Deriving User Interface from Ontologies: A Model-based Approach

Ben Liu, Hejie Chen, Wei He
Department of Computer Science, Beijing Institute of Printing
{bliu, hewei}@bigc.edu.cn

Abstract

In this paper we present an approach to derive User Interface (UI) from ontologies. It automatically generates UI according to declarative model specifications, and the UI helps a novice user to construct valid queries against a Knowledge Base (KB). The UI represents queries by OWL, and then transfers them to concrete applications for matchmaking. A novel aspect of our approach is that the ontological KB used by UI generator is exactly the one used by applications; hence the refining of ontologies needs not refining UI. Further, during a user-machine interaction session, the system eliminates every illegal option that engenders a clash in the KB, so the consistent checking of user inputs is accomplished as a side effect. It boosts the development of a knowledge dense system, as the building of a KB in progress, the UI is generated simultaneous.

1. Introduction

The ongoing semantic web envisions a better platform for human-machine and machine-machine interaction; to achieve this we need ontology as the Rosetta stone. With the emerging of web ontology language – OWL [1], researchers from the world have built a large KB that consists of ontologies (cf. http://www.daml.org/ontologies). There are also tools for authoring and browsing an ontology like protégé [2], OilEd [4], etc, and methods for ontology aligning and merging [6, 7]. But manually exploring such a huge KB is an awful work, even with some mechanisms of registration and retrieving [8], it is hardly for a novice user to construct a valid query. Hence, a flexible and effective method to explore the web ontologies is crucial; especially the need of a mechanism to extract a fragment of ontology.

Model-based interface generates UI automatically from a declarative Model and has been a paradigm of constructing UI [9]. The potential benefits of the paradigm are summarized as follows [10]: a powerful runtime tools for generation of designed-time and end-user interface, consistency and reusability that we have realized in our works, support for early conceptual design and iterative development. We exploited the model-based approach to derive UI from web ontologies according to a custom model ontology that specified which fragments of ontologies are needed. Besides the benefits of general model-based approach, our framework has following features.

• A mechanism enables designers to specify which fragment of ontology is extracted and how it is rendered. The UI automatically rendered according to the specification will guide users to explore a KB and generate queries expressed by the Description Logics (DL) [12].

• The ontological KB used by UI generating is exactly the one used by information processing; hence the evolution of KB does not result in the re-engineering of UI.

• During a process of interaction, any illegal option that engenders a clash in a DL reasoner has been removed, and the validity of user inputs is guaranteed.

• It boosts the development of a knowledge dense system, as the progress in building a KB, the UI is generated simultaneous. Furthermore, the UI realizes the consistent check without extra code; the option filter process shares most routines with the real works.

• The model specification language is OWL-DL.

The remainder of this paper is structured as follows. Next we briefly review related works and discuss their relationships with ours. In section 3, we introduce backgrounds of our works and motivations; to make further discussion clearly we also introduce elementary description logics and notations here. Next sections describe our framework, the model specification language, and its use cases. Before we close with conclusions, we analyze aspects of our approach, including performance, usability, and improvements.

2. Related work
The cornerstone of a model-based interface system is declarative models abstracting relative aspects of UI. Then the systems automatically generate UI according to respective declarative model specifications, and provide power tools for UI design [9, 10, 14]. With the emerging of mobile computing, there are works focus on the automatic generation of platform neutral UI, e.g., PDAs [15, 16]. Our works share the concept of model with these works, but have two substantial differences. Firstly we exploit a formal logics base to describe the model, more specifically, describe the model by OWL-DL, while above works use specific model language respectively. Secondly the validation of inputs is one of our premier objectives and we solve it by reasoning on the KB, and the others omit it. Clearly our works and above complement each other. We argue that a common model specification language would greatly benefit UI design and generation.

E. Furtado, et al. [17] suggest a triple-layer UI design method, a semi-frame (e.g., slot, facet,) based ontology models the top level – concept level, a logic level describing models, and the physical level rendering models. The authors argue that applying separate concerns facilitates UI design and generation. Our works differ from them in the way that we exploit formal logics to describe domain ontology, and the UI is responsible for the validation of user inputs. If a combination of concepts is incompatible with respect to ontology and without any means to detect, it is possible to generate unreasonable options – we give an example in section 3.2.

