Structural Descriptions of Process Models Based on Goal-Oriented Unfolding

Chen Qian[1], Lijie Wen[1(✉)], Jianmin Wang[1], Akhil Kumar[2], and Haoran Li[3]

[1] School of Software, Tsinghua University, Beijing 100084, China
qc16@mails.tsinghua.edu.cn, {wenlj,jimwang}@tsinghua.edu.cn
[2] Smeal College of Business, Penn State University, 16802 State College, USA
akhilkumar@psu.edu
[3] The Affiliated School of Peking University, Beijing 100080, China
lihaoran1@stu.pkuschool.edu.cn

Abstract. Business processes are normally managed by designing, operating and analysing corresponding process models. While delivering these process models, an understanding gap arises depending on the degree of different users' familiarity with modeling languages, which may slow down or even stop the normal functioning of processes. Therefore, a method for automatically generating texts from process models was proposed. However, the current method just involves ordinary model patterns so that the coverage of the generated text is too low and information loss exists. In this paper, we propose an improved transformation algorithm named Goun to tackle this problem of describing the process models automatically. The experimental results demonstrate that the Goun algorithm not only supports more elements and complex structures, but also remarkably improves the coverage of generated text.

Keywords: Process model · Natural language text generation · Extended process structure tree · Matching · Unfolding

1 Introduction

Requirements analysis is the primary step in software development. In this step, there are two kinds of important roles named domain expert and systems analyst [3]. Through oral communications and documents, they try to thoroughly understand how business processes run. After that, a systems analyst converts what he or she has comprehended into business process models which can be expressed by Petri net, Business Process Modeling and Notation (BPMN), Event-driven Process Chains (EPC) or other modeling languages [1,8,14]. However, there is a high possibility that the domain expert lacks the confidence to understand these models, or even feels that it is too time-consuming to learn formal process modeling, which may directly lead to interruption or even abortion of the project [3]. Therefore, our research goal here is to automatically generate the corresponding natural language text for a given process model since natural language texts could be read and understood by almost everyone.

E. Dubois and K. Pohl (Eds.): CAiSE 2017, LNCS 10253, pp. 397–412, 2017.
DOI: 10.1007/978-3-319-59536-8_25

Against this background, Leopold et al. proposed a model-to-text transformation algorithm (abbr. Hen) which transforms a BPMN model into its corresponding textual description to avoid understanding gaps between different roles [3]. Hen linearized a process model by traversing its corresponding Refined Process Structure Tree (RPST) [2]. When processing the leaf nodes of the tree, a data structure - Deep Syntactic Tree - is derived to represent a natural language sentence. After refinement and realization [15], the textual description with bullet points is generated. An example for illustrating the model-to-text transformation scenario is presented next.

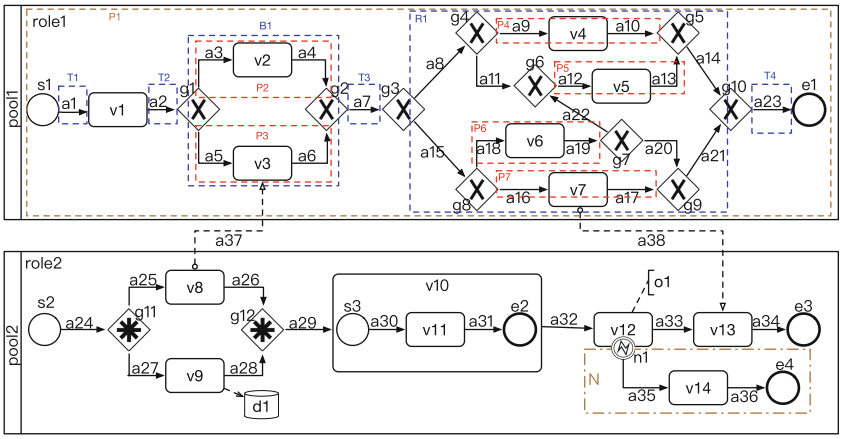


Fig. 1. A sample of BPMN 2.0 model (Color figure online)

Example 1. Taking the *pool*1 in Fig. 1 as an example, the Hen text is (for illustration, we add 9 line numbers below to mark every sentence):

1. *The 'pool1' process begins when the 'role1' does 'v1'. Then one of the following branches is executed.*
2. • *The 'role1' does 'v2'.*
3. • *The 'role1' does 'v3'.*
4. *Once one of the above branches is executed, there is a region which allows for different execution paths. One option from start to end is the following.*
5. • *The 'role1' does 'v6'. Subsequently, the 'role1' does 'v5'.*
6. *However, the region allows for a number of deviations.*
7. • *After the region begins, the 'role1' can also do 'v4'.*
8. • *After the region begins, the 'role1' can also do 'v7'.*
9. *Then, the 'pool1' process is finished.*

The sentences 1, 4 and 6 in *Example* 1 are pre-defined language templates to express the semantics of certain patterns. Note that there is a nested structure above to show the depth of a sentence using indentation with bullet points '•'.

