Information Systems 37 (2012) 714–736

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

An iterative approach to synthesize business process templates from compliance rules

Ahmed Awad [a], Rajeev Goré [c], Zhe Hou [c], James Thomson [c], Matthias Weidlich [b,*]

[a] Faculty of Computers and Information, Cairo University, Egypt
[b] Hasso Plattner Institute, University of Potsdam, Germany
[c] School of Computer Science, The Australian National University, Australia

ARTICLE INFO

Available online 11 May 2012

Keywords:
Process synthesis
Analysis of business process compliance specification
Process mining

ABSTRACT

Companies have to adhere to compliance requirements. The compliance analysis of business operations is typically a joint effort of business experts and compliance experts. Those experts need to create a common understanding of business processes to effectively conduct compliance management. In this paper, we present a technique that aims at supporting this process. We argue that process templates generated out of compliance requirements provide a basis for negotiation among business and compliance experts. We introduce a semi-automated and iterative approach to the synthesis of such process templates from compliance requirements expressed in Linear Temporal Logic (LTL). We show how generic constraints related to business process execution are incorporated and present criteria that point at underspecification. Further, we outline how such underspecification may be resolved to iteratively build up a complete specification. For the synthesis, we leverage existing work on process mining and process restructuring. However, our approach is not limited to the control-flow perspective, but also considers direct and indirect data-flow dependencies. Finally, we elaborate on the application of the derived process templates and present an implementation of our approach.

© 2012 Elsevier Ltd. All rights reserved.

1. Introduction

Compliance management has received increasing attention in recent years. Numerous financial scandals in large companies have fostered increasing attention on compliance management and has led to legislation initiatives such as SOX [1]. When aiming to control business operations, compliance checking focuses on many different aspects of a business process, for example compliance of business requirements may restrict the order in which activities are executed. Often constraints on the execution of activities are also based on the data context, and dedicated data values may require the execution, or the absence of any execution, of an activity. There may even be restrictions on role resolution to realize a separation of duty.

Driven by these trends, numerous approaches have been presented to address compliance management of business processes, and they can be classified as follows. First, compliance rules may guide the design of a business process [2,3] so that compliance is ensured *by design* since compliance violations are identified in the course of process model creation. Second, existing process models are *verified* against compliance rules [4,5]. Given compliance requirements and a process model as input, these approaches identify violations on the process model level.

Clearly, addressing compliance during the design of business operations has many advantages. Non-compliant processes are prevented at an early stage of process

* Corresponding author. Tel.: +49 331 5509180.
E-mail addresses: a.gaafar@fci-cu.edu.eg (A. Awad), Rajeev.Gore@anu.edu.au (R. Goré), zhe.hou@anu.edu.au (Z. Hou), jimmy.thomson@anu.edu.au (J. Thomson), matthias.weidlich@hpi.uni-potsdam.de (M. Weidlich).

0306-4379/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2012.05.001

implementation and costly post-implementation compliance verification along with root cause analysis of non-compliance is not needed. In most cases, process models that are synthesized from compliance rules cannot be directly used for implementing a business process. Instead, they should be seen as a blueprint that is used as a basis for negotiation between business and compliance experts. Hence, we refer to these process models as *process templates* in order to emphasize that further refinements are needed to actually implement the business process [6]. While this approach has been advocated by other authors, e.g., [2,7–9], existing approaches are limited when it comes to data-dependent compliance requirements.

In this paper, we build on our previous work [10] and present an iterative approach to the synthesis of compliant process templates. We start with a set of compliance rules expressed in Linear Temporal Logic (LTL). Hence, we do not require the definition of explicit points in time as in [2,7], but focus on relative execution order dependencies. Further, we also consider data flow dependencies between activity executions. This is neglected in [8], whereas other work requires the pattern-based coupling of control flow routing and data conditions [9]. These rules are then enriched with general constraints related to business process execution. To reach the ultimate goal of generating a process template and hence to have a common understanding of the compliance requirements, we go through a set of steps that might be repeated several times. First, we check whether the compliance requirements are satisfiable. If the compliance requirements are not satisfiable, we analyze the reasons for this inconsistency. If the requirements are satisfiable, we generate all possible execution sequences (traces) and check them for underspecification. If requirements are well-specified, a process template is generated from these traces. At the time a specific step fails, e.g., the requirements are inconsistent, our approach suggests some remedies, the requirements are updated by the user and a new iteration starts. Finally, we also illustrate how generated templates are applied during process design and how the template generation may identify inconsistencies and open questions. Hence, the template guides further refinements of the process model and the compliance requirements. To evaluate the applicability of our approach, we present a prototypical implementation.

This paper revises and extends our initial approach presented in [10] in various directions. In this paper, we show how analysis is conducted if the compliance rules are inconsistent, an aspect neglected in our previous work. Further, we now use a theorem proving approach to analyze the generated traces and extend the trace criteria to be verified. That is, we explicitly verify correctness of the traces regarding the order of execution as induced by data dependencies. In addition, our new approach comes with resolution strategies when the trace correctness criteria are not satisfied. Finally, we present a novel approach for the actual synthesis, which incorporates techniques from the field of process restructuring. Compared to the synthesis presented in [10], it has the advantage that we synthesize well-structured process models and that we take indirect data dependencies (between activities that do not follow each other directly) into account. As such, our contribution is a comprehensive approach to process design grounded in compliance rules.

The remainder of this paper is structured as follows. The next section introduces preliminaries for our work, such as the applied formalism. Section 3 gives an overview of our approach of synthesizing process templates from a given set of compliance rules and discusses the LTL encoding. Section 4 elaborates on how to detect inconsistencies if the obtained LTL specification is not satisfiable. The generation of traces along with their analysis is presented in Section 5. Section 6 shows how to cope with issues that are detected during the analysis of traces. Then, we discuss the actual synthesis of a process template and its application in Section 7. A prototypical implementation of our approach is presented in Section 8. Finally, Section 10 reviews related work, before we conclude in Section 11.

2. Preliminaries

This section gives preliminaries for our work. Section 2.1 clarifies our notion of execution semantics. Section 2.2 discusses the reason we choose LTL instead of other logics. Section 2.3 presents LTL as the logic used in this paper. Section 2.4 summarizes existing work on generating a behavioral model from a given set of LTL formulae.

2.1. Process runs as linear sequences

In this paper, we rely on trace semantics for process models. An execution sequence σ of a process model is referred to as a *process run* or *trace* – a finite and discrete linear sequence of states $\sigma : s_0,s_1,\ldots,s_n$ with a start state s_0 and an end state s_n. Clearly, a process model, as well as a set of compliance requirements, might allow for many conforming traces, e.g., due to choice points in the process model. Each state of a trace is labeled with propositions that refer to *actions* and *results*. Actions are the driving force of a trace and refer to the execution of business activities. This, in turn, may effect or be constrained by results, which relate to data values of the business process. As an example, think of an activity 'risk analysis' (*ra*) and a data object 'risk'. The action that represents the execution of this activity may have the result of setting the value of the data object to 'high' or 'low'. The execution of another activity, i.e., another action, may be allowed to happen solely if a certain result, e.g., the object has been set to 'high', occurred. Both actions and results are represented by Boolean propositions at each state. For instance, proposition *ra* being 'true' at a state s_i means that the action, i.e., execution of activity 'risk analysis', happened at state s_i. In contrast, proposition *ra* being 'false' at state s_i means that the action did not happen at state s_i. Given a trace $\sigma : s_0,s_1,\ldots,s_n$, we write $p \in s_i$ to indicate that proposition p is true in state s_i, for $0 \leq i \leq n$ and $p \in \sigma$ if there is a state s_i in σ where $p \in s_i$, for some $0 \leq i \leq n$.

We represent an execution sequence as a linear sequence of states where states are labelled with both actions and results, and (unlabelled) edges between states represent the temporal ordering in the sequence.

2.2. A suitable logic for process modelling

Various logics have been proposed for the purpose of process modelling. Deontic modalities were used to define policies and business rules by Padmanabhan et al. [11], similarly, Geodertier et al. [2,7] adopted PENELOPE, a deontic logic with temporal assignments, to enact control-flow-based processes. van der Aalst et al. [12–14], on the other hand, developed a declarative workflow management system that uses Linear Temporal Logic (LTL) [15] to drive the design and execution of processes.

The study of the logic for process modelling is still ongoing, we do not have a conclusion that one is definitely better than the other. Following our intuition to capture process runs, however, LTL seems to be a suitable logic to reason about relations of actions and results in a sequence of executions. First of all, with the temporal operators, LTL brings about the power to specify the uncertain order relation between actions. This advantage, in van der Aalst et al.'s approach, is exploited to declaratively enact flexible processes, and thus can be considered as an important future extension to our approach. Conversely PENELOPE enforces exact time for events with its temporal assignments, and consequently fixes the structure of the process. Second, many workflow management systems record process executions as "event-logs", which contain the ordering of events in each process run. Recorded "event-logs" can be used for process mining, and by adopting machine learning techniques, van der Aalst et al. have shown that it helps support the design of processes by rediscovering patterns from past experience. To this end, choosing LTL is plausible since it naturally captures the ordering of events in linear time. Third, by LTL theorem proving, we will show in Section 2.4 that we can derive all possible traces from the rules in an LTL specification. This means we guarantee that our process model generated from the rules are logically correct and contains all permitted executions described by the rule with all invalid executions excluded. Finally, there are many versions of Deontic Logics that enables different sets of axioms [16], it is nontrivial to decide which one is the best in the process modelling context. There may be missing axioms that could be useful, or present ones that could be harmful. Therefore, for simplicity, LTL enables us to encode and reason about processes under our setting, and also provides us with possibilities to extend our work in related areas in the future.

Before we had LTL as our underlying logic set in stone, we considered other logics such as Computation Tree Logic (CTL) and Propositional Dynamic Logic (PDL). The expressive power of CTL and LTL is incomparable, there are sentences that can be expressed in one but not the other. Nevertheless, since the domain knowledge and rules we define for the process are actually constraints on all traces, we can just add the universal path quantifier in the front of every state quantifier to form the corresponding CTL encoding. In this way, however, the full expressive power of CTL is unexploited. Even though CTL is branching, it is hard to utilise the existential path quantifier to merge all possible paths in one model. That is, to extract all correct traces, we still have to consider all models in CTL and the search space is not reduced. Thus CTL does not benefit us in an obvious way. PDL, as a modal logic, has similar semantics as deontic logic, which in a sense, enables us to describe what must occur and what may occur. However, eventualities in temporal logics, which is the key to flexible structured processes, are hard to simulate in PDL. Although PDL naturally supports distinguishing control-flow from data-flow (actions as programmes and results as propositions), it does not help to express the temporal relations that we want to encode.

As a result, we choose LTL as the suitable logic for our approach, the details of the encoding and other techniques relating to LTL will be presented in the rest of this paper.

2.3. Linear temporal logic

Linear Temporal Logic is a logic specifically designed for expressing and reasoning about properties of linear sequences of states. The formulae of LTL are built from atomic propositions using the connectives of $\vee$ (or), $\wedge$ (and), $\neg$ (not) and $\Rightarrow$ (implication), and the following temporal connectives: **X** (next), **F** (eventually), **G** (always), **U** (until) and **B** (before). There are two logical constants $\top$ (verum) and $\bot$ (falsum) which are always true and always false respectively. The temporal connectives are interpreted as follows:

- $\mathbf{X}\varphi$: in the neXt state, φ holds
- $\mathbf{F}\varphi$: there is some state either now or in the Future where φ holds
- $\mathbf{G}\varphi$: in every state Globally from now on, φ holds
- $\varphi\mathbf{U}\psi$: there is some state, either now or in the future, where ψ holds, and φ holds in every state from now Until that state
- $\varphi\mathbf{B}\psi$: Before ψ holds, if it ever does, φ must hold.

Note that $\mathbf{F}\varphi$ can be defined as $\top\mathbf{U}\varphi$ and similarly $\mathbf{G}\varphi$ can be defined as $\bot\,\mathbf{B}\neg\varphi$. We apply LTL to encode compliance requirements. Hence, we obtain a set Γ of LTL formulae expressing the constraints to which compliant traces have to conform.

2.4. Finding all LTL-models of a set of LTL formulae

Given a collection of compliance requirements (constraints) expressed as a set Γ of LTL-formulae, we try to find a behavioral model that captures all *formula-models*, i.e., traces in our setting, which satisfy Γ. That is, such a structure describes all linear sequences of states $s_0,s_1,\ldots,s_n$ such that Γ is true at s_0. Since Γ may contain *eventualities*, such as $\mathbf{F}\varphi$ or $\psi_1\mathbf{U}\psi_2$, ensuring that Γ is true at s_0 may require us to ensure that φ or ψ_2 is true eventually at some state s_i with $0\le i\le n$. In contrast to model checking [17] we are not given a single trace, but construct all traces satisfying the given constraints.

The first step is to determine whether the constraints Γ are satisfiable. If not, the specification is erroneous since no trace can conform to the given constraints. The second step is the creation of the behavioral model that describes *all* traces.

For both steps, we use a graph-based tableaux method introduced in [18,19]. In essence, this approach works as follows. We start by creating a root node containing Γ and proceed in two phases. First, a finite (cyclic) graph of tableau nodes is created by applying tableau-expansion rules that capture the semantics of LTL and by pruning nodes containing local contradictions [18]. Second, once the graph is complete, a reachability algorithm is used to determine which nodes do not satisfy their eventualities. These nodes are removed and the reachability algorithm is reapplied until no nodes may be removed. The set of formulae Γ is satisfiable, if and only if the root node has not been removed [18]. Further, the graph created by the tableau algorithm, referred to as the *pseudomodel*, describes all possible formula-models, i.e., possible traces [18]. We use this pseudomodel to extract possible traces during our synthesis approach.

In this paper, two types of tableaux methods are used for different purposes. One is the graph-based method mentioned above, which is used for extracting traces (cf. Section 5.1). The other one is the tree-based method with back-jumping, which is used for analysing the cause of inconsistency (cf. Section 4.2). Deciding satisfiability in LTL is PSPACE-complete [20], and the tableaux methods we use are exponential in time. It is also commonly acknowledged that the time complexity of the model checking problem in LTL is exponential in the size of the formula as well [21], so theoretically other methods based on LTL model checking do not have an advantage over our theorem proving method in terms of run time.

3. The basis of the synthesis approach

Section 3.1 gives an overview of the synthesis of process models from a set of compliance rules and introduces an example set of compliance rules used to illustrate all subsequent steps. Section 3.2 describes the LTL encoding of the compliance rules and additional domain knowledge.

3.1. Overview

The process model in Fig. 1 visualizes the steps to synthesize a process template out of a set of compliance rules. First, a set of compliance rules is collected and formulated in LTL, we refer to the set of rules as $\mathcal{CR}$, step **A**. In order to identify whether these requirements are consistent and thus a process template can be synthesized, related domain-specific knowledge is identified, we refer to domain knowledge as $\mathcal{DK}$, step **B**. In Section 3.2 we give details on the LTL encoding of both compliance rules and domain knowledge.

For the conjunction Γ of the LTL formulae in $\mathcal{CR}$ and $\mathcal{DK}$, we verify satisfiability as summarized in Section 2.4, step **C**. If Γ is not satisfiable then no trace can be constructed to satisfy the given LTL formulae. At that point, inconsistency analysis is conducted, step **D**, to identify the causes of inconsistency, details of this step are given in Section 4. Note that all of the three substeps introduced in Section 4.4 in the inconsistency analysis (checking domain knowledge, rules, and the conjunction of them resp.) are conducted in step **D**. That is, in step **C** the aim is to generate a pseudomodel out of the specification. But if the conjunction of domain knowledge and rules is not satisfiable, there can be no pseudomodel, thus checking the satisfiability in step **C** is not for inconsistency analysis but for deciding whether the process should go to step **D** or step **E**. As discussed in Section 2.4, since the purpose of the satisfiability checking in step **C** and step **D** is different, we use a graph-based theorem prover in the former and a tree-based theorem prover in the latter.

