Ensuring Correctness During Process Configuration via Partner Synthesis

Wil M.P. van der Aalst[a,c,*], Niels Lohmann[b], Marcello La Rosa[c]

[a] *Eindhoven University of Technology, The Netherlands*
[b] *Universität Rostock, Germany*
[c] *Queensland University of Technology, Australia*

Abstract

Variants of the same process can be encountered within one organization or across different organizations. For example, different municipalities, courts, and rental agencies all need to support highly similar processes. In fact, procurement and sales processes can be found in almost any organization. However, despite these similarities, there is also the need to allow for local variations in a controlled manner. Therefore, many academics and practitioners have advocated the use of *configurable process models* (sometimes referred to as reference models). A configurable process model describes a family of similar process models in a given domain. Such a model can be configured to obtain a *specific* process model that is subsequently used to handle individual cases, for instance, to process customer orders. *Process configuration is notoriously difficult as there may be all kinds of interdependencies between configuration decisions.* In fact, an incorrect configuration may lead to behavioral issues such as deadlocks and livelocks. To address this problem, we present a novel verification approach inspired by the "operating guidelines" used for partner synthesis. We view the configuration process as an external service, and compute a characterization of all such services which meet particular requirements via the notion of *configuration guideline*. As a result, we can characterize all feasible configurations (i. e., configurations without behavioral problems) at design time, instead of repeatedly checking each individual configuration while configuring a process model.

Keywords: Configurable process model, operating guideline, Petri net, C-YAWL

1. Introduction

Although large organizations support their processes using a wide variety of process-aware information systems, the majority of business processes are

[*]Corresponding author

Email addresses: `w.m.p.v.d.aalst@tue.nl` (Wil M.P. van der Aalst), `niels.lohmann@uni-rostock.de` (Niels Lohmann), `m.larosa@qut.edu.au` (Marcello La Rosa) still not directly driven by process models [1]. Despite the success of Business Process Management (BPM) thinking in organizations, Workflow Management (WfM) systems — today often referred to as *BPM systems* — are not widely used. One of the main problems of BPM technology is the "lack of content"; that is, providing just a generic infrastructure to build process-aware information systems is insufficient as organizations need to support specific processes. Organizations want to have "out-of-the-box" support for standard processes and are only willing to design and develop system support for organization-specific processes. Yet most BPM systems expect users to model basic processes from scratch. Enterprise Resource Planning (ERP) systems such as SAP and Oracle, on the other hand, focus on the support of these common processes. Although all ERP systems have workflow engines comparable to the engines of BPM systems, lion's share of processes supported by these systems are not driven by models. For example, most of SAP's functionality is not grounded in their workflow component, but hard-coded in application software. ERP vendors try to capture "best practices" in dedicated applications designed for a particular purpose. Such systems can be configured by setting parameters. System configuration can be a time consuming and complex process. Moreover, configuration parameters are exposed as "switches in the application software", thus making it difficult to see the intricate dependencies among certain settings.

A model-driven process-oriented approach toward supporting business processes has all kinds of benefits ranging from improved analysis possibilities (verification, simulation, etc.) and better insights, to maintainability and ability to rapidly develop organization-specific solutions [1, 2]. Although obvious, this approach has not been adopted thus far, because BPM vendors have failed to provide content and ERP vendors suffer from the "Law of the handicap of a head start". ERP vendors manage to effectively build data-centric solutions to support particular tasks. However, the complexity and large installed base of their products makes it hard to refactor their software and make it process-centric.

Based on the limitations of existing BPM and ERP systems, we propose to use *configurable process models*. A configurable process model represents a *family of process models*; that is, a model that through configuration can be customized for a particular setting. By developing comprehensive collections of configurable models, one could capture all possible process variations in a particular domain. For example, one could create a configurable process model for logistics, one for insurance, etc. From the viewpoint of ERP software, configurable process models can be seen as a means to make these systems more process-centric, although quite some refactoring would be needed as processes are hidden in table structures and application code.

Various configurable languages have been proposed as extensions of existing languages (e. g., C-EPCs [3], C-iEPCs [4], C-WF-nets [5], C-SAP and C-BPEL [6]) but few are actually supported by enactment software (e. g., C-YAWL [6]). In this paper, we are interested in models in the latter class of languages, which, unlike traditional reference models [7–10], are executable after they have been configured. Specifically, we focus on the *verification of configurable executable process models*. In fact, because of configuring a process model, the resulting configured model may suffer from behavioral anomalies such as deadlocks and livelocks. This problem is exacerbated by the total number of possible configurations a model may have, which may be very large, and by the complex domain and data dependencies which may exist between various configuration options. Checking the *feasibility* of each single configuration can be time consuming as this would typically require performing state-space analysis. Moreover, characterizing the "family of correct models" for a particular configurable process model is even more difficult and time-consuming as a naive approach would require solving an exponential number of state-space problems, which is unfeasible for large real-life models.

As far as we know, our earlier approach [5] is the only one focusing on the verification of configurable process models which takes into account behavioral correctness and avoids the state-space explosion problem. Other approaches either only discuss syntactical correctness related to configuration [3, 11, 12], or deal with behavioral correctness but run into the state-space problem [13, 14]. In this paper, we propose a novel verification approach where we consider the configuration process as an "external service" and then synthesize a "most permissive partner" using the approach described by Wolf [15] and implemented in the tool Wendy [16]. This most permissive partner is closely linked to the notion of *operating guidelines* for service behavior [17]. In this paper, we define for any configurable model a so-called *configuration guideline* to characterize all correct process configurations. This approach provides the following advantages over our previous approach [5]:

- We provide a *complete characterization of all possible (correct) configurations at design time*; that is, the *configuration guideline*.
- Computation time is moved *from configuration time to design time* and results can be reused more easily.
- *No restrictions are put on the class of models* which can be analyzed. The previous approach [5] was limited to sound free-choice WF-nets. Our new approach can be applied to models which do not need to be sound, which can have complex (non-free choice) dependencies, and which can have multiple end states.

To prove the practical feasibility of this new approach, we have implemented it as a component of the toolset supporting C-YAWL.

The remainder of this paper is organized as follows. In Section 2 we elaborate on the need for process configuration and define the problem in a language independent manner. Section 3 introduces basic concepts such as open nets and weak termination. These concepts are used in Section 4 to formalize the notion of process configuration. Section 5 presents the solution approach for correctness ensuring configuration. Often configurable process models cannot be freely configured and domain constraints and data dependencies need to be taken into account. For example, one cannot skip an activity that produces data to be used in a later phase of the process. Therefore, Section 6 shows how to incorporate such constraints. Section 7 discusses tool support. Related work is discussed in Section 8. Section 9 concludes the paper.

2. Motivation

The need for configuring business processes arises in many domains. For example, there are about 430 municipalities in The Netherlands. However, in principle, they all execute variants of the same set of processes. For example, they all support processes for registering a marriage or a divorce.

Variability also occurs in the insurance domain. For example Suncorp, the largest Australian insurance group, offers various insurance products using different brands such as Suncorp, AAMI, APIA, GIO, Just Car, Bingle, Vero, etc. There are insurance processes for each product (home, motor, commercial, liability, etc.) and these processes exist for the different Suncorp brands. In fact, there are up to 30 different variants of the process of handling an insurance claim at Suncorp.

Organizations such as Suncorp need to support many variants of the same process (intra-organizational variation). Likewise, different municipalities in a country need to offer similar services to their citizens, and, hence, need to manage similar collections of process models. However, due to demographics and political choices, municipalities are handling things differently. Sometimes these differences are unintentional; however, often these differences can be easily justified by the desired “Couleur Locale” (inter-organizational variation). Clearly, it is undesirable to support these intra- and inter-organizational variations by making copies of the same process models (and related IT systems!) that are subsequently adapted. Hence, it is important to support variability directly at the process model level.

In this paper, we support process variability by means of *configurable process models*. A configurable process model captures the behavior of all possible process variants and can be configured to each specific variant. Accordingly, configuration corresponds to *removing process behavior*. Clearly, other viewpoints are possible. For example, some authors also consider refinement and model extension as configuration primitives [12–14, 18, 19]. We will discuss such alternative approaches in Section 8.

Specifically, we focus on two operators to remove process behavior: i) *hiding*, i.e. bypassing, and ii) *blocking*, i.e. inhibiting, which are applied to the variation points of a configurable process model. Hiding and blocking are two basic operators for removing process behavior [5], since they apply to single activities. In fact, all other configuration mechanisms for removing process behavior that are available in the literature, can be expressed in terms of hiding and blocking (e.g., [3, 6, 20]).

The configured model that results from a configuration can be analyzed using traditional process verification approaches. In this paper, we call a configuration *feasible* if the configured model is considered to be correct. In particular, we use the notion of *weak termination* as a correctness criterion, according to which a process instance can always terminate correctly. This notion excludes the possibility of anomalies such as deadlocks and livelocks in the configured model. Other variants of *soundness* can be also used [21], but this requires adaptations with respect to the analysis technique used.

Using existing techniques, it is already challenging to verify a single concrete model. However, in process configuration, there can potentially be many configured models which would need to be verified. In fact, if a model has n variation points which are "allowed" by default and can all be configured as "hidden" or "blocked", there are 3^n possible configurations, each leading to a configured model. For example, the configurable process model we constructed from the VICS documentation[1]— an industry standard for logistics and supply chain management — comprises 50 activities, which result in 3^{50} possible configurations (an extremely large number) [22]. Thus, as pointed out in Section 1, it is unrealistic to assume that the brute-force approach depicted in Fig. 1 will work in practice.

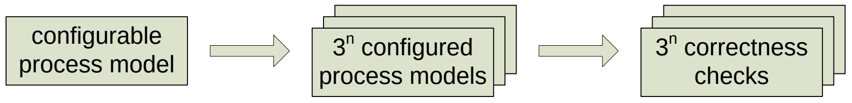


Figure 1: Classical, brute force, approach to verify the correctness of configurations.

One could use a trail-and-error approach when configuring a configurable process model. However, ideally the information system should support the configuration process by proactively removing configuration possibilities that result in incorrect process models. In fact, one would like to have a characterization of *all* feasible configurations and an "auto-complete" option that automatically completes a partial configuration while ensuring the correctness of the final model.

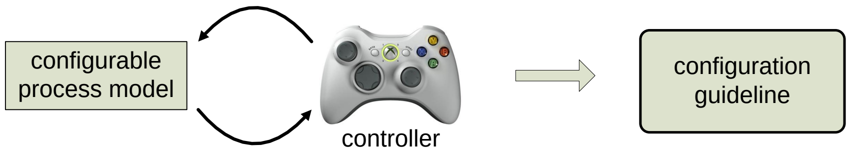


Figure 2: The approach presented in this paper. We reason about controllers that ensure feasible configurations. Using partner synthesis, we create one configuration guideline capturing all feasible configurations.

Given the above requirements, we propose a completely different approach which is shown in Fig. 2. Rather than exhaustively trying all 3^n possible configurations, we construct a "controller" that configures the process model correctly. Using controller synthesis [15, 16], we synthesize a so-called "most

[1]See `www.vics.com`

permissive partner" (see Section 5.2). This is the controller that does not remove any feasible configurations. Note that such a partner can be seen as an external service that configures the configurable model. The most permissive partner will serve as a *configuration guideline* steering the designer toward a good configuration. This provides us with a complete characterization of all feasible configurations at design time. Unlike existing approaches, we do not need to impose all kinds of syntactical restrictions on the class of models to be considered. Moreover, computation is moved from configuration time to design time and advanced functionality such as "auto-completion" comes into reach.

The ideas presented in this section are generic and do not depend on a particular representation. However, in order to explain the approach and to formalize the concepts, we use Petri nets. The next section introduces some preliminary concepts on Petri net.

3. Business Process Models

For the formalization of the problem we use Petri nets, which offer a formal model of concurrent systems. However, the same ideas can be applied to other languages (e. g. C-YAWL, C-BPEL), as it is easy to map the core structures of these languages onto Petri nets. Moreover, our analysis approach is quite generic and does not rely on specific Petri net properties.

Definition 1 (Petri net). A *marked Petri net* is a tuple $N = (P, T, F, m_0)$ such that: P and T ($P \cap T = \emptyset$) are finite sets of places and transitions, respectively, $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation, and $m_0 : P \to \mathbb{N}$ is an initial marking.

A Petri net is a directed graph with two types of nodes: places and transitions, which are connected by arcs as specified in the flow relation. If $p \in P$, $t \in T$, and $(p, t) \in F$, then place p is an input place of t. Similarly, $(t, p) \in F$ means that p is an output place of t.

