DOI 10.2298/CSIS101120042M

Avoiding Unstructured Workflows in Prerequisites Modeling

Boris Milašinović and Krešimir Fertalj

Faculty of Electrical Engineering and Computing, Unska 3,
10000 Zagreb, Croatia
{boris.milasinovic, kresimir.fertalj}@fer.hr

Abstract. Integrating prerequisite relationships, partially defined as graph components, produces a directed graph that corresponds to a well-defined and well-behaved workflow consisting only of and-splits and and-joins. Such a workflow often cannot be transformed to a structured workflow. This paper presents an approach to producing a corresponding structured workflow that will, with some adjustments in the runtime, correspond to the original unstructured workflow. The workaround is based on element cloning and on a workflow wrapper handling clones in order to avoid multiple element instances. An algorithm for finding clones and an algorithm for reducing the number of clones are proposed. Correctness of the algorithms is analyzed and some drawbacks and possible improvements are examined.

Keywords: Workflow management, structured workflows, unstructured workflows, modeling prerequisite relationships, and-splits

1. Introduction

In order to follow the key guidelines for designing a business layer of an application [17] one of the tasks is to extract a workflow component that defines and coordinates multistep business processes. Although traditionally related to enterprise systems, workflow management has other applications. Applications of prerequisite relations between workflow elements can be found in merging dependencies between UML components [8], in modeling course prerequisites in a learning management systems [4][14][22], in modeling relationships between workflow sub-components of a learning management system [27] or in modeling workflow-based data for e-learning [25]. Development of proprietary workflow management software can enhance flexibility and help in avoiding limitations of existing systems in dealing with synchronization points in unstructured workflows [18], but it requires a lot of work and raises compatibility and interoperability issues. On the other hand the use of existing workflow management software induces problems during the design of the business process model, because the workflow management systems impose different syntactical restrictions on models. One of the restrictions is that the workflow model should (and often must) be structured. Perspectives regarding this restriction vary. In [9] authors propose that workflow models should be structured in order to avoid the so called “spaghetti business process modeling”, but there might be good reasons to use unstructured business processes [5].

In this paper an approach to avoiding unstructured workflows in prerequisite modeling using directed graphs is described. The idea was introduced in [19], but here we elaborate on how to produce a structured workflow that will, with some additional adjustments in the runtime environment, correspond to the original unstructured workflow. An overview of a related work is given in the second section. The third section describes the process of merging partially defined prerequisite relationships into a directed graph that corresponds to a workflow with only and-split and and-joins and further expands on the idea of a modification based on the graph’s vertex cloning. Finding vertices that have to be cloned is done using an original vertex labeling algorithm formally described in the fourth section. The fifth section deals with reducing the number of clones prior to transforming an integrated graph into a concrete workflow model. For the sake of presentation simplicity we use the Windows Workflow Foundation model [23] (in further text WF-model). The algorithm for reducing the clones is described in the sixth section. The seventh section proves the correctness of the proposed algorithms and the eighth section discusses their complexity. The ninth section deals with possible improvements of the proposed algorithms.

2. Related Research

In [11] authors define a well-behaved workflow as one that can never lead to a deadlock or result in multiple active instances of the same activity. Another body of research [7] defines well-formed workflows, relates them to well-behaved workflows and defines prerequisites that a well-formed workflow must have in order to be structured.

Tools for transforming a process model into a corresponding structured model [21] and for translating an unstructured workflow into a structured BPEL model [13], [2] automate the transformation, but not all models can have their structural pair. In [11] and [12] it is shown that there are well-behaved workflow models that cannot be modeled as structured workflow models. Moreover, [15] shows that a workflow containing only and-splits [24] is always well-behaved but does not have a structured mapping if it does not meet certain requirements. Intuitively expressing those requirements, in order to have a structured mapping, such workflow must have and-splits paired with corresponding and-joins in a parallel routing form [1].

An approach for prerequisite modeling in which dependencies are modeled in a tree form, which has the target element as the root node of the (sub)tree and its prerequisites as its children [8], enables direct transformation to the structured workflow, but can produce large trees with same elements in different subtrees. For such elements the same dependency subtree has to be added several times and the tree can grow rapidly. As there might be many elements having a common prerequisite, which is quite common in course modeling, it is better to use directed graphs. The more common prerequisites exist, the more obvious the benefit of this approach is. The drawback of this idea is that such graphs usually produce unstructured workflows.

The idea how to produce a structured workflow that will, with some additional adjustments in the runtime environment, correspond to the original unstructured workflow introduced in [19] is based on workflow elements cloning (duplication), which in normal circumstances leads to multiple instances of an element but one can resolve these in the runtime. As clones will formally be different elements, the formal verification techniques will treat such a workflow as a structured workflow. Although this approach also introduces duplicates, their number should be significantly less than using trees due to common split and merge points.

3. Partially Defined Prerequisite Relationships

Each prerequisite relationship can be graphically presented as a directed arc between two vertices. Direction of the arc defines the dependency of a target vertex upon a source vertex. The vertices and arcs form a single (not necessary connected) integrated directed graph in which each vertex can have one or more incoming and outgoing arcs. Each component of a graph with one source and one sink vertex can be paired with a well-formed workflow model containing only and-splits and and-joins. **Fig. 1** shows this straightforward process of transformation from a directed graph into a well-formed workflow model. Each vertex having two or more outgoing arcs is presented with one workflow element connected to an and-split with outgoing arcs moved from the vertex to an and-split. Each vertex having two or more incoming arc is presented with one and-join and one workflow element, where incoming arcs are moved to the and-join.

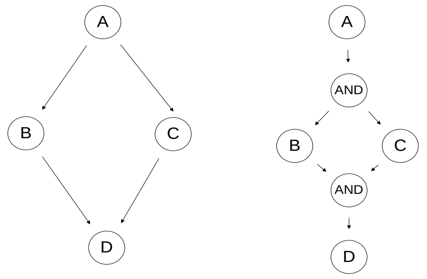

Fig. 1. From a directed graph to a well-formed workflow model

With some modifications described later in this section, it is guaranteed that the integrated graph will be connected and will have one source and one sink vertex and therefore can be represented with one well-formed workflow. In the remainder of the paper, workflow models will be depicted as directed graphs without explicitly shown and-split and and-join elements in order to simplify the figures.

The duplication of vertices [11] in the process of transformation of simple workflows without parallelism into structured workflows cannot be directly applied because it would lead to multiple instances of certain elements. If it can be guaranteed that multiple instances are handled in the way that only one of them is effectively executed during the runtime, then the corresponding structured workflow can be created. We propose cloning common vertices (workflow elements) and putting them into parallel branches. Concrete implementation of the workflow model must ensure that the corresponding clones behave as a single element and that no data duplication occurs in the runtime. This can be ensured by building a workflow wrapper that would expose only unique instances of elements to other layers of the application.

In order to briefly illustrate the idea, two graphs are shown in the **Fig. 2**. The graph on the left hand side corresponds to an unstructured workflow. If the vertex C is cloned into two instances C1 and C2, and if C1 is prerequisite for E and C2 is prerequisite for F the outcome is a structured workflow model presented in the graph on the right hand side. The graph on the right hand side is equivalent to a structured workflow that can be easily transformed into a WF-model (or any other concrete workflow model). In the runtime it has to be guaranteed that C1 and C2 are treated as a single instance of C. After the integration, vertices such as vertex C in this example have to be labeled in order to be cloned later in the process of the WF-model creation.

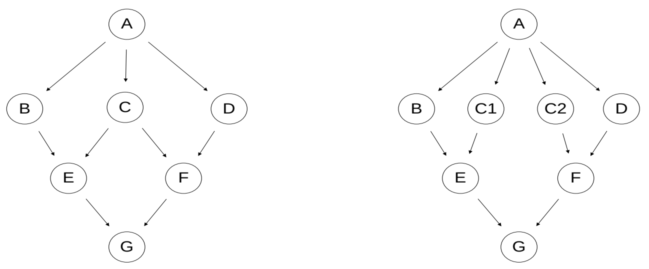

Fig. 2. Unstructured workflow and resulting structured workflow

The complete modeling process consists of several steps as shown in the Fig. 3. Firstly, partially defined prerequisite relationships have to be merged into an integrated graph.

The efficient way of doing the initial integration is to represent the graph with an adjacency matrix. Vertices have to be uniquely mapped to positive numbers ranging from 1 and the total number of the vertices. Creating adjacency matrix eases the detection of (in terms of prerequisites) illicit situations, such as the existence of graph cycles of length 1 and 2. In order to find cycles of length more than 2, algorithms shown in [10] or [20] can be used. The latter is suitable when short cycles are expected.

