Process Compliance Checking using Taint Flow Analysis

Completed Research Paper

Alexander Seeliger
Technische Universität Darmstadt
Darmstadt, Germany
seeliger@tk.tu-darmstadt.de

Timo Nolle
Technische Universität Darmstadt
Darmstadt, Germany
nolle@tk.tu-darmstadt.de

Benedikt Schmidt
Technische Universität Darmstadt
Darmstadt, Germany
benedikt.schmidt@tk.informatik.tu-darmstadt.de

Max Mühlhäuser
Technische Universität Darmstadt
Darmstadt, Germany
max@tk.tu-darmstadt.de

Abstract

Due to the growing complexity of processes, regulations, policies and guidelines (e.g., Sarbanes-Oxley-Act) computer-assisted business process analysis - known as process mining - is becoming more and more relevant for organisations. One discipline of process mining is backward compliance checking, which aims to detect non-compliant process variants based on historic data. Most existing approaches compare the "as-is" view with desired process models. However, most organisations do not maintain such models, making such approaches less attractive. This paper proposes a process flow analysis which uses graph-reachability to check whether the actual "as-is" process graph violates compliance constraints. Our approach is inspired by the taint flow algorithm which is used in code analysis to identify security vulnerabilities in software applications. We conducted a case study evaluating the compliance of event logs and performed a benchmark to show that our approach outperforms the LTL checker and the PetriNet pattern approach in ProM.

Keywords: compliance checking; process mining; conformance checking; taint flow analysis.

Introduction

Process mining aims to extract interesting knowledge from event logs, recorded by *Process-Aware Information Systems* (PAISs). One important research topic of process mining is compliance checking, e.g. ensuring that all purchase orders have been approved by the division manager (Abdullah et al. 2010). Businesses are forced to guarantee that process executions are compliant to different regulations, policies and guidelines such as the Sarbanes and Oxley (2002) Act or Six Sigma. Because compliance violations in processes may lead to legal procedures and unexpected costs, organisations need tools and methods that support them to avoid such violations. Compliance checking can be divided into forward and backward compliance checking (Ramezani et al. 2012). Forward compliance checking approaches focus on the enforcement of compliance by implementing technical restrictions and monitoring mechanisms that operate during the execution of a process. Using such methods helps to actively prevent malicious process executions by just allowing specific actions. Such methods may be needed in the financial sector where a strict process execution must be enforced. However, restricting the system to a predefined sequence of actions reduces the ability to react to the dynamic environment: for example allowing to bypass certain process steps during order peaks. Each change needs to be reflected in the executing information system, which may lead to high adjustment costs. In contrast, backward compliance checking aims to detect compliance violations from historic process execution data recorded in event logs (also known as "after-the-fact" detection), allowing to prevent compliance violations in the future. This approach has the advantage that the agility and flexibility during the process execution is not restricted. It still allows to gather information about when and why compliance violations happened in order to take further action. The approach presented in this paper is a backward compliance checking method.

A main goal of compliance checking is to compare the desired behaviour of a process (the way a process ought to be executed) with the observed behaviour (e.g. gathered from recorded event logs). Many backward compliance checking approaches use business process models (e.g. modelled in BPMN) to determine differentiating executions or formulate Linear Temporal Logic (LTL) expressions to check compliance rules. Due to the lack of up-to-date models, the use of predefined business process models to check compliance is often not possible. Additionally, the description of such models needs to be very precise in order to provide a correct and reliable analysis result. However, in reality small- and middle-sized businesses do not maintain such machine-readable process models. Besides missing formal process descriptions in the real world, the execution of processes often differs because of technical limitations or temporary instructions which are not reflected in the model. Often only a textual documentation of the desired process is available which cannot easily be used in automatic backward compliance checking methods. The use of LTL model checkers is more promising because defining compliance rules from the textual documentation is easier and less time consuming. However, writing LTL expressions is complicated for non-experts and the computation time for large event logs without reducing the search space is high (Leoni et al. 2012). Furthermore, LTL checkers do not provide useful detailed diagnostics for an investigation of the violation.

This paper proposes an algorithm for compliance checking which does not require a precise process model and, therefore a definition of all compliant execution paths. Instead, we detect compliance violations by transforming the problem into a graph-reachability problem. Image a process graph where each node corresponds to an activity and each edge represents an executed transition between two activities. Various activities in the graph may produce a sensitive flow (e.g. the order of a high amount of goods) that needs to be approved (e.g. by the department manager) before further actions can be executed safely. The proposed approach detects a compliance violation, if there exists a realisable flow between a sensitive producer activity and a consumer activity without a sanitiser activity in between (e.g. an order is released without an approval of the manager). We adapted the taint flow analysis (Guarnieri et al. 2011) approach for compliance checking and solve the graph reachability problem using the tabulation algorithm (Reps et al. 1995). In comparison to the LTL model checker, our approach delivers detailed diagnostics why a specific compliance rule is violated. It returns the realisable paths (the sequences of activities) which are not compliant. Graphically the system delivers a process graph with coloured paths.

The main contributions of this paper are:

- The use of a taint flow analysis to check compliance of processes by formulating compliance checking as a graph-reachability problem which can efficiently be solved using the tabulation algorithm.
- The modifications to the original algorithm so that it can be applied to historic process execution data to detect compliance violations.
- A case study using two real event logs extracted from SAP ERP systems of two organisations and a synthetic event log generated from a BPMN model. We compared our approach with the existing LTL checker and the PetriNet pattern approach in ProM.

The paper is structured as follows. First, we introduce related work. Second, we describe how to transform compliance checking into a graph-reachability problem. Third, we introduce required modifications to the original tabulation algorithm that is used to solve the graph reachability problem. Next we describe our case study of the approach and compare it to the existing LTL checker and the PetriNet pattern approach. Finally, the paper concludes with a discussion section and future outlook as well as a short summary.

Related Work

Backward compliance checking is a large research field in process mining. Historic event logs are used to determine which process executions did comply with the desired execution strategy and which ones violate it. Because there exist various backward compliance check approaches (Fellmann and Zasada 2014), we limit ourselves to related work which performs compliance checking on the control flow (Curtis et al. 1992).

One approach of backward compliance checking is to compare the discovered process model with a desired model (Cook and Wolf 1999). Desired process models are often created manually from textual documentations. Petri-nets, Event-driven Process Chains (EPCs) or Business Process Model Notation (BPMN) are common description languages for such process models. To solve the problem of missing process models, reference models can also be used for comparison (Gerke et al. 2009). The challenge for all model-based approaches is to find a match between the event log and the process model (Baier et al. 2014). Leoni et al. (2012) use declarative models to check compliance. To quantify the conformance of an event log Van Der Aalst et al. (2012) propose a fitness value which expresses how much an observed event log differs from a process model, and therefore, gives an indicator for violation level. In Muñoz-Gama et al. (2010) different fitness measures are proposed which calculate the relation between the number and frequency of escaping edges. Weidlich et al. (2011) compute various compliance measures based on behavioural characteristics of the process. Other approaches extend the existing BPMN with descriptions of immediate effects of activities and sub-processes to identify compliance violations (Ghose and Koliadis 2007).

Instead of comparing process models, other approaches use predefined compliance rules to detect violations which has the advantage that no complete desired process model needs to be created. Complex compliance rules can be checked via Linear Temporal Logic (LTL) (Van Der Aalst et al. 2005). To simplify the definition of formal LTL rules, in Awad et al. (2008) and Awad and Weske (2009) a visual query language (BPMN-Q) is introduced which allows to design compliance rules (such as presence, absence, and/or the order of activities) which are then formed into past linear temporal logic (PLTL) expressions. A model checker then allows to query for non-compliant cases. (Governatori and Rotolo 2008; Sadiq et al. 2007) introduce a Formal Contract Language to express normative specifications of the process. SeaFlows (Ly et al. 2010) uses a graph representation to define process-independent compliance rules. Verification for a concrete process instance is realized by using domain models which contain the relations between compliance rule and concrete process. Ramezani et al. (2012) propose 55 predefined control flow oriented PetiNet patterns which can be easily configured for specific processes. Temporal compliance checking is performed by aligning the event log to the corresponding pattern. A different approach uses SQL queries for declarative process mining, allowing to check certain event log constraints (Schönig 2015). In Caron et al. (2013) a business rule taxonomy for process mining is introduced which also covers the control flow perspective, using a combination of first order logic, linear temporal logic and logical connectors for patterns.