On the other hand, If Γ is satisfiable then the satisfiability checker automatically returns the pseudomodel which is a behavioral model of all traces that obey the given constraints. As a next step, finite traces are extracted from the pseudomodel by following all choice points and stopping when a trace becomes cyclic, step **E**. Having a finite set of traces that satisfy the compliance rules, we check them for some correctness criteria, step **F**. Failure of these criteria hints at issues in the specification, so that a new iteration of the synthesis may be started with refined compliance rules or adapted domain knowledge. Steps **E** and **F** are discussed in Section 5. In steps **G** and **H**, we check whether it is possible to restore the correctness criteria of the traces. Both steps are described in Section 6.

If the traces obey the correctness criteria, we use a process synthesis algorithm to extract a process template,

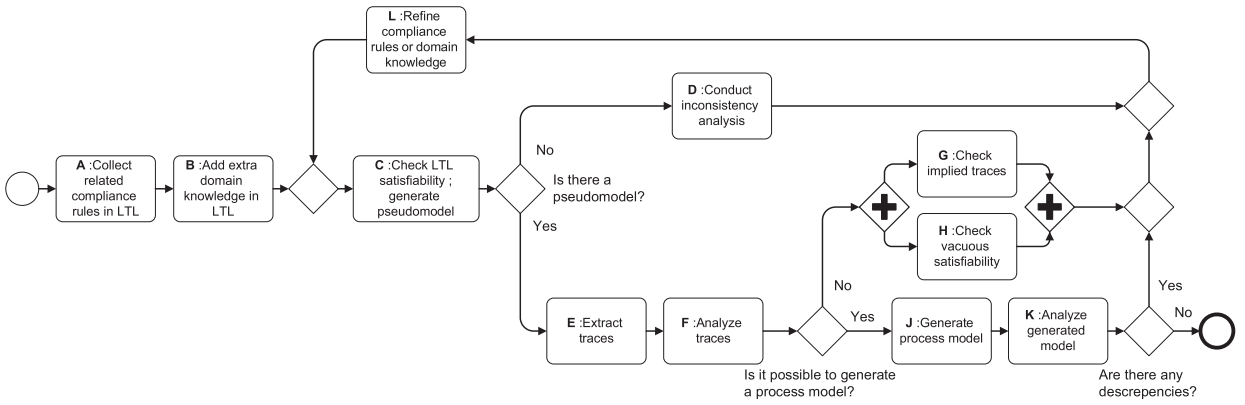

Fig. 1. Process synthesis approach.

step **J**. The synthesized template is then analyzed to identify discrepancies that stem, e.g., from under-specification, step **K**, which is basically a manual step handled by a human expert. Depending on the result of this analysis, again, a new iteration of the synthesis may be started. Section 7 describes the template generation step.

Example. We illustrate our approach with an example from the financial domain. Anti money laundering guidelines [22] address financial institutes like banks, and define a set of checks to prevent money transfers with the purpose of financing criminal actions. We focus on the following guidelines for opening new bank accounts:

*R*1: A risk assessment has to be conducted for each 'open account' request.
*R*2: A due diligence evaluation has to be conducted for each 'open account' request.
*R*3: Before opening an account the risk associated with that account must be low. Otherwise, the account is not opened.
*R*4: If due diligence evaluation fails, the client has to be added to the bank's black list.

3.2. LTL encoding

Once the compliance rules have been collected, a behavioral model that represents all traces conforming to these rules is created. In order to arrive at such a model, we need to collect extra domain-specific rules. Much of the domain-specific rules can be generated automatically from a higher level description. Such a description needs to be defined by a human expert in the first place and comprises the following information where 2^X is the set of all subsets of X.

Actions and Goals: The set of all *actions* is denoted by A. The set of *goal* actions $G \subset A$ comprises actions that indicate the completion of a trace. Moreover, we capture contradicting actions that are not allowed to occur together in one trace in a relation $CA : A \rightarrow 2^A$.

Results and Initial Values: The set of all *results* is denoted by R, and we define $\overline{R} = \{\neg r | r \in R\}$ as the set of negated results. The initial values of data objects are defined by a set $IV \subset R \cup \overline{R}$. Similar to contradicting actions, we capture contradicting results in a relation $CR : R \rightarrow 2^R$.

Relation between Actions and Results: The mapping from actions to sets of results is given as a relation $AM : A \times 2^{R \cup \overline{R}}$. To reflect the fact that results are caused by actions, we define results according to their corresponding actions. That is, for some $a \in A$ that has results, we define R_a as the set of results of action a, thus $R_a = \{r \in R : \exists S.(a,S) \in AM \ \wedge \ r \in S \text{ or } \neg r \in S\}$. It is easy to see that R is the union of the R_as for the actions a that have results.

Exclusive Results Set: Mutually exclusive sets of results RE is defined as $RE \subseteq 2^R$ which satisfies $\forall E \in RE, \forall r_1, r_2 \in E \ s.t. \ r_1 \neq r_2$, r_1 and r_2 cannot hold jointly in one state.

Based on this information and two additional actions *start* and *end* that represent the initial and final states of a trace (independent of any goal states), we derive LTL rules to represent the domain knowledge according to Table 1. Common process description languages, e.g., BPMN or EPCs, assume interleaving semantics, which is enforced by formula *interleave* and *progress*. The information on exclusiveness constraints and on contradicting actions and results yields the formulae *mutex* and *contra*. The formula *causality* guarantees correct implementation of dependencies between actions and results. Finally, the formulae *once*, *final*, *goals*, and *initial* ensure correct initialization and successful termination of any trace. The conjunction of all these formulae yields the formula *domain*, which represents the domain knowledge:

$$\begin{aligned} domain = {} & start \wedge \mathbf{G}\ initial \wedge \mathbf{F}\ goals \wedge \mathbf{F}\ end \\ & \wedge \mathbf{G}\ interleave \wedge \mathbf{G}\ progress \wedge \mathbf{G}\ mutex \\ & \wedge \mathbf{G}\ causality \wedge \mathbf{G}\ once \wedge \mathbf{G}\ contra \wedge \mathbf{G}\ final \end{aligned}$$

Example. For our example, an expert first identifies the following actions and results.

$Actions = \{ra, edd, og, od, bl\}$
ra: conduct a risk assessment.

Table 1
The formulae making up the domain knowledge.

Constraint description	Formalization
To realize interleaving semantics, the formula *interleave* ensures that at most one action can be true, i.e., one activity can be executed, at any state.	$interleave(a) = a \Rightarrow (\bigwedge_{b \in A \setminus \{a\}} \neg b))$
	$interleave = \bigwedge_{a \in A} interleave(a)$
The formula *progress* guarantees that at least one action occurs at each state.	$progress = \bigvee_{a \in A} a$
The mutual exclusion constraints given in *RE* are enforced by the formula *mutex*, i.e., exclusive results cannot be true at the same time.	$mutex(S) = \bigwedge_{a,b \in S, a \neq b} \neg(a \wedge b)$
	$mutex = \bigwedge_{S \in RE} mutex(S)$
Knowledge on contradicting actions or results is taken into account by the formulae, *con* and *conRes*.	$con(a) = a \Rightarrow \mathbf{G} \bigwedge_{b \in CA(a)} \neg b$
	$conRes(r) = r \Rightarrow \mathbf{G} \bigwedge_{s \in CR(r)} \neg s$
	$contra = \bigwedge_{a \in A} con(a) \wedge \bigwedge_{r \in R} conRes(r)$
To implement the relation between actions and results, formula cau_1 states that for every entry $(a,S) \in AM$ the action a must cause at least one of the results in S. Formula cau_2 states that for every result r, that result can only be changed by one of the actions which can cause it.	$cau_1(a,S) = a \Rightarrow \bigvee_{r \in S} r$
	$cau_2(r) = r \Rightarrow (\mathbf{X} \bigvee_{(a,S) \in AM, \{r, \neg r\} \cap S \neq \emptyset} a) \mathbf{B} \neg r$
	$causality = \bigwedge_{(a,S) \in AM} cau_1(a,S) \wedge \bigwedge_{r \in R \cup \overline{R}} cau_2(r)$
The formula *once* enforces that all actions other than *end* occur at most once, in order to avoid infinite behavior. The formula *final* enforces that *end* persists forever to represent the process end.	$once(a) = a \Rightarrow \mathbf{XG} \neg a$
	$once = \bigwedge_{a \in A \setminus \{end\}} once(a)$
	$final = end \Rightarrow \mathbf{G} end$
The formula *goals* is used to require that eventually the outcome of the process is determined, while *initial* ensures correct initial values for all objects.	$goals = \bigvee_{g \in G} g$
	$initial = start \Rightarrow \bigwedge_{v \in IV} v$

edd: evaluate due-diligence.
og: grant a request to open an account.
od: deny a request to open an account.
bl: blacklist a client.
Results = {*ri*,*rh*,*rl*,*ei*,*ef*,*ep*}
ri: risk assessment is initial.
rh: risk was assessed as high.
rl: risk was assessed as low.
ei: due-diligence evaluation is initial.
ef: due-diligence evaluation failed.
ep: due-diligence evaluation passed.

Note that the results are all descriptive statements, while the actions refer to activities. Moreover, we introduce *positive* representations for the values 'high' and 'low' of the risk object, even though both values are opposites. For example, the risk object has three possible values: high, low, or initial. The same holds true for the due-diligence object.

Based on these actions and results, the compliance rules are encoded in LTL. As a process to open a bank account is considered, the process is assumed to start by receiving such a request. Therefore, rules *R*1 and *R*2, mentioned above, are interpreted as "A risk assessment has to be conducted" and "A due diligence evaluation has to be conducted" respectively. The third rule is interpreted to mean that the risk associated with opening an account must be low *at the time the request is granted*, rather than at some point in the past. Similarly in the case when denying the open request, the risk has to be high. Based on this interpretation the rules are formalized as follows:

*R*1: A risk assessment has to be conducted:
$\mathbf{F}\ ra$
"Eventually *ra* must hold"
*R*2: A due diligence evaluation has to be conducted:
$\mathbf{F}\ edd$
"Eventually *edd* must hold"
*R*3: The risk associated with opening an account must be low when the request is granted:
$\mathbf{G}(og \Rightarrow rl) \wedge \mathbf{G}(od \Rightarrow rh)$
"Always, *og* only if *rl*, and always, *od* only if *rh*"
*R*4: If due diligence evaluation fails, the client has to be added to the bank's black list:
$\mathbf{G}(edd \wedge ef \Rightarrow \mathbf{F}\ bl)$
"Always, *edd* and *ef* imply eventually *bl*"

As a next step, the domain knowledge is defined in more detail. For instance, the action mapping *AM* defines $ra \mapsto \{rh,rl\}$ and $ra \mapsto \{\neg ri\}$. The former says that action *ra* causes the risk object to take a concrete value of 'high' or 'low'. The latter means that *ra* causes the risk to stop being 'initial' by forcing *ri* to not hold. Excluding results are defined, e.g., $\{ri,rl,rh\} \in RE$ states that at most one of the propositions *ri*,*rh*,*rl* can hold at each state. The goal of the process is defined as {*og*,*od*} and the set of initial values {*ri*,*ei*} signifies that initially, both risk and due-diligence objects, are put to an initial, unknown, value. There are also contradicting actions, $\{og \mapsto \{od\}, od \mapsto \{og\}\}$, ensuring that we cannot grant and deny a request within the same trace.

Based on Table 1, this specification is converted into LTL. For example, this yields the formula $progress = ra \vee edd \vee og \vee od \vee bl \vee start \vee end$. The final set of LTL formulae is the union of the *domain* formula and all four formulae representing the compliance rules.

Given a set of LTL formulae, we apply the technique summarized in Section 2.4 to determine whether the constraints are satisfiable. If the constraints are unsatisfiable, this indicates an inconsistent specification. The details of inconsistency detection is covered in Section 4. On the other hand, if the constraints are satisfiable, we can obtain a set of traces that represents how such rules can be fulfilled and these traces are further examined before a process template is generated.

4. Analysis of domain knowledge and rules inconsistency

In this section, we describe our approach to discover the cause of inconsistency in the domain knowledge and compliance rules expressed in temporal logic. First, related techniques and definitions are discussed in Section 4.1. Then we present the details of our approach to find the inconsistent subset of rules in Section 4.2. If the user wants to further know the exact reason of inconsistency, we refine the resulting subset to a minimal unsatisfiable core, as is illustrated in Section 4.3. Finally, we show how the techniques are incorporated in the business process synthesis context in Section 4.4.

4.1. Related methods and definitions

Various approaches attempt to find the explanations for the inconsistency of a set Γ of given formulae [23–26]. Most of them focus on extracting minimal unsatisfiable cores of Γ, since they narrow down the reason that causes inconsistency. Although a vast amount of work has been done to investigate the minimal unsatisfiability problem in propositional logic, the same problem for LTL has not drawn much attention. Hantry and Hacid proposed a conflict-driven tableau depth-first-search for LTL, the complexity of their approach is theoretically EXPTIME [25]. Schuppan demonstrated approaches to compute unsatisfiable cores of LTL formulae from various aspects [24], but there was no requirement to find a minimal one.

There are several different notions of unsatisfiable cores in the literature, here we adopt a series of definitions by Lynce et al. [27], which are appropriate under the context of business process modelling.

Definition 1 (*Unsatisfiable Core*). Given a set Γ of formulae, which is the LTL encoding of domain knowledge $\mathcal{DK}$ and compliance rules $\mathcal{CR}$, a set $\mathcal{UC}$ of formulae is an unsatisfiable core for Γ iff $\mathcal{UC} \subseteq \Gamma$ and $\mathcal{UC}$ is unsatisfiable.

Thus an unsatisfiable core can be any subset of Γ that is unsatisfiable. That is, the largest unsatisfiable core is Γ itself, and in the worst case it can be the minimal unsatisfiable core as well, the definition of which is shown as follows.

Definition 2 (*Minimal Unsatisfiable Core*). An unsatisfiable core $\mathcal{UC}$ for a given set Γ of formulae is a minimal unsatisfiable core iff $\forall \varphi \in \mathcal{UC}.\ \mathcal{UC} \backslash \{\varphi\}$ is satisfiable.

If there are multiple minimal unsatisfiable cores in Γ, a minimum unsatisfiable core is one that has the least cardinality, as below.

Definition 3 (*Minimum Unsatisfiable Core*). A minimal unsatisfiable core $\mathcal{UC}$ for a given set Γ of formulae is a minimum unsatisfiable core iff every unsatisfiable core $\mathcal{UC}'$ of Γ has $|\mathcal{UC}'| \geq |\mathcal{UC}|$.

Note that there could even be multiple minimum unsatisfiable cores for a given set of formulae, if these are of the same size. Sometimes it is useful to find all the minimal unsatisfiable cores to allow the user to select the "best" explanation.

Unfortunately, deciding if a set of LTL formulae is a minimal unsatisfiable core is in PSPACE, and is conjectured to be PSPACE-complete [25], thus it is expensive to pinpoint the exact reason of inconsistency. However, in many cases, it may be important to find (not necessarily minimal) unsatisfiable cores, since they still provide the user with useful information.

