Diagnosis from first principles for workflow executions

G. Friedrich, V. Ivanchenko
Diagnosis from first principles for workflow executions

Gerhard Friedrich and Volodymyr Ivanchenko

Universitaet Klagenfurt, Universitaetsstrasse 65, 9020 Klagenfurt, Austria,
firstname.lastname@uni-klu.ac.at

Abstract. Workflow executions may fail because executed activities do not be-
have as intended. In this paper we present the concept of diagnosis for faulty
workflow executions based on first principle diagnosis. This approach requires
the definition of the system behavior by logical sentences. Since Colored-Petri
nets (CPN) are among the most popular methods for expressing the semantics of
workflows we show how CPN-based workflow descriptions can be transformed in
a set of logical sentences. These sentences provide a semantics which is equiva-
 lent to the original semantics based on CPN. We show that the logical description
of a workflow together with a description of an observed execution behavior al-
 low a correct and complete characterization of diagnoses. Consequently, methods
from first principle diagnosis can be applied to correctly and completely compute
diagnoses for workflow executions.

Keywords: diagnosis, workflows, model-based, orchestrated Web Services

1 Introduction

In the execution of orchestrated Web-services (which correspond to workflows (WF)
where the activities are implemented by Web-services) some services may behave in
a non-intended (faulty) way. The goal of the European project WS-Diamond (wsdia-
mond.di.unito.it) is to provide a framework for designing and implementing self-healing
Web Services by diagnosis, monitoring, and repair methods. For optimizing the repair
of faulty WF executions, the basic reasoning task is to analyze which activity execu-
tions were faulty. In diagnosis/repair systems this is called the diagnosis step. In this
paper we define the concept of diagnosis for workflow executions based on first prin-
ciple diagnosis [1] (also called model-based diagnosis) which offers a complete and
correct method for generating diagnoses. However, this method requires a logical de-
scription of the system to be analyzed. Such a logical description must be equivalent to
the semantics of WFs in order to assure correctness and completeness of the diagnosis
 generation process. We base the description of the WF semantics on Colored Petri-
nets (CPN) due to its broad usage. In particular, we follow the workflow control-flow
patterns approach as described in [2]. In order to make our exposition as concise as
possible and to avoid decidability problems we restrict our formalism to the follow-
 ing control patterns described in [2]: WCP-1 trough WCP-8 (which includes sequence,
parallel split, synchronization, exclusive choice, simple merge, multi-choice, structured
synchronizing merge, multi-merge), WCP-16 (deferred choice), and WCP-33 (general-
ized AND-join). Although we exclude cycles, the above mentioned patterns allow the
specification of many real world problems. Furthermore, these restrictions lead to Col-
ored Petri-Nets which are save, i.e. each place in the Petri-net can only contain at most
one token.
Faulty executions generate faulty states of artifacts (called objects) which may serve as an input to other activity executions thus propagating failures. In order to deal with such dependencies we extend the control CPN formalism by objects. This is in the spirit of [3] who however do not provide a formalization based on CPN. Our CPN formalization is presented in Sec. 3 after introducing a motivating example in Sec. 2. In Sec. 4 we provide a logical model of WFs which is complete and correct w.r.t. the CPN semantics. Sec. 5 introduces the concept of diagnosis for WF executions. We show how the observed execution behavior of a WF can be expressed by logical sentences. Furthermore, we present a complete and correct characterization of diagnoses for WF executions based on the description of the execution behavior and the description of the semantics of a WF. As a consequence this paper describes the foundation for computing diagnoses using complete and correct methods of diagnosis from first principles.

2 Example

For supporting the introduction of our concepts we use the example workflow depicted in Fig. 1. The graphical WF representation applies the control patterns defined in [2] extended by objects and functions manipulating these objects. We call this class of CPNs Workflow/Data-Petri-nets (WFD).

A WFD consists of places and transitions. In Fig. 1 ellipses depict places and rectangles are transitions. If the end point or start point of an arrow is marked with a name of a place then this means that the arrow connects to the referred place. E.g. the upper arrows of transition $T1$ connect to place $X$. Double arrows are just an abbreviation for two single arrows. Places are either control places (e.g. $p0 \ldots p7$) or object places (e.g. $X, O1, O2, GO$). $p0$ is an input place since no arc points to $p0$. Transition $E$ is an output transition since the control flow stops there; $E$ has no outgoing arc to a control place. Places can be empty or hold one token. Note, this is a simplification of the general definition of CPN and allows a simpler formalization and presentation. Control places can hold a distinct token $id$ which has the intuitive semantics that the execution of a WF case has arrived at that place. Tokens of object places represent objects which are read or written (resp. affected) by transitions. In CPN every place must be typed. OBJ
is the object type, BOL represents the Boolean type which we consider as a subtype of OBJ, and CID is the type of control places which just contains id. In Fig. 1 ellipses are labeled with their types.

