Workflow Soundness Revisited: Checking Correctness in the Presence of Data While Staying Conceptual

Natalia Sidorova, Christian Stahl, and Nikola Trčka

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{N.Sidorova,C.Stahl,N.Trcka}@tue.nl

Abstract. A conceptual workflow model specifies the control flow of a workflow together with abstract data information. This model is later on refined to be executed on an information system. It is desirable that correctness properties of the conceptual workflow would be transferrable to its refinements. In this paper, we present *classical workflow nets extended with data operations* as a conceptual workflow model. For these nets we develop a novel technique to verify *soundness*. This technique allows us to conclude whether at least one or any refinement of a conceptual workflow model is sound.

1 Introduction

Information systems are a key technology in today's organizations. Prominent examples of information systems are Enterprise Resource Planning Systems and Workflow Management Systems. Processes are the core of most information systems [9]. They orchestrate people, information, and technology to deliver products. In this paper, we focus on *workflows*—processes that are executed by an IT infrastructure.

A workflow is usually iteratively designed in a bottom-up manner. First the control flow of the workflow is modeled. The control flow consists of a set of coordinated tasks describing the behavior of the workflow. Later the control flow is extended with some data information. The resulting model is an abstract or *conceptual workflow model*, which is typically constructed by a business analyst. This conceptual model can be used for purposes of documentation, communication, and analysis. It may abstract from concrete data values, such as the condition of an if-then-else construct, and it does usually not specify how concrete data values are stored.

To actually execute this workflow on a Workflow Management System, the conceptual workflow model is instantiated with full details, a task typically done by business programmers (who often have insufficient background knowledge of the process) and not by the business analysts themselves. For instance, the business programmer specifies concrete data values and how they are stored.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 530–544, 2010.

Modeling both abstract and executable workflows is supported by industrial workflow modeling languages available on the market.

Designing a workflow model is a difficult and error-prone task even for experienced modelers. For a fast and thus cost-efficient design process it is extremely important that errors in the model are detected during the design phase rather than at runtime. Hence, verification needs to be applied at an early stage—that is, already on the level of the conceptual model.

Formal verification of a conceptual workflow model imposes the following two challenges. First, it requires an adequate formal model. With adequate we mean that the model captures the appropriate level of abstraction and enables efficient analysis. So the question is, *how can we formalize commonly used conceptual workflow models?* Second, verification must not be restricted to the control flow, but should also incorporate available data information. Thereby the conceptual workflow model may specify abstract data values that will be *refined* later on. Here the question is, *can we verify conceptual workflows in such a way that the results hold in* any *possible data refinement* (i.e., in all possible executable versions)?

In this paper, we investigate these two questions. We focus on one of the most important requirements for workflow correctness, namely the *soundness property* [1]. Soundness guarantees that every task of the workflow can be potentially executed (i.e., it is not dead), and that the workflow can always terminate (i.e., it is free of deadlocks and livelocks). However, current techniques for verifying soundness are restricted to control flow only.

To answer the first question, we propose *workflow nets with data* (WFD-nets) as an adequate formalism for modeling conceptual workflows. A WFD-net is basically a workflow net (i.e., a Petri net tailored towards the modeling of the control flow of workflows) extended with conceptual read/write/delete data operations. WFD-nets generalize our previous model in [18,19] by means of supporting arbitrary guards (previously only predicate-negation and conjunctions were allowed). As a second contribution, we develop a novel technique for *analyzing soundness of WFD-nets*. Unlike existing approaches which could give false positives (i.e., the analysis gives sound, but the workflow is actually unsound when the data information is refined) or false negatives (i.e., the analysis gives unsound, but the workflow with refined data is sound), our proposed technique gives neither false positives nor false negatives. It is based on may/must semantics [14] that guarantees the results to be valid in *any* concrete data refinement. If a WFD-net is proven must-sound, then it is sound in any possible data refinement; if it is not may-sound, then no data refinement can make it sound. In case where a WFD-net is may-sound but not must-sound, our approach gives an honest answer "I do not know; it is sound in some data refinement and unsound in some other". We interpret WFD-net as hyper transition systems, not as standard may/must transition systems. Doing this we achieve much better precision (i.e., less "I do not know" answers).

The paper is structured as follows. In Sect. 2, we show the potential problems with soundness verification for conceptual workflow models by means of two

examples. Afterwards, we introduce the conceptual workflow model in Sect. 3 and its semantics in Sect. 4. In Sect. 5, we define soundness for workflow nets with data in the may/must setting. Finally, Sect. 6 draws the conclusion and discusses future work.

2 Motivating Examples

We illustrate the idea of modeling conceptual workflows with WFD-nets on the WFD-net in Figure 1 modeling a shipping company. Ignoring the transition guards (shown within squared brackets above transitions), and the read and write operations (denoted by `rd` and `wt` inside the transitions), Figure 1 depicts an ordinary workflow net that consists of 10 places (depicted as circles) and 9 transitions (depicted as squares). There are two distinguished places `start` and `end` modeling the initial and the final state, respectively.