The *marking* of a Petri net describes the distribution of tokens over places and is represented by a *multiset of places*. For example, the marking $m = [a^2, b, c^4]$ indicates that there are two tokens in place a, one token in b, and four tokens in c. Formally m is a function such that $m(a) = 2$, $m(b) = 1$, and $m(c) = 4$. We use $\oplus$ to compose multisets; for instance, $[a^2, b, c^4] \oplus [a^2, b, d^2, e] = [a^4, b^2, c^4, d^2, e]$.

A transition is *enabled* and can *fire* if all its input places contain at least one token. Firing is atomic and consumes one token from each of the input places and produces one token on each of the output places. $m_0 \xrightarrow{t} m$ means that t is enabled in marking m_0 and the firing of t in m_0 results in marking m. We use $m_0 \xrightarrow{*} m$ to denote that m is reachable from m_0; that is, there exists a (possibly empty) sequence of enabled transitions which by firing lead from m_0 to m.

For our configuration approach, we use *open nets*. Initially, open nets were introduced as an extension of *workflow nets* (WF-nets) [1], sometimes also referred to as loosely coupled inter-organizational workflows [23], open WF-nets (oWF-nets) [24], or workflow modules [25]. Whereas WF-nets are closed, open nets are enriched with communication places to model channels for sending/receiving messages to/from other open nets. In other publications the requirement, typical of WF-nets, of having a dedicated source place (i.e. a place without input transitions) and a dedicated sink place (i.e. a place without output transitions) was dropped [26]. Accordingly, in this paper we allow multiple source and sink places, an arbitrary initial marking and multiple final markings. As discussed in Section 5.2, many results for open nets with asynchronous communication can be extended to open nets with synchronous communication [15]. In fact, in this paper we will *only* use synchronous communication as this simplifies the modeling of process configuration. Therefore, we define a variant of open nets without communication places but with transitions that can synchronize with transitions in other open nets based on common labels. Hence, open nets extend classical Petri nets with the identification of final markings Ω, a set of labels L, and a function ℓ which assigns labels to transitions.

Definition 2 (Open net). A tuple $N = (P, T, F, m_0, \Omega, L, \ell)$ is an *open net* if

- (P, T, F, m_0) is a marked Petri net (called the *inner net* of N),
- $\Omega \subset P \to \mathbb{N}$ is a finite set of *final markings*,
- L is a finite set of *labels*,
- $\tau \notin L$ is a label representing invisible (also called silent) steps, and
- $\ell : T \to L \cup \{\tau\}$ is a *labeling function.*

We use *transition labels* to represent the activity corresponding to the execution of a particular transition. This way we can model the situation where an activity appears multiple times in a model. Two transitions having the same visible label refer to the same activity. The special label τ refers to an invisible step, sometimes referred to as "silent". We use invisible transitions to represent internal actions which do not mean anything at the business level. We use visible labels to denote activities that may be configured. Later, in Section 5 we use these labels to synchronize two open nets.

Note that Definition 2 does not allow for communication places; it is sufficient to introduce transition labels that can be used to model synchronous communication. Therefore, the *inner net*, i.e., the process without communication capabilities, is simply the core Petri net and its initial marking.

Figure 3 shows an example open net which models a typical travel request approval. The process starts with the preparation of the travel form. This can either be done by an employee or be delegated to a secretary. In both cases, the employee personally needs to arrange the travel insurance. If the travel form has been prepared by the secretary, the employee needs to check it before submitting it for approval. An administrator can then approve or reject the request, or make a request for change. Now, the employee can update the form according to the administrator's suggestions and resubmit it. In Fig. 3 all transitions bear

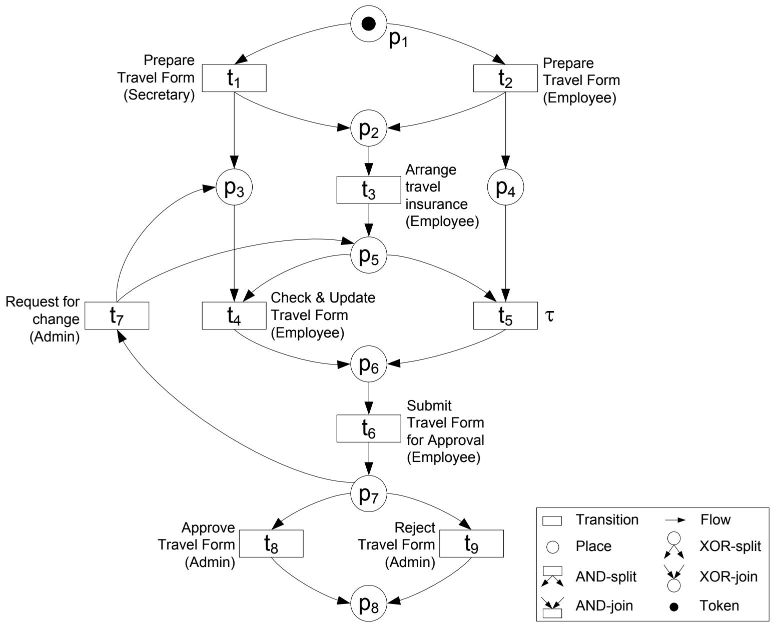

p1
Prepare Travel Form (Secretary)
t1
t2
Prepare Travel Form (Employee)
p2
t3
Arrange travel insurance (Employee)
p3
p4
p5
Request for change (Admin)
t7
t4
Check & Update Travel Form (Employee)
t5
τ
p6
t6
Submit Travel Form for Approval (Employee)
p7
Approve Travel Form (Admin)
t8
Reject Travel Form (Admin)
t9
p8
Transition
Flow
Place
XOR-split
AND-split
XOR-join
AND-join
Token

Figure 3: The open net for travel request approval ($\Omega = \{[p_8]\}$).

a visible label, except for t_5 which bears a τ-label as it has only been added for routing purposes.

Unlike our previous approach [5] based on WF-nets [1] and hence limited to a single final place, here we allow *multiple final markings*. Good runs of an open net end in a marking in set Ω. Therefore, an open net is considered to be erroneous if it can reach a marking from which no final marking can be reached any more. An open net *weakly terminates* if a final marking is reachable from every reachable marking.

Definition 3 (Weak termination). An open net $N = (P, T, F, m_0, \Omega, L, \ell)$ *weakly terminates* if and only if for any marking m with $m_0 \xrightarrow{*} m$ there exists a final marking $m_f \in \Omega$ such that $m \xrightarrow{*} m_f$.

The net in Fig. 3 is weakly terminating. Weak termination is a weaker notion than soundness, as it does not require transitions to be quasi-live [21]. This correctness notion is more suitable as parts of a correctly configured net may be left dead intentionally.

In this paper, we will restrict ourselves to open nets that are *bounded*, i.e., only finitely many states are reachable from the initial marking. The definitions in this paper also apply to unbounded nets. However, since open nets describe the life-cycle of one process instance in isolation (e.g., a travel request), there is no need to consider unbounded behavior. Moreover, most synthesis problems are known to be undecidable for unbounded nets [27].

4. Process Model Configuration

We use open nets to model configurable process models. An open net can be configured by blocking or hiding activities. Blocking activity a means that all transitions labeled a are removed. The corresponding activity is no longer available, i.e., paths visiting a transition corresponding to the blocked activity cannot be taken any more. Hiding activity a means that all transitions labeled a are skipped rather than executed. Hiding a transition means that it can be bypassed, i.e., the path containing the transition can still be taken. If a transition is neither blocked nor hidden, we say it is allowed, meaning it remains in the model. Configuration is achieved by setting visible labels to *allow*, *hide* or *block*.

Definition 4 (Open net configuration). Let N be an open net with label set L. A mapping $C_N : L \to \{\mathit{allow}, \mathit{hide}, \mathit{block}\}$ is a *configuration for* N. We define:

- $A_N^C = \{t \in T \mid \ell(t) \neq \tau \ \wedge \ C_N(\ell(t)) = \mathit{allow}\}$,
- $H_N^C = \{t \in T \mid \ell(t) = \tau \ \vee \ C_N(\ell(t)) = \mathit{hide}\}$, and
- $B_N^C = \{t \in T \mid \ell(t) \neq \tau \ \wedge \ C_N(\ell(t)) = \mathit{block}\}$.

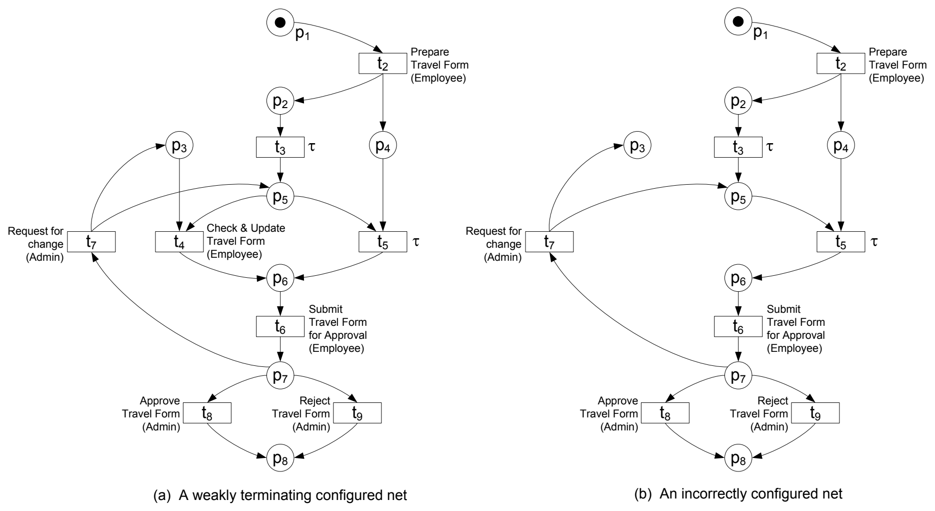
An open net configuration implicitly defines an open net, called *configured net*, where the blocked transitions are removed and the hidden transitions are given a τ-label.

Definition 5 (Configured net). Let $N = (P, T, F, m_0, \Omega, L, \ell)$ be an open net and C_N a configuration of N. The resulting *configured net* $\beta_N^C = (P, T^C, F^C, m_0, \Omega, L, \ell^C)$ is defined as follows:

- $T^C = T \setminus (B_N^C)$,
- $F^C = F \cap ((P \cup T^C) \times (P \cup T^C))$, and
- $\ell^C(t) = \ell(t)$ for $t \in A_N^C$ and $\ell^C(t) = \tau$ for $t \in H_N^C$.

As an example, Fig. 4(a) shows the configured net derived from the open net in Fig. 3 and the configuration C_N(*Prepare Travel Form* (*Secretary*)) = *block* (to allow only employees to prepare travel forms), C_N(*Arrange Travel Insurance* (*Employee*)) = *hide* (to skip arranging the travel insurance), and $C_N(x) = \mathit{allow}$ for all other labels x.

A configured net may have disconnected nodes and some parts may be dead (i. e., can never become active). Such parts can easily be removed. However, as we impose no requirements on the structure of configurable models, these disconnected or dead parts are irrelevant with respect to weak termination. For example, if we block the label of t_2 in Fig. 3, transition t_5 becomes dead as it cannot be enabled any more, and hence can also be removed without causing any behavioral issues. Nonetheless, not every configuration of an open net results in a weakly terminating configured net. For example, by blocking the label of t_4 in



p1
t2
Prepare Travel Form (Employee)
p2
t3
τ
p4
p3
p5
Request for change (Admin)
t7
t4
Check & Update Travel Form (Employee)
t5
τ
p6
t6
Submit Travel Form for Approval (Employee)
p7
Approve Travel Form (Admin)
t8
Reject Travel Form (Admin)
t9
p8
(a) A weakly terminating configured net
(b) An incorrectly configured net

Figure 4: Two possible configured nets based on the model in Fig. 3.

the configured net of Fig. 4(a), we obtain the configured net in Fig. 4(b). This net is not weakly terminating because after firing t_7 tokens will get stuck in p_3 (as this place does not have any successor) and in p_5 (as t_5 can no longer fire).

Blocking can cause behavioral anomalies such as the deadlock in Fig. 4(b). However, hiding cannot cause such issues, because it merely changes the labels of an open net. In this paper we are interested in all configurations which yield weakly terminating configured nets. We use the term *feasibility* to refer to such configured nets.

Definition 6 (Feasible configuration). Let N be an open net and C_N a configuration of N. C_N is *feasible* if and only if the configured net β_N^C weakly terminates.