As the goal is to find at least one cycle and since the topological order is necessary for the next steps of the process, the better approach would be to use the algorithm for incremental cycle detection and topological ordering shown in [6].

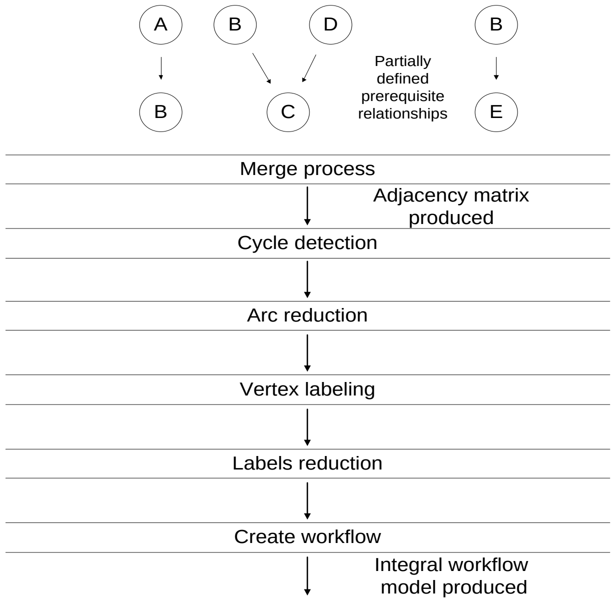
A
B
B
D
C
Partially defined prerequisite relationships
B
E
Merge process
Adjacency matrix produced
Cycle detection
Arc reduction
Vertex labeling
Labels reduction
Create workflow
Integral workflow model produced

Fig. 3. Phases of the workflow creation algorithm

After creating the adjacency matrix and completing the detection of cycles, the algorithm proceeds with arcs removal. Due to transitivity of prerequisite relation, there is a possibility that some of the arcs could be removed. In a nutshell, if for some arc *a = (A,B)* an alternative path from vertex *A* to vertex *B* exists, then the arc *a* can be removed because it is redundant. For example arc *a = (A,B)* in the **Fig. 4** is removed since *A* is a transitive predecessor of *B* via *C* and *D*.[1] This step is similar to the transitive reduction of a graph [3].

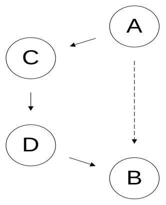

Fig. 4. Removing a redundant arc from A to B

[1] In terms of logic, the situation from **Fig. 4** can be expressed as B=>A, B=>D, D=>C, C=>A, which is equivalent to B=>D, D=>C, C=>A. (B=>A is a surplus)

An alternative to cycle detection and transitive prerequisites removal is the use of algorithms for Boolean satisfiability problem (SAT). Each arc *a=(i,j)* means that *i* is a prerequisite for *j* and such dependency can be logically represented as *j* entails *i*. The logic implication *j=>i* is evaluated to false only in case that *j* is true and *i* is false which corresponds to the situation where some activity is started without its prerequisite being finished. All other combinations (both false, both true, only the prerequisite has finished) are considered to be valid. Other options can be the algorithms presented in [16] and [26], but as prerequisite relationships produce only and-splits and and-joins the initial approach is enough.

Depending on the arrangement of the prerequisites it might happen that a graph is not connected which is not suitable for the algorithm steps that should follow. Therefore, after the reduction of arcs algorithm, two new vertices have to be added to the graph: *Start* and *End*. *Start* vertex will be connected with all vertices having number of inward arcs equal to zero in such a way that *Start* is their predecessor. All vertices having their outward degree equal to zero will be connected with the *End* vertex in such way that *End* is their successor. Consequently, it is guaranteed that the graph is connected that is essential for the vertex labeling algorithm, illustrated in the next section.

4. The Vertex Labeling Algorithm

The vertex labeling algorithm can be applied to a directed graph $G = (V, A)$ with the set of vertices V and set of directed arcs A according to the following presumptions:

- Graph G is connected and there are no cycles in the graph
- There is only one start vertex $v \in V$ such that its indegree $\deg^{-}(v) = 0$
- There is only one end vertex $v \in V$ such that its outdegree $\deg^{+}(v) = 0$
- For each pair of vertices $p \in V$ and $q \in V$ such that exists an arc between them ($a \in A$, $a = (p, q)$) there is no other path $p v_1 v_2 \ldots v_n q$ ($v_i \in V$) from p to q in graph G.

Cycle detection, arcs reduction and the introduction of *Start* and *End* vertices ensure that the previous presumptions are satisfied.

Definition 1. Vertex label is a string that contains only natural numbers and dots.

Set of all possible labels is marked with $\mathcal{L}$. Mapping $\ell \colon V \to L$, where $L \subset \mathcal{L}$, assigns one or more labels to each vertex in the graph. The vertex labeling algorithm is presented in the **Table 1**.

Labeling starts from the *End* vertex that receives label 1 and is added to the set of “open” vertices. Open vertex is a vertex that still has not forwarded all its labels to its predecessors. The labeling algorithm will in each step take a new vertex from the set of open vertices. The chosen vertex will forward its labels to its predecessors according to the following rules.

- If the vertex has only one predecessor then all its labels are added to the predecessor’s label set without being changed.
- If the current vertex has N predecessors (N ≥ 2) then each label is concatenated with *.i* where *i* is a predecessor’s order number. In this way the first predecessor in the order will receive all vertex labels concatenated with *.1*. For the second predecessor *.2* will be concatenated and same principle respectively applies to all other predecessors until the *N*-th predecessor which receives labels concatenated with *.N*. The current vertex will be removed from the set of open vertices and its predecessors will be added to the set.

Table 1. Vertex labeling algorithm

```
1. Assign label '1' to End vertex and add it to the open vertices set
        $\ell$ (End) := {'1'} , O := {End}

2. Let curr be the last vertex in the topological order of set O and let P
   be a set of its predecessors in the graph.
      $curr := w \in O$ such that $v < w, \forall v \in O, v \neq w$
      $P := \{ v \mid \exists a \in A$ such that $a = (v, curr)\}$

      if |P| = 1 then
            $\ell(v) := \ell(v) \cup \ell(curr)$ where $v \in P$
      if |P| > 1 then
            for each $v \in P$ do
                  $\ell(v) := \ell(v) \cup concat(\ell(curr),$ '.orderNumber')
                  where concat is function that will add text from the
                  second parameter as a suffix to each label in label
                  set in the first parameter.
                  orderNumber represents the order number of each
                  predecessor in P (possible values are 1 to |P|)
      O := (O ∪ P) \ {curr}

3. If |O| > 0 repeat step 2.
```

If a random or depth first algorithm is applied in choosing a new vertex in each step of the algorithm, the chosen vertex could be added to and removed from the set of open vertices several times depending of the vertex order and the number of its successors. This can be solved by tagging the labels as they are forwarded. Instead, we choose to take vertices in a reversed topological order, which ensures that at the time of dealing with a vertex all its successors have already forwarded their labels.

