个主机上发生的事件更加细致地表达, 但却导致同一类型的不变式经主机编号划分后被重写了多次. 出现这种情况是研究偏序日志划分事件类型时存在的固有问题. 而全序日志将不同主机上的同名事件认为是同一事件, 得到的不变式数量较少而且关键, 但也会丢失一些细粒度的不变式. 这时将全序日志与偏序日志相结合, 权衡这类不变式的表示, 可以描述得更加简洁和直观, 并不丢失有意义的细粒度不变式. 总体来说, 由于全序日志具有绝对的事件顺序, 事件关系简单, 不变式易于挖掘, 偏序日志没有绝对的事件顺序, 事件关系相对复杂, 有些不变式不易被挖掘, 但所能描述的信息更多. 用户可根据需求选用这两种日志获得不变式, 寻求更高程度的准确性.

方案的另一个角度是为算法优化预处理环节. 在数据库中真正存储的不是已经构建好的执行路径, 而是数据表, 包括记录请求信息的 Task 表、每个事件相关信息的 Report 表、有向边信息的 Edge 表等表格. 预处理阶段就是需要利用这些表的信息实现请求树的重构, 重构的目的是形成直观的执行路径, 为用户了解路径结构提供便利. 重构请求树的方法是首先将数据库中存储的 Report 表中的事件信息依据发生的时间顺序整理成树形拓扑结构, 保证每个事件节点最多有一个父节点; 在得到拓扑结构后, 借助深度优先搜索方法将节点按照树的前序遍历的形式进行排列, 对应于系统中事件发生的时间顺序由前至后; 为了得到完整的偏序日志, 需要将向量时钟按照得到的路径排序为每个事件逐一添加. 在该序列中, 是否利用第 1.3 节中的规则(2)取决

于相邻两事件是否为父子关系, 即是否存在一条边. 如果存在边, 即当后一事件在请求树中的深度大于前一事件在请求树中的深度时, 需要正常合并两向量时钟的较大分量并赋予后一事件, 反之, 按照规则(1)在对应分量加 1 即可; 得到向量时钟后, 即完成了预处理环节, 但如果执行路径中节点种类较多(事件类型较多)时, 预处理速度较慢. 可以选择采取以下 3 种做法以缩短预处理时间: ① 根据用户的查询条件对数据集进行筛选, 减少许多不相关的无意义的计算, 从而缩短预处理时间; ② 由于向量时钟数组的长度为日志中节点数量, 而在一个执行路径中所涉及的节点可能只是其中的少部分, 因此在求偏序关系之前, 需要对向量时钟根据需要进行处理, 处理为相关节点的向量; ③ 由于在同一主机中, 事件保持绝对的先后顺序, 偏序关系必然成立, 因此求偏序关系时仅考虑不同主机的事件.

3 实验和讨论

本节通过实验对 SOTIMiner 的功能和效率进行验证, 首先介绍实验设置, 然后与 Synoptic 进行对比实验, 分析 SOTIMiner 及其改进方案的效率和准确性, 最后对方法进行讨论.

3.1 实验设置

实验数据选用 TraceBench 数据集[16]和 CIDD[17]数据集: TraceBench 是一个利用白盒插桩技术收集的细粒度开源路径数据集, 该数据集包含很多子数据集, 分别记录了不同场景下 HDFS 中文件读、文件写、RPC 等请求的执行路径; CIDD 是一个面向云入侵检测领域的开源数据集, 主要记录了 Linux 系统在执行用户请求时的系统调用顺序的路径. 表 4 给出了实验所用数据集的具体信息, 包括每个数据集中包含的路径数量、事件类型数量、事件实例数量. 根据两个数据集中的数据格式进行转换, 使其符合 SOTIMiner 和对比方法的要求.

表 4 数据集信息

数据集		路径数量	事件类型数量	事件实例数量
Tracebench	NM_CL_w_05	49	244	8 727
	NM_CL_rpc_05	339	572	10 822
	NM_CL_r_05	211	250	24 837
	NM_CL_r_10	415	300	50 079
	NM_CL_r_15	640	350	77 451
	NM_CL_r_20	788	400	94 367
CIDD		132	50	112 346

由于 Synoptic 生成的不变式的种类与 SOTIMiner 相同, 且均是由日志生成的有意义的不变式, 因此实验选用 Synoptic 与 SOTIMiner 进行对比. Synoptic 提供了 3 种时序不变式挖掘方法, 分别用 Synoptic-0、Synoptic-1、Synoptic-2 表示. Synoptic-0 是一种基于传递闭包的算法, 而 Synoptic-1 和 Synoptic-2 均为基于统计量的方法, 其中, Synoptic-2 将 Synoptic-1 中|/|的计算删去, 仅能求得 4 种时序不变式.

