A Static Analysis Technique to Detect Unsatisfiable Conditions in Ontology-based Workflows

Gabriele Weiler[1,2], Arnd Poetzsch-Heffter[2] and Stephan Kiefer[1]

[1] Fraunhofer Institute for Biomedical Engineering, St. Ingbert, Germany
{gabriele.weiler, stephan.kiefer}@ibmt.fraunhofer.de
[2] Software Technology Group, University of Kaiserslautern, Germany
poetzsch@informatik.uni-kl.de

Abstract: Static analysis techniques for consistency checking of workflows allow to avoid runtime errors. This is in particular crucial for long running workflows where errors, detected late, can cause high costs. In many classes of workflows, the data perspective is rather simple, and the control flow perspective is the focus of consistency checking. In our setting, however, workflows are used to collect and integrate complex data based on a given domain ontology. In such scenarios, the data perspective becomes central and data consistency checking crucial.

In this paper, we focus on detecting unsatisfiable conditions, a data inconsistency which can lead to non-reachable tasks in workflows. We describe an algorithm to detect such inconsistencies in workflows with an ontology-based data perspective. The algorithm utilizes semantic web reasoning. We discuss soundness and completeness of the technique.

1 Introduction

Workflows describe the processing of data according to a well-defined control flow. Workflows often interact with humans and integrate them into the processing. We consider workflows in which the processed data has semantic metadata in terms of a given domain ontology, like e.g. for medical trials. Such semantic metadata is not only crucial to allow data integration, but can also help to improve data quality. In our work we show that it can be utilized to detect data inconsistencies in the workflow definition at design time of the workflow. Such *static* checking prevents executing faulty workflows where errors might occur only months after the workflow was started, causing potentially huge costs. We divide data inconsistencies into two categories:

1. *Data-dependent Control Flow Inconsistencies* causing e.g. undesired abortion of workflow executions or unreachable tasks due to unsatisfiable conditions.
2. *Semantic Data Inconsistencies* causing data collected during workflow execution to be inconsistent with the knowledge of the underlying domain that we assume to be given by a domain ontology[1].

[1]Similarly, one could check consistency with complex XML-schema information.

It is important to eliminate the first kind of inconsistencies to guarantee reliable workflow executions. Avoiding the second kind guarantees consistency of the collected data, a prerequisite to get reliable results from data analysis and to enable data integration.

The class of workflows we consider consists of human processable tasks, comprising forms that users have to fill in at execution time. In [WPHK09] we have presented a language called SWOD for this class of workflows, putting special emphasize on a formally defined data perspective based on an existing domain ontology. Furthermore, we have sketched a static analysis technique utilizing description logic and its reasoning services to check both categories of data inconsistencies, but have in particular focused on checking Semantic Data Inconsistencies. We have not described in detail the semantics of conditions in SWOD and how conditions are handled in the consistency checking algorithm. We will close this gap in this paper. The contribution of this paper is to augment the work described in [WPHK09] by describing in detail the semantics of SWOD focusing on semantics of conditions and a static analysis technique to detect unsatisfiable conditions in SWOD workflow descriptions.

SWOD shall not be seen as a substitute to existing powerful workflow languages, but as an example, how these languages can be augmented with ontology-based data perspectives and profit from techniques described in this work. The reader is refered to the Technical Report accompanying this paper [Wei09], where detailed formal descriptions can be found.

The paper is structured as follows: In Sec. 2, we describe our motivating application the ontology-based trial management system ObTiMA. In Sec. 3, we describe the workflow language SWOD, its semantics focusing on the semantics of conditions and data inconsistencies. In Sec. 4, a technique to detect these inconsistencies, in particular unsatisfiable conditions, is described. We conclude with discussion and related work.

2 ObTiMA - An Ontology-based Trial Management System

The motivating application for this work is ObTiMA, an ontology-based trial management system. ObTiMA has been developed for the European project ACGT (Advancing Clinico Genomic Trials on Cancer) [TB+08], which aims to provide a biomedical grid, for semantic integration of heterogeneous biomedical databases using a shared ontology for cancer trials, the ACGT Master Ontology (ACGT-MO) [BWC+08].

ObTiMA allows trial chairmen to set up patient data management systems with comprehensive metadata in terms of the ACGT-MO to facilitate integration with data collected in other trial management systems or biomedical data sources. Forms to collect patient data during the trial can be designed and questions on the forms can be created from the ontology in a user friendly way. Furthermore, treatment plans can be designed in ObTiMA, which are workflows to guide doctors through the treatment of a patient, and a form with ontology annotation can be assigned to each task of these treatment plans to document the patient's treatment. The algorithms described in this work will enable ObTiMA to detect data inconsistencies in the treatment plans.

3 Specifying Workflows

In this section we describe a simple workflow language with an ontology-based data perspective (SWOD). A workflow description in SWOD consists of a workflow template describing the control flow and the forms, and a workflow annotation containing the semantic description for the data.

3.1 Workflow Template

A workflow template describes tasks and their transitions. Tasks describe a piece of work, which has to be executed by a human user. Each task contains one form that has to be filled in to document the work. A form contains questions also called items. SWOD supports acyclic workflows with the basic control flow patterns sequence, XOR-/AND-split and XOR-/AND-join. Each outgoing transition of an XOR-split has an associated condition. For reasons of simplicity we describe in the following SWOD and the algorithm for workflows without concurrency, an extension for concurrent workflows can be found in [Wei09]. To illustrate our language and our algorithm, we use a simple example treatment plan. The outline is shown in Fig. 1.