Transitions represent activities of a WF which can manipulate objects or which can decide about the control flow (e.g. XOR). A transition has a set of input arcs and output arcs which are labeled by so called arc expressions. The places of the input arcs are the input places of a transition (e.g. p1, X are input places of T1). Likewise, p2, X, O1 are the output places of T1. In our simplified version of CPN the input arcs of a transition are just labeled with an identifier bound to the token of the corresponding input places. A transition is enabled if all input places hold a token, e.g. if p1 and X hold a token then T1 is enabled. If a transition is enabled then it can fire. On firing, a transition consumes the tokens of the input places and assigns the empty set to these places. E.g. if T1 fires then p1 and X are set to nil. Symbol nil represents the empty set. On firing a transition the tokens of the output places are computed exploiting the bindings of the arc expressions of the input arcs. E.g. if T1 fires then the arc expression c (of arc (p1, T1)) is bound to id and the arc expression X (of arc (X, T1)) is bound to the token of place X. Note, we could have used any names for the identifier but we use c for bindings to id and the name of object places for bindings to objects. The output tokens of a fired transition are computed by applying the arc expressions of the outgoing arcs and the bindings of the identifiers. E.g. firing T1 assigns the token bound to arc expression X to object place X, so T1 just reads the token of place X and does not change it. However, firing T1 also generates a token by applying function f2 to X and assigns this token to object place O1. In other words, T1 reads object X and writes object O1 by applying the object-function f2. Beside such object-functions there are condition-functions which assign tokens to control places. E.g. transition XOR1 assigns the function cond1 on input object O1. If the result is true then token id (bound to c) is assigned to p3 else token id is assigned to p6. Such condition-functions may be just the expression c. Firing T1 assigns the token id to place p2.

The semantics of the depicted WF is that transition S produces object X which serves as an input to the WF. T1 produces an object O1 based on X. O1 is exploited in XOR1 for deciding on the control flow. In the upper branch of XOR1, T2 computes object O2 based on input X. Depending on O2, XOR2 decides on the control flow. In the upper branch of XOR2 the transition T3 assigns a new token to X, i.e. the object state written by S is replaced by the state produced by T3. Next, a so called guard transition G is executed. In the sense of [4] guards monitor the execution of a WF in order to detect a faulty behavior of the WF. For diagnosing we make some assumptions which transitions may (or may not) fail. Inputs to a WF are usually assumed to be correct, hence S is assumed to work correctly. Similarly, we assume the WF is designed correctly and therefore XOR1 and XOR2 work correctly. However, we assume that transitions T1, T2, T3 may fail. Guard transitions assign tokens to guard objects which are of type Boolean. E.g. transition G assigns either true or false to GO. If a guard object has value false then the guard transition has discovered a non-indent behavior, e.g. this corresponds to a triggered exception.

Let us assume that a WF execution corresponds to the execution sequence ⟨S, T1, XOR1, T2, XOR2, G⟩ and the execution of G discovers a failure. Then for per-
forming some repair actions in order to complete this WF instance correctly, the main question is which transitions did not perform as intended. Our first answer might be that behaviors of $T_1$ or $T_2$ are probably faulty since the flow of execution involved both of them. Note, that $G$ triggers an exception on the object state of $X$ produced by $S$. This implies that the object state of $X$ must be changed. Consequently, any correct behavior of $T_1$ must generate an object $O_1$ where the upper branch of $XOR_1$ is taken. If the lower branch is followed on the value of $X$ generated by $S$ then the state of $X$ is not changed and guard $G$ triggers an exception. Consequently, $T_1$ did not misbehave in the execution sequence. In fact the misbehavior must be caused by the execution of $T_2$. Moreover, we know that the correct behavior of $T_2$ must lead to an execution of $T_3$. Note, for our reasoning we have exploited the observed execution behavior of functions, e.g. which branch was taken during execution (this corresponds to the result of executing condition-functions), which objects are read and written by object-functions, and on which object-states guard-functions evaluate to true or false. We will call this the history of execution.

Note, that the presented considerations did not exploit the definition of functions associated to transitions. In practice these definitions are not available for a number of reasons. E.g. in case of Web-services the functions of WFs are realized by providers which do not disclose their implementations.

The question we will answer in the rest of the paper is how we can exploit this information about the history of execution together with the description of the WF in order to identify transitions whose misbehavior is the source of the WF misbehavior.

### 3 Semantics of workflow models

For defining the semantics of WFs we follow [2] and extend these concepts by objects and guards. As stated above, we call such CPNs Workflow/Data-Petri-nets which allow the modeling of all acyclic and save workflow patterns presented in [2].

**Definition 1.** A Workflow/Data-Petri-net (WFD) is a Colored Petri-net with the following properties:

1. The set of types consist of $OBJ$ (which is the set of all possible object states), $BOL$ (which is a subtype of $OBJ$), and $CID$ the set of possible case identifiers. Because of the safety property $CID$ contains only one element $id$ identifying an instance of the workflow. Types represent colors in CPN.
2. $P$ is a finite set of places. We use the symbol $p$ as a name for the place $p \in P$ abusing slightly notation. Whether $p$ is a place or the name of a place, should be clear by the context.
3. $TR$ is a finite set of transitions.
4. $A$ is a finite set of arcs. Each arc is represented as ordered pair $(x, y)$ where $(x, y) \in P \times TR$ or $(x, y) \in TR \times P$.
5. The sets $P$, $TR$, and $A$ are disjoint.
6. Every place has either type $CID$, $OBJ$, or $BOL$. Places typed by $CID$ are the control places. The type of a place $p$ is referred by $type(p)$. $contPs(P)$ defines the set of control places. Places typed by $OBJ$ or subtypes of $OBJ$ are the object...
For each object $o$ read by transition $tr$ there is an outgoing arc $(o, tr)$ to $tr$ which is labeled by the arc expression $o$.