However, to the authors' knowledge, Hen has the following four main drawbacks. First, Hen fails to generate the text when a model contains some kinds of complex elements, such as text annotation, complex gateway, message flow, or subprocess [1]. For the *pool*2 in Fig. 1, Hen fails to generate sentences for the complex gateways $g11$ and $g12$, the subprocess $v10$ and the error path N. Second, Hen only works on free-choice models and omits some descriptions of trivial paths, i.e. it suffers from information loss. For example, the Hen text in *Example* 1 is somewhat incomplete because it omits some behavior such as when only $v5$ is executed (without $v6$). Third, if we were to change the direction of $a17$ and $a16$ to create a loop structure, Hen fails to generate text for it due to the existence of the loop structure in the non-structured part (i.e. $R1$). Fourth, the generated text of non-structured part is a linear structure with limited expressive power. Based on these insights, one of our goals is to solve these drawbacks of Hen and improve the expressive power of model-to-text conversion.

Our research contributions are as follows. First, we proposed a fresh data structure to express more model patterns. Second, we developed a new linearization technique based on the basic architecture of Hen to transform a model into a structured one and then convert it into textual description, which is the biggest challenge. Third, we evaluated different methods along different dimensions and discussed the results of the comparison.

The remainder of this paper is organized as follows. Section 2 briefly introduces the definitions used in this paper. Section 3 presents the new data structure EPST to express more model patterns, while Sect. 4 introduces Goun and exemplifies it with a specific model. Later, Sect. 5 evaluates the Goun algorithm and compares it with the Hen algorithm. Finally, Sect. 6 describes related work and Sect. 7 draws our conclusions.

2 Preliminaries

This section introduces Business Process Modeling and Notation (BPMN) and Refined Process Structure Tree (RPST) [2–4].

2.1 BPMN

BPMN is a graphical process modeling language. BPMN version 2.0 contains *swimlanes* (pools, lanes), *nodes* (events, activities, gateways), $flows$ (sequence flow, message flow) and $artifacts$ (e.g. text annotation, data stores) [4].

Example 2. The full process model in Fig. 1 contains two pools denoted by the two large rectangles. Events are represented as circles with thin or thick borders. The corner-rounded rectangles $v1, v2$ etc. are activities. In particular, activity $v10$ is a subprocess because it can be decomposed into three nodes $s3, v11, e2$ linked by edges $a30$ and $a31$. The diamonds with 'X' symbols $g1$, $g2$, etc. are exclusive gateways which split or merge multiple branches. $d1$ and $o1$ are the data store and text annotation, respectively. The message flows $a37$ and $a38$

transfer messages between two activities belonging to different pools as shown by dotted lines. The error path N occurs if there is an error while processing $v12$. Node $v1$ precedes gateway $g1$; and $v2, v3$ follow $g1$; thus, the pre- and post-set of $g1$ are $\bullet g1 = \{v1\}$ and $g1\bullet = \{v2, v3\}$.

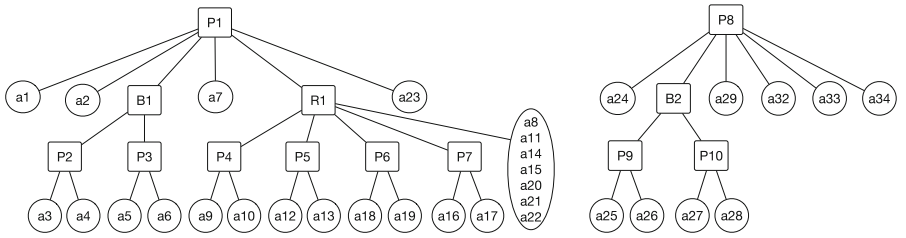
2.2 RPST

RPST is a kind of tree structure that indicates SESE (single entry single exit) parts of a process model. We call these parts process components. A *process component* is *canonical* iff (i.e., if and only if) it does not overlap with any other process component, meaning that any two canonical process components are either disjoint or nested. Therefore, canonical process components naturally form a hierarchy, which leads to the definition of RPST [6].

Definition 1 (Trivial, Polygon, Bond, Rigid). *Let C be a process component of a process model. C belongs to one of T, P, B or R components.*

- *C belongs to a Trivial (T) component, iff C contains only a single edge.*
- *C belongs to a Polygon (P) component, iff C contains a set of components linked consecutively.*
- *C belongs to a Bond (B) component, iff all components in C share one source node and one sink node.*
- *C belongs to a Rigid (R) component, iff C is none of T, P or B.*

Definition 2 (RPST). *The refined process structure tree (RPST) of a process model is the set of all its canonical process components.*



(a) The RPST of *pool*1 (b) The RPST of *pool*2

Fig. 2. The two RPSTs of the two pools in Fig. 1

Example 3. In Fig. 1, the colored dotted-line rectangles represent different depths in decomposition (blue, red represent depth 1, 2, respectively). Every single node in Fig. 1 attaches to its following Trivial component. Except Trivial, the other three kinds of canonical components can be decomposed. In *pool*1, $P1$ contains six components $T1, T2, B1, T3, R1, T4$. More precisely, component $B1's$ sub-polygon components $P2, P3$ share one source node $g1$ and one sink

node $g2$. $R1$ is unstructured since the edge $a22$ connects two separate branches $P5$ and $P6$, which leads to the unstructured Rigid component. Every component in a Rigid component is bounded by a gateway node such as $P4, P5, P6, P7$ and non-decomposable Trivial components such as $a8, a11$, etc. These visual decompositions naturally form a hierarchy so that we can get the undirected tree representation of a RPST in Fig. 2. For ease of representation, seven edges are shown in the same ellipse in Fig. 2(a) but, in fact, they are seven independent leaves of $R1$. The RPST of *pool*2 is generated in the same way.

