Automated Modelling of Deadlock-free Petri Nets Using Duplicated Transition Labels

Román Pomares-Angelino, Ernesto López-Mellado
CINVESTAV Unidad Guadalajara
45019 Zapopan, Jal. Mexico
{jirpomares, elopez}@gdl.cinvestav.mx

Abstract—The paper addresses the problem of automated synthesis of deadlock-free Petri nets (PN) from event sequences. A method for determining substructures that yield deadlocks is presented. Based on structural patterns that may appear during the synthesis of PN from some types of event sequences, deadlock substructures, called inconsistent, are detected. Afterwards, a repairing technique is applied to correct every inconsistent substructure by adding a new transition, which is associated to an event symbol already assigned to a transition in the inconsistent substructure. The algorithms derived from the technique, which have polynomial-time complexity, have been implemented and tested on examples of diverse structures.

Keywords—Petri net discovery, Model repair, Structural patterns, duplicated transitions labels.

I. INTRODUCTION

Automated modelling of discrete event processes from logs of observed behaviour is currently a problem addressed by several research groups in the areas of discrete manufacturing systems [1] and workflow process management [2]. In each field, this problem is called identification and discovery respectively. Most of the proposed identification/discovery methods aim to obtain Petri net (PN) models, which are well suited to represent complex discrete-event processes when discovery is used as a resource for reverse engineering [3], [4], [5]. These methods often hold in the problem statement the constraint that the labelling of transitions in the PN is defined as bijective function on the events (or tasks) alphabet. Consequently, some traces of the events log cannot be represented in the model; thus, some transitions labelled with ε are used; they are called silent, invisible, or unobservable transitions.

For example, the log λ = {ABC, AC} can be represented by the PN shown in Figure 1, in which the silent transition labelled with ε allows firing the sequence AC.

![Fig. 1. PN with an ε-transition](image)

In the last few years, several discovery methods have been proposed for relaxing this constraint allowing obtaining workflow nets (WFN) including silent transitions [6][7]; the methods called alpha# and alpha$ respectively allow determining successfully silent transitions of type initialize, finalize, skip, redo, and switch.

Recently, other simplest method based on a pre-processing of the set of traces in the event log λ has been proposed [8]; λ is classified into three classes: a) the normal traces, which lead to a basic WFN that may include initialize and finalize silent transitions, and b) the abnormal short traces, and the abnormal long traces, which are used to refine the basic model by adding skip and redo silent transitions respectively. Later, in [9] an event clustering method that finds switch transitions is presented.

In this paper, an alternative approach to construct PN with ε-transitions is proposed. The discovered PN may have several transitions using the same event symbol. Figure 2 shows an equivalent PN to represent the log λ = {ABC, AC}. Furthermore, the use of replicated event symbols is optimised.

![Fig. 2. PN using the same label on two transitions.](image)

In this study, the subclass of PN called workflow nets (WFN) is dealt, but the proposed repairing method can be applied for the synthesis of ordinary PN.

The paper is organised as follows. Section 2 presents the notation on Petri nets and the problem formulation. Section 3 describes the pattern-based technique for determining deadlock substructures. Section 4 presents the technique for rewriting such structures as deadlock-free ones with additional transitions using repeated event symbols. Section 5 outlines the developed software tool and presents tests on several case studies.

II. BACKGROUND, PROBLEM AND APPROACH

A. Petri nets basics [2]

Definition 1. An ordinary Petri net structure G is a bipartite digraph represented by the 3-tuple G = (P, T, F); where: P = {p1, p2, ..., p|P|} and T = {t1, t2, ..., t|T|} are finite sets of nodes named places and transitions respectively; \( F \subseteq P \times T \cup T \times P \) is a relation representing the arcs between the nodes.
For any node \( x \in P \cup T \), \( x = \{ y | (y, x) \in E \} \) and \( x^* = \{ y | (x, y) \in E \} \). The incidence matrix of \( G \) is \( C = [c_{ij}] \); where \( c_{ij} = -1 \) if \( (p_i, t_j) \in E \), \( t_j \in E \) and \( (t_j, p_i) \in E \); \( c_{ij} = 1 \) if \( (t_j, p_i) \in E \) and \( (p_i, t_j) \in E \); \( c_{ij} = 0 \) otherwise.

