Automated Domain-Aware Form Understanding with OPAL

with a Case Study in the UK Real-Estate Domain

Xiaonan Guo

St Cross College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2012

Abstract

Web forms are the interfaces to the deep web, and automated form understanding is the key to unlock its contents. It is a fundamental problem in many applications and research fields, such as deep web crawling, data integration, or information extraction. It is also essential for improving web usability and accessibility.

Form understanding is an inherently empirical problem. Existing form understanding approaches are restricted by exploiting limited and domain independent feature sets leading to overly generic and monolithic algorithms. In response, we present OPAL (***O**ntology based web **P**attern **A**nalysis with **L**ogic*), a domain-aware form understanding approach, that addresses all these limitations through a novel multi-scope approach. OPAL achieves this through a domain independent form labeling and a domain dependent form interpretation. In form labeling, OPAL associates texts with fields as labels through three domain independent scopes exploiting textual, structural, and visual information. In form interpretation, OPAL integrates the form labeling obtained with a layer of high-level domain knowledge to classify form fields and to repair the form model.

To ease the task of designing domain schemata, we develop the template language OPAL-TL to express domain types and their structural constraints. With OPAL-TL, we describe common design patterns as templates maintained in a library. Thus, the adaption to new domains often requires only instantiation of the templates with corresponding domain types.

We conduct extensive experiments, that cover both domain independent cross-domain testing with standard form understanding benchmarks, and a domain-aware evaluation with two domain datasets randomly selected from real estate and used car domain. OPAL outperforms previous works by a significant margin and pushes the state of the art to near perfect accuracy ($> 98\%$).

In an effort to integrate OPAL with an entire data extraction pipeline, we plan to extend OPAL with form probing and to exploit information obtained by other data extraction components, e.g., result page analysis.

Acknowledgements

Studying at Oxford has been a great honor. The scene when I moved to the city four years ago from thousands of miles away is still vivid in my mind. Anxiously waiting for a visa, travelling nearly thirty hours, it was tiring, but exciting.

I arrived exactly one day before the matriculation for the Michaelmas term, otherwise I would have to wait for a whole term to start my study. I went through the probation year and transferred from MSc research program to D.Phil program directly, otherwise I would have to spend at least one more year for my studies. I became a member in the DIADEM group, where I took part in great research and met outstanding people. It is a great pleasure to thank everyone who helped and supported me. The knowledge and experiences I have acquired here will stay with me for the rest of my life.

I am sincerely and heartily grateful to my brilliant supervisor, Georg Gottlob, for the invaluable guidance and support through both my research and my life. Thank you for offering me the opportunity to study here, for supporting my direct transfer to D.Phil study, and for involving me in the DIADEM project. Thank you for all the encouragement, and also for the three simple words, "Are you happy?" which always cheered me up.

I wish to express my heartfelt gratitude to four extraordinary computer scientists: Chrisitan Schallhart, Tim Furche, Giovanni Grasso, and Giorgio Orsi. Thank you for your great suggestions in my research. Thank you for the time and patience to discuss any problem I encountered during my research and my life, to help sort out my ideas and clear doubts, to improve my academic writing, ... This dissertation would not have been possible without your help and support.

It has been a pleasure to study in Oxford and to meet great people. Thank you to Andrea Cali, Luying Chen, Omer Gunes, Andrey Kravchenko, Andrew Sellers, and Cheng Wang. Especially, I would like to thank Andreas Pieris for all the tutoring and discussions in Datalog for me. Thank you to Niki Trigoni and Georg Lausen for helpful discussions and great suggestions for improving both my thesis and my research. I would also like to express my appreciation to Dr Zhong You, for all the advice on my studies and all the help during my stay here in Oxford. Thank you to the Department of Computer Science and to St Cross College for all the support during my study.

No words can express my gratitude to my family. Dear Mom and Dad, thank you for your great advice, your encouragement, your criticism, and for everything you have given

to me. I could not achieve anything without your unconditional love and unwavering support. Dear Aunt and Uncle, thank you for your support and humor that always clear away my anxiety. My two dear cousins, you are no different than brothers to me and you are the best I can imagine. Dear Grandpa, you had no idea how much I wished you could come to see my graduation, although it now is only a dream. I just want to let you know that I am doing great.

Last but not least, I wish to thank all my friends all over the world for your caring and sharing, for your help and support. Thank you for being my friends.

Contents

Chapter 1

Introduction

Forms are the gates to the web. In contrast to the information accessible on the surface web through static URLs, much of today's web contents are generated dynamically from databases which are only accessible through web forms.

Automatic form understanding is essential for deep web search, web extraction and integration, as well as for automated adaptation of forms to improve accessibility or usability, for example, on mobile devices. It discloses the deep content for applications ranging from crawlers to meta-search engines. As early as 1999 [57], the deep web amounted already to over 80% of the information published on the web, with an increasing tendency. However, the problem of form understanding has received far-from-enough attention other than as component in specific applications such as crawlers.

This dissertation presents OPAL, short for ***O****ntology based web* ***P****attern* ***A****nalysis with* ***L****ogic*, a comprehensive approach to automatic form understanding. OPAL targets at two main tasks involved in form understanding: form labeling and form interpretation, and pushes the state of the art in both problems. As an inspiration for OPAL, we conduct a comprehensive study in the UK real estate domain which underlines the importance of forms in web data extraction and demonstrates the appropriateness of OPAL's domain schemata in form understanding.

This chapter first motivates the form understanding problem in general, before illustrating OPAL's approach on a concrete example. In the end of the chapter, we summarize our contributions and lay out the organization of the thesis.

1.1 Motivation

In a white paper [10] from July 2000, it was estimated via analyzing the overlap between pairs of search engines that public information on the deep web is currently 400 to 550 times larger than the surface web. A subsequent study [14] in April 2004 estimated 450,000 online databases (among which 348,000 were structured). The hidden contents

are generally not accessible through standard search engines as they are not programmed to retrieve dynamic pages resulting from database queries, although there are efforts like Google's deep web surfacing [48] that adds results from pre-computed web form submission, and Yahoo! Subscriptions that makes a website containing invisible data searchable if it supplies the search engines with its hidden data.

Web forms are the gateway to these high quality, large quantity, and well managed deep web contents, which are provided dynamically in response to user queries [10, 21]. Form understanding is thus fundamental for many deep web related areas: crawling and surfacing the deep web [13, 48, 56], integrating web data [19, 32, 34, 67] for providing a unified and consistent view of heterogeneous data sources, classifying the domain of web databases [5] for web site classification, sampling the contents of web databases [3, 49], improving data quality [15], and matching interfaces across domains [11, 66]. Moreover, websites are increasingly used on devices or in contexts not anticipated by their developers, where form understanding in such cases can significantly improve accessibility of forms and support assistive technologies [37], e.g., for mobile use such as keyboard layouts, gestures, and auto-completion appropriate to the type of field. It is also essential in developing advanced form-based web applications, e.g., for automated testing or validation. With an attempt to uncover the hidden information, previous works have (partially) addressed the problem with an emphasis on specific aspects, such as matching form or data source schema [18, 31, 53, 55, 59, 64, 65], automatically filling out forms [2, 61, 62], and locating relevant data sources [4]. This dissertation first focuses on the general form understanding problem and then applies its results in an automated form filling tool.

Form understanding is an empirical problem that relies on observations of form design patterns from different aspects of views (e.g., structural, visual observations). It requires not only accuracy but also robustness in order to keep pace with the frequently changing web technologies and design practices. As part of the DIADEM system [22], which implements a fully automated web data extraction pipeline, OPAL as its form understanding component must (i) detect relevant forms, (ii) find their form fields, (iii) group fields into segments, (iv) assign labels to the fields and segments, (v) classify the fields and segments semantically, and (vi) align the forms with a domain ontology. Beyond form understanding, form filling is also required to reach actual data.

For its importance, form understanding has attracted a number of approaches in deep web search [48, 56, 66, 67], web querying [19, 68] and web extraction [60], which have focused on observing commonalities of web forms in general to exploit these with specifically tailored algorithms and heuristics. Despite reportedly good performance, such approaches prove to be inherently limited: (1) There is a tendency towards *sophisticated heuristics* using a single type of observations. For example, [19] analyzes a form only

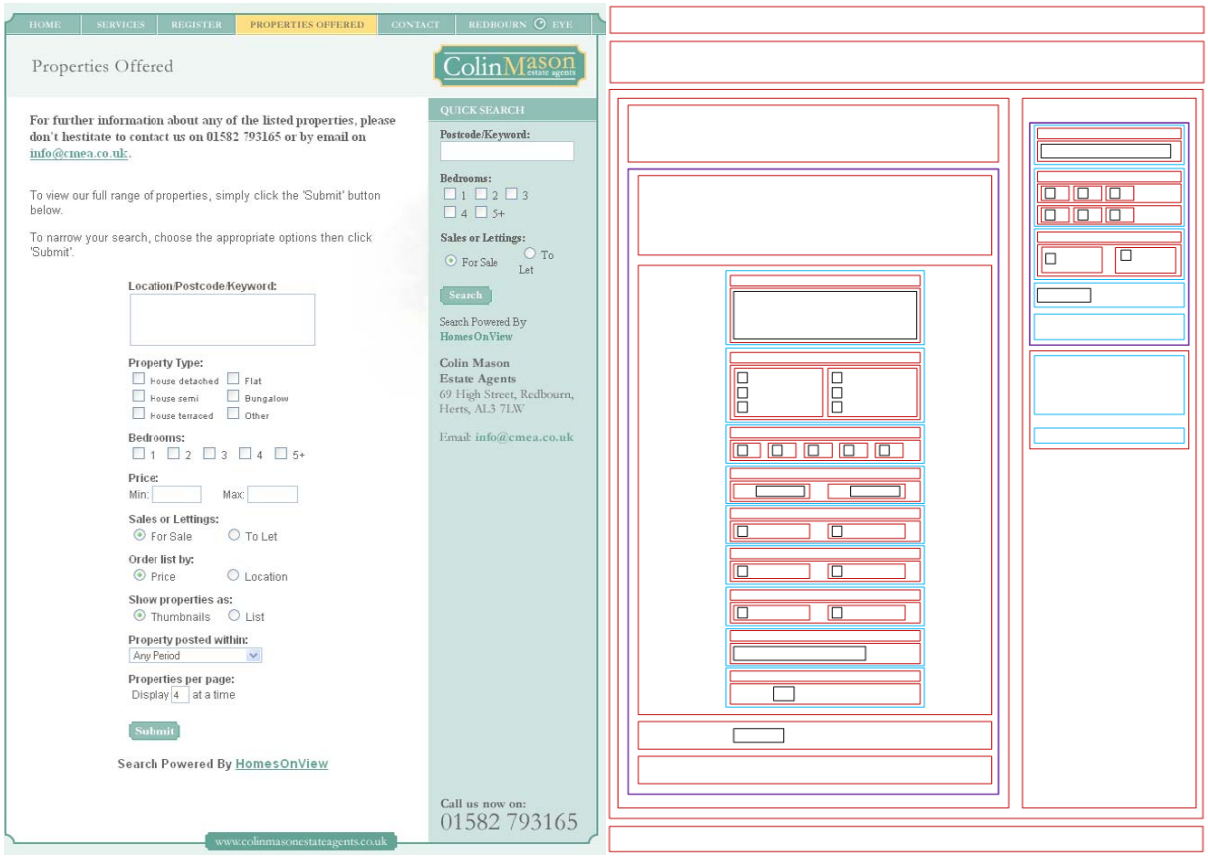
visually, whereas [38] focuses on token streams encoding only textual content and field types. (2) These heuristics are translated into *monolithic algorithms,* resulting in systems that are hard to maintain and adapt. For example, [66] and [52] encode assumptions on alignment and spatial distance of fields and labels into their approaches, [38] employs hard-coded token classes for specific concepts, such as, "min", "from" vs. "max", "to". (3) Targeted at forms across all domains, these heuristics must be general enough to *fit various layout styles.* Although the need for domain-specific methods in high accuracy settings is acknowledged in [52] and [38], domain knowledge is not adopted in form understanding. We discuss these form understanding approaches in detail in Section 2.3.

Form understanding, as a fundamental step in the web data extraction process, however, is not addressed at all by most data extraction approaches [41, 42, 50, 60]. They focus only on data extraction from a given set of result pages, falling short in providing an automated and integrated extraction pipeline. A few of them [6, 47] adopt user-guided form understanding. In particular, existing web data extraction approaches fall into the following categories: (1) In *manual wrapper programming*, a user needs to fill the web form and specify all data to be extracted, e.g., Jedi [36]. Since such wrappers are manually programmed, they support at best form understanding and filling. (2) In *supervised wrapper generation*, a user manually navigates to the desired information before labeling data examples. By generalizing these examples, such approaches generate a wrapper to automatize the recorded actions including form filling. For example LiXto [6] and Wiccap [47] fall into this category. No form understanding is integrated, as the user has to record the query and navigation steps leading to the result pages, offloading the analysis, form filling, and query submission strategy to the user. This process is recorded and replayed during the actual extraction. (3) In *machine learning* approaches, wrappers are induced from examples annotated by users, e.g., [41, 42, 50]. However, these tools take a set of result pages as input, disregarding the fact that form understanding is necessary to reach these pages. (4) *Unsupervised data extraction* aims at fully automatized data extraction. Among these approaches, DeLa [63] is the only tool that presents a more complete data extraction pipeline—including form understanding and query generation. It starts with form crawling, supported by HiWe [56], which collects labels on a given web form and queries the form. Afterwards, it generates regular expression wrappers based on the HTML tag structure, arranges the extracted data in a table, and labels the table columns, exploiting information gathered during form understanding, such as field labels. The same group introduced ODE [60], also exploiting information from web for data labeling, yet ODE itself does not include a form understanding phase.

To attain an accurate and robust form understanding, it is necessary to overcome the limitations of existing approaches. Thus, we aim for (i) a non-monolithic approach which

integrates several different features to achieve stable results without requiring overly complex heuristics. In particular, we want to combine textual, structural and visual analysis. (ii) incorporation with domain knowledge in order to semantically understand a form. For broad applicability and to develop a well-engineered tool, the domain knowledge should be parameterizable, where the system have no dependency on any domain and domain schema can be easily switched just as a parameter to the system.

1.2 OPAL by Example



(a) (b)

Figure 1.1: *Colin Mason* web page and box model.

This section briefly introduces the OPAL approach (***O**ntology based web **P**attern **A**nalysis with **L**ogic*) to form understanding using the *Colin Mason*[1] real estate agency website as an example. Figure 1.1 presents the web page on the left and its simplified CSS (Cascading Style Sheets) box model on the right. The page contains two forms: one for detailed

[1] `http://www.cmea.co.uk/properties.asp`

search (the larger form on the left) and the other for quick search (on the right). Each of these forms is marked up with a proper `<form>` element.

In Figure 1.3(a), the detailed form is subdivided into (a) a text input area, to provide location related information, (b) six check boxes, to select the preferred property type, (c) five check boxes, to specify the number of bedrooms, (d) two text boxes, to limit the property price, (e) two radio buttons, to specify the purpose of the search, (f) two radio buttons, to indicate the search result order, (g) two radio buttons, to choose the style for displaying search results, (h) a text box, to set the number of results per page, (i) a drop down list, to restrict the search to recently posted offers, and (j) a submit button. All components of the quick search are: a text input area for post code, five check boxes for bedrooms, two radio buttons for search purpose specification, a button for form submission.

At the HTML DOM level, the detailed form consists of a `<table>` element which contains all form fields and some guidance above its fields. In the main part, each of the components (a)-(j) is defined by another `<table>` element, which consists of two `<tr>` elements with the first one containing the component header in bold face and the second one covering the fields with their labels. For example, in (c) (Figure 1.2(a)) "Bedrooms:" occurs in the first `<tr>` whereas the other five check boxes are grouped in the second `<tr>`, with each field and its corresponding text label presented in a single `<td>` element (Figure 1.2(c)). For the property types in (b) (Figure 1.2(b)), the second `<tr>` is subdivided into two `<td>` elements, i.e., the six check boxes are grouped into two columns, where three fields are interleaved by three text nodes with no single HTML element for individual fields (Figure 1.2(d)). Each field in the components (c)-(g) is presented in a dedicated `<td>` element together with its corresponding label. The quick search form in the right side bar is designed in a similar fashion with two `<tr>` elements for each component and with each field and its label having a dedicated `<td>` element. Figure 1.1(b) briefly illustrates the structural properties of both example forms. In the remaining of the section, we disregard the quick search form and focus on the main form only.

Given a web page, OPAL proceeds in four stages as detailed in Sections 1.2.1 to 1.2.4: First, it extracts the page model, obtaining representation of the rendered web page. Second, it performs a domain independent form labeling, organized in three scopes, which expands the analysis area from the direct field neighbors to the entire page. Third, OPAL integrates the obtained form labeling with a layer of high-level domain knowledge to produce a form interpretation. Finally, as an application, OPAL fills the analyzed form with values supplied in a master form.

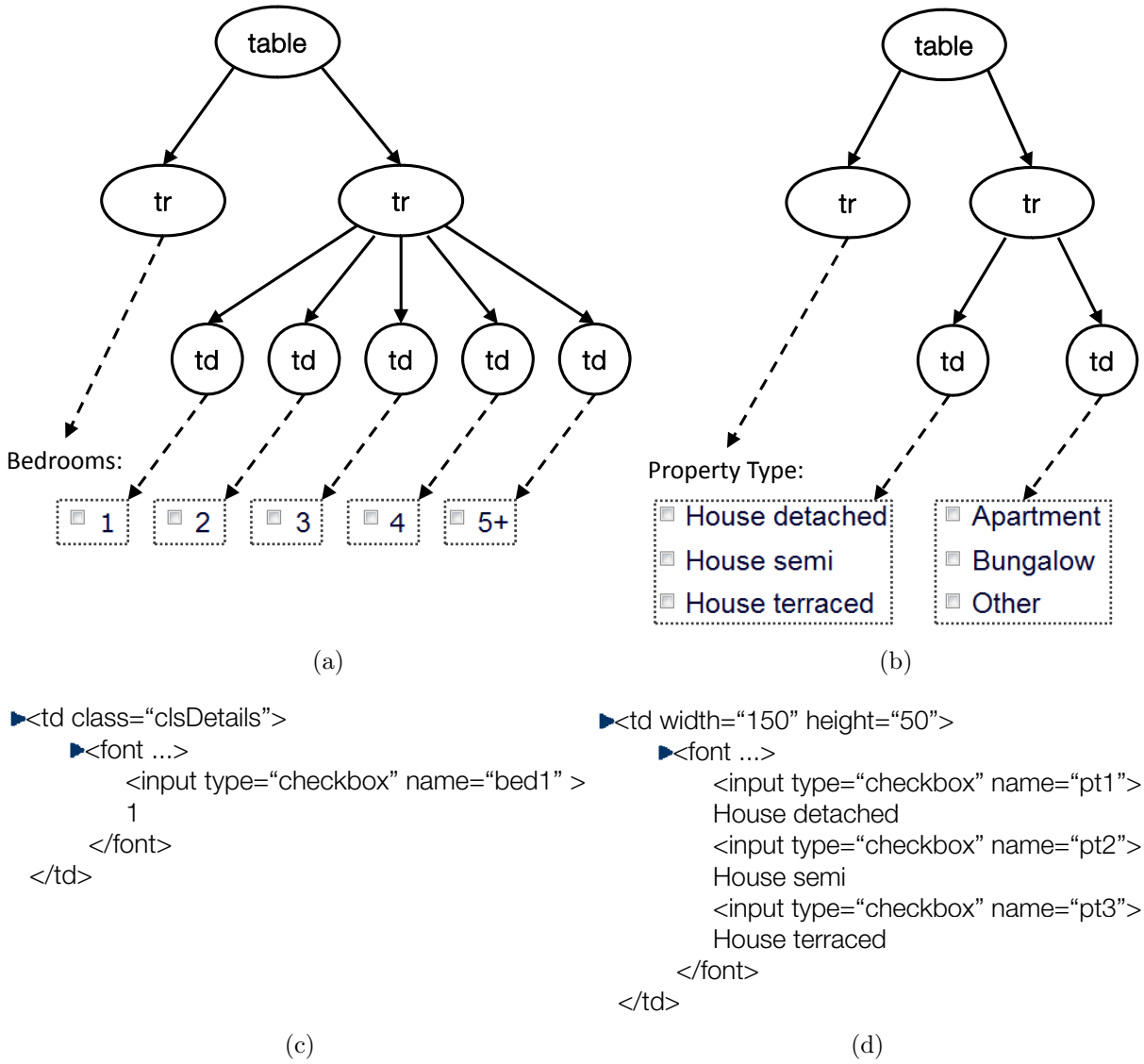
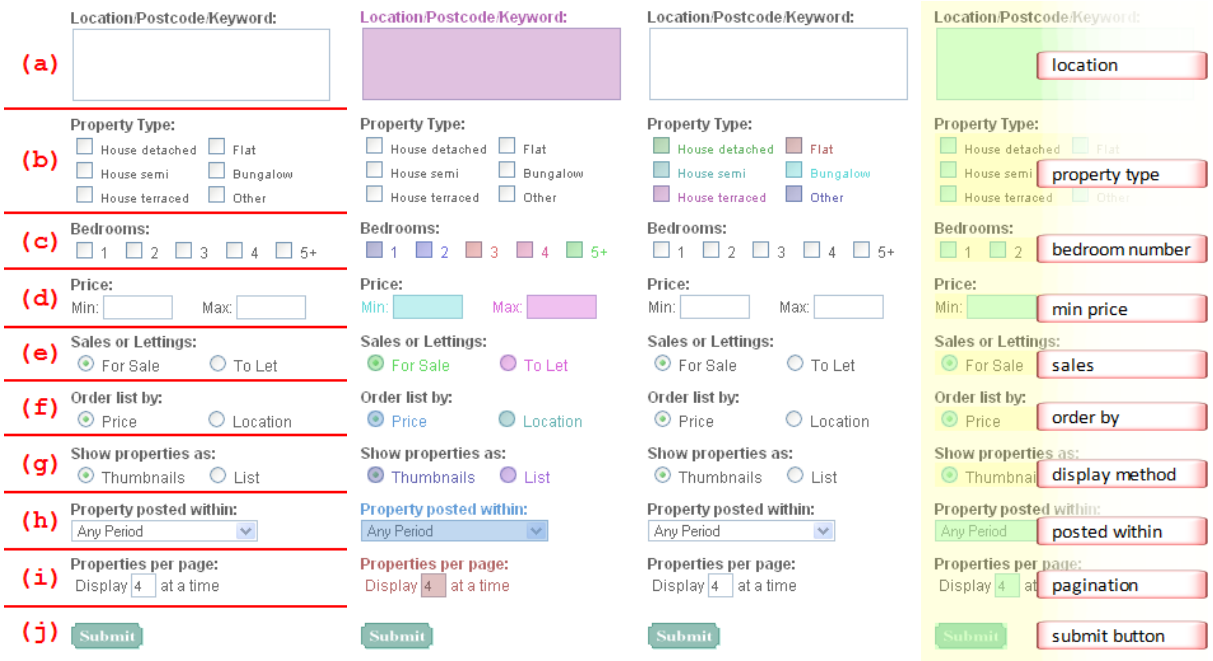


Figure 1.2: DOM for Bedrooms and Property Types Segment.

1.2.1 Extracting the Page Model

We first need to represent the contents and layout information of a given web page, before we proceed with the analysis. OPAL constructs the *page model*, which models the DOM structure of the given web page and all CSS-related visual information. CSS data is used to construct a box model of the page that describes the visual alignment of the page elements. Additionally, the textual contents on the web page are annotated by general and domain-specific linguistic annotators. In the *Colin Mason* example, terms such as "location" and "postcode" refer to geographic information, and terms such as "terraced" and "flat" are annotated as property types. For form understanding, the linguistic annotations are provided in reference to a domain ontology with a twofold goal: (i) to annotate



(a) original form (b) field scope (c) segment scope (d) domain scope

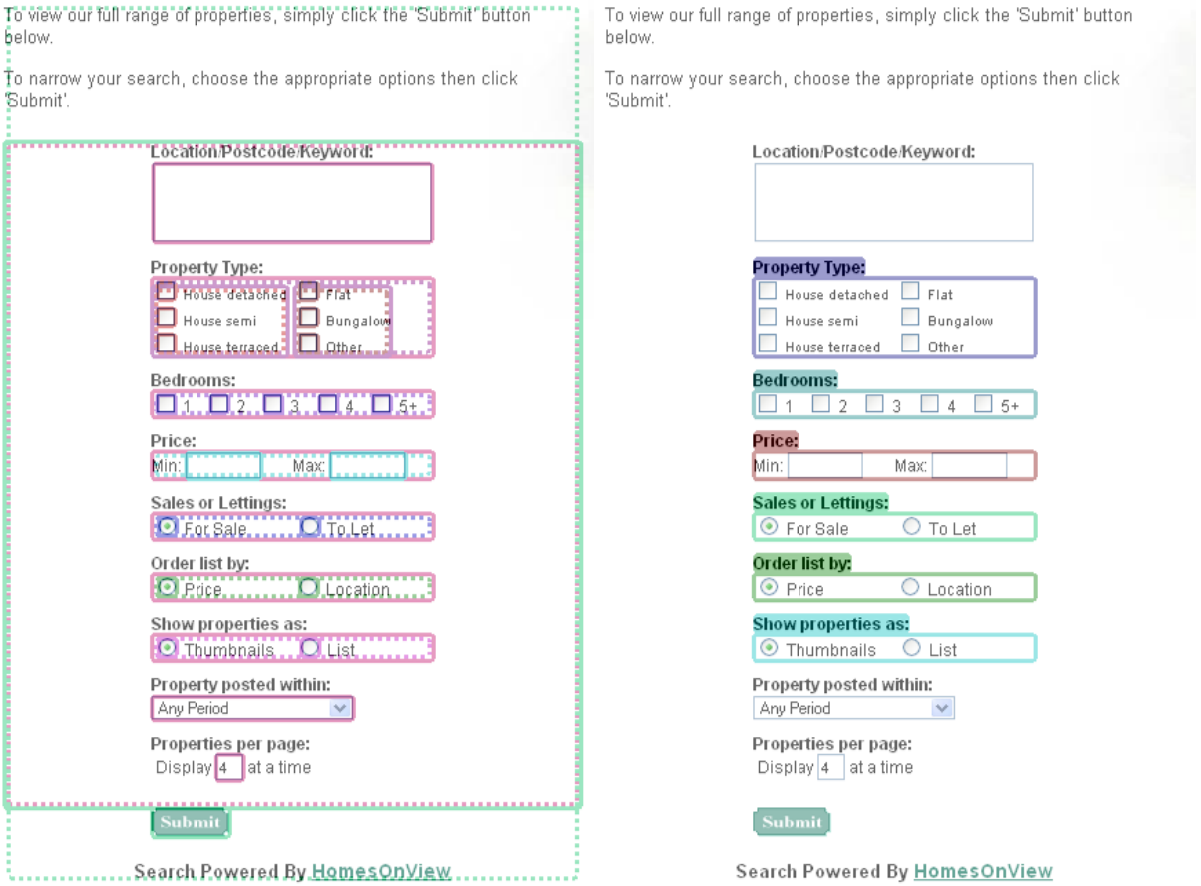
Figure 1.3: *Colin Mason* multi-scope labeling.

the elements in the form with domain-specific types, and (ii) to validate the produced form model.

1.2.2 The Domain Independent Scopes

With the complete page model as input, we firstly identify all visible fields, and then continue with our multi-scope form analysis procedure, encompassing the field, segment, and layout scope, working intuitively as follows: At field scope, OPAL applies local heuristics on individual fields. At segment scope, it exploits the segment hierarchy of the form. At layout scope, OPAL identifies visually related texts and fields using the visual rendering of the page.

Field Scope. OPAL starts with analyzing the information directly associated with individual fields. We associate labels to fields with heuristics that construct basic units called *elements*. The assignment relies on (i) the `for` attribute of the `<label>` element that maps to the `id` attribute of a field, and (ii) the least ancestor of a unique field. This least ancestor is the ancestor of a field closest to the document root which has no other field as descendant. OPAL associates all text descendants of the least ancestor to its unique field descendant. Heuristic (i) proves to be highly reliable since designers seldom set that attribute to the wrong value. In all our evaluations (encompassing approximately



(a) form segmentation (b) segment labeling

Figure 1.4: *Colin Mason* segmentation and labeling.

700 forms), we only observed *for* attributes pointing to non-existing fields but we never encountered *for* attributes referring to wrong fields. Heuristic (ii) is also reliable since the structural grouping of labels and the related fields is a very common design pattern.

With reference to the *Colin Mason* example, there is no direct reference provided by `<label>` elements, as the labels are represented by simple text nodes. The greatest unique ancestor heuristic correctly labels all fields except the ones in component (b): We associate the numbers in (c) with the check boxes in front of them, and similarly we label all fields in component (d)-(g). The text box in (i) is labeled with three text nodes, namely "Properties per page:", "Display", and "at a time". The fields in (a) and (h) take those texts as labels which appear directly above them. At field scope, we do not assign any label to the six check boxes in (b), since there is no DOM element satisfying the least ancestor requirement. Figure 1.3(b) shows the labeling results for the form at this scope: assigned texts share the same color as the field.

Segment Scope. In order to assign the remaining labels we increase the scope of the analysis from the fields to groups of fields, i.e., *segments*. Before assigning any labels at this scope, we need to construct the segmentation by identifying suitable DOM nodes.

Roughly speaking, OPAL constructs the segment tree by exploiting iteratively the DOM and selecting nodes that have at least one field as descendant and more than one child, and do not conflict with the groups of similar fields. Fields are similar if (i) they occur in sequence, (ii) they are similar to each other in terms of their HTML types, style information, or value of certain attributes, e.g., *class*, and (iii) their least common ancestor contains no other segments or fields.

If a given page contains at least one `<form>` element, OPAL processes the contents of each `<form>` element as one form. Otherwise, if there is no `<form>` element, OPAL proceeds with its analysis considering the entire page as a single form to only determine the resulting form(s) during the domain scope. In the *Colin Mason* example (Figure 1.1), there are two forms clearly represented by a `<form>` element each.

As a result of form segmentation, (the second `<tr>` of) each component on the form yields a segment if it contains more than one field, i.e., (b)-(g). Among these segments, OPAL further subdivides (b) into two sub-segments, each covering one column corresponding to a `<td>` element (Figure 1.4(a)). The results on the quick search form are similar except for the "Bedrooms" segment, which is further subdivided into two rows corresponding to the HTML table row design.

After segmenting the form, the labeling heuristics are applied to fields inside the segments. OPAL identifies the repeated patterns of interleaving fields and texts to align them accordingly. Thus, we correctly assign each check box in (b) with the text appearing after it as shown in Figure 1.3(c). Since all the members of the other segments are already labeled at fields scope, we do not perform any segment scope labeling.

Besides field labeling, OPAL also associates segments with texts, which are referred to as segment labels (see Figure 1.4(b)). This is achieved by taking yet unassigned texts which start an interleaving pattern or are descendants of the least ancestor of the segment (but not occurring within the segment in concern). For example, the latter assigns the text in bold face appearing on top of each segment as the label, e.g., "Price:" becomes the label for the segment consisting of the two text boxes associated with "Min:" and "Max:".

Layout Scope. Whenever the analysis at the previous two scopes leaves some of the fields unlabeled, we further enlarge the scope of the analysis from the segments to the entire form. At this level, the heuristics use visual information from the CSS box model to assign the missing labels. In general, OPAL prefers texts to the top-left of a field. On

the *Colin Mason* form, even by exploiting visual information, OPAL does not produce any further label assignments. We discuss this scope in Section 4.4 in detail.

1.2.3 The Domain Scope

Once the form has been analyzed, both structurally and visually, the analysis proceeds to the domain scope. At this scope, we employ domain knowledge to classify, consolidate, and verify the labeling and segmentation obtained so far. To this end, OPAL is configured with constraints specified in OPAL-TL (Section 5.1) that model typical patterns of forms in a given domain. Figure 1.3(d) shows the final form element classification as produced by OPAL (for sake of clarity, the figure shows only the classification of elements on the left hand side of the form).

Classification OPAL annotates fields and segments with types based on the annotations associated with the text labels. On the form (Figure 1.3(a)), the fields in (a), (b), (e), (g), (h), and (i) are classified immediately into domain concepts. However, fields in (c) and (d) obtain generic annotations, since their labels do not carry domain specific information. The two radio buttons in (f) are miss-classified as price and location element respectively. OPAL corrects these issues in subsequent steps.

Completion For each field, OPAL completes its classification by exploiting information associated with its parent segment or neighboring fields. Thus, the five check boxes in (c) are categorized as bedroom element, and the two text boxes in (e) are successfully classified as price element with price type "min" and "max" respectively. Furthermore, the two radio buttons in (f) are both annotated as order-by elements in addition to their initial price and location classification.

Model Repair With the classifications, OPAL verifies the resulting form model against domain specification and repairs the model whenever necessary.

Disambiguation. Fields may be associated with multiple and potentially inconsistent annotations. In our case, both fields in (f) bear multiple classifications, e.g., as order-by and price. Since we specify in the domain schema that the order-by concept has higher precedence than the price concept, we resolve this ambiguity by classifying both elements as "order by" elements.

Verification. The segmentation produced so far relies on the DOM structure. Therefore, OPAL must create virtual segments and eliminate unnecessary segments whenever the syntactic and semantic structure do not match. In area (b) there are two segments, each containing three check boxes. Both belong to the "Property Type:" segment, and

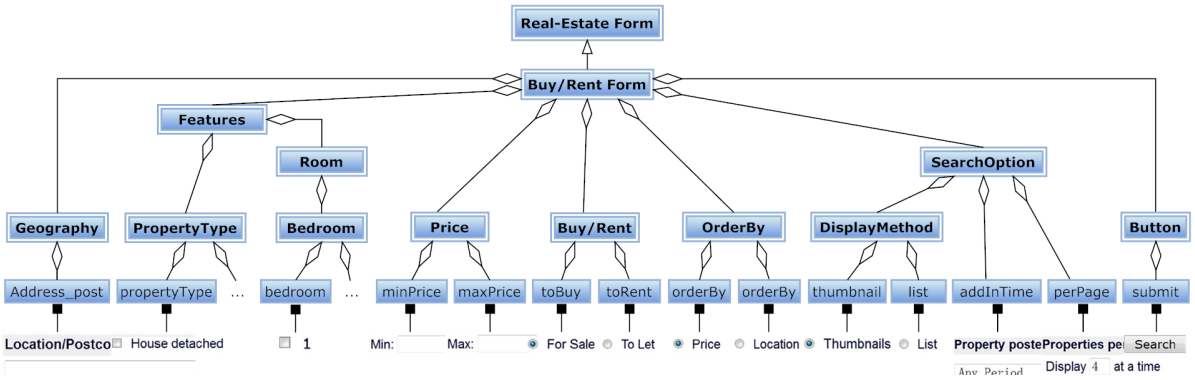


Figure 1.5: *Colin Mason* form model

contain only fields of the same type. For this reason, we only keep their parent segment in the result. The same argument applies to the "Bedrooms:" segment in the quick search form.

Reaching the form model. As the result of OPAL's multi-scope analysis, we conclude that the detailed form is indeed a real estate form with a model as detailed in Figure 1.5.

1.2.4 Filling the Form

Using the form interpretation constructed in the preceding stages, OPAL is able to fill forms: For www'12 [30], we implemented a tool which fills a given form according to the values provided in a master form.

Figure 1.6 presents the GUI for OPAL's form analysis and filling: The top panel allows visualizing OPAL's results by switching on and off the corresponding scopes. The bottom panel shows a master form for the UK real estate domain to specify search requirements. With the field categorization, OPAL maps the fields directly from the master form to the actual form and fills the latter with the provided values.

OPAL fills both forms in the *Colin Mason* example, mapping the master values to the options made available the occurring form controls: For location and min-max prices, the values specified in the master form are typed in directly; For bedroom number, the request value is compared with each member of the check box list until the match is found.

1.3 Contributions

The thesis includes results previously published in [22, 23, 24, 26, 30].

