# Analyzing control-flow and data-flow in workflow processes in a unified way

# Analyzing Control-Flow and Data-Flow in Workflow Processes in a Unified Way

Nikola Trčka, Wil 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.** Workflow correctness properties are usually defined based on one workflow perspective only, e.g. the control-flow or the data-flow. In this paper we consider workflow correctness criteria looking at the control flow extended with the read/write/destroy information for data items. We formalize some common control-flow errors, and we introduce behavioral *anti-patterns* related to the handling of data. In addition to extending, refining, and classifying existing methods, our paper provides a unifying framework for complete workflow verification, using the well-known, stable, adaptable, and effective model-checking approach.

## 1 Introduction

The workflow concept has defacto become the standard paradigm for process modeling in business process management (BPM) systems. Workflows are typically looked from three perspectives: 1) the control perspective, describing the logical order of tasks; 2) the data perspective, describing the information exchange between tasks; and 3) the resource perspective, describing the originators of tasks.

The topic of workflow verification, i.e. on checking workflows for logical correctness, has been studied since the mid-nineties. Today, even the commercial BPM vendors tend to provide more support for various form of workflow analysis. Several methods and tools exists to capture errors in the control-flow (finding, e.g., deadlocks and livelocks) [2], errors in the flow of data [15] (e.g., reading from an uninitialized element type of errors), and resource-deficiency type of problems. However, all of these works suffer at least one of the following problems: 1) they look at only one perspective in isolation; 2) the exact details of the erroneous scenarios are not always clear, being hidden in the verification algorithms; 3) every method has its own set of properties to verify, as there is no common agreement on the desired set of correctness requirements; 4) the methods are not adaptive enough as properties cannot be easily modified or

added; 5) tool support is lacking; 6) it is not always clear what is (or could be) the content and the quality of the diagnostics report; and 7) the underlying workflow model has limited expressivity.

Although it makes sense to abstract from the resource information during verification, as resources are external and dynamic in nature, the same does not hold for the other two perspectives. On the one side, ignoring the data aspect while checking for the control-flow correctness could cause some errors to pass undetected or to be falsely reported. This is because the routing decisions in a workflow are typically based on data, while in the absence of data information they can only be considered as non-deterministic and fair. On the other side, checking data correctness without looking at the control-flow is also incomplete. To capture, e.g., errors related to data-inconsistency, we clearly must know which tasks are possible to happen in parallel. Therefore, the control-flow and data-flow must be verified in a combined fashion.

In this paper we identify, and systematically classify, several generic control- and data-flow errors. We use the very expressive temporal logic $\mathsf{CTL}^*$ [4] (and its subsets CTL and LTL) for this purpose, thus removing all ambiguities inherent to formulations in a natural language. In this way we not only establish a unifying formal framework for detecting these errors, but also build a highly adaptive (new properties are easily added) and stable (model-checking has been successfully used for years) setting with excellent diagnostic features (model-checking provides error-traces). We, moreover, do need not to build our own tool, nor use a particular incomplete one, but are able to choose from several verification tools available on the market.

The flow-oriented nature of workflow processes makes the Petri net formalism an obvious candidate for the modeling of workflows. The language is very generic, well-studied, and supports (almost) all the workflow patterns recognized in [3]. Here we use the (syntactically) restricted form of Petri nets, called workflow nets [1] (WF-nets), that more directly capture the concept of a workflow. Most workflow management systems provide a graphical language that is close to WF-nets. Moreover, even when the routing elements are different from WF-nets, the informal semantics of the language is typically token-based and hence a (partial) mapping is relatively straightforward. The data information is incorporated into the standard WF-net model by allowing tasks to perform reading, writing, and destroying operations on data, and by guarding (i.e. blocking) tasks solely on the basis of data. By categorizing data operations into this simple set of primitives, and using the guard concept, we make our model

generic enough to support specifications from many existing (commercial) tools, (such as Protos [12], e.g.), and to capture the elements of the CRUD (create, read, update, delete) lifecycle known from the database world.

The structure of the rest of this paper is as follows. Section 2 gives some preliminaries and introduces our workflow nets with data model. Section 3 is the main section of the paper. There we define correctness properties, both control-flow and data-flow related, and show how they can be verified by model checking. In the last section we conclude the paper, discuss related work, and provide directions for future work.

## 2 Preliminaries

We first recall the basics of Petri nets and WF-nets. Then we define our workflow model, called "workflow nets with data". At the end, we introduce the temporal logic $\mathsf{CTL}^*$ that is used to describe desirable correctness properties.

### 2.1 Workflow nets with data

Workflow nets with data are based on Petri nets and workflow nets, so we define these two models first.

*Petri nets* The language of Petri nets is a well established formalism, able to model sequential and concurrent behavior, choices, and different types of communication. We now give a formal definition of a Petri net.

**Definition 1.** *A* Petri net *is a tuple* $N = \langle P, T, F \rangle$*, where:*

- $P$ *and* $T$ *are two disjoint non-empty finite sets of* places *and* transitions *respectively;*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (from transitions to places and from places to transitions), called the* flow relation. □

Fig. 1a shows a Petri net with places $p_1, \ldots, p_7$ and transitions $t_1, \ldots, t_5$. The directed arcs determine the flow relation.

For $t \in T$, we define the *preset of* $t$ as ${}^\bullet t = \{p \mid (p, t) \in F\}$, and the *postset of* $t$ as $t^\bullet = \{p \mid (t, p) \in F\}$. Analogously we define ${}^\bullet p$ and $p^\bullet$ for pre- and postsets of places. For the Petri net from Fig. 1a we have, e.g., ${}^\bullet t_2 = {}^\bullet t_4 = \{p_2\}$, ${}^\bullet t_5 = \{p_4, p_5\}$, $p_2^\bullet = \{t_2, t_3\}$ and $t_1^\bullet = \{p_2, p_3\}$.