3 Extending Expressive Patterns of RPST

From *Example* 3, we know that the lack of corresponding textual expressions for some of the patterns in RPST prevents Hen from generating the corresponding text, and impairs its expressive ability. Thus, we proposed a new data structure EPST and added some language templates to enhance RPST.

Definition 3 (Extended Process Structure Tree). *Extended Process Structure Tree (EPST) is a multi-tuple* $E_m = (R_S, r_O, E_R, M, O, M_S, \vartheta, \mu)$.

- R_S *is a finite set of RPSTs for all pools in the whole model.*
- r_O *is the super root node.* r_O *connects every* $r_s \in R_S$ *through edge set* E_R.
- M *is a finite set of sub-models, which contains models corresponding to a subprocess, an exceptional path, a group element, etc.*
- O *is a finite set of artifacts and data objects, i.e. text annotations, IT systems, data objects and data stores.*
- M_S *is a set of message flows that link two elements in different pools.*
- ϑ *is a function attaching a* T *component and a corresponding element in* M.
- μ *is a function attaching a* T *component and a corresponding element in* O.

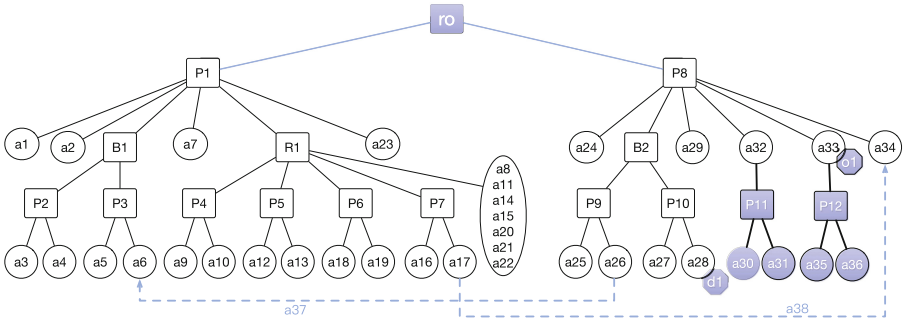


Fig. 3. The EPST of the BPMN 2.0 model in Fig. 1

Example 4. In Fig. 1, *pool2* contains patterns that RPST cannot express such as subprocess $v10$, exceptional path N, data store $d1$ and text annotation $o1$. Besides, there are two message flows transmitting messages between the two

pools. The EPST of the whole model in Fig. 1 is shown in Fig. 3. In the EPST, $R_S = \{P1, P8\}$; and, $P1$ and $P8$ connect with super root node r_O. For $M = \{P11, P12\}$, $P11$ and $P12$ are respectively attached to subprocess $v10$ and error event $n1$. For $M_S = \{a37, a38\}$, $a37$ and $a38$ are message flows. For $O = \{o1, d1\}$, $o1$ and $d1$ are attached onto corresponding elements $a33$ and $a28$. Therefore, the set of patterns EPST can express is the superset of that of RPST.

4 Structural Description

Figure 4 shows the overview of Goun in which blue parts highlight our work based on the Hen architecture. As Fig. 4 indicates, the input is a process model passing through five main steps to generate the final textual description. Here, converting a random model into a well-structured one is our biggest challenge.

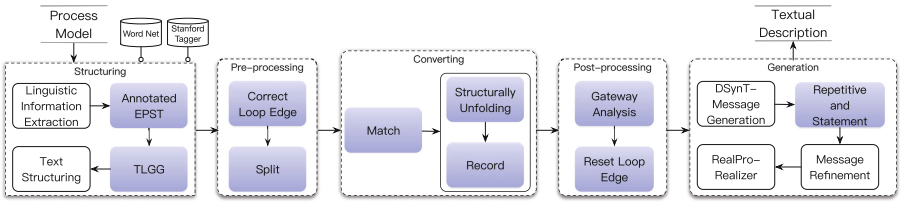


Fig. 4. The overview of Goun (Color figure online)

(1) *Structuring Step.* This step generates two main data structures of a process model, i.e. EPST and TLGG that will be introduced in this section.
(2) *Pre-processing Step.* Loop edges in a model are edges that can cause loops. For the converting step, the model should be pre-processed by reversing certain loop edges to break cycles. Splitting each MEME (multi-entry and multi-exit) aims to simplify the complexity of the model.
(3) *Converting Step.* Based on the pre-processed model, we can match and record each node with its goal nodes for structural unfolding, which is different from the unfolding of Petri nets [8]. The aim of unfolding is to create a strictly structured model with repetitive or omissive polygons (Sect. 4.4).
(4) *Post-processing Step.* This step is the reverse of pre-processing. The gateways that were split earlier are merged, and the edges that were reversed are reset.
(5) *Text generation Step.* After obtaining the post-processed model, we can generate the textual description by Deep Syntactic Tree analysis, cf. [3,15] for details. As for repetitive or omissive polygons, we complement their behavior descriptions by attaching additional messages.

