Karlsruhe Reports in Informatics 2016,10
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

A Practical Data-Flow Verification Scheme for Business Processes

Christine Tex, Jutta Mülle, Klemens Böhm

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

A Practical Data-Flow Verification Scheme for Business Processes

Christine Tex[a], Jutta Mülle[a], Klemens Böhm[a]

[a] *Karlsruhe Institute of Technology, KIT*
Institute for Program Structures and Data Organization
76131 Karlsruhe, Germany

Abstract

Data in business processes is becoming more and more important. Current standards for process-modeling languages like BPMN 2.0 which include the data flow reflect this. Ensuring the correctness of the data flow in processes is challenging. Model checking, i. e., verifying properties of process models, is a well-known technique to this end. An important part of model checking is the construction of the state space of the model. State-space explosion however typically is in the way of an effective verification. We study how to overcome this problem in our context by means of reduction. More specifically, we propose a reduction on the level of the process model. To our knowledge, this is new for the data-flow analysis of processes. To accomplish this, we specify regions relevant for the verification of properties describing the data flow. Our evaluation shows that our approach works well on real process models.

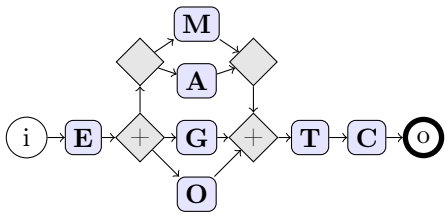
Keywords: Workflow Management, Business Process, Data-Flow Correctness, Property Verification, State-Space Reduction for High-Level Languages

1. Introduction

Recent business-process-modeling languages like BPMN 2.0 [1] do not only support the control flow but also the handling of data. Ensuring the correctness of the data flow in processes is challenging. Existing approaches tend to focus on data flow in languages like Petri Nets, e.g., [2]; formal verification techniques are mature for this type of language. In industry and elsewhere however, process designers use high-level modeling languages like BPEL, BPMN, EPC or OTX [3]. This paper features an approach for the data-flow correctness for such languages.

Example 1. *Our application scenario is commissioning of vehicles. Commissioning means configuring and testing the (electronic) components of a vehicle right after its production. The BPMN graph on the left of Figure 1 contains*

Email addresses: christine.tex@student.kit.edu (Christine Tex), jutta.muelle@kit.edu (Jutta Mülle), klemens.boehm@kit.edu (Klemens Böhm)



node	DI_O	DI_M	DO_O	DO_M
E	$\emptyset$	$\emptyset$	$\emptyset$	{DO1, DO2}
A	$\emptyset$	$\emptyset$	$\emptyset$	{DO2}
XOR	$\emptyset$	{DO1}	$\emptyset$	$\emptyset$

Figure 1: BPMN Workflow Graph with Data. Tasks not in the Table do not Use Any Data.

tasks which typically are part of the commissioning. For instance, a factory worker has to configure the transmission and to activate the anti-theft system. The transmission can either be manual, i. e., Task M does the configuration, or automatic (Task A). Task T activates the anti-theft system. Before the activation, a central computer needs to generate a master key (Task G), and it opens the connection to the specific control unit (Task O). The connection has to be closed before the process finishes (Task C). The configuration of the transmission and the activation of the anti-theft system require a running engine; Task E turns it on.

Data objects are part of the process, as input or output of tasks, of gateways or of the process itself. They can be optional or mandatory `DataInput` or `DataOutput` of a task.

Example 2. *The table in Figure 1 specifies the usage of two data objects DO1 and DO2 in the example. DO1 is mandatory* `DataOutput` *of Task E and mandatory* `DataInput` *of the XOR-node, DO2 is mandatory* `DataOutput` *of Tasks E and A.*

In this article, we use tables to specify the usage of data objects, rather than the "official" BPMN diagram elements. This is because the graphical notation of BPMN only allows to express the usage of a data object as input or output of a task or of an event, but not whether its use is optional or mandatory. BPMN diagrams also do not allow to specify the usage of a data object as input of a gateway.

Assigning data to the control-flow elements of a process specifies its flow. Correctness of data flow is a crucial issue. In this article, correctness of the data flow of a process model means that certain anti-patterns are guaranteed to not appear in the model. To verify the data flow, one typically has to inspect all execution paths where tasks use a data object.

Example 3. *Think of the execution path <E, A, G, O, T, C>. Here, the write done by Task E on DO2 gets lost. This is because A writes DO2 as well without*

reading it previously. With <E, M, G, O, T C> in turn, the write by E of DO2 does not get lost. This is the well-known data-flow error "Weakly Lost Data" [2].

Verification techniques like model checking require building the state space. They typically transform a high-level process model into a formal representation, e. g., Petri Nets. Then the construction of that space is feasible. However, there is state-space explosion [4], i. e., the space grows exponentially with the size of the model. This renders those techniques impractical for large or even medium-sized models. One problem is parallel branches in the model. Taking data objects into account makes this problem much more serious. This is because expanding the Petri Net to cover the data-flow semantics yields several parallel paths reflecting whether a certain object already exists or not.

To overcome state-space explosion, reduction techniques can be used. Reduction is feasible either (a) during construction of the state space or (b) on the level of the process model. In preliminary experiments by ourselves, methods of Category (a) do not do away with state-space explosion. Approaches of Category (b) in turn rely on the notion of relevance. A task (or a subprocess) is not relevant if there is no relationship to the property to be verified, i. e., this part of the model does not need to be expanded for verification. Relevance depends on the properties the model checker must prove. Thus, a core challenge is the definition of relevance in the data-flow context. One must identify a process abstraction where the data-flow semantics is representable and the block structure of the high-level process model is not lost. The representation should be abstract enough to give way to reduction for high-level languagues. One important contribution of this article is the specification of relevance functions for data-flow errors. These functions help to reduce the state space of the process model, to facilitate checking its data-flow correctness. Process models used in industry, in vehicle manufacturing in particular, often have a certain size that is in the way of model checking without any reduction. To accomplish the reduction, we classify the anti-patterns based on characteristics such as the kinds of usage of data objects. More specifically, we propose two classifications. The second one is a refinement of the first one, in that it takes the data object in question into account. A class of the first classification contains all anti-patterns of a data object, so that the relevant regions of the process are identical. The second classification reflects the kinds of usage of a data object by the anti-patterns. We hypothesize that its reduction tends to be more effective. To illustrate, the anti-pattern for "Weakly Lost Data" of Example 3 has the same data usage as the anti-pattern "Strongly Lost Data", see Table 2, and thus they are in the same class.

Example 4. *To verify the data flow of the process in Figure 1, one has to analyze all execution paths where tasks use a data object. Task E and the upper branch of the AND-node with the lower branch of the XOR-node containing Task A is a path/a possible execution containing DO2. It will be important to differentiate between the cases that a data object is used or not. An example for the first case is the path <E, M, G, O, T, C>. For this path, there does not exist*

any task after Task E writing DO2. To illustrate the second case, see the path <E, A, G, O, T, C>. The states representing this path typically are not needed to check for data-flow correctness, as will be explained.

Other contributions of this article are the following ones: Our goal is to enable checking of data-flow correctness for processes not only in BPMN, but also in other process-modeling languages. To this end, we propose an approach that uses an intermediate representation for processes with data. We have exemplarily implemented it for BPMN and OTX. Next, we provide an evaluation of the impact of our relevance functions on the verification of dataflow correctness of process models. We have used criteria like the size of the process models, the number of data objects used in the process, number of tasks that need a certain data object, the kind of usage, i.e., optional or mandatory, and several data-flow errors occurring in the process models. Our evaluation shows that our approach works well with processes in our domain, vehicle commissioning, which is comparable with other scenarios of vehicle manufacturing.

