计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(1):156−165 [doi: 10.15888/j.cnki.csa.008902]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 RPST 的业务过程一致性运行时检查方法①

华梦青[1,2], 龚　平[1,2], 陈志德[1,2]

[1](福建师范大学 计算机与网络空间安全学院, 福州 350007)
[2](福建省网络安全与密码技术重点实验室 (福建师范大学), 福州 350007)
通信作者: 龚　平, E-mail: tnfair@126.com

摘　要: 一致性检查是关于计算流程模型与其执行实际之间相符情况的问题. 运行时一致性检查因反馈的实时性和良好的应用前景, 成为当前一致性检查的新问题. 针对每个新产生的事件, 如何以较小的性能代价计算得到最优的一致性检查结果是运行时一致性检查的难点. 基于流程模型的结构信息 (refined process structure tree, RPST) 提出一致性监控树 (conformance monitoring tree, CMT), 基于 CMT 提出求解最优一致性结果的动态规划算法. 通过 3 个实验数据集表明, 对比已有相关工作, 本文算法具备较明显的性能优势.

关键词: 过程挖掘; 一致性检查; 运行时验证; 基于流程模型的结构信息 (RPST); 动态规划

引用格式: 华梦青,龚平,陈志德.基于 RPST 的业务过程一致性运行时检查方法.计算机系统应用,2023,32(1):156–165. http://www.c-s-a.org.cn/1003-3254/8902.html

RPST-based Conformance Checking at Runtime for Business Process

HUA Meng-Qing[1,2], GONG Ping[1,2], CHEN Zhi-De[1,2]

[1](College of Computer and Cyber Security, Fujian Normal University, Fuzhou 350007, China)
[2](Fujian Provincial Key Laboratory of Network Security and Cryptology (Fujian Normal University), Fuzhou 350007, China)

Abstract: Conformance checking refers to the alignment between a computational process model and its actual execution. Conformance checking at runtime has become a new problem in current conformance checking due to the real-time feedback and positive application prospects. For each newly generated event, how to calculate and obtain the optimal conformance checking at a low performance cost is a difficult point for conformance checking at runtime. Based on the refined process structure tree (RPST) of the process model, this study proposes a conformance monitoring tree (CMT) and a dynamic programming algorithm to obtain the optimal conformance result based on the CMT. Through three experimental datasets, it is shown that compared with the existing work, the proposed algorithm has obvious performance advantages.

Key words: process mining; conformance checking; runtime verification; refined process structure tree (RPST); dynamic programming

1　引言

随着大数据和信息技术的迅速发展, 企业越来越依赖信息管理系统进行组织实施和管理业务. 现代信息系统支持企业内部不同业务流程的执行, 可以从信息系统中提取出描述业务流程执行的事件数据. 过程挖掘[1,2]是一门较新的研究学科, 它介于数据挖掘和过程建模和分析之间. 它可以通过从当前信息系统的事件日志中提取信息来发现、监控和改进实际的流程. 过程挖掘由 3 组不同的函数组成, 包括挖掘 (discovery)、一致性检查和优化 (enhancement).

① 基金项目: 国家自然科学基金 (61841701); 福建省自然科学基金 (2020J01171, 2018J01781)
收稿时间: 2022-05-18; 修改时间: 2022-06-20; 采用时间: 2022-07-06; csa 在线出版时间: 2022-09-01
CNKI 网络首发时间: 2022-11-16

© 中国科学院软件研究所 http://www.c-s-a.org.cn

在业务流程管理系统应用中, 业务过程的实际运行受资源分配、组织协调、过程瓶颈等各个方面的影响, 可能导致业务实际执行不满足预期的模型描述[3]. 一致性检查[4] 是用以检查业务过程运行事件数据与其行为定义模型之间符合情况的方法. 早期的一致性检查方法主要基于令牌重放 (token-based replaying)[5], 但是当偏差过多时, 这种方法会导致不明确结果[4]. 为了以一种明确的方式解释和量化偏差, Adriansyah[6] 提出了对齐的方法, 通过将观察到的行为序列映射到流程模型所描述的可执行序列中, 再根据对齐结果可以查看是否存在缺失行为或不当行为.

目前, 对齐已成为一种主流的一致性检查方法. 最优对齐的查找是一个 NP-hard 问题, 复杂度较高, 为了提高对齐的效率, 部分研究者提出了基于分解技术的一致性检查方法, Polyvyanyy 等[7] 提出了简化的 RPST (refined process structure tree) 结构对业务流程进行分解. Schuster 等[8] 利用过程树的层次结构进行一致性检查, 并采用分而治之的思想计算近似的对齐结果. 除分解方法外, 还可以通过降低搜索空间的方法降低算法的复杂度. Song 等[9] 利用全局日志的统计信息, 采用启发式规则和迹重放等方法降低搜索空间. 田银花等[10] 提出了新的快速对齐方法 (rapid align), 以代价函数为约束条件, 使得搜索空间无冗余结点. 韩咚等[11] 通过计算事件日志的花型模型, 减少计算乘积模型与可达图的次数, 进而提高对齐的效率.

然而, 这些方法都属于离线一致性检查方法, 需要全局的日志信息作为输入, 且日志中的每一条迹都必须是完整的[12]. 并且只能在流程全部执行完后进行偏差分析, 并不能在流程运行的过程中发现偏差并及时调整. 因此运行时一致性检查成为当前一致性检查研究的新问题[6,13,14]. 运行时一致性检查是指将业务流程运行时所产生的数据流作为输入, 判断是否与参考模型中的行为一致[14]. 其难点在于, 针对每个新产生的事件, 如何有效缓解因计算最优对齐结果所带来的高复杂性.

Burattin[13] 提出了一种动态量化偏差行为的方法, 通过构造支持过程重放的一致性迁移系统, 可以实现在线分析事件流. 但是这种方法无法定位出偏差发生的位置. Burattin 等 [14] 还提出了根据流程活动的所有行为模式来计算运行时一致性的方法, 可以实现从中间阶段进行监控 (热启动). 但这两种方法为了性能的提升都降低了结果的准确率, 只能得到对齐的近似解.