Compliance Checking as a Reachability Problem

Backward compliance checking approaches try to find past compliance violations in historic process executions. In general, compliance checking can focus on four different process perspectives (Caron et al. 2013; Curtis et al. 1992): functional, control-flow, organisational and data process perspective. We will focus on the control-flow perspective, analysing the execution of processes regarding the existence and order of activities. For example, in the procurement process goods must be received before the invoice is payed. Instead of comparing the desired execution model (e.g. formalised in BPMN) with the "as-is" process model, we search for sensitive flows (e.g. sequences of activities where the invoice is payed before goods are received) in the process graph to find compliance violations.

Our approach is divided into three computational steps (see Figure 1). In the first step, we convert the input event log into a process graph using the process variants computed from the event log. The resulting process graph is then used to compute an exploded supergraph that maintains the sensitive paths that correspond to a single compliance check. In the third and last step, our approach replays the process variants onto the exploded supergraph to identify the variants that violate the given compliance specifications. Furthermore, our approach returns the paths (the sequences of activities) in the graph that

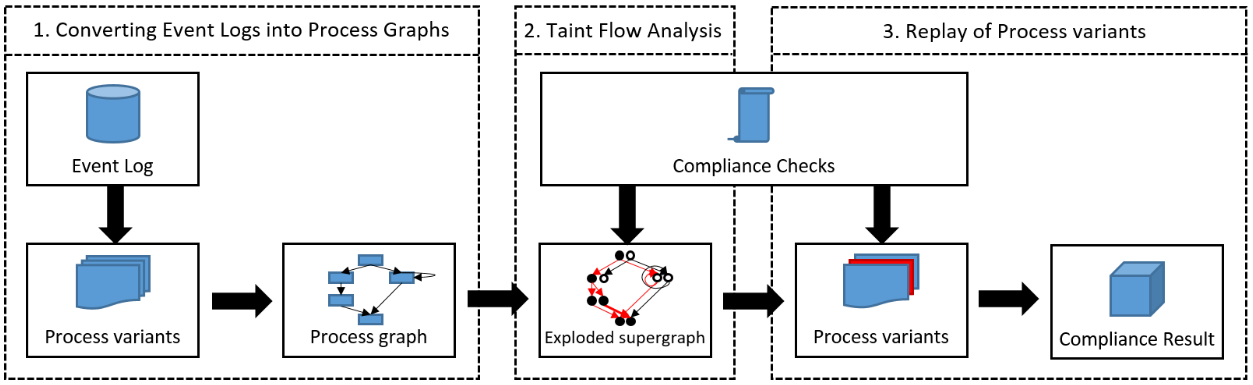


Figure 1. Overview of the steps performed during the Compliance Checking using Taint Flow Analysis.

are not compliant. If we know the violating process variants, we can also find out which cases violate against the compliance check. Consequently further analysis why a specific violation occurs can be performed. In the following we will describe our approach in detail.

Converting Event Logs into Process Graphs

Before our approach can be applied, we need to convert the event log into a process graph. An event log usually consists of multiple cases that can be distinguished by a unique case identifier (case id). Each case consists of a sequence of activities that has been executed in a specific order.

Definition 1. ***(Event Log)*** *An event log $L = \{s_1, s_2, \dots, s_m\}$ is a set of m cases $s_i = \{a_1, a_2, \dots, a_n\}$. Each case is a sequence of n activities a_j.*

The process graph is constructed so that it contains all activity sequences of the event log. Each node represents an activity and each edge corresponds to an observed transition between two activities (the transition exists at least one time in the event log). If activities are executed multiple times in a sequence, then the process graph will also contain loops because there will be an edge from the end of the repeating sequence to the beginning node. We generate the process graph from seen sequence activity variants in the event log which reduces the computation time. A variant is a distinct sequence of activities, that occurred one or more times in the event log.

Definition 2. ***(Process Graph)*** *A process graph PG is a directed graph $PG = (N^*, E^*)$ with N^* as nodes and E^* as edges. Each PG has a unique start node s_p, and a unique exit node e_p. Other nodes represent activities in the process which are connected using edges $e_i = (n_{source}, n_{target})$ to describe the transition between two activities.*

The process graph PG^* is generated from the following formalisation: Let L be an event log then $PG^* = (N^*, E^*)$ with

$$\begin{aligned} N^* &= \{a \mid \forall s \in L : a \in s\} \\ &\quad \cup \{s_p, e_p\} \\ E^* &= \{(a_i, a_j) \mid \forall s \in L, \forall (a_i, a_j) \in s\} \\ &\quad \cup \{(s_p, a_0) \mid \forall s \in L : (a_0, a_n) \in s\} \\ &\quad \cup \{(a_n, e_p) \mid \forall s \in L : (a_{n-1}, a_n) \in s\} \end{aligned}$$

The process graph now contains all seen activity sequences in a summarized graph. Additionally, a start and an exit node exist that are connected to the start respectively the last activity in the sequences. We can describe each possible activity sequence of the process graph as a path:

Definition 3. ***(Path)*** *A path in PG^* of length j from node m to node n is a sequence of j edges. The sequence is denoted by $[e_1, e_2, \ldots, e_j]$, such that for all $i, 1 \leq i \leq j-1$, the target of edge e_i is the source of edge e_{i+1}.*

Because the process graph is generated from activity sequences by connecting all successor activities, the graph may contain more possible paths than seen in the original log. Additionally, loops allow the production of an unlimited number of sequences, increasing the search space of violating process paths. We will see later how both issues are efficiently solved by the tabulation algorithm.

Taint Flow Analysis

Our proposed approach is inspired by the static analysis of program code to find security vulnerabilities in software applications called taint flow analysis. Guarnieri et al. (2011) use the taint flow analysis algorithm to identify possible dangerous data flows in JavaScript web applications. The basic idea is that every data source (e.g. a user input) needs to be sanitised before it can be securely processed by a sensitive data consumer. Compliance checking can be achieved using the same idea: a process graph consists of different activities that may produce sensitive flows which need to be sanitised before a specific activity can be executed securely. For example, in the procurement process the order of goods is such a sensitive producer. Before the activity *"Invoice payment"* is executed, the process should verify that the *"Goods received"* activity has been executed (see Figure 2, left). In this case our approach searches for activity sequences (sensitive flows) in the process graph that do not have such a sanitiser activity. In general we can say that a sensitive flow needs to be handled by a sanitiser activity before another specific activity can be executed securely.

We can formulate such a compliance check (CC) as a triple, consisting of producer, sanitiser and consumer activities:

Definition 4. ***(Compliance Check)*** *A compliance check is defined as a triple*

$$CC^* = (S_{producer}, S_{sanitiser}, S_{consumer})$$

where $S_{producer}, S_{sanitiser}, S_{consumer}$ are sets of activities with the following properties:

1. *A* ***producer*** *activity denotes the production of a sensitive flow in the process graph. The sensitive flow requires that another activity needs to be executed before a consumer activity is executed.*
2. *A* ***sanitiser*** *activity destroys a corresponding sensitive flow, so that a following consumer activity can be executed without a compliance violation.*
3. *A* ***consumer*** *activity denotes the activity that is sensitive to a flow. If there exists a sensitive path between a consumer and a producer, the corresponding compliance check would detect a violation.*

Each CC defines a path in the process graph where the order of activities is defined as $producer \xrightarrow{requires} sanitiser \xrightarrow{before} consumer$. Between a producer and a sanitiser respectively sanitiser and consumer a random number of other activities (e.g. technical activities) can be executed. Our approach does not need to know which activities are executed besides the ones defined in a CC. This opens up the ability to use the proposed method in scenarios where the full process is not known beforehand. A compliance violation is detected if there exists a sensitive flow between the producer and the consumer without a sanitiser in between. For each CC a separate sensitive flow is created, allowing to define and validate multiple compliance checks at the same time. A sensitive flow can only be sanitised by the corresponding sanitiser, likewise a compliance violation is only detected if the corresponding sensitive flow exists between a producer and consumer.