Our study leverages several recent research results. In particular, there is the insight that only parts of the process model typically are needed to verify a property [5], i. e., the general notion of relevance, and there exist proposals to detect data-flow errors in BPMN 2.0 processes using anti-patterns [6]. But relevance for data-flow constraints and the efficiency of verification approaches building on it have not been investigated yet. This however is necessary for data-flow verification of processes comparable in size and complexity to the ones examined here.

2. Notations for Process Models

This section lists the conceptual building blocks of this paper. We assume that the reader is familiar with the core notions that follow; our intention mainly is to introduce certain specifics and the notation used subsequently. The following also is rather broad, because we want to provide an approach for several high-level process model languages. To this end, we use an intermediate representation for high-level process models with data, namely an extended version of workflow graphs; the extension is that there is a separate table specifying the data usage, see the discussion of Figure 1. This will facilitate reducing the model to relevant parts to check data-flow correctness. Another motivation for this intermediate representation is to transform the process model into a representation compatible with existing model-checking tools, namely Petri Nets and CTL, see Subsections 2.4 and 2.5. Because the pruning step of our approach builds on the tree structure of a process and not on a graph structure, we propose a two-step transformation. The first step is a transformation to an RPST, see Subsection 2.2, which identifies hierarchically structured parts of the process. The second step is the transformation to a process tree. For the transformation of an RPST to a process tree, we propose a new algorithm in Subsection 2.3.

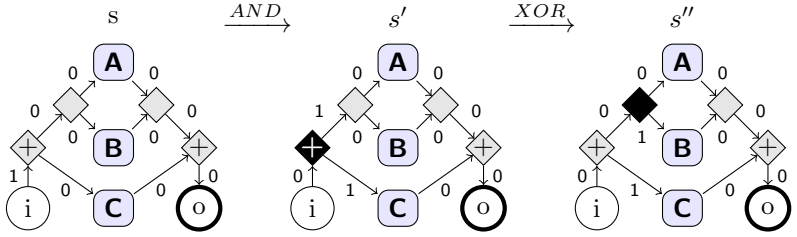


Figure 2: Three Different States s,s' and s'' of a Workflow Graph

2.1. Workflow Graph

We use the notion of workflow graph from [7]. N stands for its nodes, E for its edges, s for its state, and W for the graph itself. Here, a node can be a task node or an *AND*- or *XOR*-gateway. We use a BPMN-like graphical notation, i. e., gateways are represented as diamonds, where the symbol + stands for AND, and no decoration stands for XOR. A *workflow graph with data* is a graph W together with a set D_W of data objects and a function that maps each task to a set of objects that the task reads or writes optionally or mandatorily. We refer to these sets as `InputSet`s, respectively `OutputSet`s of the task. The function also maps a split XOR gateway to a set of data objects read mandatorily. XOR uses these objects to evaluate branching conditions. DO1 in Figure 1 serves as an example. The state of W is represented by tokens on the edges of the graph, as with Petri Nets.

Example 5. *The left graph of Figure 2 shows the initial state s of a workflow graph, i. e., with one token in the one outgoing edge and no token anywhere else. s changes to s' by executing the AND-node (in black in the second graph). This results in removing one token from each incoming edge and generating a token in each outgoing edge of the AND-node. Next, s' changes to s'' by executing the XOR-node (in black in the right graph of Figure 2).*

2.2. Refined Process Structure Tree

A *fragment* F is a subset of the edges of W that forms a connected subgraph of W with an entry and an exit node. *A fragment of* W i*s canonical* if it does not overlap with any other fragment of W. The set of all canonical fragments of W is the *RPST decomposition of W [7].* The corresponding parse tree is the *refined process structure tree (RPST) of* W, i. e., the tree of the canonical fragments such that the parent tree node of a canonical fragment F properly contains F. For an example see Figure 3 b); rectangles represent inner nodes and rectangles with round corners leaf nodes.

2.3. Process Tree

In a process tree, the inner nodes are the gateways of the RPST, and the leaves are the tasks. Transforming an RPST into a process tree is straightforward if the RPST does not have loops. With loops, one must consider the following points:

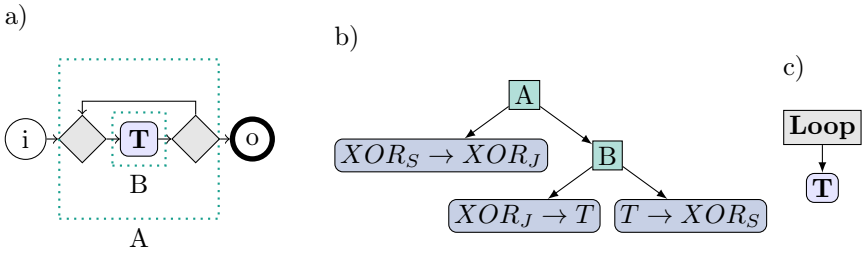


Figure 3: A Workflow Graph with a Loop (a) and the Corresponding RPST (b) and Process Tree (c).

An XOR-gateway with an edge from the split gateway XOR_S to the join gateway XOR_J represents a loop in an RPST, see Figure 3 for an example. Such a loop results in the fragments A and B and an RPST where the following holds for the RPST node A and one of its children B: $A.exit.type = B.exit.type = XOR_S$ and $A.entry.type = B.entry.type = XOR_J$ and A contains $XOR_S \rightarrow XOR_J$. See our Algorithm 1 that transforms an RPST including loops to a process tree. We first transform the root of the RPST into a SEQ node and then call Transform-RPST2PT(root). Creating a process-tree node (PT-node) includes connecting it to its parent-node.

Algorithm 1 Transform-RPST2PT(parent-node)

```
1:  for all children n of parent-node do
2:      if n.entry.type = n.exit.type ∈ {XOR, AND} then
3:          t ← n.entry.type
4:          if n has a child c with c.entry = n.entry and c.exit = n.exit then
5:              if n.contains(n.exit → n.entry) then
6:                  create PT-node of type LOOP
7:              else create PT-node of type SEQ
8:              end if
9:              Transform-RPST2PT(c)
10:         else
11:             create PT-node of type t
12:             Transform-RPST2PT(n)
13:         end if
14:     else
15:         if n.exit.type = TASK then
16:             create PT-node of type TASK
17:         end if
18:     end if
19: end for
```

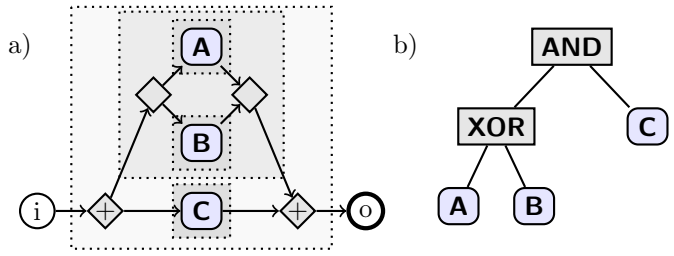


Figure 4: The Canonical Fragments for a Workflow Graph and the Resulting Process Tree

Example 6. *Figure 4 shows the set of all canonical fragments for a workflow graph W, one fragment per dotted box, and the corresponding process tree.*

2.4. Petri Nets

Our definition of Petri Nets is the one from [8]. P stands for its places, T for its transitions, and A for its arcs. Its state is a function $M : P \to \mathbb{N}_0$ that maps every place to a nonnegative number of tokens. The set of possible states from a start state of a Petri Net is its state space.