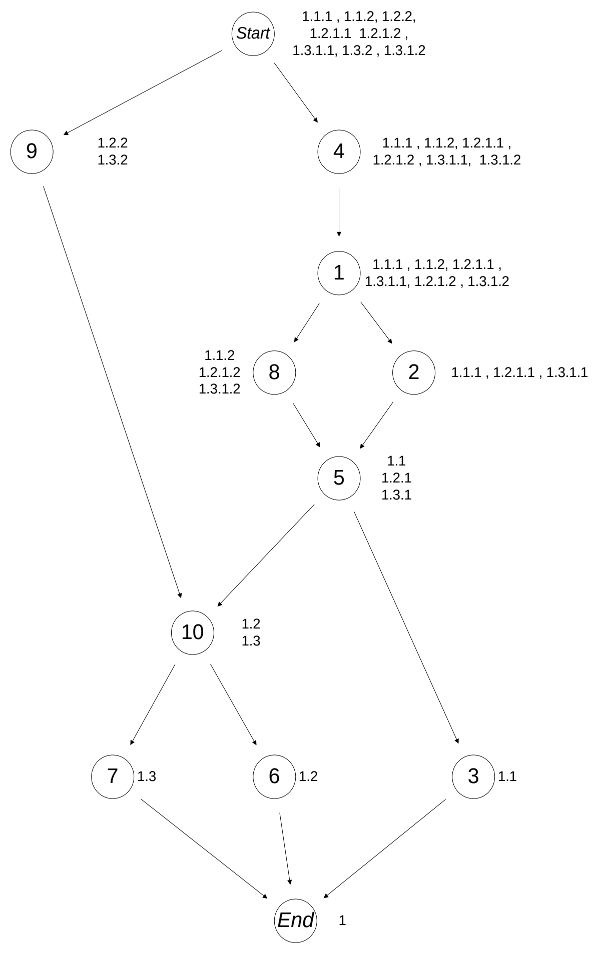

Fig. 5. An illustration of the vertex labeling algorithm operation

An example of the vertex labeling algorithm is shown in the **Fig. 5**. Presuming the topological order was *Start*, 9, 4, 1, 8, 2, 5, 10, 7, 6, 3, *End*, in the first step of the algorithm, label 1 is assigned to *End* vertex and *End* is added to the set of open vertices. Vertex *End* has 3 predecessors – 3, 6 and 7 respectively. As *End* has more than one predecessor, label 1 is suffixed with .1 and added to the label set of vertex 3, to label set of vertex 6 with suffix .2 and to label set of vertex 7 with suffix .3. After that, vertex *End* is removed from the set of open vertices and vertices 3, 6 and 7 are added to that set. Vertex 3 is the last in the topological order of all open vertices and it is processed in the next step. It has only one predecessor (vertex 5) and its labels (the only label is 1.1) are added to vertex 5 without modification. Vertex 3 is removed from the set of open vertices and vertex 5 is added to the set. The process proceeds further with vertex 6 (that copies its label to vertex 10) and after that with vertex 7. At this point the set of open vertices consists of vertices 5 and 10. Vertex 10 is processed because it is behind vertex 5 in the topological order. Vertex 10 has two predecessors and each of them receives labels from vertex 10. This way, vertex 5 receives 1.2.1 and 1.3.1 and vertex 9 receives 1.2.2 and 1.3.2. Step 2 is repeated until the *Start* vertex with no predecessors is processed making the set of open vertices empty.

5. The Label Reduction Algorithm

Since the cardinality of a vertex label set is the number of the clones of the vertex while producing a WF-model, it is reasonable to reduce the cardinality of each set when possible. A brief look at the **Fig. 5** leads to a conclusion that vertices 1 and 5 should have the same labels. There is no arc from vertex 1 that is not joined in vertex 5 and there is no inbound arc for vertex 5 that cannot be traced back to vertex 1. For example, a part of the WF-model is presented in the **Fig. 6** where parallel activities occur after vertex 1 followed by activity created for vertex 5 that was added after the synchronization point. Therefore label set of vertex 1 can be reduced to have same labels as vertex 5.

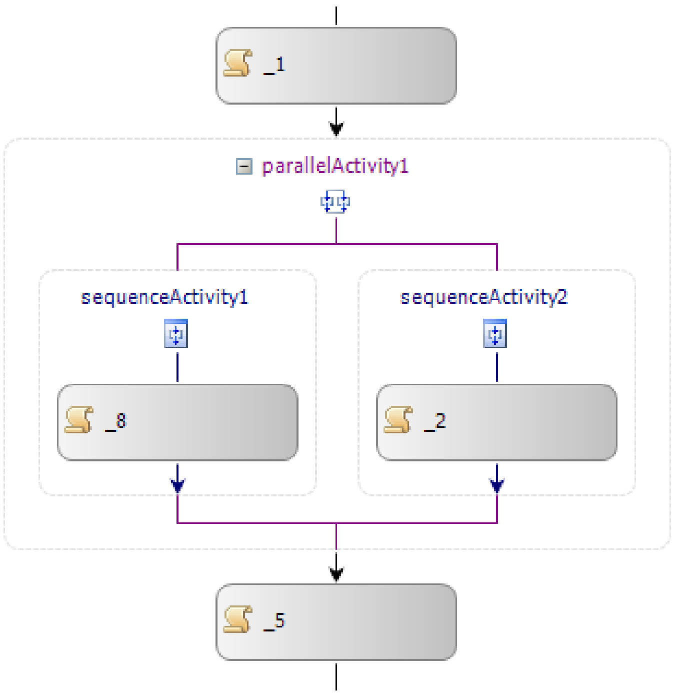

Fig. 6. WF-model for excerpt of graph from Fig. 5

Definition 2. Function *level*: $\mathcal{L} \rightarrow \mathbb{N}$ (where $\mathcal{L}$ is the set of all possible labels) is defined as the number of dots inside a label.

In our example, *level* (1) is 0 and *level* (1.3.1.1) is 3. The relationship between labels will be expressed as a parent-child and an ancestor-successor relation. For instance, label 1.3.1.1 is a child of label 1.3.1 and a successor of label 1. Vice versa, label 1 is an ancestor of labels 1.3.1 and 1.3.1.1, while label 1.3.1 is a parent of label 1.3.1.1.

Definition 3. Label l is a parent of label m if *level(*m*) = level(*l*)+1* and m and l are the same until the last dot in both labels.

Definition 4. Label l is an ancestor of label m if l is a parent of m or there exists a chain of labels $l_1 \dots l_n$ such that l is a parent of l_1, $l_1$is a parent of l_2,… and l_n is a parent of m.

The basic idea behind the label reduction algorithm is to replace labels with their common parent. The replacements are done for vertices having all children of a particular parent label. For example, if all children of label 1.2 in a graph are labels 1.2.1, 1.2.2, 1.2.3 then for all vertices that contain all those three labels, those three labels are replaced with 1.2.

The label reduction algorithm groups labels by value of level function and starts checking labels backwards from the next to the last level until it comes to the first level where the return value of function *level* for the first level is equal to zero. For each label l in current level (marked as set U in the following algorithm), set S is a set containing all children of the label l. The next step is finding all vertices (set V') that contain all labels from the set S. This way, vertices containing all children of the label l are found and for those vertices the replacement can be done. In the end, for each vertex from set V', all labels from the set S are removed from its label set and label l is added.

The algorithm continues until all labels on all levels have been checked, i.e. until label 1 (the only label at the level 0) is checked. Formal definition of the algorithm follows in the **Table 2**.

Table 2. Label reduction algorithm

```
1.  curr_level := $\max_{l \in \mathcal{L}} level(l)$ – 1

2.  Let U be the set of all labels from current level
        U: = { $l \in \mathcal{L}$ | level($l$) = curr_level}
        for each label $l$ from set U do
            S := { $m$ | $m \in \mathcal{L}$ such that $l$ is a parent of $m$}
            If S ≠ ∅ then
                V' := { $v$ | $v \in V$ such that $m \in \ell(v)$, $\forall m \in$ S}
                for each $v \in$ V' do
                    $\ell(v)$ := ($\ell(v)$ \ S) ∪ { $l$ }
3.  curr_level := curr_level - 1
      If curr_level ≥ 0 repeat step 2.
```

For the graph from the **Fig. 5** label reduction occur for the vertices 1, 4 and *Start*. Labels 1.3.1.1 and 1.3.1.2 are replaced with 1.3.1, labels 1.2.1.1 and 1.2.1.2 are replaced with 1.2.1 and labels 1.1.1 and 1.1.2 are replaced with 1.1. Next replacements occur only for the vertex *Start*. Newly added labels 1.3.1 and label 1.3.2 are replaced with 1.3. Newly added labels 1.2.1 and 1.2.2 are replaced with 1.2. Therefore, a new replacement can be done. Labels 1.1, 1.2 and 1.3 are replaced with 1.

After the algorithm ends, vertex *Start* must only contain label 1. **Fig. 7** shows the graph after the label reduction has been applied to the graph in the **Fig. 5**.

