Extending OWL with Explicit Dependency

Jean-Paul Calbimonte, Fabio Porto

Ecole Politechnique Fédéral de Lausanne – EPFL
Database Laboratory - Switzerland
firstname.lastname@epfl.ch

Abstract. Functional Dependency has been extensively studied in database theory. It provides an elegant formalism for specifying key constraints and is the basis for normalization theory used in Relational database design. Given its known axiomatization through logical implications it is expected that the ontology community would be interested in investigating its applicability to conceptual modeling. This paper investigates the extension of OWL ontologies with functional dependencies. In particular, we focus on key dependency and a new class referred to as explicit dependency. The latter is a functional dependency in which the function that correlates the left-hand side to the right-hand side of the dependency is known. We extend current formalism to allow functions to appear in the consequent of a dependency declaration. A mapping of the proposed formalism to Horn clause is specified and permits FDs to be evaluated as logic rules. The ontology therein produced may present different behaviors depending on the interpretation given to functional dependencies. Interesting enough is that explicit dependency adds one extra possible interpretation, similar to views in databases. We have integrated the extended formalism into OWL and used SWRL to map dependencies to logic rules. A prototype has been implemented and illustrates the proposed approach.

1. Introduction

Data dependency has been introduced as a general model for a large class of database constraints that augments the expressivity of simple database models [1]. Functional dependencies (FD) compose a particularly interesting data dependency [2] with which one can design correct Relational database schemas. This is due to the fact that FDs elegantly model the type of relationships existing between attributes of a relation, used, for instance, in defining primary keys and in avoiding redundant data representation by applying normalization rules.
The semantics expressed through functional dependencies are equally relevant when specifying a conceptual schema. As a matter of fact, it has been observed [3] that in data-centric applications users expect ontologies to offer mechanisms similar to those found in the database area that guarantee the correctness of entered data. There has been quite a few works investigating the adoption of FD in ontologies [4,5,6,7]. In these works, functional dependency (including keys) is incorporated into variants of description logic languages and an analysis of decidability of the new language is provided. Albeit the differences in expressivity of the different FD approaches, the semantics of the introduced FD construct remain essentially invariant. A semi-formal definition can be stated as:

Given a concept C with sets of properties X and Y, we say that X determines Y ($X \rightarrow Y$) iff given two instances I_1, I_2 of C with two bindings (x_1, x_2) of X and (y_1, y_2) of Y, where $x_i, y_i \in I_i$, then if $x_1 = x_2$ implies $y_1 = y_2$ **(1)**

As an example of functional dependency that follows the semantics described in (1) and models flight information, one may specify that for a given flight number and date a single pilot can be allocated:

$$Flight: hasNumber, hasDate \rightarrow hasPilot \quad \textbf{(2)}$$

The semantics in (1) is suitable for specifying keys of concepts and to constrain the cardinality of dependent properties. In fact, OWL offers the Functional (respectively InverseFunctional) property construct that restricts the maximal cardinality of a single property to 1 [8].
In this paper we extend the class of data dependencies that can be expressed through functional dependencies. Consider for instance the rule requiring pilots of a flight to hold passports issued by the same country as the one represented by the airline company flag. This rule could be stated as:

hasAirline.hasFlag(x) and
hasPilot.hasPassport.issuedBy (y) -> x=y **(3)**

Observe that differently to the expression in (2), the one in (3) implies an explicit functional relationship between the instances in the range of predicates *hasFlag* and *isIssuedBy*. Thus, in addition to specifying constraint cardinality, the rule in (3) specifies the value the dependent property shall hold as a function of the determining property value. We refer to this class of functional dependency as Explicit Dependency (ED), emphasizing the explicit specification of the dependency function.

In other to accommodate ED with keys and traditional functional dependency definitions, we extend current formalism by allowing functions to appear in the consequent of a FD. As proposed in the database literature [1], a Horn clause representation is adopted, allowing natural mapping of FD formal definition into logic rules. The ontology obtained by adding FDs may induce different behaviors according to the interpretation given to FDs [3]. Basically, the business rules implied by FDs can be interpreted as constraints, as in databases, or as part of the theory (TBox) affecting reasoning. Interesting enough, EDs allow further interpretation, similar to views in databases.

The extended FD specification has been integrated into OWL and a mapping to SWRL [9] language allows rules to be evaluated and results to be integrated back into OWL. An initial prototype has been implemented validating the proposed extensions.

The remaining of this paper is organized as follows. Section 2 discusses related work. Next, Section 3 motivates the need for functional dependencies in ontologies. A formal framework for extending DL ontologies with functional dependencies is presented in Section 4. Section 5 discusses the extension of OWL with FD and Section 6 describes our prototype implementation. Finally, Section 7 concludes.

2. Related Work

Functional dependency has been extensively studied in databases with the main intention of adding more semantics to database schemas [1]. Important results of FD in databases include the specification of key dependency within a relation and referential integrity between relations, known as inclusion dependency. Although inclusion dependency is placed in the realm of ontology languages, key dependency has long being neglected. Most of the work in DL concerning FD explores alternatives to integrating the new feature into the language and analyze the decidability and complexity of satisfiability of axioms under the extended language.

