Si YW, Yung WH. Unraveling iterative control structures from business processes. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(1): 158–190 Jan. 2021. DOI 10.1007/s11390-020-9516-3

Unraveling Iterative Control Structures from Business Processes

Yain-Whar Si and Weng-Hong Yung

Department of Computer and Information Science, University of Macau, Macau, China

E-mail: fstasp@um.edu.mo, harold.yung.8925@gmail.com

Received February 26, 2019; accepted December 4, 2020.

Abstract Iterative control structures allow the repeated execution of tasks, activities or sub-processes according to the given conditions in a process model. Iterative control structures can significantly increase the risk of triggering temporal exceptions since activities within the scope of these control structures could be repeatedly executed until a predefined condition is met. In this paper, we propose two approaches to unravel iterative control structures from process models. The first approach unravels loops based on zero-one principle. The second approach unravels loops based on branching probabilities assigned at split gateways. The proposed methods can be used to unfold structured loops, nested loops and crossing loops. Since the unfolded model does not contain any iterative control structures, it can be used for further analysis by process designers during the modeling phase. The proposed methods are implemented based on workflow graphs, and therefore they are compatible with modeling languages such as Business Process Modelling Notation (BPMN). In the experiments, the execution behavior of unfolded process models is compared against the original models based on the concept of runs. Experimental results reveal that runs generated from the original models can be correctly executed in the unfolded BPMN models that do not contain any loops.

Keywords business process management, iterative control structure, unravelling, zero-one principle, branching probability

1 Introduction

Business process management (BPM) deals with controlling the execution of application processes and activities according to pre-defined process models (workflow schemas) [1]. In many organizations, BPM is used to effectively model and enact their business processes with respect to organizational strategies and available resources. Workflow Management Systems (WfMS) can be used to define and use workflow systems [2]. WfMS can systematically monitor the execution associated with a business process. Iterative control structure (loop) is one of the most commonly used control flow constructs in business processes. Loops allow the repeated execution of tasks, activities or sub-processes according to the given conditions in a business process model. The descriptions of various loops are addressed in [3, 4]. During the enactment of business processes, iterative control structures in process models can significantly increase the risk of triggering temporal exceptions since activities within the scope of these control structures could be repeatedly executed until a predefined condition is met. It is also difficult for the workflow administrator to monitor the status of the process execution due to the iterations of these activities. However, a process designer may need to include these iterative control structures to better reflect the actual business processes during the modeling phase.

One possible solution to this problem is to transform the iterative control structures into workflows that do not contain any loops. After these loops are transformed, the resulting workflows can be analyzed for potential problems by the process designers. Loop unrolling is one of the well-known methods used in optimizing compilers [5]. Loop unrolling can be used to increase the efficiency of programs by reducing instructions that control the loop [6]. Simple approaches for entirely removing loops in process models have been proposed in recent years. Yu *et al.* [7] proposed a method

Regular Paper
The work was supported by University of Macau under Grant No. MYRG2019-00136-FST.

for removing iterative routing constructions. However, their method only considers simple structured loop pattern and the maximal number of iterations must be specified for each loop beforehand. Eder and Pichler [8] also proposed a method for transforming a simple structured loop into several executions paths for calculating a duration histogram. In their approach, the number of iterations and corresponding probabilities are calculated from the historical logs. The main purpose of their approach is to analyze the temporal properties of the workflow and thus it is not suitable for use in the process models which often contain complex iterative control structures.

In this paper, we propose two approaches to unravel (unroll) structured loops, nested loops and crossing loops. The proposed methods are based on workflow graphs and therefore they are compatible with modeling languages such as Business Process Modelling Notation (BPMN). The first method proposed in this paper supports Zero-One principle [9]. In this principle, loops are allowed to be executed at most once. Zero-One principle can be further extended for designing loops which are restricted by a maximum number of iterations. The second proposed method allows unravelling of loops based on branching probabilities assigned at the split gateways. These probabilities can be extracted from audit trails or previous execution logs. Branching probabilities allow process designers to control the iteration of loops based on a probability distribution. Both Zero-One principle and branching probabilities enable process designers to achieve finer control on how these iterative control structures are to be executed in the workflows. The unraveled workflow models can be analyzed for any potential issues by process designers or by automated verification programs. For example, after unraveling a crossing loop based on branching probabilities assigned at the split gateways, a process designer can estimate the times each task in the crossing loops is going to be executed. The unravelling also provides the detailed visualization of the control flow of the process model after all loop-back branches are removed.

In this article, several examples are used to illustrate the step-by-step unfolding of various process models based on branching probability in decision nodes. Based on the concept of runs, we have also conducted several experiments to analyze the execution behavior of unfolded process models by comparing against the original models. From the experimental results, we find that each set of 10 000 runs generated from the original models can be correctly executed in the unfolded BPMN models that do not contain any loops. In summary, the contributions made in this paper are two folds.

- In this paper, we propose two novel unravelling methods that can unfold not only simple structured loop patterns, but also nested and crossing loops.
- The proposed methods can unravel loops based on branching probabilities in addition to Zero-One principle.

The rest of the paper is organized into five sections. A review of recent work is given in Section 2. The methods for unravelling of iterative control structures are given in Section 3. In Section 4, we detail the experimental results for verifying the correctness of unfolded models. In Section 5, we summarize our ideas and future work.

2 Related Work

Transforming or reducing loops from programs has been extensively studied in the past. In the context of code optimization by compilers, loop unrolling [10] is one of the optimization techniques for reducing the cost of loop overhead [11]. Loop unrolling (also called loop unwinding) is used by compilers to optimize the loops and utilizes the CPU pipeline in the best possible way [12]. In recent work, loops were unrolled based on a predefined number of fixed iterations [13] or Zero-One principle [9]. In [13], loop unrolling transforms a loop by a factor of K in which the loop body is repeated K times and the loop iteration space is reduced or eliminated when the loop is fully unrolled.

According to [14], loop unrolling can significantly improve the performance because programs usually spend a significant fraction of their execution time inside the loops. In [15], Huang and Leng proposed an approach for unrolling different types of loop constructs such as FOR-loops and WHILE-loops. In [12], Velkoski *et al.* studied the performance impact of loop unrolling on various processor types and memory patterns. Their study concluded that loop unrolling achieves greater speed-up on Intel, rather than AMD CPU. Recently, Weinhardt [16] proposed a transformation which allows the optimization of WHILE loops nested within a FOR loop. The proposed transformation was designed to produce a significant increase in speed over software execution in large FPGAs by allowing multiple parallel tasks to be executed. The difference between existing unrolling methods and the unravelling methods proposed in this paper is that in the former methods,

structured loops and simple-nested loops are only considered. Whereas in our approach, we also consider crossing loops.

In [17], a loop unrolling approach for loops with arbitrary (variable) execution counts was proposed for worst-case execution time (WCET) reduction in real-time applications. In traditional unrolling approaches, loops with unknown execution courses are unrolled with fixed unrolling factors. However, the approach proposed in [17] is based on the code predication method used in software pipelining of loops. The proposed approach in [17] performs code predication in conjunction with the unrolling steps. The technique starts from a simple data dependent loop (iteration dependent on the value of a variable) and directly generates an unrolled and predicated version. The approach proposed in [17] is also combined with other unrolling techniques for data dependent loops and loops with fixed execution counts.

The iterative control patterns (loop patterns) can be classified into structured loop (with pre-test or post-test condition) and arbitrary loop. A pre-test structured loop is a while loop construct equivalent to the while-do structure in programming languages. A post-test structured loop is a repeat loop construct which is equivalent to a repeat-until structure. In general, it is possible to convert these two types of structured loop constructs into one another without any changes of semantics. The arbitrary loop can be used to model the repetition of processes in an unstructured way without the need for specific looping operators or restrictions [3]. Structured loop pattern is commonly supported in many workflow products. In the context of scientific workflows, due to the exploratory nature of large-scale experiments, workflow configuration is likely to change during its life cycle. In [18], a form of iteration called dynamic loop is introduced where the condition of the loop and the data being processed in the loop may be adapted during the execution. In the context of manufacturing systems, loops are often denoted as rework loops to model the rework of defective parts. Due to their significant impact on the performance of manufacturing systems, rework loops were also extensively studied in several performance evaluation methods [19, 20]. In transactional workflows area, Choi *et al.* [21] adopted an approach based on cycle analysis graph (CAG) to formally represent a cycle as a graph to determine whether it will terminate or not. In their approach, the cycles in BPTrigger [22] are categorized as incremental, decremental, replacement, and rework.

Recently, a number of researchers have proposed methods to transform an arbitrary cycle model into a model with structured loops, each containing only one entry point and one exit point. Structure loops are supported by modeling languages such as Business Process Execution Language (BPEL). In recent work, a number of techniques [23–25] have been proposed to restructure or transform the unstructured loops from process models into equivalent structured loops. One of the widely-used techniques for analyzing arbitrary loops or unstructured flow graphs in programming languages or complier theory is the construction of a Structuring Tree. The aforementioned technique first decomposes an undirected or a directed process model graph into a tree structure. Next, part of the unstructured flow in the tree is extracted to detect potential errors such as endless-loop or deadlock in the process model. Several techniques are proposed to transform the unstructured model into a structure one according to the detected errors. Polyvyanyy *et al.* [23] proposed a method for searching hidden unstructured regions by the SQRT-tree technique in a process model. In their approach, unstructured regions are transformed into structured regions. They also presented the transformation of a multi exit arbitrary loop into an equivalent structured loop which can be accepted by BPEL.

In [26], Eshuis and Kumar defined an automated method to convert an unstructured process model containing a structure loop into an equivalent semi-structured process model, which contains blocks and synchronization links between parallel branches. In the context of enhancing reliability in long-running programs in Cloud and Fog computing, Siavvas and Gelenbe [27] proposed a mathematical model to estimate the average execution time of the program. The proposed model also considers programs containing single loops and nested loops.

In [28] Williams and Ossher proposed methods for converting loops with multiple entry or exit points into equivalent structured forms. The approach proposed in [28] does not entirely remove the loops. Rather, it eliminates any unstructuredness from the control flow diagrams and converts them into an equivalent flow diagrams. Koehler and Hauser [24] also proposed a novel transformation method based on Continuation to map unstructured cyclic business process models to functionally equivalent workflows which support structured cycles. They transformed the loop structure into a set of abstract continuation equations similar to the SET equations and applied the rules to manipulate the links

in the graph. After the transformation, all unstructured cycles will be completely replaced by well-structured cycles. However, the transformed structured models can only be designed in modeling language such as Business Process Execution Language for Web Services (BPELWS). Kiepuszewski *et al.*[29] analyzed potential methods for transforming cycles and provided examples for transforming arbitrary loop structures into structured ones by re-allocating nodes as well as modifying relevant conditions. The method by Kiepuszewski *et al.*[29] duplicates the nodes which occurred in the paths of multi entry or exit loops. Their approach also uses conditional nodes to eliminate the unstructured entry or exit path in order to restructure the model with structured loop pattern.

In the workflow verification area, Heinze *et al.*[30,31] proposed methods for unfolding conditional control flow graphs containing loops into unconditional control flows. The objective of their approach is to produce a verifiable Petri Nets-based process model from the input process model. In [32], Choi *et al.* proposed a structural verification approach for cyclic workflow models by means of acyclic decomposition and reduction of loops. In their stepwise verification approach, nested loops are reduced in bottom-up order to become acyclic subgraphs. The difference between the approach presented in [32] and our approach is that, in our approach, crossing loops are also considered for unfolding.

Dumas *et al.*[25] also proposed a method for unraveling unstructured process models into structured models. In their method, Programming Structuring-Tree (PST) technique is used to decompose the model and the relationship between each component is determined. Then the model is restructured without altering the semantics. Although cyclic models cannot be handled directly in their method, a technique was proposed for transforming the cyclic models into GOTO-to-WHILE loop constructs. However, all these methods focus on transforming unstructured process models into structured ones and the resulting models may still contain structured loops.

The most relevant start-of-the-art methods based on node duplication are proposed in [7,8]. Yu *et al.*[7] proposed a method for unfolding a structured loop based on node duplication. However, their method only considers the simple structured loop pattern. In addition, the maximum number of iterations must be pre-defined in unfolded models. The method proposed in this paper is similar to Yu *et al.*'s approach in unfolding. However, in our approach, we consider nested and crossing loops as well as the probability distributions at the gateways to better reflect the actual business logic for repeating a set of activities. Eder and Pichler[8] proposed a method to split the probability iteration into sequences and conditions in order to analyze the temporal property. Their method copies a node incrementally into a set of execution paths respectively based on the iteration number. Their method is useful for analyzing the duration of the nodes within the iterative control structure. The main purpose of their approach is to analyze the temporal properties of the workflow. In contrast to their approach, in this paper, we propose two novel unravelling methods that can unfold not only simple structured loop patterns, but also nested and crossing loops.

3 Unravelling Iterative Control Structures in Business Processes

Three types of iterative control structures are considered in our approach. A structured-loop[3] is a control flow structure with only one entry and one exit point. Generally, a structured loop has either a pre-test or a post-test condition to determine the start and end of a loop. In our method, we mainly consider the post-test case. In Fig.1, the structured loop contains a loop entry point (join gateway $g1$) and a loop exit point (split gateway $g2$). The condition of looping can be an event trigger, a counter, or a probability condition. A crossing loop is a control flow structure comprising two or more loops and only some part of the loops in the structure are overlapped. An example of a simple crossing loop is depicted in Fig.2. A simple crossing loop contains only two loops which are crossing each other. In this example, one of the outgoing branches from gateway $g4$ connects to a join gateway $g2$ which is inside another loop. A complex crossing loop may contain more than two crossing loops. An example of a complex crossing loop is given in Fig.3. A nested loop contains loops which are embedded in other loops. In a simple nested-loop, there is only one structured-loop surrounded by another loop. Fig.4 is an example of a simple nested-loop. On the contrary, if one or more loops or a complex loop structure is surrounded by an outer loop, we call this structure a complex nested-loop. In Fig.5(a), there are two structured-loops surrounded by an outer loop. In Fig.5(b), the inner loop is a nested-loop. We made following assumptions for the proposed methods.

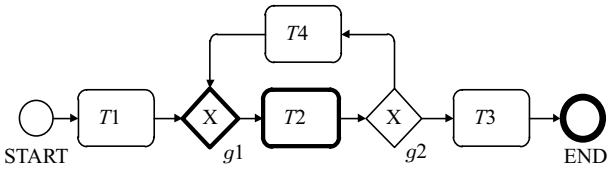


Fig.1. Example of a structured loop.

1) Loops structures with exclusive decisions are only considered in our method.

2) Only activities are allowed in back edge of the loop. A back edge is an edge that a workflow instance can take to return the start point of the loop.

3) Tasks and gateways in a loop are independent with entities or other loops.

According to assumption 1, we only consider the loop with exclusive gateways and loops with parallel gateways at decision point are not considered. Therefore, both the split and the join gateway for each loop are Exclusive OR gateway. This assumption is to ensure that only one outgoing branch will be chosen. In addition, we assume that each gateway should only have no more than two out-going (split) or in-coming (join) branches. However, it is possible to transform gateways with multiple branches into gateways containing only two out-going branches by creating new gateways and by assigning the branches to these gateways. Accord-

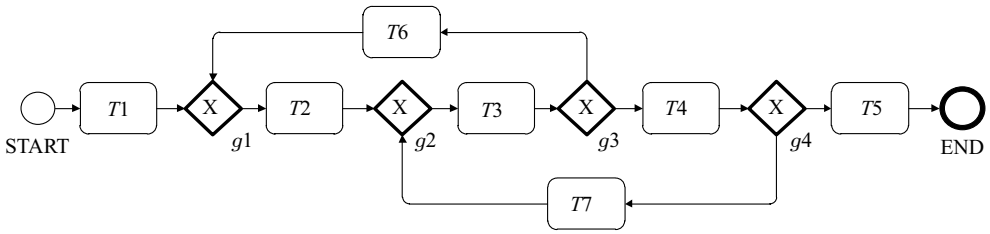


Fig.2. Simple crossing loop.

