Process Windows

Andrey Mokhov†, Jordi Cortadella‡, Alessandro de Gennaro†

†Newcastle University, United Kingdom
‡Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*—We describe a method for formally representing the behaviour of complex processes by *process windows*. Each window covers a part of the system behaviour, i.e. a part of the underlying transition system, and is easier to understand and analyse than the complete transition system. Process windows can overlap and have shared states and transitions so that the complete system behaviour is the *union* of window behaviours. We demonstrate the advantage of such representations when dealing with complex system behaviours, and discuss potential applications in circuit design and process mining.**

As a motivational example we consider the problem of covering transition systems by *marked graphs*, or more generally choice-free Petri nets. The obtained windows correspond to choice-free behavioural *scenarios* of the system, wherein one window can take over, or *wake up*, after another window has become inactive. The corresponding *wake-up conditions* and *wake-up markings* can be derived automatically.

I. Introduction

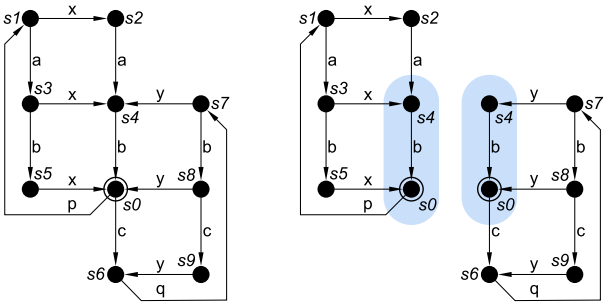
Understanding and specifying the behaviour of a complex concurrent system is a difficult task. Transition systems often suffer from the state space explosion and even Petri nets struggle to represent the behaviour of many real systems in a concise way, because of multiple behavioural scenarios entangled in a single net.

In this paper we show how a transition system can be decomposed into a set of simpler behavioural models, further referred to as *windows*. The original behaviour can be recovered as the *union* of the windows. A window models only one aspect of the system's behaviour that can be characterised by simple event relationships. By choosing windows with simple representations, such as marked graphs, choice-free nets, free-choice nets, etc., one can make sure that each individual window is simple enough to understand, visualise and specify.

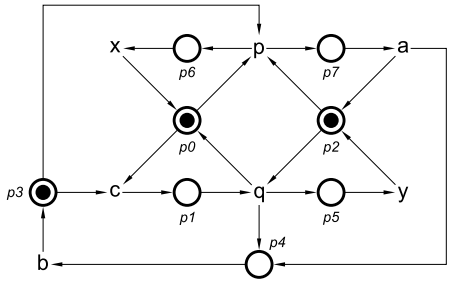
A. Motivational example

Consider a transition system and its Petri net representation shown in Fig. 1(a,c). Arguably, the Petri net is more difficult to understand than the transition system, due to a mix of concurrency and choice. One can decompose the transition system into two simpler ones, which we call windows, that are choice-free, as shown in Fig. 1(b). The windows have very simple Petri net representations W_1 and W_2 shown in Fig. 1(d): the choice aspect of the system behaviour has been abstracted away from individual windows, and is implicitly represented by the alternation of behaviour between windows.

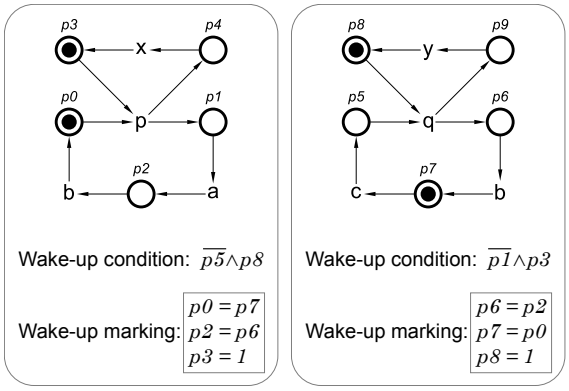
The decomposition can be automated aiming to produce windows with specific behavioural properties. In this example, the resulting windows are *marked graphs*, a class of Petri nets with particularly well-understood structural properties.



(a) Transition system. (b) Window decomposition.



(c) Petri net.



(d) Windows W_1 (left) and W_2 (right).

Fig. 1: Motivational example.

An important property of the decomposition is that windows can have *overlapping behaviours*, which provide bridges between windows. As seen in Fig. 1(b) the windows contain two shared states: $s0$ (the initial state) and $s4$. The current state of the system can therefore occasionally *be seen in* two windows simultaneously, as shown in Fig. 2. By firing p or c in $s0$ the system leaves the shared state and proceeds according to one of the windows until it eventually comes to $s0$ or $s4$, *waking up* the inactive window. Note: the system can stay in the second window indefinitely by looping through states $\{s6, s7, s8, s9\}$.

