Using OWL Design Patterns for Modeling and Reasoning with Organizational Policies

Mor Peleg[1] and Dizza Beimel[2]

[1]Department of Management Information Systems, University of Haifa, Israel, 31905, morpeleg@mis.hevra.haifa.ac.il
[2]Department of Industrial Engineering and Management, Ruppin Academic Center, Israel dizzab@ruppin.ac.il

Abstract. Organizations use policies to manage business cases effectively and efficiently. However, when they start defining their policies, they are often incomplete. In order to enact the incomplete set of policies they define a default policy that is used for cases that do not match the initial policy set. Since matching a business case to a set of policies is a classification problem, we suggest that the Web Ontology Language (OWL), a conceptual modeling language founded on ontological foundations, could be used to represent and reason with policies. However, default policies require closed-world reasoning, which is not easily supported in OWL. As a modeling approach, we defined a set of ontology design patterns (ODPs) that we created to support closed-world reasoning in OWL for policy-related classification problems. We distinguish among three sub-classes of this classification problem. To validate the ODPs, we represent the real-world examples using our ODPs and reason with them.

Keywords: OWL, conceptual modeling, closed-world reasoning, policy-based classification, Ontology Design Patterns.

1 Introduction

In the modern age, more and more organizations are defining their work practices. Clear definitions of business logic standardize and improve the quality in which business cases are handled, helping the organization's personnel operate effectively and efficiently, while reducing errors. To define business logic, organizations phrase *policies*, or rules, that define how business cases possessing certain characteristics should be handled. Because business cases are characterized by many variables that can each have a large range of values, the number of possible cases can be very large. In order to manage this large number of cases via policies, organizations select a manageable number of these variables, which we refer to as *context variables*, and based on them they define policies for *context classes*, i.e., classes of cases for which the range of possible values for the set of context variables is restricted. For example, in the medical insurance domain, patients who are in their first or second trimester of pregnancy and did not undergo genetic testing previously and are enrolled in a gold-level insurance plan are eligible to undergo a set of ten genetic tests. Another example, taken from the access control domain: family doctors may view the records of their patients only when they try to access the records from their work place.

While generating a set of policies, two contradicting requirements guide the policy creator: to cover all necessary context classes but at the same time to keep the collection of context classes to a minimum in order to support clarity and ease maintenance. An acceptable solution to these requirements follows the life-cycle of a developing prototype; an organization conceives an initial policy set that covers a basic set of context classes and specifies how they should be handled, such that the organization can start providing services before characterizing all necessary context classes. This set can be augmented later when the organization acquires more experience. In order to provide policies for all other real-life business cases that were not represented by the initial set, a *default policy* is defined by specifying a default way in which all cases not matching any context class should be handled. The use of a default policy is motivated by our aim to complete partial knowledge, inspired by Minsky [1].

In order to implement a mechanism that could suggest a policy-based recommendation for handling context classes, each context class in the initial set contains not only the definition of the context but also specifies the recommended way to handle such a context class. Based on the initial set of context classes and the default policy, a case under question is matched (exact match) to the defined context classes and the appropriate way of handling that case is inferred. If the case does not fall into one of the existing context classes then the default way of handling it is recommended (e.g., refer the case to the head administration person in the insurance company or, in the second example, deny access to patient data).

Matching a case in question into a set of possible context classes is a classification problem. Such problems are naturally represented using description-logics-based ontologies, such as the Web Ontology Language (OWL) [2]. OWL is a conceptual modeling language founded on ontological foundations, which is used to represent concepts in a domain, their properties, relationships, and restrictions. Inference (i.e., matching a case to a context class and from the context class implying the correct way to handle the case) can then be provided using OWL reasoners, such as Pellet [3]. However, as OWL uses open-world assumption, representing defaults and reasoning with them for real-world problems that require closed solutions is difficult. As a modeling approach, we defined a set of ontology design patterns (ODPs) that we created to support closed-world reasoning in OWL for the sake of policy-related classification problems. The policy-related classification problems that we address have multiple context classes that are handled via a small set of behaviors, which we refer to as *response type*. We distinguish among different sub-classes of this classification problem that depend on the number of possible responses that an organization can provide for a given case and on the number of possible response types from which the responses' values can be taken.

2 Background

In this section we review the OWL language, which we used to implement our policy-based classification of business cases, paying attention to methods for enabling closed-world reasoning in OWL. To formulate policy-based classification in OWL, we applied design principles for defining new ODPs that implement our solution. We review existing ODPs, some of which were used in our solution.

2.1 Web Ontology Language (OWL)

Web Ontology Language (OWL) [2] is a standard description-logics ontology language developed by the World Wide Web Consortium. OWL is based on a logical model which makes it possible for concepts to be defined, rather than just described. It has a rich set of operators, including intersection, union, and negation. Complex concepts can thus be defined in terms of simpler concepts. Furthermore, the logical model allows the use of a reasoner, which can check whether or not all of the statements and definitions in the ontology are mutually consistent, and can recognize which concepts fit under which definitions, forming classification hierarchies. An OWL ontology consists of Classes, Properties, Individuals (members of classes), and Restrictions. Restrictions (sometime referred to as Conditions) specify facts that must be satisfied by an individual for it to be a member of the class. The possible restriction types include: (a) the quantifier restrictions (i.e., all-values-from (∀) and some-values-from (∃)), (b) the has-value restriction, and (c) the cardinality restrictions (i.e., has-minimum-cardinality, has-maximum-cardinality, and has-exact-cardinality). We have used Protégé-OWL (or Protégé for short) [4] a free, open source ontology editor and knowledge-base framework, to define policies for context classes and used the Pellet [3] reasoner to reason with them (i.e., match a case to a context class).