目前主流的运行时一致性检查方法是一种计算事件流和过程模型之间最优前缀对齐的方法[6]. 部分研究在该方法的基础上提出了一定的改进, Zelst 等[15] 提出一种增量计算前缀对齐的算法, 通过复用先前计算的前缀对齐提高了求解性能. Schuster 等[16] 在 Zelst 等[15] 的基础上通过增量扩展搜索空间和重用先前计算的中间结果简化 A*算法[6]. 这些方法都可以得到前缀对齐的最优解, 虽然保证了结果的最优性, 但是会导致过高的时间复杂度和巨大的内存消耗.

针对基于 A*算法的前缀对齐方法[6,15,16] 并没有考虑到过程模型的结构性信息, 具有一定的局限性, 同时计算最优前缀对齐的复杂度过高. 为了解决这些问题, 本文提出了一种基于 RPST 的一致性监控树模型, 利用过程模型的结构信息进行偏差的快速定位, 同时, 一致性监控树可以动态保存运行时的状态, 采用动态规划的思想, 在计算前缀对齐时, 复用结构化片段的最优前缀对齐, 并采用重置机制保证结果的最优性, 在减少内存资源消耗的同时保证了对齐的最优解. 将本文的方法与已有方法[15,16] 进行比较说明, 验证了本文的算法在保证最优前缀对齐结果有效的基础上, 提高了运行效率并节省了内存空间.

本文的其余部分结构如下: 第 2 节介绍基本概念, 第 3 节介绍一致性监控树的概念. 第 4 节介绍具体的算法实现. 第 5 节对提出的算法进行实验评估, 并给出实验结果. 第 6 节提出总结与展望.

2 基本概念

下面给出一个现实世界的订单处理流程图, 如图 1 所示: 活动 A 代表收到订单并检查, 活动 C 表示处理订单, 在活动 C 之前任意次更改订单, 即活动 B. 之后, 产品发货 (D), 同时, 将发送发票 (E), 随后, 向客户发送邮件或者寄信通知. 完成前面的所有活动之后, 将订单存档 (H), 至此, 订单处理流程结束.

以该订单处理流程为动机例子, 本节主要介绍迹, 事件日志, 事件流, 过程模型, 对齐, 前缀对齐, RPST 的相关概念.

定义 1 (迹, 事件日志)[6]. 设 A 是一个所有活动集合. 若活动序列 $\sigma \in A^*$, 则称 σ 是一条迹. 若 $\exists L \in \beta(A^*)$ 是迹的一个有限非空多重集, 则称 L 为一个事件日志.

定义 2 (事件流)[15]. 设 C 是作为时刻序号的非负整数集合, A 是所有可能的活动的集合. 事件流 S 是这两个集合组成的二维序列 $C \times A$ 上的无界序列, 表示为

© 中国科学院软件研究所 http://www.c-s-a.org.cn

$S \in (C \times A)^*$.

定义 3 (过程模型)[17]. 基于 BPMN[18] 的过程模型为 $PM = (T, G, F)$, 其中: T 表示任务集合. G 表示网关, 包括 XOR 网关、OR 网关和 AND 网关. F 表示控制流关系. 主要连接两个元素对象, 并显示元素对象的执行顺序, 对象可以是 T 或 G.

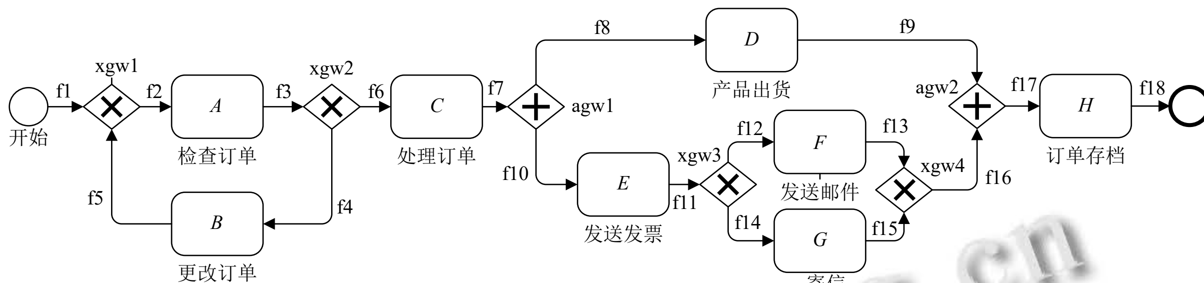

图 1 订单处理流程的流程模型

定义 4 (常规对齐)[6]. 设 A 是活动的集合, $\sigma \in A^*$是一个活动序列, $PM = (T, G, F)$是一个过程模型, 且 $>> \notin A \cup T$, σ和 N 的对齐$\gamma \in (A^{>>} \times T^{>>})$是满足如下条件的一个序列:

1) γ的第 1 行元素 (排除>>) 为活动序列, 该序列等于迹σ;

2) γ的第 2 行元素 (排除>>) 为模型中的移动序列, 可以使模型从开始任务到达结束任务.

我们将对齐记为$\gamma_M(\sigma)$, 并将合法移动$(a, t) \in \gamma_M$进行分类:

$(a, >>)$称为日志移动, 其中$a \in A$;

$(>>, t)$称为模型移动, 其中$t \in T$;

(a, t)称为同步移动, 其中$a \in A$, $t \in T$.

其中, 偏差由日志移动和模型移动表示: 日志移动表示的是在迹中所观察到的事件是模型不允许发生的, 模型移动表示的是根据模型本应该发生的事件在迹中并没有被观察到. 通常情况下, 一条迹会对应多个对齐结果, 成本函数为每个可能的移动分配一个成本, 具有最低成本的对齐即为最优对齐. 标准成本函数为每个日志移动和模型移动分配的成本为 1, 为每个同步移动分配成本为 0.

例如, 以图 1 的过程模型为例, 对迹$\sigma_1 = < A, D, F, G, H >$的最优对齐结果为$\lambda(\sigma_1)$, 如图 2 所示, 其中$(>>, C)$、$(>>, E)$是模型移动, 成本为 2, $(G, >>)$是日志移动, 成本为 1, 其他均为同步移动, 成本为 0, 所以该对齐结果的总成本为 3.