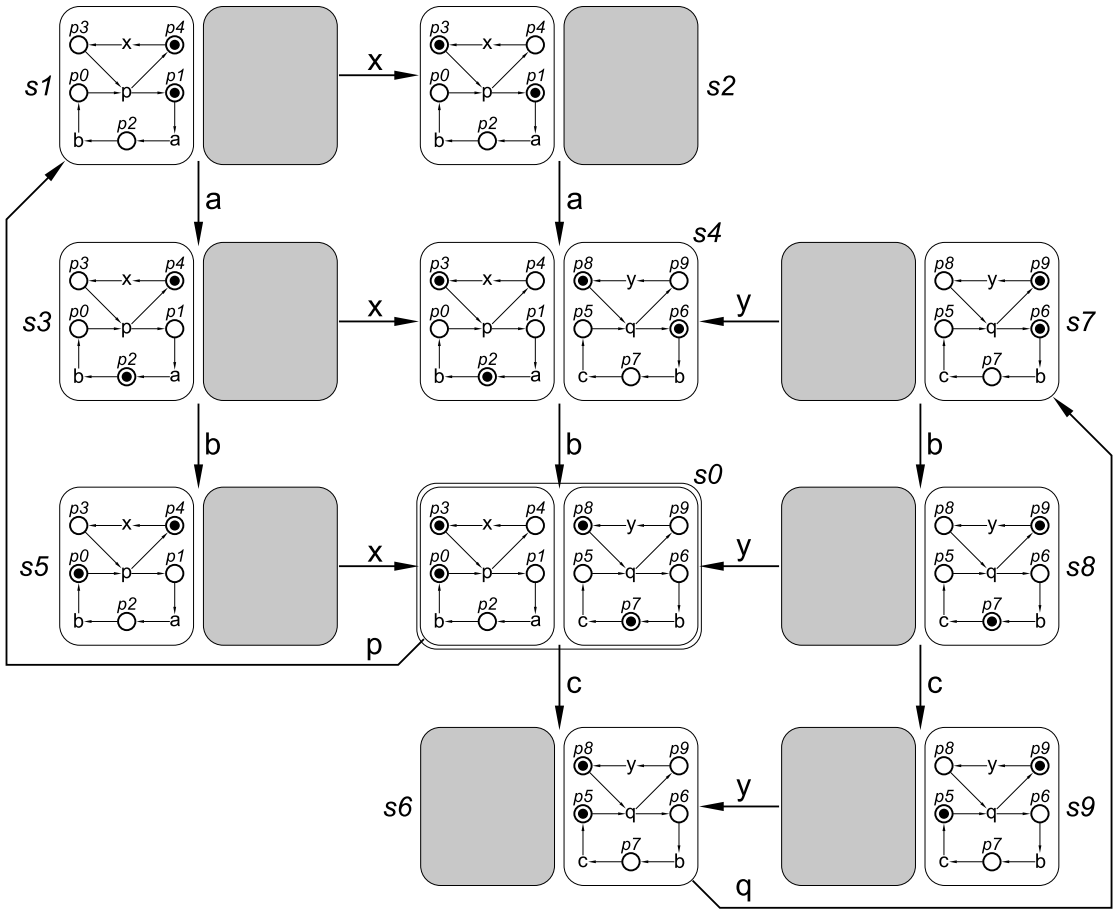


Fig. 2: Simulating windows from Fig. 1 (active windows are transparent, inactive windows are opaque).

The main contributions of this paper are as follows:

- We introduce **process windows**, both informally by examples and formally, in Sections II and III.
- Process windows can be discovered automatically using the **window decomposition algorithm** that is presented in Section IV.
- A method for automated **derivation of wake-up conditions and markings** is presented in Section V.
- We explore **applications of process windows** in circuit design and process mining in Section VI, and integrate the presented algorithms and design methodology into the open-source WORKCRAFT modelling framework [1][2].

Section VII discusses the potential of process windows for modelling systems that contain resource arbitration and OR causality. The related work is reviewed in Section VIII.

II. THE EXAMPLE IN MORE DETAIL

In this section we clarify the correspondence between a transition system and its representation using process windows. Fig. 2 shows the transition system from Fig. 1(a), where each state is expanded into the corresponding state of the two process window nets in Fig. 1(d).

A window is *active* whenever it covers the current state, otherwise it is *inactive*; inactive windows are drawn opaque in Fig. 2. For example, both windows are active in states $s4$ and $s0$, and the state transition $s4 \xrightarrow{b} s0$ is reflected in both windows by firing the transition b. Depending on which transition occurs in $s0$, either W_1 or W_2 becomes inactive. Indeed, W_1 does not cover the transition c, and W_2 does not cover the transition p. Transitions $\{s7 \xrightarrow{y} s4, s8 \xrightarrow{y} s0\}$ *wake up* window W_1 and, similarly, $\{s2 \xrightarrow{a} s4, s3 \xrightarrow{x} s4, s5 \xrightarrow{x} s0\}$ wake up window W_2. Note that there is always at least one active window, because every state of the original transition system must be covered. Furthermore, since every transition must be covered too, there is always at least one window that is active both before and after a transition occurs.

A. Wake-up markings

Table I provides an additional illustration of the correspondence between the states of the original transition system and the markings of the obtained nets W_1 and W_2. As one can see, each state is covered by at least one net marking. When

TABLE I: States and net markings in Fig. 1.

State	Marking of W_1	Marking of W_2
$s0$	$\{p0, p3\}$	$\{p7, p8\}$
$s1$	$\{p1, p4\}$	-
$s2$	$\{p1, p3\}$	-
$s3$	$\{p2, p4\}$	-
$s4$	$\{p2, p3\}$	$\{p6, p8\}$
$s5$	$\{p0, p4\}$	-
$s6$	-	$\{p5, p8\}$
$s7$	-	$\{p6, p9\}$
$s8$	-	$\{p7, p9\}$
$s9$	-	$\{p5, p9\}$

a window becomes inactive, the marking of the corresponding net has no meaning and is forgotten. When the window subsequently wakes up, the net must be initialised with a correct marking matching the current state of the system. Using Table I one can obtain the following *wake-up markings*:

- When W_1 wakes up in state $s0$, it must be initialised with the marking $\{p0, p3\}$.
- When W_1 wakes up in state $s4$, it must be initialised with the marking $\{p2, p3\}$.
- When W_2 wakes up in state $s0$, it must be initialised with the marking $\{p7, p8\}$.
- When W_2 wakes up in state $s4$, it must be initialised with the marking $\{p6, p8\}$.

One can remove all references to the original transition system and its states $s0$ and $s4$, thereby making the process windows based description self-contained from the point of view of the two nets, as follows:

- When W_1 wakes up, it must have a token in $p3$, plus a token in $p0$ (if $p7$ is marked in W_2) or $p2$ (if $p6$ is marked in W_2). All other places should be unmarked.
- When W_2 wakes up, it must have a token in $p8$, plus a token in $p7$ (if $p0$ is marked in W_1) or $p6$ (if $p2$ is marked in W_2). All other places should be unmarked.

The wake-up markings are shown in Fig. 1(d) below the nets.

B. Wake-up conditions

In addition to wake-up markings, we also derive *wake-up conditions*: the wake-up condition of a window evaluates to 1 in all states where the window can wake up and therefore requires an initialisation using the wake-up marking.

In our running example, both windows can wake up in states $s4$ and $s0$. From the point of view of W_1, it needs to wake up when W_2 has marking $\{p6, p8\}$ or $\{p7, p8\}$. This can be expressed by the Boolean condition $(p6 \vee p7) \wedge p8$, which can be simplified to $\overline{p5} \wedge p8$ using Boolean minimisation. Similarly, window W_2 needs to wake up when W_1 has marking $\{p0, p3\}$ or $\{p2, p3\}$, as captured by the condition $(p0 \vee p2) \wedge p3$ or, equivalently, $\overline{p1} \wedge p3$.

Fig. 1(d) shows minimised wake-up conditions below each net. A general method for deriving wake-up conditions and markings is presented in Section V.

III. Labeled Transition Systems and Windows

A. Labelled Transition Systems

A Labelled Transition System (LTS) is a tuple (S, Σ, T, s_0) where S is a finite set of states, Σ is the alphabet of labels, $T \in S \times \Sigma \times S$ is the set of labelled transitions and $s_0 \in S$ is the initial state.

We use $s \xrightarrow{e} s'$ to denote the labelled transition $(s, e, s') \in T$. An event $e \in \Sigma$ is said to be enabled in $s_1 \in S$ if there exists $s_1 \xrightarrow{e} s_2$ for some $s_2 \in S$.