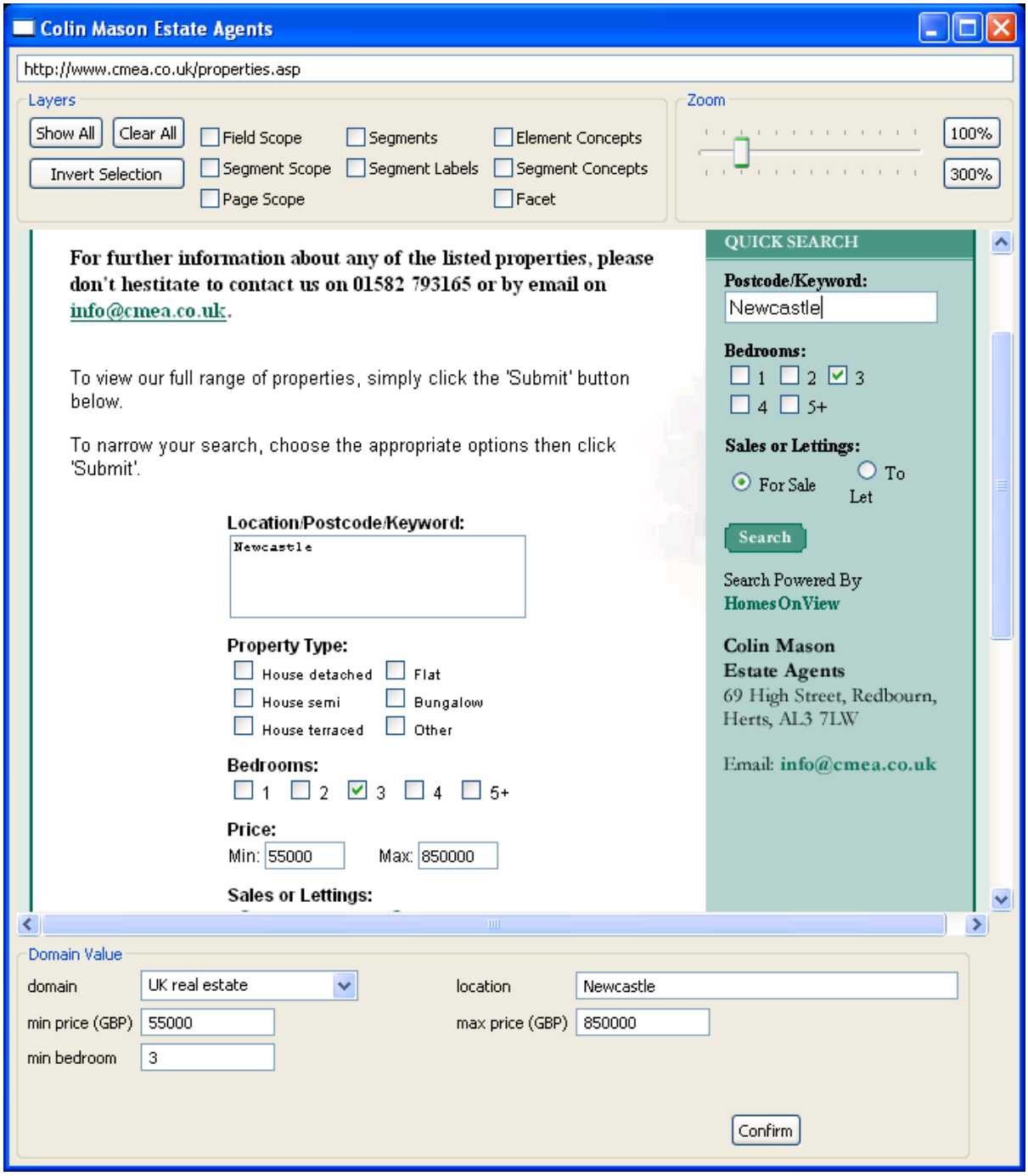


Figure 1.6: *Colin Mason* form filling

(1) **Multi-scope.** OPAL carries out a domain-independent analysis at three sequential scopes, combining textual, structural, and visual features to establish a form labeling by associating textual labels to form fields. (a) At field scope, we consider structural relations between individual fields and labels, e.g., that if a text node and a field have a common ancestor not shared by any other field, then the text node is a label for the field. (b) At segment scope, we exploit regular patterns of fields and labels

occurring in a group. (c) At layout scope, we analyze the relative position of fields and texts in the visual rendering of the page, i.e., whether texts are "visible" or not from the perspective of a field.

(2) **Domain Awareness.** OPAL is domain-aware by continuing with an optional, domain-dependent classification and form model repair stage after the domain-independent analysis. With a thin layer of high-level domain knowledge, (a) OPAL classifies form fields and segments based on textual annotations of their labels and values assigned in the domain-independent scopes, and (b) in case of an imperfect classification due to missing or misunderstood labels, OPAL disambiguates and completes the resulting form model in a repair step according to a domain ontology. This ontology contains classification and structural constraints on domain types, expressed in OPAL-TL, described next. Currently, we cover 40 domain types for the domain of UK real estate forms and 30 for used car forms.

(3) **Template Language OPAL-TL.** To specify domain schema, we introduce OPAL-TL. It extends Datalog [1] to express common patterns as parameterizable templates. It allows for compact and declarative specification of domain schemata. Common phenomena, such as a group consisting of a minimum and maximum field for some domain type, are kept in a template library, such that the adaption to new domains often requires only instantiations of these templates with domain specific types. OPAL-TL preserves the polynomial data complexity of Datalog.

(4) **Declarative Implementation.** All four scopes are implemented as declarative rules to allow easy modification and natural access to the domain model. To this end, we provide a logical representation of all involved data, the DOM tree, the visual rendering of HTML element, the segmentation hierarchy, the textual annotations, and the domain form model. OPAL performs its multi-scope analysis with about 300 rules.

(5) **Extensive Evaluation.** In an extensive experimental evaluation, OPAL achieves near perfect accuracy ($> 98\%$) for form understanding of current forms in the UK real estate and used car domain. It is able to deal with the diverse range of forms occurring in both domains, from sophisticated forms of major aggregators using modern web technologies to simplistic forms of small agencies, often created by non-experts. To compare with existing approaches, even OPAL's first three domain-independent scopes achieve already 94% - 100% accuracy on the 5 domains of the ICQ benchmark and 92% - 97% on the 8 domains of Tel8.[2] Thus, even without domain scope, OPAL

[2] `metaquerier.cs.uiuc.edu/repository/`

outperforms existing approaches in most domains by 5% or more. The experiments also illustrate the contribution of each scope to the final form model, showing the necessity of each scope in OPAL's analysis. The evaluation also indicates that the last 5% - 8% in accuracy require the domain knowledge.

(6) **Assisted Form Filling.** As one application of OPAL, we develop a system to assist domain form filling. The system provides users with a master form consisting of common fields of common domain types, where they specify their search criteria. OPAL then, with its form analysis result at hand, fills each domain form with the provided values. In our evaluation, we show that OPAL fills the analyzed forms perfectly, once a correct form interpretation has been obtained: this requires a mapping of the filling values to the options available offered by different form controls on the form.

1.4 Organization of the Thesis

In Chapter 2, we define formally the form understanding problem and discuss existing research approaches on the topic. Chapter 3 provides an overview of the OPAL approach, initially implemented with an imperative prototype. This prototype proved to be instrumental in showing the need for domain knowledge, while also indicating the limitations of this imperative implementation. Chapters 4 and 5 detail OPAL's domain-independent form labeling and domain-aware form interpretation. This version of OPAL is backed by a mature declarative implementation, introducing our schema design language OPAL-TL. In Chapter 6 we present a domain study of the UK real estate market. We conduct extensive evaluations, both domain independently on multiple domains and domain dependently on the real estate and used car domain, leading to the results discussed in Chapter 7. At the end of the thesis, we present assisted form filling as an application of OPAL. Conclusion and future work are discussed in Chapter 9.

Chapter 2

The Form Understanding Problem

Form understanding is an inherently empirical problem, since every form is different, following culturally determined conventions. Thus, there is a wide variation both in the form structure and vocabulary. Consequently, there is no standard rule for a search engine to understand and fill an arbitrary form. On the other hand, a study [14] in 2004 estimated the number of existing online databases to reach 450,000, rendering automated form understanding necessary to access the valuable contents of these databases.

Form understanding is fundamental in executing deep web related tasks. Hence it is necessary to seek for effective automated web form understanding approaches in order to keep pace with the fast growing (deep) web and the related challenges. In this chapter, we first give a general introduction to web forms and their real world design. Then we formally define the problem of form understanding split into domain independent form labeling and domain-aware form interpretation. At last, we discuss existing approaches addressing the issue and suggest possible improvements.

2.1 Web Forms

A web form (also referred to as web query interface) is used to pass data to a server. It is an effective way for clients to submit parameterized requests to webhosts, expecting, e.g., a confirmation on a transaction, or results on a search query. In general, HTML supports various components to build web forms, such as text fields, radio buttons, check boxes, etc. The purpose of web forms varies greatly, including tasks such as member registration, product search, or complex interactions in modern social media sites.

A web form is usually defined with the HTML `<form>` element, with which a form processing agent must be specified using the *action* attribute. Users interact with forms through fields, which are referred to formally as form controls.

- `<input>` - defines a form control for a user to enter input. Depending on the *type* attribute, an input field can be (i) a text field which provides a single-line text input

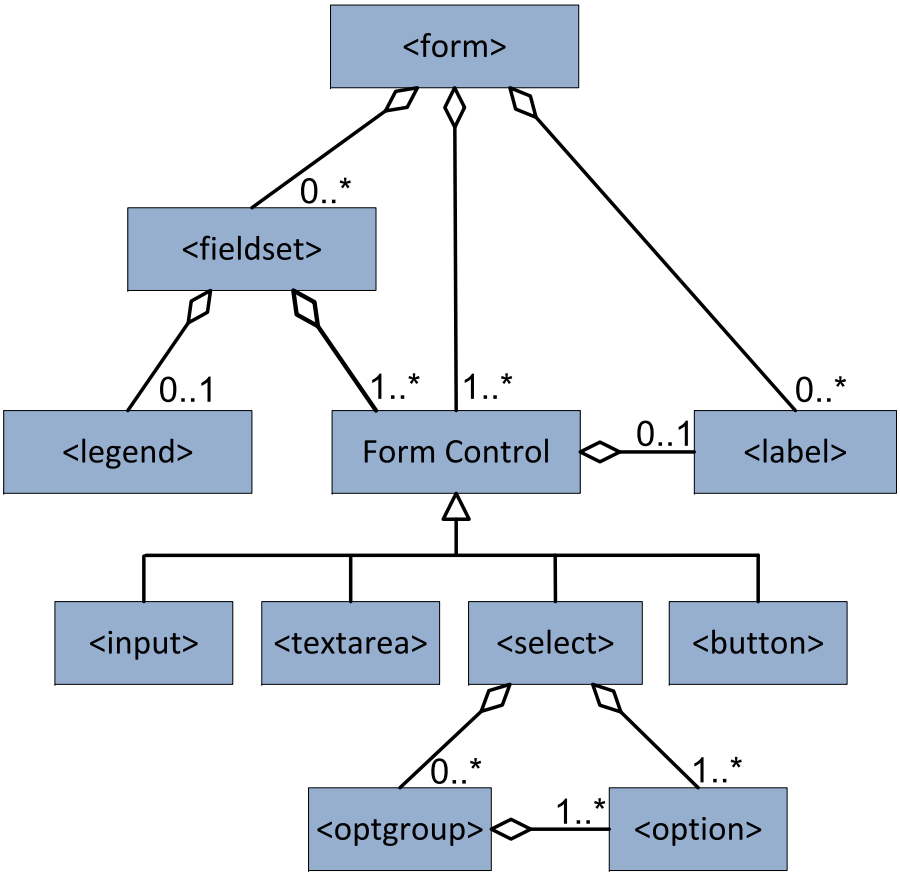


Figure 2.1: HTML web form design

field; (ii) a checkbox or a radio button, where the former can be checked without changing the state of other checkboxes but the latter is the unique selection with the radio button group (by specifying the same *name* attribute on each `<input>`); (iii) a button for submitting or resetting the form, etc.

- `<textarea>` - defines a multi-line text input.
- `<select>`, `<option>` and `<optgroup>` - defines a selection of options. A `<select>` element contains one or more `<option>` elements, which can be grouped using `<optgroup>` elements. Multiple selection is enabled when *multiple* attribute is set.
- `<button>` - defines a submit, reset, or push button. Different from `<input>`, `<button>` elements allow arbitrary child content, such as images.

There are also HTML elements for form styling:

- `<label>` - associates a label with a form field through the matching of *for* attribute of a `<label>` element to *id* attribute of that field.

- `<fieldset>` - defines a group of form fields, which allows a form to be divided into smaller and more manageable parts.
- `<legend>` - defines a caption for a `<fieldset>`, where the `<legend>` appears at the beginning of the `<fieldset>`.

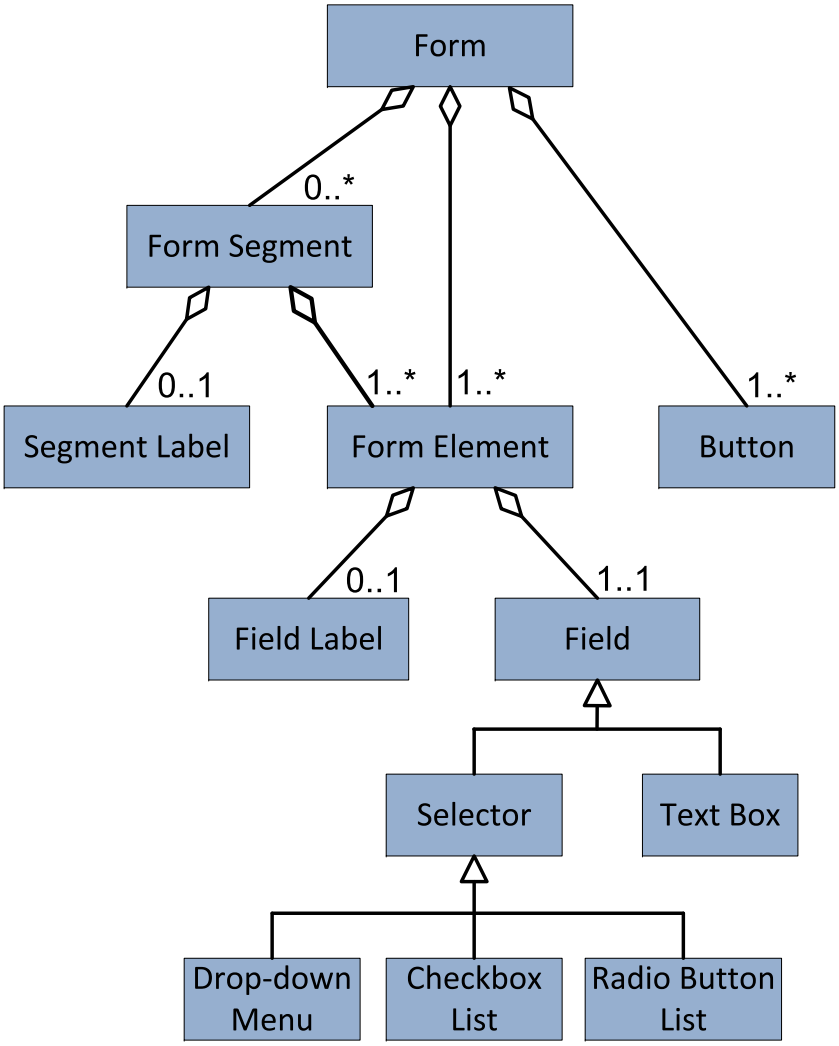


Figure 2.2: Web form in users' view

Figure 2.1 shows an ideal design of web form using the HTML elements introduced above. A `<form>` element contains at least one form field which can be one of the four types: `<input>`, `<textarea>`, `<select>`, `<button>`. Form fields can be grouped into `<fieldset>`, which may or may not have a `<legend>` for its caption. A form field can be associated with a `<label>` element, which provides the label of the field with its text content. A `<select>` element provides all selection values via a number of `<option>` elements or groups of them (`<optgroup>`).

Figure 2.2 presents web objects as understood by users. A web form contains one or more buttons and form elements (text input, checkbox, radio button, drop-down list), which can be grouped into form segments. Form segments and elements can be associated with segment labels and element labels respectively. Moreover, there is also containment relationship between form elements or segments.

However, most of these web design techniques are not adopted by current web designers, for example, `<label>` elements not properly associated with form fields or table cells miss-used for layout purpose. Moreover, the growing popularity of client-side scripting renders the task of form understanding more difficult, by adding e.g. some javascript to implement a submit facility without an `<input>` or `<button>` element.

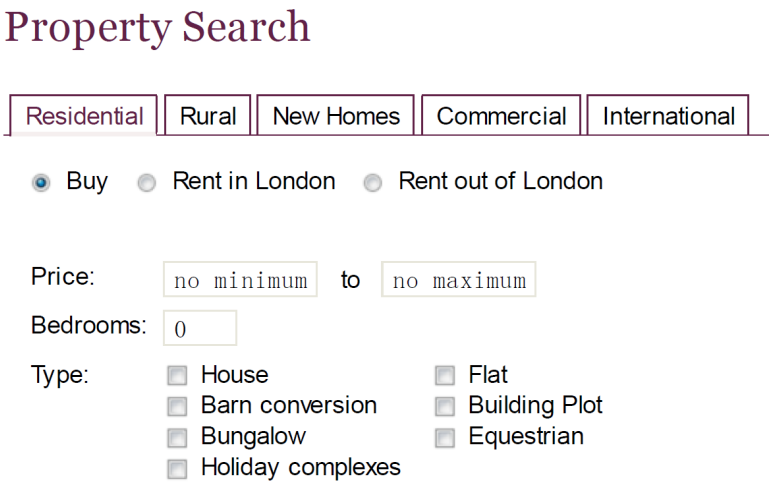
As reflected in Figure 2.2, the conceptual understanding of a form can only be partially mapped back to the physical form design (Figure 2.1), e.g., form segment to `<fieldset>`. However, without standard design practices and considering different possibilities of visual layouts, it is not a trivial task for machines to understand conceptually a form relying on its physical design.

Figure 2.3 shows the web form of *Chesterton Humberts* estate agent and its corresponding HTML code. On this form, the labels associated with the first two radio buttons are correctly defined using the *id* and *for* attribute. However, the values for the third pair do not match. But this would not affect users' understanding of the form. Figure 2.4 presents another web form from *Alexander and Co.* This form uses no `<label>` element at all. Instead, every form component is organized in a table cell for design reasons. Visually, users would associate "From" and "To" with the text input directly following them, and they would take "Price Range:" as the group label for these two text inputs.

2.2 Problem Statement

The deep web consists of data that is accessible exclusively through form-based interaction. The traditional way to access this data by manually filling HTML forms is not scalable at all with the increasing number of relevant forms and the growing size of their underlying databases. Hence, automated understanding of web forms is essential in automatically processing the deep web.

Form understanding is the process of automatically assigning semantic information to form components and structures. It includes the following tasks: (1) identifying forms and form fields, where forms are not necessarily defined by `<form>` element; (2) segmenting a form, where a form is divided into segments using structural or visual information; (3) labeling form fields and segments, i.e., associating texts with related fields or segments; (4) semantically annotating the labels, where text labels are assigned with semantic



(a)

```
<fieldset>
  <legend class="nodisplay">Purchase type</legend>
  <input id="srchBuy" class="radio" type="radio" checked="checked">
  <label for="srchBuy">Buy</label>
  <input id="srchLet" class="radio" type="radio" onclick=
  /search.dtx?advsearch=none'" name="purchasetype">
  <label for="srchLet">Rent in London</label>
  <input id="srchLet2" class="radio" type="radio" onclick=
  /search.dtx'" name="purchasetype">
  <label for="radAll">Rent out of London</label>
  <br>
 </fieldset>
```

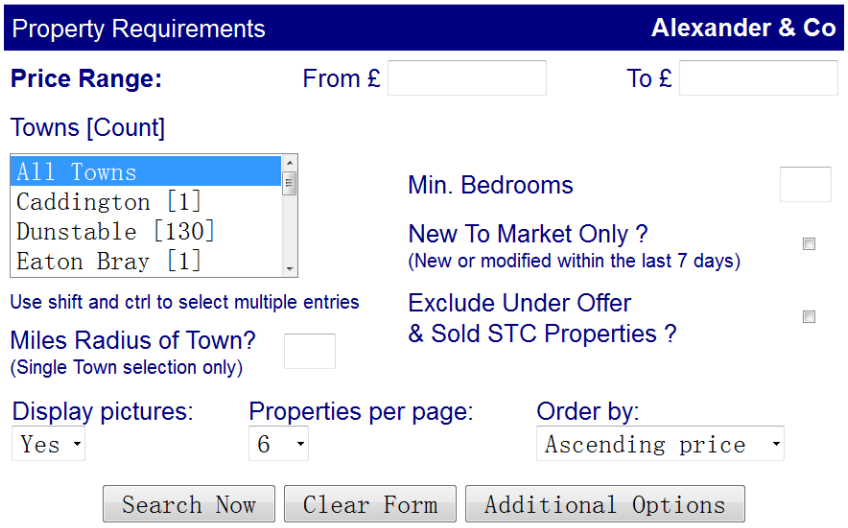
(b)

Figure 2.3: *Chesterton Humberts* Web Form.

meanings occurring in a given domain; (5) building a typed form model, which involves pruning of unnecessary segments, disambiguating among multiple annotations, etc.

As presented in the example in Section 1.2, OPAL constructs a conceptual model of a form consistent with a domain schema, which describes the form patterns occurring in a given domain. OPAL solves the form understanding problem by grouping the before mentioned sub-tasks into two problems: form labeling and form interpretation. The former identifies forms and their fields, arranges the fields into a tree, and assigns text nodes to the fields and segments as their labels. The latter aligns a form labeling with the given domain schema and thereby classifies the form fields according to the annotations on their labels.

The following sections provide formal definitions to these problems.



(a)

```
<td width="100%" colspan="2">
 <table class="SEARCH_Sub" width="100%" cellspacing="0" cellpadd
  <tbody>
   <tr>
    <td>
     <span class="bold">Price Range:</span>
    </td>
    <td align="right">
     From £
     <input type="text" value="" maxlength="12" size="10" name=
    </td>
    <td align="right">
     To £
     <input type="text" value="" maxlength="12" size="10" name=
    </td>
   </tr>
  </tbody>
 </table>
</td>
```

(b)

Figure 2.4: *Alexander and Co* Web Form.

2.2.1 Form Labeling

A **web page** is a DOM tree $P = ((U)_{U \in \mathsf{Unary}}, R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibl}}, R_{\mathsf{attribute}})$ where $(U)_{U \in \mathsf{Unary}}$ are unary type and special relations, R_{child} is the parent-child, $R_{\mathsf{next\text{-}sibl}}$ the direct next sibling, and $R_{\mathsf{attribute}}$ the attribute relation. Further XPath relations (such as $\mathsf{descendant}$) are derived from these basic relations as usual [7]. U contains relations for types as in XPath (element, text, attribute, etc.) and two kinds of relations, namely label^l for text nodes containing string l, and box^b for elements with bounding box b in the canonical rendering of the page. To normalize the representation of textual content, we represent the value of an attribute as text child node of the attribute (thus,

label^l also applies to attributes).

Definition 1. *A **form labeling** of a web page P is a tree F with mappings ϕ and ψ, such that ϕ maps the nodes of F into P. Leafs in F are mapped to form fields and inner nodes to* form segments*, that is an element grouping a set of fields. Each node n in F is also mapped to a set $\psi(n)$ of text nodes, the* labels *of n.*

A form labeling represents both the form structure and the node labeling. It has a representative for each form occurring on the given web page. The representative contains all fields and segments of the corresponding form. This allows us to distinguish multiple forms on a single page, even if no `<form>` element is present or multiple forms occur in a single `<form>` element (by spliting F into sub-trees, each representing one form on the web page, with the knowledge that each form has a submit facility). The set of text nodes mapping to each node in the form labeling can have arbitrary size, in other words, a node can be labeled with no, one, or many labels.

Definition 2. *Given a DOM tree P, the **form labeling problem** (or schema-less form understanding problem) asks for a form labeling F where for each form f in P (i) there is a node $r \in F$ such that $\phi(r)$ is a suitable representative of f and (ii) for each field e in f, there exists a leaf node $n_e \in F$ such that n_e is a descendant of r and $\phi(n_e) = e$ where $\psi(n_e)$ is a suitable label set for e.*

We call a form labeling *complete* for a web page, if, for all e, $\psi(n_e)$ contains *all* text nodes suitable as labels for e. Also note that, the suitability of a form representative $\phi(r)$ and a label set $\psi(n_e)$ cannot be defined formally, but needs to be evaluated against gold standard built by human annotators.

2.2.2 Form Interpretation

In order to define the form interpretation problem, we formalize the notion of schema and introduce a form model. We define an annotation schema that provides the necessary knowledge to interpret text nodes, a domain schema that describes domain classification and structural constraints, and a form model as a form labeling extended with type information consistent with a given domain schema.

The annotations are obtained automatically using GATE [17] application. The domain knowledge necessary for these annotations is organized in terms of gazetteers to annotate types such as price labels, and in terms of regular-expression-like rules to annotate instances of a type, such as price values. Form fields having labels with the right annotation types are classified with the corresponding domain type, e.g., price field. Segments are classified with domain types satisfying structural constraints, e.g., price segment consists of one minimum price field and one maximum price field.

Definition 3. *An* ***annotation schema*** $\Lambda = (\mathcal{A}, \sqsubset, \prec, (\mathit{isLabel}_a, \mathit{isValue}_a : a \in \mathcal{A}))$ *defines a set* $\mathcal{A}$ *of annotation types, a transitive, reflexive subclass relation* $\sqsubset$*, a transitive, irreflexive, antisymmetric precedence relation* $\prec$*, and two characteristic functions* $\mathit{isLabel}_a$ *and* $\mathit{isValue}_a$ *on text nodes for each* $a \in \mathcal{A}$*.*

For each annotation type $a \in \mathcal{A}$, we distinguish proper labels and values, with $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ as corresponding characteristic functions. Proper labels are text nodes, such as *"Price:"*, describing the field type. Texts such as *"pcm"*, i.e., per calendar month, are also considered as proper labels, restricting values applicable to the field. Values are instances of the type, representing possible values of the field, such as *"between £300 and £800"*. Hence, the following holds: $\mathsf{isLabel}_{\mathit{price}}$("Price:"), $\mathsf{isLabel}_{\mathit{price}}$("pcm"), and $\mathsf{isValue}_{\mathit{price}}$('between £300 and £800").

The $\sqsubset$ relation holds for subtypes, e.g., *postcode* $\sqsubset$ *location*, and the $\prec$ relation defines precedence on annotation types used to disambiguate competing annotations. For example, given a radio button group with *"Choose result order"* as group label, *"price"* and *"location"* are its two options associated with a radio button each. The two radio buttons, aside the direct annotation received from their option values, i.e., *price* and *location*, both of them are also annotated with *order-by*. If *order-by* $\prec$ *price* and *order-by* $\prec$ *location*, we pick *order-by* for both radio buttons.

Definition 4. *A* ***domain schema*** $\Sigma = (\Lambda, \mathcal{T}, \mathcal{C}_\Lambda, \mathcal{C}_\mathcal{T})$ *defines an annotation schema* Λ*, a set of domain types* $\mathcal{T}$*, and* $\mathcal{C}_\Lambda$ *and* $\mathcal{C}_\mathcal{T}$ *that map domain types to classification and structural constraints.*

$\mathcal{C}_\Lambda$ specifies constraints that node classification should follow. For example, $\mathcal{C}_\Lambda$(PRICE) requires an annotation *price* and prohibits *order-by* annotations for a field to be typed as PRICE. In general, such classification constraints on a domain type prohibit any annotation that precedes the annotation on the type. $\mathcal{C}_\mathcal{T}$ regulates the form model construction. For example, the structural constraint $\mathcal{C}_\mathcal{T}$(RANGE) for a RANGE segment requires a MIN and MAX field or a RANGE field, i.e., a segment on price range shall contain either min and max price fields or a single field specifying range. Structural constraints also cover cases such as $\mathcal{C}_\mathcal{T}$(PRICE-RANGE) allows no more than one occurrence of CURRENCY field

We write $S \models C$, if a constraint set C is satisfied by a set S of annotation or domain types. The empty constraint set is always satisfied.

Formally, a *form interpretation* (F, τ) is a form labeling F with a partial type-of relation τ, relating nodes in F with the types $\mathcal{T}$ of Σ. Given a node n in F, we denote with $\mathcal{A}(n) = \{a \in \mathcal{A}_\Lambda : \exists l \in \psi(n)$ with $\mathsf{isValue}_a(l)$ or $\mathsf{isLabel}_a(l)\}$ the set of annotation types associated with n via its labels, and with *child-*$\mathcal{T}(n) = \bigcup_{(n,n')\in F} \tau(n')$ the set of domain types of the children of n.

Definition 5. *A form interpretation (F, τ) is a* ***form model*** *for Σ, iff $\mathcal{A}(n) \models \mathcal{C}_\Lambda(t)$ and child-$\mathcal{T}(n) \models \mathcal{C}_\mathcal{T}(t)$ for all $n \in F$, $t \in \tau(n)$.*

Definition 6. *Given a domain schema Σ and a form labeling F, the* ***form interpretation problem*** *asks for a form model (F', τ) for Σ such that F' differs from F only in inner nodes.*

The only difference between F and F' is the internal structure of the two trees, given that both of them share the same form representatives, leaf nodes (form fields), and labels. The inner nodes (form segments) of F' may be rearranged in order to follow the form patterns prescribed by the structural constraints of Σ.

2.2.3 Form Understanding

A solution to the form interpretation problem applied to a solution for the complete form labeling is a solution to the form understanding problem. A formal definition is as follows:

Definition 7. *Given a domain schema Σ and a DOM tree P, the* ***form understanding*** *(or schema-based form understanding)* ***problem*** *asks for a form model (F, τ) of P under Σ, such that F is a solution of the complete form labeling problem for P and for each form field e in P, there is a leaf node n_e in F with $\phi(n_e) = e$ and $\tau(n_e)$ is a suitable concept from Σ for e.*

2.3 Existing Form Understanding Approaches

A recent survey on form understanding approaches [39] classifies these works into three goal-based classes: (1) to increase content visibility, such as dynamic page indexing [43, 48] and deep content repository creation [56, 58, 63], (2) to improve the intra-domain searchability, such as [31, 32, 35, 55, 64, 67], and (3) to build up knowledge as in [8]. As we consider form understanding as a stand-alone problem, a goal-based classification is unsuitable for OPAL. The following of the section categorizes the approaches based on their solutions only.

2.3.1 Flat vs. Hierarchical Approaches

Existing approaches to form understanding can be categorized based on the analysis result: flat and hierarchical approaches. Earlier approaches [37, 40, 48, 52, 56] treat web interfaces as flat collections of fields. Consequently, the main problem addressed in these approaches is field labeling. This is different from the form labeling problem defined in Section 2.2 as there is no "inner node" in a flat structure. Heuristics such as text similarity

in labels and certain HTML attributes (e.g., *name*, *id*) are frequently adopted for label identification. However, in many cases, these attributes are not in use in the current web design. Moreover, without proper structuring of the fields, even correct labeling can be miss-leading, as in the example of *order-by* radio button group with *price* and *location*. To summarize, a flat representation of form fields loses sight of the semantic relationships between them which are necessary in form understanding.

Different from the flat representation, some other approaches [19, 33, 66] model web interfaces hierarchically, i.e., with a tree-like structure. These approaches rely on assumptions on human form understanding and the corresponding web form design principles. For example, some of these rules are stated explicitly in [19]. [66] based their heuristics on common design patterns.

OPAL, as focusing on the semantic understanding of a web form, falls into the hierarchical category with its tree-representation of the form model.

2.3.2 Rule and Heuristic vs. Machine Learning Approaches

Based on the strategies used, existing approaches roughly fall into two categories: rule and heuristic approaches, such as MetaQuerier [68], ExQ [66], and SchemaTree [19], and machine learning approaches, such as LabelEx [52] and HMM [38].

Rule and Heuristic Approaches

The rule and heuristic based approaches mostly rely on human observations on web appearance and web design.

MetaQuerier [68] follows the observation that forms "seem to reveal some "concerted structure", by sharing common building blocks" and presupposes the existence of a *hidden syntax*, shared by most web forms, which "connects semantics to presentations". To uncover this hidden syntax, MetaQuerier treats form understanding as a parsing problem, interpreting the page as a sequence of "atomic visual elements", each carrying a number of attributes, e.g., its bounding box. This formalization of "hidden syntax" into a visual language is described in a *2P grammar* that is based on common patterns and relative preferences between these patterns. It provides a declarative description of the phenomena commonly encountered in web forms. This approach is the most closely related in the spirit to OPAL, in the sense that it encodes the common patterns. In a study covering 150 forms, MetaQuerier adopts 21 mainly visual-based design patters. In contrast, OPAL achieves near perfect accuracy with only 6 generic patterns by combining visual, structural, and textual features during its domain-independent analysis.

SchemaTree [19] is mainly visual-based except for two HTML attributes (*tabindex* and *for*) that reveal grouping and labeling information. It is designed to comply with

nine observations formulated in purely visual terms, stating e.g., that query interfaces are organized top-down and left-to-right, that all labels of the members of a group have the same text-style, or that the label of a group have different text-style from that of its member groups. SchemaTree renders the form and extracts a list of tokens from the embedded browser. These tokens are either text, field, or image tokens, each associated with its bounding box. After parsing, SchemaTree builds two trees, the field tree and the text tree. The field tree is constructed in a bottom-up fashion: First the fields are ordered in semantic order following the reading order of a human user. Second, all fields between two so-called inflection points are grouped together. An inflection point marks a change in the reading direction, e.g., when the reader reaches the end of a line and returns to the beginning of the next line. The text tree is constructed from a clustering of the available text tokens into groups of similar style. Each text token is associated with a scope that expands downwards to the right until reaching neighboring text tokens. Then, the text tokens of the smallest group covering the entire form are taken as children of the root of the text tree. The procedure continues recursively for each new child. As results, each field is associated with an ordered list of candidate labels: Each label whose bounding box intersects with the top, left, right, or bottom neighborhood of a field becomes a candidate label of this field, where the neighborhoods do not include diagonal spaces and are bounded by neighboring fields. The candidate ordering is given by the distance of the label to the field. The field and text tree are used to construct the *schema tree*. First, all explicit label assignments via the `for`-attribute are fixed. Then, SchemaTree iteratively assigns labels to all sibling sets in the field tree, such that all siblings receive labels with the same style and orientation, i.e., they are consistently positioned at the same side of the fields. Although, it achieves better accuracy than most existing approaches (except for OPAL), such monolithic design makes it difficult for maintenance or modification when the "observations" change with the web design practices.

ExQ [66] is also a visual-based approach adopting features such as a bias for the top-left located labels. It works in two phases, (1) it arranges the fields of the form in a tree (unlabeled), and (2) it assigns labels to both the individual fields at the tree leaves and the form segments at the inner nodes. For both, only or primarily visual features are used. Though that insulates the approach from the often messy HTML structure, it also misses those cases where the form is already well-structured in HTML. The tree construction is based on spatial clustering of the fields (or more specifically, attribute blocks which may contain multiple fields such as radio button groups). It explores topological relations between tree nodes, e.g., containment or disjointness, 4-way neighborhood relations, and alignment relations. During the label assignment process, ExQ uses *annotation blocks* which are the bounding boxes around a label and the labeled elements. Guided by a

number of assumptions, such as that annotation blocks do not overlap, ExQ processes the element tree in a bottom up manner and assigns labels to all elements in a group simultaneously in each step. The labeling process can be summarized into three steps: (i) candidate labels are generated for each form node, such that the box containing the label and all fields of the form node does not overlap with any other label or field. (ii) the candidate labels are pruned based on distance and direction from the field, favoring left and top direction. (iii) final labels are selected among the remaining candidates according to a number of heuristics: if a field has only one candidate label, the label is assigned to it, and second, if a remaining label has only a single field, it is assigned to it. For the remaining cases, labels that are not below or to the right of the corresponding fields are preferred.

Different from the three visual-based approaches, Wise-iExtractor [33] is implemented in two phases: It performs an *attribute extraction* and a separate *attribute analysis* which aims at revealing "hidden" meta-information. In this process, Wise-iExtractor distinguishes *logical attributes* which are represented by *physical elements* on the form. The attribute extraction tokenizes the form to obtain an *interface expressions (IEXP)*, distinguishing text fragments, form elements, and delimiters, such as line breaks. Based on this IEXP, the elements forming a single attribute are grouped together. To this end, Wise-iExtractor computes the *association weight* between any given element and the labels in the same line and the two preceding lines and assigns labels accordingly. The association weight is computed with heuristics that exploit ending colons, similarities in the HTML name attributes of elements and labels, and the distance between element and label. During *attribute analysis*, Wise-iExtractor determines the domain values, the default value, and the unit of form elements. It also groups together the elements forming a single logical attribute and analyzes their relationships: for example, they are classified to be in one out of *four relation types,* namely range (e.g. from, to), part (e.g. first and last name), group (e.g. radio buttons), or constraint (e.g. exact match required).