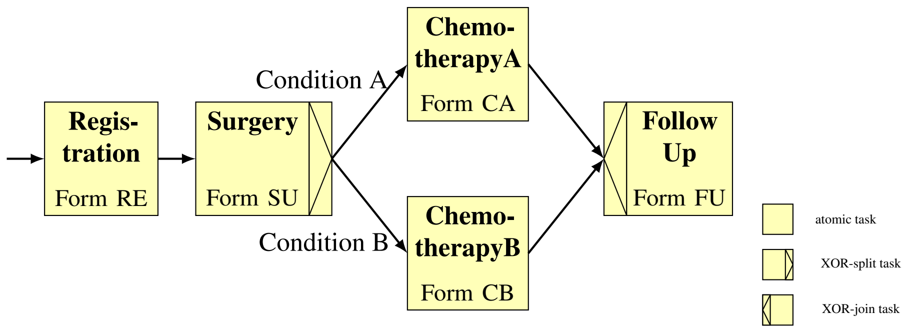

Figure 1: Example workflow: “example treatment plan”.

3.2 Workflow Annotation

A workflow annotation describes the semantic annotation of the data in terms of an existing domain ontology. It assigns a description from the ontology to each item (described in item annotations) and to each condition (in condition annotations). With this information the data, stored in form based data sources, can be queried in terms of the ontology.

Each workflow has a so-called *focal point*, which denotes the subject of a workflow execution (e.g. for a treatment plan it is a patient). Each annotation for an item or a condition refers to that focal point.

```
WFAnnotation ExampleTreatmentPlan
 focal PPatient;
 FormAnnotation FormRE
  ItemAnnotation(IDia OntoPath(d:HumanBeing(PPatient) hasTumor(PPatient,
   PTumor) d:Tumor(PTumor) hasDiameter(PTumor, PDia) float(PDia)) Value(Max(20.0))
 FormAnnotation FormSU
  ItemAnnotation(ITum OntoPath(d:HumanBeing(PPatient) hasTumor(PPatient,
    PTumor) d:Tumor(PTumor)) Specify(Case(breasttumor d:Breasttumor)
    Case(nephroblastoma d:Nephroblastoma)
    Case(other NOT(d:Breasttumor OR d:Nephroblastoma))))
  ItemAnnotation(IMet OntoPath(d:HumanBeing(PPatient)
   hasMetastasis(PPatient, PMet) d:Metastasis(PMet) Exist()))
 ConditionAnnotation(ConditionA d:Tumor(?tum) ∧ hasTumor(PPatient, ?tum)
  ∧ hasDiameter(?tum, ?dia) ∧ float(?dia) ∧ ≤(?dia, 4))
 ConditionAnnotation(ConditionB d:Tumor(?tum) ∧ hasTumor(PPatient, ?tum)
  ∧ hasDiameter(?tum, ?dia) ∧ float(?dia) ∧ >(?dia, 4))
```

Listing 1: Extract from workflow annotation for example treatment plan.

The workflow annotation for the example treatment plan is shown in Lst. 1. It describes the item annotations for items “Diameter of tumor” “Type of tumor” and “Does patient have metastasis?” and the condition annotations for Condition A and B. The underlying domain ontology is a simple example ontology, which is partly shown in the next paragraph. All classes of the domain ontology are prefixed with ’d:’.

Domain Ontology. We have chosen description logics (DL) as language for the domain ontology, since this family of knowledge representation formalisms provides the base of most modern ontology languages. A basic understanding of DL is required in the following, and we refer to [BCM+03] for a detailed description.

In DL, an *ontology* $\mathcal{O}$ introduces the terminology of an application domain. In the following an extract from the domain ontology for our example treatment plan is shown:
$\mathsf{HumanBeing} \sqsubseteq ((\leq 1\ \mathsf{hasMother.HumanBeing}) \sqcap (\forall\ \mathsf{hasAge.nonNegativeInteger}[< 150]))$
$\mathsf{Breasttumor} \sqsubseteq (\mathsf{Tumor} \sqcap (\forall\ \mathsf{locatedIn.Breast}))$
An ontology introduces the terminology as a set of axioms of the form $\mathsf{C} \sqsubseteq \mathsf{D}$ (i.e. D subsumes C) and $\mathsf{C} \equiv \mathsf{D}$ (i.e $\mathsf{C} \sqsubseteq \mathsf{D}$ and $\mathsf{D} \sqsubseteq \mathsf{C}$), where C and D are general concepts. DL languages can be distinguished by the constructors they provide for defining general concepts. We currently consider the DL language ALCQ(D) for the domain ontologies. This language provides amongst others the constructors number restriction (e.g. ≤ 1 $\mathsf{hasMother.HumanBeing}$), value restriction (e.g. $\forall$ $\mathsf{locatedIn.Brain}$) and data type restriction (e.g. $\forall$ $\mathsf{hasAge.nonNegativeInteger}[< 150]$).

In DL an *ABox* $\mathcal{A}$ describes assertions about individuals in terms of an ontology. An ABox contains concept assertions $\mathsf{C(a)}$ (i.e. individual a is an instance of concept C) and role assertions $\mathsf{R(a,b)}$ (i.e. individual b is a filler of the role R for a). Ind($\mathcal{A}$) is the set of individuals that occur in $\mathcal{A}$. *DL-reasoners* exist, which can e.g. check if a concept subsumes another, if an ontology is consistent, if an ABox is consistent wrt. an ontology, i.e. they are not contradictory or if an ABox $\mathcal{A}$ implies an assertion β (written $\mathcal{A} \models \beta$).