Let's consider a simple example (see Figure 2): the activity *"Purchase Order"* is a sensitive producer, *"Goods Received"* a sanitiser and *"Invoice Payment"* a sensitive consumer. We can see that a sensitive flow is created at process node n_2 (depicted by the red path and closed dots) because the n_2 is a sensitive producer. The flow is then forwarded at node n_3 because the corresponding activity is not present in the CC. At node n_4 the sensitive flow is destroyed because *"Goods Received"* is a sanitiser. To check the compliance of this example, we only need to check if there exists a flow between n_2 (*"Purchase Order"*) and n_5 (*"Invoice Payment"*). In the left example this is not the case because the flow has already been

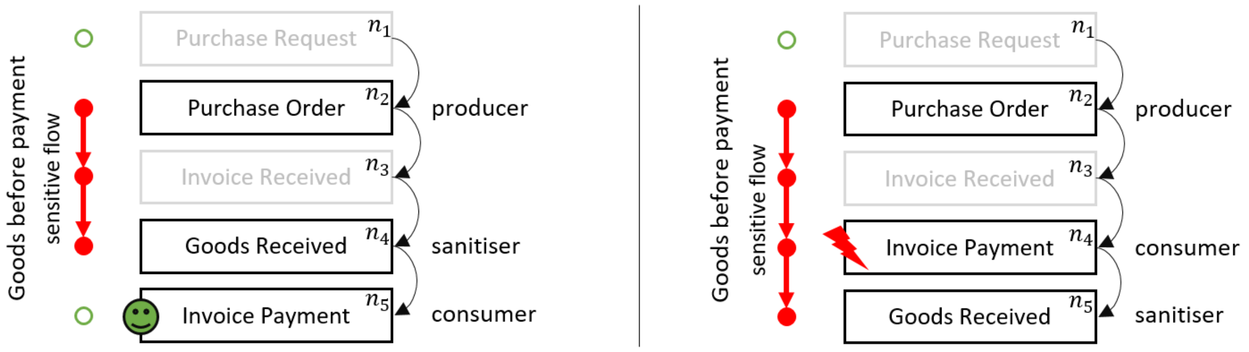


Figure 2. Example of a simple procurement process graph. On the left side, the process execution is compliant because the activity *Goods Received* (sanitiser) is executed between *Purchase Order* (producer) and *Invoice Payment* (consumer).

destroyed before at node n_4. On the right-hand side the compliance rule is violated because there exists a flow between nodes n_2 and n_4.

To solve such data flow problems, Reps et al. (1995) have proposed a general analysis framework which allows to transform various data flow problems into graph-reachability problems. Instead of searching for paths between producer and consumer activities in the graph - which would require to traverse the graph for each compliance check separately -, the framework generates a single exploded supergraph that contains information about the realisation of paths (i.e. the flow between producer and consumer) in the original process graph. The supergraph is constructed so that searching for paths between two nodes can be achieved by calculating the graph-reachability. If there exists a realisable path in the exploded supergraph between the start node and a consumer node, which corresponds to a flow between a producer and consumer node in the process graph, then a violation of the corresponding compliance check is detected. Paths in the exploded supergraph are created for each occurrence of a producer node and destroyed for each occurrence of a sanitiser node. However, producer and sanitiser can only create or destroy paths that correspond to their compliance check.

In the following, we provide a full definition of the taint flow analysis also and highlight required changes to the original work (Reps et al. 1995) to check compliance of processes. The taint flow analysis problem in the context of process compliance checking is defined as follows:

Definition 5. *(Taint Flow Analysis)* *An instance TFA of a Taint Flow Analysis problem is a four-tuple, $TFA = (PG^*, D, F, M)$, where*

1. *PG^* is a process graph as defined in definition 2.*
2. *D is a finite set of compliance checks (see definition 4).*
3. *$F \subseteq 2^D \rightarrow 2^D$ is a set of distributive functions.*
4. *$M: E^* \rightarrow F$ is a map from PG^*'s edges to distributive functions.*

A taint flow analysis consists of the process graph, a set of compliance checks that should be verified, a set of distributive functions that describe the production, destruction and forwarding of sensitive flows as well as a map that assigns graph edges to distributive functions. A distributive function determines if a sensitive flow is produced, destroyed or forwarded to the next activity. Because a separate sensitive flow is maintained for each compliance check, distributive functions are defined such that they consume and return a set of sensitive flows. The sensitive flows of all compliance checks need to be calculated for each transition between two activities. M contains the assignment of a distributive function to the edges in PG^*. For a moment, let us assume we have defined a distributive function df that handles sensitive flows correctly.

The next definition shows that distributive functions can be concatenated so that the outcome of a sensitive flow can be calculated for a given activity path q in the process graph:

Definition 6. *Let $TFA = (PG^*, D, F, M)$ be a TFA problem instance, and let $q = [e_1, e_2, \ldots, e_j]$ be a non-empty path in PG^*. The corresponding path function $pf_q =_{df} f_j \circ \ldots \circ f_2 \circ f_1$ for all i, $1 \leq i \leq j, f_i = M(e_i)$. The path function for an empty path is the identity function.*

This definition lets us calculate if there exists a path between the start node s_p and any other node n in the process graph. This allows us to determine if there is sensitive flow between s_p and node n. If n is a sensitive consumer for this sensitive flow (it corresponds to the same compliance check), the compliance check will return a violation. We can calculate sensitive flows for each compliance check and for each node using the following formula, which is also the result of the taint flow analysis:

Definition 7. *Let $TFA = (PG^*, D, F, M)$ be a TFA problem instance. The meet-over-all-valid-paths solution of TFA is a collection of values MVP_n defined as follows:*

$$MVP_n = \bigcup_{q \in TVP(s_p, n)} pf_q(\top) \quad for\ each\ \ n \in N^*$$

$TVP(m, n)$ is a function that returns a set of all possible paths between nodes m and n. MVP_n contains all compliance checks $d \in D$ that are violated at node n, respectively there exists a path between s_p and n for compliance check d which is not destroyed by a sanitiser. Finally, we only need to check each sensitive consumer node n if MVP_n contains a compliance check $d \in D$ that contains n as a sensitive consumer. A compliance violation is detected, if the sensitive flow and the consumer correspond to the same compliance check.

Distributive Functions

Distributive functions determine if a sensitive flow is produced, destroyed or forwarded between two process activities in the process graph. Each sensitive flow corresponds to exactly one compliance check which in turn consists of three sets: producer, sanitiser and consumer activities. Besides the individual sensitive flows that are handled for each compliance check, an additional **0**-flow is maintained that will always be forwarded. This **0**-flow is called the taint-flow. Each flow that origins from a **0**-flow will also be tainted. We will see later, why the **0**-flow is required and how it allows an efficient detection of compliance violations. We define a distributive function $f : 2^D \rightarrow 2^D$ as follows:

Definition 8. ***(Compact representation of a distributive function)*** *The representation relation $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$ for a function f is defined as follows:*

$$\begin{aligned} R_f &=_{df} \{(\mathbf{0}, \mathbf{0})\} \\ &\cup \{(\mathbf{0}, y) : y \in f(\mathbf{0})\} \\ &\cup \{(x, y) : y \in f(\{x\}) \ \ and \ \ y \notin f(\mathbf{0})\} \end{aligned}$$

R_f is a relation based on the given function f which describes the sensitive flow between two activities given an input flow. f returns a set of sensitive flows. Graphically, R_f is a graph with $2 \cdot (|D| + 1)$ nodes, where each node represents a compliance check $d \in D$ except for the **0**-nodes. The **0**-flow is tainted by definition and taint flow can also only be produced by creating a connection from the **0**-flow.

