Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows

Nikola Trčka, Wil M.P. van der Aalst, and Natalia Sidorova

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{n.trcka,w.m.p.v.d.aalst,n.sidorova}@tue.nl

Abstract. Despite the abundance of analysis techniques to discover control-flow errors in workflow designs, there is hardly any support for data-flow verification. Most techniques simply abstract from data, while data dependencies can be the source of all kinds of errors. This paper focuses on the discovery of data-flow errors in workflows. We present an analysis approach that uses so-called “anti-patterns” expressed in terms of a temporal logic. Typical errors include accessing a data element that is not yet available or updating a data element while it may be read in a parallel branch. Since the anti-patterns are expressed in terms of temporal logic, the well-known, stable, adaptable, and effective model-checking techniques can be used to discover data-flow errors. Moreover, our approach enables a seamless integration of control flow and data-flow verification.

1 Introduction

A *Process-Aware Information System* (PAIS) is a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models [6]. Examples of PAISs are workflow management systems, case-handling systems, enterprise information systems, etc. Many of these systems are driven by explicit process models, i.e., based on a process model a system is configured that supports the modeled process. In this paper, we primarily focus on the analysis of the models used to configure workflow management systems [2,9,11,21]. However, our approach is also applicable to other PAISs.

In the last 15 years, many analysis techniques have been developed to analyse workflow models [2]. Most of these techniques focus on verification, i.e., on the discovery of design errors. Although many process representations have been used or proposed, most researchers are using Petri nets as a basic model [1,20]. The flow-oriented nature of workflow processes makes the Petri net formalism a natural candidate for the modeling and analysis of workflows. Most workflow management systems provide a graphical language that is close to Petri nets, or that has a token-based semantics making a (partial) mapping to Petri nets relatively straightforward. Industrial languages like Business Process Modeling

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 425–439, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Notation (BPMN), extended Event-driven Process Chains (eEPCs) and UML activity diagrams, are examples of languages that can be translated to Petri nets.

Unfortunately, lion's share of attention has only been devoted to control flow while other perspectives such as data flow and resource allocation have been completely ignored. Existing analysis techniques typically check for errors such as deadlocks, livelocks, etc. while abstracting from data and resources. In most cases the abstraction from resource information is unavoidable, as resources are often external and dynamic in nature. The role of data in the workflow is however important: Routing choices in a workflow are typically based on data, which makes the control flow data dependent. Moreover, the data flow can be erroneous itself. Another limitation of the most of the existing workflow verification approaches is the way they communicate to the user: they are not configurable, and it is not always clear what types of errors they capture (the details are typically hidden in the verification algorithms).

To address some of the limitations of existing approaches, we propose a new analysis framework based on (a) workflow nets with data, (b) temporal logic, and (c) "anti-patterns". A *WorkFlow net with Data* (WFD-net) is a special type of a Petri net, with a clear start and end point and annotations related to the handling of data (a task can *read*, *write*, or *destroy* a particular data element). Assuming a WFD-net representation, we define several *anti-patterns* related to the data flow. The term "anti-patterns" was coined in 1995 by Andrew Koenig [12]. He stated that "An anti-pattern is just like pattern, except that instead of solution it gives something that looks superficially like a solution, but isn't one" [12]. The goal of anti-patterns is to formally describe repeated mistakes such that they can be recognized and repaired. In this paper, we use the temporal logic CTL^* (and its subclasses CTL and LTL) to formalize our anti-patterns. This formalization can be used to discover the occurrence of such anti-patterns in WFD-nets by standard model-checking techniques [4]. Although not elaborated on in this paper, the same techniques can be used to define correctness notions related to the control flow and check these in an integral way.

An example of an anti-pattern is *DAP 1: Missing data*. This anti-pattern describes the situation where some data element needs to be accessed, i.e. read or destroyed, but either it has never been created or it has been deleted without having been created again. This property can be expressed in both CTL and LTL. Hence, given a WFD-net it can be easily checked using standard model checkers.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 introduces WFD-nets. This representation is used in Section 4 to define a comprehensive set of data-flow anti-patterns. The formalization of these anti-patterns is given in Section 5. Section 6 concludes the paper.

2 Related Work

Since the mid-nineties, many researchers have been working on workflow verification techniques [1,16]. It is impossible to give a complete overview of the

related work here (see [3] for references). Therefore, we only mention the work directly relevant to this paper, namely verification approaches in which control and data flow are both taken into account for verification.

The importance of data-flow verification in workflow processes was first mentioned in [15]. There, several possible errors in the data flow are identified, like, e.g., the missing and the redundant data error, but no means for checking these errors is provided. Later, [18] conceptualized the errors from [15] using UML diagrams, and gave supporting verification algorithms. This work was further extended and generalized in [19]. None of these approaches consider data removal. The exact details of the erroneous scenarios are not always clear, being hidden in the verification algorithms, and good diagnostics are missing. Moreover, the methods are not adaptive enough, as new properties cannot be easily added to the checks.

In [8], a model called *dual workflow nets* is proposed, that can describe both the data flow and the control flow. The notion of classical soundness from [1] is extended to support the case when data flow can influence control flow. No explicit data correctness properties are considered.

The ADEPT$_{\text{flex}}$ tool [14] supports a limited set of checks for data-flow correctness. The focus is entirely on dynamic changes in workflow models.