4.2. Finding unsatisfiable cores

We now outline our procedure for finding unsatisfiable cores of the given domain knowledge and compliance rules. The technique we use is known in the automated reasoning and artificial intelligence communities as "back-jumping" or "use-check" [28], and is integrated into our tree-based tableaux method.

The tree-based tableaux method for checking satisfiability attempts to build a linear model of nodes for the given set of formulae Γ. Each node in the model contains the set of formulae which are true at that node. It does this by starting with an initial root node that contains Γ as a set of formulae of LTL. Under the assumption that Γ is LTL-satisfiable (i.e., has a model), the tableaux method adds further nodes below the current node which must also be LTL-satisfiable. It adds these nodes by applying a rule from a given finite set of rules to one of the formulae in the node to give a new node containing new formulae.

If the current node contains the formula $\varphi \wedge \psi$, as well as the set of formulae X, then the tableau rule for $(\wedge)$ can be applied to obtain a child node that is identical to the current node, except that $\varphi \wedge \psi$ is replaced by two formulae φ and ψ. The rule can be written as shown below at left:

$$(\wedge)\frac{X;\varphi\wedge\psi}{X;\varphi;\psi} \quad (\vee)\frac{X;\varphi\vee\psi}{X;\varphi\,|\,X;\psi} \quad (\bot)\frac{X;p;\neg p}{}$$

$$(\mathbf{X})\frac{Z;\mathbf{X}\varphi_1,\mathbf{X}\varphi_2;\cdots;\mathbf{X}\varphi_n}{\varphi_1,\varphi_2;\cdots;\varphi_n}$$

Z only contains literals and $(\bot)$ not applicable

The rule $(\vee)$, shown above in the middle splits the current branch into two branches: one where the child node contains φ instead of $\varphi\vee\psi$, and the other which contains ψ instead of $\varphi\vee\psi$. These branches capture the intuition that if $\varphi\vee\psi$ is true, then φ is true or ψ is true.

The (**X**) rule intuitively creates a child node which is a "next" state to the current state. Thus, it has a side-condition which says that it can be applied only when no other rules could apply. The rule $(\bot)$, shown above on the right is a "stopping rule" which says that we can stop expansion of the current branch if we find a node that contains a contradictory pair of atomic formulae. That is, if we find such a pair, then the current node has no children. Such a node is said to be "closed".

We omit the rules for **F**,**G**,**U** and **B** since they are more complicated to explain. However, **F** and **U** behave in essentially the same way as $(\vee)$ for the purposes of backtracking, since $\varphi\mathbf{U}\psi \equiv \psi\vee(\varphi\wedge\mathbf{X}(\varphi\mathbf{U}\psi))$, and **G** and **B** behave similar to $(\wedge)$ because $\varphi\mathbf{B}\psi \equiv \neg\psi\wedge(\varphi\vee\mathbf{X}(\varphi\mathbf{B}\psi))$.

The usual way to explore these branches is to traverse them in a depth-first, left-to-right order. When we apply a $(\wedge)$ or $(\vee)$ rule to a node, we label the node as an $\wedge$-node or $\vee$-node respectively. When we find a closed node, we propagate that status upwards to the parent of the closed node as follows. If the parent is an $\wedge$-node or an **X**-node then it gets the status closed too. Else if the parent is an $\vee$-node it gets the status closed if both of its children are closed.

Suppose we have a node $X;\varphi\vee\psi$ which we split into two child nodes $X;\varphi$ and $X;\psi$. Suppose that the $X;\varphi$ branch closes somewhere below this node because it creates a node $Y;p;\neg p$. We can immediately form the set $X_l=\{p,\neg p\}$ as a "bad" set. When we backtrack up from $Y;p;\neg p$, we can trace the formulae p and $\neg p$ to the formulae which create them and replace each in X_l with its respective traced formula. We repeat this tracing procedure to higher and higher nodes and collect these larger and larger "bad" formulae into a larger and larger set. Eventually, we end up at $X;\varphi$ with some set X_l of "bad" formulae. That is, we know that $X_l\subseteq(X;\varphi)$ is unsatisfiable (i.e., contradictory).

If we have $X_l\subseteq X$ then we can conclude that φ is irrelevant to the contradiction. Hence we can conclude that $X;\psi$ will be contradictory for exactly the same reason: namely the contradictory X_l that is sitting inside X. This means that we do not even have to expand the $X;\psi$ branch, we can just declare it closed and backtrack further up the tableau branch.

On the other hand, if $X_l\not\subseteq X$ then we have to explore the right branch by expanding the right child $X;\psi$. Suppose it gives us an unsatisfiable set $X_r\subseteq(X;\psi)$. If $X_r\subseteq X$ then we can pass up X_r alone, ignoring X_l. That is, this tells us that if we had swapped the order of exploration and had explored $X;\psi$ first, then we would have back-jumped over the $X;\varphi$ child since X contains a contradictory subset X_r.

On the other hand, if $X_r\not\subseteq X$ then we have found two contradictory sets $X_l\subseteq(X;\varphi)$ and $X_r\subseteq(X;\psi)$ such that $X_l\not\subseteq X$ and $X_r\not\subseteq X$. We therefore form the set $X_{lr}=(X_l\backslash\{\varphi\})\cup(X_r\backslash\{\psi\}))\cup\{\varphi\vee\psi\}$ by replacing φ and ψ by $\varphi\vee\psi$ in the union of X_l and X_r. We can guarantee that X_{lr} is unsatisfiable since an application of the $(\vee)$-rule immediately gives us two children $X_l;X_r\backslash\{\psi\}$ and $X_r;X_l\backslash\{\varphi\}$, each of which contains an unsatisfiable subset X_l and X_r respectively.

Notice that there may be a set $W\subset X_{lr}$ which is also unsatisfiable and strictly smaller than X_{lr}, thus we do not guarantee to identify a minimal unsatisfiable core.

Finally, here is what we do at an **X** node. As described above, the child will give a "bad" set $X_b\subseteq\{\varphi_1,\cdots;\varphi_n\}$

which we know to be unsatisfiable. We therefore return the set $\mathbf{X}X_b = \{\mathbf{X}\varphi_i | \varphi_i \in X_b\}$ as an unsatisfiable core. Again, this set may not be minimal. There are also some further complications that arise because of the need to track unfulfilled eventualities. It is impractical to find minimal unsatisfiable cores immediately via back-jumping, so we further trim off the irrelevant formulae from the result here as presented in Section 4.3.

Back-jumping is an optimisation to quickly discard branches that do not lead to satisfiability. It improves the performance of theorem proving in practice, but does not change the worst case time complexity. Comparing to the graph-based method which aims at finding all models of a formula, the tree-based method is faster when the formula is satisfiable because it can stop once it has found only one model.

4.3. Refining the unsatisfiable core

The tree based tableaux method with back-jumping is used in the inconsistency analysis because we can also derive unsatisfiable cores by using this method. If the user can already determine the cause of inconsistency from the result given by back-jumping, then no further refinement is needed. However, sometimes the result returned by back-jumping may be a relatively large subset. In this case, to help the user identify the cause of inconsistency, we adopt a general algorithm to find a minimal unsatisfiable core [29] and simultaneously avoid the complicated internal operations in the tree-based tableaux method.

Two well known algorithms that use the theorem prover as a black box to find minimal unsatisfiable cores are described by Marques-Silva [29]. The deletion-based algorithm calls the theorem prover $O(m)$ times, where m is the size of the set of formulae; the insertion-based algorithm with binary search involves $O(k \times \log m)$ calls to the theorem prover, where k is a number that is much smaller than m. Despite the possible lower complexity of insertion-based algorithms and their promising results in Constraint Satisfaction Problems, they are not widely used because insertion-based methods are not effective in solving minimal unsatisfiable core problems. Moreover, the inconsistent set returned by back-jumping can often be significantly smaller than Γ, we therefore use the deletion-based algorithm.

Given an unsatisfiable core Δ returned by back-jumping, we call the theorem prover iteratively to find a minimal unsatisfiable core as follows. For each $\varphi \in \Delta$, we test if $\Delta \backslash \varphi$ is unsatisfiable. If $\Delta \backslash \varphi$ is still unsatisfiable then φ does not contribute to the inconsistency of this subset, and thus is deleted. Otherwise, we keep φ in Δ and test other formulae. This procedure terminates when all the formulae in Δ are tested, and the remaining set of formulae is guaranteed to be a minimal unsatisfiable core.

Notice that minimal unsatisfiable cores are not defined in terms of their sizes but rather on the inability to find an unsatisfiable strict subset. However, sometimes a smaller set might give the user a better intuition for correction. To find a minimum unsatisfiable core, the naive approach is to check each subset of Γ and find the smallest one that is unsatisfiable, as adopted by van der Aalst et al. to identify the cause of an error [30]. Intuitively this procedure is done in a bottom-up fashion. That is, first check those subsets that contain one formula, then check those that contain two formulae, and so on until an unsatisfiable subset is found. In the worst case, if there are n formulae in Γ, one needs to test 2^n subsets, the enormous search space makes the naive approach computationally expensive. There could be more intelligent methods for this problem, but those are out of the scope of this paper.

4.4. Inconsistency analysis of the business specification

With previously described techniques and definitions at hand, we now demonstrate our strategy to analyze the inconsistency of the domain knowledge and compliance rules. Our assumption here is that in step **C** of Fig. 1, the theorem prover has returned unsatisfiability for the specification, and now we want to analyse the reason why it is inconsistent. We proceed as follows.

First of all, we check the satisfiability of the domain knowledge $\mathcal{DK}$. It is very rare that there are inconsistencies in the domain knowledge, since most of those formulae do not constrain the behavior of the business process. Second, we check whether the set of compliance rules $\mathcal{CR}$ are satisfiable, any contradictory rules will be corrected by iterating this step. Finally, when the sets of domain knowledge $\mathcal{DK}$ formulae and compliance rules $\mathcal{CR}$ formulae are both independently satisfiable, we check whether the two of them together as $\Gamma = \mathcal{DK} \cup \mathcal{CR}$ are consistent (satisfiable).

After the first two steps, if we find any internal inconsistency in the domain knowledge or the rules, backjumping gives an unsatisfiable core as the cause of inconsistency. On the other hand, if the checks in the first two steps are passed, we then proceed to the final step, after which we report errors caused by the interaction of $\mathcal{DK}$ and $\mathcal{CR}$, if any. Each time the cause of inconsistency is reported, our process synthesis procedure loops back to step **L** in Fig. 1 to refine the domain knowledge and/or rules.

For the first two of these steps, we provide three basic options to deal with inconsistencies. If Λ, the set of formulae being tested, is not satisfiable, the back-jumping procedure will return an unsatisfiable core Δ. If the user cannot identify the cause of inconsistency from Δ, then we can either (1) refine Δ by the deletion-based algorithm and find a minimal unsatisfiable core or (2) test each subset of Δ to find a local minimum unsatisfiable core in Δ. Note that the result returned by the latter option may not be a global minimum unsatisfiable core, since there may be smaller minimal unsatisfiable cores outside Δ. Therefore, we can (3) test each subset of Λ to find a global minimum unsatisfiable core.

For the last of these steps, the situation is more complicated. The first two steps guarantee that the set of domain knowledge $\mathcal{DK}$ and the set of compliance rules $\mathcal{CR}$ are satisfiable independently. So if their conjunction Γ is unsatisfiable, then both $\mathcal{DK}$ and $\mathcal{CR}$ should have at least one formula in the unsatisfiable cores. Since $\mathcal{DK}$ is the encoding of the underlying assumptions of the business process, such as “the goal must be reached” or “occurrence

of actions should lead to corresponding results", once $\mathcal{DK}$ is verified to be correct, it should not be changed. That is, the user may be more interested in the compliance rules that break the consistency. Therefore, in addition to the previously introduced three basic options, we further provide three more to focus on the compliance rules. Suppose Γ is unsatisfiable, the back-jumping procedure returns an unsatisfiable core Δ, but the user needs a deeper analysis, we split Δ into two sets: $\Delta_{\mathcal{DK}} = \mathcal{DK} \cap \Delta$, and $\Delta_{\mathcal{CR}} = \mathcal{CR} \cap \Delta$. Then we can either (4) run the deletion-based algorithm on Δ, but only test those formulae from $\Delta_{\mathcal{CR}}$, and return the remaining formula(e) in $\Delta_{\mathcal{CR}}$ as a minimal unsatisfiable core related to $\mathcal{CR}$, or (5) test each subset of $\Delta_{\mathcal{CR}}$ together with $\Delta_{\mathcal{DK}}$ to find a local minimum unsatisfiable core in Δ related to $\mathcal{CR}$. Finally, we can also (6) test each subset of $\mathcal{CR}$ together with $\mathcal{DK}$ to find a global minimum unsatisfiable core in Γ related to $\mathcal{CR}$.

The use of the six options depends on the user. We have some general observations, but the pros and cons are to be investigated as future work. Our business process synthesis procedure requires the domain knowledge and compliance rules to be consistent, so only when all the inconsistencies are removed can we proceed to generate the set of traces for this business process. Consequently, if there are multiple minimal unsatisfiable cores in Γ, we have to correct each one of them until Γ is satisfiable. In this sense, the order of discovery and their size do not matter. In general, finding a minimal unsatisfiable core is faster than finding a minimum one, so the user might tend to choose (1) and (4) instead of (2) and (5). Similarly, since the search space of (3) and (6) are in general much larger than other options, they may be costly in terms of computational time.

As another aspect of comparison, the benefit of specifically analyzing compliance rules (options (4–6)) may not be obvious under some conditions, since it may "push" the cause of unsatisfiability into the domain knowledge. A minimum (locally or globally) unsatisfiable core related to $\mathcal{CR}$ may not be minimum in Γ, because there may be more formulae in $\mathcal{DK}$ that contribute to the inconsistency. For the same reason, merely giving a set of compliance rules as the cause of inconsistency may not be so helpful in some cases, as it is possible that the error is related more tightly to a chain of formulae in the domain knowledge, thus leaving only the compliance rules that are not obviously related to each other.

Suppose the unsatisfiable core returned by back-jumping is Δ, the size of which is denoted by $|\Delta|$. In the worst case, option (1) involves $O(|\Delta|)$ calls to the theorem prover, whereas option (2) needs $O(2^{|\Delta|})$ calls to the theorem prover. The number of times Option (3) calls the theorem prover is exponential in the size of the entire formula being tested, which is usually larger than Δ. Similarly, suppose in the third step, when we test the domain knowledge and rules together, the rules in Δ consist of the set $\Delta_{\mathcal{CR}}$ of size $|\Delta_{\mathcal{CR}}|$. Then option (4) calls the theorem prover $O(|\Delta_{\mathcal{CR}}|)$ times, and for option (5) it is $O(2^{|\Delta_{\mathcal{CR}}|})$. Option (6), by contrast, invokes the theorem prover $O(2^{|\mathcal{CR}|})$ times, where $|\mathcal{CR}|$ is the number of compliance rules.

Example. In this example, we add some rules regardless of the correctness and the applicability in the banking area. The purpose is to demonstrate our inconsistency analysis.

Trivial errors such as adding $\mathbf{G}\neg edd$ to the previous four rules can be picked up by back-jumping, resulting in a subset $\{(\mathbf{F}\ edd),(\mathbf{G}\neg edd)\}$. Since this is already a minimal (and minimum) unsatisfiable core, there is no need for further refinement.