Reps et al. (1995) show that the composition of $f \circ g$ relates to the relational composition of R_f ; R_g. In this paper we will not introduce the proof. The interested reader is referred to the original paper.

Figure 3 shows how the sensitive flows for compliance checks a and b can be modified through the transition between two activities:

1. The first example shows the **forwarding** of flows, in which no producer or sanitiser activity is involved. For each $d \in D$ the sensitive flow is forwarded, the following path is generated:
$$\langle n_{src}, d \rangle \rightarrow \langle n_{dest}, d \rangle$$
2. The second example shows the **production** of the sensitive flow for compliance check a. In this case, a producer activity is executed, that produces a sensitive flow and corresponds to compliance check a. We can see that the production of a sensitive flow a emerges from the **0**-flow.

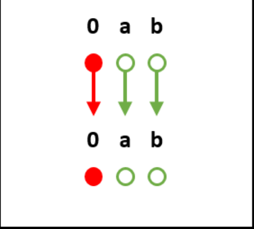
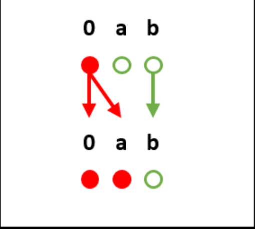
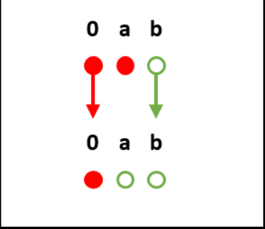
forward(a,b)	creation(a) forward(b)	destroy(a) forward(b)
$f(0) = \{\}$ $f(\{a\}) = \{a\}$ $f(\{b\}) = \{b\}$	$f(0) = \{a\}$ $f(\{a\}) = \{a\}$ $f(\{b\}) = \{b\}$	$f(0) = \{\}$ $f(\{a\}) = \{\}$ $f(\{b\}) = \{b\}$
		

Figure 3. Example flows for production, destruction and forwarding represented as a graph. Taint-flows are marked as red and with an open circle.

Because the **0**-flow is always tainted (coloured in red) by definition, the produced sensitive flow is also tainted after the execution of a producer activity. The exploded supergraph results in the following path:

$$\langle n_{src}, \mathbf{0} \rangle \rightarrow \langle n_{dest}, d \rangle$$

3. The third example shows how a sensitive flow is **destroyed** by a sanitiser activity. The corresponding sensitive flow is interrupted at the transition from a sanitiser to another activity, so that the sensitive flow is not tainted anymore. In this case, the corresponding flow is not forwarded.

To model these three behaviours we need to define a general flow function f that returns how sensitive flows need to be modified for a given transition between n_{src} and n_{dest} in a process graph:

$$f_{(n_{src}, n_{dest})}(x) = \begin{cases} \{d : d \in producers(n_{src})\} & if\ x = \mathbf{0} \\ \emptyset & if\ x \in sanitisers(n_{src}) \\ \{x\} & otherwise \end{cases}$$

$producers(n)$ and $sanitisers(n)$ are functions which return a set of compliance checks that contain n as a producer or as a sanitiser activity respectively. The defined general distributive function can be used at every edge in the process graph which results in $M\left(e_{n_{src}, n_{dest}}\right) = f_{(n_{src}, n_{dest})}$ with $e_{n_{src}, n_{dest}} = (n_{src}, n_{dest})$.

The sensitive flow information for each transition and each compliance check is stored in the exploded supergraph. The exploded supergraph's edges correspond to the previously defined relations (see definition 8) on the edges of the process graph PG^*. To maintain the flow information for each compliance check, each node exists $|C| + 1$-times in the supergraph. Besides the process graph corresponding edges and nodes, the additional **0**-flow needs to be maintained.

Definition 9. *(Exploded Supergraph)* *Let $TFA = (PG^*, D, F, M)$ a taint flow analysis problem instance, then the corresponding exploded supergraph $PG^\#$ is defined as follows:*

$$PG^\# = (N^\#, E^\#), where$$

$$N^\# = N^* \times (D \cup \{\mathbf{0}\})$$

$$E^\# = \{\langle m, d_1 \rangle \rightarrow \langle n, d_2 \rangle : (m, n) \in E^* \ and\ (d_1, d_2) \in R_{M(m,n)}\}$$

Each node in the exploded supergraph has the form $\langle n, d \rangle$ where $n \in N^*$ is the actual node (i.e. activity), and $d \in D$ a compliance check. Each edge is exploded into multiple edges $\langle m, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ where each edge represents the sensitive flow (d_1, d_2) of the corresponding compliance check. A compliance violation of the corresponding check d exists in $PG^\#$ if there is a realisable path from node $\langle s_p, \mathbf{0} \rangle$ to $\langle n, d \rangle$ (compare theorem 3.8. in (Reps et al. 1995)) and n is a sensitive consumer for d.

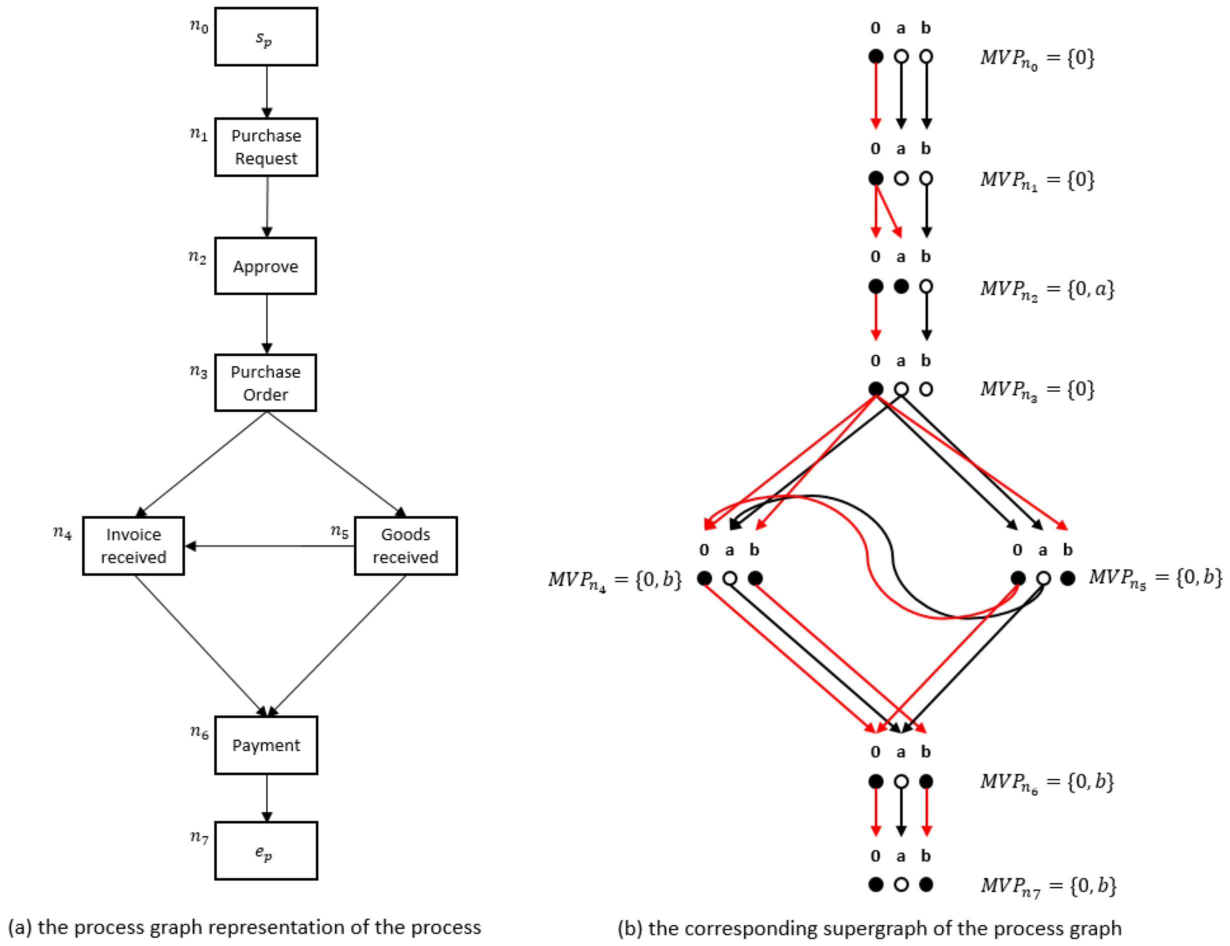


Figure 4. An example procure-to-pay process represented as a Process Graph (a) and the corresponding exploded supergraph (b). In the example, we two compliance checks [a] producer: n_1, sanitiser: n_2, consumer: n_3; [b] producer: n_3, sanitiser: n_5, consumer: n_6. Open dots indicate that this flow is not reachable from $\langle s_p, 0\rangle$ whereas closed dots indicate that there exists a flow from $\langle s_p, 0\rangle$.

