Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models

Dirk Fahland[a], Cédric Favre[b], Jana Koehler[b], Niels Lohmann[c],
Hagen Völzer[b], Karsten Wolf[c]

[a]*Humboldt-Universität zu Berlin, Institut für Informatik,*
Unter den Linden 6, 10099 Berlin, Germany
[b]*IBM Research — Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland*
[c]*Universität Rostock, Institut für Informatik, 18051 Rostock, Germany*

Abstract

We report on a case study on control-flow analysis of business process models. We checked 735 industrial business process models from financial services, telecommunications, and other domains. We investigated these models for soundness (absence of deadlock and lack of synchronization) using three different approaches: the business process verification tool Woflan, the Petri net model checker LoLA, and a recently developed technique based on SESE decomposition. We evaluate the various techniques used by these approaches in terms of their ability of accelerating the check. Our results show that industrial business process models can be checked in a few milliseconds, which enables tight integration of modeling with control-flow analysis. We also briefly compare the diagnostic information delivered by the different approaches and report some first insights from industrial applications.

1. Introduction

Various studies have shown that many business process models contain control-flow errors such as deadlocks, see for example [1] for an overview. Such errors obstruct the correct simulation, code generation, and execution of these models. Therefore, detecting and removing control-flow errors becomes crucial in view of the increasing popularity of these use cases. Preventing errors by using a restricted, for example a purely block-oriented modeling language is rarely an option because a model typically needs to reflect the real causal process structures present in an enterprise.

In this paper, we are interested in checking business process models for the classical notion of soundness [2, 3], which entails the absence of deadlocks and lack of synchronization, which are explained in more detail later. Our interest in soundness is motivated by an increased need in creating business process models not only for documentation purposes, but also for an input into a translation and code generation process where, for instance, WS-BPEL code is generated. Classical soundness is used for example as a precondition to map a process modeled in a graph-based language, such as UML Activity Diagrams or BPMN, into WS-BPEL in a way that preserves the execution semantics *and* the structure of the process, cf. [4]. This use case requires a process to be checked in a relatively short amount of time, say 500 ms or less, because checks are to be performed on each major modification, at least on each save operation on the process model. Moreover, entire libraries of up to several hundred processes have to be checked when models are exchanged between modeling tools. Short response times make it possible to integrate control-flow analysis tightly with modeling such that errors are found at the earliest possible time, which would allow the user to relate an error to the latest change in the model. Furthermore, use cases such as code generation from models also require that an analysis produces sufficient diagnostic information to allow the user to locate and repair the detected errors.

A variety of techniques for checking soundness exists in the literature. They differ in their completeness, worst-case complexity, and quality of diagnostic information returned. Most techniques can be easily combined to optimize performance. The most flexible technique is state space exploration. It is most likely applicable to other similar use cases, such as checking a relaxed notion of soundness or checking more expressive languages supporting OR-joins and other advanced synchronization constructs. However, state space exploration suffers from the state space explosion problem, i. e., the fact that the number of reachable states can be exponential in the size of the process model. On the other hand, many business process models have a simple structure, for instance, they are sequential to a large extent, hence they do not necessarily have a large state space.

At the onset of our project, it was not clear from the literature how large the state spaces of control-flow models of realistic business processes are and hence which additional techniques are needed to check their soundness as fast as required by our use case. It was open whether such a check can be performed in the required time and in such a way that sufficient diagnostic information is obtained. In addition, given the variety of available approaches, it was unclear which would be the most suitable techniques.

In this case study, we investigated three approaches implemented in three different tools as outlined in Fig. 1:

1. The Petri net model checker LoLA [5], from which we used CTL model checking with partial order reduction.
2. The business process verification tool Woflan [3], which uses a mixture of Petri net analysis techniques, most notably structural Petri net reduction and S-coverability analysis, as well as a form of state space exploration based on coverability trees.
3. The process validation technique used in IBM WebSphere Business Modeler, which combines SESE decomposition [6] with heuristics and state space exploration.

The data set for our case study was a large collection of process libraries available in the IBM WebSphere Business Modeler tool. The first two approaches required

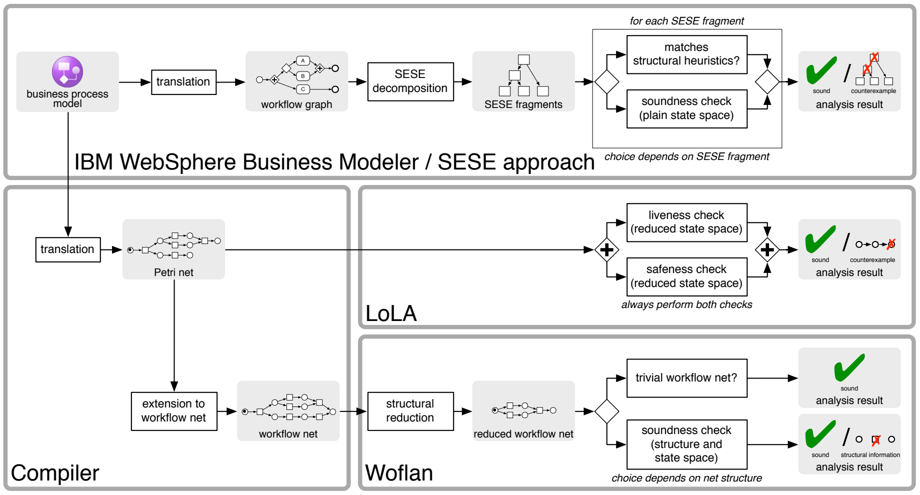


Figure 1: Three different approaches and tools to check soundness.

a translation of these models into Petri nets, whereas for the third approach, the models were translated into workflow graphs.

We obtained the following results: Based on the 735 process models that we analyzed, soundness of industrial business process models can be decided in a few milliseconds per process. Although many processes are simple enough that state space exploration alone would be sufficient to decide soundness, this method is not sufficient in general. However, all three approaches perform similarly fast, meeting the above-mentioned performance requirements. This implies that one can focus on different requirements such as the quality of the returned diagnostic information when deciding for a soundness-checking technique. Our study also shows that there is a high percentage of unsound models, confirming the need for better tool support for execution-aware modeling.

Previous studies [7, 8, 3] on checking soundness or the similar notion of *EPC soundness* of realistic business process models concentrate on error findings and error prediction. These studies do not report runtimes for the analysis. Mendling [9] reports an average analysis time of 1.8 secs and maximal time of 142 secs for checking the EPC soundness of 604 processes. His technique of using structural reduction rules that operate directly on the process model does not find all violations of soundness. A postprocessing with state space exploration is not included in these runtimes. The same set of processes was also checked for relaxed soundness [10] with a reported runtime of 46 secs per process on average [9, 1]; however, no maximal times are reported. Recent work [11] extends control-flow analysis to more advanced synchronization constructs such as OR-joins and cancellation regions, but so far no empirical results have been reported. A preliminary and incomplete version of the SESE decomposition technique that used heuristics only, but did not include state space exploration, was partially evaluated on a different set of data [6].

This paper extends a previously published conference contribution [12]. It is organized as follows: In Sect. 2, we discuss the data used in this study, their translation into workflow graphs and Petri nets, and the notion of soundness. Sections 3, 4, and 5 present the three approaches together with the results they achieved on the data. In Sect. 6, we report first insights from applying our techniques in industrial settings. In Sect. 7, we review the results in a comparison of the three approaches and draw conclusions.

2. Selecting the Empirical Data and Preparing the Case Study

2.1. Sampling the process data

We scanned a large set of real-world data available to the IBM team for our practical validation of the soundness-checking approaches and tools. These data mostly resulted from modeling activities in customer projects within an SOA context where processes were captured with the final goal of implementing them in a Service-Oriented Architecture. The models covered various industry domains such as financial services, automotive, telecommunications, construction, supply chain, health care, and customer relationship management. We also looked at large collections of reference processes that were created using different modeling styles, i. e., different ways of capturing data and control-flow at varying level of granularity. All models were available in the IBM WebSphere Business Modeler tool represented in a language that currently combines elements from UML Activity Diagrams and the Business Process Modeling Notation (BPMN), but some of them had originally been created in other tools first and then imported into the IBM product.

It turned out that only some of the model collections were useful for our purposes. Many process models are in fact quite small, as good modeling practice suggests an appropriate structuring of processes into subprocesses, and are therefore not a challenge for our soundness-checking approaches. Others, in particular those created in other tools, might not have been created with the appropriate notion of soundness in mind or might have been created by novice users and consequently turned out to be syntactically incomplete and therefore flawed in such a way that it made no sense to consider them further. In the course of our experimental studies, we therefore reduced our initial test set of approximately 3000 models to 5 libraries of 735 different models in total. We completely anonymized the data in these models; for instance, task names would be replaced by enumerations $t1, t2, \ldots$, and named these libraries A, B1, B2, B3, and C. These anonymized libraries, which have been stripped off all semantics and represent only purely structural information, were the input for the tools LoLA, Woflan, and the SESE approach. Libraries B1, B2, and B3 partially overlap—hence their similar names—as they represent a series of models from the same domain created over a period of two years, in which a library changed to the next by adding more process models and refining all models. The number of 735 different processes therefore counts only the latest library in this series, which is B3 with 421 processes, together with the 282 processes from library A and 32 processes from library C.

Table 1: Static data of the process libraries in the case study.

