Journal of Visual Languages and Computing 47 (2018) 62–76

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

# A temporal graph grammar formalism

Shi Zhan[a,b], Zeng Xiaoqin[a,*], Zou Yang[a], Huang Song[b], Li Hui[b], Hu Bin[b], Yao Yi[b]

[a] Institute of Intelligence Science and Technology, Hohai University, Nanjing 211100, China
[b] College of Command Information Systems, PLA University of Science and Technology, Nanjing 210007, China

ARTICLE INFO

*Keywords:*
Graphical language
Graph grammar
Temporal specification
Parsing algorithm
Simulation

ABSTRACT

As a useful formalism tool, graph grammars provide a rigorous but intuitive way to specify visual languages. This paper, based on the existing Edge-based Graph Grammar (EGG), proposes a new context-sensitive graph grammar formalism called the Temporal Edge-based Graph Grammar, or TEGG. TEGG introduces some temporal mechanisms to grammatical specifications, productions, operations and so on in order to tackle time-related issues. In the paper, formal definitions of TEGG are provided first. Then, a new parsing algorithm with a decidability proof is proposed to check the correctness of a given graph's structure, to analyze operations' timing when needed, and to make the computer simulation of the temporal sequence in the graph available. Next, the complexity of the parsing algorithm is analyzed. Finally, a case study on an application with temporal requirements is provided to show how the parsing algorithm of TEGG works.

## 1. Introduction

With the increasing use of graphical languages such as entity-relationship diagrams, data or program flowcharts and the business process modeling notations [1–3], etc., a framework for formally defining and analyzing graphical languages is becoming more and more important. For the specification and analysis of graphs composed of nodes and edges, graph grammar [4,5] is ideally a formal but intuitive tool. In view of the role played by a string grammar to string languages, it is not an exaggeration to say that a graph grammar forms the theoretical basis of visual languages [6].

For string languages, formal string grammars are commonly used to facilitate the language definition and its parsing algorithm. For the same reason, graphical languages also need the corresponding formal grammars. However, currently the implementation of a graphical language is usually not the case [7]. This is mostly due to the fact that the extension from one-dimensional string grammars to two-dimensional graphical ones raises several new issues [8] such as the embedding problem, the membership problem and high parsing complexity, etc. Currently, there are a number of graph grammars and their applications appearing in literature.

According to the different types of productions, graph grammars are divided into two categories: context-free and context-sensitive. The main differences between these two types are the production constraints and the expressive power. On the one hand, a context-free grammar requires that only a single non-terminal node is allowed on the left side of a production [9]. In the early stage of research, many context-free grammars were proposed [10–21]. Since the production of these graph grammars is quite simple, their expressive power is limited, which hinders the scope of their applications. On the other hand, with the increasing demands of intricate applications, context-sensitive graph grammars, such as CS-NCE (context-sensitive neighborhood controlled embedding graph grammar) [22–24], LGG(layered graph grammar) [25], RGG(reserved graph grammar) [7], SGG(spatial graph grammar) [26–28], EGG(edge based graph grammar) [29], have been appearing with more expressive power, which allow the left side of a production to be a graph instead of a node.

In 1997, Rekers and Schürr [25] proposed a graph grammar formalism called LGG for specifying visual languages. Different from most previous graph grammars, LGG is context-sensitive and can intuitively specify and parse a wide range of graphical visual languages [8]. First, productions in LGG differ widely from others by introducing some context nodes that will not be replaced in a reduction or derivation operation. Second, in order to solve the embedding problem, LGG puts a restriction on the definition of a redex by requiring that nodes in the redex, which are isomorphic with non-context nodes in productions, can only link to the context nodes in host graphs. This restriction ensures there is no creation of dangling edges when redexes in host graphs are replaced. However, the context nodes make the productions difficult to design and its parsing algorithm complicated with exponential time complexity.

To overcome the weaknesses of LGG, Zhang et al. [7] proposed another context-sensitive grammar called RGG, which is founded on the basis of LGG, but offers an improvement over it. RGG redefines the

* Corresponding author.
*E-mail address:* xzeng@hhu.edu.cn (X. Zeng).

https://doi.org/10.1016/j.jvlc.2018.06.003
Received 6 June 2015; Received in revised form 29 April 2018; Accepted 11 June 2018
Available online 18 June 2018
1045-926X/ 

structure of graphs in which each node has a two-level structure including a super vertex and a vertex. The vertexes in different nodes can be connected by edges. RGG introduces a marking mechanism to tackle the embedding problem, in which a unique label is used to identify all of the context elements. In addition, with the introduction of selection-free productions to graph grammars, a Selection-Free Parsing Algorithm (SFPA) is put forward for a confluent RGG, which only needs to consider one parsing path and thus can efficiently parse graphs with a polynomial time complexity [7]. Due to the RGG's characteristics, its application is more extensive in solving some new issues, such as Specification and Discovery of Web Patterns [30], Program Behavior Discovery and Verification [31], Visual XML Schemas [32,33], Design Pattern Evolution and Verification [34], and Generic Visual Language Generation Environments [35], etc. Next, RGG is extended by introducing some spatial notations and mechanisms. The new spatial specifications of the extended RGG, called SGG, can qualitatively express the spatial relationships of objects and may reduce the parsing complexity.

In 2008, Zeng et al. [29] proposed another a context-sensitive graph grammar formalism called EGG, which overcomes some of the shortcomings of LGG and RGG without the loss of expressive power. The main advantage of EGG over LGG is its simplified context expression by using an edge instead of a node, and that over RGG is its simplified node structure which only has one level instead of two. Based on EGG, a new attempt of transformation between BPMN and BPEL [36] was provided, which presented mapping steps and a parsing algorithm.

Up until this point, graph grammars have mostly concentrated on the syntactic aspects of graphical languages rather than the semantic aspects. Meanwhile, for graph transformation, some recent articles have focused on semantics. Szilvia Gyapay et al. [37] developed a model of time in attributed graph transformation systems inspired by the concepts of TER nets. The idea is to use the attributes of nodes as time stamps to represent the “age” of these nodes, and to update these time stamps whenever a rule is applied. Although time semantics is introduced into graph transformation, this model is difficult to solve time-related problems only with the time stamps. Besides, graph transformation is not suitable for defining graph languages and checking the correctness of a graph's structure. Meanwhile, the model of timed automata [38] proposed by Rejeev Alur has established a standard formalism for modeling the behavior of real-time systems. The timed automata adds time constraints into a finite automata, and it accepts timed words (strings of symbols) in which a real-valued time of occurrence is associated with each symbol.

Graph grammars have advantages in dealing with graph structure. However, they are difficult to deal with schedule tasks, such as a project progress schedule, business process schedule etc., due to their involvement with some temporal semantics like time sequence. Consequently, if some temporal semantic mechanisms can be introduced into graph grammars and their operations, the expressive and functional powers of graph grammars could be increased to meet more of the requirements of time-related applications.

To highlight the most common requirements, this paper takes a business process described by Business Process Modeling Notation (BPMN), a standard notation for specifying business processes [39], as an example to illustrate the requirements. Fig. 1 shows a simple example of a designed business process by BPMN, in which a diamond node with notation + indicates a parallel gateway, and × indicates an exclusive gateway. For the sake of simplicity, only these two types of gateways are used in this paper, and business processes with these gateways are called simple business processes. In Fig. 1, the process includes six activities, and each activity needs a given time period. Conceptually, the example could be regarded as an abstraction of any scheduled process, and this will be used throughout the whole paper.

From the graph in Fig. 1, it is evident that there are usually the following requirements:

1. Analyzing the correctness of the graph's structure.
2. Searching for one possible path and computing its time duration, such as pathes $a \rightarrow b \rightarrow c$, $a \rightarrow e \rightarrow f$, $a \rightarrow d$ and so on.
3. Searching for all possible paths and computing their time durations.
4. Finding the shortest time to the completion of the process, which should be 18 days.
5. Determining the start time of a given activity, such as the start time of activity *f*, which is the 8th day.

Obviously, existing graph grammars could be difficult to meet the above time-related requirements but it is hoped that the introduction of a temporal mechanism into graph grammars could be helpful to them. So, in this paper, our research work is to extend the existing graph grammar with a new mechanism for expressing temporal semantics and performing temporal operations. It is expected that the extension could be applicable, by means of a series of grammatical operations, to simulate the running of a schedule, and to check its correctness in both structure and timing in advance.

The contribution of the paper is as follows. First, a new graph grammar formalism TEGG is proposed, which introduces a temporal specification mechanism to a graph grammar. Second, a new parsing algorithm is provided, which could be used to check the correctness of the graphs' structure, to analyze the temporal semantics in a graph and to make the computer simulation of the temporal sequence in the graph available. Third, several properties of the parsing algorithm are explored and theoretically proven, and the complexity of the parsing algorithm is analyzed. Finally, the proposed temporal mechanism is general enough and could easily be applied to other graph grammars, regardless of whether they are context-free or context-sensitive.

The rest of the paper is organized as follows. Section 2 introduces some basic definitions and operations of EGG as the preliminaries to Section 3, which presents the formal definitions of TEGG with the introduction of some new mechanisms, such as an attribute set, some T-productions, etc. Based on TEGG, Section 4 provides a parsing algorithm with its decidability proof and analyzes the complexity of the parsing algorithm. In Section 5, a case study on a time-related application is given. Finally, Section 6 concludes the whole paper.

## 2. Preliminaries

Different from other graph grammars, EGG introduces some new concepts, such as the dangling edge, core graph and so on, to deal with the embedding problem. The so-called dangling edge means an edge that only has one node as its source or target node, and the core graph is a sub-graph in which all dangling edges are removed from a graph. In EGG, dangling edges with unique marks are allowed in both sides of a production to indicate the context. Based on these concepts, the embedding problem could be solved in grammatical operations by avoiding ambiguity. Below are some formal definitions of EGG, which are fundamental to the discussion of TEGG in the next section.

**Definition 2.1.** *A graph G on a given label set Lis defined as a quintuple, denoted as* $G \equiv (N, E, l, p, q)$, *where*

- $N$ is a set of nodes, which can be expressed as $N = N_T \cup N_N$, where $N_T$ represents a set of terminal nodes and $N_N$ represents a set of non-terminal nodes.
- $E$ is a set of edges.
- $l$: $N \cup E \rightarrow L$ is a mapping from $N$ and E to $L$, and $L$ can be expressed as $L = L_n \cup L_e . L_n = T \cup \overline{T}$ is a set of node labels, where $T$ represents a label set of $N_T$ and $\overline{T}$ represents a label set of $N_N$; $L_e$ is a set of edge labels.
- $p$: $E \rightarrow N$ is a mapping from $E$ to $N$, indicating the source node of an edge.
- $q$: $E \rightarrow N$ is a mapping from $E$ to $N$, indicating the target node of an edge.

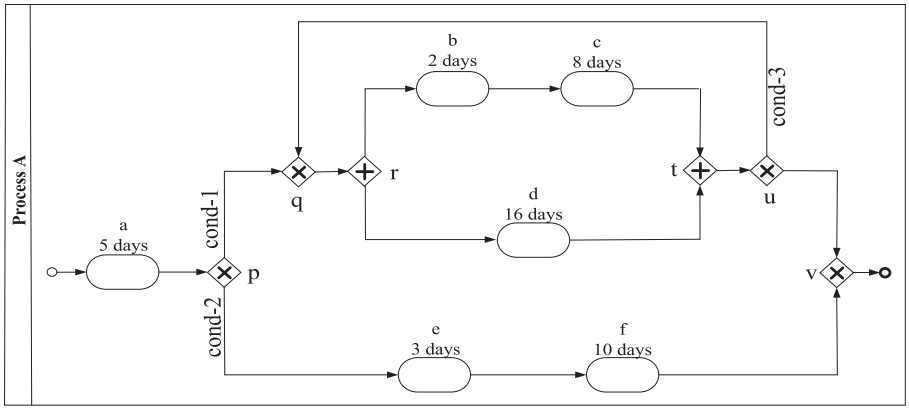



**Fig. 1.** A business process.

Graphs defined by the above definition are the most common ones which do not include dangling edges. Then, in order to represent different nodes in this paper, each node needs to be given a unique identifier called *id*. A node with an *id* is denoted as $n_{id}$ or $n_i$ ($id = i = 0, 1, 2, ...$). It should be noted that a node's *id* is not part of the above definition, and is just used to indicate the node. If the label of node $n_i$ is $A$, this label can be denoted as $n_i.A$. Obviously, in a graph there may be several nodes which have the same label.