Initially there is a token on `start`. The shipper receives goods from a client (`receive good`) to be shipped. In the model, transition `receive good` writes the data of the client (`cl`), the goods (`gds`), and the destination of the goods (`ads`). Then the shipper calculates the price (`calculate price`). Afterwards the shipment department and the customer adviser execute their tasks concurrently. If the price of the goods is high (i.e., `isHigh(price)` evaluates to true; the exact bound is left unspecified), express shipment is used (`ship express`). Otherwise, the cheaper standard shipment is used (`ship normal`). Based on the same condition, the customer advisor either calculates a bonus (`calculate bonus`) for the client and registers this bonus (`register bonus`) or no bonus is calculated (`no bonus`). In addition, clients sending goods of a high price are notified about their bonus and the shipment (`inform by call`), other clients receive only a notification of the shipment (`inform by mail`).

Clearly, this WFD-net is sound, i.e., starting with a (colored) token on `initial` it is always possible to reach a marking where only one token is on place `end`. In contrast, if we abstract from data and consider only the underlying workflow net, the net may deadlock. For example, without data the shipment department may decide to use express shipment, but the customer advisor does not calculate any bonus. This yields a token in `p5` and in `p8`, and the net gets stuck. This shows that ignoring data information in verification can lead to obtaining false negatives.

Suppose now that instead of the same predicate `isHigh(price)`, two possibly different predicates (say `isHighLeft(price)` and `isHighRight(price)`) are used in the left and the right part of the workflow (this is realistic, because these parts correspond to two different departments of the shipper). As we did not change the control flow, the classical WF-net method would still say that the workflow is not sound. Our previous work [19] on soundness of (simplified) WFD-nets would give the same verdict: the workflow is not sound due to the possibility that the predicates can have different truth values. With the methods we will introduce in this paper we will be able to give the correct answer: "I do not know—the workflow is sometimes sound (when `isHighLeft(price)`

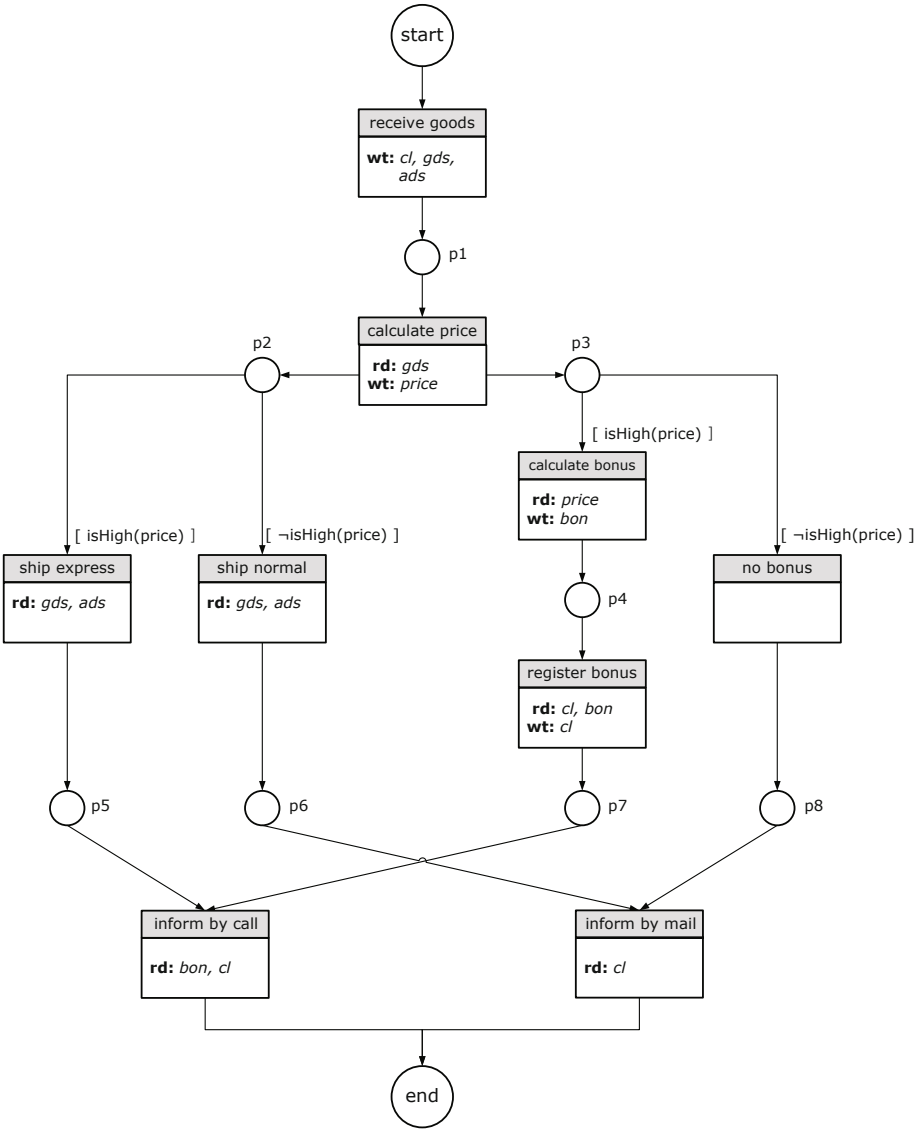

Fig. 1. WFD-net modeling a shipper – The workflow is sound, but verification ignoring data leads to the verdict "unsound"

and `isHighRight(price)` always evaluates to the same values), and sometimes it is not (when `isHighLeft(price)` and `isHighRight(price)` valuations can differ)".