	A	B1	B2	B3	C
Avg. / max. number of nodes	14 / 46	17 / 69	16 / 67	18 / 83	27 / 118
Avg. / max. number of edges	33 / 127	29 / 147	31 / 202	33 / 195	33 / 145
Avg. / max. node inflow	2.52 / 13	1.76 / 15	1.90 / 69	1.86 / 27	1.84 / 4
Avg. / max. node outflow	1.03 / 8	0.94 / 13	0.99 / 15	1.05 / 30	1.83 / 4

Table 1 characterizes the data from our process libraries in terms of the number of nodes that represent tasks, subprocesses, gateways, start and end events, and the number of edges that represent control- and data-flow connections between nodes. The inflow and outflow numbers capture the branching degree that occurs in the models. For libraries B1 and B2 the average outflow is smaller than 1, because many end events occurring in these models have outflow 0.

We show an average-sized example from library C in Fig. 2 to illustrate a typical process model in our collection. We split the flow into two parts: the end of the left flow continues at the beginning of the right flow. This process model contains 21 tasks representing elementary, not further distinguished process steps, 16 gateways to encode XOR-splits and -joins, and 51 edges representing data- and control-flow connections. A task can have multiple incoming and outgoing edges that encode implicit AND-splits and -joins of the control and data flows. The example model also contains several cycles: There is a large cycle that spans almost the entire process and there are three smaller cycles within this large cycle — two of them are nested within each other, whereas the third occurs at the end of the process.

2.2. Translation into workflow graphs and Petri nets

Data-flow constructs in the language of the current version of IBM WebSphere Business Modeler are similar to UML activity diagrams. Here, we only consider explicit data-flow connections and no repositories, because each such connection implies a control-flow connection. Control-flow constructs are visualized in BPMN.

The translation of the process models into the format required by the soundness checkers focuses on the following modeling elements: start and end events, tasks, subprocesses, control-flow, input and output sets of tasks (explained subsequently), and gateways. Data-flow is ignored during the translation, which means that each explicit data-flow connection is replaced by a control-flow connection. Data-flow connections from and to repositories were not considered at all. The current language supported by IBM WebSphere Business Modeler contains XOR- and AND-gateways as well as an OR-split, but no OR-join. The translation is well-known and therefore not repeated here; details are provided elsewhere [13].

A task can have multiple incoming and outgoing edges (inputs and outputs) that can be grouped into sets. Input and output sets of tasks are translated into gateway logic as illustrated in Fig. 3. In this figure, task A has inputs a, b

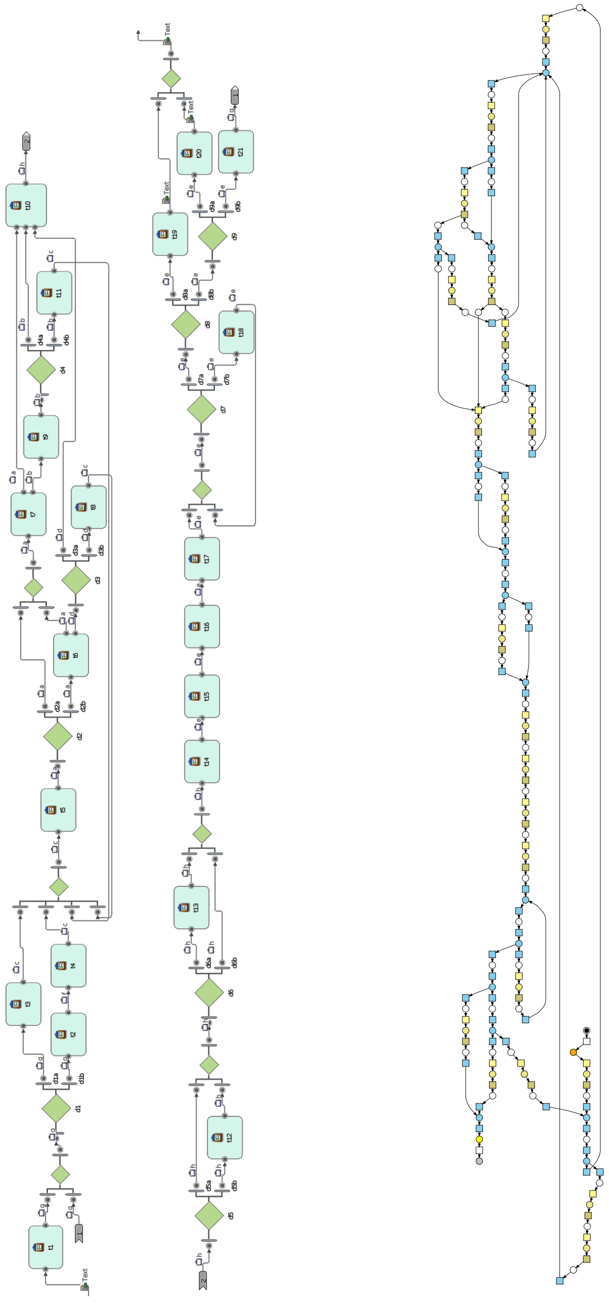


Figure 2: Example of an average-sized process model (left) and its translation into a Petri net (right).

grouped into one set and inputs c, d, e grouped into another set with the meaning that A can execute if it either receives a and b as input or c, d, and e. The alternative output sets of task A are f, g, h and i, j, k. Furthermore, a modeler can specify which input set activates which output set, but this information was not provided in any of the models in our data. For the translation, we therefore assumed that each input set can potentially activate each output set.

The presence of an input or output is expressed by placing a *token* on an edge between two nodes. Tokens move through the process as a task or gateway executes, taking the process from one state to another state in the usual way. In the middle of Fig. 3, we see the translation of task A into a *workflow graph* [2, 6], which is a control-flow graph containing only gateways and tasks. To the right, we see the resulting Petri net. Figure 2 illustrates the translation for an entire process: the process model to the left translates into the Petri net to the right.

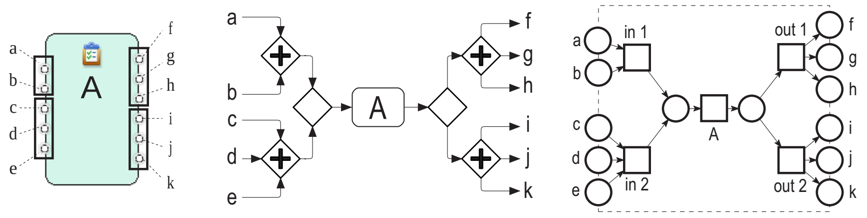


Figure 3: Translation of a task with disjoint input and output sets (left) into the corresponding workflow graph (center) and Petri net patterns (right).

In general, input and output sets can overlap which would lead to *non-free-choice* Petri nets [14] as a result of the translation [13]. However, none of the syntactically valid process models from our test set used overlapping inputs or output sets. Therefore, the translation will only return free-choice nets in our case study. This makes it possible to benefit from fast analysis techniques for free-choice Petri nets, which are applied by Woflan as explained in Sect. 4, and for workflow graphs as explained in Sect. 5.

Process models with multiple sinks. Each process model has one or more start events and one or more end events. Process execution begins by placing one token on each edge leaving a start event. The process terminates when only edges that enter an end event carry a token. The translation into Petri nets generates an initially marked source place for each start event and a sink place for each end event.

Some Petri net-based soundness analysis techniques such as Woflan (presented in Sect. 4) pose syntactic restrictions on the Petri nets that can be analyzed: a *workflow net* is a Petri net with a unique start place, a unique sink place, and each transition of the net lies on a path from source to sink place [15].

Only a few process models from our libraries have a unique end event, hence only a few of the resulting Petri nets would have a single sink place and thus would be workflow nets. However, a net N with multiple sinks can be *extended* to a workflow net N' using the algorithm of Kiepuszewski et al. [16, Proof of

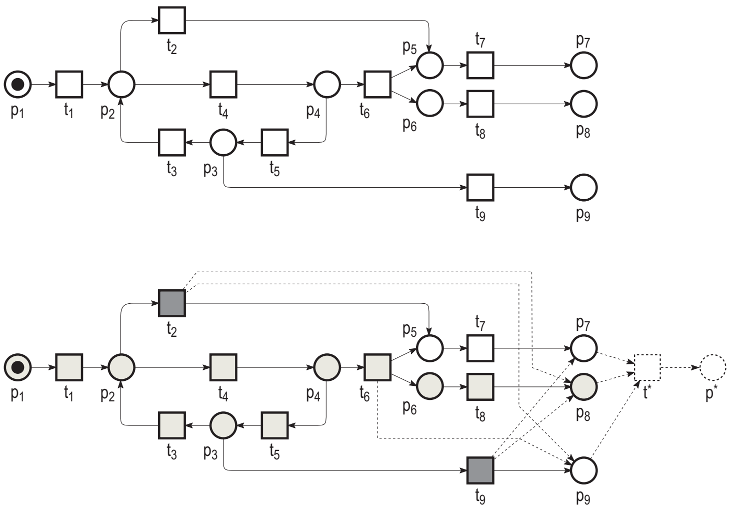


Figure 4: A net with multiple sinks (top) and its extension to a workflow net with a unique sink (bottom) by the algorithm of Fig. 5.

Theorem 5.1]. This algorithm adds new edges to N, which cause every sink place of N to be marked in every execution. All sink places of N are then synchronized by a final transition, which produces a token on a fresh unique sink place. Figure 4 depicts an example.