To intuitively express node $n_i$ with the label in a graph, the following graphic notations could be used, which are shown in Fig. 2. As we can see from Fig. 2, a graphic notation of a node is a rounded rectangle, and its *id*, label, or $n_i.A$ should be placed in the rectangle.

To deal with a business process by EGG, it should be firstly mapped to a corresponding graph of EGG, also called EGG graph. For the example of BPMN in Fig. 1, the following labels of nodes in its corresponding EGG graph are given.

1. "*xor*" of a node represents an exclusive gateway.
2. "*and*" of a node represents a parallel gateway.
3. "*x*" of a node represents an atomic activity.

Obviously, a node with the above labels is a terminal node in the EGG graph. So, for the example in Fig. 1, these labels should belong to set $T$, that is to say, $T = \{xor, and, x\}$. The notations "*T.xor*", "*T.and*", and "*T.x*" can be used to specifically point out the label of a terminal node. For example, if the label of node $n_i$ is "*xor*", this label can be denoted as $n_i.T.xor$ or $n_i.xor$. For a business process of BPMN in this paper, its corresponding graph of EGG has one type of non-terminal node, which could be labeled *s*. That is to say, $\overline{T} = \{s\}$. So, the label *s* can be directly used to represent a non-terminal node without causing ambiguity, and this label can be denoted as $n_i.s$.

If a graph $G$ of EGG is directed, the notation $d_{in}(n)$ is used to represent the numbers of edges in which target nodes are *n,* and the notation $d_{out}(n)$ is used to represent the numbers of edges in which source nodes are *n*. In the $G$, any node *n* with $d_{out}(n) = 0$ is called the end node, while n with $d_{in}(n) = 0$ is called the start node. For expression convenience, notations like *G.N, G.E, G.L, G.l*, and so on are used to represent the corresponding parts of $G$ in the following definitions.

**Definition 2.2.** *A graph with dangling edges $\overline{G}$on a given label setsLis defined as a septuple, denoted as $\overline{G} \equiv (N, \hat{E}, M, l, \overline{p}, \overline{q}, m)$, where*

- $N$, $L$, and $l$ are the same as that in Definition 2.1.
- $\hat{E}$ is a set of edges, which can be expressed as $\hat{E} = E \cup \overline{E}$, where $E$ is the same as that in Definition 2.1 and $\overline{E} = \overrightarrow{E} \cup \overleftarrow{E}$ is a set of dangling edges. $\overrightarrow{E}$ represents a set of edges which only have source nodes, while $\overleftarrow{E}$ represents a set of edges which only have target nodes.
- $M$ is a unique marks' set of $\overline{E}$, also called marks.
- $\overline{p}: (E \cup \overrightarrow{E}) \to N$ is a mapping for source nodes of an edge.
- $\overline{q}: (E \cup \overleftarrow{E}) \to N$ is a mapping for target nodes of an edge.
- $m: \overline{E} \to M$ is an injective mapping from $\overline{E}$ to $M$, indicating the mark of a dangling edge.

From the above definition, we can see that a dangling edge in the graph $\overline{G}$ only has one node as its source or target node, which is shown in Fig. 3.

Unlike Graph Theory, the graphs defined above could have dangling edges, which are employed to represent context in EGG. To clearly describe grammatical operations, the following definitions are necessary.

**Definition 2.3.** *Two graphs $G_1$ and $G_2$ are isomorphic, denoted as $G_1 \approx G_2$, if and only if there exist two bijective mappings $f_N$: $G_1.N \leftrightarrow G_2.N$ and $f_E$: $G_1.E \leftrightarrow G_2.E$, and the following conditions are satisfied:*

- $\forall\, n((n \in G_1.\, N) \to (G_1.\, l(n) = G_2.\, l(f_N(n))))$.
- $\forall\, e((e \in G_1.\, E) \to (G_1.\, l(e) = G_2.\, l(f_E(e))))$.
- $\forall\, e(e \in G_1.E \to ((G_2.\, p(f_E(e)) = f_N(G_1.\, p(e))) \wedge (G_2.\, q(f_E(e)) = f_N(G_1.\, q(e)))))$.

**Definition 2.4.** *A graph W without dangling edges is the sub-graph of G, if and only if the following conditions are satisfied:*

- $(W.\, N \subseteq G.\, N) \wedge (W.\, E \subseteq \overline{G}.\, E) \wedge (W.\, L_e \subseteq \overline{G}.\, L_e) \wedge (W.\, L_n \subseteq \overline{G}.\, L_n)$.

**Definition 2.5.** *A graph W is the core graph of $\overline{G}$, denoted as $Core(\overline{G}) = W$, if and only if the following conditions are satisfied:*

- $(W.\, N = \overline{G}.\, N) \wedge (W.\, L_e \subseteq \overline{G}.\, L_e) \wedge (W.\, L_n \subseteq \overline{G}.\, L_n)$.
- $\forall\, e(e \in W.\, E \to (e \in \overline{G}.\, E \wedge \overline{p}(e) \in W.\, N \wedge \overline{q}(e) \in W.\, N))$.

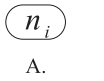



**Fig. 2.** Different expressions of a node with label A.

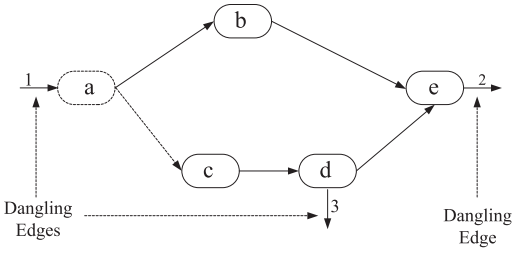



**Fig. 3.** A graph *g* with dangling edges.

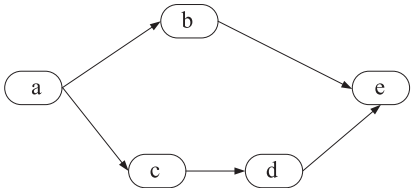



**Fig. 4.** A core graph of *g*.

The core graph in the above definition is a sub-graph in which all dangling edges are removed from a graph. The graph in Fig. 4 is the core graph of the one in Fig. 3.

**Definition 2.6.** *If graph W is the sub-graph of graph G, and Q is a graph with dangling edges, W is a redex of G with respect to Q, denoted as* $W \in Redex(G, Q)$*, if and only if there exists a bijective mappings* $f_N$*:* $W.N \leftrightarrow Q.N$ *and the following conditions are satisfied:*

- $W \approx Core(Q)$.
- $\forall\, n_i (n_i \in W.N \rightarrow ((d_{in}(n_i) = d_{in}(f_N(n_i))) \wedge (d_{out}(n_i) = d_{out}(f_N(n_i)))))$.

To understand the above definition more easily, an example is given in the following three figures. Fig. 5 shows a graph *Q* with dangling edges, and Fig. 6 presents a given host graph *G*. Obviously, the graph *W* in Fig. 7 is the sub-graph of *G*. From the above Definition 2.6, *W* is a redex of *G* with respect to *Q*.

**Definition 2.7.** *A production p is the expression* $\overline{G_L} \Leftrightarrow \overline{G_R}$*, which consists of a left graph* $\overline{G_L}$ *and a right graph* $\overline{G_R}$*, where*

- $(\overline{G_L}.M = \overline{G_R}.M) \wedge (M \subset \{0, 1, 2, ...\})$.

In addition to dangling edges and core graphs, a marking mechanism is also introduced into EGG. Similar to the mark of a vertex in RGG, the mark of a dangling edge in EGG is used to indicate the context. Each pair of corresponding dangling edges in the left and right graphs of a production is marked by a unique integer to maintain their relationship. It is the marking mechanism of EGG that solves the embedding problem. By means of dangling edges and their corresponding marks, the replacement of a redex with either a left or right graph in a production can be done without any ambiguity. Fig. 8 is an example of a set of EGG productions to specify the business process in Fig. 1.

To facilitate understanding, we take the second production in Fig. 8 as an example. On the right side there is a node labeled *T.x*, which is a terminal node and represents an activity in a business process. The left side is a non-terminal node labeled *s*. So, the transformation between any atomic activity and a non-terminal node can be achieved when using this production.

Usually, there are two typical grammatical operations that have reverse functions. One is called L-application (derivation) and the other is called R-application (reduction). The former is used to find a redex that is isomorphic to the left graph of a production and to replace the redex with the production's right graph, which could be used to define a graph language or to perform graph transformations. While the latter is used to find a redex that is isomorphic to the right graph of a production and to replace the redex with the production's left graph, which is the basis of a parsing algorithm for analyzing the structural correctness of a given host graph. More detailed information about the content of EGG is available in the literature [29].

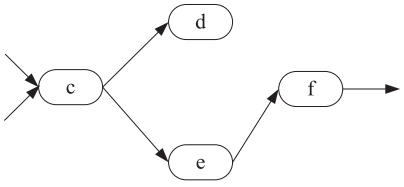



**Fig. 5.** A graph *Q* with dangling edges.

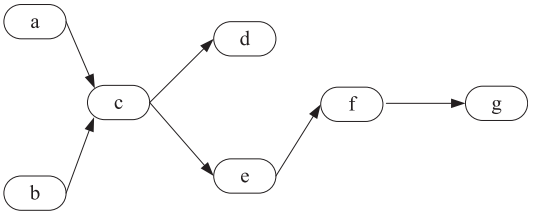



**Fig. 6.** A host graph *G*.

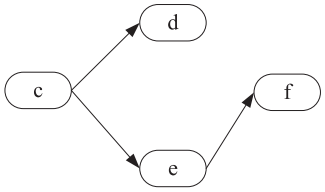



**Fig. 7.** The sub-graph *W* is a redex of *G* with respect to *Q*.

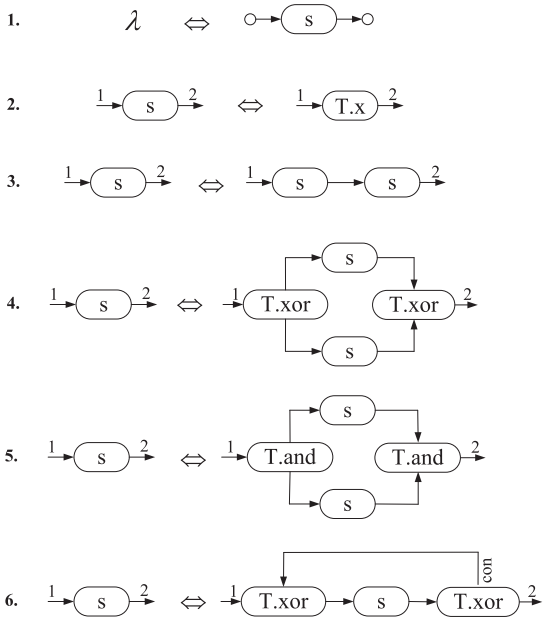



**Fig. 8.** A set of EGG productions defined for business processes.

## 3. Formalism of TEGG

By aiming to allow a graph grammar to have the ability to deal with time-related issues, on the one hand, TEGG inherits the methods of EGG. On the other hand, it introduces a new temporal specification mechanism including a set of temporal attributes for each node and edge in graphs, a global variable for a graph and some time-related productions, called T-productions, to operate on the attributes, and so on. In this way, it is hoped that TEGG will have the ability to tackle time-related requirements in real applications, such as the timing simulation of a system represented by a graph, and to check the structural correctness of this graph. Parallel to the definitions of EGG, the following are the definitions of TEGG.