2.5. CTL

We use CTL [9] to express a property ϕ to be verified. Its syntax is as follows: An atomic proposition p is a CTL formula. If ϕ_1 and ϕ_2 are CTL formulas then $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\mathsf{AX}\ \phi_1$, $\mathsf{EX}\ \phi_1$, $\mathsf{AG}\ \phi_1$, $\mathsf{EG}\ \phi_1$, $\mathsf{AF}\ \phi_1$, $\mathsf{EF}\ \phi_1$, $\mathsf{A}[\phi_1\ \mathsf{U}\ \phi_2]$, $\mathsf{E}[\phi_1\ \mathsf{U}\ \phi_2]$ are CTL formulas. The operators always occur in pairs: a *path operator* (A or E) and a *state operator* (X, G, F or U). A means that the formula holds in *all* succeeding execution paths. E means that at least one execution path *exists* where the formula holds. X means that the formula holds in the next state, G means the formula holds in all succeeding states, F means that the formula holds at least in one succeeding state, and $[\phi_1\ \mathsf{U}\ \phi_2]$ means that ϕ_1 holds until ϕ_2 is reached.

3. Data-Flow Correctness of High-Level Process Models

High-level process models are graph-based, e. g., BPMN 2.0, or block-based like OTX. The latter one is a subset of the first one. In BPMN 2.0, data objects are `DataInputs` and `DataOutputs` of tasks, events and *XOR*-gateways. They can be either optional or mandatory. Additionally, it is possible to have several alternative `InputSets` or `OutputSets`. Splits like *XOR*-gateways can have data objects only as `DataInputs` that are mandatory. OTX, which we use in our evaluation, allows to specify block-based models with similar characteristics of data, i.e., data as optional or mandatory inputs or outputs of tasks, but not alternative sets of data objects. A verification approach for data flows of high-level languages that is sufficiently general must be able to deal with a language that comprises graph-based and block-based models. The approach

must not be restricted to a specific existing process-modeling language. A design decision of ours has been to take workflow graphs as starting point. However, as we need process models with data, and workflow graphs do not support data objects, the graph must be enhanced with data.

Definition 1 (Kinds of Data Object Usage)**.** *A flow node n* uses *a data object if n reads or writes the object. We discern between four kinds of data-object usage, namely reading or writing optionally or mandatorily.*

To define correctness of a data flow of a business process, we use so-called anti-patterns. If such an anti-pattern for a certain data object can be detected during the execution of a process, the respective error occurs in the data flow. A model checker allows to prove before process execution whether the process model fulfills an anti-pattern. Table 1 lists the data-flow errors that we support. The anti-patterns are defined as generic data-flow anti-patterns (DAP) in CTL, i.e., they are not yet instantiated for a certain process model and a certain data object. $I_M(d)$ means that d is a mandatory data input of a task, $I_O(d)$ that d is an optional input. $O_M(d)$ and $O_O(d)$ are analogous for outputs. $IS(f)$ and $OS(f)$ denote `InputSet`s and `OutputSet`s of flow element f. $exec(f)$ means that f is ready to be executed, i. e., the respective transition is activated. *term* denotes the termination of the process. If an anti-pattern is fulfilled, the corresponding error occurs in the process.

We refer to anti-patterns with their acronyms. For instance, *MD* stands for *Missing Data*. DAP is the set of all generic anti-patterns in Table 1. For verification, the anti-patterns must be instantiated with the data objects of W, respecting the semantics of the data usage. The instantiation result is CTL formulas on the state of the Petri Net that in turn is the result of the transformation of the workflow graph with data. For $a \in DAP$ and data object $d \in D_W$, $a(d)$ is the instantiation of a with d, i. e., a CTL formula that must be verified against the Petri Net. For example, *MD(d)* is the property that must be verified to check whether a *Missing Data* error with respect to d occurs in W.

Definition 2 (Set of properties for a data object d)**.** *For a data object d in a workflow graph W and a set of generic data-flow anti-patterns $A' \subseteq DAP$, the set of properties of d is $\phi_{A'}(d) := \{a(d) \mid a \in A'\}$, the set of all elements of A' instantiated with d.*

The transformation of a BPMN process model with data to Petri Nets, in order to do data-flow verification, is feasible, see [6]. In what follows, we propose an algorithm for data-flow verification respecting alternative and optional data usage for workflow graphs, $verify\_wg$. It works analogously to the approach for BPMN and consists of the following steps:

1. It transforms the workflow graph into a Petri Net and inserts subnets representing the data usage. These subnets reflect the execution semantics of the optional or mandatory usage of the objects as input or output of the

Table 1: BPMN 2.0 Generic Data-Flow Anti-Patterns for a Data Object d

Anti-Patterns	**Formalization**
1 MD: Missing Data	$\mathrm{E}\big(\neg O_M(d)\ \mathrm{U}\ I_M(d)\big)$
2 MOD: Missing Optional Data	$\mathrm{E}\big(\neg(O_O(d) \vee O_M(d))\ \mathrm{U}\ I_O(d))\big)$
3 SRD: Strongly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(\mathrm{A}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
4 WRD: Weakly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
5 ROD: Redundant Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
6 SLD: Strongly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(\mathrm{A}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)])\big)$
7 WLD: Weakly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}\big(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
8 LOD: Lost Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}\big(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
9 OLD: Optionally Lost Data	$\mathrm{EF}\big(\big(O_M(d) \vee O_O(d)\big) \wedge \big(\mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_O(d)])\big)\big)$
10 ID: Inconsistent Data	$\bigvee_{f \in \{E \cup T\} \wedge d \in OS(f)\}} \mathrm{EF}\big[exec(f) \wedge \bigvee_{f' \neq f \wedge (d \in IS(f') \vee d \in OS(f'))} exec(f')\big]$

nodes. Table 2 lists the data usage according to the CTL formula given for each data-flow error.

2. For each generic data-flow anti-pattern of Table 1 for each data object of the model, $verify\_wg$ does the following:
   (a) It instantiates the anti-pattern with the object usage.
   (b) A model checker verifies whether the anti-pattern holds in the resulting Petri Net.

4. Verification of High-Level Process Models with Data

To deal with state-space explosion, a general idea is to reduce the process model before transforming it to a Petri Net. The hope is that the state space of these reduced Petri Nets is small. To this end, it is necessary to identify regions of a process relevant for verification of a property, in our case data-flow properties. Subsection 4.1 introduces our scheme for practical data-flow verification, featuring pruning of irrelevant regions. Our core contribution is the specification of a relevance function for data-flow errors, see Subsection 4.2. (The algorithm to verify the data flow without taking relevance into account,

Table 2: Data Usage of the BPMN 2.0 Generic Data-Flow Anti-Patterns for a Data Object d

Anti-Patterns	**Data Usage**
1 MD: Missing Data	I_M, O_M
2 MOD: Missing Optional Data	I_O, O_M, O_O
3 SRD: Strongly Redundant Data	I_M, I_O, O_M
4 WRD: Weakly Redundant Data	I_M, I_O, O_M
5 ROD: Redundant Optional Data	I_M, I_O, O_O
6 SLD: Strongly Lost Data	I_M, I_O, O_M
7 WLD: Weakly Lost Data	I_M, I_O, O_M
8 LOD: Lost Optional Data	I_M, I_O, O_M, O_O
9 OLD: Optionally Lost Data	I_M, I_O, O_M, O_O
10 ID: Inconsistent Data	I_M, I_O, O_M

namely $verify\_wg$, has been in Section 3.) Our transformation of a reduced process tree to a workflow graph with data that incorporates relevance is in Subsection 4.3.