The places in \( P \) can be empty or marked by one or more tokens. A marking \( M: P \rightarrow \mathbb{N} \) determines the number of tokens within the places; \( \mathbb{N} \) is the set of non-negative integers. A marking \( M \) is usually denoted by a vector \(( M^p_0 \) \(^\text{T}\) \) \(^\text{T}\) \(^\text{T}\). The current state of the modelled system is a set of traces \( \sigma = \{ \tau | \sigma \in \Sigma^* \} \). A Petri Net system or Petri Net (PN) is the pair \( N = (G, M_0) \), where \( G \) is a PN structure and \( M_0 \) is an initial marking. \( R(G, M_0) \) denotes the set of all reachable markings from \( M_0 \).

**Definition 2**. A Petri Net system or Petri Net (PN) is the pair \( N = (G, M_0) \), where \( G \) is a PN structure and \( M_0 \) is an initial marking. \( R(G, M_0) \) denotes the set of all reachable markings from \( M_0 \).

**Definition 3**. A Petri system is 1-bounded or safe iff, for any \( M \in R(G, M_0) \) and any \( p \in P \), \( M(p) \leq 1 \). A PN system is live iff, for every reachable marking \( M \in R(G, M_0) \) and \( t \in T \) there is a \( M_t \in R(G, M_t) \) such that \( t \) is enabled in \( M_t \).

**Definition 4**. A t-invariant \( Yt \) of a PN is a non-negative integer solution to the equation \( CY = 0 \). The support of \( Yt \), denoted as \( \langle Yt \rangle \), is the set of transitions whose corresponding elements in \( Yt \) are positive. \( Yt \) is minimal if its support is not included in the support of other t-invariant. A t-component \( G(Yt) \) is a subnet of \( PN \) induced by a \( Yt \) with \( C(Yt) \text{=} (P_t, T_t, F_t) \), where \( P_t = \langle Yt \rangle \cup \langle Yt \rangle \), \( T_t = \langle Yt \rangle \), \( F_t = (P_t \times T_t \cup P_t \times T_t) \cap F \).

In a t-invariant \( Yt \), if we have initial marking \( M_0 \) that enables a \( t_i \in \langle Yt \rangle \), when \( t_i \) is fired, then \( M_0 \) can be reached again by firing only transitions in \( \langle Yt \rangle \).

**Definition 4.** A WorkFlow net (WFN) \( N \) is a subclass of PN owning the following properties (van der Aalst, 2004): (i) it has two special places: \( i \) and \( o \). Place \( i \) is a source place: \( i = \emptyset \), and place \( o \) is a sink place: \( o^* = \emptyset \). (ii) If a transition \( t_i \) is added to \( PN \) connecting place \( o \) to \( i \), then after that the PN (called extended WFN) is strongly connected.

**Definition 5**. A WFN \( (N, M_0) \) is said to be sound if any marking \( M_t \in R(N, M_0) \), \( o \in M_t \rightarrow M_t = [o] \) and \( [o] \in R(N, M_t) \) and \( (N, M_0) \) contains no deadlocks. An extended WFN sound is live and bounded. A WFN can represent a process behaviour by associating task labels to some transitions.

**Definition 6**. A labelled WFN is a four-tuple \((N, M_0, \Sigma, L)\) where \( \Sigma \) is a finite set of event labels, and \( L: T \rightarrow \Sigma \cup \{ \epsilon \} \) is the labelling function. Transitions labelled with \( \epsilon \) are called silent or unobservable, otherwise they are called observable. Additionally, \( \forall t_i, t_j \in T, t_i \neq t_j \) if \( L(t_i), L(t_j) \in \Sigma \) then \( L(t_i) \neq L(t_j) \); i.e., two transitions cannot have the same label from \( \Sigma \).

**B. Problem statement**

**Definition 7**. Let \( \Sigma \) be a finite set of tasks labels; an event log \( \lambda \) is a set of traces \( \sigma = \{ A_1 \cdots A_k \} \) \( \in \Sigma^* \), \( |\sigma| = k, A_j \in \Sigma, 1 \leq j \leq k \) refers to the task at position \( j \).

In a workflow net, the first and last tasks in every trace correspond to transitions in \( i^* \), \( o^* \) respectively.

**Definition 8**. Given a workflow log \( \lambda = \{ \sigma_1, \sigma_2, \ldots, \sigma_l \} \) generated by a sound WFN, the discovery problem consists of synthesising a safe PN structure using observable transitions in which the event symbol \( \epsilon \) is not used and the number of repeated symbols in \( \Sigma \) is minimum; that is, the constraints of Definition 7 are relaxed. The number of places is unknown.