10. For each object $o$ read by transition $tr$ but not written there is an outgoing arc $(tr, o)$ to $o \in \text{objPs}(P)$ which is labeled by the arc expression $o$.

11. For each object $o_{\text{out}}$ written by transition $tr$, there is an outgoing arc $(tr, o_{\text{out}})$ to $o_{\text{out}} \in \text{objPs}(P)$ which is labeled by the arc expression $f(o_1, \ldots, o_n)$ where each $o_j \in \{o_1, \ldots, o_n\}$ is an object read by $tr$ and $f$ is the name of an object-function which evaluates to an object state.

12. There is a subset $I$ of the set of places called the input places. Input places have no incoming arc.

13. There is a subset $O$ of the transitions called the output transitions. Output transitions have no outgoing arc to a control place.

14. WF/Data-Petri-nets may contain guard transitions $G$. Guard transitions are transitions as defined. However, their object-functions write guard objects typed by $BOL$. These object-functions are called guard-functions. Places of guard objects are denoted by $\text{guardPs}(P)$. □

For the definition of the semantics of a Workflow/Data-Petri-net we have to define the semantics of the functions included in the arc expressions. For this definition we have to consider the fact that the result of evaluating an object-function could be indeterministic. Evaluating an object-function at different time points on the same input values could result in different output values.

**Definition 2.** A $n$-place object-function $f(o_1, \ldots, o_n)$ is defined by a $n+1$-place relation $R_f \subseteq \text{OBJ}_1 \times \ldots \times \text{OBJ}_{n+1}$ where $\text{OBJ}_i$ is the object type. In the tuple $(v_1, \ldots, v_n, v_{\text{out}}) \in R_f$ the values $v_1, \ldots, v_n$ are the input values and $v_{\text{out}}$ is an output value. For given input values $v_1, \ldots, v_n$ multiple tuples (i.e. multiple output values) could exist in $R_f$. We call $R_f$ the defining relation of function $f$.

The evaluation $\text{eval}(f(v_1, \ldots, v_n))$ for input values $v_1, \ldots, v_n$ returns as a result $v_{\text{out}}$ where $(v_1, \ldots, v_n, v_{\text{out}})$ is in $R_f$. If the function $f$ is a condition-function then $R_f \subseteq \text{OBJ}_1 \times \ldots \times \text{OBJ}_n \times \text{BOL}$ where $\text{BOL} = \{\text{true}, \text{false}\}$ and $\text{true} \neq \text{false}$. We assume that condition-functions $f$ are deterministic, i.e. if $(v_1, \ldots, v_n, v_{\text{out}_1}) \in R_f$ and $(v_1, \ldots, v_n, v_{\text{out}_2}) \in R_f$ then it must hold that $v_{\text{out}_1} = v_{\text{out}_2}$.

Places $p$ are marked with at most one token $v$ where $v$ is a member of the type associated to $p$. The mark of $p$ may be empty. Intuitively for an object place $p$ the token

---

1 Note, we could easily restrict the inputs of this expression to a subset of the objects read by $tr$. However, in order to make our presentation simpler we do not consider this case.
\( v \) refers to an object state. A marking is a set \( \{(p,v)|p \in P\} \) where \( v \) is a token or nil.

A marking represents the state of a CPN. Note, for every place one tuple is contained in the marking. Function \( \text{token}(p, m) \) denotes the token of \( p \) in marking \( m \).

The marking of a WF/Data-Petri-net can be changed by the firing of transitions. Changing a marking \( m_t \) to \( m'_t \) is called a step. In principle such transitions can fire in parallel which means that a step contains multiple transitions. However, as usually assumed in transaction management we require serializability. Any outcome of a step containing multiple transitions corresponds to a sequential firing of theses transitions. Consequently, we define that every step contains at most one transition.

All object places are initialized by the value nil except places for guard objects. These places are initialized by the token true. All input places are initialized by the token id. All other control places are initially empty.

**Definition 3.** Given a transition \( tr \), the set of input places is denoted by \( inP(tr) = \{p|(p, tr) \in A\} \) and the set of output places is defined by \( outP(tr) = \{p|(tr, p) \in A\} \).

A transition \( tr \) is enabled at a marking \( m_t \) iff for all \( p \) of \( inP(tr) \) the token of \( p \) is not empty, i.e. \( \text{token}(p, m_t) \neq \text{nil} \).