Suppose the user comes up with an idea that the due diligence evaluation should be done before the risk assessment, since if one fails the due diligence evaluation and is blacklisted, his bank account should not be opened, and thus there is no need to do risk assessment anymore (we do not consider if this is the case in real life). Therefore, risk assessment is only required when one passes the due diligence evaluation ($\mathbf{G}(edd \wedge ep \Rightarrow \mathbf{F}\ ra) \wedge \mathbf{G}(ra \Rightarrow ep)$). Moreover, the user gets confused at this point and specifies that the bank should blacklist anyone whose open-account request is denied ($\mathbf{G}(od \Rightarrow \mathbf{F}\ bl)$), and if his bank account is granted the bank should evaluate his due diligence again for double checking ($\mathbf{G}(og \Rightarrow \mathbf{F}\ edd)$).

The above rules give the following set of LTL formulae:

$$\{\mathbf{G}(bl \Rightarrow ef),\mathbf{G}(edd \wedge ep \Rightarrow \mathbf{F}\ ra),\mathbf{G}(ra \Rightarrow ep),$$

$$\mathbf{G}(od \Rightarrow \mathbf{F}\ bl),\mathbf{G}(og \Rightarrow \mathbf{F}\ edd)\}$$

Note that $\mathbf{G}(bl \Rightarrow ef)$ is added to complete the semantics of *R*4. The independent tests of the domain knowledge and compliance rules show that they are both satisfiable, but the conjunction of them yields a set of unsatisfiable formulae. However, this time the back-jumping procedure gives a large unsatisfiable core Δ which must contain formulae from both the domain knowledge and the compliance rules. To refine Δ, we use option (1) to find a minimal unsatisfiable core $\Delta^1 \subseteq \Delta$, as shown below.

$$\Delta^1 = \Delta^1_{\mathcal{CR}} \cup \Delta^1_{\mathcal{DK}}$$

where

$$\Delta^1_{\mathcal{CR}} = \{\mathbf{G}(og \Rightarrow \mathbf{F}\ edd),\mathbf{G}(od \Rightarrow \mathbf{F}\ bl),\ \mathbf{G}(ra \Rightarrow ep),\mathbf{G}(bl \Rightarrow ef),\mathbf{G}(og \Rightarrow rl),\mathbf{F}\ ra\}$$

$$\begin{aligned}\Delta^1_{\mathcal{DK}} = \{&\mathbf{G}(edd \Rightarrow \mathbf{X}\mathbf{G}\neg edd),\\ &\mathbf{G}(\neg ef \Rightarrow \mathbf{X}\ edd\ \mathbf{B}\ ef),\\ &\mathbf{G}(ef \Rightarrow \mathbf{X}\ edd\ \mathbf{B}\neg ef),\\ &\mathbf{G}(ei \Rightarrow \mathbf{X}\ edd\ \mathbf{B}\neg ei),\\ &\mathbf{G}(ri \Rightarrow \mathbf{X}\ ra\ \mathbf{B}\neg ri),\\ &\mathbf{G}\neg(ef \wedge ep),\mathbf{G}\neg(ei \wedge ep),\mathbf{G}\neg(ri \wedge rl),\\ &\mathbf{G}(ra \Rightarrow \neg end \wedge \neg start \wedge \neg edd \wedge \neg og \wedge\\ &\neg od \wedge \neg bl),\\ &\mathbf{F}(og \vee od),\mathbf{G}(start \Rightarrow ri \wedge \neg rh \wedge \neg rl \wedge\\ &ei \wedge \neg ep \wedge \neg ef),start\}\end{aligned}$$

Interpreting the meaning of those formulae may be time consuming, so suppose the user is still not satisfied with this answer and wants to focus on compliance rules, in which case option (4) is used on Δ, and returns $\Delta^4 = \{\mathbf{G}(og \Rightarrow rl),\mathbf{G}(od \Rightarrow rh),\mathbf{G}(bl \Rightarrow ef),\mathbf{G}(ra \Rightarrow ep),\mathbf{G}(od \Rightarrow \mathbf{F}\ bl),\mathbf{G}(og \Rightarrow \mathbf{F}\ edd)\}$ as a minimal unsatisfiable core of compliance rules. Note that this core is different from $\Delta^1_{\mathcal{CR}}$, this highlights the fact that there may be multiple

minimal unsatisfiable cores that give different causes of the inconsistency.

It seems that the minimal unsatisfiable core Δ^4 is still lengthy, but removing any formula φ from it gives a satisfiable set $\Delta^4 \backslash \{\varphi\}$. The reason for its large size is that it captures two interacting causes of inconsistency. First, $\{\mathbf{G}(bl \Rightarrow ef)\}$ unveils that *bl* should only be executed when one fails the due diligence evaluation, and $\{\mathbf{G}(ra \Rightarrow ep)\}$ indicates that the client's risk will be assessed only when he passes the due diligence evaluation. However, formulae $\{\mathbf{G}(od \Rightarrow rh), \mathbf{G}(od \Rightarrow \mathbf{F}\ bl)\}$ enforce that if the client's risk is assessed to be high, then his opening account request will be denied, and he will be blacklisted afterwards. Thus it is possible to blacklist a client even if he passes the due diligence evaluation, and this violates the rule that restricts blacklisting to only happen when *ef* is true.[1] Second, $\{\mathbf{G}(og \Rightarrow rl), \mathbf{G}(og \Rightarrow \mathbf{F}\ edd)\}$ manifests that *og* occurs only when the risk is assessed to be low, which implies that *ra*, and hence *edd*, have already been executed. But *og* will lead to *edd* again, which is not allowed by our "*once*" rule in the domain knowledge. The domain knowledge specifies that the goal of this process is either to grant the opening of an account (*og*), or to deny it (*od*). However, Δ^4 closes the option of *od* since it will cause *bl* to be executed incorrectly. The only remaining goal *og*gives rise to the restarting of *edd*, which forms a cycle that never ends.

When the set of unsatisfiable core is reported to the user, a new iteration is triggered so that business experts and compliance experts can discuss and redefine the domain knowledge or compliance rules. It is definitely not trivial to automatically correct the set of unsatisfiable formulae, so human involvement is needed to ensure that the intended business process is captured.

5. Trace generation and analysis

If a set of compliance rules is satisfiable, we obtain a pseudomodel that describes all traces that conform to the domain knowledge and compliance requirements. Section 5.1 shows how we extract traces from such a pseudomodel. Then, we discuss how to check a property against a set of traces using logic in Section 5.2. This technique is applied in Section 5.3 to verify correctness criteria over these traces.

5.1. Extracting traces

Given a pseudomodel, we extract traces as follows. Any sequence $\sigma = s_0, \ldots, s_n$ of states, starting at the root node of the pseudomodel can be extended into a trace. As we are modeling finite sequences with an end state, we consider a trace to be complete if $end \in s_n$. Because of the *once* constraint introduced in the Section 3, there will be no loops in the pseudomodel between the start and the end. Hence, the finite set of paths in the pseudomodel between the root state and a state labeled with *end* is the set of correct traces.

Note that it is possible to extract traces that take repetition of activities into account by omitting the *once* constraint in the domain knowledge. Still, for our purpose, this does not seem to be appropriate. Business experts rarely explicitly forbid the repetition of activity execution, but we feel that this is implicitly intended in many cases. Additionally, modeling all potential loops blurs the structure of a generated process template. As this hinders discussions between business and compliance experts, we explicitly forbid repetition for our synthesis approach.

The time complexity of our traces extraction procedure is linear in the size of traces. If the process we are trying to model is well-structured, the number of traces will usually be small. For flexible processes, however, there could be a large number of traces. The number of states on each trace, which is also the number of actions to execute on a trace (it happens to be this case because of the way we encode the domain knowledge, cf. Table 1), is another factor that determines the size of traces.

Example. Back to the rules we formulated in Section 3.2, some of the traces extracted from the pseudomodel are illustrated in Table 2. Here, the states of a trace are characterized by the conjunction of propositions that hold true in the respective state.

5.2. Process mining using logic

Traditional process mining takes the so-called event log as input, which contains the execution sequences of events that are recorded. It is common that event logs may contain noise and be incomplete [31], but in our approach, the set of traces that represents the execution sequences of actions is generated by a theorem prover for LTL. Therefore, incorrectness of our traces usually indicates that the rules are not well defined. Moreover, we do not have to consider probabilistic or heuristic approaches for handling the errors, and thus can focus on the analysis by using logic, which provides a more flexible and extensible way to reason about the information that those traces imply.

To incorporate the process mining procedure in our context, we adopt the idea of theorem proving to analyze the set of traces. To query the LTL theorem prover, we ask if $\Gamma \Rightarrow query$ is valid, where Γ is the encoded domain knowledge plus the formulated compliance rules. Note that the implementation is a satisfiability tester so that the validity of $\Gamma \Rightarrow query$ is converted to the satisfiability of $\Gamma \wedge \neg query$, and if this turns out to be unsatisfiable, then $\Gamma \Rightarrow query$ is valid. Or, we can simply express the query as $\Gamma \wedge query$ to test whether the query is satisfiable.

Since the set Γ of formulae, which can be used to construct the set of traces, does not change in our process mining procedure, we use the set of traces instead for the

[1] Note that the way we capture the semantics of a process is that if a result (e.g., *ef*) is true at a state, then this result holds at that time. If an action is true at a state, then this action happens at that time. We enforce each action to only happen once, so each action can only be true once in a trace, but the truth value of a result can persist. Therefore $\mathbf{G}(action \Rightarrow result)$ means the action can only happen when the result holds, while $\mathbf{G}(action_1 \Rightarrow \mathbf{F}\ action_2)$ means if $action_1$ happens, then $action_2$ must happen after that. They cannot occur at the same state because we have the "interleave" constraint.

Table 2
Excerpt of the extracted traces.

σ_1 : $start \wedge ei \wedge ri, edd \wedge ep \wedge ri, ra \wedge ep \wedge rh, bl \wedge ep \wedge rh, od \wedge ep \wedge rh, end \wedge ep \wedge rh$
σ_2 : $start \wedge ei \wedge ri, edd \wedge ep \wedge ri, ra \wedge ep \wedge rh, od \wedge ep \wedge rh, end \wedge ep \wedge rh$
σ_{14} : $start \wedge ei \wedge ri, edd \wedge ef \wedge ri, bl \wedge ef \wedge ri, ra \wedge ef \wedge rl, og \wedge ef \wedge rl, end \wedge ef \wedge rl$
⋮
σ_{32} : $start \wedge ei \wedge ri, ra \wedge rl \wedge ei, og \wedge rl \wedge ei, edd \wedge ep \wedge rl, end \wedge ep \wedge rl$
⋮
⋮
σ_{37} : $start \wedge ei \wedge ri, bl \wedge ei \wedge ri, edd \wedge ep \wedge ri, ra \wedge ep \wedge rh, od \wedge ep \wedge rh, end \wedge ep \wedge rh$
⋮
σ_{42} : $start \wedge ei \wedge ri, bl \wedge ei \wedge ri, ra \wedge rl \wedge ei, og \wedge rl \wedge ei, edd \wedge ep \wedge rl, end \wedge ep \wedge rl$

queries, so that there is no need for invoking the theorem prover for each query. As a consequence, the query is simplified to asking whether a formula can be satisfied by the set of traces. This greatly reduces the time cost of the procedure compared to repeated calls to the theorem prover.

The testing of a query formula φ against the set of traces is based on the semantics of LTL. To know whether $\Gamma \wedge p$ is satisfiable, where p is an atomic proposition, we check whether a given state (in this case, the first state of each trace) contains p, if p is in that state, then $\Gamma \wedge p$ is satisfiable. Formulae consisting of $\neg, \wedge, \vee, \rightarrow$ are tested according to the semantics of propositional logic. Those involving temporal operators **X**,**F**,**G** are tested respectively by checking whether the formula is true at the next state, somewhere after (including) the current state, and all the states from the current state. This is different from model checking in the sense that we are testing φ against all the possible models that Γ produces. Therefore, as long as there is a trace that satisfies φ, the trace checker will return true, otherwise it will return false. Since Γ and the set of traces $\mathcal{P}$ represent the same information in our context, we will denote "test φ against $\mathcal{P}$" as the query formula $\Gamma \wedge \varphi$ in the rest of the paper.

The above method is particularly useful when we need to check a series of properties against the same set of traces, and those properties can be translated into very small formulae. For example, those formulae being tested in the following sections usually only contain one temporal operator, thus the querying procedure only visits each state on each trace once in the worst case. The "next" operator **X** is easy to handle since we only need to test the next state, but too many eventualities such as **F** and **U** will complicate the computation. This method is not efficient if one wants to test a large formula, in which case we prefer to use theorem proving to solve the problem.

5.3. Analysis of extracted traces

As stated earlier, the goal of synthesizing a process template out of compliance rules is to support experts in getting a better understanding of the compliance aspects and to discover missing or under-specified requirements. However, it is possible to detect such under-specification by analyzing the extracted traces before proceeding to synthesizing a process template. Yet, not every semantical error in the specification can be detected, so a human expert has to validate the synthesized process template. In this section,we address the issue of under-specified LTL specifications by checking correctness criteria for the extracted traces.

Let $\mathcal{P}$ be a set of traces derived from a pseudomodel, cf. Section 5.1. We leverage the information whether an action $a \in A$ is optional for completing the process.

Definition 4 (*Optional Actions*). Given a set of actions A and a set of traces $\mathcal{P}$, the set A_O of optional actions is defined as $A_O = \{a \in A : \exists\ \sigma \in \mathcal{P}. a \notin \sigma\}$. The set A_M of mandatory actions is thus the complement of A_O, i.e., $A_M = A \backslash A_O$.

To detect optional actions, we simply test the satisfiability of $\Gamma \wedge \mathbf{G}\neg a$ for every action a. If this is satisfied by some traces, then a is optional.

We argue that the correctness of a specification where some activity is optional requires the existence of a specific data condition under which the optional activity is executed. Even if the choice of executing an activity is to be made in a non-deterministic way, an appropriate result-predicate which is set by an artificial initial activity must be part of the model. Then, we still obtain a complete specification of the behavior and, therefore, are able to ensure compliance of the created process template with the requirements. For the traces in Table 2, for instance, *og* and *od* are optional activities. The condition under which *og* executes is $(rl \wedge ef) \vee (rl \wedge ep) \vee (rl \wedge ei)$, i.e., the risk object assumes the value 'low'. Action *og* is executed independently from the value of the due diligence evaluation object. For action *od* the condition is $(rh \wedge ef) \vee (rh \wedge ep) \vee (rh \wedge ei)$, i.e., the risk is 'high'. In contrast, action *bl* is executed under the condition $(ei \wedge ri) \vee (ei \wedge\ rh) \vee (ei \wedge rl) \vee (ef \wedge ri) \vee (ef \wedge rh) \vee (ef \wedge rl) \vee (ep \wedge rh) \vee (ep \wedge rl) \vee (ep \wedge\ ri)$. Hence, none of the objects influences the decision of executing *bl*, since *bl* appears with *all* combinations of data values. Yet, *bl* is optional. This indicates an under-specified LTL specification as conditions for executing optional activities are not stated explicitly.

Definition 5 (*Optional Action Execution Condition*). Let A_O be the set of optional actions w.r.t a set of traces $\mathcal{P}$, and *RE* the set of mutually exclusive results. For an action $a \in A_O$, the execution condition is defined as $cond_a = \{\{r_1, \ldots, r_n\} : \exists\ \sigma \in \mathcal{P}. \exists\ s \in \sigma. a \in s \wedge r_1 \in s \wedge r_1 \in S_1 \wedge S_1 \in RE \wedge \ldots \wedge r_n \in s \wedge r_n \in S_n \wedge S_n \in RE \wedge n = |RE|\}$ where the sets S_i, $1 \leq i \leq n$, are different.