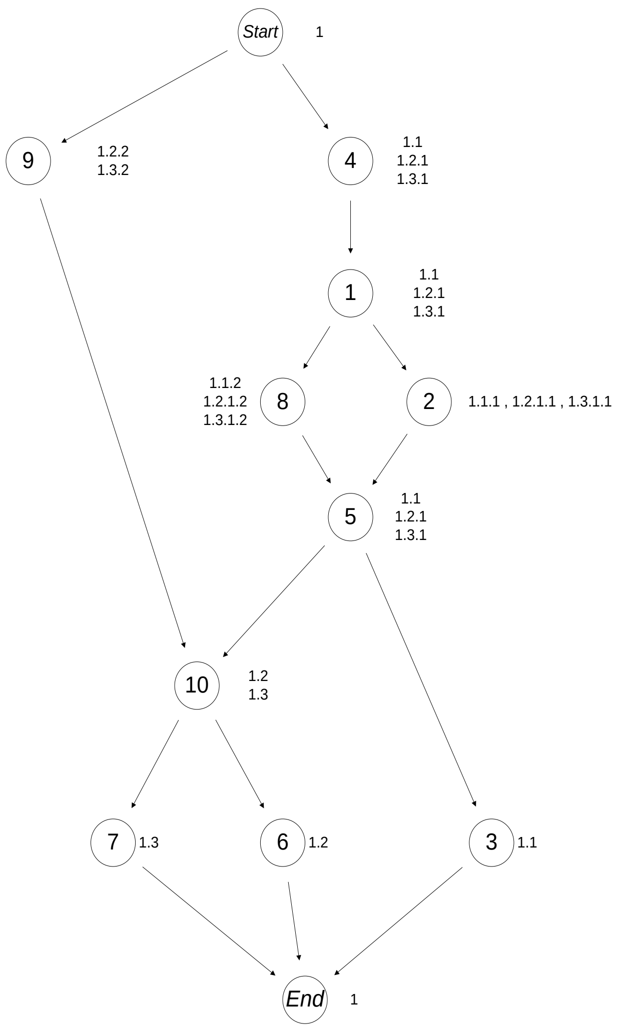

Fig. 7. An example of a graph after the reduction of labels

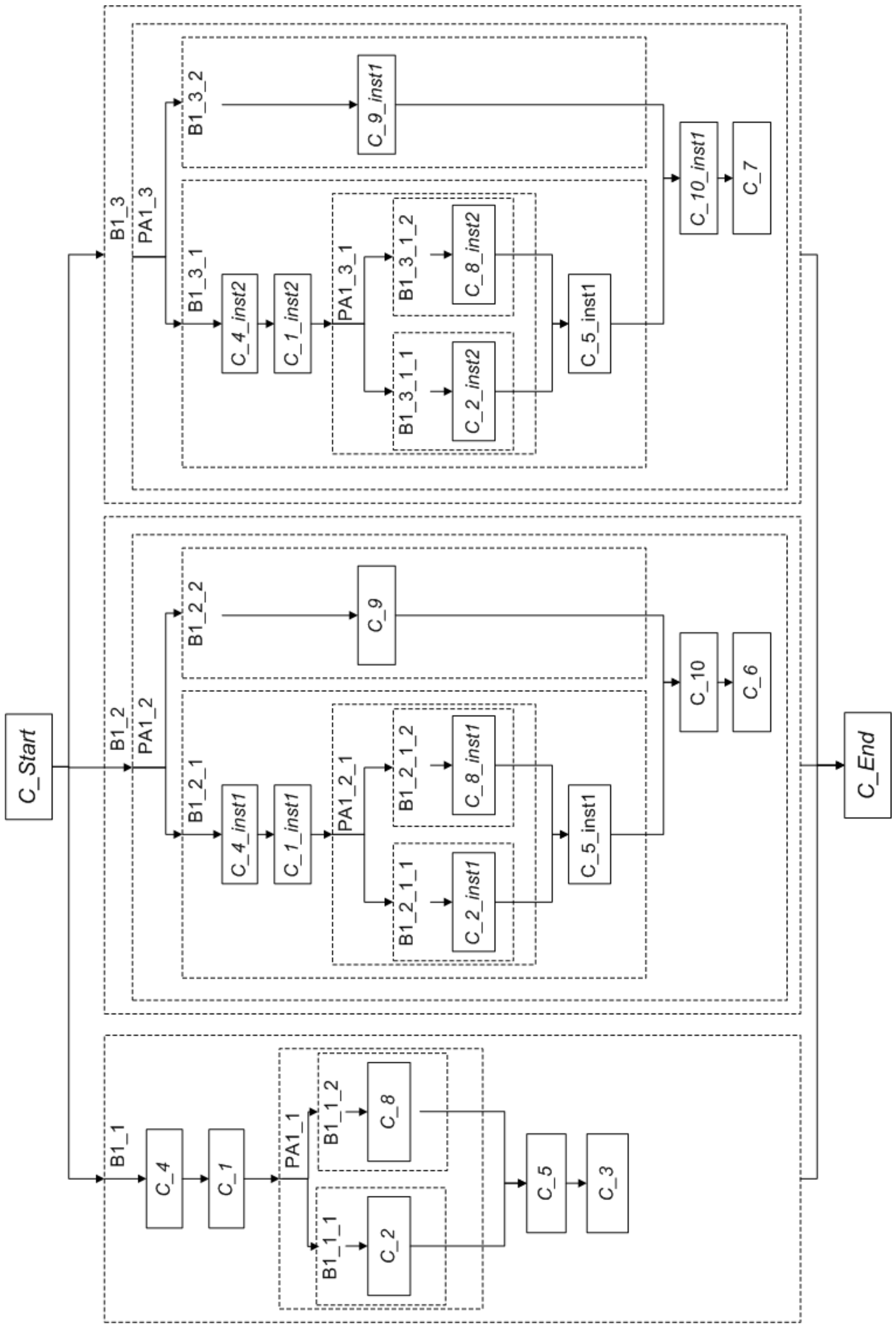

Fig. 8. WF-model for the labeled graph from Fig. 7

After the labels have been reduced, a WF-model can be created. Some vertices will be cloned and the corresponding WF-elements will be added as several parallel branches. WF-model for the previous example is presented in the **Fig. 8** where the elements are named with C_{vertex number}, parallel activities are named with PA_{label name} and branches of parallel activities are named with B_{label name}. In case a vertex had to be cloned, inst_{clone instance number} is appended to the element name in order to ensure that the element names remain unique. The principle of the WF-model creation is described in the next section.

6. WF-model Creation

Each element of a created WF-model will be either an activity that represents a clone of a vertex in the labeled graph or a built-in activity (*ParallelActivity* or *SequenceActivity*). In every WF-model at least one (main) sequence must exist, which contains all elements having label 1. For any other label corresponding new parallel activities and parallel branches will be created and assigned to the particular label. For example, vertex with the label 1.3.1.2 would be added in a branch assigned to the label 1.3.1.2. If such a branch does not exist it is created as a branch of a parallel activity assigned to the label 1.3.1. If such a parallel activity does not exist, it is created in the branch assigned to the label 1.3.1. This procedure is recursive and finishes on the main sequence assigned to the label 1. Names of the activities must be unique and follow variable naming rules in which parallel activities are prefixed with PA_ and branches of parallel activities take prefix B_.

For the graph from the **Fig. 7** a label is uniquely assigned to a particular parallel branch in **Fig. 8** and branch where an activity for a vertex will be added can be uniquely determined. It is important to note that a label does not have to uniquely identify a branch as it can be seen in **Fig. 9** and **Fig. 10**. Label is rather a indicator where a particular element would be nested within the workflow relative to its current position. The main benefit of this principle is that label set can be further reduced.

Vertices from the graph are processed in the topological order, which ensures that all merge points are added after its parents. For instance, a topological sort for the graph in **Fig. 9** can be *Start*,1,2,5,3,4,6,7,8,*End*. The algorithm starts by adding vertex 1 with label 1 (WF-element has name C_1) to the main sequence activity. Afterwards, an element representing vertex 2 with label 1.1 is processed and must be added to the branch assigned to label 1.1. As such branch still does not exist and a parallel branch assigned to label 1 does not exist, a new parallel activity (named PA_1) is added to the main sequence and assigned to label 1. Subsequently branch of a parallel activity PA_1 is created, named B_1_1 and assigned to label 1.1 upon which WF element for vertex 2 is added to the newly created branch.

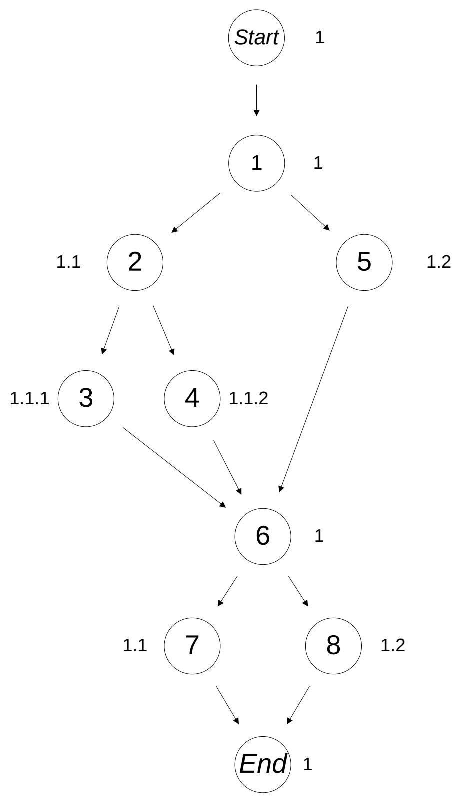

Fig. 9. A simple graph illustrating the creation of a WF-model

Similarly, element that represents vertex 5 is added to the branch B_1_2 assigned to label 1.2. Vertex 3 has label 1.1.1, which means that there must exist a branch assigned to label 1.1.1 as a branch of a parallel activity assigned to 1.1. As both do not initially exist, they will be created. The same procedure follows for all other vertices until vertex 6 is reached. Vertex 6 is the synchronization point of branches 1.1 and 1.2. As it has label 1 it will be added as the next child of the main sequence (after parallel activities). It is important to note that appearance of a label that is an ancestor of an existing label causes that all mappings between successor labels and parallel activities and branches get removed. These way vertices 7 and 8 are not added to the “old” branches B_1_1 and B_1_2 respectively, but a new parallel activity PA_1_inst1 is created containing branches B_1_1_inst1 and B_1_2_inst1. PA_1_inst1 is added to the main sequence after the element C_6. The new mapping for labels 1.1 and 1.2 is active until the next occurrence of label 1. WF-model of the graph in the **Fig. 9** is shown in the **Fig. 10**.