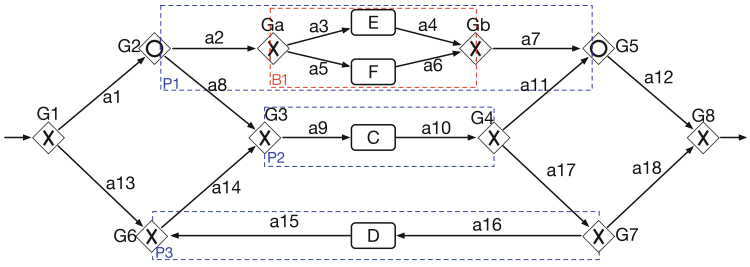
In order to overcome the drawbacks of Hen in handling Rigid components, we propose the Goun algorithm. Here, we first define TLGG with respect to the top level. Goun will run recursively so that nodes on other levels can be processed in the next successive iterations.

Definition 4 (Top-Level Gateway Graph). *Given a Rigid component node in an EPST, we treat its direct children as Polygon components. A top-level gateway graph (TLGG) is a triple* $M_{TLGG} = (G, F, \nu)$ *where:*

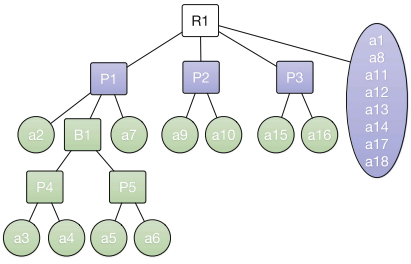
- G *is a finite set of gateway nodes.*
- F *is a finite set of directional edges connecting two different nodes in* G*.*
- ν *is a binary function to mark whether a node in* G *can cause concurrent behavior or not.*

To illustrate, we give an example in Fig. 5(a) of a NFC (non-free choice) model with a loop structure that Hen fails to handle. This whole model belongs to a Rigid component, whose EPST is shown in Fig. 5(b). For ease of presentation, eight independent edges are shown in one ellipse on the right. Note that the light green parts in Fig. 5(b) are not at the top level of the tree. Moreover, the top level (depth = 1) components are highlighted in blue. In general, we can treat every blue EPST node as a Polygon. Thus, this Rigid component contains 11 top-level Polygons.

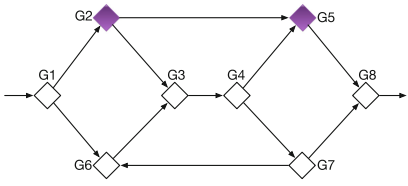
The TLGG of the original model is derived in Fig. 5(c). In this figure, only the concurrent (or parallel) gateway nodes are shown in purple, while the "non-concurrent" nodes (e.g. $G3, G4$) are in white. The non-top-level parts of the original model (e.g. $B1, E, F$, at level 2) are not of interest here and were removed.



(a) An example process model



(b) The EPST of (a)



(c) The TLGG of (a)

Fig. 5. An example process model and its EPST and TLGG (Color figure online)

4.1 Pre-processing

Loop edges in a model are edges which can cause loop structures [6]. In general, any edge attaches to its source node $source(a)$ and target node $target(a)$. The in- and out-degrees of a node g are $|\bullet g|$ and $|g \bullet|$.

For the conversion step, the model should be pre-processed to create an acyclic model. We trivially traverse TLGG by breadth first search (BFS) to number every node with its traversed depth (td). If there are two nodes g_i, g_j and an edge a_l, where $source(a_l) = g_i \ \wedge \ target(a_l) = g_j \ \wedge \ td(g_i) > td(g_j)$, then a_l is marked as a loop edge *and* its direction is reversed.

Further, if $|\bullet g_k| > 1 \wedge |g_k \bullet| > 1$, i.e. g_k is a MEME node, then g_k is split into two nodes g_{ka} and g_{kb}, where $|\bullet g_{ka}| = |\bullet g_k| \ \wedge \ |g_{kb} \bullet| = |g_k \bullet| \ \wedge \ (g_{ka}, g_{kb}) \in F$.

As a result, the TLGG contains no loop structure so that we can get dominators and goals from TLGG, which will be the basis of Goun.