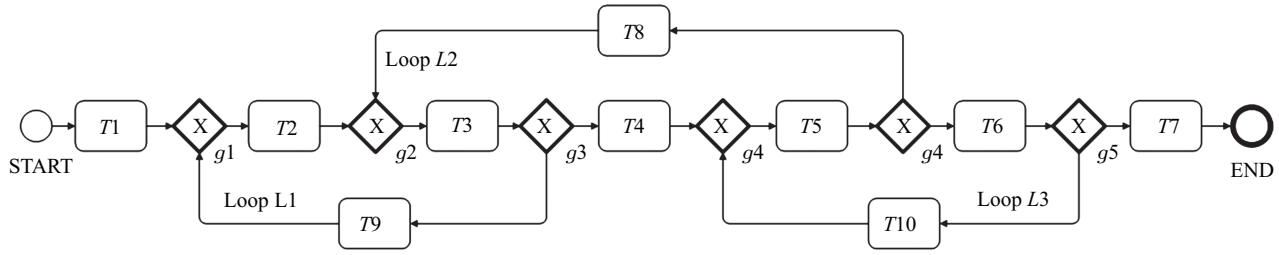


Fig.3. Complex crossing loops.

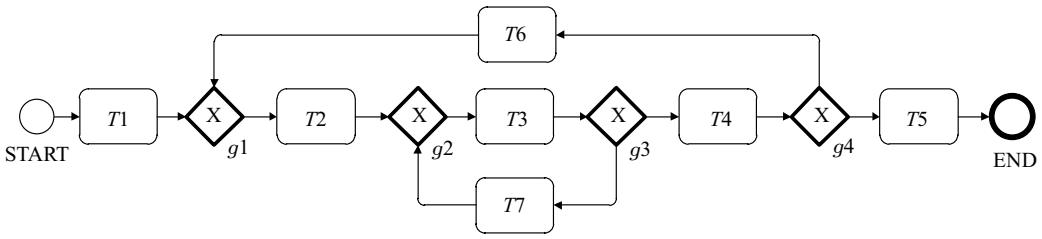


Fig.4. Simple nested-loop.

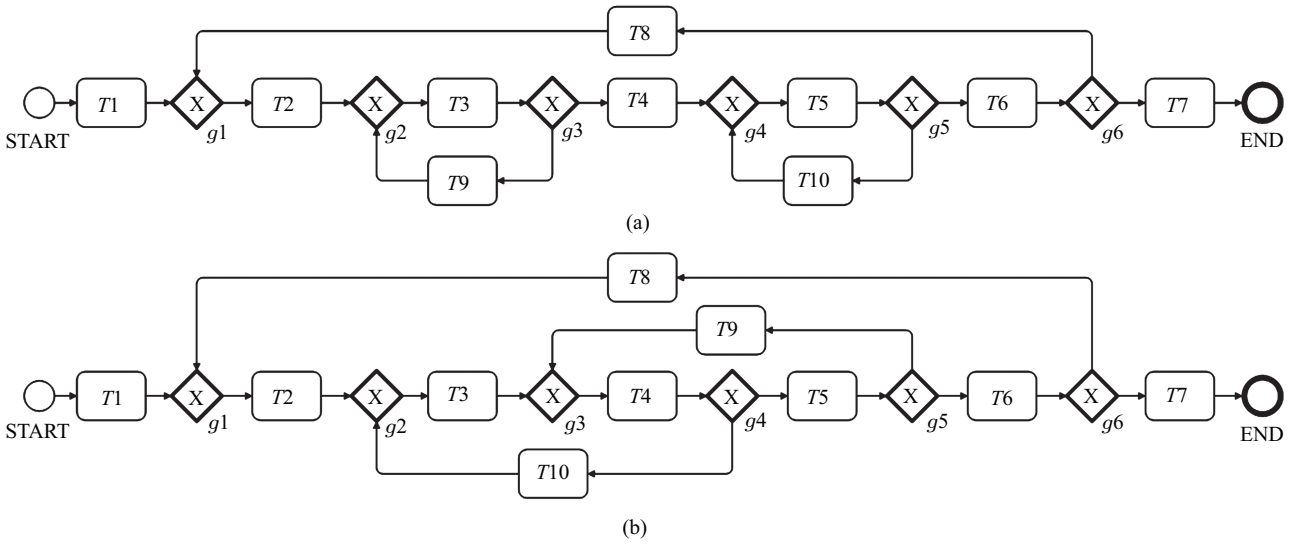


Fig.5. Examples of complex nested-loops.

ing to assumption 2, the back edge of the loop should only contain tasks or activities since any gateways in a back edge increase the difficulty in detecting and unravelling of the loops. According to assumption 3, the attributes of tasks and gateways among different loops are independent no matter whether they are overlapped or nested. Therefore, the conditions of looping back are independent.

In the proposed unravelling method, we consider business processes which are modelled in Business Process Modelling Notation (BPMN). The overview of the unravelling process is given in Fig.6. First, Loop Exploring traverses the imported BPMN model using the Depth-First Search (DFS) algorithm to find loops in the process model. Second, the loops from the previous step are parsed by Loop Parsing module. It then stores the loops information in a data structure called Loop Table. Loop parsing also uses a table to extract the probabilities assigned to the split gateways in another data structure called Probability Table. Third, Loop Unfolding module selects and unfolds a loop from the Loop Table. Loop Unfolding is repeatedly executed until there are no more loops in the process model.

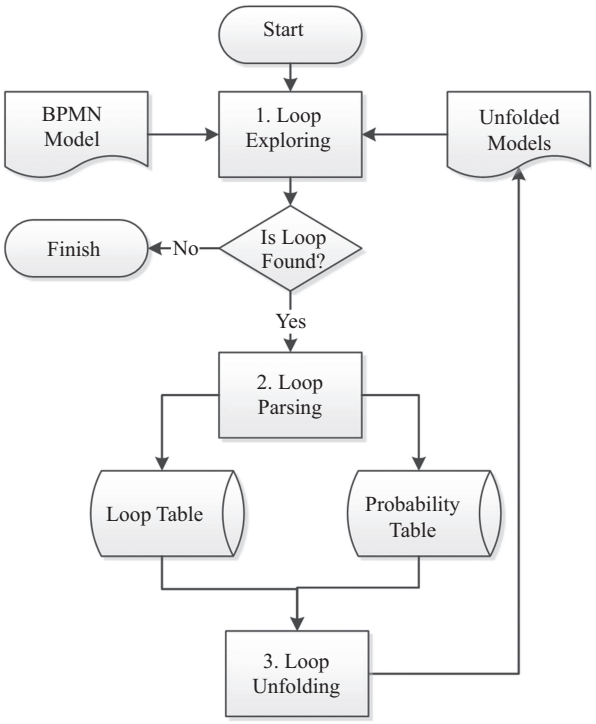


Fig.6. Overview of the unravelling process.

3.1 Loop Exploring

The Loop Exploring is a procedure to search all loops in a given BPMN model using a DFS algorithm. In the proposed method, all paths are traversed from the start event to the end event of a process model. A process model may contain more than one path when it contains a decision construct.

A workflow graph is a tuple of directed graph $WG = (N, T, G, PG, EG, E)$.

N is a finite set of nodes or vertices. $T \subset N$ is a finite set of tasks or activities. $G \subset N$ is a finite set of gateways. $PG \subset G$ is a finite set of parallel gateways. $EG \subset G$ is a finite of exclusive gateways. $G = PG \cup EG$. $N = T \cup G$. $E \subseteq N \times N$ is finite set of directed edges that are also the flow relation between two nodes. T^{S} is the start event of workflow and T^{E} is

the end event of the workflow.

The Loop Explorer module uses a set of paths E_{Path} in traversing. For the nodes containing one out-going route, the Loop Explorer adds them into the current path. For any nodes containing more than one out-going route, it clones the current path and inserts the new path for later traversing. The exploring is stopped when an end event or a visited join gateway is reached. A join gateway is used in discovering potential loops at the later stage.

An example of exploration is illustrated in Fig.7. The exploration of all paths is ended when traversing each of them reaches the end event. The set of paths found by Loop Explorer is as follows.

$$
\begin{aligned}
&E_{\text{Path}} = \{e_1, e_2, e_3\},\\
&e_1 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, T4), (T4, g3), \\ (g3, T3), (T3, g4), (g4, \text{END}) \end{array} \right\},\\
&e_2 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, g2), (g2, T2), \\ (T2, g3), (g3, T3) \end{array} \right\}\\
&\quad \cup \{(T3, g4), (g4, \text{END})\},\\
&e_3 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, g2), (g2, g4), \\ (g4, \text{END}) \end{array} \right\},\\
&e_1 \cup e_2 \cup e_3 \subseteq E.
\end{aligned}
$$

Identifying Loop. The workflow model is likely to contain a loop when a previously visited join gateway is reached during the traversal. To identify the existence of a loop in a given path, we define the function $loop\,(e, x)$ in (1). Parameter e is an exploring path and x is the join gateway which was previously visited.

$$loop(e, x) : x \in \{m | (n, m) \in e\}, \quad e \subset E_{\text{Path}}, x \in N. \tag{1}$$

In (1), n and m represent the start and the end nodes of an edge in the exploring path e respectively. The function $loop(e, x)$ checks all paths from the set of paths (E_{path}) found by the Loop Explorer module. The function returns true when a previously visited node x is equal to the target node m of one of the edges in a path. The following example in Fig.8 is used to illustrate the loop identification process.

In the following paragraphs, we illustrate a step-by-step example of identifying loops based on an SESE process model depicted in Fig.8. In this example, we assume that the path exploring module has already reached gateway $g2$ and recorded two paths $e1$ and $e2$.

$$
\begin{aligned}
&E_{\text{Path}} = \{e_1, e_2\},\\
&e_1 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, T2), \\ (T2, g2), (g2, T4) \end{array} \right\},\\
&e_2 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, T2), \\ (T2, g2), (g2, T3) \end{array} \right\}.
\end{aligned}
$$

Next, path e_1 is used to identify a loop. According to (1), function $loop\,(e_1, g1) = \text{true}$ because $g1$ has been already visited. Path e_2 does not contain any loop.

$$
\begin{aligned}
&E_{\text{Path}} = \{e_1, e_2\},\\
&e_1 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, T2), \\ (T2, g2), (g2, T4), (T4, g1) \end{array} \right\},\\
&e_2 = \left\{ \begin{array}{c} (\text{START}, T1), (T1, g1), (g1, T2), \\ (T2, g2), (g2, T3), (T3, END) \end{array} \right\}.
\end{aligned}
$$

Any loops found during the exploration are stored in a data structure called Loop Table which can be represented as an eight tuple $LT = (ID,\ EG^J,\ EG^S,\ E^{FS},\ E^{FE}, E^{BS}, E^{BE}, G^F)$. ID is the unique identifier of

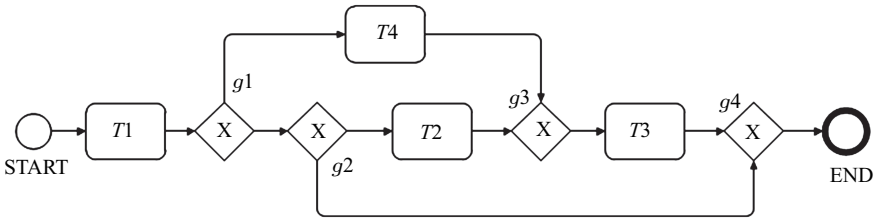


Fig.7. Example of loop exploration.

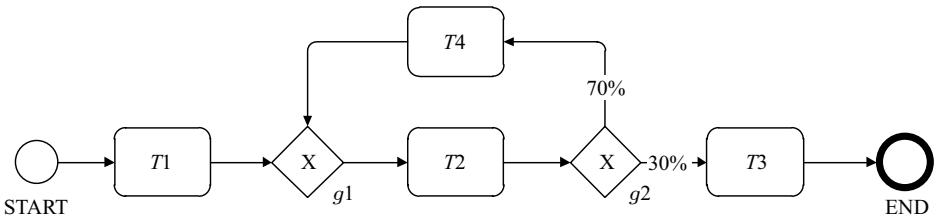


Fig.8. Example of loop identification.

the loop. $EG^J \in EG$ is the loop entry point of a loop. $EG^S \in EG$ is the loop exit point of a loop. $E^{FS} \subset E$ is the edge at the start of a forward flow. $E^{FE} \subset E$ is the edge at the end of a forward flow. $E^{BS} \subset E$ is the edge at the start of a backward flow. $E^{BE} \subset E$ is the edge at the end of a backward flow. $G^F \subset G$ is the set of gateways in a forward flow.

The loop entry point is an Exclusive OR join gateway leading to a loop region. The loop exit point is an Exclusive OR split gateway. The forward flow is the flow from the loop entry point to the loop exit point. We can define the Loop Table for the example given in Fig.8 as follows.

$$LT\,(1) = \begin{pmatrix} L1, g1, g2, (g1, T2)\,, (T2, g2)\,, \\ (g2, T4)\,, (T4, g1)\,, \{g1, g2\} \end{pmatrix}.$$

3.2 Loop Parsing

Loop Parsing is a procedure for removing all misrecognized loops in the Loop Table. It is also used to identify the relation among nested-loops, crossing loops and structured loops. The overview of the loop parsing process is illustrated in Fig.9.

Duplicate loops are the loops which are recognized more than once during loop parsing process. Individual loops are used to denote any remaining loop after parsing the process model. An individual loop is a structured loop without any nesting or crossing. The reason for removing duplicate loops is that in some situations, a loop may be recognized twice in two paths which are created by a split gateway. There are three situations in which a loop can be identified more than once.

Case 1. When a loop has two or more entry points (XOR join gateways) and one exit point (XOR split gateway).

In Fig.10, there are two paths entering the loop from gateways $g2$ and $g3$. Therefore, we obtain two loops $L1$ and $L2$ as the result of Loop Exploring.

$$LT\,(1) = \begin{pmatrix} L1, g2, g4, (g2, T2)\,, (T3, g4)\,, \\ (g4, g2)\,, (g4, g2)\,, \{g2, g3, g4\} \end{pmatrix},$$

$$LT\,(2) = \begin{pmatrix} L2, g3, g4, (g3, T3)\,, (T3, g4)\,, \\ (g4, g2)\,, (T2, g3)\,, \{g3, g4\} \end{pmatrix}.$$

Let l and m be two loops in a Loop Table and they are considered duplicate loops if the following function $dup1\,(l, m)$ returns true.

$$dup1\,(l, m) = \begin{cases} \text{true, if } \left(l.EG^S = m.EG^S\right) \text{ AND} \\ \qquad\qquad \left(l.G^F = (m.G^F \cup l.EG^j)\right), \\ \text{false, otherwise.} \end{cases} \quad (2)$$

The function $dup1\,(l, m)$ returns true when 1) the loop exit points of both loops (l and m) are the same and 2) the set of gateways from loop l in the forward flow is equal to the union of the set of gateways from loop m in the forward flow and the loop entry point of loop l. In the preceding example, we can observe that $dup1\,(LT\,(1)\,, LT\,(2)) = \text{true}$. Therefore, $L1$ will be selected for unfolding since $L2$ contains a gateway in the forward flow.

Case 2. In this case, a loop has more than one entry point (XOR join gateway) and one exit point (XOR split gateway). If a loop contains multiple exit points, this loop will be recognized more than once during loop exploring. The entry and the exit points of each loop recorded will also be different.

In Fig.11, there are two paths entering the loop from gateways $g2$ and $g4$. Therefore, we obtain two loops $L1$ and $L2$ from the Loop Exploring process.

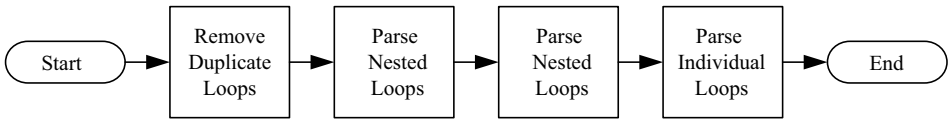


Fig.9. Overview of the loop parsing process.

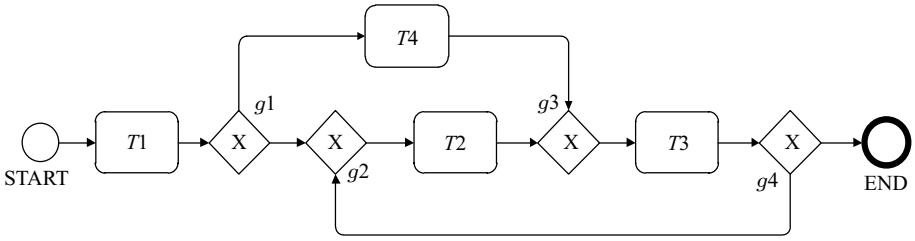


Fig.10. Example workflow model for case 1.

