Where did I go wrong?

Explaining errors in business process models

Niels Lohmann[1] and Dirk Fahland[2]

[1] Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
[2] Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
niels.lohmann@uni-rostock.de, d.fahland@tue.nl

Abstract. Business process modeling is still a challenging task — especially since more and more aspects are added to the models, such as data lifecycles, security constraints, or compliance rules. At the same time, formal methods allow for a detection of errors in the early modeling phase. Detected errors are usually explained with a path from the initial to the error state. These paths can grow unmanageably and make the understanding and fixing of errors very time consuming. This paper addresses this issue and proposes a novel explanation of errors: Instead of listing the actions on the path to the error, only the decisions that lead to it are reported and highlighted in the original model. Furthermore, we exploit concurrency to create a compact artifact to explain errors.

1 Introduction

Business process modeling is a sophisticated task and received a lot of attention in the past decades. With the advent of domain-specific languages and a growing scientific community, the act of creating and managing business process models has become a discipline on its own. Despite all efforts, design flaws may still occur. This can have different impacts, ranging from syntactically incorrect models, which are harder to understand, up to catastrophic faults and down times in the execution that yield to a loss of money or a legal aftermath. Consequently, a large branch of research focuses in the detection, correction, and avoidance of errors in business process models. Whereas plain control flow analysis is now well understood, other aspects such as data, business rules, or security may introduce more subtle flaws that are harder to detect.

The most prominent property of business process models is *soundness* [1], which combines several desirable properties such as proper termination and the absence of deadlocks, livelocks, and dead code. For this fundamental "sanity check", more and more sophisticated techniques and tools have been introduced in the last years. Recent experiments [2] suggest that soundness checks for industrial business process models can be conducted within microseconds. This allows for a tight integration of verification steps into the process of modeling.

Staying with the soundness property, we can classify existing approaches into three classes: (1) Some approaches exploit certain structural constraints of the business process model, for instance by focussing on workflow graphs that only consist of AND/XOR-gateways, for instance [3]. (2) Other approaches rely on the definition of soundness which can be defined in terms of standard Petri net properties such as boundedness, liveness, or the existence of place invariants [4]. The two mentioned approaches are *domain-specific* in the sense that they exploit the fact that they investigate business

process models. In contrast, (3) general purpose verification tools (usually called *model checkers* [5, 6]) can check all kinds of properties as long as they can be expressed in terms of temporal logics. As this is the case for soundness, these tools are also applicable for the verification of business process models.

By nature, only domain-specific approaches may exploit the special nature of business process models and their correctness criteria are best suited for corresponding verification tasks — in particular, since the approaches are specially tailored to the need of the modelers. In contrast, general purpose verification tools are not "aware" of the background of the property or the model under investigation and hence may only produce results of limited value. At the same time, the ongoing evolution of business process modeling languages, the growing number of aspects that need to be covered by a business process model, or the trend toward executable business process models, makes state-of-the-art business process model verification a moving target. As a consequence, specific approaches may become inapplicable for novel demands, leaving only general purpose approaches as stable tools for the future.

Goal. This paper tries to improve the applicability of general purpose approaches to business process models. We thereby try to combine the advantages of a vast set of supported correctness criteria (and hence, the flexibility to keep up with the fast evolution of novel modeling languages and correctness criteria) with the domain-specific diagnosis results of existing business process verification tools. This paper thereby can be seen as a follow-up to the reports for Fahland et al. [2] where comprehensive diagnosis results where only reported for domain-specific approaches, in particular [3].

Problem description. In principle, a model checker takes a formal model (e.g., a Petri net) and a formal description of the property to check (usually described by temporal logics) as input and tries to proof the property by an exhaustive investigation of the model's states. In case the property is violated (e.g., a deadlocking state is detected), a path to this error state is reported [5, 6]. The path contains all actions of the model that need to be executed to reach the error state from the initial state. Due to this operational nature of paths, the scenario that led to the error can be simulated. It is furthermore possible to explain the scenario in terms of the original model; that is, to map the states of the Petri net back to events of a BPMN model.

Unfortunately, the size of the paths correlates with the size of the model and paths of industrial models can thus be very long and hardly understandable. Furthermore, the path can contain a lot of irrelevant or diverting information that makes the comprehension of the error very difficult. For instance, the path usually contains actions that only "set up" the process (e.g., initializations and login procedures). These inevitable actions are certainly *necessary* to be able to reach the error state, but are usually not the *cause* of it. Another aspect that makes paths hard to understand is the fact that business process models may span several components where activities are executed in parallel. On the path, these originally unordered activities are reported in a fixed — and possibly arbitrary — order which may yield confusion due to unintuitive error descriptions.

Contribution. To solve the mentioned problems, this paper makes four contributions. First, we shorten paths by focussing on the choices made rather than on each individual action. Second, we perform additional verification steps to further reduce the path. Third, we exploit the concurrency of the model to undo the aforementioned arbitrary ordering and to express concurrent parts of the error independently of each other. Fourth, we take

the investigated property into account to remove aspects of the part which are irrelevant to the detected error. We shall use a large case study as experimental evaluation of our proposed approach.

Organization. The next section introduces the basic concepts we build our approach on, including Petri nets as formal model and a brief introduction to model checking. Section 3 introduces a novel representation of paths by focussing on the made choices. In Sect. 4, we discuss how the path can be further shortened by performing additional verification steps. The combination of paths and concurrency is described in Sect. 5. Section 6 demonstrations how the size of the resulting artifacts can be further reduced. All reduction steps are evaluated by experimental results with more than 1,000 industrial business process models. Finally, Sect. 7 summarizes the results and concludes the paper.