More precisely, given a configurable process model N, we are interested in the following two questions: i) Is a *particular* configuration C_N feasible? ii) How to characterize the set of *all* feasible configurations?

The remainder of this paper is devoted to a new verification approach answering these questions. This approach extends the work in [5] in two directions: (i) it imposes *no unnecessary requirements* on the configurable process model (allowing for non-free-choice nets [28] and nets with multiple end places/markings), and (ii) it checks a *weaker correctness* notion (i. e. weak termination instead of soundness). For instance, the net in Fig. 3 is not free-choice because t_4 and t_5 share an input place, but their sets of input places are not identical. The non-free-choice construct is needed to model that after firing t_1 or t_7, t_5 cannot be fired, and similarly, after firing t_2, t_4 cannot be fired.

5. Correctness Ensuring Configuration

To address the two main questions posed in the previous section, we could use a direct approach by enumerating all possible configurations and simply checking whether each of the configured nets β_N^C weakly terminates or not (see Fig. 1). As indicated before, the number of possible configurations is exponential in the number of configurable activities. Moreover, most techniques for checking weak termination typically require the construction of the state space. Hence, traditional approaches are computationally expensive and do not yield a useful characterization of the set of all feasible configuration. Consequently, we propose a completely different approach using the synthesis technique described in [15]. As shown in Fig. 2, *the core idea is to see the configuration as an "external service" and then synthesize a "most permissive partner".* This most permissive partner represents all possible "external configuration services" which yield a feasible configuration. The idea is closely linked to the notion of *operating guidelines* for service behavior [17]. An operating guideline is a finite representation of all possible partners. Similarly, our *configuration guideline characterizes all feasible process configurations.* This configuration guideline can also be used to *efficiently check the feasibility of a particular configuration without exploring the state space of the configured net.* Our approach consists of three steps:

1. Transform the configurable process model N into a *configuration interface* N^{CI}.
2. Synthesize the "most permissive partner" (our *configuration guideline*) Q^{C_N} for the configuration interface N^{CI}.
3. Study the composition of N^{CI} with Q^{C_N}.

In the remainder of this section we explain these three steps. We will use two types of configuration interfaces: one where everything is *allowed by default* and the external configuration service can block or hide labels and one where everything is *blocked by default* and the external configuration service can "unblock" (i. e., allow or hide) labels. Section 5.1 provides some more preliminaries needed to reason about configuration interfaces. In Section 5.2, we informally describe how to check the existence of a partner and how to construct a most permissive partner. The configuration interface in which everything is allowed by default is presented in Section 5.3. The configuration interface in which everything is blocked by default is presented in Section 5.4. Section 5.5 shows another example to illustrate the concepts.

5.1. Composition and Controllability

For our solution approach, we compose the configurable process model with a "configuration service" Q. To do so, we first introduce the notion of *composition.* Open nets can be composed by synchronizing transitions according to their visible labels. In the resulting net, all transitions bear a τ-label and labeled transitions without counterpart in the other net disappear.

Definition 7 (Composition). For $i \in \{1, 2\}$, let $N_i = (P_i, T_i, F_i, m_{0_i}, \Omega_i, L_i, \ell_i)$ be open nets. N_1 and N_2 are *composable* if and only if the inner nets of N_1 and N_2 are pairwise disjoint. The *composition of two composable open nets* is the open net $N_1 \oplus N_2 = (P, T, F, m_0, \Omega, L, \ell)$ with:

- $P = P_1 \cup P_2$,
- $T = \{t \in T_1 \cup T_2 \mid \ell(t) = \tau\} \cup \{(t_1, t_2) \in T_1 \times T_2 \mid \ell(t_1) = \ell(t_2) \neq \tau\}$,
- $F = ((F_1 \cup F_2) \cap ((P \times T) \cup (T \times P))) \cup \{(p, (t_1, t_2)) \in P \times T \mid (p, t_1) \in F_1 \vee (p, t_2) \in F_2\} \cup \{((t_1, t_2), p) \in T \times P \mid (t_1, p) \in F_1 \vee (t_2, p) \in F_2\}$,
- $m_0 = m_{0_1} \oplus m_{0_2}$,
- $\Omega = \{m_1 \oplus m_2 \mid m_1 \in \Omega_1 \wedge m_2 \in \Omega_2\}$,
- $L = \emptyset$, and $\ell(t) = \tau$ for all $t \in T$.

Via composition, the behavior of each original net can be limited; for instance, transitions may no longer be available or may be blocked by one of the two original nets. Furthermore, final markings have an impact on weak termination: final markings of the compositions consist of the final markings of each composed net. Hence, it is possible that N_1 and N_2 are weakly terminating, but $N_1 \oplus N_2$ is not. Similarly, $N_1 \oplus N_2$ may be weakly terminating, but N_1 and N_2 are not. The labels of the two open nets in Def. 7 serve now a different purpose: they are not used for configuration, but for synchronous communication as described in [15]. As discussed in Section 3, we consider open nets without communication places as we restrict ourselves to synchronous communication. Therefore, the inner nets need to be pairwise disjoint.

With the notions of composition and weak termination, we define the concept of *controllability*, which we need to reason about the existence of feasible configurations.

Definition 8 (Controllability). An open net N is *controllable* if and only if there exists an open net N' such that $N \oplus N'$ is weakly terminating.

Open net N' is called a *partner* of N if $N \oplus N'$ is weakly terminating. Hence, N is controllable if there exists a partner. Wolf [15] presents an algorithm to check controllability: if an open net is controllable, this algorithm can synthesize a partner.

5.2. Checking Controllability and Partner Synthesis

We briefly describe an algorithm from Wolf [15] to construct a partner for an open if one exists. The approach is limited to bounded open nets. For infinite state systems, a related controllability notion is undecidable [27]. The algorithm does not directly construct an open net, but a finite-state automaton. The latter can be transformed into a Petri net model using standard techniques and tools [29, 30].

To construct a partner for an open net N, we first overapproximate the behavior of any open net that is composable to N. As the marking of N is not observable by the partner, we can only make assumptions based on the previous communication with N. These assumptions and the uncertainty about the exact state is modeled by a *set* S of markings the open net can reach at a certain point of interaction. This set S contains all markings that can be reached in N without requiring any actions of the environment. That is, S contains those states that are reachable only by τ-labeled transitions. Hence, the set S can be treated as a state of a partner of N.

The successors of the set S can be constructed according to the labels of N. Given a label $l \in L$, we construct a new set S' that contains all markings that can be reached by an l-labeled transition from a marking $m \in S$. Again, we add all markings to S' that can be reached only by τ-labeled transitions. The resulting set S' is then the l-labeled successor of S.

In the resulting automaton, we treat those sets that contain a final marking of N as final states of the partner. The initial state is the set that contains exactly the initial marking m_0 of N and those markings that can be reached from m_0 with only τ-labeled transitions.

For a bounded open net N, the set construction eventually terminates, because only a finite number of marking sets exist. However, it is not guaranteed that the resulting automaton ensures weak termination. Therefore, in a last step, we need to remove all sets S that contain markings from which no final marking of N is reachable. This removal has to be continued until a fixed point is reached. Unless all sets are removed, the resulting automaton is a partner of N and we can conclude that N is controllable.

The partner automaton N found using the above procedure represents the "most permissive partner".

Details on the algorithm as well as a formal definition can be found in [15]. The algorithm is implemented in the tool Wendy [16] which also implements several reduction techniques that avoid the generation of sets that would be removed in later steps.

5.3. Configuration Interface: Allow by Default

After these preliminaries, we define the notion of a *configuration interface*. One of the objectives of this paper was to characterize the set of all feasible configurations by synthesizing a "most permissive partner". To do this, we transform a configurable process model (i. e., an open net N) into an open net N^{CI}, called the configuration interface, which can communicate with services which configure the original model. In fact, we shall provide two configuration interfaces: one where everything is *allowed by default* and the external configuration service can block and hide labels, and the other where everything is *blocked by default* and the external configuration service can allow and hide labels. Similarly, one can construct a *hide by default* variant, which we do not illustrate in this paper. In either case, the resulting open net N^{CI} is controllable if and only if there exists a feasible configuration C_N of N. Without loss of generality, we assume a

1-safe initial marking; that is, $m_0(p) > 0$ implies $m_0(p) = 1$. This assumption helps to simplify the configuration interface and any net whose initial marking is not 1-safe can easily be converted into an equivalent net having a 1-safe initial marking.

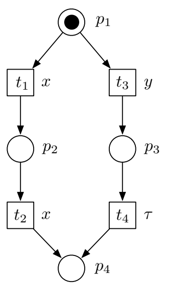
Definition 9 (Configuration interface; allow by default). Let $N = (P, T, F, m_0, \Omega, L, \ell)$ be an open net. We define the open net with configuration interface $N_a^{CI} = (P^C, T^C, F^C, m_0^C, \Omega^C, L^C, \ell^C)$ with

- $P^C = P \cup \{p_{\text{start}}\} \cup \{p_x, p_x^a, p_x^b, p_x^h \mid x \in L\}$,
- $T^C = T \cup \{t_{\text{start}}\} \cup \{b_x, h_x \mid x \in L\}$,
- $F^C = F \cup \{(p_{\text{start}}, t_{\text{start}})\} \cup \{(t_{\text{start}}, p) \mid p \in P \wedge m_0(p) = 1\} \cup \{(t, p_x), (p_x, t) \mid \ell(t) = x\} \cup \{(b_x, p_{\text{start}}), (p_{\text{start}}, b_x) \mid x \in L\} \cup \{(h_x, p_{\text{start}}), (p_{\text{start}}, h_x) \mid x \in L\} \cup \{(p_x^a, b_x), (p_x, b_x), (b_x, p_x^b) \mid x \in L\} \cup \{(p_x^a, h_x), (h_x, p_x^h) \mid x \in L\}$,
- $m_0^C = [p^1 \mid p \in \{p_{\text{start}}\} \cup \{p_x, p_x^a \mid x \in L\}]$,[2]
- $\Omega^C = \{m \oplus \bigoplus_{x \in L} m_x^* \mid m \in \Omega \ \wedge \ \forall_{x \in L}\ m_x^* \in \{[p_x, p_x^a], [p_x^b], [p_x, p_x^h]\}\ \}$,[3]
- $L^C = \{\text{start}\} \cup \{\text{block}_x, \text{hide}_x \mid x \in L\}$
- $\ell^C(t_{\text{start}}) = \text{start}$, $\ell^C(b_x) = \text{block}_x$ and $\ell^C(h_x) = \text{hide}_x$ for $x \in L$, and $\ell^C(t) = \tau$ for $t \in T$.

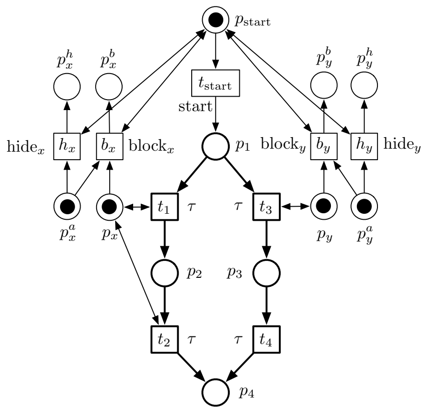
Figure 5 illustrates the two configuration interfaces for a simple open net N. In both interfaces, the original net N consisting of places $\{p_1, p_2, p_3, p_4\}$ and transitions $\{t_1, t_2, t_3, t_4\}$ is retained, but all transition labels are set to τ. Let us first focus on the configuration interface where all activities are allowed by default (Fig. 5(b)). The configuration interface consists of three parts: First, places p_x and p_y are added and connected with biflows to each transition of the original net. These places are used to control, for each label, whether a transition is blocked (i. e., the place is unmarked) or may fire (i. e., the place is marked). Second, the status of each label is modeled by the places p_x^a and p_y^a (allowed), p_x^b and p_y^b (blocked), and p_x^h and p_y^h (hidden). As we consider an allow-by-default scenario, places p_x, p_x^a, p_y, and p_y^a are initially marked. With two transitions for each label (b_x and h_x for blocking and hiding x-labeled transitions, and b_y and h_y for blocking and hiding y-labeled transitions), the status can be changed by the environment by synchronizing via labels block_x, hide_x, block_y, and hide_y, respectively. Finally, transition t_{start} has been added to ensure configuration actions take place *before* the original net is activated. This

[2] $[p^k \mid p \in X]$ denotes the multiset where each element of X appears k times. Initially, p_{start} contains one token. Since everything is allowed by default, also p_x and p_x^a contain a token in the initial marking ($x \in L$).