In most modern ontology languages, concepts are called classes and roles are called properties. We use this notation in the following sections.

Item Annotation. An item annotation is the ontology description for an item. It consists of an ontology path and an item constructor. The grammar of an item annotation is depicted in Lst. 2.

```
itemAnnotation ::= ItemAnnotation(itemID ontoPath itemConstructor);
ontoPath ::= OntoPath(focPointDesc ontoPathPart);
ontoPathPart ::= (relAssert varAssert)∗;
itemConstructor ::= existItem|specifyItem|valueItem;
valueItem ::= Value(Min(minv)? Max(maxv)?);
specifyItem ::= Specify(cases);
cases ::= case | cases case;
case ::= Case(acode ontdescr);
existItem ::= Exist();
focPointDesc ::= varAssert;
relAssert ::= rel(srcvar, tarvar);
varAssert ::= type(var);
```

Listing 2: Grammar of an item annotation.

Ontology Path. An ontology path is the basic ontology description of an item. In a simplified notation it can e.g. be “Patient hasTumor Tumor” or “Patient hasTumor Tumor hasWeight Weight”. Formally an ontology path consists of variable assertions and relation assertions. A *variable assertion* is defined as type(vname), where type can be a class or a primitive data type from the domain ontology and vname is the name representing the variable. Variables of the former kind are called *object variables*, variables of the later kind *data type variables*. E.g. in Lst. 1, line 5 the object variable PTumor of type d:Tumor and the data type variable PDia of type float are described. Relations between variables can be expressed with *relation assertions* (e.g. in Lst. 1, line 5 PTumor and PDia are related with hasDiameter). A relation between an object variable and a data type (resp. an object) variable has to be a data type (resp. an object) property from the domain ontology. For each workflow description one focal variable is declared, e.g. PPatient (Lst. 1, line 2). Each ontology path for an item starts with this variable.

Item Constructor. Different kinds of item descriptions can be assembled from an ontology path depending on how the value of the item is considered in the semantics (s. Sec. 3.3). Therefore, we defined different item constructors. The last variable in the ontology path is the associated variable of the item constructor.

1. *Value-Items* query values of data type properties from the ontology, e.g. “diameter of tumor” (Lst. 1, line 4-5) or “weight of patient”. Numerical value-items can have associated range restrictions, denoted with Min and Max. The associated variable has to be a data type variable. E.g. the associated variable for item “diameter of tumor” is PDia and the maximum value is 20 cm.

2. *Exist-Items* query if an individual for the associated variable exists, e.g. “Does patient

have metastasis?" (Lst. 1, line 11-12). The associated variable has to be an object variable.

3. *Specify-Items* are multiple choice items, which restrict an existent variable of a class by one of its subclasses, e.g. "Type of tumor?" with answer possibilities: "breasttumor", "nephroblastoma", "other" (Lst. 1, line 7-10). The associated variable has to be an object variable. The answer possibilities need to have associated class descriptions from the ontology which have to be subsumed by the ontology classes which are defined as type of the associated variable.

Condition Annotation. A condition annotation is the ontology description of a condition. It is formalized similar to bodies of SWRL-rules (Semantic Web Rule Language [HPSB^{+}04]), but in a simplified form. Conditions refer to the focal variable of the workflow also called focal variable of the condition. Furthermore, the conditions refer to condition variables, which are prefixed with "?" (e.g. ?v). Conditions consist of a conjunction of atoms of the form class(x), dataType(x), objProp(x,y), dataProp(x,y) or cmpOp(x,y), where class is a class description, dataType a data type, objProp is an object property, dataProp an data type property from the domain ontology, cmpOp is a comparison operator like $\leq$ and x and y are either the focal variable of the condition, condition variables or data values. Each condition variable has to be related by property atoms to the focal variable. E.g. the meaning of Condition B (Lst. 1, line 15-16) is "Patient has a tumor with weight greater than 4". The grammar for a condition annotation is as follows:
conditionAnnot ::= **ConditionAnnotation**(conID condition);
condition ::= atom $\wedge$ condition | atom;
atom ::= class(x) | dataType(x) | objProp(x, y) | dataProp(x, y)| cmpOp(x, y);

3.3 Semantics of Workflow Description

We describe the semantics of a workflow description by defining workflow executions for it. A workflow execution starts with the begin task of the workflow description. Then tasks are executed in the order they are related with transitions. For an XOR-split the task succeeding the condition which is satisfied for the current execution is executed next. This must be exactly one in a consistent workflow description.

During execution of a task a user has to fill in a value into each item of the associated form of the task. Since we do not consider concurrency, at each point of execution the executed part of the workflow can be described by a sequence of already filled items (described by their item annotation and the filled in value). We call such a sequence executed workflow data path (EWDP). From such an EWDP an ABox can be calculated (s. Sec 3.3.1), representing the state, i.e. the data which has been collected until this point in the workflow execution. This ABox is the base to query the collected data in terms of the ontology.

Whether a condition is satisfied for an execution can be determined with the help of the ABox calculated from the EWDP ending with the last item before the condition. A condition is satisfied for the execution if it has a valid binding to this ABox (s. Sec. 3.3.2).