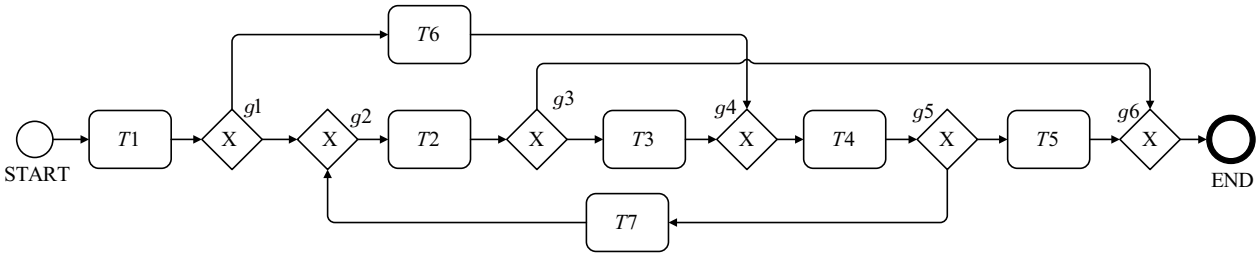


Fig.11. Parsing a duplicate loop in case 2.

$$LT\,(1) = \begin{pmatrix} L1, g2, g5, (g2, T2)\,, (T4, g5)\,, \\ (g5, T7)\,, (T7, g2)\,, \{g2, g3, g4, g5\} \end{pmatrix}.$$

$$LT\,(2) = \begin{pmatrix} L2, g4, g3, (g4, T4)\,, (T2, g3)\,, \\ (g3, T3)\,, (T3, g4)\,, \{g3, g4, g5, g2\} \end{pmatrix}.$$

Let l and m be the two loops in the Loop Table and they are considered duplicate loops if the following function $dup2\,(l, m)$ returns true.

$$dup2\,(l, m) = \begin{cases} \text{true, if } (l.EG^S \neq m.EG^S) \text{ AND} \\ \qquad (l.G^F = m.G^F), \\ \text{false, otherwise.} \end{cases} \tag{3}$$

The function $dup2\,(l, m)$ returns true when 1) the set of gateways from both loops (l and m) in the forward flow are the same and 2) the loop exit points of both loops (l and m) are different. In this example, we can observe that $dup2\,(LT\,(1)\,, LT\,(2)) = dup2\,(LT\,(2)\,, LT\,(1)) = \text{true}$. In this case, we select only one of the loops to unfold.

Case 3. Duplicate loops will be recorded when there are two or more paths leading to the loop. For the example given in Fig.12, two loops will be recorded in the Loop Table and one of the loops will be removed after parsing.

$$LT\,(1) = \begin{pmatrix} L1, g3, g4, (g3, T3)\,, (T3, g4)\,, \\ (g4, g3)\,, (g4, g3)\,, \{g3, g4\} \end{pmatrix},$$

$$LT\,(2) = \begin{pmatrix} L2, g3, g4, (g3, T3)\,, (T3, g4)\,, \\ (g4, g3)\,, (g4, g3)\,, \{g3, g4\} \end{pmatrix}.$$

Parsing Nested Loops. A hierarchical tree structure is used to record the relationship of loops in a nested loop. The root node of the tree is the outmost loop and all leaf nodes are the inner loops. A loop is not a nested loop if there are no edges connecting to the nodes in the tree. We define function $CHILD(nl)$ for constructing the hierarchical tree. Function $CHILD(nl)$ returns the set of immediate child nodes from the sub-tree at node nl. To identify the relationship among loops in a nested loop, the proposed method will compare all loops in the Loop Table to build the hierarchical tree. In total, $S_n = k(k-1)/2$ rounds of comparison are needed where k is the total number of loops in the Loop Table. The algorithm for parsing the loops and building the hierarchical tree is given in Algorithm 1.

An example process model containing seven nested loops is depicted in Fig.13. The relationship structure given in Fig.14 can be obtained from applying Algorithm 1 to the process model from Fig.13. According to the computed tree, $L4$ is the outermost loop. $L3$, $L5$ and $L7$ are the inmost loops and $L6$ does not have any relationship to other loops.

Parsing Cross Loops. The relationship of loops within a crossing loop can be defined as a directed graph. An edge from the graph denotes the crossing relationship between two loops. The node at the target node of a directed edge drawn in a solid line is the left loop and the node at the target node of a directed edge drawn in a dashed line is the right loop. We define two functions for constructing the relationship graph.

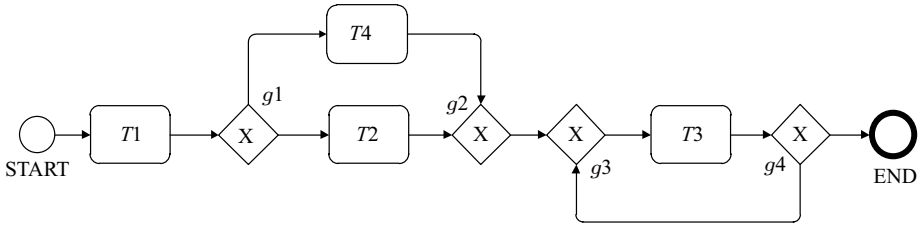


Fig.12. Example process model containing two or more paths leading to the loop.

Function $LEFT(cl)$ returns the left loops of node cl. Function $RIGHT(cl)$ returns the right loops of node cl. To build the relationship graph among crossing loops, Algorithm 2 performs pairwise comparisons among all the loops from the Loop Table. A total of $k(k-1)/2$ comparisons will be performed and k is the total number of loops in the Loop Table.

An example of a process model containing seven crossing loops is depicted in Fig.15. The results of applying Algorithm 2 to the process model are given in Fig.16. In this example, $L4$ and $L7$ are the leftmost loops. $L1$, $L3$ and $L5$ are the rightmost loops. $L5$, $L6$ and $L7$ are crossing with each other's and they are part of a non-chained complex crossing loop structure.

After parsing the process model for nested loops and crossing loops, remaining iterations are labelled as individual loops.

3.3 Loop Unfolding

Unfolding is the core component of the proposed method. Loop unfolding includes the algorithms for calculating probability conditions and step-by-step unfolding of each loop.

3.3.1 Probability Computation

In real-world business processes, a loop should not be allowed to execute indefinitely. However, it is difficult to accurately predict the exact number of iterations during run-time [16]. The number of iterations can be estimated from the past execution history, experts' opinion, and relevant statistics of the business process. In this paper, we propose a probability-based method for estimating the maximum number of iterations. Specifically, we use a pre-defined tolerance value to determine the upper limit of iterations. Whenever a loop is iterated, an accumulated probability assigned to the split gateway is reduced. When the accumulated probability is smaller than the tolerance, the loop will be stopped from iterating. Intuitively, a loop is less likely to be executed again when the accumulated probability is low.

Algorithm 1. Function *BuildNLTree*( ) for Building the Relationship Tree of a Nested Loop

Input: Loop Table L which is recorded in Loop Explorer
Output: Nested-Loop Relation Tree NL

```
1:  Let sizeofLT be the size of L
2:  for i = 1 ... to (sizeofLT − 1)
3:      Let m as the loop at L[i]
4:      for j = 2 ... to sizeofLT do
5:          Let n be the loop at L[i + 1]
6:          if m's loop exit point is not equal to n's loop exit point and
7:              m's loop entry point is not equal to n's loop entry point and
8:              m's loop exit point and m's loop entry point are inside n then
9:              % Two loops are Nested-Loop; m is the inner loop and n is the outer loop
10:             if sub-function SuccExist(CHILD(m), m) returns false then
11:                 Insert m into CHILD(n)
12:                 Call function NOT_ROOT(m) to set m as not the root of the tree
13:             end if
14:         end if
15:     end for
16: end for
17: return NL
18: sub-function SuccExist(s, target):
19: begin
20:     if s is empty then
21:         return false
22:     else if s contains target then
23:         return true
24:     else
25:         for elem in s
26:             if SuccExist(CHILD(elem), target) returns true then
27:                 return true
28:             end if
29:         end for
30:     end if
31: end
```

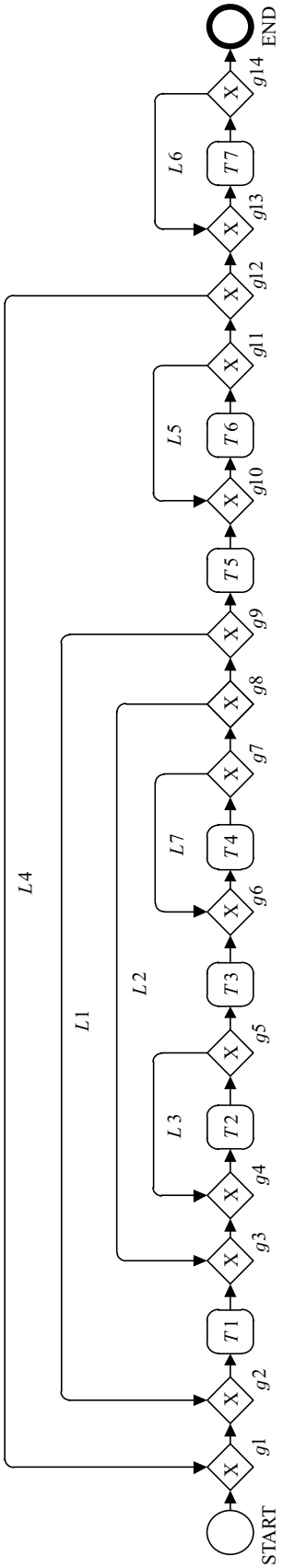


Fig.13. Example process model for building the relationship structure of nested loops.

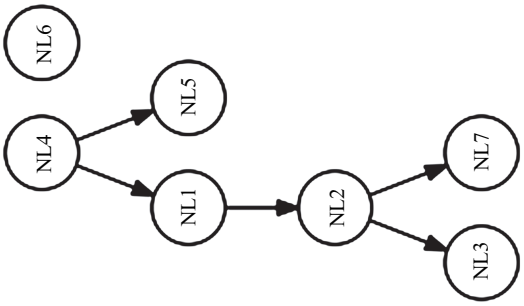


Fig.14. Nested loop relationship structure for the process model in Fig.13.

Algorithm 2. Function *BuildCLGraph*( ) for Building Relation Graph of Crossing-Loop

Input: Loop Table L which is recorded by Loop Explorer
Output: Crossing-Loop Relation Graph CL

```
1:  Let sizeofLT be the size of L
2:  for i = 1 ... to (sizeofLT − 1)
3:      Let p be the loop at L[i]
4:      for j = 2 ... to sizeofLT
5:          Let q be the loop at L[i + 1]
6:          if p's loop exit point is not equal to q's loop exit point and
7:              p's loop entry point is not equal to q's loop entry point then
8:              if p's loop exit point is inside q's forward path and
9:                  q's loop entry point is inside p's forward path then
10:                 Insert p into LEFT (q)
11:                 Insert q into RIGHT (p)
12:             else if p's loop entry point is inside q's forward path and
13:                 q's loop exit point is inside p's forward path then
14:                 Insert p into RIGHT (q)
15:                 Insert q into LEFT (p)
16:             end if
17:         end if
18:     end for
19: end for
20: return CL
```

An example process model for applying Algorithm 3 is given in Fig.17. In this example, the probability of looping back p is assumed to be 0.3 (30%). After the first iteration, the accumulated probability becomes $accu = 1 \times 0.3 = 0.3$. If we assume that the tolerance value is $tol = 0.003$, then the maximum number of iterations max is 4 because $accu = (0.3)^5 = 0.002\,43 < tol$.

3.3.2 Flow Copying

During the unfolding process, we may need to duplicate the forward and backward flows inside the loop depending on the maximum number of iterations calculated. The function *CopyFlow* in Algorithm 4 is used to duplicate the forward and backward flows. The algorithm simply copies the nodes and connects them accordingly for unfolding.

An example of flow copying is illustrated in Fig.18. In this example, the workflow model contains a structured loop. The forward path is from task $T2$ to $T4$ and the backward path contains only one task $T5$. For the purpose of discussion, we assume that the maximum number of iterations max is equal to 3. Therefore, the nodes inside the flow will be duplicated three times. Next, we apply the function $CopyFlow()$ to duplicate the loop's forward path. In Fig.19, the first node $T2$ is copied three times. To distinguish the original node from its clones, we label the duplicated node with the current round number of unfolding and a counter. Since the process is at the first round of unfolding, we label the duplicated nodes as $T2\_1(1)$, $T2\_1\,(2)$ and $T2\_1(3)$. For the complex workflow models containing more than one loop, flow copying may be performed for several rounds. The resulting process model after the first round of copying nodes is depicted in Fig.19.

Next the algorithm continues to copy all the nodes in the forward flow. The resulting process model is depicted in Fig.20.

After the copying of nodes is completed, the algorithm connects the copied nodes. The connections (directed edges) between the copied nodes are based on the original process model. In Fig.21, we can observe that the nodes are connected by directed edges. Therefore, the algorithm returns following three flows.

$$\{T2_{1(1)}, T3_{1(1)}, T4_{1(1)}\},$$
$$\{T2\_1(2), T3\_1(2), T4\_1(2)\},$$
$$\{T2\_1(3), T3\_1(3), T4\_1(3)\}.$$

3.3.3 Unfolding Structured Loops

In this subsection, we provide the algorithm of unfolding a structured loop. The function *UnfoldStLoop* given in Algorithm 5 uses functions *CalMaxIT* and *CopyFlow* which are defined in Subsection 3.3.2.

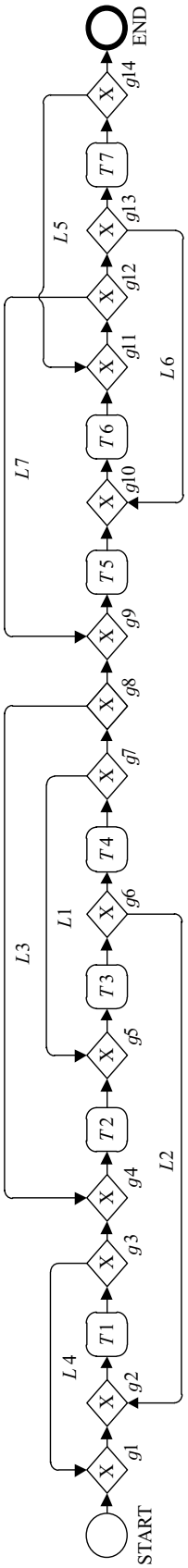


Fig.15. Example process model containing crossing loops.

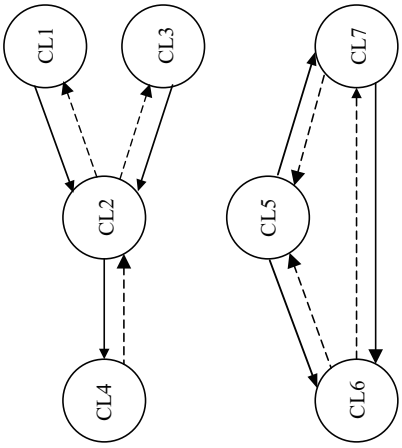


Fig.16. Crossing loop relationship structure for the process model in Fig.15.

Algorithm 3. Function $CalMaxIT()$ for Computing the Maximum Number of Iterations

Input: tolerance value *tol* for the upper limit of iterations
loop l for computing its maximum number of iterations
Output: the maximum number of iterations of the loop
1: Retrieve p as the probability value of looping back for loop l from the probability table
2: Set *max* as the maximum number of iterations
3: Set *accu* as the accumulated probability value
4: **do**
5: *max* increased by 1
6: *accu* multiplied by p
7: **while** *accu* is less than *tol*
8: **return** *max*

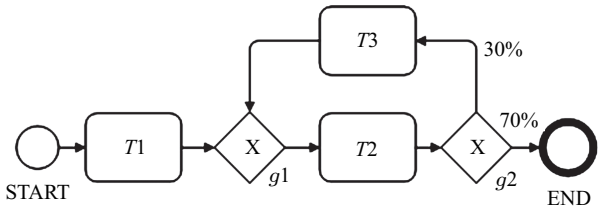


Fig.17. Example process model annotated with probability for looping back.

