Catalog-based Token Transportation in Acyclic Block-Structured WF-nets

Ahana Pradhan and Rushikesh K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Powai, Mumbai-400076, India

Abstract. The problem of workflow instance migration occurs during dynamic evolutionary changes in processes. The paper presents a *catalog-based* algorithm called the *Yo-Yo Algorithm* for consistent instance migration in Petri net workflow models. It uses a technique of folding and unfolding of nets. The algorithm is formulated in terms of *Colored Derivation Trees*, a novel representation of the runtime states of workflow nets. The approach solves the problem for certain types of changes on acyclic block-structured workflow nets built in terms of primitive patterns moving much of the computation to schema level on account of the use of two critical ideas, a catalog and the folding order. The approach is illustrated with the help of examples and comments on its correctness.

Keywords: Block structured Workflows, Dynamic Evolution, Structural Compatibility, Token Transportation, Workflow Specification

1 Introduction

Organizational goals are realized by executing business processes that involve people, resources, schedules and technology. In order to cope up with changing environments, changing requirements or new internal challenges, business processes need to be changed. Traditional workflow management systems (WFMS) are well-suited for rigid processes. However, the volatile nature of business processes requires intricate facilities for changing the workflows at runtime in WFMS in a valid and consistent manner. In absence of this support the information system susceptible to changes needs to be tackled by slower porting processes, it not being immediately usable due to the not easily bridgeable gap between *the pre-planned* and *the evolved actuality*.

An evolutionary change includes process change at schema level and also instance migration for all running cases. This paper describes the *Yo-Yo* algorithm for Petri net models of acyclic block-structured workflows to carry out consistent runtime instance migration in this context. At the schema level a Yo-Yo compatibility property is specified to define the scope of the proposed instance migration algorithm. A specialty of the algorithm is that it gives the consistent token transportation based on pre-computed catalog solutions. Secondly, from the two workflow net schemas, we are able to separate immediately migratable and immediately not migratable markings. This work uses the Petri

net based workflow model called WF-net, which was introduced by Van der Aalst [1]. We follow a block-structured formulation of WF-nets, in terms of blocks, which are primitive workflow patterns namely the Sequence, the Parallel Block and the Exclusive-choice Block. The algorithm is presented for carrying out runtime token transportation under a set of change operations, which are the inter-convertibilities among the primitive pattern blocks. The intuition of this algorithm is presented in our earlier work [2]. The paper provides the full formulation of the algorithm and its proof of correctness. The formulation is developed in terms of a new runtime workflow state representation called the Colored Derivation Tree.

The paper is organized as follows. After discussing the related work, we first present our pattern based block structured workflow specification approach in Section 3. Following this, the novel representation called *Derivation Tree* and its colored form are developed in Section 4. In Section 5, the intuition behind the algorithm is first outlined. The ingredients of the algorithm are then discussed developing the notions of the *Yo-Yo compatibility* property between two nets, instance level correctness of migration in the form of a valid and consistent catalog of token transportation and folding, unfolding operations on WF-net. Lastly, the algorithm, its working and its applicability are explained with the help of a practical example scenario. A proof of correctness of Yo-Yo algorithm is given in Section 6. The algorithm works on colored derivation tree representing the old net and produces colors in the derivation tree of the new net, i.e. colors corresponding to the marking in the new net. The catalog is used for color transfer at each iterative step in the algorithm.

2 Related Work & Contributions of the Paper

In this section, we present a brief account of existing research on dynamic evolutionary changes in Petri net models of workflows to highlight the achievements so far and respective limitations. The literature in this field can be categorized into two kinds. Firstly, the *change region* based approaches are designed to work with arbitrary structural changes. The second category is of *state based* approaches.

2.1 Change Region Based Approaches

An earlier among the change region based approaches is the approach of Ellis et al. [3], representing dynamic change as replacement of a part of the old net and preserving the same history of execution by an altered making after the replacement. In their approach called *token transfert*, the old part referred to as the *old change region* is replaced by a new change region. However, some cases are unsafe to migrate when the old change region is marked, and hence, the transfers are delayed until the tokens in the old net reach a safe state. This delayed execution of changeover requires migration to be withheld till a consistent point of execution is reached when tokens come out of the change region. This early work in the field does not suggest a method for identification of unsafe

change regions or an algorithm for consistent migration. Ellis and Keddara [4] demonstrate realization of *token transfert* with the help of transitions called *flow jumpers* connecting places in the old and the new nets. However, no algorithm to compute the flow jumpers has been suggested in this work.

Aalst [5] presents an algorithm for computing change regions in old and new nets, the notion of which was introduced earlier [3]. Outside the change regions, the marking in the old net can be carried forward into the new net one on one without violating validity. However, the adopted consistency criterion uses a notion of validity based on marking equality in terms of place labels, ignoring the notion of consistency in terms of history equivalence. Therefore, for several types of changes, the computed change region does not ensure consistent migration.

Sun and Jiang [6] present a variation of the algorithm given by Aalst [5] for generating the change region. Their work handles dynamic changes for *upward compatibility*, where the behavior of the old net in terms of execution traces is preserved in the new net. For consistent instance migration, a weaker version of history equivalence criteria is specified. Unlike usual notion of state in Petri net formalism, in addition to marking, this work represents the runtime state by considering execution trace. The work also formulates a property for migratability at the level of instance, including those inside change regions. However, this approach does not provide an algorithm for consistent instance migration.

The work of Cicirelli et al. [7] describes an implementation and a case study of dynamic evolution based on the theory founded in the works of Ellis et al. [3] and Van der Aalst [5]. Their work uses the change region generation algorithm given by Van der Aalst to compute the unsafe regions for instance migration. The migration strategy is termed as *decentralized migration*, since the executions in different parallel branches are independently inspected and set for migration. The tokens in the old instance are then tagged according to their presence inside or outside the change region. In a particular state set for migration, some tokens may be inside the change region, whereas some are outside. Tokens outside change region are migrated immediately. Tokens inside change region continue till they come out of it and enter in a safe state suitable for migration. That point of execution creates a valid marking in the new schema.

2.2 State Based Approaches

The difference between this category and the earlier one is that the state based approaches do not pre-compute the change regions. Instead, they directly provide state based mappings in the new net. If consistent mapping does not exist in a particular state, this approach can not make use of any possibility of *delayed migration* as in change region based approaches.

It has been noted [6] that the change region based approach is a pessimistic approach since inside a change region, there may be migratable markings. In the state based approach, this drawback is removed with the additional cost of instance based solutions. Another shortfall of the existing change region computation algorithms is that they overlook the history equivalence criteria. For example, a change region computation focuses on finding a mapping for state

(p_1, p_2, p_3) to state (p'_1, p'_2, p'_3) ignoring one-to-many or many-to-one mappings of other kinds such as mappings to states (p'_1, q), (p'_1, p'_2) or (p'_1, p'_2, p'_3, p'_4) in this case which have the same history of transition firings. The class of state based approaches solve this problem by keeping the observable behavior of the nets in focus, thereby exploring richer markings which need not be identical.