The work closest to ours is [7]. There, model checking is used to verify business workflows, from both the control- and data-flow perspective. The underlying workflow language is UML diagrams as opposed to the Petri net approach taken in this paper. Only a few data data correctness properties are identified and no systematic classification is presented. Data can only be read or written, but not destroyed. Finally, [7] only considers LTL model-checking while several of our anti-patterns are not expressible in LTL.

In the field of software verification model checking have been successfully used to discover program bugs that are caused by, e.g., non-initialized or dead variables [17]. In this, totally different, application domain, concurrency issues are rarely treated and systematic classification of errors is missing.

3 Workflow Nets with Data

Workflow nets (WF-nets) are commonly used as a basic representation for workflow processes [1]. A WF-net is a Petri net with one unique source place and one unique sink place such that all nodes are on a path from the source place to the sink place. The transitions in a WF-net represent tasks. A WF-net is instantiated for a particular case by putting a token on the source place. The completion of this instance is denoted by a token on the sink place. WFD-nets extend WF-nets with data elements and define four relationships between tasks and these data elements. First, a task may *read* a particular data element. This data element is thus expected to have a value before the task is executed. Second, a task may *write* (to) a particular data element. This means that this data element gets a new value. If it did not have a value yet, it is created; otherwise it is overwritten. Third, a task may *destroy* a data element, leaving it with no value. Finally, a task may use a particular data element in its *guard* (optional).

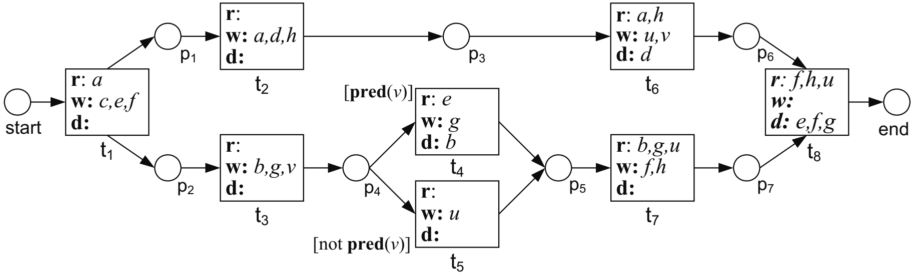

Fig. 1. A WFD-net

We only consider data elements that are case-related, i.e., that belong to an individual process instance and cannot be shared among different cases and/or different processes. The techniques of this paper, however, can be applied to support complete data interplay, if all processes are modeled and combined into one (huge) WFD-net. In addition, we assume workflows to start from an empty data state; starting with some existing data can easily be modeled with an artificial start task.

The following definition introduces *Workflow nets with Data* (WFD-nets).

Definition 1 (WFD-net). *A tuple* $\langle P, T, F, \mathcal{D}, \mathcal{G}_\mathcal{D}, \text{Read}, \text{Write}, \text{Destroy}, \text{Guard}\rangle$ *is a* Workflow net with data (a WFD-net) *iff:*

- $\langle P, T, F\rangle$ *is a WF-net, with* places P*,* transitions T *and* arcs F*;*
- $\mathcal{D}$ *is a set of* data elements*;*
- $\mathcal{G}_\mathcal{D}$ *is a set of* guards *over* $\mathcal{D}$*;*
- $\text{Read} : T \to 2^\mathcal{D}$ *is the* reading data labeling function*;*
- $\text{Write} : T \to 2^\mathcal{D}$ *is the* writing data labeling function*;*
- $\text{Destroy} : T \to 2^\mathcal{D}$ *is the* destroying data labeling function*; and*
- $\text{Guard} : T \to \mathcal{G}_\mathcal{D}$ *is the* guarding function*, assigning guards to transitions.* □

Note that a WFD-net is just an annotated WF-net; its formal semantics will be given in Section 5 using the concept of unfolding to a WF-net.

Fig. 1 shows an example of a WFD-net. There are 10 data elements ($a, \ldots, h$, u, and v), and these elements are linked to tasks in the process. Task t_6, e.g., reads from data elements a and h, writes to u and v, and destroys d. Thus: $\text{Read}(t_6) = \{a, h\}$, $\text{Write}(t_6) = \{u, v\}$, $\text{Destroy}(t_6) = \{d\}$, and $\text{Guard}(t_6) = \mathit{true}$ (i.e., no guard). If one ignores the read, write, destroy, and guard annotations, Fig. 1 is a WF-net with source place *start* and sink place *end*. All cases start with task t_1 and end with task t_8. In-between, t_2 and t_6 are executed in sequence and this is done in parallel with the lower process fragment that starts with t_3 and ends with t_7. In-between t_3 and t_7 either t_4 or t_5 is executed. This choice depends on the evaluation of $\mathsf{pred}(v)$; if this predicate evaluates to true, t_4 is selected, otherwise t_5.