Machine Learning Approaches

Machine learning approaches offer an interesting alternative to rule based tools, like OPAL. As they do not require the explicit rules or heuristics, they can be adapted to specific domains by supplying a training set with forms chosen within a single domain.

LabelEx [52] is a machine learning approach for pure label assignment without form segmentation: (1) It generates field-label candidate associations and extracts features for each such candidate association. (2) It either learns or applies two classifier stages to prune these candidates, and (3) It selects the final associations in a reconciliation step. For the candidate generation, LabelEx associates to a field all those labels which are

located within a rectangular neighborhood around the field in question. To take dynamic form elements into account, it simulates user actions by triggering all event handlers in the form. For the machine learning phase, the features for each field-label association include their HTML markup, such as its type, i.e., checkbox or button, their visual appearance (e.g., font size and shape) and spatial features, such as alignment, their relative position, or a normalized distance measure. This phase consists of a naive Bayes classifier and a subsequent decision tree, where the former prunes away false positives and the latter is used to select an association among the candidates. In the final reconciliation step, the returned associations are supplemented, given that a field does not occur in any selected mapping. The final label for a field is chosen to maximize a score, computed from the distance between field and the domain relevance of the terms occurring in the label. The relevance of a term is determined by the frequency of this term on forms from the domain, distinguishing isolated occurrences and occurrences within phrases. As mentioned, this approach doex not consider field groups, i.e., form segmentation, and thus misses the semantic information hidden behind such groups.

HMM [38] consists of two hidden Markov models (HMM) to model an "artificial web designer". The employed HMMs are essentially finite state machines with probabilistic transitions and output symbols. The first HMM, called T-HMM (Tagging-HMM), has four states, representing attribute tags, search query operators (e.g. "exact match" or "fuzzy match"), input fields, and miscellaneous texts. The second HMM, named S-HMM (Segmentation HMM), has also four states, to start or end a segment, and to stay inside or outside, respectively. The output symbols for both HMMs are the same, including e.g., string, string with colon, parenthesized string, text area, or selection list. Moreover, HMM also uses predefined knowledge on typical terms in forms, for example, strings to describe ranges, i.e., "between", "min" (and other strings indicating the beginning of a range), and "max" (and other strings for the end of a range). During web form analysis, the HMMs are used to explain the phenomena observed on the page: The two state sequences most likely producing the given web form are taken as explanation of the occurring elements. There is no visual feature adopted during the analysis.

Although the machine learning approaches can be adapted to a specific domain by using domain-specific training data, the evaluation in [38] shows little effect of such domain-based training, where a biological domain training set outperforms domain-specific training set in three out of four domains excluding the biological domain itself.

2.3.3 Probe based approaches

The approaches mentioned in Sections 2.3.1 and 2.3.2 conduct their analysis based purely on information available on the web forms. Alternatively, there is also an indirect route for

semantic form understanding, i.e., by querying through the forms and analyzing the query results, in other words, by incorporating a specific set of observations and assumptions on result pages. For example, the work in [48] determines whether a form element is a "binding" or a "free" input by generating the result pages. [64] performs text-label assignment by "query probing". [56] derives the domain of form element values using form submissions. However, form analysis is not the focus of these approaches. Also, since our approach does not involve any result page analysis, we do not address them in detail.

2.3.4 About Form Filling

One direct application of form understanding is to assist form filling. Most frequently seen are the auto-fill toolbars for web browsers (e.g., Mozilla Firefox Autofill add-on,[1] Google Toolbar Autofill[2]). Most of them ask users to fill a predefined form (i.e., master form), and fill in targeted forms with the provided information. This is generally completed through a text analysis with at best syntactical string matching. Moreover, they focus mostly on personal information (e.g., names, contacts), although the Firefox add-on allows user extension of pre-configured fields. Instead of using a master form, [61, 62] fill forms by analyzing free text inputs.

Other than the idea of master form, existing works also address the problem through analyzing previously filled form values. For example, the Safari browser has an auto fill feature that reuses data from previously filled forms but through a simple string matching of field names. Closest to the spirit of OPAL, [2] fills web forms by connecting fields at the conceptual level, but with WordNet [54] instead of proper annotations. Furthermore, OPAL produces a much richer form model that is verified against a domain schema.

2.4 Discussion

Not considering the probing-based approaches, form understanding is an empirical problem: any approach for form understanding must rely on assumptions on human form understanding and corresponding design principles. Most of these rules describe how logical attributes are mapped onto physical form elements. However, existing approaches suffer from a number of limitations.

HMM focuses only on token streams encoding only textual content and field types. Though it relies on the meaning of specific strings, such as "between", "min", or "max", it does not use explicitly a domain knowledge—aside implicitly adapting to certain domains

[1] `autofillforms.mozdev.org`
[2] `toolbar.google.com`

by selecting domain specific training sets, which proves to be of little contribution by its own evaluation.

ExQ uses a hierarchical segmentation model and favors for left and top directions in the visual labeling heuristics. However, it is limited by using only visual information, disregarding most structural clues. Furthermore, it uses no domain information and thus can not easily deal with over- or under-segmentation.

LabelEx does not consider the form segmentation and is therefore unable to describe segments of semantically related fields or to align fields and labels based on the group structure. Furthermore, segment labels may be wrongly assigned as field labels. It does not adopt domain information though it can be trained with a domain specific training set.

Both ExQ and LabelEx encode explicitly assumptions on alignment and spatial distance of fields and labels into their approaches, which leaves potential disruption on the form analysis results if these hard-encoded values change.

SchemaTree's approach exploits only the visual structure of a form and derives heuristics that comply with nine observations following left-to-right human reading direction. However, it encodes its rules into a sophisticated algorithm (field and text tree merging), making it rather inaccessible for adaptation, where many highly reliable structural hints are ignored.

With the *2P grammar* discussed in [68], the formalization as a grammar does not lend itself easily to the integration of the observations (i.e. facts) with domain-specific ontological knowledge. It is also unclear how to contextualize grammar rules so as to select which set of rules best applies to a given form.

ODE [60], as mentioned in the introduction as a web data extraction approach, although uses domain knowledge, it is combined with heuristics that focus on one dominant design pattern or sets of clues.

To summarize,

(1) Most of existing approaches show a tendency towards sophisticated heuristics using a single class of features, e.g., purely visual [19] or textural with field type features [38].

(2) In order to cover all their observations, these approaches tend to define the heuristics into either too many "rules" or too sophisticated algorithms. Additionaly, they encode into their solutions the derived assumptions, e.g., the spatial distance [52, 66], or token class [38]. Such hard-coding lacks the flexibility in adaptation.

(3) Most approaches are domain-independent and focused on coverage, rather than accuracy, i.e., they aim to deal with a large number of forms from many domains, and

thus are limited to observations that hold for forms across all domains. This limitation is acknowledged in [38, 52], but only through domain specific training data. Yet in [38], it is mentioned that on top of the set of generic design rules underlying all domains, specific domains parameterize these design patterns in ways different from other domains.

(4) Many approaches proceed on the assumption that web forms are identified correctly or that only web forms in HTML `<form>` elements are considered [33, 38]. Others do not even mention how a web form is identified. However, this assumption does not scale to the deep web size, especially with the rising of scripting language where the `<form>` element no longer serves its original purpose of representing forms.

OPAL is designed with these limitations in mind. First we integrate different classes of features for "domain-independent" analysis, e.g. all textural, structural and visual observations, and we define the preference of the observations so that results retrieved using certain classes of observations are more "trusted" than the others. Second, we incorporate domain knowledge explicitly by adopting a domain ontology and introduce a phenomenology layer in between the domain independent and dependent parts. "Phenomenology" explicitly describes the mapping from physical form elements to logical attributes in a domain. With domain knowledge, over or under analyzed form can be easily pruned or extended to give a more accurate form model. Thirdly, identification of form areas and handling of multiple forms are explicitly dealt with. Finally, we use declarative rules for implementation to allow easy modification and natural access to the domain knowledge.

Chapter 3

OPAL System Overview

To solve the form understanding problem, OPAL is divided into two parts, a domain-independent part to address the form labeling problem and a domain-aware part to handle the form interpretation problem (Figure 3.1). At field scope, only fields and their immediate neighbourhood are considered and thus only the HTML DOM serves as input. At segment scope, form fields are grouped based on structural similarities, and these groups are arranged into a segment tree. In each segment, text nodes are assigned to fields according to their interleaving pattern. At layout scope, we exploit the visual information (primarily the CSS boxes) to assign visually related texts and fields. At domain scope, OPAL turns the labeling model into a form model, relying on domain-specific annotations of the textual contents occurring in labels and suggested field values. The form model is obtained from the consolidated classification of the identified fields and segments. The domain-independent scopes increasingly enhance the labeling result, and are combined at domain scope, where OPAL classifies form fields and repairs the form model with parameterizable domain knowledge.

3.1 Multi-scope Domain-aware Form Understanding

OPAL proceeds with a domain-independent multi-scope approach that incrementally constructs a form labeling combining textual, structural, and visual features. As presented in Figure 3.1, the three classes of features are encoded into three domain independent scopes: field, segment, and layout scope. Each scope builds on the partial form labeling of the previous scope and uses the information from the additional input to find labels for previously unlabeled fields (or segments). Only the segment scope contributes to the form tree, whereas field and layout scope only labels the fields. Coming to the domain-dependent analysis, i.e., at domain scope, OPAL extends the labeling with domain-specific annotations and turns the form labeling into a form model that is consistent with a given domain schema.

3.1.1 Domain Independent Form Labeling

Starting with field scope, the labeling analysis assigns texts to fields according to local structural properties that exploit (i) form design techniques provided by HTML, such as the *for* attribute of a `<label>` element, and (ii) structural hints involving only single fields, such as the greatest unique ancestor heuristic, which associates an individual field with all those texts that share a common ancestor with this field but no other field. These heuristics construct the basic units of a form, i.e., form elements, each consisting of a field and its associated labels.

At segment scope, OPAL groups elements into segments using structural similarities. For example, if adjacent fields share the same *class* attribute value, or if adjacent, similarly styled fields are of same type (e.g. text input), then these fields form a potential segment. However, since the potential segment is derived from structural information, this heuristic requires the existence of a HTML element to act as segment root, determined as common ancestor of all fields in the potential segment. If such a segment root is found, then we consider the segment valid. As a result, OPAL obtains a projection of the DOM tree by organizing the form elements in a hierarchy of segments.

OPAL then extends the existing form labeling produced at the field scope with the segment information by distributing labels to fields in the same segment. For each segment, we construct a field list and a text set list. The former contains the fields listed in document order. The latter is obtained by firstly arranging all text nodes in document order and then partitioning the list according to the interleaved fields. The members of

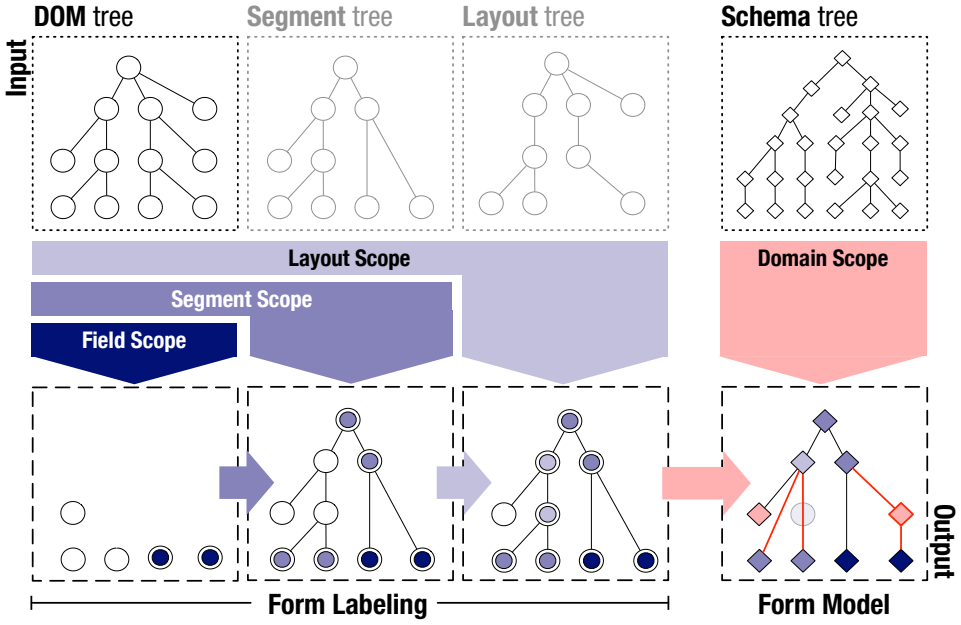


Figure 3.1: OPAL Overview

one list is then assigned to the other in a one-to-one manner. The assignment takes into account the labeling results from field scope by preserving the already matched pairs and handling the sub-lists between these pairs recursively.

Besides field labels, segments are also associated with labels. This is done by treating each segment as a single node (by taking the segment root node) and applying similar techniques as we used on individual fields at field scope on the segment root nodes. Segment label also preserves any text left-overs from the interleaving procedure.

The first two scopes are purely structural and focus on individual fields and their position in the parent segments only. However, these two scopes are insufficient to identify all label associations. Hence, we enlarge the analysis to the entire page.

At layout scope, OPAL analyzes the visual layout of each form (or the entire page if no `<form>` element occurs on the page), and considers only texts in the top-left direction of each yet unlabeled field. Labels are selected considering overshadowing cases, which in general define the visibility of the texts through the analysis of overlapping area of two fields. In the top-left region of a field, visible texts appear in the area not overlapping with the top-left region of other fields atop the current field. Texts are also visible if they are the right-most in the overlapping area of the field's direct left neighbor. All other texts remain invisible.

3.1.2 Domain Aware Form Interpretation

The three domain-independent scopes together deal with the form labeling problem, i.e., they arrange the fields into a tree model and label each node (whenever possible) with a set of texts. Next, OPAL proceeds with form interpretation.

At domain scope, OPAL builds the domain-specific form model: (i) it extends the labeling model by assigning the textual content of proper labels and values with domain-specific annotations; (ii) it classifies fields and segments based on the associated annotations following the classification constraints; and finally, (iii) it resolves any disambiguation in the classification and repairs any violation of structural constraints.

Both classification and structural constraints are specified in the domain schema with OPAL-TL. OPAL-TL allows easy querying of labelings and annotations. It also enables template design, where generic templates of constraints can be simply instantiated for concrete domain types. One example of these is the range template for defining range values. It can be instantiated for price in many commercial related domains, for room number in housing related domains, etc.

3.2 An Imperative Implementation

Following the OPAL approach, we firstly implement a prototype in Java. The prototype covers the structural scopes, i.e., field and segment scope, plus a thin layer of domain classification and verification.

3.2.1 The Java Implementation

HTMLUNIT [27] serves as underlying browser and supplies all necessary information to the subsequent analysis, extracted from the DOM tree of the parsed web pages. Starting with a URL of a web page, we first extract the DOM tree of the page. We then retrieve form-related HTML elements, i.e., `<form>`, `<input>`, `<select>`, `<label>`, `<button>`, and `<textArea>`.

In the actual web design, many forms do not follow the applicable standards, which brings difficulty to the form analysis. For example, there are cases without a `<form>` element, containing fields which occur outside `<form>` elements, containing multiple conceptual forms within a single `<form>` element, or using `<a>` links for submission instead of buttons. Besides, web designers often use scripting languages to develop interactive forms, e.g., values selected by users change the appearance of the forms.

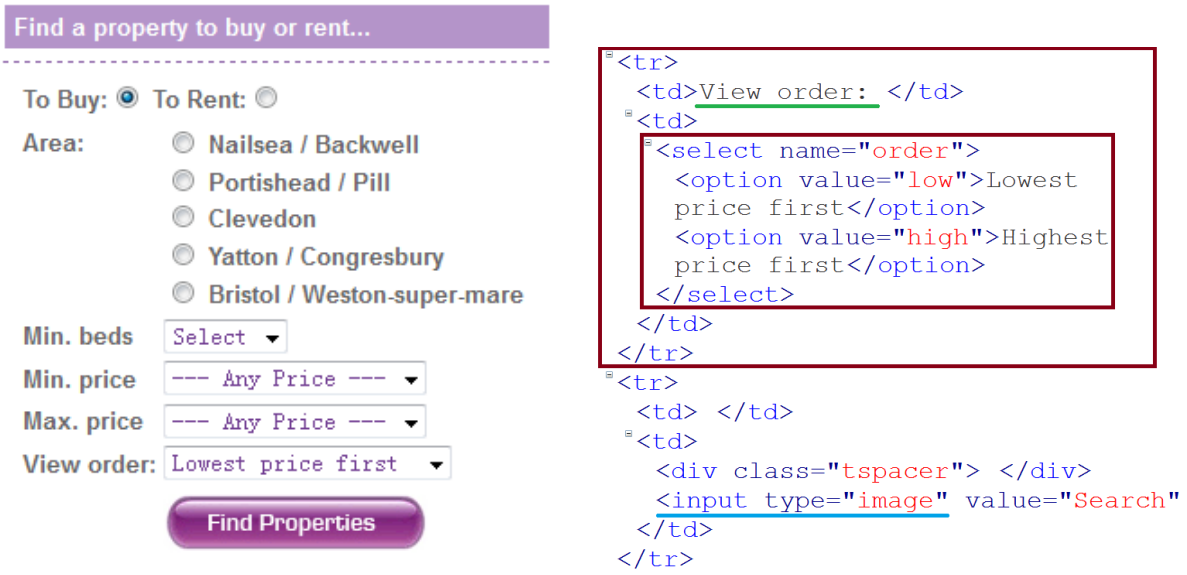
We define Java classes Element, Segment, Label as basic concept classes, representing form fields, groups of fields, and their labels correspondingly. An instance of Element is also considered as a special Segment which contains only one field. An Element is associated with a list of Labels and carries a boolean value indicating whether the field is a button. A Segment contains a list of Segments (and Elements), which describes the part-of relationship between segments. Every Segment is also associated with a list of Labels. It maintains the node corresponding to the segment root, and an indicator for whether the group consists buttons only. A Label represents a text node that is assigned to a field or segment, and also stores the canonical path to the node, and the text content.

This prototype implements the field and segment scope by translating the textural and structural observations into six sequentially executed steps. Using *Heritage Estate Agent*[1] in Figure 3.2 as a running example, we illustrate the implementation of our prototype.

(a) Matching *for* attributes of `<label>` elements with *id* attributes of form fields.

Given a web page, if a `<label>` element occurs, and its *for* attribute does match the *id* of a form field, then the corresponding association is almost always correct. But in many cases, `<label>` elements are misused by assigning non-existing id values to the *for* attribute. In such cases we treat the `<label>` elements as ordinary text nodes.

[1] `http://www.heritage4homes.co.uk/`



(a) (b)

Figure 3.2: *Heritage Estate Agent* Web Form.

On the *Heritage* form, there is no such straightforward clues using `<label>` elements, and hence this rule does not apply.

(b) Marking special form fields, i.e., button fields.

Button fields are commonly `<input>` elements with attribute *type* submit, reset, or image, and `<button>` elements with submit or reset type. It is necessary to treat button fields specially, since most forms do not explicitly label form buttons. The existence of possible fields without labels affect the field-text pattern analysis in the following steps. The *Heritage* form uses an image `<input>` as the submit button. Special cases, where the submit facility is implemented using other HTML elements with javascript, are ignored in the prototype.

(c) Labeling fields with texts satisfying the greatest unique ancestor condition.

Given a field, we traverse its ancestors in a bottom-up fashion and identify each ancestor that contains no other field. From these unique ancestors, we choose the one closest to the document root as the field's greatest unique ancestor. For each field, we assign to it all texts in the sub-tree rooted at its greatest unique ancestor. For example, the "View order" drop-down list is highlighted in the corresponding DOM tree presented in Figure 3.2(b). The inner rectangle draws out the `<select>` element and the outer rectangle shows its greatest unique ancestor. As a result, "View order:" is assigned to the field. Labels are similarly assigned to all other drop-down lists.

(d) Constructing form segmentation based on field similarities.

Similar fields must (i) be consecutive, (ii) have either the same *class* attribute value, or share the same *type* and *style* values, and (iii) share a common ancestor which has no other descendant fields. The third condition makes sure that segmentation result we derived from HTML similarities remains consistent with the HTML structure. Fields which are mutually similar are grouped into a single segment even if they are further subdivided in the HTML encoding.

In the second condition, consistency of the *type* attributes for the form fields alone is insufficient to guarantee field similarity, because most standalone fields are represented with a single type of form field. Here, standalone means fields which belong directly to the entire form, not being part of a sub-segment with other semantically related fields. For example, on the form in Figure 3.2, although there are four "adjacent" drop-down menus, only the two price fields are semantically related.

The system identifies on the *Heritage* form the five radio buttons right to "Area" satisfying all three requirements and hence we group them into a segment.

(e) Exploiting the HTML structure.

The similarity segmentation is insufficient to build a complete hierarchy of the form. Thus, we supplement the hierarchy with inner DOM nodes as further segments: first, we take all similarity-based segments and the remaining form fields as the initial segments, and second, we recursively add DOM nodes containing more than one segments until reaching a single node that covers all. The DOM nodes selected at each round are the first nodes encountered that covers more then one segments at the lowest level in the DOM tree. This tree construction is achieved in a bottom-up manner. For example (Figure 3.2), the first two radio buttons are missed by the similarity comparison but are grouped into a segment by this rule.

(f) Assigning labels following text-field interleaving patterns.

For each segment containing only fields as children, the system builds a list containing all fields in the segment, and a list containing sets of texts that are interleaved by the fields. The text sets are assigned to the fields on a one-to-one basis. For example, the segment containing the first two radio buttons have a text-field pattern while the five radio buttons have a field-text pattern. In the case where there are more text sets than fields, the system preserves the information by associating the texts with the group, and actually there can be at most one more text set than the fields with the text sets construction. In the case where there are more fields, the process will not take place.

No visual feature is encoded in the prototype.

We manually build a sample gazetteer for the UK real estate domain in order to integrate our domain-independent analysis with domain knowledge.

The gazetteer contains a small collection of terms for common domain types in the UK real estate domain, e.g., location, and is stored in text files. In order to link the gazetteer to the form understanding result, the system provides an interface that represents both the tree structure (mainly the part-of relation) of the form and the assigned labels of the fields. We adopts a simple comparison of strings between labels and gazetteered terms and classify the fields accordingly. The only verification implemented in the prototype is to check whether the form is a real estate form by checking occurrences of location or price fields together with submit buttons. If a proper button element does not occur, the system tries to identify links with both event handler and terms related to "submit" as the submit button.

3.2.2 More UK Real Estate Forms

This section presents a few examples of forms from the UK real estate domain with a walk-through of how the form structures and text labels are analyzed.

Finders Keepers.

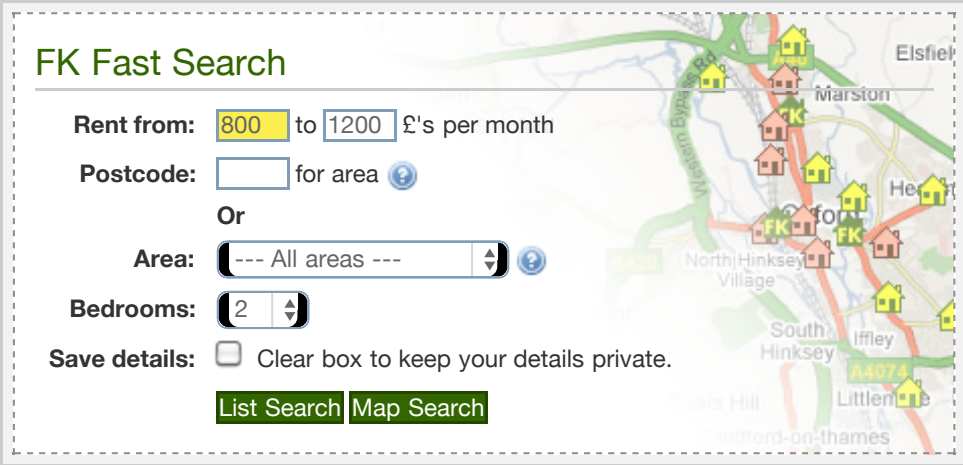


Figure 3.3: Form on Finders Keepers

The Finders Keepers[2] web form (Figure 3.3) contains two input fields for the price range with interleaved labels, and an input field for restricting the search to a given area determined by a postcode. It is also possible to specify the location from a drop-down list. The next input field allows the user to specify the desired number of bedrooms. At

[2] `http://www.finders.co.uk/`

the bottom, the form contains two search buttons determining the format of the results page: it is possible to return them as a list of results or as points on an active map.

From the structural point of view, the form is organized as a list of HTML `<div>` elements, one for each row in the form. This structure does not reflect the actual logical structure: The rows containing the postcode text box and area search list form together a logical segment as they represent alternative ways to restrict the search base on location. Also, the row containing the term "or" (and represented as a `<p>` element in the HTML tree) carries the information that the two fields are to be used exclusively.

The analysis of the labels and the content of the drop-down lists also contributes to the semantic understanding of web forms. For example it is possible to derive that the purpose of this form is to rent a house (and not to buy one) from the terms "rent" and "per month" in the first row. This can also been inferred using the values "800" and "1200" coupled with the pound symbol in the price fields of the first row; it is reasonable to assume that these values cannot represent a sale price for any house in the domain. For the price fields we also need to recognize that the user must use the fields to specify a minimum and a maximum price (i.e., a range). This can be done by means of the labels with value "to" and "from" between the two price fields.

Vebra.

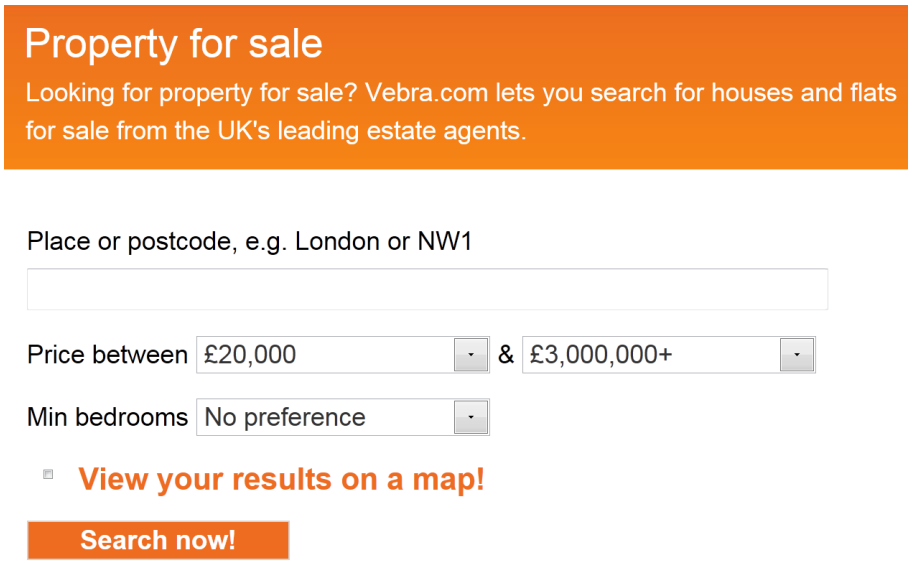


Figure 3.4: Buying Form on Vebra

The Vebra[3] form in Figure 3.4 is an example of a less sophisticated form. Each row of the form is represented by a `<p>` element in the DOM, except for the first two rows,

[3] `http://www.vebra.com/vebra/`

where the text and the input field share one single <p> element. In contrast to Finders Keepers, the location of the property can only be specified with a free text input field. The price is specified in a similar way to Finders Keepers as a range, but the values are selected from drop-down lists whose contents clearly express sale prices as it is unlikely to rent a house for 3 million pounds even in London. This can also be easily inferred by the term "for sale" in the header of the form. Same as on Finders Keepers, there is a drop-down list for bedroom number and also a means of specifying the presentation of the results page (but, through a check box). The button at the bottom of the form is easily recognized as a search button.

Holbrook Moran.

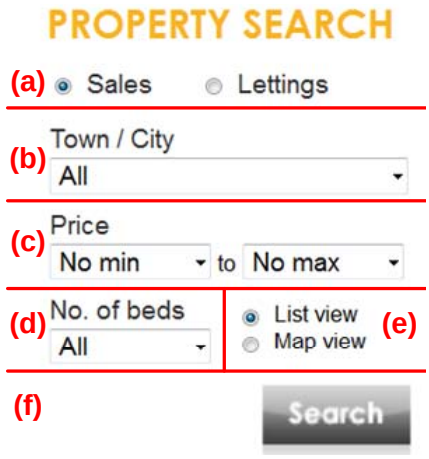


Figure 3.5: Form on Holbrook Moran

The Holbrook Moran[4] form in Figure 3.5 is an example where textual and structural features alone are not enough for a complete form understanding. Each row is encoded with a element, except for area (d) and (e), where the complete area is represented by a single element respectively.

Fields in area (d) and (e) and their corresponding labels satisfy the greatest unique ancestor rule. The form segment rule applies to fields in area (a) and (c). However, in (c), the word "price" and the two fields are in two rows appearing in separate elements. This also happens in (b). Without visual features, it is not possible to relate the texts to the fields without domain information (e.g., by values in the drop-down lists are instances of the domain types).

[4] http://www.holbrookmoran.co.uk/

3.2.3 Discussion on the OPAL approach and the Imperative Implementation

We conduct an experimental evaluation of the approach with the imperative implementation on a random sampling of 50 UK real estate websites. In order to evaluate the precision of the prototype, we annotate by hand all 50 pages. We note the number of form fields and form segments each form contains.

The evaluation results are presented in Appendix A in detail. We test the prototype in two settings: with only field and segment heuristics, and with integration of domain knowledge. We measure the percentage of fields correctly retrieved, the percentage of fields correctly labeled, and whether the prototype performed a correct segmentation.

With domain independent rules, the prototype correctly identifies 97.61% of form fields and 92.19% of form segments, and achieves a precision of 94.91% for field labeling. When further tested incorporating domain knowledge, the three figures are improved to 100% (by identifying special fields, e.g., links as submit buttons), 95.31% (by grouping semantically related fields), and 97.42% (by classifying based on field content annotations rather than labels) respectively.

Despite the promising results that demonstrate the efficacy of combining structural web design information with domain knowledge, the prototype also reveals disadvantages in the insufficient features being considered and the actual implementation:

(1) Pure structural analysis is inadequate for domain-independent form labeling. However, based on lessons learnt from existing visual-bases approaches, visual features alone are not robust enough as many of them may clash with the actual form appearances due to different page rendering or simply a don't-care design (e.g., same text style for the whole form). Hence, we should seek for a balance among textual, structural, and visual analysis, so that a high accuracy can be achieved without too detailed observations.

(2) The hard-coded translation of the heuristics into procedural algorithms limits the maintenability and the adaptability to new design patterns. This has been proved by many existing form understanding approaches as we discussed in the related work. An implementation with logical rules can resolve the issue.

(3) The java implementation prevents a straightforward integration with domain knowledge. It is more natural to present domain knowledge as facts (interpreted by meta rules), and this is a good suggestion to adopt query and rule language for the implementation.

Due to the limitation of our prototype, we decide to implement the OPAL approach declaratively. Both the approach and the implementation are explained in detail with Chapters 4 and 5.

Chapter 4

Domain Independent Form Labeling

OPAL adopts a three-scope (*field*, *segment*, and *layout* scope) approach in its domain-independent form labeling process. Each scope exploits a particular class of web form features, i.e., textual, structural, and visual, the combining of which enables OPAL to capture the diverse range of form design patterns. This distinguishes OPAL from previous approaches that rely on limited classes of features. The declarative implementation allows easy extension or adaptation to new observations that follow future web design trends.

The form labeling F is constructed by applying each scope in sequence to yet unlabeled fields. The scope's application order reflects the level of "confidence" in the respective heuristics: if it is possible to assign a label at a more confident scope, it is unlikely that the successive scopes perform better. This addresses the problem of competing label assignments. With the presence of a domain schema, the form labeling is extended to a form model in the domain-dependent form interpretation process (Chapter 5).

This Chapter discusses in detail the OPAL's form labeling approach, starting with a description of the logical representation of a given web page P, followed by the three domain-independent scopes, field, segment, and layout scope, in the applied sequence.

4.1 Input Preparation

4.1.1 Page Model

Before OPAL starts with its analysis, a logical data model is necessary to represent textual, structural, and visual information on the given web page as rendered by a browser. This has been introduced first in our proof-of-concept paper [23]. We represent the DOM tree of a web page in a relational format including elements, text nodes, attributes and their values. The textual content is stored in a large character object (clob) with references from the DOM nodes to the start and end positions in the clob. The clob allows efficient storage and querying of string values and later annotations for domain-aware analysis.

The rendering also includes all css attributes and bounding boxes of all HTML elements. We adopt the start-end encoding [29, 46] (an example of region labeling scheme for XML) during page parsing.

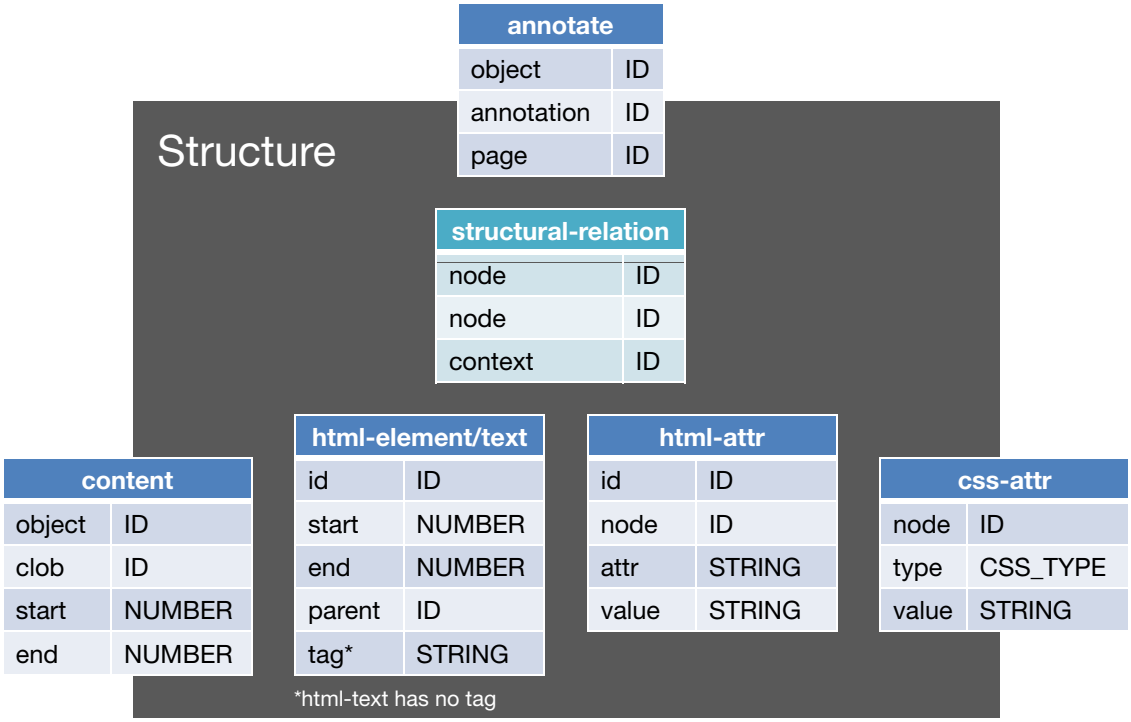


Figure 4.1: Logical model of HTML DOM structure

Structural information.

The structural information is represented in the relations `html-element`, `html-text`, and `html-attr`, see Figure 4.1. `html-element` represents an HTML element, while `html-text` encodes text nodes which have no tags. Text content is stored in `node-text` relation, which carries a reference to a text node and its actual text. In the actual implementation, an additional "doc" attribute is attached to the `html-element` and `html-text` to distinguish between different web pages.