4.1. Practical Data-Flow-Error Detection

Algorithm 2 gives an overview of our approach. Our starting point is a workflow graph W with data usage for tasks. Φ is the properties to be verified, specified in CTL. The algorithm consists of the following steps: Line 1 is the transformation of W into a process tree. More specifically, there is a transformation to an RPST as a first step and from the RPST to a process tree as a second step. For the first step, we use the algorithm from [10]. It allows to decompose any workflow graph. The only assumption is that each node must be on a path from a source to a sink, which is not a restriction in reality. The transformation is necessary because the prune algorithm works on a process-tree representation. See Example 7. Our transformation preserves the usage of data objects by tasks and *XOR*-gateways.

Example 7 (Process-Tree Transformation)**.** *Figure 5 shows the process tree resulting from the transformation of our running example.*

Algorithm 2 DataFlowVerification(W)

1: $P \leftarrow$ graph2tree(W)
2: **for all** $\phi \in \Phi$ **do**
3: P' $\leftarrow$ prune_df(P, ϕ)
4: W' $\leftarrow$ tree2graph(P', ϕ)
5: verify_wg(W', ϕ)
6: output: data flow correct
7: or data-flow error detected
8: **end for**

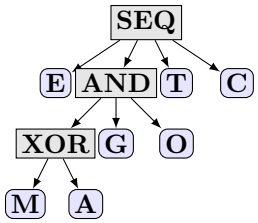


Figure 5: The Process Tree of the Commissioning Process of Figure 1.

In Line 2, the algorithm starts the reduction of W for every ϕ in Φ, i.e., the data-flow anti-patterns. The reduction algorithm *prune_df* in Line 3 relies on the relevance function, see Subsection 4.2. To apply $verify\_wg$, in Line 4 the reduced process tree needs to be retransformed into a workflow graph. See Subsection 4.3. Line 5 is the call of $verify\_wg$. This is the algorithm to check the correctness of the data flow, i.e., the data-flow anti-patterns, see Section 3. This verification is performed for each property. Line 6 returns whether the data-flow error is present or not.

4.2. Process-Model Reduction by Relevance Criteria

The question which regions of the process (subtrees of the process tree) are relevant depends on the properties to be verified. We observe that there are groups of properties where certain characteristics are identical. For all properties in one group, the relevance of process regions can be defined in the same way.

4.2.1. Characteristics of the Properties

Let W be a workflow graph with data and DAP the set of data-flow anti-patterns of interest. The derivations that follow rely on the following two important observations:

Observation 1. *Each data-flow anti-pattern refers only to one data object. Interdependencies between data objects do not exist in the data-flow anti-patterns.*

Observation 2. *There exist anti-patterns that only specify some kind of data-object usage.*

An example for Observation 2 is that *Missing Data* considers only mandatory usage of data objects, in contrast to optional usage. This is because the CTL formula of this anti-pattern in Table 1 only refers to the mandatory usage of data objects, i.e., to I_M and to O_M, see also Table 2.

4.2.2. Classification of the Properties

Based on these observations, we now specify two characteristics of properties, allowing to classify them.

Classification Φ_{DAP}. Taking Observation 1 into account, we define the classification Φ_{DAP} as follows. It contains $|D_W|$ classes, i.e., a class of equivalent anti-patterns for each data object.

Definition 3 (Classification of properties by data objects)**.** *For the set of data-flow anti-patterns DAP we define the classification*
$\Phi_{DAP} := \{\phi_{DAP}(d) \mid d \in D_W\}$.

For the definition of the sets of properties $\phi_{DAP}(d)$, see Definition 2. Note that Definition 3 relies on Observation 1: All instantiated data-flow anti-patterns belonging to one data object form a class. Therefore, Φ_{DAP} is a partitioning of the properties, without any overlap.

Classification $\Phi_{(DAP/\sim)}$. Taking Observation 2 into account, we define classes of equivalent anti-patterns based on data usage as follows.

Definition 4 (Classification of anti-patterns by data usage)**.** *$a, a' \in DAP$ are equivalent, written as $a \sim a'$, iff a has the same kinds of data usage as a'. We define $DAP/\sim$ as set of all equivalence classes of DAP with respect to $\sim$.*

Example 8 (Classification of anti-patterns)**.** Weakly Lost Data $\sim$ Weakly Redundant Data *holds, because both anti-patterns consider just flow nodes which have mandatory* `DataOutputs` *and optional or mandatory* `DataInputs`*, see Table 2. Analogously, all anti-patterns in {WLD, SLD, WRD, SRD} are equivalent according to Definition 4, defining the equivalence class* $[WLD]$.

The other classes can be established in the same way. We get five disjoint classes. I. e., $DAP/\sim$ splits DAP into five disjoint subsets. The five subsets are: [MD] = {MD}, [MOD] = {MOD}, [WLD] = {WLD, SLD, WRD, SRD}, [ROD] = {ROD}, and [OLD] = {OLD, LOD, ID}. This gives way to Definition 5:

Definition 5 (Classification of properties by data object and data usage)**.** *We define $\Phi_{(DAP/\sim)} := \{\phi_{\tilde{A}}(d) \mid d \in D_W, \tilde{A} \in (DAP/\sim)\}$.*

$\Phi_{(DAP/\sim)}$ contains $5 \cdot |D_W|$ classes. The factor 5 corresponds to the fact that there are five disjoint subsets.

Example 9 (Classification by data object and data usage)**.** *In our running example (see Figures 1 and 5), $D_W = \{DO1, DO2\}$. For $DO2$, there exist only operations where the write is mandatory. Therefore, $\phi_{\tilde{A}}(d)$ is identical for all $A \in \{[MD], [WLD], [ROD], [OLD]\}$, and there are only two equivalence classes regarding $DO2$ and its usage.*

4.2.3. Relevance for Data-Flow Anti-Patterns

A *relevance function* states whether a node in the process tree is necessary for verifying a $\phi \in \Phi$ or not, i.e., whether this node is part of an execution path of the process that renders an anti-pattern true. We also say that the node *influences the verification of the property.*

Definition 6 (Abstract relevance function)**.** *Let Φ be a classification of the properties to be verified.* relevant *is a function for a class $\phi \in \Phi$ and $n \in N$:*

$$relevant(\phi, n) = \begin{cases} true & n \text{ may influence the verification of a property} \\ & \text{contained in } \phi \\ false & otherwise \end{cases}$$

Now we propose the concrete relevance functions implied by Definitions 3 and 5.