Definition 5 (Dominator, Goal). *For a pair of nodes A, B in a TLGG, if starting from an arbitrary edge originating from A, there always exists one path ending at B, then we say that A is B's pre-dominator and B is A's post-dominator. If the post-dominator set of A is* $A_D = \{A_{D1}, A_{D2}, \ldots, A_{Dn}\}$, *and there exists i* $(1 \leq i \leq n)$ *such that* $|A_{Di} \bullet| = 0 \wedge len(A, A_{Di}) = min\{len(A, A_{Dj})\}$ *for any j* $(1 \leq j \leq n)$, *we say* A_{Di} *is a goal node of A.*

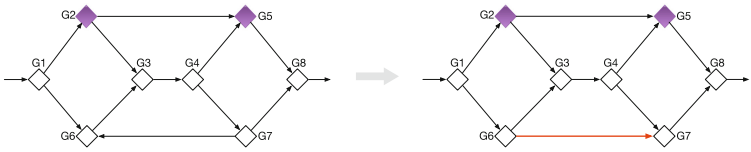


Fig. 6. An example of the pre-processing step

Example 5. After pre-processing the TLGG in Fig. 6, the direction of the loop edge $(G7, G6)$ has been reversed. Note that $G1$ has two out edges in the pre-processed model. Starting along both edges, one can reach all the nodes $G3, G4, G5, G7$ and $G8$. Therefore, the set of these five nodes is the post-dominator set of $G1$. Note that $|G8 \bullet| = 0 \Rightarrow goal(G1) = \{G8\}$. When a certain node Gi has many goal nodes, they all belong to the goal set of Gi.

4.2 Converting

This part is the most crucial step in Goun. We match and record each node with its goal node set for structural unfolding [6–8]. The purpose of matching and unfolding is to create a strictly structured model with repetitive and omissive edges (introduced in Sect. 4.4). More details are given next.

Matching. In this part, we match every valid TLGG node with its goal set. Goun begins at collecting nodes without incoming edges (*entry list*) and nodes without outgoing edges (*exit list*). For a valid entry node, there should be at least one exit node corresponding to it. If the sizes of the entry and exit lists are both equal to one, then the unique entry u and the unique exit v are matched and recorded, meaning v is the goal node of u. As for multiple exit nodes, we use a heuristic rule to guide the matching procedure. For a node u_1 and two exit nodes $v1$ and $v2$, the *len* function calculates the minimum length paths that originate from $u1$ to $v1$ and $v2$. If $len(u1, v1) < len(u1, v2)$, then $v2$ is no more a goal of $u1$.

The node matching problem is slightly similar to the parentheses matching problem. At every step, we delete matched nodes and their connecting edges. Subsequently, whether the TLGG is still reasonable should be taken into consideration. If the exit (entry) list of the remaining graph contains concurrent nodes (purple), and its entry (exit) list does not, then an unreasonable situation is said to occur. If so, we reset the deleted source nodes to keep the reasonableness of the current graph intact.

After every deletion step, a simplification operation is performed. We trivially regard ZEZE (zero entry and zero exit) and SESE nodes as invalid nodes because they both do not change the behavior of the present model and can simplify the model if removed. Thus, the algorithm checks whether an invalid node g in TLGG such that ($|\bullet g| = |g \bullet| = 0$ or $|\bullet g| = |g \bullet| = 1$) exists, and, if so, erases it.

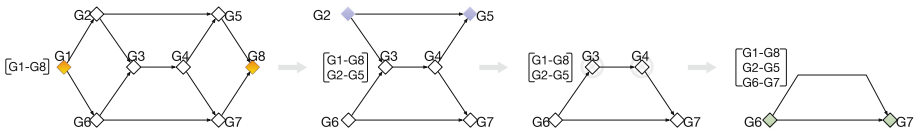


Fig. 7. An example of the matching step

In Fig. 7, $G8$ is marked as the goal node of $G1$ since they are the entry and exit nodes in the TLGG. After deleting them six nodes are left, and $G2$ and $G6$ need to be matched because they are entry nodes now. Note that $G2$ can always reach $G5$ and $G6$ can always reach $G7$, thus $goal(G2) = \{G5\}; goal(G6) = \{G7\}$. If a conflict occurs in matching, the *minimum path length first* heuristic rule should apply. In the simplification step, the SESE nodes $G3$ and $G4$ are erased. Lastly, $G7$ is marked as the goal node of $G6$; thus, the goal set is: $\{(G1, \{G8\}), (G2, \{G5\}), (G6, \{G7\})\}$.

Unfolding. Based on the matched node set, this step aims to create a structured process model with repetitive and omissive edges which is essential for structured text generation. Specifically, a *repetitive edge* is one that exists in the original model once, and in the unfolded model more than once (i.e. it is unfolded many times). An *omissive edge* exists in the original model but not in the unfolded model.

Goun begins at creating reachable paths from node u to v. In every path, it checks whether there is a goal node, and, if so, it skips all the succeeding nodes of the goal node, and continues recursive unfolding. In doing so, some nodes may be copied many times to avoid generating a Rigid component. If not, it unfolds every gateway directly. For every edge in the current path, the corresponding polygon is added.

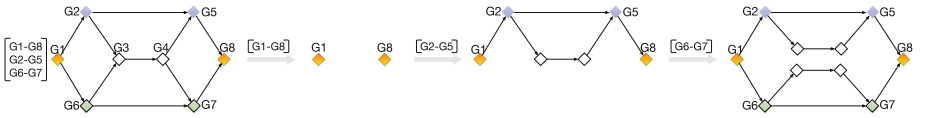


Fig. 8. An example of the unfolding step

In Fig. 8, after unfolding $(G1, \{G8\})$ (at level 1), there are two nodes (yellow) in the unfolded model. Then, after unfolding $(G2, \{G5\})$ and $(G6, \{G7\})$ (at level 2), there are ten nodes in the unfolded model. When all levels are similarly unfolded, the model is strictly structured, i.e. the EPST of it contains no Rigid components.