Given an event $e \in \Sigma$, the *Enabling Set* of e is the set of states in which e is enabled, i.e.,

$$ES(e) = \{s \in S \mid \exists s' \in S : s \xrightarrow{e} s'\}.$$

Similarly, we define the *Backward Enabling Set* of e:

$$BES(e) = \{s \in S \mid \exists s' \in S : s' \xrightarrow{e} s\}.$$

B. Windows

Given an LTS $A = (S, \Sigma, T, s_0)$, a *window* of A is another LTS $W = (S_w \cup \{\bot_w\}, \Sigma_w, T_w, s_{0_w})$ such that

- $S_w \subseteq S$, $\Sigma_w \subseteq \Sigma$, and $T_w \subseteq T$. Moreover, Σ_w strictly contains the labels associated to T_w and S_w strictly contains the states associated to T_w.
- $\bot_w \notin S$ represents the inactive state.
- If $s_0 \in S_w$, then $s_{0_w} = s_0$, otherwise $s_{0_w} = \bot_w$.

Thus, any window is fully determined by a subset of transitions of the LTS.

C. Window Decomposition

Given an LTS $A = (S, \Sigma, T, s_0)$, a *Window Decomposition* (WD) of A is a set of LTS windows, $\{W_1, \ldots, W_n\}$, with $W_i = (S_i \cup \{\bot_i\}, \Sigma_i, T_i, s_{0_i})$, such that

$$S = \bigcup_i S_i, \qquad \Sigma = \bigcup_i \Sigma_i, \qquad T = \bigcup_i T_i$$

and all $\bot_i$ are different. The definition implies that every state and every transition of A is covered by at least one window. Additionally, the following conditions hold for every W_i:

- The underlying graph induced by T_i is connected.
- $T_i \not\subseteq T_j$ for any $j \neq i$.

Intuitively, the previous conditions guarantee that all components are *minimal*, i.e., that the states are connected (except $\bot_i$) and no window is redundant.

D. State space and transition steps

A window decomposition $W = \{W_1, \ldots, W_n\}$ is a set of windows that evolve synchronously at every transition step according to the transitions of the LTS they represent. W has a global state space in which each state $\vec{s}$ is a vector of state components, $\vec{s} = (s_1, \ldots, s_n)$, each one belonging to a different window. There is a one-to-one correspondence between the states of W and the states of the associated LTS. For every state $s_x \in S$, the associated state in W will be $\vec{s_x} = (s_1, \ldots, s_n)$ such that for every W_i:

$$s_i = s_x \quad \text{if } s_x \in S_i, \qquad s_i = \bot_i \quad \text{if } s_x \notin S_i$$

i.e., all the active states are identical.

1) Example: If we consider the LTS in Fig. 1(a) and the WD with two windows, W_1 (left) and W_2 (right), in Fig. 1(b), we observe that the initial state s_0 is represented by the vector $\vec{s_0} = (s_0, s_0)$. When firing transition $s_0 \xrightarrow{c} s_6$, the WD moves to state $\vec{s_6} = (\bot_1, s_6)$. We next describe a possible trace of the WD:

$$(s_0, s_0) \xrightarrow{c} (\bot_1, s_6) \xrightarrow{q} (\bot_1, s_7) \xrightarrow{y} (s_4, s_4)$$
$$\xrightarrow{b} (s_0, s_0) \xrightarrow{p} (s_1, \bot_2) \ \cdots$$

As one can see from the above trace, at every state each window can be either active or inactive. Some transitions may deactivate a window, e.g., $(s_0, s_0) \xrightarrow{p} (s_1, \bot_2)$, whereas other transitions may activate (wake up) a window, e.g., $(\bot_1, s_7) \xrightarrow{y} (s_4, s_4)$. There is always at least one active window that keeps track of the current state so that others can wake up and appropriately initialise themselves when their time comes.

E. Structural properties

From the theory of Petri nets, we know that certain subclasses, such as Marked Graphs or Free-Choice Petri nets, are well suited for good-looking visual structures [3]. Additionally, these structures are also desirable for performance and reachability analysis.

An interesting contribution in this area is the work by Best and Devillers characterising the state spaces of behaviours that can be generated by Marked Graphs [4] and choice-free Petri nets [5]. Free-choice Petri nets is another subclass with visually-friendly structural properties [6]. The synthesis of this subclass has been proposed in [7] by combining the theory regions and label splitting to force all choices to be "free".

In this work, we apply the results from [5] and [7] for the decomposition of behaviours into process windows. One of our main goals is to create a framework in which the analysis of processes can be highly automated by providing algorithms to extract windows. The proposed algorithms are based on solving a SAT formulation of the problem that encodes *local*[1] properties of the windows.

We next present local properties presented in [5][7] to enforce structural properties on the synthesised Petri nets. The first two properties (forward and backward persistence) are necessary, but not sufficient, for the synthesis of choice-free Petri nets. The third property is necessary for the synthesis of Free-Choice Petri nets.

1) Forward and backward persistence: Given an LTS, two events a and b are said to be *forward persistent* if the following condition holds:

$$\forall s_1 \in \mathit{ES}(a) \cap \mathit{ES}(b),\ \text{s.t. } s_1 \xrightarrow{a} s_2\ \wedge\ s_1 \xrightarrow{b} s_3 : \\ s_2 \in \mathit{ES}(b)\ \wedge\ s_3 \in \mathit{ES}(a).$$

This condition guarantees that a does not disable b, and vice versa. Note that forward persistence always holds when $\mathit{ES}(a) \cap \mathit{ES}(b) = \emptyset$.

A dual property is defined for backward persistence. Two events a and b are said to be *backward persistent* if the following condition holds:

$$\forall s_1 \in \mathit{BES}(a) \cap \mathit{BES}(b),\ \text{s.t. } s_2 \xrightarrow{a} s_1\ \wedge\ s_3 \xrightarrow{b} s_1 : \\ s_2 \in \mathit{BES}(b)\ \wedge\ s_3 \in \mathit{BES}(a).$$

An LTS is said to be forward (backward) persistent if forward (backward) persistence holds for all pairs of events.

2) Free choiceness: When two events are not forward persistent, then a conflict (choice) occurs between them. In this case, we may desire the conflict to be free, i.e., the choice conditions to be symmetric for both of them.

Let a and b be two events that are not forward persistent. Then, a and b are said to be in *forward free choice* if $\mathit{ES}(a) = \mathit{ES}(b)$. Similarly, if a and b are not backward persistent, they are said to be in *backward free choice* if $\mathit{BES}(a) = \mathit{BES}(b)$.

[1]By local we refer to properties that can be formulated in terms of states or transitions and a small neighbourhood around them.