2.2 Closed-world reasoning in OWL

Our policy-based classification mechanism is based on a *closed-world assumption* [3] that the information that we have about the case in question is complete. Under these circumstances, if a formula cannot be proved true or false in a knowledge base, it is considered to be false. Consequently, since we assume that all information about the individual is known then the class membership of the individual can be determined.

As OWL was developed for the purpose of reasoning over the semantic web, it follows an open-world approach. However, matching a case to a context class where a default response is mandatory requires closed-world reasoning. Theoretical work has been done on extending description logics to support closed-world reasoning in the form of non-monotonic logic [5, 6] or combinations of description logics with a logic programming approach with negation as failure [7, 8]. Since implementations of closed-world inference in OWL reasoners are in a preliminary state [9], we developed some practical solutions to "close" the ontology. Our solution involves a combination of ontology patterns that enable closed-world reasoning for policy-based classification using existing monotonic OWL reasoners.

2.3 Ontology design patterns

Ontology design patterns (ODPs) [10] are abstractions of workable solutions for modeling common problems in a knowledge domain. Noy et al. [11] discussed ODPs that they used to represent the National Cancer Institute’s Thesaurus in OWL DL. Their ODPs target inheriting possible information, exceptions, role chaining using Semantic Web Rule Language (SWRL), reciprocal restrictions, and defining classes by numeric ranges of properties. Stevens et al. [10] presented ODPs that they found useful for modeling biological knowledge. These include ODPs that provide workarounds for OWL limitations, specifically for modeling n-ary relationships, lists, and exceptions.

Rector [12] pointed out that many biomedical applications appear to require default reasoning, at least if they are to be engineered in a maintainable way. In default reasoning, plausible assumptions based on default rules are made when the information that we have about the world is uncertain. Rector analyzed the different uses for defaults and exceptions and distinguished among four types for which he proposed ODPs: (1) specialization that merely narrows, but does not contradict, the original statement. Specializations are represented routinely in OWL; (2) a single local exception: a statement that has been entered is in fact too general and more precise statements that make further distinctions are used to handle the local exception and the rest of the cases; (3) a modest number of dimensions of context, i.e., context variables that characterize the cases to be handled. By making context explicit and then generalizing common information where possible, the different context classes can be defined (Explicit Context ODP); (4) an unpredictable number of exceptions. For this case, hybrid reasoning about a knowledge base of contingent facts built around the core ontology is necessary.

3 Policy-based classification

As discussed in the Introduction, policy-based classification for an incomplete set of policy rules requires using defaults for those cases for which context classes were not explicitly defined. Although Rector [12] has characterized four types of defaults and exceptions, as summarized in Section 2.3, we propose another type that is not captured by Rector's work: (5) *Multiple Context Few Response* (*MCFR*) - many context classes but few *responses*, i.e., few ways in which the organization handles the context classes. In this problem type, the number of context variables is of medium size and the range of each context variable has a medium size number of possible values. Therefore, the number of possible context classes (i.e., meaningful combinations of allowed values for the different context variables) can become quite large. However, the number of ways in which these context classes should be handled is small. The purpose of using OWL is (1) to represent the various business cases along with their respected organizational responses via set of context classes, (2) to enable the inference of responses for cases in question based on the set of context classes, and (3) to specify a default response for all cases that do not match the set of context classes.

We characterized three sub-types of MCFR problems that are based on the number of possible response types (e.g., approve and deny are two possible response types) that define all possible behaviors for handling cases and on whether a single case could receive one or more responses whose values are taken from the set of response types. The three categories (C1-C3) of MCFR problems include:
C1 - **One** response for each case with **two** alternative response types (e.g., positive/negative), one of which is the default response. Multiple policy rules (that relate to different context variables) determine cases for which a positive response should be given; for other cases, a negative response should be provided.

An example of this subclass of MCFR problems is situation-based access control (SitBAC) [13] – a model that determines whether to grant or deny access to electronic health records based on characteristics of the situation in which access is requested. For example, while a physician usually does not have access to the laboratory tests of any patient when he is trying to access the data from his home, he may have home access to laboratory test data of patients for whom he is the family doctor.
C2 - **One** response for each case with **many** alternative response types and one default response.

An example of this subclass of MCFR is a system addressing student discipline problems. Depending on the characteristics of the discipline problem (e.g., student attacked a teacher, hit another student, brought a weapon to school, exhibited disrespectful behavior), different response types may be applicable. For example, a student can be dropped out, suspended, put on probation, or a letter to his parents may be sent as a warning. In case no context class has been specified for addressing the discipline case in question, the school principal needs to be contacted (default response).
C3 - **Many** responses possible for each case with **many** alternative response types and one default response.

An example of this class of MCFR is a system for determining the types of credit cards that a person is eligible for (e.g., regular card, student card, secure credit card, or no card), depending on factors that include, among others, his credit history and whether he is a student.

The case of multiple responses possible for each case with only two response types (one of which is the default response) is not possible because when there are only two response types there cannot be multiple responses per case without conflict.