**Assumptions.** Considering that \( C \) is generated by a WFN \( N_t \) with \( \epsilon \)-transitions, the discovered model \( N_s \) will be equivalent to \( N_t \) by holding the following assumptions:

- A1. The process is well behaved, i.e., there are no deadlocks nor overflows during the observation period.
- A2. All the traces that do not fire \( \epsilon \)-transitions are included in the event log.

**C. Overview of the proposal**

The strategy of the method is repairing an incorrect model obtained by a discovered technique that does not deal with the behaviour of PN including \( \epsilon \)-transitions. For example, the method CoMiner [10] obtains from \( \lambda=\{ABC, AC\} \) the following set of causal dependencies \{[1, A], [A, B], [A, C], [B, C], [C, -] \} that lead to the PN shown in Figure 3, which cannot replay the log \( \lambda \). Notice that the causal dependency \{A, C\} is built through the places \( p_1 \) or \( p_3 \); this provokes a deadlock since any of the traces in \( \lambda \) cannot fire.

![Fig. 3. PN obtained by a method not able to discover \( \epsilon \)-transitions.](image)

The proposed method locates these substructures in an incorrect model built from behaviours issued from silent transitions Skip, Redo, and Switch implicit in the traces of \( \lambda \). Afterwards, the inconsistent substructures are transformed into an equivalent one using additional transitions labelled with an event symbol used in the wrong structure.

### III. DETECTING INCONSISTENT SUBSTRUCTURES

**A. The method CoMiner**

Any method that does not deal with event logs involving silent events will construct wrong PN structures when the sequences follow a behaviour corresponding to silent events. We outline a method that may build WFN from non-complete (thus reduced) event logs [10].

The method processes the event log \( \lambda \) for determining conjunct occurrence classes, which are sets of events that always occur in a same set of sequences; this allows inferring some concurrent relations and causal relations between event symbols not observed explicitly in \( \lambda \). [A, B] expresses that A and B are in a causal relation, whilst \( C||D \) states that C and D are concurrent.

Both computed and inferred dependencies determine net structures that can be composed by transition fusion. Sequential compositions build paths; if there are two dependencies [A, B]
and \([B, C]\), it can be built a path of three transitions \(A, B\), and \(C\). The causal dependencies \([A, B]\) and \([A, E]\) lead to a composed dependency \([A, B][E]\) or \([A, B+E]\) if \(B\) and \(E\) are concurrent or not respectively. Figure 4 summarises the composition operations. Successive compositions lead to a connected WFN in which every transition has associated a unique event symbol used in \(\lambda\).

![Fig. 4. Composition of dependencies [10].](image)

**B. Inconsistent structures**

The set of causal dependencies \(\{\cdot, A\}, [A, B], [A, C], [B, C], [C, \cdot]\) can be composed yielding the set of causal dependencies \(\{\cdot, A\}, [A, B+C], [A+B, C], [A, \cdot]\). However, the composition will construct the model shown in Figure 2 which expresses that \(B\) and \(C\) are concurrent by creating a dependency \([A, B+C]\) through \(p_2\) and \(p_1\); similarly, the dependency \([A|B, C]\) is created. These structures contradict the composed dependencies, and then the PN does not represent the behaviour of \(\lambda\).

Notice that the composed dependencies that build such inconsistent relations are OR dependencies; \([A, B+C]\) is named Or-split and \([A+B, C]\) is named Or-join; the set of all these dependencies is called compOr.

**Definition 9.** Let \(r_1, r_2 \in \text{compOr}\) be two composed Or dependencies, where \(r_1\) is an Or-split, and \(r_2\) is an Or-join. Then the pair \((r_1, r_2)\) is an inconsistent structure (IS) iff these dependencies lead to a substructure that has an And-split or an And-join relation.

Let us analyse the relationship between IS and the behaviour due to the silent transitions.

1) **Skip transition**

A structure derived from traces that involve a skipping behaviour, such as \(\{ABCDE, AE\}\) is shown in Figure 5. The causal dependencies are \(\{\cdot, A\}, [A, B], [B, C], [C, D], [D, \cdot], [A, E], [E, F], [F, D], [B, F]\), the composed Or dependencies are \(r_1=[A+B+E], r_2=[A+D+E], r_3=[A+C, B], r_4=[C, B+D]\).

