Received: 29 August 2019 | Revised: 29 October 2019 | Accepted: 20 December 2019

DOI: 10.1111/exsy.12529

SPECIAL ISSUE PAPER

Expert Systems WILEY

A novel completeness definition of event logs and corresponding generation algorithm

Chuanyi Li[1,2] | Jidong Ge[1,2] | Lijie Wen[3] | Li Kong[1,2] | Victor Chang[4] | Liguo Huang[5] | Bin Luo[1,2]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

[2]Department of Software engineering, Software Institute, Nanjing University, Nanjing, China

[3]Department of Software Engineering, School of Software, Tsinghua University, Beijing, China

[4]School of Computing and Digital Technologies, Teesside University, Middlesbrough, UK

[5]Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas

Correspondence
Jidong Ge, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China.
Email: gjd@nju.edu.cn

Present address
Jidong Ge, Room 924, Fei Yimin Building, No.22 Hankou Road, Gulou District, Nanjing, Jiangsu, China

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 61802167, 61572251; Open Foundation of State key Laboratory of Networking and Switching Technology (Beijing University of Posts and Telecommunications), Grant/Award Number: SKLNST-2019-2-15

Abstract

As the promotion of technologies and applications of Big Data, the research of business process management (BPM) has gradually deepened to consider the impacts and challenges of big business data on existing BPM technologies. Recently, parallel business process mining (e.g. discovering business models from business visual data, integrating runtime business data with interactive business process monitoring visualisation systems and summarising and visualising historical business data for further analysis, etc.) and multi-perspective business data analytics (e.g. pattern detecting, decision-making and process behaviour predicting, etc.) have been intensively studied considering the steep increase in business data size and type. However, comprehensive and in-depth testing is needed to ensure their quality. Testing based solely on existing business processes and their system logs is far from sufficient. Large-scale randomly generated models and corresponding complete logs should be used in testing. To test parallel algorithms for discovering process models, different log completeness and generation algorithms were proposed. However, they suffer from either state space explosion or non-full-covering task dependencies problem. Besides, most existing generation algorithms rely on random executing strategy, which leads to low and unstable efficiency. In this paper, we propose a novel log completeness type, that is, #TAR completeness, as well as its generation algorithm. The experimental results based on a series of randomly generated process models show that the #TAR complete logs outperform the state-of-the-art ones with lower capacity, fuller dependencies covering and higher generating efficiency.

KEYWORDS
big data analytics, business process management, event log completeness, log generation, testing

1 | INTRODUCTION

In recent years, with the rapid development of computer and network technologies, many advanced information technologies have been developed in depth, such as Internet of Things (IoT), Cloud Computing, Data Mining and Machine Learning. The continued maturity of these technologies has led to the development of another information technology, that is, Big Data. IoT technology can collect various data by making different

Abbreviations: ANA, anti-nuclear antibodies; APC, antigen-presenting cells; IRF, interferon regulatory factor.

types of sensors installed on all kinds of devices, which makes it possible to produce big data. The cloud computing platform provides sufficient storage space for big data and sufficient computing power for mining valuable information from big data. Meanwhile, data mining and machine learning technologies provide a wealth of theoretical and concrete algorithms for making full use of big data to mine and compute valuable information. Conversely, Big Data raises the level of requirements on these techniques from many perspectives, such as dealing with privacy issues, controlling information security, ensuring data processing efficiency and providing capability of jointly processing different types of data. All of these constitute the Big Data Analytics (BDA; Russom, 2011) technologies, which refers to the process of data collection, transfer in centralised cloud data centres, pre-processing, analysis and visualisation (Ahmed et al., 2017).

The most straightforward way to transform the IT value of BDA into business value is to apply it to business process management (BPM) applications (Wang, Kung, & Byrd, 2018). This will also fully realise the business value of IoT, Cloud Computing, Data Mining and so on. For example, banks, healthcare organisations and e-commerce platforms are applying BDA for improving business processes and workforce effectiveness, supporting evidence-based decision-making and action-taking, reducing enterprise costs and attracting new customers, respectively (Wang et al., 2018). Figure 1 briefly shows the relationship between related technologies and applications using BDA in BPM. The goals of business data visualisation (De Alvarenga, Barbon Jr, Miani, Cukier, & Zarpelão, 2018; Schwank, Schöffel, & Ebert, 2018) include building concise, accurate and rich visualisations of big historical business records from control flow, data and resource perspectives, as well as providing efficient interfaces for monitoring runtime performance of the business process. Further concrete applications of mining and analysing contain behaviour patterns discovery, business element (action, time and resources, etc.) predicting, decision-making and so on. The mining and analysis results can be either simply visualised independently or integrated with other data visualisation approaches.

However, applying BDA techniques including visualisation and mining in business processes is non-trivial. The special business domain knowledge determines that the problem of big data in business processes cannot be solved by simply using general big data processing technology (Zhu, Ge, Song, & Gao, 2018). Explicit or implicit features of business data should be well considered in designing big business data analytics based on general big data techniques. To ensure the correctness, robustness and efficiency of the designed approaches, full testing work should be taken based on data covering business features as many as possible. Then, testing based solely on existing business processes and their system logs is far from sufficient. No one knows the designed method would be applied to what kind of business process and what kind of its corresponding business log in the future. So, large-scale randomly generated models and corresponding complete logs should be used in testing.

As shown in Figure 1, besides testing, the generated complete logs could be used as theoretical references in runtime business behaviour measuring or evaluation. The generated log represents the design intent while the true log is representing the online behaviour. By comparing the two types of logs, the gap between the reality and the ideal, or shortcomings of the management, would be detected. It is common that the process model of the continuously improved online business process system turns to be out of date. In order to find the most similar process models in the repository, you could check the similarity between its runtime log and one existing model's generated log (De Medeiros, Aalst, & Weijters, 2008; Dong, Wen, Huang, & Wang, 2014; Gerke, Cardoso, & Claus, 2009; Wang et al., 2010).

Business process event logs (Aalst, Weijters, & Maruster, 2004) generation algorithm is widely discussed for designing and testing process model discovery (Carmona, 2012; Eck, Buijs, & Dongen, 2014) and similarity measuring (Dijkman, Dumas, Van Dongen, Käärik, & Mendling, 2011; La Rosa, Dumas, Uba, & Dijkman, 2010; Weidlich, Mendling, & Weske, 2010; Zha, Wang, Wen, Wang, & Sun, 2010) approaches. Figure 1 shows that the input of log generating are process models and the outputs are corresponding complete event logs. For designing generating algorithm, the complete type of event logs should be pre-defined. There are mainly two kinds of completeness: global and local (Hee, Liu, & Sidorova, 2011). Global complete event logs contain all possible execution traces of process models while local complete ones only contain all direct succession (Yang, Wen, & Wang, 2012) relations or transition adjacent relations (TARs; Zha et al., 2010) between tasks. However, the globally complete event logs have larger capacities than the locally complete ones, and it costs much more generating time. But locally complete logs cannot represent the behaviour of process as accurate as the globally complete ones, especially for implicit dependencies (Wen, Van der Aalst, Wang, & Sun, 2007). Although the state-of-the-art TAR* completeness (Wang & Wang, 2012) compensates the weakness of local completeness, its representation is

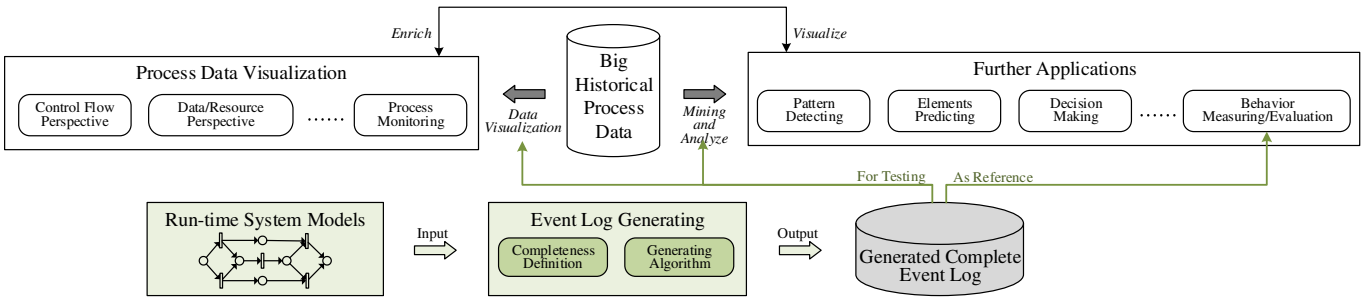


FIGURE 1 Applications of BDA in business process management and the functionality of log generating

not compatible with the content of the event logs. Besides, the generation algorithms for local and TAR* complete logs rely too much on random executing algorithms of process models, which results in inefficient and unstable log generation.

In this paper, we address the aforementioned problems of existing complete event logs and their generation algorithms by defining a novel type of event log completeness and its corresponding generating algorithm. The new completeness definition mainly considers three aspects, namely, moderate log size, complete behavioural expression and efficient generation algorithm. It is defined based on TAR complete and we name it #TAR completeness. To conclude, we made the following contributions in this paper:

1. Define #TAR completeness of event logs formally, as well as fully compare it with both global and local completeness of event logs from three aspects, namely, log capacity, accuracy in expressing model behaviour and generating algorithm.
2. Propose the algorithm for generating #TAR complete event logs according to given process models. The generating algorithm is based on the task set (TS)-invariant theory of Petri-nets. To the best of our knowledge, it is the first study of applying TS-invariant theory to generate complete event logs of process models.
3. Enhance the process mining algorithms for handling #TAR complete event logs by checking task sets consistence between input log and derived model.

The rest of this paper is organised as follows. Section 2 introduces the preliminaries and motivating example of this work. Formal definition of #TAR completeness is presented in Section 3. In Section 4, the three steps for generating the logs of #TAR completeness are described. Enhancement of exiting process mining algorithms for handling #TAR complete logs is illustrated in Section 5. Section 6 shows experimental results for evaluation. Related work is briefly introduced in Section 7. Section 8 concludes the paper and provides the remarks and directions of future works.

2 | PRELIMINARIES AND MOTIVATIONS

2.1 | Workflow nets

Workflow net (WF-net; Van Der Aalst, 1998), a subset of Petri-net (Desel & Esparza, 2005), is used as the modelling tool in this paper. Petri-net, which is composed of transitions, places and arcs between them, is specialised in modelling concurrency systems. A Petri-net is a triplet $PN = (P, T, F)$ where P, T and F are finite sets of places, transitions and the directed arcs between the places and transitions. All the pre-elements of an element $x \in P \cup T$ are in the set $\bullet x = \{y \in P \cup T | (y, x) \in F\}$ and the post-elements of x are in $x\bullet = \{y \in P \cup T | (x, y) \in F\}$. The element sequence $x_1, x_2, \ldots, x_n$ is called a path from x_1 to x_n while these elements satisfy $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n) \in F$. The snapshot of the distribution of tokens (represented by black dots) in the places is called a marking at one point of time and the marking is denoted by M. If the initial marking of the net PN is M_0, then all the possible markings of the net could be denoted by $M = [PN, M_0\rangle$. The token number in PN under marking M is denoted by $M(p)$. Transition t is enabled under marking M *iff* $\forall p \in \bullet t$: $M(p) \geq 1$. If a transition is enabled, it could change its state and this process is called fire. After t firing the marking of PN turns to M' and this transform is marked as $M \xrightarrow{t} M'$. The set of all enabled transitions under marking M is denoted by ***enabled***(M).

A WF-net requires the Petri-net to have (a) a single Start place, (b) a single End place and (c) each node must be on one path from Start to End. The soundness property further enforces that (d) there is no dead task, and (e) the process with only one token in the Start place can always terminate with only one token in the End place. The initial and final states of a WF-net with input place **i** and output place **o** are denoted by [**i**] and [**o**]. The methods proposed in this paper are only applicable for process models that can be represented by sound WF-nets. Figure 2 is a sample sound WF-net.

As shown in Figure 2, the transitions and places are represented by rectangles and circles, respectively. Places with multi outputs are the beginning of xor-splits (i.e., p_1, p_7 in Figure 2) and transitions with multi outputs are the beginning of and-splits (i.e., d and j). Places and transitions having multi-inputs are the endings of xor-joins and and-joins, respectively, such as p_2, p_5 and q, n. Sub-pathes between xor-splits and their corresponding xor-joins are choices. Sub-pathes between and-splits and their corresponding and-joins are parallelisms. Choice can form loop structures. For example, p_7, g and p_5 is a one-step loop in Figure 2. By decomposing the choices or parallelisms with the TS-invariant theory of

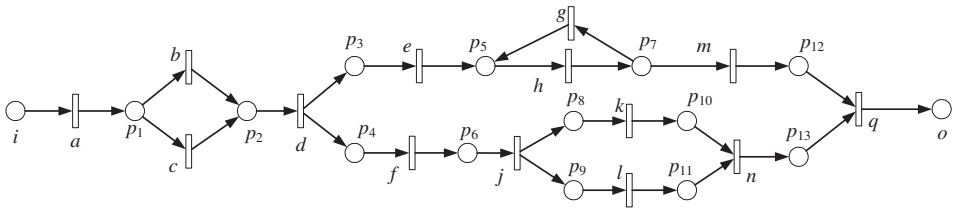


FIGURE 2 This is a sample of workflow-net

Petri-nets, we will get two simple sub-nets of the WF-nets: T-workflows and S-workflows (Hee et al., 2011). There are no choices in T-workflows and no parallelisms in S-workflows. After decomposing all choices with T-invariant of the model in Figure 2, three sub-T-workflows are derived as shown in Figure 3. Sub-T-workflow without loops is Main Path (**MP**) and the one with loops is Loop Sub-Model (**LSM**).

Decomposing the parallelisms in the sub-T-workflows in Figure 3, we will get the sub-S-workflows as shown in Figure 4.

2.2 | Event log and completeness

Event logs are composed by event entries, and each entry is connected to a task and a running case (i.e., a running instance of the business process). Let *EL* be an Event Log of a sound WF-net *PN*, then Event $e = (cid, t)$ is an element of *EL*, *cid* is the identification of the case containing the event and *t* is the execution task of the event. There are two functions on Event *e*: (a) $\boldsymbol{caseID}(e) = cid$, (b) $\boldsymbol{task}(e) = t$.