It is also possible to construct examples where the verification of classical WF-nets (without data information) would give a false positive, and our previous work on WFD [19] would give a false negative. For instance, consider the loop

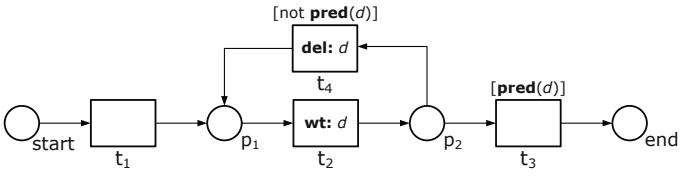

Fig. 2. WFD-net that is (non-)sound in some, but not in all, data refinements

in Figure 2 that is exited when some predicate depending on a data element d evaluates to true. The data element d is initialized inside this loop (transition t_2) and it is deleted (denoted by **del** inside transition t_4) and written in every loop cycle. In this case, proper termination is only possible if d *eventually* sets the exiting predicate to true. Therefore, the correct answer is again, "I do not know", as the soundness property depends on the concrete data refinement. However, when using classical WF-nets, the data information is ignored and the possibility to exit the loop is considered to be available, which results in the verdict that the WF-net is sound. Similarly, the previous WFD-net method would see that there is a possible bad execution sequence (namely the infinite looping) and would report non-soundness.

Remark 1. It is important to note that the problems addressed in this paper are independent of the actual version of workflow soundness definition used. Although we restrict ourselves to classical soundness, ignoring data when considering other notions of soundness can result in obtaining false positive or false negative answers. If we, for example, consider relaxed soundness (requiring that for every task there is a completing path executing this task) (see [8]), the workflow in Figure 1 would be correctly reported relaxed sound. However, if we used different predicates in the left and in the right side, the workflow was not relaxed sound. The verification result on the workflow without data is then obviously incorrect. Moreover, by swapping the guards on the right side of the net in Figure 1 we construct a workflow that has no completing execution at all. Note that also the unfolding of a workflow to have the data information incorporated into the control-flow (by the means of [18,19]) does not help in this situation either: Consider again the example in Figure 2. Unfolding the workflow, we obtain a relaxed sound net. It is, however, obvious that there are data refinements that certainly prevent t_4 from being ever executed. Similar problems arise with all other soundness notions. These problems can also be addressed by using the may/must approach we are going to present in this paper.

3 Workflow Nets with Data

Given the motivation for incorporating data into workflow nets, this section formally defines workflow nets with data (WFD-nets). WFD-nets are based on Petri nets and workflow nets, so we define these two models first.

Definition 1 (Petri net). *A* Petri net $N = \langle P, T, F \rangle$ *consists of two disjoint non-empty, finite sets* P *of* places *and* T *of* transitions *and of a* flow relation $F \subseteq (P \times T) \cup (T \times P)$.

For a transition $t \in T$, we define the *pre-set of* t as ${}^\bullet t = \{p \mid (p,t) \in F\}$, and the *post-set of* t as $t^\bullet = \{p \mid (t,p) \in F\}$. Analogously we define the pre-set ${}^\bullet p$ and the post-set $p^\bullet$ for a place $p \in P$. 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*, depicted as black dots. A state of a Petri net, called a *marking*, is a 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 marking m above we have, e.g., $m \leq [3p + 2q + r]$ and $m + [q + 3r] = [2p + 2q + 3r]$. A pair (N, m), where N is a Petri net and m is a marking, is a *marked* Petri net.

A transition $t \in T$ is *enabled* at a marking m, denoted by $m \xrightarrow{t}$, if $m \geq {}^\bullet t$. An enabled transition t may *fire*, which results in a new marking m' defined by $m' = m - {}^\bullet t + t^\bullet$. This firing is denoted as $m \xrightarrow{t} m'$.

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 (WF-net). *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 every place and every 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 called *tasks*. A *case* is a workflow instance, i.e., a marked WF-net in which the place start is marked with one token and all other places are empty. Executing a workflow means to create a running instance of this workflow. Such an instance is a called a *case*. Several cases of a workfow may coexist. Cases are assumed to be completely independent from each other (they only possibly share resources). Hence, each case is modeled as a copy of the corresponding workflow net N. We refer to the properties of $(N, [\mathsf{start}])$ as the properties of N.

Example 1. Ignoring the transition guards and the read and write operations, Figure 1 depicts a WF-net. Places start and end are the source place and the sink place, respectively. Clearly, every place and transition is on a path from start to end. Each transition, such as `Inform by call`, models a task. The pre-set of `Inform by call` is the set {p5, p7}, and the post-set is the set {end}.

Adding data information. A workflow net with data elements is a workflow net in which tasks can read from, write to, or delete data elements. A task can

also have a (data dependent) guard that blocks its execution when it is evaluated to false.