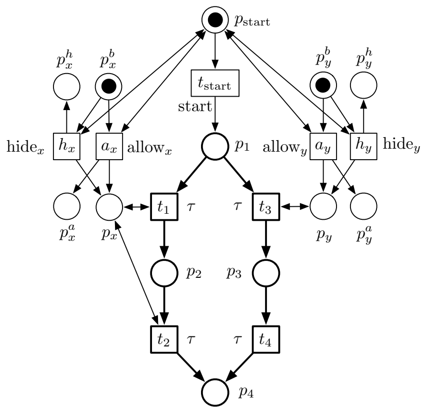
[3] Recall that $m_1 \oplus m_2$ denotes the composition of two multisets. The set of final markings imposes no restrictions on the newly added places. For label x, any of the three possible states — allowed $[p_x, p_x^a]$, blocked $[p_x^b]$, or hidden $[p_x, p_x^h]$ — is possible.



(a) Open net N ($\Omega = \{[p_4]\}$)



(b) Configurable interface N_a^{CI} (allow by default)



(c) Configurable interface N_b^{CI} (block by default)

Figure 5: An example open net (a) and its two configuration interfaces: (b) shows the allow by default variant and (c) shows the block by default variant.

way, we avoid "configuration on the fly". Note that currently the only constraint with respect to the final marking is that the original net must reach its final marking — all added places may be marked arbitrarily. In Sect. 6, we shall refine this final marking to encode domain knowledge and data dependencies. We shall discuss the construction of the configuration interface where all activities are blocked by default later on.

Consider now a *configuration service* represented as an open net Q. $N_a^{CI} \oplus Q$ is the composition of the original open net (N) extended with a configuration interface (N_a^{CI}), and the configuration service Q. In the initial phase, i. e., before *start* fires, only blocking and hiding transitions such as b_x, b_y, h_x, and h_y can fire (apart from unlabeled transitions in Q). Next, transition *start* fires after which blocking and hiding transitions such as b_x, b_y, h_x, and h_y can no longer fire. Hence, only the original transitions in N_a^{CI} can fire in the composition after firing *start*. The configuration service Q may still execute transitions, but these cannot influence N_a^{CI} any more. Hence, Q represents a feasible configuration if and only if N_a^{CI} can reach one of its final markings from any reachable marking in the composition. So Q corresponds to a feasible configuration if and only if $N_a^{CI} \oplus Q$ is weakly terminating; that is, Q is a partner of N_a^{CI}.

To illustrate the basic idea, we introduce the notion of a *canonical configuration partner*; that is, the representation of a configuration $C_N : L \to \{\mathit{allow}, \mathit{hide}, \mathit{block}\}$ in terms of an open net which synchronizes with the original model extended with a configuration interface.

Definition 10 (Canonical configuration partner; allow by default). Let N be an open net and let $C_N : L \to \{\mathit{allow}, \mathit{hide}, \mathit{block}\}$ be a configuration for N. $Q_a^{C_N} = (P, T, F, m_0, \Omega, L^Q, \ell)$ is the canonical configuration partner with:

- $L^* = \{x \in L \mid C_N(x) \neq \mathit{allow}\}$ is the set of labels other than "allow",
- $P = \{p_x^0, p_x^\omega \mid x \in L^*\}$,
- $T = \{t_x \mid x \in L^*\} \cup \{t_{\text{start}}\}$,
- $F = \{(p_x^0, t_x), (t_x, p_x^\omega), (p_x^\omega, t_{\text{start}}) \mid x \in L^*\}$,
- $m_0 = [(p_x^0)^1 \mid x \in L^*]$,[4]
- $\Omega = \{\ [\ ]\ \}$,
- $L^Q = \{\text{block}_x, \text{hide}_x \mid x \in L^*\} \cup \{\text{start}\}$,
- $\ell(t_x) = \text{block}_x$, if $C_N(x) = \text{block}$, $\ell(t_x) = \text{hide}_x$, if $C_N(x) = \text{hide}$, and $\ell(t_{\text{start}}) = \text{start}$.

The set of labels which need to be blocked or hidden to mimic configuration C_N is denoted by L^*. The canonical configuration partner $Q_a^{C_N}$ has a transition for each of these labels. These transitions may fire in any order after which the transition with label *start* fires. We observe that in the composition $N_a^{CI} \oplus Q_a^{C_N}$ first all transitions with a label in $\{\text{block}_x, \text{hide}_x \mid x \in L^*\}$ fire in a synchronous manner (i. e., t_x in $Q_a^{C_N}$ fires together with b_x or h_x in N_a^{CI}), followed by the transition with label *start* (in both nets). After this, the net is configured and $Q_a^{C_N}$ plays no role in the composition $N_a^{CI} \oplus Q_a^{C_N}$ any more.

The following lemma formalizes the relation between the composition $N_a^{CI} \oplus Q_a^{C_N}$ and feasibility.

Lemma 1. *Let N be an open net and let C_N be a configuration for N. C_N is a feasible configuration if and only if $N_a^{CI} \oplus Q_a^{C_N}$ is weakly terminating.*

Proof. ($\Rightarrow$) Let C_N be a feasible configuration for N and let N_a^{CI} be as defined in Def. 9. Consider the composition $N_a^{CI} \oplus Q_a^{C_N}$ after the synchronization via label *start* has occurred. By construction, (1) $N_a^{CI} \oplus Q_a^{C_N}$ reached the marking

[4]Recall that $[p^k \mid p \in X]$ denotes the multiset where each element of X appears k times. $[\ ]$ denotes the empty multiset.

$m = m_0 \oplus m_1 \oplus m_2$ such that m_0 is the initial marking of N, m_1 marks all places p_x^a, p_x^b, and p_x^h of the labels x with $C_N(x) = \text{allow}$, $C_N(x) = \text{block}$, and $C_N(x) = \text{hide}$, respectively. Furthermore, place p_x is marked for all unblocked labels x. Marking m_2 is the empty marking of Q^{C_N}. Furthermore, (2) all transitions which bear a synchronization label (i. e., t_{start} and all b_x and h_x transitions) and all blocked transitions $t \in B_N^C$ are dead in m and cannot become enabled any more. From N_a^{CI}, construct the net N^* by removing these transitions and their adjacent arcs, as well as the places added in the construction (p_{start} and p_x^a, p_x^b, and p_x^h for all labels $x \in L$). The marking of these places does not change any more, i. e., they either always contain a token or remain unmarked, and we already removed the transitions that are blocked. The resulting net N^* coincides with β_N^C (modulo renaming of labels which has no effect on termination). Hence, $N_a^{CI} \oplus Q_a^{C_N}$ weakly terminates.

($\Leftarrow$) Assume $N_a^{CI} \oplus Q_a^{C_N}$ weakly terminates. From $Q_a^{C_N}$, we can straightforwardly derive a configuration C for N in which all labels are blocked which occur in $N_a^{CI} \oplus Q_a^{C_N}$. With the same observation as before, we can conclude that β_N^C coincides with the net N^* constructed from N_a^{CI} after the removal the described nodes. Hence, β_N^C weakly terminates and C is a feasible configuration for N. □

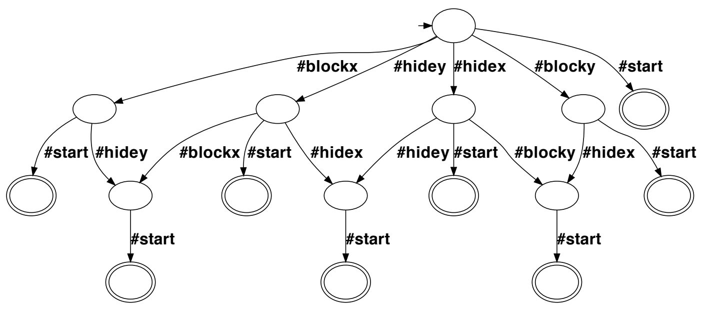
Lemma 1 states that checking the feasibility of a particular configuration can be reduced to checking for weak termination of the composition. However, the reason for modeling configurations as partners is that we can synthesize partners and test for the existence of feasible configurations.

Theorem 1 (Feasibility coincides with controllability)**.** *Let N be an open net. N_a^{CI} is controllable if and only if there exists a feasible configuration C_N of N.*

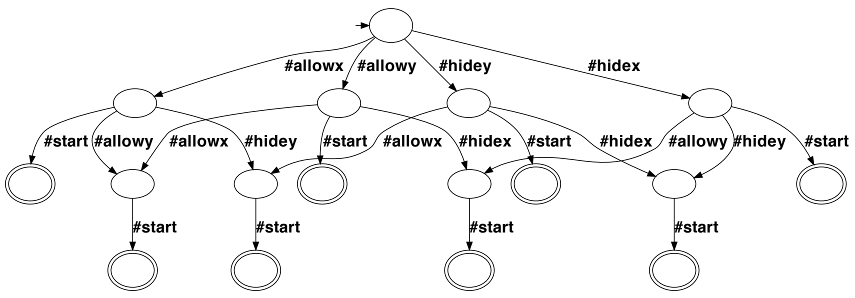
Proof. ($\Rightarrow$) If N_a^{CI} is controllable, then there exists a partner N' of N_a^{CI} such that $N_a^{CI} \oplus N'$ is weakly terminating. Consider a marking m of the composition reached by a run σ from the initial marking of $N_a^{CI} \oplus N'$ to the synchronization via label *start*. Using the construction from the proof of Lemma 1, we can derive a net N^* from N_a^{CI} which coincides with a configured net β_N^C for a configuration C_N. As $N_a^{CI} \oplus N'$ is weakly terminating, C_N is feasible.

($\Leftarrow$) If C_N is a feasible configuration of N, then by Lemma 1, $N_a^{CI} \oplus Q_a^{C_N}$ weakly terminates and by Def. 8, N_a^{CI} is controllable. □

As shown in Section 5.2, it is possible to synthesize a partner which is *most-permissive.* This partner simulates any other partner and thus characterizes all possible feasible configurations. In previous papers on partner synthesis in the context of service oriented computing, the notion of an *operating guideline* was used to create a finite representation capturing all possible partners [17]. Consequently, we use the term *Configuration Guideline* (CG) to denote the most-permissive partner of a configuration interface. Figure 6(a) shows the configuration guideline CG^a for the configurable model in Fig. 5(a), computed from the configuration interface N_a^{CI} in Fig. 5(b).



(a) CG^a Allow by default



(b) CG^b Block by default

Figure 6: Two configuration guidelines characterizing all possible configurations.

A configuration guideline is an automaton with one start state and one or more final states. *Any path in the configuration guideline starting in the initial state and ending in a final state corresponds to a feasible configuration.* The initial state in Fig. 6(a) is denoted by a small arrow and the final states are denoted by double circles. The leftmost path in Fig. 6(a) (i. e., $\langle \text{block}_x, \text{start} \rangle$), corresponds to the configuration which blocks label x. Path $\langle \text{block}_y, \text{start} \rangle$ corresponds to the configuration which blocks label y. The rightmost path (i. e., $\langle \text{start} \rangle$) does not block any label. The three paths capture all three feasible configurations that do not consider hiding steps. As hiding and allowing have the same effect on the original net (i. e., the respective labeled transitions may fire), each configuration that does not block a transition (and hence allows it by default) may further hide that transition. This yields a large number of further possible configurations. Figure 6(a) lists all feasible configurations, and, for example, shows that blocking both labels is not feasible. Since there are only two labels and eight feasible configurations, the conclusions based on Fig. 6(a) are rather obvious. However, configuration guidelines can be automatically computed for large and complex configurable process models.

5.4. Configuration Interface: Block by Default

Thus far, we used a configuration interface that allows all configurable activities by default, that is, blocking and hiding are explicit actions of the partner. It is also possible to use a completely different starting point and initially block all activities.