SOTIMiner 的实现语言为 Python, 实验所用机器为笔记本电脑, 配置为 Intel i5 (2.5 GHz) CPU、4 GB 内存, 运行环境为 Windows 8 操作系统上的虚拟机, 虚拟机分配了 1 GB 内存、30 GB 存储空间.

3.2 挖掘结果

为了对本文提出的算法准确性与实效性进行分析, 需要从功能和效率两方面设计实验来加以验证. 本文分别用 Synoptic 和 SOTIMiner 对 CIDD 和 TraceBench 数据集进行不变式挖掘. 处理 CIDD 数据集时, 依据其中的会话 ID 对数据集进行划分, 再按照执行的系统调用所在目录恢复请求路径, 最后利用算法获得系统调用之间的时序不变式. 通过对比发现: 两种方法挖掘的不变式完全相同, 均包含 1 243 个不变式. 下面给出挖掘 CIDD 数据集得出的若干时序不变式的示例及描述.

(1) *mmap*←*munmap*: 其中, *mmap* 系统调用表示将一个文件或者其他对象映射进内存, *munmap* 系统调用表示解除内存映射. 其含义为映射对象进入内存操作总先于解除内存映射的操作;

(2) *lstat*←*fchdir*: 其中, *lstat* 系统调用表示获取文件相关的信息, *fchdir* 系统调用表示更改当前工作目录. 不变式说明用户总是在更改工作目录之前查看文件的相关信息, 如在 shell 命令解释器中, 用户在执

行 *cd* 命令前总会执行 *ls* 这样的命令来选择将工作目录更改到何处. 通过挖掘时序不变式, 也可以发现用户的某些偏好行为.

挖掘 TraceBench 数据集时, 两种方法也能得到相同的不变式结果. 表 5 给出了其中部分数据集挖掘得到的时序不变式数量及示例.

表 5 时序不变式数量及示例

	NM_CL_w_05	NM_CL_rpc_05	NM_CL_r_05	示例
$e_i \rightarrow e_j$	415	231	485	RPC: getListing_0→getListing_50
$e_i \nrightarrow e_j$	18 161	9 564	13 511	RPC: complete_0↛fs-ls_0
$e_i \leftarrow e_j$	376	105	477	fs-ls_0←RPC: getListing_0
$e_i \| e_j$	13 690	100	2 948	getFileInfo_50‖RPC: getListing_0
$e_i \nparallel e_j$	3 001	45	19 325	RPC: getFileInfo_0∦complete_50

以上结果表明, SOTIMiner 能够挖掘出有价值的时序不变式. 但是与 Synoptic 类似, SOTIMiner 将系统表现出来的行为近似认为是系统固有的内在规律, 这会导致挖掘结果不可避免地存在误报和漏报. 误报是指在系统中将不具有时序相关性的两个事件报告存在时序不变式, 如 CIDD 数据集中报告 kill→setpgrp, 而实际上 kill(杀死进程)和 setpgrp(将目前进程所属的组识别码设为目前进程的进程识别码)不存在确定的先后顺序. 漏报则是指将具有时序相关性的两个事件报告不存在时序不变式, 如无法从 NM_CL_r_50 数据集中挖掘出 blockSeekTo_16→chooseDataNode_16, 这是由于节点 16 有一次没有选择数据节点发送数据块所致, 大部分情况下该不变式是成立的.

为了进一步量化分析结果的有效性, 为数据集 NM_CL_r_05 构建时序不变式基准[18], 并据此计算该数据集挖掘结果的查准率和查全率. 其中, 查准率=挖掘出的正确不变式的数量/挖掘出的不变式的总数×100%, 查全率=挖掘出的正确不变式的数量/基准中的不变式的总数×100%. 表 6 给出了 5 种不变式的结果. 可以看到, 各不变式的查准率均处于较高水平, 说明 SOTIMiner 可以较为准确地发现系统已表现出的时序特征. 但是, 作为一种基于日志的挖掘方法, SOTIMiner 不能有效发现系统未表现出的行为, 而部分类型的时序特性在数据集 NM_CL_r_05 中的展示不够充分, 导致对应类型的时序不变式查全率相对偏低. 此外, 系统异常等问题会导致一些偶发的事件时序组合, 这也会影响挖掘结果的有效性, 第 3.5 节对此进行了具体分析并给出了相应的改进方案.