2 Preliminaries

2.1 Petri nets

Business process modeling languages are usually semiformal and hence are not directly applicable to a mathematically rigorous proof of correctness criteria. However, the operational semantics can be captured in formalisms such as Petri nets or process calculi. With the advent of executable languages such as WS-BPEL 2.0 or BPMN 2.0, such a formalization became much easier, because a precise execution semantics yielded more careful language specifications.

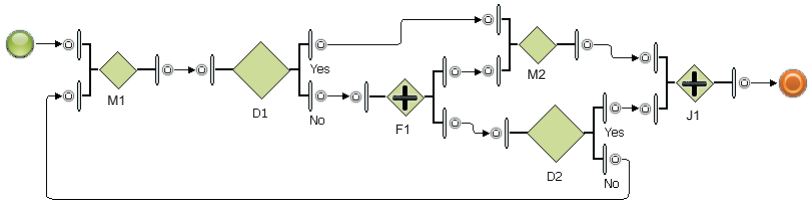
Our framework is based on *Petri nets* [7] and is hence not tied to a specific business process modeling language. In fact, for most of today's languages from industry or academia (including BPMN, WS-BPEL, UML activity diagrams, YAWL, or EPC), translations into Petri nets exists [8]. We chose Petri nets as formalism for two reasons: First, it is a *graphical formalism* that closely resembles languages such as BPMN and allows to easily translate findings from the original model into the Petri net model, and vice versa. Second, *concurrency* (i.e., the independent and yet parallel execution of actions) can be expressed naturally in terms of Petri nets. This is especially helpful as the behavior of a Petri net can be expressed by a set of *distributed runs*, an artifact we shall use in Sect. 5–6 of this paper.

Intuitively, a Petri net is a directed graph, consisting of active components called *transitions* (depicted as squares) which model actions and decisions of business processes and passive components called *places* (depicted as circles) which model locations of resources such as documents, messages, or the current control flow. The flow of resources is modeled by arcs between places and transitions, and vice versa. A state of a Petri net is expressed by a distribution of tokens (depicted by black dots) on the places (called a *marking*) which models the current presence of the respective resource. Formally:

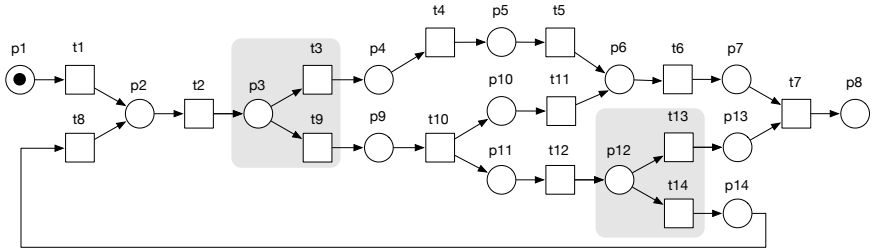
Definition 1 (Petri net). *A* Petri net *is a tuple* $N = [P, T, F, m_0]$ *where* P *is finite a set of places,* T *is finite a set of transitions* ($T \cap P \neq \emptyset$)*, a flow relation* $F \subseteq (P \times T) \cup (T \times P)$*, and an initial marking* $m_0 : P \to \mathbb{N}$.

Example. Figure 1(a) depicts a small business process model from [2] which contains two subtle control flow errors: a lack of synchronization and a local deadlock. Its translation into a Petri net is shown in Fig. 1(b). As we see, the structure is very similar to the original model. The concrete mapping from the models from [2] into Petri nets is described in [9].

The initial marking m_0 defines an initial distribution of tokens on the places. The marking can change by *firing* transitions.



(a) business process model from [2]



(b) Petri net model

Fig. 1. A business process model (a) and its translation into a Petri net (b)

Definition 2 (Firing rule). *Let $N = [P, T, F, m_0]$ be a Petri net, $t \in T$ be a transition, and $m : P \to \mathbb{N}$ be a marking of N. Transition t is* activated *in m (denoted $m \xrightarrow{t}$) iff $m(p) > 0$ for all $p \in {}^\bullet t$. An activated transition can* fire*, yielding a new marking m' (denoted $m \xrightarrow{t} m'$) with*

$$m'(p) = \begin{cases} m(p) + 1, & \text{iff } p \in t^\bullet \setminus {}^\bullet t, \\ m(p) - 1, & \text{iff } p \in {}^\bullet t \setminus t^\bullet, \\ m(p), & \text{otherwise.} \end{cases}$$

Thereby, let for a node $x \in P \cup T$ be ${}^\bullet x = \{y \mid [y, x] \in F\}$ the preset *of x and $x^\bullet = \{y \mid [x, y] \in F\}$ be the* postset *of x.*

The behavior of a Petri net is defined by the *reachability graph* which has all reachable markings as nodes, m_0 as initial node, and a t-labeled edge between node m and m' iff $m \xrightarrow{t} m'$. The reachability graph is a very versatile tool when investigating the behavior of Petri net models as all interesting properties of Petri nets can be checked using this reachability graph. This includes the most prominent correctness criteria for business process models such as soundness.

2.2 Model checking

Model checking [5, 6] is an approach to proof that a system satisfied a given correctness criterion; for instance soundness, the absence of a deadlocking state, the presence of a sound process configuration, correct data life cycles, or compliance to business rules. In contrast to *theorem provers*, which sometimes need the manual inputs, or *testing*, which can only proof the existence of errors, but never their absence, model checking is an automated and complete way to investigate systems.

