

# Data-Flow Anti-Patterns: Discovering Dataflow 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, 22]. 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 analysis techniques focus on verification, i.e., 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, 21]. 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. Even when the routing elements are different from Petri nets, the informal semantics of the

language is typically token-based and hence a (partial) mapping is relatively straightforward.

Unfortunately, lion’s share of attention has been devoted to control-flow while ignoring other perspectives such as data-flow and resource allocation. Analysis techniques typically check for errors such as deadlocks, livelocks, etc. while abstracting from data and resources. Existing approaches typically suffer from the following two problems: (1) they look at only one perspective in isolation (e.g., only control-flow), and (2) the types of errors they capture are usually not configurable and mainly driven by the verification algorithms themselves rather than by user requirements.

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 (e.g., an activity may *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 (see [20]).

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<sub>flex</sub> tool [14] supports a limited set of checks for data-flow correctness. The focus is entirely on dynamic changes in workflow models.

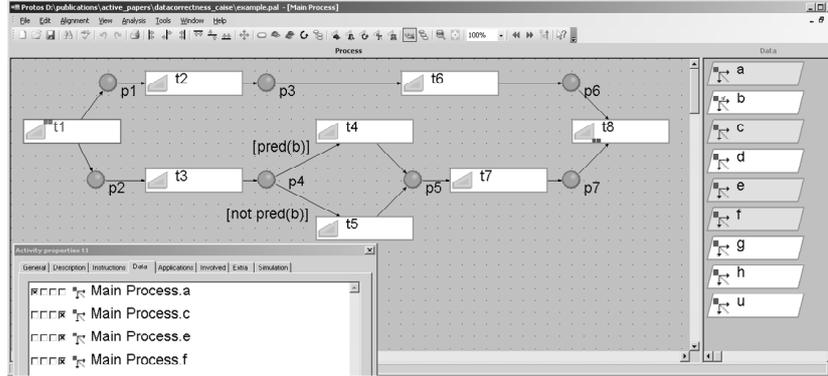
The work closest to ours is [7]. There, model checking is used to verify business workflows, from both the control- and data-flow perspective. The underlying workflow language is UML diagrams as opposed to the Petri net approach taken in this paper. Only a few data correctness properties are identified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

In the field of software verification, static analysis of software and 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

In this paper, we focus on the verification of workflow models with data. Although any workflow design will incorporate data, few analysis techniques take data into account. To illustrate the relevance of this, consider the Protos model shown in Fig. 1. Protos (Pallas Athena) uses a Petri-net-based 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. 1 shows the control flow while the right-hand side shows the different data elements. There are 9 data elements ( $a, \dots, h, u$ ), and these elements are linked to tasks in the process. The left-most task ( $t1$ ) is the initial task that reads data element  $a$  and writes data elements  $c$ ,  $e$ , and  $f$ . The colors show the relationships of  $t1$ , and the bottom window of Fig. 1 shows the nature of these relationships.

In this paper, we consider four types of relationships between tasks and data elements. First of all, a task may *read* a particular data element. This implies that the data element is expected to have a value before the task is executed.



**Fig. 1.** A Protos model showing both the control-flow and data-flow

Second, a task may *write* a particular data element. This means that the 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 particular data element. Afterwards, the data element has no value. Finally, a task may use a particular data element in its *guard* (optional). In Fig. 1 there is a choice following  $t3$  that depends on data element  $b$ . If the boolean expression  $\text{pred}(b)$  evaluates to true,  $t4$  is selected, otherwise  $t5$ .

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 the four relationships between tasks and data elements mentioned earlier (read, write, destroy, and guard). 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  $\mathcal{D}$ ;
- $\text{Read} : T \rightarrow 2^{\mathcal{D}}$  is the reading data labeling function;
- $\text{Write} : T \rightarrow 2^{\mathcal{D}}$  is the writing data labeling function;
- $\text{Destroy} : T \rightarrow 2^{\mathcal{D}}$  is the destroying data labeling function; and
- $\text{Guard} : T \rightarrow \mathcal{G}_{\mathcal{D}}$  is the guarding function, assigning guards to transitions.