3.3.1 Creation of ABoxes for workflow executions

To derive the ABox $\mathcal{A}_p$ for an EWDP starting with an empty ABox we defined a calculus, called “ABoxRules”. In the following, we describe calculi informally and simplified, but sufficient to understand the basic ideas of the described technique. Formal descriptions can be found in [Wei09]. In Table 1 an ABox derived from an example workflow execution with calculus ABoxRules is shown. The calculus extends $\mathcal{A}_p$ for each filled item as follows:

- *Ontology Path.* For each of the object variables in the ontology path an individual is created in $\mathcal{A}_p$ by adding its variable assertion as a concept assertion. Each of the object relation assertions is added as a role assertion into $\mathcal{A}_p$. The individual created for the focal variable of the workflow is called the focal individual (e.g. PPatient in Table 1).
- *Item constructor.* For a value-item the according individual in $\mathcal{A}_p$ is related with the appropriate data type relation to the value of the item. For a specify-item the according individual is restricted with the class description associated to the answer filled into the form. For an exist-item an individual for the associated variable is only created if value is “yes”.

Introducing Item	ABox $\mathcal{A}_p$
Diameter of tumor: 2.1 cm	{d:HumanBeing(PPatient), hasTumor(PPatient, PTumor), d:Tumor(PTumor), hasDiameter(PTumor, 2.1)}∪
Type of tumor: breasttumor	{d:Breasttumor(PTumor)}∪
Has metastasis: Yes	{hasMetastasis(PPatient, PMet), d:Metastasis(PMet)}

Table 1: ABox $\mathcal{A}_p$ created for execution of example treatment plan. On the left the values filled into the items during execution are shown, whereas on the right the assembled ABox $\mathcal{A}_p$ is shown. Assertions are ordered according to the items on the left.

3.3.2 Checking if condition is satisfied

A condition is satisfied for a workflow execution if a valid binding exists between the condition and the ABox created from the EWDP ending with the last item before the condition. A valid binding between a condition and an ABox exists if each of the variables from the condition can be bound to an individual or a constant of the ABox (represented as tuple in a set called binding $\mathcal{B}$), where the focal variable of the condition is bound to the focal individual, and for each atom in the condition the constraint described in Table 2 holds on the ABox and the binding. If the condition does not have a valid binding to the ABox, it is either not satisfied or, due to missing data, it can not be determined if it is satisfied or not.

E.g. Condition A (s. Lst. 1) has a valid binding to the ABox $\mathcal{A}_p$ described in Table 1 and is therefore satisfied for the example workflow execution. A valid binding is $\mathcal{B}_A$ = {(PPatient, PPatient); (PTumor, ?tum); (2.1, ?dia)}, since for each of the atoms in Condition A the constraint described in Table 2 holds. For Condition B no valid binding exists. Binding $\mathcal{B}_A$

Atom	Constraint
C(var)	$\exists$ x $\in$ Ind($\mathcal{A}$).((x,var) $\in \mathcal{B}$) $\wedge$ ($\mathcal{A} \models$ C(x))
P(srcvar, tarvar)	$\exists$ x, y $\in$ Ind($\mathcal{A}$).({(x, srcvar), (y, tarvar)} $\subseteq \mathcal{B}$) $\wedge$ ($\mathcal{A} \models$ P(x,y))
cmpOp(var, const)	$\exists$ x $\in$ Ind($\mathcal{A}$). ((x, var) $\in \mathcal{B}$) $\wedge$ (x cmpOp const)

Table 2: Constraints for a binding $\mathcal{B}$ between a condition $\mathcal{C}$ and an ABox $\mathcal{A}$, which have to hold for each atom in $\mathcal{C}$, in order that $\mathcal{B}$ is valid. C is a class description or data type, P is a data type property or an object property, cmpOp is a comparison operator.

is not a valid binding between Condition B and $\mathcal{A}_p$ because the constraint for atom >(?dia, 4) does not hold. Therefore, in the example workflow execution, after the task Surgery the task Chemotherapy A is executed.

3.4 Inconsistencies

A workflow description is consistent if none of the following inconsistencies occurs.

Unsatisfiable condition. If a condition is not satisfied for any workflow execution (has no valid binding to the ABox of its EWDP), we call the condition unsatisfiable for the workflow description. That means that for each workflow execution it is either not satisfied or, due to missing data, it can not be determined if it is satisfied or not. In such a case it can occur that the tasks after the condition are unreachable.

When in the example treatment plan Condition A is replaced with “Patient does not have a tumor”, formalized as (=0 hasTumor.d:Tumor)(PPatient), this condition is unsatisfiable. It is not satisfied for any execution, because each patient for which the forms are filled in has a tumor. The condition “Patient has brain tumor” is also unsatisfiable at this point, since it can not be decided if the patient has a brain tumor from the information filled into the forms. In both cases task Chemotherapy A is unreachable, what is clearly a design error.

XOR-stall. If in a workflow execution none or more than one of the conditions at an XOR-split are satisfied, the task to be executed next can not be determined unambiguously. We call such a situation XOR-stall. In such a case workflow execution aborts.

Semantic Data Inconsistency. If at any point of a workflow execution the calculated ABox is inconsistent wrt. the domain ontology, a Semantic Data Inconsistency occurs. In such a situation the workflow description contradicts the domain ontology causing collected data to be erroneous. In such a case workflow execution aborts.