With the start-end encoding, we extend the structural relation with all XPath [9] axes as predicates. For instance, the `descendant(Des,Anc)` axis considering both elements and text nodes are defined by comparing the start and end number of the nodes, where "_" represents a don't-care value:

```
descendant(Des,Anc):-html_element(Anc,Astart,Aend,_,_,_),
        html_element(Des,Dstart,Dend,_,_,_),
        Astart<Dstart,
        Aend>Dend.
descendant(Des,Anc):-descendantTextOf(Des,Anc).
```

```
descendantTextOf(DesT,Anc):-html_element(Anc,Astart,Aend,_,_,_),
        html_text(DesT,Dstart,Dend,_,_),
        Astart<Dstart,
        Aend>Dend.
```

Such derived XPath axes include `child`, `descendant`, `descendant-or-self`, and `following`, as defined below:

- `child(C,P)`, reads C is a child of P. It is defined by comparing the parent number referred by C with the start number of P. It takes care of both child nodes and child texts since HTML treats text nodes differently from element nodes, i.e. C can be either an HTML element or an HTML text node.
- `descendant(D,A)`, reads D is a descendant of A. It is defined by comparing the start end position of the two nodes, such that the range of A covers that of D. Similar to `child`, it handles text nodes explicitly as well.
- `descendantOrSelf(D,A)`. This relationship is satisfied if either D is a descendant of A or D and A refer to the same node.
- `following(F,X)`. F is following X if F starts after X ends, i.e. the start number of F is larger than the end number of X.

Since form labeling greatly involves texts, we also define `childTextOf(CT,P)` and `descendantTextOf(DT,P)` predicate for easy text retrieval.

Textual information.

Textual relations are presented in Figure 4.2. DOM nodes are linked by the content relation to textual information through the node id. The content relation specifies its start and end position in the corresponding clob. The two positions allow annotations crossing element borders and encode the complete nested content of the node. For ease of use, the querying result from combing `content` and `clob` is presented by a convenient relation `node-text`. As in the following example, text node t_223 has text content "Bedroom", which is a non-white value.

```
html_text(t_223,223,224,208,page1).
node_text(t_223,"Bedroom",true,page1).
```

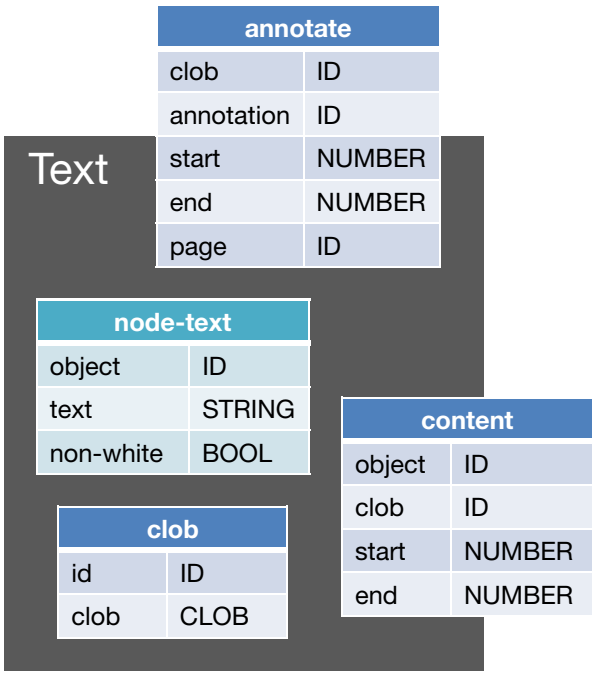


Figure 4.2: Logical model of HTML textual content

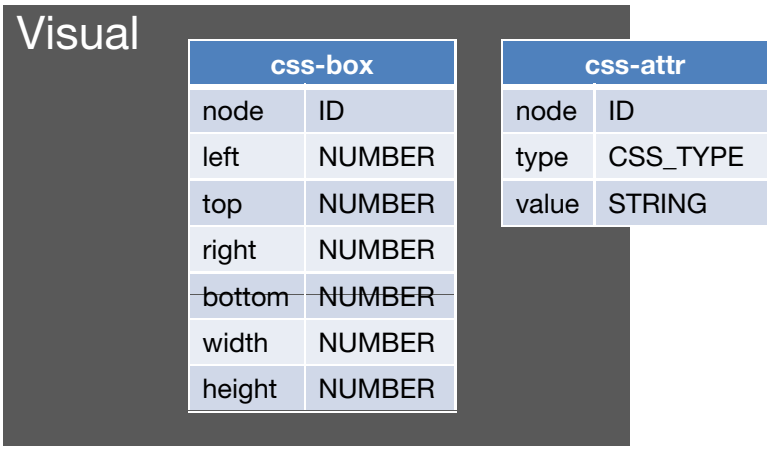


Figure 4.3: Logical model of HTML rendering

Visual information.

Figure 4.3 shows the visual relations. All css attributes [20] and element bounding boxes are extracted from the visual rendering of the web page through `css-attr` and `css-box` respectively. From these information, we also derive a number of spatial relations: `contain`, `top`, `strictTop`, `strictLeft`. Neighboring and alignment properties are further defined by these.

- `contain(X,Y)` is satisfied if X contains Y. This is done by comparing the four edges of the two boxes rendered by the browser.

- `top(X,Y)` holds if X is on top of Y. The top condition is satisfied as long as vertically more than half of the X box is on top of Y box.
- `strictTop(X,Y)` requires the bottom edge of X appears on top of Y.
- `strictLeft(X,Y)` requires the right edge of X is strictly to the left of Y.

The strict and non-strict definitions for the `top` relation are necessary considering the rendering behavior of web browsers resulted from the unformulated design of web pages and the extensive involvement of scripts.

The following example presents the `strictTop` predicate:

```
strictTop(X,Y):-box(X,Lx,Tx,Rx,Bx,Wx,Hx),
        box(Y,Ly,Ty,Ry,By,Wy,Hy),
        X!=Y, Bx<=Ty.
```

Example.

To illustrate the complete logical data model of a web page, we refer to the HTML code in Figure 4.4, and the rendering of the code is shown in Figure 4.5. It represents a `<table>` containing two `<div>` elements, with one of them having an input field and its corresponding label. The figure shows the textual and structural facts. The bounding boxes of the nodes are presented in Figure 4.6.

```
<table>
    <div>...</div>
    <div>
        <label for="price">Price:</label>
        <input type="text" id="price" tabindex="1">
    </div>
</table>
```

Figure 4.4: Page Model Example (source code)

Field visibility

OPAL's domain independent analysis is conducted on "visible" fields only. The visibility of a DOM node is determined either by its associated css attribute or by the HTML *style* attribute. In HTML there are two ways to specify the invisibility of an element, i.e., set *visibility* to "hidden", or set *display* to "none". Moreover, the invisible status of a node is inherited by all its descendants unless specified differently. To this end, we define `hiddenHtmlElement` and `hiddenHtmlText` for status check whenever necessary.

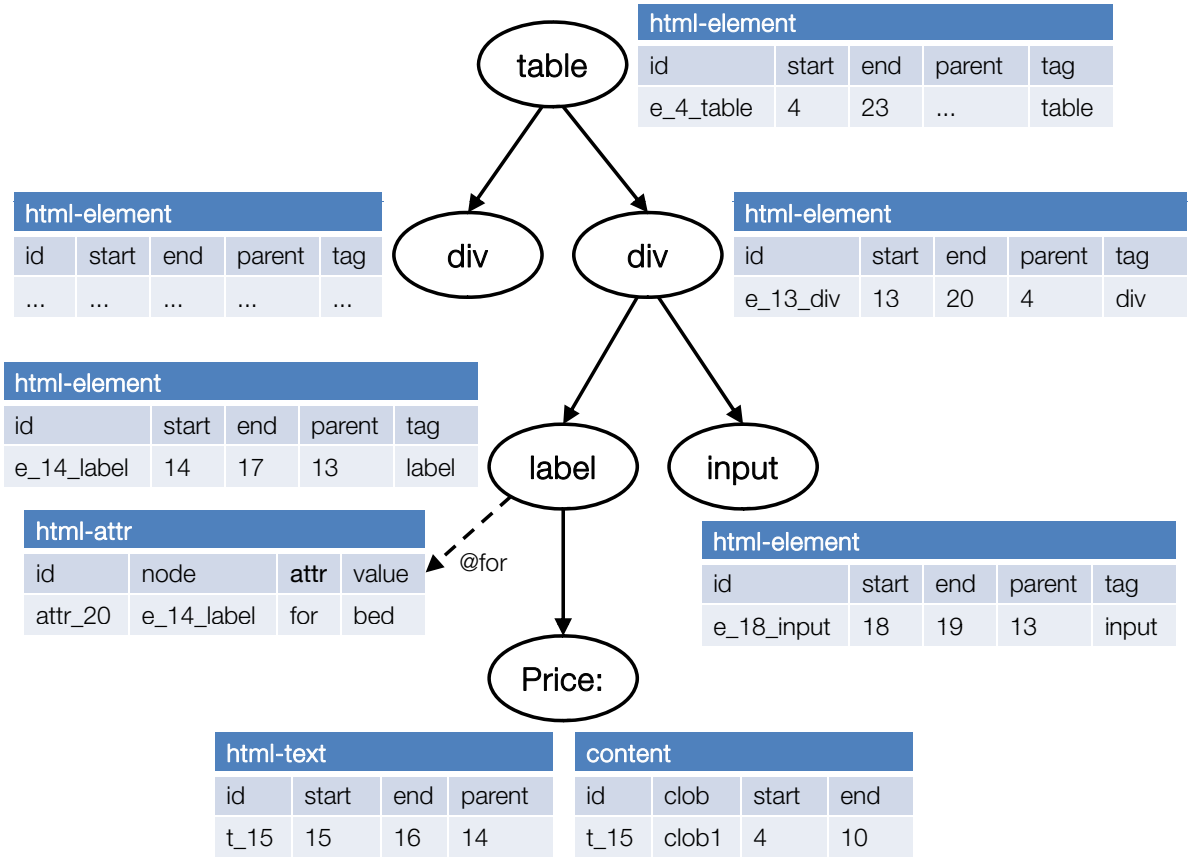


Figure 4.5: Page Model Example (structure and textual content)

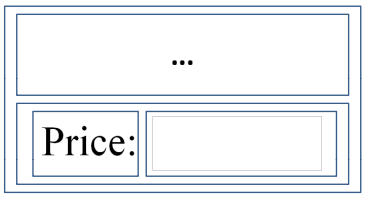


Figure 4.6: Page Model Example (rendering)

4.1.2 Annotation

GATE [17] application is used to generate all linguistic annotations and obtained information are represented as facts. These annotations mainly cover the part of domain knowledge that is organized (1) in terms of gazetteers to annotate concepts such as minimum price labels, and (2) in terms of regular-expression-like rules to annotate concepts such as price values, e.g. a number following a currency symbol.

The annotation is performed on the clobs marking the start and end position of the term being annotated. With each annotation, we arrange into facts the type of annotation, e.g., a gazetteer look-up, and all the featured annotation recognized by GATE, e.g., real estate form label for price. This is illustrated in the example provided in Section 4.1.3.

GATE annotations do not take place in the domain-independent analysis.

4.1.3 Fact examples

Figure 4.7: *Heritage Estate Agent* Web Form

This section gives an example of facts generated during web page rendering and GATE annotation. On the *Heritage* form in Figure 4.7, the first radio button in area (b) is coded in HTML as:

```
<input type="radio" value="nailsea" name="location">
```

Its representation in the page model is partially displayed below. It is an `html-element` with tag input, and its start and end number in document sequence is 320 and 321 respectively. Its parent element has the start number 319. It has attribute *name* with value "location" and attribute *type* with value "radio", etc. Its rendered box takes the area specified by the first four numbers in the box relationship. The last two figures are the width and height of the box. It also has css attribute *bottom*, *clear*, etc., as presented with the following facts:

```
html_element(e_320_input,320,321,319,input,d1).

html_attr(e_320_input_name,e_320_input,name,"location",d1)
html_attr(e_320_input_type,e_320_input,type,"radio",d1).
...
box(e_320_input,106,253,120,267,14,14).

css_attr(e_320_input,bottom,"auto").
css_attr(e_320_input,clear,"none").
```

```
10 css_attr(e_320_input,color,"rgb(0,0,0)").
   ...
```

The annotation is performed on all texts in the HTML source code. For example, "nailsea" gets the following annotations, which reveals the fact that it is a location and a member of the district_county_etc gazetteer.

```
annotation(attrclob_d1_5560,attrclob_d1,80,87,"Nailsea").
2 annotationFeature(attrclob_d1_5560,"majorType","location").
annotationFeature(attrclob_d1_5560,"minorType","district_county_etc").
```

Below is another example, "Min. price" in area (d) is annotated as a real estate form label:

```
annotation(elclob_d1_1707,elclob_d1,2028,2038,"Min. price").
2 annotationFeature(elclob_d1_1707,"modifier","min").
annotationFeature(elclob_d1_1707,"minorType","price").
4 annotationFeature(elclob_d1_1707,"majorType","reform.label").
```

4.2 Field Scope

The analysis at field scope starts with a focus on individual fields. OPAL looks for form labeling that satisfies the structural relations required at this scope as illustrated in Algorithm 1. OPAL firstly colours (lines 1–6) all nodes in P that are ancestors of a field and do not have other field descendants with orange. The least ancestor that violates the condition is coloured red. It then identifies (line 7–10) all form fields and initialises the form labeling F with one leaf node for each such field. Finally for form labeling, it considers (lines 11–12) explicit HTML `label` elements with *direct reference* to a form field, and it labels (lines 13–16) each field f with all text nodes t whose *least common ancestor* with f has no other form field as descendant. Each value v of a field f (in `select`, `input`, or `textarea` element) also becomes a label for f, as the least common ancestor of f and v is f.

From the algorithm, it is clear that OPAL bases its labeling analysis on two heuristics at field scope.

(1) HTML `<label>` element

This heuristic spots the direct references provided in an HTML document. It matches the value of *for* attribute of the `<label>` element to an HTML element that has the same value for its *id* attribute. This reference explicitly assigns the text node associated to the `<label>` element to a field in the form. See Figure 4.4 for an example where the value "price" is mapped between the *id* and *for* attributes. Although, such labeling for form fields is strongly encouraged for web accessibility [16], this is seldom

Algorithm 1: `FieldScopeLabelling`(*DOM P*)

```
1  foreach field f in P do
2      n ← f;
3      while n has a parent do
4          if n is already coloured then colour n red; break;
5          colour n orange;
6          n ← parent of n;
7  F ← empty form labeling ;
8  foreach field f in P do
9      n ← new leaf node in F;
10     φ(n) ← f;
11     if ∃l ∈ P with for attribute referencing f then
12         assign all text node descendants of l as labels to n ;
13     p ← parent of f;
14     while p not coloured red do
15         f ← p; p ← parent of f;
16     assign all text node descendants of f as labels to n ;
```

used properly by developers: either no `<label>` element presents at all or no matching *id* value can be found.

Despite the inappropriate usage of the `<label>` element, this heuristic produces reliable labeling results since it is unlikely that the developer mistakenly assigned the label to the wrong field. Among the near 800 forms we have explored so far, there are no such cases.

In OPAL, retrieving this association is performed by a declarative rule that joins the `html-element`, `html-attr`, and `field` relations, where `field` is the union of all types of HTML elements that are form controls: `hasBasicLabel(F,L,T)` holds if a field F takes a text node L as its label with text content T, with L being the child node of a `<label>` element which has a for attribute matching the id of F. The name `hasBasicLabel` is used due to the stratification requirement at later steps.

```
hasBasicLabel(F,L,T) :- field(F),
  html_attr(_,F,id,ID,_),
  html_element(N,_,_,_,label,_),
  html_attr(_,N,for,ID,_),
  child(L,N),
  node_text(L,T,true).
```

(2) Least common ancestor

Given a form field f, OPAL looks for a least common ancestor n that (i) is an ancestor of f, (ii) is at a minimum distance (in terms of depth) from the document root and,

(iii) does not contain any other form field as its descendant.

In other words, n is the root of the sub-tree T that contains only f as a form field. OPAL assigns all text nodes in T to the field f. With this heuristic, OPAL allows multiple labels to be assigned to the same field. Take the form from *Ideal Property Services* (Figure 4.8) as an example. The highlighted area of the form in (a) well illustrates the idea. Its corresponding HTML code is displayed in (b). The `<select>` element is the only form field in the `<tr>` element. As a result of applying this heuristic, both "With at least" and "Bedrooms" are assigned to the field as labels. The semantic meaning is imcomplete without either one of the texts, as the two working together reveal the information that the drop-down list is for minimum bedroom selection.

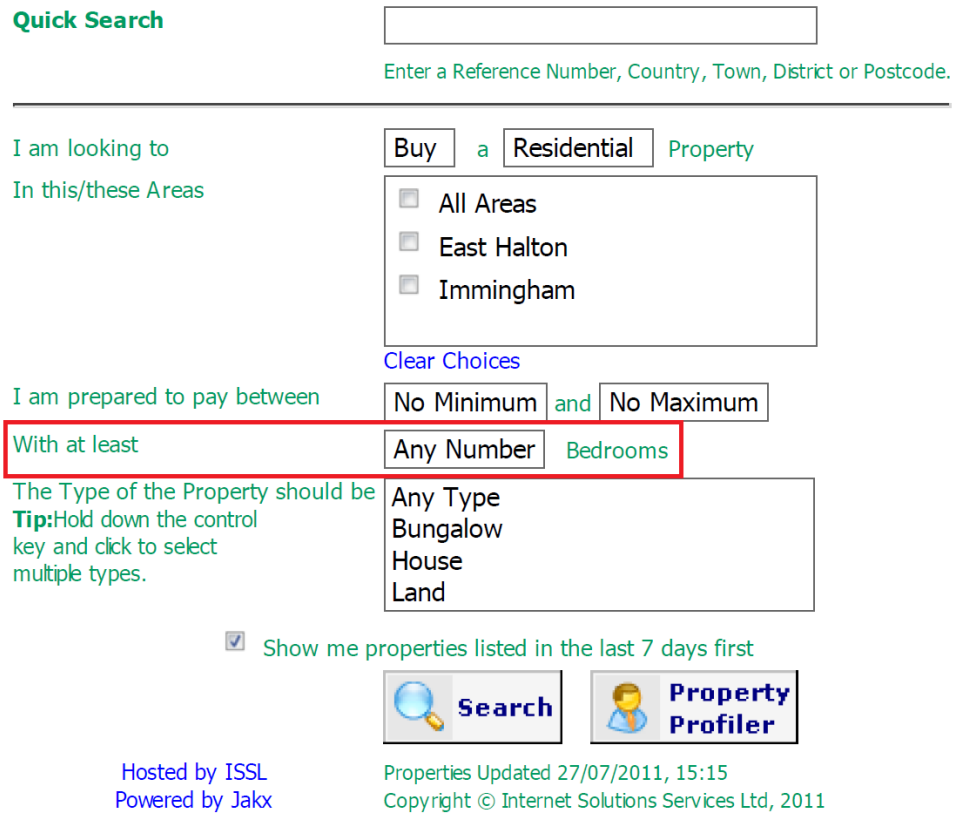
This heuristic is directly translated into declarative rules: first, the least common ancestor n is computed for each field by applying the conditions specified above; then each text node appearing as a descendant of n is associated as a label to the field being considered. These rules rely on DOM relations and the derived navigation relations (corresponding to XPath axis).

```
hasBasicLabel(F,L,T) :- field(F),
        leastCommonAncestor(A,F),
        descendant(L,A),
        html_text(L,_,_,_,_),
        node_text(L,T,true).
```

In addition to the two heuristics, OPAL also captures the special form fields, i.e., buttons. Typical form buttons are implemented with HTML `<input>` element with *type* submit, reset, and image value, or HTML `<button>` element with *type* submit and reset. In general, buttons play a different role from other form controls. It is important to handle form fields explicitly. It is often the case that the buttons do not have any text labels other than the information displayed on the form button, e.g., as provided by the *value* attribute. This will have an effect on later segment scope labeling (see Section 4.3). There are also cases where such button fields are not even present. With proper javascript code, any HTML element is possible to serve the button purpose. OPAL solves these cases by looking for such alternatives with the knowledge of no-presence of typical form buttons (Chapter 5).

4.3 Segment Scope

At segment scope, OPAL enlarges the area of the analysis from individual fields to groups of them. OPAL groups labels and fields into more complex structures called *segments* and



(a)

```
▼<tr id="trBedrooms">
  ▼<td valign="top">
     <span id="lblBedroomsWithAtLeast" class="Normal" for="ddlBedrooms">With at least</span>
   </td>
  ▼<td>
    ▼<label for="ddlBedrooms">
     ▶<select name="ddlBedrooms" id="ddlBedrooms" class="field">…</select>
     </label>
     "
     "
     <span id="lblBedrooms" class="Normal">Bedrooms</span>
   </td>
 </tr>
```

(b)

Figure 4.8: *Ideal Property Services* Web Form

then organizes them into a hierarchy (segment tree) on the basis of certain structural properties. OPAL then performs its form labeling based on these segments.

Figure 4.9 shows the data model that is used at segment-scope. Each segment is labeled with text nodes and is related to other segments in a hierarchical relation. Segments are specialized into forms and form elements. *Form elements* are basic segments that group together a form field with its label(s). Ideally a form is identified with an HTML `<form>` element that structurally contains a certain number of form segments/elements.

In the case where the <form> element is not present, we takes the least common ancestor of all segments/elements as the form root.

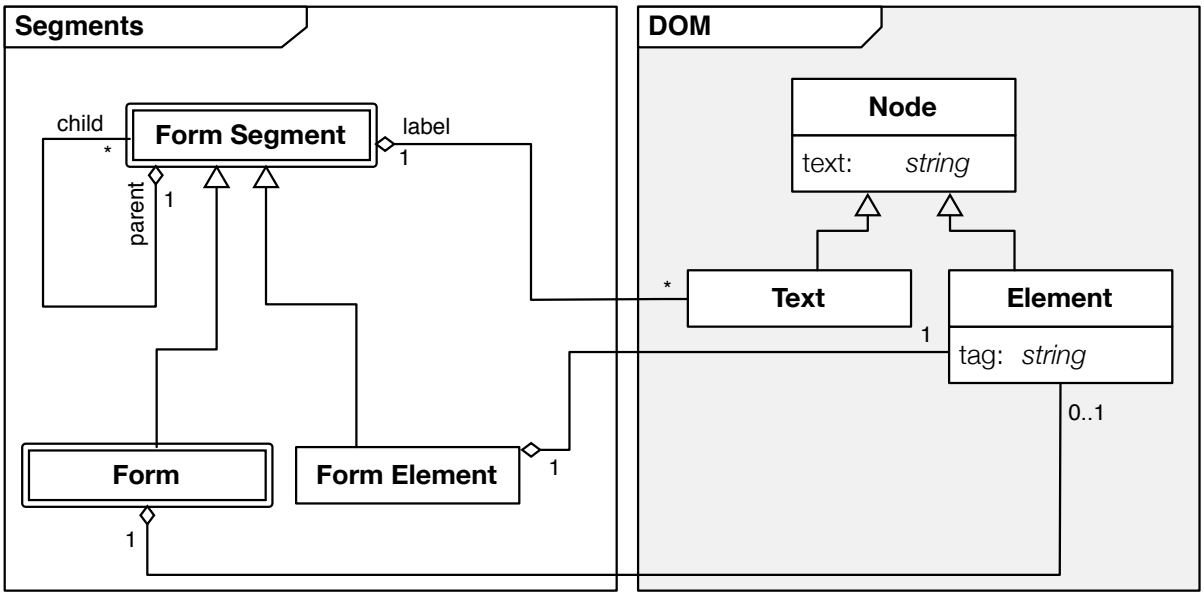


Figure 4.9: Segment scope: Segment model

4.3.1 Segment tree

Before proceeding to the labeling phase, OPAL firstly needs to identify the segments. The grouping condition for fields is essentially based on structural similarities while the heuristics are based on experimental observation.

As the DOM reflects the grouping of fields provided by the form designer, it often serves as a fair approximation of the semantic form structure. Based on this observation, OPAL starts its segmentation step with the DOM structure, with an elimination of superfluous nodes: (i) a node without field descendants. Fields are leaves in the segment tree. Thus every internal node in this tree must have at least one field descendant. (ii) a node with only one child. The elimination of such nodes mimics the collapsing of a node path from A to B where all nodes in the path have a same unique field descendant. (iii) a node with all its field descendants being style-equivalent to the fields in its neighbouring nodes. These nodes are usually for form layout purpose, e.g., representing six check boxes in two columns.

Two fields are **style-equivalent** ($\sim$) if they carry the same *class* attribute (used to indicate a formatting or semantic class) or the same *type* attribute together with *style* information. If all field descendants of the parent of an inner node n are style-equivalent, then n should be eliminated from the segment tree, as it artificially breaks up the sequence of style-equivalence fields, which is referred to as *equivalence breaking*.

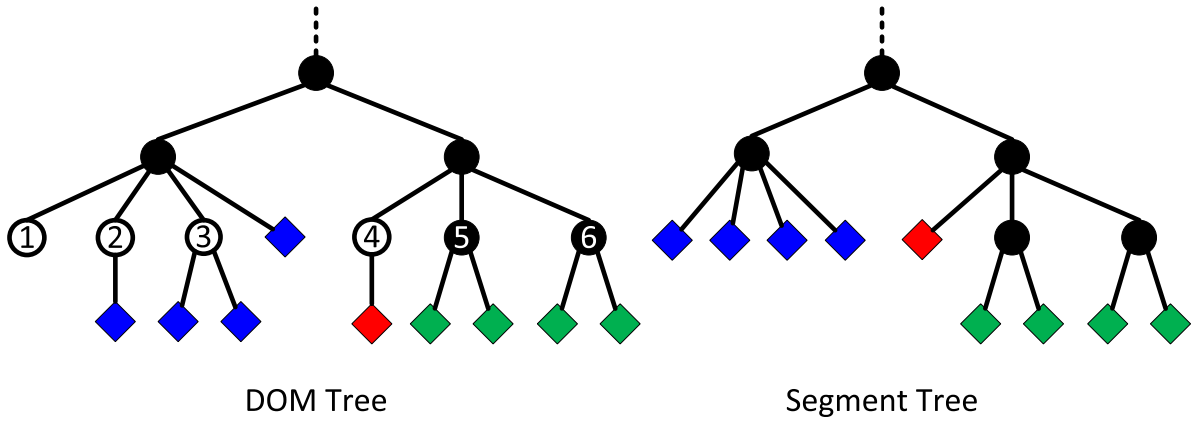


Figure 4.10: Example DOM and Segment Tree

Figure 4.10 illustrates the construction of the segment tree with an example. The left is the original DOM tree, whereas the right presents the resulting segment tree. Fields are represented with diamonds with same coloring indicating their equivalence status. Node 1 is eliminated since it has no field descendant. Nodes 2 and 4 are removed as they have single child. Node 3 is also discarded as it is equivalence breaking. Nodes 5 and 6 stay in the segment tree because of the red field, as we require all field descendants of their parent to be style-equivalent in order to determine their equivalence breaking status.

Definition 8. *The* ***segment tree*** P' *of a form page* P *is the maximal DOM tree included in* P *(i.e., obtained by collapsing nodes) such that the leafs of* P' *are all fields and for all its inner nodes* n
(1) $\exists d \in P' : R_{\mathit{descendant}}(d, n) \wedge d$ *is a field,*
(2) $|\{c \in P' : R_{\mathit{child}}(c, n)\}| > 1$*, and*
(3) n *is not equivalence breaking.*

The **segment tree** P' of a DOM tree P can be computed in $O(n \times d)$ where n is the size and d the depth of P. It takes $O(n)$ time to eliminate nodes under condition (1) as the worst case is to traverse each node. It takes $O(n \times d)$ to collapse nodes under condition (2) and (3), since in the worst case, we collapse $d - 2$ inner nodes and thereby move $O(n)$ leafs $d - 2$ times.

Algorithm 2 computes the segment tree P' for any DOM tree P. In lines 2–3, nodes that are not fields or do not have field descendant are eliminated. This ensures all leaves of P' are fields. Lines 4–5 guarantees every inner node of P' has more than 1 child. The equivalence breaking nodes are removed in lines 6–13, where OPAL computes a Representative for each inner node in a bottom-up fashion: If all field children (line 7) and the representatives of all inner children (line 8) are style-equivalent (line 9–10; $\sim$ is an equivalence relation and it suffices to compare a representative to each of the elements

Algorithm 2: `SegmentTree`$(DOM\ P)$, $\perp \not\sim n$ for any n

```
1  P' ← P;
2  while ∃n ∈ P' : n not a field ∧(∄d : R_descendant(d, n) ∈ P' ∧ d a field) do
3      delete n and all incident edges from P';
4  while ∃n ∈ P' : |{c ∈ P' : R_child(c, n) ∈ P'}| = 1 do
5      delete n from P' and move its child to the parent of n;
6  foreach inner node n in P' in bottom-up order do
7      C ← {f : R_child(f, n) ∈ P' ∧ f is a field};
8      C ← C ∪ {Representative(n') : R_child(n', n) ∈ P'} ;
9      choose r ∈ C arbitrarily ;
10     if ∀r' ∈ C : r ∼ r' then
11         Representative(n) ← r;
12         delete all non-field children of n and move their children to n;
13     else Representative(n) ← ⊥ ;
14 return P';
```

in C), a representative is chosen arbitrarily (line 11) and all inner children of that node are collapsed (line 12). Otherwise, we assign $\perp$ as representative (line 13), which is not style-equivalent to any node or to itself. It prevents this node (and its ancestors) from ever being collapsed. This can not introduce new violations to condition (1) and (2), as the algorithm never decreases the number of children, it never turns a leaf into an inner node, and it never removes fields.

In OPAL's declarative implementation, this is achieved by constructing recursively node groups (segments) starting with groups of fields at the bottom of the tree. The least common ancestor of all the nodes in a segment is taken as the segment root. The binary `part-of` relation represents the nested association between segments. The root of the largest segment (i.e., the segment containing all fields) is taken as the form root. In this way, OPAL handles web forms with or without the HTML `<form>` element. The following are some representative rules (for simplicity, stratified negation is expressed directly using "not").

```
segment(Es) :- similarFieldSequence(Es),
               leastCommonAncestor(A,Es),
               not hasAdditionalField(A,Es).
leastCommonAncestor(A,Es) :- commonAncestor(A,Es),
               not ( child(C,A), commonAncestor(C,Es) ).
partOf(E,A) :- segment(Es),member(E,Es),
               leastCommonAncestor(A,Es).
```

4.3.2 Segment Labeling

Once the segment tree is constructed, OPAL proceeds to the actual labeling. The goal is to align fields and text nodes appearing in the same segment. OPAL identifies patterns of field and text positions, which are used for the alignment, see Algorithm 3. First (lines 2–5), we create a form segment node n_s in the form labeling for each inner node s in the segment tree and choose s as representative for n_s ($\phi(n_s) = s$). For each segment with regular interleaving of text nodes and field or segment nodes, we use those text nodes as labels for these nodes, preserving any already assigned labels and fields (from field scope). In detail, we iterate over all descendants c of each segment in document order, skipping any nodes that are descendants of another segment or field itself contained in n (line 13). In the iteration, we collect all field or segment nodes in Nodes, and all sets of text nodes between field or segment nodes in Labels, except those text nodes already assigned as labels in field scope (line 14), as we assume that these are outliers in the regular structure of the segment. We assign the i-th text node group to the i-th field, if the two lists have the same size (possibly using the first text node as labels of the segment, line 17–19). Note that, by taking all text nodes between field or segment nodes, OPAL regards consecutive text nodes (i.e., not interleaved by a field) as a text group. This technique addresses the common case of multi-words form labels that lead to more than one text node, as for instance "Max price" where "Max" is rendered in bold (`<div><b>Max</b>price</div>`).

The segment labeling relies mainly on the two lists collected by OPAL: the Nodes and the Labels list. We discuss through all possibilities to illustrate the texts-node matching:

(1) $|\mathsf{Labels}| < |\mathsf{Nodes}|$. There are more fields than text groups. In this case, OPAL does not do any form labeling.

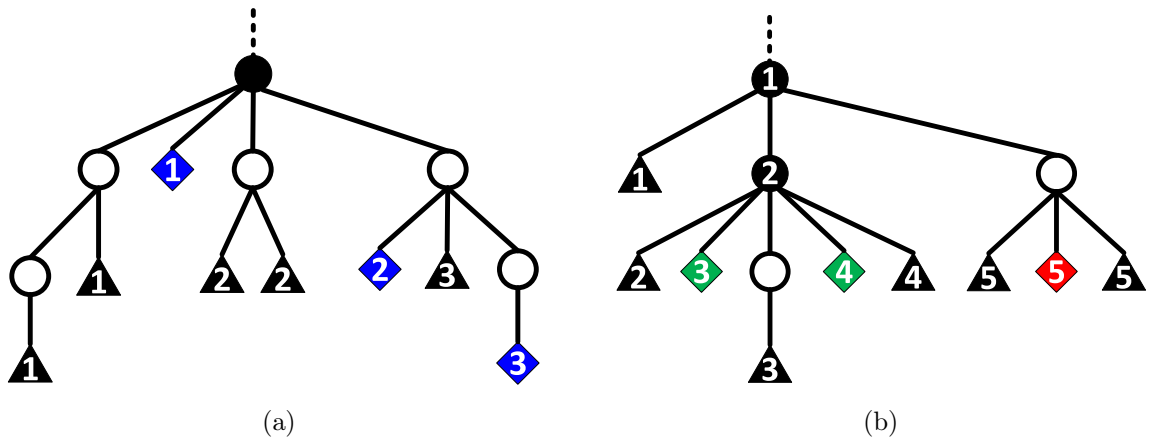
(2) $|\mathsf{Labels}| = |\mathsf{Nodes}|$. The two lists have exactly the same size. OPAL performs a direct one-to-one matching.

(3) $|\mathsf{Labels}| = |\mathsf{Nodes}| + 1$. There is one more text group. It is more likely that these texts carry semantic information that applies to the whole segment. Thus this particular group is assigned to the segment root as segment label and the rests continue with the one-to-one association.

(4) $|\mathsf{Labels}| > |\mathsf{Nodes}| + 1$. This case cannot happen since the Labels list is constructed by interleaving all texts with nodes in the Nodes list, and n nodes result in at most n+1 text groups.

Algorithm 3: `SegmentScopeLabeling`(*DOM P, Form Labeling F*)

```
1  S ← SegmentTree(P) ;
2  foreach inner node s in S in bottom-up order do
3      create a new segment n_s in F;
4      φ(n_s) ← s;
5      create an edge (n_s, c_s) in F for every φ(c_s) child of s;
6  foreach segment n in F do
7      Nodes, Labels ← new List();
8      textGrp ← ∅ ;
9      foreach c : R_descendant(c, φ(n)) ∈ P in document order do
10         if ∃f ∈ F : φ(f) = c then
11             if textGrp ≠ ∅ then Labels.add(textGrp); textGrp ← ∅;
12             Nodes.add(c);
13             skip all descendants of c in the iteration ;
14         else if c is a text node ∧ ∄d ∈ F : c ∈ ψ(d) then
15             textGrp ← textGrp ∪ {c};
16     if textGrp ≠ ∅ then Labels.add(textGrp); textGrp ← ∅;
17     if Labels.size() = Nodes.size() + 1 then
18         add Labels[0] to ψ(n);
19         delete Labels[0] from Labels;
20     if Labels.size() = Nodes.size() then
21         foreach i do add Labels[i] to ψ(Nodes[i]);
```



(a) (b)

Figure 4.11: Segment Scope Labeling

Figure 4.11 presents two examples for the segment scope labeling. Fields are denoted with diamonds with style-equivalence presented in same color. Triangles are text nodes, black circles are segments, and white circles are DOM nodes removed for the segment tree. The association between texts and fields are represented by same numbering. In Figure 4.11(a), the three fields are style-equivalent and thus result in one segment. The

Nodes [1,2,3] list contains the three fields and the Labels list [{1,1},{2,2},{3}] has three text groups as interleaved by the fields. With this regular structure of (text+, field)+, OPAL assigns the i-th text group to the i-th field. In Figure 4.11(b), segment 2 has two fields [3,4] with three text groups [2,3,4]. Thus the text 2 is promoted as the segment label. The remaining nodes have a regular structure (field, text)+ and are associated directly on a one-to-one basis. For segment 1 containing segment 2 and field 5 being labeled at field scope, there is only one text node left, and thus 1 is assigned as the segment label.