□

Fig. 2 shows an example of a WFD-net. This WFD-net is essentially the same workflow model as shown in Fig. 1 (using the Protos notation), but now

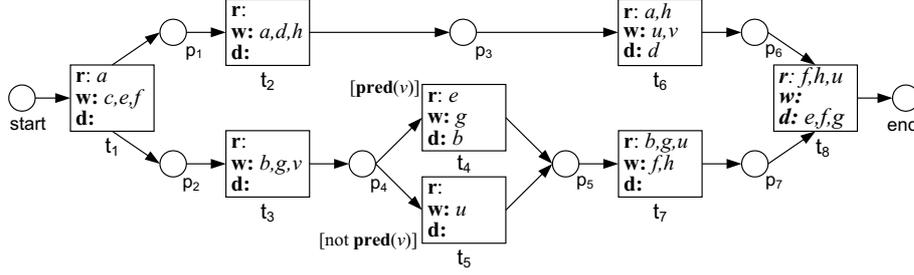


Fig. 2. A WFD-net

all annotations are shown explicitly. For example:  $\text{Read}(t_6) = \{a, h\}$ ,  $\text{Write}(t_6) = \{u, v\}$ ,  $\text{Destroy}(t_6) = \{d\}$ , and  $\text{Guard}(t_6) = \text{true}$  (i.e., no guard). If one ignores the read, write, destroy, and guard annotations, Fig. 2 is simply 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, in parallel with the lower process fragment that starts with  $t_3$  and ends with  $t_7$ . Between  $t_3$  and  $t_7$  either  $t_4$  or  $t_5$  is executed. This choice depends on the evaluation of  $\text{pred}(b)$ .

WFD-nets can be seen as an abstraction from notations deployed by widely used modeling tools. As illustrated by Fig. 1, the language used by Protos is close to Definition 1. Also other notations such as the Business Process Modeling Notation (BPMN), extended Event-driven Process Chains (eEPCs), UML activity diagrams, etc. 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 most widely used correctness notion for workflows. The basic idea is that the process cannot deadlock or livelock, i.e. that 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. 2 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 WFD-net shown in Fig. 2. Using data for the evaluation of guards is similar to reading data. 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. 2 will thus 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. 2, 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  $t_7$ , which 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 chosen. 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. 2, 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. 2. 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  $d_1, \dots, d_k$ , 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. 2, 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. 2,  $g$  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. 2,  $u$  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.

**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, in Fig. 2  $a$  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.

**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 cannot be read again independently of the choices made in the workflow.*

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. This anti-pattern is especially important for scientific workflows, where data is often large, and its unnecessary storage should be avoided.

## 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 dataflow 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.*  $\square$

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  $s_0, s_1, s_2, \dots$  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  $\pi = s_0, s_1, s_2, \dots$  and some  $k \geq 0$ ,  $\pi^k$  denotes the path  $s_k, s_{k+1}, s_{k+2}, \dots$ .

We now define the syntax of CTL\*.

**Definition 3.** *The classes  $\Phi$  of CTL\* state formulas and  $\Psi$  of CTL\* path formulas are generated by the following grammar:*

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

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

Validity of CTL\* formulas is defined as follows.

**Definition 4.** We define when a CTL\* state formula  $\phi$  is valid in a state  $s$  (notation:  $s \models \phi$ ) and when a CTL\* path formula  $\psi$  is valid on a path  $\pi$  (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  $s \models \phi_2$ ;
- $s \models E\psi$  iff there exists a path  $\pi$  from  $s$  such that  $\pi \models \psi$ ;
- $\pi \models \phi$  iff  $s$  is the first state of  $\pi$  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  $\pi \models \psi_2$ ;
- $\pi \models X\psi$  iff  $\pi^1 \models \psi$ ; and
- $\pi \models \psi \cup \psi'$  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  $X\psi$  says that  $\psi$  holds *next*, i.e. in the second state of a considered path. A formula  $\psi \cup \psi'$  says that, along a given path,  $\psi$  holds (at least) *until*  $\psi'$  holds. We standardly write  $F\psi$  for  $\top \cup \psi$  (“In the future  $\psi$ ” or “ $\psi$  will hold eventually”),  $G\psi$  for  $\neg F\neg\psi$  (“Globally  $\psi$ ” or “ $\psi$  holds *always* along a path”), and  $A\psi$  for  $\neg E\neg\psi$  (“ $\psi$  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  $A\psi$ , where  $\psi$  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  $\psi \cup \psi'$  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 of LTL is that formulas can be checked on-the-fly. 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\*. 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.