The algorithmic definition of the extension was given implicitly originally. We provide an explicit pseudocode algorithm in Figure 5. Figure 4 illustrates the algorithm for the sink place p_8. The set $\mathit{backwards}(p_8)$ comprises the places p_1 to p_4, p_6 and p_8 and the transitions t_1, t_3 to t_6 and t_8 and is highlighted in light gray. Transitions t_2 and t_9 (highlighted dark gray) consume from a place in $\mathit{backwards}(p_8)$ but p_8 cannot be reached from t_2 or t_9. In other words, an occurrence of t_2 or t_9 interrupts a token on its way to p_8. The new edges (t_2, p_8) and (t_9, p_8) ensure that the token still reaches p_8. The same is done correspondingly for all other sink places of the net. Thus, t^* is eventually enabled which leads to a single token on the unique sink place p^*.

Kiepuszewski et al. [16] show that soundness is preserved by the extension assuming that the original net N is a free-choice Petri net. As we discussed in Sect. 2, all processes in our data have the free-choice property. It is also not difficult to prove that the extension also preserves unsoundness. Extending N only requires a depth-first search in N for each of its sink places.

Tool support. Two different translations into workflow graphs and Petri nets were implemented independently, although the free-choice Petri nets could also be directly obtained from the workflow graphs by a well-known construction [2].

The tool that translates the original process models to Petri nets also implements the algorithm for extending a Petri net with multiple sink places to a Petri net with a unique sink place [16] as well as structural Petri net reduction

```
input:  A free-choice Petri net $N = (P, T, F)$ with places $P$, transitions $T$,
        and flow edges $F$.
output: A Petri net with a unique sink place.
begin
  for each sink place $p$ of $N$ do
    let $backwards(p)$ be the set of all nodes having a path to $p$
        along the edges $F$ of $N$
    for each place $q \in backwards(p)$ do
      for each transition $t$ having an edge $(q, t) \in F$ do
        if $t \notin backwards(p)$ then
          add edge $(t, p)$ to $F$
  add a fresh transition $t^*$ to $N$
  add edge $(p, t^*)$ to $N$, for each sink place $p$ of $N$
  add a fresh place $p^*$ to $N$
  add edge $(t^*, p^*)$ to $N$
end
```

Figure 5: Algorithm for creating a unique sink place for a free-choice Petri net.

rules [17], which have been applied in the course of our case study. In addition, the translation to Petri nets can be configured regarding properties of the input (e. g., consider only processes with disjoint input and output sets), the applied translation (e. g., regarding the termination semantics of a process and the intended correctness criterion), and the output into various Petri net file formats.

The translation tool to Petri nets as well as the process models used in this case study are available at `http://www.service-technology.org/soundness` in their original format of IBM WebSphere Business Modeler and as Petri nets in different formats including PNML [18].

2.3. Soundness

Soundness was initially defined for workflow nets [15]. Its main condition is that a *safe termination state* always remains reachable during the execution of the workflow. A state is said to be a *termination* state of a Petri net if each token is on a sink place. A termination state is *safe* if no sink place contains more than one token. Furthermore, a transition of a Petri net is said to be *dead* if there is no reachable state that enables it. A Petri net is *sound* if (1) for each reachable state s, there exists a safe termination state that is reachable from s and (2) no transition is dead.

The same definition can be applied to workflow graphs. However in workflow graphs and, equivalently, free-choice Petri nets, the situation simplifies and soundness can be characterized in terms of two types of local errors, viz. *local deadlock* and *lack of synchronization.* Figure 6 shows a workflow graph without any tasks, which is taken from the middle part of the process in Fig. 2 and to which we added a start and an end event. This process model contains a *lack of*

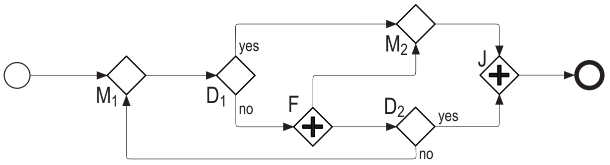


Figure 6: A workflow graph with deadlock and lack of synchronization errors.

synchronization error as well as a *local deadlock*, which are not so easy to spot in the first place.

A *local deadlock* is a reachable state s of the process that has a token on an incoming edge e of an AND-join such that each state that is in turn reachable from s also contains a token on e, i. e., the token is 'stuck' on e. A local deadlock arises for example, if two alternative paths are merged by an AND-join or if an AND-join occurs as an entry to a cycle. In the example in Fig. 6, a deadlock occurs when a token travels the *yes* edge leaving the XOR-split D_1. Eventually, this token will reach the AND-join J via the upper incoming edge. However, no other token will ever arrive at the lower incoming edge of J.

A reachable state s contains a *lack of synchronization* if there is an edge that has more than one token in s. If such an edge contained a task, this task would be executed twice. A lack of synchronization arises for example, if two parallel paths are merged by an XOR-join or if the exit of a cycle is an AND-split. In the example in Fig. 6, a lack of synchronization occurs when a token travels the *no* edge leaving the XOR-split D_1. This token will activate the AND-split F, which leads to a token reaching the XOR-join M_2 and another token traveling the cycle D_2, M_1, D_1, F. This can result in multiple tokens on the edge from F to M_2. A lack of synchronization in a Petri net is usually called an *unsafe* state.

For our different approaches in the paper to check soundness, we will exploit several equivalent characterizations of soundness. If G is a workflow graph or a free-choice Petri net, then the following statements are equivalent:

- G is sound.
- G has neither a local deadlock nor a lack of synchronization.
- For each reachable state s of G, there is a safe termination state that is reachable from s.
- For each reachable state s of G, there is a termination state that is reachable from s and G has no lack of synchronization.
- G has no lack of synchronization and no *global deadlock*, where a *global deadlock* is a reachable state s that is no termination state such that there is no state $s' \neq s$ that is reachable from s.

Further formalization of these statements can be found elsewhere [2, 3, 16, 6].

Soundness in the process data. Table 2 summarizes the results of our analysis for the Petri nets describing the control-flow of the processes in our libraries. On average, only 46% of all process models are sound ranging from 37% for library B1 to 53% for library A. Row 4 shows the number of processes with more than one million reachable states, which include error states, and processes that have infinitely many reachable states such as the process shown in Fig. 6. To exclude those, we measured the size of the state space of each *sound* process, which is always finite, which still returned a few processes with more than one million states. The average values, however, suggest that such processes are rare.

Table 2: Dynamic data of the process libraries in the case study.

	A	B1	B2	B3	C
Processes in library	282	288	363	421	32
sound	152	107	161	207	15
unsound	130	181	202	214	17
Processes with >1000000 states	26	19	29	38	7
Processes with >1000000 states (only sound)	0	1	4	4	0
Avg. number of states (only sound, <1000000 states)	26	71	322	4911	680
Max. number of states (only sound, <1000000 states)	213	2363	28641	588507	8370

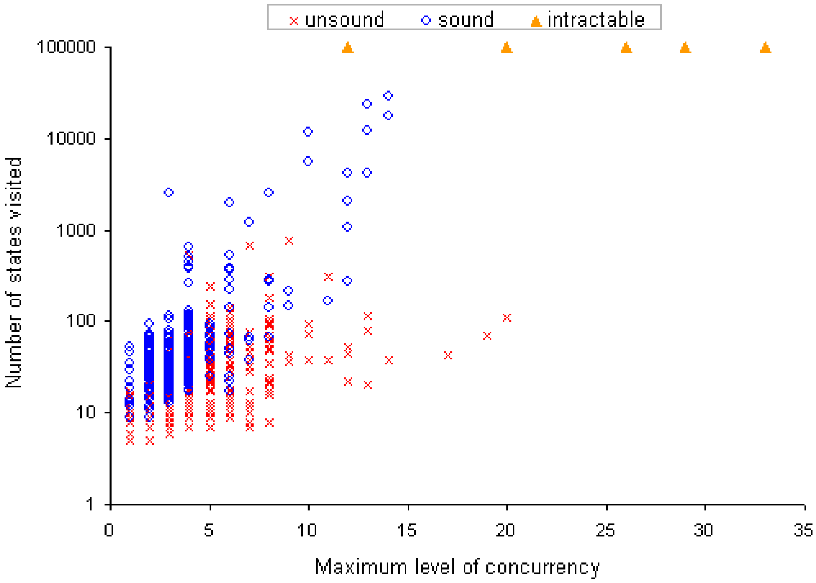


Figure 7: Number of states visited until first error.

Next, we measured the hardness of instantaneous verification in our data. For each process, we explored the states of the corresponding workflow graph until discovering a control-flow error or, when no error exists, until all states have been visited. If we visited more than 100000 states, we stopped the exploration and classified the process as intractable. Reaching this threshold correlates with a runtime of at least a few seconds which is not acceptable in the use case of instantenous verification as described in Sect. 1. Thus, the number of visited states roughly corresponds to the ‘effort’ needed to check soundness using plain state space exploration. Figure 7 depicts the number of visited states for the processes in libraries A, B3, and C with respect to the *maximum level of concurrency.* The *maximum level of concurrency* is given by the maximum number of tokens that occur in a single reachable non-error state of the workflow graph. This number indicates to which extent plain state space exploration faces the state space explosion problem in the respective process. We used the last characterization of soundness from the enumeration above for this measurement.

Altogether, we observed that most of the processes have a relatively small state space: less than 9% of the processes have more than 100 states, less than 3% have more than 1000 states. Only a small portion of the state space needs to be visited in order to find the error for unsound processes. The number of visited states for sound models correlates with the level of concurrency. Using plain state space exploration, we could not analyze sound processes with a level of concurrency above 15: a sound process having 15 concurrent task sequences of length 2 already has a state space of at least $3^{15} = 14,348,907$ states. In the following sections, we will discuss techniques which allow us to analyze processes beyond this bound.