A trace is a combination of events that belong to the same running instance. Let $PN = (P, T, F)$ is a sound WF-net, and *EL* is the event log of *PN*, then $\sigma = (t_1, t_2, \ldots, t_n)$ is an execution trace of *PN iff*: (a) $\forall i \in [1, n]$, $t_i \in T$, and (b) $\exists cid \in N \Rightarrow \forall i \in [1, n]$, $\exists e_i \in EL$ where $\boldsymbol{caseID}(e_i) = cid \wedge \boldsymbol{task}(e_i) = t_i$. Function $\boldsymbol{len}(\sigma)$ returns the number of events the trace σ has.

Completeness (Aalst et al., 2004) is a very important property of event logs. The extent of completeness of event logs is determined by the percentage of behaviour of the corresponding processes covered by the logs. There are mainly two levels of completeness: global and local completeness. Global completeness requires the log containing all possible execution traces of the process. Local completeness requires the log contain all **TAR** between tasks. The TAR between task *a* and *b* is represented by $a > b$. Let $PN = (P, T, F)$ be a sound WF-net, and $a, b \in T$, then $a > b$ *iff* there is an execution trace $\sigma = t_1, t_2, \ldots, t_{n-1}$ in the globally complete event log of *PN* and $\exists i \in [1, n-2]$ such that $t_i = a$ and $t_{i+1} = b$.

The sets of TARs determined by a given event log *EL* and execution trace σ are denoted as $>L$ and $>\sigma$, respectively. Local completeness is TAR completeness. Let *EL* be an event log of a sound WF-net $PN = (P, T, F)$, then *EL* is locally complete *iff*: (a) for any execution trace σ in the globally complete event log of *PN*: $> \sigma \subset > EL$, and 2) $\forall t \in T \Rightarrow \exists \sigma \in EL$ such that $t \in \sigma$.

But locally complete event logs cannot cover behaviour of the implicit dependencies (Wen et al., 2007) in the process model. Let $PN = (P, T, F)$ be a sound WF-net with input and output places **i** and **o**, then $\forall a, b \in T$, there is an implicit dependency between *a* and *b iff*: (a) $a\bullet \cup \bullet b \neq \emptyset$, (b) $\neg\exists M \in [PN,[\mathbf{i}]\rangle \Rightarrow a \in \boldsymbol{enabled}(M) \wedge b \in \boldsymbol{enabled}(M - \bullet a + a\bullet)$, (c) $\exists M \in [PN,[\mathbf{i}]\rangle \Rightarrow a \in \boldsymbol{enabled}(M)$, and $\exists M' \in [PN, M - \bullet a + a\bullet\rangle \Rightarrow b \in \boldsymbol{enabled}(M')$.

In order to express implicit dependency in the locally complete event log, TAR* completeness is defined (Wang and Wang (2012)) by appending statements of the implicit dependencies instead of appending additional event entries.

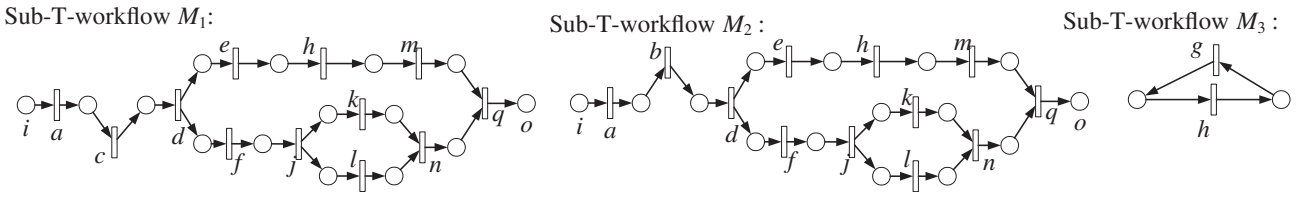


FIGURE 3 Sub-T-workflows of the model in Figure 2

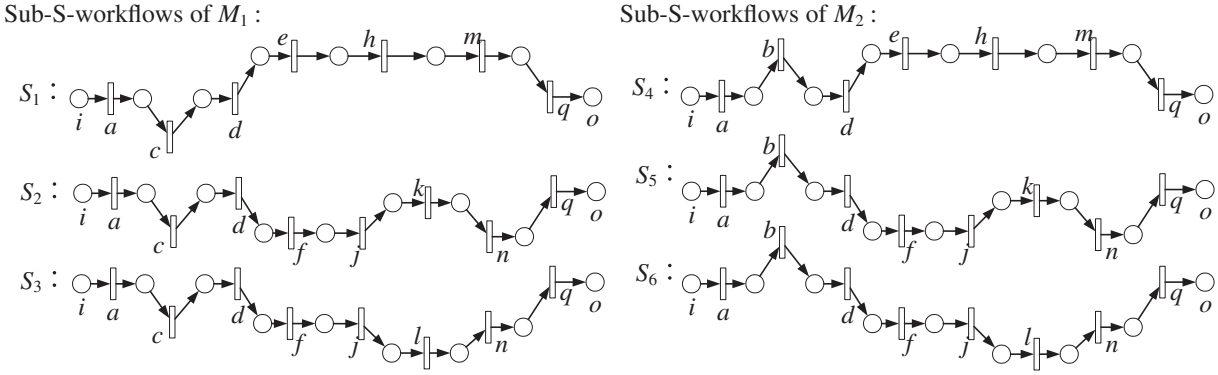


FIGURE 4 Sub-S-workflows of M_1 and M_2 in Figure 3

2.3 | Complete log generation algorithms

One certificate of the rationality of the completeness of event logs is if there is a reasonable and efficient algorithm to generate that kind of complete event logs according to a process model. Furthermore, only if the defined complete event logs of any process model can be generated by the algorithm, the generating algorithm is reasonable. To find the reasonable generating algorithm can help both understanding the definition and the usage of the proposed completeness and bringing suggestions for checking whether a given log satisfies the conditions of the completeness without former process models.

Three ways have been proposed to generate complete event logs based on process models. (a) For global Completeness: construct a coverability tree of the process model and to record the complete firing sequence of the model by traversing the coverability tree (Dong et al., 2014). The coverability tree can be traversed in two ways: Breadth-First and Depth-First. But both the time and space complexities of the traversal methods are too high to be accepted by normal computing machines. (b) For local completeness: derive all TARs first and then randomly execute the process and record running traces. One concrete way to calculate TARs is to decompose the process model by virtualising the part between an and/xor-split and an and/xor-join as a sub-model (Zha, Wang, & Wen, 2007). Then, the sub-models would only contain sequential dependencies between tasks. (c) For TAR* completeness: decompose the model according to unfolding theory of Petri-nets (Wang, Wen, & Tan, 2011) to derive all TARs and implicit dependencies. Then, rely on randomly executing strategy to collect trace records and checking if all TARs are covered. Upon the TARs are covered, statements of implicit dependencies are attached.

In summary, existing complete log generating algorithms suffer from either state space explosion or non-full-covering task dependencies problem. These problems lead to the low and unstable efficiency of generating algorithms. Root causes of these problems are definition of log completeness and randomly executing strategy. The proposed #TAR completeness and corresponding generating algorithm would solve these problems. Next, the motivations of proposing #TAR and the novel generating algorithm will be discussed.

2.4 | Motivation analysis

Table 1 summarises the characteristics, as well as limitations of definitions of different log completeness and corresponding generating algorithms. The proposition of #TAR completeness is motivated by overcoming limitations of existing ones. The last column shows the ideal properties of well-defined #TAR completeness from five perspectives.

Complete aspect and log size: The best coverability of existing definitions is Global Completeness, whose logs contain all possible traces that could be produced by the processes. However, the log size of trace complete log could be very large, especially for parallelisms and loops. For process model discovery algorithms, covering TAR and Implicit Dependencies is enough. The reduction of coverability also reduces the log capacity. So, the ideal #TAR completeness should at least cover all TARs and Implicit Dependencies in the process. The ideal minimal log size should be far smaller than the Global Completeness ones and the same as that of TAR* completeness would be the best.

Log content: Appending statements to event entries is not good for designing general process discovery algorithms. Besides, it is impossible to record statements about implicit dependencies while the business system is running. The ideal #TAR completeness logs should contain only event entries.

Decomposing strategy: For generating complete traces, no matter what kind of strategy is taken, the algorithm complexity would be high since the result log size is large. For the other two complete types, the strategy should detect the relations between any two tasks. However, all strategies applied in the existing approaches only transform the process model into either another entire data structure or fine-grained model pieces. Complex operations should be taken to generate logs from these data structures. The ideal strategy for generating #TAR complete logs should perceive both overall and details of the input process model. In this paper, TS-invariant theory of Petri-nets is adopted.

Generating strategy and efficiency: In order to avoid complex operations for combining TARs, existing algorithms for generating TAR or TAR* complete logs are highly relying on randomly executing strategy. Upon a trace is produced, the completeness of existing log would be checked.

TABLE 1 Properties of existing completeness definition and the ideal #TAR completeness

	Global	Local (TAR)	TAR*	Ideal of #TAR
Complete aspect	Trace	TAR	TAR+implicit dependency	Cover TAR+implicit dependency
Log size	Large	Small	Small	Small, large
Log content	Events	Events	Events with statements	Only events
Decomposing	Coverability tree	Region	Unfolding	TS-invariant
Generating	Tree traversal	Randomly simulating	Randomly simulating	Partial constructing+inserting
Efficiency	Low	Unstable	Unstable	High and stable

No one could make sure when would the generating algorithm stop. So, the efficiencies of the algorithms are not stable. The ideal generating efficiency of #TAR completeness logs should be High and Stable.

3 | #TAR COMPLETENESS

According to the motivation analysis, the key property of #TAR completeness should be covering implicit dependency. As mentioned in Section 2.2, removing the implicit dependencies in the model would have no impacts on its locally complete event logs, that is, TAR coverability. So, another perspective for representing implicit dependency should be established.

Task Set (**TS**) represents the collection of tasks that an execution trace contains. Let ts be a task set and $\sigma = (t_1, t_2, ..., t_n)$ be an execution trace of a case, then $ts =$ ***taskSet***(σ) *iff*: (a) $\forall t \in ts \Rightarrow \exists i \in [1, n]$ satisfies $t_i = t$, (b) $\forall i \in [1, n] \Rightarrow \exists t \in ts$ satisfies $t = t_i$.

If the function ***taskSet*** takes a T-workflow Tw or an event log EL as input, it returns all the task sets of the model or log. For the event logs of the model in Figure 2, there are at most four kinds of task sets and they are $\{a, c, d, e, f, g, j, k, l, m, n, q\}$, $\{a, c, d, e, f, g, j, k, l, m, n, q, h\}$, $\{a, b, d, e, f, g, j, k, l, m, n, q\}$, and $\{a, b, d, e, f, g, j, k, l, m, n, q, h\}$.

We find that the existence of implicit dependency in the model would influence the task sets of its corresponding complete event logs. The model without implicit dependency would have more task sets in the complete log than adding an implicit dependency to the model. So, we define the completeness of task sets as a part of the #TAR completeness. In this way, the one-to-one mapping between models and complete logs could be ensured.

in #TAR represents the pre-condition for TAR completeness, that is, the log should contain all task sets of T-workflows the initial process model has. For event traces of each task sets, TARs of corresponding T-workflow should be covered. That is to say, #TAR completeness firstly requires task set completeness and then TAR completeness. Implicit dependencies could only exist in either the main path or the loop sub-models, which means it could never bridge the main path and the loop. So, to represent implicit dependency, only complete task sets of both MPs and LSMs are necessary. To be specific, the pre-condition is that event logs should contain enough possible task sets of the process model to derive **TS**s of all single **MP**s and **LTS**s (i.e., loop task set) of all single **LSM**s. If TARs of all single MPs and single LSMs are covered, then TARs of the model must be covered. Considering these, here is the formal definition of #TAR completeness.

Definition 1 (#TAR Complete). Let $PN = (P, T, F)$ be a sound WF-net, EL be the event log of PN, $Tws = \{T_{M1}, T_{M2}, ..., T_{Mn}, T_{L1}, T_{L2}, ..., T_{Lm}\}$ be the set of sub-T-workflows of PN, where T_{Mi} represents a Main Path and T_{Lj} stands for a Loop Sub-Model, then EL is a #TAR Complete log of PN *iff*:

1. $\{$***taskSet***$(T_{Mi})|\forall T_{Mi} \in Tws, i \in [1, n]\} \subset$ ***taskSet***(EL),
2. $\forall T_{Lj} \in Tws, j \in [1, m], \exists TS \in$ ***taskSet***$(EL) \Rightarrow$ ***taskSet***$(EL) \subset TS$,
3. $\forall TAR \in > PN \Rightarrow TAR \in > EL$.

Example 1 For the model in 2, as said in Section 2.2, there are at most four task sets of its event logs and they are $TS_1 = \{a, c, d, e, f, h, j, k, l, m, n, q\}$, $TS_2 = \{a, c, d, e, f, h, j, k, l, m, n, q, g\}$, $TS_3 = \{a, b, d, e, f, h, j, k, l, m, n, q\}$, and $TS_4 = \{a, b, d, e, f, h, j, k, l, m, n, q, g\}$. However, if the log contains any three of the four task sets, the other one could be computed out. For example, $TS_1 = TS_2 - (TS_4 - TS_3)$, $TS_2 = TS_1 + (TS_4 - TS_3)$, $TS_3 = TS_4 - (TS_2 - TS_1)$, and $TS_4 = TS_3 + (TS_2 - TS_1)$. There are three sub-T-workflows of the model and their task sets or loop task sets are $TS_{Tw1} = \{a, c, d, e, f, h, j, k, l, m, n, q\}$, $TS_{Tw2} = \{a, b, d, e, f, h, j, k, l, m, n, q\}$ and $LTS_{Tw3} = \{g, h\}$. It is obvious that $TS_{Tw1} = TS_1$ and $TS_{Tw2} = TS_3$. LTS_{Tw3} can be got from $TS_2 - TS_1$ or $TS_4 - TS_3$. So, for event logs of the model in 2, if they contain any three of the pre-mentioned four task sets, they satisfy the first requirement of #TAR complete.