At any time a place contains zero or more *tokens*, drawn as black dots. The state of the Petri net, called a *marking*, is the distribution

of tokens over its places. Formally, a marking is defined as a mapping $m : P \to \mathbb{N}$, i.e., as a multiset over $P$. We use standard notation for multisets and write, e.g., $m = [2p + q]$ for a marking $m$ with $m(p) = 2$, $m(q) = 1$, and $m(x) = 0$ for $x \in P \setminus \{p, q\}$. We define $+$ and $-$ for the sum and the difference of two markings, and $=, <, >, \leq, \geq$ for comparison of markings in the standard way. For the above marking $m$ we have, e.g., $m \leq [3p + 2q + r]$ and $m + [q + 3r] = [2p + 3q + 3r]$. A pair $(N, m)$, where $N$ is a Petri net and $m$ is a marking, is called a *marked* Petri net.

A transition $t \in T$ is *enabled* in a marking $m$ if ${}^\bullet t \leq m$. An enabled transition $t$ may *fire*, which results in a new marking $m'$ defined by $m' \stackrel{\text{def}}{=} m - {}^\bullet t + t^\bullet$. This firing is denoted as $m[t\rangle m'$. In Figure 1a, $t_1$ is enabled and its firing will result in the state that marks places $p_2$ and $p_3$ with one token. In this state, $t_2$, $t_3$, and $t_4$ are enabled. If, e.g., $t_2$ fires now, $t_3$ becomes disabled, but $t_4$ remains enabled. Similarly, if $t_4$ fires, $t_3$ becomes disabled, but $t_2$ remains enabled, etc.

The *reachability graph* of a marked Petri net shows its dynamic behavior. Every node in this graph represents a reachable marking, and every labeled arc indicates the firing of a transition. For the marked net $(N, m_0)$, the reachability graph is formally defined as the tuple $\langle S, \to\rangle$ where $S$ and $\to$ are the smallest sets satisfying the following: 1) $m_0 \in S$, and 2) if $m \in S$ and $m[t\rangle m'$, then $m' \in S$ and $(m, t, m') \in \to$. In this paper we assume that the reachability graph of a Petri net is always finite. This property can be checked prior to any analysis (and, in most cases, already on the structure of the net). Fig. 1b shows the reachability graph of the Petri net from Fig. 1a.

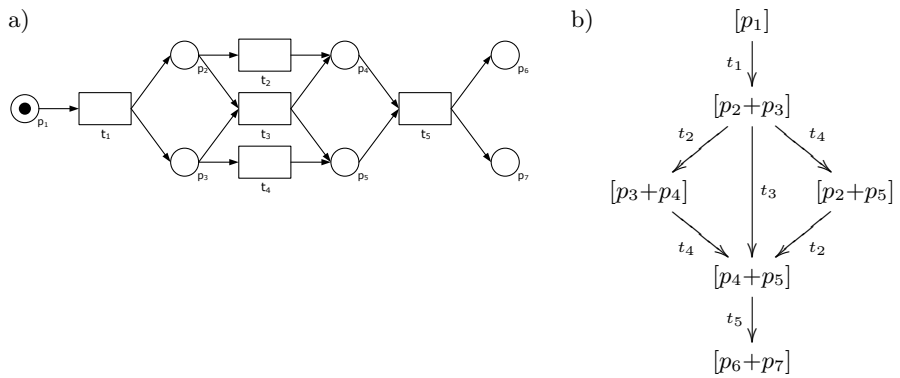



**Fig. 1.** a) A Petri net and b) its reachability graph

*Workflow nets* Workflow nets [1] impose syntactic restrictions on Petri nets to comply to the workflow concept. The notion was triggered by the assumption that a typical workflow has a well-defined starting point and a well-defined ending point.

**Definition 2.** *A Petri net* $N = \langle P, T, F \rangle$ *is a* Workflow net (WF-net) *iff:*

1. *there is a single source place* $\mathsf{start}$*, i.e.,* $\{p \in P \mid {}^\bullet p = \emptyset\} = \{\mathsf{start}\}$*;*
2. *there is a single sink place* $\mathsf{end}$*, i.e.,* $\{p \in P \mid p^\bullet = \emptyset\} = \{\mathsf{end}\}$*;*
3. *every node is on a path from* $\mathsf{start}$ *to* $\mathsf{end}$*, i.e., for all* $n \in N \cup T$*,* $(\mathsf{start}, n) \in F^*$ *and* $(n, \mathsf{end}) \in F^*$*, for* $F^*$ *being the reflexive-transitive closure of* $F$*.* □

The Petri net from Fig. 1a is not a workflow net because it has two sink places, $p_6$ and $p_7$. If these two places (and their corresponding arcs) are merged into one place $p_{67}$, the net becomes a workflow net, with one source place $p_1$ and one sink place $p_{67}$.

Transitions in a WF-net are also called *tasks*. A *case* is a workflow instance, i.e., a marked WF-net in which the $\mathsf{start}$ place is marked with one token and all other places are empty. In this paper we study workflow properties related to one single case in isolation, assuming that different cases are completely independent from each other. Therefore, when we talk of the dynamic properties of a WF-net, we actually mean the properties of its cases. The *final state* of a case (and of the workflow net) is the marking in which the end place is marked with one token and all the other places are empty.

*Workflow nets with data* A workflow net with data elements is a workflow net in which transitions can read, write and/or destroy some data. Every transition has, in addition, a data dependent guard that can influence the enableness of this transition. We formalize the concept of a guard first.

**Definition 3.** *Let* $\mathcal{D}$ *be a set of data elements. A* predicate **pred** *(on* $d_1, \ldots, d_n \in \mathcal{D}$*) is an expression* $\mathbf{pred}(d_1, \ldots, d_n)$ *that evaluates to* true *or* false*. A* guard *is either a predicate, the negation of a predicate, or the value* true*. The set of all guards over* $\mathcal{D}$ *is denoted* $G_{\mathcal{D}}$*. The set of data elements used in a guard* $g$ *is denoted* $\mathbf{data}(g)$*.* □

We can now define workflow nets with data.