4 Consistency Checking Algorithm

In the following, we describe an algorithm, which is able to detect unsatisfiable conditions in SWOD workflow descriptions. The algorithm is also able to detect XOR-stalls and Semantic Data Inconsistencies, but to describe detection of these two inconsistencies is not in the scope of this paper. The most obvious algorithm for this problem is a simulation of all possible workflow executions. We have described such an algorithm, called CCABoxes-algorithm, in [Wei09]. This algorithm calculates for each possible workflow path the set of possible ABoxes, called $\mathcal{X}_p$. This algorithm is by definition sound and complete, but it does not always terminate. This is due to the fact that primitive data types (e.g. integer) have infinite data ranges, that means e.g. into a value-item with data type integer an infinite number of different values can be filled in. Since in the CCABoxes-algorithm an ABox is created for all possible executions, this results in the creation of an infinite number of ABoxes. Therefore, we need to replace the calculated set of ABoxes with a finite abstraction, which preserves the information needed to detect the described inconsistencies, to gain a terminating algorithm.

We use an ontology as abstraction, which describes all ABoxes sufficiently to detect the inconsistencies. The resulting "CCOnto-algorithm" calculates a so-called path ontology $\mathcal{O}_p$ for each workflow path. The idea of our abstraction is to represent individuals by newly created classes in $\mathcal{O}_p$. Each individual created during any workflow execution is represented by at least one class, called its corresponding class. The information about the relations of the individual and the classes it belongs to are preserved in $\mathcal{O}_p$ by appropriate restrictions on its corresponding class. A class in $\mathcal{O}_p$ can have an infinite number of corresponding individuals from different ABoxes in $\mathcal{X}_p$. This is possible since data values of primitive data types in the ABoxes are described as data ranges in the classes of $\mathcal{O}_p$.

Furthermore, a so-called condition ontology is created in the CCOnto-algorithm as abstraction of a condition. A condition ontology contains a so-called focal condition class to represent the condition. This class is restricted according to the axioms in the condition. The focal condition class is constructed such that, after a preprocessing step, a class in $\mathcal{O}_p$ is a subclass of the focal condition class if and only if it has a corresponding focal individual in an ABox which has a valid binding to the condition. In the preprocessing step the data ranges relevant for the condition are merged into the data ranges appearing in $\mathcal{O}_p$.

Algorithm 1: CCOnto-algorithm

```
input: SWOD WF description swodWf, domain ontology $\mathcal{O}_D$
Set $\mathcal{W} \leftarrow$ determineWFDPs(swodWf);
foreach WFDP $\in \mathcal{W}$ do
    $\mathcal{O}_{pi} \leftarrow \mathcal{O}_D$;
    $\mathcal{O}_p \leftarrow$ ontoRules($\mathcal{O}_{pi}$, WFDP);
    if $\mathcal{O}_p \equiv$ ERROR(kind, id) then
        if kind $\equiv$ "XOR-Stall" OR "Semantic Data Inconsistency" then
            ABORT with ERROR(kind, id)
        if kind $\equiv$ "Condition unsatisfiable for WFDP" then
            delete WFDP from $\mathcal{W}$
foreach condition con in swodWf do
    if con not in any of WFDP $\in \mathcal{W}$ then
        ABORT with ERROR("Condition unsatisfiable", id);
```

4.1 Outline of Algorithm

We describe in the following the CCOnto-algorithm (see Alg. 1), especially how it detects unsatisfiable conditions. The input for the algorithm is the SWOD workflow description swodWf and the domain ontology $\mathcal{O}_D$. If the algorithm does not abort with an error, swodWf is consistent.

Determine WFDPs. A workflow path for a workflow description is any sequence of tasks, which are connected through transitions and start with the begin task. The path ontology $\mathcal{O}_p$ for a workflow path is an abstraction of the set of all ABoxes $\mathcal{X}_p$, which can be created for any possible workflow execution of this path. That means $\mathcal{X}_p$ contains an ABox for each possible combination of values filled into the items of the workflow path.

To calculate $\mathcal{O}_p$ for a workflow path we have to consider the items in the flow as well as each condition, because the ABoxes for which the condition is not satisfied are not any more possible after its application. We describe this information by a workflow data path (WFDP), which is a sequence of item and condition annotations, according to the following grammar:

wfdp::= wfdp itemAnnotation| wfdp conditionAnnot | itemAnnotation | conditionAnnot;

The first step of the CCOnto-algorithm is to determine all possible workflow paths from swodWF, which end with the end task and store the corresponding WFDPs in a set $\mathcal{W}$. E.g. one of the two WFDPs for the example treatment plan consists of the item annotations IDia, ITum, IMet and Condition A.

Create $\mathcal{O}_p$ for each WFDP in $\mathcal{W}$. The path ontology $\mathcal{O}_p$ is created by the function ontoRules($\mathcal{O}_{Pi}$, WFDP), where $\mathcal{O}_{Pi}$ is the initial path ontology comprising the axioms from the domain ontology (s. Sec. 4.2). The result of the function is either the consistent path ontology or an error term describing an inconsistency which occurred during creation of $\mathcal{O}_p$. If the result is an error term ERROR(kind id) with kind "Condition unsatisfiable for WFDP" the processed WFDP is deleted from $\mathcal{W}$, since the corresponding workflow path can never be taken during workflow execution.

After $\mathcal{O}_p$ is created for each WFDP, it is checked if a condition is unsatisfiable for the workflow by checking if any of the conditions does not appear in any of the WFDPs left in $\mathcal{W}$, which represent the possible flows through the workflow. In that case the algorithm aborts with an appropriate error.