The approach of Agostini and Michelis [8] implement the feature of dynamic change in their MILANO workflow system. This work allows a set of change operations, which are *parallelization*, *sequentialization* and *swapping*. The mappings of runtime states between the old and the new workflows are precomputed over the entire state space modeled as *reachability graph*. Instead of identifying regions, state to state mappings are generated for valid migration points.

Van der Aalst and Basten [9] have looked into the problem of dynamic change in light of *inheritance* relations between the nets in a migration pair. If the new workflow *specializes* the *observable behavior* of the old workflow by *hiding* or *blocking* some of the additional tasks, then the new workflow is considered as a subclass of the old one. They show that if two nets are related by inheritance, it is always possible to have a correct instance migration from the old to the new workflow. The work also shows that addition or deletion of cycle, sequence, parallel or choice branches preserve inheritance relation. The mapping between the runtime states of the two processes is given by transfer rules guaranteeing *soundness*. However, the problem of consistency in terms of history is not formally addressed, though the authors point out a supporting example.

2.3 Contributions of Our Work

The paper presents an algorithm for token transportation to ensure the consistency criterion of history equivalence by applying catalog solutions without replaying the history, unlike most of the existing approaches. For a practical scenario of evolution, where thousands of instances need to be migrated, replaying history for each of them or solving the *state equation* [10] along with the solution for legal firing sequence problem [11] may be computation intensive. The Yo-Yo algorithm improves the runtime by pre-computing *migrations among primitive patterns*, and by generating what is called *Yo-Yo compatible derivation trees* at the schema level. Moreover, for the chosen types of nets and change patterns, the Yo-Yo algorithm successfully carries out consistent migration even for those cases many of which are not suitable for migration as per the change region based approaches due to their pessimistic prediction of non-migratability. Yo-Yo approach does not compute change region, but in turn, it looks for catalog based transportation which succeeds if the case is migratable by the consistency criteria of history equivalence.

3 The Pattern Based Approach of Workflow Modeling

The Yo-Yo token transportation approach offloads some of the complexities incurred by traditional change region based or state based approaches by means of

its block structured workflow specification. Confining the scope to block structured workflows is in line with the philosophy of block structured executable process models in BPEL [12] and pattern based models such as [13].

The permissible change operations on workflow schemas have been referred to as *change patterns* in the literature [14]. The change patterns in the Yo-Yo approach are *inter-convertibilities among the primitive patterns* shown in Fig. 1. The following six kinds of pattern changes are considered: SEQ to AND, AND to SEQ, SEQ to XOR, XOR to SEQ, AND to XOR, and XOR to AND.

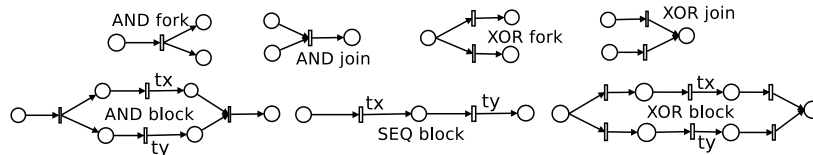


Fig. 1. Primitive Gateways and Patterns

3.1 Workflow Primitives

Patterns are commonly occurring configurations in architecture. Control flow behavior of patterns in workflow processes were described by Aalst et al. [15]. In our work, we formulate and use a grammar for workflow nets in terms of the primitive workflow patterns. Fig. 1 shows the Petri-net models of primitive fork-join gateways and pattern blocks. It can be noted that the transitions in the gateways are kept unlabeled, since these are used only to model the control-flows and not the workflow tasks. Consequently they are omitted from the specifications in the string based language of WF-nets which is introduced below. The string based language captures the control flow dependencies through delimiters.

3.2 CWS: A Compact Block Structured Workflow Specification

Block-structured workflows are composed by nesting the primitive patterns. This approach simplifies complex processes in terms of blocks. The block structures are directly folded in or out in the Yo-Yo algorithm. For the purpose of our work we assume that there is no repetition of transition-labels in a net.

Now, a compact string-based specification language called CWS is introduced for specifying block structured acyclic WF-nets. Unlike graphical and tuple based existing description methods for Petri net based workflows, in CWS, the places are dropped and only the labeled transitions are included. The reason for excluding the places is that the consistency criterion based on task execution traces does not require any role from the places. However, places shown in the pictorial models can be regenerated by parsing CWS specifications. The execution control transitions used in fork-join patterns are implicitly encoded into the delimiters, and only the application transitions are included in the specification.

$Start \rightarrow SEQ$;
$SEQ \rightarrow SEQ$ **t** SEQ **t** $SEQ \mid SEQ\ AND\ SEQ \mid SEQ\ XOR\ SEQ \mid \epsilon$;
$AND \rightarrow$ **(** SEQ **t** SEQ **)** **(** SEQ **t** SEQ **)** ;
$XOR \rightarrow$ **[** SEQ **t** SEQ **]** **[** SEQ **t** SEQ **]** ;

Workflow Net	**CWS Specification**
SEQ block in Fig. 1	$t_x t_y$
AND block in Fig. 1	$(t_x)(t_y)$, $(t_y)(t_x)$
XOR block in Fig. 1	$[t_x][t_y]$, $[t_y][t_x]$
The net in Fig. 8(a)	$t_1(t_2(t_3)(t_4))((t_5)(t_6)t_7)t_8$
The net in Fig. 8(b)	$t_1t_2t_3t_4(t_5)(t_6)t_7t_8$

Fig. 2. CWS Grammar, Example Nets, and their Specifications

The CWS grammar is shown in Fig. 2. A terminal symbol **t** represents a transition corresponding to a task in the workflow. Round and square bracket pairs are used to mark AND and XOR fork-join patterns respectively. The top level pattern is always a Sequence that can generate either an empty string or a nesting of blocks. Example nets specified in CWS are given in Fig. 2. It can be seen that parallel or choice branches can be specified in any order, which creates multiple equivalent specifications.

4 Derivation Tree of a Workflow Net

Parsing of workflow models into hierarchical blocks has been implemented earlier in the approach of Refined Process Structure Tree (RPST) [16]. It provides unique parsing of a WF-graph in terms of canonical single-entry-single-exit regions which can be of arbitrary length. However, this approach results in an infinite-sized catalog, which counters the advantage of our approach.

For Yo-Yo algorithm, the nets are required to be parsed in a hierarchy of fixed-size ingredient blocks. This parsing obtains the Derivation Tree representation of a WF-net. The derivation tree is obtained after cleaning up the delimiters from the CWS parse tree. For the primitive nets shown in Fig. 1, their respective parse trees and derivation trees are shown in Fig. 3. Table 1 shows the correspondence between symbols in the derivation tree and WF-net.

The terminals in the parse tree are application transitions, delimiters and empty sequences; and the non-terminals represent block-structured configurations in the net. The derivation tree excludes delimiters and empty sequences resulting in *leaf non-terminals* (e.g. n_i and n_x''' in Fig. 3f).