In this paper, we assume that the model under investigation is given as a Petri net. The correctness criterion is typically motivated by the domain of the original model (i.e., a business process). For an automated check, this correctness criterion needs to be expressed in terms of a *temporal logics*. Temporal logics extend classical propositional logics by temporal operators that express the relationships of propositions (i.e., that a

state B is reached *after* state A is reached) and path quantifiers that express whether some or all successors of a state need to satisfy a property. The most prominent temporal logics used in model checking is CTL*. We refrain from a formal definition, as we shall concentrate on the evaluation of errors in incorrect systems rather than in the correctness criteria and their formalization themselves. Detailed introductions provide [5, 6].

For the remainder of the paper, we assume the existence of a model checking tool that takes a Petri net N and a temporal logical formula φ as input. If the formula is satisfied by the Petri net (e.g., if the Petri net is sound), this is reported as "yes" to the modeler. In case the formula is violated (e.g., a deadlocking marking m is found), this is reported as "no" to the modeler. In addition, a path $\pi = t_1 \cdots t_n$ is given to the modeler which explains how m is reachable from the initial marking m_0; that is, $m_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} m$. Depending on the nature of the formula φ, the marking reached by the reported path either is a proof that the formula is not satisfied by the behavior of the Petri net N and is called a *counterexample* or marking itself is the proof that the formula is satisfied (e.g., if φ expresses the reachability of that marking m) and is called a *witness*. In this paper, we do not distinguish the semantics of the marking m and always refer to m as *goal marking*.

Example (cont.). The business process from Fig. 1(a) has a lack of synchronization. This can be detected by checking the Petri net from Fig. 1(b). The following path π describes how a marking m can be reached which puts two tokens on place p_6.

$$\pi = t_1\ t_2\ t_9\ t_{10}\ t_{11}\ t_{12}\ t_{14}\ t_8\ t_2\ t_3\ t_4\ t_5 \qquad m = \{p_6 \mapsto 2\}$$

The path contains 12 transitions. In the remainder of this paper, we use this path to exemplify the proposed reductions.

It is worthwhile to mention that model checking suffers a devastating worst case complexity due to the well-known state explosion problem which yields reachability graphs with exponential blow-ups compared to the size of the Petri nets. However, even industrial business process models can be model checked in few microseconds, because heuristics that fight the state space explosion proved to be very effective in this domain [2].

3 Representing paths by made choices

3.1 The problem: long paths = big problems

In the remainder of the paper, we focus on the following problem:

> Given a path π to a goal marking m of a Petri net model N, how can the reason for the error modeled by m be briefly and comprehensively explained to the modeler of N?

Apparently, π describes how the goal marking m can be reached from the initial marking m_0 of N. Consequently, reporting the transitions of π together with the intermediated markings to the modeler should help to understand the reasons m was reached. Unfortunately, this approach is futile in case π contains dozens of transitions. The reasons for such long paths are:

Detours: Model checkers usually investigate the markings of a Petri net in a depth first search. As a result, the reported paths do not need to be optimal and may contain some transitions that model "detours" in the reachability graph that do not contribute in the actual reaching of the goal marking. Note that breadth-first approaches are not applicable to many classes of formulae.

Interleaving of concurrent transitions: A marking of N may activate two transitions t_1 and t_2 which are not mutually exclusive. That is, firing either transition first does not disable the other one. A typical reason for this is that t_1 and t_2 do not share any resources. Consequently, the order in which t_1 and t_2 occur on the path π is arbitrary. If each transition belongs to different components of the underlying business process model, then these arbitrary interleaving of the transitions may be irritating to the modeler if she tries to understand the path π. In the example path, transition t_{11} and t_{12} are concurrent and the reported order in path π (t_{11} before t_{12}) is arbitrary.

Indisputable parts: Though the path π is an actual proof *that* the goal marking m can be reached in N, not every transition on the path is an actual cause of m. In the example process, *any* path will begin with firing t_1 and hence does not need to be reported to the modeler as reason for an error.

3.2 The solution: don't report the obvious

To tackle the problem of long paths with redundant or unhelpful information, we shall exploit two aspects to shorten paths in the remainder of this chapter: *progress* and *conflicts*.

Progress is the assumption that the model never "gets stuck" in case a transition is activated. That is, if a marking activates one or more transitions, then this marking is eventually left by firing on of these transitions. Progress is a natural assumption for business process models in which the execution of tasks also cannot be postponed indefinitely. Though the actual occurrence of message or timer events cannot be precisely predicted, the respective states are always assumed to be eventually left by the modeled actions.

A *conflict* is a situation in which there exist more than one possible continuations. In terms of Petri nets, it is a marking in which two transitions t_1 and t_2 are enabled, but after firing either of them, the other transition is disabled. This situation is dual to concurrent transitions (see above) that do not disable each other. A detailed discussion of these aspects can be found in [7].

The combination of these aspects brings us to the following intuitive observation: *Only the conflicts on the path π carry information on how to reach the goal markings.* Any other marking m on the path between the initial and the goal marking either (1) enables no transition: Then this must be the goal marking itself, because it has no successor marking. Alternatively, (2) marking m enables exactly one transition: Then this transition is eventually fired due to the assumption of progress. Consequently, this transition does not need to be reported to the modeler as its firing was already determined by the previous transition on π thad lead to m. Finally, (3) marking m enables several concurrent transitions. These transitions may fire independently, and if all of them are on π, then the exact order is arbitrary.

In the remainder of this section, we shall give a formal definition of conflicts and sketch an algorithm to reduce paths based on these conflicts. Finally, we shall report on experimental results applying this algorithm to thousands of business processes. We shall first formalize a conflict:

Definition 3 (Conflict marking, conflict transition). *Let m be a marking with $m \xrightarrow{t_1}$ and $m \xrightarrow{t_2}$. Marking m is a* conflict marking *and t_1 and t_2 are* conflict transitions *iff (1) $m \xrightarrow{t_1} m_1$, (2) $m \xrightarrow{t_2} m_2$, and (3) $m_1 \not\xrightarrow{t_2}$ or $m_2 \not\xrightarrow{t_1}$.*

The above definition relies on markings. However, conflict transitions can be approximated using the structure of the Petri net. Intuitively, transitions may be conflict transitions if they share a place in their presets. Desel and Esparza [10] extended this observation toward a decomposition of a Petri net into its *conflict clusters*.