Calvanese et al. [4] extend $\mathcal{DLR}$ with multiple-keys (keys constituted by more than one attribute) for concepts and n-ary relations. Components of keys are simple properties that cannot map into a path of relationships [10]. Keys are in fact modeled as cardinality constraints over roles in concept constructors and relations without requiring further extension to the language, which permitted to keep the satisfiability of concepts in EXPTIME.

Borgida and Wedell [10] extend the Classic description logics with functional dependencies and show how subsumption and views can be computed using description graphs even in the presence of FDs. An interesting point raised by the authors is the integration of FD definitions with the TBox. In particular, the extension of DL through extra constructs for FD representation may fail to provide any interesting deduction based on associated concepts relationships. In order to circumvent this problem the FD interpretation is specified with reference to the domain Δ making it valid for any subsumption assertion related to the FD definition. In our approach, as FDs are mapped into SWRL rules, the antecedent formula is composed of conjunctions of roles forming paths with a common root concept. In this context, any ABox instances matching the predicates fires the rule, including those related to the involved terms through subsumption relationship with the respective classes. The representation of views in Classic was also inspiring, as described in Section 4.

Lutz and Milicic [6] propose an extension of description logics $\mathcal{ALC}(\mathcal{D})$ with concrete domains to support functional dependencies with paths in the antecedents and consequents. A set of functional dependencies over a concept *C* forms a key-box. The work shows that for large class of practical scenario key-boxes, computing the satisfiability of concepts is decidable. Our formalization for FDs equally considers paths in composing the left and right hand side of its implication with the difference of not distinguishing between concrete and abstract features.

Toman and Weddell [7] investigate the decidability of a terminology in $\mathcal{DLFD}$ when FDs are part of concept constructors using paths. It is shown that decidability is maintained with path functional dependencies iff the latter is restricted to appear in the right-hand side of monotone concept

constructors. Our results do not consider FD in concept constructor; rather we add a new constructor for FDs and integrate it to the TBox through logic rules derived from the FD formulation.

Motik and colleagues [3] elegantly discuss the role of constraints in ontologies and trace an interesting comparison to constraints in databases. Indeed, the very same business rule may be interpreted as terminological axiom or as a constraint. The authors extend the definition of a knowledge base to admit a special constraint component that adverts against non-compliance to the rules without being part of the terminology. The constraint component is one of the elements of our conceptual model with very similar behavior.

Finally, Ludascher et al [11] introduced a distinguishable witness predicate for holding instances not conforming to specified constraints. The approach we adopted for integrating constraints enforcement to our framework was inspired by the aforementioned proposal.

3. Preliminaries

As we have seen, functional dependencies can be used to express different types of relationships between ontology elements. In classical FDs and the special case of keys, the function that links the antecedent and consequent is a selection function, which is based on semantic interpretations of the domain of interest. For instance consider the following example in the flight ontology: *"the destination airport of a flight functionally determines the departure gate"*. This constraint would ensure that passengers from two different flights with the same destination should pass through the same gate. In terms of DL, let's suppose that we have defined the flights `F1` and `F2`, gates `G1`, `G2` and airport `HEATHROW`:

```
hasDestination(F1,HEATHROW)
hasDestination(F2,HEATHROW)
departsThrough(F1,G1)
```

The ABox assertions above would imply that it is not possible to have the following assertion:

```
departsThrough(F2,G2)
```

It would indeed violate the FD. However the selection of `G1` instead of `G2` is rather arbitrary. It depends absolutely on the user's perception and interpretation of reality. In fact if we change `G1` by `G2` for both flights, the FD constraint is respected anyway.
This is understandable because the function that establishes a relationship between antecedent and consequent is not explicitly stated. But there are cases where we might want to formally represent this function in the FD declaration. Consider this example: *"the flight origin and destination functionally determine the price of the ticket"*. In this case the ticket price depends on a known function, which takes the flight origin and destination as parameters. For example, we could calculate the price based on the distance of the two airports. Now we cannot arbitrarily choose between all possible price values, the function computes it and all instances should agree on it:

```
hasDestination(F1,HEATHROW)          hasDestination(F2,HEATHROW)
hasOrigin(F1,GENEVA)                 hasOrigin(F2,GENEVA)
ticketForFlight(TICKET1,F1)          ticketForFlight(TICKET2,F2)
price(T1,300)                        price(T2,300)
```

In the example we see that two completely different flights that agree on origin and destination (`GENEVA` and `HEATHROW`), also agree on the ticket price, which is computed by a function *f*.

So we see the need of defining all these flavours of FDs in DL. In OWL-DL, only basic FDs are expressible thanks to the `FunctionalProperty` and the `InverseFunctionalProperty`. If a property *P* is declared as functional it satisfies the following rule for all instances *x*, *y* and *z*:

$$P(x,y) \wedge P(x,z) \rightarrow y = z \qquad (4)$$

And for the inverse case:

$$P(y,x) \wedge P(z,x) \rightarrow y = z \qquad (5)$$

It is easy to see that this is very limited in terms of expressivity, not even compound keys or FDs can be constructed. By using the key-box constructs introduced in [6] and Path-FD's of [7] we are able to

define quite complex FD but we are required to modify the reasoners in order to check compliance of the theory to the FDs. Moreover, we are anyway unable to express explicit functions with FDs like in the ticket price example.