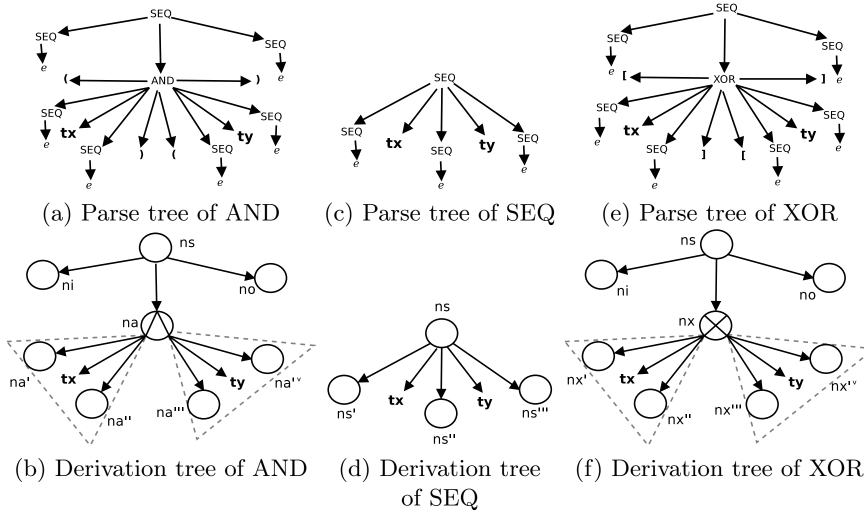
The child nodes of AND and XOR non-terminals are organized into two *triplets* representing the two fork-join branches. Each triplet contains two places and a transition. The arcs in a triplet are ordered left to right, showing a transition sandwiched between pre- and post-places respectively.

A derivation tree can be viewed as a hierarchical composition of the *derivation tree patterns*, which are the derivation trees of the primitive patterns (Figs. 3b,d,f). An Example of derivation tree patterns in a bigger non-primitive net appears in Fig. 5, where the derivation tree patterns are marked as dotted ellipses.

In a derivation tree pattern, the arcs coming out of a SEQ node are ordered from left to right. A SEQ node representing only a sequence of two transitions has five arcs, two for the transitions and three for the places. A SEQ node representing the grammatical reduction of an AND or an XOR branching has three arcs to connect to the branching and the places before and after it.

Table 1. Mapping of Derivation Tree Symbols

Derivation Tree Element/CWS Element	Symbol Used	Correspondence with WF-net
Leaf non-terminal/empty SEQ	Unfilled circle	Unfolded Place
Non-leaf non-terminal/SEQ	Unfilled circle	Abstraction (Folded place)
Non-leaf non-terminal/AND	Circle marked $\wedge$	Two parallel branches
Non-leaf non-terminal/XOR	Circle marked $\times$	Two exclusive-choice branches
Terminal/**t**	Symbol t_{label}	Labeled transition t_{label}



(a) Parse tree of AND (c) Parse tree of SEQ (e) Parse tree of XOR

(b) Derivation tree of AND (d) Derivation tree of SEQ (f) Derivation tree of XOR

Fig. 3. CWS Parse Trees and Derivation Trees of Primitive Patterns

4.1 Yield of a Non-terminal

Yield of a non-terminal is a *sequence* obtained by depth-first traversal on the terminals in the entire subtree rooted at the non-terminal of a derivation tree, where elements of the sequence are terminals or sets of terminals which can be further nested. During the traversal, swapping the traversal order of triplets under an AND or XOR node does not alter the yield. Hence, yield of an AND or XOR node is formulated as a set of yields of the two triplets. Operator $yield(n)$ generates this traversal for a non-terminal n. For example, $yield(n_s) = \{t_x, t_y\}$ in Fig. 3b, $yield(n_s) = t_x t_y$ and $yield(n'_s) = \epsilon$ in Fig. 3d, yield of the root in Fig.

5a is a sequence with one element $\{t_1t_2t_3, t_4t_5t_6\}$, and the root of the derivation tree for the net given in Fig. 8a has yield $t_1\{t_2\{t_3, t_4\}, \{t_5, t_6\}t_7\}t_8$.

4.2 Local Terminal Coverage of a Pattern

To recall, a pattern is any of the tree structures shown in Figs. 3(b,d,f). The notion of *local terminal coverage* (LTC) defined on patterns establishes pattern to pattern correspondence called *peers* in two nets. LTC of a pattern p, i.e. $LTC(p)$, is a cross-product $s \times o$ of set s of terminals in the pattern and boolean value o indicating whether the set is ordered (i.e. SEQ block). The individual elements can be accessed through a dot operator as $p.s$ and $p.o$.

Peer Patterns: Two LTCs can be compared by comparing their terminal sets and the ordering. The comparison operator (equality) called *peer* is defined as follows. Let $=_s$ be the set equality operator and $=_o$ be the ordered set equality operator. We can define the comparison operator $peer(p, q)$ for two patterns p, q in terms of their respective LTCs, as an operator returning a boolean value: $peer(p, q) = ((p.o \wedge q.o) \wedge (p.s =_o q.s)) \vee (\neg(p.o \wedge q.o) \wedge (p.s =_s q.s))$. If both sets are ordered (first part of the disjunction), then the comparison operator checks for element ordering. This condition defines peer relation between sequences. For example, Sequence t_xt_y and t_yt_x are not peers, their LTCs being $(\{t_x, t_y\}, 1)$ and $(\{t_y, t_x\}, 1)$. If one of the sets is unordered (second part of the disjunction), the comparison operator checks for set equality not considering the ordering of elements. This condition defines peer relation between two patterns when one of them is not a sequence. For example, Sequence t_xt_y and AND $(t_y)(t_x)$ are peers, their LTCs being $(\{t_x, t_y\}, 1)$ and $(\{t_y, t_x\}, 0)$.

The *peer* operator is thus used to identify pattern to pattern correspondence between two nets, which contributes to the formulation of hand-in-hand folding and unfolding of the nets. For formulation of Yo-Yo compatible derivation trees of a given pair of nets, identification of peer pattern pairs is the very first requirement. The Yo-Yo algorithm carries out token transportation by transferring colors between the peer patterns. Examples of peer patterns can be seen in Fig. 4, where any two derivation trees satisfy the peer relation.

4.3 Colored Derivation Trees

Coloring of a derivation tree represents a *marking* of the corresponding net. Non-terminals can be colored following Definitions 1 and 2. Fig. 4 shows examples of colorings of derivation trees and the corresponding net markings .

Definition 1 Black Non-terminal*: (i) A leaf non-terminal corresponds to a marked place in the net, and (ii) a non-leaf non-terminal abstracts a marked subnet in which no labeled-transition has been fired yet.*

Definition 2 Red Non-terminal*: It is a non-leaf non-terminal that abstracts a marked subnet where at least one labeled-transition is fired.*

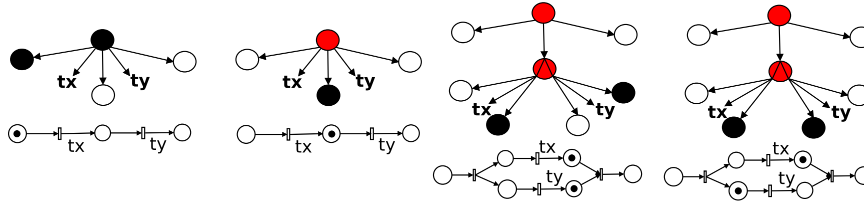


Fig. 4. Examples of Derivation Tree Coloring

5 The Yo-Yo Algorithm for Token Transportation

This section first develops an intuition to the Yo-Yo algorithm, and then discusses the ingredients of the algorithm, which are the consistency and validity notions, token transportation catalog, pattern hierarchy and folding order. The algorithm is discussed at the end of this section.