The most similar works to ours is [3], our works share with them on using a DL-based KB and guiding users to choose “right” options. These two systems differ on aspects of model and target applications. The former uses a layer separating underline DL with applications to help a user construct reasonable queries; while ours exploits a more structured and expressive model and targets a general UI.

3. Background

The primary motivation of our works is to construct a vertical application of the semantic web, since the works have been published elsewhere [11], we briefly describe the system here.

3.1. The application

We have built an intelligent query system in a B2C scenario of flowers markets, the system can answer questions as under certain conditions (we name it a situation), e.g. intention, festival, recipient, etc, which flowers are suitable as a present. To achieve this, we have constructed a KB that consists of domain ontology. During the process of an interaction between a user and the system, the system constructs queries expressed by the DL according to the user’s actions, then matches them against the KB and finds answers. We rewrote all rules of flower-present and expressed them as DL concepts described by OWL class; hence the matching process is to find the least super concept that contains the query, and the matching is done via the RACER (cf. http://www.cs.concordia.ca/~haarslev/racer/) engine.

3.2. Motivation

During the interaction, the UI provides a user with options and by selecting these options the user tells the system what situation it is. If the UI rendered all concepts and their properties without any means of checking, the combo of options would look unreasonable. For example, there is a situation that a recipient is a pregnant woman and some flowers are candidates. If a user told the system that the recipient is a woman, the Pregnancy should be an option. But for a male recipient, the Pregnancy option is ridiculous according to nowadays technologies. We have encoded those rules in our knowledge, e.g., equation 3.3 states that only AdultFemalePerson may be Pregnancy. A consequence of E.3.1-3.3 is that a man has not the property of Pregnancy.

\[\text{MalePerson} \cap \text{FemalePerson} \equiv \bot \quad \text{E3.1}\]
\[\text{AdultFemalePerson} \subseteq \text{FemalePerson} \quad \text{E3.2}\]
\[\exists \text{has Pregnancy}.T \subseteq \text{AdultFemalePerson} \quad \text{E3.3}\]

Hardwiring these rules in UI has two major drawbacks. First, to ensure the consistent, refining the KB has to refine the UI rules, worse yet, not all clashes are evident - some of them are the consequences of a complex reasoning process. Second, keeping above constraints while mapping DL concepts to UI titles is troublesome, and without carefully factoring, it will lead to much repetitious code and is error prone.

Inspired by the literature, we defined a model specification language expressed by OWL. We use the language to declare which part of the KB is to be extracted and how it is rendered to the interface. A UI generating module interprets the model specification, extracts terms according to the model, and checks the consistent of a term against the current situation. If the term passed the verification, the module produces an option and renders it to the canvas.

3.3. Description Logics
A typical DL-based KB has two components, ABox and TBox. TBox has an analogy to classes in the object oriented programming language, and ABox has an analogy to instances. The standard reasoning methods applied on a TBox is to check whether a concept is satisfiable or a concept C contains another concept D. The reasoning on ABox is mainly to decide whether an instance belongs to a certain concept. Using tableau calculus, reasoning on TBox is more efficient than on ABox [12]. Therefore, we only reason on TBox in our applications.

The basic constructing blocks for TBox are atom concepts (unary predicates) and atom roles (binary predicates). By convention, capital letters C and D are used to represent atom concepts, R and S are used to represent atom roles. To form complex concepts, various constructors are need, and the more constructors, the more expressive power and the more complexity of computing.

Each role R has a domain C and a range D, in other words, a concept C has a property R whose values belong to D. When some restrictions are applied on C and it transform to C', the range slot R of C' may alter correspondingly. For example, concept human has age range form 0 to 200, for a married human (human ∩ hasMarried.True) concept it has age range from 20 to 200 according to the law, it also gains new slot of has mate. When the system generates UI and maps concepts to options, we must segment the range of a slot properly to make reasoning process useful for our application, and each segment of the range is contained by the range, in other words, each segment is a sub class of the range. We discuss the methods of segmenting concepts in section 4.2.

4. Framework

Our initial implement did not realize a visual model authoring tools, and the design time architecture shared with the runtime architecture. We also have two assumptions on the underline KB. First, the language should be decidable. Second, reasoning is performed only on the TBox, since we do not feature reasoning on the ABox.

4.1. The framework

We describe our initial interaction model of the generated UI as a series of iteration. During each round, the user selects an option and the generated UI checks whether there are options inconsistent with the current situation or there are missing options, then the UI updates panels according to the results of the checking process.