It is well known that there is an intrinsic relation between Horn clauses and functional dependencies, as described in [2] in the context of databases. So it seems natural to use Horn clauses for FDs in DL as well. First we analyze the general and common case of a classical FD. Consider the flight and gate example described previously. We can translate the FD as a Horn clause rule in the following way:

$$hasDestination(x_1, y) \wedge hasDestination(x_2, y) \wedge departsThrough(x_1, z_1) \wedge hasDestination(x_2, z_2) \rightarrow z_1 = z_2 \quad (6)$$

It is interesting to see that similar rules can be constructed for the case of keys. For instance if we define that *"two passports issued for the same country and the same passport numbers are the same"*, then we can express this constraint with the following rule:

$$issuedInCountry(x_1, y) \wedge issuedInCountry(x_2, y) \wedge passNumber(x_1, z) \wedge passNumber(x_2, z) \rightarrow x_1 = x_2 \quad (7)$$

Finally we can write a similar rule for the ticket price example:

$$ticketForFlight(t, x_1) \wedge hasOrigin(x_1, w) \wedge hasDestination(x_1, z) \rightarrow price(t, f(w, z)) \quad (8)$$

Notice that *f* is a known deterministic function that takes two instances, the origin and destination, and returns a price value.

The examples above denote a requirement to a more expressive DL knowledge base. Some proposals suggest the extension of DL with functional dependency and keys but, to the best of our knowledge, no work has considered *explicit dependency*. Section 4 describes a formal framework that accommodates: keys, functional and explicit dependencies.

4. Formal Framework

Section 3 illustrates that complex keys and explicit dependency definitions are relevant business rules that require an extended DL knowledge base to be expressed. The extension we propose is introduced by a new concept FD that is added to a DL knowledge base. In fact, instances of FD are rules descriptions that are mapped to logic rules using a rule language. We study the consequence of adding FDs as rules and provide three interpretations for their evaluation. As a result of these different interpretations our conceptual model distinguishes elements of the terminology and the assertion base therein produced.

4.1. Conceptual Model

Functional dependencies establish particular relationships between elements of the knowledge base. Keys, for example, induce equality of values between non-key properties for instances identified to be the same. Similarly, explicit dependencies relate values of dependent properties to those of determining ones by means of an explicit function. The referred relationships are expressed by means of predicates in the head of FD rules that, as a result of rule evaluation, may produce new facts. Thus, one may say that the facts produced by running FD rules correspond to the subset of assertions in the knowledge base that agrees on the FD relationship. In this respect, an important issue is to analyze the correspondence of such subset with the ABox without FD inferences.
We propose three different interpretations:

- Assertions – usual ABox facts;
- Constraints– witnesses [11] facts about ABox assertions conflicting with the FD rule;
- Views – facts whose mapping to the FD rule defines a necessary and sufficient condition for belonging to the resulting head predicate [10];

In order to accommodate these different interpretations we extend the conceptual model proposed in [3] according to the following definition.

Definition 1: An extended DL-FD knowledge base is a quintuple $\mathcal{K}=(\mathcal{T},\mathcal{A},C,C_{\mathcal{A}},\mathcal{V},\mathcal{FD})$ such that (**9**)

- $\mathcal{T}$ is a finite set of standard TBox axioms,
- $\mathcal{A}$ is a finite set of standard ABox assertions,
- C is a finite set of axioms specifying constraints (i.e. witnesses),
- $C_{\mathcal{A}}$ is a finite set of assertion hurting some constraint in C and expressed as *witnesses* facts,
- $\mathcal{V}$ is a finite set of view definitions of type $\mathcal{V}=\{v_1 \equiv fdv_1, \ldots, v_n \equiv fdv_n\}$, where $fdv_i \in \mathcal{FDV} \subseteq \mathcal{FD}$, and
- $\mathcal{FD}$ is a finite set of functional dependency definition instances with $\mathcal{FDa} \cup \mathcal{FDc} \cup \mathcal{FDv} \cup \mathcal{FDk} = \mathcal{FD}$, where
  - $\mathcal{FDa}$ – is a finite set of assertion FDs,
  - $\mathcal{FDc}$ – is a finite set of constraint FDs,
  - $\mathcal{FDv}$ – is a finite set of views FDs, and
  - $\mathcal{FDk}$ – is a finite set of key FDs.

In Definition 1, we have that elements of $\mathcal{K}$ are pair wise disjoint. The constraint interpretation corresponds to the behavior in which a ABox is searched for assertions not conforming to a FD rule. Note that predicates reflecting the FD constraint must be negated in the body of the rule. ABox assertions conforming to the paths in the FD rule body lead to the literals that match with the variables of a particular *witness* predicate in the rule head. *Witnesses* are specified as distinguished constraint concepts in C. The produced *witness* facts are added to the constraint assertion box $C_{\mathcal{A}}$.
The view interpretation specifies queries whose answers are computed by the *explicit dependency* function over determining property values. The view characterization defers from simple assertions in that the FD rule definition specifies necessary and sufficient conditions for ABox assertions to match with predicates in FD. In fact, computing view results is only dependent on predicates on the body of FD rule. $\mathcal{V}$ comprehends view labels mapped to corresponding $\mathcal{FDV}$ instances. Observe that the equivalence between a $\mathcal{V}$ instance and an $\mathcal{FDV}$ one allows querying from both $\mathcal{FDV}_i$ or view evaluation. The consequent of an $\mathcal{FDV}$ rule is a predicate defined in $\mathcal{T}$ and having the resulting assertions due to the view evaluation added to the usual ABox. One may clearly observe that $\mathcal{V} \cap \mathcal{FDk} = \varnothing$.
The assertion interpretation assumes that the results of $\mathcal{FDa}$ evaluation are added to the set $\mathcal{A}$. Predicates in the head of an $\mathcal{FDa}$ functional dependency are defined, as usual, in $\mathcal{T}$. Thus, is expected that assertions corroborate to the complete theory as expressed by $\mathcal{T} \cup \mathcal{A}$.