Unfolding structured loops from two example process models is illustrated in Fig.22. In both process models in Fig.22, gateway $g1$ is identified as the loop entry point EG^J. In Fig.23(a), the algorithm first disconnects the edges $(T1, g1)$, $(T4, g1)$ and $(g1, T2)$. Next, it connects the edge $(T1, T2)$. In Fig.23(b), the algorithm disconnects the edges $(T1, g1)$ and $(g1, T2)$. Next, it connects the edge $(T1, T2)$. The results of the above steps are shown in Fig.23.

Next, the algorithm inserts the loop entry point $g1$ right after $g2$. For the example model given in Fig.23(a), the algorithm disconnects the edge $(g2, END)$. For the model in Fig.23(b), the algorithm disconnects the edge $(g2, END)$. After that, the algorithm connects the edge $(g2, g1)$ and $(g1, END)$ in both models. The results of these steps are shown in Fig.24.

Unfolding a Structured Loop Based on Zero-One Principle. When unfolding based on Zero-One principle, a loop is permitted to be executed (iterated) at most once. Hence, the maximum number of iterations is set as 1. The process of unfolding structured loops from two example process models given in Fig.22 based on Zero-One principle is illustrated in Fig.25 and Fig.26.

Firstly, according to Fig.25(a), the task $T2$ is copied once and SF is set to $S\{T2\_1(1)\}$. Next, tasks $T2$ and $T3$ are copied once and SF is set to $\{T2\_1(1), T3\_1(1)\}$ (see Fig.25(b)).

For the model given in Fig.25(a), the algorithm then connects task $T4$ and task $T2\_1(1)$. Finally, it connects $T2\_1(1)$ and the loop entry point $g1$. The final result is shown in Fig.26(a).

Algorithm 4. Function $CopyFlow[ERR : md : MbegChr = 0x0028, MendChr = 0x0029, nParams = 0]$ for Copying a Flow

Input: the graph WG of workflow model; the flow f which will be copied; the number of iterations *num* will be copied
Output: the set of flows S copied from f
1: Create a set of flows S
2: **for** node n contained in f
3: Copy n in *num* times
4: **end for**
5: Connect copied nodes and push the flows into S
6: **return** S

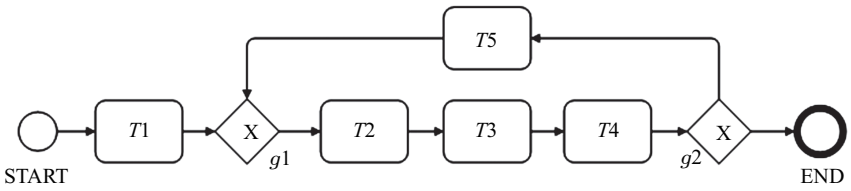


Fig.18. Example process model for flow copying.

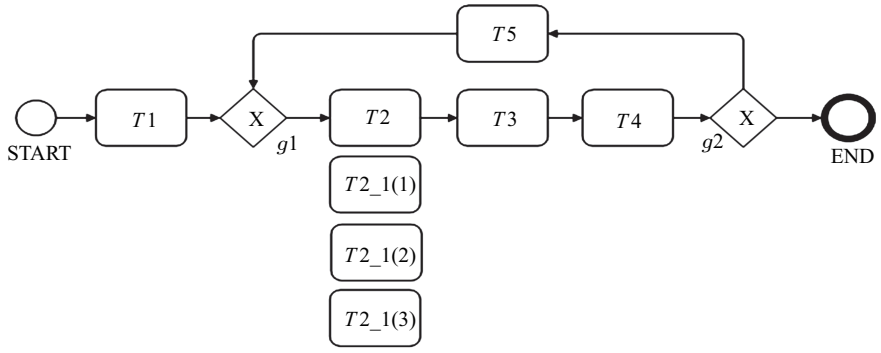


Fig.19. Process model after the first node in the forward flow is copied in the first round.

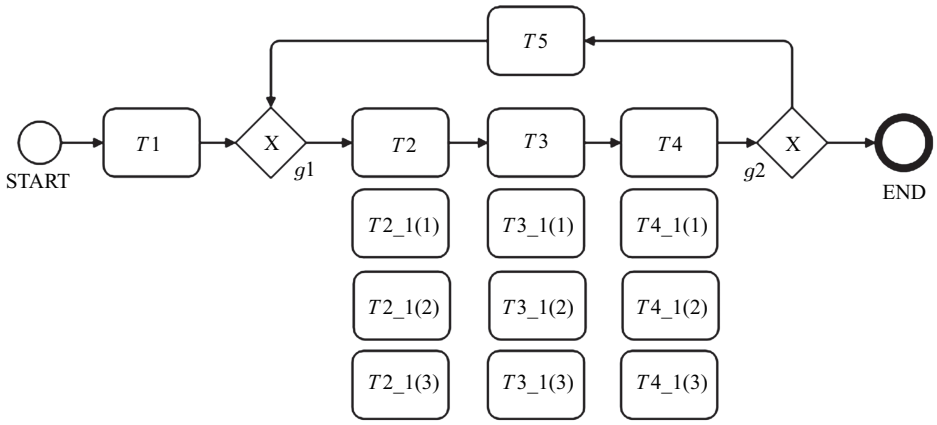


Fig.20. Copying of all nodes in the forward flow.

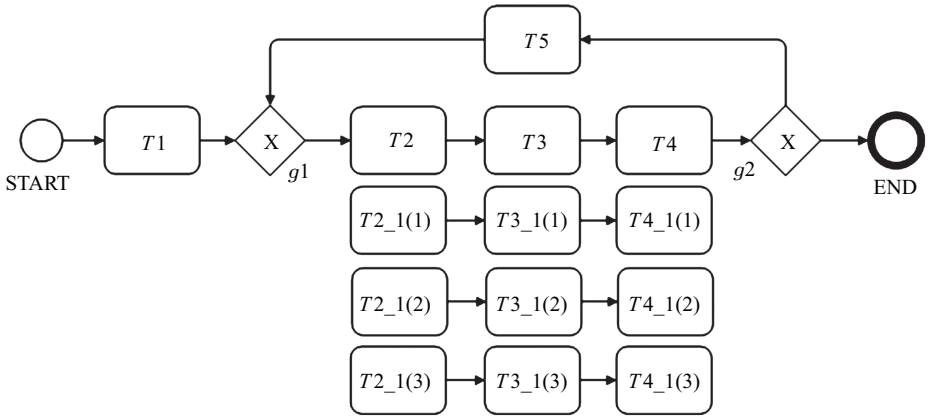


Fig.21. The last stage of flow copying.

For the model given in Fig.25(b), the algorithm connects loop exit point $g2$ and task $T2\_1(1)$. Next, it connects task $T3\_1(1)$ and loop entry point $g1$. The final results of unfolding are shown in Fig.26(b). We can observe that the process models after unfolding contain no more structured loops.

Unfolding a Structured Loop Based on Probability-Based Principle. In this subsection, we propose a novel method to unfold a structured loop based on the probability. To illustrate our ideas, we will use the same running examples given in Figs.22(a) and 22(b). For the purpose of discussion, we assume that the maximum number of iterations max returned from function $CalMaxIT()$ is 3. The step-by-step unfolding process of the example model given in Fig.22(a) is illustrated in Fig.27. The step-by-step unfolding process of the example model given in Fig.22(b) is illustrated in Fig.28.

Algorithm 5. Function $UnfoldStLoop(\,)$ for Unfolding a Structured Loop

Input: the graph WG of workflow model; the structured loop $strLoop$ which will be unfolded

Output: updated WG after unfolding of a structured loop

1: Create a set of gateways SG
2: Create two sets of flows SB and SF
3: Let $fFlow$ be the loop's forward path which begins at the successor node of the loop entry point and ends at the predecessor node of the loop exit point
4: Let $bFlow$ be the loop's backward path which begins at the successor node of the loop exit point and ends at the predecessor node of the loop entry point
5: Disconnect all incoming and outgoing edges of the loop entry point
6: Insert the disconnected loop entry point directly after the loop exit point
7: Call function $CalMaxIT\,(strLoop)$ to calculate the maximum number of iterations max
8: Copy loop exit point max times and insert it into SG
9: Call function $CopyFlow\,(WG, fFlow, max)$ to copy $fFlow$ as SF
10: **if** $backFlow$ is not empty **then**
11: Call function $CopyFlow\,(WG, bFlow, max-1)$ to copy $bFlow$ as SB
12: Set $pointer$ as the last node in flow $backFlow$
13: **else**
14: Set $pointer$ as the loop exit point
15: **end if**
16: **for** $counter = 1 \ldots$ to $max - 1$
17: Connect $pointer$ and the first node in flow $SF\,[counter]$
18: Connect the last node in flow $SF\,[counter]$ and gateway $SG\,[counter]$
19: Connect gateway $SG\,[counter]$ and loop entry point
20: **if** $backFlow$ is not empty **then**
21: Connect gateway $SG\,[counter]$ and the first node in flow $SB\,[counter]$
22: Set $pointer$ as the last node in flow $SB\,[counter]$
23: **else**
24: Set $pointer$ as the gateway in $SG\,[counter]$
25: **end if**
26: **end for**
27: Connect $pointer$ and the first node in flow $SF\,[max]$
28: Connect the last node in flow $SF\,[max]$ and loop entry point
29: **return** WG

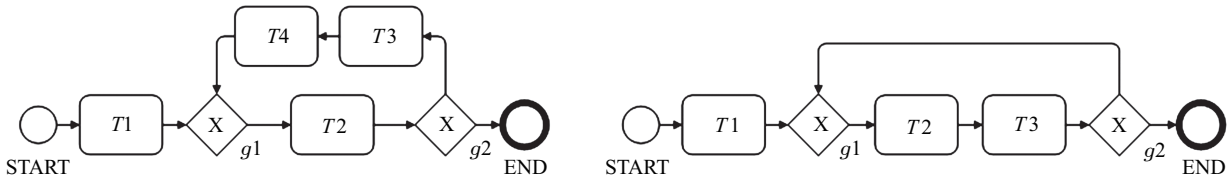


Fig.22. Unfolding structured loops from two example process models.

3.3.4 Unfolding Nested Loops

Recall that nested loops contain structured loops as their basic components. Therefore, we can reuse the function $UnfoldStLoop(\,)$ which is previously defined in Algorithm 5 to unfold nested loops. The algorithm for unfolding a nested loop is described in Algorithm 6. We define function $ROOT(nl)$ which returns true if the loop nl is the root of the tree.

The process of unfolding a nested loop can be illustrated with the process model given in Fig.29. In this example, there are two nested loops $L1$ and $L2$.

$$LT\,(1) = \begin{pmatrix} L1, g1, g4, (g1, g2)\,, (T3, g4)\,, \\ (g4, g1)\,, (g4, g1)\,, \{g1, g2, g3, g4\} \end{pmatrix},$$

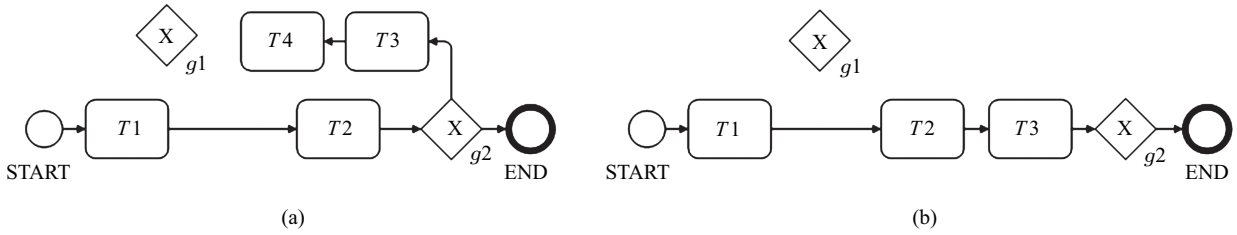


Fig.23. Disconnecting incoming and outgoing edges at the loop entry point $g1$.

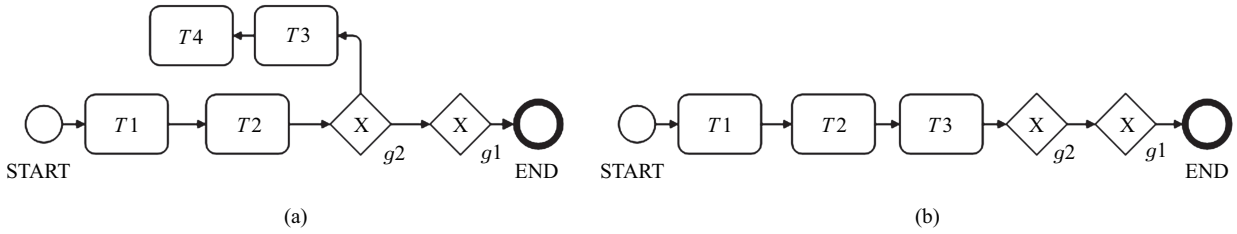


Fig.24. Process models after inserting a loop entry point.

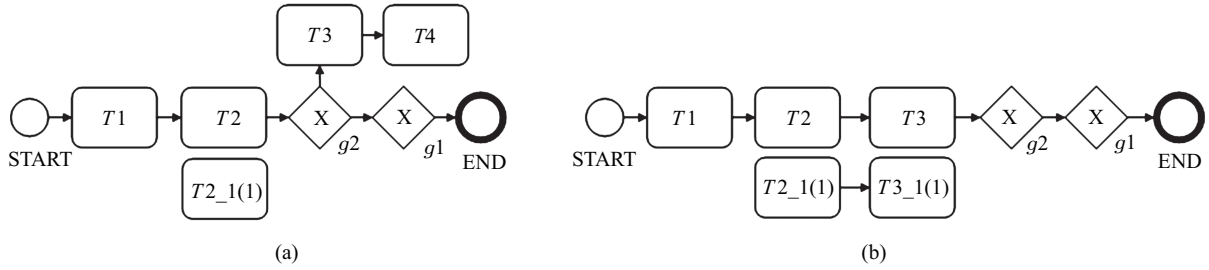


Fig.25. Intermediate results of flow copying during unfolding structured loops based on Zero-One principle.

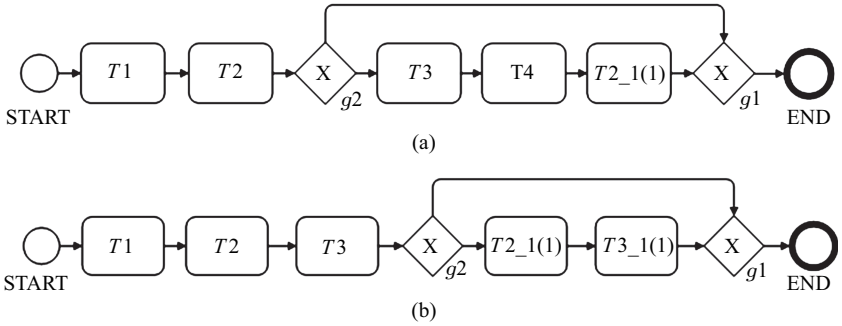


Fig.26. Final results of unfolding structured loops based on Zero-One principle.

$$LT\,(2) = \begin{pmatrix} L2, g2, g3, (g2, T2)\,, \\ (T2, g3)\,, (g3, g2)\,, (g3, g2)\,, \{g2, g3\} \end{pmatrix}.$$

Unfolding a Nested Loop Based on Zero-One Principle. Similar to structured-loops, we first explain the unfolding of nested loop in Zero-One Unfolding. The results of unfolding from inner to outer direction are depicted in Fig.30. First, the inner loop is unfolded in Fig.30(a). Next, the outer loop is unfolded, and the results are shown in Fig.30(b).