In Figure 4 (b) each reachable node is marked as a closed circle whereas a non-reachable node is marked as an open circle in the graph. The example depicted in the figure shows two different CCs a and b. CC a consists of the producer activity *"Purchase Request"*, the sanitiser activity *"Approve"* and the consumer activity *"Purchase Order"*. CC a is not violated because every sequence in the process graph contains an *"Approve"* activity between *"Purchase Request"* and *"Purchase Order"*. We can see that there is no forwarding edge $\langle n_2, a\rangle \rightarrow \langle n_3, a\rangle$ because n_2 is a sanitiser for compliance check a. CC b consists of the producer activity *"Purchase Order"*, the sanitiser activity *"Goods received"* and the consumer activity *"Payment"*. In the example we have three paths that are relevant for b. For the first path $\langle n_2, b\rangle \rightarrow \langle n_4, b\rangle \rightarrow \langle n_6, b\rangle$ the compliance check will fail because there is no sanitiser activity on the path. The second $\langle n_2, b\rangle \rightarrow \langle n_5, b\rangle \rightarrow \langle n_6, b\rangle$ and third path $\langle n_2, b\rangle \rightarrow \langle n_5, b\rangle \rightarrow \langle n_4, b\rangle \rightarrow \langle n_6, b\rangle$ do have a sanitiser activity. Although two paths exist that contain the sanitiser n_5, the compliance check will fail because it is sufficient to find a single path that is not compliant.

If the control flow compliance check can be formulated as a triple of activities (cf. definition 4) then the presented approach delivers the compliance violations in the process graph correctly. The formal proof for the conversion of the flow problem into a graph-reachability problem can be found in the original paper (Reps et al. 1995). Because our approach solves the compliance checking problem via graph-reachability, we can return sequences of activities that fail the compliance check. In comparison to other approaches which only provide the fact that there is a violation, our approach delivers information that allows a deeper understanding why a specific compliance check is violated. There is no additional calculation required to extract this valuable information.

Tabulation Algorithm

In this section the Tabulation Algorithm (Reps et al. 1995) is presented in detail which can solve the realisable-path reachability problem described in the previous sections. The original version of the tabulation algorithm is used for static code analysis of computer applications. It considers function calls and return sides which are not required in the context of compliance checking.

The Tabulation Algorithm is a worklist algorithm which manages a list of all existing *PathEdges* in graph $PG^{\#}$. A PathEdge represents a realisable path in the process graph. Instead of managing path edges from various producer nodes, only realizable paths from the start node s_p (taint flows) are considered such that a PathEdge is always of the form $\langle s_p, d_1\rangle \rightarrow \langle n, d_1\rangle$. The worklist is initialised with the PathEdge $\langle s_p, \mathbf{0}\rangle \rightarrow \langle s_p, \mathbf{0}\rangle$. For each iteration of the main loop in the ForwardTabulateSLRP procedure (lines [21]-[35]), the algorithm looks for path edges that have not yet been processed. Because the process graph may contain loops, only new and unseen path edges are added to the PathEdge list. The algorithm only needs to calculate the flow between two activities in the process graph one time, so the algorithm will terminate after each process graph edge has been visited. New path edges $\langle s_p, \mathbf{0}\rangle \rightarrow \langle m, d_3\rangle$ are propagated when there exists an edge of the form $\langle n_2, d_2\rangle \rightarrow \langle m, d_3\rangle \in E^{\#}$ (lines [26]-[28]). In the actual implementation, the algorithm calls the corresponding distributive function $f_{(n_2,m)}$ to determine the flow of the transition between n_2 and m for compliance check d_2.

The following pseudo code demonstrates how the Tabulation Algorithm works:

```
1   declare PathEdge, WorkList : global edge set
2   algorithm Tabulate($G^{\#}$)
3   begin
4           let $(N^{\#}, E^{\#}) = G^{\#}$
5           $PathEdge := \{\langle s_p, 0\rangle \rightarrow \langle s_p, 0\rangle\}$
6           $WorkList := \{\langle s_p, 0\rangle \rightarrow \langle s_p, 0\rangle\}$
7           ForwardTabulateSLRPs()
8           for each $n \in N^{*}$ do
9                   $X_n := \{d_2 \in D : \exists\, d_1 \in (D \cup \{0\})\ such\ that\ \langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle \in PathEdge\}$
10          od
11  end
12
13  procedure Propagate(e)
14  begin
15          if $e \notin PathEdge$ then
16                  insert $e$ into $PathEdge$
17                  insert $e$ into $WorkList$
18          fi
19  end
20
21  procedure ForwardTabulateSLRPs()
22  begin
23          while $WorkList \neq \emptyset$ do
24                  select and remove an edge $\langle n_1, d_1\rangle \rightarrow \langle n_2, d_2\rangle$ from $WorkList$
25
26                  for each $\langle m, d_3\rangle$ such that $\langle n_2, d_2\rangle \rightarrow \langle m, d_3\rangle \in E^{\#}$ do
27                          Propagate($\langle s_p, d_1\rangle \rightarrow \langle m, d_3\rangle$)
28                  od
29
30                  if $WorkList = \emptyset$ then
31                          pop and remove $e$ from $Stack$
32                          insert $e$ into $WorkList$
33                  fi
34          od
35  end
```

In the final step of the Tabulation Algorithm a set of failed compliance checks $d \in D$ is generated for each node n (lines [8]-[10]). A compliance check d fails if there is a realisable path for d to node n which is equivalent to the existence of a path edge such that $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in PathEdge$ and $d_1 \in (D \cup \{\mathbf{0}\})$. The variable X_n for a node n contains the compliance check sensitive flows for which a realisable path from the start node s_p to node n exists. If node n is a consumer activity of a compliance check $d \in D$ and d is in the set of X_n then a compliance violation is detected.

The cost of the tabulation algorithm depends on the number of edges and the number of compliance checks that should be verified. For each edge, the tabulation algorithm needs to calculate the outcome of the distributive function, determine which sensitive flows are generated, forwarded and destroyed. In our case, the existence or order of activities checking can only affect the outcome of a single compliance check d. For most edges, the distributive function will be the identity function, returning the same output as the given input. If we assume that all primitive operations have the same cost, the tabulation algorithm will have the runtime of $O(ED)$ where E is the number of edges and D the number of compliance checks plus the **0**-node.

Replay of Process Variants onto Exploded Supergraph

Now, we have presented how the taint flow analysis is constructed and how the analysis can be efficiently calculated using the tabulation algorithm. The last step is to replay the process variants that occurred in the event log onto the taint flow analysis result. As described earlier, the constructed process graph may allow the production of activity sequences which have not been seen in the event log, thus we would report compliance violations that have never been seen in the event log. To overcome this issue, we replay each process variant onto the taint flow result and check if there is a realisable-path from a producer to a sensitive consumer - a taint flow. Because all possible realisable-paths have already been calculated during the taint flow analysis, the replay of the process variants can be achieved very efficiently.