Definition 4 (Conflict cluster). *Let $x \in P \cup T$ be a node of a Petri net. The* conflict cluster *of x, denoted $[x]$ is the minimal set of nodes such that: (1) $x \in [x]$. (2) If $p \in P$ and $p \in [x]$, then $p^\bullet \subseteq [x]$. (3) If $t \in T$ and $t \in [x]$, then ${}^\bullet t \subseteq [x]$.*

The conflict clusters of a Petri net can be determined by a union-find-algorithm with effectively constant amortized time complexity.

Note that *free-choice Petri nets* [10] have the following property: If one transition in a conflict cluster is activated in a marking m, then m activates *all* transitions of that conflict cluster. That is, an additional check is not required. However, not all aspects of business process models can be formalized using free-choice Petri nets, for instance errors, complex gateways, or timeouts. To this end, we decided not to constrain our approach to this class of Petri nets, but to make it applicable to arbitrary Petri nets. However, checking whether a transition is activated given a concrete marking has linear complexity in the size of the net and can usually be assumed to be constant as transitions hardly have all places in their preset, but only a very small subset.

Now we can reduce the path π as follows:

1. Calculate the conflict clusters of N.
2. For each transition t of π activated by a marking m reached by a (possibly empty) prefix of π: Report t as part of the reduced path if and only if t is a conflict transition; that is, if and only if $\{t' \in T \mid t' \in [t] \ \wedge \ m \xrightarrow{t'}\} \neq \{t\}$.

We discuss the implementation of this algorithm in Sect. 7.

Example (cont.). The conflict clusters with more than one transition of our running example are shaded gray in Fig. 1(b): transitions t_3 and t_9, as well as t_{13} and t_{14} are conflicting. Consequently, we can reduce the path π as follows:

$$\pi_{\text{reduced}} = t_9 \ t_{14} \ t_3 \qquad m = \{p_6 \mapsto 2\}$$

The firing of all other transitions is clear from the context from the intermediate markings and the assumption of progress. Note that the transition names need to be translated back into the terms of the original model. A different representation of π_{reduced} could be: "After (1) decision D1: *No*, (2) decision D2: *No*, and (3) decision D1: *Yes*, a lack of synchronization occurs after after merge M2."

3.3 Experimental results

To evaluate the path reduction algorithm, we applied it to a large collection of industrial process models created by IBM customers using the *IBM WebSphere Business Modeler*. The models were first presented in a report by Fahland et al. [2], where the 1386 process models were checked for soundness using different approaches. As one general-purpose model checker, *LoLA* [11], took part in this investigation, the process models were also translated into Petri nets.[3] The models are partitioned into five libraries (A, B1, B2, B3,

[3] The original models and their Petri net translations are available for download at http://service-technology.org/soundness.

library	A	B1	B2	B3	C
avg. path length before / after	17.51 / 1.83	17.52 / 2.11	16.06 / 1.54	20.34 / 1.67	13.40 / 2.30
max. path length before / after	53 / 8	66 / 7	56 / 6	54 / 5	21 / 3
sum of path lengths before / after	1699 / 178	1419 / 171	1349 / 129	1688 / 139	134 / 23
reduction	89.52 %	87.95 %	90.44 %	91.77 %	82.84 %

Table 1. Paths from the checks for local deadlocks

library	A	B1	B2	B3	C
avg. path length before / after	30.83 / 3.17	10.47 / 0.66	12.16 / 0.68	11.50 / 0.59	51.00 / 7.57
max. path length before / after	89 / 13	52 / 7	100 / 8	103 / 14	120 / 17
sum of path lengths before / after	1079 / 111	1047 / 66	1459 / 82	1507 / 77	357 / 53
reduction	89.71 %	93.70 %	94.38 %	94.89 %	85.15 %

Table 2. Paths from the checks for lack of synchronization

library	A	B1	B2	B3	C
avg. path length before / after	12.06 / 2.79	13.82 / 2.55	18.13 / 2.33	14.27 / 2.55	11.27 / 2.33
max. path length before / after	44 / 7	70 / 7	95 / 7	95 / 7	27 / 3
sum of path lengths before / after	19699 / 4557	5707 / 1054	13835 / 1777	17494 / 3130	169 / 35
reduction	76.87 %	81.53 %	87.16 %	82.11 %	79.29 %

Table 3. Paths from the checks for noninterference

C) and stem from different business areas, ranging from financial services, automotive, telecommunications, construction, supply chain, health care, and customer relationship management.

Soundness. Using these models, we repeated the soundness checks to created paths for those Petri nets with unsound behavior. In the original report [2], each Petri net was checked twice to proof soundness: once for weak termination (i.e., whether the final marking is reachable from every reachable marking) to rule out *local deadlocks* and once for unsafe markings (i.e., whether a marking m is reachable with $m(p) > 1$ for a place p) to rule out *lack of synchronization*.

From the 1386 models, 642 control-flow errors were found — 355 Petri nets were not weakly terminating and 393 Petri nets contained unsafe markings.[4] Consequently, we could apply our reduction to 748 paths.