Example 2 Let EL be an event log of the model in Figure 2 and it satisfies the first condition of #TAR complete with three task sets: $TS_1 = \{a, c, d, e, f, h, j, k, l, m, n, q\}$, $TS_2 = \{a, c, d, e, f, h, j, k, l, m, n, q, g\}$, and $TS_3 = \{a, b, d, e, f, h, j, k, l, m, n, q\}$. The three sub-T-workflows of the model and their task sets are $TS_{Tw1} = \{a, c, d, e, f, h, j, k, l, m, n, q\}$, $TS_{Tw2} = \{a, b, d, e, f, h, j, k, l, m, n, q\}$ and $TS_{Tw3} = \{g, h\}$. There is a big difference between the TARs of the whole model and the TARs of the three sub-T-workflows. For example, the elements in the set of TAR $TARs = \{(g, f), (g, j), (g, k), (g, l), (g, n), (f, g), (j, g), (k, g), (l, g), (n, g)\}$ can be derived from the whole model but are not contained in the sub-T-workflows. It is enough for EL to be a #TAR complete log of the model in Figure 2 if it contains all TARs in the three sub-T-workflows, which means EL need not to contain the TARs in the set tars.

Comparing with the existing kinds of complete event logs, the #TAR complete event logs have the following advantages generally:

Log capacity: Fewer number of traces

For any process model, there are many different event logs that satisfy the same requirements of completeness. In order to compare the capacities of different complete logs of the same process model, we define the Minimal Complete logs of different types of completeness firstly.

Definition 2 (Minimal Complete Event Log). Let $PN = (P, T, F)$ is a sound WF-net, C is a kind of log completeness and $\{EL_1, EL_2, ..., EL_n\}(n \in \mathbf{N}^*)$ is the set of all event logs of PN that satisfy C. Then EL_i $(1 \le i \le n)$ is PN's minimal complete event log that satisfies C *iff*: $\forall j \in [1,n]$ and $j \neq i \Rightarrow \boldsymbol{cap}(EL_j) \ge \boldsymbol{cap}(EL_i)$.

The minimal globally complete event logs of a process must contain all possible execution traces of the process. So, it is obvious that the capacity of a minimal #TAR complete log will be never higher than that of a minimal globally complete one. Here are three examples for comparing the logs of models that contain structures of selection, parallelism and loop.

Example 3 As for the model in Figure 5(1), its minimal globally complete log is EL_g = {*acdef*, *acedf*, *bcdef*, *bcedf*}. The minimal #TAR complete log of the model is $EL_\#$ = {*acdef*, *bcedf*}.

Example 4 The globally complete event log of the model in Figure 5(2) contains infinite number of traces because of the loop structure. In order to simplify the comparison, we assume that the loop task g executes only once. Then EL_g = {*hacdef*, *hacedf*, *hbcdef*, *hbcedf*, *hagacdef*, *hagbcdef*, *hagacedf*, *hagbcedf*, *hbgbcdef*, *hbgbcedf*, *hbgacdef*, *hbgacedf*} while $EL_\#$ = {*hacdef*, *hbcedf*, *hagacedf*, *hbgbcdef*}.

Example 5 The difference in Figure 5(3) is that the loop structure is bound to a branch of the parallelism instead of the selection. We still assume that the loop task *g* executes only once. Then the minimal #TAR complete log is $EL_\#$ = {*acdef*, *bcedf*, *acdgdef*}, but the minimal globally complete one is EL_g = {*acdef*, *acedf*, *bcdef*, *bcedf*, *acdgedf*, *acdegdf*, *acedgdf*, *bcdegdf*, *bcedgdf*, *bcdgedf*}.

According to the examples, we find that the minimal #TAR complete log ($EL_\#$) always has fewer traces than the minimal globally complete one (EL_g).

Behaviour accuracy: Reflecting implicit dependency

Owning to the completeness of executing traces, globally complete event logs can fully express behaviour of any process models, including implicit dependency. But locally complete ones cannot reflect implicit dependencies correctly because of reducing the number of traces. This is why α-Algorithm (Aalst et al., 2004), that is, the most popular process mining algorithm, cannot reconstruct implicit dependencies from locally complete event logs. While giving a locally complete event log of a process model containing implicit dependencies as input, the result model derived by α-Algorithm would lose the structures of implicit dependencies. The locally complete event logs of the result model would contain more traces than the initial input logs. Since there are no appending information or constrictions of the locally complete event logs, you cannot define more processing steps for enhancing α-Algorithm. The limitation of locally complete event logs in reflecting implicit dependencies also restricts the optimisation of process mining algorithm.

However, if the event log satisfies the definition of #TAR completeness, the influences of implicit dependencies on model behaviour will be fully reflected. In other words, executing traces that are not allowed by the implicit dependencies are excluded by the #TAR complete logs, while there are no explicit declarations about the behaviour of implicit dependencies in locally complete logs. We find that definition of TAR* completeness also contains statements about implicit dependency behaviour. However, its restriction on the log is reflected through the declaration of not allowed TARs related to implicit dependencies, instead of the declaration of allowed executing traces. Although #TAR complete logs have extra restrictions on task sets of the traces, they look like having no differences with either globally or locally complete ones.

With the explicit restrictions on task sets of event logs, it is easier for us to expend mining algorithm, for example, α-Algorithm, to detect lost places of implicit dependencies after generating a first round result model. First, generate a new #TAR complete event log of the first round result model. Second, compare the new log with the input one. If they have the same task sets, there is no lost places in the result model. Otherwise, there must be some places representing implicit dependencies missing in the result model. Last, check the differences of the two event logs in

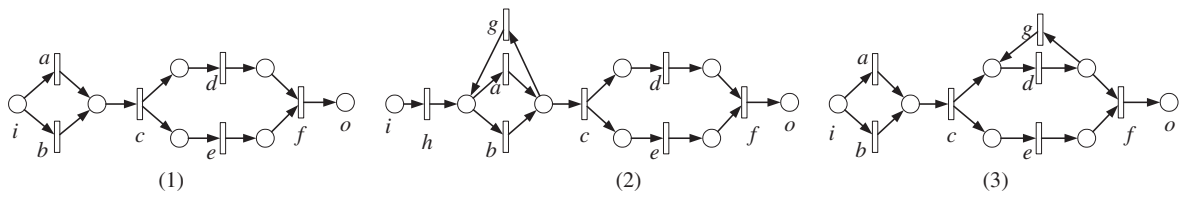


FIGURE 5 Three different models used in the comparison of different minimal complete logs

detail and then complement the missing places. Concrete expansion of the mining algorithm discovering implicit dependencies from #TAR complete event logs would be introduced in detail in Section 5.

Generating efficiency: No need to generate all TARs

Since there are much fewer traces in #TAR complete event logs than in globally complete ones, it will cost much less time for generating a #TAR complete event log than a globally complete one. While generating locally complete event logs, the algorithm should decompose the input model into fine-grained pieces for constructing all TARs. However, only executing paths of the model should be extracted for constructing event traces of the #TAR complete logs. Decomposing the input model into TAR pieces would theoretically take much more time than decomposing it into executing paths that consist of TAR pieces. The generating algorithm for #TAR complete event logs proposed in this paper will be described in detail in Section 4. The experimental results of comparing efficiencies of generating algorithms of different complete logs are presented in Section 6.

The characteristics of #TAR complete logs also bring benefits for improving the efficiency in discovering process models from very large event logs. Since the #TAR complete logs contain all task sets of the former model's sub-T-workflows which are connected only with places, all traces belonging to different task sets can be processed independently which means being handled in parallel. Then, a specific parallelised mining algorithm for #TAR complete event logs can be designed in theory (will be discussed in Section 5). Besides, checking similarity between complex processes with their #TAR complete event logs would also save time considering both the low capacity and accurate behaviour expression of #TAR complete logs. Before comparing process models using their logs, we need to generate those logs if there are no existing ones. In the next section, we introduce a reasonable approach for generating #TAR complete logs of given process models.

4 | GENERATING ALGORITHM

Proposing a novel definition of event log completeness and analysing its goodness is not our ultimate purpose. How to judge whether a log is of this kind completeness and how to make full use of this kind of event logs are more important and meaningful. But the foundation is that you know how to generate this kind of event logs according to a given process model. This will help to design the strategy for checking completeness or to measure the similarity between processes by their event logs. In this section, we mainly propose the algorithm for generating #TAR complete event logs of given processes.

If there is a restriction that the result logs should be minimal complete ones, each required TAR should show up as few times as possible (Wang et al. (2011)) The TARs of the model were acquired first, then after generating each executing trace the contained TARs and the trace would be recorded. If there is no new TAR contained by the executing trace, the trace would be dropped and go on generating new ones until all TARs have been contained once. Obviously, there is a high risk in wasting a lot of time in generating the result logs. However, checking time would be saved if we weaken the minimal restriction on the result log. Besides, the generating procedure will be simplified and solidified if we follow a pre-defined framework to generate event traces. More importantly, it will get rid of the dependency on randomly simulating the execution of process models.

So, in this paper, we propose a novel framework for generating #TAR complete logs of weakened minimal restriction. Content of this kind of #TAR complete log is defined as following:

1. For each parallelism, it satisfies Trace Completeness, that is, Global Completeness;
2. For each Main Path, its corresponding task set should be contained in the log;
3. For each Loop Sub-model, there must be at least one task set in the log containing the task set of the Loop Sub-model as the single loop at the same level.

Figure 6 shows the generating framework for the defined #TAR complete logs. Step 1 is to decompose the model with TS-invariant. First, split the model into several sub-models according to T-invariants of the model, then main paths and loops, as well as the level of loops could be derived. At the same time, connections among main paths and loops are also calculated. Then, for each main path or loop sub-model, calculate parallelisms with its S-invariant. Since a parallelism may be contained by several sub-models, we need to find all sub-models that share a same parallelism in order to save time in generating execution traces of parallelisms. In loops, there are also parallelisms, so connections among loops and parallelisms should also be derived. Step 2 is to generate complete execution traces of parallelisms. That is to generate all reasonable combinations of all tasks in the parallelism. Of course, you could generate TAR or local complete traces for parallelisms using randomly simulating strategy just like the one proposed in Wang et al. (2011) Step 3 is to insert traces of parallelisms into main path or loop sub-models. For each main path and loop that has only one single parallelism, generate its traces using those of the parallelism. Then, generate traces for main path and loop having more than one parallelism. The last step is to combine traces of loops with traces of main path or traces of main path with upper-level loops.

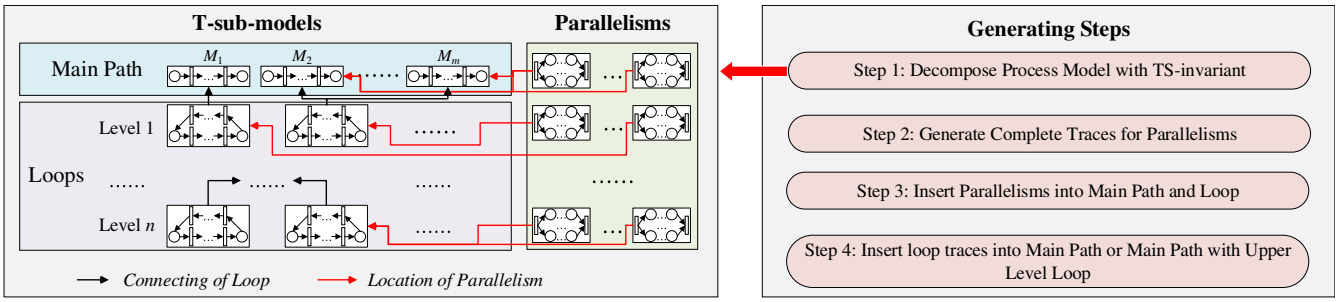


FIGURE 6 Main steps used in generating the #TAR complete event log of a given process

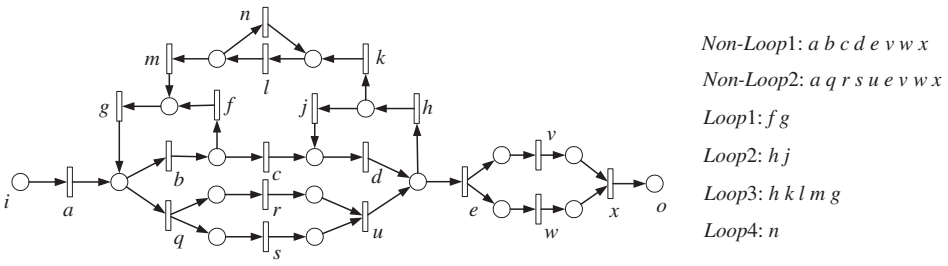


FIGURE 7 A complex example model with four loops to be used in the rest of this paper

The four steps are detailed in the following four subsections, respectively. The complex model in Figure 7 will be used as an example throughout these sections. There are two non-loop sub-models and four loop sub-models in the process. The task sets of each sub model are listed at the right in the figure.

4.1 | Decomposing with TS-invariant

The method for decomposing the process model with TS-invariants of Petri-nets has been briefly introduced in Section 2.1. But how to store the derived sub-models and how to describe their relations are not mentioned. For these, we have to observe the details of decomposing. Relational Matrix and Minimal Semi-positive T-invariant of Petri-nets will be used. Here are their definitions.

Definition 3 (Relational Matrix and T-invariant (Desel & Esparza, 2005)).

1. Any Petri-net $PN = (P, T, F)$ could be described as a Relational Matrix between places and transitions, PN: $P \times T \rightarrow \{-1, 0, 1\}$, the element $PN(p,t)$ in the matrix is defined as:

$$PN(p,t) = \begin{cases} -1 & \text{if } (p,t) \in F \\ 0 & \text{if } ((p,t) \notin F \wedge (t,p) \notin F) \vee ((p,t) \in F \wedge (t,p) \in F) \\ 1 & \text{if } (t,p) \in F \end{cases}$$

2. For a Petri-net $PN = (P, T, F)$, its T-invariants are the integer solutions of the Equations $PN \cdot Y = 0$. The set of solutions is $S = \{S_1, S_2, \ldots, S_n\}$, $n \in \mathbf{N}^*$. S_i ($i \in \mathbf{N}^* \wedge i \in [1, n]$) is semi-positive *iff* $\forall t \in T \Rightarrow S_i(t) \geq 0 \wedge S_i \neq 0$.
3. If there is no other semi-positive T-invariant S_k contained by the semi-positive T-invariant S_i, that is, $S_k \subset S_i$, then S_i is one of the minimal semi-positive T-invariants of PN. Any non-minimal semi-positive T-invariant is a linear combination of some minimal semi-positive T-invariants.
4. Let S_k be a minimal semi-positive T-invariant of PN, σ be the Parikh vector (Desel & Esparza, 2005) of S_k and M_0 be the starting state of PN, then $PN \cdot S_k = 0$ and this means $M_0 \xrightarrow{\sigma} M_0$ (i.e., starting from the state M_0, after the transitions in S_k firing, PN will get back to the state M_0).

