Applied Informatics University Klagenfurt

ALPEN-ADRIA UNIVERSITÄT KLAGENFURT

Technical Report

2008/002

Institute of Applied Informatics

Intelligent Systems and Business Informatics

June 2008

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 behave 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 equivalent 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 allow 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 (*wsdiamond.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 executions 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 principle diagnosis [1] (also called model-based diagnosis) which offers a complete and correct method for generating diagnoses. However, this method requires a logical description 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 decideability problems we restrict our formalism to the following 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 (generalized AND-join). Although we exclude cycles, the above mentioned patterns allow the specification of many real world problems. Furthermore, these restrictions lead to Colored 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 object. We call this class of CPNs Workflow/Data-Petri-nets (WFD).

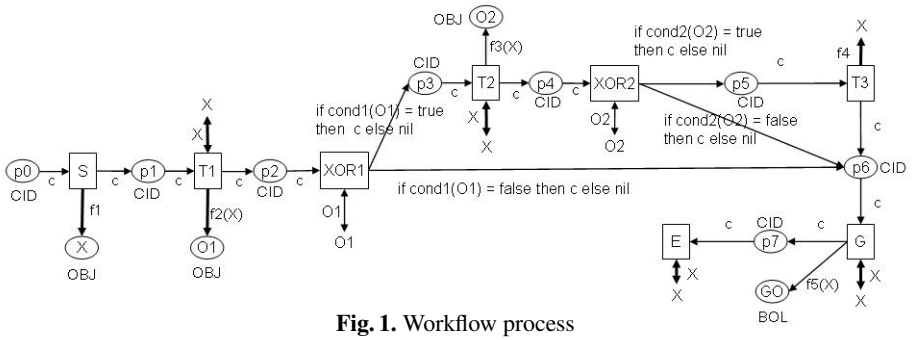

Fig. 1. Workflow process

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. $XOR1$). 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 $XOR1$ reads object $O1$, evaluates 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 corrct, 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-indented behavior, e.g. this corresponds to a triggered exception.

Let us assume that a WF execution corresponds to the execution sequence $\langle S,\ T1,\ XOR1,\ T2,\ XOR2,\ G\rangle$ 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 $T1$ or $T2$ 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 $T1$ must generate an object $O1$ where the upper branch of $XOR1$ 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, $T1$ did not misbehave in the execution sequence. In fact the misbehavior must be caused by the execution of $T2$. Moreover, we know that the correct behavior of $T2$ must lead to an execution of $T3$. 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

places. $objPs(P)$ defines the set of object places. There is exactly one object place $o \in P$ for each object o read or written by an activity of the workflow.

7. Each ingoing arc (p, tr) from a control place p to a transition tr (called a control arc) is labeled by the arc expression c.
8. Each outgoing arc (tr, p) from a transition tr to a control place p (also called a control arc) is either labeled by c or by an expression "*If* $cond(o_1, \ldots, o_n) = true$ *then* c *else nil*" or by an expression "*If* $cond(o_1, \ldots, o_n) = false$ *then* c *else nil*" where $o_1, \ldots, o_n \in objPs(P)$ are the objects read by transition tr and $cond$ is the name of a condition-function which evaluates to $true$ or $false$.[1]
9. For each object o read by transition tr there is an ingoing 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 $o \in objPs(P)$ which is labeled by the arc expression o.
11. For each object o_{out} written by transition tr, there is an outgoing arc (tr, o_{out}) to $o_{out} \in objPs(P)$ 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 $guardPs(P)$. □

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 definded by a n+1-place relation* $R_f \subseteq OBJ_1 \times \ldots \times OBJ_{n+1}$ *where* OBJ_i *is the object type. In the tuple* $\langle v_1, \ldots, v_n, v_{out} \rangle \in R_f$ *the values* $v_1, \ldots, v_n$ *are the input values and* v_{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 $eval(f(v_1, \ldots, v_n))$ *for input values* $v_1, \ldots, v_n$ *returns as a result* v_{out} *where* $\langle v_1, \ldots, v_n, v_{out} \rangle$ *is in* R_f*. If the function* f *is a condition-function then* $R_f \subseteq OBJ_1 \times \ldots \times OBJ_n \times BOL$ *where* $BOL = \{true, false\}$ *and* $true \neq false$*. We assume that condition-functions* f *are deterministic, i.e. if* $\langle v_1, \ldots, v_n, v_{out_1} \rangle \in R_f$ *and* $\langle v_1, \ldots, v_n, v_{out_2} \rangle \in R_f$ *then it must hold that* $v_{out_1} = v_{out_2}$*.*

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 $token(p, m)$ 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 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. $token(p, m_t) \neq nil$.

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

A finite occurrence sequence, starting at marking m_k and ending at m_n, is a sequence of markings and transitions $m_k[tr_k\rangle m_{k+1} \ldots m_{n-1}[tr_{n-1}\rangle m_n$. 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 $p \in guardPs(P)$ $token(p, m_t) = true$.*

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: $\forall p \in P, m \in M, v1, v2 \in type(p) : val(p, m, v1) \wedge val(p, m, v2) \wedge v1 \neq v2 \rightarrow \bot$.