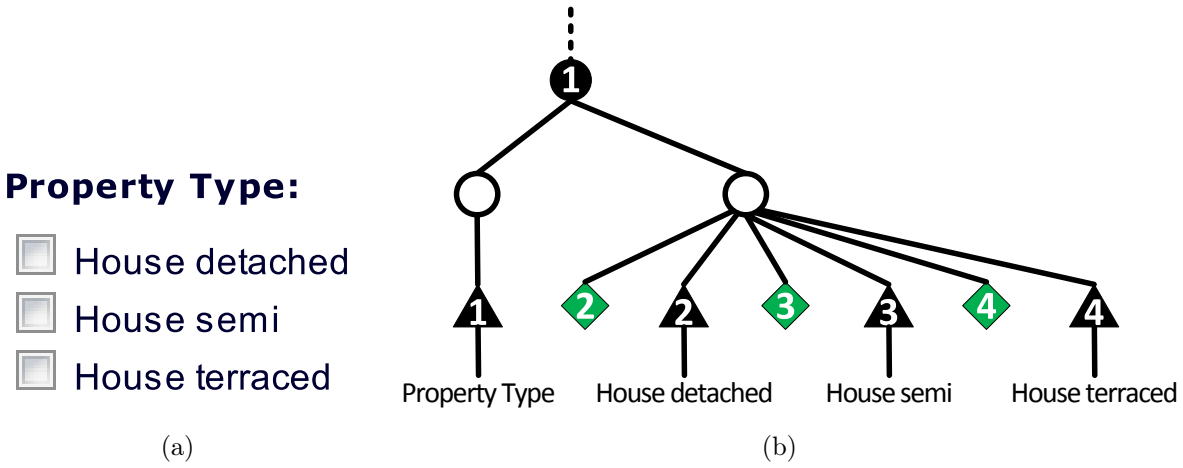


Figure 4.12: Segment Scope Form Example

The part of a web form in Figure 4.12 shows a real example where segment scope applies. The segment has a group label and three check boxes interleaving the other three text nodes as presented in 4.12(b).

OPAL's logical rules for segment labeling runs mainly with two list constructions for the Nodes and Labels list and one recursion for one-to-one association between the two lists. The following presents the top level rules for the label assignment. In the `matchOneToOne` predicate, each time we associate the heads of the two lists until the two base case where either both lists become empty or the Nodes list is empty (meaning that there is one more text group left, which is assigned as the segment label).

```
matchOneToOne(N,L,Nodes,Labels) :-
        labelingLists([N|Nodes],[L|Labels]),
        #length([N|Nodes],Nsize),
        #length([L|Labels],Lsize),
        Nsize<=Lsize.
matchOneToOne(N,L,Nodes,Labels) :-
        matchOneToOne(_,_,[N|Nodes],[L|Labels]).
matchOneToOne(N,Ls,Nodes,Labels) :-
        matchOneToOne(N,_,Nodes,[L|Labels]),
```

```
10          #length(Nodes,Nsize),Nsize=0.
```

4.4 Layout Scope

At layout scope, OPAL further expands the form labeling by exploring visual relationships between texts and yet unlabeled fields. In general, OPAL considers visible text nodes in the west, north-west, or north quadrant of a field, if the text nodes are not overshadowed by any other field. To this end, OPAL constructs a layout tree from the CSS boxes of the DOM nodes.

Definition 9. *Given a DOM P, tuple $(N_P, \lhd, \mathsf{w}, \mathsf{nw}, \mathsf{n}, \mathsf{ne}, \mathsf{e}, \mathsf{se}, \mathsf{s}, \mathsf{sw}, \mathsf{aligned})$ represents the* ***layout tree****, where N_P is the set of DOM nodes from P, $\lhd, \mathsf{w}, \mathsf{nw}, \mathsf{n}, \ldots$ the "belongs to" (containment), west, north-west, north, ... relations from RCR [51], and $\mathsf{aligned}(x, y)$ holds if x and y have the same height and are horizontally aligned.*

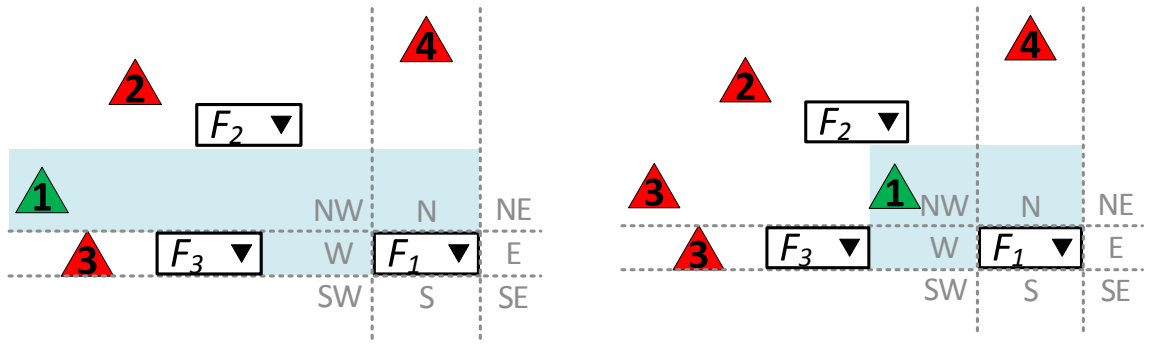
$\mathsf{w}, \mathsf{nw}, \mathsf{n}, \ldots$ are the neighbour relations. The layout tree is at most quadratic in size of a given DOM P and can be computed in $O(|P|^2)$. For convenience, we write, e.g., w-nw-n to denote the union of the relations w, nw, and n.

We observe in cultures with left-to-right reading direction that there is a strong preference with labels occurring in the w-nw-n region of a field. Moreover, with the interspersing of field and segment labels, it is necessary to take care of the overshadowing region. In general, given a field f, a visible text node t is overshadowed by another field f' if t is visible from f' and also closer to f'. In the case where f and f' are aligned, we relax this condition by allowing text nodes occurring on top to be visible from f.

Definition 10. *Given a text node t, a field f'* ***overshadows*** *another field f if* ***(1)*** *f and f' are unaligned, $\mathsf{w\text{-}nw\text{-}n}(f', f)$, and $\mathsf{w\text{-}nw\text{-}n\text{-}ne\text{-}e}(t, f')$ or* ***(2)*** *f and f' are aligned and* ***(i)*** *$\mathsf{w}(t, f')$ or* ***(ii)*** *$\mathsf{nw\text{-}n}(t, f')$ and there is a text node t' not overshadowed by another field with $\mathsf{ne\text{-}e}(t', f')$ and $\mathsf{w\text{-}nw\text{-}n}(t', f)$.*

The overshadowing cases are well illustrated with the examples in Figure 4.13. Consider F_1 as the targeted field. With condition (1), text nodes 2 and 4 are overshadowed by F_2. Text node 3 is to the west of F_3 and is overshadowed as well (by condition (2)(i)). In Figure 4.13(a), text node 1 is not covered by any overshadow condition and thus is associated with F_1. The condition (ii) for the aligned case is presented in Figure 4.13(b), where text node 3 is overshadowed by F_3 because of the presence of text node 1.

The design of the logic rules follows directly the definition of the w-nw-n region and the overshadowing region. To save computation effort, instead of considering the entire web page, the analysis is bounded by the form root (either the HTML `<form>` element if



(a) (b)

Figure 4.13: Layout Scope Labeling

exists or the root determined at the segment scope. Again, stratified negation is expressed directly using "not" for simplicity.

```
hasTextBoxWN(F,B) :- visibleField(F),
        formRoot(R),descendant(F,R),
        html_element(B,_,_,_,_,_),
        top(B,F),left(B,F),contain(R,B),
        not (Segment(S),contain(S,B),not descendant(F,S)),
        descendantOrSelf(BD,B),node_text(BD,BDText,true).
hasTextBoxNoOvershadow(F,B) :-
        fieldWestNorth(E,F),not fieldWest(E,F),
        hasTextBoxWN(F,B),not hasTextBoxWN(E,B),not top(B,E).
hasTextBoxAlignW(F,B) :- fieldWest(E,F),
        hasTextBoxWN(F,B),left(B,F),not left(B,E).
hasTextBoxAlign(F,B) :- fieldWest(E,F),
        hasTextBoxWN(F,B),top(B,F),
        hasTextBoxWN(E,B),top(B,E),
        rightMostBox(B).
```

We first look for all text boxes that fall into the w-nw-n region of a field f with the predicate `hasTextBoxWN`: line 2 ensures we use the correct form root (in case of multiple forms on a single page). Lines 3–6 find all proper text boxes in the w-nw-n region of f and prevent assignment of labels from unrelated form segments (line 5). Predicate `hasTextBoxNoOvershadow` expresses condition (1) by removing all boxes in the w-nw-n-ne-e region of f' (line 9) for unaligned f' with w-nw-n(f', f) (line 8). Now we consider the aligned case by considering text nodes only in the w region and the special case (as text node 1 in the example of Figure 4.13(b)). The former is straightforward by removing text nodes to the west of f' (line 10–11). The latter looks for the right-most text nodes that is in the nw-n region of f' (line 12–15).

Figure 4.14 presents a real example where layout scope takes place. This form is in a tabular design, i.e., each line of texts or fields appears as a table row in the HTML tree

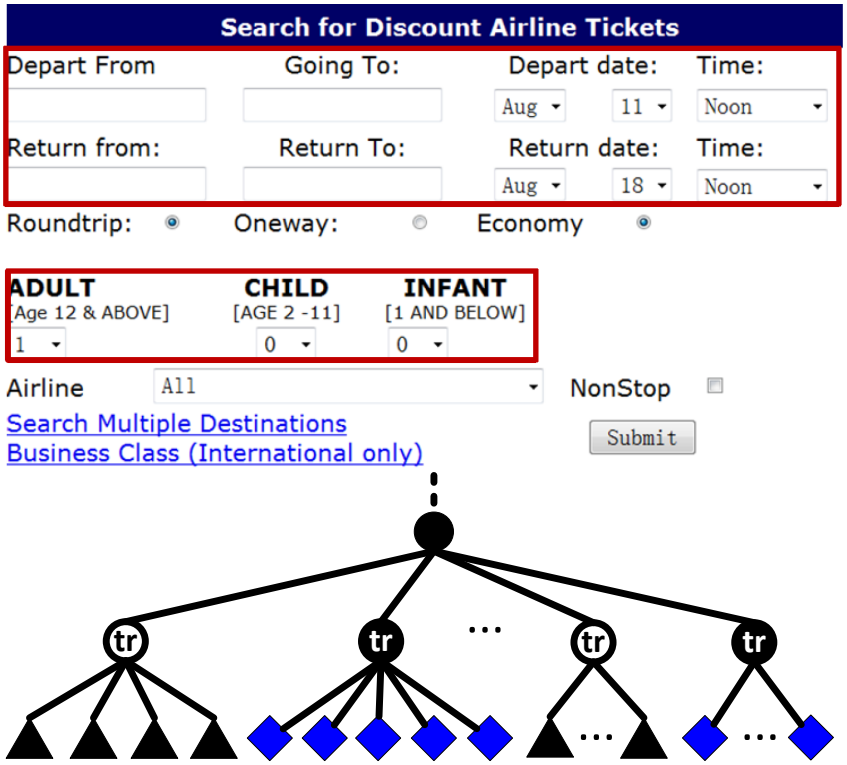


Figure 4.14: Layout Scope Form Example

presented below (triangles for texts , diamonds for fields, and black circles for segments). Hence, for the fields in the two highlighted areas, none of the techniques at the previous two structural-based scopes work. However, with visual-based analysis, OPAL successfully aligns all labels with fields correctly: it associates "Depart date" and "Return Date" with both fields appearing directly below the texts, and the remaining texts and fields are assigned on a one-to-one basis.

Chapter 5

Domain Dependent Form Interpretation

Form interpretation is necessary for form understanding: there is no straightforward mapping from form fields to domain concepts, and from the fields' structural or visual appearance to their semantic relationship. Even seemingly domain-independent concepts often exhibit domain specific peculiarities. For example, "price" may carry different semantic meaning even in a single domain, e.g., they may refer to "guide price on a house" or "monthly flat rental fee" in the real estate domain. Without proper form interpretation, form labeling itself is inadequate to provide a trustworthy form understanding.

We recall from the problem definition for form interpretation in Section 2.2.2 that a form model (F', τ) for a schema Σ is derived from a form labeling F by extending F with types and restructuring its inner nodes to fit the structural constraints of Σ. To this end, OPAL constructs a form model from a form labeling F in two steps: (1) *classification*, where OPAL classifies nodes in F according to the domain types $\mathcal{T}$ to obtain a partial typing τ_P. This step relies on the annotation schema Λ and its typing of labels in F; (2) *model repair*, where the segmentation structure derived in the segmentation scope (Section 4.3) is aligned with the structural constraints of Σ.

OPAL specifies the domain schema with a template language OPAL-TL, where it expresses both classification and structural constraints. To parameterize OPAL for a new domain, we only need to provide (1) a set of annotators implementing $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ and (2) an OPAL-TL specification of the domain types and their classification and structural constraints.

This chapter starts with an introduction to the language OPAL-TL, followed by OPAL's domain-dependent analysis, where OPAL performs classification and model repair towards a form interpretation.

5.1 Schema Design with OPAL-TL

OPAL provides a template language, OPAL-TL, for domain schema specification, consisting of annotation types, concepts, and constraints. OPAL-TL extends Datalog with templates and predefined predicates for convenient annotation querying. It allows reusing of common concepts and constraints with its template essence.

An OPAL-TL program is executed against a form labeling F and a DOM P. Relations are mapped in the obvious way from P to F. The `child(C,P)` relation remains the same in F. The document and sibling order is extended from P to F: `follows(X,Y)` for $X, Y \in F$, if $R_{\mathsf{following}}(\phi(X), \phi(Y)) \in P$ and no other node in F occurs between X and Y in document order; `adjacent(X,Y)`, if $R_{\mathsf{next\text{-}sibling}}(\phi(X), \phi(Y)) \in P$ or vice versa. We abbreviate $\mathsf{label}^l(\phi(X))$ as $l(X)$.

5.1.1 Annotation Querying

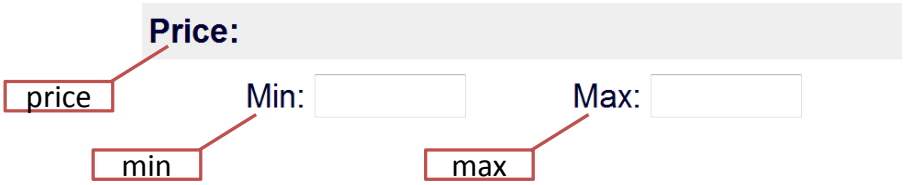
Annotations identify instances of domain types. It is necessary for OPAL to query for nodes with appropriate annotations (on their associated labels) in order to proceed with classification.

Annotations are characterised by an external specification of the characteristic functions $\mathsf{isLabel}_a$ and $\mathsf{isValue}_a$ for each $a \in \mathcal{A}$. In the current version of OPAL, the two functions are implemented with GATE [17] gazetteers and transducers either provided by human domain experts or derived from external sources such as DBPedia [44, 12] and Freebase[1]. Currently, OPAL contains a large set of such artefacts for common domain types such as location, price, contact, etc.

Annotations are attached to labels of fields and segments. Nodes associated with labels having same annotation types may not necessarily be classified with same domain type. Thus we need annotation query to select the right ones. The query distinguishes proper labels (e.g., "Min Price") and values (e.g., "£800"). It considers indirect labels inherited from parent nodes if necessary, e.g., when labels of a single node is not enough for domain classification. It also rejects if a node has more annotations with higher precedence, for example "lowest price first" indicates ordering rather than price.

Examples in Figure 5.1 illustrate some of the situations. In 5.1(a), the two labels "min" and "max", directly associated with the two text boxes, do not carry any information that indicates price type. This is available only when considering the indirect label, i.e., "Price:". In 5.1(b), the drop-down menu for result ordering receives two price annotations, two bedroom annotations, and five order-by annotations. This field should

[1] `http://www.freebase.com/`



(a)

(b)

Figure 5.1: Label Annotation Examples

not be considered for price type or for bedroom type, since order-by annotation precedes the other two and the field has more annotations of order-by.

Definition 11. *Given a form labeling F on a DOM P and an annotation schema Λ, an* **OPAL-TL *annotation query*** *is an expression of the form: $X@A\{d, p, e\}$ where X is a first-order variable, $A \in \mathcal{A}$, and d, p, and e are annotation* modifiers. *An annotation query $X@A\mu$ with $\mu \subseteq \{d, p, e\}$ holds for all $X \in [\![A\mu]\!]$ with*

$$[\![A\mu]\!] = \{n \in P : \mathit{Allow}_\mu(n) \cap \mathit{Match}_\mu(A) \neq \emptyset\} \setminus \mathit{Block}_\mu(A)$$

with

$\mathit{Allow}_\mu(n)$ set to $\psi(n)$ for $d \in \mu$, and $\psi(n) \cup \psi(\text{parent of } n)$ otherwise.

$\mathit{Match}_\mu(A)$ is to $\{l : \bigcup_{A' \sqsubset^* A} \mathit{isLabel}_{A'}(l)\}$ for $p \in \mu$, and $\{l : \bigcup_{A' \sqsubset^* A} (\mathit{isLabel}_{A'}(l) \vee \mathit{isValue}_{A'}(l))\}$ otherwise.

$\mathit{Block}_\mu(A)$ to $\{n : \exists A' \prec A, |\mathit{Match}_\mu(A)| < |\mathit{Match}_\mu(A')|\}$ if $e \in \mu$, and $\emptyset$ otherwise.

Intuitively, an annotation query $X@A$ returns all nodes having labels that are annotated with A. We use the modifiers to put more restrictions on the query result. The modifier d (direct) asks for labels associated with a node n only. Without d, OPAL considers annotations on the labels n's parent. The modifier p (proper) restricts the query to consider proper labels ($\mathsf{isLabel}_A$) only, otherwise both proper labels and value labels ($\mathsf{isValue}_A$) are taken into account. The e (exclusive) modifier requires checking on annotation precedence, i.e., a node should have more labels with annotation A than those with other annotation types which has precedence over A.

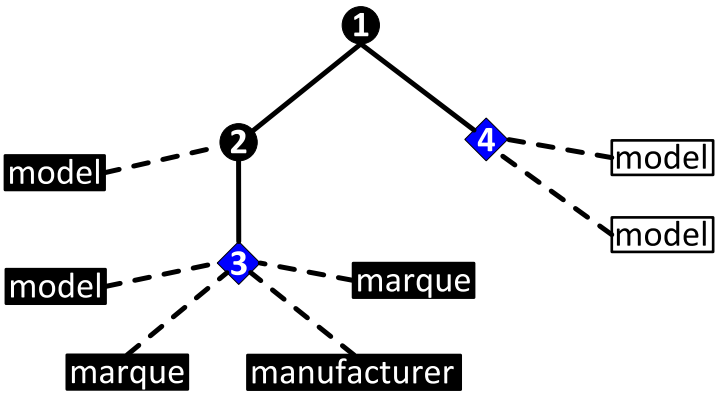


Figure 5.2: Form Labeling Example with Annotation

To illustrate the idea, consider the example in Figure 5.2, with diamonds representing fields, circles denoting segment, and black rectangles proper labels, white rectangles value labels. The labels are further annotated with an annotation type (*model*, *marque*, or *manufacturer*). A query on annotation *model* without any modifier, i.e., $X@\mathit{model}\{\}$, returns node $2, 3, 4$.

Given the condition that $\mathit{marque} \prec \mathit{model}$, $X@\mathit{model}\{d, e\}$ matches $2, 4$ but not 3, since 3 has more labels of *marque* than of *A*, which violates the exclusive modifier. $X@\mathit{model}\{p, e\}$ matches $2, 3$ but not 4, since the query considers no value label with the proper modifier. This query matches 3 despite the presence of e, as the annotation associated with its parent is also being considered (without direct modifier d) and thus there are two *model* labels.

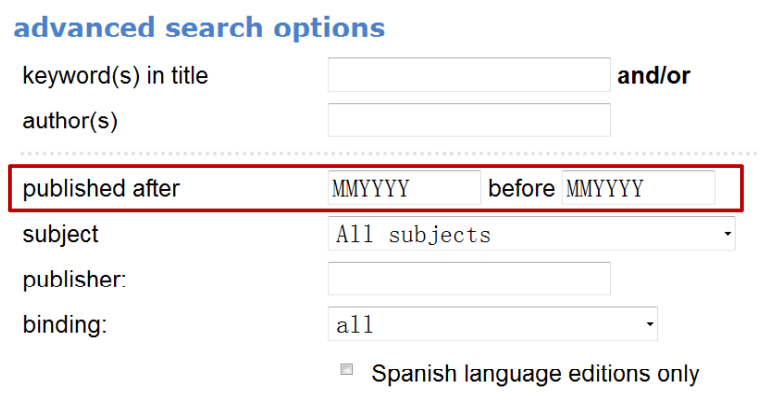
Given that $\mathit{manufacturer} \sqsubset \mathit{marque}$ and $\mathit{marque} \prec \mathit{model}$, the query $X@\mathit{model}\{p, e\}$ return node 2 only. With the given condition that *manufacturer* is a subclass of *marque*, even considering parent labels, there are still three *marque* and two *model* labels, which does not obey the exclusive modifier as $\mathit{marque} \prec \mathit{model}$.

5.1.2 OPAL-TL Templates

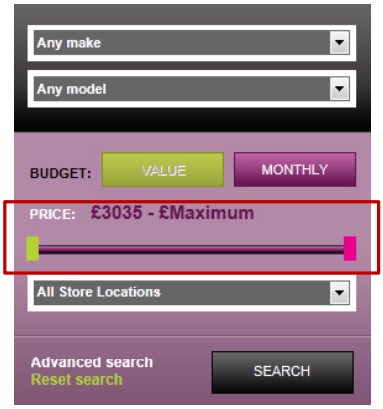
Template

We observe ubiquitous patterns when defining domain concepts and constraints. Thus, to reduce the effort in domain schema design, OPAL-TL extends Datalog$^\neg$ (Datalog with stratified negation) by templates which are reusable throughout various concepts and domains. Examples of such patterns are basic classification patterns that derive a domain type from a conjunction of annotation types, or structural patterns, e.g., min-max range, which express multiple fields with related annotations in a group and clues that they represent a range. For example in Figure 5.3, although the three forms are taken from

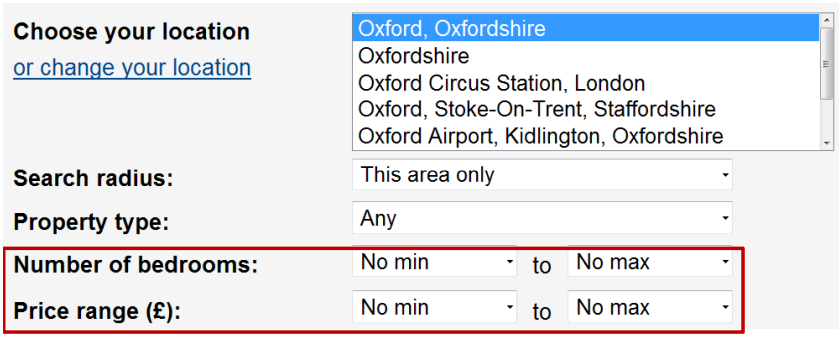
three different domains, all of them contain range patterns (highlighted with red boxes) for various domain concepts.



(a) abc.nl (book)



(b) carshop.co.uk (car)



(c) rightmove.co.uk (real estate)

Figure 5.3: Range Pattern in Various Domains

In general, there are two types of template patterns, one for classification constraints, one for structural constraints. The former specify patterns for relationships between annotations and domain concepts, e.g., a field with *city* annotation is a "location" field; the latter express the abstract structure between domain concepts, e.g., a real estate form must have a "location" field. These will be addressed with detail in the following sections 5.2 and 5.3.

Definition 12. *An* OPAL-TL ***template*** *is an expression* $TEMPLATE\ N{<}D_1,\ldots,D_k{>}\ \{\ p \leftarrow expr\ \}$ *where* N *names the template,* $D_1,\ldots,D_k$ *are template parameters,* p *is a template atom, expr a conjunction of template atoms and annotation queries. A* template atom $p{<}C_1,\ldots,C_k{>}(X_1,\ldots,X_n)$ *consists of first-order predicate name* p*, template variables* $C_1,\ldots,C_k$*, and first-order variables* $X_1,\ldots,X_n$*.*

Multiple rules with the same head express union. For convenience, we use $\vee$ and $\neg$ over conjunctions, which are translated to pure Datalog$^\neg$ rules .

As an example, the following template defines a pattern of constraints that associate the domain type D to a node N whenever N is labeled by an exclusive direct annotation of type A.

```
TEMPLATE basic_concept<C,A> { concept<C>(N) :- N@A{d,e}  }
```

Instantiation

Given a domain constraint of a common pattern, OPAL-TL only needs to instantiate the corresponding template.

A template *tpl* is *instantiated* to produce a family of rules where the formal template variables $D_1, \ldots, D_k$ are instantiated using values $v_1^i, \ldots, v_k^i$ from a *template instantiation* expression of the form

```
INSTANTIATE tpl<D_1,...,D_k> using { <v_1^1,...,v_k^1> ... <v_1^n,...,v_k^n> }
```

For example, the following expression instantiates `basic_concept` replacing D with domain type `radius` and replacing A with annotation type *radius*

```
INSTANTIATE basic_concept<C,A> using {<radius, radius>}
```

The instantiationg produces the following rule:

```
concept<radius>(N) :- N@radius{d,e}
```

Datalog Translation

A template atom $p{<}v_1, \ldots, v_k{>}(X_1, \ldots, X_n)$ is translated as a n-ary Datalog predicate $t(X_1, \ldots, X_n)$ where the predicate name t is obtained by concatenating p with $v_1, \ldots, v_k$. As an example, consider the instantiated rule above, the translation into Datalog rule will produce the following output,

```
concept_radius_field(X) :-
  hasAnnotation(X,AnnotationID,radius),
  precedingAnnotation(Annotation,radius),
  not (count{AnnotationID:hasAnnotation(X,AnnotationID,radius)} <
     count{AnnotationID:hasAnnotation(X,AnnotationID,Annotation)}).
```

where the `hasAnnotation` predicate is derived from the `content` and `annotation` facts linked with the labels of nodes.

```
hasAnnotation(X,AnnotationID,Annotation) :-
        hasLabel_field(X,L),
        content(L,Clob,Start,End),
        annotation(AnnotationID,Clob,Astart,Aend,_),
        Start<=Astart,End>=Aend,
        annotationFeature(AnnotationID,Feature,Annotation).
```

Proposition 1. OPAL-TL *has the same data complexity as Datalog*$^\neg$.

Proof. After instantiation OPAL-TL rules are translated to Datalog with stratified negation and inequality by producing unique names for concept predicate names, and expanding $\vee$ into multiple rules. Though this can yield a Datalog program exponential in the size of the OPAL-TL specification, data complexity remains unaffected. □

5.2 Classification

OPAL carries out classification following the classification constraints of the domain schema specified with OPAL-TL. The specification with templates enables reuse of domain concepts and concept patterns. We identify three patterns that suffice to describe most classification constraints in our target domains (UK real estate and used car domain), which effectively capture common semantic entities in the web forms and instantiated with various concepts.

A classification constraint for a domain concept C consists of a domain type (or concept) C and an annotation type A. None of the classification constraint patterns involve more than one annotation type as template parameter, although the annotation query in the body of the pattern template often require checking on additional but fixed set of annotation types (e.g., preceding types).

```
TEMPLATE basic_concept<C,A> { concept<C>(N) :- N@A{d,e} }
2
TEMPLATE concept_by_segment<C,A> { concept<C>(N) :- N@A{e,p} }
4
TEMPLATE concept_minmax<C,C_M,A> {
6 concept<C_M>(N_1) :-
        child(N_1,G),child(N_2,G),adjacent(N_1,N_2),
8       N_1@A{e,d},(concept<C>(N_2) ∨ N_2@A{e,d})
 concept<C_M>(N_2) :-
10      child(N_1,G),child(N_2,G),follows(N_2,N_1),
        concept<C>(N_1),N_2@range_connector{e,d}, ¬(A_1 ≺ A, N_2@A_1{d})
12 concept<C_M>(N_1) :-
        child(N_1,G),child(N_2,G),adjacent(N_1,N_2),
14      N_1@A{e,p},N_2@A{e,p},
        ((N_1@min{d,p,e},N_2@max{d,p,e}) ∨
           (N_1@max{d,p,e},N_2@min{d,p,e}))
16 }
```

The classification constraint templates are presented through lines 1–16.

- *Basic concept.* This template (line 1) captures direct classification of a node N with domain concept C, if N matches the annotation query `X@A{d,e}`, i.e., has labels of type A such that there is no more labels of any other type A' with $A' \prec A$. This

template is used by far most frequently, primarily for concepts with unambiguous proper labels.

Consider the example in Figure 5.4. The highlighted field receives both *price* and *currency* annotations. However, with the specification that *price* $\prec$ *currency*, the field is classified with the `price` concept.

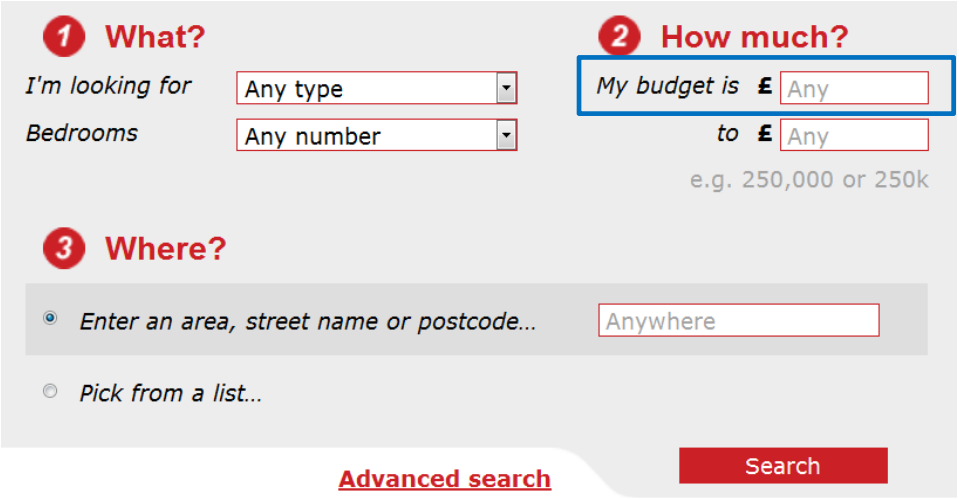


Figure 5.4: Example for Basic Concept Considering Precedence

- *Concept by segment.* This template (line 3) relaxes the requirement by considering also indirect labels (i.e., labels of the parent segment). In the real estate and used car domains, it is instantiated primarily for control fields such as `order_by` (price, bedroom) or `display_method` (grid, list, map), where the possible values of the field are often misleading.

  The form fragment in Figure 5.5 presents such an example. Considering only direct labels, the two radio buttons are annotated with *price* and *location* respectively, which will lead to miss-classification if using the `basic_concept` template. However, with `concept_by_segment`, OPAL considers indirect annotations as well, in this case *order_by* annotation associated with their parent. With the precedence over *price* and *location*, the two radio buttons are classified as `order_by` concept.

- *Min-max concept.* Web forms often represent a range of values for a feature (e.g., the acceleration of a car) with a pair of (min and max)fields. We express this pattern with three rules (line 5–16) that respectively describe three configurations of segments with fields missing proper labels.

  This pattern template is the only one that has two concept parameters, i.e., C and C_M, where $C_M \sqsubset C$ is the "minmax" variant of C, e.g., `min_price`. The first

Figure 5.5: Example for Concept by Segment

rule classifies adjacent pairs of nodes if both carry the annotation A, or a single such node if the other node is already classified as C. The second rule classifies the second node, that directly follows the first node (already classified with C), and has a `range_connector` (e.g., "to" in the "from...to" case), and is not annotated with an annotation type with precedence over A. The last rule classifies both nodes of an adjacent pairs with C_M if they carry a combination of `min` and `max` annotations.

For example, Figure 5.6 represents a case where the second field is classified as `max_price` using the second rule. In this example, the first field is associated with label "Price between" and is directly classified with the `basic_concept` rule. The second field has only "&" as label with the *range_connector* annotation, and thus satisfies the second rule. Figure 5.7 gives an example where the third rule applies. For the highlighted segment, "Price Range" is associated with the segment as label and the two fields present only range property. The third rule allows annotation queries over indirect labels. Thus both fields are classified as `price`.

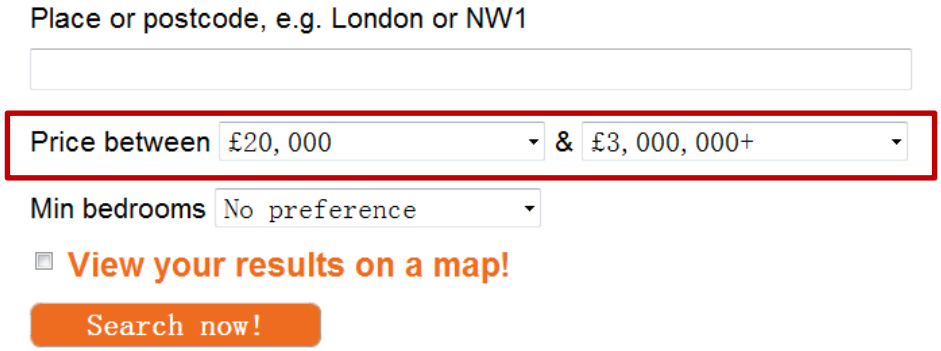


Figure 5.6: Example for Concept Min-Max with Rule 2

In addition to these templates, there is also a small number of specific patterns. We expect a submit control on every web form. However, with scripting language, this is not always implemented with proper form controls, i.e., HTML `<input>` or `<button>` element. In such cases, we use the following rule to describe forms that use HTML `<a>` links for submission. The identification of such a link (without probing or analysis of

Figure 5.7: Example for Concept Min-Max with Rule 3

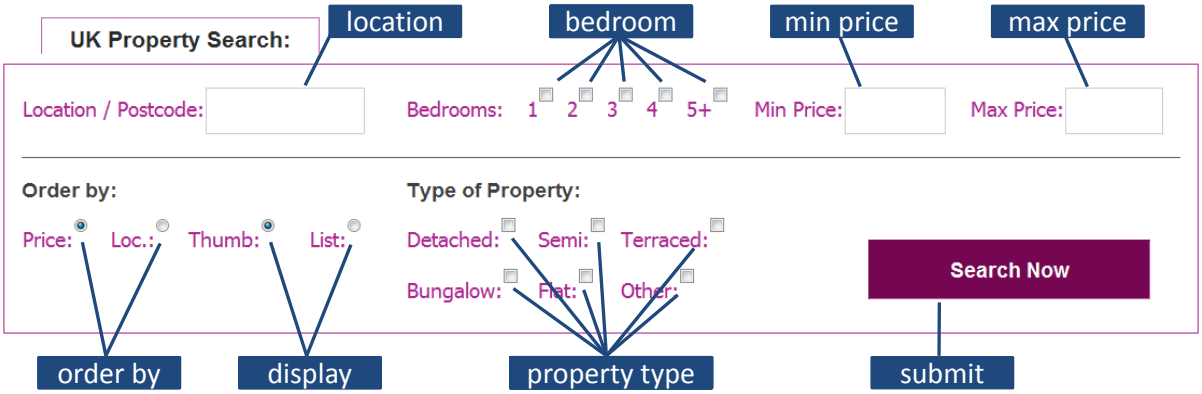


Figure 5.8: Classification on *Farlowestates* Real Estate Form

javascript event handlers) is performed based on annotation type for typical content, *href*, *title*, or *alt* attribute of contained images. The concept of `link_button` is mostly domain independent but also carries domain specific peculiarity. For example, in the real estate domain, a link for submission may be expressed without using any "submit" or "search" related keywords, e.g., a "buy" or "rent" link.

```
concept<link_button>(N1) :-
        form(F),descendant(N1,F),link(N1),
        N1@link_button{d},
        ¬(descendant(N2,F),(concept<button>(N2) ∨ follows(N1,N2)))
```

Figure 5.8 presents the classification results on the given form. Most of the classifications are straightforward with `basic_concept`. Only the two `order_by` radio buttons are classified with `concept_by_segment` as they both carry miss-leading annotations.

5.3 Model Repair

We need to make sure that the segment structure derived in the segment scope (Section 4.3) aligns with the structural constraints of the given domain, we also need to verify all the classifications achieved from the classification constraints and resolve any

issue raised during the step. To this end, OPAL proceeds to a model repair step, where it verifies and repairs the structure of the form to obtain a form model consistent with the domain schema. We use OPAL-TL to specify the structural constraints, with which OPAL also carries out verification and repairing.