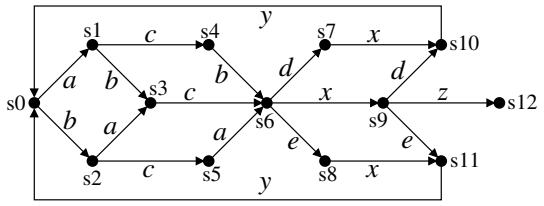


Fig. 3: LTS with different persistence and choice properties.

3) Example: Fig. 3 depicts an LTS with different properties between pairs of events. For example, the pair (a, b) is forward persistent, but not backward persistent, since a and b are both backward enabled in $s6$, but a is not backward enabled in $s4$ and b is not backward enabled in $s5$. The pairs (a, c) and (b, c) are both forward and backward persistent.

The pairs (d, x) and (e, x) are also forward and backward persistent. The pair (d, e) is not forward persistent, but it is in forward free choice. Finally, the pair (d, z) is not in forward free choice, since they are not forward persistent and z is not enabled in $s6$. Similarly for the pair (e, z).

IV. Algorithm for Window Decomposition

This section describes an algorithmic method for discovering a window decomposition of an LTS. The method is inspired by the one presented in [3] for the extraction of LTS slices during process mining. It is based on a SAT formulation of the problem, using $|T|$ variables, that models all possible windows that conform to a certain set of properties. The problem is solved via Pseudo-Boolean Optimisation [8].

With an abuse of notation, we define a Boolean variable t for each transition $t \in T$. A window is fully determined by a subset of transitions, that can be modeled as a Boolean assignment to the corresponding variables. All those variables asserted in the model correspond to the selected transitions for the window.

The formula $W(T)$ that models all possible windows that can be extracted from an LTS is defined as:

$$\textsc{Window}(T) = P_1(T)\ \wedge\ \cdots\ \wedge\ P_n(T) \qquad (1)$$

where each $P_i(T)$ represents a set of constraints associated to a property.

Next, the constraints associated to different properties are described. A common and important feature of these properties is that they are local, i.e., they can be specified as Boolean relationships among variables that represent transitions in a small region of the LTS.

A. Forward and backward persistence

Forward persistence ensures that no event disables another event in the extracted window (see Sect. III-E). The Boolean formulation of this property is as follows:

> Let $s, s_1, s_2 \in S$ and $a, b \in \Sigma$, with $a \neq b$, such that $t_1 = (s, a, s_1)$ and $t_2 = (s, b, s_2)$. Let $T_{s_2,a} = \{t_i = (s_2, a, s_i) \mid s_i \in S\}$ be the set of transitions enabled in s_2 with event a. Then the

following constraint is added to the formula to guarantee the persistence of a:

$$(t_1 \wedge t_2) \implies (t_{i_1} \vee \cdots \vee t_{i_k})$$

where $t_{i_1}, \ldots, t_{i_k}$ are the elements of $T_{s_2,a}$. Notice that, by symmetry, this constraint will also be applied for b's persistence. It also works for non-deterministic LTSs in which $|T_{s_2,a}| > 1$. In case $T_{s_2,a} = \emptyset$, the constraint is reduced to: $\neg t_1 \vee \neg t_2$.

Backward persistence has a dual formulation that intuitively corresponds to the one for forward persistence when the direction of the transitions is reversed.

The constraints for forward and backward persistence must be formulated for pairs of events enabled in any state of the LTS.

B. Determinism

In some cases, it is desirable that the extracted windows are deterministic. This implies that any non-deterministic choice enforces the system to move to a different window. Determinism can be easily enforced by adding *"at-most-one"* constraints over all the transitions enabled with the same event in non-deterministic states.

C. Connectedness

This is a property that cannot be guaranteed with only local constraints. Instead, a hybrid approach combining Boolean constraints and algorithmic post-processing is used.

The Boolean constraints guarantee that no new source/deadlock states are generated in a window. Formally, for any state $s \in S$, we define $T_{\text{in}}(s) = \{t_{\text{in}_1}, \ldots, t_{\text{in}_m}\}$ and $T_{\text{out}}(s) = \{t_{\text{out}_1}, \ldots, t_{\text{out}_n}\}$ as the set of incoming and outgoing transitions of s, respectively. For any state in which $T_{\text{in}}(s) \neq \emptyset$ and $T_{\text{out}}(s) \neq \emptyset$, the following constraint is added:

$$(t_{\text{in}_1} \vee \cdots \vee t_{\text{in}_m}) \iff (t_{\text{out}_1} \vee \cdots \vee t_{\text{out}_n}).$$

This constraint guarantees that any state with incoming and outgoing transitions in the original LTS will also have incoming and outgoing transitions in any window.

D. Algorithm

We next present an algorithm for the discovery of a window decomposition of an LTS (see Algorithm 1).

The algorithm is based on the sequential extraction of process windows that fulfil a set of properties. The extraction is shepherded by a SAT formula that encodes the constraints of the chosen properties (see line 2 and equation (1)).

The variable T' contains the set of transitions that have not yet been covered by the previously extracted windows.

For each window, the maximum number of uncovered transitions is selected (besides other transitions). This selection is steered by a cost function encoded as a Pseudo-Boolean constraint (line 6). The variable T_i is the set of selected transitions.

It is theoretically possible that T_i has more than one connected component. In that case, the largest component is selected to create a new process window W_i. The initial state can be either s_0 or $\perp_i$ depending on whether s_0 belongs to one of the selected transitions.

The main loop of the algorithm iterates until all the transitions of the LTS have been covered by some window.

Algorithm 1 Generation of a Window Decomposition

```
1:  function WindowDecomposition((S, Σ, T, s_0))
2:      F ← SAT formula (1) for property encoding
3:      T' = T            ▷ T' contains the uncovered transitions
4:      i ← 1                                  ▷ Window index
5:      while T' ≠ ∅ do                ▷ while uncovered transitions
6:          Cost ← Σ_{t∈T'} t          ▷ max uncovered transitions
7:          T_i ← PseudoBooleanOptimization(F, Cost)
8:          T_i ← LargestConnectedComponent(T_i)
9:          Σ_i ← The alphabet associated to T_i
10:         S_i ← States from S adjacent to T_i
11:         s_ini ← s_0 ∈ S_i ? s_0 : ⊥_i
12:         W_i = LTS(S_i, Σ_i, T_i, s_ini)
13:         T' ← T' \ T_i
14:         i ← i + 1
15:     return {W_1, ..., W_n}                    ▷ The WD
```

E. Implementation details

The current algorithm has been implemented to discover window decompositions with forward and backward persistence that can later lead to choice-free Petri nets. To avoid an excessive fragmentation of the selected transitions into multiple connected components, the connectedness condition discussed in Sect. IV-C has been added to formula F.