Lemma 1 (Relevance function for $\Phi = \Phi_{DAP}$)**.** *Let $\phi_{DAP}(d) \in \Phi_{DAP}$ and $n \in N$. Then $relevant(\phi = \phi_{DAP}(d), n)$ is as follows:*

$$relevant(\phi, n) = \begin{cases} n \text{ uses } d & n \text{ is a task} \\ n \text{ has at least one relevant child} & n \text{ is an XOR or a Loop} \\ \quad \vee\ n \text{ reads } d \text{ mandatorily} & \\ n \text{ has at least one relevant child} & n \text{ is an AND or a SEQ} \end{cases}$$

Proof. Let $\phi_{DAP}(d) \in \Phi_{DAP}$ and $n \in N$. In a process tree, a node n is either an XOR, AND, SEQ, Loop or a task. While tasks use data objects in different ways (e.g., optionally vs. mandatorily), this distinction is not important here. So relevance is given iff the task uses the data object in question. An inner node (XOR, AND, Loop, or SEQ) of the tree is always relevant if any child of the node is relevant. Next, AND and SEQ nodes do not use data objects and therefore do not have to be considered further, in contrast to tasks, XORs and Loops. So, from now on, n is a Loop or an*XOR*-node. Loops and XORs have just one kind of data usage: They only can read a data object mandatorily. The properties in $\phi_{DAP}(d)$ take all kinds of data usage into account, including mandatory read. So n might influence the verification. □

Example 10. *In our running example (see Figures 1 and 5), $D_W = \{DO1, DO2\}$. I.e., $\phi_A(DO1)$ is a set of properties of $DO1$ in $\Phi_{DAP} = \{MD, MOD, SRD, WRD, ROD, SLD, WLD, LOD, OLD, ID\}$. $relevant(\phi_{DAP}(DO1), M)$ is false, because task M does not use the object $DO1$.*

On the other hand, $relevant(\phi_{DAP}(DO1), E)$ returns true, because E writes $DO1$ mandatorily. For the inner node XOR, $relevant(\phi_{DAP}(DO2), XOR)$ returns true as well. This is because A, which belongs to the subgraph of XOR, writes $DO2$ mandatorily.

Note that the relevance function in this case is independent from the anti-pattern. For the classification $\Phi_{(DAP/\sim)}$, coming up with the relevance is more complex. To take the usage characteristics of the anti-patterns into account, there will be a different function for each equivalence class. We exemplarily show the relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [WLD]$. For the definition of $[WLD]$, see Example 8. We use the following notation: For $n \in N$, $DI_O(n)$ is the set of all optional `DataInputs` of n. $DO_M(n)$ and $DI_M(n)$ are defined analogously. E.g., $d \in DI_O(n)$ if n has an `InputSet` which contains d as an optional `DataInput`.

Lemma 2 (Relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [WLD]$)**.** *Let $n \in N$ and $\tilde{A} = [WLD] \in \Phi_{(DAP/\sim)}$. Then $relevant(\phi = \phi_{\tilde{A}}(d), n)$ is as follows:*

$$relevant(\phi, n) = \begin{cases} d \in DI_O(n) \vee d \in DI_M(n) & n \text{ is a task} \\ \quad \vee\ d \in DO_M(n) & \\ n \text{ has at least one relevant child} & n \text{ is an XOR or Loop} \\ \quad \vee\ n \text{ reads } d \text{ mandatorily} & \\ n \text{ has at least one relevant child} & n \text{ is an AND or SEQ} \end{cases}$$

Proof. Let $n \in N$ and $\tilde{A} = [WLD] \in \Phi_{(DAP/\sim)}$. Analogously to the proof of Lemma 1, inner nodes are relevant if any of their children is relevant. Let n be a task, an XOR or a Loop. Differently from Lemma 1, where it only is important whether n does use d or not, we now must additionally consider the kinds of data usage. First, let n be a task. All anti-patterns in the class $[WLD]$ only take mandatory `DataOutputs` but both kinds of `DataInputs` into account. Therefore, in contrast to Lemma 1, n is not relevant if n only writes d mandatorily. Otherwise, it is relevant. Now let n be an XOR or a Loop. All anti-patterns in the class $\tilde{A}$ take mandatory `DataInputs` into account, see above. Thus the relevance in this case is analogous to the one in Lemma 1. □

For the other $\phi_{\tilde{A}}(d) \in \Phi_{(DAP/\sim)}$ the relevance functions are analogous, see Appendix Appendix A. Most of the functions only differ for the task case. With *Missing Optional Data* however, i.e., $\tilde{A}$ =[MOD] = {MOD}, there is a complication with the *XOR*-nodes: For this anti-pattern, mandatory `DataInputs` are not relevant. Thus, an *XOR*-node is not relevant because it only reads d.

Example 11. *In our running example, the XOR-node reads DO*1 *mandatorily, and none of its children, i.e., neither M nor A, uses DO*1*. Hence, the XOR-node is not relevant to verify* Missing Optional Data *with respect to DO*1*.*

4.2.4. The prune*-Algorithm*

Algorithm 3, borrowed from [5], is the reduction algorithm. It starts with the root of the process tree (Line 1) and then calls two subalgorithms (Lines 2 and 3). Algorithm 4 recursively traverses the process tree. It executes the relevance function on each node (Line 1). If the function evaluates to false, and the parent of the node is not an XOR-node, the algorithm deletes the node

(Line 9). Otherwise, the node is replaced by λ, i.e., an empty node (Line 7). We discuss the reason for this later. If the function evaluates to true, the node is kept (Lines 2 - 4). Algorithm 4 decides whether to prune a node, using our results from Lemma 1 or 2. Algorithm 5 trims the pruned tree, i.e., it replaces inner nodes with only one child by this child (Lines 2 - 4) – *prune* does not prune *XOR*-nodes in a straightforward way. This is not feasible for *XOR*-nodes in order to detect, say, *Strongly Lost Data* and *Weakly Lost Data* correctly.

In the next two examples we first illustrate the reduction of our running example, and then motivate a slightly adapted data usage that illustrates the pruning of *XOR*-gateways.

Example 12. *For our example from Figure 1 we get the tree in Figure 6 a) after the reduction for $\phi_{DAP}(d) = \phi_{DAP}(DO2)$. See Figure 1 for the data usage.*

Example 13. *Assume that for our example from Figure 1, task* `A` *does not write $DO2$ mandatorily but optionally. This does not have any impact on Example 12. But if we perform a reduction for $\phi_{[WLD]}(DO2)$, task* `A` *does not appear in the reduced process tree, so that it only contains the nodes* `i`*,* `E` *and* `o`*.*

In the following, $relevant_{DAP}$ denotes the function from Lemma 1 and $relevant_{(DAP/\sim)}$ the one from Lemma 2.

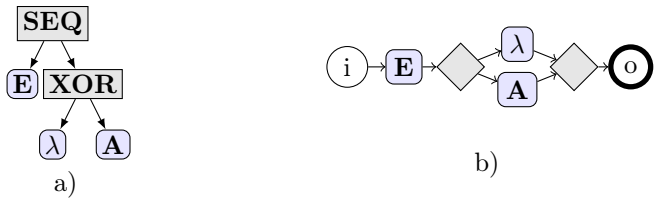


Figure 6: a) The Tree from Figure 5 after the Execution of Algorithm 3 $\phi_{DAP}(DO2)$. b) The Corresponding Workflow Graph.