![Fig. 5. PN built from a log involving a skip behaviour](image)

2) **Redo transition**

The log \(\{ABCD, ABCBCD\}\) represents a Redo behaviour; it includes a trace that involves an iteration. The causal dependencies are \(\{\cdot, A\}, [A, B], [B, C], [C, D], [D, \cdot], [C, B]\), the composed Or dependencies are \(r_1=[A+C, B], r_2=[C, B+D]\).

![Fig. 6. PN built from a log including a Redo behaviour](image)

3) **Switch transition**

The log \(\{ABCD, AEFD, ABF\}\) includes a trace that involves a switch transition in a WFN. The causal dependencies are \(\{\cdot, A\}, [A, B], [B, C], [C, D], [D, \cdot], [A, E], [E, F], [F, D], [B, F]\), the comp-Or dependencies are \(r_1=[A, B+E], r_2=[A+D, E], r_3=[B+E, F]\).

![Fig. 6. PN built from a log including a Redo behaviour](image)

4) **Patterns of inconsistent structures**

The previous analysis shows that the behaviours of a WFN built using IS are not correct. In the three cases, the form of the Or dependencies is similar and follows the same pattern stated below.
Proposition 1. Let \( r_i, r_o \in \text{compOr} \) be two composed Or dependencies. The pair \((r_i, r_o)\) is an inconsistent structure iff these dependencies have the following form: \( r_i = [a, b + d + \ldots + r] \), \( r_o = [a + c + \ldots + k, d] \), where \( a, b, c, d, r, k \in \Sigma \).

Proof. \((\rightarrow)\) \( r_i \) states a causal relation between \( a \) and \( d \) through a place \( p_i \), whilst \( r_o \) states a causal relation between \( c \) (and some other transitions) and \( d \) through another place \( p_j \). This constructs an And-join structure between \( a \) and \( c \) with \( d \), which leads to a deadlock situation. \((\leftarrow)\) Conversely, by Definition 9, an IS is characterized by the structures in dependencies \( r_i \) and \( r_o \).

C. Determining inconsistent structures

The previous pattern is useful to detect IS in a PN discovered by a method that does not deal with silent transitions. A matching test can determine the set of IS from the comp-Or substructures built by such a method.

In order to verify if a pair of dependencies \((r_i, r_o)\) match the pattern stated in Proposition 1, one must fulfill the following conditions. For a dependency \( r = [S, T] \in R \), the operator \( \ast \) selects the parts of the dependency: \( 'r' = S \) (left) and \( 'r' = T \) (right).

Proposition 2. Let \( r_i = [W, X] \), \( r_o = [Y, Z] \) be two comp-Or dependencies. \((r_i, r_o)\) is an IS if \( a) \ r_i \cap r_o \neq \emptyset \) and \( b) \ r_i \cap r_o \neq \emptyset \).

Proof. Condition \( a) \) implies that the single transition in \( W \) is also included in \( X \). Similarly, \( b) \) implies that the single transition in \( Z \) is included in \( Y \). This fulfills the matching of the pattern stated in Proposition 1.

Consider \( R \) the set of composed dependencies computed by a method that does not deal with silent transitions. We need to analyse the compOr dependencies to determine if some pair of dependencies fulfill the conditions in the pattern. First, the Or-dependencies are selected; the conditions of Proposition 2 are tested.

The procedure is summarised in Algorithm 1. The type of relations in \( S \) or \( T \) is obtained by the function \( \text{type-} r : \text{type-} r(S) \in \{\text{Or}, \text{And}\} \).

Algorithm 1. Ident-IS: gets the set of inconsistent structures

Input: \( R \) // composed dependencies
Output: \( \text{Ce}i \) // set of IS

1: \( \text{Ce}i \leftarrow \emptyset \); \( \text{compOr} \leftarrow \emptyset \)
2: \( \forall d \in R \)
3: \( \text{if} \ \text{(type-} r(d) = \text{Or}) \text{ or (type-} r(d') = \text{Or}) \)
4: \( \text{then compOr} \leftarrow \text{compOr} \cup \{d\} \)
5: \( \forall r_i \in \text{compOr} \)
6: \( \forall r_o \in \text{compOr} \)
7: \( \text{if} \ r_i \cap r_o \neq \emptyset \) and \( r_i \cap r_o \neq \emptyset \)
8: \( \text{then Ce}i \leftarrow \text{Ce}i \cup \{(r_i, r_o)\} \)
9: \( \text{Return Ce}i \)

IV. Rewriting inconsistent structures

A. Strategy

Inconsistent structures create concurrent dependencies through transitions that become and-split or and-join as described above. The main idea is to duplicate such transitions and use the same event symbol.