5.3.1 Structural Constraints

We identify three patterns that cover most structural requirements in our targeted domain, i.e., UK real estate and used car domain. All of the patterns require the presence of an outlier among the siblings of the segment: `outlier<C>(G)` holds if at least one of G's siblings is not a C segment. In other words, this is to make sure that concept C is not the unique concept among G's siblings, otherwise the existence of G and its siblings are meaningless as all their children are of the same type.

```
TEMPLATE segment<C>{
 segment<C>(G) :-
        outlier<C>(G),child(N1,G),
        ¬(child(N2,G),¬(concept<C>(N2)∨segment<C>(N2))) }

TEMPLATE segment_range<C,CM> {
 segment<C>(G) :-
        outlier<C>(G),concept<CM>(N1),concept<CM>(N2),
        N1 ≠ N2,child(N1,G),child(N2,G) }

TEMPLATE segment_with_unique<C,U> {
 segment<C>(G) :-
        outlier<C>(G),child(N1,G),concept<U>(N1,G),
        ¬(child(N2,G),N1 ≠ N2,¬(concept<C>(N2)∨segment<C>(N2))). }

TEMPLATE outlier<C>{
 outlier<C>(G) :-
        root(G)∨(child(G,P),child(G′,P),¬(segment<C>(G′))) }
```

- *Basic segment.* A segment is a C segment, if it consists of only concepts typed with C or other segments. This rule captures most segment structures, for example, the `room`, `property_type` segment in the real estate domain, or the `fuel_type`, `transmission` segment in the used car domain as in the example of Figure 5.9.

- *Minmax segment.* A segment is a C segment, if it has at least two field children typed with C_M where $C_M \sqsubset C$ is the minmax type for C. This is typically used for segment representing ranges, e.g., a `price` segment consisting of `min_price` and `max_price` in both domains, and similarly for `bedroom` in the real estate domain, and `engine_size` in the used car domain. Examples of such segments have been presented in Figure 5.3 (Section 5.1.2).

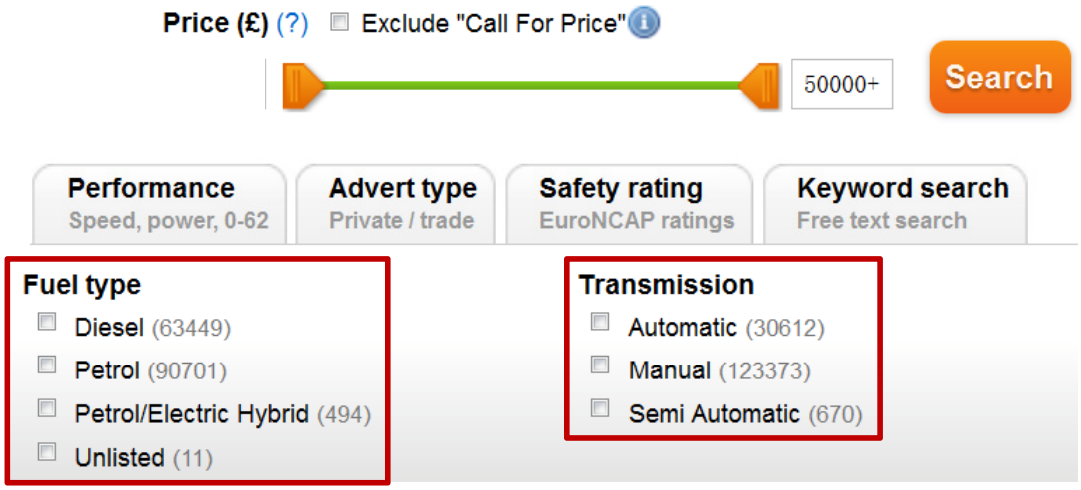


Figure 5.9: Example for Basic Segment

- *Segment with mandatory unique.* A segment is a C segment, if its children are only segments or concepts typed with C except for one (mandatory) field child typed with U where $U \not\sqsubset C$. This is used for `geography` segments where only one `radius` may occur, or `price` segments where `currency` is allowed. Figure 5.10 highlights a form fragment of such kind, where a postcode field works together with the distance field to restrict the geographic region.

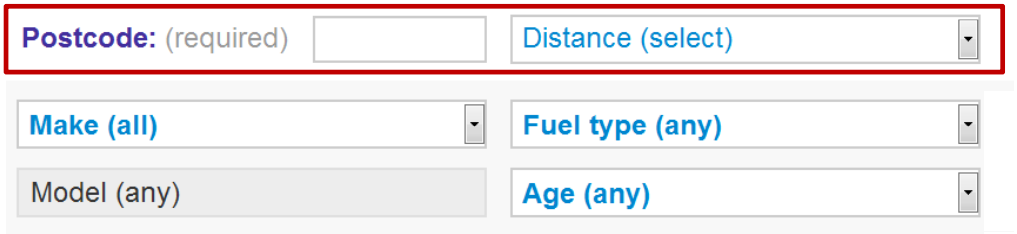


Figure 5.10: Example for Segment with Unique

As in the classification case, there are also additional patterns. We define the template `unique<C>(`N_1`,G)` that holds if N_1 is the only C typed concept child of G. For example, below we require the presence of a unique `submit` (or `link_submit` $\sqsubset$ `submit`) field in a `button` segment.

```
INSTANTIATE unique<C> using {<submit>}
segment<button>(N) :- unique<submit>(N,F),form(F).
```

An example for domain specific pattern is the `buy_rent` segment in the real estate domain. A `buy-rent` segment allows to search for properties to rent or to buy within the same form. This segment requires the presence of a single `buy_rent` field or both a `buy` and a `rent` field.

```
segment<buy_rent>(G) :-
        child(N1,G),child(N2,G),
        concept<buy>(N1),concept<rent>(N2).
segment<buy_rent>(G) :-
        child(N1,G),concept<buy_rent>(N1).
```

5.3.2 Form Model Repair

The classification yields a form interpretation F. However, this is not necessarily a model under domain schema Σ, and may contain violations of structural constraints. In order to fix such violations, OPAL corrects the obtained form interpretation to ensure its consistency with Σ. This is achieved with the rewriting rules, which express the adaptation of the classification and segment hierarchy following the guidance of Σ. OPAL performs the rewriting in a stratified manner to guarantee termination and introduces at most n new segments where n is the number of fields in the form.

(1) *Under Segmentation:* Intuitively, under segmentation occurs when a segment of certain domain type requires the presence of certain type of child segment. This also applies to the case where a segment consists of fields or segments of the wrong type.

Formally, if there is a segment n with type t such that $\mathcal{C}_\mathcal{T}(t)$ requires additional child segments of type $t_1, \ldots, t_k \notin \textit{child-}\mathcal{T}(n)$, we try to partition the children of n into $k+1$ partitions $P_1, \ldots, P_k, P_n$ such that $P_i \models \mathcal{C}_\mathcal{T}(t_i)$ and $P_n \cup \{t_1, \ldots, t_k\} \models \mathcal{C}_\mathcal{T}(t)$. For each P_i we add a new segment node as child of n, classify it with t_i, and move all nodes assigned to P_i from n to that segment.

In general, under segmentation requires value invention (to create segment nodes not existing in the DOM). As the number of segments is always bounded by the number of fields in the form, we provide a pool of unused segments in the segmentation.

(2) *Over Segmentation:* This is introduced by the DOM nodes only for form layout design purpose. For example, at segment scope, a multi-row form design often results in multiple segments each representing a single row. There is also the case where OPAL fails to identify some of these nodes as equivalence breaking because of lack of design similarities among fields of the same domain type.

If there is a segment n of type t with children $c_1, \ldots, c_k$ such that $\bigcup \textit{child-}\mathcal{T}(c_i) \cup \bigcup_{n' \in C} \tau(n') \models \mathcal{C}_\mathcal{T}(t)$ where C is the set of children of n without $c_1 \ldots c_k$, then we move the children of each c_i to n and delete all c_i.

OPAL restructures the segmentation by deleting unnecessary segment nodes and moving their children one level up in the segment tree.

(3) *Under Classification:* Sometimes, there are not enough hints for OPAL, from either annotations or constraints, to classify all nodes in F. Such cases are referred to as under classification. However, with repairing rules, OPAL may infer the classification for such nodes from their neighbouring or structural information.

If there is a segment n of type t with untyped children $c_1, \ldots, c_k$ and corresponding types $t_1, \ldots, t_k$ such that $\textit{child-}\mathcal{T}(n) \cup \{t_1, \ldots, t_k\} \models \mathcal{C}_\mathcal{T}(t)$ and, for each c_i, $\textit{child-}\mathcal{T}(c_i) \models \mathcal{C}_\mathcal{T}(t_i)$ holds, then we type c_i with t_i.

(4) *Over Classification:* Over classification refers to cases where a node receives multiple annotation types. Multiple annotation is usually caused by ambiguities with entity recognition, e.g. "ford" is both a car maker and also possibly part of a location name, or annotations obtained from indirect (parent) labeling. The former is generally resolved by type precedence during classification. The latter is handled by aligning with structural constraints.

If there is a segment node n of type t with child c typed t_1 and t_2 such that $\{t_1\} \cup \bigcup_{c' \in C} \tau(c') \models \mathcal{C}_\mathcal{T}(t)$ where C is the set of children of n without c, we drop t_2 from $\tau(c)$.

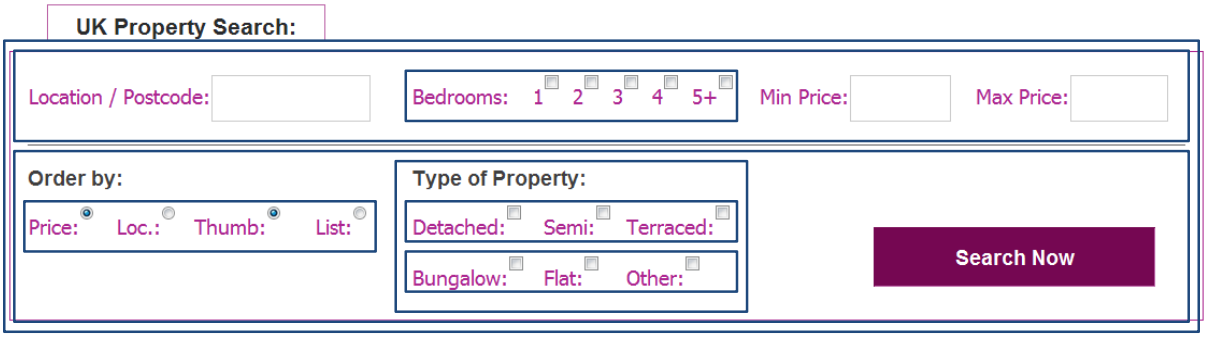
(5) *Miss Classification:* During form interpretation, it is also possible that OPAL classifies nodes with wrong types. This may be caused by incorrect labeling or misleading segmentation. If OPAL notices that the typing is inconsistent with the specification in domain schema, it removes the classification and continues with its inference as for under classification cases.

If there is a node n of type t where $\textit{child-}\mathcal{T}(n) \not\models \mathcal{C}_\mathcal{T}(t)$, then we delete the classification of n as t.

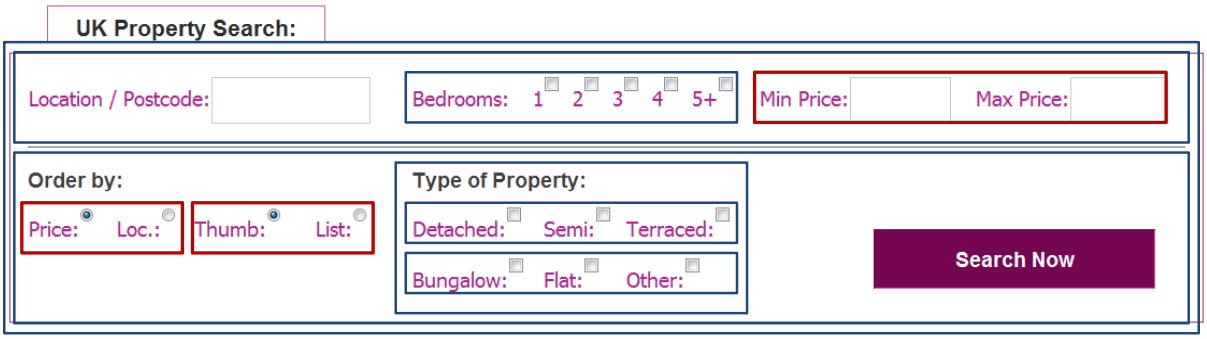
We recall from the classification in Figure 5.8, and continue with OPAL's interpretation on the *Farlowestates* form (see Figure 5.11). 5.11(a) shows the segmentation OPAL obtained at segment scope. There are several problems with this segmentation:

- The `min_price` and `max_price` fields are not arranged into a range segment as no such node is present in the DOM. This is a case of under segmentation. Following the `segment_range` constraint, OPAL introduces a price range segment to include both fields as in Figure 5.11(b).

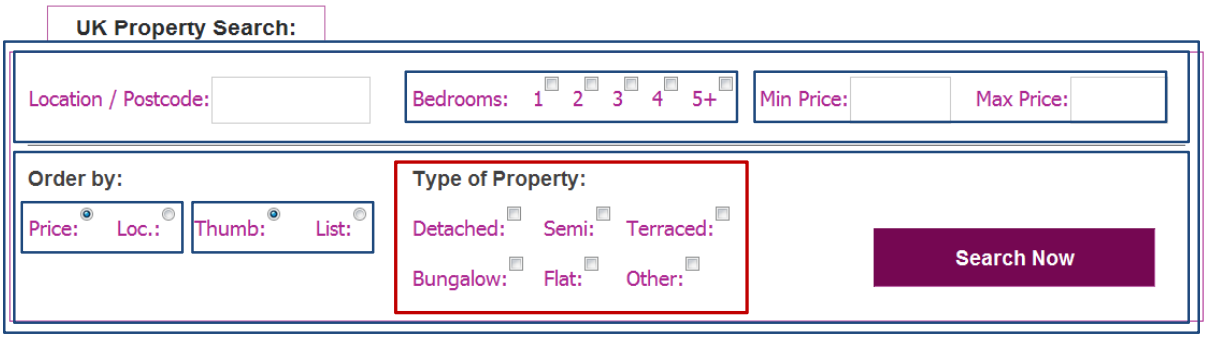
- The four radio buttons under "order by" are of two different domain types, i.e., `order_by` for the first two and `display` for the last two. However, all four are in the same segment. As specified with the structural constraint for basic segment, OPAL requires all children with the same domain type. This is also a case of under



(a) segmentation in segment scope



(b) resolving under segmentation



(c) resolving over segmentation

Figure 5.11: Model Repair on *Farlowestates* Real Estate Form

segmentation, where OPAL needs to introduce a new segment node into the form model (Figure 5.11(b)).

- As a result of the original segment with four radio buttons grouped together, the last two radio buttons in the four are also typed as `order_by` in addition to their `display` type. OPAL resolves this over classification by removing the `order_by` following the restructuring of the segment.

- The `property_type` segment is subdivided into two segments in the original segmentation, since OPAL identifies no style-equivalence among the six checkes due to

lack of similarity. However, with form interpretation, OPAL notices two segments of `property_type` and both are contained in a single parent segment. Thus, the two segments are removed with all their children directly contained in the larger segment (Figure 5.11(c)). This is an example of over segmentation.

- The segmentation obtained at segment scope preserves the two DOM nodes representing two form rows. However, in the domain schema, these nodes do not carry semantic meanings, and thus are treated as over segmentation and removed by OPAL (Figure 5.11(c)).

5.4 Domain Adaptation to Used Car Domain

This section presents domain adaptation to the UK used car domain, where classification and structural constrains are instantiated with 20 domain types. For illustration purpose, we ignore the sub-types and uncommon domain types.

```
INSTANTIATE basic_concept<C,A> using
        {  <price, price>
           <currency, currency>
           <mileage, mileage>
           <engine-size, engine-size>
           <location, location>
           <location, postcode>
           <location, area>
           <location, locality>
           <location, uk-county>
           <location, district-county-etc>
           <location, town-city-etc>
           <radius, radius>
           <radius, distance>
           <orderby, orderby>
           <pagination, pagination>
           <make, make>
           <model, model>
           <door, doors>
           <colour, colour>
           <transmission, transmission>
           <used-new, used-new>
           <road-tax-bracket, road-tax-bracket>
           <fuel-type, fuel-type>
           <car-type, car-type>
           <seat, seats>
           <emission, emissions>
           <dealer-type, dealer-type> }
```

```
INSTANTIATE concept_by_segment<C,A> using
        {  <price, price>
           <mileage, mileage>
           <engine-size, engine-size>
           <order-by, order-by>
           <pagination, pagination> }

INSTANTIATE concept_minmax<C,C_M,A> using
        {  <price, price_M, price>
           <mileage, mileage_M, mileage>
           <engine-size, engine-size_M, engine-size> }

INSTANTIATE TEMPLATE segment<C> using
        {  <price>
           <mileage>
           <engine-size
           <location>
           <orderby>
           <pagination>  }

INSTANTIATE TEMPLATE segment_range<C,C_M> using
        {  <price, price_M>
           <mileage, mileage_M>
           <engine-size, engine-size_M>  }

INSTANTIATE TEMPLATE segment_with_unique<C,U> using
        {  <price, currency>
           <location, radius>  }
```

Chapter 6

UK Real Estate Domain

The DIADEM system targets at implementing a fully automated web data extraction pipeline. As the Internet holds data for a variety of domains, e.g. job market, car dealers, etc, instead of getting data from unrelated domains, we are more interested in domain-centric data, which refers to data from one or several domains given that the data from these domains can be correlated. Such domains, for example, may contain information about travel packages provided by numerous travel agents, or details on restaurant offers, etc. To efficiently access domain-centric data on the web becomes critical in this fast-growing information world. It might be crucial for a company to make a fast and proper decision based on information gathered from a great number of web pages about its competitors. It would be important for financial consultants to make a more accurate estimation or prediction based on current market and other related financial information which requires data collection from large numbers of websites. It may also be valuable for a house buyer to track all houses on the market sold by various real-estate agencies within a certain price range and located in a certain area.

OPAL, as part of the DIADEM project, is designed to lead DIADEM passing through the gateway to these data. As a pioneer in the project, we conduct a study on the UK real estate market to understand the domain websites and their data. Particularly, we build an ontology to describe web forms in this domain.

6.1 A Study on UK Real Estate Websites

6.1.1 The UK Real Estate Market

The real-estate market is significant for individuals, the economy, and financial markets. It is important to monitor house prices and track their growth so as to predict future market trends. A House Price Index (HPI) is developed to describe the movements in the price of residential housing. There are several established House Price Indices (HPIs)

in use in the UK, which can be categorized into Government HPI and Private Sector HPI. Land Registry HPI is an example of Government HPI. It uses actual transaction prices for houses to build the index. However, the data about transactions are always delayed by months, and this leads to the delay of the market analysis, which surely affects the prediction. The most famous Private Sector indexes are Nationwide and Halifax, both of which use a characteristics-based method to analyze data from mortgage lending. Few housing websites (home.co.uk and rightmove.co.uk) have their own indices published monthly as well.

These indices are constructed from housing data, which can be improved in a handful of areas. The existing indices are published at certain intervals (normally a month) with large delay. Most of the indices provide analysis over nation-wide or regional data only. Such granularity is not satisfying in many cases, e.g., supervisory control on particular cities or towns. The analysis of the liquidity varies a lot across sub-markets, which is not adequately tracked by the existing HPIs. Besides, the housing market might be influenced by other non-housing related factors such as the job market. By adding this information we would expect a better prediction.

In the United Kingdom, besides individual agencies there is another kind of house information provider, which is generally referred to as "aggregator", for example, *Rightmove*.[1] Individual agencies publish their properties to the aggregators. People looking for houses could actually search from the aggregators directly, where there will be links provided to reach the original agencies. It seems obvious that the wrapping process can be greatly simplified if the aggregators are chosen as the sources. However, information on the aggregators' web pages is not updated immediately and sometimes the information may even be inaccurate if compared to that on the original websites. Thus, for the purpose of building a more reliable HPI, websites of individual agencies are preferred.

Job market is one of the areas that might affect the housing market. Official vacancy data sources (ONS vacancy data[2] and Jobcentre vacancy data[3]) have been chosen for this study. We aim to track daily changes in the properties of vacancies as well as the applications for individual vacancies. The properties include the job sector (e.g. banking), type (e.g. part-time), and the proposed salary. The data could be related to the housing market data by defining regions (city, postcode area, etc.)

We analyzed the information on the websites of over 80 real-estate agencies which provide residential sale service in Oxfordshire. Only 65 of them had houses on sale at the time we conducted our study (Sep, 2009). For simplicity, our study concentrates on residential house sales only, the conclusions from which can be easily extended to cover

[1] `www.rightmove.co.uk`

[2] `www.ons.gov.uk/ons/taxonomy/index.html?nscl=Vacancies`

[3] `data.gov.uk/dataset/job_centre_vacancies`

other customer-related aspects in the real estate domain, e.g. renting. We extracted oxford housing data daily from several selected websites for two weeks (with Lixto[4]). Despite data manipulation issues (such as duplication detection), we do see the possibility of deriving certain housing-related information by analyzing these data.

With proper data extraction, we foresee the index to be improved in several respects: it would be more timely as the data could be retrieved frequently enough from web pages. It could be more accurate than some of the existing indices since the data is directly extracted from the individual real-estate websites instead of the aggregators which do not have up-to-date information. The analysis could be brought down to low granularity such as postcode area (e.g. OX2), which makes more sense to individual users or small business groups. Also, it would take into account the influences by other markets or domain. This index could be expected to provide an improved analysis for the housing market.

However, Lixto requires manual wrapper construction for each website, and the wrappers may fail with any small changes on the web pages where navigation or extraction are involved. With our experience on Lixto, we see the necessity of developing a fully automated data extraction system, such as DIADEM.

6.1.2 Observations and Statistics

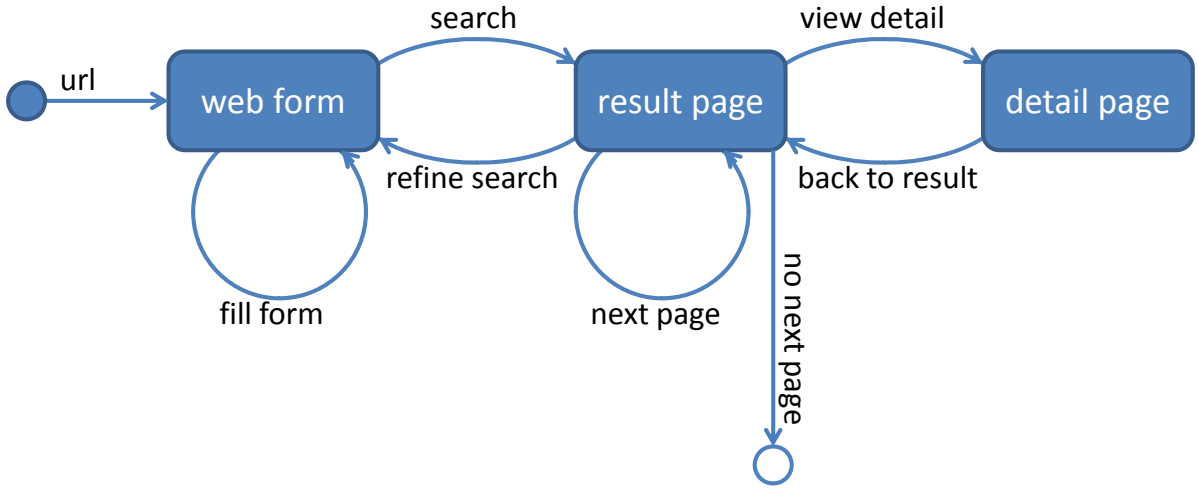


Figure 6.1: Property Search State Chart.

First let us take a look at the common property search process as presented in Figure 6.1. Given a real estate website, starting from the search engine page, other than advertisements and browsing panels, there is usually a particular module for searching (which is generally a web form). A user may enter part of an address or city/county

[4] www.lixto.com

names or postcode into a text field which asks for a search location. The purpose would be served by other means (selecting from maps or a list of locations) if the options are provided. Various form controls are also used to provide users with more search options, e.g., "to buy" instead of "to rent", looking for "house" instead of "flat", etc. Number of bedrooms and range of house prices are also popular options among real estate web forms. This process of providing search criteria can be completed on a single page for most of the search engines, although there are cases where the process is divided into several steps with each appearing on a single web page (this is indicated by the self-loop for the "web form" object). Then the submission of the form activates the search engine to form the query based on the values provided and to perform the query against the database and to display the search result.

On the search result page, if the results are not satisfying, most websites allow users to revise or refine their searches. On the result page, there is always a main search result panel where a list of houses is displayed (unless no results are found). If the returned results are displayed with multiple pages, navigation via a "Next" button allows access to each of the result pages. This navigation ends once there is no next page. On a single result page, typically each house is defined in a single container. All such containers are expected to share a common (or similar) traversal path for their corresponding root nodes in the DOM tree and the only difference between the paths is the positions of the nodes in the result panel. In every such container, there is usually a button or an image which links to the detail page (via corresponding *href* attributes in the source code). When we navigate to the detail page, more information of the particular house is available. After getting detailed information of one house, a user generally would go back to the search result to view other houses.

A more complete search process is presented in Figure 6.2, with the most common navigation paths highlighted. The numbers in the circles refer to the number of websites going through the path. Among the 65 websites, 57 generally follow the process above, 5 do not have the search engine page and the process starts directly from search result page, and 3 of them break the search engine page into sub-pages (page for specifying county level location and page for specifying areas within the county and other requirements). In other words, in order to proceed with any data extraction, over 92% of the websites requires form submission. Therefore, automatic form understanding is a necessary step towards automatic web data extraction.

Figure 6.3 summarizes the total number of websites (among the 60 websites with search engines) on which a particular search option is presented and the form controls used for their implementation. Each color represents the format that a search option is presented in. The list of search options is given along the horizontal axis. The full length

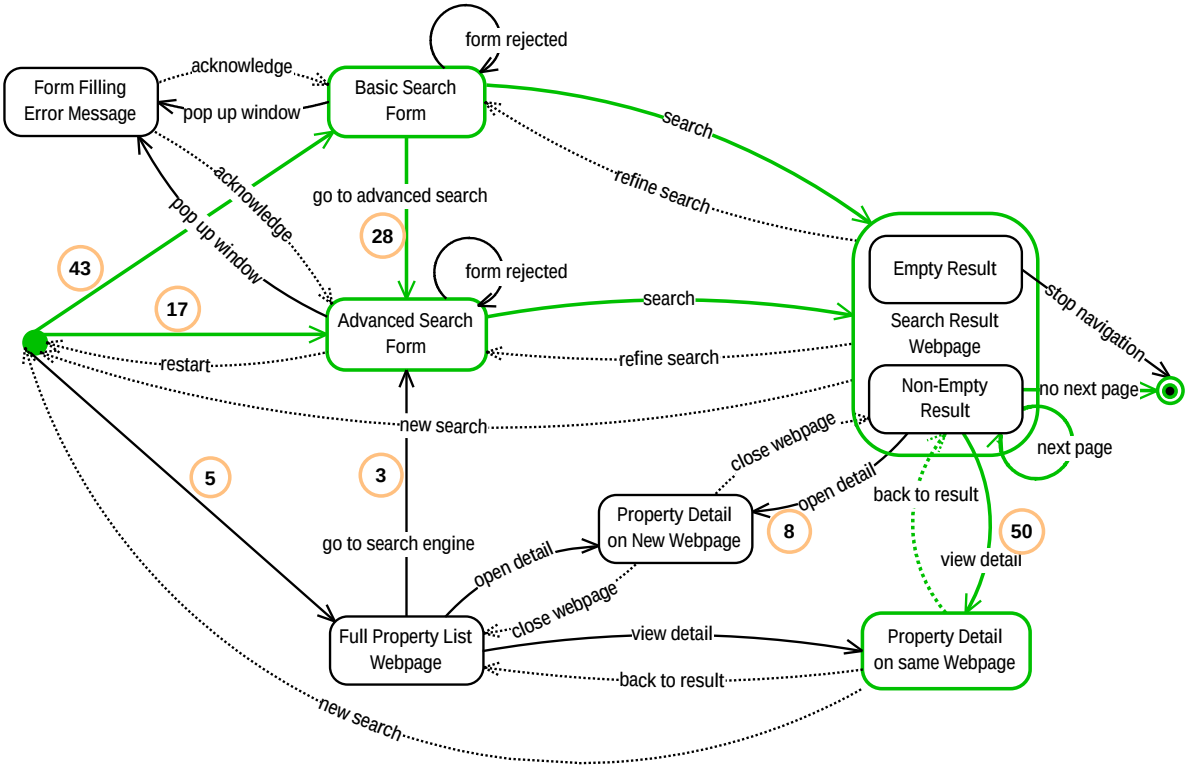


Figure 6.2: Navigation on Real Estate Website.

of a bar suggests the number of websites that an option is available. As presented in the figure, the most popular ones are location, price range, and the minimum bedroom number. Sometimes, a single search option may be provided in more than one format on the web page. For example, many websites allow buyers to enter a location by typing into a text box or define the location by selecting from a map.

6.2 UK Real Estate Web Form Domain Ontology

There are over 17,000 real estate agencies in the United Kingdom as provided on the yellow page.[5] Based on our observation from Oxford agencies, although the web forms are designed differently, there are a number of structural or semantic patterns that these web forms follow. In general, one could argue that existing "low level" annotators may help in locating these similarities, for example, a text box with "postcode" may be associated with a field for providing postcodes, or numbers following signs of currency symbols might be the price information that should be extracted. However, this is not enough for a thorough form interpretation. On a real estate website, we further observe that such elementary objects could be modeled into a structure from a "high level" point of

[5] `www.yell.com`

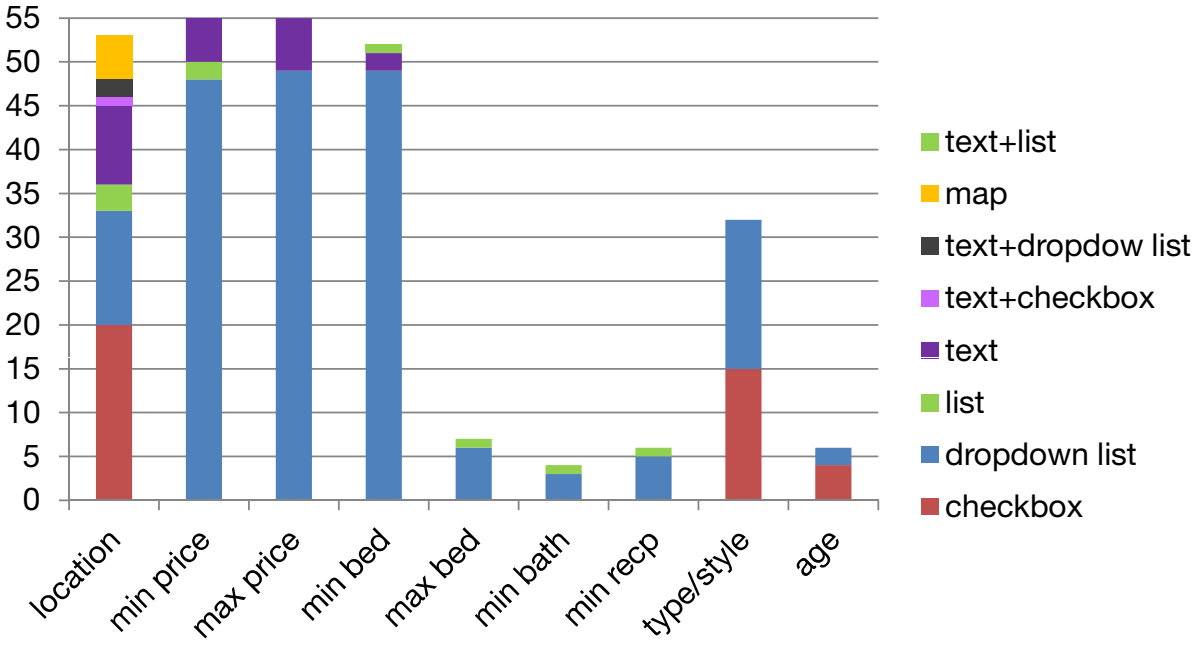


Figure 6.3: Form Controls for Various Search Options on UK Real Estate Websites.

view, in other words, having them grouped into larger (and more conceptual) objects. For instance, the concept of geographic search facility would contain an input box for free text and a map for region selecting. The web form itself makes the top concept in the hierarchy, i.e., real estate web form.

We define such common "high level" objects on the websites in the domain of real estate web forms and have them classified with domain types. People may define terms and concepts within the domain into mechanical and computable units for machine processing. Suitable conceptual models are hence derived from the construction of these objects and their relationships. These models drive the design of domain schema, which is the guidance for OPAL's form interpretation.

In order to understand the conceptual and logical structure of a web form, we require a reference description of how such forms occur in practice on the considered websites. In addition, such a description must be formal and machine readable since we need to combine it with the information coming from the browser page model and the phenomenological rules to produce a meaningful model of the web form. Due to the intrinsic object-oriented and hierarchical nature of a web form, a natural solution is to resort to an ontological description [28], where each component of a web form is represented as an ontological concept, corresponding to a class in the object-oriented terminology. The adoption of an ontological language also enables reasoning over such entities by means of standard inference procedures and technologies (e.g., rule-based reasoning or tableaux reasoning).

The role of the ontology in web form understanding is twofold: (i) The ontology formally models the design-patterns of web forms. Such reference structures can then be used to perform coherence checks against the facts provided by the phenomenological rules. This is needed since the heuristics implemented in mapping rules (from physical objects to conceptual entities) may provide inconsistent information because there are, in general, multiple valid labeling, grouping, and classification possibilities for the same form elements. (ii) The information extracted domain-independently represents only a partial description of the web form. We need such an ontology to exploit reasoning procedures to further interpret the form by producing a form model carrying semantic information.

To this end, we design an ontology to describe conceptually the web forms in the UK real estate domain. We also prove its feasibility by conducting a study showing the mapping from actual forms to the concepts in the ontology.

6.2.1 Ontology Representation

The ontology represents the hierarchical structure of a web form in terms of its components, their properties and the relationships among them. The vocabulary of our ontology includes terms to denote basic components of a web form such as *labels*, *buttons* and *input-fields* (e.g., text fields, drop-down lists, check boxes, etc.) along with the logical relationships among them. In particular, we introduce two logical structures that do not always have a counterpart in the DOM tree of the form namely *form elements* and *segments*. A form element is a logical element consisting of an input field and a label describing it. A *segment* denotes a group of form elements with a common goal with respect to the user interaction such as a price input-facility or a group of search options. Though in many forms we can find a representative DOM node for these logical concepts, in some cases we need to invent specific representative nodes and inject them into the form hierarchy.

For the representation of such structures, we adopt a simple ontological language consisting of the following constructs:

- *Classes (or concepts)*, represented in the ontology as unary first-order predicates and denoting the type of an object in the form e.g., an input field for the number of bedrooms or a price segment.

- *partOf* relation, represented in the ontology as a binary relation between two concepts and used to encode the hierarchical structure of the components of a web form e.g., a minimum and a maximum price field are part of a price segment.

- *Attributes*, represented in the ontology as binary relations between a concept and an XML data-type (e.g., strings, integers, etc.) and denoting the class attributes. An attribute may, for example, represent the fact that a room field is a minimum bedroom field or that an order-by input field is ordering in ascending or descending order.

In addition, we need to model some additional constraints for the attributes and the *partOf* relation that are required to fully capture the constraints we identified on the real-estate web forms. In particular we introduce:

- *Attribute constraints.* These constraints are expressed over the attribute values of certain objects. For example the fact that all such objects with a given type have mandatory values for their attributes or that a group of objects must agree on the values of their attributes.

- *Cardinality constraints.* These constraints are useful to specify constraints over the hierarchical structure of the ontology. For example the fact that an object cannot be part of two other objects at the same time (i.e., the web forms are conceptually trees) or that a certain component is optional in a web form.