**Definition 3.1.** *Temporal attributes of a node in TEGG are relevant to time sequence, which can be divided into the following two types of attributes:*

- one is a set of time–related attributes, called non-key temporal attributes, which represents the available temporal interval and

execution duration of each node, such as start time, duration time and so on.
- the other is a status–related attribute, called a key temporal attribute and denoted as a "statusflag", which represents the status of a node.

In TEGG, the key temporal attribute *statusflag* should have finite values. It is noted that the *statusflag* could have more values according to different applications.

**Definition 3.2.** *A node n on a label set $L_n$ is defined as a two-tuple, denoted as $n \equiv (lab, Natts)$, where*

- $L_n$ is the same as that defined in Definition 2.1.
- $lab \in L_n$, representing the label of n.
- Natts is a set of temporal attributes owned by node n, while $Natts = Nka \cup \overline{Nka}$ with $Nka = \{statusflag\}$ being a set of key temporal attributes of n, and $\overline{Nka}$ a set of non-key temporal attributes of n.

The above definition introduces the *Natts* for handling temporal semantics. Different applications should include the key temporal attribute of all nodes. For example, to deal with the business process shown in Fig. 1, there is the *statusflag* attribute with the values *WAITING, WORKING*, or *COMPLETED*. According to these three values of *statusflag*, node *n* could have one of the following statuses:

$n.Natts.statusflag = WAITING$, that is to say, *n* is in the waiting status.
$n.Natts.statusflag = WORKING$, that is to say, *n* is in the working status.
$n.Natts.statusflag = COMPLETED$, that is to say, *n* is in the completed status.

To intuitively express the status of a node in a TEGG graph, the following graphical notations of a solid triangle or solid circle are used, which are shown in Fig. 9. The notations are placed in the upper right of a node.

Of course, different applications may require time-related attributes in the $\overline{Nka}$ of the *Natts*. For example, the attributes *starttime* and *durationtime* need to be introduced into each node in Fig. 1, which represent the start and duration time of an activity respectively. For a given activity in Fig. 1, after the value of its *durationtime* need to be given, the value of its *starttime* could be calculated according to the business process including this activity. An explanation of these values' calculations will be provided later.

To deal with a business process by TEGG, it should be firstly mapped to a corresponding graph of TEGG. In TEGG graph, to distinguish each node, a unique identifier also needs to be given for each node, which is called *id*. If a node with an *id* is expressed as $n_i$, the attribute *att* in the set *Nka* of the node can be denoted as $n_i.Nka.att$, or $n_i.Natts.att$, while the notation $n_i.lab$ represents the label.

**Definition 3.3.** *A temporal attribute of an edge in TEGG is also relevant to time sequence, denoted as an "edgeflag", which represents a sequence–related attribute.*

In TEGG, the temporal attribute *edgeflag* should have finite values and it can be seen as the switch that controls the execution of next activity nodes. For the business process shown in Fig. 1, there is an *edgeflag* attribute with the values *PASSING*, or *STOPPING* in the corresponding TEGG graph. The next activities can be executed when the value of the attribute *edgeflag* is *PASSING*, and the next activities can not be executed when the value is *STOPPING*. It is noted that the *edgeflag* could have two or more values according to different applications.

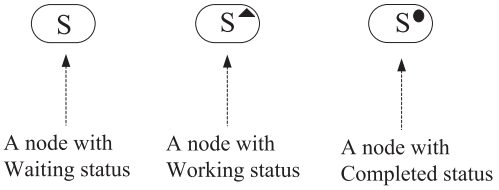



**Fig. 9.** Nodes with waiting, working, and completed statuses.

**Definition 3.4.** *An edge e on a label set $L_e$ is defined as a quadruple, denoted as $e \equiv (lab, n_s, n_t, Eatts)$, where*

- $L_e$ is the same as that defined in Definition 2.1.
- $lab \in L_e$, representing the label of e.
- $n_s$ is a node defined in Definition 3.2, representing the source node of e.
- $n_t$ is a node defined in Definition 3.2, representing the target node of e.
- Eatts is a set of attributes owned by edge e, while $Eatts = Eka \cup \overline{Eka}$ with $Eka = \{edgeflag\}$being a set of temporal attribute of an edge e, called key attributes of e, and $\overline{Eka}$ being a set of attributes which could have finite or infinite values, called non-key attributes of e.

Usually, set $L_e$ may include multiple elements and each element indicates each of edge types in a TEGG graph. This means edges have strong expressive power. For simplicity, the business processes in this paper only contain a control flow of BPMN, so there is only one edge type in the corresponding TEGG graphs. In TEGG, the label of this edge type could be *controlflow*. So, $L_e = \{controlflow\}$.

The *Eatts* is an attributes' set which can be divided into two sets, and the attributes in these two sets can be varied for different applications. In general, edges with different labels have different non-key attributes. For the business processes involved in this paper, the non-key attribute "*cond*" is assigned to each edge, so $\overline{Eka} = \{cond\}$. This attribute represents the conditions of a control flow in BPMN.

To intuitively express an edge with the temporal attribute *edgeflag* in a TEGG graph, the following lines with different colors are used, which are shown in Fig. 10.

Similar to the representation of a node, a unique identifier is also given for an edge, which is denoted as $e_i$ ($i = 0, 1, 2, \ldots$), to distinguish the different edges. If there is an edge $e_i$, the attribute *att* in the set *Eka* of the edge can be denoted as $e_i.Eka.att$, or $e_i.Eatts.att$, while the notation $e_i.lab$ represents the label.

**Definition 3.5.** *A temporal-attributed graph TAG on a label set L is defined as a quintuple, denoted as $TAG \equiv (N, E, l, p, q)$, where*

- *N* is a set of nodes defined in Definition 3.2, and $N = N_T \cup N_N$ like that in Definition 2.1.
- *E* is a set of edges defined in Definition 3.4.
- *L*, *l*, *p*, and *q* are the same as those defined in Definition 2.1.

**Definition 3.6.** *A temporal-attributed graph with dangling edges $\overline{TAG}$ on a label set L is defined as a septuple, denoted as $\overline{TAG} \equiv (N, \widehat{E}, M, l, \overline{p}, \overline{q}, m)$, where*

- *N*, *L*, and *l* are the same as those defined in Definition 3.5.
- $\widehat{E}$ is a set of edges defined in Definition 3.4, and it can be expressed as $\widehat{E} = E \cup \overline{E}$, which is the same as that defined in Definition 2.2.
- *M*, $\overline{p}$, $\overline{q}$, and m are the same as those defined in Definition 2.2.

**Definition 3.7.** *Two temporal-attributed graphs with dangling edges $\overline{TAG}_1$ and $\overline{TAG}_2$ are isomorphic, denoted as $\overline{TAG}_1 \approx \overline{TAG}_2$, if and only if there exist two bijective mappings $f_N: \overline{TAG}_1.N \leftrightarrow \overline{TAG}_2.N$ and $f_E: \overline{TAG}_1.\widehat{E} \leftrightarrow \overline{TAG}_2.\widehat{E}$, and the following conditions are satisfied:*


**Fig. 10.** Edges with different values of the edgeflag.

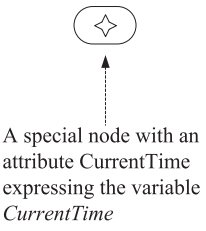



**Fig. 11.** A special node to express the variable *CurrentTime*.

- $\forall\, n\ \forall\, att((n \in \overline{TAG}_1.\ N) \rightarrow (n.\ lab = (f_N(n)).\ lab \wedge ((att \in n.\ Nka) \rightarrow (n.\ Nka.\ att = (f_N(n)).\ Nka.\ att)))$
- $\forall\, e\ \forall\, att((e \in \overline{TAG}_1.\ \widehat{E}) \rightarrow (e.\ lab = (f_E(e)).\ lab) \wedge (\overline{TAG}_2.\ \overline{p}(f_E(e)) = f_N(\overline{TAG}_1.\ \overline{p}(e))) \wedge (\overline{TAG}_2.\ \overline{q}(f_E(e)) = f_N(\overline{TAG}_1.\ \overline{q}(e))) \wedge ((att \in e.\ Eka) \rightarrow (e.\ Eka.\ att = (f_E(e)).\ Eka.\ att)))$

**Definition 3.8.** *A temporal graph TG on a label set L is defined as a six-tuple, denoted as TG ≡ (N, E, CurrentTime, l, p, q), where*

- *N,E,L, l,p*, and *q* are the same as those defined in Definition 3.5.
- CurrentTime is a self-increasing global variable and can be changed by some operations discussed in the following.

To show the *CurrentTime* in a temporal graph, a special node can be used, which is given in Fig. 11. The node is independently placed in a graph and *CurrentTime* is assigned to it as an attribute.

In order to analyze a given graph of a specific field, the graph should be first transformed to a TEGG graph. This transformation can be easily done by mapping nodes and symbols of the graph to one of the TEGG graph, setting attributes for nodes and edges in the TEGG graph, and adding a new *CurrentTime* node. For example, the business process in Fig. 1 can be transformed to a host graph of TEGG as Fig. 12 shown.

In the following, unless specified otherwise, graphs mean temporal graphs or TEGG graphs.

**Definition 3.9.** *A graph TS is the state of a graph TG, if and only if there exist two bijective mappings $f_N$: TS.N↔TG.N and $f_E$: TS.E↔TG.E, and the following conditions are satisfied:*

- $(TS.\ N = TG.\ N) \wedge (TS.\ E = TG.\ E) \wedge (TS.\ L_n = TG.\ L_n) \wedge (TS.\ L_e. = TG.\ L_e)$
- $\forall\, n((n \in TS.\ N) \rightarrow (TS.\ l(n) = TG.\ l(f_N(n))).$
- $\forall\, e((e \in TS.\ E) \rightarrow (TS.\ l(e) = TG.\ l(f_E(e))).$
- $\forall\, e(e \in TS.\ E \rightarrow ((TG.\ p(f_E(e)) = f_N(TS.\ p(e))) \wedge (TG.\ q(f_E(e)) = f_N(TS.\ q(e)))))$

$\exists\, n\ \exists\, e((n \in TS.\ N) \wedge (n.\ Nka.\ statusflag \neq (f_N(n)).\ Nka.\ statusflag) \vee (e \in TS.\ E) \wedge (e.\ Eka.\ edgeflag \neq (f_E(e)).\ Eka.\ edgeflag))$

From the above definition, a graph state depends on the values of two key attributes, they are *statusflag* for nodes and *edgeflag* for edges. Once these values change, which are caused by executing some operations based on *CurrentTime*, the graph state also change. Obviously, the states of a graph *TG* are not unique and can consist of a sequence, denoted as $TG^0$, $TG^1$, $TG^2$, ..., or $TG^i$ ($i$ = 0, 1, 2, ...). In fact, the state of a graph *TG* when *CurrentTime* is equal to a specific value is defined by Definition 3.9. Generally, if a *TG* has a sequence of all states, it can be denoted as *State(TG)*. In *State(TG)*, each state except the first one is a result from its former state with some changes to the key attributes of nodes or edges. A final state of *TG* is a state in which the key attributes' value of end node *n* is equal to a specific value, such as *n. Natts. statusflag = COMPLETED*. In the TEGG, it is the state changes of a given graph due to *CurrentTime* changes that make the simulation and analysis of a system's time sequence available.