As discussed previously, a net is considered as a composition of primitive patterns, and the derivation tree of the net is a hierarchy of derivation trees of the component patterns. Due to the hierarchical structuring of patterns, the patterns in the upper level have places that are abstractions of patterns in the lower level. In other words, a pattern is *folded into a place* of a pattern which is at a higher level in the hierarchy. The Yo-Yo algorithm transports tokens from old net to new net by transporting tokens between peer patterns of two derivation trees starting from the top level. The tokens move into their places in the new net as they trickle down when the *folded places unfold*. Resemblance between the stretching and squeezing of the string of the Yo-Yo toy, and the nets being folded and unfolded caused the nomenclature of the algorithm. A *token transportation catalog* is constructed for the purpose of peer to peer token transportation. Transportation in a larger net is thus carried out by applying the cataloged solutions repetitively through the process of folding and unfolding of the patterns organized in the hierarchy. Given two pattern hierarchies the *Folding Order* is formulated, using which both of the nets are folded hand-in-hand.

5.1 Yo-Yo Compatibility at Schema Level

Yo-Yo compatibility at schema level is a structural property, which is necessary for instance migration by the Yo-Yo algorithm. It ensures that the old and the new nets can be folded and unfolded hand-in-hand. During a workflow life-cycle, at the time of building the new net from the old net, if the changes are confined to only the allowed pattern alterations, the schema compatibility can be achieved.

A pattern in a derivation tree can be from any of AND, XOR and SEQ blocks. A pattern occurring at any level in the derivation tree can be replaced by another pattern without changing the tasks involved in the pattern. Consequently, the tree of the old net is modified by replacing a derivation tree pattern by another. Two such replacements can be observed in the tree pair shown in Fig. 7. Syntactically, when a Sequence is changed into an AND and XOR, triplets are formed by including additional nodes according to the grammar. The reverse

happens in the case of a change from AND or XOR to Sequence. These syntactic alterations among the primitive patterns can be observed in Figs. 3b,d,f.

The operator $compatible(n_1, n_2)$ defines the Yo-Yo schema compatibility between two nets whose derivation trees are rooted at nodes n_1 and n_2 respectively: $compatible(n_1, n_2) = (yield(n_1) =_y yield(n_2))$, where the yield equality operator $=_y$ compares two yields considering that swaps of triplets in a fork-join pattern are permissible. An example pair of compatible yields is $t_1\{t_2\{t_3, t_4\}, \{t_5, t_6\}t_7\}t_8$ and $t_1t_2t_3t_4\{t_5, t_6\}t_7t_8$ for the trees shown in Fig 8a,b.

5.2 Correctness of Token Transportation

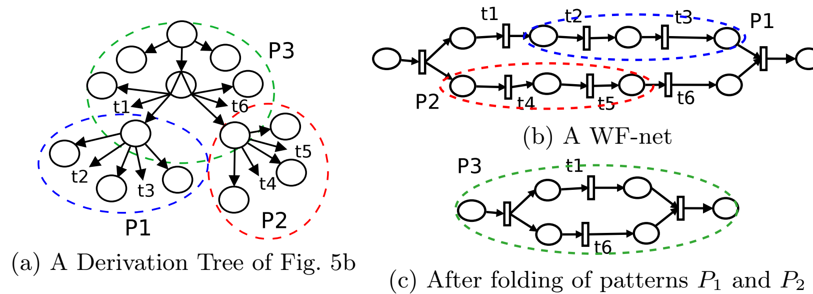
In order to ensure the correctness of the applied dynamic change, validity and consistency of the resultant marking in the new net must be ensured. For Yo-Yo migratability, the following models of consistency and validity are adopted.

Axiom 1 Consistency*: The tasks which are already completed in the old net are also completed in the new net, and vice-versa.*

Axiom 2 Validity*: Resulting marking in the new net is reachable from its initial marking.*

5.3 Pattern Hierarchy and Folding Order

The process of abstracting a single primitive pattern into a place in a net is called *folding*. The reverse, i.e. expansion of a folded place into a pattern is referred to as *unfolding*. Folding operation simplifies the structure of a net consisting of multiple patterns converging into a single pattern at the top level. Folding operation can be applied multiple times, each one simplifying the net further until the whole net is folded into a single pattern at the top. The original net can be obtained by the reverse process of unfolding abstract places into patterns.



(a) A Derivation Tree of Fig. 5b

(b) A WF-net

(c) After folding of patterns P_1 and P_2

Fig. 5. Derivation Tree and Folding Operation

Pattern hierarchy of a derivation tree is a partial order capturing the nesting hierarchy of derivation tree patterns. In the corresponding net, it gives the

hierarchy of folding of primitive patterns into places. For example, the derivation tree shown in Fig. 5a for the net in Fig. 5b shows the patterns in dotted ellipses for which the pattern hierarchy is given by bottom-up partial order $(\{P_3 \leftarrow P_2, P_3 \leftarrow P_1\})$. Each folding expression $P \leftarrow C$ in the tree gives a pair of parent-child patterns, where child C is folded into a place in parent P. Expressions in round brackets represent sequences, and sets represent no ordering constraint among its folding expression elements.

Given two pattern hierarchies, a *folding order* is formulated for applying the Yo-Yo algorithm which is the order to *fold the peer patterns hand-in-hand.* All parent-child relations among the patterns for each of the two trees are covered (directly or transitively) in the folding order. The top-down folding order expression for the two bottom-up pattern hierarchy expressions $(\{P_3 \leftarrow P_2, P_3 \leftarrow P_1\})$ and $(P'_1 \leftarrow P'_2, P'_3 \leftarrow P'_1)$ is a sequence $(< P_3\text{-}P'_3 >, < P_1\text{-}P'_1 >, < P_2\text{-}P'_2 >)$ consisting of folding expressions $< P_i\text{-}P'_i >$, $i \in \{1, 2, 3\}$, where P_i, P'_i are peer patterns. An expression in angular brackets is pair of peer patterns.