3. State Space Verification with LoLA

LoLA [5] is a tool that decides numerous properties of a given Petri net by an inspection of the state space. To make state space inspection feasible, it offers several state space reduction techniques. The subsequently described experiments were carried out with version 1.11 of LoLA [19].

Soundness as a model-checking problem. Process models are translated into Petri nets prior to the verification as sketched in Sect. 2.2. To verify soundness, LoLA works in two runs on the resulting Petri nets. In the first run, it checks for local deadlocks. In the second run, it checks for lack of synchronization. This order is motivated by the observation that it is cheaper in terms of run time and memory consumption to qualify a model as unsound due to a local deadlock rather than due to a lack of synchronization. Subsequently, we explain the reasons for this behavior in more detail.

A process has no local deadlock if and only if a termination state is reachable from every reachable state. The latter can be easily expressed as a state predicate in LoLA, which we call *termination.* To check this predicate, LoLA has several options.

The first option is to express this predicate as a formula *AG EF termination* in the temporal logic CTL [20] and to use LoLA’s CTL model checker. Informally, *AG* represents “in every reachable state. . . ” whereas *EF* means “. . . there exists an execution where eventually. . . ”. In LoLA, a CTL formula is verified by a nested depth-first search for counterexample or witness paths. For the formula above, LoLA searches for a counterexample path that leads from the initial state to a state from which no termination state is reachable. The process has no local deadlock if LoLA cannot find such a path. LoLA is able to verify the property on a reduced state space, which means that it computes only a subset of the reachable states in such a way that the reduced system satisfies the given formula if and only if the original system does. The applied technique is known as *partial order reduction*.

Partial order reduction occurs in several different, independently evolved flavors such as the *stubborn set method* [21], the *persistence set method* [22], or the *ample set method* [23]. LoLA uses variations of the stubborn set method which performs best on Petri net models. Partial order reduction has originally been developed for the temporal logic LTL. Unfortunately, AG EF type formulae cannot be expressed in LTL and therefore, CTL is used in this case. A partial order reduction for CTL is significantly less powerful than the one known for LTL [24]. If a CTL formula uses no other operators than AG and EF, LoLA additionally exploits the improvements reported in [25, 26]. In any case, LoLA verifies properties *on-the-fly* while the state space is being generated.

The second option to check termination is a special algorithm for AG EF type temporal logic properties. This algorithm is based on the observation that a property AG EF φ can be rewritten as "In every terminal strongly connected component of the state space, there exists a state satisfying φ". Consequently, LoLA first computes a subset of the state space that, by construction, contains elements of every terminal strongly connected component, but not necessarily all elements of the components. Then, it picks one state from each such component and verifies reachability of a φ-state. The advantage of this procedure is that variations of partial order reductions [27, 28] can be used that yield significantly better reduction than the generic CTL preserving methods mentioned above. As soon as LoLA detects a violation, it stops and returns the violating state. Once an error state has been found, a reachability check is used to produce a trace to this error state.

Lack of synchronization, i. e., unsafeness of states, can be expressed in LoLA as the state predicate $\bigvee_{p \in P} m(p) > 1$, where P is the set of places of the Petri net. As this set can become very large, e. g., on our test data, a maximum of 275 places occurred, we simplified this predicate to optimize performance. We can assert by construction for several places in the Petri net that they cannot obtain more than one token unless a preceding place is also able to do so. In essence, only places that represent an XOR-join or an exit of a cycle need to be considered. The resulting state predicate is checked for reachability by LoLA. If the predicate is satisfied, a lack of synchronization is identified and LoLA produces a trace to the error state.

Although LoLA's partial order reduction specifically addresses reachability queries, the given formula typically yields only a marginal reduction. The method described in [28] requires that certain transitions need to be fired in every state of the reduced state space. For a state predicate $m(p) > 1$ this is just the set of those transitions that put tokens onto p, but for formulae of the

Table 3: Impact of order of soundness checks using LoLA.

	A	B1	B2	B3	C
Processes	282	288	363	421	33
Complete number of checks	564	576	726	842	66
Actual required checks (first check for lack of synchronization)	529	476	606	711	58
Actual required checks (first check for deadlock)	434	395	524	625	49

shape $\varphi \vee \psi$ it is the union of sets that are sufficient for φ and ψ, respectively. For deeper insights, we refer the reader to [28]. As the unsafeness predicate involves many places of the Petri net, there is not much room for partial order reduction. This explains why we first check for local deadlocks and only then for lack of synchronization. This order, however, has the twist that some of the nets that exhibit lack of synchronization actually have infinitely many reachable states. In this case, our local deadlock algorithm may not terminate. Fortunately, LoLA has a switch that causes any state space generation to be stopped if an *unsafe* state is generated during the first run to find local deadlocks. A state is unsafe if a single place contains more than one token, which indicates a lack of synchronization in the original process model. This simultaneous check for lack of synchronization in the first run prevents that LoLA tries to generate an infinite state space and also optimizes performance for finite state spaces. If an unsafe state is found, a trace to this state is returned immediately.

The partial order reduction for the first run only preserves deadlocks, but not lack of synchronization. Therefore, some lack of synchronization errors might remain undetected. Thus, if no error has been detected during the first run, LoLA is invoked a second time on each net, this time explicitly checking for lack of synchronization.

Table 3 summarizes the impact of the order of the checks on the required number of checks for the process libraries. For each process, two checks are required in principle. When we check for lack of synchronization first, we already detect some unsound processes and could skip the subsequent check for deadlocks. When we change the order of the checks, we actually detect all unsound processes during the first check. The subsequent check only made sure that the sound processes indeed did not contain any lack of synchronization.

For the example depicted in Fig. 2, LoLA detects a lack of synchronization in the first run, concludes that the net is unsound, and returns an error trace consisting of 36 states.

Experimental setup. After translating the process models into Petri nets with our compiler [13, 29], we performed the two checks explained above. We ran the experiments on a notebook with a 2.16 GHz processor and 2 GB RAM. We set a bound of one million states for each net and classified a net as *intractable* if this bound was reached.

Experimental results. The Petri nets that we obtained have about 5.5 times more nodes and edges when compared to the original models, see Table 1. This

Table 4: Analysis statistics for LoLA.

	A	**B1**	**B2**	**B3**	**C**
Intractable processes (no partial order reduction)	0	2	5	4	0
using partial order reduction					
Avg. number of explored states	50.42	40.60	37.52	60.76	127.28
Max. number of explored states	187	1591	1591	6467	1469
Avg. length of error trace	30.24	10.81	12.12	11.21	53.17
Max. length of error trace	67	110	75	103	120
Analysis time for library [ms]	2680	2356	3184	3878	305
Analysis time for library (struct. reduced) [ms]	2523	2192	3025	3575	275

is caused by the more fine-grained representation of the process logic in Petri nets as illustrated by Fig. 3 where 1 task node translates to 18 Petri net nodes. The largest net with 558 nodes and 607 edges results from a process model in library C.

Without partial order reduction, not all nets could be analyzed, see row 1 of Table 4. When partial order reduction is used, no intractable process is left. In fact, the largest explored state space consists of only 6467 states. On average, only around 100 states need to be explored. The overhead of applying partial order reduction is more than compensated by its effect to only generate a small fraction of the states. For those processes that could be analyzed without partial order reduction, analysis times are up to 10 times longer. During the experiments, LoLA never consumed more than 2 MB of memory, which enables an unobtrusive verification process, which was an open question before running the experiments. Table 4 summarizes the results.

In a variant of the experiments, we also applied structural Petri net reduction rules [17] to each Petri net before checking it with LoLA. These rules reduce the size of the net, while preserving soundness. The last row of Table 4 shows that structural net reduction has almost no effect on the runtime. Note that the numbers do not contain the time spent on structural reduction.

The longest error trace contains 120 Petri net states. When mapped to the original process model, this trace corresponds to a sequence of 40 tasks.

4. Soundness Verification with Woflan

Woflan [3] is a tool for verifying the soundness of business processes modeled as Petri nets. It poses syntactic restrictions on the Petri nets it can analyze, requiring that the net is a *workflow net* [15]. Section 2.2 presented an algorithm for extending any process model from our libraries to a workflow net while preserving soundness. In our experiments, we checked soundness with Woflan on these extended models.

The tool Woflan. Woflan implements a complex algorithm [3] to check soundness. It uses various techniques from Petri net structure theory as well as state space exploration. If the workflow net is a free-choice net, which is the case in our experiments, Woflan's algorithm reduces to the following procedure (recall also Fig. 1):

(1) First, soundness-preserving structural reduction rules from Petri net theory [17] reduce the size of the input. If the resulting net is *trivial*, i. e., it has only one transition, Woflan immediately concludes that it is sound. (2) Otherwise, Woflan checks the *S-coverability* of the net [3] to exploit the following properties: (2a) A free-choice Petri net that is not S-coverable is unsound, and Woflan quits; the unsoundness can be caused by a deadlock or a lack of synchronization. (2b) A Petri net that is S-coverable has no lack of synchronization, but may contain a local deadlock [3]. (3) If step (2b) applies, Woflan searches for local deadlocks — in Petri net terms a *dead* or a *non-live* transition — by state space exploration, which amounts to constructing the net's coverability graph. The techniques underlying steps (2) and (3) have exponential worst-case complexity in the size of the net.