WFD-nets can be seen as an abstraction from notations deployed by popular modeling tools. To illustrate this we show in Fig. 2 the Protos model corresponding to the WFD-net from Fig. 1. Protos (Pallas Athena) uses a Petri-net-based

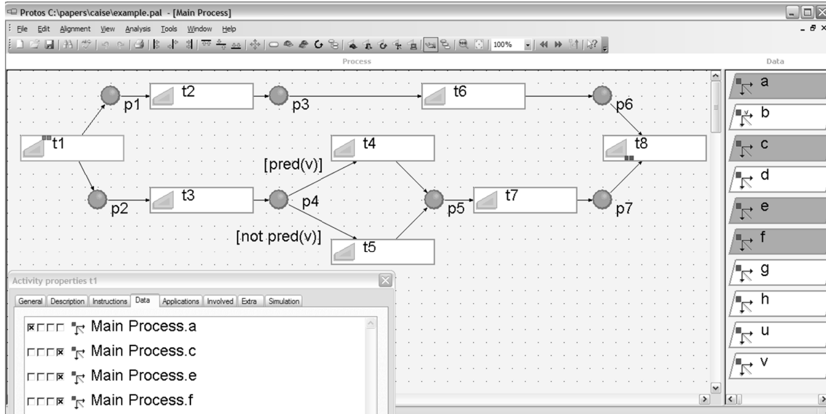

Fig. 2. A Protos model showing both the control flow and data flow

modeling notation and is a widely-used business process modeling tool. It is used by more than 1500 organizations in more than 20 countries and is the leading business process modeling tool in the Netherlands. Like most other tools it allows for the modeling of both control flow and data flow. The left-hand side of Fig. 2 shows the control flow while the right-hand side shows the different data elements. The colors (different shades of grey in this case) show the relationships between t_1 and these data elements, and the bottom window of Fig. 2 shows the nature of these relationships.

As illustrated by Fig. 2, the language used by Protos is close to Definition 1. Other popular notations such as the Business Process Modeling Notation (BPMN), extended Event-driven Process Chains (eEPCs), UML activity diagrams, etc. also allow for the modeling of both control flow and data flow. In fact, the basic idea to link data elements to tasks originates from IBM's Business Systems Planning (BSP) methodology developed in the early eighties. Here a so-called *CRUD matrix* is used showing Create, Read, Update, and Delete relationships between tasks and data elements. The Read relationship in a CRUD matrix is similar to the Read function and the Delete relationship is similar to the Destroy function in Definition 1. The Update relationship is similar to the Write function, but may also refer to a combination of read and write. The Create relationship can be seen as the first write action for a data element. In Protos a variant of the CRUD matrix is used and the basic relations are Mandatory, Created, Deleted, and Modified. Other tools use other variants. However, all of these operations can be translated into the primitives given in Definition 1. Hence, the applicability of the results presented in the remainder extends to other notations (BPMN, eEPCs, etc.) and variants of the CRUD matrix.

Soundness [1] is the mostly used correctness notion for workflows. The basic idea is that the process cannot deadlock or livelock and it is always still possible to terminate properly. However, the classical soundness notions do not consider data. This is serious limitation. For example, the workflow design shown in Fig. 1

is sound but has some serious design flaws when considering the data annotations. For example, data element b may be destroyed in task t_4 while it is needed in the following task t_7 for reading. To identify such problems we use so-called *data-flow anti-patterns.*

4 Data-Flow Anti-Patterns

In this section we introduce data-flow anti-patterns and explain them using the example WFD-net shown in Fig. 1. For the sake of readability, when saying "data element d is read" in the descriptions of anti-patterns, we actually mean "data element d is read or used for the evaluation of a guard". Evaluating predicate *pred* on data element v in Fig. 1 thus will be interpreted as reading v by transitions t_4 and t_5.

DAP 1 (Missing Data). *This anti-pattern describes the situation where some data element needs to be accessed, i.e. read or destroyed, but either it has never been created or it has been deleted without having been created again.*

In Fig. 1, data elements a and b are missing. Note that a needs to be read immediately by the first task, although it has not been created yet. Data element b is created by t_3, but it can be destroyed by t_4 before it reaches t_7 that needs to read it.

Unlike some other anti-patterns we will present later, we do not introduce a strong and a weak variant for missing data depending on the fact whether we will certainly miss a data element, or we miss it only at some execution paths that might be choosen. We require that data should be present independently of the choices made in the workflow—the absence of data necessary for an action indicates a flaw in the workflow.

DAP 2 (Strongly Redundant Data). *A data element is strongly redundant if there is a writing activity after which in all possible continuations of the execution this data element is never read before it gets destroyed or the workflow execution is completed.*

In Fig. 1, data elements c and d are strongly redundant. Task t_1 creates c but it is never read in the workflow, while task t_2 creates d and t_6 destroys d without reading it.

DAP 3 (Weakly Redundant Data). *A data element is weakly redundant if there is* some *execution scenario in which it is written but never read afterwards, i.e. before it is destroyed or the workflow execution is completed.*

If a data element is strongly redundant (DAP 2), it is also weakly redundant (DAP 3), while the opposite does not hold in general. Consider data element e in Fig. 1. It is created by t_1 and it is only read by t_4. In case t_5 and not t_4 is chosen, e remains unread, and hence it is weakly redundant. On the other hand,

if t_5 is chosen, e is read between its creation and destruction, and therefore e is not strongly redundant.

Strongly redundant data indicates in most situations a real flaw in the workflow. Weakly redundant data can in principle refer not to a flaw but to a design decision aimed e.g. at the uniformization/simplification of data requests (asking all clients to provide data $d_1, \ldots, d_k$, while d_k is of interest only for the clients with a particular value of d_1) or at the improvement of the performance (computing some weakly redundant data element d in parallel to some other activity whose result will make it clear whether d is needed afterwards or not; in case d is needed, it is immediately available, and it is ignored otherwise).