Table 1 summarizes the results from the reduction of the paths for Petri nets with local deadlocks. We list, for each library, the average path length, the maximal path length, and the sum of all path lengths for the respective library — once before and once after the reduction. The numbers suggest that the reduction is very effective: The average path length could be reduced from 13–20 transitions to 1.5–2.3 transitions. This means a reduction of 82–91 %. Table 2 reports similar results for Petri nets with a lack of synchronization. In summary, the longest path for a soundness violation contains at most 17 transitions, compared to 120 before the reduction.

Information flow security. Furthermore, the same business process models were used in a recent report [12] on information flow security. In this case study, *noninterference* [13] was verified. This correctness criterion ensures that decisions from a secure domain cannot be reproduced by investigating public runtime information of the business process. To perform this check, each business process model needed to be checked

[4] 24 Petri nets had both kind of errors and hence failed both checks.

several times: For each participant (i.e., swimlane of the process), one check is required. In that case study, 4050 errors were reported, yielding 4050 paths to investigate.[5]

Table 3 summarizes the reduction results for the paths. Again, we can report a reduction between 76–87 %. The maximal reduced path of the whole case study consists only of 7 transitions, whereas it was 95 transitions before the reduction. On average, not more than 2.79 transitions are reported per detected error.

The experiments report promising results. Though the reduced paths consist of Petri net transitions, they can be easily translated back into the nomenclature of the original model. For each model translated from the IBM WebSphere Business Modeler into a Petri net, a file was created that maps the Petri net nodes to a construct of the original model, see [9]. Consequently, conflict transitions can be easily linked to the respective gateways.

4 Further reduction: remove spurious conflicts

4.1 Motivation and formalization

In the previous section, we showed how paths to errors in business process models can be reduced by only reporting conflict transitions. This reduction decided, for each marking that activates a transition, whether conflicting transitions are also activated. This check is local in the sense that it is not checked whether those transitions that were not taken in the decisions actually could have avoided the next conflict transition on the path.

We can formalize this idea as follows:

Definition 5 (Spurious conflict). *Let $\pi = t_1 \cdots t_n$ be a reduced path and, for $0 \leq i < n$ be m_i the marking that is reached from m_0 by firing the first i transitions of π. In marking m_i, transition t_i is a* spurious conflict *iff, for all $t \in [t_i] \cap T$ with $t \neq t_i$ and $m_i \xrightarrow{t}$ holds: $m_i \xrightarrow{t} m_i'$ and for all paths beginning with m_i', marking m_{i+1} is eventually reached, activating the next conflict transition t_{i+1} on π.*

Intuitively, a transition t_i on a reduced path π is a spurious conflict iff every transition t in conflict to t_i eventually reaches the marking m_{i+1} which enables the next transition t_{i+1} on path π. In this case, choosing any transition from the conflict cluster $[t_i]$ will eventually enable the next conflict on the path to the goal state. Consequently, reporting the spurious conflict t_i is of little help to the modeler to understand the error itself.

The check for spurious transitions defined above can be straightforwardly be implemented using a model checker.[6] We integrated this check as postprocessing step after reducing the paths as described in the previous section. Note that executing a model checker can be very time and memory consuming. However, even if a check is not finished with a reasonable amount of resources, we just failed to proof whether a conflict is spurious and can continue with the investigation of the next transition. That said, the postpocessing can be aborted at any time — any intermediate result is still correct.

[5] The original models were not designed with noninterference in mind. However, the authors of [12] decided to use the processes from [2] as case study to investigate their algorithms.

[6] We check whether N with initial marking m_i' satisfies the CTL formula $\varphi = \mathbf{AF}\, m_{i+1}$.

library	A	B1	B2	B3	C
avg. path length before / after	1.84 / 0.91	2.11 / 0.67	1.54 / 0.57	1.67 / 0.41	2.30 / 0.90
max. path length before / after	8 / 2	7 / 1	6 / 1	5 / 1	3 / 1
sum of path lengths before / after	178 / 88	171 / 54	129 / 49	139 / 34	23 / 10
reduction	50.56 %	68.42 %	62.79 %	75.54 %	60.87 %
aborted checks	1	0	0	0	0

Table 4. Reduced paths from the checks for local deadlocks

library	A	B1	B2	B3	C
avg. path length before / after	3.17 / 0.86	0.66 / 0.17	0.68 / 0.14	0.59 / 0.09	7.57 / 1.00
max. path length before / after	13 / 2	7 / 2	8 / 2	14 / 2	17 / 2
sum of path lengths before / after	111 / 30	66 / 17	82 / 17	72 / 12	53 / 7
reduction	72.97 %	54.55 %	79.27 %	84.42 %	86.79 %
aborted checks	1	4	0	0	4

Table 5. Reduced paths from the checks for lack of synchronization

library	A	B1	B2	B3	C
avg. path length before / after	2.79 / 0.99	2.55 / 0.75	2.33 / 0.55	2.55 / 0.63	2.33 / 0.40
max. path length before / after	7 / 2	7 / 2	7 / 2	7 / 2	3 / 1
sum of path lengths before / after	4557 / 1614	1054 / 310	1777 / 423	3130 / 772	35 / 6
reduction	64.58 %	70.59 %	76.20 %	75.34 %	82.86 %
aborted checks	12	4	4	7	0

Table 6. Reduced paths from the checks for noninterference

4.2 Experimental results

We applied the reduction of spurious conflicts to the case studies described in the previous section. Table 4.2–5 summarize the results. In all three experiments, the paths could be *further* reduced by 50–86%. Now, in all experiments, at most two transitions are reported as to reach the error. All other transitions are either nonconflicting or are spurious conflicts for which *any* resolution eventually reaches the next conflict on the path. Note that in some cases, the check for spurious conflicts has been aborted after more than 2 GB of memory were consumed. In these cases, the conflict was kept in the path and the check proceeded with the next conflict.