The PBLib library [9] has been used for Pseudo-Boolean optimisation and Minisat [10] as SAT solver. The maximisation of the cost function has been implemented by incrementally strengthening the constraints that encode the cost function until the problem becomes unsatisfiable [9].

F. Termination

Let us call *simple* traces those that correspond to simple paths in the LTS, i.e., those that do not visit the same vertex more than once. Termination is always guaranteed by considering the following observation: any simple trace is a window with forward and backward persistence.

Therefore, if $T' \neq \emptyset$, it is always possible to find a window consisting of a simple trace that covers at least one of the transitions in T' and fulfils persistence. Hence, T' is reduced at each iteration.

The previous observation also provides an upper bound on the number of iterations required to discover a WD: the number of simple traces cannot be larger than the minimum number of traces required to cover all transitions of the LTS.

In practice, the number of iterations is substantially smaller since every window covers multiple traces exhibiting concurrent behaviours.

V. Deriving wake-up conditions and markings

In this section we present a method for deriving Boolean equations for wake-up conditions and markings. The method is integrated with the window decomposition algorithm presented in Section IV and is available as part of the WORKCRAFT framework [1].

A. Deriving wake-up conditions

Let W be a window covering the set of states S_w. The truth table of its wake-up condition $c(W, s)$ can be specified as follows:

- $c(W, s) = 0$ for all states s that are outside the window, that is:
$$\forall s \notin S_w : c(W, s) = 0.$$
- $c(W, s) = 1$ for all states s where we can enter the window, that is:
$$\forall s \in S_w : (\exists s' \notin S_w : s' \xrightarrow{e} s) \Rightarrow c(W, s) = 1.$$
- Otherwise, for all states s that are covered by the window but cannot be entered from outside, the value $c(W, s)$ is unconstrained, i.e. it is a *don't care*. Indeed, there is no harm if the wake-up condition is true when the window is already active, and we can use this flexibility to obtain a simpler equation for the wake-up condition.

For the example discussed in Section II-B, the above definition yields the following constraints for W_1:

$$\begin{cases} c(W_1, s6) = c(W_1, s7) = c(W_1, s8) = c(W_1, s9) = 0, \\ c(W_1, s0) = c(W_1, s4) = 1. \end{cases}$$

Since the windows are represented by safe Petri nets, it is natural to represent their states by sets of marked places. This leads to the following standard Boolean logic synthesis problem that can be solved by the ESPRESSO tool [11], where sets are encoded by Boolean vectors:

Marking s	Boolean vector	Wake-up condition $c(W_1, s)$
$\{p5, p8\}$	$(1, 0, 0, 1, 0)$	0
$\{p6, p9\}$	$(0, 1, 0, 0, 1)$	0
$\{p7, p9\}$	$(0, 0, 1, 0, 1)$	0
$\{p5, p9\}$	$(1, 0, 0, 0, 1)$	0
$\{p7, p8\}$	$(0, 0, 1, 1, 0)$	1
$\{p6, p8\}$	$(0, 1, 0, 1, 0)$	1

By synthesising the above into a Boolean equation, one obtains a very succinct wake-up condition: $c(W_1, s) = \overline{p5} \wedge p8$.

It is not difficult to prove that an equation with only positive literals can always be obtained for wake-up conditions. This comes from the fact that the activation of a window always coincides with the arrival of a token in some other windows. This *monotonic* behaviour guarantees a positive relationship between a set of variables and the wake-up conditions. The details of the proof are out of the scope of this paper.

We implemented the *positive mode* in our tool to derive wake-up conditions without negative literals, which produces $c(W_1, s)_{pos} = (p6 \wedge p8) \vee (p7 \wedge p8)$ in this case. One can see that the obtained equation may be simplified by factoring out the common term $p8$. Our tool can derive such factored equations if requested by the user. In our example the result is $c(W_1, s)^{opt}_{pos} = (p6 \vee p7) \wedge p8$, as expected.

B. Deriving wake-up markings

To derive wake-up markings we use a similar approach. Let $m(W, p, s)$ stand for the wake-up marking of place p in window W and state s. Its truth table is as follows:

- $m(W, p, s) = 1$ if we can enter W in state s and p must be marked.
- $m(W, p, s) = 0$ if we can enter W in state s and p must be unmarked.
- Otherwise, the value $m(W, p, s)$ is unconstrained. Indeed, we only care about the value in the states when we enter window W and wake it up.

For the example discussed in Section II-A, the above definition yields the following constraints for W_1 and $p0$:

$$\begin{cases} m(W_1, p0, s0) = 1, \\ m(W_1, p0, s4) = 0. \end{cases}$$

This can be equivalently expressed by the following truth table:

Marking s	Boolean vector	Wake-up marking $m(W_1, p0, s)$
$\{p7, p8\}$	$(0, 0, 1, 1, 0)$	1
$\{p6, p8\}$	$(0, 1, 0, 1, 0)$	0

It is not difficult to see that the truth table can be synthesised into a very simple equation $m(W_1, p0, s) = p7$.

VI. Applications

Process windows are useful whenever one needs to understand the behaviour of a complex system where concurrency and choice are intertwined. Such systems often make commonly used behavioural models, such as Petri nets, difficult to comprehend, as we have already demonstrated in the motivational example in Section I.

In this section we discuss two application areas where process windows are particularly useful: asynchronous circuit design and process mining.

A. Asynchronous circuit design

Asynchronous circuits operate without a global clock signal, and individual gates can therefore fire truly concurrently. Design of such circuits is a very challenging task, and even circuits with a few gates may require a substantial specification and analysis effort from the designer.

Signal Transition Graphs (STGs) are commonly used as the specification language in asynchronous circuit design [12]. STGs are Petri nets whose transitions are labelled with *signal transitions*, i.e. events corresponding to signals changing their value from 0 to 1 (the *rising* transition, denoted by a^+ for signal a) and 1 to 0 (the *falling* transition, denoted by a^-). The key benefit of using STGs compared to transition systems is that STGs can represent concurrency very compactly using the true concurrency semantics, which is very natural for asynchronous circuits. Transition systems, on the other hand, use the interleaving semantics for representing concurrency, which leads to inadequately large models.

Despite the fact that STG models of asynchronous circuits are often compact, they may still be hard to understand, particularly by users who are not experts in the concurrency theory, such as industrial hardware engineers. Process windows help

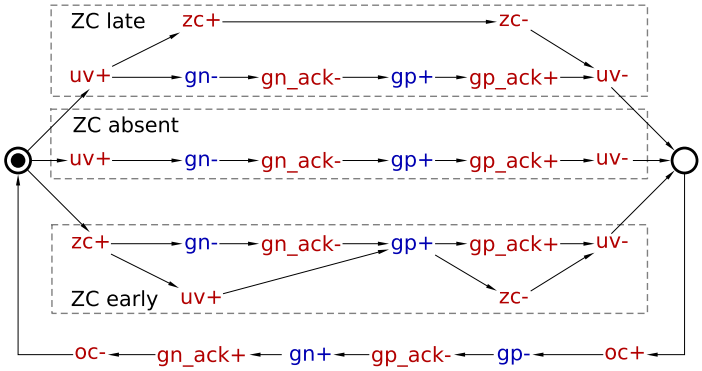


Fig. 4: STG specification of a basic buck controller [13].

make STG models easier to understand by abstracting away the choices in the system, and representing each behavioural scenario separately.