If a vertex has more than one label, previously described steps are repeated for each label.

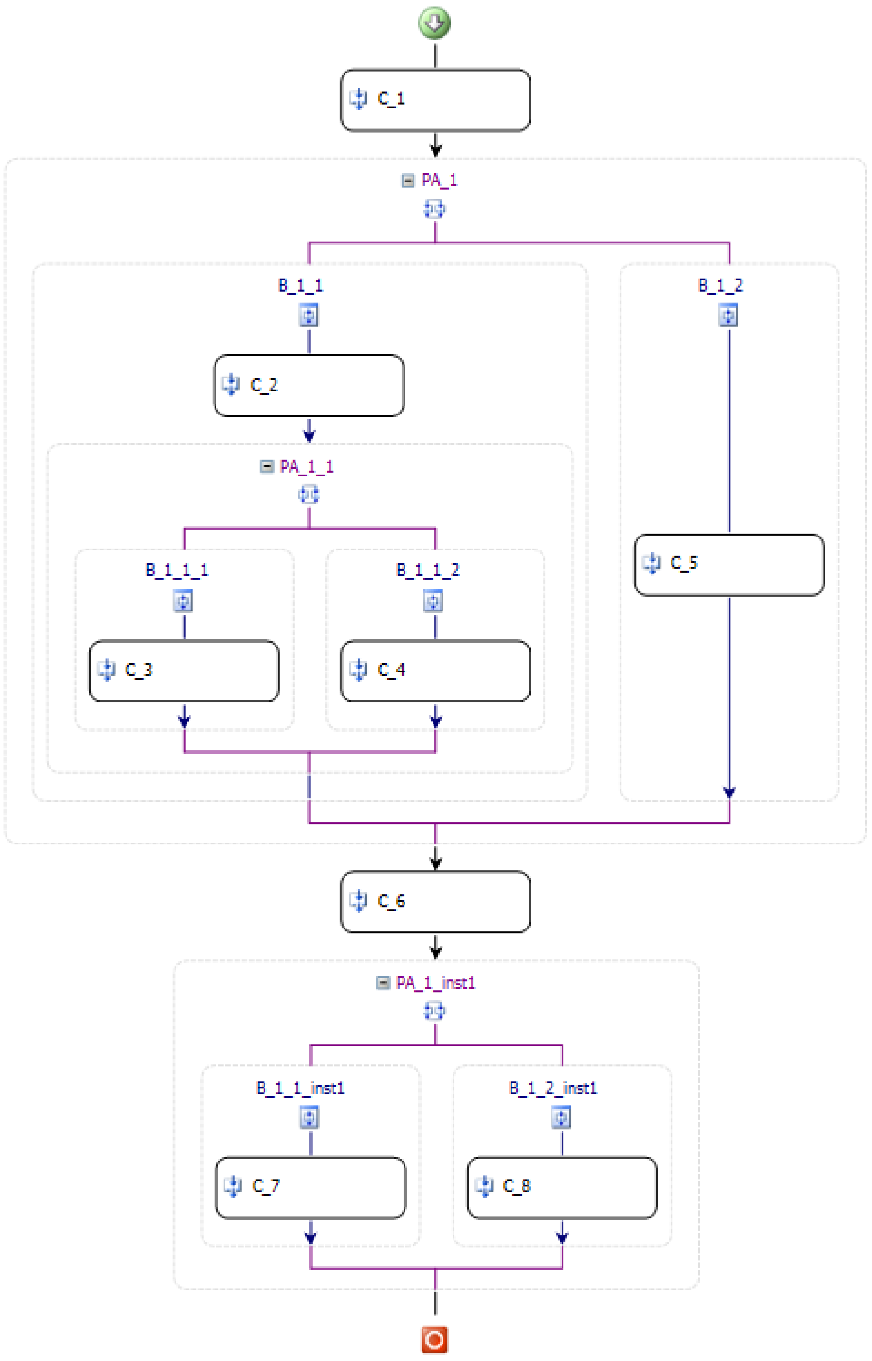

Fig. 10. An WF-model for the labeled graph in Fig. 9

Pseudo code for the WF-model creation algorithm is given in the **Table 3**. Prior to the formal definition of the algorithm two mappings must are defined.

Definition 5.
$p : \mathcal{L} \to$ PA is mapping between a particular label and the assigned parallel activity, where $\mathcal{L}$ is a set of all labels and PA is set of all parallel activities in WF-model

$b : \mathcal{L} \to$ B is mapping between a particular label and the assigned branch, where $\mathcal{L}$ is a set of all labels and B is a set of all branches for parallel activities in WF-model

with $b(1)$ initially defined in order to assign the main sequence to label 1.

Table 3. The WF-model creation algorithm based on a labeled graph

```
Let $G(V, A)$ is a graph labeled according to the algorithms from Table 1 and Table 2

for i:=1 to i≤ $|V|$ do
    $v$ := ith element from the topological order of the graph $G$

    for each label $l \in \ell(v)$ do
        D := { $m$ | $m \in \mathcal{L}$ such that $l$ is an ancestor of $m$}

        if D ≠ ∅ then
            for each $m \in$ D do
                $p(m)$ := undefined
                $b(m)$ := undefined

        if $b(l)$ is not defined
            createBranch($l$)

        add element C_$v$[2] in branch $b(l)$
```

Recursive function *createBranch* creates a branch for a given label. It depends on the function for creating parallel activity. Both functions are described in the next table.

[2] Name of the element can contain suffix *_instX* where X is the number of elements already created for vertex v

Table 4. The algorithm for creating a branch and parallel activity for a given label

```
createBranch($l$)
    if $p(parent(l))$ is not defined
        createParallelActivity($parent(l)$)
    $b(l)$ := create branch B_ $l$[3] as a branch of parallel activity $p(parent(l))$

createParallelActivity($m$)
    if $b(m)$ is not defined
        createBranch($m$)
    $p(m)$ := create parallel activity PA_$m$ as a next activity in branch $b(m)$
```

7. Correctness of the Workflow Creation Algorithm

In order to prove the correctness of the workflow creation algorithm it is necessary to prove two main claims. The first one is that algorithms for graph integration, arc reduction, vertex labeling and label reducing preserve all of the initial prerequisite relationships and create no new prerequisites. The second one is that the WF-model creation algorithm creates correct sequences and parallel activities.

By looking into the steps of the complete process of workflow creation it becomes clear that the first two steps only transform the partially defined prerequisites into a graph representation without any modifications. The third step (arc reduction) removes arcs representing prerequisite relationships that are already present through transitivity of some other prerequisites. The graph is modified but the meaning has not been changed – all prerequisites are kept either in the original relationship or as a set of transitive prerequisite relationships.

To show the correctness for the remaining algorithm steps (labeling, label reduction and WF-model creation) the following claims should be proven:

- All vertices are in correct order, which means that all prerequisite relationships are satisfied.
- Execution of each element depends only on its prerequisites; i.e. there is no waiting for elements which are not prerequisites.

From the design of the WF-model it is obvious that in order to have two elements run in parallel, it is imperative that their labels are not in an ancestor-successor relationship. The vice versa statement does not have to be valid (see **Fig. 10**, labels 1.1 and 1.2). The claim is formally stated through the following theorem.

Theorem 1. The non existence of an ancestor-successor relationship between labels l_x and l_y, where $l_x \neq l_y$, is required in order to have a WF-

[3] Name of the element can contain suffix *_instX* where X is the number of elements already created for vertex v

element for vertex X with label l_x and a WF-element for vertex Y with label l_y both running in parallel.

Proof: Without loss of generality it can be assumed that vertex X is before vertex Y in the topological order. There are two possible cases that must be discussed. In the first case there is no vertex between X and Y in the topological order having a label which is a common ancestor of labels l_x and l_y. The second case is that such vertex exists which shows that the presumption about non existence of the ancestor-successor relationship is not enough to run the elements in parallel.