5 Combining paths and concurrency

5.1 Motivation and formalization

So far, we focused on reducing paths by removing any transitions whose firing provides no information to the modeler on why the goal state was actually reached. Thereby, we could exploit the Petri net structure to calculate conflict clusters to identify possible conflict transitions. This allowed for a quick check whether a transition is actually a conflict.

However, we still considered paths as a *sequences* of transitions leading to the goal state. As discussed earlier, this sequence may be an arbitrary linearization of originally concurrent behavior. Therefore, communicating paths to the modeler — for instance by means of animation or simulation — still uses this arbitrary and hence unintuitive ordering. This may be especially confusing if the underlying process spans several components (e.g., participants of a choreography or lanes) and the path constantly switches between actions of different components.

To this end, this section aims at exploiting the concurrency of the Petri net model and to use it to reorganize paths. We thereby try to undo the arbitrary ordering and to provide partially-ordered paths, called *distributed runs* in the literature [7]. As sketched earlier, two transitions t_1 and t_2 may fire concurrently in a marking if they do not disable each other; that is, any ordering of t_1 and t_2 are possible. The definition of Petri nets further ensure that firing any order of concurrent transitions yield the same marking. This has one interesting effect: by depicting concurrent transitions as concurrent (i.e., unordered), a distributed run implicitly expresses all possible orderings of these transitions.

Before we continue, we formalize the concept of a distributed run. The underlying structure of such a distributed run is a *causal net*:

Definition 6 (Causal net). *A* causal net *is a Petri net $C = [P, T, F]$ without initial marking such that (1) for each place p holds: $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$, (2) the transitive closure F^+ of the flow relation F is irreflexive, and (3) any node has only finitely many predecessors with respect to F^+.*

Intuitively, a causal net is (1) conflict free and begins and ends with places, (2) is acyclic, and (3) the prefix of any element is finite. A causal net does not have an initial marking — its places with empty preset represent initially marked places. This becomes clear in the definition of a distributed run of a Petri net N, defined as follows:

Definition 7 (Distributed run). *Let $N = [P_N, T_N, F_N, m_0]$ be a Petri net, $C = [P_C, T_C, F_C]$ be a causal net, and $\beta \subseteq (P_C \times P_N) \cup (T_C \times T_N)$ be a mapping. Further assume, without any loss of generality, that $m_0(p) \leq 1$ for all $p \in P_N$. The pair $[C, \beta]$ is a* distributed run *of N iff: (1) for all $p_C \in P_C$ with ${}^\bullet p_C = \emptyset$ holds: $m_0(\beta(p_C)) = 1$ and (2) for each $t_C \in T_C$ with $\beta(t_C) = t_N$ holds: β bijectively maps ${}^\bullet t_C$ to ${}^\bullet t_N$ and $t_C^\bullet$ to $t_N^\bullet$.*

A distributed run is a causal net C whose nodes are mapped to those of a Petri net, such that (1) those places of C without predecessors map to the initially marked places of N and (2) the preset and postset of a transition of C bijectively maps to the preset and postset of the respective transition of N.

5.2 Translating paths into distributed runs

Intuitively, we can translate a path into a distributed run by copying fired transitions with their preset and postset to the distributed run and "glue" those places representing resources created by one transition and consumed by another transition.

In more detail, a path π of a Petri net N can be translated into a distributed run $[C, \beta]$ as follows: First, add, for each initially marked place p_N of N, a place p_C to P_C and define $\beta(p_C) = p_N$. Then, for each firing $m \xrightarrow{t_N} m'$ in π, (1) add a transition t_C to T_C and define $\beta(t_C) = t_N$, (2) for each place $p_N \in {}^\bullet t_N$, find a place p_C of C with $\beta(p_C) = p_N$ and $p_C^\bullet = \emptyset$ and add an arc $[p_C, t_C]$ to F_C, and (3) for each place $p_N \in t_N^\bullet$ add a place p_C to P_C and define $\beta(p_C) = p_N$ and add an arc $[t_C, p_C]$ to F_C.

As paths are acyclic, the created distributed run is finite by definition. Furthermore, paths are conflict-free (i.e., every intermediate marking has exactly one successor) such that the created Petri net structure is indeed a causal net.

The translation into a distributed run now allows for a reasoning of the relationship between occurrences of transitions on the path. We distinguish two cases: On the one hand, if there is a directed path between t_1 and t_2, then $\beta(t_1)$ was fired *causally before*

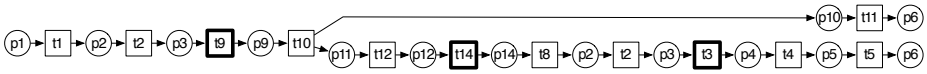


Fig. 2. The path π as a distributed run with highlighted conflict transitions

$\beta(t_2)$. On the other hand, $\beta(t_1)$ and $\beta(t_2)$ were fired *concurrently*, if there exists neither a path from t_1 to t_2 nor from t_2 to t_1. In this case, the order on path π was arbitrary and should not be reported as such to the modeler.

Example (cont.). Figure 2 depicts path π as distributed run. Note that the cycle in the model is unfolded, yielding two copies of transition t_2. Two places p_6 without successors model the target marking $\{p_6 \mapsto 2\}$. Furthermore, note transition t_{11} is displayed concurrently to all transitions following t_{10}.

5.3 Applying the conflict reduction to distributed runs

We implemented the translation of paths into distributed runs. This construction algorithm only works for unreduced paths, because it requires that all intermediate markings are used to create places in the underlying causal net. Therefore, the reduction reported in Sect. 3–4 are not directly applicable.