In a marking places have a token or the place is empty: $\forall p \in P, m \in M : (\exists v \in type(p) : val(p, m, v)) \vee val(p, m, nil)$.

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

In order to reference the token of place p_i in logical sentences we associate to p_i a distinct logical variable $V_i = var(p_i)$. 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 $\mathrm{VAL}(m)$ denote the conjunction $\forall V_0 \in type(p_0) : val(p_0, m, V_0) \wedge \ldots \wedge \forall V_n \in type(p_n) : val(p_n, m, V_n)$ where $p_0, \ldots, p_n$ are *all* places of the Petri-net WF. E.g. for our example $\mathrm{VAL(m)}$ is $\forall V_0 \in type(p_0) : val(p_0, m, V_0) \wedge \ldots \wedge \forall V_7 \in$

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

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

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

For transition $T1$ this sentence is: $\forall m \in M : \forall V_0 \in type(p_0) : val(p_0, m, V_0) \wedge \forall V_1 \in type(p_1) : val(p_1, m, V_1) \wedge \ldots \wedge \forall V_8 \in type(X) : val(X, m, V_8) \wedge \ldots \wedge \forall V_{11} \in type(GO) : val(GO, m, V_{11}) \rightarrow (p_1 \neq nil \wedge X \neq nil \leftrightarrow enabled(T1, m))$. Note, we can optimize these sentences by reducing obviously unnecessary $\mathrm{VAL}(m)$ 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 $\forall m_t \in M : enabled(tr, m_t) \wedge \mathrm{VAL(m_t)} \wedge LHS \rightarrow \exists m_{t'} \in M : RHS$ 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 $\wedge$. Note, we follow exactly the definition of firing a transition.

- If p_i is in $inP(tr)$ but not in $outP(tr)$ then $val(p_i, m_{t'}, nil) \in RHS$.
- If p_i is not in $inP(tr)$ and not in $outP(tr)$ then add $(val(p_i, m_{t'}, V_i)$ to RHS.
- If the object-function $f(o_1, \ldots, o_n)$ is the arc expression of arc (tr, p_{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 $val(p_{out}, m_{t'}, V)$ to RHS.
- If the condition-function $cond(o_1, \ldots, o_n) = true$ or $cond(o_1, \ldots, o_n) = false$ is part of the arc expression of arc (tr, p_{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 $cond(o_1, \ldots, o_n) = true$ then add $((val(p_{out}, m_{t'}, id) \wedge V = true) \vee (val(p_{out}, m_{t'}, nil) \wedge V = false))$ to RHS.
  If the condition-function contains $cond(o_1, \ldots, o_n) = false$ then add $((val(p_{out}, m_{t'}, id) \wedge V = false) \vee (val(p_{out}, m_{t'}, nil) \wedge V = true))$ to RHS.
- If o is the arc expression of arc (tr, p_{out}) and o is an object place then add $val(p_{out}, m_{t'}, V_i)$ to RHS.
- If c is the arc expression of arc (tr, p_{out}) then add $val(p_{out}, m_{t'}, id)$ to RHS.

In our example we generate for transition $T1$ the following sentence:

$\forall m_t \in M : enabled(T1, m_t) \wedge \forall V_0 \in type(p_0) : val(p_0, m_t, V_0) \wedge \ldots \wedge \forall V_7 \in type(p_7) : val(p_7, m_t, V_7) \wedge \forall V_8 \in type(X) : val(X, m_t, V_8) \wedge \forall V_9 \in type(O1) : val(O1, m_t, V_9) \wedge \forall V_{10} \in type(O2) : val(O2, m_t, V_{10}) \wedge \forall V_{11} \in type(GO) : val(GO, m_t, V_{11}) \wedge \forall \mathbf{V} \in OBJ : f2(V_8, \mathbf{V}) \rightarrow \exists m_{t'} \in M : val(p_0, m_{t'}, V_0) \wedge val(p_1, m_{t'}, \mathbf{nil}) \wedge val(p_2, m_{t'}, V_2) \wedge \ldots \wedge val(p_7, m_{t'}, V_7) \wedge val(X, m_{t'}, V_8) \wedge val(O1, m_{t'}, \mathbf{V}) \wedge val(O2, m_{t'}, V_{10}) \wedge val(GO, m_{t'}, V_{11})$.