**Definition 4 (WFD-net).** *A tuple* $\langle P, T, F, \mathcal{D}, \mathbf{r}, \mathbf{w}, \mathbf{d}, \mathbf{grd} \rangle$ *is a* Workflow net with data (a WFD-net) *iff:*

– $\langle P, T, F \rangle$ *is a WF-net;*
– $\mathcal{D}$ *is a set of* data elements*;*
– $\mathbf{r} : T \rightarrow 2^{\mathcal{D}}$ *is the* reading data labeling function*;*
– $\mathbf{w} : T \rightarrow 2^{\mathcal{D}}$ *is the* writing data labeling function*;*
– $\mathbf{d} : T \rightarrow 2^{\mathcal{D}}$ *is the* destroying data labeling function*; and*
– $\mathbf{grd} : T \rightarrow G_{\mathcal{D}}$ *is the* guarding function*, assigning guards to transitions.* □

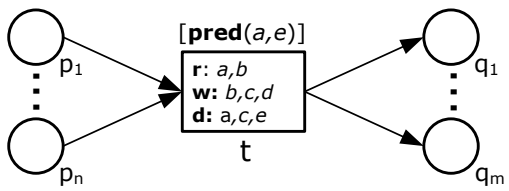



**Fig. 2.** A transition in a workflow net with data

Fig. 2 shows (the visualization of) a typical task in a workflow net with data. This task takes the data elements $a$ and $b$ as input, and stores its output in $b$, $c$ and $d$. We implicitly assume that inside a task reading always precedes writing, so when $t$ is completed the old version of $b$ is overwritten and thus lost. If $c$ and $d$ existed before the occurrence of $t$, their old values are lost as well. If they did not exists, they were created by $t$. After producing its output, $t$ destroys $a$, $e$, and the just created $c$. Task $t$ is only enabled, and thus can fire, if the input places $p_1, \ldots, p_n$ are marked, and the guard **pred**, that depends on the data elements $a$ and $e$, evaluates to true.

Categorizing data operations into read, write and destroy operations, increases flexibility and allows us to express many other standard operations on data easily. For example, the elements of the CRUD (create, read, update, delete) lifecycle from the database literature can simply be represented as follows: creating data is writing to a data element without reading from the same element, reading corresponds to reading an element without writing to the same element, updating is reading and writing to the same data element, and finally, deleting corresponds to destroying.

Guards on transitions allow us to model decision points in which the choice is made based on some data elements. For example, Fig. 3 shows how a standard XOR-split construct can be specified. Any other, less

basic, choice constructs can be represented as combinations of these XORs and regular Petri net transitions (AND-splits).

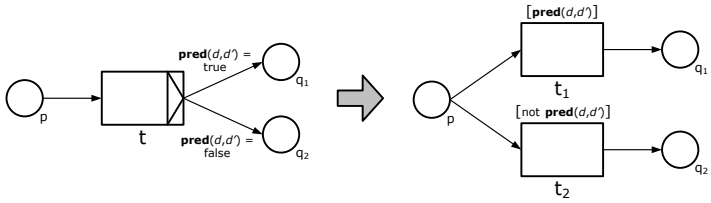



**Fig. 3.** Representing an XOR-split where the choice depends on data

### 2.2 Temporal logic CTL*

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

**Definition 5.** *A* Kripke structure *is a tuple* $(S, A, \mathcal{L}, \rightarrow)$ *where* $S$ *is a 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 $s'$, also then written as $s \rightarrow s'$. For a state $s$, $\mathcal{L}(s)$ is the set of atomic propositions that *hold* in $s$. For the remainder of the section we fix a Kripke structure $(S, A, \mathcal{L}, \rightarrow)$.

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 $k \geq 0$, $\pi^k$ denotes the path $s_k, s_{k+1}, s_{k+2}, \ldots$.

We now define the syntax of CTL* [4].

**Definition 6.** *The classes* $\Phi$ *of* CTL* state formulas *and* $\Psi$ *of* CTL* path formulas *are generated by the following grammar:*

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

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

Validity of CTL* formulas is defined as follows.

**Definition 7.** *We define when a* $\mathsf{CTL}^*$ *state formula* $\phi$ *is* valid *in a state* $s$ *(notation:* $s \models \phi$*) and when a* $\mathsf{CTL}^*$*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* $s \models \phi_2$*;*
- $s \models \mathsf{E}\psi$ *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* $\pi \models \psi_2$*;*
- $\pi \models \mathsf{X}\psi$ *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 $\psi$ holds *next*, i.e., in the second state of a considered path. A formula $\psi \,\mathsf{U}\, \psi'$ says that, along a given path, $\psi$ holds *until* $\psi'$ holds. As standard, as a shorthand we write $\mathsf{F}\psi$ for $\top \,\mathsf{U}\, \psi$ ("In the future $\psi$" or "$\psi$ will hold *eventually*"), $\mathsf{G}\psi$ for $\neg\mathsf{F}\neg\psi$ ("Globally $\psi$" or "$\psi$ holds *always* along a path"), and $\mathsf{A}\psi$ for $\neg\mathsf{E}\neg\psi$ ("$\psi$ holds along *all* paths"). The combinators $\mathsf{AG}$ and $\mathsf{EF}$ can then be interpreted as "in all states" and "in some state" respectively.

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

## 3 Correctness Properties

In this section we define correctness properties related to the control-flow and the data-flow, using CTL*. As this logic is defined on Kripke structure, we must first obtain one from a WFD-net.

### 3.1 From WFD-net to a Kripke structure

In order to generate a Kripke structure representing the behavior of a WFD-net, the most natural thing to do is to see this net as a regular WF-net (i.e. abstract from data) and build its reachability graph. The transition labels on this graph can be ignored, and a suitable state label can be generated for every marking representing this state using the read/write/destroy information. This direct and simple approach, however, leads to two problems.

The first problem is that, in a WFD-net, data elements can be used in guards, and can thus affect the reachability of a certain marking. Abstracting from data adds behavior, which could cause some errors to pass undetected or to be falsely reported. Consider, for example, the simple WFD-net from Fig. 4. After $t_1$ has performed and created the data element $d$, the predicate **pred** is evaluated. If it evaluates to true, the workflow executes $t_4$ and completes. If, however, **pred** evaluates to false, $t_2$ is executed. After $t_2$, **pred** is evaluated again for $t_3$. Since, $t_2$ does not update $d$, **pred** remains false, and so the workflow deadlocks (here a deadlock is assumed to an undesired situation). Checking this workflow without the data information would clearly not indicate this problem.

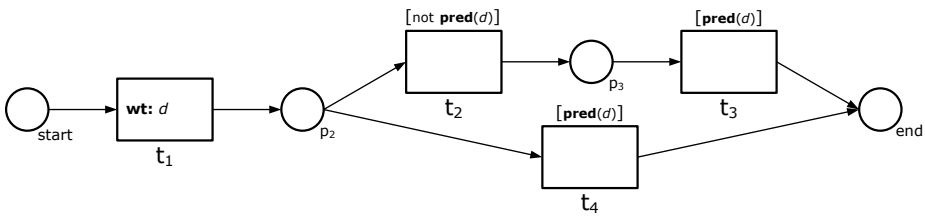



**Fig. 4.** Data can influence reachability

The second problem of the direct approach is that in the obtained reachability graph the markings do not hold enough information. We can, e.g., deduce whether two transitions are *enabled* at the same time, but not whether they are being executed at the same time. The latter info is

crucial in the verification process, as we want to make sure that, e.g., two concurrent transitions do not write to the same data.

To solve these problems we propose a preprocessing step that converts a WFD-net into a WF-net, in a more complex way comparing to the direct method, but still keeping the original structure intact. This step consists of the following smaller steps:

1. To be able to capture the situation where two transitions are executing in parallel, 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* $\mathsf{pred}$ appearing in some guard we introduce places $\mathsf{pred}_{\text{true}}$ and $\mathsf{pred}_{\text{false}}$. A token on $\mathsf{pred}_{\text{true}}$ indicates that the predicate is evaluated to true for the current set of data values. A token on $\mathsf{pred}_{\text{false}}$ means that $\mathsf{pred}$ evaluates to false. We make an arbitrary choice assuming that all $\mathsf{pred}_{\text{true}}$ places initially have a token and $\mathsf{pred}_{\text{false}}$ places do not. We can afford making an arbitrary choice as the errors related to the use of undefined data will be signaled as an error (see next section).
3. For every transition $t$ with a guard $\mathsf{pred}$ in the WFD-net, we add an arc from $\mathsf{pred}_{\text{true}}$ to $t_s$, and an arc back from $t_s$ to $\mathsf{pred}_{\text{true}}$, 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 $\neg\mathsf{pred}$ we add the arcs to the place $\mathsf{pred}_{\text{false}}$ instead.
4. A change of the value of a data element $d$ that appears in a *predicate* $\mathsf{pred}$ may potentially change the evaluation of $\mathsf{pred}$. We reflect that by assuming that every transition $t$ writing to $d$ might change the value of $\mathsf{pred}$ (or not). Therefore, we further 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. In this paper we assume that predicates do not depend on each other; our method, however, can be easily extended to support dependencies. 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. 5 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 of the workflow.

After the preprocessing step the reachability graph of the new net can be built. To obtain a Kripke structure we must then also define

the set of atomic propositions and assign labels to states. We define $A = \{p \geq i \mid p \in P, i \in \mathbb{N}\}$ for the set of atomic propositions, as this very generic set allows us to express all state properties. The labels of a state (=marking) $m$ are assigned as follows: for some $p \in P$ and $i \in \mathbb{N}$, we have $p \geq i \in \mathcal{L}(m)$ iff $m(p) \geq i$. To state the formulas in a more concise and clear way, we introduce some abbreviations. First, $p = i$ denotes $p \geq i \wedge \neg(p \geq i + 1)$. To more directly represent the fact that some transition $t$ is *executing*, we write $\mathsf{exec}(t)$ instead of $p_t \geq 1$. The final state of the workflow, denoted $\mathsf{final}$, is defined as $\mathsf{end} = 1 \wedge \bigwedge_{p \in P \setminus \{\mathsf{end}\}} (p = 0)$. To represent the fact that a data element $d \in \mathcal{D}$ is being *read*, either as input or for evaluating a guard of some transition, we write $\mathsf{r}(d)$, abbreviating thus $\bigvee_{t:d \in \mathbf{r}(\mathbf{t}) \cup \mathbf{data}(\mathbf{grd}(\mathbf{t}))} \mathsf{exec}(t)$. The constructs $\mathsf{w}(d)$ and $\mathsf{d}(d)$ are defined similarly.

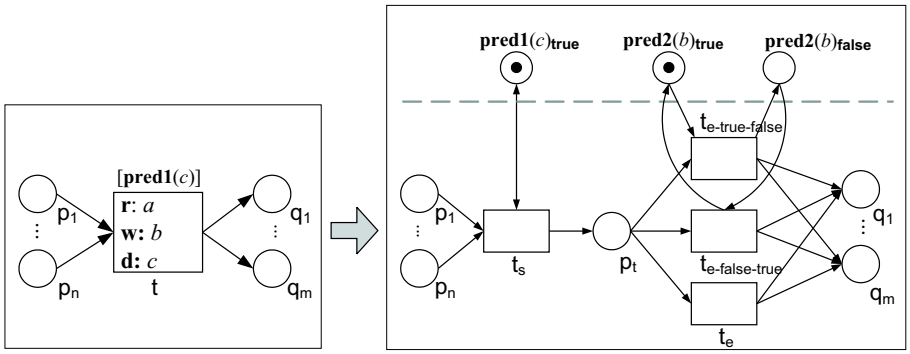



**Fig. 5.** Decomposition of a transition in a WFD-net

Having the Kripke structure formed, we can proceed with defining the desired correctness properties; we treat the control-flow aspect first.

### 3.2 Control-flow correctness

Control-flow analysis deals with questions like: "Does the workflow terminate?", "Is there a deadlock?", "Does task A ever happen?", etc. These and similar questions can be categorized into the following two correctness requirements: no-dead-transitions (every task can potentially happen), and proper-completion (the workflow reaches its final state without leaving "garbage" behind).

The no-dead-transitions requirement imposes that from the initial state every task in a workflow can potentially be performed. This is

natural and desired, as redundant tasks should not be present in the definition of a workflow. The property is expressed in $\mathsf{CTL}^*$ terms as

$$\bigwedge_{t \in T} \mathsf{EF}\, \mathsf{exec}(t).$$

Fig. 6a shows a workflow net in which transition $t_2$ is dead. This transition can never be performed (the only possible execution sequence is $t_1, t_3$) and can thus be removed. The obtained workflow, depicted in Fig. 6b, has exactly the same behavior.

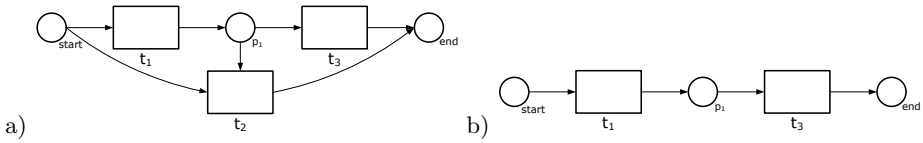



**Fig. 6.** a) A WF-net with a dead transition $t_2$, and b) a WF-net with the same behavior

The proper-completion property requires that the workflow *eventually* reaches its final state. Here the word eventually is not used in the strict, temporal logic, sense and is, thus, subject to different interpretations. We list some possibilities for the completion property inspired by or extracted from different correctness notions proposed in the literature.

*Strict completion* Strict completion requires that, starting from the initial state, every possible path of the workflow (properly) completes. A $\mathsf{CTL}^*$ formula expressing this is

$$\mathsf{AF}\, \mathsf{final}.$$

Fig. 7a shows a WF-net that strictly completes (note that data information is not needed to illustrate the differences between the proposed control-flow properties). No matter whether $t_1$ or $t_2$ fires, there will be a token in $p_1$ and a token in $p_2$. These two tokens are then consumed by $t_3$ which then puts a token in place $\mathsf{end}$, completing the workflow.

*Strict completion with fairness* Strict completion is too strong, as completion is required for *all* paths. The notion, e.g., does not allow for loops in the workflow. To eliminate this problem a weaker requirement is proposed that only takes "fair" paths into consideration. The fairness assumption is captured by some (path) formula $\psi$. In $\mathsf{CTL}^*$ terms, we require:

$$\mathsf{A}\,(\psi \Rightarrow \mathsf{F}\, \mathsf{final}).$$

Note that, for $\psi$=false, strict completion with fairness degrades to (ordinary) strict completion.

The WF-net from Fig. 7b does not complete strictly as the loop $t_2, t_4, t_2, \ldots$ might never be exited. This net, however, does complete strictly if we impose the (standard and commonly used) *strong fairness* assumption [4], i.e., let $\psi = \vee_{t \in T}[\mathsf{GF}\,(\bigwedge_{p \in {}^\bullet t} p \geq 1) \Rightarrow \mathsf{GF}\,\mathsf{exec}(t)]$. With this assumption we include only those sequences in which an infinitely often enabled transition is infinitely often executed.

*Optional completion* Optional completion does not require that every path must complete, but that no matter in which state the workflow is know, it *can* always be completed eventually. The notion is weaker than strict completion but it still captures all the important properties, like livelocks and deadlocks. Stated in $\mathsf{CTL}^*$ terms we have

$$\mathsf{AG}\,\mathsf{EF}\,\mathsf{final}.$$

Fig. 7c shows a WF-net that completes optionally but not strictly. Note that the strong fairness assumption (or any other standard one) does not help here as $t_6$ is never enabled in the infinite sequence $t_2, t_3, t_4, t_5, t_2, \ldots$.

*Eventual completion with $\phi$* This property requires that there exists a state in which $\phi$ holds and from which there is a path to completion. The intuition is that "bad" paths can be avoided if there are enough "good" paths. Typical examples for $\phi$ are the requirements that some transition is executed or that some data element is being used. Written in $\mathsf{CTL}^*$ language we have:

$$\mathsf{EF}\,(\phi \wedge \mathsf{EF}\,\mathsf{final}).$$

The workflow from Fig. 7d does not optionally complete as the execution sequence $t_1, t_4$ leads to a deadlock. It, however, eventually completes with $\phi = \mathsf{exec}(t_4)$ as the sequence $t_2, t_4$ leads to proper completion.

*Eventual completion* Eventually completion is the weakest property we study. Its only requirement is that, starting from the initial state the workflow *can* (i.e., along some path) reach the final state. In $\mathsf{CTL}^*$ terms we write this property as

$$\mathsf{EF}\,\mathsf{final}.$$

Note that eventual completion is actually eventual completion with $\phi$ for $\phi = \text{true}$.

The workflow in Fig. 7e eventually completes by executing the sequence $t_1, t_3$. It does, however, not eventually complete with $\mathsf{exec}(t_2)$, as the (only) execution of $t_2$ leads to a deadlock.

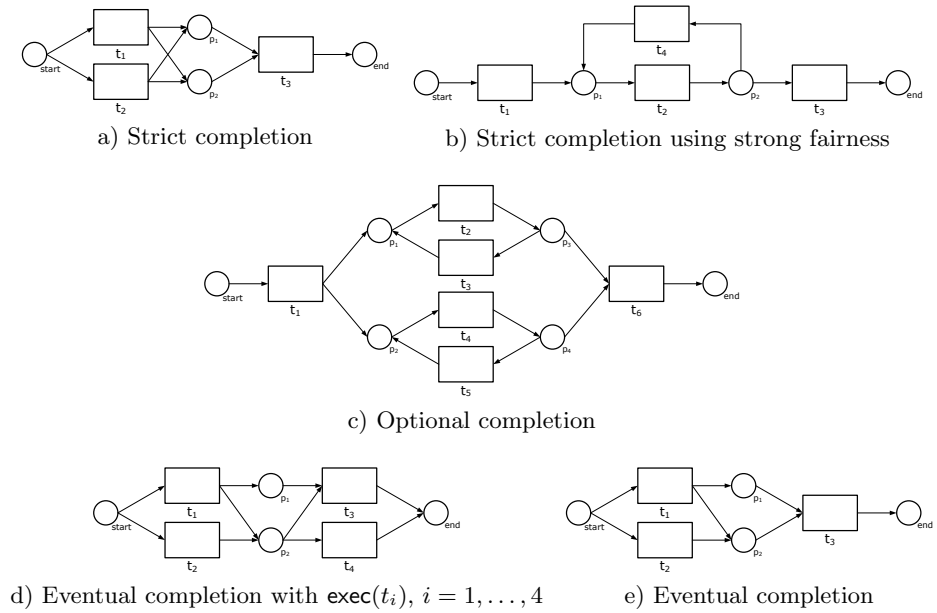
Some of the above requirements, with or without the no-dead-transition requirement, have been investigated in literature under the notion of *workflow soundness* (see [2] for an overview). *Classical soundness* [1], e.g., is the conjunction of the no-dead-task requirement and optional completion. The notion of *weak soundness* [10] imposes only the optional completion requirement. It does not consider whether some transition is dead or not. *Easy soundness* [19, 2] is even weaker, corresponding to eventual completion only. *Relaxed soundness* [6] requires that for each transition there should be at least one execution towards completion. This amounts to checking the eventual completion with $\mathsf{exec}(t)$ property, for each transition $t$. Finally, the completion requirement of [7] is strict completion with strong fairness.

In this paper we focus only on proper completion, i.e. we consider a workflow completed only when there is one token in the $\mathsf{end}$ place and there are no other tokens. Some notions of soundness from the literature weaken this requirement. For example, in *lazy soundness* [13] tokens may be left behind as long as the place $\mathsf{end}$ is marked precisely once. Although, this property does not fit into any category from above, it can be easily expressed in $\mathsf{CTL}^*$, e.g. as $\mathsf{AG}\,[\mathsf{end} \leq 1 \wedge \mathsf{EF}\,\mathsf{end} = 1]$.

### 3.3 Data correctness properties

In this section we list several properties related to the verification of the flow of data through the workflow. In contrast to the control-flow properties that we stated in the form of correctness requirements, here we define data anti-patterns that represent undesirable (data-flow) behaviors. Some of these anti-patterns are serious flaws (like, e.g., the missing data error), while some only capture an undesired and non-optimal, but not necessarily erroneous, behavior (like, e.g., the redundant data error).

*Missing data* The missing data error 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 being created again. A data element $d$ is thus missing if there is an execution path along which no writing to $d$ happens before a reading or destroying of $d$ takes place. In $\mathsf{CTL}^*$ this can be expressed as $\mathsf{E}\,[\neg\mathsf{w}(d)\,\mathsf{U}\,(\mathsf{r}(d) \vee \mathsf{d}(d))]$. A data element $d$ is also missing if it is destroyed and then no writing takes place until $d$ is read or destroyed. This can be captured by $\mathsf{EF}\,[\mathsf{d}(d) \wedge (\neg\mathsf{w}(d)\,\mathsf{U}\,(\mathsf{r}(d) \vee \mathsf{d}(d)))]$.




a) Strict completion

b) Strict completion using strong fairness

c) Optional completion

d) Eventual completion with $\mathsf{exec}(t_i)$, $i = 1, \ldots, 4$

e) Eventual completion

**Fig. 7.** Different variants of the proper completion requirement

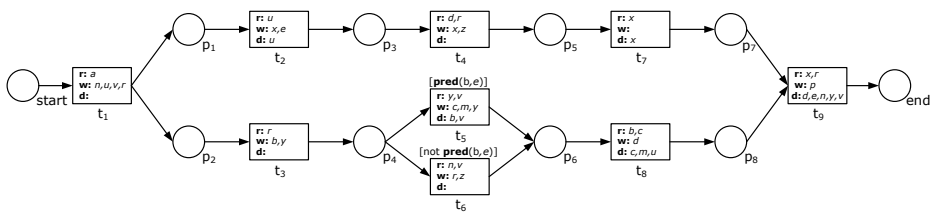



**Fig. 8.** WFD-net with data-flow errors

The disjunction of these two expressions results in the following $\mathsf{CTL}^*$ formula:

$$\mathsf{E}\,[(\neg \mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d)))\ \vee \mathsf{F}\,[\mathsf{d}(d) \wedge (\neg \mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d)))]]$$

In the example from Fig. 8, the elements $a$, $b$, $c$, $d$, $e$ and $u$ are all missing. The element $a$ needs to be read immediately by the first task; $b$ is created by $t_3$, but it can be destroyed $t_5$ before it reaches $t_8$ that reads it; $c$ is an input of $t_8$, but it is only created if the choice was made to do $t_5$ and not $p_6$; if $t_8$ does not execute before $t_4$, then $d$ is missing in $t_4$; $e$ is created in $t_2$ that might not be finished before $e$ is needed to evaluate the guard in $t_5$ and $t_6$; and finally, $u$ is missing as it is needed for removal in $t_8$ but could have been destroyed already by $t_2$.

*Strongly redundant data* Strongly redundant data element is an element that is produced (written), but never read afterwards, i.e., before the workflow is complete or the element is destroyed. In $\mathsf{CTL}^*$ terms we express this as

$$\mathsf{EF}\,[\mathsf{w}(d) \wedge \mathsf{X}\,[\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg\mathsf{r}(d))]].$$

Note that the use of the operator $\mathsf{X}$ is essential, as we do not want to capture the situation where the same task that writes to $d$ is also reading $d$, as this reading is, according to our convention, assumed to always happen before the writing. Similarly, we must put $\mathsf{d}(d) \wedge \neg\mathsf{r}(d)$ and not just $\mathsf{d}(d)$, as destroying data is the last operation performed by a task.

In Fig. 8, the elements $m$ and $n$ are redundant. Task $t_5$ creates $m$, which $t_8$ immediately destroys without reading (from) it. Similarly, $n$ is created in $t_1$ but only read in $t_6$, which need not ever execute.

*Weakly redundant data* Weakly redundant data strengthens weakens the condition for the strongly redundant data error. The bad scenario is when some data is produced, and then not read in *all* possible continuations. The formalization of this error differs from its strong counterpart by one letter only: the $\mathsf{A}$ requirement is added, since it is now required for all subsequent paths to have the undesired behavior. We thus have

$$\mathsf{EF}\,[\mathsf{w}(d) \wedge \mathsf{A}\,\mathsf{X}\ [\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg\mathsf{r}(d))]].$$

The element $m$ from Fig. 8 is weakly redundant. No path after $t_5$ can contain a transition that reads $m$. The other strongly redundant element from the figure, $n$, is not weakly redundant; there is still one path after $t_1$ in which $n$ is read, namely the path in which $t_6$ is taken.

*Strongly Lost data* Data is strongly lost if it is created, and then rewritten without being read in between. The principle behind this error is the same as for the strongly redundant data. The $\mathsf{CTL}^*$ formula capturing the error is

$$\mathsf{EF}\,[\mathsf{w}(d) \wedge \mathsf{X}\,[\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg\mathsf{r}(d))]].$$

Here we need to take $\mathsf{w}(d) \wedge \neg\mathsf{r}(d)$ instead of just $\mathsf{w}(d)$, as we assume that in a task reading always precedes writing. We take $\neg(\mathsf{r}(d) \vee \mathsf{d}(d))$ instead of just $\neg\mathsf{r}(d)$ to make sure that the data is indeed rewritten and not created again. The modality $\mathsf{X}$ is needed as without it any task that writes to $d$ but does not read $d$ would cause an error.

In Fig. 8, the elements $x$, $y$ and $z$ are strongly lost. The element $x$ is written in $t_2$ and immediately rewritten by the next task, $t_4$. The element

$y$ is lost if after $t_3$, where it is created, task $t_5$ is chosen. Finally, $z$ is lost if after $t_6$, the parallel task $t_4$ is executed.

*Weakly Lost data* Data is lost in the weak sense if it is created and then, in *all* following executions, overwritten without being read first. Therefore weakly lost data weakens the condition for the strongly lost data error. The $\mathsf{CTL}^*$ formula is the same as for the strongly lost data, but the operator $\mathsf{X}$ is replaced by $\mathsf{AX}$:

$$\mathsf{EF}\,[\mathsf{w}(d) \wedge \mathsf{A}\,\mathsf{X}\ [\neg(\mathsf{r}(d) \vee \mathsf{d}(d))\ \mathsf{U}\ (\mathsf{w}(d) \wedge \neg\mathsf{r}(d))]].$$

Among the three strongly lost data elements, only $y$ is also weakly lost. There is one path after $t_3$ in which $y$ is not written again, namely the path in which $t_6$ is taken.

*Inconsistent data* Data is inconsistent if a task is using this data while some other task (or another instance of the same task) in parallel is writing to this data or is destroying it. In $\mathsf{CTL}^*$ terms, we have:

$$\bigvee_{\substack{t \in T\,: \\ d \in \mathbf{r(t)} \cup \mathbf{data(grd(t))} \\ \cup\, \mathbf{w(t)} \cup \mathbf{d(t)}}} \mathsf{EF}\,[(\mathsf{exec}(t)\ \wedge \bigvee_{\substack{t' \neq t\,: \\ d \in \mathbf{w(t')} \cup \mathbf{d(t')}}} \mathsf{exec}(t'))\quad \vee\quad p_t \geq 2].$$

The last part of the formula captures the situation where the same task that writes to or deletes some data element is instantiated twice, in a different thread. In Fig. 8, the elements $z$ and $r$ are inconsistent. The element $z$ can be written by both $t_4$ and $t_6$ in parallel. Task $t_4$ can be reading $r$, while task $t_6$ is changing it in parallel.

The following correctness properties are related to data removal. They should be seen more as optimization objectives rather than strict correctness criteria. The properties become of high importance in environments that operate on large data like, e.g., in grids.

*Never destroyed* A data element is never destroyed if it is created but not destroyed afterwards. The principle of formulating this scenario is essentially the same as for the strongly lost data. We have:

$$\mathsf{EF}\,[\mathsf{w}(d) \wedge \mathsf{X}\,[\neg(\mathsf{d}(d) \vee \mathsf{w}(d))\ \mathsf{U}\ \mathsf{final}]].$$

The reason for writing $\neg(\mathsf{d}(d) \vee \mathsf{w}(d))$ and not just $\neg\mathsf{d}(d)$ is because rewriting is also destroying in some sense; the use of $\mathsf{X}$ is then necessary because $\mathsf{w}(d)$ and $\neg\mathsf{w}(d)$ can not both hold.

In our example from Fig. 8, the elements $b$, $z$, $r$ and $p$ are all never destroyed. There exists no task that destroys $z$, nor a task that destroys $r$. The element $b$ is only destroyed if $t_5$ executes instead of $t_6$. Finally, $p$ is created by the last task, $t_9$, that does not also destroy it.

*Twice destroyed* This error is similar to the strongly lost data error but concerns data deletion. A data element is twice destroyed if it is destroyed twice in a row without being created in between:

$$\mathsf{EF}\,[\mathsf{d}(d) \wedge \mathsf{X}\,[\neg\mathsf{w}(d)\ \mathsf{U}\ (\mathsf{d}(d) \wedge \neg\mathsf{w}(d))]].$$

Here again we use the assumption that, inside a task, data removal always takes place after data writing.

In Fig. 8, $u$ and $v$ are destroyed twice. The element $u$ can be destroyed first by $t_2$ and then again by $t_8$, or vice versa. If **pred** is false, then $v$ is destroyed in $t_5$ and then again in $t_9$.

*Not destroyed on time* This is an error when a task that is *always* the last to read some data is not also destroying this data. For a $t \in T$ that reads $d$ without destroying it this means that $d$ is never read again after $t$. An additional explanation needed here is related to the fact 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 $\mathsf{CTL}^*$ formula that captures this:

$$\bigvee_{t \in T : d \in \mathbf{r}(\mathbf{t}) \setminus \mathbf{d}(\mathbf{t})} \mathsf{AG}\,[\mathsf{exec}(t) \Rightarrow \mathsf{exec}(t)\ \mathsf{U}\ \mathsf{G}\,(\neg\mathsf{r}(d))].$$

In Fig. 8, only the elements $c$ and $x$ are destroyed on time.

Note that in some cases the error scenarios overlap. For example, if an element is twice destroyed, then it is also missing (cf. $u$ from Fig. 8), and an inconsistent data that is being written simultaneously by two tasks is also lost data (cf. $z$ from Fig. 8). To have this duality in some cases is intentional, as one might not always want to check a workflow for all the errors.

### 3.4 Verifying the properties

As explained before, $\mathsf{CTL}^*$ model checking is, in general, inefficient, and working in one of its subclasses is preferable. Moreover, tools supporting $\mathsf{CTL}^*$ are rare, while there is a plethora of $\mathsf{LTL}$ and $\mathsf{CTL}$ model-checkers accepting Petri nets as input [11, 9, 5]. In this section we show that all

the properties we introduced can, in fact, be checked with either an LTL or a CTL model-checker (or both). To achieve this we rely on, but do not show, the standard rewriting rules for CTL* [4].

The CTL* formula describing the no-dead-transitions property can also be seen as a set of CTL formulas, one for each transition $t$. As $\neg$EF rewrites to AG$\neg$, the negation of every such formula is an LTL formula. Therefore, the no-dead-transitions property can be checked with both an LTL and a CTL model-checker. Despite this fact, however, the property is hardly ever going to be proved by model checking; it can be more optimally checked while the underlying Kripke structure is generated.

The requirement for strict completion is both an LTL and a CTL formula. Strict completion with fairness is an LTL property when $\psi$ contains no state formulas. In addition, replacing F final by EF final we get an equivalent formula, that is CTL when $\psi$ is CTL. The CTL* formula for optional completion is a CTL formula for which there exists no equivalent LTL formula. The formula capturing eventual completion with $\psi$ is CTL when $\phi$ is CTL. If $\phi$ is a path formula containing no state formulas, the negation of the property is also LTL. Consequently, (ordinary) eventual completion is both a CTL and an LTL property.

In the CTL* formula for missing data error we can replace the second occurrence of $\neg\mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d))$ by $\mathsf{E}(\neg\mathsf{w}(d)\ \mathsf{U}\ (\mathsf{r}(d) \vee \mathsf{d}(d)))$ to obtain an equivalent CTL formula. Moreover, the negation of the original CTL* formula is LTL. In the strongly redundant data error we can replace X by EX to obtain an equivalent CTL formula; the negation of the original CTL* formula is also LTL. Replacing X by XA in the CTL* formula for the weakly redundant data error, the formula becomes CTL. There is no LTL formula expressing (the negation of) this property. In the strongly lost data error we can replace X by EX which leads to an equivalent CTL formula, while the negation of the original formula is LTL. The weakly lost data, on the other side, is a CTL property not expressible in LTL. The CTL* formula for the inconsistent data error can be seen as a set of CTL formulas. The negation of any formula from this set is an LTL formula. Note, however, that a check for inconsistent data only needs to identify the "bad" states. Therefore, this check be done while the Kripke structure is generated. The negation of the CTL* formula for the never destroyed property is LTL; replacing X by FX we obtain an equivalent CTL formula. Similarly to the strongly lost data, twice destroyed data error is both a CTL and an LTL property. The formula expressing the not destroyed on time error for a single transition is LTL. The above observations are summarized in Table 1.

**Table 1.** Model checking the correctness properties - summary

| CONTROL-FLOW REQUIREMENTS | |
|---|---|
| No-dead-transitions | LTL/CTL |
| Strong completion | LTL/CTL |
| Strong completion with fairness | LTL (when $\psi$ has no state formulas) / CTL (when $\psi$ is CTL) |
| Optional completion | CTL |
| Eventual completion with $\phi$ | LTL (when $\phi$ has no state formulas) / CTL (when $\phi$ is CTL) |
| Eventual completion | LTL/CTL |
| **DATA-FLOW ERRORS** | |
| Missing | LTL/CTL |
| Strongly redundant | LTL/CTL |
| Weakly redundant | CTL |
| Strongly lost | LTL/CTL |
| Weakly lost | CTL |
| Inconsistent | LTL/CTL |
| Never destroyed | LTL/CTL |
| Twice destroyed | LTL/CTL |
| Not destroyed on time | LTL |

## 4 Conclusions

Motivated by the fact that data-flow and control-flow in business workflows are not independent, we provided a framework for combined verification of these aspects. We identified several possible flaws and formalized them in terms of the temporal logics CTL*. In addition to extending, refining, and classifying the existing works that deal with similar issues, our paper provides a unifying framework for complete workflow analysis, using the well-known, stable, adaptable, and effective model-checking approach.

*Related work* As mentioned before, many researchers have been working on the topic of workflow verification already for years. It is impossible to give a complete overview of the related work here (see [2] for a survey of verification approaches focusing on the control-flow). Therefore, we only mention the approaches directly relevant to this paper, namely those 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 redundant data error, but no means for checking these errors is provided. Later, [17] conceptualized the errors

from [15] using UML diagrams, and gave supporting verification algorithms. This work was further extended and generalized in [18]. None of these approaches consider control-flow properties.

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

The work closest to ours is [7]. There also model checking is used to verify business workflows, from both 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, not destroyed. Control-flow errors are captured by means of strict completion with strong fairness, meaning that the workflow from Fig. 8c is not correct in their setting. Finally, [7] only considers LTL, so many of our correctness properties are not expressible there.

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 [16]. In this, totally different, application domain, concurrency issues are rarely treated and a systematic classification of errors is missing.

*Future work* In the future we will try to identify more errors. 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 by using an existing model-checker, e.g. [11], and finally generating a verification report for the workflow designer.

We also plan to investigate how structural techniques on Petri nets can be applied in our case and how to extend the correctness properties for the case when data elements are not only case-related, but could be shared among different cases.

## 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, 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.
3. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
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. J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Berlin, 2001.
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. 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.
10. A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.
11. Model-Checking Kit Home Page. http://www.informatik.uni-stuttgart.de/fmi/szs/tools/mckit/.
12. Pallas Athena. *Protos User Manual*. Pallas Athena BV, Plasmolen, The Netherlands, 2004.
13. F. Puhlmann and M. Weske. Interaction Soundness for Service Orchestrations. In A. Dan and W. Lamersdorf, editors, *Proceedings of Service-Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 302–313. Springer-Verlag, Berlin, 2006.
14. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflows 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. 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.
17. 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.
18. 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.

19. R. van der Toorn. *Component-Based Software Design with Petri nets: An Approach Based on Inheritance of Behavior*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.