5.4 Enumeration of the Token Transportation Catalog

The catalog handles token transportation between two *different* patterns. Transfer between the same patterns are handled by the algorithm through a simpler generic step. Consistent migrations between valid markings of different peer patterns create the cases of the token transportation catalog given in Fig. 6. The counts of valid markings for the three patterns SEQ, AND and XOR blocks are 3, 6 and 6 respectively. Each marking further generates variants based on (1) whether the influential places are folded and (2) if a marked place is folded, whether a token in it represents none, partial or full completion of the subnet abstracted in it. Some of the resultant markings that are not migratable due to the consistency criterion are omitted from the catalog. The catalog shown in Fig. 6 contains 37 entries all in all, and the transportation mappings among them. The entries are enlisted as colored derivation trees. A bidirectional arrow between migratable colorings of different patterns means that if one is the old pattern coloring the other can be the new coloring. For example, consider the mapping between case 28 and 26. Case 28 is a sequence, where the token is after t_x and before t_y. Case 26 is an AND pattern which has consistent mapping from case 28. It can be seen that there are two tokens in the two parallel branches of case 26, one after t_x and another before t_y. Thus, both the cases have completed task t_x and hence they are defined to be consistent with each other.

In some cases, a SEQ coloring can be mapped to more than one AND or XOR colorings. These ties are broken based on the conditions on non-empty yields noted in Table 2. A tag symbol is associated with each condition for use in Fig. 6. Also, if a node in the catalog is identified as a *leaf* or *non-leaf* or as a *just completed* folded place (i.e. holding a token just before the exit), the constraint has to be matched. In the table, node o is in the old SEQ pattern tree, and n_1, n_2 are in the new fork-join pattern tree. When o is the leftmost child, n_1 is the leftmost child, and n_2 is the node to the left of t_x. When o is the middle child, n_1 is the node to the right of t_x, and n_2 is the node to the left of

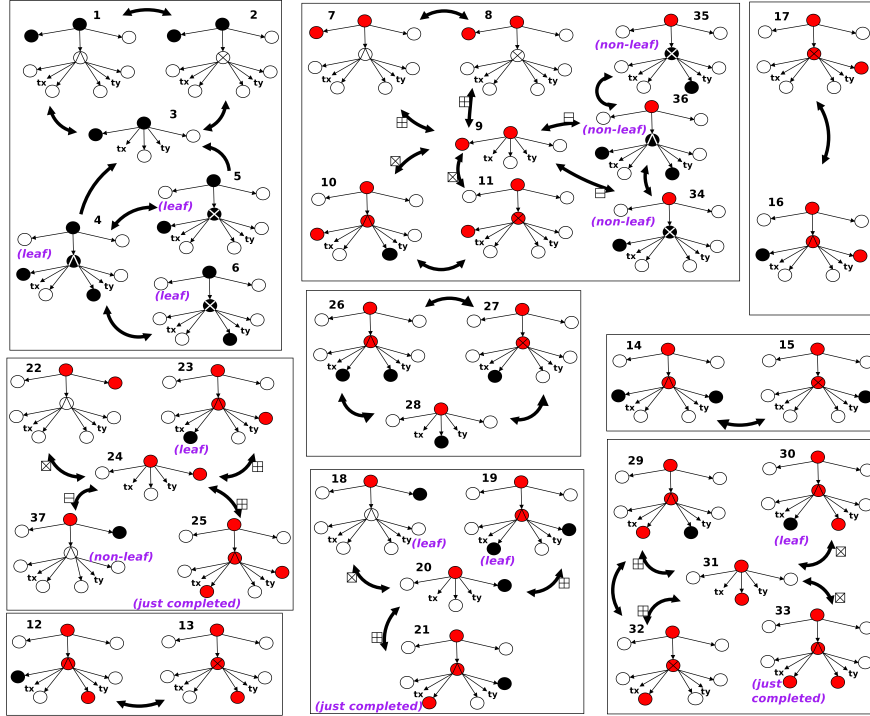


Fig. 6. Consistent and Valid Token Transportation Catalog

t_y. When o is the rightmost child, n_1 is the node to the right of t_y, and n_2 is the rightmost child. Node n is in the subtree rooted at o s.t. $yield(n) =_y yield(n_2)$. It can be noted that, since swapping of two tasks in a sequence is not included in the present catalog, the sequence t_x, t_y never becomes sequence t_y, t_x.

Table 2. Tags and Conditions for Catalog Cases

Tag	Condition to be evaluated
⊞	$yield(o) =_y yield(n_1)$ or $yield(o) =_y yield(n_1).yield(n_2) \wedge$uncolored n
⊠	$yield(o) =_y yield(n_2)$, or, $yield(o) =_y yield(n_1).yield(n_2) \wedge$ Red n
⊟	$yield(o) =_y yield(n_1).yield(n_2) \wedge$ Black n

5.5 Token Transportation Algorithm

The Yo-Yo algorithm formulates a consistent marking in the new net given the marked old net and a Yo-Yo compatible derivation tree pair of the two nets. The folding order for the derivation tree pair and the token transportation catalog are required for the computation. The algorithm is given in Algorithm 1.

At first, the marking of the old net is translated into a coloring in the derivation tree of the net. Then, the algorithm colors the new derivation tree pattern

by pattern in top-down fashion given by the *folding order*. When all the color ripples reach the leaves, the algorithm successfully terminates.

Algorithm 1: Yo-Yo Algorithm

```
Input: Old Marked Net N, Unmarked New Net N′, Uncolored Old Derivation
       Tree D, Uncolored New Derivation Tree D′, Folding Order F, Token
       Transportation Catalog
Result: Marking in N′
1 colorTree(D,N)
2 Let < p-q > be the first folding expression 'fetched' from F, where p and q are
  peer patterns
3 if modularTransport(p,q) ≠ true then return false
4 for every folding expression < p-q > 'fetched' from the remainder of F, not
  violating the partial order specified in F, where q has colored root do
5     if p is colored then
6         if modularTransport(p,q) ≠ true then return false
7     else localPropagation(q)
8 Mark the places in N′ corresponding to Black leaves in D′
9 return true
```

Procedure colorTree(Uncolored Derivation Tree D, Marked Net N)

```
Result: Coloring in D
1 for each leaf non-terminal n in D corresponding to a marked place in N do
2     color n Black
3 S ← set of colored nodes in D having uncolored parent
4 while S is not φ do
5     n ← any element from S
6     p ← colorParent(n)
7     S ← S \ {n}
8     if p is not NULL then S ← S ∪ {p}
```

Fig. 7 shows the color propagation traces in the old and then in the new trees. Old Tree: Steps 1-4 depict the bottom-up coloring of the old tree performed by procedure `colorTree`. It starts by coloring the leaf nodes black corresponding to the marked places in the net of Fig. 8a. Then for each colored node, its parent is colored either red or black by procedure `colorTree` until the root is colored.

After transferring color between the top peers, the algorithm goes through the peer patterns $< p_i, q_i >$ from the folding order confronting the following cases: (i) root of q_i is uncolored, (ii) p_i is colored, root of q_i is colored, and (iii) p_i is uncolored, root of q_i is colored. In case (i) there is no color transfer. In case (ii), procedure `modularTransport` colors q_i. When p_i and q_i are the same patterns, after replicating the color of p_i to q_i, a red color transferred to a leaf is turned

black to preserve the validity of coloring. If p_i and q_i are different, cataloged transportations are applied. In case (iii), proc. `localPropagation` colors q_i.