Case I. According to the algorithm presented in the **Table 3**, the branches of parallel activities are created in a way that a branch B assigned to label l is branch of the parallel activity PA assigned to label $parent\,(l)$. Parallel activity PA is added to the branch assigned to label $parent\,(l)$ which is a branch of the parallel activity assigned to label $parent\,(parent\,(l))$ etc. Creating activity for vertex X with label l_x will create (if such does not already exists) a branch assigned to the label l_x. The same principle applies to the vertex Y with the label l_y. If $l_x = l_y$ then it is the same branch and X and Y are in the same branch which also means they are in the same execution line. If l_x is an ancestor of l_y then, according to previously described algorithm, the branch assigned to the label l_y is contained inside the branch assigned to the label l_x. If l_y is an ancestor of the label l_x it means that branch assigned to the label l_y already exists and activity representing a clone of the element Y is added to the end of that branch which means that Y is a synchronization point for branches contained in the branch assigned to l_y putting X and Y in the same execution line. If l_x and l_y are not in an ancestor-successor relationship then it is obvious that they can be run in parallel.

Case II. Appearance of label l that is a common ancestor of labels l_x and l_y, according to the algorithm from the **Table 3**, causes all assignments between branches and successors of the label l to be removed. As such, assignments for labels l_x and l_y are also removed. The clone of vertex with label l, such as one between X and Y, is a synchronization point for all branches that have been assigned to the labels l_x and l_y. This fact causes that when an element for the vertex Y is going to be created it gets added to the newly created branch assigned to l_y and that branch is a part of the branch assigned to the label l and, as such, is in the same line of execution with X.

■

As an outcome from the proof of the Theorem 1, Corollary 2 follows directly.

Corollary 2. Clone of a vertex X with label l_x and clone of a vertex Y with label l_y run in parallel if $l_x \neq l_y$, labels l_x and l_y are not in an ancestor-successor relationship and there are no vertices between X and Y in a topological order with a label that is common ancestor of labels l_x and l_y.

By observing not only a single clone, but all clones of a vertex it can be stated that two vertices are in the same line of execution if for each clone of one vertex exists a clone of the second vertex such that those two clones are in the same line of execution. If not, these vertices can be run in parallel. For example, for the graph from the Fig. 7 vertices 4 and 10 are in the same line of execution because for each label of vertex 10 (labels are 1.2 and 1.3) exists a label of vertex 4 (labels are 1.1, 1.2.1, 1.3.1) in an ancestor-successor relationship with that label. Vertices 9 and 3 represent an opposite situation. With Lemma 4 it will be shown that it is enough to find just one pair of labels in an ancestor-successor relationship so to conclude that the two vertices are in the same line of execution.

Corollary 3. For a WF-model created immediately after the vertex labeling algorithm, two vertices X and Y are in the same line of execution if and only if for each pair of labels $l_x \in \ell$ (X) and $l_y \in \ell$ (Y) label l_x is equal to l_y or label l_x is an ancestor of label l_y or label l_y is ancestor of label l_x.

Proof: The first direction follows directly from the Corollary 2 because there are no vertices between X and Y in a topological order having a label which is a common ancestor of labels l_x and l_y. Therefore l_x and l_y must be in an ancestor-successor relationship. The second direction follows from the Theorem 1, due to fact that nonexistence of an ancestor-successor relationship between labels is required to have two clones run in parallel. If all labels are in an ancestor-successor relationship then no pair of clones of vertices X and Y can be run in parallel, hence vertices X and Y are in the same line of execution.

■

Lemma 4. After the vertex labeling algorithm and before the labels reduction algorithm, for each two vertices X and Y, such that X is predecessor of Y, and for each label $l_y \in \ell$ (Y) exists a label $l_x \in \ell$ (X) such that either $l_y = l_x$ or l_y is an ancestor of l_x.

Proof: Case I. Let X be an immediate predecessor of Y. If X is the only immediate predecessor of Y then all labels of vertex Y are added to set of labels of vertex X without any modification and in this case is ℓ (Y) ⊂ ℓ (X). From this it immediately follows that for each label $l_y \in \ell$ (Y) exists a label $l_x \in \ell$ (X) such that $l_y = l_x$. If the vertex Y has two or more immediate predecessors and the vertex X is the i-th predecessor of Y then each label of the vertex Y is suffixed with *.i* while adding to label set of the vertex X. Therefore, for each label $l_y \in \ell$ (Y) there is a label $l_y.i$ in ℓ (X).

Case II. Let X be a non-immediate predecessor of Y. Then there exists a path $Xv_1v_2..v_nY$ in the graph. Labels from the vertex Y are transferred to v_n, from the vertex v_n to the vertex v_{n-1} and so on until the vertex X. In each step one of the two possible situations from the first case can be applied. Therefore for each label $l_y \in \ell$ (Y) either exists a label $l_x \in \ell$ (X) such that $l_y = l_x$ (in case all vertices in the path had only one immediate predecessor) or l_y

is an ancestor of l_x as a consequence of successive suffixing with dot and the order number of a particular predecessor.

■

Lemma 5. After the vertex labeling algorithm two vertices cannot be in the same line of execution if one vertex is not a predecessor of the other.

Proof: Let vertices X and Y be in the same line of execution, but such that neither X is a predecessor of Y, nor Y is a predecessor of X. Let vertex Z be the first (in the topological order) common successor of the vertices X and Y. Such a vertex must exist because the graph is a connected graph having at least vertex *End* as such a vertex. Since X and Y are according to the presumption in different ancestry lines, after transferring labels from the vertex Z to its predecessors, ancestry lines toward vertex X have received labels of the vertex Z with the suffix *.i*, and ancestry line toward the vertex Y has received labels of the vertex Z with the suffix *.j*. As there was no label reduction, vertices X and Y have labels that are not in an ancestor-successor relationship and according to the Corollary 3 cannot be in the same line of execution which is in the contradiction with the assumption. Therefore X must be a predecessor of Y or Y must be a predecessor of X.

■

Lemma 4 shows that before the label reduction all predecessors and successors of a vertex (and according to the Lemma 5 only these) are in the same line of execution with the vertex due to the existence of an ancestor-successor relationship between their labels. It has to be shown that the label reduction algorithm keeps the execution line.

Lemma 6. Two vertices X and Y are in the same line of execution after the label reduction algorithm if and only if they were in the same line of execution before the label reduction algorithm.

Proof: $\Rightarrow$ Let vertices X and Y be in the same line of execution after the label reduction algorithm. If it is assumed that X and Y were not in the same execution line before the label reduction, then their lines of execution split at their common ancestor and join in the first (in the topological order) common successor. As labels coming from the common successor are propagated to two different lines of execution with different suffixes, neither one element in the different ancestry lines (including X and Y) cannot have all children of a label l from the common successor's label set and that is true for each label l of the common successor. Therefore X and Y cannot reduce their label set to have same labels as those in the label set in the other execution line or to become their ancestors or successors. According to the Lemma 3 this means that they were in the different lines of execution and no label reduction could occur.

$\Leftarrow$ Let vertices X and Y be in the same line of execution before the label reduction algorithm and let, without loss of generality, X be a predecessor of Y. This means that before the label reduction each label $l_y \in \ell$ (Y) was equal to a label $l_x \in \ell$ (X) or l_y was an ancestor of the label l_x. Label reduction algorithm can change the label set of the vertex X and/or the vertex Y and three different mutual relationships between the vertices in the same line of execution have to be observed.

In the first one, every path from the vertex *Start* to the vertex Y includes the vertex X and every path from the vertex *Start* to the vertex *End* such that includes the vertex X also includes the vertex Y (e.g. vertices 1 and 5 from **Fig. 5**). In this case, after the label reduction algorithm vertices X and Y will have the same labels because X contains all children of labels originated by coping labels of the vertex Y to its predecessors.

In the second case each path from the vertex *Start* to the vertex Y includes the vertex X, but there exists a path from the vertex *Start* to the vertex *End* that includes the vertex X and does not include the vertex Y (e.g. vertices 1 and 2 from the **Fig. 5**). If the vertex X had contained labels that have the same parent as the labels of the vertex Y, X's label set would be reduced and those labels would be replaced with their parents. In the successive reduction it may happen that for a label $l_y \in \ell$ (Y) its corresponding label $l_x \in \ell$ (X) is swapped with a label that is an ancestor of l_x. Therefore the new label is also in an ancestor-successor relationship with the label l_y, and according to the Lemma 3, vertices are still in the same line of execution.

The third case is represented with a situation in which there exists a path from the vertex *Start* to the vertex Y which does not include the vertex X (e.g. vertices 4 and 7 from the **Fig. 5**). (The existence of the vertex Y in any path from the vertex *Start* through X to the vertex *End* is irrelevant.) In this case the vertex X in the label reduction algorithm cannot remove a label $l_x \in \ell$ (X) that is a successor of a label $l_y \in \ell$ (Y), because there exists at least one label that is a successor of the label l_y in another ancestry line and as such is not an element of ℓ(X) and therefore the relationship between X and Y has not been changed.