4.2 Construction of Path Ontology

$\mathcal{O}_p$ is derived from the initial path ontology with the calculus OntoRules. The item and condition annotations in WFDP are processed as follows in the order they appear:

- *Item annotation.* For each item annotation the following steps are processed. As an example Fig. 2 depicts $\mathcal{O}_p$ as created from the item annotations in the example treatment plan.
  - *Object variable assertion.* For each object variable in the ontology path a class

is created, which is represented with the same name as the variable, called corresponding class of the variable. The class is declared to be a subclass of the type of the variable and to be disjoint to other classes created from object variables. The class created for the focal variable is called top focal path class and its subclasses are called focal path classes.

  - *Object relation assertion.* For each object relation objProp(X, Y) in the ontology path the axiom (X $\sqsubseteq$ ($\geq$1 objProp.Y)) is added to $\mathcal{O}_p$.
  - *Item constructor.* For a value-item with minimum value mi, maximum value ma, last part dataProp(X, Y) dataType(Y) in its ontology path, the axiom (X $\sqsubseteq$ $\exists$ dataProp.dataType[$\geq$ mi, $\leq$ ma]) is added to $\mathcal{O}_p$. This shows that using an ontology allows the CCOnto-algorithm to work with data ranges instead of single data values, which enables the algorithm to terminate always.

    For an exist-item with last part objProp(X, Y) class(Y) in its ontology path, classes for answer possibilities "yes" and "no" are created as subclasses for each sub class of X, which is a leaf class. A class created for answer possibility "no" has as superclass (=0 objProp.class). A class created for answer possibility "yes" has as superclass ($\geq$ 1 objProp.Y). For the item "Does patient have metastasis?" classes PPatMet and PPatNMet are created (s. Fig. 2).

    For a specify-item, classes for each answer possibility are created, which have as superclasses the according ontology description of the answer possibility and the corresponding class of the associated variable. E.g. for item "type of tumor" classes PTumB, PTumN and PTumO are created (s. Fig. 2).

    For exist- and specify-items $\mathcal{O}_p$ is expanded. We call class X related to Y (resp. Y backwards related to X) if $\mathcal{O}_p \models$ (X $\sqsubseteq$ ($\geq$ 1 objProp.Y)) holds for any object property objProp. $\mathcal{O}_p$ is expanded if any class X is related to any superclass S of a set of newly created classes setNew. Then for each class R $\in$ setNew a new class N is created as a subclass of X, and N is related to R. For the newly created classes the process is repeated. E.g all subclasses of PPatMet and PPatNMet are created in the expansion step for item "type of tumor".

- *Condition Annotation.* When a condition is applied to $\mathcal{O}_p$, all classes, which are corresponding to individuals belonging to ABoxes $\mathcal{A}_p$, for which the condition is not satisfied, have to be deleted. Therefore, the condition ontology is created, $\mathcal{O}_p$ is preprocessed for the condition, and it is reduced to its focal path classes that are subsumed by the focal condition class, to superclasses of the subsumed classes and to classes which are backwards related to these classes (for details s. below).
  - *Condition unsatisfiable.* If $\mathcal{O}_p$ has no more path classes after applying a condition, the condition is unsatisfiable for this WFDP. Then $\mathcal{O}_p$ is set to ERROR("Condition unsatisfiable for WFDP", conditionId).

Construction of Condition Ontology. We have defined a calculus to derive the condition ontology from a condition. For each object variable in the condition, a class is created called condition variable class (CC). CC is set equal to class $\top$. We call the conjunction of classes, which are set equal to CC, condition variable equal class (CEC). For each atom in the condition the condition ontology is extended as follows:

- *class(x)*: The CEC of variable x is set to (R $\sqcap$ class), where R is the CEC of x before the application of the atom.
- *objProp(x, y)*: The CEC of variable x is set to (R $\sqcap$ ($\geq$ 1 objProp.Z)), where R is the CEC of x and Z is the CC of y before the application of the atom.
- *dataProp(x, y)*: The CEC of variable x is set to (R $\sqcap$ ($\exists$ dataProp)), where R is the CEC of x before the application of the atom.
- *dataType(y)* The CEC of each variable x, which is related with a data property dataProp to y is set to (P $\sqcap$ ($\exists$ dataProp.datatype)) resp. (P $\sqcap$ ($\exists$ dataProp.odatatype $\cap$ datatype)), when it was before the atom was applied (P $\sqcap$ ($\exists$ dataProp) resp. (P $\sqcap$ ($\exists$ dataProp.odatatype).
- *cmpOp(y, const)*: The CEC of each variable x, which is related with a data property dataProp to y is set to (P $\sqcap$ ($\exists$ dataProp.[cmpOp const])) resp. (P $\sqcap$ ($\exists$ dataProp.odatatype $\cap$ [cmpOp const])), when it was before (P $\sqcap$ ($\exists$ dataProp) resp. (P $\sqcap$ ($\exists$ dataProp.odatatype).

As a first example we consider that Condition A in the example treatment plan is replaced with "Patient has metastasis" (d:Metastasis(?met) $\wedge$ hasMetastasis(PPatient, ?met)). We apply the condition to $\mathcal{O}_p$ (s. Fig. 2) to determine if it is satisfiable for the WFDP. The condition ontology is as follows, where CPatient is the focal condition class:
CPatient $\equiv$ $\geq$1 hasMetastasis.CMet
CMet $\equiv$ d:Metastasis
In this example it is not necessary to preprocess $\mathcal{O}_p$. The condition is satisfiable for the workflow path, since the focal path class PPatMet can be inferred as a subclass of CPatient. Therefore, after applying the condition, $\mathcal{O}_p$ comprises the classes PPatMet, its subclasses, its superclass PPatient and the classes, which are backwards related to these classes, which are PTumor, its subclasses and PMet. Class PPatNMet and its subclasses are deleted from $\mathcal{O}_p$.