Actually, a T-invariant corresponds to a loop structure constructed by the transitions in it. In order to decompose the model completely, a new transition t^* connecting the output place **o** and the input place **i** is added to the process. Then $PN = (P, T, F)$ turns to be $PN^* = (P, T \cup \{t^*\}, F \cup \{(\mathbf{o}, t^*), (t^*, \mathbf{i})\})$. By calculating the Equations $PN^* \cdot Y = 0$ of the model in Figure 8, we get the solutions in Table 2.

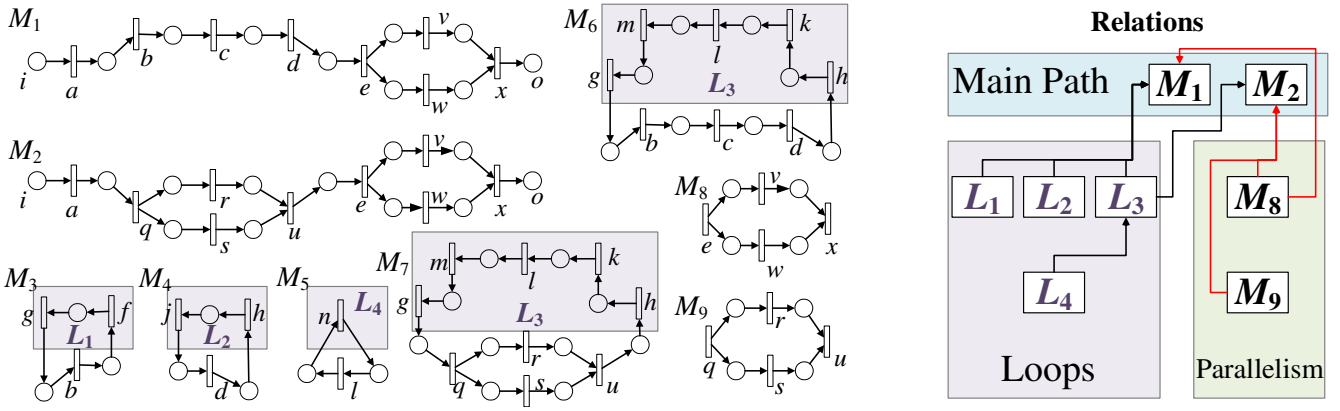


FIGURE 8 Sub-models of the model in Figure 7

The solutions with $S(t^*) = 1$ represent the non-loop sub-models and they are main path (MP). The other solutions are loop sub-models (LSM). The notation of a main path is *MP* = (*M*, *TS*) where *M* represents the corresponding sub-model and *TS* represents the task sets of *M*. Since, the loop sub-model has overlap with main path, there is no specific notation for LSMs. We only have notations for loop structure for each LSM. The notation of a loop structure is *LM* = ({*M*},{*FM*},*TS*), where {*M*} is the set of corresponding loop sub-models, {*FM*} is the set of father models that {*M*} binds to, *TS* is the task set of transitions that does not belong to models in {*FM*}. This means different LSMs may have the same loop structure and we mark them as a same *LM*. Besides, father models of a loop are models bound by all LSM containing the loop. So, although there are only four loops in the model, there are five LSMs in Table 2.

The {*FM*} in each *LM* represents the relationship between loops and main path and the relationship between loops themselves. The key problem is how to construct *MP*s and *LM*s according to the minimal semi-positive T-invariants in Table 2. The details for constructing {*FM*}s and *LM*s are shown in Algorithm 1. Firstly, the main paths are detected from the results of semi-positive T-invariants. Next, for each loop sub-model, the father models, that is, main path or upper level loops, are detected. If no father model found for the current loop sub-model, it must be bound to other unhandled loop sub-models. Then continue to handle next loop sub-model. Upon there is no loop sub-model left in the semi-positive T-invariants, all main paths, loop sub-models and relationship among them are generated. Eventually, loop structures would be generated by merging loop sub-models having the same loop task set.

Algorithm 1 Constructing Main Path and Loop Structure (model), where loop sub-models are detected firstly: *ConMPAndLSM*(*S*,*M*)

Input: Minimal Semi-positive T-invariants $S = [S_1, S_2, \cdots, S_n]$, corresponding sub-models $M = [M_1, M_2, \cdots, M_n]$.
Output: Main paths: *Set<MP> mps*, Loop structures: *Set<LM> lms*, Loop sub-models: *Set<LSM> lsms*.
Begin:
  Initialise *mps*, *lms*, *lsms*, *preInvSet*, *tmpInvSet* ← ∅
  For $S_i \in S$ **do**
    If $S_i(t_*) == 1$ **then** { *tmpInvSet* ← *tmpInvSet* ∪ {S_i}; *mps* ← *mps* ∪ {**new MainPath**(M_i, *taskSet*(S_i))}; *S.remove*(S_i) }
  end for
  preInvSet ← *tmpInvSet*; *tmpInvSet* ← ∅
  While $S \neq \emptyset$ **do**
    for $S_i \in S$ **do**
      fatherModels ← {Model in *M* for *preInv* | ∀ *preInv* ∈ *preInvSet* ⇒ (*sharedTask* ← *taskSet*(*preInv*) ∩ *taskSet*(S_i)) ≠ ∅} *tmpInvSet* ← *tmpInvSet* ∪ {S_i}; *lsms* ← *lsms* ∪ {**new** LSM(M_i, *fatherModels*, *taskSet*(S_i) − *sharedTask*)}; *S.remove*(S_i)
    end for
    preInvSet ← *tmpInvSet*; *tmpInvSet* ← ∅
  end while
  lms ← ∪{ $(M_{\forall lsm_i \in LSMGroup}, \cup(lsm_i.fatherModels)_{\forall lsm_i \in LSMGroup}, TS_{\forall lsm_i \in LSMGroup})_{\forall lsm_a, lsm_b \in lsms \wedge lsm_a.TS == lsm_b \Rightarrow lsm_a, lsm_b \in LSMGroup}$}
  return *mps*, *lms*, *lsms*;
End

Next is to find shared parallelisms (*SP*s) among sub-models. To mark the shared parallelism (*SP*) among *MP*s or *LM*s, we first introduce the notation for parallelism *para* = (*StT*, {*LiB*}, {*RE*} *EnT*) where the four elements are *starting task* (*StT*), *list of branches* ({*LiB*}), *sequential restrictions for*

TABLE 2 Minimal semi-positive T-invariants of the model in Figure 7

a	*b*	*c*	*d*	*e*	*q*	*r*	*s*	*u*	*h*	*j*	*f*	*g*	*k*	*l*	*m*	*n*	*v*	*w*	*x*	*t**
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0	0	1	0	0	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0

tasks in different branches ({*RE*}) and *ending task* (*EnT*), respectively. Each branch is a set of tasks. For example, the parallelism containing task *e*, *v*, *w* and *x* can be marked with $para_1$ = (*e*,{{*v*},{*w*}}, {}, *x*). For each branch, tasks are recorded in the item of {*LiB*} according to their sequential orders.

The notation *SP* = (*M*, *para*,{*FM*}) is for marking the shared parallelism where *para* is the notation of parallelism, *M* is the model of parallelism and {*FM*} is the set of sub-models that contain this *para*.

For finding the parallelisms, we first try to decompose the sub-model with S-invariant theory of Petri-nets. S-invariants are solutions of the equations $PNT \cdot Y = 0$ where *PNT* is the transpose of the relational matrix *PN*(*p*,*t*). It just likes viewing the places as transitions and transitions as places in the model. An S-invariant represents a group of places on a loop that keep the state of the model which are marked by transitions unchanged after these places firing. If there are more than one S-invariants of the sub-model, there must be parallelisms in it. For each Main path and Loop Sub-model, calculate the parallelisms it has. Then, shared parallelisms among different main paths or loop sub-models are detected and marked. Since no parallelism can exist in both main path and loop structure, the father models of the parallelism in {*FM*} should either all Main Path or all Loop Structures.

Example 6 There are two *MP*s, five *LSM*s and two *SP*s for the model in Figure 7 calculated by TS-invariant as shown in Figure 8 (left part). But there are only four loop structures. The right part of Figure 8 shows the relations among main path, loops and parallelisms. The notations for main paths, loops and parallelisms are: MP_1 = (M_1, {*a*, *b*, *c*, *d*, *e*, *v*, *w*, *x*}), MP_2 = (M_2, {*a*, *q*, *r*, *s*, *u*, *e*, *v*, *w*, *x*}), LM_1 = ({M_3}, {M_1}, {*f*, *g*}), LM_2 = ({M_4}, {M_1}, {*h*, *j*}), LM_3 = ({M_6, M_7}, {M_1, M_2}, {*h*, *k*, *l*, *m*, *g*}), LM_4 = ({M_5}, {M_6, M_7}, {*n*}), SP_1 = (M_8, [*e*, {{*v*}, {*w*}}, *x*], {M_1, M_2}), and SP_2 = (M_9, [*q*, {{*r*}, {*s*}}, *u*], {M_2, M_7}). As shown in Figure 8, parallelisms M_8 (i.e., SP_1) and M_9 (i.e., SP_2) are both parts of main path M_1 (i.e., MP_1), and M_8 is also a part of main path M_2 (i.e., MP_2). Loop L_1, L_2 and L_3 (i.e., LM_1, LM_2, and LM_3) are all connected to main path M_1, and loop L_3 is also connected to M_2. At last, loop L_4 (i.e., LM_4) is connected to upper level loop L_3.

4.2 | Generating traces of parallelisms

In Figure 6, the second step is to generate execution traces of Parallelisms. Since locations of parallelisms in main paths and loop structures have already been detected, there is no need to consider how to merge traces of parallelisms into the entire execution trace of a running instance in this step. How to insert traces of parallelisms into mapping paths or loops would be illustrated in the next two subsections.

As said in the beginning of this section, in the proposed #TAR complete logs, parallelisms satisfy traces completeness. So, for each *para*, all possible executing traces should be generated. For each parallelism, all branches have been generated and recorded in {*LiB*} of the *para*.

To generate traces of parallelism is to find all reasonable combinations of tasks in different branches with respect to the special sequential restrictions ({*RE*}) of tasks from different branches. The strategy is as following:

1. Initialise a set, *TmpTraces* = {}, for recording all generated traces.
2. For each branch in the Parallelism, that is, in {*LiB*}, do the following steps:
3. For each existing trace in *TmpTraces*, inserting all tasks of this branch to a reasonable position considering the sequential restrictions on the task to be inserted. That is, checking if there are special sequential restrictions in {*RE*} on the task to be inserted, and if the restriction is related to tasks in the traces to be inserted into, the inserting position should satisfy the restriction.
4. Remove all traces in *TmpTraces* and add all newly generated ones into *TmpTraces*.

Eventually, *TmpTraces* would contain all traces for the parallelism. For each parallelism, there is a single *TmpTraces* for it. Since there may be too many tasks in the parallelism, the number of traces would be too large to be recorded in the memory of computer. So, while programming for the generating work, how to manage *TmpTraces* in the memory and when to take advantage of the disk should be considered.

For example, the traces of M_8 in Figure 8 are *TmpTraces* = {(*vw*), (*wv*)}, and those of M_9 are are *TmpTraces* = {(*sr*), (*rs*)}. In this step, there is no need to insert the starting task and append the ending task of the parallelism to the traces. While inserting these traces into main path or loop structure, the starting and ending tasks would be used to find the position to insert. Upon deriving all traces of parallelisms, it is ready to generate traces for main paths.

4.3 | Inserting parallelisms to main paths

The requirement on traces of main paths of the proposed #TAR completeness log is just to cover all task sets of main paths. This is easy to accomplish by generating only one trace for each main path. If there are parallelisms in the main path, just choose one trace of the parallelism to insert into the trace of the main path.

However, the proposed #TAR completeness also requires the log covering all traces of parallelisms. So, one trace for the main path containing parallelisms is definitely insufficient. But for each main path, we should not insert traces of all parallelisms it has into the traces of it, because parallelisms may be shared among different main paths. In order to minimise the log size, we need to ensure that traces of each parallelism are inserted as few times as possible. If the parallelism is contained by two main paths, then only one main path should contain all of its traces and the other one can just contain one trace of the parallelism. This is why we calculate the relations among parallelisms and main paths.

The following are the steps for generating traces of main paths, which would ensure that the fewest number of traces would be generated.

Step 1. Choose the parallelism SP_{max} which contains the largest number of traces from all *Unhandled* ones. If there are more than one parallelism having the largest number of traces, then randomly choose one from them.

Step 2. From all *Unhandled* main paths containing the choosen SP_{max}, that is, father models in {*FM*} of SP_{max}, choose the one having the largest number of parallelism, noted as MP_{maxP}. If there are more than one main path satisfies the condition, randomly choose one from them.

Step 3. Start to generate traces for MP_{maxP}: Each time, choose one distinct trace from each *Unhandled* parallelism contained by MP_{maxP}, and insert them into the trace for the main path, until there is no trace left in SP_{max}. During the process, if any parallelism's traces have all been inserted, mark the parallelism as *Handled* and the next time just use the last one trace of the parallelism to insert. Eventually, mark MP_{maxP} as *Handled* and save all traces of it.

Step 4. Iteratively conduct Step 1 to Step 3 until there are no *Unhandled* parallelisms.

Step 5. For all *Unhandled* main paths, generate one trace for them each and save all traces. During the process, if there are parallelisms in the main path, randomly choose one trace of each parallelism to insert to the entire trace.

Eventually, entire traces for all main paths are generated. All traces of parallelisms are included in the traces of main paths as few times as possible. The formal definition of these steps is described in Algorithm 2.