4 Ontology design patterns (ODPs) for closed-world reasoning in MCFR problems

As noted, in order to infer a recommended response (handling) for every case in question, we need to "close" the domain. We used a set of ODPs to define a design solution for enabling closed-world reasoning for MCFR problems using OWL. The three types of MCFRs use a very similar design with slight changes that we discuss later in this section. We start by describing the main part of the design which is used in all three types of MCFRs.

We defined an ontology that includes a class hierarchy of policy rules (via context classes) that determine the responses for MCFR problems. Under this ontology, a case in question, for which a response needs to be made, is represented as an individual that is realized by an OWL reasoner into one (or more) context classes from which the response(s) are inferred. To construct the context classes, we developed an ODP that we call *Ocean Surrounding Islands*, which we discuss in subsection 4.1. In Section 4.2 we discuss its inference mechanism. This ODP depends on closed-world reasoning that is achieved by closing class hierarchies and individuals, as we explain in subsections 4.3 and 4.4, respectively. The design variations used to model the three types of MCFR problems are discussed in Section 4.5.

4.1. An ODP for MCFR problems (Ocean Surrounding Islands)

Our MCFR ODP can be envisioned as an ocean surrounding islands (Fig. 1). The islands represent the context classes (the policies). Each context class defines the allowed values for context variables and specifies the response type. The ocean represents the default response type that should be inferred if no matching island is found.

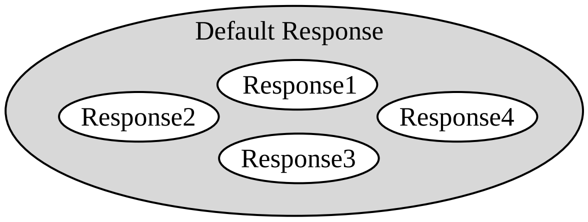
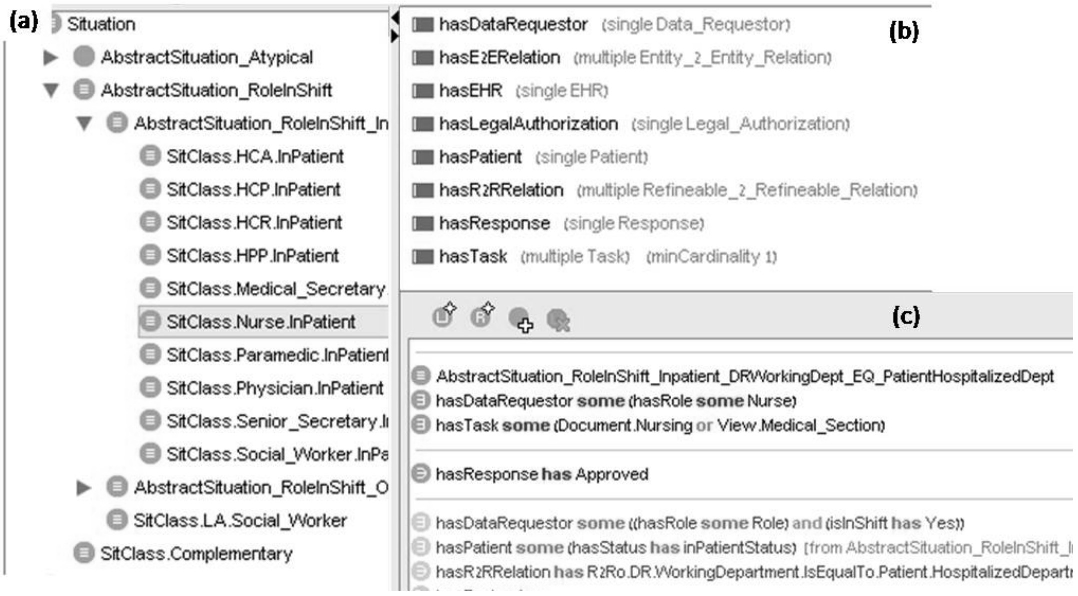


Fig. 1. The ocean surrounding islands ODP. Each policy is represented as an island class from which a response can be inferred. The ocean is the complement of all island classes. From it, a default response is inferred.

The MCFR ODP is realized via the *explicit context* ODP [12], discussed in Section 2.3. In the case of MCFR problems, each policy corresponds to a context class for which a response needs to be made. The dimensions of context are the context variables. For example, for cases of patient data access requests, the dimensions include properties of (1) the data requestor (e.g., whether he is working in the current shift), (2) the patient whose data is requested (e.g., his location), (3) the type of data requested (e.g., laboratory data), and (4) relationships between the data requestor and patient (e.g., the data requestor is the family doctor of the patient), as shown in Fig. 2 (b). Part (a) of Fig. 2 shows the hierarchy of context classes from the domain of access control to electronic health records. Each context class (called `Situation` Class, or *SitClass* for short) represents a situation of data access request for which, according to organizational policies, access should be approved. For example, the class `SitClass.Nurse.InPatient` corresponds to a context class that defines the policy for a situation where the nurse on duty wishes to document the nursing section or view the medical section of the electronic health record (EHR) for a patient hospitalized in her department. For this situation, the specified response that can be inferred is `Approved`. The definition of this context class is shown in Fig. 2 (c).