We assume a finite set $\mathcal{D} = \{d_1, \ldots, d_m\}$ of *data elements*, and we fix a set of predicates $\Pi = \{\pi_1, \ldots, \pi_n\}$. We also assume a function $\ell : \Pi \to 2^{\mathcal{D}}$, called the *predicate labeling function*, that assigns to every predicate the set of data elements it depends on. When $\ell(\pi) = \{d_1, \ldots, d_n\}$ for some predicate $\pi \in \Pi$, we sometimes write $\pi(d_1, \ldots, d_n)$ to emphasize this fact. A *guard* is constructed from predicates by means of the standard Boolean operations; the set of all guards (over Π) is denoted by $\mathcal{G}_\Pi$. The function ℓ naturally extends to guards.

We now define a workflow net with data as a WF-net where every transition t is annotated with at most four sets: a set of data elements being read when firing t, a set of data elements being written when firing t, a set of data elements being deleted when firing t, and a transition guard. Note that we do not explicitly consider the update of data elements, because this is simply the combination of read and write at the same transition.

Definition 3 (WFD-net). *A* workflow net with data *(WFD-net)* $N = \langle P, T, F, \mathrm{rd}, \mathrm{wt}, \mathrm{del}, \mathrm{grd}\rangle$ *consists of a WF-net* $\langle P, T, F\rangle$*, a* reading data labeling function $\mathrm{rd} : T \to 2^{\mathcal{D}}$*, a* writing data labeling function $\mathrm{wt} : T \to 2^{\mathcal{D}}$*, a* deleting data labeling function $\mathrm{del} : T \to 2^{\mathcal{D}}$*, and a* guard function $\mathrm{grd} : T \to \mathcal{G}_\Pi$*, assigning guards to transitions.*

Example 2. An example of a WFD-net is the workflow of a shipper in Figure 1. Its data elements are $\mathcal{D} = \{\texttt{cl}, \texttt{gds}, \texttt{ads}, \texttt{price}, \texttt{bon}\}$. Consider transition `calculate bonus`, for instance. The labeling functions are $\mathrm{rd}(\texttt{calculate bonus}) = \{\texttt{price}\}$, $\mathrm{wt}(\texttt{calculate bonus}) = \{\texttt{bon}\}$, $\mathrm{del}(\texttt{calculate bonus}) = \emptyset$, and $\mathrm{grd}(\texttt{Ship normal}) = \neg\texttt{isHigh(price)}$.

The next section assigns formal semantics to WFD-nets.

4 Semantics of WFD-nets

The model of WFD-nets is a conceptual model, a schema for characterizing several executable workflows. In this section we introduce a special semantics for WFD-nets that is based on hyper transition systems [14,17] and allows us to capture all possible refinements of a WFD-net in one graph.

4.1 Behavior of WFD-nets

In a WFD-net data values are not specified, but we can distinguish non-created data values from created ones. In our semantics we choose to keep the exact value for the predicates in a state. Predicates and guards can be evaluated to true, false, or undefined (if some data element assigned to them does not have a value). This is formalized by the three abstraction functions assigning abstract values to the data elements, the predicates, and the guards, respectively. Function $\sigma_{\mathcal{D}} : \mathcal{D} \to \{\top, \bot\}$ assigns to each data element $d \in \mathcal{D}$ either $\top$ (i.e., defined value) or

$\perp$ (i.e., undefined value); Function $\sigma_\Pi : \Pi \to \{\mathsf{T}, \mathsf{F}, \perp\}$ assigns to each predicate one of the values true, false, and undefined. A consistency requirement that $\sigma(\pi) = \perp$ whenever $\sigma(d) = \perp$ and $d \in \ell(\pi)$ is imposed. A pair $\sigma = (\sigma_\mathcal{D}, \sigma_\Pi)$ is called a *state*, and the set of all states is denoted by Σ. We use the following simplified notation: $\sigma(d) = \sigma_\mathcal{D}(d)$ for $d \in \mathcal{D}$, and $\sigma(\pi) = \sigma_\Pi(\pi)$ for $\pi \in \Pi$. As a guard $g \in \mathcal{G}_\Pi$ is built from Boolean operations, $\sigma(\text{grd}) \in \{\mathsf{T}, \mathsf{F}, \perp\}$ can be evaluated with the help of functions $\sigma_\mathcal{D}$ and σ_Π.

The next definition lifts the definition of a state of a WF-net to a WFD-net. We refer to a state of a WFD-net as a configuration[1]. A configuration consists of a marking m of a WFD-net and a state.

Definition 4 (Configuration). *Let $N = \langle P, T, F, \text{rd}, \text{wt}, \text{del}, \text{grd}\rangle$ be a WFD-net. Let m be a marking of N, and let σ be as defined above. Then, $c = \langle m, \sigma\rangle$ is a* configuration *of N. The* start configuration *of N is defined by $\langle[\mathsf{start}], (\{d_1 \mapsto \perp, \ldots, d_n \mapsto \perp\}, \{\pi_1 \mapsto \perp, \ldots, \pi_n \mapsto \perp\})\rangle$. With Ξ we denote the set of all configurations, and $C_f = \{\langle[\mathsf{end}], \sigma\rangle \mid \sigma \in \Sigma\}$ defines the set of* final configurations.

In the initial configuration, only place start is marked, all data elements are undefined, and all predicates are evaluated to undefined. A configuration is a final configuration if it contains the final marking [end].