If \( tr \) is enabled at \( m_t \) and \( tr \) is fired at \( m_t \) then this results in a marking \( m'_t \). In this case the marking \( m'_t \) is directly reachable from \( m_t \) usually denoted by \( m_t[tr]m'_t \) (in the Petri-net field). The result of firing a transition \( tr \) at a marking \( m_t \) is changing the marks of the input and output places of \( tr \) by the following rules:

- If \( p_{in} \) is in \( inP(tr) \) but not in \( outP(tr) \) then \( \text{token}(p_{in}, m'_t) = \text{nil} \).
- If the object-function \( f(o_1, \ldots, o_n) \) is the arc expression of arc \( (tr, p_{out}) \) then \( \text{token}(p_{out}, m'_t) = v \) where \( v \) is the result of \( \text{eval}(f(\text{token}(o_1, m_t), \ldots, \text{token}(o_n, m_t))) \).
- If \( o \) is the arc expression of arc \( (tr, p) \) and \( p \) is an object place then \( \text{token}(p, m'_t) = \text{token}(p, m_t) \).
- If the condition-function “If \( \text{cond}(o_1, \ldots, o_n) = \text{true} \) then \( c \) else nil” is the arc expression of arc \( (tr, p_{out}) \) and \( \text{eval}(\text{cond}(\text{token}(o_1, m_t), \ldots, \text{token}(o_n, m_t))) = \text{true} \) then \( \text{token}(p_{out}, m'_t) = \text{id} \) else \( \text{token}(p_{out}, m'_t) = \text{nil} \). If the condition-function comprises \( \text{cond}(o_1, \ldots, o_n) = \text{false} \) then in the previous rule \( \text{true} \) is exchanged by \( \text{false} \).
- If \( c \) is the arc expression of arc \( (tr, p) \) then \( \text{token}(p, m'_t) = \text{id} \) (\( p \) a control place).
- If place \( p \) is not in \( inP(tr) \) and not in \( outP(tr) \) then \( \text{token}(p, m'_t) = \text{token}(p, m_t) \).

A finite occurrence sequence, starting at marking \( m_k \) and ending at \( m_n \), is a sequence of markings and transitions \( m_k[tr_k]m_{k+1} \ldots m_{n-1}[tr_{n-1}]m_n \). A marking \( m' \) is reachable from \( m'' \) iff there exists a finite occurrence sequence starting at \( m' \) and ending at \( m'' \). An occurrence sequence correspond to an execution of the WF.

For WF/Data-Petri-nets we exploit the usual properties [5]:

- An instance of a WF/Data-Petri-net is a marking reachable from the initial marking.
- A WF/Data-Petri-net is terminating iff from all its instances a marking can be reached where all control places are empty. The marking where all control places are empty is called a final instance.
- A WF/Data-Petri-net is safe iff for every place \( p \) and for every reachable marking the number of tokens at \( p \) is less or equal to 1.
A WF/Data-Petri-net is well-behaved iff it is safe and terminating.
A WF/Data-Petri-net is acyclic iff the subgraph of the WF/Data-Petri-net where only control arcs are considered is acyclic.

We consider only well-behaved, acyclic WF/Data-Petri-nets.

We assume that exceptions should only be thrown if the WF execution does not proceed as intended. In this case a guard function discovers a failure and the state of a guard object is set to false. Consequently, the WF specification should allow only final instances where the tokens of all guard objects are true. In this case we say a WF/Data-Petri-net is valid. More formally:

**Definition 4.** A final instance $m_t$ of a WF/Data-Petri-net is valid if the guard objects are true, i.e. for all $p \in \text{guardPs}(P)$ token$(p, m_t) = true$.

## 4 Logical formalization

Subsequently, we construct a set of logical sentences $ls(WF)$ given a WF/Data-Petri-net $WF$. The basic idea of expressing the semantics of a WF/Data-Petri-net by logical sentences is that we require a decision machinery which outputs the set of possible markings of a WF, even if our knowledge about the definition of the functions is partial. We design $ls(WF)$ in such a way that marking $m$ is a final instance iff $m$ logically follows from $ls(WF)$. For expressing $ls(WF)$ we exploit First-order-logic. However, we will apply this logical description only for cases where all types of places are finite. Consequently, the set of markings and the defining relations $R_f$ are finite. In $ls(WF)$ all quantified variables are bounded by sets which are finite. Therefore, $ls(WF)$ can be transformed to propositional logic.

The fact that place $p$ in marking $m$ has token $v$ is represented by $val(p, m, v)$. We use $val$ instead of token to avoid overloading of symbols. The set of all markings is denoted by $M$. Note, that in the logical representation of a workflow a marking is represented by a symbol whereas in Petri-nets a marking is a set of tuples.

Logical sentences formulated subsequently are added to $ls(WF)$. The constrain that in a marking a place can only have one token is expressed by following sentence: \[ \forall p \in P, m \in M, v_1, v_2 \in \text{type}(p) : val(p, m, v_1) \land val(p, m, v_2) \land v_1 \neq v_2 \rightarrow \bot. \]

In a marking places have a token or the place is empty: \[ \forall p \in P, m \in M : (\exists v \in \text{type}(p) : val(p, m, v)) \lor val(p, m, nil). \]

The initial marking $m_0$ is generated by the following sentences which are added to $ls(WF)$: \[ \{ val(p, m_0, id) | p \in I \} \cup \{ val(p, m_0, nil) | p \in \text{contPs}(P) - I \} \cup \{ val(p, m_0, nil) | p \in \text{objPs}(P) - \text{guardPs}(P) \} \cup \{ val(p, m_0, true) | p \in \text{guardPs}(P) \}. \]

In order to reference the token of place $p_i$ in logical sentences we associate to $p_i$ a distinct logical variable $V_i = \text{var}(p_i)$. Whenever we use place $p_i$ and the logical variable $V_i$ in a logical sentence then $V_i$ is the variable associated to $p_i$. We will use the following abbreviations to define the construction of logical sentences.