This definition describes the conditions under which an action executes by investigating, for each observation of the action a, the data effects (results) that are true in the same state as a. If an optional activity a has an execution condition, which is a proper subset of the combination of non-exclusive results, then this indicates a well specified set of compliance rules. We formalize this trace correctness criterion as follows.

Definition 6 (*Proper Execution of Optional Actions*). Let A_O be the set of optional actions with respect to a set of traces $\mathcal{P}$ and RE the set of mutually exclusive results. We define the set of all possible results interactions as $RI = \{\{r_1, \ldots, r_n\} : r_1 \in S_1 \wedge S_1 \in RE \wedge \ldots \wedge r_n \in S_n \wedge S_n \in RE \wedge n = |RE|\}$. An action $a \in A_O$ has a proper execution iff $cond_a \subset RI$.

The rationale behind this definition is that the execution condition of an optional action oa is proper if and only if there exist some combinations of results that prevent it from occurring. To check this, we test $\Gamma \wedge \mathbf{F}(r_1 \wedge \cdots \wedge r_n \wedge oa)$, for each $\{r_1, \ldots, r_n\} \in RI$. If there is a set in RI together with oa that is unsatisfiable, then this criterion is met for oa.

The *proper execution of actions* is the first correctness criterion to be investigated on traces before synthesizing a template. Referring to the set of traces in Table 2, we find that this criterion is not met for activity *bl*.

The second correctness criterion also relates to the execution of optional actions. Even if an optional action has a proper execution condition, the set of compliance rules might be specified in a way that allows a counter-intuitive execution of optional tasks. Imagine that rule $R3$ from Section 3.2 is modified to

$$R3: \ \mathbf{G}(od \Rightarrow (rh \vee ef)) \wedge \mathbf{G}(og \Rightarrow (rl \wedge ep))$$

Then, *od* has a proper execution condition which is: $(ri \wedge ef) \vee (rh \wedge ef) \vee (rh \wedge ep) \vee (rl \wedge ef) \vee (rh \wedge ei)$. Yet, we can observe traces like the following.

$$\sigma : start \wedge ei \wedge ri, edd \wedge ef \wedge ri, od \wedge ef \wedge ri,$$
$$ra \wedge ef \wedge rl, \ldots, end \wedge ef \wedge rl$$

In this trace, we can observe that *od* has been executed before executing *ra* and it is still a compliant execution, because the condition $ef \vee rh$ holds at the point of execution of *od*. However, from the execution condition of *od* we observe that it depends on the result of both actions *ra* and *edd*. Thus, it seems reasonable to postpone the execution of *od* until the state where *ra* and *edd* have been executed. Generally, we require that an optional action must not be executed until all actions upon which it depends have been already executed. This property, related to a concept called *natural order*, is motivated by the aim of deriving a well-structured process template, suited for discussions among experts, in an imperative modeling language that emphasizes the control flow logic. Hence, only at a few branching points, data values are considered in order to decide on the continuation. The natural order, therefore, can be seen as a means to control the number of decision points which prevents the creation of overly complex models.

Definition 7 (*Natural Order*). Let A_O be the set of optional actions with respect to a set of traces $\mathcal{P}$, $CondEff_a$ be the set of results contributing to the condition of an optional action $a \in A_O$, the set $CnA_a = \{ca \in A : R_{ca} \cap CondEff_a \neq \emptyset\}$ be the set of controlling actions for a. We say that a natural order between optional action a and its controlling actions CnA_a is kept iff $\forall \sigma \in \mathcal{P}.(\forall s_i \in \sigma.a \in s_i \Rightarrow \forall ca \in CnA_a.\exists s_j \in \sigma.ca \in s_j \wedge j < i)$, where $i,j \in \mathbb{N}$.

Definition 7 ensures that the execution of an optional action a must always be preceded by all actions which contribute to the execution condition of a.

The final correctness criterion for a set of traces is *data-completeness*. A set of traces $\mathcal{P}$ is data-complete if for every possible combination of results resulting from the *mandatory* activities, there is a trace in which this combination occurs.

Definition 8 (*Traces Data-Completeness*). Let $\mathcal{P}$ be a set of traces, A_M be the set of mandatory actions and RE_M be the set of mutually exclusive results of mandatory actions, defined as $RE_M \subset RE.\ \forall E_M \in RE_M\ \forall r \in E_M.r \in R_a$ where $a \in A_M$. We define the set $RI_M = \{\{r_1, \ldots, r_n\} : r_1 \in S_1 \wedge S_1 \in RE_M \wedge \ldots \wedge r_n \in S_n \wedge S_n \in RE_M \wedge n = |RE_M|\}$. The set of traces $\mathcal{P}$ is data-complete iff $\forall C \in RI_M.\ \exists \sigma \in \mathcal{P}.\ \exists s_i \in \sigma : \forall r \in C\ r \in s_i$ where $i > 0$ and the sets S_i are different.

To verify this, we test each exclusive set of results of each mandatory action. That is, for every $\{r_1, \ldots, r_n\} \in RI_M$, we ask $\Gamma \wedge \mathbf{F}(r_1 \wedge \cdots \wedge r_n)$, if the answer is satisfiable for every set in RI_M, then the set of traces is data-complete.

A process template may be generated even if data incompleteness is detected for a set of traces. However, the template could suffer from deadlocks as for some combinations of results, continuation of processing is not defined.

6. Addressing failed trace correctness criteria

In Section 5.3 we have described three correctness criteria for the generated traces that must be fulfilled before a process template can be generated.

In this section we discuss approaches to help refine the compliance rules and thus the traces in case an improper execution condition or data-incompleteness is found in the traces.

6.1. Handling optional actions

A set of traces fails *the proper execution condition* criterion if at least one optional action appears under all possible result interactions. The primary reason of this violation is the under-specification of conditions under which an optional action shall be executed. Business experts usually tend to express the rules in the way $R4$ is specified, i.e., "If due diligence evaluation fails, then the client has to be added to the bank's black list", which is formalized as $\mathbf{G}(edd \wedge ef \Rightarrow \mathbf{F}\ bl)$. Thus, they focus on the reasons that call for executing some optional action without explicitly specifying data conditions that must hold at the point in time such actions are executed. As a result, the satisfiability checker generates some traces

that are compliant with the explicitly mentioned rules yet are meaningless from a business-expert point of view. In this behavior, optional actions are executed unnecessarily, cf. trace σ_{37} in Table 2 where the client is immediately blacklisted before any other actions occur.

In this section, we address this problem of *improper execution conditions of optional actions* by detecting so-called *implied runs*. The term implied run is inspired by the term *implied scenario* [32,33] from the requirements engineering domain. An implied scenario represents an *unnecessary* and usually *unwanted* extra behavior of a software system that was not intended by the users. The detection of implied scenarios indicates under-specification of requirements and triggers a new iteration of specification refinement.

Definition 9 (*Implied Process Run*). Given a set of actions A and a set of process runs $\mathcal{P}$, a process run $\sigma : s_0, \ldots, s_n \in \mathcal{P}$ is called *implied*, if there exists an action $a \in A_O$ and an integer k such that $a \in s_k$, and there is a process run $\sigma' : s_0, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n \in \mathcal{P}$.

Intuitively we can delete σ since the almost identical σ' can substitute for it, and the optional action a has no effect on the compliance of the process.

Based on the implied run notion, we reduce the set of traces considered for process template generation by removing implied runs. With our approach, we strive for a *minimal* representation of the compliant behavior. Hence, the potential execution of an optional action should be neglected in order to synthesize *compact* process templates. To detect such optional execution, we check all process traces that contain an optional action such that the removal of the according state results in a process run that is also in the set of all process runs. If so, this provides us with evidence that the optional action is not needed to complete the process run (trace) in a compliant manner. Hence, such an *implied process run* is not considered for process template generation. With the removal of implied runs we can obtain a proper execution condition for optional task and thus can continue trying to generate a process template. Moreover, we can suggest to the user further rules that explicitly state conditions for the execution of optional tasks of the form $\mathbf{G}(a \Rightarrow cond_a)$, where a is an optional task and $cond_a$ is now the proper condition according to Definition 6.

Definition 7 describes the natural order correctness criterion between optional actions and their controllers as lacking traces in which an optional action might appear before all of its controllers have already been executed. If this criterion is not met, we follow the same approach we did with the implied runs: we delete the traces in which the criterion is violated. This comes with the cost of not being completely faithful to the behavior stated in the rules, but we argue that experts care only about executing the optional action after its controlling actions have already completed. Moreover, maintaining the natural order helps produce structured models which will be better understood by experts and better serve as a design template for operational processes.

Based on each violating trace that we drop, we can deduce an explicit ordering rule between the optional action and its controller of the form $\mathbf{G}(start \Rightarrow (ca\ \mathbf{B}\ oa))$, where oa is the optional action and ca is one of its controlling actions.

Example. In our running example, we have identified that task *bl* has an improper execution condition, cf. Table 2. According to Definition 9, run σ_1 is an implied run because removing the state in which *bl* appears yields the trace σ_2. We can see that *bl* was unnecessarily executed in run σ_1, because the task *edd* resulted in *ep*. That is, the due diligence evaluation succeeded and there is no need to black list the customer. On the other hand, trace σ_{14} cannot be dropped as there is no other trace that looks exactly the same but avoids executing *bl* because removing the state where *bl* executes yields a non-compliant execution since rule *R*4 will not be fulfilled. After dropping the implied runs, we can suggest the following rule $\mathbf{G}(bl \Rightarrow ef)$ that explicitly states that *bl* executes only when *ef* holds.

6.2. Identifying vacuously satisfied rules

The satisfiability checker is designed in a way that generates all possible traces, that satisfy Γ. However, in some cases, the traces satisfy a rule vacuously, especially if these rules lead to contradictions with other rules. Imagine two actions a and b where a has the exclusive results $r1$ and $r2$. Suppose that we have two rules $t1$: $\mathbf{G}(a \wedge r1 \Rightarrow \mathbf{F}\ b)$ and $t2 : \mathbf{G}(\neg b)$. It is obvious that it is not possible to satisfy $t1$ by allowing a to produce the effect $r1$ because this contradicts with $t2$. Rather, $t1$ is satisfied vacuously by never making the condition of $t1$ true.

Data-incompleteness of the traces occurs when some rules are vacuously satisfied. This indicates that some result combinations will lead to a contradiction and thus the satisfiability checker avoids producing these results.

Definition 10 (*Vacuously satisfied rules*). Let $\mathcal{CR}$ be the set of LTL rules representing compliance requirements, $\mathcal{DK}$ be the set of LTL rules representing the domain knowledge where $\Gamma = \mathcal{DK} \cup \mathcal{CR}$ and let $\mathcal{P}$ be the set of traces generated for Γ. A rule $r \in \mathcal{CR} \cup \mathcal{DK}$ is vacuously satisfied by $\mathcal{P}$, written as $\mathcal{P} \models_v r$, iff $\forall \sigma \in \mathcal{P}. \nexists s \in \sigma. s \models cond(r)$, where $cond(r)$ is the condition part of the rule r.

For each rule, trace $\sigma \in \mathcal{P}$ and state $s \in S$ we need to check whether the conjunction of atomic propositions that hold true in state s logically implies the propositional formula forming the condition of the rule. The result of this scan is the set $VS = \{r \in \mathcal{CR} \cup \mathcal{DK} : \mathcal{P} \models_v r\}$.

We construct VS by finding those rules r that are unsatisfiable when we force $cond(r)$ to occur. That is, we check $\Gamma \wedge \mathbf{F}(cond(r))$ and we run the test described in Section 4. These inconsistencies are then reported to the user for correction, e.g., refining the rules, and then iterate, cf. Fig. 1.

7. Process template generation and evaluation

Given a set of traces that meets the aforementioned correctness criteria, we proceed by generating a process template. To this end, we adapt techniques from the field of process mining and process restructuring to create an initial process template in Section 7.1. Then, Section 7.2

shows how the initial template is augmented with data conditions. Finally, Section 7.3 elaborates on the evaluation of the generated process template.

7.1. Generating process templates

The generation of an initial process template builds upon techniques proposed in the field of *process mining* [34] and process restructuring [35,36]. Most mining algorithms neglect the difference between control flow dependencies and data flow dependencies when generating a process model. Therefore, we cannot apply an existing algorithm directly. Also, process templates are intended to serve as a means for discussion and negotiation between business and compliance experts. Hence, we want to ensure that the generated template is easy to understand. It has been shown that block-structured process models are more easy to understand than arbitrarily structured process models [37]. Block-structuredness refers to a topology of a process model that requires every node with multiple outgoing edges to have a corresponding node with multiple incoming flows such that both nodes form a single-entry single-exit block. To obtain such a process model structure, we combine basic techniques from the field of process mining, i.e., behavioral relations known from the α-mining algorithm [34], with techniques that aim for the construction of a block-structured process model, see [35,36].

Order of actions. As a first step, we extract the precedence of actions. To this end, we employ the order relations known from the α-mining algorithm [34].

Definition 11 (*Order Relations*). Let $\mathcal{P}$ be a set of traces and A the sets of actions. We define the following *order relations* for two actions $a_1,a_2 \in A$.

$a_1 > a_2$: iff there is a trace $\sigma : s_0,\ldots,s_n \in \mathcal{P}$, such that $a_1 \in s_i \wedge a_2 \in s_{i+1}$ for some $0 \leq i < n$.
$a_1 \rightarrow a_2$: iff $a_1 > a_2$ and $a_2 \not> a_1$.
$a_1 \parallel a_2$: iff $a_1 > a_2$ and $a_2 > a_1$.
$a_1 \# a_2$: iff $a_1 \not> a_2$ and $a_2 \not> a_1$.

For two actions ordered by $>$, we know that the first action appears immediately before the second action. We obtain the order relations by testing satisfiability for the following formula expressed in LTL: $\Gamma \wedge \mathbf{F}(a_1 \wedge \mathbf{X}\ a_2)$. If this formula succeeds, we conclude that $a_1 > a_2$. Then, the relations $\rightarrow$, $\parallel$, and # are derived from the results for the $>$ relation as stated in Definition 11. For the model synthesis, therefore, we have to test satisfiability using the method mentioned in Section 5.2 for n^2 formulae with n as the number of distinct actions.

Synthesis of process model: Having defined the order of actions, one may directly proceed by applying the α-mining algorithm [34] to construct a process model. However, this approach has drawbacks. First, the construction of a process model may result in a model that shows behavioral anomalies, such as deadlocks. The α-mining algorithm does not specify the requirements for the derivation of a sound process model without anomalies on the level of behavioral relations. Therefore, one needs to construct a model using the algorithm and check its correctness subsequently. Second, the α-mining algorithm encodes causal dependencies directly, which may lead to the interference of synchronization of parallel paths and the choice about exclusive continuations. This complicates the annotation of data conditions, since there is no unique point at which the decision is taken. For these reasons, we rely on the order relations of the α-mining algorithm, but adopt the synthesis technique presented in [35,36]. It leverages the notion of an order relations graph that is defined for a set of behavioral relations. We adapt this notion towards the relations introduced earlier.

Definition 12 (*Order Relations Graph*). Let $\{\rightarrow,\#,\parallel\}$ be order relations for a set of actions A via Definition 11. The *order relations graph* $\mathcal{G} = (V,E)$ comprises all actions as nodes, $V = A$, and the relations $\rightarrow$ and # as edges, $E = (\rightarrow \cup\ \#)$.

Edges in the order relations graph represent the order or exclusiveness of actions. Both relations are distinguished by unidirectional or bidirectional edges. Using this graph, we employ the synthesis technique introduced in [35,36]. We limit ourselves to an informal description of this synthesis and refer to [35,36] for the formal details.