Example 3. The initial configuration of the shipper in Figure 1 is defined to be $\langle[\mathsf{start}], \sigma\rangle$, where σ assigns $\perp$ to all data elements `cl`, `gds`, `ads`, `price`, `bon`. In addition, predicate `isHigh(price)` is $\perp$. Note that in Figure 1 no data element is deleted. However, we assume that upon reaching a final configuration (i.e., the case is completely executed), all data elements are deleted.

As Definition 4 lifts the notion of a state of a WF-net to a configuration of a WFD-net, we have to define the behavior of a WFD-net. For this purpose, we define when a transition t of a WFD-net N is enabled at a configuration $c = \langle m, \sigma\rangle$ of N.

The enabling of a transition t requires two conditions to be fulfilled. The first condition takes the control flow into account and requires that transition t must be enabled at marking m. The second condition considers the data values in configuration c. Any data element that is read by t or that is assigned to a predicate of t must be defined. In addition, the guard of t must evaluate to true.

An enabled transition t may fire. Firing of t changes the marking as well as the values of the data elements that have been written or deleted. As we do not know the concrete operations nor the values of the predicates, we have to consider any evaluation of the predicates. Hence, the firing of t yields a *set* of successor configurations $\langle m', \sigma'\rangle$. Each of these successor configurations has a marking m', where firing t at marking m yields marking m'.

On the data level, we assign undefined (i.e., $\perp$) to each data element d that has been deleted when firing t, and we assign undefined to each predicate that

[1] The meaning of the term *configuration* here is "a state that includes data information", and not the one related to configuring processes.

contains a data element that has been deleted. The reason is that reading always precedes writing, and writing always precedes deleting. Thus, no matter whether this data element has been also written, it is undefined after the firing of t. In addition, we assign defined (i.e., $\top$) to each data element d that has been written and not deleted when firing t, and evaluate each predicate that contains at least one data element that has been written and no data elements that have been deleted. The different evaluations of the predicates actually result in a set of successor configurations. This is formalized in the following definition.

Definition 5 (Firing rules for WFD-nets). *Let* $N = \langle P, T, F, \mathrm{rd}, \mathrm{wt}, \mathrm{del}, \mathrm{grd}\rangle$ *be a WFD-net. A transition* $t \in T$ *of* N *is* enabled *at a configuration* $c = \langle m, \sigma\rangle$ *of* N *if* $m \xrightarrow{t}$*, all data elements* $d \in \mathrm{rd}(t)$ *are defined, all data elements assigned to any predicate occurring in the transition guard* $\mathrm{grd}(t)$ *of* t *are defined, and* $\sigma(\mathrm{grd}(t)) = \mathsf{T}$*. Firing* t *yields a set* $C \subseteq \Sigma$ *of configurations with* $C = \{\langle m', \sigma'\rangle \mid m \xrightarrow{t} m' \wedge (\forall d \in \mathrm{del}(t) : \sigma'(d) = \bot \wedge \forall \pi \in \Pi : d \in \ell(\pi) \implies \sigma'(\pi) = \bot) \wedge (\forall d \in \mathrm{wt}(t) \setminus \mathrm{del}(t) : \sigma'(d) = \top \wedge (\forall \pi \in \Pi : \forall \bar{d} \in \ell(\pi) \setminus \{d\} : \sigma(\bar{d}) = \top) \implies \sigma'(\pi) \in \{\mathsf{T}, \mathsf{F}\})\}$ *and is denoted by* $c \xrightarrow{t} C$*.*

Example 4. Consider transition `calculate price` in Figure 1. Suppose there is a token in place `p1`. Transition `calculate price` is enabled if data element `gds` is defined. Firing this transition means that the token in `p1` is removed, and a token in `p2` and in `p3` is produced. In addition, `calculate price` takes data element `gds` as its input and stores its output in data element `price`. We implicitly assume that inside a task reading always precedes writing, and writing always precedes deleting. As `price` did not have a value before the occurrence of `calculate price`, a new value of `price` is created (otherwise it would have been updated). Moreover, as `price` is assigned to predicate `isHigh(price)`, this predicate is evaluated to either true or false, yielding two configurations.

4.2 Reachability

Definition 5 defines the semantics of firing a single transition. Now we extend the firing of a single transition to sequences of transitions. In other words, we define the set of reachable configurations of N. To take into account that we do not know the concrete values of predicates a priori, we define may- and must-reachability. *May-reachability guarantees that the reachability holds in* at least one *data refinement, whereas must-reachability guarantees that the reachability holds in* every *data refinement.*

Given a configuration c, a *may-step* from c specifies the existence of a successor configuration c' of c. Accordingly, a may-path of length n specifies the existence of a sequence of n may-steps from c to a configuration c'. In this case, c' is *may-reachable* from c. A *must hyper-path* of length n from c defines the set C of *all configurations* c' such that a may-path of length n exists from c to c'. In case there exists a may-path of length $n-1$ from c to a configuration c'', and c'' has no successor configuration (i.e., c'' is a *deadlock*), then c'' is also contained in the set C of configurations being reachable via a may-path of length n. In

other words, a must hyper-path of length n contains both the configurations that are reachable from c via a may-path of length n and all the deadlocks that are may-reachable from c. We refer to the set C as the set of configurations that are *must-reachable* from c.