表 6 时序不变式的查准率和查全率 (%)

		$e_i \rightarrow e_j$	$e_i \nrightarrow e_j$	$e_i \leftarrow e_j$	$e_i \| e_j$	$e_i \nparallel e_j$	总计
NM_CL_r_05	查准率	100.00	81.88	100.00	65.81	96.42	91.01
	查全率	13.80	76.77	91.73	60.66	97.02	85.12

3.3 方法效率

通过在 TraceBench 数据集上运行 SOTIMiner 和 Synoptic 中的 3 种方法, 验证 SOTIMiner 的效率. 实验主要从两个维度进行: 一个维度是数据集中涉及的请求类型, 不同请求类型具有不同的路径长度和事件类型数量等, 从而影响了挖掘速度; 另一个维度是客户端数量, 客户端数量越多, 则对应数据集中路径数也越多, 即数据规模越大, 这也会影响到挖掘的效率. 在计算方法运行时间时, 对 5 次运行时间取平均值作为结果.

图 3 给出了 4 组实验的结果. 不同组实验所用数据集涉及的请求类型不一样, 图 3(a)对应实验涉及的请求类型为 HDFS 中的文件读请求(用 *r* 表示)、图 3(b)为文件写请求(用 *w* 表示)、图 3(c)为 RPC 类请求(用 *rpc* 表示)、图 3(d)包含 3 种请求. 对于每组实验使用的 4 个数据集, 涉及的请求类型相同, 但客户端数量不一样, 分别为 5、10、15、20. 图中横坐标给出了所用数据集, 纵坐标给出了不同算法在这些数据集上的挖掘时间, 单位为 s. 可以看出: 无论涉及何种请求类型以及无论客户端数量多少, SOTIMiner 的挖掘速度均要快于另外 3 种挖掘算法. 即: 在不同规模、不同路径长度、不同事件类型的情况下, SOTIMiner 的效率均高于另外 3 种算法.

具体如下.

(1) 在时间利用率方面, Synoptic-2 相比 Synoptic-1 能够有一些提升, 但其实是忽略了对不变式|t|的求解, 减少了约 50%的不变式的计算所导致的, 尽管多数情况下该不变式对于用户解决问题的意义不大, 也不便于用户理解. 在 SOTIMiner 中, 直接计算事件集合替代对事件的顺序统计, 可以较大地缩短求解时间;

(2) 在空间利用率方面, 由于 Tracebench 数据集包含的执行路径数量较多, Synoptic 采用的方式需要维护较多的双向链表集合, 不能够及时地释放空间, 有时会产生空间不足的情况, 需要扩大 Java 堆栈的容量;

(3) 三者综合的路径数据对于挖掘速度有显著的影响. 软件系统中存在的更多的是这些复杂的操作环节, 操作越复杂, 涉及到的事件类型越多, 对挖掘的效率影响越大.