前缀对齐是在常规对齐上的改进, 以实现运行时一致性检查的对齐结果, 不需要显示从开始到结束所有事件的对齐结果, 只需要显示当前发生事件的对齐结果, 每当新的事件发生时, 对齐结果都有可能改变.

定义 5 (前缀对齐)[6]. 设 A 是活动的集合, $\sigma \in A^*$是一个活动序列, $PM = (T, G, F)$是一个过程模型, 且 $>> \notin A \cup T$, σ和PM的前缀对齐$\gamma^{pre} \in (A^{>>} \times T^{>>})^*$是一个满足如下条件的一个序列:

1) γ的第 1 行元素 (排除>>) 为活动序列, 该序列就是迹σ;

2) γ的第 2 行元素 (排除>>) 为模型中的移动序列σ_M, 该序列由 PM 中的任务 T 组成. 且存在一个移动序列σ', 使得$\sigma_M \cdot \sigma'$的发生, 可以使模型从初始任务到达结束任务.

A	>>	D	>>	F	G	H
A	C	D	E	F	>>	H

图 2 序列σ_1的对齐结果$\lambda(\sigma_1)$

例如, 迹$\sigma_2 = < A, D, F >$与图 1 的过程模型的前缀对齐的结果为$(>>, C)$, 如图 3 所示, 该对齐结果的第一行的最后一个活动不需要是模型的最后一个活动 H. 其中$(>>, C)$和$(>>, E)$是模型移动, 成本为 2, 没有日志移动, 其他均为同步移动, 所以总成本为 2.

A	>>	D	>>	F
A	C	D	E	F

图 3 序列σ_2的对齐结果$\lambda(\sigma_2)$

过程结构树 RPST 是将流程模型的图结构划分成一系列单入口单出口的模块 (SESE)[7], 图 4 将图 1 的过程模型分解为 SESE 片段, 并按照嵌套关系组织成层次结构, 这个层次结构表示为一棵树, 即 RPST, 如图 5 所示.

定义 6 ($RPST$)[7]. $RPST = (RN, r, child)$, 其中: RN 为树的结点集合; r 为树的根结点; $child$ 表示结点之间的父子关系.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

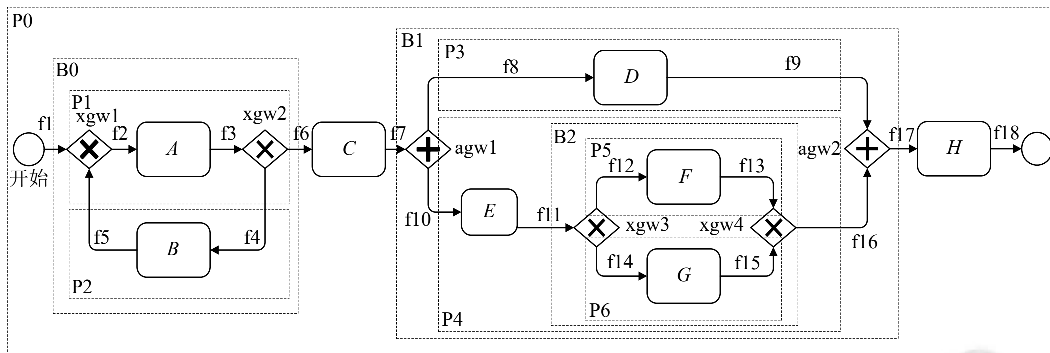

图 4　SESE 片段分解

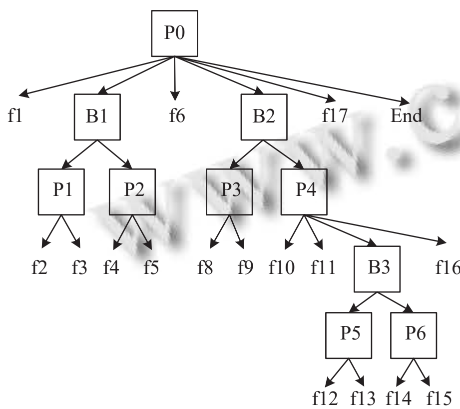

图 5　过程模型 (图 1) 对应的 RPST

给定 *PM* 为 BPMN 的流程图, *PM* 的 *RPST* 是所有规范 SESE[7] 片段的集合. 规范片段可以分为 4 种类型. 这 4 种类型分别为多边片段 (polygon fragment)、键片段 (bond fragment)、平凡片段 (trivial fragment)、刚性片段 (rigid fragment)[7], 在图 4 中, 用 P 表示多边片段, B 表示键片段, 普通叶子结点为平凡片段.

3　基于 RPST 的一致性监控树

本文提出了一种基于一致性监控树的前缀对齐算法, 利用 RPST 的结构化特点, 保存运行时的状态信息, 提高算法的性能. 首先介绍基于 RPST 的一致性监控树的概念.

因为原始的 RPST 片段中的子结点是呈无序状态的, 无法显示流程活动的执行依赖关系, 所以本文使用的是改进后的有序的过程结构树 RPST 来构造一致性监控树, 这个监控树可以体现流程模型中任务结点的执行顺序.

如图 6 所示, 基于 RPST 的一致性监控树不仅可以表示事件之间的依赖关系, 还存储了计算前缀对齐所需的相关信息.

一致性监控树具有以下几个特点:

(1) 叶子结点是任务结点.

(2) 非叶子结点是非任务结点, 并有以下 4 种类型: XOR、AND、LOOP、Polygon. 这 4 种类型分别对应过程模型中的选择结构、并行结构、循环结构、顺序结构. 其中 XOR、AND 的子结点是无序的, LOOP、Polygon 的子结点是有序的.