Algorithm 2 Generating Execution Traces of Main Paths to cover all traces of parallelisms they have: *GenMPTcoverPT*(*SP*s, *MP*s)

```
Input: All parallelisms SPs contained by all main paths MPs.
Output: Execution Traces of main paths in MPs covering all traces of parallelisms in SPs: Map<MP,Set<ET>> MPET.
Begin:
  Initialise Unhandled Main Path and Parallelism, UHandledMP, UHandledSP←∅, MPET←∅;
  for SP ∈ SPs do UHandledSP ← UHandledSP ∪ {SP} end for; for MP ∈ MPs do UHandledMP ← UHandledMP ∪ {MP} end for
  while UHandledSP ≠∅do
    SP_max ← argmax_{SP ∈ UHandledSP} NumberofTraces (SP); MP_maxP ← argmax_{MP∈{FM}of SP_max∩UHandledMP} NumberofParallelisms
    (MP) traces MP ← ∅; rootTrace ← PartialTraceOf (MP_maxP); subSPofMP ← ParallelismsOf (MP_maxP)
    while UHandledSP ∩ subSPofMP ≠ ∅ do
      tmpTrace ← rootTrace
      for SP ∈ subSPofMP do
        if SP ∈ UHandledSP then { toBeInsert, NONEXT ← NextUnhandled (SP); if NONEXT then UHandledSP.remove(SP) }
        else toBeInsert ← RandomlyChooseTrace (SP)
        tmpTrace ← InsertParallelismTrace (tmpTrace, toBeInsert, SP)
      end for tracesMP ← tracesMP ∪ {tmpTrace}
    end while
    MPET.put(<MP, tracesMP>); UHandledMP.remove(MP)
  end while
  for MP ∈ UHandledMP do
    rootTrace ← PartialTraceOf (MP); tmpTrace ← rootTrace; subSPofMP ← ParallelismsOf (MP)
    for SP ∈ subSPofMP do
```

```
        toBeInsert ← RandomlyChooseTrace(SP); tmpTrace ← InsertParallelismTrace(tmpTrace, toBeInsert, SP)
      end for
      MPET.put(<MP, {tmpTrace}>); UHandledMP.remove(MP)
    end for
    Return MPET
  End
```

Taking the model in Figure 7 as an example, the parallelisms M_8 and M_9 have the same number of traces. No matter which one is chosen as the first Unhandled parallelism, M_2 would be chosen as the first Unhandled main path. While generating each trace for M_2, traces of both M_8 and M_9 would be inserted. Assuming that in the first round, the trace (*aqrsuevwx*) is generated, then in the next round, it must be the trace (*aqsruewvx*) being generated. At this time, all parallelisms of the model are marked as *Handled*. Then generate one trace for unhandled M_1 by randomly choosing one trace from M_8 and inserting to the entire trace. The result trace is either (*abcdevwx*) or (*abcdewvx*). At last, all main paths are marked as *Handled*.

4.4 | Inserting loops

The proposed #TAR completeness event log requires that for each loop structure, there is at least one task set containing the task set of the loop structure as the single loop in the task set. Since there may be parallelisms in the loop structures, the loop traces should cover all traces of parallelisms they contain. So, firstly, the loop structures are treated as main paths to generate traces for them to cover all parallelisms. Each time, traces of loops of a same level are generated considering their corresponding parallelisms. And they are generated from Level 1 to Level *n* in order.

Upon traces for each single loop are generated, they should be inserted into traces of one main path that the loop is connected to generate new traces. In order to generate as few new traces as possible to cover all traces of one single loop, the following processing steps should be followed:

Step 1. For currently processed level of loops, preserve one trace for each upper level father model firstly. Then conduct the following sub-steps if there are loops that satisfy the condition of any sub-step until there are no *Unhandled* loops left in this level. And each time a father model has no left traces, cut the connections between it and the loop that has other father models which has traces left.

Step 1.1. Processing *Unhandled* loops having only one father model. Start from the first left trace of the father model, for each trace insert one distinct trace of the loop. If the number of left traces of the loop is more than the number of left traces of the father model, after there are no traces left in the father model, generate new traces by inserting each left trace of the loop into the last trace of the father model. Eventually, while there are no traces in the loop to be inserted, mark the loop as *Handled*.

Step 1.2. Processing *Unhandled* loops occurring to be the single loop connected to one father model FM_any of them. Start from the first left trace of the father model FM_any, for each trace insert one distinct trace of the loop. If the number of left traces of the loop is more than the number of left traces of the father model, after there are no left traces in the father model, cut the connection between the loop and the father model (i.e., remove father model from {*FM*} of the loop). Otherwise, while there are no traces in the loop left, mark the loop as *Handled*.

Step 1.3. Processing the loop with the most left traces among all *Unhandled* loops. Among its father models, choose the one having most left traces for inserting. If the loop has more left traces than the chosen father model, cut the connection. Otherwise, mark the loop as *Handled*.

Step 2. For loop level from 1 to *n*, conduct **Step 1**.

Table 3 shows the process of handling loops in the model shown in Figure 7. The first row shows the traces for each main path and loop. According to Figure 8, there are three loops in the first level. By checking the conditions of loops in the first level, we find that L_1 and L_2 satisfy the condition of Step 1.1. Then conduct Step 1.1 for L_1 and L_2 and results shown in the second row are derived. At the same time, there are no left traces in M_1, the connection between M_1 and L_3 is cut. Since there is still unhandled loop L_3 in level 1, continue to check the conditions of left loops. Then Step 1.1 is conducted for L_3. The result in the third row is derived. Next is to handle loops in the second level. Step 1.1 is conducted for the only loop L_4 and the result in forth row is generated. The last two rows show the ultimate traces in the #TAR completeness log generated using the proposed approach.

5 | ENHANCEMENT OF MINING ALGORITHM

Before introducing the enhancement for any mining algorithm in handling #TAR complete event logs, we illustrate the ideally parallelised approach for mining process models from #TAR complete event logs. The parallelised way would save a lot of time comparing with non-parallelised ones while handling large scales of logs. Firstly, event traces belonging to a same task set are allocated to a same computing node. At the same time, frequencies of each trace are counted, as well as the number of traces belonging to each task set. Then, we can drop out trace or task set outliers. Secondly, relationships among task sets are calculated to derive loop structures. Thirdly, sub-models are discovered from traces

TABLE 3 Process of inserting loops of the model in Figure 7

Initialise	$M_1 \rightarrow (abcdevwx)$; $M_2 \rightarrow (aqrsuevwx),(aqsruewvx)$; $L_1 \rightarrow (fg)$; $L_2 \rightarrow (hj)$; $L_3 \rightarrow (hklmg)$; $L_4 \rightarrow (n)$
Step 1	Check condition⇒step 1.1: $M_1 + L_1 \rightarrow (abfgbcdevwx)$; $M_1 + L_2 \rightarrow (abcdhjdevwx)$
	Check condition⇒step 1.1: $M_2 + L_3 \rightarrow (aqsruhklmgqsruewvx)$
Step 1	Check condition⇒step 1.1: $M_2 + L_3 + L_4 \rightarrow (aqsruhklnlmgqsruewvx)$
Ultimate result	$M_1 \rightarrow (abcdevwx)$; $M_2 \rightarrow (aqrsuevwx)$; $M_1 + L_1 \rightarrow (abfgbcdevwx)$; $M_1 + L_2 \rightarrow (abcdhjdevwx)$;
	$M_2 + L_3 \rightarrow (aqsruhklmgqsruewvx)$; $M_2 + L_3 + L_4 \rightarrow (aqsruhklnlmgqsruewvx)$

belonging to each group of task sets. At last, all sub-models are merged into an overall result model. Note that the second and last steps cannot be parallelised.

Besides the parallelised design, there is a repair operation which could be the enhancement of any mining algorithm for discovering implicit dependencies from #TAR complete event logs. As mentioned in Section 3, the implicit dependencies can be found by comparing task sets of the input log and those of the new log generated for the result model. The concrete repair operation can be conducted according to the following Theorem.

Theorem 1 *Let* PN *be the discovered result model,* EL *be the input event log,* tsm *and* tse *are the task sets of* PN *and* EL*, respectively, then:*

1. if $\exists ts \in tsm \wedge ts \notin tse \Rightarrow$ places indicating the implicit dependencies are missed,
2. if $\exists ts \in tse \wedge ts \notin tsm \Rightarrow$ causal dependencies between tasks in implicit dependencies are missed.

Proof of Theorem 1 Let PN' be the former model, then *tse* are also the task sets of PN'. If there are implicit dependencies missed in *PN*, *PN* will generate more possible running cases than PN', which means $\exists ts \in tsm \wedge ts \notin tse$. If there are only causal dependencies (denoted by an "arc") between tasks constructing implicit dependencies missed, *PN* will be no longer a sound one. Then less task sets could be derived correctly from *PN* than the former model PN', which means $\exists ts \in tse \wedge ts \notin tsm$.

For example, in N_1 of Figure 9, two places representing implicit dependencies are missed. There are four steps to repair: (a) Find all pairs of or-split and or-join where the or-join appears before the or-split in the result model, that is, {(*A*, *B*), *C*} and {*C*, (*D*, *E*)}. (b) Keep the pairs whose tasks of choices ever occur in the same task sets of the input event log for further processing, that is, {*A*, *C*, *D*} and {*B*, *C*, *E*}. (c) For reserved pairs, drop the ones that all possible combinations of whose choices occur together in any task set. (d) Find all combinations of choices of the reserved pairs that occur in any task set: (*A*, *D*) and (*B*, *E*) in N_1. The missed places are between (*A*, *D*) and (*B*, *E*).

For N_2 in Figure 9, three steps are needed to add the missing causal dependencies: (a) Find the task sets which are in event log but not in the result model, that is, {*A*, *E*, *D*, *F*, *G*}. (b) Get the tasks which are only in the task sets found in first step but not in other task sets, that is, *D* in {*A*, *E*, *D*, *F*, *G*} but not in {*A*, *E*, *C*, *B*, *F*, *G*}. (c) Add the causal dependencies between the task *t* found in the second step and the pre/post task of *t*'s pre/post task in any one execution trace of the task sets containing *t*. Note that (*AEDFG*) is a trace of the task set {*A*, *E*, *D*, *F*, *G*} where *A* is the pre-pre task of *D* and *G* is the post-post task of *D*. So, the causal dependencies between (*A*, *D*) and (*D*, *G*) should be added.

6 | EVALUATION

We evalulated the proposed #TAR completeness definition of event log, as well as the log generating algorithm, with theoretical analysis and experiments. Log capacity, generating efficiency and stableness and ability to express implicit dependencies of the #TAR complete logs are evaluated. In the theoretical analysis, series of manually made process models are used. For the experiments, series of randomly generated process models are used. The experiments are conducted on a personal desktop with Inter(R) 2.20GHz i7-6,650 U CPU and 16G memory running Windows 10 and JDK 1.8. The randomly generated process models are produced by the function provided in the process mining tool BeehiveZ (which can be found at: https://github.com/jintao05/BeehiveZ). Generating algorithms are also developed in BeehiveZ (which can be found at: https://github.com/lcynju/BeehiveZ-NJU).

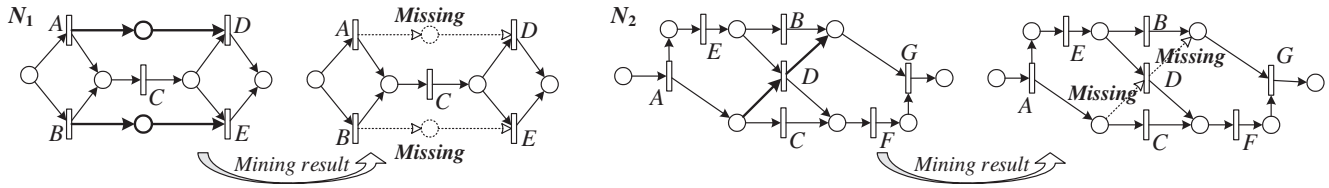


FIGURE 9 Result models discovered from event logs of models that contain two typical types of implicit dependencies

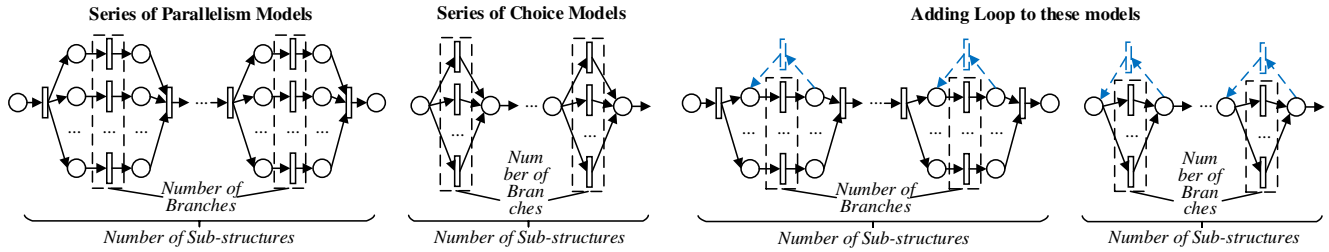


FIGURE 10 The manually generated models used in the comparison with global completeness

6.1 | Theoretical comparison of log capacity: #TAR versus global completeness

We firstly compare the theoretical log capacities of #TAR and Global complete logs. Four series of models are used in the comparison, as shown in Figure 10 and they are: (a) a series of models containing different number of parallelisms, (b) a series of models containing different number of choices, (c) a series of models containing different number of parallelisms with loops and (d) a series of models containing different number of choices with loops. Both the number of branches of each parallelism or choice and the number of parallel or exclusive sub-structures are changing as the models are changing. Assuming that the number of branches and sub-structures are noted as B and S, respectively, then the number of traces in #TAR and Global complete logs for the four series of models can be calculated and represented with B and S. The results are shown in Table 4.

For the parallelisms, there is only one task in each branch and B tasks in each sub-structure. For each sub-structure, any combination of B tasks would be one possible trace, which means there are $B!$ traces in total for each sub-structure. Since the sub-structures are connected to each other in sequence, choosing any one trace from each sub-structure would generate a possible trace for the entire model. So, there are $(B!)^S$ traces in the Global Complete log of the first series of models. However, #TAR completeness just requires that the log contains all possible traces for each parallelism instead of the entire model. Since the sub-structures are connected in sequence, the largest number of traces of all parallelisms is the least number of traces in the #TAR complete log to cover all traces of each parallelism.

The difference between choice and parallelism being a sub-structure is that, the number of traces of each choice sub-structure is the same with the number of branches, while that of each parallelism sub-structure is the number of combinations of tasks from different branches. In the second series of models, the choices are also connected in sequential. The entire number of traces should also be the number of combinations of sub-structures for the global complete log, that is, B^S. It is much fewer than that of the first series of models. As shown in Table 4, the number of traces in #TAR complete log is the same as that of the global complete one. It is because that there are B^S T-invariants while decomposing the model, which means there are B^S Main Paths of the model. There is one trace for each main path and eventually B^S traces for the model.