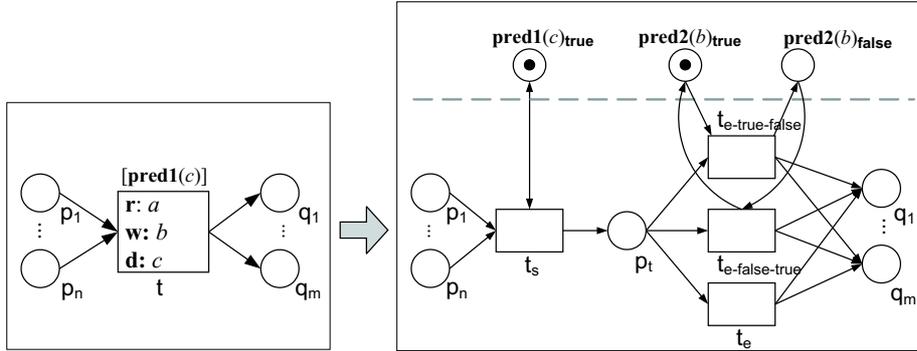


Fig. 3. Decomposition of a transition in a WFD-net

*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 still 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*  $\text{pred}$  appearing in some guard we introduce places  $\text{pred}_{true}$  and  $\text{pred}_{false}$ . A token on  $\text{pred}_{true}$  indicates that the predicate is evaluated to true for the current set of data values. A token on  $\text{pred}_{false}$  means that  $\text{pred}$  evaluates to false.
3. For every transition  $t$  with a *guard*  $\text{pred}$  in the WFD-net, we add an arc from  $\text{pred}_{true}$  to  $t_s$  and an arc back from  $t_s$  to  $\text{pred}_{true}$  to our preprocessed net. This self-loop makes sure that  $t$  is only executed only when its guard is evaluated to true. In case of guard  $\neg\text{pred}$  we add the arcs to the place  $\text{pred}_{false}$  instead of  $\text{pred}_{true}$ .
4. A *change* of the value of a data element  $d$  that appears in a *predicate*  $\text{pred}$  may potentially change the evaluation of  $\text{pred}$ . We reflect that by assuming that every transition  $t$  writing to  $d$  might change the value of  $\text{pred}$  (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.<sup>1</sup>

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  $\text{pred1}(c)$  writing to data element  $b$ . We assume that  $b$  is used in some predicate  $\text{pred2}(b)$ , guarding some other transition(s) of the workflow. Places  $\text{pred1}(c)_{true}$ ,  $\text{pred1}(c)_{false}$  (not

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

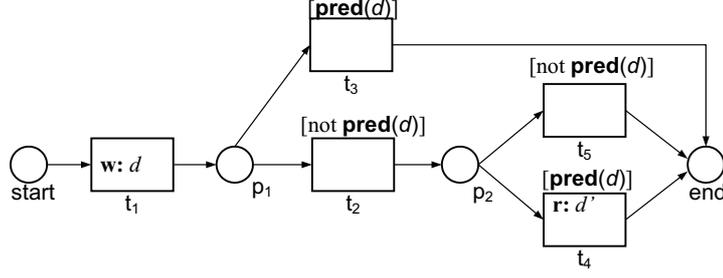


Fig. 4. Data can influence reachability

shown in the Fig.),  $\text{pred2}(b)_{\text{true}}$  and  $\text{pred2}(b)_{\text{false}}$  are added to represent the values of the predicates. The transition is split into the start transition  $t_s$ , controlled by place  $\text{pred1}(c)_{\text{true}}$ , and transitions  $t_{e-\text{true}-\text{false}}$ , changing the value of the predicate  $\text{pred2}$  from true to false,  $t_{e-\text{false}-\text{true}}$ , changing the value of the predicate from false to true, and  $t_e$  leaving the value unchanged.

We make an arbitrary choice assuming that all  $\text{pred}_{\text{true}}$  places initially have a token and  $\text{pred}_{\text{false}}$  places do 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  $\text{exec}(t)$  instead of  $p_t \geq 1$ . The workflow being in its *final state* is denoted as *term* and it is defined as  $(\text{end} = 1 \wedge \bigwedge_{p \in P \setminus \{\text{end}\}} (p = 0))$ . To represent the fact that a data element  $d \in \mathcal{D}$  is being *read* either as input or for evaluating a guard of some transition, we write  $r(d)$ , abbreviating thus  $\bigvee_{t: d \in \text{Read}(t) \cup \text{data}(\text{Guard}(t))} \text{exec}(t)$ .