To combine the advantages of both approaches — that is, reducing paths by removing nonconflicting transitions on the one hand and not ordering concurrent transitions on the other hand — we exploit the two relationships (causal order and concurrency) from above and create an artifact (called *reduced distributed run*[7]) with the following properties:

1. For each initially marked place of N, it contains a place with empty preset and the respective labeling.
2. For each place marked by the goal marking reached by π, it contains a place with empty postset and the respective labeling.
3. For each conflict transition (i.e., transitions that were not removed by the reductions in Sect. 3–4), it contains a transition with the respective labeling.
4. For each transition consuming a token from the initial marking or producing a token to the goal marking, it contains a transition with the respective labeling and the respective arcs to the places in the preset and postset.
5. For each two transitions t_1 and t_2, add a dashed arc $[t_1, t_2]$ if we can derive from the distributed run that t_1 is causally before t_2.
6. Transitively reduce the dashed arcs; that is, remove all dashed arcs $[t_1, t_3]$ for which there exists arcs $[t_1, t_2]$ and $[t_2, t_3]$.

As reduced distributed runs have more or less the same size as the reduced paths, we refrain from a detailed discussion of a case study.

[7] In fact, the described artifact is not a distributed run. Though it shares properties of distributed runs, we decided to stick to the name as it is most intuitive.

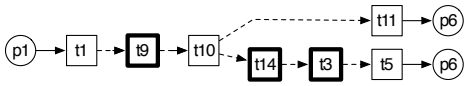


Fig. 3. The reduced path π as a reduced distributed run with highlighted conflict transitions

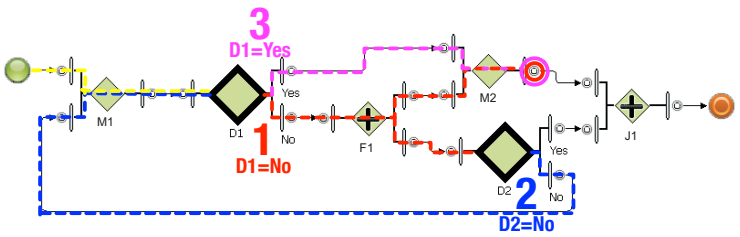


Fig. 4. Mapping back the reduced distributed run to the original process model

Example (cont.). An example is depicted in Fig. 3. It explains how the initial marking $\{p_1 \mapsto 1\}$ is transformed into the goal marking $\{p_6 \mapsto 2\}$. In case a transition consumes from the initial marking or produces to the goal marking, it is reported explicitly. Furthermore, the resolved conflicts on the original path π are reported, and their causal order is depicted by dashed arcs. Figure 4 further depicts an example how the information of a reduced distributed run can be used for a visualization of a path in a concrete business process model.

6 Cropping distributed runs

This section introduces a further reduction for distributed runs, that can be combined to reduced distributed runs as described in the previous section. We shall first concentrate on unreduced distributed runs.

When a path π is translated into a distributed run, it is a precise description on how the initial marking is translated into the target marking. However, usually only a parametrized description of the target marking is given to the model checker, for instance a formula expressing "Find a marking m such that $m(p) > 1$ for a any place p." in case of checking the absence of lack of synchronization. In case a goal marking m is found that does mark a place with more than one token, then m usually also marks other places of the Petri net. These places are, however, not relevant to the proof that unsafe markings are reachable in the Petri net. Therefore, it would be of more value if the distributed run could be "cropped" such that it only contains those places and transitions in the prefix of the unsafe places.

As each node of a causal net only has finitely many predecessors, the cropped prefix of a set of nodes is well-defined. Any other nodes that are not on this prefix can be removed. The result is still a causal net, but violates the definition of a distributed run. For the sake of a uniform nomenclature, we refer to this artifact as *cropped distributed run*. Note that reduced distributed runs can be cropped as well, producing *cropped reduced distributed runs*.

Which places are used to crop the distributed run depends on the property under investigation. We gave a straightforward example for the check for lack of synchro-

nization. For the noninterference check, the marking of a specific goal place signals a security flaw — consequently, this place can be used to crop the distributed run. For local deadlocks, this choice is not straightforward, because the final marking of the Petri net is actually unreachable in the reported goal marking. A starting point to crop distributed runs is subject of future research.

7 Concluding remarks

7.1 Summary

In this paper, we investigated how the output of model checking tools — usually a path from the initial state to a state modeling an error — can be briefly and comprehensively explained to the modeler. We presented four reductions — each focussing on a different aspect of the problem:

1. In Sect. 3, we removed all transitions whose firing is totally determined by the current marking, because there are no activated conflict transitions. As a result, we explain errors not by the complete path from the initial to the goal state, but only explain which choices on the way lead to the goal state.
2. In Sect. 4, we further removed those choices where any continuation eventually reaches the next choice on the path. This postprocessing step required additional verification runs which can be stopped at any time without jeopardizing correctness. Though the reduction seems technical, it is actually very effective in the investigated case study.
3. The underlying concurrency of the model was exploited in Sect. 5. There, we create a distributed run from the path in which concurrent transitions are not any more artificially ordered. We further demonstrated how distributed runs can be combined with the previous reductions.
4. A final reduction is presented in Sect. 6: The verification problem usually concentrates on few places of the Petri net. Distributed runs allow to remove all aspects that are irrelevant to the goal marking.

All reported reductions were implemented in a tool *Pathify* which bases on the *Petri Net API* [14] to calculate conflict clusters and can process Petri nets and paths from the *LoLA* model checking tool [11]. The tool, together with the Petri nets from the case studies can be downloaded from http://www.pirat.ly/25wg2.