定义 7 (一致性监控树结点). $CMN = (RN, type, pre, post, shortest, acts)$, 其中:

RN 表示该 *CMN* 结点关联的 RPST 结点;

type 表示结点类型, 结点分为任务结点和非任务结点, 非任务结点又分为 Polygon (多边结点), XOR (选择结点)、AND (并发结点) 和 LOOP (循环结点), 其中, XOR、AND、LOOP 都属于 RPST 中的键片段;

pre 表示当前结点的前置结点, 即发生在该结点之前, 会影响该结点发生的前一个结点;

post 表示当前结点的后置结点, 即发生在该结点之后, 受该结点影响的后一个结点;

shortest 表示该结点的最短路径, 即该结点完成所需的最少的任务结点数;

acts 表示与该结点相关的活动, 即该结点片段所包含的任务结点的集合.

图 6 中显示了一致性监控树结点的所有信息, 其中, 前置结点和后置结点的判定方法是由当前结点的父结点类型决定的. 结合图 1 中的过程模型可知, 前置结点和后置结点表示的是事件之间的依赖关系. 如事件 *D* (产品出货) 依赖于事件 *C* (处理订单), 所以 *D* 的前置结点为 *C*, *C* 的后置结点为 *D*, 只有在订单处理完成以后, 产品才能出货.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

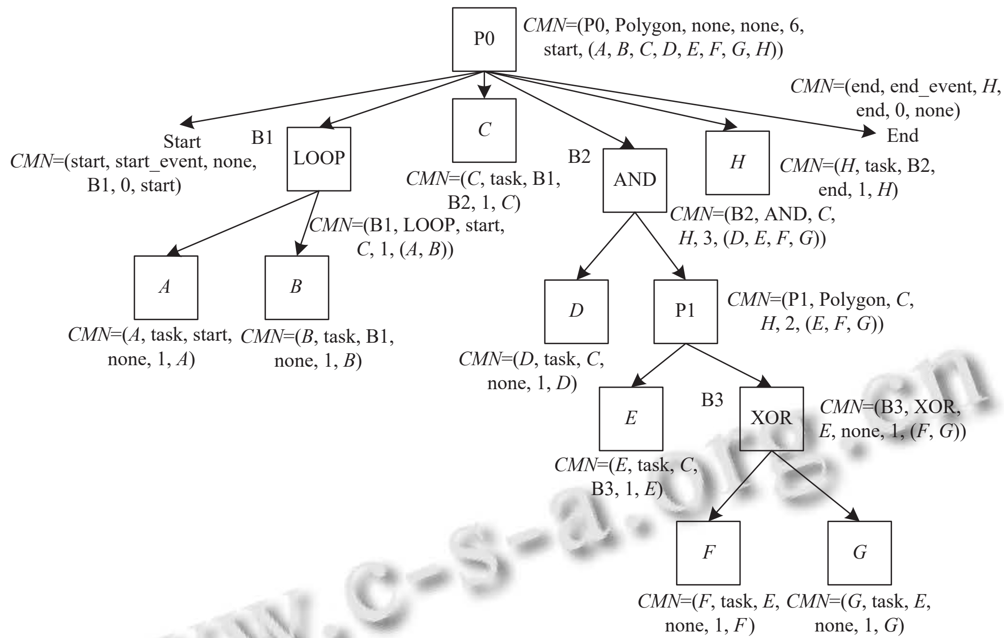

图 6 一致性监控树 (CMT)

本文提出的前缀对齐算法需要根据成本来计算最优前缀对齐, 对于任务结点, 如果在事件流中缺失, 成本为 1, 对于非任务结点, 如表 1 所示, 不同类型的结点成本的计算并不相同, *shortest* 记录了每个结点的初始成本, 即最短路径.

表 1 不同结点类型的成本计算

结点类型	成本
Task	1
XOR	所有孩子结点的成本的最小值
AND	所有孩子结点的成本之和
LOOP	循环体中所有结点的成本之和
Polygon	所有孩子结点的成本之和

一致性监控树结点 CMT 根据与其相关联的 RPST 结点 *RN* 进行转化, 接下来给出结点的转化算法, 如算法 1.

算法 1. 结点转化算法 (rn_to_cmn)

输入: RPST tree, 及结点 RN: *rn*
输出: CMT 的结点 CMN: *cmn*

```
1 type←tree.get_type(rn)
2 pre←pre_node(rn)
3 post←post_node(rn)
4 acts←get_acts(rn)
5 if type is trivial then
6   if rn is task node then
7     cmn←(rn, task, pre, post, acts)
8 else if type is bond then
9   if rn is AND node then
10    cmn←(rn, AND, pre, post, acts)
11  else if rn is XOR node then
12    cmn←(rn, XOR, pre, post, acts)
13  else if rn is LOOP node then
14    cmn←(rn, LOOP, pre, post, acts)
15  end if
16 else if type is Polygon then
17  cmn←(rn, Polygon, pre, post, acts)
18 end if
```

结点转化算法根据结点类型的不同将 RPST 中的结点转化为 CMT 中的结点. 在转化的过程中根据 RPST 的结构特点获取每个 CMT 结点的相关属性信息. 首先输入 RPST 的结点, 获取该结点的前置结点, 后置结点以及相关活动, 接下来判断该结点的类型, 如果是平凡片段, 则转化为 CMT 的任务结点 (代码第 5–7 行), 如果是键片段, 则继续判断其是否为循环结点、选择结点或并行结点 (代码第 8–14 行). 如果是多边片段, 则转化为相应的多边结点 (代码第 16–17 行).

一致性监控树 CMT 是由一致性监控树结点 *CMN* 组成的, 下面介绍一致性监控树的定义.

定义 8 (一致性监控树). $\mathrm{CMT}=(CMN,r,rt,ms)$, 其中:

CMN 表示树中一致性监控树结点 *CMN* 集合;

$r\in CMN$ 表示一致性监控树的根结点;

© 中国科学院软件研究所 http://www.c-s-a.org.cn

rt 表示一致性监控树关联的 RPST 树;

ms: $CMN \rightarrow MS$为结点集合到监控状态集合的映射, 给定结点$n \in CMN$, $ms(n) \in MS$为结点 *n* 的当前监控状态; 监控状态$ms(n) = (status, las, cost, seq, unt)$, 其中:

status 表示结点的匹配状态;

las 表示结点的最新对齐步;

cost 表示结点的成本, 记录从开始结点到当前结点的总成本;

seq 表示结点的对齐步序列, 这个对齐步序列就是最优前缀对齐;

unt 表示缺失的任务结点, 对应于传统对齐中的模型移动.