Definition 11 (Configuration interface; block by default). Let $N = (P, T, F, m_0, \Omega, L, \ell)$ be an open net. We define the open net with configuration interface $N_b^{CI} = (P^C, T^C, F^C, m_0^C, \Omega^C, L^C, \ell^C)$ with

- $P^C = P \cup \{p_{\text{start}}\} \cup \{p_x, p_x^a, p_x^b, p_x^h \mid x \in L\}$,
- $T^C = T \cup \{t_{\text{start}}\} \cup \{a_x, h_x \mid x \in L\}$,
- $F^C = F \cup \{(p_{\text{start}}, t_{\text{start}})\} \cup \{(t_{\text{start}}, p) \mid p \in P \wedge m_0(p) = 1\} \cup \{(t, p_x), (p_x, t) \mid \ell(t) = x\} \cup \{(a_x, p_{\text{start}}), (p_{\text{start}}, a_x) \mid x \in L\} \cup \{(h_x, p_{\text{start}}), (p_{\text{start}}, h_x) \mid x \in L\} \cup \{(p_x^b, a_x), (a_x, p_x^a), (a_x, p_x) \mid x \in L\} \cup \{(p_x^b, h_x), (h_x, p_x^h), (h_x, p_x) \mid x \in L\}$,
- $m_0^C = [p^1 \mid p \in \{p_{\text{start}}\} \cup \{p_x^b \mid x \in L\}]$,
- $\Omega^C = \{m \oplus \bigoplus_{x \in L} m_x^* \mid m \in \Omega \ \wedge \ \forall_{x \in L} \ m_x^* \in \{[p_x, p_x^a], [p_x^b], [p_x, p_x^h]\} \ \}$,
- $L^C = \{\text{start}\} \cup \{\text{allow}_x, \text{hide}_x \mid x \in L\}$
- $\ell^C(t_{\text{start}}) = \text{start}$, $\ell^C(a_x) = \text{allow}_x$ and $\ell^C(h_x) = \text{hide}_x$ for $x \in L$, and $\ell^C(t) = \tau$ for $t \in T$.

N_b^{CI} in Fig. 5(c) shows the configuration interface where all activities are blocked by default. The idea is analogous to the construction of N_a^{CI}. Instead of b_x and b_y, transitions a_x and a_y are added to model the explicit allowing of labels x and y, respectively. Furthermore, the initial marking was adjusted: places p_x and p_y are initially unmarked such that, by default, none of the original transitions can fire. These places can be marked by allowing or hiding the respective label. Very similar to the "allow by default" case, we define a canonical configuration partner.

Definition 12 (Canonical configuration partner; block by default). Let N be an open net and let $C_N : L \to \{\mathit{allow}, \mathit{hide}, \mathit{block}\}$ be a configuration for N. $Q_b^{C_N} = (P, T, F, m_0, \Omega, L^Q, \ell)$ is the canonical configuration partner with:

- $L^* = \{x \in L \mid C_N(x) \neq \mathit{block}\}$ is the set of labels other than "block",
- $P = \{p_x^0, p_x^\omega \mid x \in L^*\}$,
- $T = \{t_x \mid x \in L^*\} \cup \{t_{\text{start}}\}$,
- $F = \{(p_x^0, t_x), (t_x, p_x^\omega), (p_x^\omega, t_{\text{start}}) \mid x \in L^*\}$,
- $m_0 = [(p_x^0)^1 \mid x \in L^*]$,
- $\Omega = \{ \ [\ ] \ \}$,
- $L^Q = \{\text{allow}_x, \text{hide}_x \mid x \in L^*\} \cup \{\text{start}\}$,

- $\ell(t_x) = \text{allow}_x$, if $C_N(x) = \text{allow}$, $\ell(t_x) = \text{hide}_x$, if $C_N(x) = \text{hide}$, and $\ell(t_{\text{start}}) = \text{start}$.

The structure of the canonical configuration partner $Q_b^{C_N}$ is identical to that of $Q_a^{C_N}$. Only the labels are different; that is, $L \setminus L^*$ are the labels that need to be "unblocked" (i. e., allow or hide). Moreover, we obtain the same results linking feasibility to controllability.

Lemma 2. *Let N be an open net and let C_N be a configuration for N. C_N is a feasible configuration if and only if $N_b^{CI} \oplus Q_b^{C_N}$ is weakly terminating.*

Proof. Analogous to the proof of Lemma 1. □

Theorem 2 (Feasibility coincides with controllability)**.** *Let N be an open net. N_b^{CI} is controllable if and only if there exists a feasible configuration C_N of N*

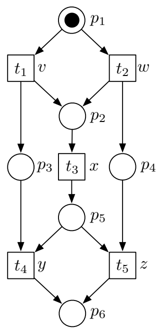
Proof. Analogous to the proof of Theorem 1. □

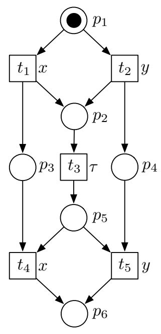
Figure 6(b) shows the configuration guideline CG^b for the configurable model in Fig. 5(a), computed from the configuration interface N_b^{CI} in Fig. 5(c). Again, any path in CG^b starting in the initial state and ending in a final state correspond to a feasible configuration. The leftmost path (i. e., $\langle \text{allow}_x, \text{start} \rangle$) corresponds to the configuration which "unblocks" label x by allowing it. Paths $\langle \text{allow}_x, \text{allow}_y, \text{start} \rangle$ and $\langle \text{allow}_y, \text{allow}_x, \text{start} \rangle$ correspond to the configuration where both x and y are allowed. The path $\langle \text{allow}_y, \text{start} \rangle$) allows y only. Similar paths exist for hiding, e.g., $\langle \text{hide}_x, \text{start} \rangle$ corresponds to the configuration which "unblocks" label x by hiding it. Again there are eight feasible configurations (see final states in Fig. 6(b)).

Clearly, the two configuration guidelines in Fig. 6 point to the same set of feasible configurations as they refer to the same original model. In can be noted that for each configuration that contains an allow_x there also exists a configuration with a hide_x, but otherwise identical actions. This is always the case; hiding and allowing are equivalent with respect to feasibility. *For this reason, we shall not depict hiding actions in the remainder of this section.* We have included them both in the constructs used because they become relevant when dealing domain knowledge and data dependencies (see Section 6). For example, if a transition produces a data element used later in the process, there is a clear difference between hiding or blocking it.

5.5. Another Example

Let us now consider a more elaborated example to see how configuration guidelines can be used to rule out unfeasible configurations. Figure 7 shows three open nets. The structures are identical, only the labels are different. For example, blocking x in N_2 corresponds to removing both t_1 and t_4 as both transitions bear the same label, while blocking x in N_3 corresponds to removing t_1 and t_5. For these three nets, we can construct the configuration interfaces using Def. 9 and then synthesize the configuration guidelines, as shown in Fig. 8.


(a) N_1


(b) N_2

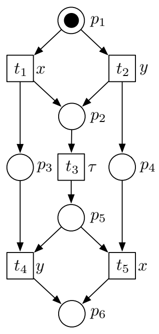
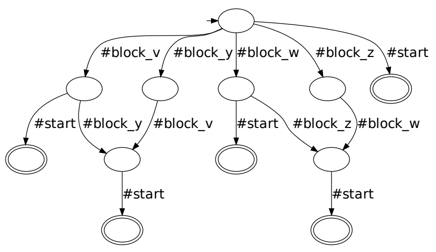
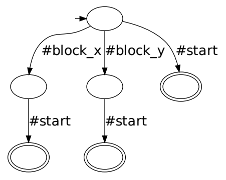

(c) N_3

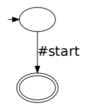
Figure 7: Three open nets ($\Omega = \{[p_6]\}$).



(a) CG_1^a



(b) CG_2^a



(c) CG_3^a

Figure 8: The configuration guidelines (allow by default) for N_1 (a), N_2 (b) and N_3 (c). Hiding actions are not depicted.

For these three nets, we can construct the configuration interfaces using Def. 9 or Def. 11, and then synthesize the configuration guidelines. Figure 8 shows the three configuration guidelines using Def. 9 (allow by default). As mentioned before, we refrained from presenting configurations that contain hiding activities.

Figure 8(a) reveals all feasible configurations for N_1 in Fig. 7(a). From the initial state in the configuration guideline CG_1^a, we can immediately reach a final state by following the rightmost path $\langle \text{start} \rangle$. This indicates that all configurations which block nothing (i. e., only allow or hide activities) are feasible. It is possible to just block v (cf. path $\langle \text{block}_v, \text{start} \rangle$) or block both v and y (cf. paths $\langle \text{block}_v, \text{block}_y, \text{start} \rangle$ and $\langle \text{block}_y, \text{block}_v, \text{start} \rangle$). However, it is not allowed to block y only, otherwise a token would deadlock in p_3. For the same reasons, one can block w only or w and z, but not z only. Moreover, it is not possible to combine the blocking of w and/or z on the one hand and v and/or y on the other hand, otherwise no final marking can be reached. Also x can never be blocked, otherwise both v and w would also need to be blocked (to avoid a token to deadlock in p_2) which is not possible. There are $3^5 = 243$ configurations for N_1. If we abstract from hiding as this does not influence feasibility (assuming we abstract from data and domain knowledge; see Section 6), there remain

$2^5 = 32$ possible configurations. Of these only 5 are feasible configurations which correspond to the final states in Fig. 8(a). This illustrates that the configuration guideline can indeed represent all feasible configurations in an intuitive manner.

Figure 8(b) shows the three feasible configurations for N_2 in Fig. 7(b). Again all final states correspond to feasible configurations. Here one can block the two leftmost transitions (labeled x) or the two rightmost transitions (labeled y), but not both.

The configuration guideline in Fig. 8(c) shows that nothing can be blocked for N_3 (Fig. 7(c)). Blocking x or y will yield an unfeasible configuration as a token will get stuck in p_4 (when blocking x) or p_3 (when blocking y). If both labels are blocked, none of the transitions can fire and thus no final marking can be reached.

In the next section we show how the partner synthesis can be further refined by ruling out specific partners based on domain knowledge and data dependencies.

6. Dealing with Domain Knowledge and Data Dependencies

Given an open net modeling a configurable process model, we can compute a configuration guideline. This configuration guideline characterizes all feasible configurations. The only consideration used to construct the configuration guideline has been weak termination of the model after configuration, i.e., it should always be possible to reach a desirable end state. However, in practice there will be many more constraints that a configuration should satisfy. For example, there may be domain constraints that inhibit particular configurations (e.g., a check activity may only be hidden when the payment activity is blocked). Data dependencies may also introduce such constraints. For example, if the only activity producing data element X is hidden, all subsequent activities using X should be blocked or hidden. These examples show that there is a significant difference between hiding and allowing whereas this was irrelevant for weak termination (cf. Section 5). Therefore, we introduce the notion of *configuration constraints*. Section 6.1 motivates the need for such constraints. Section 6.2 shows how these constraints can be formalized by simply restricting the set of final markings. As will be shown in Section 6.3, the notion of configuration constraints can be trivially embedded in the approach described in Section 5. This results in configuration guidelines that consider both weak termination and additional configuration constraints.

6.1. Configuration Constraints

Typically, configurable process models cannot be freely configured, i.e., even if the resulting configured model is free of deadlocks and livelocks, there may be good reasons for not allowing a particular configuration.

First of all, a configuration has to comply with constraints imposed by characteristics of the application domain [31]. Let us consider the travel request example in Fig. 3. In this business process there must always be an option to approve the request, and an option to reject it. Thus, we cannot block the label of transition t_8, nor that of t_9, although blocking either of these labels would still result in a feasible configuration. Moreover, we cannot hide the label of t_6 because a travel request cannot be approved or rejected if it has not been submitted first. These examples show that domain constraints may limit the space of acceptable configurations. Corporate governance and regulatory compliance may result in additional configuration constraints, e.g., there may be legal reasons for excluding particular configurations, such as the absence of a particular check activity.

In a similar vein, data dependencies among activities may prevent certain combinations of hiding and blocking. Activities typically have input data and output data. Suppose that activity a_1 creates a data element d that is later used by activity a_2. Obviously, it is not possible to hide or block a_1 while keeping a_2. Coming back to the travel request example, we cannot hide the labels of t_1 and t_2 because the data needed in subsequent steps would be missing. These two transitions create the Travel Form, which is used as input by all other transitions, e.g. transition "Arrange travel insurance" reads the Travel Form as input and writes an Insurance document as output. Although hiding the labels of t_1 and t_2 creates an obvious problem, we would not be able to observe this by just considering Definition 6. This shows that data needs to be taken into account when constructing the configuration guideline.

There is no need to take the actual data values into account; only the presence of data matters. Data presence can be checked by providing a so-called *CRUD matrix*. This is a matrix showing the relation between activities and data elements using the basic operations *Create* (C), *Read* (R), *Update* (U), and *Delete* (D). The idea to link data elements to activities originates from IBM's Business Systems Planning (BSP) methodology developed in the early eighties.