Unfolding a Nested Loop Based on Probability-Based Principle. In this subsection, we use the same example given in Fig.29 to explain the unfolding process. We assume that the tolerance value is $tol = 0.1$ and the probability values of looing back for loop $L1$ and $L2$ are $p_{L1} = 0.4$ (40%) and $p_{L2} = 0.3$ (30%). The process of unfolding from inner to outer direction is illustrated in Fig.31. The intermediate result of unfolding the inner loop based on Algorithm 5 is shown in Fig.31(b). The progress of unfolding the remaining outer loop is shown in Fig.31(c). The completely unfolded model is given in Fig.31(d).

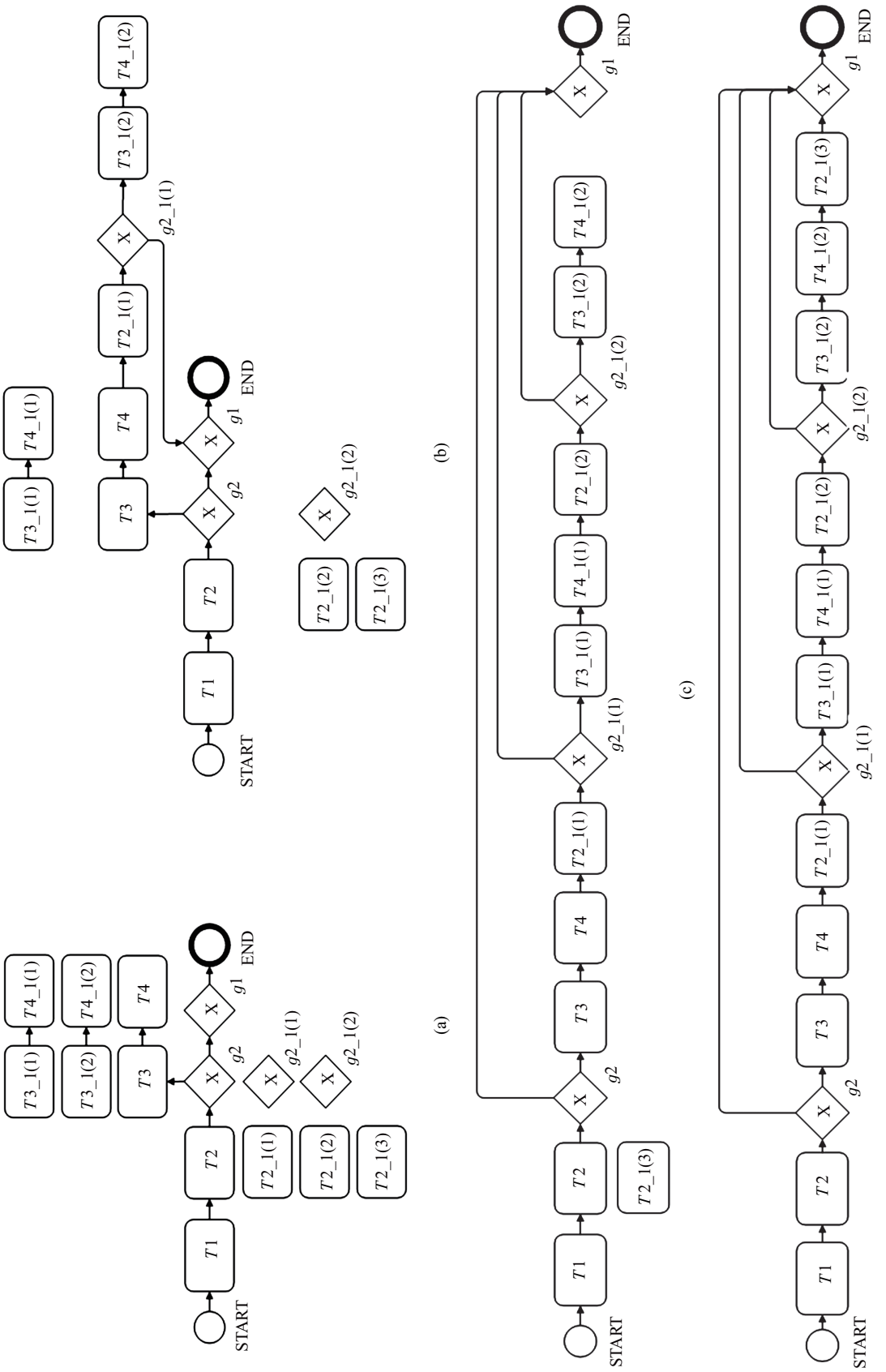


Fig.27. Step-by-step unfolding of a structured loop based on probability-based principle (based on the process model in Fig.22(a)).

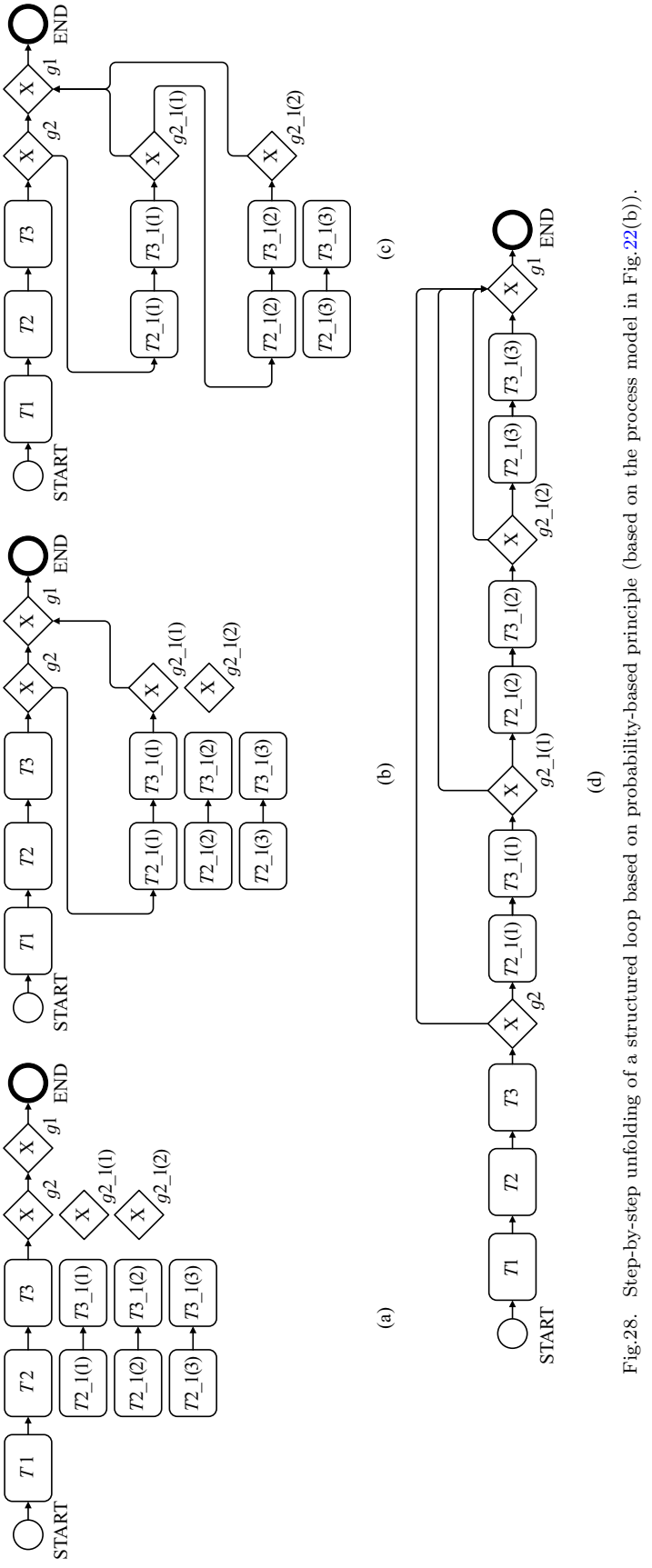


Fig.28. Step-by-step unfolding of a structured loop based on probability-based principle (based on the process model in Fig.22(b)).

Algorithm 6. Function $UnfoldNeLoop(\,)$ for Unfolding Nested Loop

Input: the graph WG of workflow model; the Nested-Loops Tree NL; loop Table LT
Output: WG which is the model after unfolded Nested Loop

```
1:  for loop rootLoop in L
2:      if ROOT (rootLoop) is true then
3:          Call sub-function UnfoldNeInnerLoop (WG, rootLoop, NL) to unfold loops
4:      end if
5:  end for
6:  return WG
7:  sub-function UnfoldNeInnerLoop(WG, loopNL)
8:  begin
9:      for loop inner in CHILD (loop)
10:         Call function UnfoldNeInnerLoop (WG, inner, NL)
11:     end for
12:     Call function UnfoldStLoop(WG, loop) to unfold loop
13: end
```

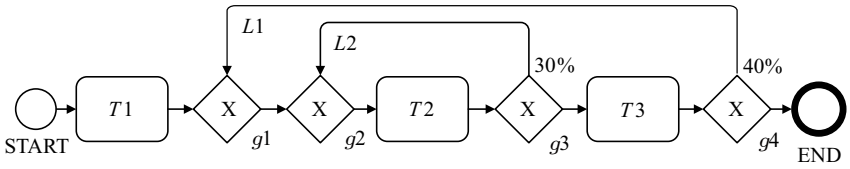


Fig.29. Example process model containing a nested loop.

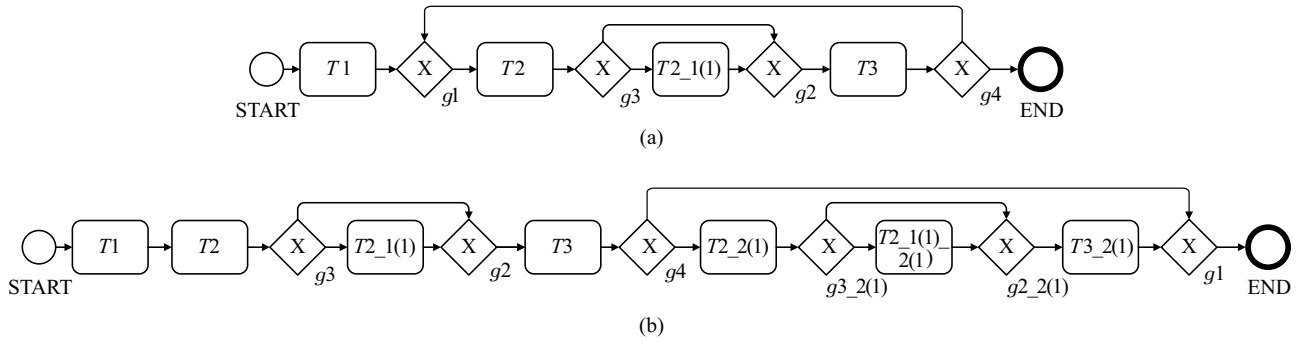


Fig.30. Unfolding from inner to outer direction based on Zero-One principle.

3.3.5 *Unfolding Crossing Loops*

In this subsection, we provide the algorithm for unfolding a crossing loop.

In this algorithm, the leftmost loop in a crossing loop will be unfolded first. The example process model for illustrating the unfolding process is given in Fig.32.

In Fig.32, there are two loops and they are recorded in loop tables as follows.

$$LT\,(1) = \left(\begin{array}{c} L1, g1, g3, (g1, T1)\,, (T2, g3)\,, (g3, g1)\,, \\ (g3, g1)\,, \{g1, g2, g3\} \end{array} \right),$$

$$LT\,(2) = \left(\begin{array}{c} L2, g2, g4, (g2, T2)\,, (T3, g4)\,, (g4, g2)\,, \\ (g4, g2)\,, \{g2, g3, g4\} \end{array} \right).$$

Unfolding of a Crossing Loop Based on Zero-One Principle. We use Algorithm 7 to unfold the crossing loop given in Fig.32. First, the leftmost loop $L1$ is unfolded using the function $UnfoldStLoop(\,)$ in Algorithm 5. The step-by-step unfolding process is illustrated in Fig.33. We can notice that in Fig.33(b), the loop exit point $g4$ of loop $L2$ still contains three outgoing branches. In Fig.33(c), we create a copy gateway called $g4\_C$ from $g4$. Next, the algorithm inserts $g4\_C$ directly after the loop entry point $g2$ of loop $L2$. Next, the algorithm disconnects the edge $(g4, g2\_1(1))$ and creates a new edge $(g4\_C, g2\_1(1))$. After that, $L2$ is unfolded using Algorithm 5. After $L2$ is unfolded, there are no more remaining loops in the process model. The final result of unfolding is depicted in Fig.33(d).

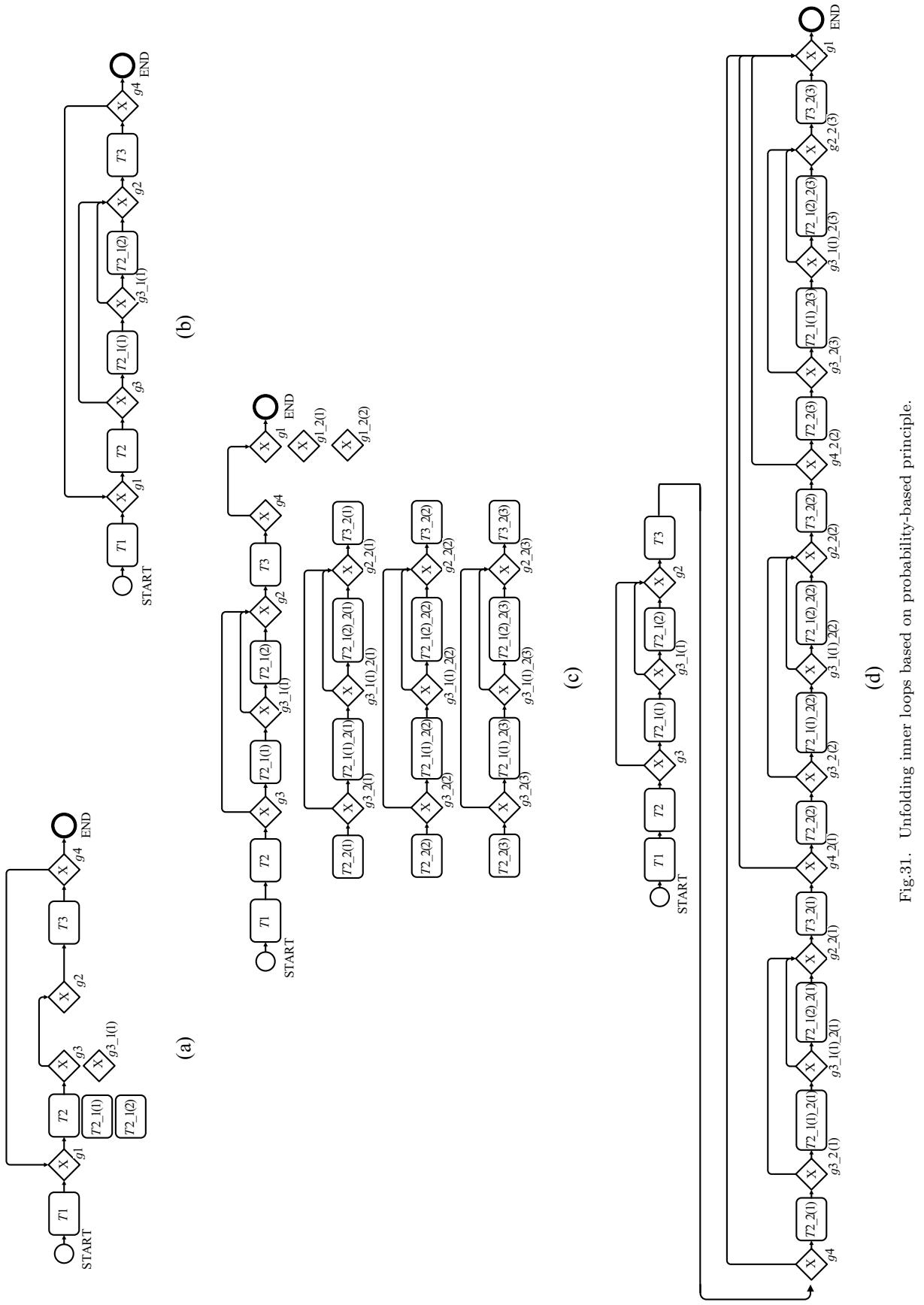


Fig.31. Unfolding inner loops based on probability-based principle.