The synthesis employs the modular decomposition [38] of the order relations graph. It detects subgraphs, which show uniform relations with all other nodes of the graph. The modular decomposition technique parses a graph into a rooted hierarchy of these subgraphs that are maximal and non-overlapping in terms of their contained nodes. The modular decomposition technique distinguishes different types of detected subgraphs. Trivial subgraphs comprise only a single node. Subgraphs that are complete graphs are referred to as being XOR-complete, subgraphs that are edgeless are referred to as AND-complete. Further, a subgraph is linear, if and only if all of its nodes are sequentially ordered. Finally, subgraphs that do not meet any of these requirements are called primitive. Algorithms to obtain a modular decomposition tree run in time linear in the size of the order relations graph [38]. The size of the decomposition tree is also linear in the size of the graph. The size of the graph, in turn, is determined by the number of actions.

For the generation of a process template, we require the order relations graph to be free of primitives. If this is not the case, user input is required on the behavioral dependencies between the actions that are part of this subgraph. If the order relations graph is free of primitives, the synthesis algorithm iteratively constructs a process model from the identified subgraphs, cf. [35,36]. That is, trivial subgraphs contain a single action, which is added as an activity to the process model. Subgraphs that are XOR-complete or AND-complete are represented by a block that is bordered by gateways with either XOR- or AND-logic. Finally, subgraphs that are linear lead to the construction of edges between the respective activities.

Example. After we adapted the set of constraints for our running example as discussed above, we derive the order relations graph, visualized as the first graph in Fig. 2. Since

the graph is free of primitive subgraphs, we proceed by applying modular decomposition. Fig. 2 illustrates this decomposition which identified subgraphs comprising nodes with uniform relations to all other nodes. Based on these subgraphs, the model synthesis returns the process template visualized in Fig. 3. Note that this template does not correctly represent the action 'black list a client' as an optional action. This is due to the absence of any exclusive action that is executed instead of this activity. Such anomalies are corrected when annotating the template with data conditions.

7.2. Annotating data conditions

The process template created so far lacks details on the data conditions that lead to the execution of optional activities. We next augment the process template with such conditions. We already introduced the notion of an execution condition for optional actions in Definition 5. It is important to see that these conditions are not local (i.e., consider only the directly preceding actions as in our previous work [10]). Instead, indirect data dependencies in terms of results that have been obtained by actions that happened long before are also taken into account.

To annotate the process template with these execution conditions one may guard the execution of each optional activity with the respective condition separately. However, this may lead to overly complex templates. Imagine that there is a sequence of actions where all actions are guarded by the same execution condition. Then, inserting decisions points for all of them separately would inflate the template drastically compared to inserting just one decisions point that guards the whole sequence of actions. Hence, we proceed as follows.

1. We consider all XOR-complete subgraphs that have been detected during the model synthesis. For each of these subgraphs, we investigate whether the actions in each of the child subgraphs show the same execution condition. If so, we annotate the edge leading from the respective splitting gateway with XOR-logic to the actions of the child subgraph with this execution condition. We keep the information on whether all actions of a child of an XOR-complete subgraph could be treated in this way.
2. For all optional actions that are have not been treated by the previous step, we insert two additional gateways with XOR-logic directly before and after the action. Note that the process template is acyclic and all actions have at most one predecessor and at most one successor. The edge between the gateway before the action and the action itself is annotated with the execution condition. Further, an edge is defined between the gateway before the action and the one after the action. This edge is annotated with the disjunction of all results under which the execution of the optional action is not observed. Those results are determined by removing the result terms of the execution condition from the set of all possible result combinations of the objects references in the execution condition.

Again, this step does not impose computational challenges. The data conditions for all actions have been determined before. The annotation of process templates first requires iteration over all subgraphs and comparison of the data conditions for all children. In a second step, we iterate over all optional actions that have not been treated before.

Example. For our running example, we first observe that the XOR-complete subgraph that spans the actions *og* and *od* can be treated as follows. The edge leading to action *og* is annotated with the result *rl*, whereas the edge leading to action *od* is annotated with the result *rh*. That is, the request to open an account is granted only if the risk is considered to be low. If the risk is high, the request is denied. In addition, we have to deal with the optional action *bl*, which is not part of a child of an XOR-complete subgraph. As such, we introduce two XOR-gateways and annotate the edges as outlined above. Here, the edge bypassing the action *bl* is annotated with $ei \vee ep$ since the evaluation object must have a value other than *ef* to avoid executing *bl*. However, due to the action *edd* which must have occurred already, the object may only have value *ef*

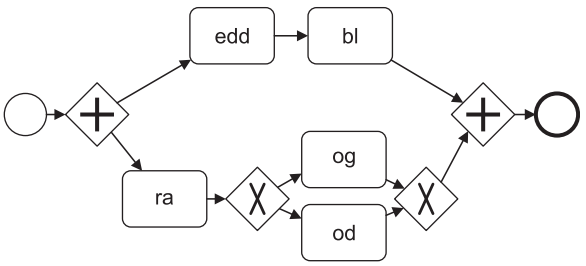

Fig. 3. Intermediate process template synthesized for the example, annotations to constrain the execution of optional activities are still missing.

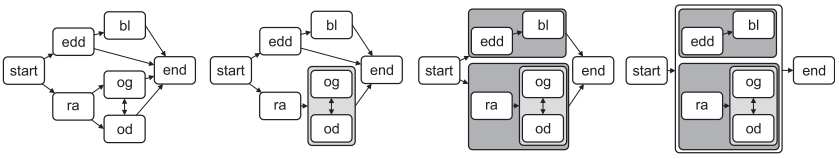

Fig. 2. Order relations graph and the single steps of the modular decomposition.

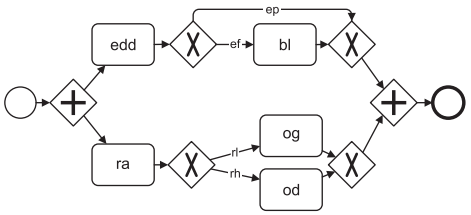

Fig. 4. Annotated process template synthesized for the example.

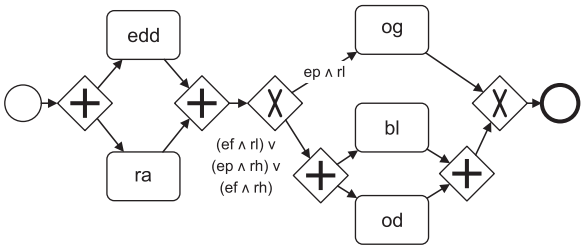

Fig. 5. A compliant process template where *bl* and *og* are exclusive and the conditions for their execution have been adjusted.

or *ep*. The obtained annotated process templates is shown in Fig. 4.

7.3. Evaluation of the synthesized process template

Process templates aim to support experts in getting a better understanding of the compliance aspects and to discover missing or under-specified requirements. Such under-specification is manifested in the process template in terms of semantical problems. Those problems can only be detected by human experts. In this section, we will further elaborate on the running example to illustrate such problems. Using the process template in Fig. 4 as a basis of the discussion between compliance expert and business expert, they identify that the template allows for executing both black listing the client and granting the request to open the account in the same trace. This is an example of the aforementioned semantical problems caused by under-specified compliance rules. The compliance expert refines the set of constraints by indicating that black listing and granting the request to open an account are contradictory, cf. the *CA* relation in Section 3.2, formalized as $\mathbf{G}(og \Rightarrow \mathbf{G}(\neg bl))$ and $\mathbf{G}(bl \Rightarrow \mathbf{G}(\neg og))$. Repeating the steps of our approach reveals that the adapted set of compliance rules yields a set of traces that is data incomplete. This is explained based on the two added constraints as follows. By forcing *bl* and *og* to be exclusive, we implicitly require *bl* to be executed only with the condition $ef \wedge rh$, while *og* is executed only with the condition $ep \wedge rl$. Other combinations of results are not considered. There is no trace that addresses the situation where $ef \wedge rl$ holds in some state. This contradicts our requirement to execute either *og* or *od* in each run. Since the condition $ef \wedge rl$ enables neither of them, it is not observed in any of the generated traces.

As a consequence, another adaptation of our set of compliance requirements is needed. One solution is to update the conditions under which the actions *og*, *od*, and *bl* are executed, i.e., $\mathbf{G}(og \Rightarrow ep \wedge rl)$, $\mathbf{G}(od \Rightarrow (ef \vee rh))$, and $\mathbf{G}(bl \Rightarrow (ef \vee rh))$. With these updated constraints, another iteration of behavior synthesis is started. This time, the generated set of traces shows data completeness. The final generated process template is visualized in Fig. 5.

When the regulation or the law changes, many related rules have to be modified, added, or deleted, and consequently the process template will be different. Moreover, any process models refined from these templates will have to be adapted. Much previous work has been done by a number of researchers to investigate dynamic workflow changes, and many of them dealt with the change by reconstructing graphs [39–41]. In our approach, however, there is no need to worry about the correctness of changed graphs, since we only need to change the rules and the corresponding new graph of process templates will be generated automatically. Our idea of combining logic and process mining helps make the change easier for the user, since we can also detect incorrect changes and check the properties of the changed process by the same means. Iterations in enacting the process template in our paper can be seen as small changes in rules, and we have shown how the changed process template is generated and evaluated. As future work, we will look for methods to check if the current executing instance complies with the changed process, and how to cope with changes that involve too many rules. Those, however, are beyond the scope of this paper.

8. Implementation

We created a prototypical implementation to validate our approach. Fig. 6 shows a snapshot of it. It relies on a specification of domain knowledge, such as activity results and contradicting activities, which has to be defined once by a human expert. When the domain knowledge is defined, we generate the LTL encoding formulae automatically according to Table 1. Then a set of compliance rules has to be defined by the human expert to control the behavior synthesis and to enforce the semantics of the domain knowledge. The user can check the rules that have been entered, and those that are generated automatically in the "LTL preview" tab. The theorem prover is implemented according to Wolper's method for checking LTL satisfiability [18]. If the rules are satisfiable, the graph-based tableaux method LTL theorem prover generates the pseudomodel of all possible traces. Next, our implementation extracts finite traces, which are shown in the "Traces view" tab. Then the set of traces is analysed against the correctness criteria in Section 5.3. The analysis of traces is done by querying properties against the set of traces, as described in Section 5.2. If the extracted traces pass the quality tests, then the tool computes the order relations between actions as in process mining by the same querying method, and constructs an order relations graph. In contrast to our original approach [10] which uses traditional process mining to analyse traces and derive relations in the process, our new implementation to query the set of traces is from a logic perspective. That is, we test the properties of the set of

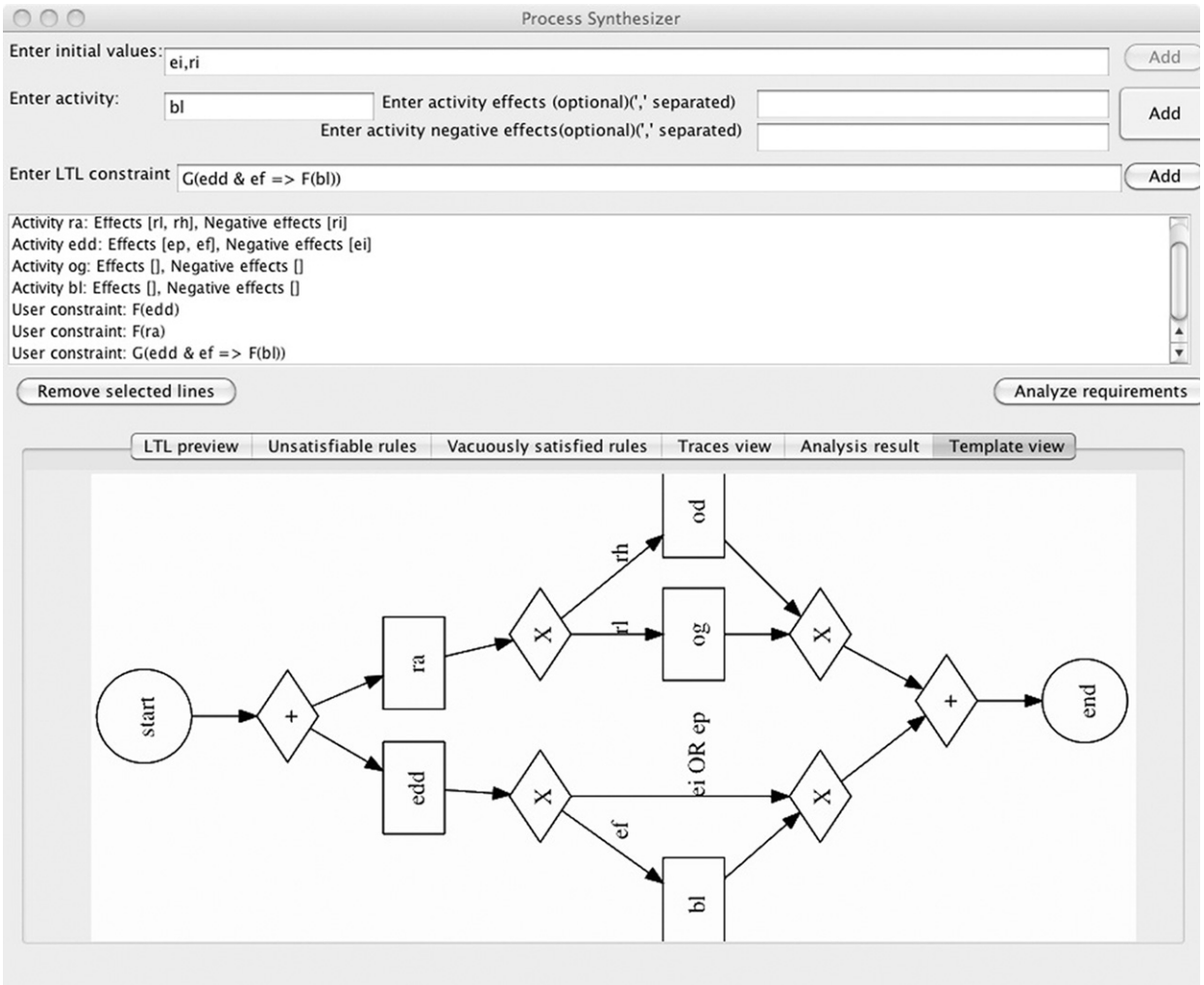

Fig. 6. A snapshot of the process synthesis tool.

traces based on the semantics of LTL. Since our encoding of the domain knowledge and compliance rules are both in LTL, the new approach coherently provides us with logical correctness, and can be easily extended for further use. Finally, a first overview of the process template is given using the approach of [10] with a GraphViz [42] based visualization in the "Template view" tab. Further, artefacts to generate the final template with the approach introduced in this paper are also provided.

In case that our theorem prover returns unsatisfiability for the domain knowledge and the rules, our implementation conducts the inconsistency analysis in three phases to check the problems in the domain knowledge, the rules, and the conjunction of them respectively, cf. Section 4.4. For the first two phases, we provide the user with three options to check the cause of inconsistency, while in the last phase six options are enabled for the user to choose from. Each of those options is realised by different methods and thus gives different performance and results. The unsatisfiable core is reported in the "Unsatisfiable rules" tab.

In case that traces do not pass checks described in Section 5.3, the problems found are reported on the "Analysis result" tab. At that point, the user is offered the option to apply the remedy strategies discussed in Section 6. In particular, our tool will check for vacuously satisfied rules and the result is reported in "Vacuously satisfied rules" tab.