DAP 4 (Strongly Lost Data). *A data element is strongly lost if there is a writing activity after which in all possible continuations of the execution this element gets overwritten without having been read first.*

In Fig. 1, element f is strongly lost, since t_1 writes to f, t_7 rewrites it, and f cannot be read in between.

DAP 5 (Weakly Lost Data). *A data element is weakly lost if there is an execution sequence in which it is overwritten without been read first.*

Strongly lost data (DAP 4) implies weakly lost data (DAP 5) but, in general, not the other way around. In Fig. 1, g and h are weakly lost. Task t_3 writes to g, then g may be overwritten by t_4, after which g is read by t_7. In case t_5 is chosen instead of t_4, g is read by t_7 without having been overwritten. The example of h shows a concurrency-related instance of this anti-pattern. In case of the execution sequence $t_1t_2t_6t_3t_4t_7t_8$, t_2 writes to h, t_6 reads it, t_7 writes again and t_8 reads h. If t_6 is scheduled to be executed later, and the execution sequence is $t_1t_2t_3t_4t_7t_6t_8$, t_2 writes to h, then t_7 rewrites it, and only then h is read. Note that in the latter case both t_6 and t_8 use the value of h produced by t_7.

Strongly lost data normally indicates a real flaw in the workflow, while weakly lost data may be a design decision, but may also be a flaw. Examples where weakly lost data may be an instance of a normal behavior are, e.g., reading some client's data (address, telephone number, etc.), which might remain possible along the whole workflow. The fact that they are updated (overwritten) without ever having been read is then a normal scenario.

DAP 6 (Inconsistent Data). *Data is inconsistent if a task is using this data while some other task (or another instance of the same task) is writing to this data or is destroying it in parallel.*

In Fig. 1, u is inconsistent since t_5 and t_6 may write to u in parallel and it is not clear which version of u will be used by t_7 and t_8. Data element v is also inconsistent, as t_6 might change its value before or after the predicate **pred** is evaluated. Inconsistent data normally indicates a real flaw in the workflow.

The following anti-patterns are related to data removal. They should be seen more as efficiency drawbacks rather than strict correctness problems. These anti-patterns are especially important for scientific workflows, where data is often

large and its unnecessary storage should be avoided, while automatic garbage collection is rare.

DAP 7 (Never destroyed). *A data element is never destroyed if there is a scenario in which it is created but not destroyed afterwards.*

For example, a is never destroyed after its creation by t_2, which indicates the possibility of leaving garbage by the workflow.

DAP 8 (Twice Destroyed). *A data element is twice destroyed if there is a scenario in which it is destroyed twice in a row without having been created in between.*

This anti-pattern is similar to the strongly lost data error but concerns data deletion. It can be seen as a special instance of DAP 1.

DAP 9 (Not Deleted on Time). *A data element is not deleted on time when there is a task that reads it without destroying it, and after this task the data element is never read again in the workflow, independently of the choices made.*

For example, t_7 is the last (and the only) task reading g, but g is deleted later, by t_8. Thus g is not deleted on time.

5 Formalization and Implementation

After introducing the anti-patterns in an informal manner, we now show that these patterns can be formalized and supported by standard model checking tools. We first introduce CTL^* and its subclasses LTL and CTL. Then we provide a translation of WFD-nets to Kripke structures to facilitate the verification of the desired temporal properties, and we provide formalizations for the anti-patterns formulated in Section 4. Finally, we discuss how the approach can be supported by existing tools.

5.1 Temporal Logic CTL^*

CTL^* [4] is a powerful (state-based) temporal logic combining linear time and branching time modalities. It is typically defined on Kripke structures, so we introduce this model first.

Definition 2. *A* Kripke structure *is a tuple* $(S, A, \mathcal{L}, \rightarrow)$ *where* S *is a finite set of states,* A *is a non-empty set of* atomic propositions*,* $\mathcal{L} : S \rightarrow 2^A$ *is a* (state) labeling function*, and* $\rightarrow \subseteq S \times S$ *is a* transition relation. □

If $(s, s') \in \rightarrow$, then there is a *step* from s to s', then also written as $s \rightarrow s'$. For a state s, $\mathcal{L}(s)$ is the set of atomic propositions that *hold* in s.

A *path* from s is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $s = s_0$, and either $s_k \rightarrow s_{k+1}$ for all $k \in \mathbb{N}$, or there exists an $n \geq 0$, such that $s_k \rightarrow s_{k+1}$ for all $0 \leq k < n$, $s_n \not\rightarrow$, and $s_k = s_{k+1}$ for all $k \geq n$. For a path $\pi = s_0, s_1, s_2, \ldots$ and some $k \geq 0$, π^k denotes the path $s_k, s_{k+1}, s_{k+2}, \ldots$.

We now define the syntax of CTL^*.

Definition 3. *The classes Φ of* CTL^* state formulas *and Ψ of* CTL^* path formulas *are generated by the following grammar:*

$$\begin{aligned} \phi &::= \ a \mid \ \neg\phi \ \mid \ \phi \wedge \phi \mid \ \mathsf{E}\psi \\ \psi &::= \ \phi \ \mid \ \neg\psi \ \mid \ \psi \wedge \psi \mid \ \mathsf{X}\psi \mid \ \psi \,\mathsf{U}\, \psi \end{aligned}$$

with $a \in A$, $\phi \in \Phi$, and $\psi \in \Psi$. □

Validity of CTL^* formulas is defined as follows.