For a free logical variable $m$ (which will be bound to markings by a quantifier in logical sentences) let $\text{VAL}(m)$ denote the conjunction $\forall V_0 \in \text{type}(p_0) : val(p_0, m, V_0) \land \ldots \land \forall V_n \in \text{type}(p_n) : val(p_n, m, V_n)$ where $p_0, \ldots, p_n$ are all places of the Petri-net $WF$. E.g. for our example $\text{VAL}(m)$ is $\forall V_0 \in \text{type}(p_0) : val(p_0, m, V_0) \land \ldots \land \forall V_7 \in \{...\}$. 


type(\(p_7\)) : val(\(p_7, m, V_7\))∧\(\forall V_8 \in type(X) : val(X, m, V_8)\)∧\(\ldots\∧\forall V_{11} \in type(GO) : val(GO, m, V_{11})\).

For a transition \(tr \in TR\) let \(\text{NOTEMPTY}(tr)\) denote the conjunction \(V_i \neq \text{nil} \∧ \ldots \∧ V_j \neq \text{nil}\) where \(V_i = \text{var}(p_i), \ldots, V_j = \text{var}(p_j)\) are the variables associated to the input places of \(tr\), i.e. \(p_1, \ldots, p_j \in \text{in}(tr)\) and all input places of \(tr\) are mentioned in \(p_1, \ldots, p_j\). If transaction \(tr\) has no input place then \(\text{NOTEMPTY}(tr)\) is true.

The following logical sentences describe the properties of enabled transitions. For each \(tr \in TR\) include the following sentence in \(is(WF)\): \(\forall m \in M : \text{VAL}(m) \rightarrow (\text{NOTEMPTY}(tr) \leftrightarrow enabled(tr, m)).\)

For transition \(T1\) this sentence is: \(\forall m \in M : \forall V_0 \in type(p_0) : val(p_0, m, V_0) \ ∧ \ \forall V_1 \in type(p_1) : val(p_1, m, V_1) \ ∧ \ldots \ ∧ \forall V_4 \in type(X) : val(X, m, V_4) \ ∧ \ldots \ ∧ \forall V_{11} \in type(GO) : val(GO, m, V_{11}) \rightarrow \ (p_1 \neq \text{nil} \∧ X \neq \text{nil} \rightarrow enabled(T1, m)).\)

Note, we can optimize these sentences by reducing obviously unnecessary \(\text{VAL}(m)\) literals depending on the transition \(tr\). We do not describe this in order to simplify matters.

If a transition \(tr\) is enabled at marking \(m_t\) it can fire and on firing \(tr\) creates an immediate successor marking \(m_{t'}.\) Consequently for every enabled transition there is a marking which can be reached by firing \(tr\).

For all transitions \(tr \in TR\) we construct a logical sentence with the structure \(\forall m_t \in M : enabled(tr, m_t) \land \text{VAL}(m_t) \land LHS \rightarrow \exists m_{t'} \in M : RHS\) where we subsequently specify the details of \(LHS\) (Left-Hand-Side) and \(RHS\) (Right-Hand-Side) depending on \(tr\). All addition to \(LHS\) and \(RHS\) are connected by operator \(\land\). Note, we follow exactly the definition of firing a transition.

- If \(p_i\) is in \(\text{in}(tr)\) but not in \(\text{out}(tr)\) then \(val(p_i, m_{t'}, \text{nil}) \in RHS\).
- If \(p_i\) is not in \(\text{in}(tr)\) and not in \(\text{out}(tr)\) then add \(\text{val}(p_i, m_{t'}, V_i)\) to \(RHS\).
- If the object-function \(f(o_1, \ldots, o_n)\) is the arc expression of arc \((tr, out)\) and \(R_f\) is its defining relation then add \(\forall V \in OBJ : R_f(V_1, \ldots, V_n, V)\) to \(LHS\) where \(V\) is a unique variable, and add \(\text{val}(out, m_{t'}, V)\) to \(RHS\).
- If the condition-function \(cond(o_1, \ldots, o_n) = \text{true}\) or \(cond(o_1, \ldots, o_n) = \text{false}\) is part of the arc expression of arc \((tr, out)\) and \(R_{cond}\) is its defining relation then add \(\forall V \in BOL : R_{cond}(V_1, \ldots, V_n, V)\) to \(LHS\) where \(V\) is a unique variable.
  - If the condition-function contains \(cond(o_1, \ldots, o_n) = \text{true}\) then add \((\text{val}(out, m_{t'}, id) \land V = \text{true}) \lor (\text{val}(out, m_{t'}, \text{nil}) \land V = \text{false})\) to \(RHS\).
  - If the condition-function contains \(cond(o_1, \ldots, o_n) = \text{false}\) then add \((\text{val}(out, m_{t'}, id) \land V = \text{false}) \lor (\text{val}(out, m_{t'}, \text{nil}) \land V = \text{true})\) to \(RHS\).
- If \(c\) is the arc expression of arc \((tr, out)\) and \(o\) is an object place then add \(\text{val}(out, m_{t'}, V_i)\) to \(RHS\).
- If \(c\) is the arc expression of arc \((tr, out)\) then add \(\text{val}(out, m_{t'}, id)\) to \(RHS\).