Note that our approach heavily relies on Petri nets and their concise semantics, a natural expression of concurrency and conflict relation, efficient algorithms, and a notion of distributed runs. These features are not available by other formalisms or modeling languages. At the same time, we are not bound to a specific input modeling language as most business process modeling languages can be translated into Petri nets.

7.2 Related work

The analysis and verification of business process models is a broad field of research. Consequently, there exists a variety of domain-specific approaches (e.g., the decomposition of workflow graphs into SESE regions to check soundness [3]). However, we are not aware of other approaches that postprocess error information from general purpose model checkers to explain these errors to the modelers. In particular, most approaches consider only a subset of the modeling language's features (e.g., BPMN without fault

handling). In contrast, our presented approach is applicable to any verification approach that produces witness paths.

Related to the presentation of error information is the automated correction of flawed business process models [15, 16]. These approaches use similarity metrics to find a correct business process model which maximally resembles the flawed model. These approaches have the benefit of avoiding lengthy manual correction steps altogether.

Back annotation of execution sequences. The problem studied in this paper is similar to *model-based analysis* in Software Engineering which is the problem of translating a (high-level) domain model (of a generic software system) into a formal model and analyzing for various properties and problems [17]. Also there, the open problem is to make the analysis result obtained on the formal level understandable by a domain expert [18]. One generic approach to relate traces of the formal model to model elements of the domain model is *back annotation* [19]. Here, the model transformation from domain model to formal model is reversed to translate steps of the formal model to steps or elements of the domain model. However, mismatches in granularity and semantics complicate the translation from one to another, requiring customized solution for each case. The technique proposed in this paper is orthogonal: rather than trying to translate entire traces, we have shown that the diagnostic information of the formal trace can be reduced to an essential minimum which is easier to map. Though a systematic integration with the back annotation approach is left open here.

This paper considered traces generated by verification and validation tools from given models. Process mining considers traces recording the actual process execution. Here, *conformance checking* is the problem of detecting how and where traces deviate from process models [20]. Deviations can be highlighted on the traces and the process model directly [21]. Also, branching processes of Petri nets can be used to greatly simplify process models in *process discovery* [22]. It is an open question whether the reduction techniques presented in this paper can used to improve the diagnostic information in conformance checking and results in process mining.

7.3 Future work

In this paper, we focused on reducing paths to error states and neglected the retranslation into the original business process model. Visualizations such as Fig. 4, possibly enriched with animations, need to be automatized and evaluated by business process modelers. Here, understandability criteria [23] could be of great value. However, this was out of scope of this paper which aimed at evaluating the idea of using conflicts to reduce paths with three experimental setups checking different correctness criteria with thousands of industrial business process models.

Beside better visualizations, also the investigation of further correctness criteria is a direction of future work. In particular the cropping of distributed runs appears to be a promising approach to help the modeler focus on the original causes of an error. Another aspect of this investigation is a better localization of errors — in particular, any behavior where the avoidance of an error is still possible should be left out, allowing to better spot the action or decision in the model that makes the error inevitable.

We see in this paper a first step toward a diagnosis framework which uses general purpose verification tools to verify business process models. As motivated in the introduction, domain-specific approaches are very closely coupled to the structure or the property under investigation, but may become inapplicable for future developments. In

contrast, the modularization (a translation into Petri nets as frontend, a general purpose model checking tool as middleware, and a diagnosis framework as backend) may be more flexible when it comes to novel business process languages and properties.

References

1. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
2. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: BPM 2009. LNCS 5701, Springer (2009) 278–293
3. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: ICSOC 2007. LNCS 4749, Springer (2007) 43–55
4. Verbeek, H.M.W., Basten, T., Aalst, W.M.P.v.d.: Diagnosing workflow processes using Woflan. Comput. J. **44**(4) (2001) 246–279
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
6. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
7. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
8. Lohmann, N., Verbeek, H., Dijkman, R.M.: Petri net transformations for business processes – a survey. LNCS ToPNoC **II**(5460) (2009) 46–63
9. Fahland, D.: Translating UML2 activity diagrams to Petri nets. Informatik-Berichte 226, Humboldt-Universität zu Berlin, Berlin, Germany (2008)
10. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press (1995)
11. Wolf, K.: Generating Petri net state spaces. In: ICATPN 2007. LNCS 4546, Springer (2007) 29–42
12. Accorsi, R., Lehmann, A.: Automatic information flow analysis of business process models. In: BPM 2012. LNCS 7481, Springer (2012) 172–187
13. Busi, N., Gorrieri, R.: Structural non-interference in elementary and trace nets. Mathematical Structures in Computer Science **19**(6) (2009) 1065–1090
14. Lohmann, N., Mennicke, S., Sura, C.: The Petri Net API: A collection of Petri net-related functions. In: AWPN 2010. CEUR Workshop Proceedings 643, CEUR-WS.org (2010) 148–155
15. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: BPM 2008. LNCS 5240, Springer (2008) 132–147
16. Gambini, M., La Rosa, M., Migliorini, S., ter Hofstede, A.: Automated error correction of business process models. In: BPM 2011. LNCS 6896, Springer (2011) 148–165
17. Bondavalli, A., Cin, M.D., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability analysis in the early phases of uml-based system design. Comput. Syst. Sci. Eng. **16**(5) (2001) 265–275
18. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of uml models. In: ASE 2002, IEEE Computer Society (2002) 267–270
19. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of simulation traces with change-driven model transformations. In: SEFM 2010, IEEE Computer Society (2010) 145–155
20. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
21. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
22. Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Inf. Syst. **38**(4) (2013) 585–605
23. Mendling, J., Reijers, H.A., Cardoso, J.: What makes process models understandable? In: BPM 2007. LNCS 4714, Springer (2007) 48–63