■

Theorem 7. Every clone of a graph's vertices transformed into WF-activities depends only on its predecessors in the graph and no clones can be executed before all predecessor clones have been finished.

Proof: The Lemma 4 proves that all labels of a vertex are propagated to its predecessors and no predecessors of the vertex are omitted. This means that all predecessors are in the same line of execution before the label reduction. Lemma 5 proves that for each vertex, a vertex in the same line of execution cannot exist if it was not a predecessor of a successor of the vertex. Lemma 6 ensures that the label reduction algorithm keeps the line of execution, which means that all predecessors are preserved and no new dependencies have been added. Mutual vertical order of the clones inside the WF-model is ensured using topological ordering.

■

8. Algorithms Complexity

Complexity of the presented algorithms depends on the number of (different) labels, which is dependent not only on the number of vertices and arcs in a graph, but also on the graph's structure. As the worst case for a particular algorithm step can lead to significant simplification of another step in the process, in this section we give main guidelines on how to calculate the number of labels and which implementations to use in order to reduce the complexity of the presented algorithms. It is important to stress out that the given complexity is the upper bound and it can be far from the average complexity.

In the step 2 of the vertex labeling algorithm from the **Table 1**, each vertex having indegree greater than one creates new labels in the graph. A contribution of the vertex to the number of new labels in a graph is a product of the cardinality of the vertex label set and the number of the inbound arcs. The cardinality of a vertex label set represents the number of different paths from that vertex to the *End* vertex and can be calculated as a sum of cardinalities of label sets of its successors. For a vertex v in a graph $G = (V, A)$ this contribution of a vertex v can be expressed with the following formula

$$c(v) = \begin{cases} deg^{-}(v) * |\ell(v)| & , deg^{-}(v) > 1, v \neq End \\ deg^{-}(End) & , deg^{-}(v) > 1, v = End \\ 0 & , deg^{-}(v) = 1 \end{cases}$$

where

$$|\ell(v)| = \sum_{w \in S} |\ell(w)|$$

and $S = \{ w \mid \exists\, a \in A$ such that $a = (v, w)\}$.

This way, total number of different labels in the graph is equal to the sum of contributions of all vertices incremented by one due to the label 1 initially added to graph and assigned to the vertex *End*.

$$|\mathcal{L}| = 1 + \sum_{v \in V} c(v)$$

In the vertex labeling algorithm from the **Table 1**, a sub-step of the step 2 is executed $|A|$ times, where $|A|$ is the number of arcs in the graph. The complexity of the algorithm then depends on the implementation of relationship between labels and vertices, implementation of the union operation (with concatenation) and the number of labels in a vertex.

As it is shown later in this section, it is convenient to implement vertex label set as a hash table and store all labels in a tree, where each node additionally contains a pointer to the workflow branch assigned to the label and a hash table of pointers to vertices that have the label in its label set. If the tree is balanced then the addition of a new node to the tree has complexity of $O(\log|\mathcal{L}|)$ and the addition of a new vertex pointer into the hash table for the retrieved node can be done in constant time. As insertion of a new vertex in the set of open vertices in such a way that vertices are always sorted by topological order and producing the union of labels is linear in relation to the number of vertices and the number of labels respectively, it can be claimed that the complexity of the vertex labeling algorithm is $O(|A| * (|\mathcal{L}| + |A|))$.

With labels stored in the tree, the label reduction algorithm can be implemented as an inorder tree traversal in which for each non-leaf node a slightly modified step 2 from the **Table 2** is executed. In that case finding the set V' for label l in step 2 from the **Table 2** is, complexity-wise, equal to finding the intersection of all hash tables with vertices pointers to the children of the node traversed and each intersection can be executed with linear complexity. Replacement of children labels with label l for all vertices from the set V' is done with the following single step: for a vertex from intersection and for a child label m of the label l, m is deleted from the vertex label set, and vertex pointer is deleted from the hash table in the node m. As the deletion from hash table can be done in constant time and as this occurs for each vertex and each label in each step of a traversal process, the complexity is $O(|V| * |\mathcal{L}|^2)$.

A single step of the inner loop in the WF-model creation algorithm consists of removing the existing mapping, creating a new branch and adding an element to the branch. This step is executed $\sum_{v \in V} |\ell(v)|$ times. If the tree is balanced, finding an assigned branch to a label is done in logarithmic complexity. When a mapping between a label and a branch has to be removed, the pointer to the branch has to be deleted from the tree. Instead of deleting pointers to the branches in entire subtree, to save on complexity it is enough to delete just the pointer in root of the subtree. This implies that the creation of a new branch has to be modified in such a way that, except assigning new pointer to a node, the deletion of all pointers in the child has to be done. According to this, it can be stated that complexity of the WF-creation algorithm is $O\left(\sum_{v \in V} |\ell(v)| * \log|\mathcal{L}|\right)$.

9. Possible Improvements of the Algorithm

Although the algorithm from the **Table 2** reduces the number of clones, there are some drawbacks to it. The **Fig. 8** shows that the two rightmost branches with labels *B_1_2* and *B_1_3* are almost the same, except for the elements *C_6* and *C_7*. The presented WF-model can be improved by deferring the parallel split just before the elements *C_6* and *C_7*. In that case all previous elements can be added to the common branch. Hence, the possible improvement consists of merging the parts of parallel branches that share the same parent parallel activity. In our example this means that branches assigned to labels 1.2 and 1.3 are the same (i.e. 1.3 can be replaced with 1.2) for all vertices before the vertex 10 (including vertex 10). Therefore, for the labels in those vertices, starting pattern 1.3 could be replaced with 1.2. For all vertices after the vertex 10 label 1.2 should be replaced with 1.2.1 and 1.3 should be replaced with 1.2.2.

It turns out that the more “regular” a graph is (in terms of adequately paired and-splits and and-joins) the surplus of the clones becomes more obvious. By looking at the **Fig. 11** one can notice that after the label reduction, label set of the vertex 3 has been reduced from {1.1, 1.2} to {1}, but the same thing does not apply to the vertices 1 and 2. Their label sets can be reduced by just replacing starting patterns of the labels which will, in turn, produce the graph shown in the **Fig. 12**.

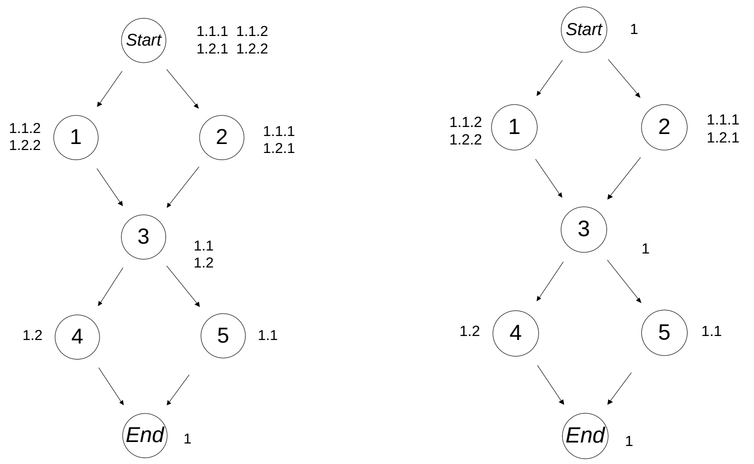

Fig. 11. A graph after the vertex labeling and the same graph after the label reduction algorithm

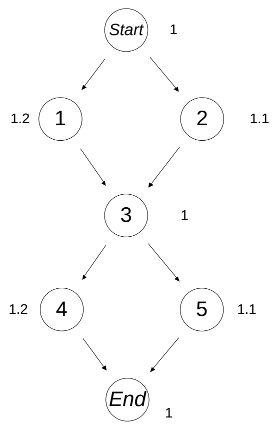

Fig. 12. Optimal labeling solution for the graph on the left hand side in the Fig. 11

Proof of these claims is based on a proof of the Theorem 8 which deals with an improvement that can be done for these vertices where the label set has not been reduced but contains two or more labels that have same parent as the vertex 10 in the Fig. 7. Replacing more labels with the same parent starts from the vertex containing those labels and propagates replacements upwards to the *Start* vertex or until the first vertex without these labels.

Theorem 8. Let the label set of a vertex v contains a subset O={l_1, … ,l_n} of labels that have same parent and let l_1 be the first among them in the lexicographic order. If the starting pattern l_iX is replaced with l_1X for the labels of the vertex v and the labels of v's predecessors and the starting pattern l_j is replaced with l_i.*k* in the labels of v's successors, where k is the position of l_i inside the set O, then it can be claimed the line of execution has been kept.