To represent classes with context, we adapt the Explicit Context ODP described in Section 2.3. According to the explicit context ODP, if X (e.g., `Situation`) is a class for which different context classes exist, then these context classes are modeled as disjoint subclasses of X. We state that all individuals who are members of X must be members of one of X's subclasses by use of a covering axiom. The conditions that define the context are not represented in the class definition of X, but are pushed down into X's subclasses. In contrast to Rector's ODP, we make two changes. First, the context classes do not have to be disjoint; they are disjoint only for one of the three multiple-criteria decision problems (i.e., C2: one response for each individual with many alternative response types and one default response). Second, in addition to the subclasses that define different context classes, we add an "else", or *complementary context*. This complementary context is defined as another subclass of `Situation` (`SitClass.Complementary`) shown on the bottom of Fig. 2 (a) and is the complement (i.e., negation) of all other context subclasses. For example, the complement of the Situation context subclasses is defined as: "`Situation and not SitClass.HCA. InPatient and not SitClass.HCP.InPatient and not SitClass.HCR. InPatient and not SitClass.HPP.InPatient and not SitClass. Medical_Secretary.InPatient and not SitClass.Nurse.InPatient and not SitClass.Paramedic.InPatient and not SitClass.Physician .InPatient and not SitClass.Senior_Secretary.InPatient and not SitClass.Social_Worker.InPatient and not SitClass.LA.Social _Worker`". For this complementary situation, the response is `Denied`. Note that the complementary context class acts as an alternative to the coverage axiom of the most generic `Situation` class.



Situation
AbstractSituation_Atypical
AbstractSituation_RoleInShift
AbstractSituation_RoleInShift_In
SitClass.HCA.InPatient
SitClass.HCP.InPatient
SitClass.HCR.InPatient
SitClass.HPP.InPatient
SitClass.Medical_Secretary
SitClass.Nurse.InPatient
SitClass.Paramedic.InPatient
SitClass.Physician.InPatient
SitClass.Senior_Secretary.I
SitClass.Social_Worker.InPa
AbstractSituation_RoleInShift_O
SitClass.LA.Social_Worker
SitClass.Complementary

hasDataRequestor (single Data_Requestor)
hasE2ERelation (multiple Entity_2_Entity_Relation)
hasEHR (single EHR)
hasLegalAuthorization (single Legal_Authorization)
hasPatient (single Patient)
hasR2RRelation (multiple Refineable_2_Refineable_Relation)
hasResponse (single Response)
hasTask (multiple Task) (minCardinality 1)

AbstractSituation_RoleInShift_Inpatient_DRWorkingDept_EQ_PatientHospitalizedDept
hasDataRequestor some (hasRole some Nurse)
hasTask some (Document.Nursing or View.Medical_Section)

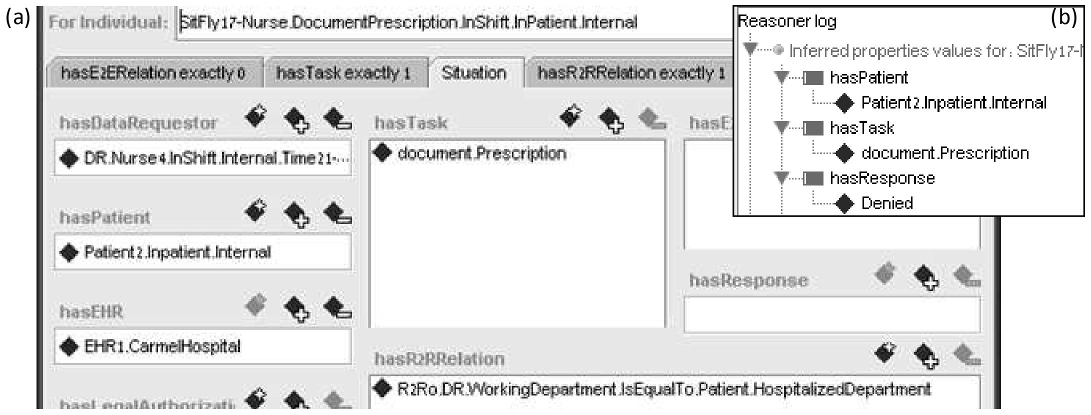
hasResponse has Approved

hasDataRequestor some ((hasRole some Role) and (isInShift has Yes))
hasPatient some (hasStatus has inPatientStatus) [from AbstractSituation_RoleInShift_I
hasR2RRelation has R2Ro.DR.WorkingDepartment.IsEqualTo.Patient.HospitalizedDepart

Fig. 2. Implementation of the MCFC ODP in Protégé for access control to electronic health records. (a) Part of the context class hierarchy. The root class *Situation* corresponds to data access requests for which a response should be made. The highlighted class `SitClass.Nurse.InPatient` and the other Situation classes (sitClasses) correspond to some of the organizational policies for granting access to a data access request cases. The class at the bottom of the figure is the complementary class, corresponding to the "ocean" in the ODP. The definition of the `SitClass.Nurse.InPatient` class is shown in part c. (b) The properties of Situation classes. (c) The definition of the `SitClass.Nurse.InPatient` subclasses, including necessary and sufficient conditions (top), necessary conditions (middle), and inherited conditions (bottom). The screenshot was taken using Protégé version 3.4.

4.2. Using the ocean surrounding island ODP to infer a response

A case in question, for which a response needs to be made, is represented as an individual of the most generic context class (e.g., `Situation` for the access control example and `CreditCardRequest` for the credit card example). For each case individual, the appropriate response type needs to be inferred. To do so, each individual is realized into its most specific context subclass, based on the necessary and sufficient axioms of the context subclass. In addition, each context subclass has a necessary axiom that defines the implied value of the response property. Such axioms are known as necessary implications [14] (about property values). In the access control example, all the context subclasses have an implied response of `Approved` (see Fig. 2 (c)), except for the complementary context subclass, which has a response of `Denied`. Fig. 3 shows an example of a situation individual realized to belong to the complementary subclass. While this individual was not asserted with a value for the `hasResponse` property (see part (a) of the figure) the value `Denied` was inferred for it (see part (b) of the figure) after it was realized as a member of the complementary subclass.