4.2. Formalizing Functional Dependencies

A FD definition *fd* is composed of the following elements: the antecedent A, consequent C, a root concept R and eventually a deterministic function *f*:

$$fd = (A, C, R, f) \quad (\mathbf{10})$$

The antecedent is a list of paths. A path u_i is in turn composed of roles r_i:

$$\begin{aligned} A &= \{u_1, u_2, \ldots, u_n\} \\ u_i &= \{r_{i,1}, r_{i,2}, \ldots, r_{i,n_i}\} \\ C &= \{u\} \\ u &= \{s_1, s_2, \ldots, s_l\} \end{aligned} \quad (\mathbf{11})$$

The consequent is defined by a single path u, which is composed of l roles u_i. The root concept R is the starting point of all paths in the antecedent and consequent. This ensures that all paths in the FD definition have a common source, which serves as a linking element between them.

In case of having the deterministic function *f* defined, it takes as parameters the ranges of the last roles of the antecedent paths. And the result of *f* must be comparable to the range of the last role of the path in the consequent.
If the FD is a key FD, then the consequent is the instance of the root concept itself (*Id*).

In the simple example of the passport with a key FD, the *fd* definition would be composed of the following antecedent, consequent and root concept:

$$A = \{u_1, u_2\} \;\; C = \{u\} \;\; R = Passport \quad (12)$$
$$u_1 = \{issuedInCountry\}$$
$$u_2 = \{passNumber\}$$
$$u = \{Id\}$$

Or in a simplified way, we could write it as:

$$Passport : \{issuedInCountry\}, \{passNumber\} \rightarrow \{Id\} \quad (13)$$

For the more complex case of the ticket price we would have:

$$A = \{u_1, u_2\} \;\; C = \{u\} \;\; R = Ticket \;\; f = f_{ticket} \quad (14)$$
$$u_1 = \{ticketForFlight, hasOrigin\}$$
$$u_2 = \{ticketForFlight, hasDestination\}$$
$$u = \{price\}$$

Or in simplified notation:

$$Ticket : \{ticketForFlight, hasOrigin\}, \quad (15)$$
$$\{ticketForFlight, hasDestination\} \xrightarrow{f_{ticket}} \{price\}$$

We can see that these abstract FD definitions can be mapped to their corresponding Horn clause rule counterparts (as seen in expressions (6), (7) and (8)). In the general case the abstract FD definition in (11) can be translated to the following Horn rule:

$$r_{1,1}(a, p_{1,1}) \wedge r_{1,2}(p_{1,1}, p_{1,2}) \wedge \ldots \wedge r_{1,n_1}(p_{1,n_1-1}, p_{1,n_1}) \wedge \quad (16)$$
$$r_{2,1}(a, p_{2,1}) \wedge r_{2,2}(p_{2,1}, p_{2,2}) \wedge \ldots \wedge r_{2,n_2}(p_{2,n_2-1}, p_{2,n_2}) \wedge$$
$$\ldots \wedge$$
$$r_{i,1}(a, p_{i,1}) \wedge r_{i,2}(p_{i,1}, p_{i,2}) \wedge \ldots \wedge r_{i,n_i}(p_{i,n_i-1}, p_{i,n_i}) \wedge$$
$$s_1(a, p_1) \wedge s_2(p_1, p_2) \wedge \ldots \wedge s_{l-1}(p_{l-2}, p_{l-1})$$
$$\rightarrow s_l(p_{l-1}, f(p_{1,n_1}, p_{2,n_2}, \ldots, p_{i,n_i}))$$

The $p_{i,j}$ and a elements are variables in the rule language. The variable a is the common root node linking all the paths in the antecedent and consequent of the FD.

Notice that the formalization in (16) implies a new assertion interpretation. The head of the rule is the new information added to the theory. In the following sub-section we discuss the different types of interpretations of functional dependencies within our framework.

4.3. FD Interpretation

As we have introduced in Section 4.1, depending on the kind of FD enforcement, its results can be interpreted differently. We have identified three basic cases of FD interpretations: as constraint, new assertions and views.