Woflan provides two kinds of diagnostic information in this setting: If step (2a) applies, it returns a list of places that are not S-coverable which contribute to a deadlock or a lack of synchronization. If Woflan detects a deadlock in step (3), it returns a list of dead and non-live transitions that create this deadlock.

Experimental setup. We verified the workflow nets resulting from the translation with a command-line version of Woflan in a batch on a notebook with a 1.66 GHz processor and 2 GB RAM. We ran the experiments twice, the first time without applying structural reduction, the second time with. Aiming at *instantaneous* verification, we interrupted Woflan if the verification time exceeded 5000 ms. In these cases, we classified the process as *intractable* for the analysis.

Experimental results. Table 5 summarizes the results of our Woflan experiments. Our first analysis on the unreduced workflow nets was intractable for 46% of library A and for 19%–28% of libraries B1 to B3. The size of these nets corresponds to the numbers presented for LoLA in Sect. 3. Surprisingly, the analysis became intractable mostly when Woflan checked S-coverability — the technique's exponential worst-case complexity explains this observation. If S-coverability completed successfully, proving absence of deadlocks by state space exploration was tractable in all, but 11 cases. Library C was analyzed completely and fairly quickly, see Table 5, row 4. The structure of its models seems to be more suitable for Woflan. We observed that without capping analysis after 5000 ms, Woflan's analysis frequently required between 15 minutes and more than 1 hour per process.

In the second experiment, we allowed Woflan to apply structural Petri net reduction rules before the analysis, which on average reduced nets in size by a factor 5. The largest net, which resulted from a process in library B3, has 74 nodes and 232 edges. About a third of all models were reduced to the trivial workflow net, see Table 5, row 5. Thus, structural reduction alone identified 53% (libraries A and C) to 80% (libraries B) of all sound processes. Woflan classified about two thirds of the remaining nets as unsound by proving that a net is not S-coverable and free-choice. These nets constitute almost 100% of all

Table 5: Analysis statistics for Woflan.

	A	**B1**	**B2**	**B3**	**C**
1) Without structural reduction					
Intractable processes	129	54	77	119	0
due to S-coverability	129	53	74	112	0
due to state space exploration	0	1	3	7	0
Analysis time [ms]	860812	288218	429343	755875	2375
2) With structural reduction - no intractable processes					
Sound by structural reduction	81	79	134	162	8
Unsound by S-coverability	130	176	197	210	11
explored state spaces	71	32	32	49	8
Max. number of explored states	8	7	8	8	12
Analysis time per library [ms]	1120	1305	1795	2315	165
per process [ms], avg. / max.	3.97 / 20	4.55 / 40	4.94 / 91	5.50 / 1142	6.11 / 90

unsound models. For example as Table 2 shows, library B3 has 213 unsound processes, out of which 210 are not S-coverable. Only for the remaining nets — between 9% (library B2) and 25% (libraries A and C) of the processes — a state space of at most 12 states was explored to complete the analysis. Woflan checks soundness of a process in about 4 to 6 ms on average, with a maximum runtime of less than 91 ms. The one exception in library B3 ran into the exponential worst-case complexity of the S-coverability check, see Table 5, row 10.

Diagnostic information. Interpreting Woflan's diagnostic information on the original process model is not trivial. Woflan succeeds in an efficient analysis by structurally reducing the original process model. If the S-coverability check determines that the reduced net is unsound, then the readily available diagnostic information is the set of places that are not S-coverable. This set of places can be mapped back to the original process model by identifying for each place of the reduced model the corresponding places of the original model. Unfortunately, expanding the diagnostic information to the original model may hide the concrete source or location of the error. For instance, in the workflow net that corresponds to the model of Fig. 6, Woflan reports *all* places to be not S-coverable.

Woflan may compute *additional* diagnostic information upon user request. A counterexample trace can be easily obtained when Woflan checks for dead or non-live transitions by state space exploration in step (3) of the algorithm. If Woflan determines unsoundness by a failed S-coverability check in step (2a), then a counterexample trace has to be constructed explicitly in a separate run of the tool [3].

To estimate the additional effort for computing diagnostic information, especially counterexample traces in case of non S-coverable processes, we ran the experiments again and enforced the construction of a counterexample trace by Woflan. We always applied structural reduction rules prior to the analysis. The obtained picture is quite diverse, see Table 6.

For libraries A and C, generating additional diagnostic information succeeded with practically no additional effort. For library B1, the additional effort

Table 6: Generating diagnostic information by Woflan.

	A	**B1**	**B2**	**B3**	**C**
1) Standard analysis - structural reduction applied, per process					
analysis time [ms] avg. / max.	3.4 / 16	3.6 / 31	3.9 / 31	4.3 / 110	6.8 / 47
explored states avg. / max.	1.2 / 8	0.7 / 7	0.7 / 8	0.8 / 8	2 / 12
2) Analysis and additional diagnostic information - structural reduction applied, per process					
intractable	0	0	2	5	0
tractable					
analysis time [ms] avg. / max.	3.4 / 16	5.1 / 250	5.5 / 250	20 / 4672	6.8 / 62
explored states avg. / max.	5 / 26	34 / 1927	38 / 1927	96 / 16404	16 / 128

was moderate, whereas for libraries B2 and B3 the generation required significant additional effort for some processes compared to the standard analysis. In more detail, generating a counterexample trace turned out to be practically infeasible for 2 processes of B2 and 5 processes of B3, i. e., requiring substantially more than 5000 ms to complete. Among all tractable processes, libraries B1 and B2 exhibit a similar complexity with respect to diagnostic information. Analysis time increases slightly to at most 250 ms per process which is still very fast. In these cases, about 34–38 states were explored on average and 1927 states in the worst case compared to 8 states in the standard analysis. For library B3, analysis time grows close to 5000 ms in the worst case, but is still at moderate 20 ms per process in the average case. Here, about 96 states were explored on average with a maximum of 16404 states compared to 8 states in the standard analysis. The shortest counterexample traces in the reduced models ranged from 2 to 8 steps. Woflan computes by default all counterexample traces. The largest trace in a structurally reduced net consisted of 30 steps (library B3).

We conclude that Woflan's soundness analysis largely benefits from sophisticated structural analysis techniques. In the standard analysis, S-coverability checking alone does not sufficiently speed up the analysis for instantaneous verification of free-choice Petri nets. However, this technique becomes very powerful in combination with Petri net reduction rules. For up to 91% of our examples, soundness or unsoundness was proven alone by these two techniques. Only in the remaining cases, a fairly simple state space exploration was required.

In a concrete application, a user may choose to quickly check for soundness of a process model. If the model is unsound, Woflan may generate detailed diagnostic information upon request. Woflan efficiently generates further structural information such as mismatches between AND-splits and XOR-joins. In those cases, where meaningful diagnostic information comes only by behavioral analysis like counterexample traces, Woflan has to construct a state space of the model [3]. Because Woflan does not implement partial order reduction, it then faces the state space explosion problem which it avoids in standard analysis. As a result, running time for providing diagnostic information can increase substantially.

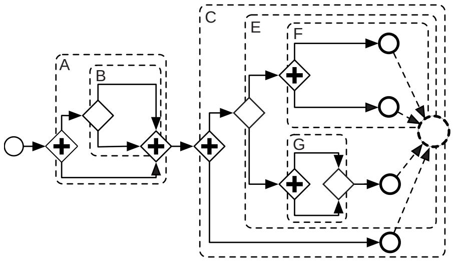


Figure 8: Decomposition of a workflow graph using the Refined Process Structure Tree.

5. The SESE Decomposition Approach

The SESE approach structurally decomposes a business process model into smaller fragments, for which soundness is analyzed by heuristics and state space exploration. If each fragment is sound, then the entire process is sound. The analysis is done on a workflow graph, which is obtained from the original process model as sketched in Sect. 2.2. The SESE approach combines the following three techniques.

State space exploration. The base technique for the SESE approach is state space exploration. Soundness of a workflow graph can be decided by checking that no explored state has more than one token on a single edge (lack of synchronization) and that each non-terminal state has a successor state (*global deadlock*). If a workflow graph has no lack of synchronization, then every local deadlock manifests itself eventually in a global deadlock in each execution. The workflow graph's state space is explored by depth-first search. The analysis terminates upon the first state that violates one of these two properties and returns a trace leading to this state. If there is no error, the entire state space must be explored.

SESE decomposition. To mitigate the state space explosion problem, we use a parsing technique called the *Refined Process Structure Tree (RPST)* [30]. The RPST decomposes a workflow graph into a hierarchy of *fragments* with a single entry and single exit (SESE) of control. A SESE fragment of a workflow graph is a subgraph that has a single entry node and a single exit node. Figure 8 shows an example of a workflow graph that is decomposed into such fragments. Multiple end nodes can be handled by adding a unique dummy end node as shown in Fig. 8. Soundness is compositional with respect to SESE fragments and therefore, each fragment can be checked in isolation [6]. To verify the soundness of a fragment, each of its child fragments can be treated as a task (node) of the workflow graph.

The soundness of a SESE fragment can be checked using plain state space exploration. Because fragments are usually considerably smaller than the entire workflow graph, the input to the state space exploration is smaller, in turn resulting in smaller state spaces to be explored. The decomposition is done in linear time and the number of fragments is at most linear in the size of the workflow graph. The time to analyze an entire workflow graph is then dominated by the size of its largest fragment.