In order to check the structural correctness and analyze the temporal semantics of a graph, there are two types of productions in TEGG. One is called E-production, which is similar to productions in the EGG for dealing with the structural aspect. While the other is called T-production, which only deals with the temporal aspect. The following are formal definitions of productions in TEGG.

**Definition 3.10.** *An E-production $p_E$ is the expression $\overline{TAG_L} \Leftrightarrow \overline{TAG_R}$, which consists of a left and right temporal-attributed graphs with dangling edges, denoted as $\overline{TAG_L}$ and $\overline{TAG_R}$ respectively, where*

- $(\overline{TAG_L}.\ M = \overline{TAG_R}.\ M) \wedge (M \subset \{0, 1, 2, ...\})$.

The E-productions defined for business processes are provided in Fig. 13, which are similar to the productions of EGG in terms of structure. However, left and right graphs in each E-production are temporal-attributed graphs with dangling edges, so each node in them also has the attribute *statusflag, starttime, durationtime* and so on.

To facilitate understanding, a brief explanation of one E-production is provided. In the second production, on the right of it is a terminal node labeled *T.x* that represents an activity in a business process, while on the left of it there is a non-terminal node labeled *s*. So, this E-production could perform the replacement of the non-terminal node with the terminal node in L-application or opposite replacement in R-application.

**Definition 3.11.** *A T-production $p_T$ is defined as a quadruple, denoted as ($\overline{TAG_L}$, $\overline{TAG_R}$, Con, Fun), which consists of left graph $\overline{TAG_L}$, right graph $\overline{TAG_R}$, condition Con and function set Fun, where*

- $\overline{TAG_L}$ and $\overline{TAG_R}$ are the same as that defined in Definition 3.10.
- *Con* is a condition which must be satisfied for the use of this T-

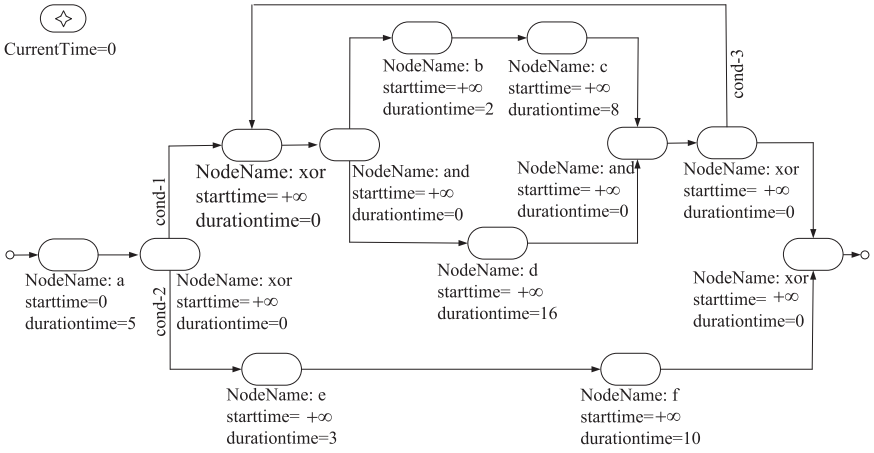



**Fig. 12.** A TEGG host graph corresponding to Fig. 1.

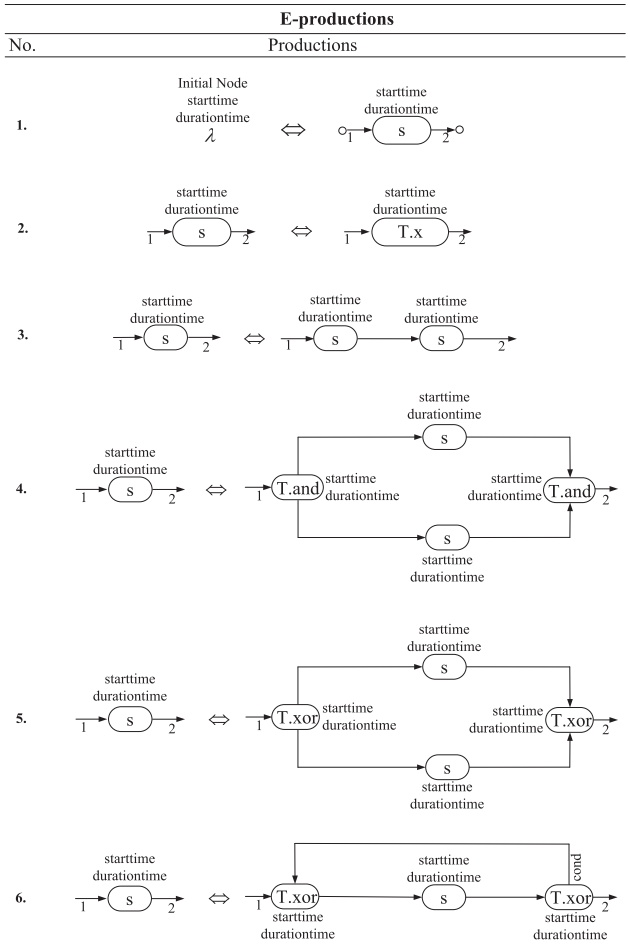



**Fig. 13.** A set of E-productions defined for business processes.

production.

- *Fun* is a group of functions which are executed once this T-production is used.

In the above definition, *Con* is used to determine whether a T-production needs to be used, while *Fun* is used to calculate and modify the values of attributes in a T-production. Generally speaking, the structures of left and right graphs in a T-production are the same, but the attributes of the nodes or edges in these two graphs could be changed by *Fun* functions.

As for the example of business processes, some T-productions are provided in Fig. 14, which consist of not only the left and right graphs, but also *Con* and *Fun*.

Since the left and right graphs in each T-production are temporal-attributed graphs with dangling edges, each node in the T-productions also has the attributes *starttime* and *durationtime*. The values of these attributes could be computed by the corresponding *Fun* in T-productions. It's noted that the return value of the function *prenode(Edge e)* in Fig. 14 is the source node of an edge *e*. At the same time, in Fig. 14 there are some extra notations in T-productions. A notation * is employed to indicate an arbitrary natural number. So, an edge with * means no edge or several edges. In addition, a notation {} containing a set of marks is employed to indicate dangling edges' marks.

To provide a better understanding, some further explanations are provided by taking three T-productions as examples. In the T-production A, its node of the left graph is in working status, while the one of the right graph is in waiting status. *Con* is $(n_1.lab \neq T.xor) \wedge (n_2.lab \neq T.xor) \wedge CurrentTime \geq max\{prenode(e_i).starttime + prenode(e_i).durationtime\} (i = 0, 1, ..., m)$, namely $n_1$ and $n_2$ which represent any atomic activity or a parallel gateway in BPMN, and the value of *Current Time* should be greater than or equal to the maximal sum of the attributes *starttime* and *durationtime* in $prenode(e_i)$ $(i = 0,1,\ldots,m)$. Therefore, this T-production can realize a transformation from a terminal node with waiting status to one with working status, and perform a group of functions that assign the values of the attributes *starttime* and *durationtime* in the right graph's node to the ones in the left graph's node.

As for T-production D, its *Con* is $(n_1.lab = =T.xor) \wedge (n_2.lab = =T.xor) \wedge (e.Eatts.cond = ='True') \wedge CurrentTime \geq n_2.Natts.starttime + n_2.Natts.durationtime$. In this T-production, $n_2$ is in working status, and $n_1$ is in completed status. In addition, $n_1.lab = n_2.lab = T.xor$, $e.Eatts.cond =' True'$, and the value of *CurrentTime* should be greater than or equal to the sum of the attributes *starttime* and *durationtime* in node $n_2$.

In T-productions, there is a special T-production which realizes the increment of a variable *CurrentTime* in a temporal graph, and this incremental quantity is a unit time of an application. T-production G is this special T-production, which is also denoted as $p_{Special-G}$, and its role is to achieve an increment of a variable *CurrentTime* in a temporal graph.

Similar to EGG, the concepts of sub-graph also need to be introduced for TEGG. However, to conveniently deal with temporal semantic, the sub-graph in TEGG includes dangling edges, which is not the same as that of EGG. Its definition is provided as follows.

**Definition 3.12.** *A graph TW is the sub-graph of TG if and only if the following conditions are satisfied:*

- $(TW.CurrenTime = TG.CurrenTime)$.
- $(TW.N \subseteq TG.N) \wedge (TW.L_e \subseteq TG.L_e) \wedge (TW.L_n \subseteq TG.L_n)$.
- $\forall e(e \in TW.E \rightarrow (e \in TG.E \wedge (p(e) \in TW.N \vee q(e) \in TW.N)))$.

For TEGG, a graph *TW* in Fig. 15 is the sub-graph of the graph in Fig. 6. Obviously, *TW* has dangling edges.

**Definition 3.13.** *If a graph TX is the sub-graph of a graph TG, and TQ is a temporal-attributed graph with dangling edges, TX is a redex of TG with respect to TQ, denoted as TX ≡ Redex(TG, TQ), if and only if there exists a bijective mappings $f_N$: TX.N↔TQ.N, and the following conditions are satisfied:*

- $TX \approx TQ$.
- $\forall n((n \in TX.N) \rightarrow (d_{in}(n) = d_{in}(f_N(n))) \wedge (d_{out}(n) = d_{out}(f_N(n))))$.
- if TQ is the left or right graph of a T-production, the condition Con of this T-production should also be satisfied.

In Fig. 16, the redexes with respect to the right side graph of E-production 2 in Fig. 13 are marked with a blue ellipse. As one can see, there are six redexes. Due to *CurrentTime = 0*, conditions of the T-production G, that is $p_{Special-G}$, are satisfied. As a result, the redex with respect to the right side graph of the T-production G is marked by a red ellipse.

**Definition 3.14.** *If a graph TX is the sub-graph of a graph TG, TQ is a temporal-attributed graph with dangling edges, and TX ≡ Redex(TG, TQ), then a function Assign(TX → TQ) is defined as the following assignment operation:*

- $\forall n_j \forall att \exists f_N((n_j \in TQ.N \wedge att \in Natts \wedge (f_N: TQ.N \leftrightarrow TX.N)) \rightarrow (n_j.Natts.att: =f_N(n_j).Natts.att))$.
- $\forall e_j \forall att \exists f_E((e_j \in TQ.\hat{E} \wedge att \in Eatts \wedge (f_E: TQ.\hat{E} \leftrightarrow TX.\hat{E})) \rightarrow (e_j.Eatts.att: =f_E(e_j).Eatts.att))$.

Since the temporal information of a *TG* could be handled by the values of the attributes in each node and edge of *TG*, the function *Assign* $(TX \rightarrow TQ)$ in the above definition is to assign the attributes' values of *TX* to *TQ*. In this way, *TX* and *TQ* have the same temporal information.