This situation is clearly exhibited in Figure 5. In the PN of Fig 5.b, \( A \) is an And-split transition and \( E \) is an And-join transition; this is the outcome of merging the transitions in the IS \([(A, B+E), (A+D, E)]\). We can see that the transitions that have to be duplicated are \( A \) and \( E \), as suggested by Figure 5.a, then an arc to link the place \( 'A' \) to the duplicated transition \( 'A' \). Similarly, transition \( 'E' \) is linked to the place \( 'E' \). This is shown in Figure 8, where \( 'A'(E') \) is the same symbol than \( A \) (E). This PN has not deadlocks and represents the log \{ABCD, AE\}; the trace AE is fired in two manners using the duplicated transitions, thus, one of them can be discarded, for example, the new transition \( A \) that involves non-determinism.

PN including Redo and Switch behaviours using duplicated transitions are shown in Figures 9 and 10 respectively.

Fig. 7. PN built from a log including a Switch behaviour

Fig. 8. PN with Skip behaviour using duplicated transitions

Fig. 9. PN with Redo behaviour using duplicated transitions

Fig. 10. PN with Switch behaviour using duplicated transitions
B. Locating transitions to duplicate

In the previous examples, which illustrate the case of Skip, Redo, and Switch behaviours, the transitions that need to be duplicated in an IS \((r_i, r_o)\) are included simultaneously in two parts of each of the composed Or dependencies \(r_i\) and \(r_o\). They can be determined using the following condition.

**Proposition 3.** Let \((r_i, r_o)\) be a IS. The transitions that must be duplicated to avoid the IS are \(r_i \cap r_o\) and \(r_i' \cap r_o'\).

**Proof.** The transitions in \(r_i \cap r_o\) and \(r_i' \cap r_o'\) are precisely those that make fulfil the conditions of Proposition 2 to be an IS; besides, as one of the parts is a singleton, the transitions are uniquely determined. ■

C. Rewriting IS

Let us analyse the strategy to rewrite each substructure according to the behaviour of the silent transition.

1) Skip

In the example of Figure 8 the transitions \(r_i \cap r_o = \{t_i\}\), where \(L(t_i)=A\), and the transitions in \(r_i' \cap r_o' = \{t_i\}\), where \(L(t_i)=E\) have to be duplicated. The rewriting is performed as follows:

- In the case of \(t_i\), a new transition \(t_k\) is created, with \(L(t_k)=A\); then, the flow is updated by deleting in \(F\), the arc from \(t_i\) to \(t_k\), and adding the new arcs \((t_i, t_k)\) and \((t_k, t_i)\) to \(F\).
- In the case of \(t_i\), a new transition \(t_i\) is created, with \(L(t_i)=E\); then, the flow is updated by deleting the arc from \(t_i\) to \(t_i\), and adding the new arcs \((t_i, t_i)\) and \((t_i, t_i)\) to \(F\).

2) Redo

In the example of Figure 9 the transitions \(r_i \cap r_o = \{t_i\}\), where \(L(t_i)=C\) and the transitions in \(r_i' \cap r_o' = \{t_i\}\), where \(L(t_i)=B\) have to be duplicated. The rewriting is performed similarly as defined for the skip behaviour with \(L(t_k)=C\) and \(L(t_i)=B\).

3) Switch

In the example of Figure 10 the transitions \(r_i \cap r_o = \{t_i\}\), where \(L(t_i)=B\), and the transitions in \(r_i' \cap r_o' = \{t_i\}\), where \(L(t_i)=F\) must be duplicated. The rewriting is performed similarly as defined for the skip behaviour with \(L(t_k)=B\) and \(L(t_i)=F\).

4) Minimising duplicated transitions

The previous strategy adds two new transitions \(t_k, t_i\) for every inconsistent structure in the discovered model before rewriting. Following this strategy, the repaired WFN is sound and represents correctly all the traces in the log.

However, it contains redundant paths regarding the traces related to silent transitions. Then, for simplifying the repaired PN, \(t_k\) can be discarded since it has the same label of \(t_i\) and both are output transitions from the same place; hence, they represent a non-deterministic choice.