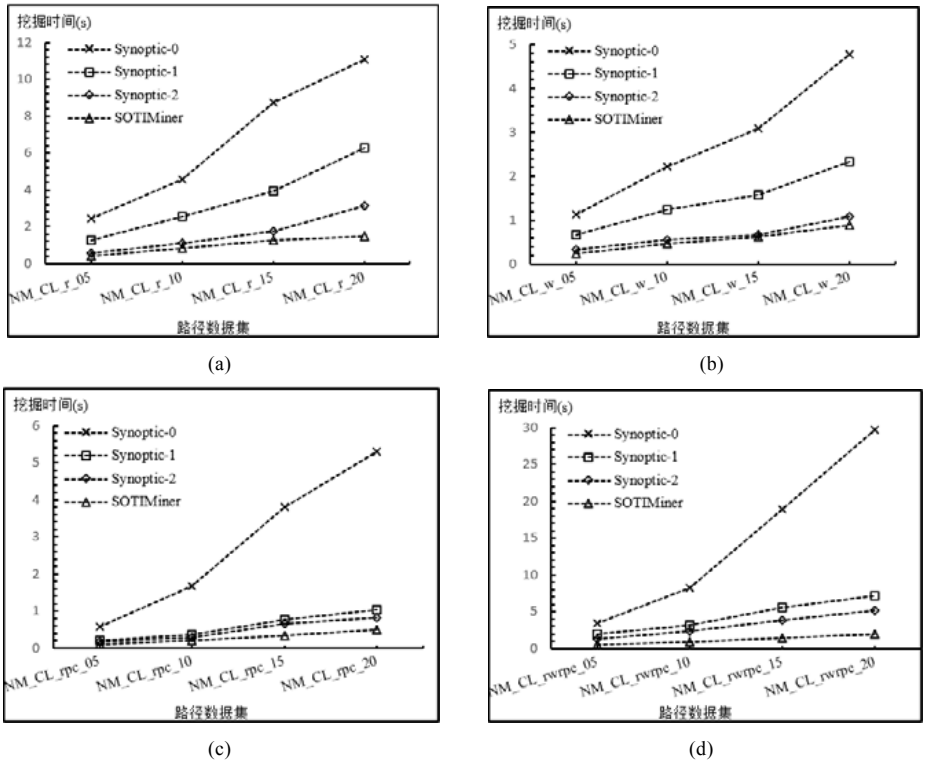


图 3 4 种挖掘方法的速度对比

3.4 对比方法分析

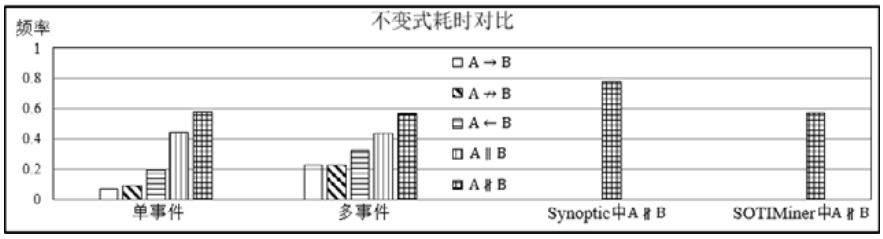
Synoptic-0 基于传递闭包分析时序不变式. 一个事件 e 的传递闭包是指在有向无环图中, 可以经有向边到达 e 或者从 e 经有向边可以到达的事件集合. 在实际案例中, 一棵请求树将对应于这些传递闭包的组合, 并表示为一个可达矩阵. 将两事件可达表示为 1, 否则表示为 0. 如果在各个可达矩阵中两事件的可达性总保持不变, 则说明它们具有固定的时序关系. Synoptic-1 和 Synoptic-2 基于统计量分析事件之间的联系, 首先对各事件发生的次数, 两事件在同一请求树中发生的次数等计数, 然后根据设定的规则推理出时序不变式. 例如: 如果在数据集中的所有请求树中, 事件 e_j 在事件 e_i 之后发生的次数与事件 e_i 发生的次数相等, 则认为 $e_i \rightarrow e_j$. Synoptic-2 发现了在利用前两种算法计算不变式|t|时需要维护所有事件先于或跟随的事件集合, 即每一对事件之间是否具有偏序关系, 而其他 4 种不变式则仅维护某一事件类型先于或跟随的事件集合. 在观察到挖掘|t|导致开销显著增加后, Synoptic-2 不进行|t|的计算.

Synoptic 所采用的两类挖掘方法均需多次遍历日志数据来维护统计结果, 进而研究结果之间的数值关系得出不变式, 这种方式涉及到分析日志中每个有向无环图中的边与事件类型. 利用边分析出事件的前驱和后继关系, 遍历边的同时合并事件类型所包含的数量 m, 为比较统计结果做准备. 由于有向无环图中边的数量与节点数量 n 是近似一致的, 因此遍历日志数据的开销为 O(mn). 在分析统计结果时, 需要遍历任意两个事件类型的数量, 通过运算和比较条件分析出时序不变式, 开销为 O(m^2). 而通常情况下, 日志中包含大量的事件实例而只有较少的事件类型(见表 4), 表现为 $m<<n$, 因此, Synoptic 算法最终的时间开销约为 O(mn), 大于 SOTIMiner 的开销 O(m^2).

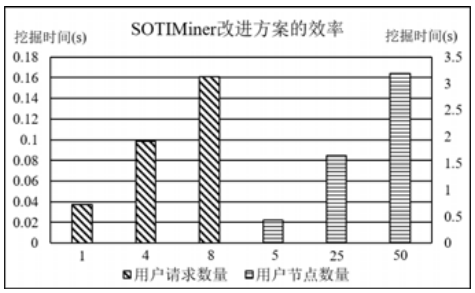
3.5 改进结果分析

总体来说, SOTIMiner 在保证了计算所有不变式的前提下, 能够有效地减少挖掘结果的时间和空间. 在对实现的 SOTIMiner 改进方案进行分析时, 首先对 5 种不变式具体的耗时比例作简要的对比, 以说明提高挖掘不变式的针对性的好处, 然后分析增加事件类型数量对改进方案的影响.