Procedure localPropagation(Uncolored Pattern q)

Result: Coloring in q

1 $r \leftarrow$ root node of q
2 **if** r *is Red* **then**
3 $\quad n_r \leftarrow$ rightmost child of r
4 $\quad$ **if** n_r *is leaf* **then** color n_r Black **else** color n_r Red
5 **else** color the leftmost child of r Black

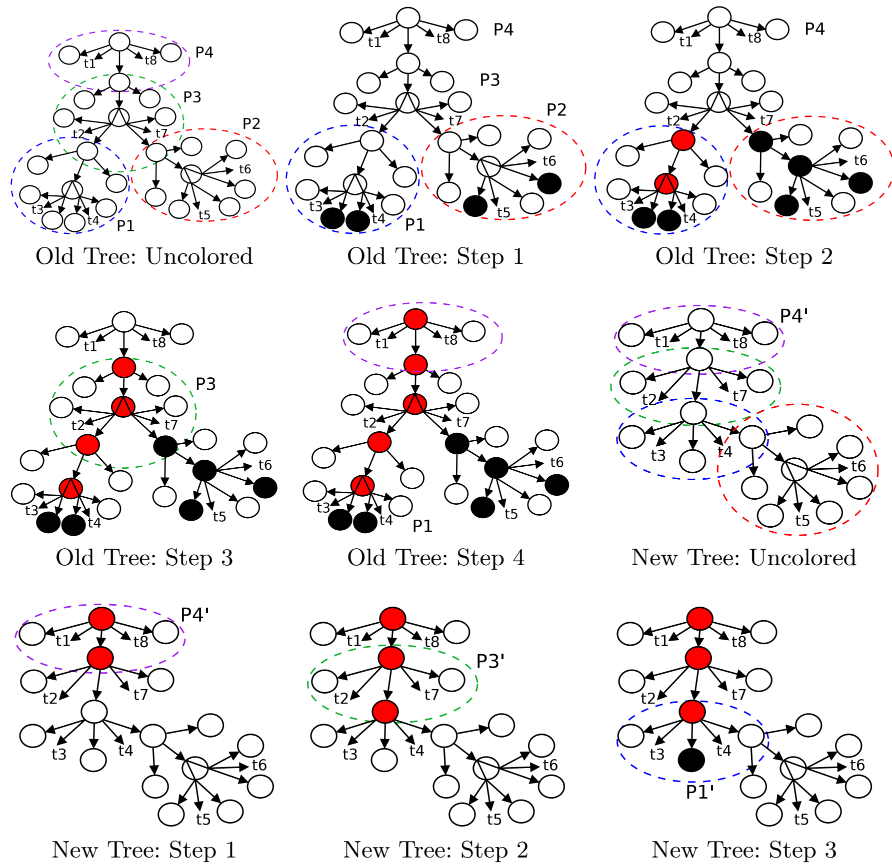


Fig. 7. Derivation Trees Traces in Yo-Yo Transportation

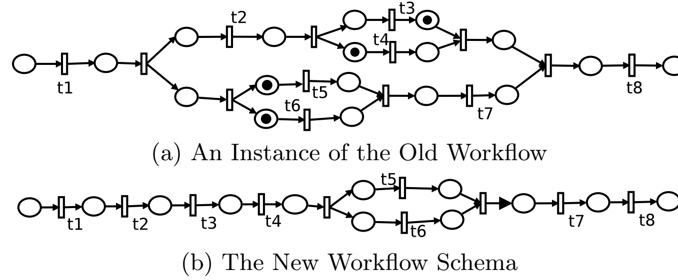
```
Function modularTransport(Colored Pattern p, Uncolored Pattern q)
   Data: Token Transportation Catalog
   Result: Coloring in q
 1 if p and q are same patterns then
 2 |  color q as p // same color transfer
 3 |  change the Red leaves of q to Black // no leaf is left Red
 4 |  return true
 5 search catalog for colored p and its mapping to q
 6 if no mapping found then
 7 |  print instance not consistently migratable, return false
 8 color q as per the search result, return true
```

```
Function colorParent(non-terminal n)
 1 if n is root node in D then return NULL
 2 p ← parent node of n
 3 if n is Red then color p Red, return p
 4 if n is of type SEQ then
 5     if p is of type SEQ then
 6         if n is leftmost child of p then color p Black else color p Red
 7     else if p is of type AND then
 8         if n is left child in any triplet from p and left child in the
 9         other triplet is Black then color p Black else color p Red
10     else
11         if n is left child in any triplet from p then color p Black
12         else color p Red
13 else
14     if non-terminal left to n is leaf then color p Black else color p Red
15 return p
```

5.6 An Example Application Scenario

A realistic scenario of dynamic evolution in the reimbursement process in an academic institute is now illustrated where the Yo-Yo algorithm is used for token transportation. The old process schema is modeled by the net depicted in Fig. 8(a). The actual tasks corresponding to each labeled-transition are given in Table 3. As per this design, a student has to first fill the reimbursement form and submit it to initiate a reimbursement request. Next, two concurrent subprocesses begin, one of which is submission of the bills and then approval by guide and head of the department. In parallel, the verification of the funding history for the applicant and funding availability is performed by the awards' committee, after a favorable result is confirmed by the committee approval. The reimbursement amount granted by these three approvals are lastly credited to the student's scholarship account thereby completing the workflow.

This design is evolved into the new schema, depicted in Fig. 8(b) due to the following reasons: every time an application is approved by the head of the



(a) An Instance of the Old Workflow

(b) The New Workflow Schema

Fig. 8. Reimbursement Workflow

Table 3. Tasks in the reimbursement process in an academic institute

Task Label	Actual Task	Task Label	Actual Task
t_1	fill form & submit	t_5	Funding history verification
t_2	submit documents	t_6	Funding availability verification
t_3	Guide's approval	t_7	Awards' committee's approval
t_4	HOD's approval	t_8	credit transaction

department only after its approval by the student's guide. In the new design, this dependency is reflected explicitly to prevent an applicant from making approval request to the HOD prior to his/her guide. Also, for some cases, though the funding background is verified by the awards' committee, reimbursement is not granted due to rejection either from respective guide or the head. Therefore, to alleviate the unusable funding verification by the awards' committee, the designed concurrency is now made sequential by moving the funding related activities in the later part of the process.

Dynamic migration of the reimbursement applications already in progress relieves the applicants from having to start fresh. Also, the process is too simple to maintain different versions. Therefore, consistent dynamic instance migration in response to the evolutionary changes are desired. The Yo-Yo algorithm carries out the consistent token transportation as shown in Fig. 8.

The visualization of the transportation in the given net pair is depicted in Fig. 9. The bottom-up coloring of the old derivation tree is equivalent to successive folding operations of the marked old net. Again, pattern by pattern top-down coloring of the new derivation tree is equivalent to unfolding a folded pattern in the new net and marking it each time. Movement of the color ripple into a leaf node is equivalent to reaching of a token into an actual place. In this case, the algorithm terminates when all the transported tokens are placed. Fig. 9 shows the nets being squeezed and released as they undergo token transportation.