(a)
For Individual: SitFly17-Nurse.DocumentPrescription.InShift.InPatient.Internal

hasE2ERelation exactly 0 | hasTask exactly 1 | Situation | hasR2RRelation exactly 1

hasDataRequestor: DR.Nurse4.InShift.Internal.Time21-...
hasPatient: Patient2.Inpatient.Internal
hasEHR: EHR1.CarmelHospital
hasLegalAuthorizati
hasTask: document.Prescription
hasResponse:
hasR2RRelation: R2Ro.DR.WorkingDepartment.IsEqualTo.Patient.HospitalizedDepartment

(b)
Reasoner log
Inferred properties values for : SitFly17-
hasPatient
Patient2.Inpatient.Internal
hasTask
document.Prescription
hasResponse
Denied

Fig. 3. (a) A situation case individual "closed" for multi-cardinality properties (`hasR2RRelation`, `hasE2E Relation`, and `hasTask`). Closing axioms are shown in grey, on the top. Note that no value has been asserted for the `hasResponse` property; (b) Inferred response. The screenshot was taken using Protégé version 3.4.

4.3. An ODP for closing class hierarchies

In the open-world reasoning approach supported by OWL, an individual cannot be inferred to be a member of the complementary context class unless the reasoner can establish that this individual cannot possibly be a member of the other context classes. In order for individuals to be realized into the complementary class, we need to "close" classes and case individuals, such that for any pair of individual and class, we can infer whether the individual is a member of that class or not. To do so, we need to make sure that (1) all possible individuals could be realized as members of existing classes and (2) we know not just the information asserted about an individual but also that no other information about the individual can be true. In this way we can decide for each pair of individual and class whether the individual is a member of that class. In this subsection and in the next subsection we provide the ODPs used to close class hierarchies and individuals.

Classes in an ontology are often arranged in hierarchies. For example, the residential status of USA residents includes visitors with different types of visas, residents, and citizens. Similarly, the hierarchy of roles in a healthcare organization includes several contexts, including healthcare professionals, healthcare administrators, and healthcare researchers. Healthcare professionals include paramedics and physicians, and paramedics are further specialized into nurses, nutritionists, and ordinary paramedics. "Closing" classes means that we restrict the range of possible values of the class to an exhaustive list. This can be done using the value partition [15] ODP. In a value partition, we create a class X that represents some class in our model (e.g., `Paramedic`), define its possible values as disjoint subclasses, and add a covering axiom for X (e.g., `Paramedic` is covered by its disjoint subclasses `Nurse` and `Nutritionist`). This is done recursively for all levels of the class hierarchy.

However, there may be cases where there exist individuals belonging to the generalization class (e.g., `Paramedic`) in addition to individuals that are members of more specific subclasses. These cases would violate the coverage axiom. To solve this problem, we add a complementary ("ordinary") subclass to represent the context of an individual belonging to the generalization class (e.g., `Ordinary_Paramedic`) and change the covering axiom so that the ordinary subclass is part of the covering axiom. This preserves the covering axiom and ensures that an individual can be realized into only one of the context classes. Thus, "ordinary" individuals have their own context subclass, defined to be the negation of all possible subclasses of the generalization class (see Fig. 4).

Some concepts in the ontology are not hierarchically organized but instead form enumerations of possible alternative states. For example, class `Response` (i.e., the response to a credit card request) is defined as a set of individuals `{ApproveSecuredCreditCard, ApproveStudentCreditCard, ApproveRegularCreditCard, Denied}`.