如图 4(a)所示, 查询单个事件与查询多个事件是指用户查询某一个事件或某几个事件的不变式结果. 这 5 种不变式的频率之和不为“1”的原因是: 在为某一种不变式计算频率时, 方法是计算这种不变式的耗时占用计算所有不变式的耗时的比例, 其中包含某些重复的操作. 例如: 求解 $e_i||e_j$ 和 $e_i\nparallel e_j$ 时, 都要利用在请求树中与某一事件一同发生的事件集合, 而计算该集合不能单一地算入 $e_i||e_j$ 或 $e_i\nparallel e_j$. 因此, 类似这样的计算将被多种不变式同时包含. 特别地, 用户若仅查询某一种不变式, 所耗时间应为该不变式占用比例与原耗时的乘积, 理论上速度可提升约 2–5 倍(占用比例的倒数). SOTIMiner 由于改变了求解方式, 节约了不变式的开销, 同时也使得不变式$\nparallel$在耗时比例上降低. 程序耗时较多部分的减少会为程序的效率带来更好的提升, 因此从不变式占比角度可以证明, SOTIMiner 的改进方案能够获得效率良好的表现. 由于改进方案的挖掘时间随用户的需求变化而变化, 涉及的情况种类繁多, 因此主要从两个方面检验方法的健壮性: 请求事件数量和节点数量. 如图 4(b)所示: 在提升了挖掘的针对性以后, 挖掘效率没有受到过多影响, 与事件或节点数量依次成近似的正比关系, 说明该方案能够较好地适应多种维度, 且保持了较快的速度.



(a)



(b)

图 4 不变式耗时对比及改进方案的效率

3.6 讨 论

本小节主要对方法可能存在的问题以及优化方向进行讨论.

在日志解析方面, 本文方法需要固定的输入格式, 即约定的偏序日志格式. 如果不具备这样的格式, 算法就不能有效地处理事件序列并得到时序不变式. 现实中日志的种类多种多样, 日志文件很多也很分散, 而且由于软件运行通常会调用其他组件, 还会导致不同种类的日志格式的生成. 因此, 解析出固定的输入格式是比较繁琐的过程. 而不进行日志解析则很难从不同的日志中提取出需要的信息, 也不能很好地分辨出事件类型, 这是方法存在的局限性. 但是, 常见的日志类型能够提供这种格式所需要的信息, 如事件的名称、事件开始和结束的时间、事件发生的位置即可利用算法完成偏序日志的生成, 因此也可以不失一般性. 不过, 结合有效的日志解析过程, 可以使算法的结论更有说服力.

在挖掘结果方面, 与 Synoptic 类似, SOTIMiner 认为, 凡是通过计算得出的均为正确的不变式. 而在复杂软件系统中, 一些事件的时序组合可能只是偶然发生或偶然未发生, 这也是影响挖掘结果有效性的一个重要原因. 一方面, 某些偶然发生的事件组合会被认为是系统中确定的时序关系, 但实际上这些事件间可能并不存在关联, 也就是将不具有时序相关性的两个事件认定为确定的时序不变式, 从而导致误报; 另一方面, 偶然出现的事件时序组合或者在日志数据集收集范围内未出现的时序组合, 可能会导致时序不变式的遗漏, 将具有时序相关性的两个事件认为是不存在时序不变式, 从而产生漏报.

例如: 假设某 HDFS 包含两个数据节点, 客户节点 Client 进行了 100 次文件读请求, 前 99 次正常完成, 最后一次发生异常. 图 5 给出了这 100 次请求对应的简化的请求路径. 基于该数据集, SOTIMiner 的挖掘结果存在误报. 其具体流程为: 用户节点首先选择数据节点, 随后建立连接并创建数据块, 发送数据块使节点接收. 利用 SOTIMiner 和 Synoptic 挖掘, 存在无效的不变式被误报的情况. 例如: 在最后一条路径中, 可以挖掘出不变式 datanode001.receiveBlock←datanode002.Failed, 该不变式对于每条路径都是成立的, 但这可被看作是误报, 因为路径右分支中某个事件发生异常(如 sendBlock)不是由于节点 1 的 receiveBlock 事件导致的. 而由于两个事件在同一路径中出现, 因此该不变式被错误地挖掘出来. 在真实系统中, 大量的误报会掩盖真正有意义的不变式结果. 另一方面, SOTIMiner 基于该数据集的挖掘结果存在漏报, 例如 client.choosenode→client.sendBlock 无法被方法挖掘得出. 这是因为, client 或 datanode002 中某一方断开连接, 使得发送数据事件 sendBlock 未能够发生. 而在其他大部分路径中, 该不变式属于正确的系统行为, 应将其报告给用户.