Definition 4. *We define when a* CTL^* *state formula ϕ is* valid *in a state s (notation: $s \models \phi$) and when a* CTL^* *path formula ψ is* valid *on a path π (notation: $\pi \models \psi$) by simultaneous induction as follows:*

- *$s \models a$ iff $a \in \mathcal{L}(s)$;*
- *$s \models \neg\phi$ iff $s \not\models \phi$;*
- *$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$;*
- *$s \models \mathsf{E}\psi$ iff there exists a path π from s such that $\pi \models \psi$;*
- *$\pi \models \phi$ iff s is the first state of π and $s \models \phi$;*
- *$\pi \models \neg\psi$ iff $\pi \not\models \psi$;*
- *$\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$;*
- *$\pi \models \mathsf{X}\psi$ iff $\pi^1 \models \psi$; and*
- *$\pi \models \psi \,\mathsf{U}\, \psi'$ iff there exists a $j \geq 0$ such that $\pi^j \models \psi'$, and $\pi^k \models \psi$ for all $0 \leq k < j$.* □

A formula $\mathsf{X}\psi$ says that ψ holds *next*, i.e. in the second state of a considered path. A formula $\psi \,\mathsf{U}\, \psi'$ says that, along a given path, ψ holds (at least) *until* ψ' holds. We standardly write $\mathsf{F}\psi$ for $\top \,\mathsf{U}\, \psi$ ("In the future ψ" or "ψ will hold *eventually*"), $\mathsf{G}\psi$ for $\neg\mathsf{F}\neg\psi$ ("Globally ψ" or "ψ holds *always* along a path"), and $\mathsf{A}\psi$ for $\neg\mathsf{E}\neg\psi$ ("ψ holds along *all* paths"). The combinators AG and EF can then be interpreted as "in all states" and "in some state" respectively.

The complexity of checking CTL^* formulas is linear in the size of the model but exponential in the size of the formula. We define two most popular (syntactic) restrictions of CTL^* that allow for more optimal verification. A CTL^* state formula of the form $\mathsf{A}\psi$, where ψ is a path formula containing no state formulas, is a *linear temporal logic* (LTL) formula. A CTL^* state formula in which every sub-formula of the type $\psi \,\mathsf{U}\, \psi'$ is prefixed by an A or E quantifier, is a *computational tree logic* (CTL) formula. The complexity of LTL model checking is the same as of CTL^*, but the advantage is that LTL formulas can be checked on-the-fly [4]. The complexity of CTL model checking is linear in both the size of the model and the size of the formula, and thus lower than for CTL^* [4]. As we will see later, all our correctness properties belong to either the LTL or the CTL subset (or both)The reason we work with CTL^* is to have a common framework, and to be allowed to (temporarily) jump outside of the restricted domain when rewriting one formula to another.

5.2 Unfolding of WFD-Net

Since we use a state-based logic, the *states* of a Kripke structure representing the behavior of a WFD-net, should include information necessary for the formalization of our anti-patterns, namely what happens with the data when some transition is executed. This information is lost if we just build the reachability graph of a WFD-net—e.g. we can see that two transitions writing to a data element d can be enabled at the same time, but we cannot see whether they can be executed at the same time.

Preprocessing. To include the information about the data operations into the states, we use a preprocessing step that converts a WFD-net into a WF-net, while keeping the original structure intact. This step consists of the following smaller steps:

1. We *split* every transition t into its start t_s and its end t_e connected by a place p_t. A token on p_t means that transition t is being executed.
2. To capture the restrictions on the behavior due to guards, we add a *"guard layer"* to our net: For every *predicate* pred appearing in some guard we introduce places $\mathsf{pred}_{\mathsf{true}}$ and $\mathsf{pred}_{\mathsf{false}}$. A token on $\mathsf{pred}_{\mathsf{true}}$ indicates that the predicate is evaluated to true for the current set of data values. A token on $\mathsf{pred}_{\mathsf{false}}$ means that pred evaluates to false.
3. For every transition t with a *guard* pred in the WFD-net, we add an arc from $\mathsf{pred}_{\mathsf{true}}$ to t_s and an arc back from t_s to $\mathsf{pred}_{\mathsf{true}}$ to our preprocessed net. This self-loop makes sure that t is executed only when its guard is evaluated to true. For the guard $\neg\mathsf{pred}$ we add the arcs to the place $\mathsf{pred}_{\mathsf{false}}$ instead of $\mathsf{pred}_{\mathsf{true}}$.
4. A *change* of the value of a data element d that appears in a predicate pred may potentially change the evaluation of pred. We reflect that by assuming that every transition t writing to d might change the value of pred (or not). Therefore, we split t_e into three transitions: two to represent possible changes of the predicate value (from true to false and from false to true), and one leaving the predicate value unchanged.[1]

Please note that in case the transition changes data items related to k predicates, it will be in general split into 3^k transitions.