The diagnostic information returned is a fragment showing the error as a trace relative to the fragment. This shows an error inside a smaller scope and shortens the error trace. Moreover, the checker can detect multiple errors at once, up to one per fragment. This includes ‘unreachable’ errors, such as a lack of synchronization in a fragment. For example, fragment G in Fig. 8 contains such a lack of synchronization. However, it cannot be reached by plain state space exploration, because this fragment is obstructed by another deadlock earlier in the process contained in fragment B.

Heuristics. In practice, many fragments have a simple structure that can be recognized as sound or unsound in linear time using structural heuristics [6]. For example, if a fragment contains only XOR-gateways, it is purely sequential and therefore sound. A fragment is unsound if it contains at least one XOR-split, but no XOR-join. In this case, the XOR-split inside the fragment can be highlighted as diagnostic information. We implemented 14 heuristics that can be evaluated based on a single count of the gateway types within a fragment. Only a fragment that does not match any of the heuristics becomes subject to state space exploration. Such a fragment is said to be *complex.* Consequently, heuristics can be expected to speed up the analysis by avoiding state space exploration.

Experimental setup. The SESE approach is implemented as part of IBM WebSphere Business Modeler, in which we also conducted the experiments.We conducted three experiments to measure the impact of the SESE decomposition and the heuristics: First, we used plain state space exploration only. Second, we decomposed each process into its SESE fragments, and *all* fragments were then analyzed by state space exploration. In the third experiment, we used decomposition in combination with heuristics and state space exploration, i. e., state space exploration was only applied to complex fragments.

The SESE experiments were conducted on a notebook with a 2 GHz processor and 3 GB RAM. The analysis time was computed as an average over five runs and collected from the debugging console. It also includes the time spent by the tool to generate the error report for the user. The overhead for loading the process models from the hard drive into memory was measured separately and factored out from the analysis time.

A process is *intractable* if more than 100000 states have to be explored. This threshold value is based on the experience that the time needed would otherwise exceed a value that is acceptable in the use case of instantaneous verification as described in Sect. 1.

Experimental results. Table 7 shows the results for the three experiments described above. For plain state space exploration, we observe that 6 out of 363 processes (considering only library B2) are intractable, which are less than 2%. Analyzing library A, which contains no intractable process, only requires 490 ms.

Table 7: Experimental results for the SESE decomposition approach.

		A	B1	B2	B3	C
1) State space exploration - reference						
Explored states per process (avg.)		42.8	826.4	1879.3	1508.1	149.7
Explored states per process (max.)		241	17176	28684	28688	2517
Intractable processes		0	2	6	5	0
Analysis time [ms]	library	490	30019	197670	135178	30019
	process (max.)	16	13186	76700	24624	62
2) Using the decomposition - no intractable processes						
Size reduction (workflow graph / largest fragment) (avg.)		4.1	3.8	3.9	4.7	2.7
Explored states	process (avg.)	52.4	31.6	86.7	38.4	61.7
	avg. reduction per process w.r.t. 1)	1.0	13.8	13.4	10.2	1.5
	process (max.)	201	268	16534	311	356
	fragment (max.)	53	117	16403	68	120
Analysis time [ms]	library	1587	1359	35495	2446	447
	process (max.)	16	16	25286	32	32
3) Using the heuristics - no intractable processes						
Explored states	process (avg.)	6.0	2.3	3.2	2.5	10.1
	avg. reduction per process w.r.t. 2)	28.3	22.9	78.4	29.6	34.1
	process (max.)	53	36	165	24	120
	fragment (max.)	53	36	165	20	120
Analysis time [ms]	library	1247	1390	1681	2303	318
	process (max.)	16	31	16	31	62

When using the decomposition into fragments, we observe that there is no longer an intractable process. However, the time for library B2 is dominated by the analysis of one particular process that required 25 seconds. All other processes took less than 1 second each. SESE decomposition reduces the size of the input to state space exploration by an average factor between 1.5 and 4. The number of states that are explored for a particular process is the sum of the number of states explored for each fragment of the process. Table 7 shows that the number of states that have to be explored for a process on average reduces by up to a factor of 13.8 with respect to experiment 1. After decomposition, there is still a fragment that has 16403 states.

Library A shows that decomposing process models does not always pay off. This library is analyzed faster without decomposition. The analyses of the other libraries, however, clearly benefit from the decomposition where it reduces the required time by a factor between 5 and 67 compared to plain state space exploration.

In addition, we recorded the length of the error trace in both experiments. Error traces are notably smaller when they relate only to a fragment, rather than to the entire workflow graph. The average length of an error trace was reduced by a factor of 4.7. Note that the error trace using the decomposition into fragments starts at the start node of the fragment and not at the start node of the workflow graph. The decomposition allows us to detect multiple errors per process, at most one per fragment. For library B2, we measured an average of 1.55 and a maximum of 7 unsound fragments per unsound process.

The third experiment showed that the heuristics speed up the analysis further. A process usually contains not more than one complex fragment. Note that on average, a process contains 16 fragments. Only the largest process, which has 122 fragments, contains two complex fragments, no process contained more. The small number of complex fragments results in a reduction factor of up to 78.4 for the average number of states that were explored to analyze a process. The use of the heuristics reduces the analysis time of library B2 by a factor of 21 with respect to experiment 2. For the other libraries, the differences in the analysis times was not significant. The maximum analysis times per process ranged from 10 to 62 ms.

Structuredness and soundness. In addition to speeding up analysis, the heuristics allowed us to obtain a deeper insight into how the process in our data set were structured and how this structure affects the soundness of the process.

In Table 8, we present the categories of fragments obtained by the heuristics for our libraries. For complex fragments, we also mention if they are sound or unsound, which was determined by state space exploration. A *sequence* is a fragment that does not contain any gateways. A fragment is *sequential* if it contains only XOR-gateways. It is *concurrent* if it contains only AND-gateways. A fragment that is sound and contains at most two gateways is called *well-structured.* For unsound fragments, we regroup the result of the heuristics into three categories: (1) An *inappropriate pair* is an unsound fragment that contains only two gateways of different logic. (2) A *cycle with inappropriate logic* is a fragment that contains a cycle, but no XOR-join (in which case there must be a deadlock) or no XOR-split (in which case there must be a lack of synchronization). (3) A fragment with an *unmatched gateway* has more than two gateways and contains a split of some logic (AND or XOR), but no join of the same logic or vice versa.

Table 8 shows that, in practice, more than 95% of the fragments are matched by heuristics. In general, 94% of the analyzed fragments are sound. Heuristics can match most of the sound fragments (more than 99%); 99% of the sound fragments matched by heuristics are even well-structured. Note the dominant number of concurrent well-structured fragments over the number of sequential well-structured fragments. This dominance can be explained by the routed data-flow modeling style used in libraries A and B. In this modeling style, passing a data-item from one task to another implies a separate control-flow edge between the two tasks. Thus, if a tasks sends two or more data-items to another, we obtain a concurrent well-structured fragment. Processes in library C were not designed using this modeling style.

A share of 64% of the unsound fragments are matched by heuristics. It is surprising to observe that more than 44% of these fragments contain inappropriate gateway pairs. One would assume that it is easy to choose the logic of the gateways properly when a fragment is composed of only two gateways. However, our experiments showed the opposite. One explanation for the large percentage of unsound fragments composed of paired gateways is the instantia-

Table 8: Type of fragments analyzed.

		A	B1	B2	B3	C
Fragments matched by heuristics		5151	4763	6121	8525	299
Sound:		5148	4493	5797	8217	265
Well-structured	Sequence	1414	1641	2056	2508	164
	Sequential (acyclic/cyclic)	177 / 47	218 / 6	200 / 6	314 / 9	75 / 12
	Concurrent	3471	2605	3493	5337	3
Unstructured	Sequential (acyclic/cyclic)	10 / 4	2 / 7	2 / 9	2 / 10	9 / 1
	Concurrent	25	14	31	37	1
Unsound:		3	270	324	308	34
Inappropriate pair (acyclic/cyclic)		0 / 0	120 / 4	140 / 4	134 / 4	13 / 0
Cycle with inappropriate logic		0	10	12	10	1
Unmatched gateway		3	136	168	160	20
Complex fragments (not matched)		175	127	140	170	6
Sound (acyclic/cyclic)		42 / 2	6 / 0	7 / 0	22 / 0	2 / 0
Unsound (acyclic/cyclic)		99 / 32	107 / 14	121 / 12	135 / 13	0 / 4

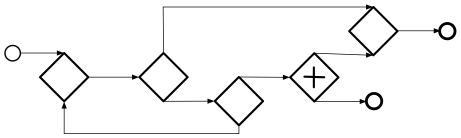


Figure 9: A complex sound acyclic workflow graph fragment.

tion semantics: unless specified otherwise, two start nodes of a process produce a token concurrently when the process is instantiated. Users often intend to model alternative start nodes, but fail to specify it properly in the editor. The translation maps concurrent start nodes of a process to a single start node followed by an AND-split in the corresponding workflow graph. We confirmed by a manual investigation of these fragments that more than half of the unsound fragments composed of paired gateways correspond to paths originating from concurrent start nodes that are joined by an XOR-join later in the flow.

It seems difficult to design a process with complex fragments as 87% of all complex fragments are unsound. 12% of the complex fragments are cyclic, but less than 3% of them are sound. Nevertheless, we found two sound cyclic complex fragments. Figure 9 depicts the structure of one of them.