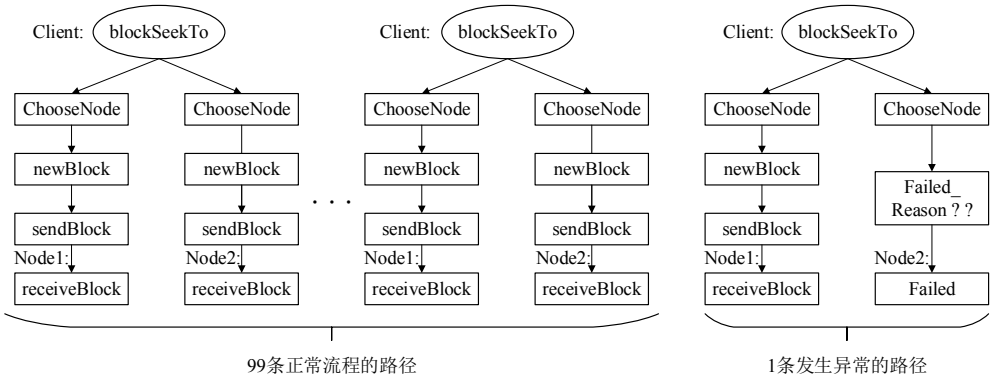


图 5 简化的包含异常的读数据块工作流程

本文给出的改进方案可在一定程度上解决误报和漏报的问题, 即采用与全序日志结合的方式, 将这些特殊不变式从全局中挑出, 作为可疑不变式单独研究. 也可以依据用户的需求提供他们需要的不变式, 使用户能够自主地利用和处理不变式. 此外, 引入概率也可以有效地解决该问题. 如果某些概率指标能够使误报不变式的概率较低, 漏报不变式的概率较高, 同时使正常报告成立的不变式的概率较高, 未报告的不应该成立的不变式的概率较低, 则可以通过设置阈值, 将误报的不变式(如 datanode001.receiveBlock←datanode002.

Failed)划分为错误的, 而将漏报的不变式(如 client.choosenode→client.sendBlock)划分为正确的, 从而减少误报和漏报. 在后续工作中, 将对这部分内容展开研究, 从而降低偶发事件组合对挖掘结果有效性的影响.

在算法的有效性和实用性方面, 本文评估算法的有效性采用的数据集较少, 原因是满足要求的大型的细粒度的开源数据集不多, 且多数数据集为人工合成的数据, 真实性较差. 本文对于得到的不变式的实用性仅作了简单的讨论, 其中, 改进方案对于在系统中注入的某些明显故障能够有一定的发现, 但将算法真正用于解决大型软件系统的故障还需要更进一步地加以研究. 本文主要评估事件的时序关系, 而不能够检测程序的数据结构等其他方面的不变式结果.

4 相关工作

本节对不变式研究相关的一些工作进行总结, 并把挖掘不变式的结果划分为程序的状态结构不变式[19−24]和时序不变式[4,8−12,14,25−30]. 多数工作是利用有向无环图[4]、有限状态机[10,26]、一阶逻辑[20,23]等形式进行直观的描述, 但实际上, 这些方法在准确率、效率、可扩展性、普适性等方面还存在各自的问题, 下面分类依次加以介绍.

程序不变式是最早从程序或日志中获取并研究的一种较为常用的不变式种类, 主要用于研究程序相关的变量、函数操作等需要满足的执行条件. Daikon[19]观察并模拟了系统的执行过程, 利用程序中函数的入口和出口点, 在多种语言找到函数操作的前置和后置条件. 许多工作因此获得启发, 在其基础上进行改进. Avenger[20]用一阶逻辑描述不变式, 首先根据用户的输入自动生成大量可能相关的潜在不变式, 然后使用已有的系统执行路径检查并剔除这些不变式中不满足实际情况的部分, 最后在真实的系统执行路径中用过滤器筛选出用户想要检查的不变式. 它可以在分布式系统中挖掘得到有助于发现不常显示的错误的不变式. Jiang 等人[21]提出了在分布式系统中挖掘与网络数据包数量、CPU 使用率等系统中流强度相关的不变式. 在相邻的位置或事件间流强度应该具有类似的曲线, 或两条曲线具有某些相关性. 该方法利用先验知识对测量得到的流强度划分相关联的集合, 在每个不可被划分的集合中寻找不变式, 最后依次检验这些不变式的有效性. Kusano 等人[22]从多线程程序动态生成可能的不变式, 并使用选择性交叉搜索的方式提高生成的不变式质量. Dinv[23]采用静态、动态混合方式检测分布式系统代码, 推断系统中不同节点的变量之间可能存在的数据属性以及分布状态. Maikel[24]划分了平面日志和分层日志两种日志格式, 利用过程树模型对日志中的函数调用等事件过程进行描述.