Fig. 3 illustrates the preprocessing for transition t with a guard $\mathsf{pred1(c)}$ writing to data element b. We assume that b is used in some predicate $\mathsf{pred2(b)}$, guarding some other transition(s) of the workflow. Places $\mathsf{pred1(c)}_{\mathsf{true}}$, $\mathsf{pred1(c)}_{\mathsf{false}}$ (not shown in the figure), $\mathsf{pred2(b)}_{\mathsf{true}}$ and $\mathsf{pred2(b)}_{\mathsf{false}}$ are added to represent the values of the predicates. The transition is split into the start transition t_s, controlled by place $\mathsf{pred1(c)}_{\mathsf{true}}$, and transitions $t_{e-true-false}$, changing the value of the predicate $\mathsf{pred2}$ from true to false, $t_{e-false-true}$, changing the value of the predicate from false to true, and t_e leaving the value unchanged.

[1] In this paper we assume that predicates do not depend on each other; our method, however, can be easily extended to support dependencies.

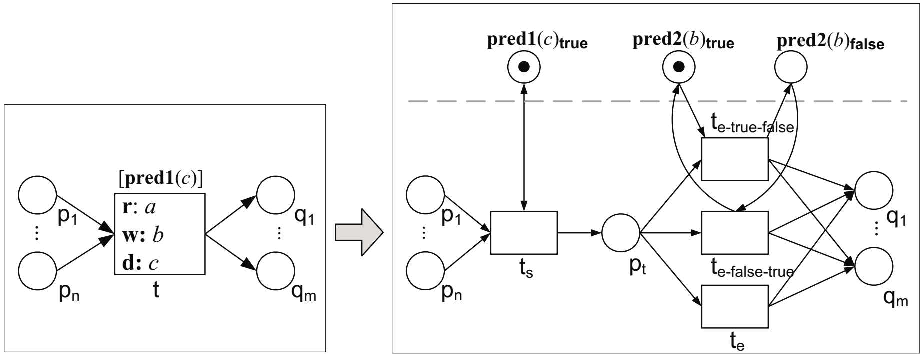

Fig. 3. Decomposition of a transition in a WFD-net

We make an arbitrary choice assuming that all pred_{true} places initially have a token and pred_{false} places not. We can afford making an arbitrary choice since the errors related to the use of undefined data for the valuation of guards is captured by DAP 1 and will be signaled as an error, in case it takes place.

Building the Kripke structure. The Kripke structure is in fact an extended reachability graph of the preprocessed net. The states of the Kripke structure are states (markings) of the reachability graph and the transition relation is the reachability relation. We define the *set of atomic propositions* $A = \{p \geq i \mid p \in P, i \in \mathbb{N}\}$ to express properties of markings ($p \geq i$ means that place p holds at least i tokens). The *labels* of states map the markings to the sets of atomic propositions as follows: for some $p \in P$ and $i \in \mathbb{N}$, $(p \geq i) \in \mathcal{L}(m)$ iff $m(p) \geq i$.

For the sake of readability, we introduce some abbreviations. We write $p = i$ for $p \geq i \wedge \neg(p \geq i + 1)$. To directly formulate that some *transition* t *is executing*, we write $\mathsf{exec}(t)$ instead of $p_t \geq 1$. That the workflow is in its *final state* is denoted term, defined by $(\mathsf{end} = 1 \wedge \bigwedge_{p \in P \setminus \{\mathsf{end}\}} (p = 0))$. To represent the fact that a data element $d \in \mathcal{D}$ is being *read* by some transition, either as its input or for evaluating a guard, we write $\mathsf{r}(d)$, abbreviating thus $\bigvee_{t:d \in \mathrm{Read}(t) \cup \mathbf{data}(\mathrm{Guard}(t))} \mathsf{exec}(t)$. The constructs $\mathsf{w}(d)$ and $\mathsf{d}(d)$ are defined similarly.

We will use a *convention* that the order of operations within a transition is fixed as first read, then write and after that destroy, which e.g. implies that transition t_8 in Fig. 1 first reads f and only after that destroys it, i.e. here there is no attempt to read a destroyed data element.

Example. We use a simplistic example to show that the addition of the guard layer reduces the number of false positives and false negatives, compared to the analysis on the net without it. Consider the WFD-net from Fig. 4. If guards are ignored while generating the behavior, d' will be reported missing in t_4. This is a false negative, as t_4 can never be enabled—t_2 can only be chosen when $\mathsf{pred}(\mathsf{d})$ is false, and the value of the predicate remains the same when it is evaluated at t_4. On the other hand, a soundness check on the net with the guard layer will

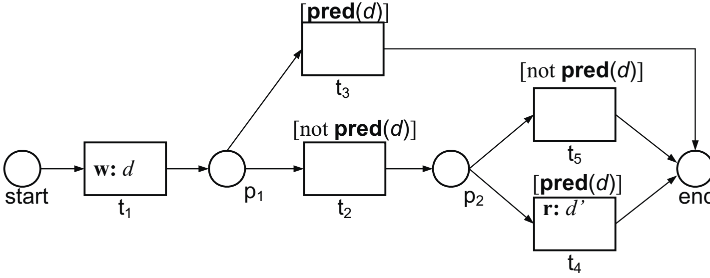

Fig. 4. Data can influence reachability

correctly report that transition t_4 is dead, while the check on the control flow would result in a false positive, saying that the net is sound.

5.3 Formalization of Anti-patterns

We explain the formalization process for some of the anti-patterns in detail, and we merely provide the corresponding CTL^* formulas for the rest.