The replay of the process variants works as follows: for each process variant

$$v_i = \{n_0, n_1, \ldots, n_{n-1}, n_n\} = \{n_0, n_1, \ldots, n_i, n_{i+1}, \ldots, n_{j-1}, n_j, \ldots, n_n\}$$

and for each compliance check $d \in D$ we search for producer and consumer activities $n_i \in producers(d)$ and $n_j \in consumers(d)$ in the activity sequence. If we found a producer and a consumer activity, we look if each node $n_{i+1}, \ldots, n_{j-1}, n_j$ lies on a taint flow that corresponds to the compliance check d:

$$\forall\, n \in \{n_{i+1}, \ldots, n_{j-1}, n_j\} \;:\; d \in MVP_n$$

If all nodes lie on a taint flow, the compliance check d will fail for the reviewed process variant. Otherwise v_i will pass the compliance check.

The result of our approach is a list of process variants for which the compliance checks fail.

Case Study

To prove that our approach can be used to analyse compliance of real event logs, we implemented our approach as a plug-in for the ProM 6 framework (van Dongen et al. 2005), which is an open-source framework for implementing process mining techniques. We evaluated against three different event logs and measured the number of compliance violations as well as the overall computation time. Our goal is to show that our taint flow analysis approach works for real event logs and performs faster than existing approaches. Thus, we compare our approach with the existing LTL checker (Van Der Aalst et al. 2005) and the PetriNet Pattern checker (Ramezani et al. 2012).

Experiment Setup

Event Logs. To show that our approach works for real event logs, we extracted two event logs from SAP ERP R/3 systems, representing the procurement process of goods. Additionally, we used a synthetic event log that was created using PLG2 (Burattin and Sperduti 2011) from a BPMN model. The probabilities for trace missing head, trace missing tail, trace missing episode, perturbed event order, doubled event and alien event were set to 20 percent. In all three event logs the procurement process consists of a shopping cart system (e.g. Amazon-like shopping for catalogue goods; here: SAP SRM), the actual order of goods, the goods receipt, the invoice receipt and the payment. Although we only concentrate on the procurement process of goods in this evaluation, our approach also works for different domains and processes. Event Log A only consists of a small time frame (1 month), whereas Event Log B consists of cases over a time period of about 2 years. The synthetic log C consists of 50 000 automatic generated cases from a manual created BPMN model. Table 1 shows the general characteristics of the event logs.

	Event Log A	**Event Log B**	**Event Log C**
Type	SAP ERP	SAP ERP	Synthetic
Time period	1 month	2 years	-
Cases	873	651 709	50 000
Events	5 077	3 880 138	486 395
Event classes	20	35	15

Table 1. General properties of the event logs used in the evaluation.

Compliance Checks. We created a set of four different compliance checks that refer to the procurement process. The rules were created based on the compliance guidelines that organisations have and wanted to check. The following four compliance checks were created:

1. **Approval of all Purchase Requests**
   Before goods can be ordered, a purchase request must be created for all free text orders. These requests must be approved before the order can be made.
2. **Approval of all Shopping Carts**
   Besides free text orders, employees can order goods via a catalogue system (like Amazon). Ordered shopping carts also need to be approved, although low value items will be approved automatically by the system.
3. **Approval of all Purchase Orders**
   After the purchase request or the shopping cart is finalized, the order for the goods is made. These purchase orders need to be approved by the central procurement department.
4. **Goods must be received before Payment**
   The last compliance rule checks, if goods have been received before the payment is made.

All four compliance checks were checked using the taint flow analysis approach, the LTL Checker and the PetriNet Pattern approach. For the taint flow analysis approach we formulated the compliance checks as defined in definition 4:

$$CC_1 = \left(\overbrace{\{PR\_ITEM\_CREATED\}}^{\text{producer}}, \overbrace{\{PR\_ITEM\_RELEASED\}}^{\text{sanitiser}}, \overbrace{\{PO\_ITEM\_CREATED\}}^{\text{consumer}}\right)$$

This compliance check corresponds to **Approval of all Purchase Requests**. The same check can be formulated as an LTL rule for the ProM LTL checker like the following:

$$[]((task == "PR\_ITEM\_CREATED" \\ \rightarrow (task! = "PO\_ITEM\_CREATED"\ \_U\ task\ == "PR\_ITEM\_RELEASE")))$$

Analogue to the LTL checker, we also created PetriNet Patterns that match the given rule for the PetriNet Pattern checker. The three other rules can be formulated the identical way, but due to space constraints we only give the explicit definition for the first rule.

Hardware Specifications. We perform our benchmark tests on two different machines. Machine A is a laptop with a dual-core Intel Core i5-5200U processor with each core running at 2.2GHz, 12GB memory and a 500GB SSHD hard drive (cf. Table 2). Machine B is a desktop computer with a quad-core Intel Core i7-6700 processor with each core running at 3.4GHz, 16GB memory and 200GB SSD storage. Each machine runs Microsoft Windows 10 with all updates (Version 10.0.10586). Not required background processes are disabled. On both machines the same Java VM 1.8.90 64-bit version is installed. Our written plug-in was tested using ProM 6.5.1 which also includes the version of the LTL checker and PetriNet Pattern plug-in we compared our approach with.

	Machine A Laptop	**Machine B** Desktop
Processor	Dual-core 2.4 GHz Intel Core i5-5200U	Quad-core 3.4 GHz Intel Core i7-6700
Memory	12 GB	16 GB
Storage	500 GB SSHD	200 GB SSD
OS	Windows 10 v10.0.10586	
Java VM	Oracle Java 1.8.90 64-bit	

Table 2. Computer specifications.

Measurement Methodology

In our case study we measured two different types of metrics: the number of violations and the computation time of the approaches.

Number of Violations. We measured the number of cases that violate against each compliance check separately for each event log. Additionally, we stored unique identifiers of each violated case to compare the result of our approach with the LTL checker and the PetriNet Pattern result.

Computation Time. In our approach we measure the computation time starting at the point where the event log is converted into process variants and ending with the replay of the variants onto the exploded supergraph. This includes all the computation time that is required to perform our approach and to retrieve the violated cases. For the LTL checker and the PetriNet pattern approach we only measure the computation time of the real rule checking, excluding the loading of the plugin and the visualisation. For all approaches we do not measure the loading time of the event log nor the creation of visualisation. To get stable results, computation time was measured 6-times for each model in a row, ignoring the first measurement due to Java VMs memory allocation.

Case Study Results

To show that our approach works for real event logs and returns the same violations as an equally configured LTL checker and PetriNet Pattern checker, we conducted a case study with three different event logs. Additionally, we measured the computation time of the taint flow analysis and the LTL checker and compared the results.

Violation Results. We run the taint flow analysis, the LTL checker and the PetriNet Pattern checker with the four compliance checks against both event logs. The cases that were identified as violating have been recorded and compared to the gold standard which has been attached to the cases manually.

Dataset	**Rule**	**Gold Standard**	**Taint Flow**	**LTL Checker**	**PetriNet Pattern**
Event Log A	(1)	65	65	65	65
	(2)	0	0	0	0
	(3)	342	342	342 (346)	342
	(4)	3	3	3 (5)	3
Event Log B	(1)	293 588	293 588	293 588	293 588
	(2)	0	0	0	0
	(3)	323 539	323 539	323 539	323 539
	(4)	2 679	2 679	2 679	2 679
Event Log C	(1)	24	24	24 (54)	24

	(2)	24	24	24 (52)	24
	(3)	52	52	52	52
	(4)	155	155	155	155

Table 3. Number of cases that are not compliant.

Table 3 shows the identified violating cases for each compliance check individually. As we can see, almost all compliance checking approaches deliver the same number of violating cases. The cases identified as violating cases are also identically to the ones in the gold standard. We did not calculate the precision and recall for all methods because all three approaches are (if implemented correctly) mathematically correct. So there is no deviation between the gold standard and the violating cases identified by the evaluated approaches. However, we noticed some additional cases classified as violating using the LTL checker. We found out that for rule (3) there exist 4 cases that do not have a purchase order at all, thus the gold standard and our approach does not consider this as a violation. The defined LTL rule, however, does fail because the logical formula is not fulfilled. Same difference goes for rule (4): there exist 2 cases in which no payment was performed.