时序不变式是本文主要的研究对象, 同时在现阶段研究发展比较迅速, 是被多数程序员所认可、接受的不变式种类. Beschastnikh[4]设计了 Synoptic 工具, 给出了两类时序不变式的不变式挖掘方法, 在遍历有向无环图后自动地对偏序日志进行分析, 进而挖掘出两个事件间的时序不变式. Li 等人[25]根据测试人员指定的时序逻辑属性, 从 UML 状态图派生目标测试序列. Javert[26]首先在函数调用序列层面定义不变式, 利用二元决策图挖掘包含不超过 3 个函数调用的不变式, 再用有限状态机的形式将这些不变式进行表示, 并通过组合简单的、细小的不变式, 从而推导出较大的更加复杂的不变式. 这是一种不变式可扩展的技术, 但是该工具仅建立在全序日志范围内, 不能挖掘偏序日志中的不变式. Lou 等人[27]在并发系统的多条路径的混合路径中定义了事件对的前向或后向的时间依赖性, 通过一个事件相对另一个事件发生的概率计算置信度来确定事件之间是否具有依赖性, 为路径构建基本的有限状态机形式的流程模型, 然后对其进行细化以获得包含更多结构化关系的流程. Wang 等人[28]认为: 使用时序不变式能够有效地区分失败执行产生的原因, 并能够系统地表示系统执行行为. Thomas 等人[29]研究了在串行路径中统计两个事件在路径中发生的位置和顺序作为 Declare 约束, 以建立事件与活动之间的映射关系. 但对于并行发生的事件路径, 则需要改进对约束行为的表达. Xosanavongsa[30]在安全领域定义了上下文动作因果依赖关系, 利用因果图和依赖图确定异构事件之间的相关性, 但处理得到的形式结果比较单一, 并具有一定的局限性. 本文对时序不变式的研究能够具体地表述事件之间构成的相关性, 对于软件开发和测试人员正确理解系统行为有很大帮助.

5 总 结

本文提出了从集合运算角度出发的挖掘时序不变式的有效算法 SOTIMiner, 该算法可以正确、快速地求解时序不变式, 并设计了改进的方案, 验证了方案的效果、优势和局限性. 从实验结果来看: 相比先前存在的算法 Synoptic, SOTIMiner 在效率层面有较大程度的提高. 这种方法对软件系统调试和理解以及可靠性的验证等领域能够有所帮助, 可以为日益复杂的软件系统快速排除故障提供合理的依据.

References:

[1] Zhou JW. Research on key technologies of distributed system monitoring for user request path [Ph.D. Thesis]. Changsha: National University of Defense Technology, 2015 (in Chinese with English abstract).

[2] Liao XK, Li SS, Dong W, Jia ZY, Liu XD, Zhou SL. Survey on log research of large scale software system. Ruan Jian Xue Bao/ Journal of Software, 2016, 27(8): 1934−1947 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4936.htm [doi: 10.13328/j.cnki.jos.004936]

[3] Sambasivan RR, Fonseca R, Shafer I, Ganger GR. So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report, CMU-PDL-14-102, Carnegie Mellon University, 2014.

[4] Beschastnikh I, Brun Y, Ernst MD, Krishnamurthy A, Anderson TE. Mining temporal invariants from partially ordered logs. ACM SIGOPS Operating Systems Review, 2011, 45(3): 39−46.

[5] Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, dapper-2010-1, Google, 2010.

[6] Twitter Inc. Zipkin, from Twitter. 2014. http://twitter.github.io/zipkin/

[7] Fidge CJ. Timestamps in message-passing systems that preserve the partial ordering. In: Proc. of the Australian Computer Science Conf. 1988. 55−66.

[8] Ohmann T, Herzberg M, Fiss S, Halbert A, Palyart M, Beschastnikh I, Brun Y. Behavioral resource-aware model inference. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. 2014. 19−30.

[9] Le TB, Le XD, Lo D, Beschastnikh I. Synergizing specification miners through model fissions and fusions. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2015. 115−125.

[10] Beschastnikh I, Brun Y, Abrahamson J, Ernst MD, Krishnamurthy A. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. IEEE Trans. on Software Engineering, 2015, 41(4): 408−428.