While adding loop tasks to the models in the way Figure 10 shows in the last two columns, the number of traces in both #TAR and Global complete logs of these models change dramatically. The state explosion problem of the global complete logs of the parallel models is much more prominent. Firstly, if all loop tasks are not executed, there will be the same number of traces as the first series of models. If all loop tasks are executed only once, there will be $\frac{(B+2)!}{3!}$ traces for each sub-structure: the trace of the branch containing the loop task has three tasks, and $B+2$ tasks for the sub-structure in total. Eventually, there will be $(\frac{(B+2)!}{3!})^S$ traces for this condition. No wonder there are conditions that some loops are executed and others are not, and different loops can be executed for different times. In Table 4, the result of all loop tasks executed only once is given. For #TAR completeness, $B!$ traces have already cover all traces of main path. In order to cover all loops, there should be at least one entire trace containing one single loop for each loop. Only one trace for the main path should be reserved and others could be used for inserting loops. The number of loops is S, so the entire number of traces is the $max(S+1, B!)$. In models of choices with loops, assuming that each loop executes at most once, then for each sub-structure, there are B (loop does not execute) + B^2 (loop executes once) possible traces. Eventually, for the entire model, there are $(B+B^2)^S$ traces in the Global Complete log. For the #TAR complete log, there are B^S traces for the main paths. After adding traces for containing each loop (there are S loops), the total number is B^S+S.

	Parallelisms	Choices	Parallelisms with loops	Choices with loops
Global	$(B!)^S$	B^S	$(\frac{(B+2)!}{3!})^S$, asumming loops always execute	$(B + B^2)^S$
#tar	$B!$	B^S	$B!$ If $B! - 1 > S$, otherwise $S + 1$	$B^S + S$

TABLE 4 Number of traces in complete logs of different models: B and S for number of branches and sub-structures respectively

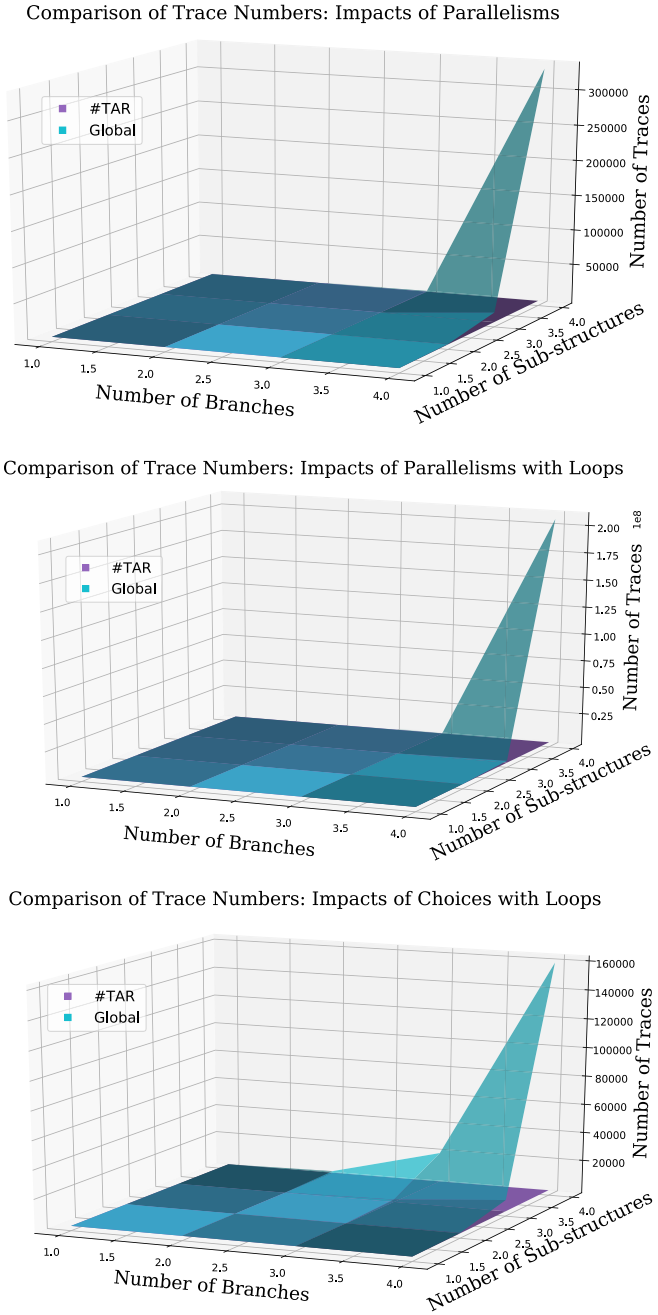


FIGURE 11 Visualised capacity comparison of complete logs of different type of models: #TAR versus Global

Figure 11 intuitively presents the large gaps between the numbers of traces in #TAR complete and Global complete logs for series of models shown in Figure 10 (except the second series of models, since there are the same number of traces in different types of logs). Among three types of basic structures in process models, parallelism has the largest influence on number of traces in global complete log. However, for #TAR complete log, the parallelism has less influence on number of traces than the choice. So, the more parallelisms the model has, the more traces would be saved in expressing the model behaviour using #TAR complete log than Global complete one. If there are loops on branches of parallelisms, far more number of traces would be saved in #TAR complete log.

6.2 | Experimental comparison: #TAR versus TAR* completeness

Table 5 shows the statistics of automatically randomly generated process models for the comparing experiments. Rows of the table represent the groups of models having different number of tasks. For each group of models, we generate five different sub-group models of different max-outputs (i.e., from 1 to 5) of each task. For each sub-group, we randomly generate 10 models and the average number of items in models of each sub-groups are shown in Table 5. The types of items are: Place, Arc, TAR, And-split, Xor-split and Loop. We find that the larger the max-outputs is, the fewer the places in the models. The large the number of tasks is, the more xor-splits and loops the model contains. Generating algorithms of #TAR and TAR* will be applied on these models to collect number of traces in the log and the generating efficiency. Besides, for investigating the properties of #TAR completeness, we generate another group of models containing 25 tasks.

1. *Log capacity*: The first sub-figure in Figure 12 shows the average number of traces in #TAR and TAR* complete logs of different sub-groups of models, respectively. Three conclusions can be drawn from this figure. First, the number of traces in both #TAR and TAR* complete logs increases as the number of tasks and the number of max-outputs of tasks increase. Second, the number of traces in TAR* complete logs always be larger than that in #TAR complete logs. Last, growing speeds of the number of traces in two types of logs are almost the same.

It is reasonable that the number of traces in both types of logs increases as the number of tasks and max-outputs of tasks increasing, since both properties of models represent the complexity of the models. The more complex the model is, the more various its behaviour is, and the more traces are needed to cover its behaviour. The size of TAR* complete log is mainly relying on the number of TARs in the model: the more TARs, the more traces. Comparing the increasing trends of the number of TARs in Table 5 and the increasing trends of number of traces in Figure 12 by aligning the number of tasks and max-outputs proves the relationship between number of TARs and number of traces for TAR* complete logs. However, the number of traces in #TAR complete log is determined by several detailed properties of the model, such as the number of main path (largely related to the number of xor-splits), number of parallelisms as well as the length of each branches. It is not easy to analyse concrete relationships between the number of traces and these detailed factors for #TAR complete log. But, it is true that the more tasks the model has and the more outputs the tasks have, the more traces the #TAR complete log has.

The reason of that TAR* complete logs are always having more traces than #TAR complete ones is that there may be duplicated traces in the TAR* complete logs while there are only distinct traces in #TAR complete logs. Theoretically speaking, there should be less traces in the TAR* complete log, especially for models with long branches parallelisms or sequentially connected xor-splits. However, its generating algorithm relies on a Random generating strategy, which leads to that a same trace may be generated for many times. Besides, there is a very low probability for randomly generated models satisfying the conditions leading to large number of traces in #TAR complete logs.

For trying to investigate the complex relations between model structures and the number of traces in corresponding #TAR complete logs, we randomly generate another 100 models having 25 tasks each and generating their #TAR complete logs with the proposed generating algorithm. Six properties representing the complexity of a model are investigated, namely, Number of TARs, Number of Loops, Sum of Average Length of Parallel Branches, Number of XOR-splits, Total Outputs of And-splits and Total Outputs of XOR-splits. Right two figures in Figure 12 show correlations between each property of the model and log capacities. Obviously, the number of combinations of tasks in parallelisms almost determines the number of traces. This is reasonable since that the goal of #TAR is to cover all traces of parallelisms. Traces of parallelisms are roughly calculated by $\sum_{mp \in MainPath}\left(\prod_{para \in mp}\left(\frac{tasks\,(para)!}{\prod_{bra \in para} tasks\,(bra)!}\right)\right)$, where *tasks*() returns the number of tasks in the given structure. This formula implies that the number of main paths also has a big influence on the number of traces. As shown in lower right of the last sub-figure in Figure 12, the number of total outputs of xor-splits, which determines the number of main paths, is correlated to the number of traces to some extents. Besides, the number of TARs also has stronger impacts than other factors. As the number of TARs increases in the model, the model turns to be more complex, which leads to the log capacity much more likely to be larger than other models with the same number of tasks.

As expected, the number of loops is not strongly correlated with the log size. But surprisingly, it seems that the more the number of loops, the smaller the log size. Although the definition of #TAR emphasises that the loop does not affect the log size, it does not say that the more loops, the smaller the scale. However, this is also understandable because the total number of tasks in the model is fixed. When the number of loop structures increases, other structures are reduced accordingly, such as concurrency and selection, so the log size becomes smaller.

Earlier analysis provides that some factors of the model have more obvious influence on the log size than the others. However, since the model is a whole, the factors do not exist independently, and the trade-offs are different. Therefore, in a broad sense, each factor has a certain impact on the size of the log. To calculate the capacity of #TAR complete log, you should dig deeply to analyse the relations among Main Path, Parallelism and Loops.

2. *Generating efficiency*: First sub-figure in Figure 13 shows the efficiency comparison of two log generation algorithms. Obviously, as the scale and complexity of the model increase, the efficiency of generating both types of logs is decreasing. Besides, the efficiency of generating #TAR complete logs is always higher than the efficiency of generating TAR* complete ones. Overall, the efficiency of generating #TAR complete logs changes smoothly as the scale and complexity of the models change. However, there is a sharp increase in the efficiency of TAR* full log generation. This is mainly due to the random execution strategy of the TAR* generation algorithm. This strategy also causes a reduction in the

TABLE 5 Average number of items in randomly generated models: P(lace), A(rc), T(AR), (A)N(D-split), (X)O(R-split), and L(oop)

	MaxOut:1				MaxOut:2						MaxOut:3						MaxOut:4						MaxOut:5					
	P	A	T	NOL	P	A	T	N	O	L	P	A	T	N	O	L	P	A	T	N	O	L	P	A	T	N	O	L
5	6	10	5	0	5	10	6	0	1	0	4	10	6	0	1	0	3	10	7	0	1	0	4	10	6	0	2	0
10	11	20	10	0	9	20	17	0	2	1	8	20	17	0	2	1	7	20	16	0	2	1	5	20	16	0	2	0
15	16	30	15	0	12	30	18	0	4	2	11	31	24	0	3	1	10	30	24	0	3	0	11	31	29	1	4	2
20	21	40	20	0	17	40	24	0	4	2	14	41	37	1	6	4	14	40	26	0	4	1	11	41	41	1	4	3
25	26	50	25	0	21	51	42	1	6	4	19	52	51	1	6	4	19	52	55	1	6	5	14	51	53	1	6	6

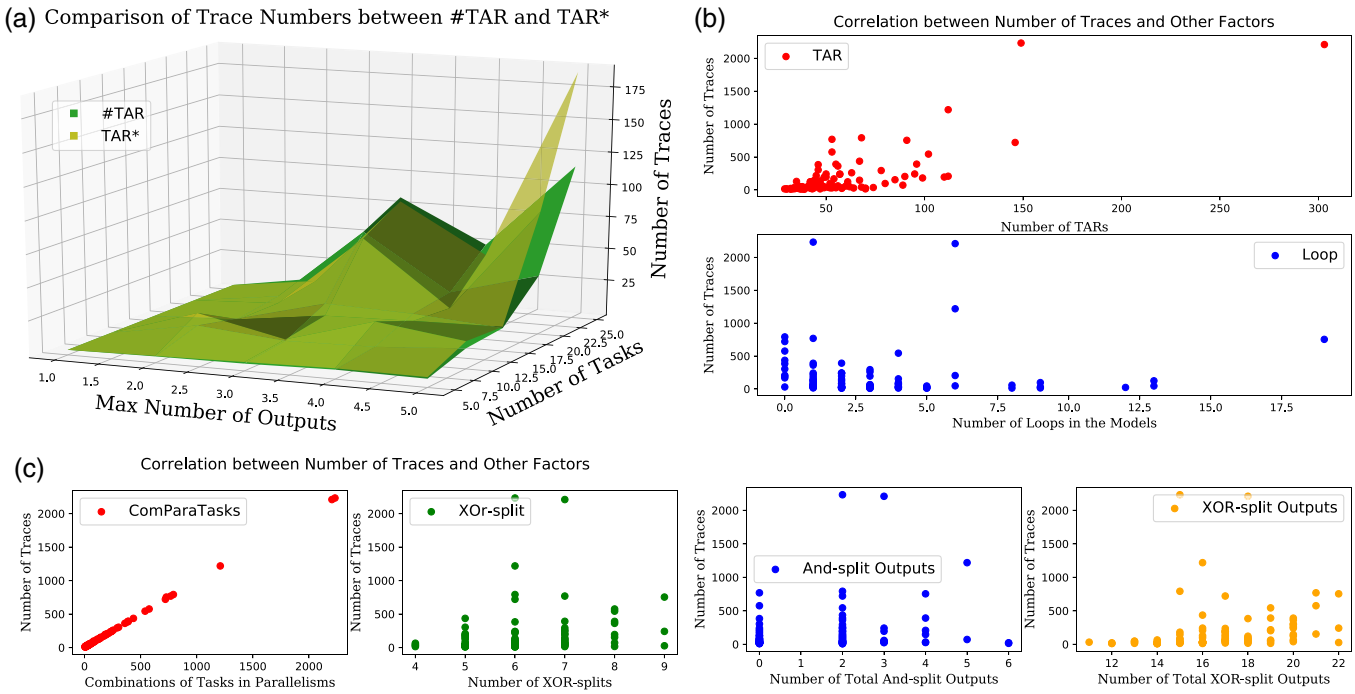


FIGURE 12 Comparisons of log capacity: #TAR versus TAR*; And influences of different properties of models on #TAR's traces number

generation efficiency while causing the number of generated traces always exceed those just covering all TARs. Another very important effect is that for the same model, the results of multiple runs of the same generation algorithm are different, that is, the stability of the algorithm is poor. The will be discussed in the next part.