5) Rewriting procedure

Now we can apply the test stated by Proposition 3 to correct all the IS of a model using the previous rewriting and simplification steps; then, we will obtain a sound WFN. The rewriting procedure is summarised in Algorithm 2.

**Algorithm 2. Repair-IS**

**Input:** \(Cei, F\)

**Output:** \(F\) (repaired)

1. If \(Cei = \emptyset\) then Return \(F\)
2. \(\forall (r_i, r_o) \in Cei\)
3. \(t_i \leftarrow \text{Item}(r_i \cap r_o); \ \text{\textbackslash\slash }\text{obtains the object in the set \{t_i\}}\)
4. \(t_j \leftarrow \text{Item}(r_j \cap r_o');\)
5. \(T \leftarrow T \cup \{t_i, t_j\};\)
6. \(F \leftarrow F \setminus \{(t_i, t_j), (t_i, t_j)\};\)
7. \(F \leftarrow F \cup \{(t_i', t_i), (t_i, t_i')\};\)
8. \(L(t_i) \leftarrow L(t_i)\)
9. Return \(F\)

V. IMPLEMENTATION AND TESTS

The algorithms derived from this method and auxiliary procedures have been implemented using Java 1.8.0 on the IDE of Netbeans 8.2. The software module is integrated with another module implementing the CoMiner and a PN editor.

The test scheme is shown in Figure 11. First, a PN is edited using PIPE [11] and the model is executed to produce an artificial event log; this allows validating in a controlled manner, that the discovered models are correct; the artificial log is the input to a module that implements CoMiner, which discovers a possible incorrect PN. The structure of such a PN, in particular, the composed dependencies \(R\) are processed by the rewriting module. Finally, the repaired PN is displayed using the Graphviz library [12].

![Implementation scheme](image)

Using this test scheme, the developed software has applied to discover WFN from artificial event logs obtained from known models with silent transitions, which involve diverse structural complexity. We include below two simple case studies.

**Test 1.** The processed log is \(\lambda=(xabcd\text{efg}, xabcde\text{fg}, x\text{abcde}f\text{bedc}\text{fg}, x\text{ABC}\text{Dy}, x\text{abc}\text{Dy}).\) The model discovered using CoMiner is shown in Figure 12, and the repaired model is shown in Figure 13, where the transitions using labels \(b, c,\) and \(e\) are duplicated, their labels are \(b', c',\) and \(e'\) respectively.

In the repaired PN, \(b'\) is a redo transition, \(c'\) is a switch transition, and \(e'\) is used to skip transition \(d.\) The processing time of the repairing algorithm in this test is 0.8162 ms.
**Test 2.** The processed log is $\lambda_2=\{xjlmknqpr, xjlknpqmoro, xjlmknqpor, xjknqmlnor, xjknmlorpqr, xjknplmrq, xjloknpqr, xjknqknpqlmor, xjknqnpqmlor, xabcef, xabdef, xaghif, xabsif, xaf, xif\}$.

The model discovered using CoMiner is shown in Figure 14, and the repaired model is shown in Figure 15; in this PN there are six transitions using labels $i$, $f$, $k$, $n$, $o$, and $s$, which are duplicated.

The corresponding duplicated transitions $i'$, $f'$, and $o'$ are skip transitions, $k'$ and $n'$ are redo transitions, and $s'$ is a switch transition. The processing time of the repairing algorithm in this test is 2.1573 ms.

**CONCLUSIONS**

We have presented a method for discovering sound workflow nets from event logs involving behaviour due to silent transitions. The models may have two or more transitions labelled with the same event symbol. The approach held is repairing an incorrect model obtained by a method that does not deal with silent transitions. The strategy is based on the use of structural patterns that determine the inconsistent substructures in the model that provoke deadlocks. Then, such substructures are rewritten using additional transitions labelled with an event symbol already used in the inconsistent substructures.

The repaired model is sound and contains the minimum number of repeated labels. The polynomial-time algorithms derived from the proposed method have been implemented and tested on examples of diverse structures.

At our knowledge, the discovery of PN including duplicated transitions as a solution to handle event logs that involves silent transitions behaviour has not been addressed in related works; Current research addresses the transformation of PN with duplicated transitions into PN with silent transitions.

**REFERENCES**


Fig. 13. PN repaired with duplicated transitions equivalent to that in Fig. 12.

Fig. 14. PN discovered by CoMiner from $\lambda_2$.

Fig. 15. PN repaired with duplicated transitions equivalent to the model in Fig. 14.