Algorithm 3 prune(P, ϕ)

1: $n_0 \leftarrow$ get root of P
2: pruneNode(n_0, ϕ)
3: trimTree(n_0, ϕ)

Algorithm 4 pruneNode(n, ϕ)

```
1: if relevant(φ, n) then
2:     for all children n' of n do
3:         pruneNode(n', φ)
4:     end for
5: else
6:     if type of parent of n = XOR
   then
7:         replace n with λ
8:     else
9:         delete n
10:    end if
11: end if
```

Algorithm 5 trimTree(n, ϕ)

```
1: if |n.children| = 1 and type of n is
   not loop then
2:     connect the children of n to
   n.parent
3:     delete n
4:     trimTree(n.parent, φ)
5: else
6:     for all children n' of n do
7:         trimTree(n', φ)
8:     end for
9: end if
```

4.2.5. Discussion

We discuss whether to select Φ_{DAP} or $\Phi_{(DAP/\sim)}$ for Φ, i. e., using $relevant_{DAP}$ or $relevant_{(DAP/\sim)}$ for reduction. We are aware of two factors. First, the number of resulting reduced process trees is interesting, because the $verify\_wg$ algorithm must transform each different tree into a Petri Net representation as a prerequisite to run the model checker. So, we first derive the number of resulting reduced trees. We have to execute $prune(P, \phi)$ for every $\phi \in \Phi$. This yields a reduced tree for every $\phi \in \Phi$. As shown before, the number of classes in Φ_{DAP} (i.e., the elements of Φ_{DAP}) is five times smaller than the one for $\Phi_{(DAP/\sim)}$. Second, the size of the resulting reduced process trees determines the size of the state space; this in turn affects the costs of the model checking. Using $relevant_{(DAP/\sim)}$ has a higher potential of reduction resulting in smaller process trees. But this higher potential is not as high as one might expect. In particular, if no optional data usage occurs in W, there is no improvement with $relevant_{(DAP/\sim)}$. This is because all classes of equivalence fall together, see Lemma 3.

Lemma 3. *Let W be a workflow graph with data without any optional data usage. Then all anti-patterns in A' = {MD, SRD, WRD, SLD, WLD, ID} are equivalent.*

Proof. The anti-patterns in A' all take mandatory reads and writes into account. They only differ in the optional data usage, which does not occur in W. □

Lemma 4. *If W is a workflow graph with data without any optional data usage, it suffices to verify the anti-patterns in A' = {MD, SRD, WRD, SLD, WLD, ID}.*

Proof. All the anti-patterns not in A' can only evaluate to true if there is optional data usage in the workflow graph. This is not the case in W'. □

Lemmata 3 and 4 give way to a simplified verification for such workflow graphs.

Corollary 1. *If W is a workflow graph with data without any optional data usage, all anti-patterns belong to the same relevance class, i.e., the pruned workflow graph to be verified is the same for all data-flow anti-patterns.*

Next, think of an object d that is read and written mandatorily as well as optionally within W. First, consider class [WLD]. The anti-pattern *Weakly Lost Data* takes all kinds of data usage into account and uses the *Allpath*-Quantifier, see the CTL formula formalizing DAP 7 in Table 1. This means that all paths are relevant for this anti-pattern, and then no pruning is possible. The anti-pattern for *Redundant Optional Data* behaves analogously for all but mandatory `DataOutputs`. So pruning is not possible either in this case. I. e., the difference between the reduction for $\phi_{[WLD]}(d)$ and $\phi_{[ROD]}(d)$ is that, for the second property, the tasks with d as mandatory `DataOutput` can be pruned. However, the resulting reduced trees do not differ much in general, see Example 13. Thus, in this case the improvement from $relevant_{(DAP/\sim)}$ in general is not large.

4.3. Transformation of a Process Tree to a Workflow Graph

To analyze the data flow, we have developed a transformation from the reduced tree back into a workflow graph. This transformation is straightforward except for the handling of *XOR*-nodes. Let n be an *XOR*-node in the reduced process tree. Let $relevant_{DAP}$ be the relevance function used. We discern between two cases:

1. n has only λ-nodes (i.e., empty nodes resulting from the *prune*-Algorithm, see Algorithm 3) as children: This means that n does not have any relevant children, but reads d. Then a task with d as mandatory `DataInput` replaces n.
2. n has at least one λ as a child, but not all children are λ: In this case, we must keep one of the λs but can transform it to a task with no data usage.

If $relevant_{(DAP/\sim)}$ has been used, this differentiation is also valid except for the special treatment in the second case. This treatment is not necessary for $\tilde{A} \in \{[MD], [MOD], [ROD], [OLD]\}$, because these anti-patterns do not use an *Allpath*-quantifier.

Example 14. *To illustrate the data-flow verification steps for our running example, see Figure 6 b). It is the workflow graph corresponding to the tree in Figure 6 a). This graph is input of the verification algorithm, which then checks the properties in $\phi_{DAP}(d) = \phi_{DAP}(DO2) = \{MD(DO2), WLD(DO2), ..\}$. After that, our verification scheme goes on with the next class of properties.*

5. Evaluation

We quantify the impact of our approach on the number of states of the Petri Nets. From another perpective, we demonstrate that it is now possible to detect data-flow errors in processes where data-flow analysis has not been possible before.

Table 3: Statistics of the Processes Used in Our Evaluation.

process	♯ data objects	♯ tasks	♯ *AND*-splits	♯ *XOR*-splits	♯ states of Petri Net
P2	9	303	13	4	$>$ 6.8 mio.
P3	24	310	7	1	$>$ 6.3 mio.
PI4	43	582	25	2	$>$ 5.9 mio.
bpmn	1	7	0	1	444

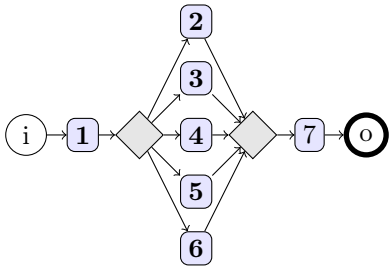
5.1. Preliminaries

Table 3 summarizes the statistics of the processes used. The first three are OTX processes. OTX is an international standard to describe test plans for vehicles. The processes are real commissioning processes. They have different numbers of tasks, gateways and data objects. In OTX, `DataInputs` and `DataOutputs` are mandatory. Thus, we use $relevant_{DAP}$ for the first three processes. To experiment with $relevant_{(DAP/\sim)}$ we use a BPMN process with optional data usage (the fourth process in the table), see Figure 7.

We first transform our processes to Petri Nets and count their states. We use the model checker [11] and stop the counting when 95 percent of the capacity of our 24 GB main memory is reached.

5.2. Results

Without reduction, the computation of the state space has not been possible for any of the OTX processes given. We now reduce the process with our approach and again compute the number of states. For the OTX processes, we introduce a differentiation between (1) the results for the data objects which are used in up to 13 tasks in the processes, i. e., used rarely, and (2) the results for the frequently used objects, used by nearly half of the tasks.



node	$DI_O(n)$	$DI_M(n)$	$DO_O(n)$	$DO_M(n)$
1	$\emptyset$	$\emptyset$	$\emptyset$	{DO}
2	$\emptyset$	$\emptyset$	{DO}	$\emptyset$
3	$\emptyset$	$\emptyset$	{DO}	$\emptyset$
4	$\emptyset$	$\emptyset$	{DO}	$\emptyset$
5	$\emptyset$	$\emptyset$	{DO}	$\emptyset$
6	$\emptyset$	$\emptyset$	$\emptyset$	{DO}
7	$\emptyset$	{DO}	$\emptyset$	$\emptyset$
XOR	$\emptyset$	{DO}	$\emptyset$	$\emptyset$

Figure 7: The BPMN Process Used for Evaluation as a Workflow Graph.