In a graph *TG*, if there is a sub-graph *TX* which is the redex of *TG* with respect to *TQ* that is a left or right side graph of a production, we

**T -productions**

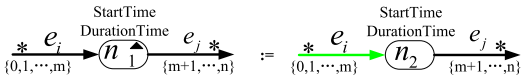
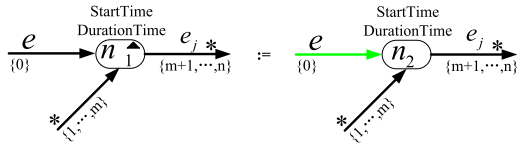
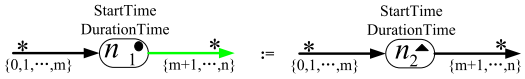
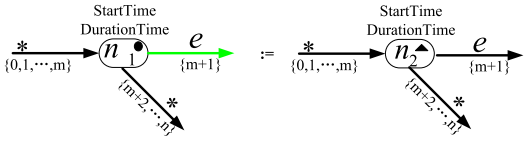
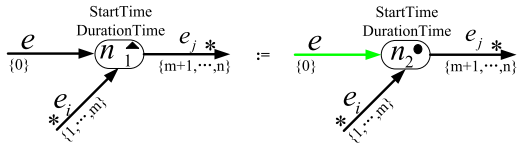
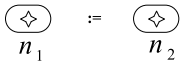
| No. | Productions | Con | Fun |
|---|---|---|---|
| A. |  | $(n_1.lab \neq T.xor) \wedge$ $(n_2.lab \neq T.xor) \wedge$ $CurrentTime \geq max\{$ $prenode(e_i).starttime +$ $prenode(e_i).durationtime\}$ $(i = 0,1,...,m)$ | $n_2.starttime = max\{$ $prenode(e_i).starttime +$ $prenode(e_i).durationtime\}$ $(i = 0,1,...,m)$ $n_1.starttime = n_2.starttime;$ $n_1.durationtime =$ $n_2.durationtime;$ |
| B. |  | $(n_1.lab == T.xor) \wedge$ $(n_2.lab == T.xor) \wedge$ $CurrentTime \geq$ $prenode(e).starttime +$ $prenode(e).durationtime\}$ | $n_2.starttime =$ $prenode(e).starttime +$ $prenode(e).durationtime$ $n_1.starttime = n_2.starttime;$ $n_1.durationtime =$ $n_2.durationtime;$ |
| C. |  | $(n_1.lab \neq T.xor) \wedge$ $(n_2.lab \neq T.xor) \wedge$ $CurrentTime \geq$ $n_2.Natts.starttime +$ $n_2.Natts.durationtime$ | $n_1.starttime =$ $n_2.starttime;$ $n_1.durationtime =$ $n_2.durationtime;$ |
| D. |  | $(n_1.lab == T.xor) \wedge$ $(n_2.lab == T.xor) \wedge$ $(e.Eatts.cond == 'TRUE') \wedge$ $CurrentTime \geq$ $n_2.Natts.starttime +$ $n_2.Natts.durationtime$ | $n_1.starttime =$ $n_2.starttime;$ $n_1.durationtime =$ $n_2.durationtime;$ |
| E. |  | $(prenode(e).lab == T.xor) \wedge$ $(CurrentTime \geq$ $prenode(e).starttime +$ $prenode(e).durationtime\}) \wedge$ $(prenode(e_i).status ==$ $'COMPLETED')(i = 1,2,...,m)$ | $n_2.starttime =$ $prenode(e).starttime +$ $prenode(e).durationtime$ $n_1.starttime = n_2.starttime;$ $n_1.durationtime =$ $n_2.durationtime;$ |
| G. |  | | $n_2.CurrentTime = CurrentTime$ $n_1.CurrentTime = n_2.CurrentTime$ $+unittime$ $CurrentTime = n_1.CurrentTime$ |

**Fig. 14.** T-productions.

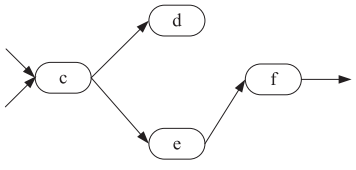



**Fig. 15.** A sub-graph *TW* in the TEGG.

will use the right or left side graph of the production to replace *TX* in *TG*. This process is similar to the one in the EGG, which is called the graph transformation or replacement. Its definition is as follows.

**Definition 3.15.** *A L-application to a graph TG is a transformation that generates a graph TG′ using an E-production L⇔R, denoted as TG′ ≡ Tr(TG, TX$_i$: L, R), where TX$_i$ ∈ Redex(TG, L), and R is used to replace TX$_i$ in TG. The L-application is also denoted as TG →$^P$TG′.*

If a sequence of L-applications for a graph *TG* is the following: $TG \rightarrow^{Pi1} TG_{i1}'$, $TG_{i1}' \rightarrow^{Pi2} TG_{i2}'$, ..., $TG_{i(n-2)}' \rightarrow^{Pi(n-1)} TG_{i(n-1)}'$, $TG_{i(n-1)}' \rightarrow^{Pin} TG_i'$, then $TG \rightarrow {}^*TG_i'$ can be used to simply express this process. For example, Fig. 17 shows a sequence of L-applications. From the initial graph of Fig. 17(a), E-production 1 can be used to derive Fig. 17(b), which in turn is a redex with respect to the left side graph of E-production 4 and can derive Fig. 17(c).

**Definition 3.16.** *A R-application to a graph TG is also a transformation that generates a graph TG″ using an E-production L⇔R, denoted as TG″ ≡ Tr(TG, TX$_i$: R, L), where TX$_i$ ∈ Redex(TG, R), and L is used to*

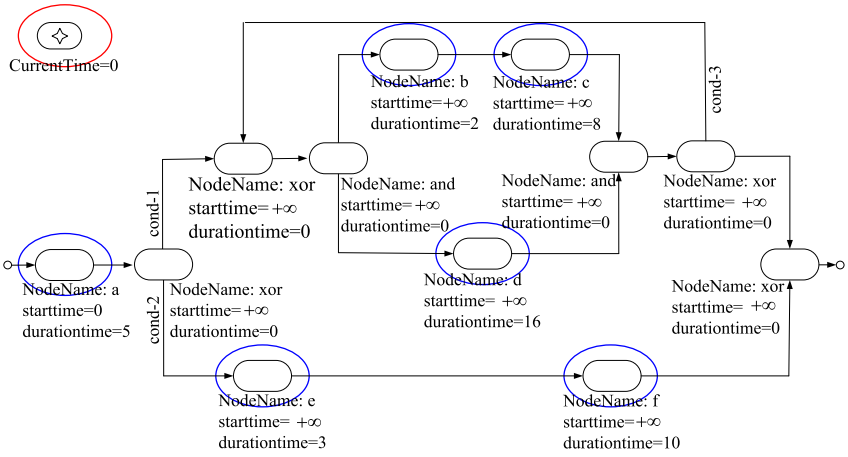



**Fig. 16.** All redexes in Fig. 12.

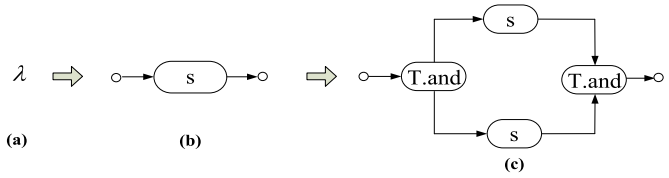



**Fig. 17.** A sequence of L-applications.

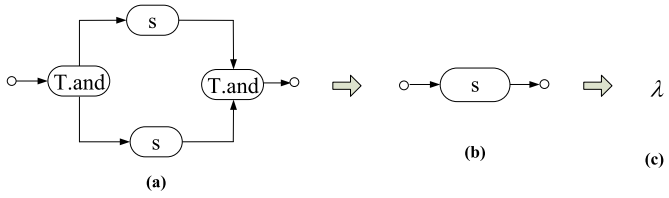



**Fig. 18.** A sequence of R-applications.

*replace $TX_i$ in TG. The R-application is also denoted as $TG \mapsto^{p} TG''$.*

In a similar way as L-applications, a sequence of R-applications, which is $TG \mapsto^{p_{i1}} TG_{i1}''$, $TG_{i1}'' \mapsto^{p_{i2}} TG_{i2}''$, ..., $TG_{i(n-2)}'' \mapsto^{p_{i(n-1)}} TG_{i(n-1)}''$, $TG_{i(n-1)}'' \mapsto^{p_{in}} TG_i''$ can be expressed as $TG \mapsto^{*} TG_i''$. For example, in Fig. 18, there is a sequence of R-applications. For the graph of Fig. 18(a), which is a redex with respect to the right graph of E-production 4 and Fig. 18(b) can be generated using R-application. Similarly, Fig. 18(b) is a redex with respect to E-production 1 and Fig. 18(c) can be generated. Obviously, this R-application is opposite process of the L-application of Fig. 17.

Since there are T-productions that only operate on temporal attributes in TEGG, a T-application needs to be defined to describe a transformation of the host graph using a T-production.

**Definition 3.17.** *A T-application to a graph TG using a T-production (L, R, Con, Fun) is defined as the following process:*

- If $TX_i$ is a redex, namely $TX_i \in Redex(TG, R)$, then $Assign(TX_i \rightarrow R)$ is executed.
- All functions in Fun of the T-production are executed.
- Replace $TX_i$ in TG with L to generate a temporal graph $TG^i$, denoted as $TG^i \equiv Tr(TG, TX_i: R, L)$.

*The above processes are denoted as $TG \Rightarrow^{p} TG^i$.*

It should be noted that we can gain the state of a graph by executing the above T-application on this graph. It is by the continuous execution of T-application that we can change the states of the host graph, which makes the computer simulation of the temporal sequence in the graph available.

**Definition 3.18.** *A TEGG is a quadruple ($\lambda$, $L_n$, $L_e$, $P$), where:*

- $\lambda$ is an initial graph.
- $L_n$ is a label set of nodes, which can be expressed as $L_n = T \cup \overline{T}$, where $T$ represents a label set of terminal nodes and $\overline{T}$ represents a label set of non-terminal nodes.
- $L_e$ is a label set of edges.
- $P$ is a set of productions including E-productions denoted as $P_E$ and T-productions denoted as $P_T$, i.e., $P = P_E \cup P_T$.
- Each production $p \in P$ should satisfy the following constraints, which are called productions' constraints:

1. p is the same as that defined in Definitions 3.10 or 3.11.
2. The right graph of p is nonempty.
3. Let $L_{n_L}$ and $L_{n_R}$ be the label sets of the left and right graphs in p, then $(L_{n_L} \subseteq L_n) \wedge (L_{n_R} \subseteq L_n)$.
4. For any E-production, the following relationship $<$ between its left graph L and right graph R is necessary:

$$L < R \Leftrightarrow (|L.N| < |R.N|) \vee (|L.N| = |R.N| \wedge |L.N_T| < |R.N_T|) \vee (|L.N| = |R.N| \wedge |L.N_T| = |R.N_T| \wedge |L.\widehat{E}| < |R.\widehat{E}|)$$

5. For any T-production, the following relationship $\neq_T$ between its left graph L and right graph R is necessary:

$$L \neq_T R \Leftrightarrow (|L.N_T| = |R.N_T|) \wedge (|L.N_N| = |R.N_N|) \wedge ((\exists n_i \exists f_N (n_i \in L.N \wedge f_N: L.N \leftrightarrow R.N \wedge (n_i.statusflag \neq (f_N(n_i)).statusflag))) \vee (\exists e_i \exists f_E (e_i \in L.\widehat{E} \wedge f_E: L.\widehat{E} \leftrightarrow R.\widehat{E} \wedge (e_i.edgeflag \neq (f_E(e_i)).edgeflag))))$$

In the above definition, productions' constraints ensure the workability and decidability of TEGG, and their fourth constraint means that the size of $R$ must be no less than that of $L$ in an E-production, which makes a host graph parsed in finite steps of R-application [8]. However, in TEGG there is an extra T-application, so its decidability needs to be proven by taking the T-application into consideration, which is provided in Section 4.

**Definition 3.19.** *Let tgg $\equiv$ ($\lambda$, $L_n$, $L_e$, $P$) be a TEGG, its language $\Gamma(tgg)$ is a set of graphs which can be defined by* $\Gamma(tgg) = \{TG | (\lambda \rightarrow^{*} TG) \wedge (TG.N_N = \varnothing)\}$.

A graph grammar is a formal tool for defining a graph language and analyzing its properties. Whether a given graph is correct or not with respect to TEGG, this could be determined by its parsing mechanism.