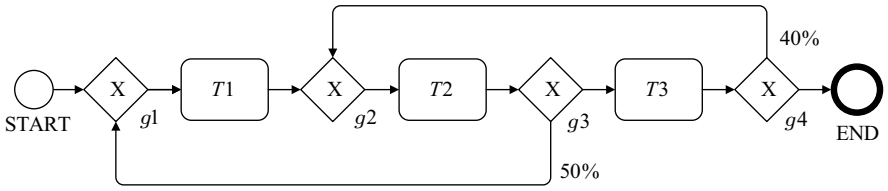


Fig.32. Example process model for unfolding a crossing loop.

Algorithm 7. Function $UnfoldCrLoop(\,)$ for Unfolding Crossing-Loop

Input: the graph WG of workflow model; the Crossing-Loops CL; loop Table L
Output: WG

```
1:  for loop loopL in L
2:      if CL → LEFT[loopL] is empty and CL → RIGHT[loopL] is not empty then
3:          % loopL is the leftmost loop in CL
4:          Call function UnfoldStLoop(WGloopL) to unfold loopL
5:          for loop loopR in CL → RIGHT[loopR]
6:              % loopR is the right loop of loopL
7:              Duplicate loopR's loop-exit point once and set it as tempGateway
8:              Insert tempGateway directly after loopR's loop-entry point
9:              for each node n where n is the successor of loopR's loop-exit point
10:                 if node n is copied from loopR's loop-entry point then
11:                     Disconnect the edge from n to loopR's loop-exit point
12:                     Connect the edge from n to tempGateway
13:                 end if
14:             end for
15:             Call function UnfoldStLoop(WGloopR) to unfold loopR
16:         end for
17:     end if
18: return WG
```

Unfolding a Crossing Loop Based on Probability-Based Principle. In this subsection, the same process model given in Fig.32 is used to explain the unfolding of a crossing loop based on probability-based principle. We assume that the tolerance value is $tol = 0.2$ and the probability values are $p_{L1} = 0.5$ (50%) and $p_{L2} = 0.4$ (40%). Therefore, the maximum number of iterations is 3 for the left loop because $p_{L1}^3 = 0.125 < tol$ and the maximum number of iterations is 2 for the right loop because $p_{L2}^2 = 0.16 < tol$.

We use Algorithm 7 to unfold the crossing loop given in Fig.32. First, the leftmost loop $L1$ is unfolded using the function $UnfoldStLoop(\,)$ in Algorithm 5. The step-by-step unfolding process is illustrated in Fig.34 and Fig.35. We can notice that in Fig.34(b), the loop exit point $g4$ of loop $L2$ contains more than two outgoing branches. According to Fig.35(a), we create $g4\_C$ from gateway $g4$. Next, the algorithm inserts the gateway $g4\_C$ directly after the loop entry point $g2$ of loop $L2$. Then, the algorithm disconnects the edges $(g4, g2\_1(1))$, $(g4, g2\_1(2))$ and $(g4, g2\_1(3))$. The algorithm also creates new edges $(g4\_C, g2\_1(1))$, $(g4\_C, g2\_1(2))$ and $(g4\_C, g2\_1(3))$. After that, $L2$ is unfolded using Algorithm 5. After $L2$ is unfolded, there are no more remaining loops in the process model. The final results of unfolding are given in Fig.35(b).

3.4 Complexity of the Algorithms

Complexity of the algorithm *UnfoldStLoop* can be written as $T_{St} = T(n) + T(m)$. $T(n)$ is for the first recursive call of copying elements in the loop when n is the size of loop. $T(m)$ is for the second recursive call to unfold the loop when m is the maximum number of iterations by calculation. $T(m)$ is equal to constant 1 for unravelling based on the Zero-One principle and the complexity of the algorithm is $O(n)$. For the probability-based unravelling, the complexity is $O(n \times m)$.

Complexity of the algorithm *UnfoldNeLoop* can be written as $T_{Ne} = T(p) \times T_{St}$. $T(p)$ is for the recursive call of unfolding all nested loops when p is the number of loops in the workflow. The complexity of Zero-One principle for *UnfoldStLoop* is $T_{St} = T(n)$ and therefore, the complexity of the algorithm *UnfoldNeLoop* for the Zero-One principle is $O(p \times n)$. For the probability-based principle, the complexity of the algorithm *UnfoldNeLoop* is $O(p \times n \times m)$.

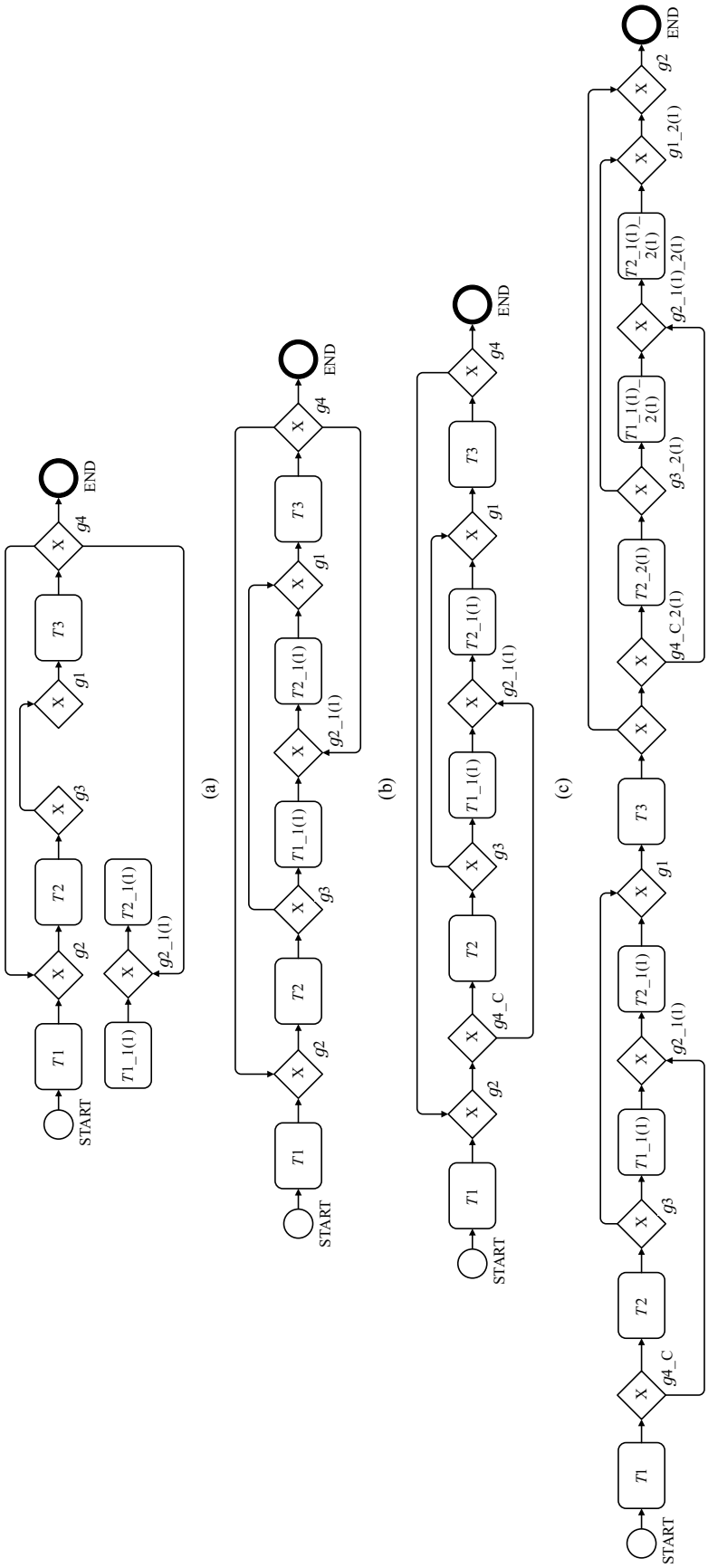


Fig.33. Unfolding a crossing loop based on Zero-One principle.

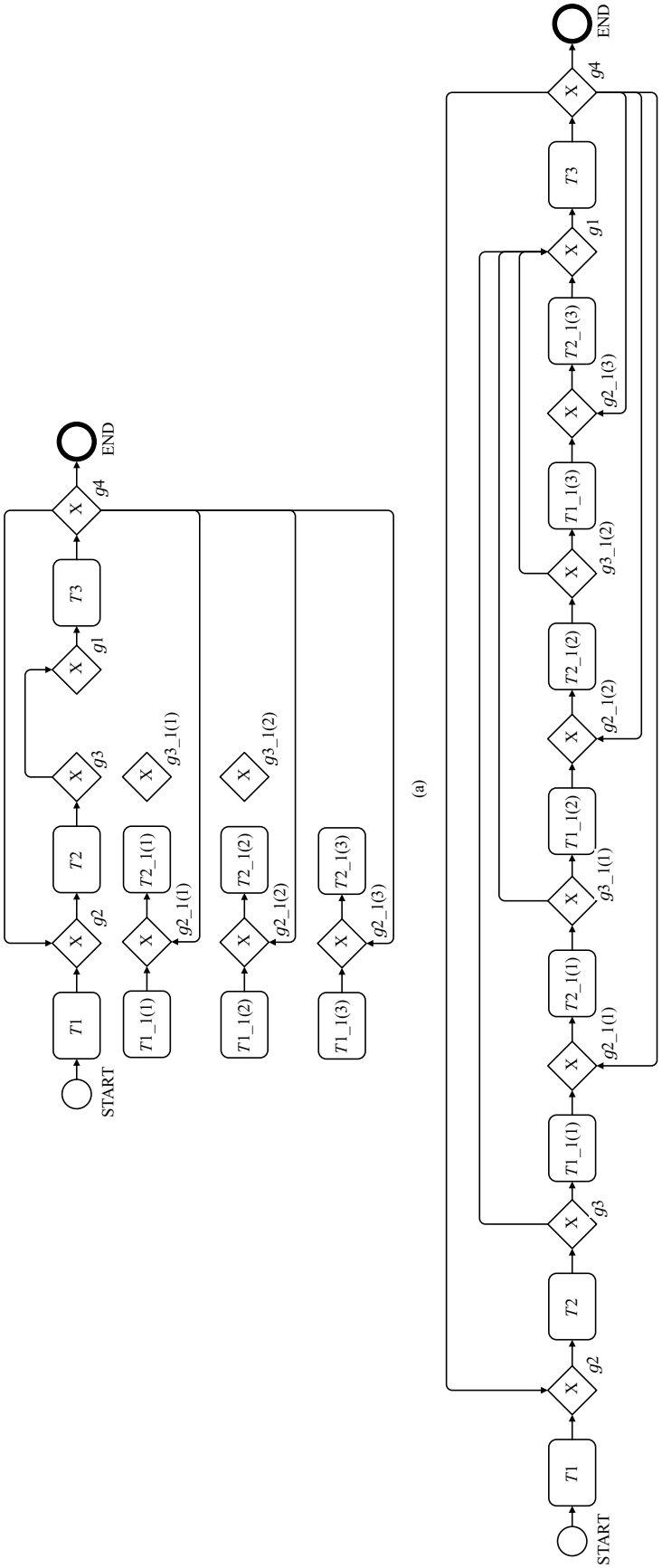
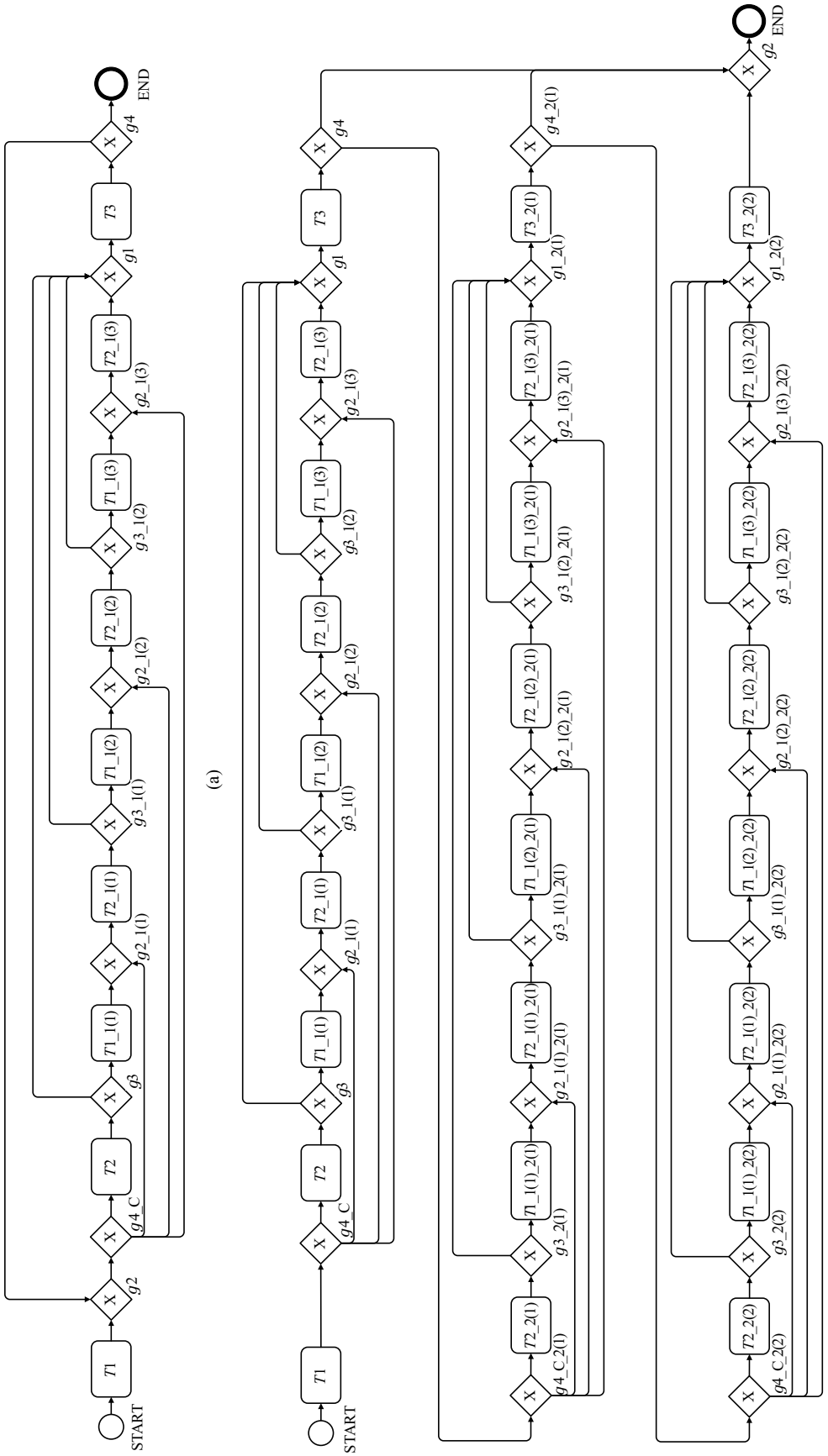


Fig.34. Unfolding a crossing loop based on probability-based principle (part 1).



(b)

Fig.35. Unfolding a crossing loop based on probability-based principle (part 2).

Complexity of the algorithm *UnfoldCrLoop* is similar to that of the algorithm *UnfoldNeLoop*. It can be written as $T_{Cr} = T(l) \times T_{St} + T(p-l) \times T_{St} = T(p) \times T_{St}$ where l is the number of loops which contain the entry point from another loop in crossing loops, p is the number of loops in the workflow and $p-l$ is the number of loops which contain the exit point from another loop in crossing loops. Therefore, the complexity for the Zero-One principle is $O(p \times n)$ and for probability-based principle is $O(p \times n \times m)$.

4 Experiments

Simulation experiments are performed to analyze the correctness of the proposed algorithms. In these experiments, all the process models are defined in BPMN, which include various types of iterative control structures. Signavio Process Manager① was used to model the processes and they were converted into Oryx② specific BPMN models. Loops in each BPMN model are annotated with probabilities and tolerance values at the split gateways. The prototype system was developed in JAVA programming language on a computer with Intel® Core™ i5-4210 CPU, 8 GB of RAM and 64 bits Windows 10 operating system. The probability and the tolerance used for the experiments are summarized in Table 1. In real cases, probability and tolerance could be defined by the process designers or calculated from historical information such as audit trails. The overview of the experimental setting is depicted in Fig.36.