DAP 1: Missing Data. A data element d is missing if there is an execution path along which no writing to d happens until reading d or destroying d takes place. This can be expressed by $\mathsf{E}[\neg\mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d))]$. A data element d is also missing if d first get destroyed and then no writing takes place until d is read of destroyed, which can be captured by $\mathsf{EF}[\mathsf{d}(d) \wedge (\neg\mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d)))]$. The disjunction of these two expressions results in the formalization given in Table 1.

DAP 2: Strongly Redundant Data. A data element is strongly redundant if there is a path leading to a writing to d (i.e. $\mathsf{EF}[\mathsf{w}(d) \wedge \ldots]$) such that in all possible continuations of this path no reading takes place until the workflow terminates or d get destroyed ($\mathsf{AX}[\neg\mathsf{r}(d)\,\mathsf{U}\,(\mathsf{term} \vee (\mathsf{d}(d) \wedge \neg\mathsf{r}(d)))]$). We need X here because we want to impose $\neg\mathsf{r}(d)$ restriction starting from the next state only, not from the state where the writing in question takes place—reading there would precede the writing, according to our convention. This convention is also the reason for including $\neg\mathsf{r}(d)$ in the conjunction $\mathsf{d}(d) \wedge \neg\mathsf{r}(d)$.

The formalization of *DAP 3 Weakly Redundant Data* differs from its strong counterpart by one letter only: the A requirement is removed, since it is sufficient to have one path showing the undesired behavior. The principle of formulating *DAP 4* is the same as for *DAP 2*, the principle of formulating *DAP 5*, *DAP 7* and *DAP 8* is the same as for *DAP 3*.

DAP 6: Inconsistent Data. A data element d is inconsistent if some transition t that changes d and some transition t' that uses d can be executed at the same time, captured by $\bigvee_{t \in T : d \in \mathsf{change}(t)} \mathsf{EF}[(\mathsf{exec}(t)\ \wedge \bigvee_{t' \neq t : d \in \mathsf{use}(t')} \mathsf{exec}(t'))]$, or if two or more instances of transition t changing d can be executed in parallel, captured by $\bigvee_{t \in T : d \in \mathsf{change}(t)} \mathsf{EF}[p_t \geq 2]$. Here $\mathsf{change}(t)$ stands for the set $\{d \mid$

Table 1. Formalization of anti-patterns for a data element d

Anti-pattern	Formalization
DAP 1 *Missing Data*	$\mathsf{E}[(\neg \mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d))) \vee \mathsf{F}[\mathsf{d}(d) \wedge (\neg \mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d)))]]$
DAP 2 *Strongly Redundant Data*	$\mathsf{EF}[\mathsf{w}(d) \wedge \mathsf{A\,X}\ [\neg \mathsf{r}(d)\ \mathsf{U}\ (\mathsf{term} \vee (\mathsf{d}(d) \wedge \neg \mathsf{r}(d)))]]$
DAP 3 *Weakly Redundant Data*	$\mathsf{EF}[\mathsf{w}(d) \wedge \mathsf{X}[\neg \mathsf{r}(d)\ \mathsf{U}\ (\mathsf{term} \vee (\mathsf{d}(d) \wedge \neg \mathsf{r}(d)))]]$
DAP 4 *Strongly Lost Data*	$\mathsf{EF}[\mathsf{w}(d) \wedge \mathsf{A\,X}\ [\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg \mathsf{r}(d))]]$
DAP 5 *Weakly Lost Data*	$\mathsf{EF}[\mathsf{w}(d) \wedge \mathsf{X}[\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg \mathsf{r}(d))]]$
DAP 6 *Inconsistent Data*	$\bigvee_{t \in T : d \in \mathsf{change}(t)} \mathsf{EF}[(\mathsf{exec}(t) \wedge \bigvee_{t' \neq t : d \in \mathsf{use}(t')} \mathsf{exec}(t'))\ \vee p_t \geq 2]$
DAP 7 *Never destroyed*	$\mathsf{EF}[\mathsf{w}(d) \wedge \mathsf{X}[\neg(\mathsf{d}(d) \vee \mathsf{w}(d))\ \mathsf{U}\ \mathsf{term}]]$
DAP 8 *Twice Destroyed*	$\mathsf{EF}[\mathsf{d}(d) \wedge \mathsf{X}[\neg \mathsf{w}(d)\ \mathsf{U}\ (\mathsf{d}(d) \wedge \neg \mathsf{w}(d))]]$
DAP 9 *Not Deleted on Time*	$\bigvee_{t \in T : d \in (\mathrm{Read}(t) \cup \mathbf{data}(\mathrm{Guard}(t))) \setminus \mathrm{Destroy}(t)} \mathsf{AG}[\mathsf{exec}(t) \Rightarrow \mathsf{exec}(t)\ \mathsf{U}\ \mathsf{G}(\neg \mathsf{r}(d))]$

$d \in \mathrm{Write}(t) \cup \mathrm{Destroy}(t)\}$, and $\mathsf{use}(t)$ stands for the set $\{d \mid d \in \mathrm{Read}(t) \cup \mathbf{data}(\mathrm{Guard}(t)) \cup \mathrm{Write}(t) \cup \mathrm{Destroy}(t)\}$.