4.3 Post-processing

This step performs pre-processing in reverse. The gateways which were split should be merged and the edges which were reversed should be restored. Guided by the matched node set $\{(G1, \{G8\}), (G2, \{G5\}), (G6, \{G7\})\}$, we have unfolded and post-processed the original model as Fig. 9 shows. This generated model is strictly structured compared with the one in Fig. 5(a). Note that some activities can be generated twice or more; therefore, some elements appear in the unfolded model many times (i.e. they are repetitive edges).

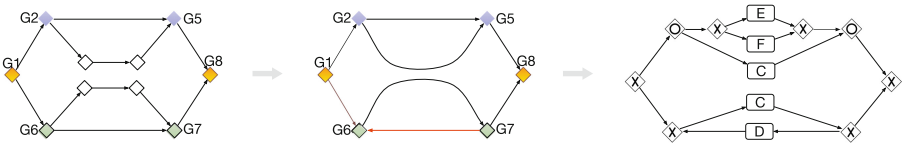


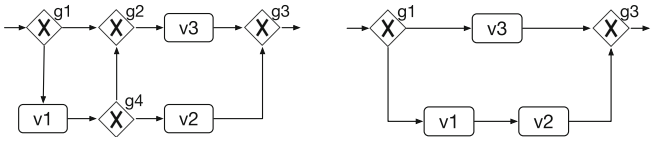
Fig. 9. An example of the post-processing step

4.4 Text Generation

Goun assumes that all generated statements capturing the new elements are correct and well-placed within the text. e.g. a text annotation should directly follow the sentence(s) of its corresponding node and be placed in parentheses.

Transformation of Trivial. After traversing an EPST until its Trivial components are found, we can generate a natural language sentence by Deep Syntactic Tree analysis of every trivial component [3,5]. Goun detects whether all elements are attached, and if so, their corresponding templates are loaded.

Adding Additional Description. Text generation consists of three further aspects as follows.



(a) An example Rigid part (b) The unfolded model of (a)

Fig. 10. An example model with an omissive edge and its unfolded model

(1) *Body text.* If the model contains no Rigid component, we can employ the traditional text generation method to solve all the components [3]. The description of this part needs to be decorated with statements for the behavior of other elements to avoid information loss.
(2) *Repetitive behavior.* From our experience, activities described more than once do not need additional language templates.
(3) *Omissive behavior.* For some models, those edges which can cause different behavior are dropped after unfolding. In Fig. 10(a), after matching $g1$ and $g3$, its unfolded model is shown in Fig. 10(b). Note that edge $(g4, g2)$ is an omissive edge since it cannot be mapped in Fig. 10(b). Accordingly, edges that cannot be mapped in the unfolded model are recorded. Every omissive edge should be clarified through a language template. Taking Fig. 10(a) as an example, the additional statement is, *"After the 'role' does 'v1', it can also do 'v3'"*.

Having described the Goun algorithm, we give an example to show how it works.

Example 6. The structured Goun text of the *pool*1 in Fig. 1 is:

01. *The 'pool1' process begins when the 'role1' does 'v1'. Then one of the following branches is executed:*
02. • *The 'role1' does 'v2'.*
03. • *The 'role1' does 'v3'.*
04. *Once one of the above branches is executed, there is a region which allows for different execution paths:*
05. *One of the following branches is executed:*
06. • *The 'role1' does 'v4'.*
07. • *The 'role1' does 'v5'.*
08. *One of the following branches is executed:*
09. • *The 'role1' does 'v6'.*
10. • *The 'role1' does 'v7'.*
11. *However, the region allows for other behavior.*
12. • *After the 'role1' does 'v6', it also can do 'v5'.*
13. *Once the region is executed, the 'pool1' process is finished.*

5 Experimental Evaluation

We have implemented Goun[1] based on an open source BPM tool named jBPT[2]. In this section, we compare Goun with Hen [3]. All the experiments were run on a Macbook Pro with Intel Core I7 CPU@2.2 GHz, 16G DDR3@1600 MHz, and OS X 10.11.4 operating system.

5.1 Experimental Setting

We created[3] artificial test cases [9] and collected real-life test cases from different sources such as books (24.6%, textbook), online tutorials (7.4%, academic), papers (27.6%, academic), modeling tools (5.9%, industry), enterprises (30.5%, industry), and so on. We divided them into 10 datasets based on their domain. Their structural characteristics vary from easy (with just one activity) to complex (with hundreds of nodes with sub-processes, message flows, boundary events, etc.). The ratio of real-life cases is 61.76%. To eliminate artificial factors as much as possible, these 130 models[4] from different sources are randomly selected and sorted. Afterwards, we run the Hen and Goun algorithms to generate two separate texts for each case. Finally, text coverage, time cost, text property and user evaluation are measured as well.

5.2 Handling Ability

We analyzed the features of the Hen and Goun algorithms, in the context of both simple and complex models, and present a qualitative comparison in Table 1 (symbols '+' for supported; '−' for not supported; '±' for partial support).