First, we generate 10 000 runs for each BPMN model. A run is a directed acyclic graph. Each run represents one possible execution path of the workflow model. A run consists of all or a subset of nodes and edges within a workflow schema beginning from the start node to the end node [9, 33]. Detailed discussion of about the runs and relevant examples can be found in [34]. Second, we unfold the BPMN models based on the probability-based unfolding algorithms. Due to the page limitation, we do not include the experiments for unfolding with the Zero-One principle in this paper. Third, we use the unfolded BPMN models to execute all the runs generated from the previous step. The main purpose of this execution is to verify whether each run can be correctly executed in the unfolded BPMN model.

Table 1. Experimental Settings

		Probability	Tolerance	Number of Runs Generated
Experiment 1	Case 1	0.6	0.10	10 000
	Case 2	0.3	0.10	10 000
Experiment 2	Case 1	0.2, 0.6	0.10	10 000
	Case 2	0.2, 0.6	0.50	10 000
Experiment 3	Case 1	0.3, 0.2	0.10	10 000
	Case 2	0.4, 0.4	0.08	10 000

We define the following conditions for generating runs.

1) The sequence of tasks and gateways will be recorded in each run. We also record the probability value when a path is selected at the gateways.

2) The choice of route for the gateway will be decided after the randomly generated probability value is compared against the pre-defined probability value at the gateways.

3) Similar to the unfolding algorithm, the run generator adheres to the maximum number of iterations calculated for each loop.

The BPMN models used for the three experiments are shown in Fig.37–Fig.41.

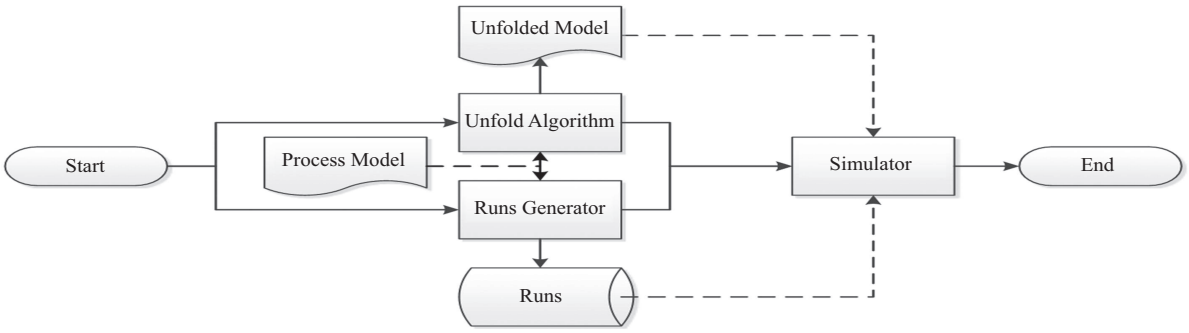


Fig.36. Overview of experimental setting.

① https://www.signavio.com/, October 2020.

② https://code.google.com/archive/p/oryx-editor/, October 2020.

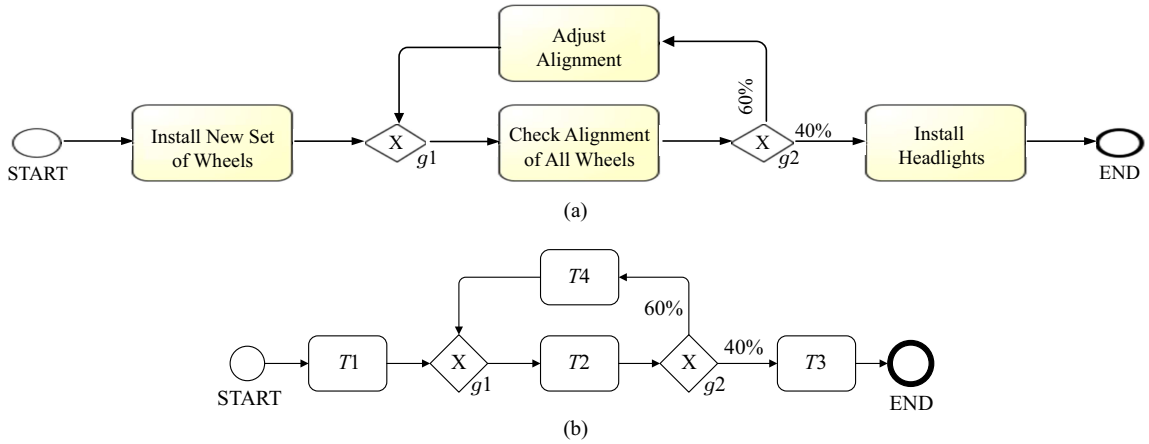


Fig.37. Process model for the case 1 of experiment 1. (a) Wheel adjustment process. (b) Simplified process model.

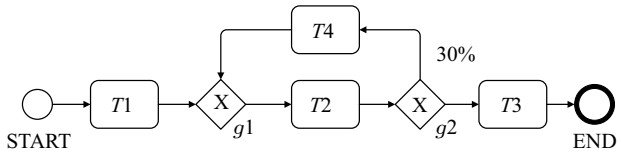


Fig.38. Process model for case 2 of experiment 1.

• The process model for case 1 of experiment 1 is depicted in Fig.37(a). The model illustrates the process of new wheels installation in a car factory. In this process, the alignment of new wheels is adjusted until they are satisfactory. Adjustment is needed in 60% of the every checking of all wheels. After all the wheels are aligned correctly, headlights can be installed. The simplified version of the new wheels installation process is depicted in Fig.37(b) where activity names are replaced with task ID. The process model for case 2 of experiment 1 depicted in Fig.38 is identical to the case 1 except the split probability at gateway $g2$.

• The process model for case 1 of experiment 2 is depicted in Fig.39(a). The model illustrates a project management process of a marketing department. In this process, the department first prepares a proposal for marketing campaign. The proposal includes objectives for achieving specific outcomes. Next, the cost of achieving these objectives is estimated. After that, the proposal is submitted to the department head for approval. Each submitted case has 20% of being rejected by the department head (split gateway $g3$). In case the proposal is rejected by the department, the adjustment to the resources required for the campaign is performed. If the proposal is endorsed by the department head, it will be submitted to the executive committee for final approval. For each proposal submitted to the executive committee, the probability of disapproval is 60% (split gateway $g4$). If the proposal is rejected by the executive committee, it will be sent back to the marketing department for revising the objectives. Revising objectives include the re-work of cost estimation and obtaining department head approval. If the proposal is approved by the executive committee, it will be announced to the customers by public relation (PR) department. The simplified version of the project management process is depicted in Fig.39(b) where activity names are replaced with task ID.

• The process model for case 1 of experiment 3 is depicted in Fig.40(a). The model illustrates a part of design process from a mobile phone factory. In this process, designers first choose the model of a central processing unit (CPU) for the new mobile phone. Next, designers decide the storage and SIM card option for the phone. For instance, the new phone may be equipped with an SD card slot and two SIM slots for dual standby. Next, an estimation for the size of battery (capacity) for the phone is made. Based on the estimated capacity, the size of battery compartment in the phone is checked for fitting. Such checking results in 20% of cases (split gateway $g2$) require adjusting the storage and SIM card options. For example, by omitting some of these options, the size of battery compartment could be increased. If the size of battery compartment is within an acceptable range, designers will continue with the selection of display panel type (e.g., IPS-LCD, OLED or AMOLED). Next, based on the preceding selection of CPU model, Storage and SIM card options, and the display, the total power consumption is calculated and compared against battery capacity. In 30% of the cases (split gateway $g4$), the estimated total power consumption is incompatible with the battery capacity. In this case, the adjustment in the size of

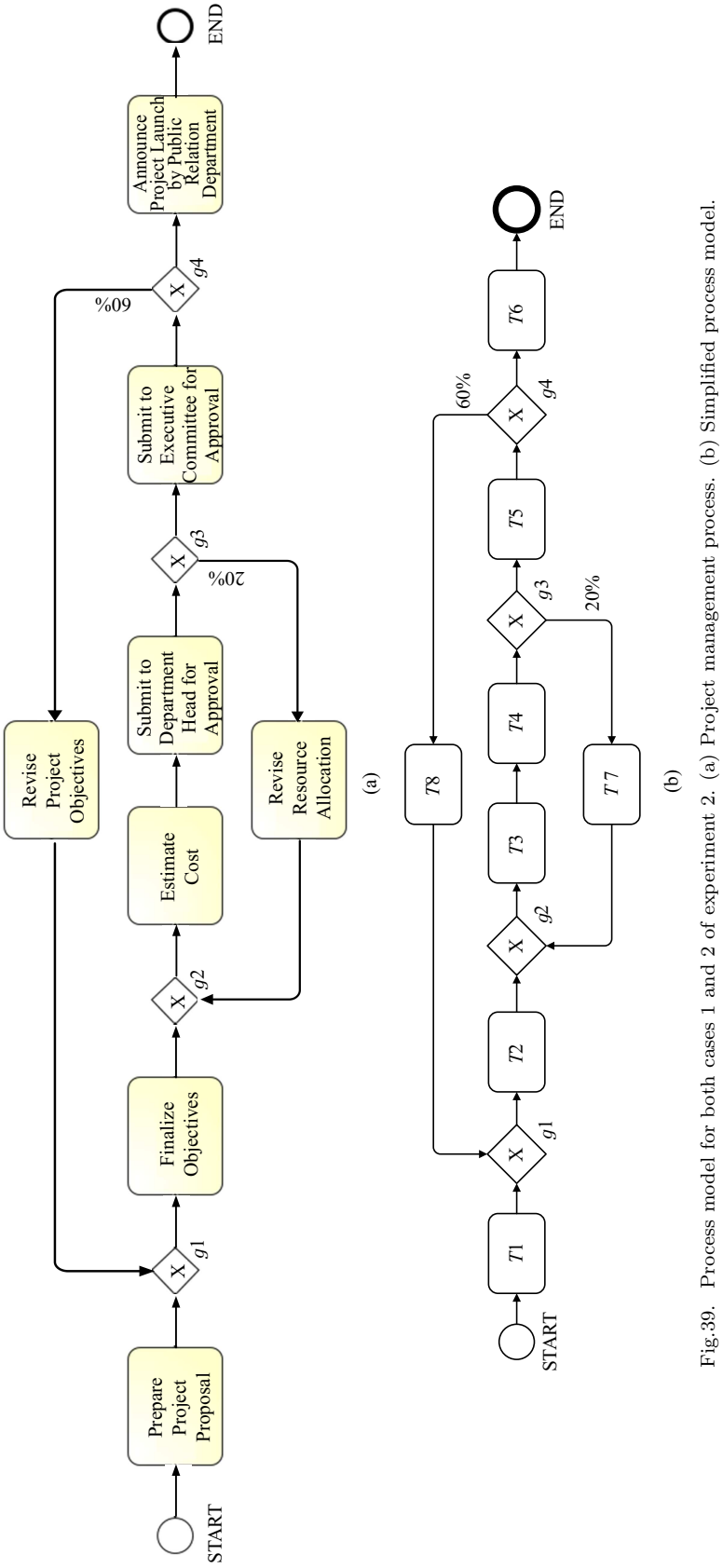
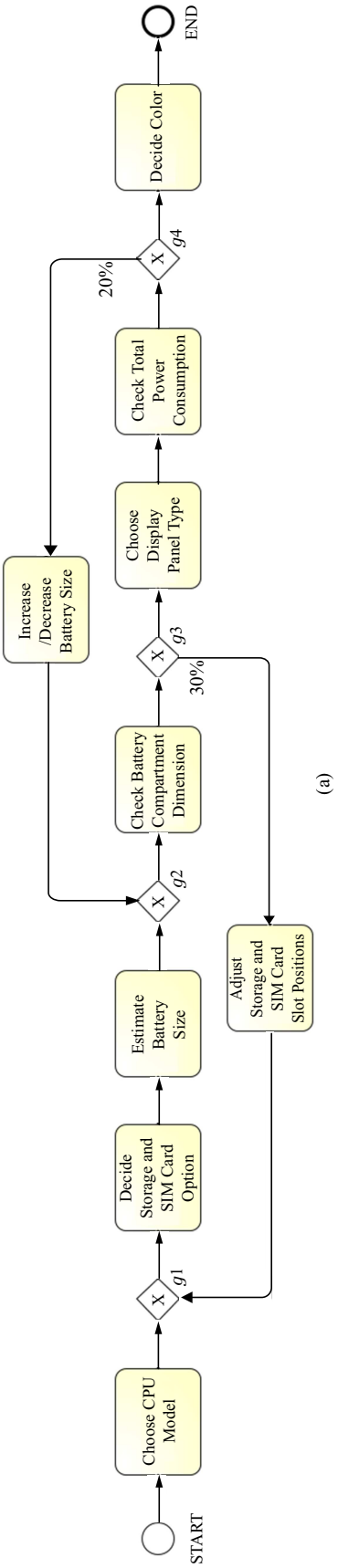
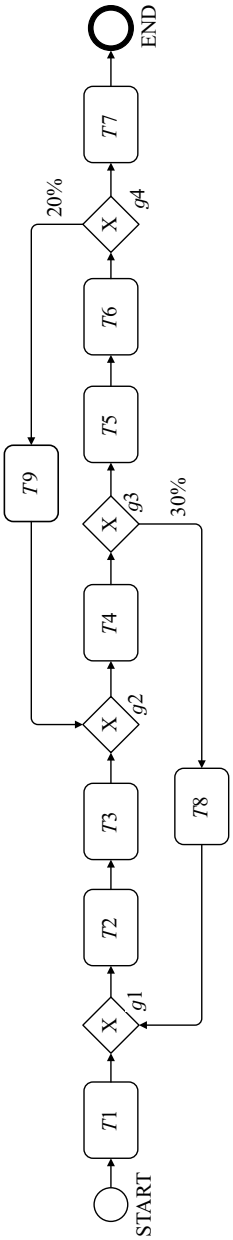


Fig.39. Process model for both cases 1 and 2 of experiment 2. (a) Project management process. (b) Simplified process model.



(a)



(b)

Fig.40. Process model for case 1 of experiment 3. (a) Mobile phone design process. (b) Simplified process model.

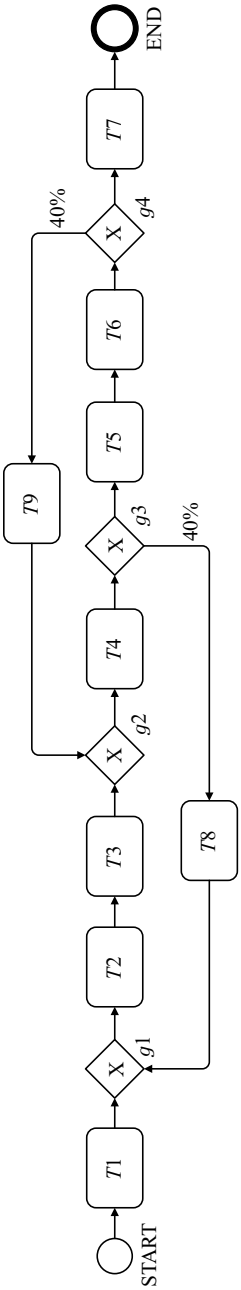


Fig.41. Process model for case 2 of experiment 3.

battery (capacity) must be made. The adjustment in battery capacity requires recalculation of battery compartment dimension and re-selection of display panel. It may also require the reconsideration of storage and SIM card options (split gateway $g3$). If the estimated total power consumption is compatible with the battery capacity, designers will proceed with the selection of colors for the back/front covers of the phone. The simplified version of the process is depicted in Fig.40(b) where activity names are replaced with task ID. The process model for case 2 of experiment 3 depicted in Fig.41 is identical to that for case 1 except the split probabilities at the gateways.

In experiment 1, tolerance value is kept constant for both case 1 and case 2 but different probabilities at the split gateway are used in testing. In experiment 2, probability values at two split gateways are kept constant for both case 1 and case 2 but different tolerance values are used for testing. In experiment 3, different probability and tolerance values are used in case 1 and case 2.

These BPMN models are unfolded based on the probability-based principle. For the purpose of visualization, the results of unfolding are exported into graphs which are formatted in the DOT language of Graphviz software[③]. We do not include the figures of the unfolded model in this paper since the unfolded models can include more than 100 nodes (tasks) and they are too large to fit in an A4 size paper.