[11] Lemieux C, Park D, Beschastnikh I. General LTL specification mining. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2015. 81−92.

[12] Sun P, Brown C, Beschastnikh I, Stolee KT. Mining specifications from documentation using a crowd. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. 2019. 275−286.

[13] Biermann AW, Feldman JA. On the synthesis of finitestate machines from samples of their behavior. IEEE Trans. on Computers, 1972, 100(6): 592−597.

[14] Dwyer MB, Avrunin GS, Corbett JC. Patterns in property specifications for finite-state verification. In: Proc. of the 1999 Int'l Conf. on Software Engineering. 1999. 411−420.

[15] Dou W, Bianculli D, Briand L. Model-driven trace diagnostics for pattern-based temporal specifications. In: Proc. of the 21st ACM/IEEE Int'l Conf. on Model Driven Engineering Languages and Systems. 2018. 278−288.

[16] Zhou JW, Chen ZB, Wang J, Zheng ZB, Lyu MR. A data set for user request trace-oriented monitoring and its applications. IEEE Trans. on Services Computing, 2018, 11(4): 699−712.

[17] Kholidy HA, Baiardi F. CIDD: A cloud intrusion detection dataset for cloud computing and masquerade attacks. In: Proc. of the 9th Int'l Conf. on Information Technology. 2012. 397−402.

[18] Legunsen O, Hassan WU, Xu X, Rosu G, Marinov D. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. 2016. 602−613.

[19] Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C. The Daikon system for dynamic detection of likely invariants. In: Proc. of the Science of Computer Programming. 2007. 35−45.

[20] Yabandeh M, Anand A, Canini M, Kostic D. Finding almost-invariants in distributed systems. In: Proc. of the 30th IEEE Int'l Symp. on Reliable Distributed Systems. 2011. 177−182.

[21] Jiang GF, Chen HF, Yoshihira K. Efficient and scalable algorithms for inferring likely invariants in distributed systems. IEEE Trans. on Knowledge and Data Engineering, 2007, 19(11): 1508−1523.

[22] Kusano M, Chattopadhyay A, Wang C. Dynamic generation of likely invariants for multithreaded programs. In: Proc. of the Int'l Conf. on Software Engineering. 2015. 835−846.

[23] Grant S, Cech H, Beschastnikh I. Inferring and asserting distributed system invariants. In: Proc. of the Int'l Conf. on Software Engineering. 2018. 1149−1159.

[24] Maikel L, Wil MPVDA, Mark GJVDB. Recursion aware modeling and discovery for hierarchical software event log analysis. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. 2018. 185−196.

[25] Li SH, Wang J, Qi ZC. Property-oriented test generation from UML statecharts. In: Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering. 2004. 122−131.

[26] Gabel M, Su ZD. Javert: Fully automatic mining of general temporal properties from dynamic traces. In: Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2008. 339−349.

[27] Lou JG, Fu Q, Yang SQ, Li J, Wu B. Mining program workflow from interleaved traces. In: Proc. of the 16th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. 2010. 613−622.

[28] Wang QQ, Brun Y, Orso A. Behavioral execution comparison: Are tests representative of field behavior? In: Proc. of the 10th IEEE Int'l Conf. on Software Testing, Verification and Validation. 2017. 321−332.

[29] Thomas B, Claudio DC, Jan M, Mathias W. Matching events and activities by integrating behavioral aspects and label analysis. In: Proc. of the Software & Systems Modeling. 2018. 573−598.

[30] Xosanavongsa C, Totel E, Bettan O. Discovering correlations: A formal definition of causal dependency among heterogeneous events. In: Proc. of the European Symp. on Security and Privacy. 2019. 340−355.

附中文参考文献:

[1] 周竞文. 面向用户请求路径的分布式系统监控关键技术研究[博士学位论文]. 长沙: 国防科技大学, 2015.

[2] 廖湘科, 李姗姗, 董威, 贾周阳, 刘晓东, 周书林. 大规模软件系统日志研究综述. 软件学报, 2016, 27(8): 1934−1947. http://www.jos.org.cn/1000-9825/4936.htm [doi: 10.13328/j.cnki.jos.004936]

孙德权(1996－), 男, 硕士生, 主要研究领域为系统可靠性, 形式化验证.

周竞文(1986－), 男, 博士, 副教授, 主要研究领域为软件工程, 系统可靠性, 数据分析.

周海芳(1975－), 女, 博士, 教授, CCF 专业会员, 主要研究领域为软件工程, 图形图像处理, 大数据.