To better understand the different usages of FD dependencies, consider the following example, again in the context of the 'flight ontology': "*The tax on a ticket price functionally depends on the passenger age-group, the departure airport and the arrival airport*". This rule may make sense if for instance a special ticket tax is added to the price depending of the age of the passenger and the airport

he is arriving to. The following graph will help us to represent the concepts and roles involved in the aforementioned rule:

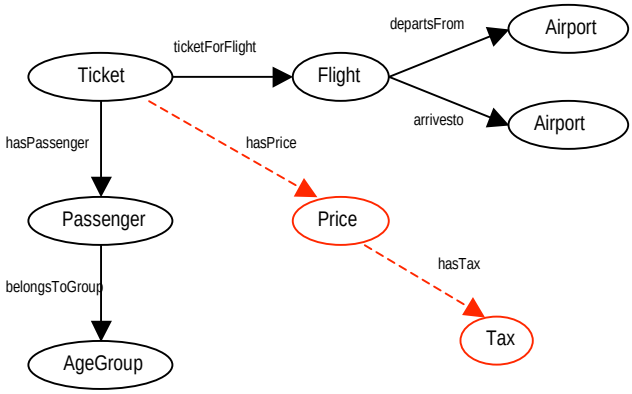

Fig. 1. Graph of concepts and roles involved in the ticket price tax example.

We can easily identify the following paths for the antecedent:

- $\{ticketForFlight, departsFrom\}$
- $\{ticketForFligh, arrivesTo\}$
- $\{hasPassenger, AgeGroup\}$

And the single path for the consequent (dashed arrows in Fig. 1):

- $\{hasPrice,hasTax\}$

In addition, we define f_{tax} as the function that computes the tax based on the departure, arrival and age group:

- $tax = f_{tax}(departureAirport,arrivalAirport,ageGroup)$

Our FD is then defined as:

$$fd_{tax} : (A, C, Ticket, f_{tax}) \tag{17}$$

$$A = \left\{ \begin{array}{l} \{ticketForFlight, deprtsFrom\}, \\ \{ticketForFlight, arrivesTo\}, \\ \{hasPassenger, ageGroup\} \end{array} \right\}$$

$$C = \{\{hasPrice, hasTax\}\}$$

In the Horn-rule version:

$$\begin{aligned} & ticketForFlight(a, p_{1,1}) \wedge departsFrom(p_{1,1}, p_{1,2}) \wedge \\ & ticketForFlight(a, p_{2,1}) \wedge arrivesTo(p_{2,1}, p_{2,2}) \wedge \\ & hasPassenger(a, p_{3,1}) \wedge ageGroup(p_{3,1}, p_{3,2}) \wedge \\ & hasPrice(a, p_1) \\ & \rightarrow hasTax(p_1, f(p_{1,2}, p_{2,2} p_{3,2})) \end{aligned} \tag{18}$$

In this case, if FDs are interpreted as assertions, the head of the rule express a new fact, which is added to the ontology as a consequence of FD evaluation. In the current scenario, the *hasTax* predicate specifies the value to be paid as computed by the know function "*f*" applied over the determining properties:

$$hasTax(p_1, f(p_{1,2}, p_{2,2} p_{3,2})) \tag{19}$$

This assertion would be added to the $\mathcal{A}$ set of the knowledge base. The property *hasTax* should exist in the $\mathcal{T}$ set of the knowledge base.

If the FD above is treated as a constraint, then we need to change the rule in such a way that we add a witness in case the theory does not obey the FDs. The rules would have to include a negation in the last predicate of the consequent:

$$ticketForFlight(a, p_{1,1}) \wedge departsFrom(p_{1,1}, p_{1,2}) \wedge \quad (20)$$
$$ticketForFlight(a, p_{2,1}) \wedge arrivesTo(p_{2,1}, p_{2,2}) \wedge$$
$$hasPassenger(a, p_{3,1}) \wedge ageGroup(p_{3,1}, p_{3,2}) \wedge$$
$$has\Pr ice(a, p_1) \wedge \neg hasTax(p_1, f(p_{1,2}, p_{2,2}p_{3,2}))$$
$$\rightarrow w_{tax}(fd_{tax}, hasTax(p_1, f(p_{1,2}, p_{2,2}p_{3,2})))$$

Notice that the witness w_{tax} indicates: the failing instance, the FD and the predicate that caused the failure. This witness would be added to the $C_{\mathcal{A}}$ set of the knowledge base. Notice that the witness can grow in complexity, similarly to exception handling classes in OO modelling. The taxonomy of the witness classes is defined in the C set of the knowledge base.

Finally, ED rules may behave like views in databases by specifying values of a dependent property as derived from the function evaluation over determining properties. Views are specified in the $\mathcal{V}$ set and correspond to labels associated to FD clauses. In the head of a view FD, a predicate, specified as part of the $\mathcal{T}$ set, is filled with matching values resulting from rule evaluation. Observe that differently to the two other interpretations, a user can explicit invoke a view FD. Clause (21) illustrates a view FD for the example (17). Note that syntactically it is equivalent to the one presented in (18).

$$ticketForFlight(a, p_{1,1}) \wedge departsFrom(p_{1,1}, p_{1,2}) \wedge \quad (21)$$
$$ticketForFlight(a, p_{2,1}) \wedge arrivesTo(p_{2,1}, p_{2,2}) \wedge$$
$$hasPassenger(a, p_{3,1}) \wedge ageGroup(p_{3,1}, p_{3,2}) \wedge$$
$$has\Pr ice(a, p_1)$$
$$\rightarrow hasTax(p_1, f(p_{1,2}, p_{2,2}p_{3,2}))$$

Having defined FDs formally and having explained the different interpretations we can find, we explain in the next section how we extended OWL with functional dependencies.