In our example we generate for transition \(T1\) the following sentence:

\(\forall m_t \in M : enabled(T1, m_t) \land \forall V_0 \in type(p_0) : val(p_0, m_t, V_0) \land \ldots \land \forall V_7 \in type(p_7) : val(p_7, m_t, V_7) \land \forall V_8 \in type(X) : val(X, m_t, V_8) \land \forall V_9 \in type(O1) : val(O1, m_t, V_9) \land \forall V_{10} \in type(O2) : val(O2, m_t, V_{10}) \land \forall V_{11} \in type(GO) : val(GO, m_t, V_{11}) \land \forall V \in OBJ : f2(V_8, V) \rightarrow \exists m_{t'} \in M : val(p_0, m_{t'}, V_0) \land val(p_1, m_{t'}, \text{nil}) \land \ldots \land val(p_7, m_{t'}, V_7) \land val(X, m_{t'}, V_8) \land val(O1, m_{t'}, V) \land val(O2, m_{t'}, V_{10}) \land val(GO, m_{t'}, V_{11}).\)
For transition XOR1 the following sentence is generated:
\[ \forall m_t \in M : \text{enabled}(\text{XOR1}, m_t) \land \forall V_0 \in \text{type}(p_0) : \text{val}(p_0, m_t, V_0) \land \ldots \land \forall V_g \in \text{type}(O1) : \text{val}(O1, m_t, V_g) \land \ldots \land \forall V \in \text{BOL} : \text{cond1}(V_0, V) \rightarrow \exists m_{t'} \in M : \text{val}(p_0, m_{t'}, V_0) \land \text{val}(p_1, m_{t'}, V_1) \land \text{val}(p_2, m_{t'}, \text{nil}) \land (\text{val}(p_3, m_{t'}, \text{id}) \land V = \text{true}) \lor (\text{val}(p_5, m_{t'}, \text{nil}) \land V = \text{false}) \land \ldots \land (\text{val}(p_6, m_{t'}, \text{id}) \land V = \text{false}) \lor (\text{val}(p_7, m_{t'}, \text{nil}) \land V = \text{true}) \land \ldots \land \text{val}(\text{GO}, m_{t'}, V_{11}). \]

In addition we have to characterize final markings. Let EMPTY be the conjunction
\[ V_i = \text{nil} \land \ldots \land V_j = \text{nil} \] where the variables \( V_i = \text{var}(p_i), \ldots, V_j = \text{var}(p_j) \) are the associate variables of all the control places \( p_i, \ldots, p_j \in \text{contPs}(P) \). Add the following sentence to \( \text{ls}(WF) \): \( \forall m \in M : \text{VAL}(m) \rightarrow (\text{EMPTY} \leftrightarrow \text{final}(m)). \)

Furthermore, we have to ensure that final instances are valid. Let GUARDSTRUE be the conjunction
\[ V_i = \text{true} \land \ldots \land V_j = \text{true} \] where the variables \( V_i = \text{var}(p_i), \ldots, V_j = \text{var}(p_j) \) are the associate variables of all the guard object \( p_i, \ldots, p_j \in \text{guardPs}(P) \).

By adding the sentence \( \forall m \in M : \text{VAL}(m) \land \text{final}(m) \rightarrow \text{GUARDSTRUE} \) to \( \text{ls}(WF) \) we enforce validity of final instances. Note, if a marking is a final instance and some guard object is marked with false then a contradiction follows.

Furthermore, we have to express properties of defining relations. For every transition \( tr \) which is enabled in some marking \( m \) the defining relations of the object-functions and condition-functions applied in the arc expressions of \( tr \) must be defined on the input objects. Let \( f(o_1, \ldots, o_n) \) be an object-function of \( tr \) and \( R_f \) its defining relation then add to \( \text{ls}(WF) \) the sentence \( \forall m \in M : \text{VAL}(m) \land \text{enabled}(tr, m) \rightarrow \exists V \in \text{OBJ} : R_f(V_1, \ldots, V_n, v). \)

In case the object-function is a guard-function or a condition-function the type \( \text{OBJ} \) is replaced by the Boolean type \( \text{BOL} \).

The definition of the functions is given in the set of logical sentences \( \overline{R} \). In case the types are finite these relations can be expressed as set of facts. \( \overline{R} \) is not included in \( \text{ls}(WF) \) in order to account for the case where function definitions are not known.

**Property 1.** Given a WF/Data-Petri-net \( WF \) and its logical description \( \text{ls}(WF) \) and the function definitions \( \overline{R} \) as described above. All final instances of \( WF \) are valid iff \( \text{ls}(WF) \cup \overline{R} \) is consistent.