Whenever a configuration c has due to the data abstraction more than one successor configuration, may-reachability considers always one successor—that is, it considers only *one* data refinement. In contrast, a must hyper-path contains all successor configurations of c. Hence, it considers *all* possible data refinements.

Definition 6 (Reachability). *Let $N = \langle P, T, F, \mathrm{rd}, \mathrm{wt}, \mathrm{del}, \mathrm{grd}\rangle$ be a WFD-net, c, c' be configurations of N, and $C, C' \subseteq \Xi$ be sets of configurations of N.*

- *A set C of configurations is* reachable *from a configuration c, denoted by $c \to C$, if and only if there is a transition $t \in T$ being enabled at c and the firing of t yields C (i.e., $c \xrightarrow{t} C$).*
- *There is a* may-step *from a configuration c to a configuration c', denoted by $c \to_{may} c'$, if and only if c' is an element of a set C of configurations that is reachable from c.*
- *A* may-path *(of length n) from a configuration c is a sequence of configurations $c_1, \ldots, c_n$ of N, $n \geq 0$, where $c_1 = c$ and $c_i \to_{may} c_{i+1}$ for every $i = 1, \ldots, n-1$; we denote the existence of a may-path $c_1, \ldots, c_n$ with $c_1 = c$ and $c_n = c'$ by $c \to^*_{may} c'$.*
- *A* must hyper-path *(of length n) from a configuration c is a set of may-paths from c inductively defined as follows: $\Omega_1 = \{c\}$ and $\Omega_{i+1} = \{\omega, c' \mid \omega \in \Omega_i \wedge \exists C : c \to C \wedge c' \in C\} \cup \{\omega \mid \omega \in \Omega_i \wedge c \not\to\}$ for $i = 1, \ldots, n-1$.*
- *By $c \twoheadrightarrow_{must} C$ we denote the existence of a must hyper-path Ω_n such that, for every $c' \in C$, there is a may-path $c_1, \ldots, c_n \in \Omega_n$ with $c_1 = c$ and $c_n = c'$.*

Example 5. The state space of the shipper is depicted in Figure 3. From the start configuration `c0` only the singleton set $\{$`c1`$\}$ can be reached by firing transition `receive goods`. Configuration `c1` consists of a marking [`p1`], and the abstraction function σ assigns the value $\top$ to data elements `cl`, `gds`, and `ads`. Predicate `isHigh(price)` is undefined in `c1`. Transition `calculate price` is enabled at `c1`. Firing this transition yields the set $\{$`c2`, `c3`$\}$ of successor configurations. The difference between both configurations is that in `c2` predicate `isHigh(price)` is evaluated to true (denoted `isHigh`), whereas in `c3` this predicate is evaluated to false (denoted $\neg$`isHigh`). From `c0` $\to$ `c1` $\to \{$`c2`, `c3`$\}$, we conclude that there is a may-step from `c0` to `c1`, a mat-step from `c1` to `c2` as well as from `c1` to `c3`. So there is a may-path of length 2 from `c0` to `c2` and from `c0` to `c3`. Consequently, $\{$`c2`, `c3`$\}$ but also $\{$`c1`$\}$ is must-reachable from `c0`, because a must hyper-path of length 2 and 1 exists, respectively. Observe that there is also a must hyper-path (of length 5) from `c0` to $\{$`c11`, `c12`$\}$. Only configuration `c13` is may-reachable from `c11`. Unlike `c11`, configuration `c12` does not have any successor. Hence, we conclude from the definition of a must hyper-path that there exists a must hyper-path of length 6 from `c0` to $\{$`c13`, `c12`$\}$.