在一致性监控过程中, 为了表示出结点的匹配状态, 需要知道每个结点的决定性结点, 因此还为每一个非任务结点定义了决定性任务结点, 非任务结点的完成状态由其孩子结点的完成状态决定, 如表 2 所示.

表 2　不同结点类型的决定性结点

结点类型	决定性结点
Task	任务结点本身
XOR	任意一个孩子结点
AND	所有孩子结点
LOOP	任意一个孩子结点
Polygon	最后一个孩子结点

同时, 本文的方案还考虑了事件之间的冲突关系, 例如, 在 XOR 结点的孩子结点之间就是冲突的, 如果 XOR 结点之下的孩子结点完成了不止一个, 则只能选择其中一个孩子结点中的事件进行对齐, 并且要在结果中表明冲突关系.

4　基于一致性监控树的前缀对齐算法

本文使用一致性监控树的结构, 采用动态规划算法求事件流和过程模型的最优前缀对齐. 当事件流中有新的事件发生时, 首先计算出事件流和模型的因果对齐, 再利用结点状态动态的记录结点的对齐步、成本及当前的最优前缀对齐, 并随着事件流的更新不断更新最优前缀对齐, 并将结果记录在 CMT 的根结点中.

4.1　因果对齐

在一致性监控过程中, 每到一个事件, 都要与一致性监控树中的对应任务结点进行匹配, 并更新其他相关结点的匹配状态. 为了表示事件与结点的匹配情况, 定义了以下 4 种匹配状态, 如表 3 所示.

表 3　事件与结点的 4 种匹配状态

匹配状态	相关解释
inactivated (未触发)	该结点的*acts*均未发生
activated (触发)	非任务结点下有结点已触发, 但是决定性结点未触发
matched (匹配)	任务结点的前置结点已触发
unmatched (不匹配)	任务结点的前置结点未触发

当没有事件发生时, 结点的初始匹配状态为未触发 (inactivated), 只有该结点的 *acts* 中的活动有发生, 该结点状态才会变为触发 (activated). 只有任务结点存在匹配 (matched) 和不匹配状态 (unmatched), 当该任务结点的前置结点 *pre* 为触发状态时, 该结点状态为匹配, 否则为不匹配.

因果对齐将匹配状态存储在对齐步中, 以便显示与当前事件有依赖关系的事件是否完成.

定义 9 (对齐步 (aligned step)). 设 *A* 是活动的集合, *T* 是事件的集合. 对齐步为$als = (ord, a, t, status)$, 其中:

ord 为该对齐步的标识, 表示事件发生的次序;

$a \in A$表示迹中的活动, 即当前对齐步的活动;

$t \in T$表示与 *a* 对应的模型中的事件;

status 为该对齐步的匹配状态.

每到一个新的事件, 都将其与一致性监控树对应的任务结点进行对齐, 根据树中结点的状态得到对应的对齐步, 并更新树中该结点与其祖先结点的匹配状态.

为了在因果对齐中表示事件的依赖关系, 还需要用输入流将对齐步连接起来, 如果得到的对齐步的状态是匹配的, 那么直接在前置结点的最近对齐步和当前对齐步中建立因果流, 如果不匹配, 则需要通过一致性监控树查找未对齐片段. 并根据通过未对齐片段在当前对齐步和最近对齐步之间建立因果流.

定义 10 (未对齐片段 (unaligned fragment)). $uaf = (src, tar, shortest)$, 其中:

src 表示源结点, 即最近处于触发状态的前置结点.

tar 表示目标结点, 即当前对齐步对应任务结点.

shortest 表示从源结点到目标结点的最短路径, 即最小成本.

未对齐片段的搜索算法如算法 2.

算法 2. 未对齐片段搜索算法 (find_uaf)

输入: 当前 *CMN* 结点 *t*, 一致性监控树 CMT
输出: 未对齐片段 *uaf*

```
1 currnode←t
2 uaf.shortest←0
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
3 while true do
4   prenode←pre_node(currnode)
5   unalignedtask.add(prenode.acts)
6   uaf.tar←t
7   if prenode is activated or start node then
8     uaf.src←prenode
9     end while
10   end if
11   uaf.shortest←uaf.shortest+
     prenode.shortest
12   currnode←prenode
13 return uaf
```

算法 2 可以根据给定的 *CMN* 结点 *t*, 在 CMT 中搜索到未对齐片段 *uaf*, 目标结点 *tar* 为当前结点 *t* (代码第 6 行), 通过不断搜索前置结点的状态来确定源结点 *src* (代码第 4 行和第 12 行), 源结点为最近的处于触发状态的前置结点 (代码 7–9 行), 并在搜索的过程中统计未对齐片段的最小成本 *shortest* (代码第 11 行). 设算法中 CMT 的结点数为 *n*, 那么该算法的时间复杂度为O(*n*).

最近对齐步为源结点中的最近对齐步, 通过搜索未对齐片段 *uaf*, 可以找到当前对齐步与监控树中最近对齐步之间所有未触发的前置结点, 构建前置链条. 并计算出缺失的最少活动数, 从而可以计算成本. 同时, 在当前的对齐步与最近的对齐步之间建立因果流, 因果流记录了事件发生的顺序关系, 在后续计算最优前缀对齐的过程中, 可以通过这个关系查找到最优对齐序列.

例如, 对迹$\sigma_1 =< A, D, F, G, H >$和图 1 的过程模型, 根据一致性监控树计算出对应的因果对齐结果, 如图 7 所示.