## 4. Graph parsing

Usually, a graph grammar needs to be equipped with a graph parsing algorithm for automatically checking whether a given graph, called host graph, is correct or valid to the graph language defined by the grammar. The parsing algorithm tries to reduce a host graph to its initial graph according to the grammar's productions. However, in this process, some semantics of the host graph can hardly be checked, such as temporal semantics. In addition, it is very difficult to achieve the purpose of the temporal simulation of a system with temporal requirements. For TEGG, its parsing algorithm alternately executes R-applications and T-applications with corresponding E-productions and T-productions to achieve the goal of checking the structure correctness of a host graph and analyzing the temporal feature of the graph. In order to improve parsing efficiency, the two processes of R-applications and T-applications could be executed separately in the TEGG's parsing algorithm sine there is no need to execute T-applications if the R-applications fail. In the following sub-sections, the parsing algorithm is designed and its decidability is proven.

### 4.1. New parsing algorithm

A new parsing algorithm has two parameters. One is a TEGG host graph *g*, the other is a group productions *P* of TEGG. An overall description of the algorithm is that it checks the structural correctness of *g* using E-productions in *P*. Then, if the structure is correct, the algorithm further analyzes temporal semantics of *g* using T-productions in *P*, otherwise the algorithm returns *NULL*.

The main steps of the parsing algorithm are designed as follows, which tackle both the structural aspect and the temporal aspect. The pseudo codes of the parsing algorithm are provided in Fig. 19.

Step 1. *Initializing*, to initialize a given graph of TEGG. Firstly, attributes of all nodes and edges in the given graph are assigned to the specific values. For example, the *statusflag* attribute of all nodes is assigned to the value *WAITING*, while the *edgeflag* attribute of all edges is assigned to the value *STOPPING*. Secondly, the global variable *Current Time* in the given graph is set to be 0. This step is realized by function *Initialize()*.
Step 2. *E-checking*, to check structure correctness of the given graph by means of R-applications with E-productions. If the graph is correct, then go to the next step, otherwise skip the next step. This step is realized by the function *GraphStructureParsing()*.
Step 3. *T-checking*, after setting the *statusflag* attribute of all start nodes to the value *WORKING*, to perform temporal analysis of the given graph by means of T-applications with T-productions, which is realized by the function *GraphTemporalParsing()*.

For function *GraphStructureParsing()*, its pseudo codes are shown in

```
Graph Parsing (Graph g, Productions P)
{
        G←Initialize( g );
        if(G!= NULL && P!= NULL)
            {
                if (GraphStructureParsing( G , E-productions  )!=false)
                    {
                        SettingStatusFlag( start nodes );
                        result = GraphTemopralParsing( G, T-productions );
                        return result;
                    }
                else
                    return NULL;
            }
}
```

**Fig. 19.** A new parsing algorithm.

```
Boolean GraphStructureParsing(Graph g, Productions P)
{
<Loop-1>    while (   g != λ   )
                {
                    Push( MARK, ERedexStack );
<Loop-2>            for all p ∈ P
                      {
                        EredexSet = FindRedex ( g, p.R );
<Loop-3>                for all eredex ∈ EredexSet
                            Push( eredex, ERedexStack );
</Loop-2>             }
                    eredex ← Pop( ERedexStack );
<Loop-4>            while ( eredex == MARK )
                      {
                        g ← Pop( GraphStack );
                        eredex ← Pop( ERedexStack );
                        if ( eredex==NULL )
                            return false;
</Loop-4>             }
                    Push ( g, GraphStack );
                    g = RApplication ( g, p, eredex );
</Loop-1>       }
                return true;
}
```

**Fig. 20.** The function *GraphStructureParsing()*.

Fig. 20. In order to better understand this function, some detailed explanations are given with introduction of two stacks GraphStack and ERedexStack. They are used to store graphs and corresponding redexes of the graphs respectively. The pseudo-codes push the flag “MARK” into the ERedexStack, which is used to separate redexes of different intermediate graphs. For specific graph *g* in GraphStack, pseudo-codes between 〈Loop-2〉 and 〈/Loop-2〉 search all redexes with all E-productions of TEGG, and push them into the ERedexStack for backtracing. Function *FindRedex()* is similar to the function *FindRedexForRight()* in [8] that searches for all possible redexes. From these codes, the function *Rapplication()* which executes R-application with respect to the redex poped out from the ERedexStack using the corresponding production is in the Loop-1. That means R-applications are repeatedly used. Section 5 will further discuss show the function *GraphStructureParsing()* varifies graphs of TEGG in a case study.

For function *GraphTemporalParsing()*, main steps are designed as follows, which makes temporal analysis of a given graph. The pseudo codes of the parsing algorithm are provided in Fig. 21.

Step 1. T-productions except for the special T-production G, as shown in Fig. 14, can be used for searching redexes in the given graph. If redexes can be found, they are pushed into a stack called *TRedexStack*, which is done by the Loop-C in Fig. 21, then go to Step 2. If there is no redex, go to the step 3.
Step 2. T-applications are performed for the redexes in *TRedexStack* until there is no redex. This step work is done by the Loop-B in Fig. 21.
Step 3. The special T-production $p_{Special-G}$ is used to find a redex, and T-application is executed with this production, which increases the *Current Time* smallest time unit of the business process.
Step 4. Steps 2–4 are repeatedly executed until the temporal graph can be transformed to the final state.

Section 5 will further discuss how the function *GraphTemporalParsing()* analyzes graphs of TEGG in a case study.

### 4.2. Decidability of the parsing algorithm

**Definitions 4.1.** *Supposing a node or an edge in a graph TG has an attribute att, the set composed of the att's values is denoted as V. Since this set is related to the attribute att, so it can also be denoted as* $V_{att}$.

```
Graph GraphTemporalParsing(Graph g, Productions P)
{
            tredex← MARK;
<Loop-A>    while ( g.EndNode.Natts.statusflag !=COMPLETED
                  || tredex!=NULL)
              {
<Loop-B>        while ( tredex!=NULL )
                  {
                    Push( MARK, TRedexStack );
<Loop-C>            for all p ∈ {p | p ∈ P ∧ p ≠ P_Special−G}
                      {
                        TredexSet = FindRedex ( g, p.R );
<Loop-D>                for all tredex ∈ TredexSet
</Loop-D>                 Push( tredex, TRedexStack );
</Loop-C>               }
                    tredex ← Pop( TRedexStack );
<Loop-E>            while ( tredex!=NULL)
                      {
                        if(tredex!=MARK)
                          g = TApplication ( g, p, tredex );
                        tredex ← Pop( TRedexStack );
</Loop-E>               }
</Loop-B>           }
                  SpeTredex=FindRedex( g, P_Special−G )
                  g = TApplication ( g, P_Special−G , SpeTredex);
</Loop-A>     }
            return g;
}
```

**Fig. 21.** The function *GraphTemporalParsing ()*.

**Theorem 4.1.** *If TG is a graph, then State(TG) is a finite set.*

*Proof:*

Suppose $TG$. $N = \{n_1, n_2, ...n_u\}$ and $TG$. $E = \{e_1, e_2, ...e_v\}$ (u and v are constants), and that is TG has finite nodes and edges. For any node, say node n, it has a key attributes' set $C_i^j$, while the key attribute *statusflag* has finite values according to Section 3. So, the set $V_{statusflag}$ is finite. For any edge, say edge e, it has a key attributes' set $Eka = \{edgeflag\}$, while the key attribute *edgeflag* has finite values according to Section 3. So,

$$|State(TG)| = (|V_{edgeflag}|)^v \cdot (|V_{statusflag}|)^u$$

Then, it is proven that *State*(*TG*) is finite.

**Lemma 4.1.** *For an arbitrary nonempty graph TG ($TG.N \neq \emptyset$) and a given tgg of the TEGG graph grammar, whether or not TG is in Γ(tgg) is decidable within finite steps.*

*Proof:*

Suppose there is an E-production $p_i$ ($p_i \in P$) of the *tgg*, its right graph is $R_i$, and $TX_i \in Redex(TG, R_i)$, then $TG \mapsto^{p_i} TG'$. From productions' constraints 4 in Section 3, it has $|TG| > |TG'|$. That means each R-application will result in decreasing the size of *TG*. Therefore, $TG \in \Gamma(tgg)$ can be checked within limited steps. On the contrary, if there is no such an E-production $p_i$ to perform R-application $TG \mapsto^{p_i} TG'$, then $TG \notin \Gamma(tgg)$ can be concluded.

**Lemma 4.2.** *If there is an arbitrary nonempty graph TG ($TG.N \neq \emptyset$) and a given tgg of the TEGG graph grammar, and TG is in Γ(tgg), whether or not TG can be transformed to a final state is decidable within finite steps.*

*Proof:*

For *TG* after setting the *statusflag* attribute of all start nodes to the value *WORKING* and the *edgeflag* attribute of all edges to the value *STOPPING*, if it can continuously find a redex of a T-production $p_j$ to perform T-application $TG| \Rightarrow^{p_j} TG^j$, then, according to productions' constraints 5 and Theorem 4.1, the state of *TG* can be changed to final state within limited steps. On the contrary, if there is no such a T-production $p_j$ to perform T-application $TG| \Rightarrow^{p_j} TG^j$, then *TG* can not be transformed to a final state.

**Theorem 4.2.** *For an arbitrary nonempty graph TG ($TG.N \neq \emptyset$) and a given tgg of the TEGG graph grammar, the parsing algorithm of TEGG can stop.*

*Proof:*

First, the parsing algorithm checks structure correctness of the given graph by means of R-applications with E-productions. According to Lemma 4.1, whether or not TG is in Γ(*tgg*) is decidable within finite steps. So, the function *GraphStructureParsing()* can stop within finite steps.

Second, if the structure of the given graph is correct, then the parsing algorithm performs temporal analysis of the given graph by means of T-applications with T-productions. According to Lemma 4.2, whether or not *TG* can be transformed to a final state is decidable within finite steps. So, the function *GraphTemporalParsing()* can stop within finite steps.

Then, the above conclusion is proven.

### 4.3. Parsing complexity

Besides the decidability of the parsing algorithm, the complexity of the algorithm is also in need of considering its computational efficiency. In this sub-section, the complexity of the parsing algorithm is analyzed.