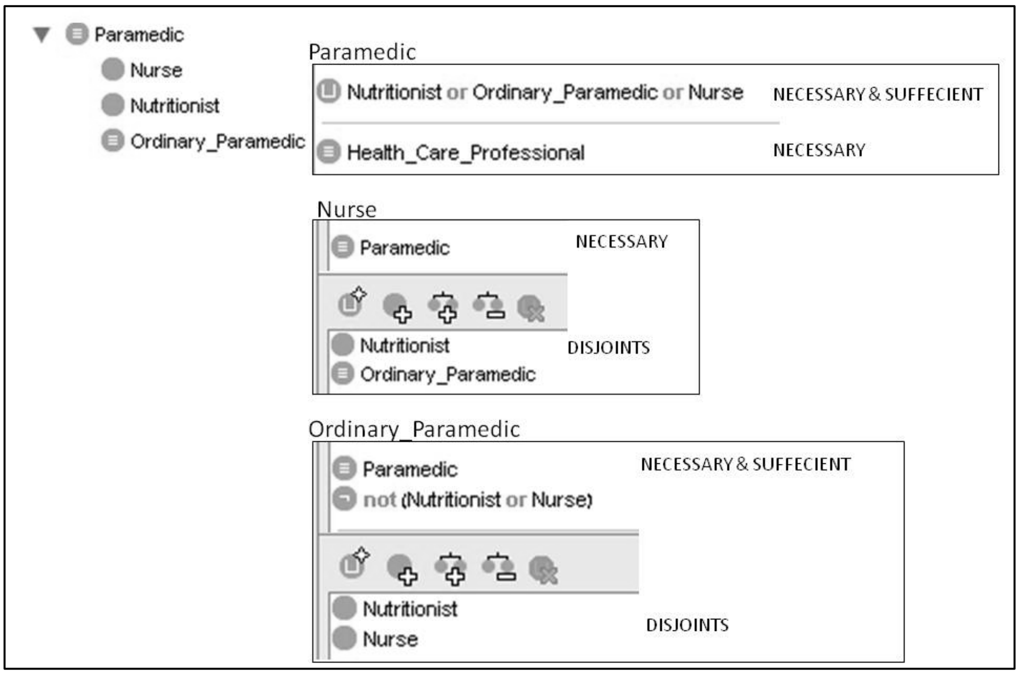


Fig. 4. A closed class hierarchy. The Paramedic class (a subclass of `Health_Care_Professional`) is covered by the subclasses Nutritionist, `Ordinary_Paramedic`, or `Nurse`. `Nurse` is disjoint from `Nutritionist` and `Ordi-nary_Paramedic` (likewise Nutritionist is disjoint from `Nurse` and `Ordinary_Paramedic` – not shown in the figure). `Ordinary_Paramedic` is defined as a `Paramedic` that is not a `Nutritionist` or `Nurse`. The screenshot was taken using Protégé version 3.4.

4.4. An ODP for closing individuals

To enable closed-world reasoning, we developed the following set of rules for closing individuals.
(1) All properties should have values.
(2) When values are known not to exist, an axiom should be used to state that the property has exactly 0 Things (see Fig. 3).
(3) In cases where a class has a property associated with a multiple cardinality restriction, we need to state the exact cardinality of an individual belonging to that class (see Fig. 3).
(4) We also need to close multi-cardinality properties in entities appearing as the range of single-cardinality slots (i.e., closing multi-cardinality slots of `Patient` and `Data_Requestor`).

Closing each and every multi-cardinality property for every individual is very tedious. To simplify this task, we structure all possible relationships between classes as class hierarchies. This allows us to create generic relationship properties (e.g., `hasRelation`, e.g., `hasR2RRelation`, `hasE2ERelation` in Fig. 3) that will accept any relationship as the range. In this way, while the number of possible relationships is very large, only one relationship property needs to be closed.

4.5. ODPs supporting the three types of MCFR problems

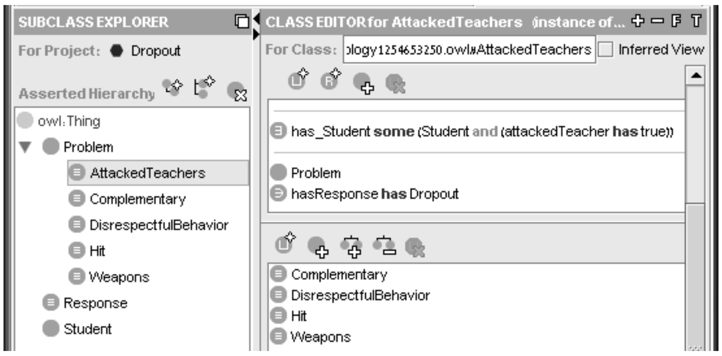
The three types of MCFR problems can be represented using variants of the *Ocean surrounding Islands* ODP. In all three MCFRs, there is a default response type that could be inferred only from the complementary context class. Since our MCFRs require non conflicting responses, we extended the Ocean surrounding Islands ODP to support the variations among the three MCFRs. Practically, the differences between the three MCFRs lie in whether context subclasses are disjoint and in the number of response types that are defined.

4.5.1. One response for each individual with two alternative response types

This MCFR was exemplified in earlier subsections of Section 4. In this MCFR, there are only two response types: `approved` and `denied`, where denied is the default response. Each context class represents rules for inferring the approved response. The complementary context class defines all the complementary cases for which a denied response is inferred. Context classes other than the complementary class do not have to be disjoint because there is only one response type for all positive responses.

4.5.2. One response for each individual with many alternative response types and one default response

Unlike the two other MCFRs, in this MCFR the classes need to be disjoint. This is because only one response should be given for each case for which a response should be given and at the same time many alternative response types are possible. Therefore, a conflict between response types may result unless the context subclasses are disjoint. In such a solution, a case in question may be realized into only a single context class, resulting in a single non-conflicting response. Fig. 5 shows an example of the definition of disjoint context subclasses for the student discipline problems case study.



SUBCLASS EXPLORER
For Project: ● Dropout
Asserted Hierarchy
owl:Thing
Problem
AttackedTeachers
Complementary
DisrespectfulBehavior
Hit
Weapons
Response
Student

CLASS EDITOR for AttackedTeachers (instance of...
For Class: ology1254653250.owl#AttackedTeachers
Inferred View
has_Student **some** (Student **and** (attackedTeacher **has** true))
Problem
hasResponse **has** Dropout
Complementary
DisrespectfulBehavior
Hit
Weapons

Fig. 5. An example of the definition of disjoint context subclasses for the student discipline problems case study. The context subclasses for this domain include the following types of discipline problems: student attacked a teacher, exhibited disrespectful behavior, Hit (another student), or brought weapons to school, and a complementary case, not falling into the previous categories. All subclasses are disjoint, assuming that a single problem may be realized into only one of these classes. The screenshot was taken using Protégé version 3.4.