There is the potential for a state space explosion, especially since the additional constraints of the process are unrestricted logical formulae. Even without pathological constraints, if there is a lot of freedom or many non-local conditions then the satisfiability checking phase can take a considerable amount of time. The *once* constraint helps limit this, and too much freedom can often be a sign that other conditions have been omitted. We aim to evaluate these issues in further case studies.

9. Feasibility of our approach

To evaluate the applicability and feasibility of our approach, two further examples and their run times are shown in the appendix. Interestingly, some technical details that affect the performance of our approach can be observed in those examples.

In A.1, a total of 593 traces are generated out of the set of rules, while in the second example less than 10 traces are generated. Consequently the trace evaluation and process mining take rather long in A.1. This is mainly caused by a lot of rules with **F**. In theorem proving, if there is an $\wedge$ operator, the child state contains both conjuncts, whereas to deal with an $\vee$ operator, we must split the current state into two children where each contains a disjunct. That is, when there are many $\vee$ operators, the search will have to go through a large number of branches in the worst case, but for $\wedge$ operators the search goes linearly. Temporal operators **F**, **U** behave like $\vee$, and **G**, **B** behave like $\wedge$. This is the reason that A.1 contains more traces than the other example. For instance, if we add $\mathbf{G}(arc \Rightarrow \mathbf{G}\ \neg rej)$ (which equals to $\mathbf{G}(\neg arc \vee \mathbf{G}\ \neg rej)$) in Appendix A.1, the theorem prover spends 12 s but generates 5298 traces. As a result, it takes 2.5 s, which is longer than in the reported examples, to evaluate and process-mine those traces. The result of the process template, however, is the same as the one we have shown in A.1. Furthermore, adding more rules with **G** and **B** sometimes helps the theorem prover to skip unsatisfiable paths earlier when searching. For example, if we add rules $\{(odr\ \mathbf{B}\ crc),(crc\ \mathbf{B}\ ivc),(ivc\ \mathbf{B}\ evl)\}$ in A.1, the theorem prover would give the same answer and generate an equivalent set of traces in 0.29 s. Note that $\mathbf{G}(action_1 \Rightarrow \mathbf{X}\ action_2)$ indicates that if $action_1$ occurs, $action_2$ must occur exactly afterwards. But this allows $action_2$ to appear on a trace solely. The rule $(action_1\ \mathbf{B}\ action_2)$ forces that if $action_2$ is on a trace, then $action_1$ must be executed before that. So the two rules together restrict the sequence of execution that we intend to model. But in A.1, the structure of the process is rather fixed. It happens to be the case that in all rules of this form, $action_2$ cannot occur solely on a trace. Therefore, even though we do not add rules with **B**, the result is the same.

As one can see, the time cost for solving the problem sometimes depends on how the rules are interpreted and how they are formulated in LTL. Thus, it is difficult to give a conclusion of run time analysis and scalability, since if the rules are not well-defined we might get a false negative answer. The intelligent way to formulate compliance rules can be a very important and interesting aspect of further work.

Although we cannot claim our approach to be scalable without further extensive experimental evaluation, the time costs for examples in the appendix are all within reasonable tolerance. The last two examples are inspired by those in previous papers, so their validity and practicality should be inherited. In all, the examples exhibited that our approach is not ad-hoc, but can be used to model small example processes in other areas in a relatively short time.

10. Related work

Compliance checking of business process models with a focus on execution order constraints has been approached from two angles: namely compliance by design and compliance checking of existing models. The latter has been tackled using model checking techniques [5,4,43]. Our work follows a compliance by design approach that has also been advocated in [7,2,3,44,45,8]. Close to our work, the authors of [2,7] employ temporal deontic assignments to specify what can or must be done at a certain point in time and synthesize a process template from these assignments. In contrast to our work, however, the approach is limited to temporal dependencies between activity executions and the underlying logic requires an encoding of these dependencies via explicit points in time. Another approach to synthesize compliant processes was introduced in [8]. The authors employ a set of compliance patterns expressed in LTL. For each pattern a finite state automaton (FSA) is defined. To synthesize a process, the FSAs of the involved patterns are composed. Next, the user is required to select for each composition an execution path in order to synthesize the process. That approach is able to generate processes with sequence and choice only. Moreover, it does not consider data flow aspects in the synthesized process. In [9], the authors develop an approach to extract process structure from software requirements expressed in the Service Oriented Requirements Language (SORL). The language provides a set of patterns which are similar to loop, sequence, and choice patterns of business processes. Thus, the extraction of the control structure of the process is rather straightforward. Also, under the heading "Object Depend", the authors mention that their approach can model processes which make decisions, but only where those decisions are based on binary results of objects (usually "true" or "false"). Compared to their work, we can generate process templates with structures that have non-local data conditions and multi-valued (more than two) data conditions. Further, without explicit data-flow, most of the related papers attempted to model choices by using extra actions [13], which complicate the process model. Some papers mention the choices in the description of their process, but those conditions are not presented in the process model [30], and thus might confuse the user when making decisions.

As we mentioned before, our aim is to produce a complaint process template. The work in [6] is interesting as it is complementary to our work. In [6], the authors have proposed a *layered* refinement process whose input is a compliant process template and its output is a fully refined compliant executable process.

Another interesting complementary work to ours is about extraction of compliance requirements from legal documents. These approaches are surveyed in [46]. According to the survey, a plethora of techniques have been used to extract and formalize compliance requirements from legal documents. What is interesting is that some techniques extract and formalize some requirements. This could form the input to our approach where we produce process templates out of these requirements. Another interesting aspect that has been covered by the survey is *law-compliant business process templates*. The surveyed papers show that the effort is mainly to manually develop a set of compliant business process templates out of legal requirements. Compared to our approach, we have the advantage of semi-automating this step.

Related to our approach to process model synthesis is work on process mining, which aims at automatic construction of a process model from a set of logs [47,34,48]. We adapted the α-algorithm [34], a standard mining

approach, for our purposes. Besides the commonalities, there are some important differences between process mining and process template synthesis. We consider control flow routing based on data values. This aspect is often neglected in process mining algorithms. Only recently, time information and data context have been considered when predicting the continuation of a trace based on its current state [49,50]. Further, process mining approaches have to be robust against incorrect data (log noise). As we derive a model from artificially generated traces, this is not an issue for our approach.

Work on declarative business process modeling is also related to our work. The authors of [13,12] propose to model processes by specifying a set of execution ordering constraints on a set of activities. These constraints are mapped onto LTL formulas; which are used to generate an automaton that is used to both guide the execution and monitor it. That is similar to our approach of generating a pseudomodel. Recently, the authors also showed how finite traces that respect interleaving semantics can be extracted from a set of LTL constraints [14]. The major difference from our work is that [14] does not model data constraints as we do. They also change the semantics of LTL rather than by using standard LTL as we do. Finally, we initially tried the approach of extracting Büchi automata from our LTL specifications for our example, but found that the automata approach required hours to return the automata whereas our LTL satisfiability checker returns a pseudomodel in a few seconds.

Model synthesis is also a relevant issue in model driven development. Synthesis of behavioral models out of scenario-based models has received attention from researchers [32]. A similar problem to the notion of implied runs, cf. Section 6.1, has been identified as implied scenarios where implicit behavior appears due to under-specification. In our work, we take the synthesis one step ahead and produce a process model rather than stopping at the state machine step. Also, the issue of requirements consistency checking is relevant. However, inconsistency appears in the form of negative implied scenarios, i.e., unwanted interactions among interacting objects. Some of these approaches, e.g., [51], reduce the detection of inconsistency to a model checking problem. On the one hand, the sequence charts for different scenarios represent the behavior of the system. On the other hand, the user requirements represent properties to be checked. Consistency is achieved when the system behavior satisfies the user requirements. In our work, the only source of specification is the set of compliance rules. Thus, our inconsistency analysis is basically checking the satisfiability of the LTL specification.

The issue of identifying causes of unsatisfiability has also been an issue in the formal verification field. In [52], the authors propose an approach to check satisfiability of PSL, a super language of LTL, as well as identification of causes of unsatisfiability. First, they consider Boolean abstraction over the specification and check whether the conflict occurs due to the Boolean formulation, e.g., $p \wedge \neg p$. If this is not the case, they reduce the satisfiability checking problem to a model checking one and based on Bounded model checking, they identify the causes of the inconsistency. Still, the identification of vacuous satisfiability is not considered.

11. Conclusion

In this paper, we introduced an approach to synthesize business process templates out of a set of compliance rules expressed in LTL. We also showed that extra domain-specific knowledge is required to decide consistency of such requirements and introduced an LTL encoding for compliance rules and domain knowledge. This was used to generate traces from which a process template is constructed.

Our approach addresses control- and data-flow aspects of compliance rules, whereas most of the existing work focuses on control-flow aspects only. The consideration of data-flow aspects comes with new challenges. Data dependencies may show rather complex interactions that are hard to handle in the first place. We cope with these challenges following an iterative approach – the required knowledge is built incrementally each time inconsistencies or semantical issues are detected. Our approach is comprehensive in the sense that it provides resolution strategies that support experts in resolving these issues. In particular, we showed how inconsistent subsets of the given knowledge base are identified. Also, we addressed the case when correctness criteria for traces are not met.

However, we also have to reflect on certain limitations of our approach. In particular, the model synthesis from a set of traces is limited to block-structured process models. Even though it is desirable to synthesize block-structured models, there may be settings in which this is not possible, but in which an inherently graph-structured process model would still be reasonable to use for negotiation between business experts and compliance experts. Note that we explicitly decided against the direct application of the alpha-mining algorithm (which is not limited to block-structured models) in order to be able to consider direct and indirect data dependencies. Those are of crucial importance for creating shared understanding on the business operations between business experts and compliance experts. In future work, however, we aim to address this limitation.

Further, we argued that while compliance rules rarely explicitly forbid the repetition of activity execution, it is often implicitly intended, and useful for clarity. Hence, by default we forbid repetitive behavior, which also helps to keep the generated process templates concise. However, there may be cases in which compliance requirements relate to the number of allowed executions of an activity. Further work is needed to extend our approach to cope with such requirements.

Future work also includes the application of our approach in case studies. In this work, we presented a prototypic implementation of the complete approach. However, many applicability aspects can only be evaluated for a concrete compliance setting. For instance, the amount of underspecification that needs to be addressed as part of the iterative processing and the amount of negotiation needed between compliance and business experts influences the applicability of our approach, but

those highly depend on the concrete process and compliance requirements. As part of that, it may be necessary to extend the approach to also consider constraints on role resolution for generating process templates.

Appendix A. Further examples

In the appendix, we present further examples modelled using our approach. On the one hand, the examples demonstrate the applicability and feasibility of our method. On the other hand, we provide run time information to illustrate practicality from the performance perspective.

A.1. The order and purchase process

This example is taken from the paper by Ellis et al. [40]. An office procedure for order processing within a typical electronics company. When a customer requests (*odr*) by mail, or in person, an electronic part, this is the beginning of a job. A form is filled out by the order administrator; the job is sent to credit check (*crc*), and then to inventory check (*ivc*). After the evaluation (*evl*), either a rejection letter (*rej*) is sent to the customer when the evaluation fails (*ef*), or the order is approved (*apr*) when the evaluation passes (*ep*) and then sent to shipping (*shp*) and billing (*bil*). The shipping department will actually cause the part to be sent to the customer; the billing department will see that the customer is sent a bill, and that it is paid. In the end of a successful purchase, the transaction will be archived (*arc*).

From their description of the process, we extract the following actions and results. Note that there is no results in their original process, but since actions "approval" and "rejection-letter" are straightforward, there ought to be no misunderstandings when executing this process. In the next examples, however, we will illustrate how our data-flow helps the user to make decisions.

$Actions = \{odr, crc, ivc, evl, apr, rej, shp, bil, arc\}$
odr: order entry
crc: credit check
ivc: inventory check
evl: evaluation
apr: approval
rej: send rejection letter
shp: shipping
bil: billing
arc: archiving
$Results = \{ei, ep, ef\}$
ei: evaluation is initial
ep: evaluation passed
ef: evaluation failed

To adapt to our approach, the following rules are translated into LTL:

$\mathbf{G}(start \Rightarrow \mathbf{X}\ odr)$
first step, the customer requests the order
$\mathbf{G}(odr \Rightarrow \mathbf{X}\ crc)$
then conduct the credit check
$\mathbf{G}(crc \Rightarrow \mathbf{X}\ ivc)$
then conduct the inventory check
$\mathbf{G}(ivc \Rightarrow \mathbf{X}\ evl)$
then evaluate the above checks
$\mathbf{G}(evl \wedge ep \Rightarrow \mathbf{X}\ apr) \wedge \mathbf{G}(apr \Rightarrow ep)$
if evaluation is passed, approve the order
$\mathbf{G}(evl \wedge ef \Rightarrow \mathbf{X}\ rej) \wedge \mathbf{G}(rej \Rightarrow ef)$
evaluation is failed, reject the order
$\mathbf{G}(apr \Rightarrow \mathbf{F}\ shp)$
if the order is approved, the item will be shipped
$\mathbf{G}(apr \Rightarrow \mathbf{F}\ bil)$
if the order is approved, the bill will be sent
$\mathbf{G}(shp \Rightarrow \mathbf{F}\ arc)$
the transaction will be archived after shipment
$\mathbf{G}(bil \Rightarrow \mathbf{F}\ arc)$
archiving is after the bill being sent also
$\mathbf{G}(rej \Rightarrow \mathbf{G}\ \neg arc)$
rejected order should not be archived

We do not add $\mathbf{G}(arc \Rightarrow \mathbf{G}\ \neg rej)$, because the rest of the rules determine that *arc* cannot occur before *rej*. The reason is, if the evaluation is failed, *rej* must be the first action to be executed (because of the next operator in $\mathbf{G}(evl \wedge ef \Rightarrow \mathbf{X}\ rej)$). Action *rej* cannot be executed if the evaluation is passed, and the execution sequence before the evaluation is fixed linearly. Thus it is impossible for *arc* to appear before *rej* on any traces.

The order relations graph is shown in Fig. A1 and the process template is shown in Fig. A2. The time to do the satisfiability check and extract the traces for this example was 8.21 s. Doing the trace evaluation and mining took 0.35 s.

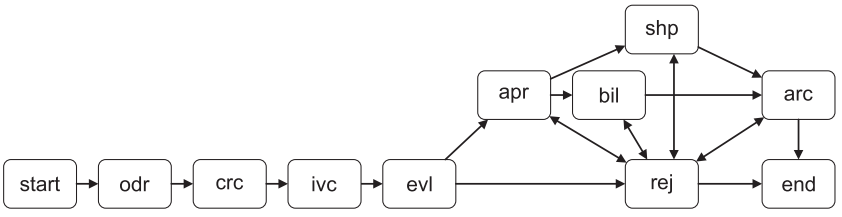

Fig. A1. Order relations graph of the order and purchase process.

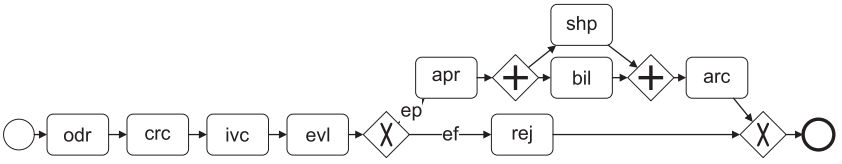

Fig. A2. Process template graph of the order and purchase process.