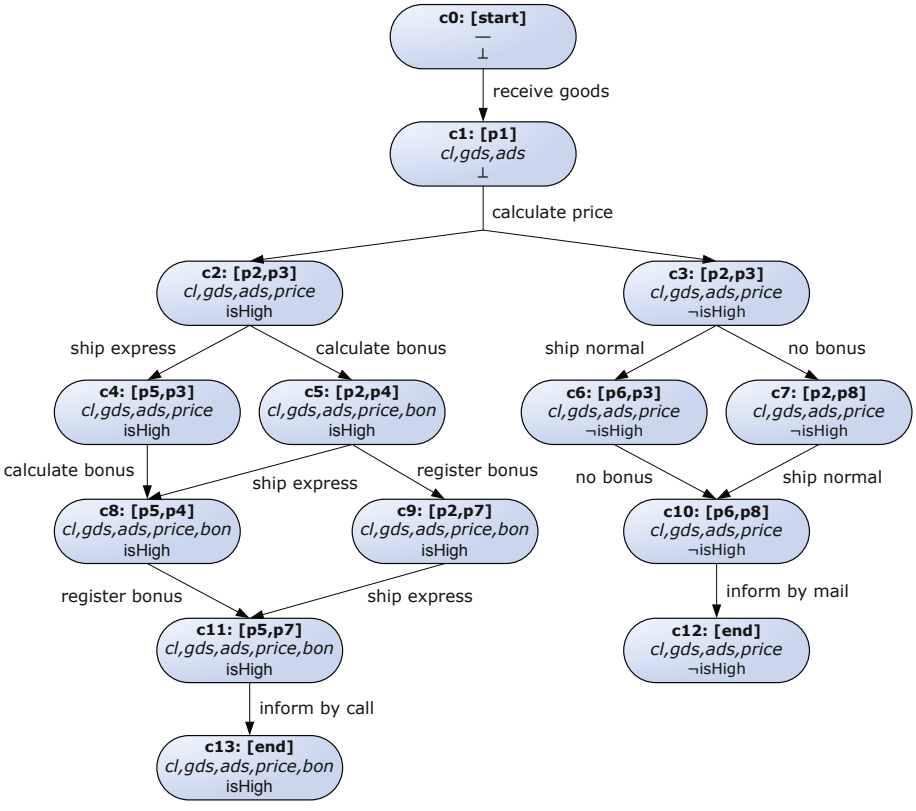

Fig. 3. State space of the shipper in Fig 1. Each rounded rectangle specifies a configuration of the shipper. In the first line, the identifier of the configuration and the marking is depicted, the second line presents all defined data elements, and the third line evaluates the predicate `isHigh(price)`.

5 Soundness

With the help of may- and must-reachability we can formalize soundness for WFD-nets. The soundness property, originally defined for WF-nets, ensures that from any reachable marking the final marking can be reached, and every task can potentially be executed. In this section, we extend the definition of soundness to WFD-nets. We present two notions: may-soundness and must-soundness. A WFD-net is may-sound if and only if *there exists* a data refinement such that the concrete workflow model (that contains all data information) is sound. In contrast to may-soundness, the notion of must-soundness guarantees that the WFD-net is sound in *all* possible data refinements.

Definition 7 (May- and Must-soundness). *Let* $N = \langle P, T, F, \mathrm{rd}, \mathrm{wt}, \mathrm{del}, \mathrm{grd}\rangle$ *be a WFD-net, let* c_0 *be the start configuration of* N*, and let* $C_f \subseteq \Xi$ *denote the set of final configurations of* N*.* N *is*

- may-sound *if and only if for every set C of configurations of N being must-reachable from the start configuration c_0 (i.e., $c_0 \twoheadrightarrow_{must} C$), there exists a configuration $c \in C$ such that a configuration $c_f \in C_f$ is may-reachable from c (i.e., $c \rightarrow^*_{may} c_f$).*
- must-sound *if and only if for every configuration c being may-reachable from the start configuration c_0 of N (i.e., $c_0 \rightarrow^*_{may} c$), there exists a set $C \subseteq C_f$ of final configurations that is must-reachable from c (i.e., $c \twoheadrightarrow_{must} C$).*

May-soundness ensures that for any set C of configurations that are must-reachable from the start configuration, there exists a configuration $c \in C$ from which a final configuration is may-reachable. The set C contains all configurations that are reachable from the initial configuration in any data refinement (because must-reachability considers all may-paths). The existence of a configuration $c \in C$ from which a final configuration is may-reachable guarantees that there exists at least one data refinement of N (i.e., one may-path) in which a final marking can be reached.

Must-soundness ensures that from any configuration c that is may-reachable from the start configuration, a subset of the final configurations is must-reachable. That means, from every marking that is reachable in the WFD-net N, a final configuration can be reached in any data refinement of N (because must-reachability considers all may-paths).

Example 6. Consider again the state space of the shipper in Figure 3. As previously mentioned, there exists a must hyper-path from the start configuration `c0` to the set $\{$`c13`, `c12`$\}$ of configurations. Both configurations, `c13` and `c12`, are final configurations (and clearly `c13` is may-reachable from `c13` and `c12` is may-reachable from `c12`). As this is the longest must hyper-path in the state space, we conclude that from any other must hyper-path, there always exists a state from which either `c13` or `c12` is may-reachable. Thus, the shipper is may-sound. It can also be easily seen that from each state being may-reachable from `c0`, there exists a must hyper-path to a final configuration. Hence, we conclude that the WFD-net of the shipper is also must-sound. In other words, the soundness property holds in any data refinement of the WFD-net in Figure 1.

Let us now come back to the modification of the shipper (cf. Section 2) where different predicates (`isHighLeft(price)` and `isHighRight(price)`) are used in the left and the right part of the shipper. In this case, `c2` corresponds to a configuration where both predicates are true, and `c3` corresponds to a configuration where both predicates are false. In addition, `c1` has two more successors, say `c2`$'$ and `c3`$'$, corresponding to configurations where `isHighLeft(price)` is true and `isHighRight(price)` is false and vice versa. In configuration `c2`$'$ transitions `ship express` and `no bonus` are enabled. Firing these transitions yields a configuration, say c, where the shipper is in the marking [`p5`, `p8`]. Configuration c is a deadlock, and it is may-reachable from the start configuration. Hence, there does not exist a must hyper-path from c to a final configuration, and thus the modified shipper is not must-sound. However, the modified shipper is may-sound: there exists a data refinement in which a final configuration can be reached, namely, if

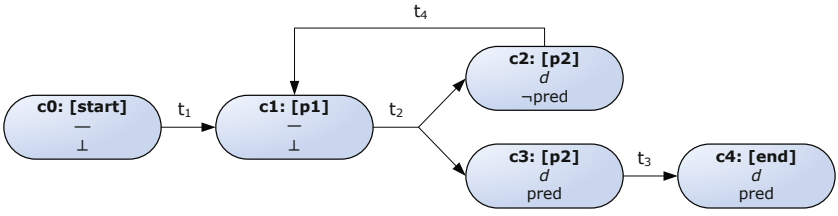

Fig. 4. State space of the example from Figure 2

the two predicates `isHighLeft(price)` and `isHighRight(price)` always evaluate to the same value.