As depicted in Figure 1, there are two models in our UI generator. The application model specifies which part of the underline KB should be extracted and rendered on the canvas. The concerns of the presentation model are layouts, windows, icons, etc. Every application model has a target concept either corresponding to a concept in the KB or a combination of some concepts of the KB. The UI generator extracts the target concepts and concepts of its slots recursively from the KB, and then renders them on the canvas according to the specifications of the corresponding presentation model. The UI generator marks the target concept as the current concept, and the current concept varies during an interaction. When a user is choosing some options, more restrictions are applying to the current concept; hence the available options should be adjusted to ensuring they are compatible with the changed current concept.

For example, at the beginning, the current concept states that the recipient is a human. After a user selected that the human has a sex of female, then the options of pregnancy became available, but all male-relationship options (e.g. fatherhood, brotherhood, etc.) were removed. Furthermore, if the user selected that the present was prepared for the Mother’s day, only motherhood relations held.
Fig2 depicts the situation that a customer is preparing a gift for the traditional Chinese festival – the Spring Festival, and there is no special restriction on the recipient. If the festival is the Father’s day, only fatherhood relationships remain, as illustrated in Figure 3. This is done by the following rule.

\[ \exists \text{hasFestival.Father'sDay} \land \forall \text{hasRelation.}(\text{ChildFather} \cup \text{GChildGFather}) \]

The rule above states that if a giver is preparing a gift for a coming Father’s day, the giver has either a relationship of child-father or of grandchild-grandfather. Other relationships disjoint with these two relationships and have been removed from the panel automatically.

Figure 3. Recipients of the Father’s day

4.2. The application models

As we mentioned early, each application model has a target concept. Although we express the target concept by the OWL in the model specification file, for simplicity we describe it by pseudo code in this paper. The class definition of the target concept is listed in Table 1; each application model holds an instance of it. The owlConcept member targets a concept in the KB, and the usedSlot member is for selective using its slots (i.e., ObjectProperty). For each slot, concepts contained by its range are rendered to panels. The details of those concepts are specified by concept descriptors; Table 2 lists the prototype.

Concept descriptors are used to describe other concepts that are ranges of some properties related to the target concept. The range of a slot may be a concept that has its own slots, so we encounter indirect referenced concepts. Whether it needs to describe those indirect referenced concepts depends on the granularity of their direct referee. We have three kinds of granularity for the UI generator to make options: atom, literal, slot.

Table1. Class of the target concept

```java
class TargetConcept{
  OWLClass owlConcept;
  OWLObjectProperty usedSlots[0..N];
};
```

The atom granularity is used in case that a property has a range of a simple taxonomy, and the system generates options for each atom concept whose sub-concept is null. In our case, we use taxonomy of festival to describe a slot of situation; the system generates festival options per atom, e.g., the Christmas day, the Spring Festival. It is worthy to note that not every atom concept engenders an option. For example, we organize festivals according to calendar, and classify them into the lunar calendar set and the solar calendar set. This is accomplished by the rdfs:subClassOf constructor. Although a representation model may use lunar festival and solar festival to group options, in case of atom settings they have no corresponding options on panels.

Table2. Class of the concept descriptor

```java
class ConceptDescriptor{
  OWLClass owlConcept;
  OWLObjectProperty usedSlots[0..N];
  enum {atom, literal, slot} granularity;
  OWLObjectProperty granSlot;
  OWLClass literal[1..N];
};
```

The slot granularity is used when a concept C, which is referenced directly or indirectly by the target concept, has a slot R whose range is a concept D. Then for each concept C ∩ R.Di (Di is one of sub concepts of D according its granularity), the generator produces an option for it, and the granularity of D is also specified by the corresponding concept descriptor. The role R is specified by the granSlot member. For example, a human has properties of age, sex, etc. If we specify the slot granularity and assign property hasAge to granSlot, concepts as baby, adult, etc will be listed on panels. If hasSex assigned to granslot, male and female person concepts appear in panels as options.

The literal granularity is applied on either atom concepts or concepts with slots. When an instance of concept descriptor invokes the literal granularity, arbitrary sub concept of its owlConcept member may engender an option, and these sub concepts are stored in the literal member. The literal here means that these sub concepts have kind of atom attribute, and the system ignores their slots even they have. In contrast with the literal, the slot granularity has not such an atom attribute, the system may generate panels for slots of the corresponding concept.