6 Correctness of the Algorithm

This section provides a sketch of the proof of correctness and comments on the runtime complexity of the Yo-Yo algorithm. A top-down proof is given based on

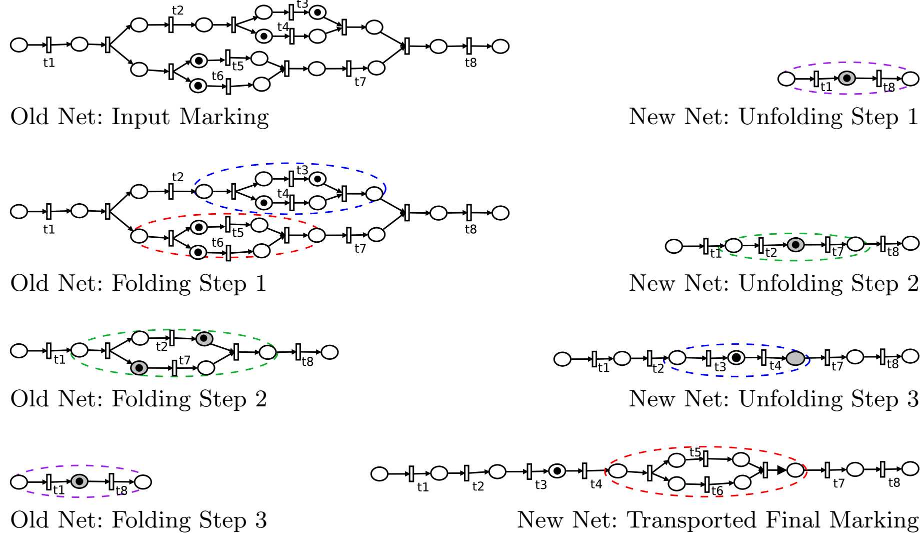


Fig. 9. Token Transportation Through Yo-Yo Steps

a precondition, which is first outlined below.

Completeness of the Catalog A derivation tree of a arbitrary-sized net is composed of derivation trees of the primitive patterns. The folding operation enables us to abstract a bigger net into a single primitive pattern configuration. Given this folding, derivation trees of primitive patterns which are located at a lower level of a bigger derivation tree are abstracted as folded leaf non-terminals in the derivation tree of a pattern located at a higher level. Therefore, a derivation tree of a primitive pattern can have one or more of the following two types of leaf non-terminals based on where the pattern is located in the entire tree: (1) leaves which represent folded lower level patterns. These are tagged as *non-leaf* in the catalog, and (2) leaves which are actual places in the entire net. These are unfolded leaves which are tagged as *leaf* in the catalog.

Coloring of derivation trees is an encoding scheme under which all places fall into either of the three color based classes shown in the Table 4.

Table 4. Coloring Scheme for Catalog Patterns

Type of node	Marking Status	Execution Status	Color
Folded (non-leaf)	Unmarked	Not applicable	Uncolored
Unfolded (leaf)	Unmarked	Not applicable	Uncolored
Folded (non-leaf)	Marked	null-executed (just started)	Black
Unfolded (leaf)	Marked	Not applicable	Black
Folded (non-leaf)	marked	full-executed (just completed)	Red
Folded (non-leaf)	marked	partially-executed	Red

Every place in a pattern can belong to one of the three classes provided that the resultant marking is a valid marking. Every marked place in a valid pattern marking can be colored black or red. This leads to 6 possible colorings of SEQ pattern given that there are 3 valid markings of the primitive SEQ pattern. There are 6 valid markings of the primitive AND pattern which leads to 20 colorings. Similarly, for 6 valid marking of primitive XOR, there are 12 colorings. Out of these 38 cases, three cases of AND, and two cases of XOR are not migratable to any other consistent and valid making in any other pattern as per the correctness criteria. This gives a total of 33 unique migratable colorings of derivation trees of all primitive patterns. However, 4 more cases need to be considered as follows.

It can be seen that the six rows of the table have been colored using three colors. Using six different colors results in a much bigger catalog. It was found that clubbing the cases reduces the size of the catalog considerably, leaving out 4 extra cases that need to be handled separately. The clubbing is done based on whether the nodes are marked, and if marked, whether at least one task in the folded section is done. In the catalog these 4 extra cases are due to pairs 18 and 37, 4 and 36, 5 and 34, 6 and 35. One case in each pair is covered in the above 33 cases. In this way we obtain 37 valid and exhaustive entries in the catalog.

Mappings among this group are given as per the consistency criteria. For some cases among these 37 cases, there are multiple mappings possible. These are resolved by yield-based tie-breaker rules as explained previously.

The Correctness Argument First, the algorithm colors the old derivation tree as per the old net marking, preserving the semantics of derivation tree coloring as given in Section 4.3. Next, it transports the color from the old top pattern to the new top pattern. If this step is not possible without violating consistency, the algorithm terminates. After the transfer, the colors are propagated further down through the descendant patterns in the new tree following the folding order. The color transfer iteratively continues until either no pattern can be colored thus, or till the algorithm terminates on finding the case not migratable. If a case is migratable, Lemma 1 proves that given the yield compatibility at the roots of a peer patterns guaranties that consistent color transfer between them leads to consistent color transfer in the immediate child patterns. To prove that this can be done repetitively for the entire tree Lemma 2 is used. Lemma 3 proves validity of each color transfer. As a result, the algorithm is guaranteed to terminate and produces correct token transportation.

Lemma 1 *For a given pattern P' in the new tree having yield compatible root with peer pattern P in the old tree, consistent color transfer to P' guaranties to find consistent coloring of the immediate child patterns of P'.*

Proof: Coloring P' either by `modularTransport` or `localPropagation` leads to coloring of the roots of the child patterns visible to P'. Given the yield compatibility between the roots of P and P', this color passing either by catalog cases or `localTransport` can result in the following variants coloring of a child root against the root of its peer in the old tree. Let Q' be a direct child pattern of

P'. Let Q' have peer Q in the old tree. (i) Roots of Q and Q' have the same color (ii) Both have no color, and (iii) One of them is colored and the other is uncolored. (Note that, if P' and P are the same patterns, color transfer to P' follows either case (i) or case (ii)).

In case (i), either one of the catalog mappings or the same color replication mapping between Q and Q' is guaranteed to be applicable. Since both of them are preserve consistency by construction, the lemma applies for this case. In case (ii), a pattern remains uncolored if either token has moved past it or has not reached in it yet. In case (iii), the colored root can be either black or red, which gives us four possibilities. If Q is uncolored and the root of Q' is black, the token has not reached in Q, whereas in Q' black color means that is in the source place indicating that no labeled-transition is fired yet. If the root of Q' is red, the token is past Q, whereas in Q' it is in the sink place just after firing the last transition in it. The other two possibilities in this case are reverse of the first two possibilities. Given the yield compatibility between the parent roots, i.e. the roots of P and P', it ensures the same relative positioning of Q and Q' with their respective parent patterns, i.e. they are both either left, right or middle children. Therefore, when P and P' are consistently colored, a token before Q and after Q' is a contradiction, which proves case (ii). Similar argument follows for case (iii) also. In this way consistent transportation for the direct children can be achieved. Case (i) is handled by `modularTransport`, case (ii) and the last two possibilities of case (iii) do not require any coloring action, and the first two possibilities of case (iii) are handled by `localPropagation`.