Consider an example STG shown in Fig. 4. The STG specifies the behaviour of a basic asynchronous buck controller used in on-chip power regulators [13]. The STG is a result of a careful analysis of the controller by the designer, who manually extracted three behavioural scenarios and represented them as separate branches of the STG in order to achieve clear representation of the controller's behaviour. Note that the STG is not entirely trivial: the two upper branches start with the same signal transition uv^+, which indicates that the scenarios partially overlap.

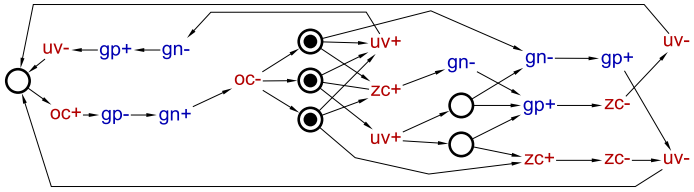
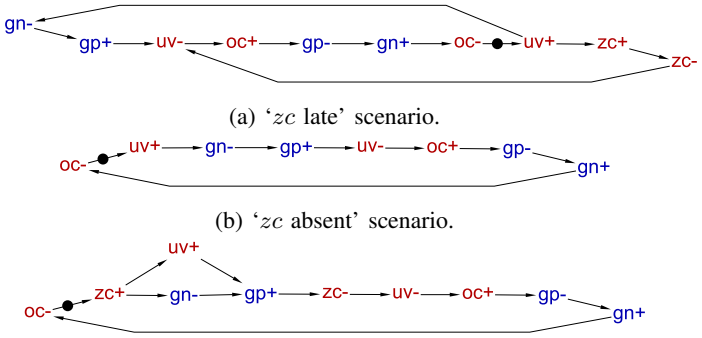


Fig. 5: Buck STG synthesised from the transition system.

To illustrate that deriving the STG specification in Fig. 4 manually is not trivial, Fig. 5 shows the STG synthesised automatically from the transition system of the buck controller by PETRIFY [14]. As one can see, once the careful manual layout based on the insight into behavioural scenarios of the system is removed, the STG becomes harder to understand.

The window decomposition method presented in this paper can automatically discover the three scenarios from the underlying transition system, without any manual intervention from the designer, thereby substantially decreasing the specification and visualisation effort. The extracted windows are shown in Fig. 6. They have have the same wake-up marking with the only token on the $oc^- \rightarrow uv^+$ arc, and symmetric wake-up conditions that monitor these places. The aspect of overlapping scenarios is even more evident in the window-based representation: the marked graph corresponding to the 'zc absent' scenario is a subgraph of the 'zc late' scenario, which means that whenever the former window is active, the latter is active too.

Representing circuit behaviour by a collection of marked graphs can be also beneficial for the following reasons:



(a) 'zc late' scenario.

(b) 'zc absent' scenario.

(c) 'zc early' scenario.

Fig. 6: Fully automated window decomposition of the buck controller (layout generated by Graphviz [15]).

- Marked graphs are a subclass of Petri nets with particularly well understood structural properties. For example, one can easily characterise marked graphs in terms of performance and energy. If the proportion of time the circuit operates in each scenario is known from the system description, it is possible to aggregate individual characteristics of the scenarios, obtaining a metric for overall system performance and energy consumption.
- Process windows permit incremental specification of a system, where each scenario is designed and verified separately as a marked graph. Such incremental specification allows to avoid monolithic STG specifications that cannot be designed in a decentralised manner by independent teams of designers.
- It is possible to synthesise asynchronous circuits for each marked graph separately and then combine them using a correct-by-construction approach on the basis of wake-up conditions and markings. This can potentially produce circuits that are easier to implement using gates available in standard technology libraries.

B. Process mining

Another potential application area is *process mining* [16], more specifically the discovery of process models from execution traces. Some of the previous work in this area [3] inspired the concept of process windows presented in this paper.

Traces:

```
adeac
acdbeacb
abceac
acbdecab
adeca
acbeca
```

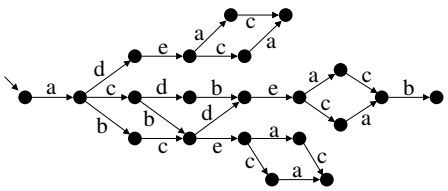
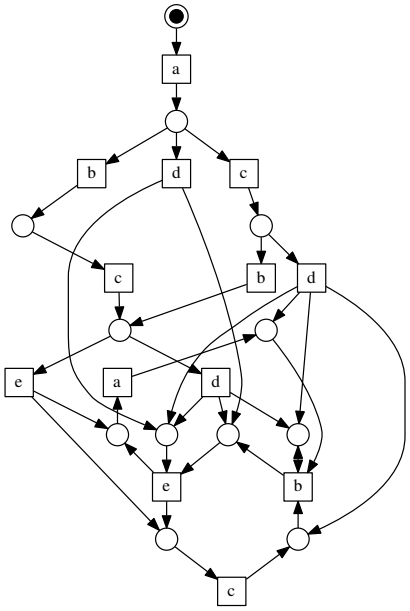
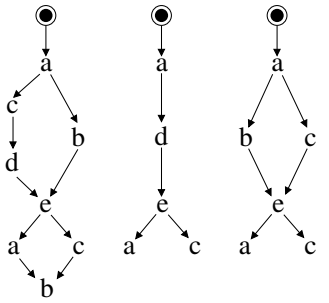


Fig. 7: Log of traces and Labelled Transition System.

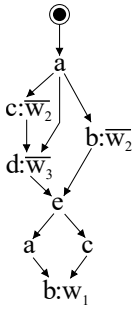
We illustrate the applicability of process windows to the area of process mining with a simple example in Fig. 7. On the left, a *log* of traces is shown with events belonging to the alphabet $\{a, b, c, d, e\}$. On the right, the LTS obtained from



(a) Petri net model automatically discovered by PETRIFY.



(b) Process windows $\{W_1, W_2, W_3\}$.



(c) Window overlay.