A model containing a Structure-Loop in Fig.37 is used to illustrate the functions of Run Generator. In this example, we define the probabilities of the branches for gateway $g2$ as 0.6 and 0.4 (for $\Pr((g2, T4))$ and $\Pr((g2, T3))$). We also assume that the tolerance of accumulated probability is 0.1. Therefore, the maximum number of iterations is 5 because $0.6^5 = 0.077\,76 < 0.1$. When the Run Generator reaches a gateway, it will randomly generate a probability value which is denoted as p and the generator will decide which route to be selected. For example, if the randomly generated probability value is 0.7, the generator will choose the edge $(g2, T4)$. The generator continues until it reaches the End node. Based on this procedure, we can generate a set of runs from the model given in Fig.37. Three of the generated runs are shown in Fig.42.

$$\begin{aligned} run1 = {} & (\text{START}, T1, g1, T2, g2(p_1 = 0.82), \\ & T4, g1, T2, g2(p_2 = 0.63), \\ & T4, g1, T2, g2(p_3 = 0.27), T3, \text{END}). \end{aligned}$$

$$\begin{aligned} run2 = {} & (\text{START}, T1, g1, T2, g2(p_1 = 0.58), \\ & T3, \text{END}). \end{aligned}$$

$$\begin{aligned} run3 = {} & (\text{START}, T1, g1, T2, g2(p_1 = 0.66), \\ & T4, g1, T2, g2(p_2 = 0.63), \\ & T4, g1, T2, g2(p_3 = 0.93), \\ & g1, T2, g2(p_4 = 0.28), T3, \text{END}). \end{aligned}$$

We proceed to unfold the process model given in Fig.37 by using the proposed unfolding algorithms and obtain the model in Fig.43. Next, we execute/compare the first run in Fig.42(a) against the unfolded model. First, the execution starts at the start node from the first run. Next, the task $T1$ from the run matches with the task $T1$ from the unfolded model. Then, the execution reaches the XOR-gateway $g1$ which contains one incoming branch and one outgoing branch. Since, this gateway only contains one outgoing branch, it can be ignored. After that, task $T2$ is reached and executed on the unfolded model. After task $T2$ is executed, the next node in sequence will be gateway $g2$. According to the run, the randomly generated probability value at $g2$ is $p_1 = 0.82$. The simulator chooses the branch from gateway $g2$ to task $T4$ since p_1 is larger than 0.6. The simulator continues the execution until the END node is reached.

From the experiment results, we find that each set of 10 000 runs generated from the original models can be correctly executed in the unfolded BPMN models that do not contain any loops.

5 Conclusions

Iterative control structures are widely used in business processes. However, the use of iterative control structures can result in unanticipated outcomes in workflow management systems, specifically, when loops are labelled with branching probabilities at split gateways. In this paper, we proposed two methods for unravelling iterative control structures in process models: the first method is based on Zero-One principle and the second method is based on probabilistic conditions at the split gateways. The proposed methods can be used to unfold structured loops, nested loops and crossing loops. In the proposed methods, the process model containing iterative control structures is transformed (unfolded) into a process model that does not contain any loops. The unravelling methods proposed in this article address all three types of iterative control structures (structured loops, nested loops, and crossing loops).

③ http://www.graphviz.org/, October 2020.

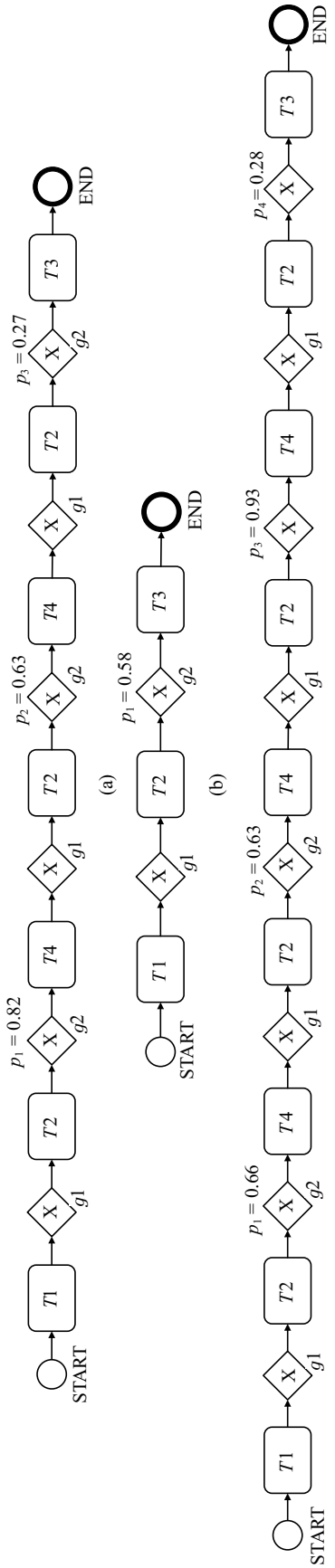


Fig.42. Examples of runs generated from the model given in Fig.37.

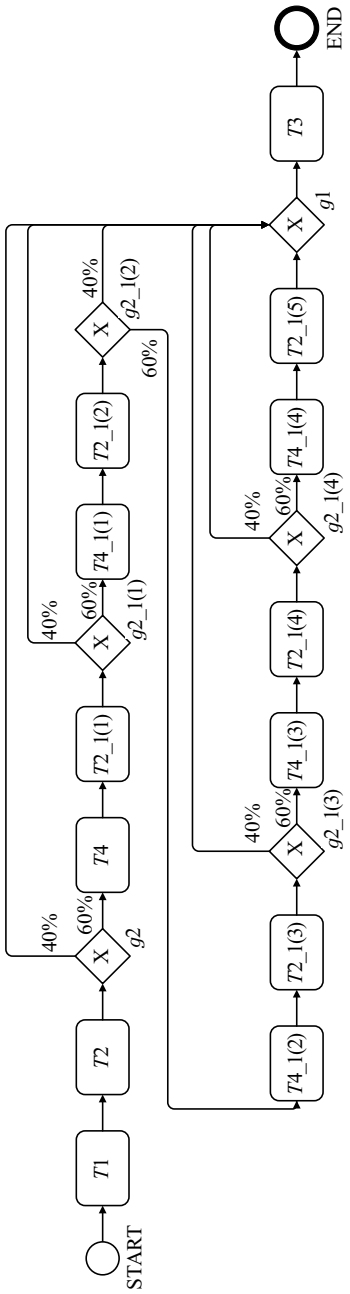


Fig.43. Unfolded model of process in Fig.37.

Our methods are based on workflow graphs and therefore they are compatible with modeling languages such as Business Process Modelling Notation (BPMN). The proposed methods are tested on a number of process models containing various types of loops. By using the concept of runs, the execution of unfolded process models is compared against the original models.

The models unfolded by the proposed approaches no longer contain loops when they are compared with the original process models. However, the size of unfolded models becomes larger since additional gateways and duplicate tasks are added during the unravelling process. Therefore, for the nested and crossing loops, the readability and the simplicity will be negatively affected by the unravelling. However, the unraveled process models can be more easily analyzed for any potential issues by process designers or by automated verification programs due to their deterministic nature. Therefore, a trade-off needs to be made by process designers in balancing the readability, simplicity, and verifiability in unravelling of iterative control structures from the models. Although the proposed unravelling approaches can be used to unfold structured loops, nested loops, and simple crossing loops, the proposed algorithms have following limitations. First, the proposed methods only consider loops structures with exclusive decisions. For unravelling, only activities are allowed in back edge of the loop. A back edge is an edge that a workflow instance can take to return the start point of the loop. We also assume that tasks and gateways in a loop are independent with entities or other loops.

Since the unfolded model does not contain any iterative control structures, it can be used for further analysis by process designers during the modeling phase. For instance, one of the analyses could focus on the temporal constraints. Since unraveled results become deterministic process models, the maximum execution time of the workflow can be estimated. By comparing with the maximum allowable execution time set by the workflow designer, potential temporal exceptions can be predicted during the design time. Another direction in further analysis could be the optimization of the programs. For example, the proposed approach could be extended for loop unrolling in real-time applications and compiler constructions. Likewise, the proposed approaches could be used for more efficient resource assignment in multistage manufacturing systems involving rework loops.

References

[1] Weske M. Adaptive workflows based on flexible assignment of workflow schemes and workflow instances. In *Proc. the Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, October 1999, pp.42-48.

[2] Aalst W M P V D, Hofstede A H M T, Kiepuszewski B, Barros A P. Workflow patterns. *Distrib. Parallel Databases*, 2003, 14(1): 5-51. DOI: 10.1023/A:1022883727209.

[3] van derAalst W, van Hee K. Workflow Management: Models, Methods, and Systems. The MIT Press, 2004.

[4] van der Aalst W M P, Barros A P, ter Hofstede A H M, Kiepuszewski B. Advanced workflow patterns. In *Proc. the 7th International Conference on Cooperative Information Systems*, September 2000, pp.18-29. DOI: 10.1007/10722620_2.

[5] Sarkar V. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 2001, 29(5): 545-581. DOI: 10.1023/A:1012246031671.

[6] Aho A V, Ullman J D. Principles of Compiler Design. Addison-Wesley, 1977.

[7] Yu Y, Xie T, Wang X. A handling algorithm for workflow time exception based on history logs. *The Journal of Supercomputing*, 2013, 63(1): 89-106. DOI: 10.1007/s11227-010-0543-7.

[8] Eder J, Pichler H. Duration histograms for workflow systems. In *Proc. the IFIP TC8/WG8.1 Working Conference on Engineering Information Systems in the Internet Context*, September 2002, pp.239-253. DOI: 10.1007/978-0-387-35614-3_14.

[9] Dumas M, García-Bañuelos L, Ho K S, Si Y W. Extended choice relation framework for workflow testing. In *Proc. the 12th Symposium on Programming Languages and Software Tools*, October 2011, pp.236-247.

[10] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach (5th edition). Morgan Kaufmann, 2011.

[11] Kukunas J. Toolchain primer. In *Power and Performance: Software Analysis and Optimization*, Kukunas J (ed.), Morgan Kaufmann, 2015, pp.207-239. DOI: 10.1016/B978-0-12-800726-6.00012-4.

[12] Velkoski G, Gusev M, Ristov S. The performance impact analysis of loop unrolling. In *Proc. the 37th International Convention on Information and Communication Technology, Electronics and Microelectronics*, May 2014, pp.307-312. DOI: 10.1109/MIPRO.2014.6859582.

[13] Cardoso J, Coutinho J, Diniz P. Source code transformations and optimizations. In *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*, Cardoso J, Coutinho J, Diniz P (eds.), Morgan Kaufmann, 2017, pp.137-183. DOI: 10.1016/C2015-0-00283-0.

[14] Cooper K D, Torczon L. Introduction to optimization. In *Engineering a Compiler* (2nd edition), Cooper K D, Torczon L (eds.), Morgan Kaufmann, 2012, pp.405-474. DOI: 10.1016/C2009-0-27982-7.

[15] Huang J C, Leng T. Generalized loop-unrolling: A method for program speedup. In *Proc. the 1999 IEEE Symposium on Application—Specific Systems and Software Engineering and Technology*, March 1999, pp.244-248. DOI: 10.1109/ASSET.1999.756775.

[16] Weinhardt M. High-level synthesis oriented restructuring of functions with while loops. In *Proc. the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops*, May 2019, pp.115-122. DOI: 10.1109/IPDPSW.2019.00029.

[17] Carminati A, Starke R A, de Oliveira R S. Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Applied Computing and Informatics*, 2017, 13(2): 184-193. DOI: 10.1016/j.aci.2017.03.002.

[18] Dias J, Guerra G, Rochinha F, Coutinho A L G A, Valduriez P, Mattoso M. Data-centric iteration in dynamic workflows. *Future Gener. Comput. Syst.*, 2015, 46(C): 114-126. DOI: 10.1016/j.future.2014.10.021.

[19] Cao Y, Subramaniam V, Chen R. Performance evaluation and enhancement of multistage manufacturing systems with rework loops. *Comput. Ind. Eng.*, 2012, 62(1): 161-176. DOI: 10.1016/j.cie.2011.09.004.

[20] Heng Z, Aiping L, Xuemei L, Liyun X, Moroni G. Modeling and performance evaluation of multistage serial manufacturing systems with rework loops and product polymorphism. *Procedia CIRP*, 2017, 63: 471-476. DOI: 10.1016/j.procir.2017.03.347.

[21] Choi I, Jung J, Mannino M, Park C. Terminability and compensatibility of cycles in business processes with a process-oriented trigger. *Data Knowl. Eng.*, 2008, 66(2): 243-263. DOI: 10.1016/j.datak.2008.03.002.

[22] Park C, Choi I. Management of business process constraints using BPTrigger. *Comput. Ind.*, 2004, 55(1): 29-51. DOI: 10.1016/j.compind.2003.11.003.

[23] Polyvyanyy A, García-Bañuelos L, Weske M. Unveiling hidden unstructured regions in process models. In *Proc. the Confederated International Conferences on the Move to Meaningful Internet Systems*, November 2009, pp.340-356. DOI: 10.1007/978-3-642-05148-7.

[24] Koehler J, Hauser R. Untangling unstructured cyclic flows—A solution based on continuations. In *Proc. the 2004 OTM Confederated International Conferences on the Move to Meaningful Internet Systems*, October 2004, pp.121-138. DOI: 10.1007/978-3-540-30468-5_10.

[25] Dumas M, García-Bañuelos L, Polyvyanyy A. Unraveling unstructured process models. In *Proc. the 2nd International Workshop on Business Process Modeling Notation*, October 2010, pp.1-7. DOI: 10.1007/978-3-642-16298-5_1.

[26] Eshuis R, Kumar A. Converting unstructured into semi-structured process models. *Data Knowl. Eng.*, 2016, 101(C): 43-61. DOI: 10.1016/j.datak.2015.10.003.

[27] Siavvas M, Gelenbe E. Optimum checkpoints for programs with loops. *Simulation Modelling Practice and Theory*, 2019, 97: Article No. 101951. DOI: 10.1016/j.simpat.2019.101951.

[28] Williams M, Ossher H L. Conversion of unstructured flow diagrams to structured form. *Comput. J.*, 1978, 21(2): 161-167. DOI: 10.1093/comjnl/21.2.161.

[29] Kiepuszewski B, ter Hofstede A H M, Bussler C. On structured workflow modelling. In *Proc. the 12th International Conference on Advanced Information Systems Engineering*, June 2000, pp.431-445. DOI: 10.1007/3-540-45140-4_29.

[30] Heinze T, Amme W, Moser S. Control flow unfolding of workflow graphs using predicate analysis and SMT solving. In *Proc. the 5th Central-European Workshop on Services and their Composition*, February 2013, pp.1-8.

[31] Heinze T, Amme W, Moser S, Gebhardt K. Guided control flow unfolding for workflow graphs using value range information. In *Proc. the 4th Central-European Workshop on Services and their Composition,* February 2012, pp.128-135.

[32] Choi Y, Kongsuwan P, Joo C M, Zhao J L. Stepwise structural verification of cyclic workflow models with acyclic decomposition and reduction of loops. *Data & Knowledge Engineering*, 2015, 95: 39-65. DOI: 10.1016/j.datak.2014.11.003.

[33] van der Aalst W M P, Hirnschall A, Verbeek H M W. An alternative way to analyze workflow graphs. In *Proc. the 14th International Conference on Advanced Information Systems Engineering*, May 2002, pp.535-552. DOI: 10.1007/3-540-47961-9_37.

[34] Si Y W, Hoi K K, Biuk-Aghai R P, Fong S, Zhang D. Run-based exception prediction for workflows. *J. Syst. Softw.*, 2016, 113(C): 59-75. DOI: 10.1016/j.jss.2015.11.024.

Yain-Whar Si is an associate professor at the Department of Computer and Information Science, University of Macau, Macau. His research interest includes business process management, information visualization, and decision support systems.

Weng-Hong Yung received his Bachelor's degree in computer and information science from University of Macau, Macau, China, in 2011. He is currently an information technology technician in the Macau government. He has an interest in the field of programming, system design, web development and information security.