**Property 2.** Let \( WF \) be valid. If \( m_{wf} = \{(p, v) | p \in P\} \) is a final instance of \( WF \) then there exists a marking \( m_t \) of the logical representation of \( WF \) s.t. \( \text{ls}(WF) \cup \overline{R} \models \text{val}(p, m_t, v) \) for all \( (p, v) \in m_{wf} \) and \( \text{ls}(WF) \cup \overline{R} \models \text{final}(m_t). \)

Conversely, if there exists a marking \( m_t \) of the logical representation of \( WF \) and \( \text{ls}(WF) \cup \overline{R} \models \text{val}(p, m_t, v) \) for a \( p \in P \) and a \( v \in \text{OBJ} \) and \( \text{ls}(WF) \cup \overline{R} \models \text{final}(m_t) \) then there is a final instance \( m_{wf} \) of \( WF \) where \( (p, v) \in m_{wf}. \)

Consequently, the logical representation of a WF/Data-Petri-net \( WF \) covers the semantics of \( WF \) as defined on the basis of CPN. In the next section we show how this logical representation can be exploited to diagnose faulty workflow executions.

## 5 Diagnosing the execution of a workflow

Diagnosis from first principles is based on a comparison of intended and non-intended behavior with the observed behavior. The observed behavior is contained in the history of a workflow execution, i.e. the trace of the execution, the results of evaluating...
condition-functions, and the results of executing object-functions. The non-intended behavior is partially characterized by the results of executing guard-functions. In case we have no exact information about these values, we can represent them just as Skolem constants. The observed execution behavior of a function $f(v_1,\ldots,v_n)$ is expressed as literal of its defining relations $R_f(v_1,\ldots,v_n,v_{\text{out}})$ where $v_{\text{out}}$ is the output value generated by the execution of $f$. The set of observed execution behaviors of a workflow execution $EX$ is denoted by $BEH(EX)$. By execution behavior we refer to the observed execution behavior for short. In our example the occurrence sequence of the execution is $m_0[S]h_1 [T1]h_2 [XOR1]h_3 [T2]h_4 [XOR2]h_5 [G]h_6$ where $m_0,h_i$ are markings. Using the Skolem constants $s_k$ the execution behavior for our example is: $R_{f_1}(s_1), R_{f_2}(s_1,s_2), R_{\text{cond}1}(s_2,\text{true}), R_{f_3}(s_1,s_3), R_{\text{cond}2}(s_3,\text{false})$, and $R_{f_5}(s_1,\text{false})$.

If functions behave as intended then the execution behaviors must be contained in their defining relation. Given the execution behavior and provided that the workflow is valid, then it must not be the case that in a final instance the value of a guard-object is false. Consequently, if a non-valid final instance exists then some of the execution behaviors are not part of the defining relations. However, the defining relations express the intended (correct) behavior. Such execution behaviors which are not indented are called faulty. Which execution behaviors are considered as faulty serves as an important information to optimize further recovery actions. Note, some execution behaviors are assumed to be always correct. I.e. some transitions provide user inputs which may be assumed as correct. In addition, if we assume a correct design of the workflow, then the condition-functions and the guard-functions are also assumed to be correct. In this case only object-functions are regarded as possibly faulty. The set of execution behaviors which are defined to be correct is denoted by $OKBEH(EX)$. For the subset of execution behaviors $\Delta \subseteq BEH(EX)$ where $\Delta$ is considered as faulty and all other execution behaviors $BEH(EX) - \Delta$ are considered to be correct the following property must hold. We call such a set $\Delta$ a diagnosis.

**Definition 5.** Given a WF/Data-Petri-net $WF$, an execution $EX$, the set of defined correct execution behaviors $OKBEH(EX)$, the overall set of execution behaviors $BEH(EX)$ and a subset $\Delta$ of $(BEH(EX) - OKBEH(EX))$.

$\Delta$ is a diagnosis iff for all functions $f$ there exists a defining relation $R_f$ which includes all the execution behaviors assumed to be correct (i.e. for all $f$ of $WF$ if $R_f(v_1,\ldots,v_n,v_{\text{out}}) \in (FIXBEH(EX) \cup (BEH(EX) - \Delta))$ then $(v_1,\ldots,v_n,v_{\text{out}}) \in R_f$) and $WF$ is valid. A diagnosis $\Delta$ is minimal if no proper subset of $\Delta$ is a diagnosis.

Consequently, a diagnosis points out which execution behaviors can be assumed as correct and which of them as faulty s.t. a valid WF-definition exists. Conversely, if $\Delta$ is not a diagnosis, then we know that there is no valid WF-definition where all the observed execution behaviors are included in the definition of the functions.