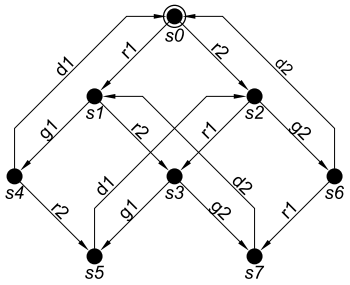
Fig. 8: Mining a process model for the log in Fig 7.

the traces is depicted. This LTS has been generated with the *prefix multiset* conversion [17], in which prefixes with the same number of occurrences of each event lead to the same state (regardless of the occurrence order).

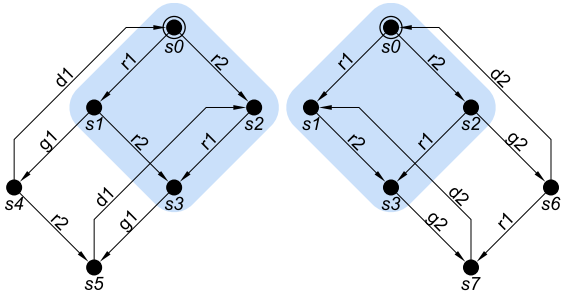
Fig. 8(a) depicts the Petri net obtained by PETRIFY [14] using the theory of regions and label splitting. The structure of the Petri net is very intricate and gives no intuition on how the process behaves. The main reason is because the net represents multiple behaviours of different nature under the same structure. By extracting choice-free process windows shown in Fig. 8(b), the structure of the behaviour becomes much more visible.

In many cases, the logs of a system contain behaviours of subsystems that interact with different causality/concurrency relations. Process windows contribute to *distill* the hidden sub-processes and show the variety of behaviours that were artificially blended in the same log.

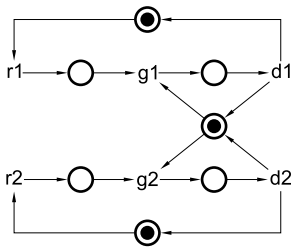
Furthermore, it is possible to combine process windows with other techniques for visualising scenarios. For example, Fig. 8(c) shows how the windows can be overlaid by matching their common fragments and using Boolean conditions to deactivate individual events. By setting $w_1w_2w_3 = 010$, i.e. choosing window W_2, one can remove events b and c whose conditions evaluate to 0, and obtain the second scenario. Such compact overlaid representations can be automatically produced by existing process mining techniques [18].



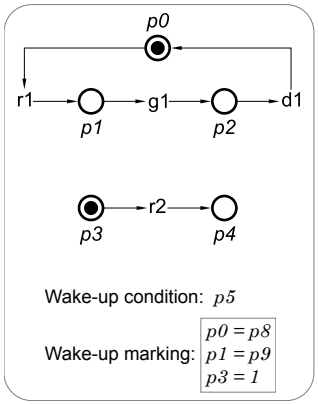
(a) Transition system.



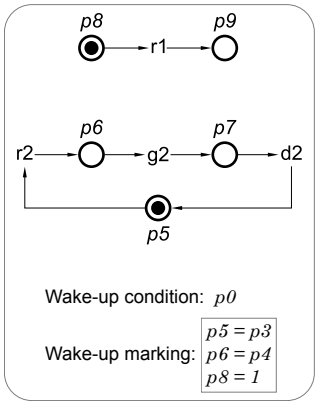
(b) Window decomposition (shared states highlighted).



(c) Petri net.



(d) Window W_1.

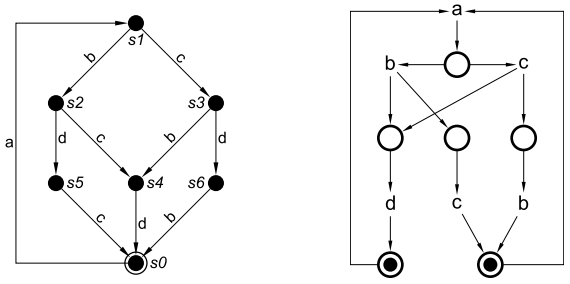


(e) Window W_2.

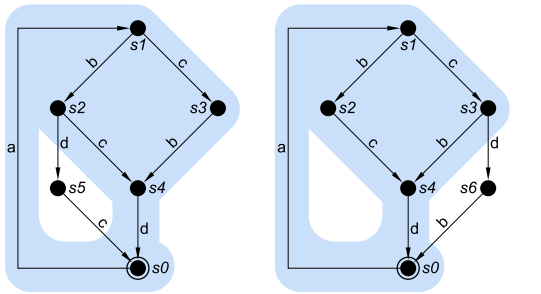
Fig. 9: Arbitration.

VII. DISCUSSION

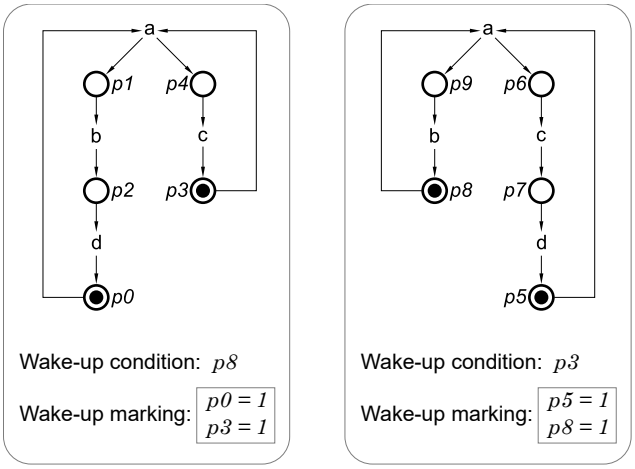
In this section we study two more examples of window decomposition, that highlight the aspects of scalability and flexibility of the proposed approach when it is applied to real-life systems that often contain such sources of complexity as resource arbitration and OR causality.



(a) Transition system. (b) Petri net.



(c) Window decomposition (shared states highlighted).



(d) Window W_1. (e) Window W_2.

Fig. 10: OR causality.

A. Resource arbitration

Fig. 9(a) shows a transition system of a *request-grant-done arbiter*. The arbiter accepts two concurrent requests $r1$ and $r2$ and issues at most one grant $g1$ or $g2$ for a pending request. The request is subsequently removed by the corresponding *done* event $d1$ or $d2$. The corresponding Petri net in Fig. 9(c) is well-known and captures the behaviour of the system in a very clear and concise way. Furthermore, it is a *scalable* representation in the following sense. If one needs to generalise the model to describe a k-of-n arbiter for handling n requests by issuing at most k grants at a time, one can simply use n request-grant-done loops arbitrating via a shared resource place initially containing k tokens.