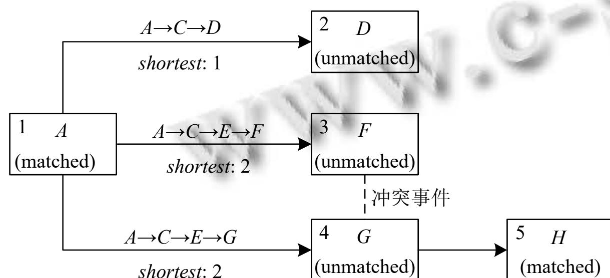

图 7 σ_1的因果对齐结果

由图 7 可知, 对齐步 2、3、4 的状态为未匹配 (unmatched), 存在未对齐片段. 事件 *F* 和 *G* 为冲突事件. 以对齐步 2 为例, 此时迹中的活动为 *D*, 在监控树中对应任务结点 *D*, 该结点的前置结点为 *C*, *C* 的状态为未触发, 则继续寻找前置结点, *C* 的前置结点为 *A*, *A* 的状态为匹配, 已触发, 则 *A* 为源结点. *A* 的最近对齐步为 1, 从 *A* 到 *D* 的前置链条为 *A*→*C*→*D*, 根据每个结点的 *shortest* 可得到 *uaf* 的最短路径为 1. 在对齐步 1 和 2 之间建立因果流, 并显示未对齐片段.

4.2 基于 CMT 的最优前缀对齐算法

基于 CMT 的最优前缀对齐算法是通过动态规划的思想, 利用结点状态记录成本和匹配序列的信息, 每到一个新的事件, 计算对齐步和对应结点的成本元组, 并更新对应结点以及所有相关结点的成本元组, 通过结点的监控状态的比较和因果流计算出最优的前缀对齐序列, 并将其存储在根结点中. 算法如算法 3 所示.

算法 3. 基于 CMT 的最优前缀对齐算法

输入: 事件流$S \in (C \times A)^*$, 一致性监控树 CMT
输出: 最优前缀对齐γ

```
1 i←1
2 while true do
3   e←S(i)
4   t←CMT.domatch(e) //t 为 CMN 结点
5   als←(i, e, t, null)
6   p←pre_node(t)
7   if p.status is finished then
8     als←(i, e, t, matched)
9     CMT.ms(t)←(matched, als,
      CMT.ms(p).cost, als, null)
10   else
11     als←(i, e, t, unmatched)
    //查找未对齐片段
12     uaf←find_uaf(t)
13     CMT.ms(t)←(unmatched, als,
      CMT.ms(uaf.src).cost+uaf.shortest,
      als, CMT.ms(uaf.src).unt+uaf.unt)
14   end if
15   update_note_state(t, CMT.ms(t))
16   γ←CMT.ms(r).seq
17   return γ
18   i←i+1
19 end while
```

算法 3 是基于 CMT 的前缀对齐算法, 事件流 *S* 和 CMT 作为算法输入, 根据事件流中的事件变化更新 CMT 的监控状态 *ms*, 每接收一个新的事件 *e*, 就在 CMT 中找到对应的 *CMN* 结点 *t* (代码 3–4 行), 根据前置结点的状态判断对齐步是否匹配, 如果前置结点已完成, 则状态为 matched, 并更新当前结点的监控状态 (代码 7–10 行). 如果前置结点未完成, 则状态为 unmatched, 并寻找未对齐片段 (代码 12–13 行), 并根据未对齐片段的源结点的状态信息更新当前结点的监控状态 (代

© 中国科学院软件研究所 http://www.c-s-a.org.cn

码第 15 行).

计算完对齐步和当前结点的监控状态之后, 还需要利用动态规划的方法更新与当前结点相关的所有结点的监控状态 (代码第 16 行), 只有当新的监控状态比原来的的监控状态更优的情况下, 才需要更新结点状态. 更新算法如算法 4 所示.

算法 4. 结点状态更新算法 (update_note_state)

输入: 当前 *CMN* 结点 *t*, *t* 的监控状态 CMT.*ms*(*t*)
输出: 一致性监控树 CMT

```
1 preseq=null
2 f=father(t)
3 currstep=t.als
4 if CMT.ms(t).seq> CMT.ms(f).seq or
(CMT.ms(f).seq= CMT.ms(t).seq and
CMT.ms(f).cost< CMT.ms(t).cost) then
5    CMT.ms(f)= CMT.ms(t)
6 end if
7 if f is Polygon then
8    preseq.add(currstep)
9    while CMT.ms(f).las!=start.las do
10       prestep=get_before_als(currstep)
11       preseq.add(prestep)
12       currstep=prestep
13     end while
14     CMT.ms(f).seq=preseq
15 end if
16 update_node_state(f, CMT.ms(f))
17 reset_node(t, CMT)
18 return CMT
```

在更新算法中, 只有当父结点类型为 Polygon 时, 才需要计算当前对齐步的前缀对齐序列 (代码 7–15 行), 父结点的对齐序列为所有孩子结点的对齐序列, 通过该算法, 最终才能将最优的对齐序列存放在根结点当中.

在更新结点的过程中, 除更新当前结点以外, 还需要遍历该结点的所有后置结点, 需要判断更新结点的后置结点需不需要重置 (代码第 17 行), 通过比较当前结点的成本与当前结点到后置结点 (已激活) 的最短路径之和是否小于后置结点原来的成本, 如果成本更小, 则将后置结点中触发的结点重置为未触发, 具体实现如算法 5 所示.

算法 5. 重置算法 (reset_node_state)

输入: 当前 *CMN* 结点 *t*, 一致性监控树 CMT
输出: 一致性监控树 CMT

```
1 cost= CMT.ms(t).cost
2 while true do
3    postnode=post_node(t)
4    if postnode is inactived then
5       cost=cost+postnode.shortest
6    else if postnode is actived then
7       if postnode.cost>cost then
8          CMT.ms(postnode)=(inactived, 0, null, null, null)
9       end if
10    end if
11 end while
12 return CMT
```

通过重置算法, 可以更新后置结点的状态信息, 使得在新的事件到来时, 可以根据新的状态计算最优的前缀对齐结果, 保证结果的最优性.

根据算法 1–5, 以图 1 的模型为例, 当迹为$\sigma_1 = <A,D,F,G,H>$时, 最优前缀对齐的结果为序列$<A,D,F,H>$, 成本为 2, 缺失任务为 *C* 和 *E*. 本文的对齐结果为最长的匹配序列, 成本为缺失的任务数. 同时, 每个结点的匹配状态显示了与该结点有依赖关系的活动是否完成, 即该结点的前置结点的完成情况. 这在传统的前缀对齐算法中是无法体现的.