Table 4: Results for the first three processes from Table 3.

frequency	# tasks	# AND splits	# XOR splits	# states of Petri Net	data-flow errors
14	1	0	0	8	RD
17	2	0	0	12	-
2	2	0	0	13	RD, LD
15	3	0	0	16	-
1	3	0	0	16	RD
1	3	0	0	18	RD, LD
4	4	0	0	20	-
5	5	0	0	24	-
1	6	0	0	31	LD
1	6	0	0	28	-
1	7	3	0	798	-
1	8	1	0	118	-
1	9	0	0	40	-
1	10	0	0	44	-
1	10	1	0	254	-
1	11	1	0	98	-
1	12	0	0	52	-
1	13	2	1	1.007	RD
1	13	1	0	138	-

Table 5: Results for the Frequently Used Data Objects in the OTX Processes.

process	♯ tasks	♯ *AND*-splits	♯ *XOR*-splits	♯ states of Petri Net
P2	152	10	1	$>$ 7.3 mio.
P2	149	9	1	$>$ 8.0 mio.
P3	114	6	0	$>$ 8.8 mio.
P3	131	6	0	$>$ 9.0 mio.
PI4	239	21	2	$>$ 6.2 mio.
PI4	204	20	1	$>$ 6.1 mio.

Table 6: Results for the BPMN Process with Optional Data Usage.

anti-pattern class	♯ tasks	♯ *AND*-splits	♯ *XOR*-splits	♯ states of Petri Net
MD	4	0	0	21
SLD	4	0	1	84
ROD	5	0	1	153

5.2.1. Case (1)

In every OTX process, all objects except for two are used by at most 13 tasks. See Table 4 for results with $relevant_{DAP}$. Note that each of our processes W results in $|D_W|$ trees, i.e., in $|D_W|$ processes to be verified. Each row represents a set of processes ("frequency" is the number of processes in the set) with a similar number of tasks, gateways and Petri Net states and the same data-flow errors detected. *RD* means that we have detected *Strongly* and *Weakly Redundant Data*, *LD* the same for *Strongly* and *Weakly Lost Data*.

The numbers of tasks in the resulting processes are much smaller than the original ones. Before reduction, the processes contained at least 300 tasks, after reduction at most 13. The reduced processes also are less complex, because they all contain at most three splits. Before they had up to 27. The number of Petri Net states has decreased by orders of magnitude. Another very important point is that we have detected errors that have not been detected before.

5.2.2. Case (2)

To begin with, the two data objects used frequently are used for procedure testing exclusively. In other words, these objects could have been omitted without

confining functionality. We report on this rather unimportant case nevertheless, because it is a setting where reduction is not expected to help much, i. e., reveals the limits of our approach. As Table 5 shows, nearly half of the tasks stay in the process after reduction, and the complexity of the process in terms of the number of split gateways is not reduced. This in turn leads to a large state space which is not computable in total. However, we could compute at least two million states more than without reduction.

5.2.3. BPMN Process with Optionality

In the process bpmn, just one object occurs. Table 6 shows the results with $relevant_{DAP}$. Each row stands for the reduction for an anti-pattern class. We do not consider the class *OLD* because we do not expect any reduction, see Section 4.2. *MOD* is not listed either. The reason is that, while instantiating the CTL-formula for *Missing Optional Data*, the program discovers that there is not any optional `DataInput` in the process. Thus, the process will never fulfill the anti-pattern.

For the other classes, we perform our reduction. It is significant for the number of control-flow elements and also for the size of the state spaces. The reduction for *MD* removes all *XOR*-paths where the only object of the process is only used as an optional `DataOutput`. This is the case for the paths including just Tasks 2 to 5. The result is that only one path, the one with Task 6, remains. Additionally, *Missing Data* does not use an *Allpath*-quantifier, and therefore a task could replace the *XOR*-gateway. The result is a process with only four sequentially ordered tasks. All in all, the state space had 444 states; after reduction the number is between 21 to 153.

5.2.4. Discussion

The effects of our reduction approach are significant. Where no analysis of the data-flow has been possible so far, we now have detected several data-flow errors. Speaking more generally, we have shown the potential of a reduction with an anti-pattern based definition of relevance.

6. Related Work

Correctness of data-flow in processes. There are other approaches for data-flow verification, e. g., [12], which uses BPEL processes, or [2] using Workflow Nets. In principle, they could profit from our verification scheme as well. This is possible as long as they use model checking for data-flow correctness. [13] features an approach to verify complex business processes in BPMN notation with multiple instances, exception handling and cancellation activities. They transform BPMN into an enhanced Petri net formalism, so-called RECATNets, which allows to handle the semantics of those complex BPMN concepts. However, data flow is only handled for data-based XOR-decision gateways. In contrast to our approach, they do not provide a mechanism for general data-flow correctness. Other definitions of correctness of data in process models exist as well. These definitions consider, say, properties restricting the allowed values of data objects [14] but not the data flow defined with data usage, or life-cyles of data objects with states [15, 16, 17]. To apply our scheme to these correctness definitions, one has to define respective properties as, e.g., anti-patterns and then to come up with a specific formula for relevance. For instance, this seems to be feasible for life-cycles of data objects. For artifact-centric business-process models, [18] proposes an integration of artifacts into the model by pre- and post-conditions. They model life-cycles of data objects and numerical data with these conditions using the constraint programming paradigm, and provide correctness checking regarding reachability and so-called weak termination. In contrast to our approach they rely on artifact-centric process models, but do not focus on control-flow centric models like BPMN.

Transformation of process models. Transformations from high-level process models to Petri Nets are described in [19]. An overview on transformations from block-based to graph-based process models gives [20]. These transformations do not comprise the data flow. Therefore, we use such transformations for the control-flow and have enhanced it, respectively. Abstraction of process models is a related topic, see for example [21, 22, 23].

Reduction for enabling process-model verification. Reduction on the level of the state space is orthogonal to the one of the process model, see [5]. In our evaluation we actually have used such a technique, namely stubborn set reduction [24], which is available in LoLA. However, this does not work for the data-flow anti-patterns, as the state space remains too large. In [5], a reduction of a high-level process model to relevant regions confines the state space. Relevance depends on the properties to be proven. Thus, one challenge behind our work has been the specification of relevance in the data-flow context, which has not been investigated yet. [5] also uses a scenario with properties called resource conditions. This means that a task needs certain data. Including this additional information into the relevance function could reduce the Petri Net further. However, resource conditions only concern one issue of data semantics in process models and thus of data-flow correctness. Other work on the level of

the process model, e. g., [25, 26, 27], mostly covers structural conditions of the model and is not applicable to more complex properties expressed in temporal logic. 1 [14, 28] speeds up the verification of data-aware properties by considering the states of a variable. It does so by abstracting from the possible values of a data object in a preprocessing step. This work is orthogonal to ours, and a combination seems feasible.

7. Conclusions

This paper has proposed a data-flow verification scheme for business processes. To deal with state-space explosion, it has reduced the process models. To do so, we have relied on the insight that only parts of the model typically are required to verify a property, i. e., use the notion of relevance. We have derived relevance for data-flow constraints and have studied the efficiency of respective verification approaches. Our evaluation has demonstrated that our approach works nicely for processes of realistic size.

References

[1] Object Management Group, Business Process Model and Notation, V2.0, OMG Specification, URL `http://www.omg.org/spec/BPMN/2.0/PDF`, 2011.

[2] N. Trčka, W. v. d. Aalst, N. Sidorova, Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows, in: CAISE, 2009.

[3] I. International Organization for Standardization, Road vehicles – Open Test sequence eXchange format (OTX), ISO, Geneva, Switzerland, 2012.

[4] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.