5. Extending OWL with FD

The formalism introduced in Section 4 is the basis for the integration of the extended FD approach into OWL. In this section we discuss such approach and use the SWRL language to express logic rules therein produced.

5.1. OWL FD Definition

In order to model the abstract FD definition presented in (16), an OWL Class FD has been specified. The FD Class, just like in the definition introduced at (16), has the following properties: `antecedent`, `consequent`, `definedFor` and `function`. The antecedent property links FD instances to one or more Path instances. A Path is a Class that contains a list of property references called PartList. The PartList class is a extension of the generic rdf:List. In order to make this a list of properties, the "first" property of this list can only accept rdf:Property instances. An example of a consequent property is shown below:

(22)

```
<owlfd:antecedent>
  <owlfd:path rdf:ID="path_1">
    <owlfd:part>
      <owlfd:partList>
        <rdf:first rdf:reference="#issuedForPerson"/>
        <rdf:rest>
          <owlfd:partList>
            <rdf:first rdf:reference="#worksFor"/>
            <rdf:rest>
              <owlfd:partList>
               <rdf:first rdf:reference="#basedInCountry"/>
               <rdf:rest rdf:resource="http://www.w3.org/
                 1999/02/22-rdf-syntax-ns#nil"/>
```

```
            </owlfd:partList>
          </rdf:rest>
        </owlfd:partList>
      </rdf:rest>
    </owlfd:partList>
  </owlfd:part>
</owlfd:path>
</owlfd:antecedent>
```

The Path instance contained in the antecedent specified in (22) specifies the following path in abstract notation:

$$path\_1 = \{issuedForPerson, worksFor, basedInCountry\} \quad (23)$$

The consequent property also links an FD instance to a Path, but to maximum 1. The path of the consequent also contains a list of rdf:Property references.
The definedFor property of the FD class corresponds to the root concept defined in (10). In case of being a FD key, we use the sub-property `keyDefinedFor`. The definedFor property links an FD instance with a reference to a Class, like in the following example:

```
<owlfd:definedFor rdf:resource="#Passport" />
```
(24)

Notice that the *Passport* class should have been previously defined somewhere in the ontology. Finally the *hasFunction* property indicates the resource id of the function corresponding to *f* in (10).
We can summarize the aforementioned characteristics of the FD class with its definition in OWL using a simplified syntax:

```
FD ⊆
owl:Thing
∀ antecedent only Path
≥ antecedent min 1
∀ consequent only Path
≤ consequent max 1
= definedFor exactly 1
= hasFunction exactly 1
```
(25)

The Path class and the PartList are defined as follows:

```
Path ⊆
owl:Thing
∃ part some partList
≥ part min 1

PartList ⊆
rdf:List
∀ rdf:first only rdf:Property
= rdf:first exactly 1
∀ rdf:rest only rdf:List
= rdf:rest exactly 1
```
(26)

In addition we have defined three subclasses of FD: $\mathcal{FD}a$, $\mathcal{FD}c$ and $\mathcal{FD}v$. These subclasses correspond to the abovementioned interpretation types: assertions, constraints and views respectively:

$$\mathcal{FD}a \subseteq \texttt{FD}$$
$$\mathcal{FD}c \subseteq \texttt{FD} \quad (27)$$
$$\mathcal{FD}v \subseteq \texttt{FD}$$

Another subclass of FD is $\mathcal{FD}k$, for the special case of key functional dependencies:

$$\mathcal{FD}k \subseteq \texttt{FD} \quad (28)$$

Once we have these classes and properties defined, we can reuse them to define FD's in any OWL ontology. For instance consider the Passport FD example in (7). We would instantiate our $\mathcal{FD}k$ class as follows:

```
<owlfd:FDk rdf:ID="fd_1">
    <owlfd:antecedent>
      <owlfd:Path rdf:ID="path_1">
        <owlfd:part>
          <owlfd:PartList>
```
(29)

```
            <rdf:first rdf:resource="#issuedForCountry" />
            <rdf:rest rdf:resource="http://www.w3.org/1999/02/
             22-rdf-syntax-ns#nil" />
          </owlfd:PartList>
        </owlfd:part>
      </owlfd:Path>
    </owlfd:antecedent>
    <owlfd:antecedent>
      <owlfd:Path rdf:ID="path_2">
        <owlfd:part>
          <owlfd:PartList>
            <rdf:first rdf:resource="#passportNumber" />
            <rdf:rest rdf:resource="http://www.w3.org/1999/02/
             22-rdf-syntax-ns#nil" />
          </owlfd:PartList>
        </owlfd:part>
      </owlfd:Path>
    </owlfd:antecedent>
    <owlfd:keyDefinedFor rdf:resource="#Passport" />
  </owlfd:FDk>
```

5.2. Mapping to SWRL

One we have defined the FD using our OWL FD classes, we can build Horn clause rules following the mapping schema as shown in (16). We used the Semantic Web Rule Language SWRL to express such rules. It is easy to see that a simple script program can easily generate the necessary SWRL rules based in a FD definition. For instance the SWRL rule counterpart to (29) would be:

(**30**)

```
<swrl:Imp rdf:ID="Rule_fd_2">
    <swrl:body rdf:parseType="Collection">
       <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate
                rdf:resource="#issuedForCountry"/>
          <swrl:argument1 rdf:resource="#p1"/>
          <swrl:argument2 rdf:resource="#c1"/>
       </swrl:IndividualPropertyAtom>
       <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate
                rdf:resource="#issuedForCountry"/>
          <swrl:argument1 rdf:resource="#p2"/>
          <swrl:argument2 rdf:resource="#c1"/>
       </swrl:IndividualPropertyAtom>
       <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate
                rdf:resource="#passportNumber"/>
                <swrl:argument1 rdf:resource="#p2"/>
                <swrl:argument2 rdf:resource="per1"/>
       </swrl:IndividualPropertyAtom>
       <swrl:IndividualPropertyAtom>
            <swrl:propertyPredicate
                  rdf:resource="#passportNumber"/>
            <swrl:argument1 rdf:resource="#p1"/>
            <swrl:argument2 rdf:resource="#per1"/>
       </swrl:IndividualPropertyAtom>
    </swrl:body>
    <swrl:head rdf:parseType="Collection">
          <swrl:IndividualPropertyAtom>
            <swrl:argument1 rdf:resource="#p1"/>
            <swrl:argument2 rdf:resource="#p2"/>
            <swrl:propertyPredicate rdf:resource="#sameAs"/>
          </swrl:IndividualPropertyAtom>
    </swrl:head>
  </swrl:Imp>
```

As explained above, the SWRL rule indicates that the `issuedForCountry` and `passportNumber` are the keys of the Passport class. In the figure below, dotted arrows mark the key properties.

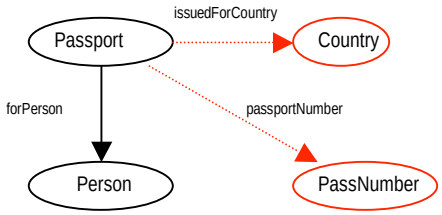

Fig. 2. $\mathcal{FDK}$ for the Passport Class.

Notice that, in this particular case, the FD is a key FD. If two passports have same instances for the properties `issuedForCountry` and `passportNumber`, then both Passports are the same. We can express this equality with the `owl:sameAs` property, and add it to the ontology.

Once the SWRL rules are generated, a SWRL reasoner can evaluate them and produce results. These results may eventually be added to the ABox or used for constraint checking. An OWL reasoner can then be applied to see if the ontology is consistent. If not, then some instances of the ABox do not comply with the FD's defined previously and they can be signaled in a consistency report.

6. Implementation

Having described our approach for adding functional dependencies to OWL, we proceed now to describe a prototype implementation demonstrating the applicability of our ideas. As it can be seen throughout this paper, we have worked with a simple ontology concerning flights. This ontology has been written in OWL. In the next subsections we will present the main components of our prototype and the main steps of the implementation workflow.

6.1. Building Blocks

In order to make possible an implementation of our constructs, we have put together several components and tools, which are detailed below. First we have added an abstract extension of FD's for OWL, which we call owlfd. Then we have used SWRL to define FDs in terms of horn clauses:

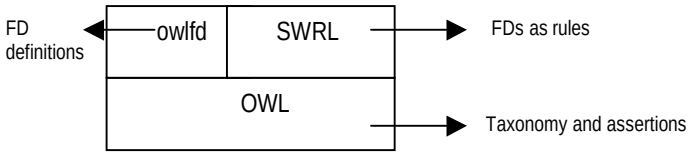

Fig. 3. OWL SWRL and FDs

Now to generate the SWRL rules based on the FD definitions, we have written a Java program that parses and interprets the FD's and returns the corresponding rules. The SWRL evaluation is performed by the Jess engine. For consistency checking we have used Pellet reasoner.

6.2. Implementation Workflow

The workflow of our implementation can be summarized in the following figure:

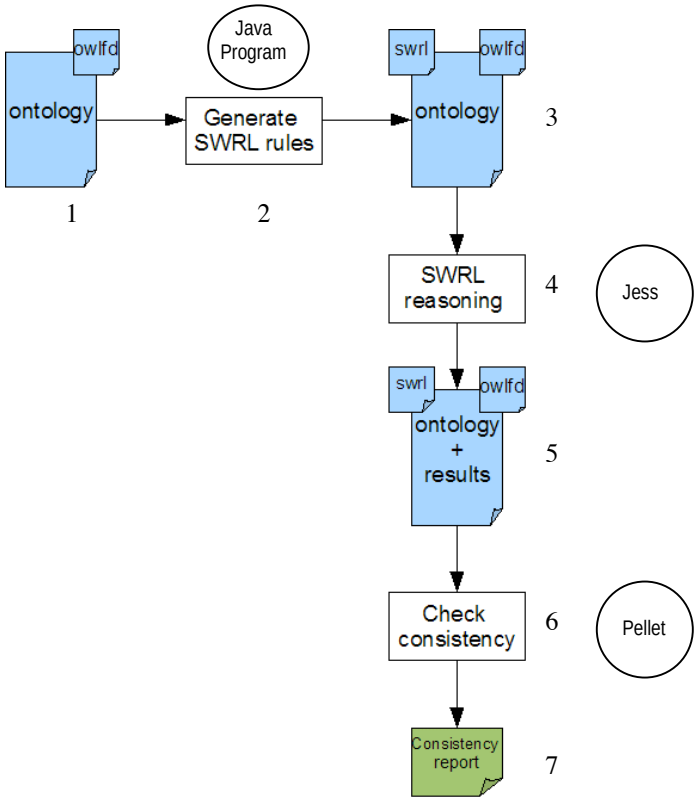

Fig. 4. Workflow of FD construction and evaluation with SWRL.