5 实验评估

本节使用不同的数据集评估基于 CMT 的最优前缀对齐算法的性能. 将本文的算法与文献 [16] 中涉及的算法 (如表 4) 进行比较, 这些算法在 PM4Py 开源库[19]中均有实现, 主要对比算法的执行时间和内存消耗. 本文的算法实现可在线访问: https://gitee.com/workingspace/conformance-monitoring-all.

本文采用 3 组真实的数据集, 如表 5 所示.

表 4 算法介绍

算法	介绍
OCC	基于A*算法的在线一致性检查算法
IAS	增量A*算法
IASR	在IAS的基础上不减少启发式重新计算

表 5 数据集情况

数据集	活动	迹	事件	事件/迹
CCC 19[20]	29	20	697	35
Receipt[21]	22	1 434	8 577	6
Sepsis[22]	16	1 050	15 214	15

本文所有实验基于如下环境: Intel(R) Core(TM) i7-10710U CPU@1.10 GHz 1.61 GHz, 16 GB 内存, Windows 10 64 位操作系统, Python 3.7, JDK 11.

5.1 实验步骤

首先, 为了得到算法输入所需的过程模型, 需要通过 Inductive Miner[23] 的过程挖掘算法从数据集中挖掘

© 中国科学院软件研究所 http://www.c-s-a.org.cn

出合适的过程模型, 该过程模型用 Petri 网表示. 基于 A*算法的在线一致性检查算法直接将过程模型和事件流作为输入, 得到最优前缀对齐结果. 本文算法的过程模型基于 BPMN, 所以需要将 Petri 网转化为对应的 BPMN, 作为算法的模型输入. 再结合事件流的输入得到对应的最优前缀对齐. 统计这几组算法的执行时间和内存消耗, 并显示对比结果.

5.2 准确性评估

使用来自 CCC 19 的数据集验证本文算法结果的准确性, 利用服从度[4] 量化一致性检查的结果. 因为 OCC 的前缀对齐结果是最优解[16], 所以实验只需验证 CMT 算法与 OCC 算法得到的结果服从度是否相同, 如果相同, 则证明 CMT 算法得到的前缀对齐是最优解. 服从度为对齐中同步移动占总移动长度的比例[12]. 在 OCC 算法的对齐结果λ中, 记同步移动数为$syn(\lambda)$, 成本移动数为$dev(\lambda)$, 成本移动包括模型移动和日志移动, 服从度的公式记为:

$$C = \frac{syn(\lambda)}{syn(\lambda)+dev(\lambda)}$$

其中, 总移动长度为同步移动数和成本移动数的总和. 在本文的最优对齐结果λ^{CMT}中, 最长匹配序列对应λ中的同步移动, 迹中的序列的长度对应λ中的同步移动和日志移动. 成本对应模型移动. 记$cost(\lambda^{\mathrm{CMT}})$ 为运用 CMT 算法得到的成本 (最小代价). n 为最长匹配序列的长度, m 为迹的长度. 则本文算法的服从度公式记为:

$$C_{\mathrm{CMT}}(\sigma_L) = \frac{n}{m+cost(\lambda^{\mathrm{CMT}})}$$

因为 n 与$syn(\lambda)$均表示迹与模型相符合的事件序列, $syn(\lambda)+dev(\lambda)$和$m+cost(\lambda^{\mathrm{CMT}})$均表示迹的长度和模型移动个数的总和, 所以$C_{\mathrm{CMT}}$和$C$所表示的内涵是相同的.

根据两种算法分别计算 CCC 19 中 20 个迹的服从度, 结果如图 8 所示.

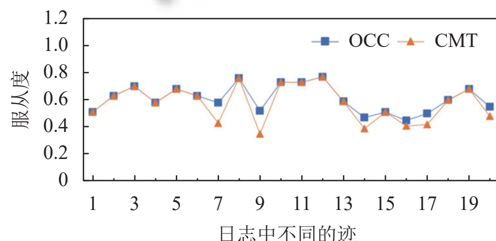

图 8　两种算法的服从度对比结果

通过对比可知, 这 20 个迹 (事件序列) 的结果中, CMT 算法有 14 个迹的服从度与 OCC 完全相同. 分析数据集可知, 造成另外 6 个迹的服从度不同的原因是迹中的事件发生了循环. 在发生循环事件时, 本文的算法只考虑一次循环中的事件, 所以服从度较 OCC 会有所降低. 实验结果证明了在不考虑循环事件的前提下, 可以计算出前缀对齐的最优解.

5.3 效率评估

针对 3 组数据集, 计算平均每个迹所需的时间, 得到如图 9 所示的结果.

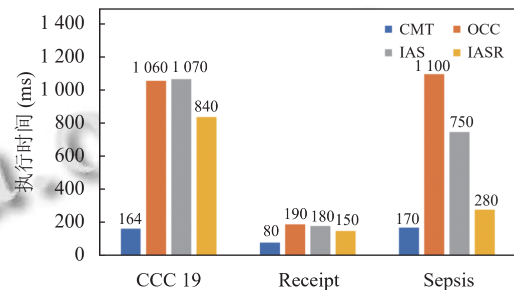

图 9　4 组算法的执行时间

图 9 的结果显示, 本文算法在时间性能上有明显的提升, 这 3 组数据集中每个迹的执行时间都有明显的减少.

同时利用这 3 组数据集对比这 4 种算法的内存消耗, 如图 10 所示.

对于数据集 CCC 19, 4 种算法的内存消耗差别不大, 但是基于 CMT 的算法的内存消耗还是比基于 A*的算法少. 而对比另外两组数据集, 基于 A*的算法的内存均远远大于基于 CMT 的算法. 尤其是数据集 Sepsis, 这组的内存消耗差距是最明显的. 从图 10 可以看出, 数据量越大, 内存消耗越大, 但是基于 CMT 的算法的内存消耗增长较为平缓. 相较于基于 A*的算法在减少内存消耗上更有优势.