Now consider the window decomposition shown in Fig. 9(b) and the windows in Fig. 9(d,e). The windows are choice-free, as desired, however, this representation is not scalable. Indeed, a k-of-n arbiter requires an exponential number of windows $\binom{n}{k}$, because there are exactly $\binom{n}{k}$ different choice-free scenarios. This shows the limitation of the proposed approach when windows are restricted to choice-free nets. A possible solution to this problem is to overlay windows, as described in Section VI-B, which results in a compact representation even for exponentially many windows. Alternatively, one can adapt the window decomposition algorithm to allow simple resource arbitration patterns within windows.

B. OR causality

OR causality [19] is known to be difficult to model with Petri nets, as it requires either 2-safe places or event splitting. Fig. 10 shows an example of a system with OR causality: the event d may be caused either by b or by c. The Petri net in Fig. 10(b) splits event b and c to model OR causality, which makes the net difficult to understand and introduces an aspect of choice into the model, even though the original transition system is persistent.

Fig. 10(c) shows how the transition system can be decomposed into a union of two backward-persistent transition systems. The resulting window decomposition is shown in Fig. 10(d); it contains two windows because the original transition system is not backward persistent.

Our implementation of the window decomposition algorithm allows the user to choose whether to respect backward persistence during the decomposition process or not, because in some situations it may be beneficial to discover windows that contain OR causality, as they may correspond to natural behavioural scenarios of the system.

VIII. Related work

Process windows are related to and inspired by a broad body of work on scenario-based system specification and analysis methods [20]. In particular, *Message-Sequence Charts* (MSCs) [21] and *Live Sequence Charts* (LSCs) [22] are widely used for the specification of protocols of the communication between system components by means of messages. MSCs and LSCs are supported by tools and can be automatically transformed to transition systems for further model-checking and synthesis.

Some approaches use Petri net based models to represent individual scenarios, for example *oclets* [23] use partial runs to represent scenarios and *anti-scenarios* (i.e. scenarios that must not occur). *Untanglings* [24] also represent a system behaviour by a collection of acyclic partial runs, but for the purpose of efficient state representation and model-checking, instead of specification. *Structured Occurrence Nets* [25] introduce a family of richer relations between scenarios, such as behavioural abstraction.

In the area of asynchronous circuit design, one can also highlight the work on *Conditional Partial Order Graphs* [26] and *Parameterised Graphs* [27], where partial orders are used for the specification of multi-scenario hardware systems, such as processors and on-chip communication controllers.

The key differentiating aspect of this paper is the *automated discovery of scenarios* from transition systems, directly inspired by [3], which allows to understand the behaviour

of complex existing systems that have not been manually decomposed into scenarios. Furthermore, unlike many of the above-mentioned approaches, the proposed method is not limited to acyclic scenarios and can seamlessly handle both cyclic and acyclic scenarios, choosing the most appropriate formalism for their representation, as has been demonstrated in Section VI.

The main difference from the approach presented [3] is the coverability of behaviours. In [3], the extraction of *slices* was oriented to discovering process models obtained from logs. In that case, each trace of the log was completely covered by at least one slice. The work presented in this paper is more general in the sense that windows cover all behaviours of the LTS but traces may travel across different windows.

IX. Conclusions

Complex processes often have intricate relationships among the participating events. Having a monolithic model to represent such behaviours often results in structures that do not give a clear intuition on the sub-processes hidden inside the complexity of the global process.

Process windows is a formalism that helps distilling and discovering sub-processes with structurally simple relationships. Every window models a partial behaviour of the complete system and has simple properties that contribute to better understanding of the interaction among events.

An important aspect of the process windows method is that it is not restricted to any particular set of properties of discovered windows. In this paper we have explored persistent windows to obtain choice-free scenarios. However, one can envisage other window properties to be used to derive windows with other useful features.

Automation is another important aspect of process windows. If the desired properties are simple and local, efficient algorithms for the discovery of windows can be derived, thereby making the interaction with the user simple, practical and interactive.

References

[1] The Workcraft framework homepage. http://www.workcraft.org/, 2009.

[2] I. Poliakov, D. Sokolov, and A. Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency*, pages 505–514. 2007.

[3] Javier de San Pedro and Jordi Cortadella. Mining structured Petri nets for the visualization of process behavior. In *31st ACM Symposium on Applied Computing*, pages 839–846, April 2016.

[4] Eike Best and Raymond Devillers. Characterisation of the state spaces of live and bounded marked graph Petri Nets. In *Language and Automata Theory and Applications*, volume 8370 of *LNCS*, pages 161–172. Springer, 2014.

[5] Eike Best and Raymond Devillers. Synthesis of bounded choice-free Petri nets. In *Proc. 26th International Conference on Concurrency Theory (CONCUR)*, pages 128–141, 2015.

[6] Jörg Desel and Javier Esparza. *Free Choice Petri nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[7] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. Deriving Petri nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.

[8] Endre Boros and Peter L. Hammer. Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1–3):155–225, 2002.

[9] Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-boolean constraints into CNF. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, 2015.

[10] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 502–518, 2003.

[11] Richard L. Rudell and Alberto L. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[12] Alexandre V Yakovlev, Albert M Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.

[13] D. Sokolov, A. Mokhov, A. Yakovlev, and D. Lloyd. Towards asynchronous power management. In *IEEE Faible Tension Faible Consommation (FTFC)*, pages 1–4, May 2014.

[14] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, 80(3):315–325, 1997.

[15] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993.

[16] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 1st edition, 2011.

[17] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[18] Andrey Mokhov, Josep Carmona, and Jonathan Beaumont. Mining Conditional Partial Order Graphs from Event Logs. In *Transactions on Petri Nets and Other Models of Concurrency XI*, pages 114–136. Springer Berlin Heidelberg, 2016.

[19] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, pages 189–234, 1996.

[20] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. Scenarios in system development: current practice. *IEEE software*, 15(2):34–45, 1998.

[21] David Harel and PS Thiagarajan. Message sequence charts. In *UML for Real*, pages 77–105. Springer, 2003.

[22] David Harel and Rami Marelly. *Come, let's play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.

[23] Dirk Fahland. Oclets–scenario-based modeling with petri nets. In *International Conference on Applications and Theory of Petri Nets*, pages 223–242. Springer, 2009.

[24] Artem Polyvyanyy, Marcello La Rosa, Chun Ouyang, H Arthur, and M Ter Hofstede. Untanglings: a novel approach to analyzing concurrent systems. *Formal Aspects of Computing*, 27(5-6):753, 2015.

[25] Maciej Koutny and Brian Randell. Structured occurrence nets: A formalism for aiding system failure prevention and analysis techniques. *Fundamenta Informaticae*, 97(1-2):41–91, 2009.

[26] Andrey Mokhov and Alex Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[27] Andrey Mokhov and Victor Khomenko. Algebra of parameterised graphs. *ACM Transactions on Embedded Computing Systems*, 13(4s), 2014.