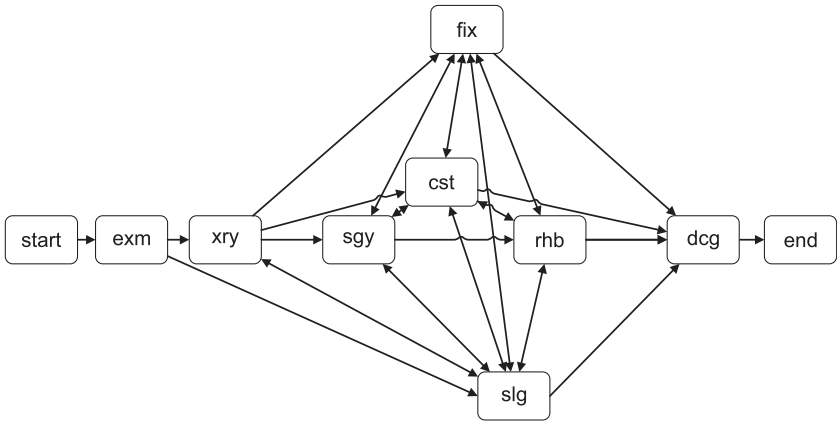

Fig. A3. Order relations graph of the hospital treatment process.

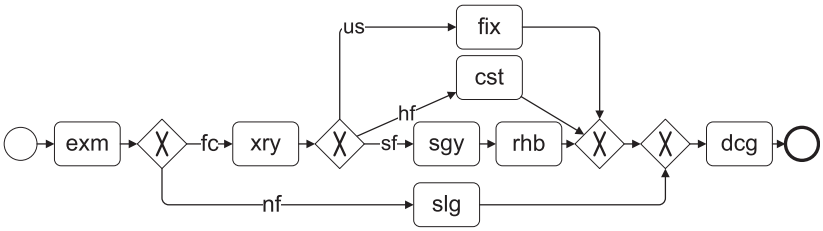

Fig. A4. Process template graph of the hospital treatment process.

A.2. The hospital treatment process

This example is a modified version of the hospital example used by van der Aalst et al. [30]. The original process model is designed in DECLARE, which is a workflow management system for flexible processes. Thus the structure of the original process model contains too many traces and is hard to be presented as a BPMN-like graph. So here we modify the process to adapt to our notation.

Initially, a specialist performs activity examination (*exm*). If the specialist diagnoses the absence of a fracture during examination, then x-ray (*xry*) is not necessary, thus sling (*slg*) is enough for the treatment. If x-ray results in a simple fracture (*sf*), then a cast (*cst*) is enough; if unstable fracture (*us*) is observed, then fixation (*fix*) is preferred; otherwise if the result is heavy fracture (*hf*), then the patient should be treated with surgery (*sgy*), and rehabilitation (*rhb*) is suggested after that. One of the treatments fixation, surgery, sling, and cast must be given to the patient before he can be discharged from hospital (*dcg*).

In DECLARE, data-flow is neglected, so even though they briefly mention the preferred conditions to make choices, the results are not reflected in their declarative model. By contrast, we identify those results and illustrate how we can handle ternary valued data-flow in this example. The set of actions and the set of results are shown respectively as below.

$Actions = \{exm, xry, fix, sgy, cst, slg, rhb, dcg\}$
exm: examination
xry: x-ray
fix: fixation
sgy: surgery
cst: cast
slg: sling
rhb: rehabilitation
dcg: discharge from hospital
$Results = \{ei, nf, fc, fi, sf, us, hf\}$
ei: examination is initial
nf: not fractured
fc: fractured
fi: fracture type is initial
sf: simple fracture
us: unstable fracture
hf: heavy fracture

To fit into our interpretation of the process, we define the following rules in LTL:

$\mathbf{G}(start \Rightarrow \mathbf{X}\ exm)$
first, examine the patient
$\mathbf{G}(exm \wedge fc \Rightarrow \mathbf{X}\ xry) \wedge \mathbf{G}(xry \Rightarrow fc)$
if fracture detected, scan x-ray
$\mathbf{G}(exm \wedge nf \Rightarrow \mathbf{X}\ slg) \wedge \mathbf{G}(slg \Rightarrow nf)$
if not fractured, sling is enough
$\mathbf{G}(xry \wedge sf \Rightarrow \mathbf{X}\ cst) \wedge \mathbf{G}(cst \Rightarrow sf)$
if simple fracture detected, execute cast
$\mathbf{G}(xry \wedge us \Rightarrow \mathbf{X}\ fix) \wedge \mathbf{G}(fix \Rightarrow us)$
if unstable fracture detected, execute fixation
$\mathbf{G}(xry \wedge hf \Rightarrow \mathbf{X}\ sgy) \wedge \mathbf{G}(sgy \Rightarrow hf)$
if heavy fracture detected, execute surgery
$\mathbf{G}(sgy \Rightarrow \mathbf{X}\ rhb) \wedge (sgy\ \mathbf{B}\ rhb)$
after surgery, the patient needs rehabilitation
$\mathbf{G}((fix \vee cst \vee sgy \vee slg) \Rightarrow \mathbf{F}\ dcg)$
one treatment must be given before discharge

Since data-flow is made explicit in our approach, conditions under which the affected actions take place are clear to the user. We do not claim that the process above is better than the original one, but we emphasise

the fact that data-flow helps the user to make decisions, as well as keeping the process model concise.

The order relations graph is shown in Fig. A3 and the process template is shown in Fig. A4. The time to do the satisfiability check and extract the traces for this example was 16.40 s. Doing the trace evaluation and mining took 0.08 s.

References

[1] Sarbanes-Oxley Act of 2002, US Public Law 107-204, 2002.
[2] S. Goedertier, J. Vanthienen, Designing compliant business processes with obligations and permissions, in: J. Eder, S. Dustdar (Eds.), Business Process Management Workshops Lecture Notes in Computer Science, vol. 4103, Springer, 2006, pp. 5–14.
[3] R. Lu, S. Sadiq, G. Governatori, Compliance aware business process design, in: BPM Workshops, Lecture Notes in Computer Science, vol. 4928, Springer, 2007, pp. 120–131.
[4] A. Förster, G. Engels, T. Schattkowsky, R. Van Der Straeten, Verification of business process quality constraints based on visualprocess patterns, in: TASE, IEEE Computer Society, 2007, pp. 197–208.
[5] A. Awad, M. Weidlich, M. Weske, Visually specifying compliance rules and explaining their violations for business processes, Journal of Visual Languages and Computing 22 (1) (2011) 30–55.
[6] D. Schleicher, T. Anstett, F. Leymann, D. Schumm, Compliant business process design using refinement layers, in: R. Meersman, T.S. Dillon, P. Herrero (Eds.), OTM Conferences (1), Lecture Notes in Computer Science, vol. 6426, Springer, 2010, pp. 114–131.
[7] S. Goedertier, J. Vanthienen, Compliant and flexible business processes with business rules, in: BPMDS, CEUR Workshop Proceedings, vol. 236, CEUR-WS.org, 2006.
[8] J. Yu, Y. Han, J. Han, Y. Jin, P. Falcarin, M. Morisio, Synthesizing service composition models on the basis of temporal business rules, Journal of Computer Science and Technology 23 (6) (2008) 885–894.
[9] J. Tian, K. He, C. Wang, H. Chen, An approach to generation of process-oriented requirements specification, Journal of Software Engineering and Applications 2 (1) (2009) 13–19.
[10] A. Awad, R. Goré, J. Thomson, M. Weidlich, An iterative approach for business process template synthesis from compliance rules, in: H. Mouratidis, C. Rolland (Eds.), Advanced information systems engineering, in: Proceedings of the 23rd International Conference, CAiSE 2011, London, UK, June 20–24, 2011. Lecture Notes in Computer Science, vol. 6741, Springer, 2011, pp. 406–421.
[11] V. Padmanabhan, G. Governatori, S. Sadiq, R. Colomb, A. Rotolo, Process modelling: the deontic way, in: APCCM '06: Proceedings of the 3rd Asia-Pacific Conference on Conceptual modelling, Australian Computer Society, Inc., Darlinghurst, Australia, 2006, pp. 75–84. 〈http://portal.acm.org/citation.cfm?id=1151864〉.
[12] M. Pesic, H. Schonenberg, W.M.P. van der Aalst, DECLARE: full support for loosely structured processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.
[13] M. Pesic, W.M.P. van der Aalst, A declarative approach for flexible business processes management, in: BPM Workshops, NCS, vol. 4103, Springer, 2006, pp. 169–180.
[14] M. Pesic, D. Bosnacki, W.M.P. van der Aalst, Enacting declarative languages using LTL: avoiding errors and improving performance, in: SPIN, Lecture Notes in Computer Science, vol. 6349, Springer, 2010, pp. 146–161.
[15] A. Pnueli, The temporal logic of programs, in: SFCS, IEEE Computer Society, Washington, DC, USA, 1977, pp. 46–57. http://dx.doi.org/10.1109/SFCS.1977.32.
[16] B. Hansson, An analysis of some deontic logics, Noûs 3 (4) (1969) 373–398. 〈http://www.jstor.org/stable/2214372〉.
[17] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[18] P. Wolper, The tableau method for temporal logic: an overview, Logique et Analyse 110–111 (1985) 119–136.
[19] P. Wolper, Temporal logic can be more expressive, Information and Control 56 (1983) 72–99.
[20] A.P. Sistla, E.M. Clarke, The complexity of propositional linear temporal logics, Journal of the ACM 32 (1985) 733–749, http://dx.doi.org/10.1145/3828.3837.
[21] E. Clarke, Model checking, in: S. Ramesh, G. Sivakumar (Eds.), Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, vol. 1346., Springer, Berlin/Heidelberg, 1997, pp. 54–56, http://dx.doi.org/10.1007/BFb0058022.
[22] Financial Services Commission, Guidelines on Anti-Money Laundering & Counter-Financing of Terrorism, 2007 〈http://www.natlaw.com/interam/jm/bk/sp/spjmbk00008.pdf〉.
[23] F. Baader, R. Peñaloza, Automata-based axiom pinpointing, Journal of Automated Reasoning 45 (2010) 91–129, http://dx.doi.org/10.1007/s10817-010-9181-2.
[24] V. Schuppan, Towards a notion of unsatisfiable cores for LTL, Fundamentals of Software Engineering (2010) 129–145. 〈http://www.springerlink.com/index/46l540451323v161.pdf〉.
[25] F. Hantry, M.-S. Hacid, Handling Conflicts in Depth-First-Search for LTL Tableau to Debug Compliance Based Languages, in: EPTCS (Ed.), FLACOS, 2011. 〈http://liris.cnrs.fr/publis/?id=5197〉.
[26] J. Bailey, P.J. Stuckey, Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization, in: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL05), Springer, 2005, pp. 174–186.
[27] I. Lynce, J. P. Marques-Silva, On computing minimum unsatisfiable cores, in: International Symposium on Theory and Applications of Satisfiability Testing, 2004, pp. 305–310 〈http://eprints.ecs.soton.ac.uk/12252/〉.
[28] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.
[29] J. Marques-Silva, Minimal unsatisfiability: models, algorithms and applications (invited paper), in: IEEE International Symposium on Multiple-Valued Logic, 2010, pp. 9–14. http://dx.doi.org/10.1109/ISMVL.2010.11.
[30] W.M.P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: balancing between flexibility and support, Computer Science - R&D 23 (2) (2009) 99–113.
[31] A. Weijters, W. van der Aalst, Rediscovering workflow models from event-based data using little thumb, Integrated Computer-Aided Engineering 10 (2001) 2003.
[32] H. Liang, J. Dingel, Z. Diskin, A comparative survey of scenario-based to state-based model synthesis approaches, in: J. Whittle, L. Geiger, M. Meisinger (Eds.), SCESM, ACM, 2006, pp. 5–12.
[33] I. Krka, From requirements to partial behavior models: an iterative approach to incremental specification refinement, in: G.-C. Roman, K.J. Sullivan (Eds.), SIGSOFT FSE, ACM, 2010, pp. 341–344.
[34] W.M.P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (9) (2004) 1128–1142.
[35] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, in: R. Hull, J. Mendling, S. Tai (Eds.), BPM, Lecture Notes in Computer Science, vol. 6336, Springer, 2010, pp. 276–293.
[36] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, M. Weske, Maximal structuring of acyclic process models, CoRR abs/1108.2384.
[37] R. Laue, J. Mendling, Structuredness and its significance for correctness of process models, Information Systems E—Business Management 8 (3) (2010) 287–307.
[38] R.M. McConnell, F. de Montgolfier, Linear-time modular decomposition of directed graphs, Discrete Applied Mathematics 145 (2) (2005) 198–209.
[39] M. Reichert, P. Dadam, Adeptflex-supporting dynamic changes of workflows without losing control, Journal of Intelligent Information Systems 10 (2) (1998) 93–129. (Special Issue on Workflow Management Systems) 〈http://dbis.eprints.uni-ulm.de/301/〉.
[40] C. Ellis, K. Keddara, G. Rozenberg, Dynamic change within workflow systems, in: Proceedings of Conference on Organizational Computing Systems, COCS '95, ACM, New York, NY, USA, 1995, pp. 10–21. http://doi.acm.org/10.1145/224019.224021. .
[41] M. Reichert, P. Dadam, A framework for dynamic changes in workflow management systems, in: Proceedings of the Eighth International Workshop on Database and Expert Systems Applications, 1997, pp. 42–48. http://dx.doi.org/10.1109/DEXA.1997.617231.
[42] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, G. Woodhull, Graphviz—open source graph drawing tools, in: Graph Drawing, 2001, pp. 483–484.
[43] Y. Lui, S. Müller, K. Xu, A static compliance-checking framework for business process models, IBM Systems Journal 46 (2) (2007) 335–362.
[44] Z. Milosevic, S. Sadiq, M. Orlowska, Translating business contract into compliant business processes, in: EDOC, IEEE Computer Society, 2006, pp. 211–220.
[45] K. Namiri, N. Stojanovic, Pattern-based design and validation of business process compliance, in: OTM Conferences (1), Lecture Notes in Computer Science, vol. 4803, Springer, 2007, pp. 59–76.
[46] S. Ghanavati, D. Amyot, L. Peyton, A systematic review of goal-oriented requirements management frameworks for business

process compliance, in: Fourth International Workshop on Requirements Engineering and Law, IEEE, 2011, pp. 25–34.
[47] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: EDBT, Lecture Notes in Computer Science, vol. 1377, Springer, 1998, pp. 469–483.
[48] W.M.P. van der Aalst, H.A. Reijers, A.J.M.M. Weijters, B.F. van Dongen, A.K.A. de Medeiros, M. Song, H.M.W.E. Verbeek, Business process mining: an industrial application, Information Systems 32 (5) (2007) 713–732.
[49] H. Schonenberg, J. Jian, N. Sidorova, W.M.P. van der Aalst, Business trend analysis by simulation, in: Advanced Information Systems Engineering, Proceedings of the 22nd International Conference, CAiSE 2010, Hammamet, Tunisia, June 7–9, 2010, Lecture Notes in Computer Science, vol. 6051, Springer, 2010. pp. 515–529.
[50] W.M.P. van der Aalst, M. Pesic, M. Song, Beyond process mining: from the past to present and future, in: Advanced Information Systems Engineering, Proceedings of the 22nd International Conference, CAiSE 2010, Hammamet, Tunisia, June 7–9, 2010, Lecture Notes in Computer Science, vol. 6051, Springer, 2010, pp. 38–52.
[51] K.G. Larsen, S. Li, B. Nielsen, S. Pusinskas, Scenario-based analysis and synthesis of real-time systems using uppaal, in: DATE, IEEE, 2010, pp. 447–452.
[52] A. Cimatti, M. Roveri, V. Schuppan, S. Tonetta, Boolean abstraction for temporal logic satisfiability, in: CAV, Lecture Notes in Computer Science, Springer, vol. 4590, 2007, pp. 532–546.