In [32] it was shown how the information contained in a CRUD matrix can be added to a process model expressed in terms of Petri nets. The operations considered in [32] are: *read* (activity a requires a data element d as input), *write* (activity a creates or updates data element d), *destroy* (activity a deletes data element d), and *guard* (a Boolean condition over data element d is used to route control to activity a). Figure 9 shows a refined version of net N_1 in Fig. 7(a), namely net N_D, which has been enriched with data manipulation aspects.

These data operations can be encoded in a Petri net which models the presence of data elements and the state of guards. There is no need to encode the explicit values of data elements to find data dependency errors. In [32] a set of *data-flow anti-patterns* is defined. For instance, the anti-pattern *DAP 1 (Missing Data)* describes the situation where some data element needs to be accessed, i.e., read or destroyed, but either it has never been created or it has been deleted without having been created again. Hiding an activity that creates a data element may easily result in the situation described by *DAP 1*. The data-flow anti-patterns are defined in the context of WF-nets. However, the ideas can easily be converted to open nets. In this paper, we do not list the various anti-patterns. Instead, we focus on the implications for process configuration.

Let us consider Fig. 9 again. This process starts when data element a is available. Based on the evaluation of the guard *pred(a)*, control is either routed

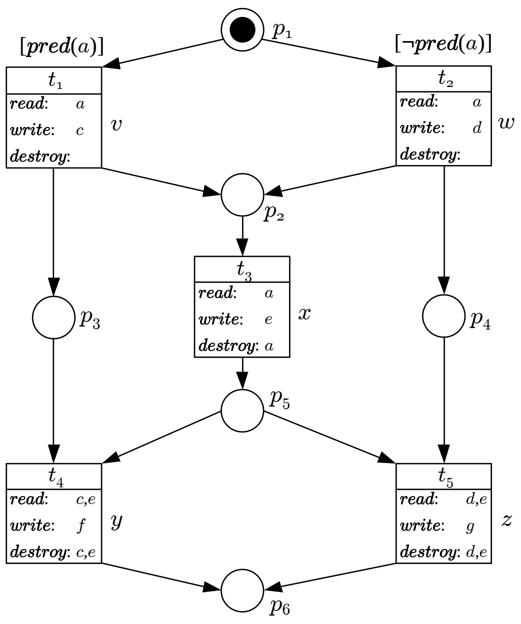


Figure 9: Net N_D models an open net annotated with information on data, e.g., transition t_1 has a guard involving data element a, t_1 reads a, and writes c.

to activity v or to activity w. Both these activities need to read a to start. v uses a to write, i.e. create, a new data element c, whereas w uses a to create d. After one of these activities is executed, control is passed to activity x which also requires a to start. This activity creates e and destroys a. The process concludes with the execution of y or z, based on the set of data elements that are available after executing x, i.e. c and e (in this case y is executed) or d and e (in this case z is executed).

From a configuration point of view, we are interested in the data dependencies that activity y has on activities v and x. v creates c and x creates e, which are both needed by y. These data dependencies imply the following data constraints for configuration: i) we can no longer block or hide v without also blocking or hiding y; and ii) we can no longer hide x without also hiding or blocking y (recall that x cannot be blocked for correctness reasons, as shown in Fig. 8(a)). Indeed, if we do not also hide or block y, this activity will deadlock waiting for a data element that will never be available. Similarly, z depends on the output produced by w and x. Thus, we can no longer block or hide w, or hide x, without also blocking or hiding z, otherwise the latter activity will deadlock. On the other hand, the dependency of activities v, w and x on a, as well as the guard on a, do not have any implication on the configuration of this net because a is an external data element. In fact, for the sake of refining the set of feasible configurations, we are only interested in the internal data dependencies among activities. More precisely, we consider reads, destroys and guards on data elements that are produced by some activity in the net.

In summary, both domain knowledge and data dependencies may limit the total number of feasible configurations. We refer to such restrictions as configuration constraints.

6.2. Formalization of Configuration Constraints

In order to consider configuration constraints in the partner synthesis, we introduce boolean constraints ("formulae") over activity labels (i.e., set L). For example, the domain constraint that the label of transition t_8 ("Approve Travel Form") cannot be blocked in our travel request process, can be expressed as $\neg\text{block}_{\ell(t_8)}$ (recall that visible transition labels correspond to activities, i.e., we do not block a specific transition, but all transitions having label x). Further, the data constraint that the labels of t_1 and t_2 cannot be simultaneously hidden, can be expressed as $((\text{hide}_{\ell(t_1)} \Rightarrow \neg\text{hide}_{\ell(t_2)}) \wedge (\text{hide}_{\ell(t_2)} \Rightarrow \neg\text{hide}_{\ell(t_1)}))$, whereas the data constraints among the transitions of net N_D can be expressed as $\text{allow}_y \Rightarrow (\text{allow}_v \wedge \text{allow}_x)$ and $\text{allow}_z \Rightarrow (\text{allow}_w \wedge \text{allow}_x)$.

Definition 13 (Formula). *Formulae* are defined inductively:
(Base) For a label $x \in L$, allow_x, block_x and hide_x are formulae. $\mathcal{F} = \bigcup_{x\in L}\{\text{allow}_x, \text{block}_x, \text{hide}_x\}$ is the set of all *atomic* formulae.
(Step) If φ and ψ are formulae, so are $\neg\varphi$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, and $(\varphi \Rightarrow \psi)$.

Examples of atomic formulae are allow_x and hide_y. Examples of non-atomic formulae are $(\text{allow}_x \vee \text{hide}_y)$ and $((\text{hide}_x \wedge \text{block}_y) \Rightarrow \neg\text{hide}_z)$.

Such formulae are generated based on domain knowledge and data dependencies. In [31] we showed how domain constraints can be transformed into such formulae while in [32] we showed how data dependencies can be extracted from a process model enhanced with data operations.

Next, we translate these formulae into constraints on the set of final markings of a configuration interface. In Definition 9, we defined that the set of final markings of the configuration interface (allow by default) is $\Omega^C = \{m \oplus \bigoplus_{x\in L} m^*_x \mid m \in \Omega \ \wedge\ \forall_{x\in L}\ m^*_x \in \{[p_x, p^a_x], [p^b_x], [p_x, p^h_x]\}\ \}$. The configuration interface defined in Definition 11 (block by default) specified the same set of final markings Ω^C. Hence, in both cases each label x is required to be in one of the following three states: $[p_x, p^a_x]$ (allowed), $[p^b_x]$ (blocked), or $[p_x, p^h_x]$ (hidden). Since this covers all three possibilities, it does not constrain the set of feasible configurations. We observe that after firing t_{start}, the state of a label does not change any more. Therefore, domain knowledge and data dependencies can be captured by removing undesirable markings from Ω^C.

Definition 14 (Translation of formulae into a set of final markings). Let $N = (P, T, F, m_0, \Omega, L, \ell)$ be an open net and φ be a formula representing the conjunction of all constraints resulting from domain knowledge and data dependencies. $\mathcal{A}_\varphi \subseteq 2^{\mathcal{F}}$ is the set of all satisfying assignments of φ.

For an assignment $A \in \mathcal{A}_\varphi$ and label $x \in L$, we define the multiset m_A^x with:

$$m_A^x = \begin{cases} [p_x, p_x^a], & \text{allow}_x \in A \\ [p_x^b], & \text{block}_x \in A \\ [p_x, p_x^h], & \text{hide}_x \in A \end{cases}$$

This allows us to redefine the set of final markings of the configuration interface:

$$\Omega^C = \{m \oplus \bigoplus_{x \in L} m_A^x \mid m \in \Omega \;\wedge\; A \in \mathcal{A}_\varphi\}$$

Each assignment corresponds to a set of final markings. The redefined set of final markings Ω^C can be used in the configuration interfaces defined in Definitions 9 and 11. By restricting Ω^C, domain knowledge and data dependencies are taken into account when checking feasibility and when constructing the configuration guideline.

6.3. Computing Configuration Guidelines Using Additional Constraints

The idea to constrain the set of partners of an open net by adjusting its final marking is inspired by the concept of *behavioral constraints* presented in [33]. By replacing Ω^C in Definitions 9 and 11 by the constrained Ω^C specified in Definition 14 we limit the set of feasible configurations to all weakly terminating configurations that do not violate any of the configuration constraints. This implies that we can still use the approach described in Section 5, i.e., using partner synthesis we synthesize a partner which is most-permissive resulting in the desired the configuration guideline.

Consider again N_D, the configurable model extended with data, shown in Fig. 9. Assume that we have two configuration constraints: $\text{allow}_y \Rightarrow (\text{allow}_v \wedge \text{allow}_x)$ and $\text{allow}_z \Rightarrow (\text{allow}_w \wedge \text{allow}_x)$. Based on these constraints we redefine the set of final markings of the configuration interface Ω^C. Based on the first configuration constraint ($\text{allow}_y \Rightarrow (\text{allow}_v \wedge \text{allow}_x)$) we remove those combinations of markings featuring $[p_y, p_y^a, p_v^b]$, $[p_y, p_y^a, p_v, p_v^h]$, $[p_y, p_y^a, p_x^b]$, or $[p_y, p_y^a, p_x, p_x^h]$. Based on the second configuration constraint ($\text{allow}_z \Rightarrow (\text{allow}_w \wedge \text{allow}_x)$) we remove all combinations featuring $[p_z, p_z^a, p_w^b]$, $[p_z, p_z^a, p_w, p_w^h]$, $[p_z, p_z^a, p_x^b]$, or $[p_z, p_z^a, p_x, p_x^h]$. This leads to the new configuration guideline $CG_{N_D}^a$ depicted in Fig. 10 (for simplicity those actions that have been hidden are not depicted). For example, now it is no longer possible to complete a configuration by blocking v only, i.e., y needs also to be blocked. It is also not possible to block only w, i.e. z also needs to be blocked. Compare Fig. 10 with Fig. 8(a) to see the differences between both configuration guidelines.

Although the example does not show the labels that have been hidden, we stress that from the viewpoint of domain knowledge and data dependencies there is a considerable difference between allowing and hiding an activity. When an activity is hidden no data is consumed nor produced. Therefore, we included hiding in the configuration interfaces (Definitions 9 and 11). Without adding

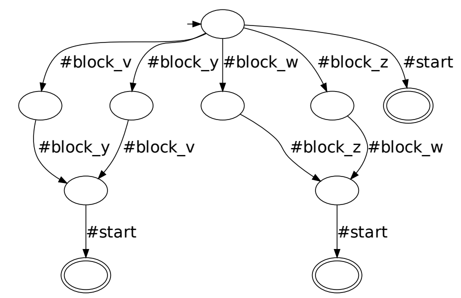


Figure 10: $CG^a_{N_D}$ the configuration guideline (allow by default) for N_D (hidden actions are not shown).

this to the corresponding interfaces, we would be unable to express constraints related to hiding.

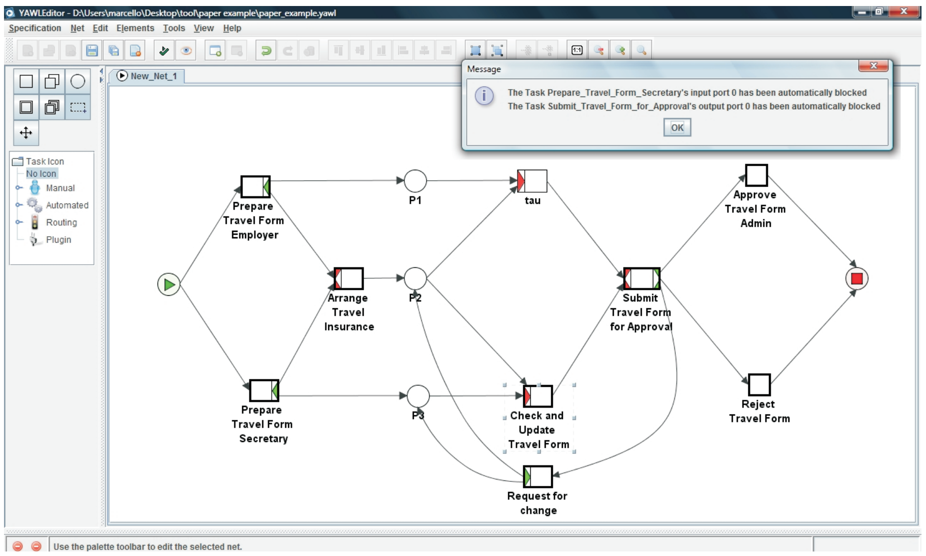
The simplicity of handling configuration constraints illustrates the flexibility of our approach based on partner synthesis.