As it seems easier to design a sound process by composing well-structured fragments, we assume that a model structured in this way is more likely to be sound than a less structured model. To verify this hypothesis on our data, we define the level of *structuredness* of a process as the number of *paired* gateways divided by the total number of gateways. Two gateways are *paired* if they are the only gateways contained in a fragment. Note that a fragment with paired gateways can be sound (well-structured fragment) or unsound (inappropriate pair). We measured that the average level of structuredness of sound processes is 0.93 with a standard deviation of 0.14, whereas the average level of structuredness of unsound processes is 0.77 with a standard deviation of 0.18. This difference is significant and allows us to confirm the positive influence of structuredness on the soundness in our data.

6. Delivering Analysis on Demand to Non-Expert Users: First Insights

6.1. IBM WebSphere Business Process Management Suite

The soundness checker based on the SESE Decomposition approach as described in Sect. 5 has been adopted by the IBM WebSphere Business Process Management Suite starting from Version 6.2. It was released to customers by the end of 2008. The soundness checker is used in the following two use cases:

- Validation of business process models created in WebSphere Business Modeler before the translation and code generation process where WS-BPEL code is generated.
- Validation of WS-BPEL processes upon deployment on WebSphere Process Server.

Furthermore, in 2009 a set of *accelerators* [31] was released on IBM developerWorks for users of WebSphere Business Modeler that add patterns, transformations, and refactorings to the business process modeling environment. The accelerators also contain a *Control-Flow Analysis* that exposes the soundness checker to the business user when creating a business process model. The control-flow analysis can be invoked by the user either on a single business process model or on an entire process library.

Three main challenges have to be addressed when providing a soundness checking capability to non-expert users who are not familiar with the notions of deadlocks and lack of synchronization and never before used a model checker or another soundness checking tool:

- **Coverage**: Make sure the soundness checker can check all or at least the majority of user created models. Users will not accept a new feature that has low coverage.
- **Immediacy**: Ensure the soundness checker returns a result instantly. Long runtimes caused by state space exploration are inacceptable and are often interpreted as a tool error.
- **Consumability**: Develop a user interface that conveys the information about detected control-flow errors in a way that is consumable by non-experts.

Full coverage was achieved for models created in the so called *technical modes* of WebSphere Business Modeler that constrain the modeling language. However, without any constraints, WebSphere Business Modeler allows users to create models with overlapping input and output sets. Although this feature is only used by a small group of users, it causes a conceptual gap for these users between the models that they can create and the models that can be analyzed. In the case of validation of models before export to WS-BPEL, the gap is shielded by the syntactic validation that rejects the model because overlapping inputs and outputs cannot be mapped to WS-BPEL and the models are rejected for this reason. However, the accelerators also address users that only model for documentation purposes where no syntactic restrictions are imposed on the models. Interestingly, also these users of the accelerators care about the feedback provided by the control-flow analysis and are disappointed when some of their models cannot be analyzed.

As the runtime results discussed in this paper have shown, immediacy is not an issue: Even if invoked on large collections of process models, the analysis results are returned instantly.

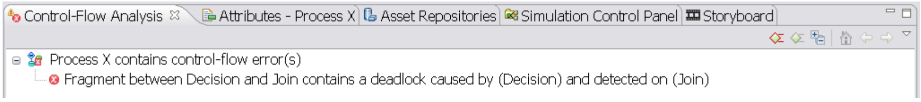
Consumability constitutes the biggest challenge and has not achieved much attention by the research community so far: How can a detected error be communicated such that the user understands the error, is able to locate it in the model, and obtains a hint on how the error could be corrected?

The SESE decomposition approach can identify several errors during one invocation, up to one error per fragment. For each error, a separate message is created. The error messages are added to a view that lists all the errors of a process. For the WS-BPEL export use cases, error messages from the soundness checker are just added to all other error messages produced by the process model validation. This can be overwhelming for some users as many errors are shown in no specific order and no hints are given on how to correct an error.

Figure 10 shows an example message of a deadlock, which follows the following schema. First, the fragment containing the error is identified by giving the name of the entry node (here, “Decision”) and the exit node (here, “Join”) that delimit the fragment. Then the type of the error is given (here, “a deadlock”) followed by a pattern of “caused by $\langle x \rangle$” and “detected on $\langle y \rangle$” where x and y are instantiated with gateways.

A deadlock is detected on the AND-join of which an incoming edge is marked in the deadlock state. It may be caused by a preceeding XOR-split as in Figure 10. Finding the location of a lack of synchronization is more difficult. It may be detected on an XOR-join that precedes the first edge carrying more than one token and caused by a preceeding AND-split. The error information is directly obtained from the heuristics and may remain incomplete, because some heuristics are not able to identify the location or cause of every error. In case a complex fragment is analyzed, which requires using state space exploration, the location of a deadlock is provided, but not its cause. For a lack of synchronization error, no information can be retrieved from a complex fragment.

An additional visualization was developed for the accelerators, which allows a user to click on an error message in the view and then visualize the error in the model. The visualization highlights the fragment that contains the error and marks the “caused by” and “detected on” nodes in the graph with error symbols as shown in Fig. 11. Users of the accelerators react very positively to



Process X contains control-flow error(s)
Fragment between Decision and Join contains a deadlock caused by (Decision) and detected on (Join)

Figure 10: Example of an error message following the “caused by - detected on” pattern.

this visualization, because they consider it as being much more informative than just the textual entry in the error view. Note that as the example illustrates, correct fragments contained within the erroneous fragment are not highlighted. A desirable extension to the visualization would be a collapsing feature that abstracts error-free fragments into a single node within the workflow graph. This would ease the navigation for the user and increase the focus on the part of the model that contains the error. Further research currently aims at a uniform intuitive visualization for all deadlocks and likewise for all occurrences of lack of synchronization.

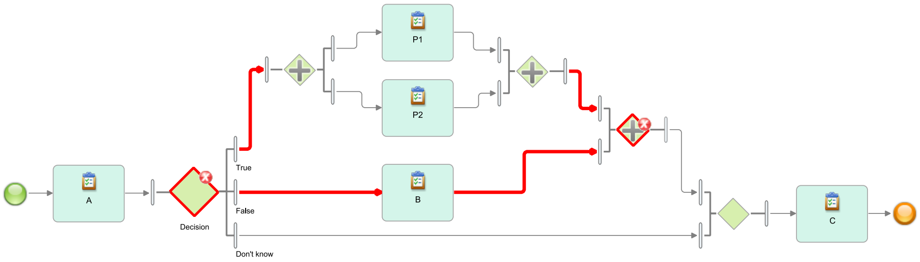


Figure 11: Coloring of the smallest fragment containing the error.

6.2. Woflan

Verbeek et al. [3] report on two case studies that applied Woflan to analyze soundness of industrial business process models. The case studies primarily aimed at how Woflan supports correcting erroneous process models. Altogether, it was shown that depending on the number of errors in a model, several analysis iterations are required to correct an erroneous model as some errors may not be reachable or be hidden by other errors. It is pointed out that besides detecting mismatches in the process logic, e. g., an AND-split closed by an XOR-join, counterexample traces provide the most useful diagnostic information for correcting an error. The two case studies in [3] also differ in how close the diagnostic information provided by Woflan maps to the original process definition. In the first case study, the original process models almost directly map to Petri nets and thus, Woflan's diagnostic information could be mapped back making correction of erroneous models straight forward. In the second case study, the original process model and the analyzed Petri net did not correspond closely. In this case, the reported effort for locating and correcting the error in the original model was significantly higher. As in the case of IBM WebSphere Business

Modeler, these results confirm the importance of consumability of diagnostic information presented to a user.

6.3. LoLA

At this time, there is no particular industrial tool that routinely uses LoLA. However, LoLA is integrated in several platforms including the *Model Checking Kit* [32], the modeling tools *CPN-AMI* [33], *Petri Net Kernel* [34], and *PEP* [35], the workflow mining framework ProM [36], and the exploration tool for biochemical pathways *Pathway Logic Assistant* [37]. Successful case studies beyond the one reported here have been conducted, for instance, on asynchronous circuits [38], Web services [39], and Web service collaborations [40]. There is evidence that LoLA's performance is stable, more or less independently from the application domain as long as it is applied to concurrent systems: "For reachability queries on Pathway Logic nets, answering a reachability query that would have taken hours using a general purpose model-checking tool takes on the order of a second in LoLA — fast enough to permit interactive use" [37].

7. Conclusion

We showed that different techniques can be used to check the soundness of industrial business process models reliably in fractions of a second. Thereby, this paper mainly focused on the verification speed and only briefly touched other important topics such as understandability of the results with respect to the correction of faulty models. To evaluate these aspect, empirical studies and experiments with modelers are required. These experiments are out of scope of this paper and are subject to future work.

For the *state space approach* using LoLA, we found that partial order reduction and on-the-fly verification are the essential factors for success. Although many processes could have been verified on a brute force state space, some state spaces exploded without the use of partial order reduction. The exploration of erroneous state spaces up to the first error was efficient, but it was difficult to handle full state spaces. Surprisingly, a prior application of structural Petri net reduction has only a minor impact on performance. This may be caused by the fact that many existing reduction rules address situations that partial order reduction also resolves on the state space.

In the *structural approach* using Woflan, we saw that the original models can be easily translated into the more restrictive notion of workflow nets with just one sink node. Another observation was that the performance of Woflan can mainly be attributed to the structural Petri net techniques. In the few cases where Woflan had to explore a state space, this state space was rather small because of prior application of structural reduction. Here, structural reduction turned out to be beneficial as Woflan does not provide partial order reduction.