Table 1. Comparison on handling abilities of Hen and Goun

Item	Hen	Goun
Structural model	+	+
Gateway	±	+
Artifacts	−	+
Message flow	−	+
Subprocess	±	+
Text annotation	−	+
Boundary events	±	+
Loop in Rigid	−	+
NFC models	−	+

[1] https://github.com/qc529491527/ModelToText/tree/master/SourceCodes.
[2] https://code.google.com/archive/p/jbpt/.
[3] www.signavio.com.
[4] https://github.com/qc529491527/ModelToText/tree/master/TestModels.

The results show that Goun enhances the completeness of structural descriptions, and the patterns it can handle is a superset of those of Hen.

5.3 Quantitative Comparison Results

We compared the coverage rate of Hen and Goun algorithms as shown in Fig. 11 (coverage-sources graphs). The element coverage, activity coverage, gateway coverage, flow coverage, message flow coverage and artifact coverage, respectively,

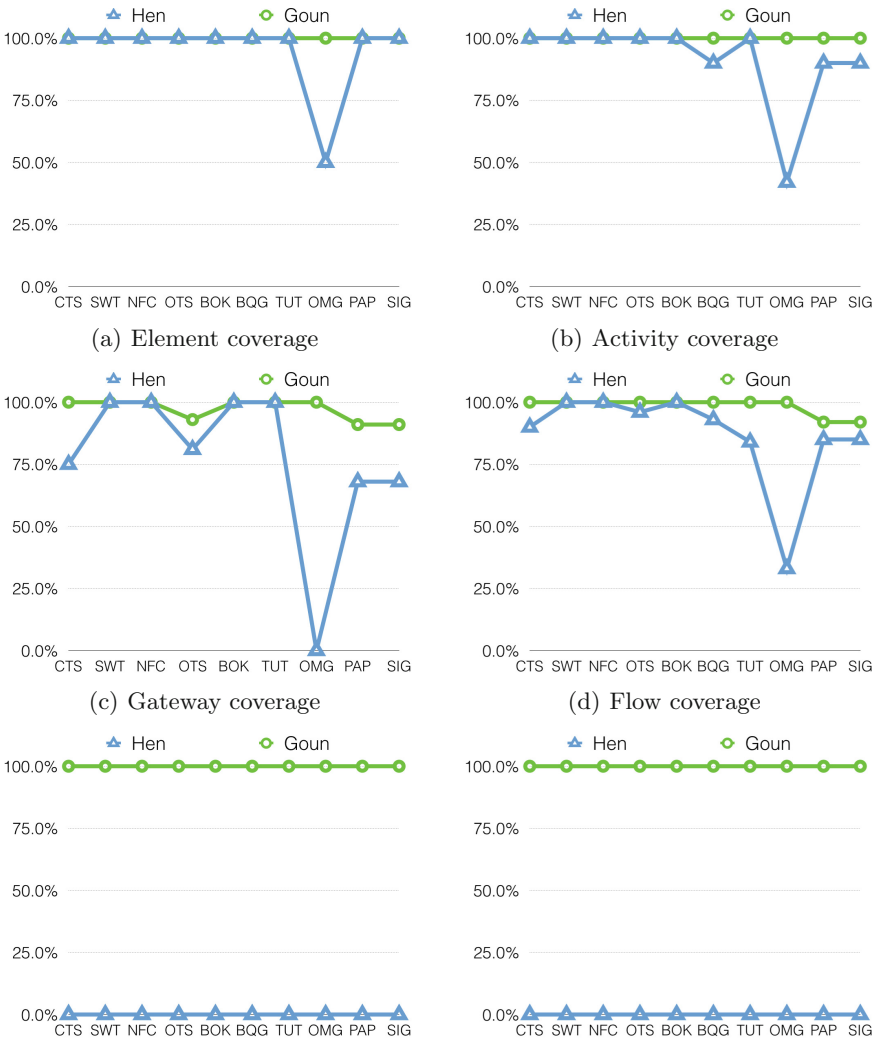


Fig. 11. The comparison results on text coverage rates

represent the ratio of covered elements, activities, gateways, control flows, message flows and artifacts to their maximum values. As for the BQG source in Fig. 11(c), since this set contains no gateway elements it is not shown. Next, we reverse constructed the process models from the generated text manually [12], and then compared the models with the "gold" models created manually. From the single source or total statistic, we can clearly see that the coverage rates of Goun are either equal to or more than those of Hen. The below-100% performance of Goun is explained by the fact that Goun ignores invalid gateways, i.e. SESE gateways, to avoid overfitting; thus, the coverage rates of these models are less than 100%. Note that for message flow coverage and artifact coverage, Goun achieved 100% while Hen got 0%. In sum, if we use $cov(f)$ to denote the coverage of method f, we can conclude that $cov(Hen) \leq cov(Goun)$.

Finally, the time cost, text property and user evaluation are measured. First, we tested these datasets and got an average generation time of around 1162 ms. Then, from the text property experiments, we find that the Goun text is more detailed than Hen's based on word count, number of sentences, and so on. Lastly, we randomly polled almost 22 BPM (business process management) researchers and experts, from graduate students to professors, to independently understand and evaluate the consistency scores of all the 130 model-text pairs on a 1–5 scale. They were not given any prior guidance. The average score of Goun (4.58) is 1.59 points higher than that of Hen (2.99).