**Theorem 4.3.** *For a given TEGG, the time complexity of its parsing algorithm is* $O\left(\left(\frac{ps}{m!}\right)^h \cdot (h^h h!)^m\right)$, *where h is the number of nodes in a host graph, ps is the number of productions in the TEGG, and m is the maximum number of nodes among all productions' right graphs.*

*Proof:*

Supposing $\sigma_1$, $\sigma_2$, and $\sigma_3$ are time complexity of the function *Initialize (), GraphStructureParsing(),* and *GraphTemporalParsing()* respectively. According to the parsing algorithm, its time complexity $\sigma$ is determined by the following equation:

$$\sigma = \sigma_1 + \sigma_2 + \sigma_3$$

It is obvious that the complexity of function *Initialize()* is *O(h)*. For function *GraphStructureParsing()*, supposing $k_1$ is the maximal number of iterations in the Loop-1, $k_2$ is that in the Loop-2, $k_3$ is that in the Loop-3, and $k_4$ is that in the Loop-4, and $t_1$, $t_2$, and $t_3$ are the time complexities of the function *FindRedex(), RApplication()*, and *TApplication()* respectively, then, similar to the complexity analysis in literature [8], the complexity of function *GraphStructureParsing()* is:

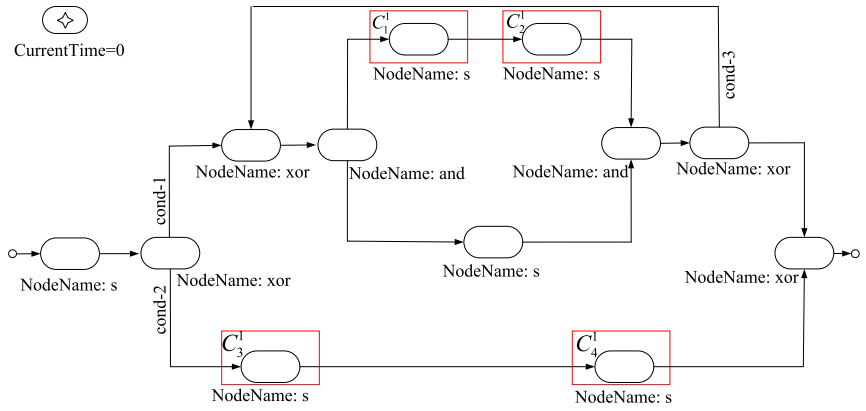
$$\sigma_2 = O(k_1 \cdot (k_2 \cdot (t_1 + k_3) + k_4 + t_2))$$
$$= O\left(\left(\frac{ps}{m!}\right)^h \cdot (h^h h!)^m\right)$$

As for the function *GraphTemporalParsing()*, supposing $k_A$ is the maximal number of iterations in the Loop-A, $k_B$ is that in the Loop-B, $k_C$ is that in the Loop-C,$k_D$ is that in the Loop-D, and $k_E$ is that in the Loop-E, then the complexities of the five loops could be further separately discussed.

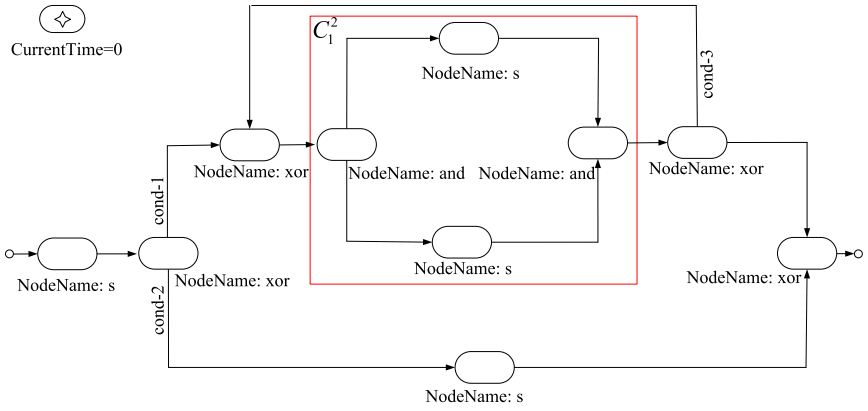
The worst case of $k_A$ is *h*, i.e. $k_A \leq h$, because a given graph with *h* nodes can be reduced to the final state using T-productions in which there is only one node that key attributes are updated in the left graph. $k_C$ can be at most to the number of T-productions, namely $k_C < ps$. $k_D$ is the number of redexes of a given production. Since the maximal number of redexes is the all possible node combinations $C_h^m$, thus $k_D \leq C_h^m = O(h^m)$. $k_B$ and $k_E$ are respectively the partial number of redexes poped from stack *TRedexStack*. Since $k_C \cdot k_D$ is the total number of graphs in the *TRedexStack*, $k_B$, and $k_E$ should be no more than $k_C \cdot k_D$ respectively. Based on the discussion, the complexity of the function *GraphTemporalParsing()* is:

$$\sigma_3 = O(k_A \cdot (k_B \cdot (k_C \cdot (t_1 + k_D) + k_E \cdot t_3) + t_1 + t_3))$$
$$\leq O(h \cdot (ps \cdot h^m \cdot (2 \cdot ps \cdot h^m)))$$
$$= O(ps^2 \cdot h^{2m+1})$$

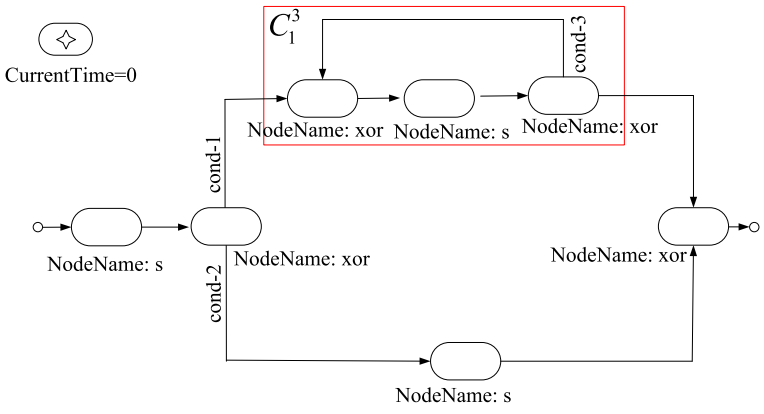
In a summary, the time complexity of the parsing algorithm is the following:




**(a). An intermediate graph after using E-production 2**




**(b). An intermediate graph after using E-production 3**




**(c). An intermediate graph after using E-production 4**

**Fig. 22.** (a). An intermediate graph after using E-production 2, (b). An intermediate graph after using E-production 3, (c). An intermediate graph after using E-production 4.

$$\sigma = \sigma_1 + \sigma_2 + \sigma_3$$
$$\leq O\left(h + \left(\frac{ps}{m!}\right)^h \cdot (h^h h!)^m + ps^2 \cdot h^{2m+1}\right)$$
$$= O\left(\left(\frac{ps}{m!}\right)^h \cdot (h^h h!)^m\right)$$

Of course, if the set of E-productions in TEGG is *selection-free*[7], the time complexity of this parsing algorithm will be greatly reduced.

## 5. Case study

In Section 1, a business process is provided as an example for highlighting temporal requirements. In order to tackle the requirements, temporal attributes for each node and each edge, a global variable *CurrentTime*, some T-productions, a new T-application on the graph and so on are introduced in Section 3. Then, how to analyze this business process by the TEGG? This section illustrates the execution processes of the parsing algorithm in Fig. 19.

The first process is initialization, in which the host graph of TEGG as Fig. 12 shown can be obtained by executing the first steps of the parsing algorithm.

The second process is grammatical analysis, in which the host graph's structure can be analyzed and checked by executing the function *GraphStructureParsing()* based on E-productions. More detailed analysis

processes are given below. In the following figures, temporal attributes of each node are not listed for representation simplicity without losing generality, and each redex is identified as $C_i^j$, where j presents the number of an intermediate graph including the redex and i specifies the redex's order.

(1) The function first tries to search for all redexes with the E-productions given in Fig. 13 to yield Fig. 16. Obviously, E-production 2 is satisfied and R-applications are then executed to obtain Fig. 22(a), in which the labels of the above redexes have been changed to "s".
(2) The function continues to search for redexes based on Fig. 22(a). As a result, the redexes $C_1^1$, and $C_2^1$ are selected and replaced by R-application with E-production 3. The similar processes are also applicable to other redexes: $C_3^1$, and $C_4^1$. So, Fig. 22(b) can be obtained.
(3) The function again searches for redexes. As a result, the redex $C_1^2$ is selected and replaced by R-application with E-production 4. Then, Fig. 22(c) can be obtained.

With iteration of the above three processes, if the host graph in Fig. 12 can finally be transformed step by step into the initial graph, then one can concluded that the host graph is structural correct, otherwise it is not.

The third process is temporal analysis, in which the host graph's temporal requirement can be analyzed and checked by executing the function *GraphTemporalParsing()* based on T-productions given in Fig. 14. Before the execution of the function *GraphTemporalParsing()*, some temporal initializations need to be done as Fig. 23(a) shown, in which "*CurrentTime*" represents a relative current time, *statusflag* attribute of a start node is assigned *WORKING, starttime* attribute in node *a* is assigned to 0. More detailed analysis processes of the function *GraphTemporalParsing()* are given below.

(1) The function first tries to search for all redexes with the T-productions given in Fig. 14. In Fig. 23(a) since *CurrentTime = 0*, conditions of these T-productions are not satisfied, and conditions of the special T-production G are satisfied, the redex $C_1^1$ with *CurrentTime = 0* can be placed by $C_1^2$ with *CurrentTime = 1* to yield Fig. 23(b).
(2) The above process is repeatedly executed until *CurrentTime = 5*, which yields Fig. 23(c).
(3) The function continues to search for redexes based on Fig. 23(c). Since *CurrentTime = 5* and conditions of the T-production C are satisfied, the redex $C_1^3$ is selected and replaced by T-application with the T-production C. So, Fig. 23(d) can be obtained.
(4) The function again searches for redexes. Since *CurrentTime = 5* and conditions of the T-production B are satisfied, the redex $C_1^4$ is selected and replaced by T-application with the T-production B. Then, its Fun is activated and executed, which assigns 5 to the attribute *starttime* of $C_1^4$. So, Fig. 23(e) can be obtained.
(5) The function again searches for redexes. Since *CurrentTime = 5 and* conditions of the T-production D are satisfied, redex $C_1^5$can be selected and replaced by T-application with the T-production D, which yields Fig. 23(f).

With iteration of the above five processes, if the state of the initial graph in Fig. 23(a) can finally be transformed step by step into the final state, then one can analyze these states and make the simulation of time sequence. Therefore, the requirements in Section 1 can be tackled with TEGG.

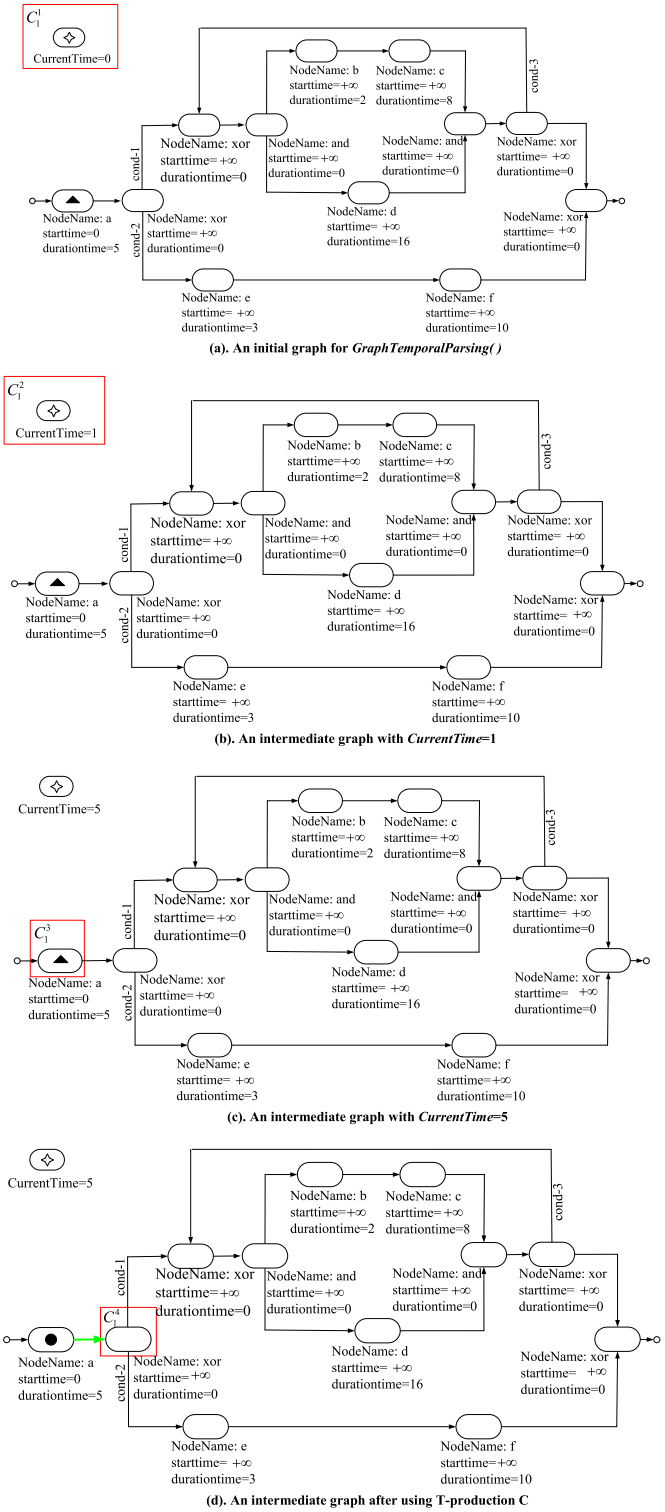
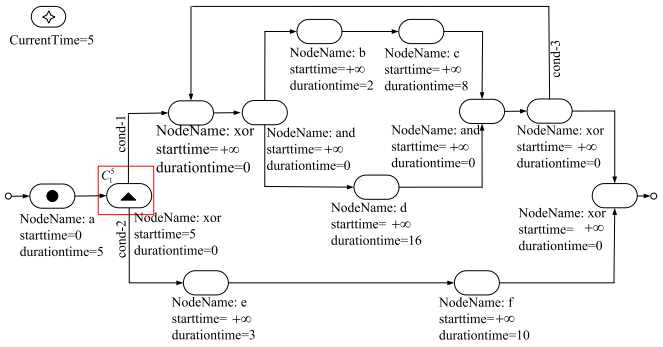
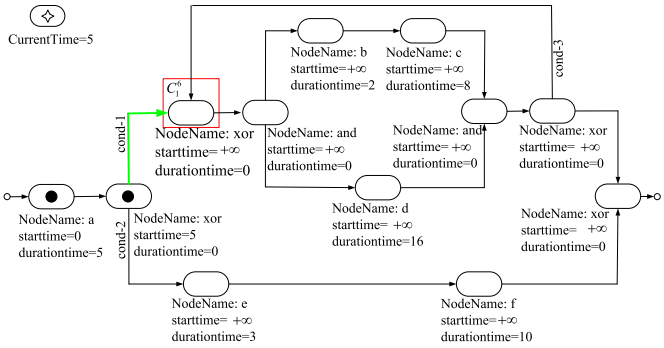