Similarly, we analysed the correlation between the efficiency of the #TAR generation algorithm and the various factors of the model. Comparing Figure 13 with Figure 12, we can see that the effects of all factors on the number of traces are similar to those on the generation efficiency. The obvious difference is that, except for the number of loops, all other factors have a greater impact on efficiency than on log size. For example, when there are slight differences in the number of TARs or traces of concurrent structures, the efficiency of the algorithm will be very different. As another example, when the total number of branches of the choices increases, the probability that the generating efficiency is reduced is greater. However, the fewer the number of loops, the less significant the effect on efficiency, because the same number of loops, the efficiency distribution is more extensive than the trace number distribution. All in all, efficiency is similar to log scale to be influenced by the combination of factors, but is more sensitive to structural changes than the log scale.

3. *Generating stableness*: Comparing with TAR* generating algorithm, the proposed #TAR generating algorithm has higher stability from two aspects, namely, generating ability and generating results.

First, as long as the process model is sound and the model scale is within a certain range, the proposed #TAR complete log generating algorithm will be able to quickly generate its corresponding, and ensure that the log scale is as small as possible. Second, for the same model, the log content and generating efficiency of each time are almost the same. However, for the TAR* generating algorithm based on the random executing strategy, the log content and algorithm efficiency will be affected due to the uncertainty of randomisation. Besides, if the model is complex enough, theoretically, you might encounter situations where you can never derive a generated log. When calculating the TAR* log of a randomly generated model, some models have been processed for more than 10 min and have not ended. The left diagram in Figure 14 shows the model features that the TAR* generation algorithm cannot handle. The most prominent of these is the impact of TAR. When the number of TARs reaches a certain level, it is difficult to generate a log covering all TARs in a short time. The right chart in Figure 14 shows the differences in log scales and execution times obtained by the two algorithms for processing the same model for multiple times. The instability of the TAR* generation algorithm is verified, and the processing time is also verified to have a high correlation with the log scale.

6.3 | Correctness in expressing implicit dependency

By mining process models from event logs of different completeness, we compare the accuracy of different logs in expressing the behaviour of the process models. Figure 15 shows the different results of mining models from #TAR complete logs and locally complete logs of the same model

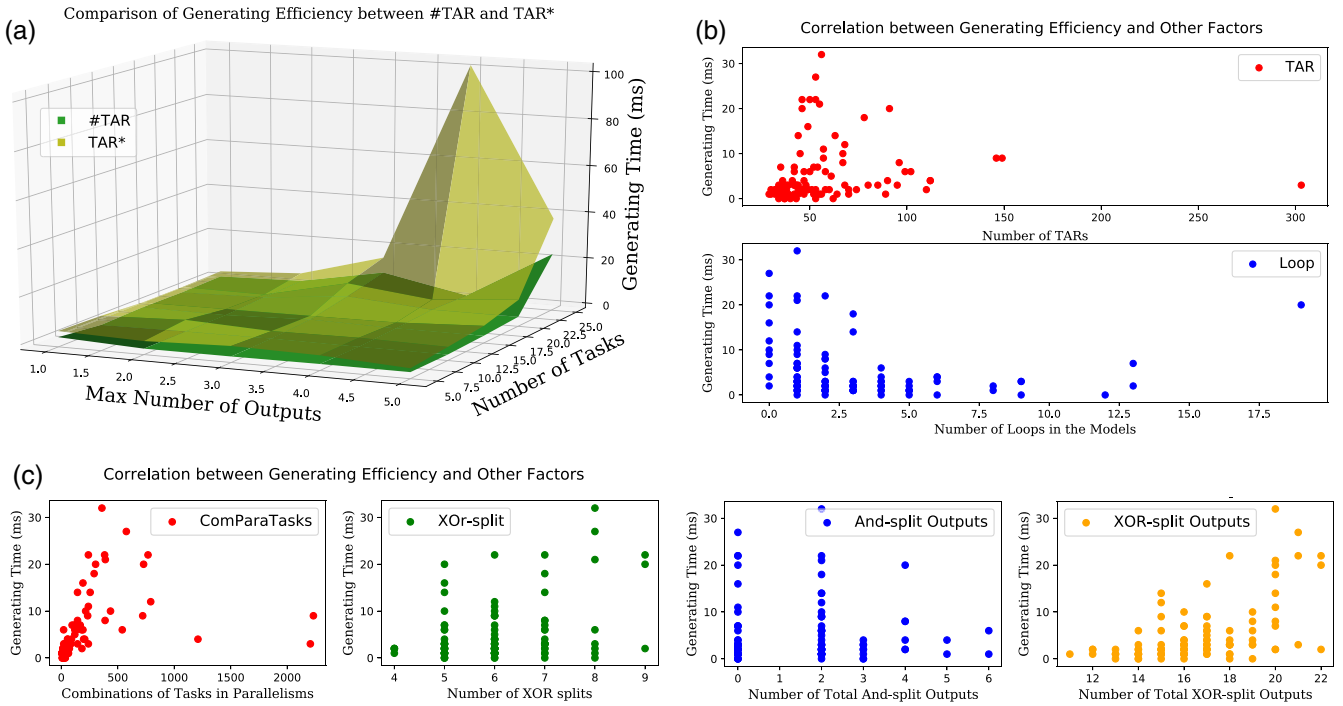


FIGURE 13 Comparisons of generating efficiency: #TAR versus TAR*; And influences of different properties of models on #TAR's generating efficiency

showing in left of Figure 15 (The α-Algorithm is applied here. While handling #TAR complete logs, the enhancement described in Section 5 is also applied.). The implicit dependencies of the former model can only be reconstructed from the #TAR complete logs. This proves that the #TAR complete logs can represent the behaviour of implicit dependency more accurately than locally complete ones. However, if we apply the enhanced α-Algorithm in processing locally complete log, we could also derive the left model in Figure 15 which is the same as the initial model. But you have no confidence in saying that you derive a correct model, because that the initial model could also be the right one in Figure 15. This means that different models may have the same locally complete event logs, but they would never have the same #TAR complete logs.

6.4 | Overall discussion

In summary, the following conclusions can be drawn from the presentation and analysis of the earlier experimental results: (a) #TAR complete log of a model has much fewer traces than the global complete log of that model, especially for model containing many parallelisms and having loops binding to parallel branches. (b) Theoretically, there are more traces in the #TAR complete log than in the TAR* log. But owning to the random executing strategy adopted in TAR*'s generating algorithm, there are more traces in TAR* complete log. (c) The generating efficiency of #TAR complete log is also higher than that of TAR* complete log. (d) The process model is a combination of various elements, and the existence of each element has an impact on the generating efficiency and the scale of the #TAR complete log. Among them, the influence of the number of parallel structures and the number of choice branches is most prominent. These two structures directly affect the number of TARs, so there is also a strong correlation between the number of TARs and log scale and generation efficiency. (e) The proposed #TAR complete log generating algorithm is more stable than that of TAR* complete log in both processing ability and result consistency. (f) The generated #TAR complete log can accurately express the behaviour of implicit dependency in the format of event trace.

Using the proposed #TAR complete log generating algorithm, you can generate complete logs of different models very quickly for designing or testing a process mining algorithm, for checking similarity between models, for checking the conformance or coverability of a given log. However, the scenes of better utilising the features of the #TAR complete log itself should be discovered more deeply. If any log of a model satisfies the restrictions of #TAR complete definition, that it can be used in what kind of task as the substitute of other kinds of logs to derive a better result or performance, should be studied in-depth. Besides, how to check the satisfaction of #TAR completeness should also be defined.

FIGURE 14 The model used in the experiments

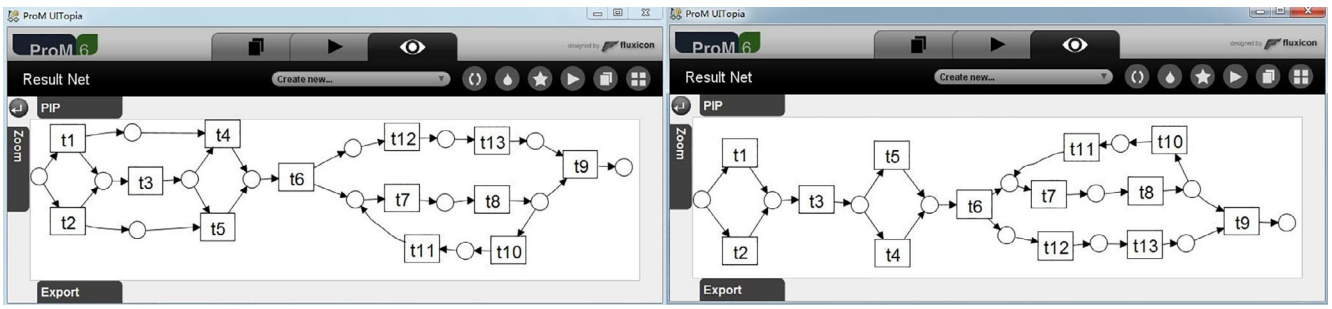


FIGURE 15 Result models of mining from #TAR complete logs (left) and locally complete logs (right)

7 | RELATED WORK

In the research of BPM, the event logs are among the most important data elements. Especially in the area of process mining, the event logs are treated as first citizen (Van Der Aalst et al., 2011). Researches that are most related to event log generating are process mining, business process similarity measuring and completeness checking.

Process mining is the idea of process discovery, conformance checking and enhancement (Rozinat & Van der Aalst, 2008; Van Der Aalst, 2012; Van Der Aalst et al., 2011). Process discovery is to construct a process model from given business logs. Different discovery algorithms (i.e., α-Algorithm and its extensions Aalst et al., 2004, region-based Carmona, 2012 and generic ones Eck et al., 2014) have different assumptions on completeness and formats of input logs (e.g., Song, Jacobsen, Ye, & Ma, 2016 define a Dependence-Complete Event Log) and these input logs determine the accuracy and efficiency of mining works eventually. Conformance checking aims to identify whether the reality, as recorded in the event logs, conforms to the model and vice versa. Different completeness of the input logs has different checking methods. The idea of process enhancement is to extend or improve an existing process model leveraging information of the actual process execution recorded in event logs.

Measuring similarities among process models can be used for finding changes of a business process quickly, reusing process conveniently or discovering service satisfying specific user requirements rapidly (Becker & Laue, 2012). Gerke et al. (2009) measure the process similarity by counting the same subsequences between the traces in their logs while Wang et al. (2010) taking the problem of infinite traces and infinite set of traces into account. De Medeiros et al. (2008) present a method to calculate the similarity of process models based on comparing traces obtained from actual process executions or by simulation. All these works assume the existences of complete logs of models. In order to break these

limitations, Wang et al. (2011) and Dong et al. (2014) proposed different methods for generating complete logs based on given process models for similarity measuring. Considering the advantages of #TAR complete logs as well as the generating algorithm, our method would be a good alternative of theirs.

Checking log completeness is known to be a difficult issue (Hee et al., 2011). Hee et al. (2011) overcome the problem of completeness checking by taking a probabilistic point of view, where a probabilistic lower bound for log completeness for three subclasses of Petri-nets, namely, WF-nets, T-WF-nets and S-WF-nets is provided. Yang et al. (2012) proposed an approach to evaluate the local completeness of an event log without knowing any information about the original process model and later in Yang, Wen, Wang, and Wong (2014), the approach was improved for estimating the local completeness of a log that contains a small number of traces precisely. However, in this paper, we do not investigate the completeness checking method for the newly proposed #TAR complete logs. We will concentrate on the completeness checking method for #TAR complete logs in the future work.

As for exploring the relationship between big data and BPM, as early as the big data concept was born, there are related scholars who propose how to practice business process mining technology in the era of big data (Aalst, 2014; Van Der Aalst, 2013). The main idea of these works is to extend existing business process mining techniques to big event data scenarios. For example, Song, Jacobsen, and Chen (2019) propose an approach for discovering scientific workflow protocol from event logs distributed in cloud platforms. Amelec Viloria et al. (2018) design a method for constructing learning processes of students from big knowledge data. However, more researches are conducted in investigating the intrinsic link between BDA and BPM. Sakr, Maamar, Awad, Benatallah, and Van Der Aalst (2018) discussed how to bridge the gap between the two by utilising Big Data Systems in analysing big business data, while Rialti, Marzi, Silic, and Ciappei (2018) arguing that BPM systems should integrate BDA capability. Moreover, Grover, Chiang, Liang, and Zhang (2018) proposed a concrete research framework for creating strategic business value from BDA. The idea of this paper is to generating process data for designing or testing related big process data analytics tools.

8 | CONCLUSION AND FUTURE WORK

While applying BDA including mining and visualisation in BPM, characteristics of business data should be well considered. Comprehensive and in-depth testing should be conducted on large-scale business logs of randomly generated process models to ensure the quality of the developed business BDA. Using randomly and automatically generated model can increase the coverage of the process model structures. However, strategies for automatically generating corresponding logs for these models are also needed.

So, in this paper, we propose a novel type of event log completeness for covering TARs and implicit dependencies of process and the corresponding log generating algorithm. The proposed completeness type names #TAR completeness, since it is an extension of TAR completeness by restricting (a) completeness of task sets without loops, (b) completeness of task sets containing only single loop of the same level. #TAR proceeds existing ones mainly in three folds: (a) lower log capacity than globally complete logs, (b) higher accuracy in expressing process behaviour than locally complete logs and (c) higher generating efficiency than both globally and locally complete logs. In other words, with #TAR complete logs, sufficient information for representing the behaviour of a process model is compressed in the logs of the lowest capacities. Owning to the sophisticated design of #TAR completeness, the generating algorithm saves the time for decomposing *TAR*s from the input model. Besides, it is no longer a simple simulating method without knowing what traces would be generated before the traces are generated. The proposed generating algorithm knows the target traces by decomposing the given process model. Main Paths, Parallelisms and Loops, as well as connection among them are detected within decomposing operation using TS-invariant analysis of Petri-nets.

However, the proposed instance of #TAR completeness is not the minimal #TAR complete log. All potential executing traces of each parallelism are generate by the proposed generating algorithm. However, for a parallelism, to cover its all TARs is enough to discover the structure accurately. To cover all TARs, fewer traces are needed. How to reduce the number of traces for parallelisms is one research direction of our future work. Besides, we will also do researches in measuring business process similarity with #TAR complete event logs to better visualise differences between models through event traces. We will try to measure similarity between different models and similarity between models and logs. Last but not least, we will design a proper method for checking whether an event log satisfies the definition of #TAR completeness to let the mining algorithm know if there is the need to repair the implicit dependencies in the result model.