Lemma 2 *Let two Yo-Yo compatible derivation trees have patterns P, Q in the old tree and their respective peer patterns P', Q' in the new tree. Let Q be child of P and Q' is child of P'. Let the roots of P and P' satisfy yield compatibility. If P' and Q' satisfy Lemma 1, then so do Q' and all of its immediate children.*

Proof: The lemma is about a structural property achieved due to Yo-Yo compatibility and folding order that ensures yield compatibility between the roots of peer patterns extracted from the folding order for coloring at each step of iteration. We use notation S_X to represent the yield sequence of the root of derivation sub-tree X. Let the transition terminals of peer patterns P and P' be denoted by t_x (left) and t_y (right). P and P' can be either of Sequence or fork-join patterns. We analysis the case where P is a Sequence and P' is a fork-join. The other three cases can be proved similarly. When P is a Sequence and P' is a fork-join, P' has at most six immediate child patterns rooted at its six leaf non-terminals. These are shown in Fig. 10a as Q'_{11}, Q'_{12}, Q'_{21}, Q'_{22}, Q'_{31}, and Q'_{32}. The term Q' in the lemma refers to each one of them. Similarly, P being a Sequence has three child patterns Q_1, Q_2 and Q_3 as shown in Fig. 10b. Since the roots of P and P' are yield compatible, $S_{Q_1} t_x S_{Q_2} t_y S_{Q_3} =_y \{S_{Q'_{11}} S_{Q'_{12}} t_x S_{Q'_{21}}, S_{Q'_{22}} t_y S_{Q'_{31}} S_{Q'_{32}}\}$. From this, at subtree level we can observe that $S_{Q_i} =_y S_{Q'_{i1}} S_{Q'_{i2}}$, $i \in \{1, 2, 3\}$. As each pattern has two transitions, to accommodate four transitions, each of Q_1, Q_2 and Q_3 is a hierarchy of two patterns as shown in Fig. 10c or 10d.

Now there are two cases: Q' can be either Q'_{i1} or Q'_{i2}. For the first case, P has subtree as Fig. 10c, and for the second P has subtree as Fig. 10d. In the folding

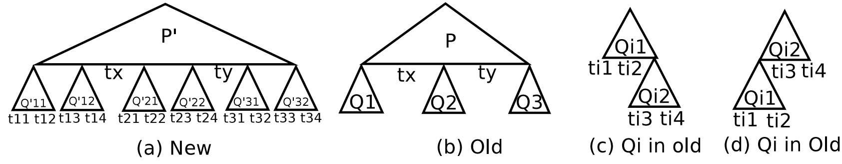


Fig. 10. Visualization of the subtrees of P' and P in Lemma 2

order corresponding to the first case, element $< Q_{i1}\text{-}Q'_{i1} >$ precedes element $< Q_{i2}\text{-}Q'_{i2} >$, and for the second case they are swapped. As the coloring follows the folding order, after every step the visible parts of the nets are composed of the patterns from the already traversed. In this regard, for the extracted element $< Q_{ij}\text{-}Q'_{ij} >$, the roots of Q_{ij} and Q'_{ij} are yield compatible and hence, consistent color transfer to Q'_{ij} guaranties consistent color transfer to the next level, thereby proving Lemma 1 for this case. The justification for other combinations of P and P' mentioned at the beginning of this argument is skipped due to lack of space.

Lemma 3 *Peer to peer color transfer preserves validity pattern of coloring.*

Proof: For catalog transfer cases validity is preserved by construction. For same color replication, validity of the newly produced color is preserved by the correctness of the old pattern coloring and the validity preserving step of the algorithm (line 3 of `modularTransport`). For local transportation, validity is preserved by following the definitions. Therefore, the lemma is proved.

Time Complexity and Brief Comparison As it can be observed from the algorithm, the asymptotic time complexity of the runtime token transportation in terms of number of patterns n is linear. This computation does not include parsing of the workflow specifications, finding out the compatible derivation tree pairs and the folding order. Therefore, the Yo-Yo approach improves the runtime migration cost by pushing much of the complexity into one-time schema level computations and design time catalog construction depending on the grammar. As compared to history-replay approach, Yo-Yo transportation does not compute or reproduce history. Transportation via pre-computed mappings among *marked patterns* achieves the desired migration.

7 Conclusions and Future Work

The paper developed a novel catalog based dynamic token transportation technique called Yo-Yo algorithm with the help of contributory concepts such as CWS specification grammar for block structured WF-nets, derivation trees and their colorings, peer patterns, Yo-Yo compatibility, catalog based transportation, and folding and unfolding of hierarchically organized patterns. The algorithm uses the ready-made consistent and valid migration solutions from the token transportation catalog repetitively to achieve correct transportation for non-primitive bigger nets. Also, immediately non-migratable markings are automatically identified by the algorithm due to the non-existence of the corresponding entries in

the catalog. We supplemented the discussion on the algorithm with an example realistic situation and also provided the sketch of the proof of correctness. We plan to integrate an implementation of this algorithm in the workflow engine described in [17]. We aim to generalize the approach to handle replacement, removal, addition and swapping of tasks.

References

1. van der Aalst, W.M.: The application of petri nets to workflow management. Journal of circuits, systems, and computers **8**(01) (1998)
2. Pradhan, A., Joshi, R.K.: Token transportation in petri net models of workflow patterns. In: 7th India Software Engineering Conference, India, 2014. (2014)
3. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of conference on Organizational computing systems, ACM (1995)
4. Ellis, C.A., Keddara, K.: A workflow change is a workflow. In: Business Process Management, Models, Techniques, and Empirical Studies, Springer-Verlag (2000)
5. van der Aalst, W.M.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers **3**(3) (2001)
6. Sun, P., Jiang, C.: Analysis of workflow dynamic changes based on petri net. Information and Software Technology **51**(2) (2009)
7. Cicirelli, F., Furfaro, A., Nigro, L.: A service-based architecture for dynamically reconfigurable workflows. Journal of Systems and Software **83**(7) (2010)
8. Agostini, A., Michelis, G.D.: Improving flexibility of workflow management systems. In: Business Process Management: Models, Techniques, and Empirical Studies. LNCS 1806, Springer (2000)
9. van der Aalst, W.M., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science **270**(1) (2002)
10. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
11. Morita, K., Watanabe, T.: The legal firing sequence problem of petri nets with state machine structure. In: Circuits and Systems, 1996. ISCAS'96., Connecting the World., 1996 IEEE International Symposium on. Volume 3., IEEE (1996) 64–67
12. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web services business process execution language version 2.0. OASIS standard **11** (2007)
13. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: Business process management. Springer (2008)
14. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. **66**(3) (2008)
15. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (2003)
16. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering **68**(9) (2009) 793–818
17. Pradhan, A., Joshi, R.K.: Architecture of a light-weight non-threaded event oriented workflow engine. In: The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, India, 2014. (2014) 342–345