In the *decomposition approach* using SESE fragments, we learned that the approach did not suffer from severe state space explosion as the state space is only computed locally for a typically small fragment of the process model.

Moreover, structural heuristics are sufficient to handle most of the fragments, which allows one to avoid state space exploration altogether.

Although being similar in their performance, the three approaches vary with respect to the diagnostic information they provide. The state space approach used by LoLA is able to return an error trace of manageable size that can be simulated or animated. The SESE approach can detect multiple errors in one analysis run and localizes each error in a particular, typically small, fragment of the original model. This also reduces the length of the error trace by a factor of 4.7 on average. Moreover, the approach can provide additional information depending on the applied heuristics. Woflan returns Petri-net specific information that needs to be interpreted carefully before it can be shown to a business user. The tool can also generate counterexample traces upon user request, but it faces state space explosion because partial order reduction is not applied.

Another notable difference between the three approaches is that Woflan is specifically built for checking soundness and the SESE approach is specifically designed to check soundness instantaneously, whereas LoLA is a generic model checker for Petri nets that could more easily be adapted to check other temporal properties of business processes.

We would like to point out that there are other interesting algorithms to check soundness, especially polynomial-time algorithms exploiting the free-choice property [14]. We have no doubt that these algorithms would perform at least as fast as the techniques considered here, if properly implemented. However, we did not include those algorithms in our case study because they do not produce suitable diagnostic information for our use case.

Finally, the three approaches in this paper could be easily combined in different ways. For example, one could apply SESE decomposition to break the model into smaller fragments, then use heuristics and structural Petri net reduction to quickly sort out sound fragments that have a simple structure, and then finally check the remaining fragments with state space exploration based on partial order reduction to obtain detailed localized error information.

Acknowledgements

We thank Eric Verbeek for his substantial support in providing a Woflan version for our experiments. Dirk Fahland is funded by the DFG-Graduiertenkolleg "METRIK" (1324). Niels Lohmann and Karsten Wolf are supported by the DFG project "Operating Guidelines for Services" (WO 1466/8-1). Jana Koehler and Hagen Völzer were partially supported by the SUPER project (`http://www.ip-super.org`) under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850).

References

[1] J. Mendling, Empirical studies in process model verification, T. Petri Nets and Other Models of Concurrency 2 (2009) 208–224.

[2] W. M. P. v. d. Aalst, A. Hirnschall, H. M. W. E. Verbeek, An alternative way to analyze workflow graphs., in: CAiSE, Vol. 2348 of LNCS, Springer, 2002, pp. 535–552.

[3] H. M. W. E. Verbeek, T. Basten, W. M. P. v. d. Aalst, Diagnosing Workflow Processes using Woflan, Comput. J. 44 (4) (2001) 246–279.

[4] OMG, Business process model and notation (BPMN) FTF beta 2 for version 2.0, omg document number dtc/10-06-04, Tech. rep. (2010).

[5] K. Wolf, Generating Petri net state spaces, in: PETRI NETS 2007, Vol. 4546 of LNCS, Springer, 2007, pp. 29–42, invited lecture.

[6] J. Vanhatalo, H. Völzer, F. Leymann, Faster and more focused control-flow analysis for business process models through SESE decomposition, in: ICSOC 2007, Vol. 4749 of LNCS, Springer, 2007, pp. 43–55.

[7] B. F. v. Dongen, M. Jansen-Vullers, H. M. W. E. Verbeek, W. M. P. v. d. Aalst, Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants, Comput. Ind. 58 (6) (2007) 578–601.

[8] J. Mendling, G. Neumann, W. M. P. v. d. Aalst, Understanding the occurrence of errors in process models based on metrics, in: OTM 2007: CoopIS, DOA, ODBASE, GADA, and IS, Vol. 4803 of LNCS, Springer, 2007, pp. 113–130.

[9] J. Mendling, Detection and prediction of errors in EPC business process models, Ph.D. thesis, Vienna University of Economics and Business Administration (May 2007).

[10] J. Mendling, H. M. W. E. Verbeek, B. F. v. Dongen, W. M. P. v. d. Aalst, G. Neumann, Detection and prediction of errors in EPCs of the SAP reference model, Data Knowl. Eng. 64 (1) (2008) 312–329.

[11] M. Wynn, H. M. W. E. Verbeek, W. M. P. v. d. Aalst, A. H. M. t. Hofstede, Business process verification: Finally a reality!, Business Process Management Journal 15 (1) (2009) 74–92.

[12] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Instantaneous soundness checking of industrial business process models, in: U. Dayal, J. Eder, J. Koehler, H. A. Reijers (Eds.), BPM, Vol. 5701 of Lecture Notes in Computer Science, Springer, 2009, pp. 278–293.

[13] D. Fahland, Translating UML2 activity diagrams to Petri nets, Informatik-Berichte 226, Humboldt-Universität zu Berlin, Berlin, Germany (2008).

[14] J. Desel, J. Esparza, Free choice Petri nets, Cambridge University Press, New York, NY, USA, 1995.

[15] W. M. P. v. d. Aalst, The application of Petri nets to workflow management, Journal of Circuits, Systems and Computers 8 (1) (1998) 21–66.

[16] B. Kiepuszewski, A. H. M. t. Hofstede, W. M. P. v. d. Aalst, Fundamentals of control flow in workflows, Acta Inf. 39 (3) (2003) 143–209.

[17] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[18] J. Billington, S. Christensen, K. M. v. Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber, The Petri net markup language: Concepts, technology, and tools, in: ICATPN 2003, LNCS 2679, Springer, 2003, pp. 483–505.

[19] LoLA v1.11 available at, `http://service-technology.org/lola`.

[20] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263.

[21] A. Valmari, Stubborn sets for reduced state space generation, in: G. Rozenberg (Ed.), Applications and Theory of Petri Nets, Vol. 483 of Lecture Notes in Computer Science, Springer, 1989, pp. 491–515.

[22] P. Godefroid, Using partial orders to improve automatic verification methods, in: E. M. Clarke, R. P. Kurshan (Eds.), CAV, Vol. 531 of Lecture Notes in Computer Science, Springer, 1990, pp. 176–185.

[23] D. Peled, All from one, one for all: on model checking using representatives, in: C. Courcoubetis (Ed.), CAV, Vol. 697 of Lecture Notes in Computer Science, Springer, 1993, pp. 409–423.

[24] R. Gerth, R. Kuiper, D. Peled, W. Penczek, A partial order approach to branching time logic model checking, in: ISTCS, 1995, pp. 130–139.

[25] K. Schmidt, Stubborn sets for model checking the EF/AG fragment of CTL, Fundam. Inform. 43 (1-4) (2000) 331–341.

[26] L. M. Kristensen, K. Schmidt, A. Valmari, Question-guided stubborn set methods for state properties, Formal Methods in System Design 29 (3) (2006) 215–251.

[27] A. Valmari, A stubborn attack on state explosion, Formal Methods in System Design 1 (4) (1992) 297–322.

[28] K. Schmidt, Stubborn sets for standard properties, in: Applications and Theory of Petri Nets 1999: 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA, June 1999. Proceedings, Vol. 1639 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 46–65.

[29] UML2oWFN compiler available at, `http://service-technology.org/uml2owfn`.

[30] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, in: Business Process Management, Vol. 5240 of LNCS, Springer, 2008, pp. 100–115.

[31] T. Gschwind, J. Koehler, J. Wong, C. Favre, W. Kleinoeder, A. Maystrenko, K. Muhidini, IBM WebSphere pattern-based process model accelerators for WebSphere Business Modeler, IBM developerWorks (2009).

[32] C. Schröter, S. Schwoon, J. Esparza, The Model-Checking Kit, in: ICATPN 2003, LNCS 2679, Springer, 2003, pp. 463–472.

[33] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, Y. Thierry-Mieg, New features in CPN-AMI 3: focusing on the analysis of complex distributed systems, in: ACSD, IEEE, 2006, pp. 273–275.

[34] M. Weber, E. Kindler, The Petri Net Kernel, in: Petri Net Technology for Communication-Based Systems, LNCS 2472, Springer, 2003, pp. 109–124.

[35] B. Grahlmann, The PEP tool, in: CAV '97, LNCS 1254, Springer, 1997, pp. 440–443.

[36] W. M. P. v. d. Aalst, B. F. v. Dongen, C. W. Günther, R. S. Mans, A. K. A. de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. E. Verbeek, A. J. M. M. Weijters, Prom 4.0: Comprehensive support for *real* process analysis, in: ICATPN 2007, LNCS 4546, Springer, 2007, pp. 484–494.

[37] C. L. Talcott, D. L. Dill, Multiple representations of biological processes, T. Comp. Sys. Biology LNCS 4220 (VI) (2006) 221–245.

[38] C. Stahl, W. Reisig, M. Krstic, Hazard detection in a GALS wrapper: A case study, in: ACSD'05, IEEE, 2005, pp. 234–243.

[39] S. Hinz, K. Schmidt, C. Stahl, Transforming BPEL to Petri nets, in: BPM 2005, LNCS 3649, Springer, 2005, pp. 220–235.

[40] N. Lohmann, O. Kopp, F. Leymann, W. Reisig, Analyzing BPEL4Chor: Verification and participant synthesis, in: WS-FM 2007, LNCS 4937, Springer, 2008, pp. 46–60.