Figure 4 shows the state space of the WFD-net from Figure 2. As there is no guarantee that both branches of t_2 will lead to proper completion (there exists an infinite sequence $c_1 \rightarrow c_2 \rightarrow c_1 \rightarrow c_2 \ldots$), we conclude that this WFD-net is not must sound. However, we see that one branch of t_2, namely the one going to c_3, always leads to completion. Therefore, the workflow is may-sound.

6 Conclusion

In this paper we showed how to obtain reliable verification results when verifying conceptual workflow nets with data information. Our work is in fact a cross-fertilization of design and modeling frameworks coming from the field of Process-Aware Information Systems (PAIS), and verification and abstraction approaches developed in the field of Formal Methods.

The final target of researchers working in the area of formal methods is usually the verification of programs/systems which may contain complex data coming from large or infinite data domains, consist of a large number of distributed components, etc. To cope with the complexity of the objects to be verified, many abstraction techniques [4,5,6,7,10,15] such as predicate abstractions, and abstraction methodologies, such as CEGAR (counter-example guided abstraction refinement) [3,13], are proposed. The concept of may/must transition systems was defined in this area [14] and then found multiple applications there.

In our work, the verification target is not a refined system but a conceptual model, which may later on be refined in different ways. We do not need to apply abstractions to cope with the complexity of data, as it is done in [16,11,12] for WS-BPEL processes—the data is still underdefined, abstract, in the models we consider.

We use WFD-nets to specify conceptual workflows. As we showed in [19], WFD-nets can be seen as an abstraction from notations deployed by popular modeling tools, like Protos of Pallas Athena, which 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.) By building on the classical formalism of Petri nets, we keep our framework easily adaptable to many industrial and academic languages.

Future work. For the future work we plan to integrate our implementation of the may/must-based soundness verification of WFD-nets in the process analysis/discovery framework ProM [2]. As a basis, we use the generic CTL model-checking algorithm presented in [17], and restrict it to the soundness property. This algorithm shall then replace the standard algorithm [19] for checking soundness. As ProM provides import functionality for many industrial process modeling languages, by integrating our implementation in ProM we will achieve direct applicability of our framework to real-world conceptual workflows.

Another item for the future work concerns the diagnostic methods for the identification of possible causes of incorrectness in the workflow, which would be fit to work within the may/must framework.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., Alves de Medeiros, A.K., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W., Weijters, A.J.M.M.: ProM 4.0: Comprehensive Support for Real Process Analysis. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 484–494. Springer, Heidelberg (2007)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided Abstraction Refinement for Symbolic Model Checking. Journ. of the ACM 50(5), 752–794 (2003)
4. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994); A preliminary version appeared in the Proc. of the POPL 1992
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages (POPL 1977), pp. 238–252. ACM Press, New York (1977)
6. Dams, D.: Abstract Interpretation and Partition Refinement for Model Checking. PhD dissertation, Eindhoven University of Technology (July 1996)
7. Dams, D., Gerth, R., Grumberg, O.: Abstract Interpretation of Reactive Systems. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(2), 253–291 (1997)
8. Dehnert, J., Rittgen, P.: Relaxed soundness of business processes. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 157–170. Springer, Heidelberg (2001)
9. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley & Sons, Chichester (2005)
10. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

11. Heinze, T.S., Amme, W., Moser, S.: Generic CSSA-based pattern over boolean data for an improved WS-BPEL to petri net mappping. In: Mellouk, A., Bi, J., Ortiz, G., Chiu, D.K.W., Popescu, M. (eds.) Third International Conference on Internet and Web Applications and Services, ICIW 2008, Athens, Greece, June 8-13, pp. 590–595. IEEE Computer Society, Los Alamitos (2008)
12. Heinze, T.S., Amme, W., Moser, S.: A restructuring method for WS-BPEL business processes based on extended workflow graphs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 211–228. Springer, Heidelberg (2009)
13. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental Verification by Abstraction. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 98–112. Springer, Heidelberg (2001)
14. Larsen, K.G.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
15. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design 6(1), 11–44 (1995)
16. Moser, S., Martens, A., Görlach, K., Amme, W., Godlinski, A.: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: 2007 IEEE International Conference on Services Computing (SCC 2007), Salt Lake City, Utah, USA, July 9-13, pp. 98–105. IEEE Computer Society, Los Alamitos (2007)
17. Shoham, S., Grumberg, O.: Monotonic abstraction-refinement for ctl. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 546–560. Springer, Heidelberg (2004)
18. Trčka, N.: Workflow Soundness and Data Abstraction: Some negative results and some open issues. In: Workshop on Abstractions for Petri Nets and Other Models of Concurrency (APNOC), pp. 19–25 (2009)
19. Trčka, N., van der Aalst, W.M.P., Sidorova, N.: Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 425–439. Springer, Heidelberg (2009)