For ontology representation, we use square boxes to denote the classes of the ontology, ellipses to denote the attributes of a class, "$\Diamond$" arrows to denote the *partOf* relation and "$\rightarrow$" arrows to express inheritance. In addition, we associate an attribute a to an aggregation operator (i.e., AND, OR and XOR) whenever we require that all the objects participating in the aggregation have compatible values for a. Given two values v_1, v_2 for an attribute a, we say that v_1 and v_2 are compatible if either $v_1 = v_2$ or at least one of them is a null (i.e., unknown) value. The notion of compatibility is easily extended to sets of values.

We describe the entire ontology in detail by cutting them into pieces of segment ontologies due to size limitation.

Real Estate Web Form

A real-estate web form (Figure 6.4) necessarily contains a price segment, used to provide the targeted prices for house search, and a segment containing the buttons for the submission of the form or other operations (e.g., clearing a form). Moreover, it must contain either a geographic-segment, containing location-based search options, or a property-feature segment, describing the features of the property. Optionally, we may find a contract segment defining the type of contract for the property and additional search options, e.g., to sort the search results according to price or room number. Each form

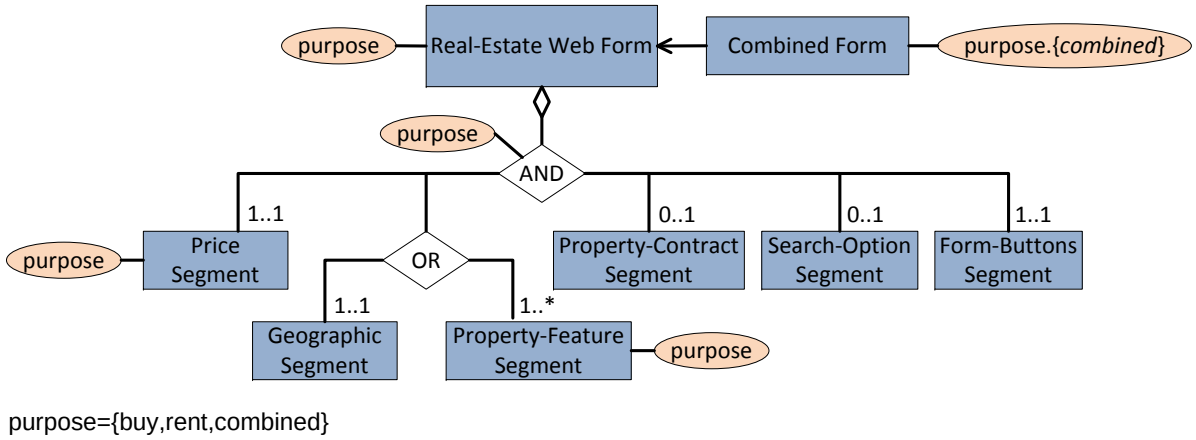


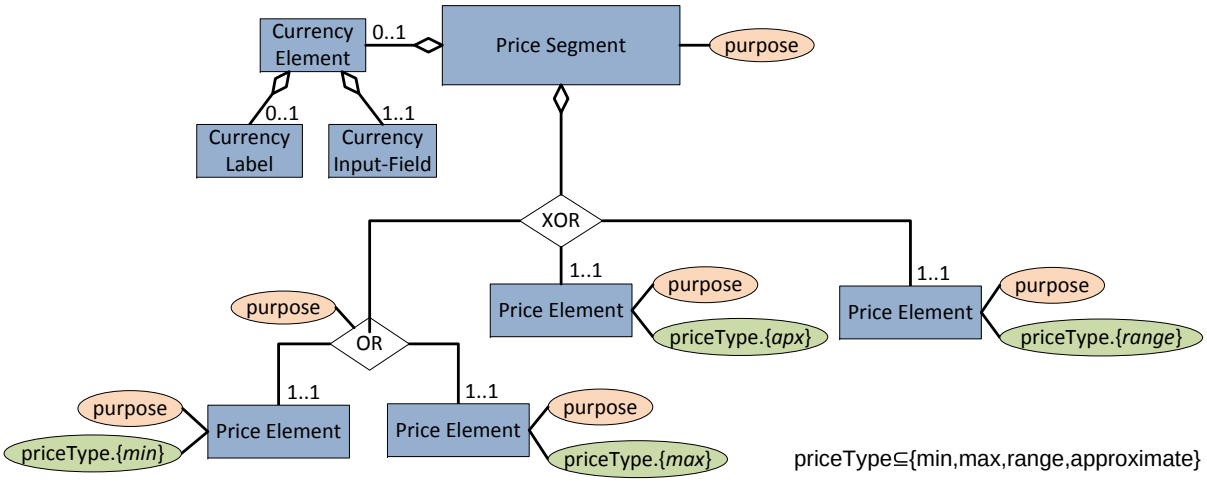
Figure 6.4: Ontology of Real-Estate Web Form

has a purpose (e.g., buy, rent or combined) represented as an attribute. Since price and property-feature segments also have a purpose attribute, the purpose value must be consistent whenever they belong to the same web form. We also say that a combined form, represented in the ontology as a subclass of a real-estate web form, must have "combined" as value for the purpose attribute. We explain each segment in detail in the following paragraphs.

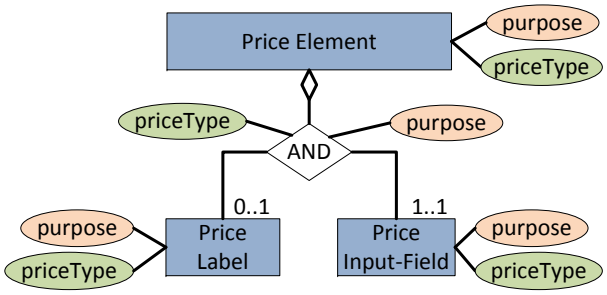
Price Segment

A price segment (Figure 6.5) on a real-estate web form is designed for putting limitation on prices. It is composed by a currency element and one or more price elements. The *priceType* attribute is used to denote different prices occurring in real-estate web forms, e.g., we may select a price range, an approximate price, or a pair composed by a minimum and a maximum prices. In the latter case, if they occur together, they must agree on the value of the purpose attribute, i.e., both prices should be rent prices or both are buy prices. Each price element consists of a label and a price input field (i.e., a list or a text field). In the case where the field itself reveals price types or purpose (e.g., £500000 is unlikely to be a rent price), we require the matching of attributes between the label and the field.

Figure 6.6 gives two examples of real-estate web forms that have Price Search Facilities, with 6.6(a) containing both a currency selector and a max price selector, and 6.6(b) holding both Min Price Input Field and Max Price Input Field (with two text boxes where people can input their own values) only.



(a)



(b)

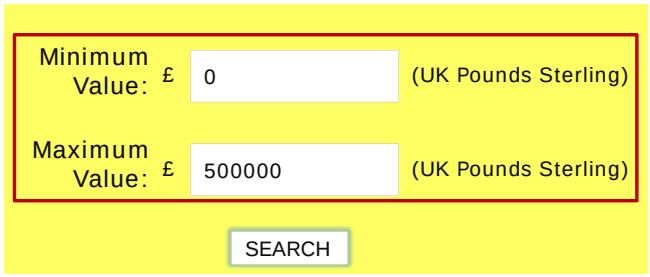
Figure 6.5: Ontology of Price Segment

Geographic Segment

A geographic segment (Figure 6.7) provides various ways to define the geographic region in a house search. An area/branch segment is part of a geographic segment and must be at granularity level of either area or branch. The granularity attribute distinguishes between one area-branch segment for counties or cities, e.g., oxford, from another area-branch segment for smaller regions, e.g., Summertown in Oxford. However, the granularity of a location value is not fixed. The idea of this attribute is only to distinguish between different geographic levels provided by an agency. As usual, an area-branch element consists of a label and an input field.

Figure 6.8 presents two examples of Geographic Search Facility. Example 6.8(a) is composed of area-branch input field, and also area-branch element (with granularity area) and a radius element with the region defined within a 5 mile radius of Oxford. The "or" keyword allows users to choose one of the two, either the input field or the area-branch element together with the radius element is enough to provide the search criteria. This design corresponds to the XOR relationship between the area-branch element and the

(a)



(b)

Figure 6.6: Price Search Facility Example

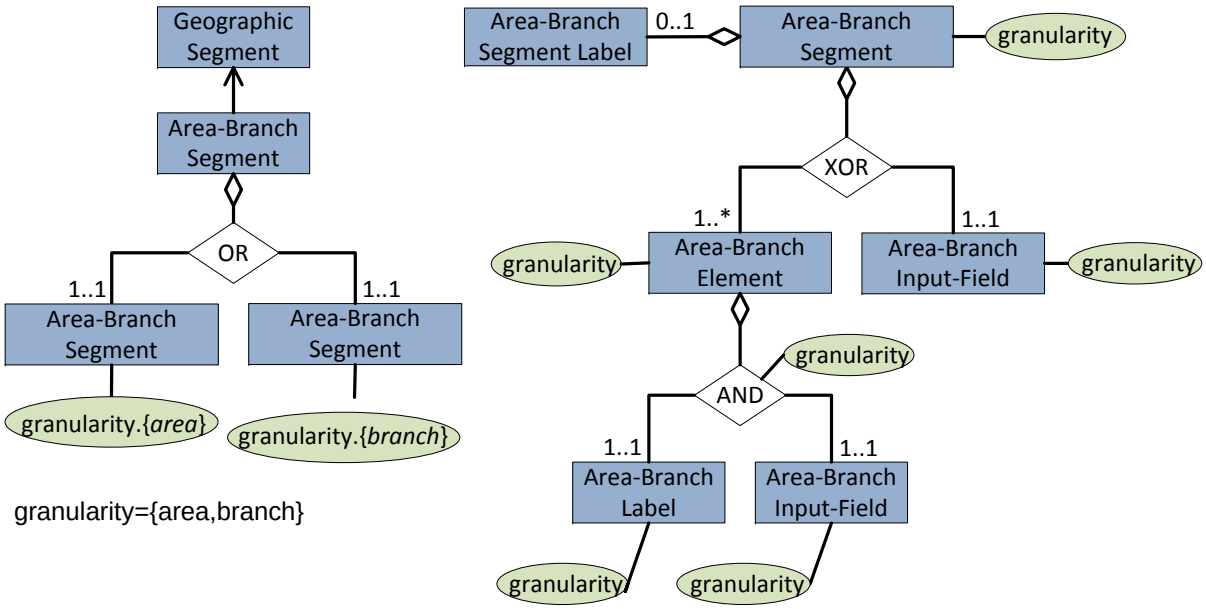
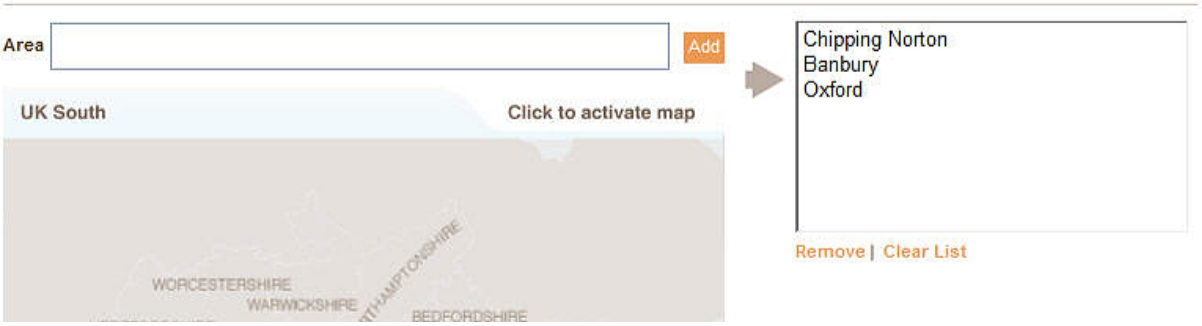


Figure 6.7: Ontology of Geographic Segment

input field. Example 6.8(b) has an area-branch input field, and an address container where three places have been selected and added into the list of searching areas. The ontology currently does not model the situation due to the interaction between the two fields. We also do not model "map" currently due to the restriction of conducting analysis on maps.

(a)



(b)

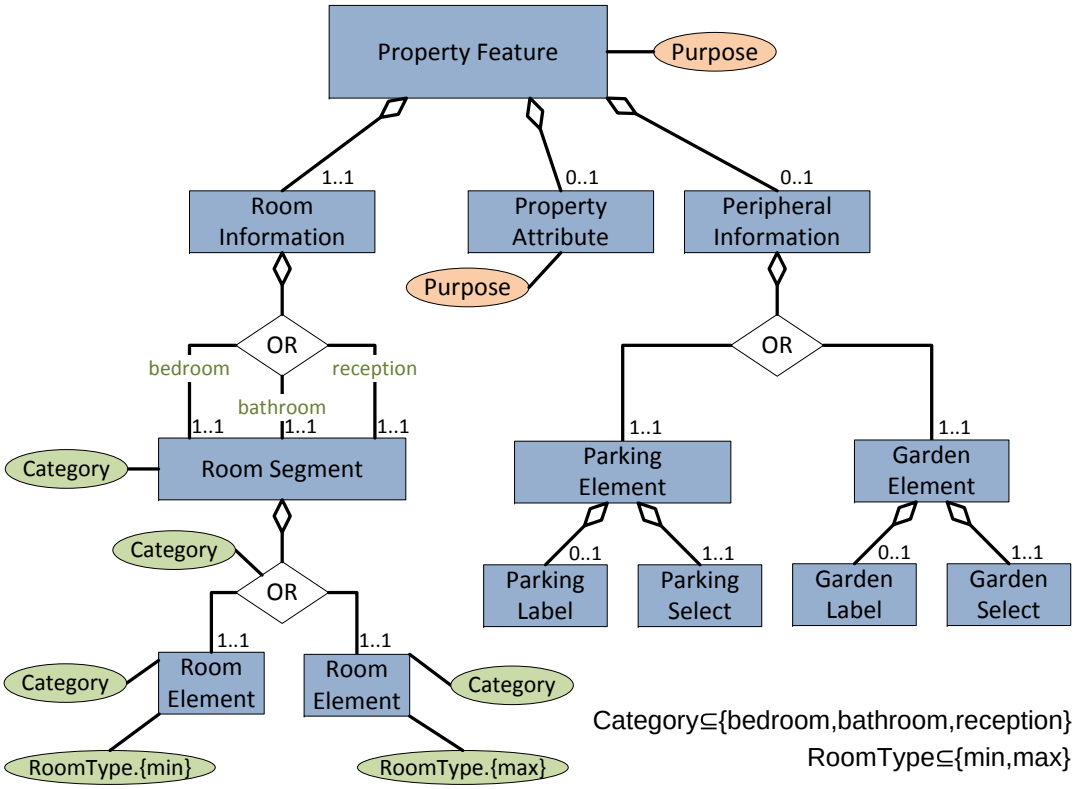
Figure 6.8: Geographic Search Facility Example

Property-Feature Segment

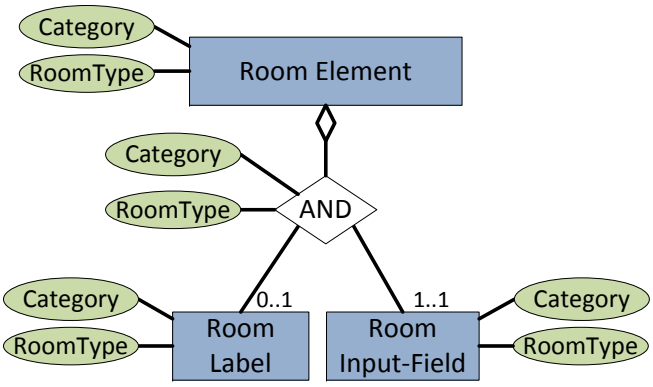
A property-feature segment (Figure 6.9) contains segments for specifying room requirements, property attributes (e.g., house type, age), and peripheral information (e.g., house with garden or car park), with the latter two being optional. The segment for room information consists of one to three room segments, i.e., a subset from room segments with category bedroom, bathroom, and reception room. A room segment has min and max room elements. The basic construction, room element, carries attribute category and room type. Figure 6.10 presents two examples of web forms containing room information. Example 6.10(a) has three room elements, for category bedroom, bathroom and reception room and all three have value min for the room-type attribute. Example 6.10(b) provides two room elements, both for category bedroom, but the first with value min and the second with value max for room-type. A peripheral information segment may contain a parking element and a garden element, each consists of the corresponding label and field.

Property Attribute segment is separately illustrated in Figure 6.11. It may have any but at most one of the following elements.

- *Property type-style element.* It allows users to state preferred house type or style. Type of a property selects over choices such as house, flat, apartment, and so on. Style of a property usually gives several options of a particular type, e.g., a house can be detached, terraced, etc. Some real-estate agents mix the two on their forms.



(a)



(b)

Figure 6.9: Ontology of Property-Feature Segment

Hence, in the ontology a property type-style element has facet type-style with value type, style, or mixed.

- *Property age element.* In UK real-estate domain, age refers to the period that a house is built within, for example, the Victorian period.
- *Property purpose element.* Users specify the purpose of the properties they are

(a) (b)

Figure 6.10: Room Number Search Facility Example

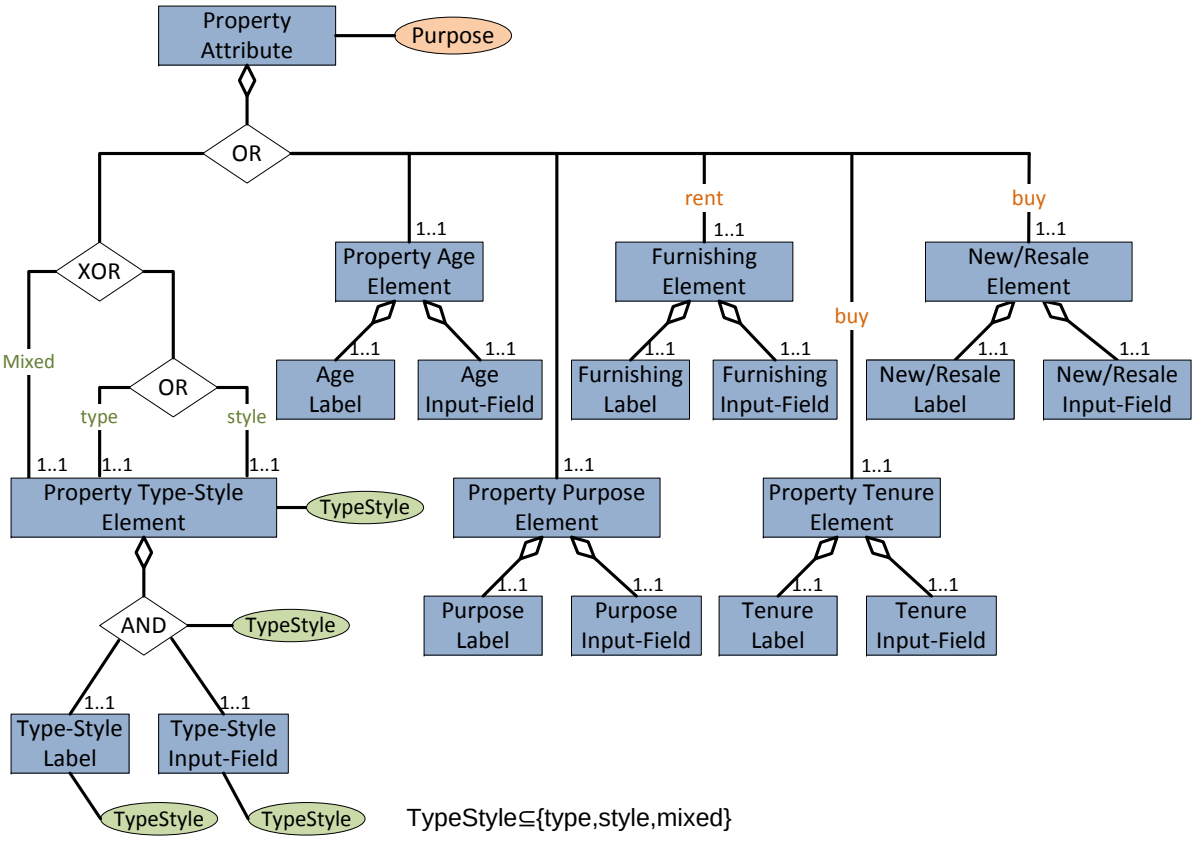
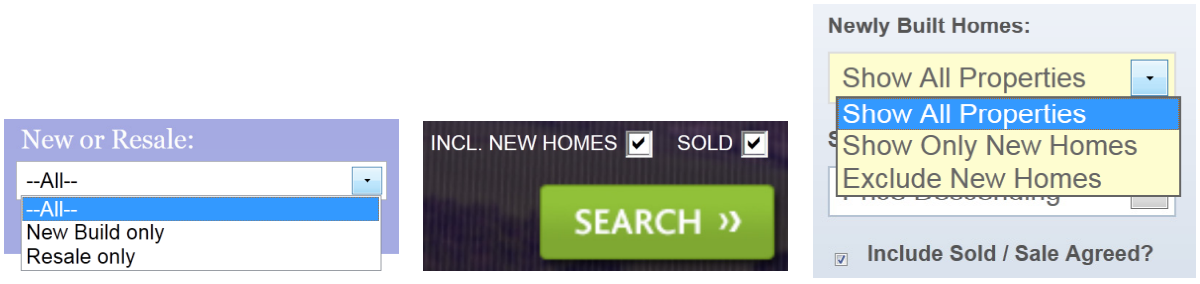


Figure 6.11: Ontology of Property-Attribute Segment

looking for with this element, e.g., residential or commercial.

- *Furnishing element.* It provides choices between furnished and unfurnished if the search aims at renting. Hence, if a furnishing element occurs on a form, we infer that the purpose facet takes the value rent.

- *Property tenure element.* A property can be leasehold or free hold. This information



(a) (b) (c)

Figure 6.12: New/Resale Selector Example

is interesting only to property buyers. Therefore, we assign the purpose facet with value buy if this element is present on the form.

- *new-resale element.* It provides users with options to include or exclude new or resale properties. For example, Figure 6.12(a) allows new or resale searches, 6.12(b) gives the option to include new homes, while 6.12(c) can choose to display only or exclude new homes.

Property Contract Segment

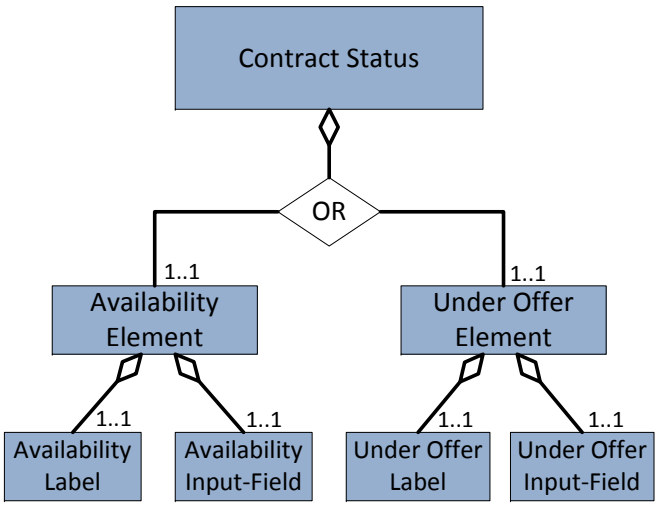


Figure 6.13: Ontology of Property-Contract Segment

A property contract segment (Figure 6.13) consists of an availability element and an under offer element. The former helps with indicating search parameters corresponding to the availability of a property, i.e., available, sold, and sold subject to contract (SSTC). An example can be found in Figure 6.12(b). The latter restricts the search to look for houses that are currently under offer.

Search-Option Segment

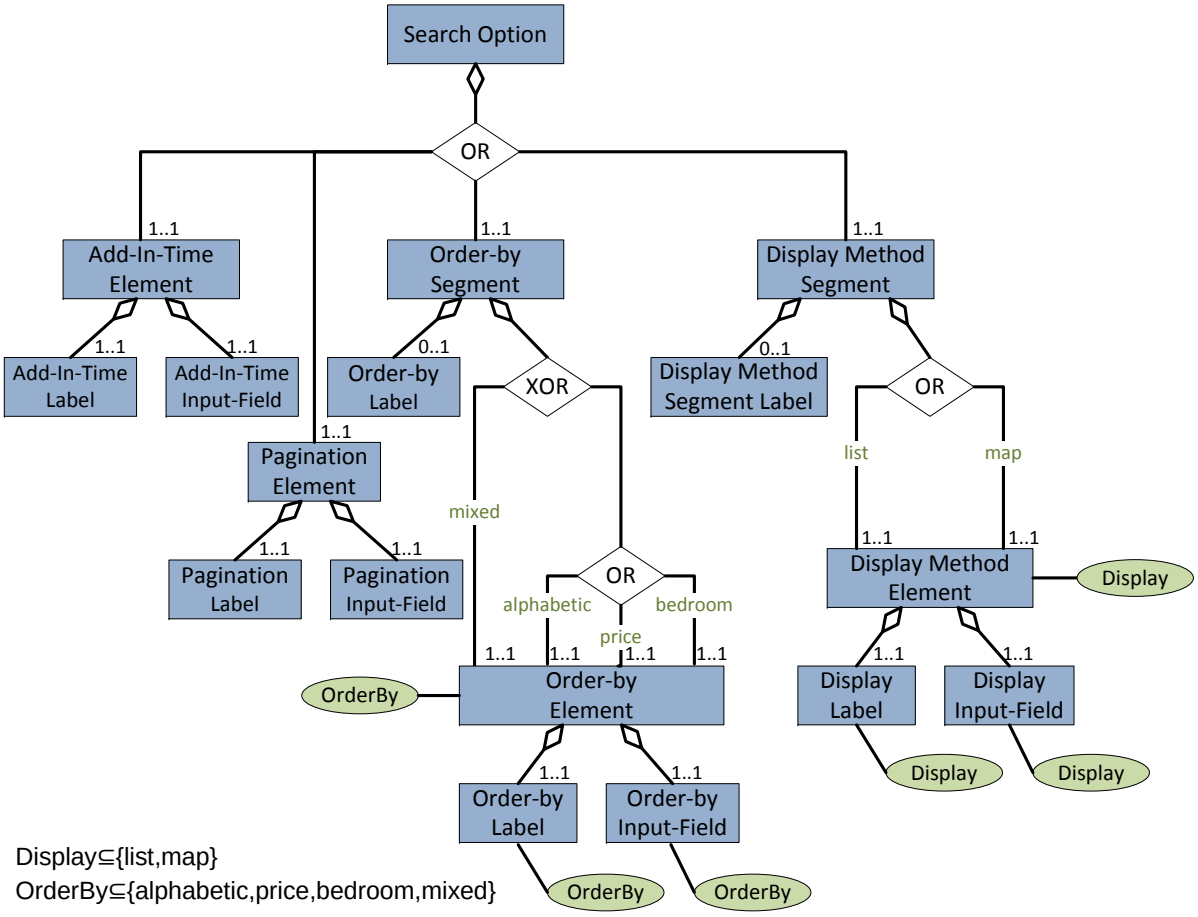


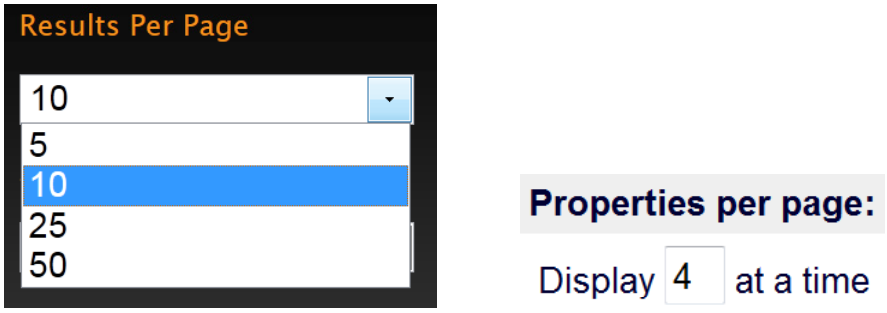
Figure 6.14: Ontology of Search-Option Segment

A real-estate web form often provides various search options for displaying search results (Figure 6.14).

- *Pagination element.* It defines the number of results per page. Two examples are given in Figure 6.15. The first is implemented with a select box and the second uses a text input.

- *Add-in-time Element.* Sometimes users are only interested in houses recently added to the websites. Add-in-time element enables users to search for houses added within a particular period (see Figure 6.16 for an example).

- *Order-by segment.* It defines the results ordering. It may have a segment label and one or more order-by elements. A order-by element can be a drop-down menu containing different ordering choices or a single field for ascending or descending ordering on one criterion, i.e., alphabetic, price, bedroom. These correspond to the



(a) (b)

Figure 6.15: Pagination Element Example

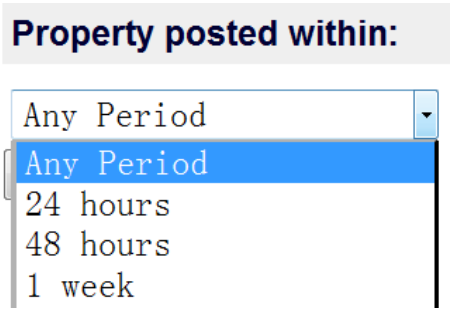
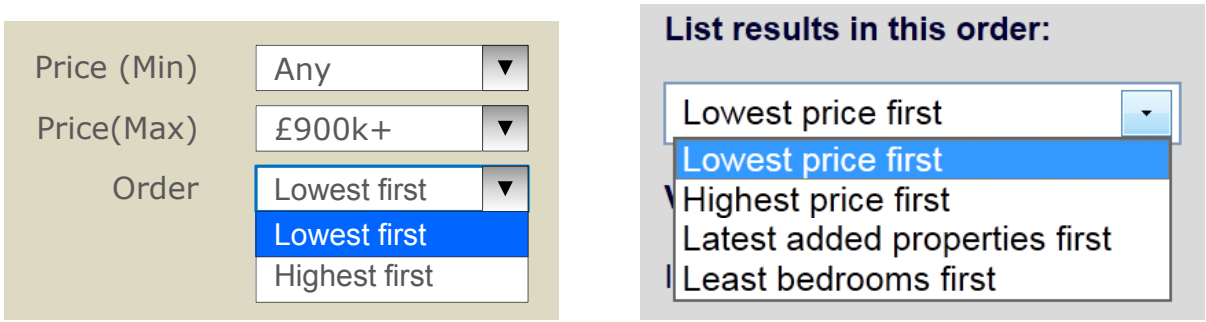


Figure 6.16: Add-in-time Element Example

four possible values of the order-by facet associated with the element. For example, Figure 6.17(a) allows ordering with each search criterion separately (here is the price), whereas 6.17(b) lists out all allowed options for sorting with price, latency of the properties, and bedroom.



(a) (b)

Figure 6.17: Order By Selector Example

- *Display-method segment.* In UK real-estate domain, users can choose to view the result as a list or on a map, represented in the ontology by facet display attached to display-method element. Figure 6.18 gives an example of the segment with two elements, one for list, the other for map.

Figure 6.18: Display Method Example

Form Button Segment

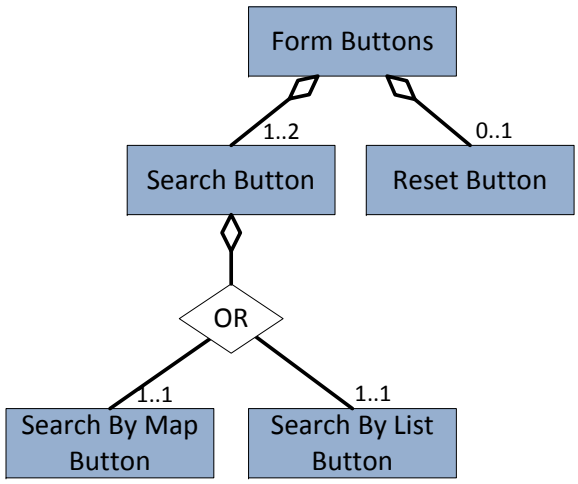


Figure 6.19: Ontology of Form-Button Segment

A form-button segment (Figure 6.19) consists of only buttons for various purposes. Note that a button is not necessarily an `<input>` or a `<button>` element, it can be any scripted HTML element that serves as a button, e.g., a scripted hyperlink. In the UK real estate domain, there are only submit and reset buttons, and a submit button sometimes also defines the result display method, i.e., by map or by list.

6.2.2 Mapping from Domain Web Forms to the Ontology

As an example, we use the ontology for UK real estate web forms to annotate the *Heritage* web form as presented (see Figure 6.20). For illustration, only geographic and price segments are shown. A more complicated example has been provided in Figure 1.5 of Section 1.2.

The construction of a form model for a given domain web form starts from recognizing element-level concepts, e.g., price element, and then continues with the buiding process of grouping smaller modules into larger segment-level concepts, until an instance of the root concept, i.e., real estate web form, is reached. As the element-level concepts are the basis, we conduct an evaluation (with 60 domain web forms) on how well our ontology fits

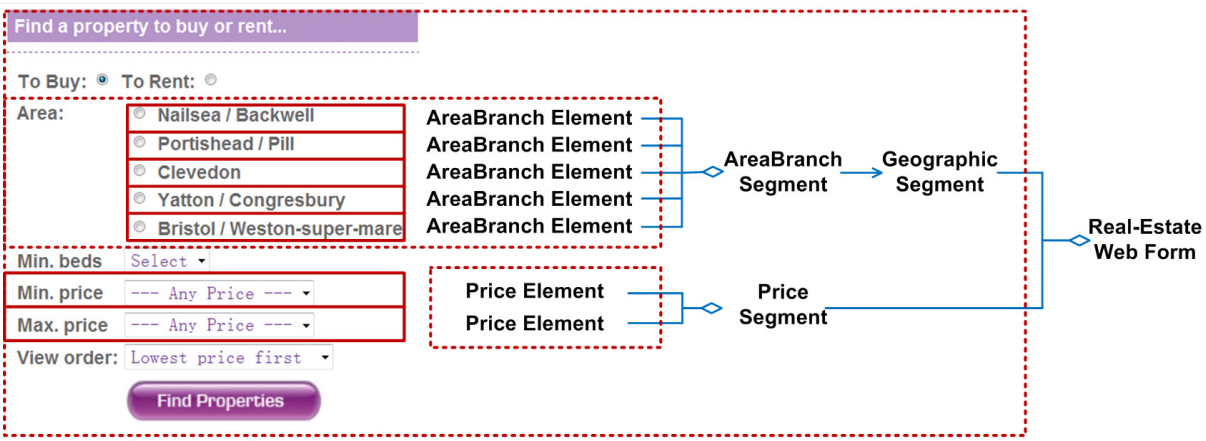


Figure 6.20: *Heritage* Simplified Form Model

to the existing domain web forms by mapping physical form elements to the element-level concepts. We successfully mapped every form element to a concept in the ontology.

This mapping evaluation also reveals design preferences among forms in the real estate domain. We consider element concepts and the combination of these concepts with their attribute values (if they exist). In addition, we also monitor how these concepts are physically implemented, e.g., with text inputs or drop-down lists. In total, we fix a set of 40 domain objects for evaluation, the list of which is provided in Appendix B.

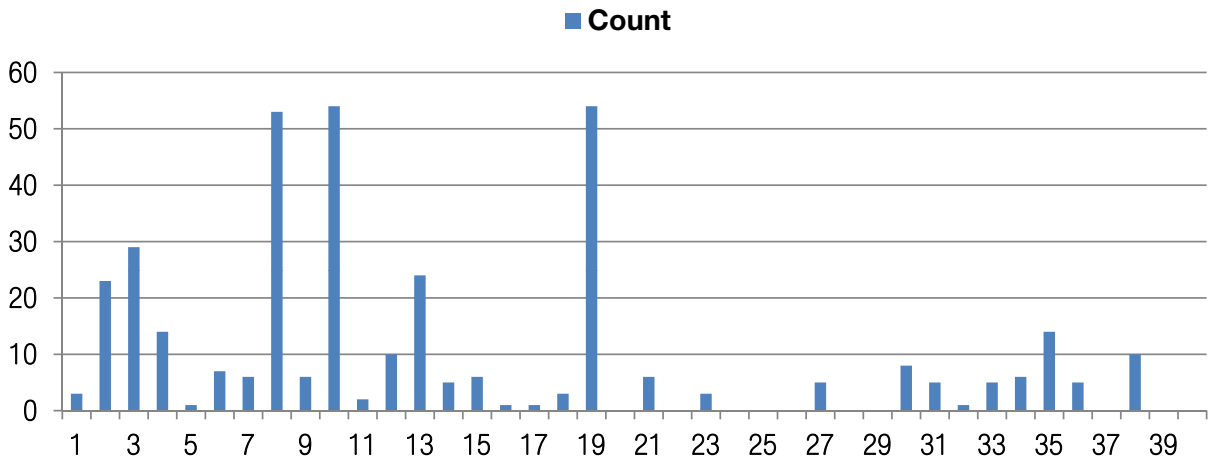


Figure 6.21: Number of Occurrences for Domain Elements

Figure 6.21 shows the number of occurrences of each object in the 60 web forms. Minimum and maximum prices implemented with drop-down lists (8 and 10) are the most popular ones. More than 50 web forms contain these two objects. It is obvious from the figure that certain physical presentations are preferred to others for certain domain concepts, for example, comparing minimum price using drop-down list (8) with using input field (7), or comparing those for maximum price (10 vs 9) and minimum bedroom

number (19 vs 18). Drop-down lists are clearly preferred over text input for these three concepts. Another observation is that some search criteria appear more often that others. For instance, the number of minimum bedrooms (18 and 19) occurs much more often than minimum bathrooms (22 and 23) or reception rooms (26 and 27).