ACKNOWLEDGEMENT

This work was supported by the National Natural Science Foundation of China (No. 61802167, 61572251), and Open Foundation of State key Laboratory of Networking and Switching Technology (Beijing University of Posts and Telecommunications) (No. SKLNST-2019-2-15).

ORCID

Chuanyi Li https://orcid.org/0000-0001-9270-5072
Jidong Ge https://orcid.org/0000-0003-1773-0942
Lijie Wen https://orcid.org/0000-0003-0358-3160

Victor Chang https://orcid.org/0000-0002-8012-5852
Liguo Huang https://orcid.org/0000-0001-7790-0195

REFERENCES

Aalst, W., Weijters, T., & Maruster, L. (2004). Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering.*, *16*(9), 1128–1142.

Aalst, W. M. P. (2014). Desire lines in big data. *Encyclopedia of Social Network Analysis and Mining.*, 351–364.

Ahmed, E., Yaqoob, I., IAT, H., Khan, I., Ahmed, A. I. A., Imran, M., & Vasilakos, A. V. (2017). The role of big data analytics in internet of things. *Computer Networks*, *129*, 459–471.

Becker, M., & Laue, R. (2012). A comparative survey of business process similarity measures. *Computers in Industry.*, *63*(2), 148–167.

Carmona, J. (2012). Projection approaches to process mining using region-based techniques. *Data Mining and Knowledge Discovery.*, *24*(1), 218–246.

De Alvarenga, S. C., Barbon, S., Jr., Miani, R. S., Cukier, M., & Zarpelão, B. B. (2018). Process mining and hierarchical clustering to help intrusion alert visualization. *Computers & Security.*, *73*, 474–491.

De Medeiros, A. K. A., Aalst, W. M., & Weijters, A. J. (2008). Quantifying process equivalence based on observed behavior. *Data & Knowledge Engineering.*, *64*(1), 55–74.

Desel, J., & Esparza, J. (2005). *Free choice petri nets*. Cambridge, England: Cambridge university press.

Dijkman, R., Dumas, M., Van Dongen, B., Käärik, R., & Mendling, J. (2011). Similarity of business process models: Metrics and evaluation. *Information Systems.*, *36*(2), 498–516.

Dong, Z., Wen, L., Huang, H., & Wang, J. (2014). CFS: A behavioral similarity algorithm for process models based on complete firing sequences, In: Meersman R. et al. (eds) *On the Move to Meaningful Internet Systems: OTM 2014 Conferences. OTM 2014. Lecture Notes in Computer Science*, vol 8841. (pp. 202–219). Berlin, Heidelberg: Springer.

Eck, M. L., Buijs, J. C., & Dongen, B. F. (2014). Genetic process mining: Alignment-based process model mutation. In: Fournier F., & Mendling J. (eds) *Business Process Management Workshops. BPM 2014. Lecture Notes in Business Information Processing*, vol 202, (pp. 291–303). Cham: Springer.

Gerke, K., Cardoso, J., & Claus, A. (2009). Measuring the compliance of processes with reference models. In: Meersman R., Dillon T., & Herrero P. (eds) *On the Move to Meaningful Internet Systems: OTM 2009. OTM 2009. Lecture Notes in Computer Science*, vol 5870, (pp. 76–93). Berlin, Heidelberg: Springer.

Grover, V., Chiang, R. H., Liang, T. P., & Zhang, D. (2018). Creating strategic business value from big data analytics: A research framework. *Journal of Management Information Systems.*, *35*(2), 388–423.

Hee, K. M., Liu, Z., & Sidorova, N. (2011). *Is my event log complete?—A probabilistic approach to process mining*. In 2011 Fifth International Conference on Research Challenges in Information Science 2011 May 19, Gosier, France (pp. 1–12). IEEE.

La Rosa, M., Dumas, M., Uba, R., & Dijkman, R. (2010). Merging business process models. In: Meersman R., Dillon T., & Herrero P. (eds) *On the Move to Meaningful Internet Systems: OTM 2010. OTM 2010. Lecture Notes in Computer Science*, vol 6426, (pp. 96–113). Berlin, Heidelberg: Springer.

Rialti, R., Marzi, G., Silic, M., & Ciappei, C. (2018). Ambidextrous organization and agility in big data era: The role of business process management systems. *Business Process Management Journal.*, *24*(5), 1091–1109.

Rozinat, A., & Van der Aalst, W. M. (2008). Conformance checking of processes based on monitoring real behavior. *Information Systems.*, *33*(1), 64–95.

Russom, P. (2011). Big data analytics. *TDWI Best Practices Report, Fourth Quarter.*, *19*(4), 1–34.

Sakr, S., Maamar, Z., Awad, A., Benatallah, B., & Van Der Aalst, W. M. (2018). Business process analytics and big data systems: A roadmap to bridge the gap. *IEEE Access.*, *6*, 77308–77320.

Schwank, J., Schöffel, S., & Ebert, A. (2018). Log-based process visualization. In: Ahram T., & Falcão C. (eds) *Advances in Usability, User Experience and Assistive Technology. AHFE 2018. Advances in Intelligent Systems and Computing*, vol 794, (pp. 741–751). Cham: Springer.

Song, W., Jacobsen, H. A., & Chen, F. (2019). Scientific workflow protocol discovery from public event logs in clouds. *IEEE Transactions on Knowledge and Data Engineering, Early Access*, https://doi.org/10.1109/TKDE.2019.2922183.

Song, W., Jacobsen, H. A., Ye, C., & Ma, X. (2016). Process discovery from dependence-complete event logs. *IEEE Transactions on Services Computing.*, *9*(5), 714–727.

Van Der Aalst, W. M. (1998). The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers.*, *8*(01), 21–66.

Van Der Aalst, W. M. (2012). Decomposing process mining problems using passages. In: Haddad S., & Pomello L. (eds) *Application and Theory of Petri Nets. PETRI NETS 2012. Lecture Notes in Computer Science*, vol 7347, (pp. 72–91). Berlin, Heidelberg: Springer.

Van Der Aalst, W. M. (2013). A general divide and conquer approach for process mining. In 2013 *Federated Conference on Computer Science and Information Systems.*, 07 November 2013, Kraków, Poland, pp. 1–10.

Van Der Aalst, W. M., Adriansyah, A., De Medeiros, A. K. A., Baier, T., Blickle, T., … Burattin, A. (2011). Process mining manifesto. In: Daniel F., Barkaoui K., & Dustdar S. (eds) *Business Process Management Workshops. BPM 2011. Lecture Notes in Business Information Processing, vol 99*, (pp. 169–194). Berlin, Heidelberg: Springer.

Viloria, A., Lis-Gutiérrez, J. P., Gaitán-Angulo, M., ARM, G., Moreno, G. C., & Kamatkar, S. J. (2018). Methodology for the design of a student pattern recognition tool to facilitate the teaching—Learning process through knowledge data discovery (big data). In: Tan Y., Shi Y., & Tang Q. (eds) *Data Mining and Big Data. DMBD 2018. Lecture Notes in Computer Science*, vol 10943, pp.670–679. Cham: Springer.

Wang, J., He, T., Wen, L., Wu, N., Ter Hofstede, A. H., & Su, J. (2010). A behavioral similarity measure between labeled Petri nets based on principal transition sequences. In: Meersman R., Dillon T., & Herrero P. (eds) *On the Move to Meaningful Internet Systems: OTM 2010. OTM 2010. Lecture Notes in Computer Science*, vol 6426, (pp. 394–401). Berlin, Heidelberg: Springer.

Wang, W. X., & Wang, J. M. (2012). TAR (*): An improved process similarity measure based on unfolding of petri nets. *Computer Integrated Manufacturing Systems.*, *18*(8), 1774–1784.

Wang, W. X., Wen, L. J., & Tan, S. J. (2011). Generating algorithm for complete log based on complete finite prefix. *Computer Integrated Manufacturing Systems.*, *17*(8), 1692–1702.

Wang, Y., Kung, L., & Byrd, T. A. (2018). Big data analytics: Understanding its capabilities and potential benefits for healthcare organizations. *Technological Forecasting and Social Change*, *126*, 3–13.

Weidlich, M., Mendling, J., & Weske, M. (2010). Efficient consistency measurement based on behavioral profiles of process models. *IEEE Transactions on Software Engineering.*, *37*(3), 410–429.

Wen, L., Van der Aalst, W. M., Wang, J., & Sun, J. (2007). Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery.*, *15*(2), 145–180.

Yang, H., Wen, L., & Wang, J. (2012). *An approach to evaluate the local completeness of an event log*. 2012 IEEE 12th International Conference on Data Mining, 10-13 Dec. 2012, Brussels, Belgium. (pp. 1164–1169). IEEE.

Yang, H., Wen, L., Wang, J., & Wong, R. K. (2014). CPL+: An improved approach for evaluating the local completeness of event logs. *Information Processing Letters.*, *114*(11), 607–610.

Zha, H., Wang, J., Wen, L., Wang, C., & Sun, J. (2010). A workflow net similarity measure based on transition adjacency relations. *Computers in Industry.*, *61*(5), 463–471.

Zha, H. P., Wang, J. M., & Wen, L. J. (2007). An algorithm of complete log generation for petri net models. *Journal of System Simulation*, *19-S1*, 271–276.

Zhu, J., Ge, Z., Song, Z., & Gao, F. (2018). Review and big data perspectives on robust data mining approaches for industrial process modeling with outliers and missing data. *Annual Reviews in Control.*, *46*, 107–133.

AUTHOR BIOGRAPHIES

Chuanyi Li is an Assistant Professor at Software Institute, Nanjing University. He received the BS and PhD degrees from Nanjing University in 2012 and 2017, respectively. His research interests include software engineering, systems and software quality assurance, process modelling, simulation and improvement, process mining.

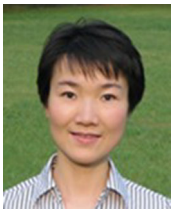
Jidong Ge is an Associate Professor at Software Institute, Nanjing University. He received his PhD degree in Computer Science from Nanjing University in 2007. His current research interests include workflow scheduling, software engineering, workflow modelling, stochastic network optimisation, process mining. His research results have been published in more than 50 papers in international journals and conference proceedings including IEEE TSC, JASE, FGCS, JSS, Inf. Sci., SCIS, ICSE, SCC, APSEC, ICSSP, HPCC, SEKE and so on.

Lijie Wen is an Associate Professor at Software Institute, Tsinghua University. He received the BE degree in computer science and technology, the PhD degree in computer software and theory in 2000 and 2007, respectively, from Tsinghua University, China. His main research interests include analysis of process and behavioural data, such as process mining, process indexing and retrieval, process similarity and process analysis. He has published more than 30 academic papers on international journals and conferences.

Li Kong is a PhD candidate at Software Institute, Nanjing University, under the supervision of Assistant Professor Chuanyi Li and Prof Bin Luo. Her research interests include software engineering, systems and software quality assurance, process modelling, simulation and improvement and process mining.

Victor Chang is a Full Professor of Data Science and Information Systems, School of Computing and Digital Technologies, Teesside University, Middlesbrough, UK, since September 2019. Previously, he was a Senior Associate Professor in Information Management and Information Systems at International Business School Suzhou (IBSS), Xi'an Jiaotong Liverpool University, China. He was a Director of PhD Program and the 2016 European and Cloud Identity winner of 'Best Project in Research'. Victor Chang was a Senior Lecturer in the School of Computing, Creative Technologies at Leeds Beckett University, UK, and a visiting Researcher at the University of Southampton, UK. He is an expert on Cloud Computing and Big Data in both academia and industry with extensive experience in related areas since 1998. He completed a PGCert (Higher Education) and PhD (Computer Science) within 4 years while working full time. He has over 100 peer-reviewed published papers. He won £20,000 funding in 2001 and £81,000 funding in 2009. He was involved in part of the £6.5 million project in 2004, part of the £5.6 million project in 2006 and part of a £300,000 project in 2013. He won a 2011 European Identity Award in Cloud Migration and 2016 award. He was selected to present his research in the House of Commons in 2011 and won the best papers in 2012 and 2015. He has demonstrated

10 different services in Cloud Computing and Big Data services in both of his practitioner and academic experience. His proposed frameworks have been adopted by several organisations. He is the founding chair of international workshops in Emerging Software as a Service and Analytics and Enterprise Security. He is a joint Editor-in-Chief (EIC) in *International Journal of Organizational and Collective Intelligence* and a founding EIC in Open Journal of Big Data. He is the Editor of a highly prestigious journal, *Future Generation Computer Systems* (FGCS). His security paper is the most popular paper in *IEEE Transactions in Services Computing*, and his FGCS paper has one of the fastest citation rates. He is a reviewer of numerous well-known journals and had published three books on Cloud Computing which are available on Amazon website. He is a keynote speaker for CLOSER 2015/WEBIST2015/ICT for Ageing Well 2015 and has received positive support. He is the founding chair of IoTBD 2016 www.iotbd.org and COMPLEXIS 2016 www.complexis.org conferences.



Liguo Huang is an Associate Professor, Department of Computer Science and Engineering at the Southern Methodist University (SMU), Dallas, TX, USA. She received both her PhD (2006) and MS from the Computer Science Department and Center for Systems and Software Engineering (CSSE) at the University of Southern California (USC). Her current research centres around process modelling, simulation and improvement, systems and software quality assurance and information dependability, mining systems and software engineering repository, stakeholder/value-based software engineering and software metrics. Her research has been supported by NSF, U.S. Department of Defense (DoD) and NSA. She had been intensively involved in initiating the research on stakeholder/value-based integration of systems and software engineering.

Bin Luo is a Full Professor at Software Institute, Nanjing University. His main research interests include cloud computing, computer network, workflow scheduling and software engineering. His research results have been published in more than 50 papers in international journals and conference proceedings including IEEE TSC, ACM TIST, JSS, FGCS, Inf Sci, ESA, ICTAI and so on. He is leading the Institute of Applied Software Engineering at Nanjing University.

How to cite this article: Li C, Ge J, Wen L, et al. A novel completeness definition of event logs and corresponding generation algorithm. *Expert Systems*. 2020;e12529. https://doi.org/10.1111/exsy.12529