Benchmark Results. We also compared the taint flow analysis, the LTL checker and the PetriNet pattern approach regarding the performance perspective on two computers with different specifications. Figure 5 and Figure 6 show the computation time for checking the four compliance rules onto the three event logs. Our approach outperforms the LTL checker and the PetriNet pattern approach for all three event logs. We noticed that the computation time of the LTL checker and the PetriNet pattern approach is significantly slower than our approach for the large event log B. The reason for this is that both other approaches check each process case separately while our approach checks each process variant. Because of this behaviour and the fact that the LTL checker and the PetriNet pattern approach could also operate on process variants, we reduced the size of the event log B to just the process variants. This reduction leads to a faster processing for both approaches. Yet, our approach is still faster.

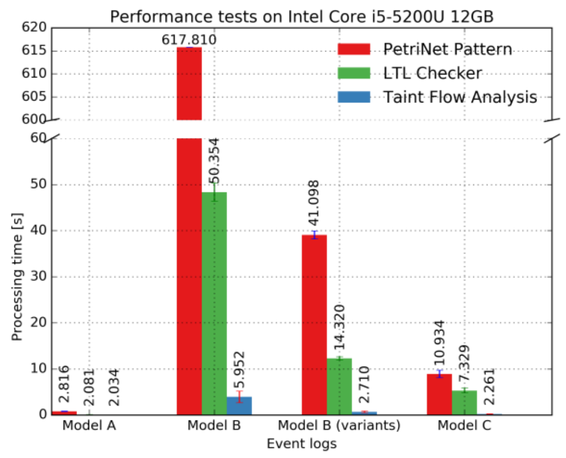


Figure 5. Benchmark results on machine A.

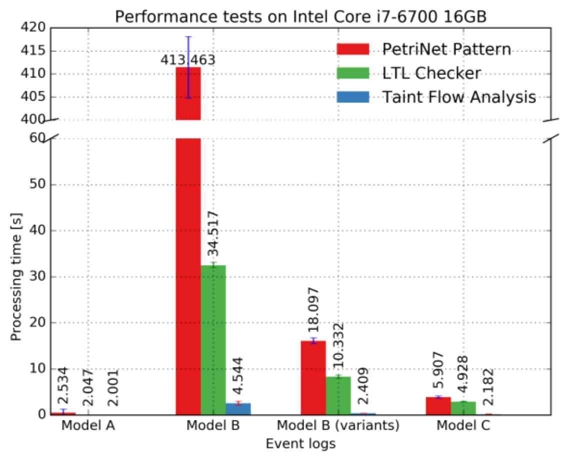


Figure 6. Benchmark results on machine B.

If we look deeper into the benchmark results, we see that our approach is, on average, up to 12.2-times faster than the LTL checker for the large event log and 5.7-times faster for the small one respectively (see Table 4). For the synthetic event log, our approach is 18.6-times faster than the LTL checker. Our approach is even faster when compared to the PetriNet pattern approach which is also slower than the LTL checker. We think that adjusting each trace in the event log to fit the created patterns dues a lot of time. We conclude that especially for larger event logs our approach performs significantly faster. In real world scenarios the complexity of the process graph does not increase much with the number of traces because the majority of traces will follow a certain process variant. Because our approach operates on top of the process graph and the tabulation algorithm has a linear computation time complexity which depends on the number of edges and nodes, we can perform faster control flow compliance checks.

	Event Log A		**Event Log B**		**Event Log B (only variants)**		**Event Log C**	
	i5 6 GB	**i7** 12 GB	**i5** 6 GB	**i7** 12 GB	**i5** 6 GB	**i7** 12 GB	**i5** 6 GB	**i7** 12 GB
PetriNet Pattern	816.4	173.6	615 810.0	421 323.0	39 098.2	16 097.4	8 933.6	3 906.6
LTL	81.4	47.4	48 353.8	32 516.8	12 319.8	8 331.6	5 328.6	2 927.6
Taint Flow	**21.6**	**1.0**	**4 087.2**	**2 544.2**	**910.6**	**408.6**	**261.4**	**182.2**

Table 4. Performance benchmark (in milliseconds) between Taint Flow Analysis, LTL checker and PetriNet Pattern.

When we reduce the amount of data in event log B so that the event log only contains a single case for each process variant, both approaches perform faster. For the used event log B, reducing the size to process variants produces an event log with 30 556 cases with 523 577 events. Our approach is still faster than the LTL checker and the PetriNet pattern approach (see Table 4), although the taint flow approach needs to calculate the process graph using the process variants, constructs the exploded supergraph and replays the variants on the graph.

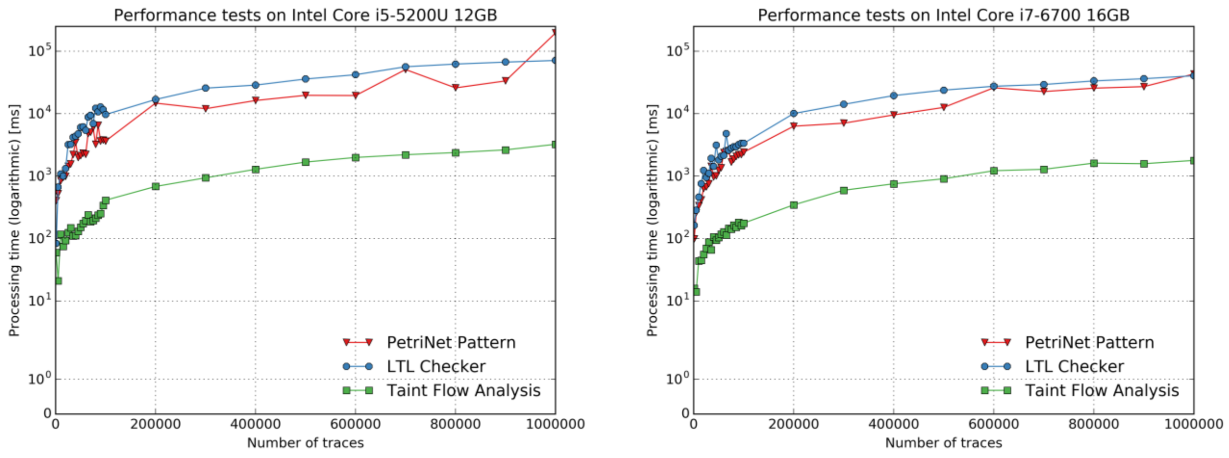


Figure 7. Calculation time for a single compliance check for different sizes of event logs.

Besides measuring the performance of the three selected event logs, we also generated synthetic event logs of different sizes from the same process model used to generate event log C. In Figure 7 the processing time for each of the event logs is depicted. We measured the processing time to perform a single compliance check. While the processing time for the PetriNet pattern approach and the LTL checker are almost identically, our taint flow analysis approach performs faster. Even for extremely large event logs (e.g. 1 million traces) our three step approach outperforms the other two methods.

Discussion and Future Work

In this section, we discuss our taint flow analysis approach that detects compliance violations in event logs by formulating compliance checking as a graph-reachability problem.

Limitation to Existence and Order of Activities

We have shown that our approach performs well for real event logs. Its computation time is significantly faster than the other two evaluated approaches. In the current implementation we only consider checking compliance regarding the existence and order of activities. However, compliance checking also includes other perspective such as organisational process perspective (e.g. checking separation of duty or who performs which activity) or informational process perspective (e.g. objects that are modified during the execution of the process). Supporting such types of checks may also be achieved using our approach by replacing the distributive function. Instead of creating or destroying sensitive flows based on the executed activity, the distributive function may use other case properties (e.g. resources) to decide if a sensitive flow is created, forwarded or destroyed. We can also imagine that the exploded supergraph may need to be extended, such that it stores temporal information on the edges to calculate the outcome of the distributive function.