**Fig. 23.** (a). An initial graph for *GraphTemporalParsing()*, (b). An intermediate graph with *CurrentTime* = 1, (c). An intermediate graph with *CurrentTime* = 5, (d). An intermediate graph after using T-production C, (e). An intermediate graph after using T-production B, (f). An intermediate graph after using T-production D.




(e). An intermediate graph after using T-production B




(f). An intermediate graph after using T-production D

**Fig. 23.** (*continued*)

## 6. Conclusion

This paper presents a new graph grammar formalism TEGG by introducing a temporal mechanism into the existing graph grammar EGG. The temporal mechanism includes the temporal attributes of each node and each edge in graphs, a global variable *CurrentTime*, T-productions, T-applications and a new parsing algorithm, etc. They make the computer simulation of the temporal sequence possible. Based on a set of properly designed productions, such as E-productions and T-productions, the parsing algorithm is able to check not only the structural correctness but also the temporal satisfaction of host graphs.

In the paper, besides the formal definition of the TEGG and the presentation of the parsing algorithm, several properties of the TEGG are pointed out and proven, the decidability and complexity of the parsing algorithm is analyzed, and a case study is provided to illustrate how the TEGG works in applications. It should be noted that the temporal mechanisms of the TEGG is a basis for graph grammars to solve time-related problems and can easily be applied to other graph grammars, such as LGG and RGG.

In addition to the application of TEGG to BPMN models described in the above, potential applications of TEGG are worthy of investigation. Actually, TEGG as a formal tool is suitable for generating and analyzing graphs in connection with time issues. For example, TEGG may be applicable to UML (Unified Modeling Language) diagrams in which time state and time sequence are important components in need of analyzing and manipulating. TEGG may also be applicable to program flow charts for efficiently scheduling parallel executions, and to BPEL (Business Process Execution Language) descriptions for correctly specifying time attribute and execution sequence, etc.

Our future research includes the above mentioned applications of TEGG and the development of a more effective parsing algorithm.

## Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant 61170089 and Natural Science Foundation of Jiangsu Province under grant 2012060.

## References

[1] M. Zur Muehlen, D.T Ho, Service process innovation: a case study of BPMN in practice, Proceedings of the 41st Annual Hawaii International Conference on System Science, 2008, p. 372.
[2] G. Mpardis, T. Kotsilieris, Bank loan processes modelling using BPMN, Proceedings of the Developments in E-systems Engineering (DESE), 2010, pp. 239–242.
[3] M. Decker, H. Che, A. Oberweis, P. Sturzel, M. Vogel, Modeling mobile workflows with BPMN, Proceedings of the 2010 Ninth International Conference on Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010, pp. 272–279.
[4] G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation vol. 1, World Scientific Publishing Co. Pte. Ltd., Singapore, 1997.
[5] H. Fahmy, D. Blostein, A survey of graph grammars: theory and applications, Proceedings of the 11th Int'1 Conf. on Pattern Recognition, Vol. 2: Conf. B: Pattern Recognition Methodology and Systems, 1992, pp. 294–299.
[6] C. Ermel, M. Rudolf, G. Taentzer, The AGG approach: language and environment, Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools vol. 2, World Scientific Publishers, 1999, pp. 551–603.
[7] D.Q. Zhang, K. Zhang, J.N. Cao, A context-sensitive graph grammar formalism for the specification of visual languages, Comput. J. 44 (3) (2001) 187–200.
[8] X.Q. Zeng, K. Zhang, J. Kong, G.L. Song, RGG+: An enhancement to the reserved graph grammar formalism, in: M. Erwig, A. Schurr (Eds.), Proceedings of the 2005 IEEE Symp. on Visual Languages and Human-Centric Computing, IEEE Computer Society Press, Los Alamitos, 2005, pp. 272–274.
[9] K. Wittenburg, L. Weitzman, Relational grammars: theory and practice in a visual language interface for process modeling, in: K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, Berlin, 1998, pp. 193–217.
[10] D. Janssens, G. Rozenberg, Graph grammars with neighbourhood-controlled embedding, Theor. Comput. Sci. 21 (1982) 55–74.
[11] G. Rozenberg, E. Welzl, Boundary NLC graph grammars–basic definitions, normal forms, and complexity, Inf. Control 69 (1986) 136–167.
[12] F. Ferrucci, G. Tortora, M. Tucci, G. Vitiello, A predictive parser for visual languages specified by relation grammars, Proceedings IEEE Symposium on Visual Languages, October, 1994, pp. 245–252.
[13] K. Wittenburg, Earley-style parsing for relational grammars, Proceedings of the IEEE Workshop on Visual Languages, 1992, pp. 192–199.
[14] L. Weitzman, K. Wittenburg, Relational grammars for interactive design, Proceedings of the IEEE Symposium on Visual Languages, 1993, pp. 4–11.
[15] K. Marriott, Constraint multiset grammars, Proceedings of the 1994 IEEE Symposium on Visual Languages, 1994, pp. 118–125.
[16] S.S. Chok, K. Marriott, Automatic construction of user interfaces from constraint multiset grammars, Proceedings of the 11th IEEE international symposium on Visual Languages, 1995, pp. 242–249.
[17] B. Shizuki, H. Yamada, K. Iizuka, J. Tanaka, A unified approach for interpreting handwritten strokes using constraint multiset grammars, Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environment, 2003, pp. 180–182.
[18] Y. Adachi, K. Tsuchida, K. Anzai, T. Yaku, Hierarchical program diagram editor based on attribute graph grammar, Proceedings of the 20th International Conference on Computer Software and Applications, 1996, pp. 205–213.
[19] Y. Adachi, S. Kobayashi, K. Tsuchida, T. Yaku, An attribute graph grammar for signal flow graphs, Proceedings of the 1999 IEEE International Conference on Control Applications, 1999, pp. 1549–1554.
[20] F. Han, S.-C. Zhu, Bottom-up/top-down image parsing by attribute graph grammar, Proceedings of the 10th IEEE International Conference on Computer Vision, 2005, pp. 1778–1785.
[21] E. Golin, A Method for the Specification and Parsing of Visual Languages, [Ph.D. Thesis] Department of Computer Science, Brown University, 1991.
[22] Y. Adachi, Visual language design system based on context-sensitive NCE graph grammar, Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics, 2004, pp. 502–507.
[23] Y. Adachi, S. Kobayashi, K. Tsuchida, T. Yaku, An NCE context-sensitive graph grammar for visual design languages, Proceedings of the 1999 IEEE Symposium on Visual Languages, 1999, pp. 228–235.
[24] Y. Adachi, Y. Nakajima, A context-sensitive NCE graph grammar and its parsability, Proceedings of the 2000 IEEE International Symposium on Visual Languages, 2000, pp. 111–118.
[25] J. Rekers, A. Schürr, Defining and parsing visual languages with layered graph grammars, J. Visual Lang. Comput. 8 (1) (1997) 27–55.
[26] J. Kong, K. Zhang, X. Zeng, Spatial graph grammars for graphical user interfaces, ACM Trans. Comput.-Hum. Interact. 13 (2) (2006) 268–307.
[27] J. Kong, K. Zhang, On a spatial graph grammars formalism, IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC), 2004.
[28] C. Zhao, J. Kong, J. Dong, K. Zhang, Pattern-based design evolution using graph transformation, J. Visual Lang. Comput. 18 (2007) 378–398.
[29] Z. Xiao-Qin, H. Xiu-Qing, Z. Yang, An edge-based context-sensitive graph grammar formalism, J. Software 19 (8) (2008) 1893–1901.
[30] A. Roudaki, J. Kong, K. Zhang, Specification and discovery of web patterns: a graph grammar approach, Inf. Sci. 328 (2016) 528–545.
[31] C.Y. Zhao, J. Kong, K. Zhang, Program behavior discovery and verification: a graph grammar approach, IEEE Trans. Softw. Eng. 36 (3) (2010) 431–448.

[32] G. Song, K. Zhang, Visual XML schemas based on reserved graph grammars, International Conference on Information Technology: Coding and Computing (ITCC), 2004, pp. 687–691.
[33] K. Zhang, D-Q. Zhang, Y. Deng, A Visual Approach to XML Document Design and Transformation, Proceedings of the 2001 IEEE Symposium on Human-Centric Computing Languages and Environments (HCC'01), Stresa, Italy, 5-7, September, IEEE CS Press, 2001, pp. 312–319.
[34] C.Y. Zhao, J. Kong, K. Zhang, Design Pattern Evolution and Verification Using Graph Transformation, Proceedings of the 40th Hawaii International Conference on System Sciences (HICSS'07), Big Island, Hawaii, IEEE CS Press, 2007, pp. 290–296 3-6 January.
[35] K. Zhang, D. Zhang, J. Cao, Design, construction, and application of a generic visual language generation environment, IEEE Trans. Software Eng. 27 (2001) 289–307.
[36] Z. Shi, X. Zeng, T. Zhang, et al., Bidirectional transformation between BPMN and BPEL with graph grammar, Comput. Electr. Eng. 51 (2016) 304–319.
[37] S. Gyapay, R. Heckel, D. Varró, Graph transformation with time: causality and logical clocks, Graph Transformation Lecture Notes Comput. Sci. 2505 (2002) 120–134.
[38] R. Alur, D. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (2) (1994) 183–235.
[39] S. Mazanek, M. Hanus, Constructing a bidirectional transformation between BPMN and BPEL with a functional logic programming language, J. Visual Lang. Comput. 22 (2011) 66–89.