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  $t_3$  in Fig. 2 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  $\text{pred}(d)$  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 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  $E[\neg w(d) \cup (r(d) \vee d(d))]$ . A data element  $d$  is also missing if  $d$  first get destroyed and then no writing takes place until  $d$  is read or destroyed, which can be captured by  $EF[d(d) \wedge (\neg w(d) \cup (r(d) \vee d(d)))]$ . 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.  $EF[w(d) \wedge \dots]$ ) such that in all possible continuations of this path no reading takes place until the workflow terminates or  $d$  get destroyed ( $AX[\neg r(d) \cup (\text{term} \vee (d(d) \wedge \neg r(d)))]$ ). We need  $X$  here because we want to impose  $\neg r(d)$  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  $\neg r(d)$  in the conjunction  $d(d) \wedge \neg r(d)$ .

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

$$\bigvee_{t \in T: d \in \text{change}(t)} EF[(\text{exec}(t) \wedge \bigvee_{t' \neq t: d \in \text{use}(t')} \text{exec}(t'))],$$

or if two or more instances of transition  $t$  changing  $d$  can be executed in parallel, captured by

$$\bigvee_{t \in T: d \in \text{change}(t)} EF[p_t \geq 2].$$

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

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

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

*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. a transition  $t \in T$  with  $d \in (\text{Read}(t) \cup \text{data}(\text{Guard}(t))) \setminus \text{Destroy}(t)$ , such that the execution of this transition is never followed by reading  $d$ . This means that for all paths whenever  $t$  is executed (i.e.  $AG[\text{exec}(t) \Rightarrow \dots]$ ),  $d$  is never read again after finishing the execution of  $t$  (captured by  $\text{exec}(t) \cup G(\neg r(d))$ ). An additional explanation needed here is related to the fact that there can be several consecutive states for which  $\text{exec}(t)$  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 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 or use some particular one, but we 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 flow of data in business workflows. We formulated these errors 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 first 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 generates a verification report for the workflow designer.

## References

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

- and Network Technologies, and Information Management (APWeb/WAIM 2007 Workshops), volume 4537 of *Lecture Notes in Computer Science*, pages 433–444. Springer-Verlag, Berlin, 2007.
9. D. Georgakopoulos, M. Hornick, and A. Sheth. 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. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
  12. A. Koenig. Patterns and Antipatterns. *Journal of Object-Oriented Programming*, 8(1):46–48, 1995.
  13. M. Mäkelä. Maria: Modular Reachability Analyser for Algebraic System Nets. In *Applications and Theory of Petri Nets 2002 (ICATPN'2002)*, volume 2360 of *LNCS*, pages 434–444. Springer, 2002.
  14. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
  15. S.W. Sadiq, M.E. Orlowska, W.Sadiq, and C. Foulger. Data Flow and Validation in Workflow Modelling. In *Fifteenth Australasian Database Conference (ADC), Dunedin, New Zealand*, volume 27 of *CRPIT*, pages 207–214. Australian Computer Society, 2004.
  16. W. Sadiq and M.E. Orlowska. Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, 25(2):117–134, 2000.
  17. D.A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'98)*, pages 38–48. ACM, 1998.
  18. S.X. Sun, J.L. Zhao, J.F. Nunamaker, and O.R. Liu Sheng. Formulating the Data Flow Perspective for Business Process Management. *Information Systems Research*, 17(4):374–391, 2006.
  19. M.H. Sundari, A.K. Sen, and A. Bagchi. Detecting Data Flow Errors in Workflows: A Systematic Graph Traversal Approach. In *17th Workshop on Information Technology & Systems (WITS-2007), Montreal*, 2007.
  20. Nikola Trčka, W.M.P. van der Aalst, and N. Sidorova. Analyzing Control-Flow and Data-Flow in Workflow Processes in a Unified Way. Technical Report CS 08/31, Eindhoven University of Technology, 2008.
  21. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
  22. M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Berlin, 2007.