Following the concept of diagnosis from first principles, minimal diagnoses characterize a lower bound of the set of all possible diagnoses, i.e. a subset of a minimal diagnosis is not a diagnosis. Among the minimal diagnoses there are the minimum cardinality diagnoses. Minimal (rsp. minimum cardinality) diagnoses are computed by methods presented in [1] exploiting theorem provers for consistency checking. As previously stated, this could be done in our case by propositional theorem provers. However, the
translation of the presented logical sentences to propositional logic is beyond the scope of the paper. In order to apply diagnosis by first principles we have to relate the concept of diagnosis to consistency checking. If an execution behavior \( R_f(v_1, \ldots, v_n, v_{out}) \) of function \( f \) is assumed to be in the defining relation \( R_f \) then this is expressed by the literal \( ok(R_f(v_1, \ldots, v_n, v_{out})) \). Consequently, by asserting \( ok(R_f(v_1, \ldots, v_n, v_{out})) \) we make a correctness assumption. By such an assumption the corresponding execution behavior should be included in \( R_f \). We express this by the set of logical sentences \( ls(EX) \) where for each execution behavior \( R_f(v_1, \ldots, v_n, v_{out}) \) of a workflow execution \( EX \) we add the sentence \( ok(R_f(v_1, \ldots, v_n, v_{out})) \leftrightarrow R_f(v_1, \ldots, v_n, v_{out}) \).

Given our logical model of the workflow (which covers the semantics of the WF by exploiting Property 1) and the execution behavior we can express the following relation.

Property 3. Given WF/Data-Petri-net \( WF \), an execution \( EX \), the set of defined correct execution behaviors \( OKBEH(EX) \), the overall set of execution behaviors \( BEH(EX) \) and a subset \( \Delta \) of \( (BEH(EX) - OKBEH(EX)) \).

\( \Delta \) is a diagnosis iff \( ls(WF) \cup ls(EX) \cup \{ok(B)|B \in (BEH(EX) - \Delta)\} \cup \{-ok(B)|B \in \Delta\} \) is consistent.

As a result we have a logical characterization of diagnoses for WF/Data-Petri-nets which is correct and complete w.r.t. the semantics of these WF/Data-Petri-nets.

In our example \( R_{f_3}(s_1, s_3) \) is a diagnosis. By Property 3, we have to show the consistency of: logical sentences \( ls(WF) \cup ls(EX) \) joined with the assumption that execution behavior of \( f_3 \) is not ok joined with the assumption that all other execution behaviors are ok. Since we exclude \( R_{f_3}(s_1, s_3) \) any firing of transition \( T2 \) will produce a new value \( s' \) for which the behavior of \( cond2(s') \) is unknown. Since we do not know the value of the evaluation of \( cond2(s') \) there are two cases (logical interpretations): either the control flow branches to \( p5 \) or to \( p6 \). In case \( p5 \) gets the id token, transition \( T3 \) is fired changing the value of \( X \). Since the value of \( X \) is changed the guard function does not imply that \( GO = false \). Consequently, we have found a logical model and the set of sentences is consistent. However, \( R_{f_2}(s_1, s_2) \) is not a diagnosis. If \( R_{f_2}(s_1, s_2) \) is assumed to be faulty then this results in an unknown behavior of \( cond1 \). Although we do not know if \( cond1 \) evaluates to \( true \) or \( false \), there is a firing of \( XOR1 \) implying the consequence that either \( p3 \) or \( p6 \) gets the token \( id \). In case \( p6 \) gets the id token, \( X \) is not changed and the guard-function will assign \( false \) to the guard object. In case \( p3 \) gets the id token we know from the execution history that firing \( T2 \) on value \( s_1 \) of \( X \) results in \( s_3 \) for \( O2 \) and then firing \( XOR2 \) on \( s_3 \) will result in an evaluation of \( cond2(s_3) = false \). Consequently, the value of \( X \) is not changed and the subsequent evaluation of the guard-function results in \( false \). Both execution branches imply a final, non-valid instance. A contradiction follows. Provided that conditional-functions and guard-functions are correct and function \( f1 \) serves as an input function for the WF which is also correct then the only minimal diagnosis for our example is \( R_{f_3}(s_1, s_3) \).

6 Related work

In [6] a diagnosis approach is formulated based on the composition of automata. Diagnoses are computed by tracing data dependencies. However, tracing data dependencies is known to miss some possible faulty behaviors. In the work of [7] dynamic
slicing techniques are introduced based on model-based diagnosis and exploited for Web-services in [8]. However, the set of diagnoses produced by this method may be too large. E.g. by exploiting the semantics of WFs as presented spurious diagnoses could be eliminated. Relevant slices are introduced in [9] where the authors point out the shortcomings of dynamic slices. Since we can prove completeness and correctness of our approach we are able to avoid the shortcomings of dynamic slices. However, to the best of our knowledge completeness and correctness results of relevant slicing are open. Furthermore, dynamic and relevant slicing is always defined on a trajectory of the program execution. We designed our methods s.t. executions and re-executions of functions can be in any order, which is important to handle cases where failures may occur during the repair of faulty WF executions (e.g. in case activities are re-executed).

7 Conclusions

In this paper we extended the CPN-semantics for expressing control-patterns of WFs by functions reading and writing objects and by guard-functions signalling a faulty behavior of a WF. We showed how a logical description based on a CPN-description of a WF can be generated s.t. this logical description is equivalent to the CPN-description. Based on the execution behavior of WFs we defined the concept of diagnosis. We showed how to describe the execution behavior by logical sentences. These logical descriptions together with the logical description of the WF semantics serve as a complete and correct characterization of diagnoses for WF executions. This characterization is based on consistency checking and allows the correct and complete generation of diagnoses by using methods developed in the field of first principle diagnosis.

References