Compliance Checking on Process Models

Because our approach works on top of process graphs that are constructed from process variants, the graph allows the production of activity sequences that are not existent in the event log. To overcome this issue, we replay each process variant onto the exploded supergraph to identify the violating process cases. Another possible solution for this issue is the use of process models instead of process graphs. Process mining discovery algorithms like the α-miner (Van Der Aalst et al. 2004), heuristics miner (Weijters et al. 2006) or the fuzzy-miner (Günther and van der Aalst 2007) deliver a good abstraction of the event log and provide a more expressive model than the process graph. However, using a process model requires some adjustments to the overall approach: models may consist of concurrent activities. Another issue is the conversion of such a model representation into a flow graph (e.g. challenge to transform non-free-choice nets into flow graphs) (Favre et al. 2015).

Conclusion

In this paper, we proposed the usage of the existing taint flow analysis concept to identify compliance violations in processes. Our main contribution is the transformation of the original taint flow analysis algorithm to the context of compliance checking in processes. The approach works on top of process graphs which can be generated from historic event logs extracted from PAISs, allowing to perform backward compliance checking. We have shown that compliance checking can be formulated as a graph-reachability problem that can efficiently be solved using the tabulation algorithm. To define a compliance check, only three sets of activities (producers, sanitisers and consumers) need to be specified. This information is sufficient to specify the order of activities which the process needs to follow.

Testing our approach regarding applicability and performance, we conducted a user study with two different-sized event logs from two organisations. Both event logs reproduce the real procurement process of goods in two different organisations. As a result, we can show that our approach works for real event logs and detects the exact same violated cases as the LTL checker and the PetriNet pattern approach. The overall computation time for the taint flow analysis approach is significantly faster than for the LTL checker and the PetriNet pattern method, even if we reduce the event log to process variants.

Although our current implementation has some limitations, the idea of using the taint flow analysis concept to check compliance in processes looks very promising. The user study has shown that the taint flow analysis works with real event logs and performs significantly faster than the two evaluated existing methods in our scenario.

Acknowledgement

This project (HA project no. 479/15-21) is funded in the framework of Hessen ModellProjekte, financed with funds of LOEWE – Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz, Förderlinie 3: KMU-Verbundvorhaben (State Offensive for the Development of Scientific and Economic Excellence) and by the LOEWE initiative (Hessen, Germany) within the NICER project [III L 5-518/81.004].

References

Abdullah, N. S., Sadiq, S., and Indulska, M. 2010. "Information Systems Research: Aligning to Industry Challenges in Management of Regulatory Compliance," *14th Pacific Asia Conference on Information Systems* (14:August 2016), pp. 546–557.

Awad, A., Decker, G., and Weske, M. 2008. "Efficient Compliance Checking Using Bpmn-Q and Temporal Logic," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (5240 LNCS), pp. 326–341.

Awad, A., and Weske, M. 2009. "Visualization of Compliance Violation in Business Process Models," *5th Workshop on Business Process Intelligence BPI* (9), pp. 182–193.

Baier, T., Mendling, J., and Weske, M. 2014. "Bridging Abstraction Layers in Process Mining," *Information Systems* (46:September), pp. 123–139.

Burattin, A., and Sperduti, A. 2011. "Plg: A Framework for the Generation of Business Process Models and Their Execution Logs," in *Lecture Notes in Business Information Processing*. pp. 214–219.

Caron, F., Vanthienen, J., and Baesens, B. 2013. "Comprehensive Rule-Based Compliance Checking and Risk Management with Process Mining," *Decision Support Systems* (54:3), pp. 1357–1369.

Cook, J. E., and Wolf, A. L. 1999. "Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model," *ACM Transactions on Software Engineering and Methodology* (8:2), pp. 147–176.

Curtis, B., Kellner, M. I., and Over, J. 1992. "Process Modeling," *Communications of the Acm* (35:9), pp. 75-90.

Favre, C., Fahland, D., and Volzer, H. 2015. "The Relationship between Workflow Graphs and Free-Choice Workflow Nets," *Information Systems* (47), pp. 197-219.

Fellmann, M., and Zasada, A. 2014. "State-of-the-Art of Business Process Compliance Approaches: A Survey," *Proceedings of the 22th European Conference on Information Systems*), pp. 1–17.

Gerke, K., Cardoso, J., and Claus, A. 2009. "Measuring the Compliance of Processes with Reference Models," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (5870 LNCS:PART 1), pp. 76–93.

Ghose, A., and Koliadis, G. 2007. "Auditing Business Process Compliance," *Proceedings of the International Conference on Service-Oriented Computing (ICSOC-2007)*), pp. 169–180.

Governatori, G., and Rotolo, A. 2008. "An Algorithm for Business Process Compliance," *Frontiers in Artificial Intelligence and Applications* (189:1), pp. 186–191.

Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. 2011. "Saving the World Wide Web from Vulnerable Javascript," *Proceedings of the 2011 International Symposium on Software Testing and Analysis*), pp. 177–187.

Günther, C. W., and van der Aalst, W. M. P. 2007. "Fuzzy Mining – Adaptive Process Simplification Based on Multi-Perspective Metrics," *Communications of the ACM* (4714), pp. 328–343.

Leoni, M. d., Maggi, F. M., and van der Aalst, W. M. P. 2012. "Aligning Event Logs and Declarative Process Models for Conformance Checking," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (7481 LNCS:257593), pp. 82–97.

Ly, L. T., Rinderle-Ma, S., and Dadam, P. 2010. "Design and Verification of Instantiable Compliance Rule Graphs in Process-Aware Information Systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (6051 LNCS), pp. 9–23.

Muñoz-Gama, J., Carmona, J., Hull, R., Mendling, J., and Tai, S. 2010. "A Fresh Look at Precision in Process Conformance," (6336), pp. 211–226.

Ramezani, E., Fahland, D., and van der Aalst, W. M. P. 2012. "Where Did I Misbehave? Diagnostic Information in Compliance Checking," in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. pp. 262–278.

Reps, T., Horwitz, S., and Sagiv, M. 1995. "Precise Interprocedural Dataflow Analysis Via Graph Reachability," in *Proceedings of the 22nd Acm Sigplan-Sigact Symposium on Principles of Programming Languages - Popl '95*. New York, New York, USA: ACM Press, pp. 49–61.

Sadiq, S., Governatori, G., and Namiri, K. 2007. "Modeling Control Objectives for Business Process Compliance," in *Business Process Management*. Springer, pp. 149–164.

Sarbanes, P., and Oxley, M. 2002. "Sarbanes-Oxley Act of 2002."

Schönig, S. 2015. "Sql Queries for Declarative Process Mining on Event Logs of Relational Databases," (612052).

Van Der Aalst, W. M. P., Adriansyah, A., and van Dongen, B. 2012. "Replaying History on Process Models for Conformance Checking and Performance Analysis," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (2:2), pp. 182–192.

Van Der Aalst, W. M. P., Beer, H. T. d., and van Dongen, B. F. 2005. "On the Move to Meaningful Internet Systems 2005: Coopis, Doa, and Odbase," in: *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 130-147.

Van Der Aalst, W. M. P., Weijters, T., and Maruster, L. 2004. "Workflow Mining: Discovering Process Models from Event Logs," *IEEE Transactions on Knowledge and Data Engineering* (16:9), pp. 1128–1142.

van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H. M. W., Weijters, A., and Van Der Aalst, W. M. 2005. "The Prom Framework: A New Era in Process Mining Tool Support," in *Applications and Theory of Petri Nets 2005*. Springer, pp. 444–454.

Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., and Weske, M. 2011. "Process Compliance Analysis Based on Behavioural Profiles," *Information Systems* (36:7), pp. 1009–1025.

Weijters, a. J. M. M., Van Der Aalst, W. M. P., and Medeiros, a. K. A. D. 2006. "Process Mining with the Heuristicsminer Algorithm," *Cirp Annals-manufacturing Technology* (166), pp. 1–34.