DAP 9: Not Deleted on Time. To conclude that a data element d is not deleted on time, we need a transition that reads d without destroying it (i.e. $t \in T$ with $d \in (\mathrm{Read}(t) \cup \mathbf{data}(\mathrm{Guard}(t))) \setminus \mathrm{Destroy}(t)$), such that the execution of this transition is never followed by reading d. This means that for all paths whenever t is executed ($\mathsf{AG}[\mathsf{exec}(t) \Rightarrow \ldots]$), d is never read again after the execution of t is finished (captured by $\mathsf{exec}(t)\,\mathsf{U}\,\mathsf{G}(\neg \mathsf{r}(d))$). An additional explanation needed here is that there can be several consecutive states for which $\mathsf{exec}(t)$ is true, which means that there are events happening in parallel branches while t continues its execution. The resulting formula is given in Table 1.

All the formulas except for the last one (DAP 9) are (or can be rewritten to) CTL formulas. Negations of formulas for DAPs 1, 3, 5, 6, 7 and 8 can be rewritten to LTL. DAP 9 is a set of LTL formulas itself.

5.4 Tool Support

As explained in the previous section, all anti-patterns we identified (or their negations) can be expressed in one of the two most commonly used temporal logics, CTL or LTL. Therefore, to check for data correctness we do not need to build our own tool but can choose from a number of Petri-net model-checkers available (e.g. [10,13,5]). The Model-Checking Kit [10] allows for both CTL and LTL model-checking, and supports a variety of Petri-net modeling languages as

input. Maria [13] is an LTL model-checker, and CPN Tools [5] is a powerful framework for modeling and analysis of Colored Petri nets with the CTL model-checking facility. We used CPN Tools in our verification experiments, and we were able to easily state all the CTL anti-patterns, and to check them within a fraction of a second.

6 Conclusion

This paper provides a systematic classification of possible flaws related to the data flow in business workflows. We formulated these flaws as data-flow anti-patterns. To avoid ambiguities inherent to formulations in a natural language, we formalized the anti-patterns in the temporal logic CTL^*. All anti-patterns belong to one of the two (or both) most popular subsets of CTL^*: CTL and LTL. This opens a way to easy tool support for our approach, since there are many model-checkers for both CTL and LTL.

Our approach is a first step towards a unifying framework for the integral analysis of workflows taking into account both control and data flow. As we showed in the example related to Fig. 4 (Subsection 5.2), by including data flow along with control flow into consideration when checking classical properties of workflow like soundness, we can reduce the number of false positives and false negatives caused by (unavoidable) abstraction of data values.

In the future we will try to identify more anti-patterns. We will also build an integrated tool-chain that starts with the check for boundedness, then performs the preprocessing transformations and Kripke structure generation, proceeds in looking for anti-patterns' instances by using an existing model-checker, e.g. [10], and finally generating a verification report for the workflow designer.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2004)
3. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. BPM Center Report BPM-08-02, BPMcenter.org (2008)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
5. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page, `http://wiki.daimi.au.dk/cpntools/`
6. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley & Sons, Chichester (2005)
7. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Software Engineering Methodology 15(1), 1–38 (2006)

8. Fan, S., Dou, W.C., Chen, J.: Dual Workflow Nets: Mixed Control/Data-Flow Representation for Workflow Modeling and Verification. In: Chang, K.C.-C., Wang, W., Chen, L., Ellis, C.A., Hsu, C.-H., Tsoi, A.C., Wang, H. (eds.) APWeb/WAIM 2007. LNCS, vol. 4537, pp. 433–444. Springer, Heidelberg (2007)
9. Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases 3, 119–153 (1995)
10. Institute of Formal Methods in Computer Science, Software Reliability and Security Group, University of Stuttgart. Model-Checking Kit Home Page, `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/mckit/`
11. Jablonski, S., Bussler, C.: Workflow Management: Modeling Concepts, Architecture, and Implementation. International Thomson Computer Press, London (1996)
12. Koenig, A.: Patterns and Antipatterns. Journal of Object-Oriented Programming 8(1), 46–48 (1995)
13. Mäkelä, M.: Maria: Modular Reachability Analyser for Algebraic System Nets. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 434–444. Springer, Heidelberg (2002)
14. Reichert, M., Dadam, P.: ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. Journal of Intelligent Information Systems 10(2), 93–129 (1998)
15. Sadiq, S.W., Orlowska, M.E., Sadiq, W., Foulger, C.: Data Flow and Validation in Workflow Modelling. In: Fifteenth Australasian Database Conference (ADC), Dunedin, New Zealand. CRPIT, vol. 27, pp. 207–214. Australian Computer Society (2004)
16. Sadiq, W., Orlowska, M.E.: Analyzing Process Models using Graph Reduction Techniques. Information Systems 25(2), 117–134 (2000)
17. Schmidt, D.A.: Data Flow Analysis is Model Checking of Abstract Interpretations. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1998), pp. 38–48. ACM, New York (1998)
18. Sun, S.X., Zhao, J.L., Nunamaker, J.F., Liu Sheng, O.R.: Formulating the Data Flow Perspective for Business Process Management. Information Systems Research 17(4), 374–391 (2006)
19. Sundari, M.H., Sen, A.K., Bagchi, A.: Detecting Data Flow Errors in Workflows: A Systematic Graph Traversal Approach. In: 17th Workshop on Information Technology & Systems (WITS 2007), Montreal (2007)
20. Verbeek, H.M.W., Basten, T., van der Aalst, W.M.P.: Diagnosing Workflow Processes using Woflan. The Computer Journal 44(4), 246–279 (2001)
21. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin (2007)