Preprocessing path ontology in order to apply condition. To preprocess $\mathcal{O}_p$ for a condition, $\mathcal{O}_p$ is for each data type variable dVar in the condition extended as follows. Let dRC be the data range of dVar and dataProp be the data property, which relates an object variable to dvar. Let L be a leaf class in $\mathcal{O}_p$, for which holds $\mathcal{O}_p \models$ L $\sqsubseteq$ ($\exists$ dataProp.dRP) and dRP overlaps with dRC, for any data range dRP. Then for each such class L, two subclasses are created, one as a subclass of ($\exists$ dataProp.dRP $\cap$ dRC) and the other as a subclass of ($\exists$ dataProp.dRP$\setminus$dRC). $\mathcal{O}_p$ is expanded as described above.

As an example we consider Condition A as described in Lst. 1. The derived condition ontology is:
CPatient $\equiv$ $\geq$1 hasTumor.CTumor
CTumor $\equiv$ d:Tumor $\sqcap$ $\exists$ hasDiameter.float[$\leq$ 4]
$\mathcal{O}_p$ is preprocessed for the condition as follows. For the only data type variable ?dia the data range is float[$\leq$ 4]. Object variable ?tum is related with property hasDiameter to ?dia. The classes PTumO, PTumB and PTumN are subclasses of data type restriction ($\exists$ hasDiameter.float[$\leq$ 20]). For each of them two subclasses are created, which we call small tumor and big tumor class, for PTumB e.g. big tumor class PTumBG and small tumor class PTumBL are created and thus following axioms are added to $\mathcal{O}_p$:

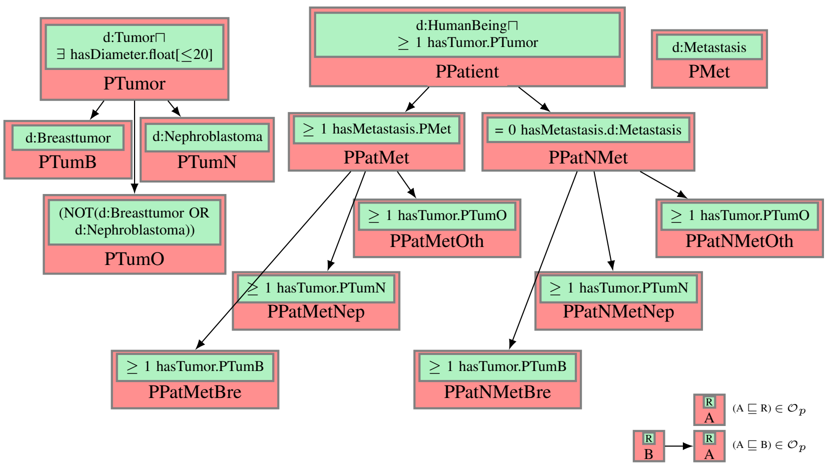

Figure 2: Main content of $\mathcal{O}_p$ for example treatment plan after processing all item annotations.

$\mathsf{PTumBL} \sqsubseteq \mathsf{PTumB} \sqcap (\exists\ \mathsf{hasDiameter.float}[\leq 4])$
$\mathsf{PTumBG} \sqsubseteq \mathsf{PTumB} \sqcap (\exists\ \mathsf{hasDiameter.float}[> 4, \leq 20])$
In the expansion step for each focal path class, which is a leaf class, two subclasses are created, one related to the appropriate small tumor class and the other related to the appropriate big tumor class.

When the condition is applied to the preprocessed $\mathcal{O}_p$, it can be detected that the condition is satisfiable for WFDP, since all focal path classes, which are related to small tumor classes, are subclasses of the focal condition class. After applying the condition, all focal path classes related to big tumor classes are deleted from $\mathcal{O}_p$ as well as all big tumor classes itself, since they are no longer backwards related to any focal path class.

Using “Patient does not have a tumor”, formalized as $(\mathsf{=0\ hasTumor.d{:}Tumor})(\mathsf{PPatient})$, as a replacement of Condition A is an example for an unsatisfiable condition. The condition ontology is: $\mathsf{CPatient} \equiv (\mathsf{=0\ hasTumor.d{:}Tumor})$. None of the focal path classes in $\mathcal{O}_p$ is a subclass of $\mathsf{CPatient}$, since all are subclasses of $(\geq 1\ \mathsf{hasTumor.PTumor})$. That means the condition is unsatisfiable for the WFDP and since Condition A appears in only one WFDP, it is unsatisfiable for the workflow description.

4.3 Soundness and Completeness

In this section we sketch a proof showing that detection of unsatisfiable conditions with the CCOnto-algorithm is complete, i.e. all inconsistencies are detected, and sound, i.e. detected inconsistencies are inconsistencies. From the definition of the inconsistency follows soundness and completeness for the CCABoxes-algorithm. Completeness and soundness of the CCOnto-algorithm is proved by showing that it detects the inconsistency in the same cases as the CCABoxes-algorithm, as outlined in the following. The steps of the two algorithms are the same, with the difference, that CCOnto utilizes $\mathcal{O}_p$ instead of a set of ABoxes, $\mathcal{X}_p$, to detect inconsistencies. We call $\mathcal{O}_p$ derived from WFDP with calculus OntoRules an abstraction of $\mathcal{X}_p$ derived from the same WFDP with calculus SOABoxesRules (utilized in the CCABoxes-algorithm). We can prove by induction over the rules of the two calculi, that following abstraction criteria hold between $\mathcal{X}_p$ and its abstraction $\mathcal{O}_p$: Each individual in any ABox of $\mathcal{X}_p$ has a corresponding class in $\mathcal{O}_p$ and each leaf class in $\mathcal{O}_p$ has a corresponding individual in any ABox of $\mathcal{X}_p$.