Proof: Without the loss of generality it can be claimed that the vertex v contains two labels *l.i* i *l.j*, where *l* is the mutual prefix of the labels and i<j. All predecessors of v, among other labels received from their other successors, have labels with the patterns *l.i.suffix* i *l.j.suffix*, where suffix is a string of dots and numbers that were concatenated during the labeling algorithm. (Optionally, v can have other labels that do not start with *l* and, for those labels, in v's predecessors will exist labels, with an appended suffix, that are the successors of those labels so v is in the same line of execution with its predecessors). By replacing prefix *l.j* with *l.i* in the vertex v and in its predecessors, it is possible that a label set of a particular vertex gets reduced, but line of execution will be kept because for each label of the vertex v there exists a label in his predecessors in an ancestor-successor relationship with label from the vertex v. It remains to be shown that replacing prefix *l.i* with *l.i.1* and *l.j* with *l.i.2* in all labels in a v's successor keeps the line of execution. The vertex v has labels *l.i* and *l.j* which means that at least one successor of the vertex v containing label *l* exists. That successor has at least one more predecessor and that predecessor contains label *l.k* (in any other situation the label reduction algorithm would reduce labels *l.i* i *l.j* with label *l* in the vertex v*).* The replacement of *l.i* with *l.i.1* and *l.j* with *l.i.2* implies that branches assigned to labels *l.i.1* i *l.i.2* join one step before the join with a branch assigned to *l.k* (and any other branches).

If set O had more than two labels, the proof is similar.

■

Although this improvement can reduce the number of the labels, it significantly raises complexity as it adds labels of the higher value of a level function after the level has already been processed. Appearance of those labels requires algorithm reset and several re-runs.

10. Conclusion

Dividing prerequisite relationships into partially defined components helps maintaining a relationship between elements and increases readability. Nevertheless, as shown in the introductory example, even for simple models it may be impossible to transform the integrated graph directly into a real workflow model with an existing workflow modeling language because such workflow models do not have to be structured. As an opposite to developing proprietary workflow management software to support unstructured models, an approach consisting of using existing workflow management software is proposed. The approach consists of element cloning (duplication) and a workflow wrapper ensuring the clones are shown as unique elements in the runtime and no data duplication occurs. The description of the wrapper is out of the scope of this paper.

The paper presented the steps for the integration of prerequisite relationships into directed graphs and algorithms for vertex labeling and label reduction. Initial algorithms and enhancements are presented and their correctness has been proved. Furthermore, the complexity of the algorithms has been discussed and the worst case complexity has been given, with the remark that finding average complexities would require extensive tests since the worst case in one of the algorithm's steps can significantly reduce complexity of another step.

References

1. van der Aalst, W., van Hee, K.: Workflow management. Models, Methods and Systems. MIT Press, Cambridge, Massachusetts, USA. (2004)
2. van der Aalst, W., Lassen, K.: Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation. Information and Software Technology, Vol. 50, No. 3, 131-159. (2008)
3. Aho, A.V., Garey, M.R., Ullman, J.D.: The Transitive Reduction of a Directed Graph. SIAM Journal on Computing, Vol.1, No 2, 131-137. (1972)
4. Cesarini, M., Monga, M., Tedesco, R.: Carrying on the e-Learning process with a Workflow Management Engine. In Proceedings of the 2004 ACM Symposium on Applied Computing. Nicosia, Cyprus, 940-945. (2004)
5. Gruhn, V., Laue, R.: Good and bad excuses for unstructured business process models. In Proceedings of 12th European Conference on Pattern Languages of Programs, Irsee, Germany. (2007)
6. Haeupler, B., Telikepalli, K.: Mathew, R.; Siddhartha S.; Tarjan, R.E.: Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. (2008). [Online]. Available http://people.csail.mit.edu/haeupler/incremental-topological-ordering-journal.pdf (October 2010)
7. Hausser, R., Friess, M., Küster, J.M., Vanhatalo, J.: Combining Analysis of Unstructured Workflows with Transformation to Structured Workflows. In Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, Hong Kong, China, 129-140. (2006)
8. Hlaoui, Y.B., Ayed, L.J.B.: An Interactive Composition of UML-AD for the Modelling of Workflow Applications. Ubiquitous Computing and Communication Journal, Vol. 4, No. 3, 599-608. (2009)
9. Holl, A., Valentin G.: Structured business process modeling (SBPM). Information Systems Research in Scandinavia (IRIS 27), Falkenberg, Sweden. (CD-ROM). (2004)
10. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing, Vol. 4, No. 1, 77-84. (1975)

11. Kiepuszewski, B., ter Hofstede, A., Bussler, C.: On Structured Workflow Modelling. Lecture Notes in Computer Science, Vol. 1789. Springer-Verlag, Berlin Heidelberg, 431-445. (2000)
12. Kiepuszewski, B.: Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows. PhD Thesis, Queensland University of Technology, Brisbane, Australia. (2003)
13. Lassen, K., van der Aalst, W.: WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. Lecture Notes in Computer Science, Vol. 4275. Springer-Verlag, Berlin Heidelberg, 127-144. (2006)
14. Lin, L., Ho, C., Sadiq, W., Orlowska, M.E.: Using Workflow Technology to Manage Flexible e-Learning Services. Educational Technology & Society, Vol. 5, No. 4, 116-123. (2002)
15. Liu, R., Kumar, A.: An analysis and taxonomy of unstructured workflows. Lecture Notes in Computer Science, Vol. 3649. Springer-Verlag, Berlin Heidelberg, 268-284. (2005)
16. Mendling, J., Simon, C.: Business Process Design by View Integration. Lecture Notes in Computer Science, Vol. 4103. Springer-Verlag, Berlin Heidelberg, 55-64. (2006)
17. Microsoft Patterns and Practices: Microsoft Application Architecture Guide, 2nd Edition, Microsoft Press, Redmond, USA. (2009)
18. Milašinović, B., Fertalj, K., Nižetić, I.: On some Problems while Writing an Engine for Flow Control in Workflow Management Software. In Proceedings of the 29th International Conference on Information Technology Interfaces, Cavtat, Croatia, 489-494. (2007)
19. Milašinović, B., Fertalj, K.: Using partially defined workflows for course modelling in a learning management system. ICIT-2009, The 4th International Conference on Information Technology, Amman, Jordan. (2009).
20. Nishizawa, K.: A method to Find Elements of Cycles in an Incomplete Directed Graph and Its Applications – Binary AHP and Petri Nets. Computers & Mathematics with Applications, Vol. 33, No. 9, 33-46. (1997)
21. Polyvyanyy, A., García-Bañuelos, L., Dumas, M.: Structuring Acyclic Process Models. Lecture Notes in Computer Science, Vol. 6336, Springer-Verlag, Berlin Heidelberg 276-293. (2010)
22. Sadiq, S., Sadiq, W., Orlowska, M.: Workflow Driven e-Learning – Beyond Collaborative Environments. International NAISO Congress on Networked Learning in a Global Environment, Challenges and Solutions for Virtual Education. Berlin, Germany. (2002)
23. Windows Workflow Foundation. [Online]. Available: http://msdn.microsoft.com/en-us/netframework/aa663328.aspx (October 2010)
24. Workflow Management Coalition. Terminology & Glossary: Document Number WFMC-TC-1011 - Document Status - Issue 3.0 [Online]. Available: http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html (October 2010)
25. Xi, S., Yong, J.: New Data Integration Workflow Design for e-Learning. Lecture Notes in Computer Science, Vol. 4402, Springer-Verlag, Berlin Heidelberg 699-707. (2007)
26. Ye, Y., Roy, K.: A graph-based synthesis algorithm for AND/XOR networks. In Proceedings of the 34th Annual Design Automation Conference, Anaheim, California, USA, 107-112. (1997)

27. Yong, J.: Workflow-based e-learning platform. In Proceedings of the 9th International Conference on Computer Supported Cooperative Work in Design, Coventry, UK, Vol. 2, 1002-1007. (2005)

Krešimir Fertalj is a full professor at the Department of Applied Computing at the Faculty of Electrical Engineering and Computing, University of Zagreb. Currently he lectures a couple of computing courses on undergraduate, graduate and doctoral studies. His professional and scientific interest is in computer-aided software engineering, complex information systems and in project management. He participated in a number of information system designs, implementations and evaluations. Fertalj is member of ACM, IEEE, PMI, and Croatian Academy of Engineering.

Boris Milašinović is a research assistant at the Department of Applied Computing at the Faculty of Electrical Engineering and Computing, University of Zagreb. He graduated in 2001 at the Department of Mathematics at Faculty of Science, University of Zagreb. He received MSc degree in 2006 and Ph. D. degree in 2010 in Computing at the Faculty of Electrical Engineering and Computing. His main research interests include software development and workflow management. Currently he lectures a couple of undergraduate courses in Computing and he is teaching assistant to one graduate course in Computing.

Received: November 20, 2010; Accepted: August 5, 2011