7. Tool Support

To prove the feasibility of our approach, we applied it to the configuration of C-YAWL models [6] and extended the YAWL system accordingly.[5] The YAWL language can be seen as an extension of Petri nets which provides "syntactic sugaring" (shorthand notations for sequences and XOR-splits/joins). An atomic activity is called a task in YAWL. Composite tasks represent subprocesses. YAWL also provides advanced constructs such as cancelation sets, multiple instance tasks and OR-joins. YAWL is based on the well-know workflow patterns.[6] The YAWL system supporting this language is one of the most widely used open source workflow systems. For configuration, we restrict ourselves to the basic control-flow patterns supported by most systems. Thus we leave out YAWL's cancelation sets, multiple instance tasks and OR-joins. This allows us to easily map a YAWL model onto an open net.

A C-YAWL model is a YAWL model where some tasks are annotated as *configurable*. Configuration is achieved by restricting the routing behavior of configurable tasks via the notion of *ports*. A configurable task's joining behavior is identified by one or more *inflow* ports, whereas its splitting behavior is identified by one or more *outflow* ports. The number of ports for a configurable task depends on the task's routing behavior. For example, an AND-split/join and an OR-join are each identified by a single port, whereas an XOR-split/join is identified by one port for each outgoing/incoming flow. An OR-split is identified by a port for each combination of outgoing flows. To restrict a configurable task's routing behavior, inflow ports can be hidden (thus the corresponding

[5]`http://www.yawlfoundation.org`
[6]`http://www.workflowpatterns.com`


YAWLEditor - D:\Users\marcello\Desktop\tool\paper example\paper_example.yawl

Message

The Task Prepare_Travel_Form_Secretary's input port 0 has been automatically blocked
The Task Submit_Travel_Form_for_Approval's output port 0 has been automatically blocked

Use the palette toolbar to edit the selected net.

Figure 11: The C-YAWL model for travel request approval.

task will be skipped) or blocked (no control will be passed to the corresponding task via that port), whereas outflow ports can only be blocked (the outgoing paths from that task via that port are disabled). For instance, Fig. 11 shows the C-YAWL model for the travel request approval in the YAWL Editor, where configurable tasks are marked with a thicker border.

The *YAWL Editor* can be downloaded from `www.yawlfoundation.org`. It provides a graphical interface to conveniently configure and check C-YAWL models and subsequently generate configured models. Given a configuration, the tool can show a preview of the resulting configured net by graying out all model fragments which have been blocked, and commit the configuration by removing these fragments altogether.

To assist end users in ruling out all unfeasible configurations in an interactive manner, we developed a new component for the YAWL Editor named *C-YAWL Correctness Checker*. Given a C-YAWL model in memory, the component first maps this model into an open net. More precisely, it maps each condition to a place, each configurable task's port to a labeled transition, and each non-configurable task to a silent transition. Also, for each task it adds an extra place to connect the transition(s) derived from its inflow port(s) with the transition(s) derived from its outflow port(s). By using silent transitions we prevent non-configurable tasks from being later configured via a configuration interface. Next, the component passes the generated open net to the tool *Wendy* [16]. Wendy creates the corresponding configuration interface (allow by default), and produces the configuration guideline (allow by default) from the latter artifact.

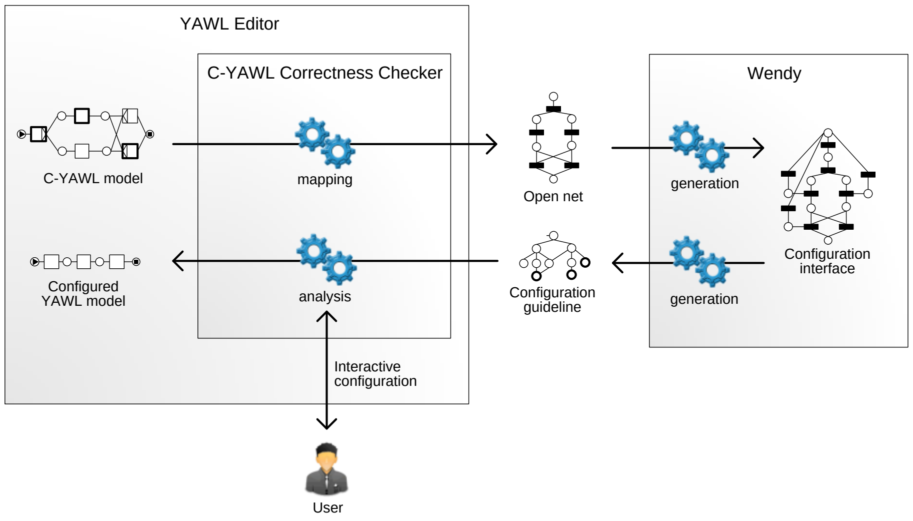


Figure 12: The interaction between the C-YAWL Correctness Checker and the Wendy tool.

Wendy is a free and open source tool[7] which implements the algorithms for partner synthesis [15] and to calculate operating guidelines [17]. Wendy itself offers no graphical user interface, but is controlled by input/output streams. In our setting, Wendy's output is piped back into the Correctness Checker, where it can be parsed. The component's interaction with Wendy is illustrated in Fig. 12.

The construction of the configuration guideline is based on partner synthesis as described in Section 5.2. Hence, we inherit the complexity of the partner synthesis algorithm. To synthesize a partner for an open net N with label set L, first a graph is constructed that over-approximates any potential interaction with N using the communication labels from L. Second, any behavior that violates weak termination is removed. If this results in a non-empty graph, we conclude N is controllable. We refer to [15] for a detailed description of the synthesis algorithm.

The complexity of this algorithm is dominated by the potential size of the over-approximation which is at most exponential i) on the size of N (because the reachability graph of N needs to be generated), and ii) on the size of L. The tool Wendy implements several reduction techniques to address potential problems due to this worst case complexity. First, the internal behavior that does not influence the interaction behavior is removed from the reachability graph. Second, the reachability graph is further preprocessed to discover non-terminating behavior as early as possible. Third, efficient data structures further minimize the memory footprint of the over-approximation stage. These optimizations

[7]Available for download at `http://service-technology.org/wendy`.

speed-up Wendy significantly when applied to real-life examples. Practical experiences show that Wendy is able to analyze industrial models having millions of states. Wendy is also capable of synthesizing partners of similar sizes [16].

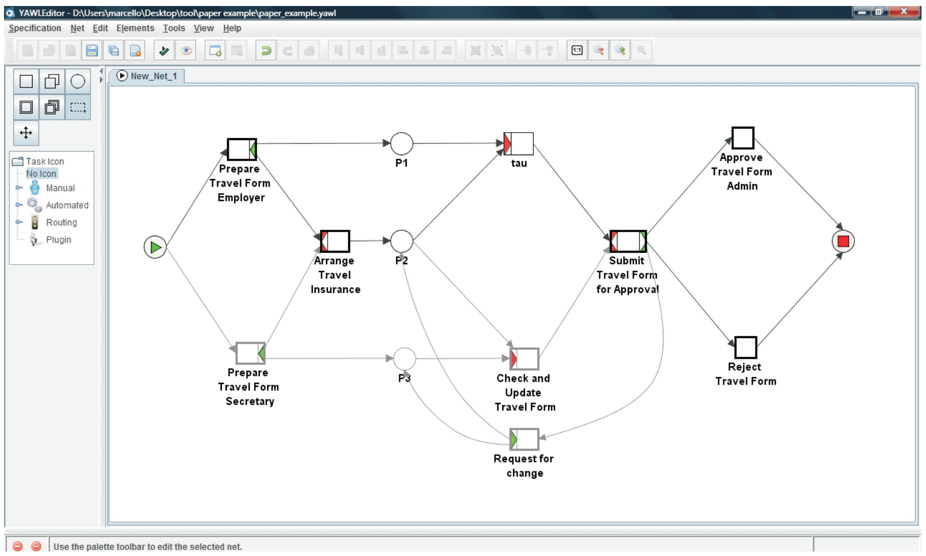
At each configuration step, the Correctness Checker scans the set of outgoing edges of the current state in the configuration guideline, and prevents users from blocking those ports not included in this set. This is done by disabling the block button for those ports. As users block a valid port, the Correctness Checker traverses the configuration guideline through the corresponding edge and updates the current state. If this is not a *consistent* state, that is, a state with an outgoing edge labeled "start", further ports need to be blocked, because the current configuration is unfeasible. In this case the component provides an *"auto complete" option.* This is achieved by traversing the shortest path from the current state to a consistent state and automatically blocking all ports on that path. After this, the component updates the current state and notifies the user with the list of ports that have been automatically blocked. For example, Fig. 11 shows that after blocking the input port of task *Check and Update Travel Form*, the component notifies the user that the input port of task *Prepare Travel Form for Approval (Secretary)* and the output port of task *Submit Travel Form for Approval* to task *Request for Change* have also been blocked. Figure 13 shows the preview of the resulting configured net. From this we can observe that condition p_3 and task *Request for Change* will also be removed from the net as a result of applying the earlier configuration. Similarly, the component maintains a consistent state in case users decide to allow a previously blocked port. In this case the component traverses the shortest backward path to the closest consistent state and allows all ports on that path. By traversing the shortest path we ensure that the number of ports being automatically blocked or allowed is minimal.

This auto-completion feature can be extended by prompting the user with the set of paths from the current state to a consistent state of a given length (e.g. five states). In this way the user can select which combinations of ports to block/allow in order to keep the configuration feasible.

The C-YAWL example of Fig. 11 comprises ten inflow ports and nine outflow ports. In total more than 30 million configurations are potentially possible. If we abstract from hiding we obtain 524,288 possible configurations, of which only 1,593 are feasible according to the configuration guideline. Wendy took an average of 336 seconds (on a 2.4 GHz processor with 2GB of RAM) to generate this configuration guideline which consumes 3.37 MB of disk space. Nonetheless, the shortest path computation is a simple depth-first search which is linear in the number of nodes in the configuration guideline. Thus, once the configuration guideline has been generated, the component's response time at each user interaction is instantaneous.

8. Related Work

Traditional reference models [7–10] are typically not executable. For example, the well-known SAP reference model is disconnected from the actual system


YAWLEditor - D:\Users\marcello\Desktop\tool\paper example\paper_example.yawl
Specification
Net
Edit
Elements
Tools
View
Help
New_Net_1
Task Icon
No Icon
Manual
Automated
Routing
Plugin
Prepare Travel Form Employer
P1
tau
Approve Travel Form Admin
Arrange Travel Insurance
P2
Submit Travel Form for Approval
Prepare Travel Form Secretary
P3
Check and Update Travel Form
Reject Travel Form
Request for change

Use the palette toolbar to edit the selected net.

Figure 13: The preview of a configured net for the example in Fig. 11.

and has many internal inconsistencies [34]. Such models focus on training and documentation rather than enactment. Configurable process models [3, 4, 6, 20, 35] can be seen as executable reference models.

In this paper, we take the viewpoint that configuration is achieved by restriction (i.e., hiding and blocking). Other paradigms are possible. For example, in the Provop (Managing and Configuring Process Variants) approach process designers create process variants by applying well-defined change operations on some reference model [13, 14]. Another example is the Application-based Domain Modeling (ADOM) approach which allows both reuse (i.e., restriction) and specialization [18, 19]. Specialization corresponds to the concretization of a reference model element into a specific business process model element. In [12] both configurative and generic adaptation techniques are proposed for adaptive reference modeling. In [36, 37] an approach is described to discover a reference model for which the edit distance to a given set of variants is minimal. This approach assumes that configuration corresponds to a sequence of change operations. Nevertheless, the majority of process configuration approaches use restriction of behavior as a configuration mechanism [3, 4, 6, 20, 35]. This paper also assumes that configuration is done through behavioral restrictions (hiding and blocking).

Since configurable process models are actively used to support processes, they need to be correct and only correct configurations should be allowed. Many researchers have worked on the verification of business processes, workflows, and services (e.g., [21, 32, 34, 38–41]). However, these approaches focus on the analysis of one process in isolation and can only be used to exhaustively verify all possible configurations to create a configuration guideline. In this respect, they face the state-space explosion problem.

To the best of our knowledge, our earlier approach [5] is the only one focusing on the behavioral correctness of process configurations which avoids state-space explosion. Other approaches either only discuss syntactical correctness related to configuration [3, 11, 12], or deal with behavioral correctness but run into the state-space problem [13]. For example, [3] preserves syntactic correctness by construction of the configured EPC model from a C-EPC, whereas [11, 12] prompt users with a list of syntactic issues detected during process configuration, which need to be manually fixed. Finally, [13] proposes to check the correctness of each single configured process model.