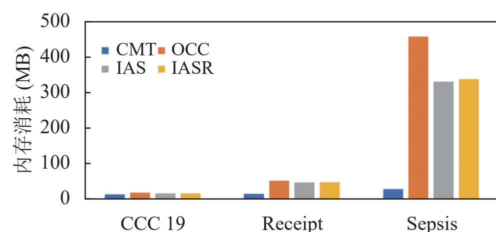

图 10　4 组算法的内存消耗

综上表明, 基于 CMT 的前缀对齐算法具有较高的性能优势. 因为运行时一致性检查无法获得完整的迹, 每到一个新事件, 最优前缀对齐的结果都有可能发生改变, 传统的基于 A*的算法的方法是使用线性规划重新计算最优前缀对齐, 这种方法会导致过高的时空复杂度. 而本文的算法利用重置机制保证结果的最优性,

© 中国科学院软件研究所　http://www.c-s-a.org.cn

如果最优前缀对齐发生改变, 只需要将一致性监控树中当前对齐步所在结点之后的所有结点状态重置为初始状态, 就可以保证之后计算的对齐结果也是最优的.

6 总结与展望

针对业务过程运行时一致性检查的复杂度过高的问题, 本文基于 RPST 提出了一致性监控树 CMT, 利用过程模型的结构性信息进行偏差的快速定位, 并利用基于 CMT 的动态规划算法求解事件流与过程模型的最优一致性检查结果. 最后, 将本文方法与其它在线一致性检查方法进行性能比较分析, 比较了它们的准确率和效率. 结果表明, 本文提出的方法在保证了有效性的同时提升了性能. 但是, 如果过程模型中存在循环结构, 本文的方法只考虑一次成本最小的局部对齐, 因此, 在未来的工作中, 将考虑当过程模型中存在循环结构时, 计算出循环多次执行时的最优前缀对齐结果.

参考文献

1 van der Aalst W, Weijters T, Maruster L. Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering, 2004, 16(9): 1128–1142. [doi: 10.1109/TKDE.2004.47]

2 闻立杰. 基于工作流网的过程挖掘算法研究 [博士学位论文]. 北京: 清华大学, 2007.

3 Pichon F, Destercke S, Burger T. A consistency-specificity trade-off to select source behavior in information fusion. IEEE Transactions on Cybernetics, 2015, 45(4): 598–609. [doi: 10.1109/TCYB.2014.2331800]

4 van der Aalst W. Process Mining: Data Science in Action. Berlin: Springer, 2016.

5 Rozinat A, van der Aalst WMP. Conformance checking of processes based on monitoring real behavior. Information Systems, 2008, 33(1): 64–95. [doi: 10.1016/j.is.2007.07.001]

6 Adriansyah A. Aligning observed and modeled behavior [Ph.D. Thesis]. Eindhoven: Technische Universiteit Eindhoven, 2014.

7 Polyvyanyy A, Vanhatalo J, Völzer H. Simplified computation and generalization of the refined process structure tree. Proceedings of the 7th International Workshop on Web Services and Formal Methods. Hoboken: Springer, 2010. 25–41.

8 Schuster D, van Zelst S, van der Aalst WMP. Alignment approximation for process trees. Proceedings of ICPM 2020 International Workshops on Process Mining Workshops. Padua: Springer, 2020. 247–259.

9 Song W, Xia XX, Jacobsen HA, *et al*. Efficient alignment between event logs and process models. IEEE Transactions on Services Computing, 2017, 10(1): 136–149. [doi: 10.1109/TSC.2016.2601094]

10 田银花, 杜玉越, 韩咚, 等. 基于 Petri 网的事件日志与过程模型对齐方法. 计算机集成制造系统, 2019, 25(4): 809–829. [doi: 10.13196/j.cims.2019.04.003]

11 韩咚, 田银花, 杜玉越, 等. 基于 Petri 网可达图的业务对齐方法. 计算机集成制造系统, 2020, 26(6): 1589–1606. [doi: 10.13196/j.cims.2020.06.016]

12 马婷婷. 基于对齐的业务流程在线一致性检测 [硕士学位论文]. 淮南: 安徽理工大学, 2021.

13 Burattin A. Online conformance checking for Petri nets and event streams. Proceedings of the 15th International Conference on Business Process Management. Barcelona: BPM, 2017. 1–6.

14 Burattin A, van Zelst SJ, Armas-Cervantes A, *et al*. Online conformance checking using behavioural patterns. Proceedings of the 16th International Conference on Business Process Management. Sydney: Springer, 2018. 250–267.

15 van Zelst SJ, Bolt A, Hassani M, *et al*. Online conformance checking: Relating event streams to process models using prefix-alignments. International Journal of Data Science and Analytics, 2019, 8(3): 269–284. [doi: 10.1007/s41060-017-0078-6]

16 Schuster D, van Zelst SJ. Online process monitoring using incremental state-space expansion: An exact algorithm. Proceedings of the 18th International Conference on Business Process Management. Seville: Springer, 2020. 147–164.

17 van der Aa H, Leopold H, Reijers HA. Efficient process conformance checking on the basis of uncertain event-to-activity mappings. IEEE Transactions on Knowledge and Data Engineering, 2020, 32(5): 927–940. [doi: 10.1109/TKDE.2019.2897557]

18 Al-Ali H, Damiani E, Al-Qutayri M, *et al*. Translating BPMN to business rules. Proceedings of the 6th IFIP WG 2.6 International Symposium on Data-driven Process Discovery and Analysis. Graz: Springer, 2016. 22–36.

19 Berti A, van Zelst SJ, van der Aalst W. Process mining for python (PM4Py): Bridging the gap between process and data science. arXiv:1905.06169, 2019.

20 Munoz-Gama J, de la Fuente R, Sepúlveda M, *et al*. Conformance Checking Challenge 2019. https://doi.org/10.4121/uuid:c923af09-ce93-44c3-ace0-c5508cf103ad. (2019-02-14).

21 Buijs J. Receipt phase of an environmental permit application process (‘WABO’). https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6. (2014-08-04).

22 Mannhardt F. Sepsis cases-event log. https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460. (2016-12-07).

23 Ghawi R. Process discovery using inductive miner and decomposition. arXiv:1610.07989, 2016.

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn