Workflow Completion Patterns

Nikola Trčka Wil van der Aalst Natalia Sidorova
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: {n.trcka, w.m.p.v.d.aalst, n.sidorova}@tue.nl

***Abstract*— The most common correctness requirement for a (business) workflow is the *completion requirement*, imposing that, in some form, every case-instance of the workflow reaches its final state. In this paper, we define three workflow completion patterns, called the *mandatory*, *optional* and *possible completion*. These patterns are formalized in terms of the temporal logic CTL*, to remove ambiguities, allow for easy comparison, and have direct applicability. In contrast to the existing methods, we do not look at the control flow in isolation but include some data information as well. In this way the analysis remains tractable but gains precision. Together with our previous work on data-flow (anti-)patterns, this paper is a significant step towards a unifying framework for complete workflow verification, using the well-developed, stable, adaptable, and effective model-checking approach.**

I. Introduction

A *business process* is a set of *tasks* (atomic units of work) that need to be performed in a specific way to achieve a particular and well-defined business goal. Examples of business processes are a handling of an insurance claim, a booking of a flight, and a loan approval. In a loan approval process the goal is to grant or reject the loan; a typical task in this process is to check the client's credit history.

The concept of a *workflow* has been introduced to facilitate the documentation, management and execution of complex business processes. The workflow is a *model* of a business process, specifying the rules for desired coordination and cooperation of individual business tasks. A typical business workflow defines the 1) *control-flow* perspective (the ordering of tasks); 2) *data-flow* perspective (document sharing and exchange); and 3) the *resource* perspective, defining the (groups of) resources, such as humans, machines, or services, suitable for performing certain tasks.

Designing business workflow is similar to software programming, and thus an error-prone task. The presence of "bugs" in a business workflows may lead to angry customers, dissatisfied employees, legal issues, and increased production time and cost. Unfortunate situations are not uncommon; we do sometimes have a wrong book delivered, never receive our requested flight information, get two bills for our holiday trip, or hear about a business fraud discovered too late.

Control-flow analysis is the most widely employed type of workflow verification, and arguably the most important one. This analysis tries to answer questions like "Does task A ever happen?", "Is there a deadlock?", "Does task A always precede task B?", etc. Although many control-flow properties are domain specific, like e.g. the requirement that some task follows another, the properties like deadlock freedom, are applicable in any business context. Most of these generic requirements are considered with work-progress and can simply be combined into one property ensuring that the workflow (its every case-instance, to be precise) eventually reaches its final state. We call this unifying property the *workflow completion property*.

There already exist several verification techniques having the completion property as their central requirement (most of them appeared under the name *soundness* [1], [2]). The exact form of the completion requirement, however, changes dependent on the author or the context. Sometimes the workflow is allowed to reach its end state and leave some work behind, while sometimes no leftover "garbage" is allowed. Sometimes the workflow *must* always complete eventually, while sometimes only the existence of one path from the beginning to the end is required. There is a plethora of different (sub)conditions and combinations. Moreover, not all of these notions have been defined in the same setting, which makes their comparison difficult.

The absence of a systematic classification and of a unifying analysis method are not the only weaknesses of existing approaches to workflow completion verification. Another important weakness is that these approaches consider the control flow in isolation and completely ignore the other two perspectives of the workflow. Although it makes sense to abstract from resources, as they are external and dynamic in nature, the same does not hold for data. Ignoring the data perspective could, e.g., cause a deadlock error 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. Moreover, a workflow obviously cannot complete if some scheduled task is requiring a document that no other task has produced.

To overcome the limitations of the existing approaches this paper proposes an analysis framework based on a) workflow nets with data information, b) model-checking, and c) workflow completion patterns. A *WorkFlow net with Data* (WFD-net) is a special type of a Petri net with a clear start and end point, with annotations describing read/write data operations, and with data-dependant guards on transitions. The idea of this (conceptual) formal model is to fully cover the control-flow perspective while also including data information in

some limited form. The model supports the interplay of data elements but abstracts from their concrete values. This allows for a more precise, albeit still tractable, workflow analysis. Note that the industrial languages like Business Process Modeling Notation (BPMN), extended Event-driven Process Chains (eEPCs) and UML activity diagrams, are all examples of languages that can, from a pragmatic point of view, be translated to WFD-nets.

Assuming a WFD-net representation of our process, we define three *patterns* related to workflow completion (called *mandatory*, *optional*, and *possible completion* respectively), and show how the standard completion requirements from the literature fit into these patterns. The patterns are formalized in terms of the temporal logic CTL* [3] (and its subsets CTL and LTL). In this way we not only remove all ambiguities inherent to formulations in a natural language, but also automatically obtain a highly configurable (properties can be easily changed, added or removed) and stable (model-checking has been successfully used for years) verification setting, with excellent diagnostic features (model-checking provides error traces). Moreover, together with the data-flow (anti-)patterns we formulated in [14], our results enable a seamless integration of all aspects of workflow verification.

The structure of the rest of this paper is as follows. Section II gives some preliminaries (Petri nets, workflow nets, and the logic CTL*) and introduces workflow nets with data. Section III presents the actual completion patterns. In Section IV we conclude the paper and discuss future work.

II. Preliminaries

In this section we define the WFD-net model and present the temporal logic CTL*.

A. Workflow nets with data

WFD-nets are based on Petri nets and workflow nets, so we define these two models first.

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, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of *arcs*, called 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. A place p is called a *source* place if ${}^\bullet p = \emptyset$, and a *sink* place if $p^\bullet = \emptyset$.

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 defined as a mapping $m : P \to \mathbb{N}$. 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 $m(p) \geq 1$ for all $p \in {}^\bullet t$. An enabled transition t may *fire*, which results in a new marking m' defined by $m'(p) = m(p) + 1$ if $p \notin {}^\bullet t$ and $p \in t^\bullet$, $m'(p) = m(p) - 1$ if $p \in {}^\bullet t$ and $p \notin t^\bullet$, $m'(p) = m(p)$ otherwise. This firing is denoted as $m[t\rangle m'$.

The *reachability graph* of a marked Petri net is a labeled directed graph in which every node represents a reachable marking, and every arc indicates the firing of a transition. For a marked net (N, m_0), this graph is formally defined as the tuple $\langle S, \rightarrow\rangle$ where S and $\rightarrow$ 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 $(m, t, m') \in \rightarrow$. 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.

Workflow nets [1] impose syntactic restrictions on Petri nets to comply to the workflow concept.

Definition 2: A Petri net $N = \langle P, T, F\rangle$ is a *Workflow net (WF-net)* if it has a single source place start and a single sink place end, and if its every node (place or transition) is on a path from start to end (i.e. if $(\mathsf{start}, n) \in F^*$ and $(n, \mathsf{end}) \in F^*$ for all $n \in P \cup T$, where F^* is the reflexive-transitive closure of F).

Transitions in a WF-net are also called *tasks*. A *case* is a workflow instance, i.e., a marked WF-net in which the start place is marked with one token and all other places are empty. In this paper we study the completion property related to one single case in isolation, assuming that different cases are completely independent from each other.

A workflow net with data elements is a workflow net in which tasks can read from or write to some data element. A task can also have a (data dependent) guard that can block its execution. We formalize the concept of a guard first.

Definition 3: Let $\mathcal{D}$ be a set of data elements. A *predicate* (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 or the negation of a predicate. The set of all guards over $\mathcal{D}$ is denoted $G_\mathcal{D}$.

We now define workflow nets with data.

Definition 4: A tuple $\langle P, T, F, \mathcal{D}, \mathbf{r}, \mathbf{w}, \mathbf{grd}\rangle$ 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*, $\mathbf{r} : T \to 2^\mathcal{D}$ is the *reading data labeling function*, $\mathbf{w} : T \to 2^\mathcal{D}$ is the *writing data labeling function*, and $\mathbf{grd} : T \rightharpoonup G_\mathcal{D}$ is the *guarding function*, assigning guards to some transitions.

Fig. 1 shows a WFD-net representing a simplified loan approval process. The first task in the process is to register the request of a client. This task creates two data elements: c to store clients information and r for the actual request. In the next task the client's history is checked and a decision d is made. The guard client_ok(d) evaluates to true if d denotes a positive decision; otherwise it evaluates to false. In the positive case the task Approve executes, producing a document (a) describing the approved loan (amount, conditions, etc.). This document is then communicated to the client (task Inform client). The actual payment is done in parallel via the Utilize loan task that also produces the final report (stored in p). The two parallel branches are synchronized by the task Sync. In the case of a negative decision, a report is made (the same data element p is used) and sent to the client. Finally, regardless of the decision, the client's record is updated, based on the information given in the report. We implicitly assume that inside a task reading always precedes writing, so when the last task is executed the old version of c is overwritten and thus lost.

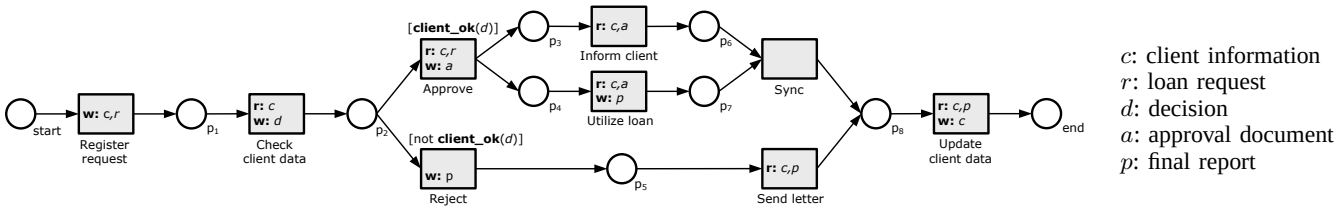


Fig. 1. A WFD-net representing a loan approval process

B. Temporal logic CTL*

The (state-based) temporal logic CTL* [3] is a powerful temporal logic combining linear time and branching time modalities. The logic is usually 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.

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

We now define the syntax of CTL* [3].

Definition 6: The classes Φ of CTL* *state formulas* and Ψ of CTL* *path formulas* are generated by the following grammar:

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

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

Validity of CTL* formulas is defined as follows.

Definition 7: We define when a CTL* state formula ϕ is *valid* in a state s (notation: $s \models \phi$) and when a CTL* path formula ψ is *valid* on a path π (notation: $\pi \models \psi$) by simultaneous induction as follows:

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

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

A CTL* state formula of the form $\mathsf{A}\psi$, where ψ contains no non-atomic state formulas, is a *Linear Temporal Logic* (LTL) formula. A CTL* state formula in which every subformula of the type $\psi \,\mathsf{U}\, \phi$ occurs in a pair with the quantifier A or E, is a *Computational Tree Logic* (CTL) formula.

C. From a WFD-net to a Kripke structure

The reachability graph of a WF-net can simply be seen as a Kripke structure if transition labels are ignored and if suitable atomic propositions are generated for every state using its underlying marking information. For a WFD-net, however, data information must also be incorporated as data can influence state reachability. In [14] we proposed a preprocessing step that converts a WFD-net into a WF-net with the same structure, but with explicit encoding of dependencies between data elements and guards. We do not elaborate on this transformation here (details are given in [14]), but assume for the rest of the paper that a Kripke structure representing the behavior of a WFD-net has been built, with atomic propositions being 1) $p \diamond i$, for $p \in P$, $i \in \mathbb{N}$ and $\diamond \in \{\leq, \geq, =\}$, valid in a state(=marking) m where $m(p) \diamond i$, and 2) $\mathsf{exec}(t)$, for $t \in T$, valid in a state where transition t is executing.[1]

III. Completion patterns

In this section we define three patterns for workflow completion: the *mandatory*, *optional*, and *possible completion* pattern. For each pattern we give:

1) Description with motivation.
2) Formalization in terms of a (parametric) CTL* formula. – The parameter τ will appear in all the patterns. It is a CTL* state formula representing the final state of the workflow, allowing the user to choose for the most appropriate notion of successful termination. One typical value for τ is $\mathsf{end} \geq 1$, which says that the completion of at least one workflow tread was successful (place end has at least one token). Another, more commonly used value for τ is $\mathsf{end} = 1 \wedge \bigwedge_{p \in P \setminus \{\mathsf{end}\}} (p = 0)$. This formula ensures that all parallel treads have been properly synchronized and that the final state is reached without any leftover work (place end is marked with exactly one token and all the other places are empty). Every pattern-instance

[1]The transformation of [14] splits every transition into its start and its end, allowing us to capture transition execution as a state property.

having this formula for τ will be referred to as *proper completion*.

3) Several instances, in form of concrete completion requirements, with relations to the existing notions from the literature.
4) Discussion on verification. – CTL^* model checking is, in general, inefficient, so working in one of its subclasses is preferable. Moreover, tools supporting CTL^* are rare, while there is a plethora of LTL and CTL model-checkers accepting Petri nets as input [11], [4]. For these reasons we show when the pattern can be rewritten to an equivalent LTL or CTL formula.
5) Comparison with the other two patterns.
6) Examples showing the difference with other patterns and between the different instances of the pattern.

We now proceed with pattern definitions.

Pattern 1: Mandatory completion

Description: Mandatory completion pattern captures the requirement that, starting from the initial state, *all* execution paths of the workflow must complete. It is a very restrictive pattern that is based on the intuitive idea that the workflow should not specify any non-completing behavior. Besides τ, the pattern has two other parameters. The first parameter is used to exclude some paths from consideration, and e.g. allow the completion property to hold in the presence of loops. The second parameter strengthens the completion property by imposing extra requirements on the completing paths, like e.g. that some transition is executed or that some data is always available.

Formalization in CTL^**:* For the path formula ψ that selects the "interesting" paths, the path formula φ denoting the additional requirement for completing paths, and the state formula τ representing the final state of the workflow, the pattern specifies that all paths satisfying ψ must also satisfy φ and eventually reach the final state described by τ. In CTL^* terms we write this as:

$$\mathsf{A}\,(\psi \Rightarrow \varphi \wedge \mathsf{F}\,\tau).$$

Instances: For $\psi = \mathrm{true}$ we have the *classical* mandatory completion property. The parameter ψ, however, is mostly used as a fairness assumption [7], to remove inadequate, i.e. unfair, infinite paths. The *strong fairness* assumption includes only those sequences in which an infinitely often enabled transition is infinitely often executed, and is formalized as $\psi = \vee_{t\in T}[\mathsf{GF}\,(\bigwedge_{p\in {}^\bullet t} p \geq 1) \Rightarrow \mathsf{GF}\,\mathsf{exec}(t)]$. The *weak fairness* assumption, requires only that continuously enabled transitions are infinitely often executed. Its formalization is the same as for the strong fairness but with the first GF replaced by FG.

In [8], workflow completion is required for all paths and no leftover work is allowed. It is thus proper mandatory completion with $\psi = \mathrm{true}$. The completion property of [6] adds strong fairness (for all transitions) to the requirement of [8]. In [9], the formula $\mathsf{AG}(\textit{initial state} \Rightarrow \mathsf{AF}\,\textit{goal state})$ (that does not fit into our patterns) is used to describe generic workflow properties. If there is only one initial state (as in our case), and the goal is to reach states that satisfy τ (and τ implies that no transition is enabled), the formula can simply be rewritten to $\mathsf{AF}\tau$, which is mandatory completion. The *missing data* anti-pattern from [14] captures the data-flow error where a task that reads from some data precedes a task in which this data is created. The negation of this anti-pattern can also be seen as mandatory completion with $\psi = \mathrm{true}$, $\tau = \mathrm{true}$ and $\varphi = (\neg\mathsf{read}(d)\ \mathsf{U}\ \mathsf{write}(d))$. Here $\mathsf{read}(d)$ abbreviates $\bigvee_{t:d\in\mathbf{r(t)}\cup\mathbf{data(grd(t))}} \mathsf{exec}(t)$ and $\mathsf{write}(d)$ is defined similarly.

Examples: The WF-net from Fig. 2a has the proper mandatory completion property with $\varphi = \mathsf{F}\mathsf{exec}(t_3)$ (and thus also with $\varphi = \mathrm{true}$). 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 puts a token in place end and (properly) completes the workflow.

The WF-net from Fig. 2b does not satisfy the mandatory completion requirement when $\psi = \mathrm{true}$ as the loop $t_2, t_4, t_2, \ldots$ might never be exited. This net, however, does have the proper mandatory completion property with the strong (but not weak) fairness assumption implying that t_4 must eventually change d in such a way that pred evaluates to true.

It is not hard to see that the WFD-net from Fig. 2c satisfies the requirement of proper mandatory completion with e.g. $\varphi = (\neg\mathsf{read}(d)\ \mathsf{U}\ \mathsf{write}(d))$. In this perfectly realistic example, however, if data information is removed, the sequence t_1, t_2, t_5 becomes possible (the information that $\mathsf{pred(d)}$ is then true in both parallel branches is lost) which leads to a deadlock. The example clearly illustrates the importance of including data information into control-flow verification.

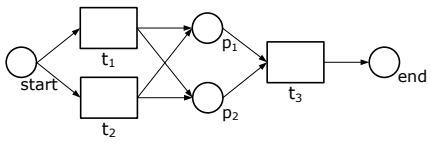
Finally, Fig. 2d is an example of mandatory completion with $\psi = \varphi = \mathrm{true}$ and $\tau = (\mathsf{end} = 1)$. The path t_1, t_3, t_4 leads to the end state but leaves some work behind (token in p_1); the completion is, therefore, not proper.

Verification: If ψ and φ contain only atomic state formulas, and τ is not of the form $\mathsf{E}\tau'$, then mandatory completion is an LTL property. For $\psi = \mathrm{true}$ and $\varphi = \mathrm{true}$ the formula for mandatory completion is clearly also an CTL formula. Moreover, if ψ is of the form $\mathsf{G}\psi'$, neither ψ' nor τ are of the form $\mathsf{E}\rho$, and $\mathsf{F}\tau$ implies φ, then the formula for mandatory completion can be rewritten to the equivalent CTL formula $\mathsf{AF}(\neg\psi' \vee \tau)$.

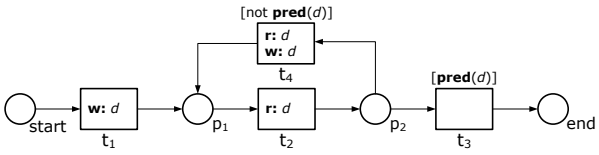
Pattern 2: Optional completion

Description: Optional completion does not require that every path must complete, but that from *every state* of the workflow there *exists* a path to completion. The focus is thus shifted from (initial) paths to states, which also eliminates the need for an additional path-restricting parameter like ψ. The pattern, however, still has one parameter for imposing extra requirements on states. This parameter is typically used to check whether a place always has less than n tokens, or whether some transition could be executed in future.

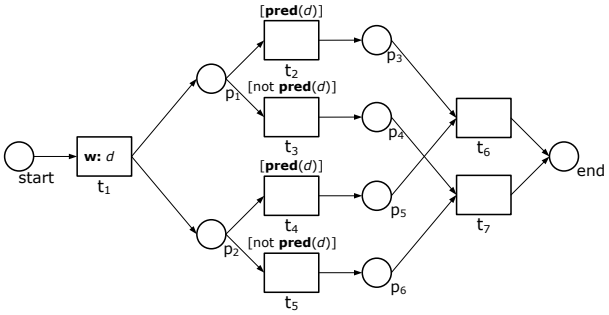
The motivation for the optional completion pattern comes from the fact that mandatory completion cannot satisfactorily



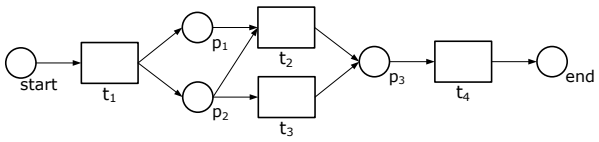
a) Proper mandatory completion with $\varphi = \mathsf{F}\mathsf{exec}(t_3)$



b) Proper mandatory completion with strong fairness



c) Proper mandatory completion where data information is important (without data a deadlock would be reported)



d) Mandatory completion with $\tau = (\mathsf{end} = 1)$ that is not proper

Fig. 2. Mandatory completion pattern

deal with arbitrary loops (see example below), and that a better and more direct capturing of progress is needed.

Formalization in CTL^**:* Let ϕ be a CTL^* state formula representing an extra requirement for states. Then every state must satisfy ϕ and be the start of a completing path:

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

Instances: The notion of *weak soundness* [10] corresponds to proper optional completion with $\phi = \mathrm{true}$. *Classical soundness* [1] adds the requirement that there are no dead tasks, i.e., that every task can potentially be executed. This can be seen as proper optional completion where $\phi = [\mathsf{start} = 1 \Rightarrow \bigwedge_{t \in T} \mathsf{EF}\,\mathsf{exec}(t)]$. In the notion of *lazy soundness* [12] tokens may be left behind as long as the place end is marked precisely once. This can be expressed as optional completion with $\phi = (\mathsf{end} \leq 1)$ and $\tau = (\mathsf{end} = 1)$. Finally, [13] strengthens the notion of classical soundness by requiring that all places are *safe*, i.e. that they always hold at most one token. This notion, called *safe classical soundness*, is (without the requirement for no dead tasks) proper optional completion with $\phi = \wedge_{p \in P}(p \leq 1)$.

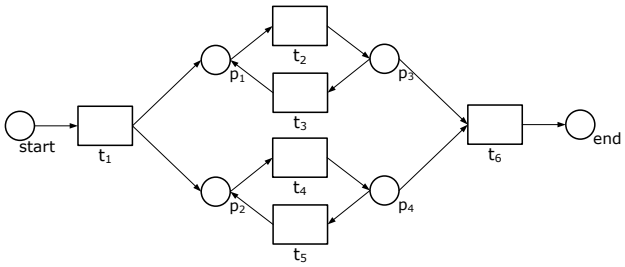


Fig. 3. Proper optional completion with $\phi = \mathsf{F}\,\mathsf{exec}(t_6)$

Examples: Fig. 3 shows a WF-net that has the optional, but not the mandatory, completion property. Note that neither of the two standard fairness assumptions for mandatory completion help here as t_6 is never enabled in the infinite sequence $t_2, t_3, t_4, t_5, t_2, \ldots$.

Comparison: Optional completion is less restrictive than mandatory completion when $\psi = \mathrm{true}$ and φ implies $\mathsf{G}\phi$. Note however that optional completion still captures all important properties like livelocks and deadlocks. For $\psi = \mathrm{true}$ and $\varphi = \mathsf{G}\phi$ the two notions agree on workflows without loops.

Verification: Optional completion is a CTL^* formula. When $\phi = \mathrm{true}$, the formula can be rewritten to $\mathsf{AGEF}\,\tau$, which is CTL. For this formula there exists no equivalent LTL formula (unless all paths are finite, in which case we can equivalently use mandatory completion).

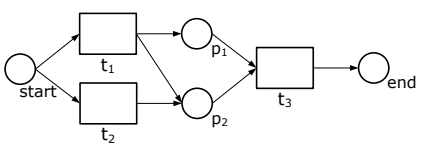
Pattern 3: Possible completion

Description: Possible completion is the least restrictive form of completion; it only requires that from the initial state there *exists* a (special) path towards completion. The notion, therefore, allows for deadlocks and livelocks in the workflow. The main idea behind the requirement is the assumption that users always make intelligent choices and avoid bad situations. The pattern has one parameter for putting extra requirements on the completing path.

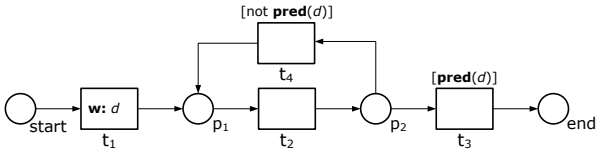
Formalization in CTL^**:* Let φ be a path formula denoting some additional requirement for the completing path, and let, as before, τ represent the final state of the workflow. Then the possible completion pattern can be expressed as: In the beginning there is at least one completing path satisfying φ. In CTL^* terms we write

$$\mathsf{E}\,(\varphi \wedge \mathsf{F}\,\tau).$$

Instances: The notion of *Easy soundness* [13], [2] corresponds to proper possible completion with $\varphi = \mathrm{true}$. *Relaxed soundness* [5] requires that for each transition there is at least one execution towards completion. This amounts



a) Proper possible completion with $\varphi = \mathsf{Fexec}(t_3)$



b) Proper possible completion where guards restrict behavior

Fig. 4. Possible completion

to checking proper possible completion with $\varphi = \mathsf{Fexec}(t)$, for each transition t.

Examples: The workflow from Fig. 4a does not have the optional completion property as the execution of t_2 leads to a deadlock. It does, however, satisfy the requirement for proper possible completion with $\varphi = \mathsf{exec}(t_3)$ (and thus with $\varphi = \mathrm{true}$ too) as the sequences t_1, t_3 leads to (proper) completion.

The workflow in Fig. 4b illustrates proper possible completion with $\varphi = \mathsf{Fexec}(t_2)$. If t_1 sets the value of to pred to true, the sequence t_1, t_2, t_3 is completing properly. The net, however, does not have the optional completion property as it enters the livelock $t_2, t_4, t_2, \ldots$ when the value of pred is false. Note that without the data information this workflow would be reported as satisfying the mandatory completion requirement with strong fairness.

Finally, Fig. 5 shows a workflow that does not complete according to the possible completion pattern; task t_3 can never be executed as it needs a token in *both* p_1 and p_2.

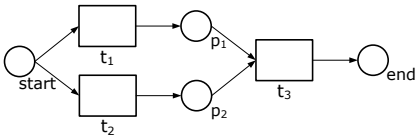


Fig. 5. A workflow that does not have the possible completion property

Comparison: Possible completion is implied by optional completion whenever φ is implied by $\mathsf{G}\phi$. The opposite holds only (in the non-interesting case) when there are no explicit choices in the workflow and φ implies $\mathsf{G}\phi$.

Verification: The formula capturing possible completion is a CTL formula when $\varphi = \mathrm{true}$. If φ is a path formula containing no state formulas of the form $\mathsf{E}\varphi'$, then the negation of the property is in LTL.

Remark 1: We could now also define the fourth pattern that would be the "state based" version of possible completion (like optional completion is for mandatory completion). For this pattern, however, we found no practical application.

IV. Conclusions and Future Work

We defined three generic patterns for representing the workflow completion requirement, and we formalized these patterns in terms of CTL*. We showed that the patterns encompass most of the variants of the completion requirement described in the literature. In many cases, pattern-instances were either CTL or LTL formulas, which immediately enables the use of many tools available on the market.

Our method does not fully abstract from data but includes information about data reading, data writing, and guards. In this way we keep the size of the system tractable, but increase the precision of the analysis. Although our method can still give false positives and false negatives, the number of cases where this happens is less compared to the existing methods which do not take data into account.

We are in the process of making a CTL* based verification tool for WFD-nets, aiming at a framework for flexible, adaptive and complete workflow verification. The patterns of this paper, and the patterns for data-flow errors we defined in [14], are important steps in this process.

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] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

[4] CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. http://wiki.daimi.au.dk/cpntools/.

[5] 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.

[6] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering Methodology*, 15(1):1–38, 2006.

[7] N. Francez. *Fairness*. Springer, New York, 1987.

[8] C. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater. Model Checking of Workflow Schemas. In *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC'00)*, pages 170–181, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[9] J. Koehler, G. Tirenni, and S. Kumaran. From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods. In *6th International Enterprise Distributed Object Computing Conference (EDOC'02), 17-20 September 2002, Lausanne, Switzerland*, pages 96–106. IEEE Computer Society, 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] F. Puhlmann and M. Weske. Investigations on Soundness Regarding Lazy Activities. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 145–160. Springer-Verlag, Berlin, 2006.

[13] 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.

[14] N. Trčka, W.M.P. van der Aalst, and N. Sidorova. Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows. In *21st International Conference on Advanced Information Systems (CAiSE'09)*, 2009. LNCS. To appear.