The approach in [5] presents a technique to derive propositional logic constraints from configurable process models that, if satisfied by a configuration step, guarantee the behavioral correctness of the configured model. This approach allows correctness to be checked at each intermediate step of the configuration procedure. Whenever a configuration value is assigned to a transition label (e.g. x is blocked), the current set of constraints is evaluated. If the constraints are satisfied, the configuration step is applied. If on the other hand the constraints are violated, a reduced propositional logic formula is computed, from which additional configuration values are determined, that also need to be applied in order to preserve correctness. Unfortunately, this approach requires the configurable process model to be a sound, free-choice Workflow net. Thus, these requirements limit the applicability of the approach. In the current paper, we do not impose such restrictions.

This paper is an extended version of [42]. In [42], we already described the idea of synthesizing a configuration guideline based on the partner synthesis approach [15]. However, in [42] we abstracted from hiding and only showed one configuration interface (allow by default). The actual construction presented in Section 5 is different from that in [42] to be able to deal with hiding. Moreover, we showed the configuration interface where everything is blocked by default. Hiding, blocking and allowing are now symmetric. In principle, we could have also provided a configuration interface that hides by default. Finally, we showed how constraints can be incorporated in our approach. These constraints may be derived from the domain in which the configurable process model has been constructed, or from data dependencies that exist among process tasks.

9. Conclusion

Configurable process models are a means to compactly represent families of process models. However, the verification of such models is difficult as the number of possible configurations grows exponentially in the number of configurable elements. Due to concurrency and branching structures, configuration decisions may interfere with each other and thus introduce deadlocks, livelocks and other anomalies. The verification of configurable process models is challenging and only few researchers have worked on this. Moreover, existing results impose restrictions on the structure of the configurable process model and fail to provide insights into the complex dependencies among different process model configuration decisions.

The main contribution of this paper is an innovative approach for ensuring correctness during process configuration. Using partner synthesis we compute the configuration guideline — a compact characterization of all feasible configurations, which allows us to rule out configurations that lead to behavioral problems. The approach is highly generic and imposes no constraints on the configurable process models that can be analyzed. Moreover, all computations are done at design time and not at configuration time. Thus, once the configuration guideline has been generated, the response time is instantaneous stimulating the practical (re-)use of configurable process models. The approach is implemented in a checker integrated in the YAWL Editor. This checker uses the Wendy tool to ensure correctness while users configure C-YAWL models.

In previous work, we have shown that our Wendy tool can cope with models generating millions of states. Furthermore, we have extensively evaluated the use of process configuration by "behavior restriction" to various domains such as logistics, municipalities and the screen business [4, 22, 35]. However, we acknowledge that further evaluations need to be conducted, in order to claim that the specific approach presented in this paper can be used in practice. This is an avenue for future work.

Several interesting extensions to this work are possible. First, it is possible to create more compact representations of configuration guidelines (e.g. exploiting concurrency [29]). The "diamond structures" in the example configuration guidelines illustrate that regions can help fold the guidelines and separate unrelated configuration decisions. However, more research is needed to understand how to best present the configuration guidelines to end-users (see e.g. our earlier work on questionnaire-based variability modeling [22]). Second, one could consider configuration at run-time, that is, while instances are running, configurations can be set or modified. This can be easily embedded in the current approach, but would be impossible when using conventional techniques. Finally, we are interested in relating this work on process configuration to process mining. Process mining has been focusing on the analysis of individual processes. However, as more and more variants of the same process need to be supported, it is interesting to analyze differences between these variants based on empirical data.

Acknowledgments.. We thank Jingxin Xu for his help with the implementation and testing of the C-YAWL component, and its integration with the Wendy tool. We also thank the anonymous reviewers for their valuable feedback which helped us improve the paper.

References

[1] W.M.P. van der Aalst, C. Stahl, Modeling Business Processes: A Petri Net Oriented Approach, MIT press, Cambridge, MA, 2011.

[2] F. Leymann, D. Roller, Production Workflow: Concepts and Techniques, Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

[3] M. Rosemann, W.M.P. van der Aalst, A Configurable Reference Modelling Language, Information Systems 32 (1) (2007) 1–23.

[4] M. La Rosa, M. Dumas, A. ter Hofstede, J. Mendling, Configurable Multi-Perspective Business Process Models, Information Systems 36 (2).

[5] W.M.P. van der Aalst, M. Dumas, F. Gottschalk, A. ter Hofstede, M. Rosa, J. Mendling, Preserving Correctness During Business Process Model Configuration, Formal Aspects of Computing 22 (3) (2010) 459–482.

[6] F. Gottschalk, W.M.P. van der Aalst, M. Jansen-Vullers, M. L. Rosa, Configurable Workflow Models, International Journal of Cooperative Information Systems 17 (2) (2008) 177–221.

[7] T. Curran, G. Keller, SAP R/3 Business Blueprint: Understanding the Business Process Reference Model, Upper Saddle River, 1997.

[8] P. Fettke, P. Loos, Classification of Reference Models - A Methodology and its Application, Information Systems and e-Business Management 1 (1) (2003) 35–53.

[9] S. Huan, S. Sheoran, G. Wang, A Review and Analysis of Supply Chain Operations Reference (SCOR) Model, Supply Chain Management - An International Journal 9 (1) (2004) 23–29.

[10] R. Addy, Effective IT Service Management. To ITIL and Beyond!, Springer-Verlag, Berlin, 2007.

[11] K. Czarnecki, M. Antkiewicz, Mapping Features to Models: A Template Approach Based on Superimposed Variants, in: Proceedings of the 4th Int. Conference on Generative Programming and Component Engineering, Springer-Verlag, Berlin, 2005, pp. 422–437.

[12] J. Becker, P. Delfmann, R. Knackstedt, Adaptive Reference Modeling: Integrating Configurative and Generic Adaptation Techniques for Information Models, in: J. Becker, P. Delfmann (Eds.), Reference Modeling: Efficient Information Systems Design Through Reuse of Information Models, Physica-Verlag, Springer, Heidelberg, Germany, 2007, pp. 27–58.

[13] A. Hallerbach, T. Bauer, M. Reichert, Guaranteeing Soundness of Configurable Process Variants in Provop, in: CEC, IEEE, 2009, pp. 98–105.

[14] A. Hallerbach, T. Bauer, M. Reichert, Capturing Variability in Business Process Models: The Provop Approach, Journal of Software Maintenance and Evolution: Research and Practice 22 (6-7) (2010) 519–546.

[15] K. Wolf, Does my Service Have Partners?, in: K. Jensen, W.M.P. van der Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, Vol. 5460 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 152–171.

[16] N. Lohmann, D. Weinberg, Wendy: A Tool to Synthesize Partners for Services, in: J. Lilius, W. Penczek (Eds.), International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Vol. 6128 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 297–307

[17] N. Lohmann, P. Massuthe, K. Wolf, Operating Guidelines for Finite-State Services, in: J. Kleijn, A. Yakovlev (Eds.), 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings, Vol. 4546 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 321–341.

[18] I. Reinhartz-Berger, P. Soffer, A. Sturm, Organizational Reference Models: Supporting an Adequate Design of Local Business Processes, International Journal of Business Process Integrations and Management 4 (2) (2009) 134–149.

[19] I. Reinhartz-Berger, P. Soffer, A. Sturm, Extending the Adaptability of Reference Models, IEEE Transactions on Systems, Man and Cybernetics - Part A: Systems and Humans 40 (5) (2011) 1045–1056.

[20] A. Schnieders, F. Puhlmann, Variability Mechanisms in E-Business Process Families, in: W. Abramowicz, H. Mayr (Eds.), Proceedings of the 9th International Conference on Business Information Systems (BIS'06), Vol. 85 of LNI, GI, 2006, pp. 583–601.

[21] W.M.P. van der Aalst, K. Hee, A. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, M. Wynn, Soundness of Workflow Nets: Classification, Decidability, and Analysis, Formal Aspects of Computing 23 (3) (2011) 333–363.

[22] M. Rosa, W.M.P. van der Aalst, M. Dumas, A. ter Hofstede, Questionnaire-based Variability Modeling for System Configuration, Software and Systems Modeling 8 (2) (2009) 251–274.

[23] W.M.P. van der Aalst, Process-oriented Architectures for Electronic Commerce and Interorganizational Workflow, Information Systems 24 (8) (2000) 639–671.

[24] P. Massuthe, W. Reisig, K. Schmidt, An Operating Guideline Approach to the SOA, Annals of Mathematics, Computing & Teleinformatics 1 (3) (2005) 35–43.

[25] E. Kindler, A. Martens, W. Reisig, Inter-Operability of Workflow Applications: Local Criteria for Global Soundness, in: W.M.P. van der Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management: Models, Techniques, and Empirical Studies, Vol. 1806 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, pp. 235–253.

[26] R. Heckel, Open Petri Nets as Semantic Model for Workflow Integration, in: H. Ehrig, W. Reisig, G. Rozenberg, H. Weber (Eds.), Petri Net Technology for Communication Based Systems, Vol. 2472 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 281–294.

[27] P. Massuthe, A. Serebrenik, N. Sidorova, K. Wolf, Can I find a Partner? Undecidablity of Partner Existence for Open Nets, Information Processing Letters 108 (6) (2008) 374–378.

[28] J. Desel, J. Esparza, Free Choice Petri Nets, Vol. 40 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, UK, 1995.

[29] E. Badouel, P. Darondeau, Theory of Regions, in: W. Reisig, G. Rozenberg (Eds.), Lectures on Petri Nets I: Basic Models, Vol. 1491 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 529–586.

[30] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, IEICE Transactions on Information and Systems E80-D (3) (1997) 315–325.

[31] M. La Rosa, J. Lux, S. Seidel, M. Dumas, A. ter Hofstede, Questionnaire-driven Configuration of Reference Process Models, in: CAiSE'07, Vol. 4495 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 424–438.

[32] N. Trcka, W.M.P. van der Aalst, N. Sidorova, Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows, in: P. van Eck, J. Gordijn, R. Wieringa (Eds.), Advanced Information Systems Engineering, Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09), Vol. 5565 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 425–439.

[33] N. Lohmann, P. Massuthe, K. Wolf, Behavioral Constraints for Services, in: BPM 2007, Vol. 4546 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 271–287.

[34] J. Mendling, H. Verbeek, B. van Dongen, W.M.P. van der Aalst, G. Neumann, Detection and Prediction of Errors in EPCs of the SAP Reference Model, Data and Knowledge Engineering 64 (1) (2008) 312–329.

[35] F. Gottschalk, T. Wagemakers, M. Jansen-Vullers, W.M.P. van der Aalst, M. Rosa, Configurable Process Models: Experiences From a Municipality Case Study, in: P. van Eck, J. Gordijn, R. Wieringa (Eds.), Advanced Information Systems Engineering, Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09), Vol. 5565 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 486–500.

[36] C. Li, M. Reichert, and A. Wombacher, Discovering Reference Models by Mining Process Variants Using a Heuristic Approach, in: U. Dayal, J. Eder, J. Koehler, H. Reijers (Eds.), Business Process Management (BPM 2009), Vol. 5701 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 344–362.

[37] C. Li, M. Reichert, A. Wombacher, The Minadept Clustering Approach for Discovering Reference Process Models Out of Process Variants, International Journal of Cooperative Information Systems 19 (3) (2010) 159–203.

[38] H. Lin, Z. Zhao, H. Li, Z. Chen, A Novel Graph Reduction Algorithm to Identify Structural Conflicts, in: Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-35), IEEE Computer Society Press, 2002.

[39] J. Dehnert, P. Rittgen, Relaxed Soundness of Business Processes, in: K. Dittrich, A. Geppert, M. Norrie (Eds.), Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), Vol. 2068 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 157–170.

[40] W. Sadiq, M. Orlowska, Analyzing Process Models using Graph Reduction Techniques, Information Systems 25 (2) (2000) 117–134.

[41] N. Sidorova, C. Stahl, N. Trcka, Soundness Verification for Conceptual Workflow Nets With Data: Early Detection of Errors With the Most Precision Possible, Information Systems 37 (7) (2011) 1026–1043.

[42] W.M.P. van der Aalst, N. Lohmann, M. Rosa, J. Xu, Correctness Ensuring Process Configuration: An Approach Based on Partner Synthesis, in: R. Hull, J. Mendling, S. Tai (Eds.), Business Process Management (BPM 2010), Vol. 6336 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2010, pp. 95–111.