Using the abstract FD OWL extension described in the previous section, we have written FD definitions on the flight ontology (step 1 in fig. 4). These definitions define antecedents and consequents just like described in 16. Since our FD classes and properties are specified in OWL, it is possible to use a generic OWL editor like Protégé. In order to generate SWRL rules based on the FD definitions (step 2 in fig. 4); we have written a Java program that generates such rules following the mapping we established in section 4. It is important to notice that our implementation currently focuses on the identity function as the *f* component of a FD described in section 4.

Once the program has generated the SWRL rules, a SWRL reasoner can evaluate them (step 4 in fig. 4). We have used Jess with the JessTab extension of Protégé to evaluate the rules. Once the SWRL reasoner produces new assertions these are added to the ontology as new knowledge. Notice that it could also be possible to treat the FD as constraints and never add the results of the SWRL evaluation back to the theory.

At this point it is possible to use a standard OWL reasoner to check if the ontology is still consistent (step 6 of fig. 4). We used Pellet to do so. In case of finding inconsistencies, it could be due to several reasons. For example, the existent instances do not comply with the rules. But it could also be the case that the rules are wrong or badly defined. Since they are part of the knowledge base and their semantic interpretation depends on the domain and context, we can only let the user know that inconsistencies have been found. A deeper analysis should be performed by an expert in the domain of the concerned ontology.

7. Conclusions

The extension of DL knowledge based with functional dependencies has been acknowledged as relevant in producing more expressive ontologies. In this paper we investigate the extension of knowledge bases with three kinds of functional dependencies: classical, keys and explicit. In fact, to the best of our knowledge, this is the first work in ontologies that explores explicit functional dependencies, which are those where an explicit function relating dependent to determining properties is known. We propose a formal framework to extend ontologies with these three functional dependencies and study the different behaviors that can be considered when running FD rules. We identified three main interpretations: constraints, assertions and view and show how to integrate them within a common framework. The conceptual representation is implemented in OWL by a new FD

concept holding functional dependencies definitions. Moreover, a mapping function translates FD instance definitions into SWRL rules, allowing inferences to produce the desired FD behavior. The framework has been implemented in an initial prototype under Protegé and using Jess as the rule execution engine.

Currently, the mapping of explicit dependencies to SWRL only consider *identify* function. We intend as future work to explore the extension of SWRL with other builtin functions and the possibility of including user defined functions.

Our approach to extend the knowledge base with a new FD class has both advantages and disadvantages. The advantage is that it can be easily adopted without requiring any extension to the language. Furthermore, as the FD evaluation is done through SWRL it does not affect subsumption reasoning in the TBox. It turns out that this same aspect can be seen as a disadvantage as subsumption cannot be expressed over constrained concepts with FD.

Another interesting issue that we leave for future investigation is the case of key FD with multi-valued non-key attributes. In this scenario deciding on equality of sets seems not evident. Similarly, if properties in the head of a FD are allowed to be multi-valued then existential quantification over the set is required.

Finally, we intent to further explore the view interpretation, investigating how queries should be expressed and the possible effects of integrating view results with ABox assertions.

References

1. S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*, Addison Wesley, 1995.
2. Ronald Fagin. Horn Clauses and Database Dependencies. In *Journal of the Association for computing Machinery*, Vol 29, no 4, pages 952-985, 1982.
3. Boris Motik, Ian Horrocks, Ulrike Sattler. Bridging the Gap Between OWL and Relational Databases. In *Proceedings of the 16th international conference on World Wide Web*, pages 807-816, 2007.
4. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini. Keys for free in Description Logics. In *Proc. of the 2000 International Workshop on Description Logics (DL'2000)*, 2000.
5. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini. Identification Constraints and Functional Dependencies in Description Logics. In *Proc. of IJCAI 2001*, pages 155--160, 2001.
6. Carsten Lutz, Maja Milicic, Description Logics with Concrete Domains and Functional Dependencies. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, 2004.
7. David Toman, Grant E. Weddell, On Keys and Functional Dependencies as First-Class Citizens in Description Logics. On *IJCAR* 2006: 647-661, 2006.
8. S. Bechhofer, F. V. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P.F. Patel-Schneider, L. A. Stein. OWL Web Ontology Language Reference, http://www.w3.org/TR/owl-ref/, W3C Recommendation 10-02-2004.
9. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL – A Semantic Web Rule Langguage Combining OWL and RuleML, W3C member submission, 21th May 2004.
10. Alexander Borgida and Grant E. Weddell. Adding uniqueness constraints to description logics (preliminary report). In *Proceedings of the Fifth International Conference on Deductive and Object Oriented Databases*, pages 85--102, 1997..
11. B. Ludascher, Amarnath Gupta, M. E. Martone, Model-Based Mediation with Domain Maps, 17th International Conference on Data Engineering, Heidelberg, Germany, 2001.