4.3. The representation models

The representation model of our works is similar to [10]. The design patterns of the swing have
substantially influenced us, e.g. the decorator pattern, the observer pattern [13]. Each concept in an application model corresponds to a panel in the representation model, and the panel is decorated by layouts and controls. A panel may consist of multiple panels if the corresponding concept has multiple slots. If a concept satisfies certain conditions, the model can guide the UI generator to group its “atom (according to the granularity)” sub-concepts as depicted in Figure 2.

4.4. The usage of the framework

At the end of an interaction session, the UI constructs a valid DL-concept (i.e. an OWL class), and various applications make use of it respectively. There are many applications could benefit from our framework, for example, considering the case of a documents retrieving system. A documents classifier has classified documents according to ontology, and assigned each document as instances of concepts. The retrieving of documents in this case is straightforward - just finding all instances belong to the concept gained at the end of the interaction session. In our application, the query system exploits these concepts to retrieve products that satisfy them.

5. Performance analyzing and improvement

Our initial implement uses a KB consists of 434 concepts and 36 slots. We use OWLAPI [5] to manipulate OWL statements, and reason through the DIG compliant RACER. We test our prototype on a machine has a 1.7GHz CPU and 512 MB RAM. If we did not optimize the initial interaction model, its performance is unsatisfactory – the duration of each round ranges form 0.1s to 180s.

After we applied our optimization, the duration of each round is negligible. The optimization consists of two components, preprocessing and refined interaction model. We sketch the preprocessing as follows.

- Every situation (i.e., current concept) has a map stores its slots and their ranges, we should decompose these range concepts to “atoms” according to its granularity.
  - Decomposing ranges for roles of the target concept.
  - For each used slot of the target concept R and its decomposed range {C0, C1, ..., Cn}, we construct a set of concepts {R.C0, R.C1, ..., R.Cn}. For each constructed concept, we decompose its ranges of slots.
  - Repeat last two steps for all direct or indirect referenced concepts and their slots.

During each round of the refined interaction model, when a user clicks, the UI finds which slot R has changed options to the concept D. Then the UI backtrack to the concept C that does not applied restrictions on the slot R. For each slot S other then R, the UI computes the intersection between the range of S of C and the range of S of R.D, and gets the decomposed ranges of the current concept C ∩ R.D, and updates panels.

For a KB of large scale, we suppose there are two ways to keep our optimization working. The first is offline reasoning, and store results into a database. The second is to exploit a multiple stages interaction model, since each stage focuses on limited concepts, we can expect a quick response time.

6. Conclusion

We have proposed a model-based ontology driven UI generating method, and present our initial implement and its performance. Our framework boosts the development of a knowledge-tense system - it generates UI according to declarative models automatically. The UI guides a user to construct valid DL-based concepts, and the concrete application exploits the concepts to retrieve knowledge form the KB. The main contribution of this paper is twofold.

- We exploit a formal logics base in our model. Hence we ensure that only reasonable options appear in canvas and the validation of user inputs is always done.
- We develop a structural method to extract fragments of a KB and render them on panels. More specifically, we introduce the granularity which has been proved a flexible and powerful mechanism in our initial implement. We also refined the interaction model and its performance has improved significantly.

Since our project is still ongoing, we did not provide visual tools for designers to generate model specifications at the current stage, and we author model specifications through protégé 2000 and the OWL plug-in. Our future works will refine the representation model, and may consult the task model in the literature [14]. There are two main objectives in these researches. The first is to implement visual design-time tools to make our framework a powerful and ease of use tools for generation of UI in knowledge dense applications. The second is to map the representation model to XForm [18] model so our framework gains the feature of platform (e.g., desktop device, mobile device, handset, etc.) neutral.

The other field we will explore in future works is to integrate our frame with ontology portal services. These portals provide queries based on semantics, and respective interfaces [19]. It also reveals that people
often need information of specific format [20]. We have found that the cooperation between application models and representation models facilitate supporting these requirements. Moreover, with a powerful visual design tools, users even customize their own portal interfaces.

7. References

http://potato.cs.man.ac.uk/papers/dl00.pdf
[18] http://www.w3.org/2004/02/01/RecWaveform/