For transition $XOR1$ the following sentence is generated:

$\forall m_t \in M : enabled(XOR1, m_t) \wedge \forall V_0 \in type(p_0) : val(p_0, m_t, V_0) \wedge \ldots \wedge \forall V_9 \in type(O1) : val(O1, m_t, V_9) \wedge \ldots \wedge \forall \mathbf{V} \in BOL : cond1(V_9, \mathbf{V}) \rightarrow \exists m_{t'} \in M : val(p_0, m_{t'}, V_0) \wedge val(p_1, m_{t'}, V_1) \wedge val(p_2, m_{t'}, \mathbf{nil}) \wedge ((val(p_3, m_{t'}, id) \wedge V = true) \vee (val(p_3, m_{t'}, nil) \wedge V = false)) \wedge \ldots \wedge ((val(p_6, m_{t'}, id) \wedge V = false) \vee (val(p_6, m_{t'}, nil) \wedge V = true)) \wedge \ldots \wedge val(GO, m_{t'}, V_{11})$.

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

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

By adding the sentence $\forall m \in M : \mathrm{VAL(m)} \wedge final(m) \rightarrow \mathrm{GUARDSTRUE}$ to $ls(WF)$ 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 $ls(WF)$ the sentence $\forall m \in M : \mathrm{VAL(m)} \wedge enabled(tr, m) \rightarrow \exists v \in OBJ : R_f(V_1, \ldots, V_n, v)$. In case the object-function is a guard-function or a condition-function the type OBJ is replaced by the Boolean type 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 $ls(WF)$ 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 $ls(WF)$ and the function definitions $\overline{R}$ as described above. All final instances of WF are valid iff $ls(WF) \cup \overline{R}$ 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_l of the logical representation of WF s.t. $ls(WF) \cup \overline{R} \models val(p, m_l, v)$ for all $(p, v) \in m_{wf}$ and $ls(WF) \cup \overline{R} \models final(m_l)$.

Conversely, if there exists a marking m_l of the logical representation of WF and $ls(WF) \cup \overline{R} \models val(p, m_l, v)$ for a $p \in P$ and a $v \in OBJ$ and $ls(WF) \cup \overline{R} \models final(m_l)$ 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 $R_f(v_1, \ldots, v_n, v_{out})$ where v_{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 $m_0[S\rangle h_1\ [T1\rangle h_2\ [XOR1\rangle h_3\ [T2\rangle h_4\ [XOR2\rangle h_5\ [G\rangle h_6$ where m_0, h_i are markings. Using the Skolem constants s_k the execution behavior for our example is: $R_{f1}(s_1)$, $R_{f2}(s_1, s_2)$, $R_{cond1}(s_2, true)$, $R_{f3}(s_1, s_3)$, $R_{cond2}(s_3, false)$, and $R_{f5}(s_1, false)$.

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 Δ 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 Δ 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 Δ of $(BEH(EX) - OKBEH(EX))$.*

Δ 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. forall f of WF if $R_f(v_1, \ldots, v_n, v_{out}) \in (FIXBEH(EX) \cup (BEH(EX) - \Delta))$ then $\langle v_1, \ldots, v_n, v_{out}\rangle \in R_f$) and WF is valid. A diagnosis Δ is minimal if no proper subset of Δ 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 Δ 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 Δ of $(BEH(EX) - OKBEH(EX))$.

Δ is a diagnosis iff $ls(WF) \cup ls(EX) \cup \{ok(B) | B \in (BEH(EX) - \Delta)\} \cup \{\neg ok(B) | B \in \Delta\}$ 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 $R_{f3}(s_1, s_3)$ 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 $f3$ is not ok joined with the assumption that *all other* execution behaviors are ok. Since we exclude $R_{f3}(s_1, s_3)$ 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, $R_{f2}(s_1, s_2)$ is not a diagnosis. If $R_{f2}(s_1, s_2)$ 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 $R_{f3}(s_1, s_3)$.

6 Related work

In [6] a diagnosis approach is formulated based on the composition of automatons. 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 short comings 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

1. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **23**(1) (1987) 57–95
2. Russell, N., ter Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow control-flow patterns: A revised view. In: BPM-06-22. BPM Center Report, BPMcenter.org (2006)
3. Reichert, M., Dadam, P.: $\text{Adept}_{\text{flex}}$-supporting dynamic changes of workflows without losing control. J. Intell. Inf. Syst. **10**(2) (1998) 93–129
4. Eder, J., Lehmann, M.: Workflow data guards. In: OTM Conferences (1). Volume 3760 of Lecture Notes in Computer Science., Springer (2005) 502–519
5. Kiepuszewski, B., Hofstede, A., van der Aalst, W.: Fundamentals of control flow in workflows. Acta Informatica **39**(3) (2003) 143–209
6. Yan, Y., Dague, P.: Modeling and diagnosing orchestrated web service processes. In: International Conference on Web Services. (2007) 51–59
7. Wotawa, F.: On the relationship between model-based debugging and program slicing. Artif. Intell. **135**(1-2) (2002) 125–143
8. Ardissono, L., Console, L., Goy, A., Petrone, G., Picardi, C., Segnan, M., Dupré, D.T.: Enhancing web services with diagnostic capabilities. In: European Conference on Web Services, IEEE Computer Society (2005) 182–191
9. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.: Incremental regression testing. In: ICSM '93: Proceedings of the Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (1993) 348–357