[5] R. Mrasek, J. Mülle, K. Böhm, A new verification technique for large processes based on identification of relevant tasks, Information Systems 47 (2014) 82–97.

[6] S. von Stackelberg, S. Putze, J. Mülle, K. Böhm, Detecting Data-Flow Errors in BPMN 2.0, Open Journal of Information Systems OJIS 1 (2) (2014) 1–19.

[7] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, Data & Knowledge Engineering 68 (9) (2009) 793–818.

[8] W. M. P. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits, Systems and Computers 08 (01) (1998) 21–66.

[9] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263.

[10] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified Computation and Generalization of the Refined Process Structure Tree, in: Web Services and Formal Methods, no. 6551 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 25–41, 2011.

[11] LoLA a Low Level Analyser, URL `http://www.informatik.uni-rostock.de/tpp/lola/`, last visited on 05/2015.

[12] W.-J. van den Heuvel, A. Elgammal, O. Turetken, Using Patterns for the Analysis and Resolution of Compliance Violations, Coop.Inf.Systems 21 (1).

[13] A. Kheldoun, K. Barkaoui, M. Ioualalen, Specification and Verification of Complex business Processes - A High-Level Petri Net-Based Approach, in: BPM2015, no. 9253 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 55–71, 2015.

[14] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, P. Dadam, On Enabling Data-Aware Compliance Checking of Business Process Models, in: J. Parsons, M. Saeki, P. Shoval, C. Woo, Y. Wand (Eds.), Conceptual Modeling – ER 2010, no. 6412 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 332–346, 2010.

[15] A. Meyer, M. Weske, Weak Conformance between Process Models and Synchronized Object Life Cycles, in: ICSOC 2014, no. 8831 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 359–367, 2014.

[16] A. Meyer, A. Polyvyanyy, M. Weske, Weak Conformance of Process Models with respect to Data Objects, in: Services und ihre Komposition, vol. 847, CEUR-WS.org, 74–80, 2012.

[17] A. Meyer, M. Weske, Extracting Data Objects and their States from Process Models, in: EDOC'2013 - 13th Enterprise Distributed Object Computing Conference, 27–36, 2013.

[18] D. Borrego, R. Gasca, M. Gómez-López, Automating correctness verification of artifact-centric business process models, Information and Software Technology 62 (2015) 187–197.

[19] N. Lohmann, E. Verbeek, R. Dijkman, Petri Net Transformations for Business Processes – A Survey, in: K. Jensen, W. M. P. van der Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, no. 5460 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 46–63, 2009.

[20] J. Mendling, K. B. Lassen, U. Zdun, et al., Transformation strategies between block-oriented and graph-oriented process modelling languages, in: Multikonferenz Wirtschaftsinformatik, vol. 2, 297–312, 2006.

[21] A. Polyvyanyy, S. Smirnov, M. Weske, Business Process Abstraction, in: M. Rosemann, J. vom Brocke (Eds.), Handbook on Business Process Management, vol. 1, Springer Berlin Heidelberg, 149 – 166, 2010.

[22] S. Smirnov, H. Reijers, M. Weske, A Semantic Approach for Business Process Model Abstraction, in: CAISE, vol. 6741 of *LNCS*, Springer Berlin Heidelberg, 497 – 511, 2011.

[23] A. Meyer, M. Weske, Data Support in Process Model Abstraction, in: ER, vol. 7543 of *LNCS*, Springer-Verlag Berlin Heidelberg, 292–306, 2012.

[24] K. Schmidt, Stubborn Sets for Standard Properties, in: S. Donatelli, J. Kleijn (Eds.), Application and Theory of Petri Nets 1999, no. 1639 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 46–65, 1999.

[25] W. M. P. v. d. Aalst, A. Hirnschall, H. M. W. Verbeek, An Alternative Way to Analyze Workflow Graphs, in: A. B. Pidduck, M. T. Ozsu, J. Mylopoulos, C. C. Woo (Eds.), Adv. Inform. Systems Engineering, no. 2348 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 535–552, 2002.

[26] H. Lin, Z. Zhao, H. Li, Z. Chen, A novel graph reduction algorithm to identify structural conflicts, in: System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on, 2002.

[27] B. F. van Dongen, W. M. P. van der Aalst, H. M. W. Verbeek, Verification of EPCs: Using Reduction Rules and Petri Nets, in: O. Pastor, J. F. e. Cunha (Eds.), Adv. Inf. Syst. Eng., no. 3520 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 372–386, 2005.

[28] L. T. Ly, D. Knuplesch, S. Rinderle-Ma, K. Göser, H. Pfeifer, M. Reichert, P. Dadam, SeaFlows Toolset – Compliance Verification Made Easy for Process-Aware Information Systems, in: P. Soffer, E. Proper (Eds.), Information Systems Evolution, no. 72 in Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 76–91, 2011.

Appendix A. Relevance Functions for the Equivalence Classes of $\Phi_{(DAP/\sim)}$

This appendix defines the relevance functions for the classes in $\Phi_{(DAP/\sim)}$ not defined in Section 4. Their proofs are analogous to the proof of Lemma 1 in Section 4 .

Lemma A1 (Relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [MD]$)**.** *Let $n \in N$ and $\tilde{A} = [MD] \in \Phi_{(DAP/\sim)}$. Then $relevant(\phi = \phi_{\tilde{A}}(d), n)$ is as follows:*

$$relevant(\phi, n) = \begin{cases} d \in DI_M(n) \vee d \in DO_M(n) & n \text{ is a task} \\ n \text{ has at least one relevant child} & n \text{ is an XOR or Loop} \\ \quad \vee\ n \text{ reads } d \text{ mandatorily} & \\ n \text{ has at least one relevant child} & n \text{ is an AND or SEQ} \end{cases}$$

Lemma A2 (Relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [MOD]$)**.** *Let $n \in N$ and $\tilde{A} = [MOD] \in \Phi_{(DAP/\sim)}$. Then $relevant(\phi = \phi_{\tilde{A}}(d), n)$ is as follows:*

$$relevant(\phi, n) = \begin{cases} d \in DI_O(n) \vee d \in DO_M(n) & n \text{ is a task} \\ \quad \vee d \in DO_O(n) & \\ n \text{ has at least one relevant child} & n \text{ is an XOR or Loop} \\ & \text{or AND or SEQ} \end{cases}$$

Lemma A3 (Relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [ROD]$)**.** *Let $n \in N$ and $\tilde{A} = [ROD] \in \Phi_{(DAP/\sim)}$. Then $relevant(\phi = \phi_{\tilde{A}}(d), n)$ is as follows:*

$$relevant(\phi, n) = \begin{cases} d \in DI_O(n) \vee d \in DO_O(n) & n \text{ is a task} \\ \quad \vee d \in DI_M(n) & \\ n \text{ has at least one relevant child} & n \text{ is an XOR or Loop} \\ \quad \vee\ n \text{ reads } d \text{ mandatorily} & \\ n \text{ has at least one relevant child} & n \text{ is an AND or SEQ} \end{cases}$$

Lemma A4 (Relevance function for $\phi_{\tilde{A}}(d)$ with $\tilde{A} = [OLD]$)**.** *Let $n \in N$ and $\tilde{A} = [OLD] \in \Phi_{(DAP/\sim)}$. Then $relevant(\phi = \phi_{\tilde{A}}(d), n)$ is as follows:* $relevant(\phi, n) = relevant(\phi_{DAP}(d), n)$