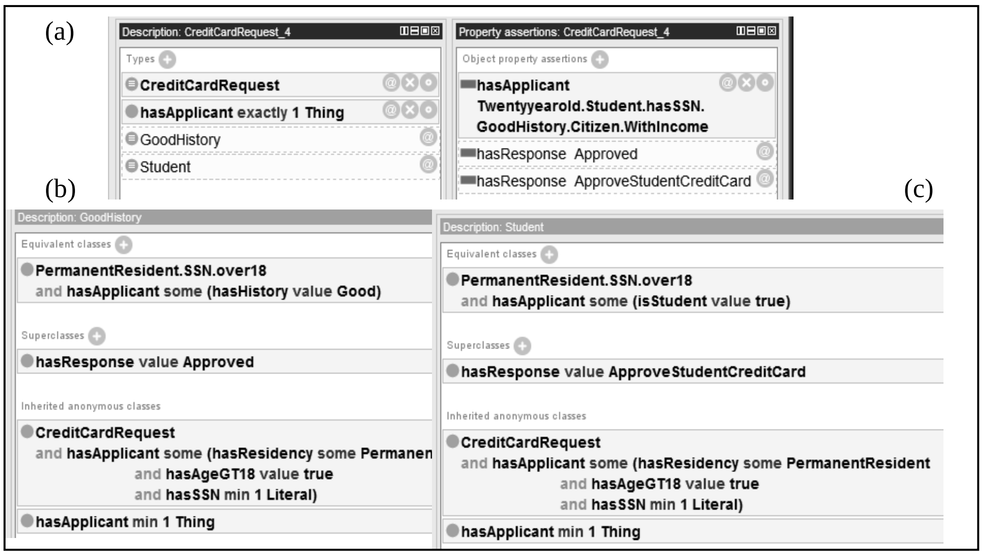
4.5.3. Many responses possible for each individual with many alternative response types and one default response

In this MCFR, a case in question may be realized into several context classes, which each may have a different response type. Classes are obviously not disjoint, several response types are defined, and the response property is not functional (i.e., has multiple cardinality). Fig. 6 shows an example of an individual realized into more than one context subclass.

5. Discussion

MCFR problems occur often in real-world problems. We characterized three types of MCFR problems and developed a model for their representation and reasoning using a set of OWL ODPs. Used together, the ODPs can support policy-based reasoning that results in recommending a response for a business case in question. We demonstrated the use of the ODP-based solutions for three case studies that involve policy-based classification in three domains: access control, student discipline problems, and credit card issuing. The OWL access-control ontology [13] is available at the National Center for Biomedical Ontology at the following URL: http://bioportal.bioontology.org/visualize/39798. All the case study ontologies are available at http://mis.hevra.haifa.ac.il/~morpeleg/Policy.htm.

Our ODP-based construction of the ontology results in a structured and uniform ontology that eases extending and maintaining the ontology. Moreover, our design assures completeness, no conflicts, and minimality, as explained below. An incoming case of a MCFR problem will always have a response (**completeness**). This is achieved because a case individual whose property values are all "closed" can be determined to be a member or a non-member of each defined context subclass in the ontology; if it is not a member of any of the context subclasses, then, by definition, it is a member of the complementary context class. Once it is realized, its response would be implied by the necessary implication. However, there could be cases in question where a case request is submitted with some property values not known (e.g., the patient's status or the data requestor's shift status may be unspecified). In those cases, the case individual might not be realized to any context class or to the complementary class. The policy for those cases is to provide the default response. Logging such cases would allow database managers (or privacy officers in the case of access control problems) to supply the missing information so that the individual could be realized. To help in diagnosis of such cases, explanation facilities could be developed to indicate which property values were not closed and for which context subclasses membership of the case individual could not be inferred.



Description: CreditCardRequest_4

Types

CreditCardRequest

hasApplicant exactly 1 Thing

GoodHistory

Student

Property assertions: CreditCardRequest_4

Object property assertions

hasApplicant Twentyyearold.Student.hasSSN.GoodHistory.Citizen.WithIncome

hasResponse Approved

hasResponse ApproveStudentCreditCard

Description: GoodHistory

Equivalent classes

PermanentResident.SSN.over18 and hasApplicant some (hasHistory value Good)

Superclasses

hasResponse value Approved

Inherited anonymous classes

CreditCardRequest and hasApplicant some (hasResidency some Permanen and hasAgeGT18 value true and hasSSN min 1 Literal)

hasApplicant min 1 Thing

Description: Student

Equivalent classes

PermanentResident.SSN.over18 and hasApplicant some (isStudent value true)

Superclasses

hasResponse value ApproveStudentCreditCard

Inherited anonymous classes

CreditCardRequest and hasApplicant some (hasResidency some PermanentResident and hasAgeGT18 value true and hasSSN min 1 Literal)

hasApplicant min 1 Thing

Fig. 6. Realization of a credit card request into two types of positive response types (`Approved` - a regular card and `ApproveStudentCreditCard`). As shown on the right hand side of the credit card request individual (a) the applicant of the request is a 20-year old student with a social security number (SSN), good credit history and has an income. Following the rules defined by the context classes for GoodHistory (b), and Student (c) the request (a) is realized into both of these context classes and draws their inferred responses. The screenshot was taken using Protégé version 4.0.