6 Related Work

The interdisciplinary, natural language text generation of process model, has been studied for several years. The model-to-text generation system proposed in [3] neglects many complex elements and structures which may be used in enterprises. Our approach is developed based on the Hen architecture. It not only overcomes some drawbacks but also makes the text generation system more robust, and generates texts that are more compliant with the original models.

The tool *Realizer* takes a Deep Syntactic Tree as input, and outputs its English description [15]. We used the tool for sentence generation.

Meziane et al. developed an application which generates texts for UML class diagrams [10]. Malik et al. provide SBVR rules to map corresponding elements to its language templates [16]. We applied some of their generation rules in our system with modifications.

The simplification or conversion of model structure is valuable [6–8] since converted structures are in some ways easier to understand. Our approach applied the ideas of behavioral equivalence and proper complete prefix unfolding.

Model construction is the reverse of text generation [5,12,13]. Our approach can be applied to evaluate such systems also, especially when using machine learning techniques since raw texts are required as input. Under this circumstance, our system will enormously reduce the manual efforts of modeling.

Furthermore, consistency checking between models and texts is proposed in [11]. This technique can also be applied in a generation system to quantitatively evaluate multiple generated results.

7 Conclusion

In this paper, we proposed a new data structure EPST, and an algorithm for transforming general business process models to their textual descriptions. The EPST is an extended version of RPST since it can describe more patterns of process models. In addition, we implemented the Goun method which computes the goal sets of the original models, unfolds them into strictly structured ones and generates corresponding structured texts for them. Afterwards, test cases were collected to evaluate the Goun and Hen transformation methods along different dimensions. Experiments show that Goun not only extends the expressive patterns, but also presents a strictly structured text to readers. Furthermore, we showed that it improves the text coverage and reduces information loss.

Although the Goun technique can handle arbitrary Rigid components with loops and possesses good extensibility, the Goun texts are expressed by similar expression templates and are somewhat longer than those of Hen. In the future, we will try to decorate and shorten generated texts, and also do more extensive testing using enterprise models. In the meantime, we will give formal proofs to validate the properties and abilities of Goun. We will also employ automatic consistency checking to avoid subjective evaluation [11,13].

Acknowledgement. The work was supported by the National Key Research and Development Program of China (No. 2016YFB1001101) and the National Nature Science Foundation of China (Nos. 61472207, 61325008 and 71690231).

References

1. BPMN Task Force, Business Process Model and Notation (BPMN) Version 2.0, Object Management Group, 2011 (OMG Document Number formal 2011-01-03)
2. Vanhatalo, J., Vlzer, H., Koehler, J.: The refined process structure tree. Data Knowl. Eng. **68**(9), 793–818 (2009)
3. Leopold, H., Mendling, J., Polyvyanyy, A.: Supporting process model validation through natural language generation. IEEE Trans. Softw. Eng. **40**(8), 818–840 (2014)
4. Leopold, H., Mendling, J., Polyvyanyy, A.: Generating natural language texts from business process models. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 64–79. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31095-9_5
5. Friedrich, F.: Automated generation of business process models from natural language input. Humboldt University at zu Berlin, Berlin, Germany, Institute of Information Systems (2010)
6. Polyvyanyy, A., Garca-Bauelos, L., Dumas, M.: Structuring acyclic process models. Inf. Syst. **37**(6), 518–538 (2012)
7. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**, 121–141 (1979)
8. McMillan, K.L., Probst, D.K.: A technique of state space search based on unfolding. Form. Methods Syst. Des. **6**(1), 45–65 (1995)
9. Yan, Z., Dijkman, R., Grefen, P.: Generating process model collections. Softw. Syst. Model. **16**, 1–17 (2015)

10. Meziane, F., Athanasakis, N., Ananiadou, S.: Generating natural language specifications from UML class diagrams. Requir. Eng. **13**, 1–18 (2008)
11. van der Aa, H., Leopold, H., Reijers, H.A.: Detecting inconsistencies between process models and textual descriptions. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 90–105. Springer, Cham (2015). doi:10.1007/978-3-319-23063-4_6
12. Schumacher, P., Minor, M., Schulte-Zurhausen, E.: Extracting and enriching workflows from text. In: 2013 IEEE 14th International Conference on Information Reuse and Integration (IRI), pp. 285–292. IEEE (2013)
13. Zhang, Z., Webster, P., Uren, V., Varga, A., Ciravegna, F.: Automatically extracting procedural knowledge from instructional texts using natural language processing. In: International Conference on Language Resources and Evaluation (2012)
14. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes, 1st edn. Springer, Heidelberg (2011)
15. Lavoie, B., Rambow, O.: A fast and portable realizer for text generation systems. In: Proceedings of the fifth conference on Applied natural language processing. Association for Computational Linguistics (1997)
16. Malik, S., Bajwa, I.S.: Back to origin: transformation of business process models to business rules. In: La Rosa, M., Soffer, P., et al. (eds.) BPM Workshops 2012. LNBIP, vol. 132, pp. 611–622. Springer, Heidelberg (2013)