To prove that both algorithms detect the inconsistency in the same cases, we have to prove that $\mathcal{O}_p$ for a workflow data path ending at the condition is set to an error term "Condition unsatisfiable for EWDP" if and only if $\mathcal{X}_p$ is set to an error term. $\mathcal{O}_p$ is set to an error term if $\mathcal{O}_p$ has no path classes after applying the condition and $\mathcal{X}_p$ is set to an error term if $\mathcal{X}_p$ is empty after applying the condition. From the abstraction criteria, it follows that $\mathcal{O}_p$ has no path classes if and only if $\mathcal{X}_p$ is empty. (for full proof and definition of corresponding individuals and classes s. [Wei09]).

5 Discussion

We have described a static analysis technique to detect unsatisfiable conditions in workflows with an ontology-based data perspective. This technique has the capability to increase reliability of workflow executions.

Implementation. A prototypical implementation of the CCOnto-algorithm is currently developed in Java using the OWL API [HBN07] and the DL-reasoner Pellet [SPG$^+$07]. It is intended to be integrated into ObTiMA, to support users in defining consistent treatment plans.

Related Work. Consistency checking algorithms exist to check structural consistency (e.g. [VvdAtH07], [QXW$^+$07]), but few integrate data (e.g. [SZ$^+$06] or [Esh02]). These techniques have only limited ability to detect unsatisfiable conditions in workflows. Sun et al. [SZ$^+$06] developed a framework for detecting data flow anomalies, which is e.g. capable of detecting missing data in conditions, but is not able to check the satisfiability of conditions in a workflow. Eshuis [Esh02] describes a framework for verification of workflows based on model checking. His framework is able to check satisfiability of conditions, but these checks consider only Boolean expressions and their dependencies.

We are not aware of an algorithm, which is able to detect unsatisfiable conditions in workflows with complex data perspectives based on semantic annotations.

Future Work. We plan to extend the data perspective of SWOD e.g. by allowing to create more complex items from the ontology and to define constants or constraints between items. We aim for a more expressive control flow perspective comprising e.g. cyclic workflows with respect to research on workflow-patterns [RtHvdAM06]. We plan to consider time in the data and control flow perspectives.

Conclusion. Ontology-based data perspectives in data intensive workflows are well suited to provide the basis to detect unsatisfiable conditions. Integrated with existing algorithms for checking structural consistency (e.g. [VvdAtH07], [QXW+07]) the technique described here can have the capability to guarantee soundness of complex workflows.

References

[BCM+03] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook, Theory Implementation and Applications*. Cambridge University Press, 2003.

[BWC+08] M. Brochhausen, G. Weiler, C. Cocos, H. Stenzhorn, N. Graf, M Doerr, and M. Tsiknakis. The ACGT Master Ontology on Cancer - A new Terminology Source for Oncological Practice. In *Proc. of 21st IEEE International Symposium on Computer-Based Medical Systems*, pages 324–329, 2008.

[Esh02] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workow Modelling*. PhD thesis, University of Twente, http://www.ctit.utwente.nl/library/phd/eshuis.pdf, 2002.

[HBN07] M. Horridge, S. Bechhofer, and O. Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. In *OWLED 2007, 3rd OWL Experiences and Directions Workshop, Innsbruck, Austria*, June 2007.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language. W3C Member Submission, 2004.

[QXW+07] Yi Qian, Yuming Xu, Zheng Wang, Geguang Pu, Huibiao Zhu, and Chao Cai. Tool Support for BPEL Verification in ActiveBPEL Engine. In *Proceedings of the 2007 Australian Software Engineering Conference*, pages 90–100, 2007.

[RtHvdAM06] N. Russel, A. ter Hofstede, W.M.P van der Aalst, and N Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPMcenter.org, 2006.

[SPG+07] B. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2), 2007.

[SZ+06] X.S. Sun, J.L. Zhao, et al. Formulating the Data-Flow Perspective for Business Process Management. *Information Systems Research*, 17(4):374–391, 2006.

[TB+08] M. Tsiknakis, M. Brochhausen, et al. A Semantic Grid Infrastructure Enabling Integrated Access and Analysis of Multilevel Biomedical Data in Support of Postgenomic Clinical Trials on Cancer . *IEEE Transactions on Information Technology in Biomedicine*, 12(2):205–217, 2008.

[VvdAtH07] H.M.W. Verbeek, W.M.P van der Aalst, and A.H.M. ter Hofstede. Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. *Computer Journal*, 50(3):294–314, 2007.

[Wei09] G. Weiler. Consistency Checking for Workflows with an Ontology-Based Data Perspective. unpublished, available at: http://softech.informatik.uni-kl.de/twiki/bin/view/Homepage/PublikationsDetail?id=133, 2009.

[WPHK09] G. Weiler, A. Poetzsch-Heffter, and S. Kiefer. Consistency Checking for Workflows with an Ontology-Based Data Perspective. In *20th International Conference on Database and Expert Systems Applications*, 2009.