To enable unambiguous responses with **no conflicts**, we maintain a **minimal** knowledge base that does not contain unnecessary specialized context subclasses whose knowledge is already contained in more generic classes. For example, in the access control example a situation (context) subclass that permits a nurse to access the demographics section of the EHR for patients hospitalized in the emergency department is redundant if another situation subclass permits a nurse to access the entire EHR (i.e., with the property defining section left empty) for such patients. To discover redundant classes, we use a reasoner to classify the ontology's classes and check for new inferred subclass relationships between context subclasses. If we find new subclass relations, we can examine them and delete the unnecessary class. The only exception that we allow is an unnecessary super class that is used to generalize axioms that are reused in several case classes, which eases maintenance of the knowledge base. These super classes do not contain necessary implications about the response. We thus achieve a knowledge base where each individual can be realized into at most one concrete context class, ensuring no conflicts in responses.

Other works [16-18] have also used OWL-based representation and a DL-reasoner to handle policies. However, our work is different as we use the reasoner in real time in order to make a decision concerning a case in question, while others use the reasoner mainly to maintain consistent policies. In addition, we provide a set of ODPs to "close" the domain, thus, we always provide a response.

MCFR problems are very appropriate for classification and realization-based reasoning mechanisms, which are well-supported by OWL. However, since MCFR problems require closed-world reasoning, we had to find a solution for enabling closed-world reasoning in OWL. Our solution works with existing reasoners that support monotonic open-world reasoning. However, the complexity of the reasoning may be exponential. For example, the complexity of the student dropout example is ALCOF(D) (PSpace) and the complexity of the SitBAC and the credit card ontologies is equivalent to ALCON(D), making it PSpace-complete. However, polynomial time algorithms have been developed for sub-ALC logics [19]. Therefore, we believe that if complexity becomes an issue for scalability we could develop a special-purpose reasoner with polynomial complexity.

Acknowledgments. We thank Timothy Redmond, Samson Tu, and Martin O'Connor for the useful discussions.

References

1. Minsky M. L.: A framework for representing knowledge. In: Winston P. H. (ed). The Psychology of Computer Vision. pp. 211--277. McGraw Hill, New York (1975)
2. WebOnt Working Group. Web Ontology Language (OWL), http://www.w3.org/2001 /sw/WebOnt/
3. Sirin E., Parsia B., Grau B. C., Kalyanpur A., Katz Y.: Pellet: A Practical OWL-DL Reasoner. J. Web Semantics 5(2), 51--53 (2005)
4. Knublauch H., Fergerson R., Noy N. F., Musen M.A.: The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications. In: Third International Semantic Web Conference; Hiroshima, Japan (2004)
5. Grimm S., Hitzler P.: Semantic Matchmaking of Web Resources with Local Closed-World Reasoning. Intl. J. of e-Commerce 12(2), 89--126 (2008)
6. Katz Y., Parsia B.: Towards a nonmonotonic extension to OWL. In: Proc. Workshop on OWL Experiences and Directions; Galway, Ireland (2005)
7. Rosati R.: On the decidability and complexity of integrating ontologies and rules. Journal of Web Semantics 3(1), 61--73 (2005)
8. Eiter T., Lukasiewicz T., Schindlauer R., Tompits H.: Combining answer set programming with description logics for the semantic web. In: Proc. Ninth Intl Conf Principles of Knowledge Representation and Reasoning (2004)
9. Clark & Parsia LLC.: Does Pellet support closed world reasoning?, http://clarkparsia.com/pellet/faq/closed-world/
10. Stevens R., Aranguren M. E., Sattler U., Drummond N., Horridge M., Rector A.: Using OWL to model biological knowledge. Intl J Human-Computer Studies 65(7), 583--594 (2007)
11. Noy N. F., de-Coronado S., Solbrig H., Fragoso G., Hartel F. W., Musen M. A.: Representing the NCI Thesaurus in OWL DL: Modeling tools help modeling languages. Applied Ontology J. 3(3), 173--190 (2008)
12. Rector A.: Defaults, context, and knowledge: alternatives for OWL-indexed knowledge bases. In: Pac. Symp. Biocomput. pp. 226--37 (2004)
13. Peleg M., Beimel D., Dori D., Denekamp Y.: Situation-Based Access Control: privacy management via modeling of patient data access scenarios. J. Biomed. Inform. 41(6), pp.1028--1040 (2008)
14. Rector A., Drummond N., Horridge M., Rogers J., Knublauch H., Stevens R., et al.: OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors and Common Patterns. In: Proc. European Conference on Knowledge Acquistion. Springer-Verlag. Northampton, England. pp. 63--81 (2004)
15. Horridge M., Knublauch H., Rector A., Stevens R., Wroe C.: A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools, Ed. 1.0. University of Manchester, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf
16. Finin T., Joshi A., Kagal L., Niu J., Sandhu R., Winsborough W. H, et al.: ROWLBAC - Representing Role Based Access Control in OWL. In: Proc. 13th Symposium on Access control Models and Technologies, pp. 73--82 (2008)
17. Bradshaw J., Uszok A., Jeffers R., Suri N., Hayes P., Burstein M., et al.: Representation and Reasoning for DAML-Based Policy and Domain Services in KAoS and Nomads. In: Intl. Conf. Autonomous Agents, pp. 835--842 (2003)
18. Kagal L., Berners-Lee T., Hendler J.: The Semantic Web and Policy. J. Web Sem. 7(1), pp.1-56 (2009)
19. Baader F., Lutz C., Suntisrivaraporn B.: CEL - A Polynomial-time Reasoner for Life Science Ontologies. Proc. 3rd Intl. Joint Conf. on Automated Reasoning, LNAI, vol. 4130, pp. 278--291. Springer-Verlag (2006)