We also summarize the availability of several main segment-level objects, see Figure 6.22. In the domain of UK real estate web forms, top search criteria provided are location, price, and room numbers.

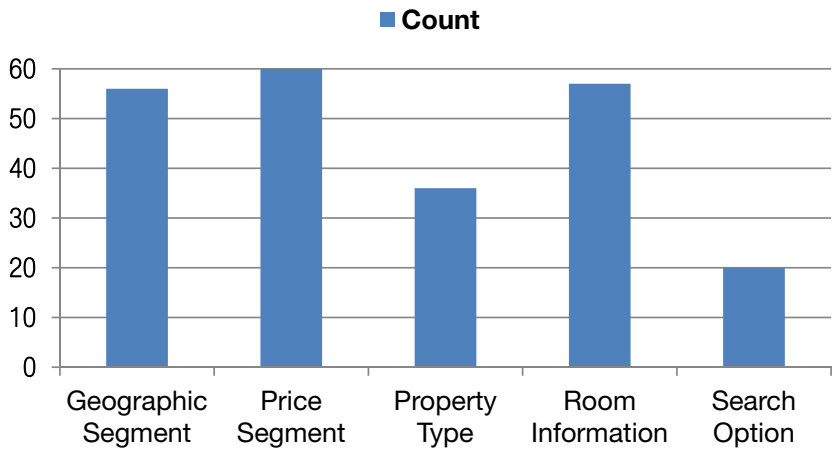


Figure 6.22: Number of Occurrences for Domain Segments

Chapter 7

Evaluation

OPAL implements all the heuristics as declarative rules, using DLV [45] as reasoning engine. Also, we embed the Mozilla as the live browser for web pages and represent the DOM as a relational data model. The linguistic annotations are generated by GATE.

We perform experiments on several domains across four different datasets. Two of them are random selection of web forms from the UK real estate and used car domains respectively. The other two are publicly available datasets as benchmarks for form understanding. We use the latter to compare with existing approaches, on which we only evaluate OPAL's domain independent form labeling for fair comparison, as these approaches only label forms and do not use domain knowledge. We also conduct an introspective analysis on the impact of each of OPAL's four scopes and the performance and scalability of OPAL.

The experiments focus on field labeling. We measure the proper assignment of text nodes to form fields by precision P (correctly labeled fields over total labeled fields), recall R (correctly labeled fields over total number of fields), and F_1-score (the harmonic mean $F_1 = 2PR/(P + R)$ of precision and recall). For all four datasets, we compare the form understanding result to a manually constructed gold standard. The effectiveness of segmentation can be revealed from its impact on the classification and the improvement from benchmark datasets (where no form interpretation is performed) to the two domain dependent datasets.

In this chapter, we first introduce in detail the datasets for the experiments. Then we focus on OPAL's field labeling accuracy on these datasets. We also provide a cross domain validation on the benchmarks to demonstrate the versatility of OPAL and show that even without domain knowledge OPAL outperforms existing approaches by over 5%. At the end of the chapter, we present the contributions of OPAL's scopes and the scalability of our approach.

7.1 Evaluation Datasets

We select two domains for domain dependent experiments. For the UK real estate domain, we build a dataset by randomly selecting 100 real estate agencies from the UK yellow page.[1] For the used car domain, we randomly select 100 used car dealers from Autotrader[2] the UK largest aggregator in the domain.

The two benchmarks, ICQ and Tel-8, are taken from the University of Illinois at Urbana Champaign Web Repository.[3] ICQ presents web forms from five domains: air travelling, car dealer, book, job, real estate. There are in total 100 web pages with 20 for each domain. However, 2 pages are no longer available and thus excluded from the evaluation. The Tel-8 dataset contains web forms from eight domains: books, car rental, jobs, hotels, airlines, auto, movies, and music records. The dataset amounts to 477 forms, but only 436 of them are still accessible.

The two benchmark datasets were both created in 2003. We observe significantly different characteristics between the two domain datasets we created and the two benchmarks, especially for more fashionable domains, e.g., car domain. This is mainly due to changes in web technology and web design practices, such as the usage of CSS stylesheets for layout and AJAX features.

7.2 Field Labeling Accuracy

We evaluate OPAL's accuracy in field labeling on all four datasets. For the two benchmarks, the evaluation is conducted on OPAL's domain independent analysis. Only for the real estate and used car datasets, we employ OPAL's form interpretation to further improve the labeling.

Figure 7.1 shows the results for all four datasets. The first two bars represent the results for the two domain datasets. OPAL achieves perfect precision and 98.6% recall for the real estate domain, with an overall F-score of 99.2%. For the used car domain, the result is slightly lower, with 98.2% precision and 99.2% recall, which amount to 98.7% F-score. The result is affected in this domain due to the highly interactive design of the forms. For example, it is often the case that users are allowed to select the model of a car only after they choose the car manufacturer, which is reflected on the forms by enabling or even creating the model field after the manufacturer selection. However, as OPAL so far processes forms statically, the placeholder for the model field before it appears introduces noise to field labeling and thus classification.

[1] `www.yell.com`
[2] `www.autotrader.co.uk`
[3] `metaquerier.cs.uiuc.edu/repository`

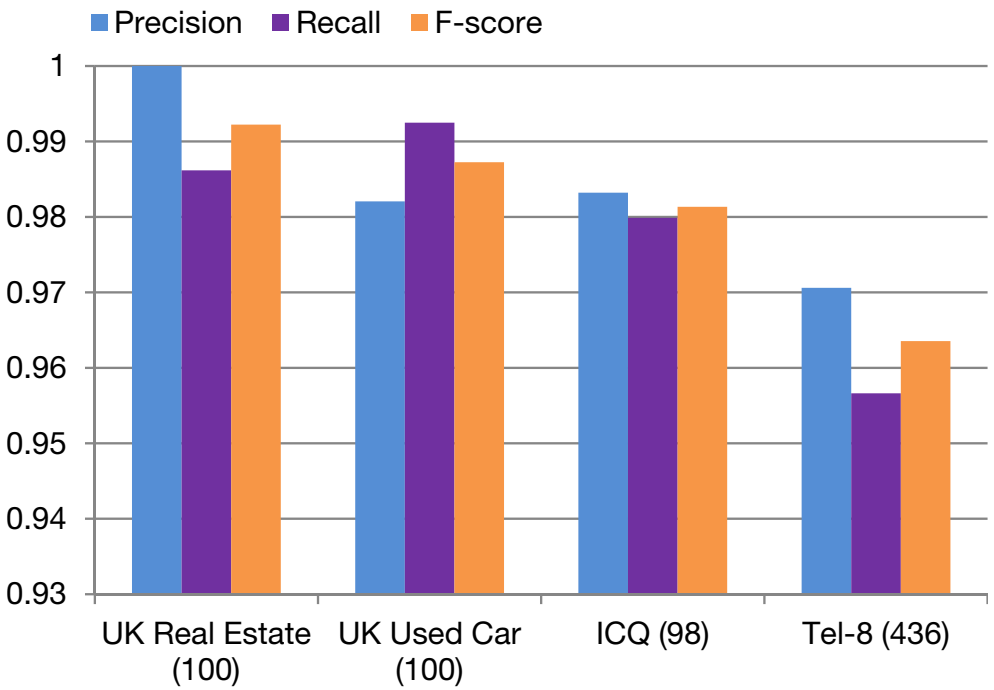


Figure 7.1: OPAL on 734 Forms

The domain schema for the real estate domain consists of 40 concepts and their corresponding constraints. In the used car domain, there are about 30 domain concepts. In our experience, creating an initial domain schema (including gazetteers and testing) for a domain takes a single person familiar with the domain and OPAL-TL roughly a week. In fact, domain experts can provide common concepts involved for typical domain objects. These may be used as the bootstrap of schema design for domain web forms.

The third and fourth bars in Figure 7.1 presents the result of field labeling on ICQ and Tel-8 datasets. Worth mentioning that, although both benchmarks contain real estate and car domains, these are actually different domains from the ones we analyzed in the United Kingdom. For example, British agencies seldom sell houses based on price per square meter, and thus size is rarely provided as a search criterion. However, the real estate forms in the benchmarks cover mainly the US market, where size is a popular characteristic. Hence, on the two benchmarks, OPAL applies without form interpretation and relies only on its domain independent scopes (field, segment, layout). Nonetheless, OPAL reports high accuracy (overall above 95%) also on these forms, confirming the effectiveness of our domain independent analysis. Yet not unexpected, comparing with the two domain datasets, OPAL performs significantly better in presence of domain knowledge.

Although on the two domain datasets OPAL produces high quality results, there are classes of forms that can be considered as outliers. Among them, the most prominent has the distinctive feature of using images to label form field. The form on the O'Connor

Kennedy Turtle agency,[4] for instance, uses images for all its fields, although this form is solved by OPAL using attribute values associated with the fields themselves. Without presence of such field attributes, a straightforward extension of our heuristics could search for alternative HTML elements (other than ones containing texts) that may also serve as label, such as `<img>`, and by analyzing the attributes associated with these elements to obtain field classification. However, this does not work when such values are missing.

7.3 Cross Domain Evaluation

We use the two benchmarks to compare OPAL's performance of field labeling against existing approaches. Both benchmarks contain various domains. OPAL achieves over 98% for ICQ and over 95% for Tel-8 as illustrated in Figure 7.1.

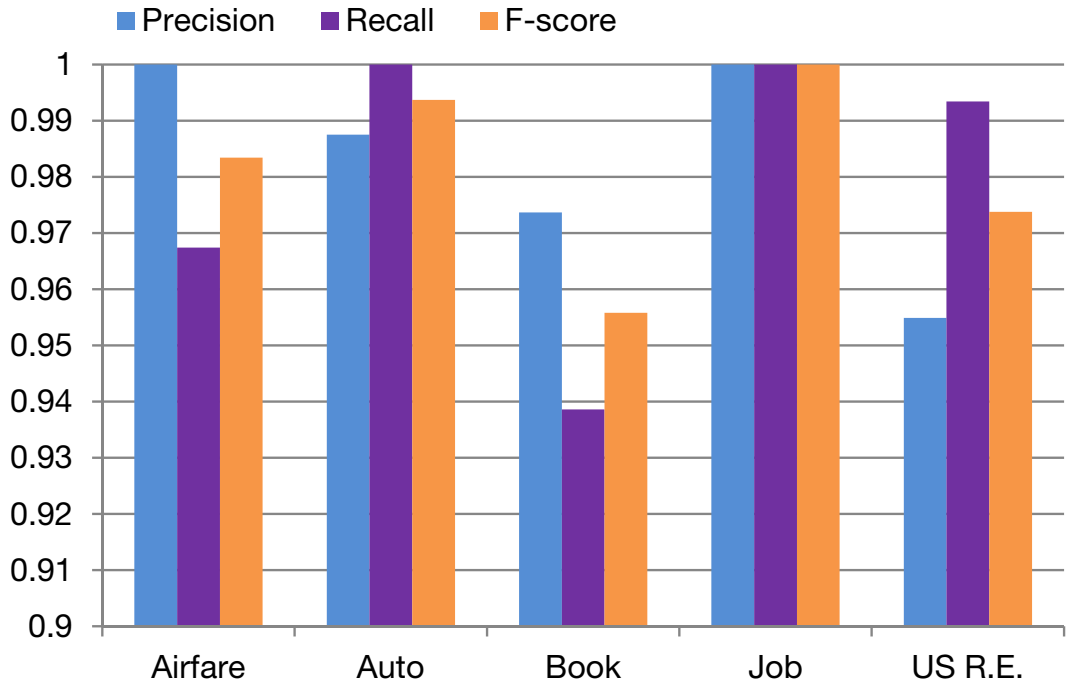


Figure 7.2: ICQ Results

Figure 7.2 illustrates the results on each domain of the ICQ dataset. OPAL shows remarkable F-score values for the job domain (100%) as well as auto (99.3%) and air travelling (98.2%). For comparison, [19], a visual based approach, reports 92% F-score for labeling on average, which we outperform even in the most difficult domain (books). [66] reports slightly better precision (92.1%) and recall (93.9%), but OPAL still outperforms it by several percents.

[4] `www.okt.co.uk/advanced_search.php`

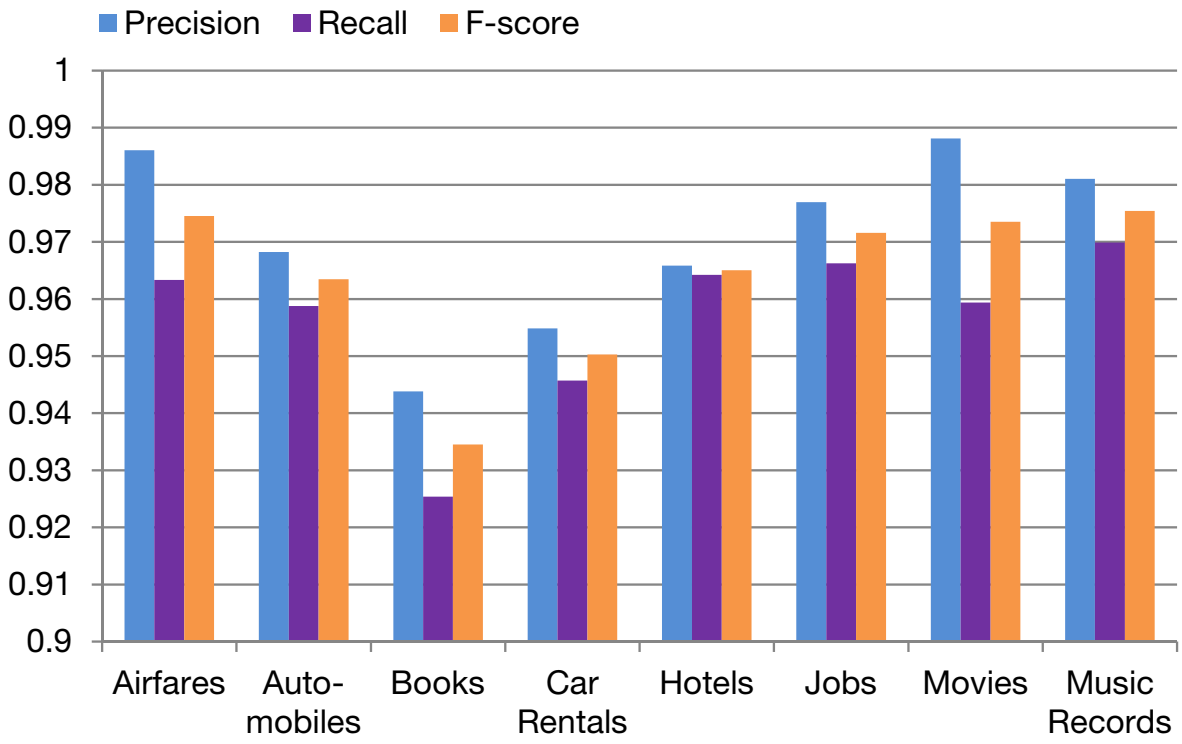


Figure 7.3: Tel-8 Results

The results for the Tel8 dataset are depicted in Figure 7.3. Here, the overall F-score is 95.1%, which is mostly affected by the performance in the book domain. Note that, especially on Tel8, OPAL obtains very high precision compared to recall. Indeed, lower recall means OPAL is not able to assign labels to all fields, missing the labeling for some of them. For comparison, [19] reports $88-90\%$ overall F-score, which we again outperform by a wide margin even on the worst domain. [52] reports F-scores between 89% and 95% for four domains in the Tel8 dataset. Though they perform slightly better on books, we outperform them on the three other domains included in their results.

We further investigate OPAL's performance in the book domain of both benchmark datasets to figure out the reason for the low recall. It turns out that, in the book domain datasets provided, (i) forms are frequently designed poorly by abusing the HTML `<table>` element, which leads to failure of structural heuristics; (ii) texts are not surrounded by dedicated HTML element, with which our current browser model is limited to calculate the CSS boxes for such texts, and hence leads to failure of visual analysis.

7.4 Scope Contribution

We demonstrate the effectiveness of combining different types of features during the analysis by measuring to what extent each of our four scopes contributes to the overall

quality of form understanding, again focusing on field labeling and classification. We use the two domain datasets from the previous experiment, and for both we evaluate the recall.

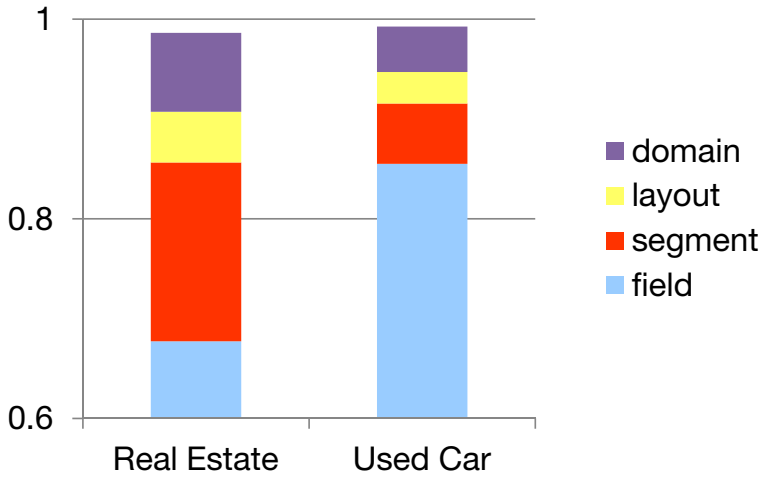


Figure 7.4: Scope Contribution

As illustrated in Figure 7.4, in the real-estate dataset, the field scope already contributes significantly (67%). The Segment scope increases by 18%. Layout and domain scope add together another 13%. In the used car domain, field scope alone brings even more significant impact with 85%, together with each of the other three scopes contribute towards the near perfect result.

The great impact of structural scopes indicates the form design practices nowadays. People tend to consider more the semantic relationship between fields, which is reflected not only by the visual presentation but also through the underlying structural design.

The domain scope also produces considerable contribution for OPAL to achieve such great result. This indicates the importance of domain knowledge in form understanding. This is also proved by our preliminary work (Section 3.2): there is a 2% gain with the integration of a basic layer of domain knowledge (with less than 10 concepts).

The experiment with the prototype involves testing on structural heuristics only and the domain. With only structure-based analysis, OPAL achieves 96% correct labeling and 93% correct segmentation. With the integration of a basic layer of domain knowledge, the two figures are improved to 97% and 95% respectively.

7.5 Scalability

As presented in Chapter 4, OPAL's overall analysis is bounded by $O(n^2)$ due to the layout scope. As expected, OPAL's actual performance follows a quadratic curve, but

with very low constants. There is a significant amount of outliers, partially due to long page rendering time and partially due to variance in the depth and sophistication of the HTML structure.

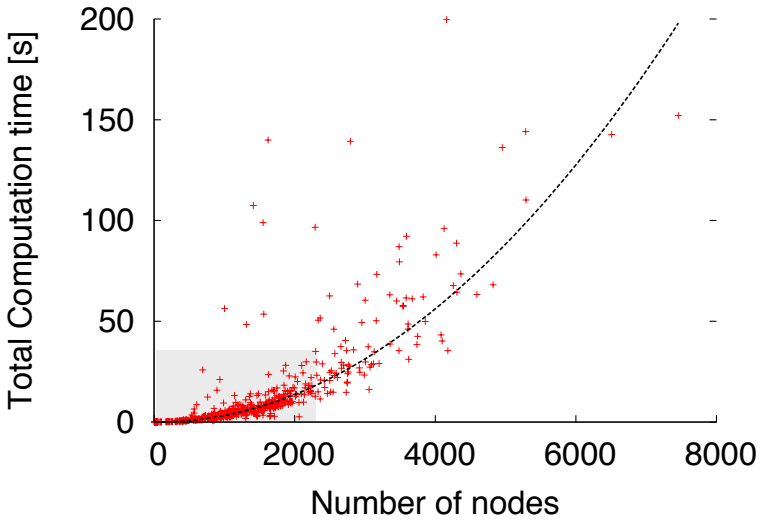


Figure 7.5: Running Time

Figure 7.5 reports OPAL performance (including page rendering time) on all 534 forms in the Tel-8 and ICQ benchmarks. The highlight area covers 80% of the forms with 2200 nodes. It requires at most $30s$ for page rendering and the analysis of OPAL on these forms. Further study on the effect of increasing field or form numbers confirms that these have little effect and page size is the dominant factor.

Chapter 8

OPAL-Assisted Form Filling

Given a form, a human can easily figure out the meaning and functionality of each field by associating textual labels to the corresponding fields and by grouping logically and visually coherent fields. Once reaching such high-level understanding, one can fill the form accordingly.

However, this is not a trivial task for machines, and the main bottle neck is proper form understanding. In this chapter, we present an assisted form filling system based on our domain-aware form understanding approach, OPAL. With the system, we demonstrate the achievement of OPAL at each of its scopes, and fill in the domain form using OPAL's analysis results.

8.1 Automated Form Filling

Automated form filling involves both automated form understanding and automated filling. Given a domain web form, OPAL achieves form understanding by analyzing the form and produces a form model consistent with the domain schema, we then use OPAL's result to actually fill the form.

Here is a scenario as a motivation for automated form filling: one needs to search on every single Oxford real estate website for a house (in Oxford with 2 bedrooms and costs no more than £300,000) before he or she can be sure that they find the best offer. However, existing auto-fill toolbars (e.g., Mozilla Firefox Autofill add-on, Google Toolbar Autofill) work mainly for personal information (by asking users to fill in a pre-defined master form, the values of which are then used to fill in targeted forms).

Similar to these auto-fill toolbars, OPAL also provides users with a master form of a particular domain, where users input their requirements of domain objects. OPAL then fills each domain form with the values provided based on its understanding of the form.

Figure 8.1 presents the interface for our automated form filling system. In the master form, the domain field can be switched between real estate and used car. By changing

Figure 8.1: OPAL GUI

this value, the master form changes accordingly to cover the most popular concepts of the switched domain. Users fill in the master form with their search requirement. The confirm button tells the system to store the values provided. During an actual run, the system (i) loads the web page of a given URL into the browser, (ii) runs OPAL's form

understanding analysis, and (iii) fills the form if a form in the domain specified in the master form is found. With the current implementation, the filling is based on loaded values for the master form (as previously confirmed by users). Newly specified values are used only in the next run.

In case of free text fields, OPAL fills in the values directly. In case of drop down menus, or lists of check boxes or radio buttons, OPAL tries to find the best matching one. OPAL compares the values occurring on the form with the values provided with text matching. Particularly, it handles different formats for price representation, and selects min max values that cover the specified price range. However, if no match is found, OPAL highlights the field and allows users to choose manually.

The interface also allows browsing through OPAL's form understanding results (the top panel), i.e., the labeling obtained at each scope and the classification.

8.2 Form Filling Evaluation

The performance of current system depends mostly on OPAL's form analysis, which proves to be highly effective with an overall of 95% accuracy. Nevertheless, we did a preliminary evaluation on the form filling process. We randomly select 50 web pages with 25 from each of the two domains, and observe no case where OPAL understands a form correctly, but does not fill it successfully. However, as OPAL performs static analysis on a given form, the system is currently limited on a few cases where the form changes dynamically or uses heavily scripted UI elements.

Currently, OPAL does not incorporate any automatic validation mechanism to judge its form understanding efficacy on an arbitrary form. Thus, it cannot avoid filling in wrongly understood forms. However, as a form filling assistant, the current system fills a form as understood, and allows users to manually adjust any inappropriate fillings before submitting the form.

Chapter 9

Conclusion and Future Work

OPAL is an approach on automated web form understanding, and pushes the state of the art significantly by combining various classes of features during its domain independent form labeling. More importantly, it differs from existing approaches in the problem definition, where it integrates domain knowledge to achieve a domain-aware form interpretation. We provide a template language OPAL-TL for domain schema design together with a template library of common design patterns.

1. OPAL employs a domain independent form labeling that expands through three scopes. At field scope, OPAL considers structural relations between texts and individual fields, taking care of both information explicitly provided by the DOM and the local field structure. At segment scope, OPAL groups fields into a hierarchy (segment tree) both by exploring the similarities between neighboring fields and by integrating information from a cleaned-up DOM tree. The labeling is then derived from the text-field patterns identified in the segments. At layout scope, OPAL further expands the analysis to the entire page and analyzes the visual relationships between fields and texts. OPAL's three domain independent scopes exploit different classes of features during the analysis and produce a very accurate labeling. OPAL achieves overall 95% accuracy in a cross domain evaluation.

2. OPAL is domain-aware in producing a form interpretation consistent with a given domain schema. Different from existing approaches which stop at domain independent form labelings, OPAL integrates high-level domain knowledge to classify textual annotations on the labels and to complete the form model according to classification and structural constraints of the domain schema. OPAL's form interpretation pushes the analysis result towards near perfect accuracy ($> 98\%$).

3. To specify domain schemata, OPAL adopts a template language OPAL-TL which extends Datalog with templates. It allows compact and declarative expressions of common form design patterns as parameterizable templates. We provide a library to maintain

these templates, which enables OPAL's easy adaption to new domains, often only requiring template instantiations with domain specific annotation and concept types. To implement a new domain, one only needs to provide a set of textual annotators and an OPAL-TL specification of domain schemata consisting of the domain types together with their classification and structural constraints.

OPAL's multi-scope analysis is implemented with declarative rules to allow easy modification and natural access to domain knowledge, addressing many limitations in previous research work. In extensive experiments, the most reliable field scope contributes over 60%. Segment scope and layout scope contribute around 20% and 10% respectively. Overall, OPAL improves upon the strongest competitive results by at least 5% accuracy with its domain independent analysis alone. Domain knowledge further contributes significantly to produce an almost perfect form understanding.

The application sequence of the domain independent scopes is important. It represents our confidence in the different types of features being exploited. It also allows OPAL to avoid complicated heuristics or algorithms in its design. For example, compared with the purely visual-based approaches [19, 66, 68], instead of dozens of observations on visual appearances, OPAL exploits only basic visual relationships (e.g., texts to the top right of a field). However, OPAL applies its visual analysis with the knowledge of successful labeling in the previous two scopes. Hence, OPAL resolves those cases where complex visual analysis is necessary by handling them with structural analysis at earlier steps.

Automated form filling is one application of OPAL: Based on its form analysis, OPAL continues with filling the forms with user-specified domain values. It fills successfully all correctly interpreted forms, leading to practically perfect results given our highly accurate form analysis.

Limitation and Future Work

OPAL has achieved near perfect form understanding incorporating a multi-scope domain independent form labeling and a domain dependent form interpretation, yet it is limited in several respects.

OPAL performs static analysis on forms, which is insufficient with the current trend of dynamic web form design. To handle dynamic form changes, OPAL currently requires rerun of the complete form understanding process on the changed web page after each individual action. This is a huge waste of resources. The limitation may be addressed in two ways. One way is to detect changes on the form and combine the analysis result of these changes with the original form understanding results. However, this requires modification of the presentation for the part of DOM tree corresponding to the changes.

There we seek for a more capable yet efficient way of tree encoding scheme (instead of the basic start-end encoding) that allows modifications to the changed part only without affecting the rest. The other way is to employ dynamic analysis in the form understanding process, for example, through Javascript analysis, which may not be as reliable as taking the action for real, as form labeling is limited (both structurally and visually) without having the actual web pages.

As discussed in Section 7.3, OPAL's performance is affected (with lower recall) in the book domain in both benchmark datasets due to the poor design of the forms in this domain. To solve this problem, we can improve the browser model by calculating the CSS boxes for each text node that has no dedicated HTML element. The calculation can be achieved by cutting off the sibling boxes of the text node from their parent box, or by inserting an HTML element for each such text node and the box calculation remains the same as for other nodes.

Currently, all experiments on OPAL's performance are compared against a manually defined gold standard. However, in real world applications, it is necessary to have a verification mechanism to automatically judge how well a form is understood. We may give confidence to the label-field associations, based on the scopes and the information used to generate them. Confidence may be associated with the form interpretation as well, by considering both confidence in text annotations and the amount of repairs OPAL performs to obtain the form model. We may establish a comprehensive scoring system to calculate a score for every single labeling or interpretation result OPAL generates. Verification may also be addressed by linking form understanding with result page analysis through probing, which requires contributions from other research areas (e.g., result page analysis) rather than purely form understanding.

Incorporation with other research areas, such as result page analysis, may also enhance OPAL's overall form understanding performance.

To further improve the accuracy and robustness, we plan to extend OPAL by integrating more "scopes'. In particular, we are developing a site scope to perform form-form and form-result alignment.

Many websites contain multiple forms, e.g., simple, advanced, or refinement forms. Instead of treating these forms in isolation, we may align them to combine the clues found on individual forms.

Even more promising is alignment with result page models, as produced by another sub-task of DIADEM [25]. Particularly in combination with probing, i.e., test submissions of a form, this allows us to guess the type of previously untyped fields, to verify assumptions, and to further improve the form model.

To allow probing, the form model can be expanded with information relevant to form submission, e.g., integrity and access constraints. These may be derived by analysis on the Javascript code for client side validation, or derived directly from interaction and submission of forms.

To minimize the effort in domain adaption, we seek for effective approaches for ontology creation, based on which, OPAL may generate classification and structural constraints automatically into OPAL-TL rules. To this end, OPAL may also improve the existing domain schema semi-automatically by verifying the existence of domain types if untyped fields or segments occur.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Samur Araujo, Qi Gao, Erwin Leonardi, and Geert-Jan Houben. Carbon: domain-independent automatic web form filling. In *Proceedings of the 10th International Conference on Web Engineering*, ICWE '10, pages 292–306, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Ziv Bar-Yossef and Maxim Gurevich. Random sampling from a search engine's index. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 367–376, New York, NY, USA, 2006. ACM.

[4] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 441–450, New York, NY, USA, 2007.

[5] Luciano Barbosa and Juliana Freire. Combining classifiers to identify online databases. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 431–440, New York, NY, USA, 2007. ACM.

[6] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 119–128, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[7] Michael Benedikt and Christoph Koch. Xpath leashed. *ACM Computing Survey*, 41:3:1–3:54, January 2009.

[8] Sidi Mohamed Benslimane, Mimoun Malki, Mustapha Kamal Rahmouni, and Djamal Benslimane. Extracting personalised ontology from data-intensive web application: an html forms-based reverse engineering approach. *Informatica, Lithuanian Academy of Sciences*, 18(4):511–534, 2007.

[9] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) Version 2.0. Recommendation, W3C, 2010. `http://www.w3.org/TR/xpath20/`, checked 2011/07/10.

[10] Michael K. Bergman. White Paper: The Deep Web: Surfacing Hidden Value. *The journal of electronic publishing*, 7(1), 2001.

[11] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 69–80, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semantics*, 7(3):154–165, September 2009.

[13] Michael Cafarella, Edward Chang, Andrew Fikes, Alon Halevy, Wilson Hsieh, Alberto Lerner, Jayant Madhavan, and S. Muthukrishnan. Data management projects at google. *SIGMOD Record*, 37(1):34–38, March 2008.

[14] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured databases on the web: observations and implications. *SIGMOD Record*, 33:61–70, September 2004.

[15] Kuang Chen, Harr Chen, Neil Conway, Heather Dolan, Joseph M. Hellerstein, and Tapan S. Parikh. Improving data quality with dynamic forms. In *Proceedings of the 3rd International Conference on Information and Communication Technologies and Development*, ICTD '09, pages 487–487, Piscataway, NJ, USA, 2009. IEEE Press.

[16] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. HTML Techniques for Web Content Accessibility Guidelines 1.0. Recommendation, W3C, November 2000. `http://www.w3.org/TR/WCAG10-HTML-TECHS/`.

[17] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. Department of Computer Science, University of Sheffield, 2011.

[18] Anhai Doan, Pedro Domingos, and Alon Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Mach. Learn.*, 50(3):279–301, March 2003.

[19] Eduard C. Dragut, Thomas Kabisch, Clement Yu, and Ulf Leser. A hierarchical approach to model web query interfaces for web source integration. In *In Proceeding of International Conference on Very Large Data Bases*, VLDB, pages 325–336, 2009.

[20] Elika J. Etemad. Cascading style sheets (css) snapshot 2007. Working draft, W3C, May 2008. `http://www.w3.org/TR/css-beijing/`, checked 2009/09/16.

[21] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: a survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[22] Tim Furche, Georg Gottlob, Giovanni Grasso, Omer Gunes, Xiaonan Guo, Andrey Kravchenko, Giorgio Orsi, Christian Schallhart, Andrew Sellers, and Cheng Wang. DIADEM: Domain-centric, Intelligent, Automated Data Extraction Methodology. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 267–270, New York, NY, USA, 2012. ACM.

[23] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, and Christian Schallhart. Real understanding of real estate forms. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pages 13:1–13:12, New York, NY, USA, 2011. ACM.

[24] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, and Christian Schallhart. Opal: automated form understanding for the deep web. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 829–838, New York, NY, USA, 2012. ACM.

[25] Tim Furche, Georg Gottlob, Giovanni Grasso, Giorgio Orsi, Christian Schallhart, and Cheng Wang. Little Knowledge Rules The Web: Domain-Centric Result Page Extraction. In *International Conference on Web Reasoning and Rule Systems*, RR '11, pages 61–76, 2011.

[26] Tim Furche, Georg Gottlob, Xiaonan Guo, Christian Schallhart, Andrew Jon Sellers, and Cheng Wang. How the minotaur turned into ariadne: Ontologies in web data extraction. In *International Conferences Workshops and Exhibitions*, ICWE '11, pages 13–27, 2011.

[27] Gargoyle Software Inc. HtmlUnit. Online only, 2010. Retrieved at 14/09/2010.

[28] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[29] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating xpath evaluation in any rdbms. *ACM Transactions on Database Systems*, 29(1):91–131, March 2004.

[30] Xiaonan Guo, Jochen Kranzdorf, Tim Furche, Giovanni Grasso, Giorgio Orsi, and Christian Schallhart. Opal: a passe-partout for web forms. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 353–356, New York, NY, USA, 2012. ACM.

[31] Bin He and Kevin Chen-Chuan Chang. Statistical schema matching across web query interfaces. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 217–228, New York, NY, USA, 2003. ACM.

[32] Bin He, Zhen Zhang, and Kevin Chen-Chuan Chang. Towards building a meta-querier: Extracting and matching web query interfaces. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 1098–1099, Washington, DC, USA, 2005. IEEE Computer Society.

[33] Hai He, Weiyi Meng, Yiyao Lu, Clement Yu, and Zonghuan Wu. Towards deeper understanding of the search interfaces of the deep web. *wwwjournal*, 10:133–155, 2007.

[34] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Wise-integrator: an automatic integrator of web search interfaces for e-commerce. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 357–368. VLDB Endowment, 2003.

[35] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of web search interfaces with wise-integrator. *The VLDB Journal*, 13(3):256–273, September 2004.

[36] Gerald Huck, Peter Fankhauser, Karl Aberer, and Erich J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems*, COOPIS '98, pages 32–43, Washington, DC, USA, 1998. IEEE Computer Society.

[37] Oliver Kalijuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient web form entry on pdas. In *Proceedings of the 10th international conference on World Wide Web*, www '01, pages 663–672, 2001.

[38] Ritu Khare and Yuan An. An empirical study on using hidden markov model for search interface segmentation. In *The 18st ACM International Conference on Information and Knowledge Management*, CIKM '09, pages 17–26, 2009.

[39] Ritu Khare, Yuan An, and Il-Yeol Song. Understanding Deep Web Search Interfaces: A Survey. *SIGMOD Record*, 39(1):33–40, 2010.

[40] Nicholas Kushmerick. Learning to invoke web forms. In *Proceedings of International Conference Ontologies, Databases and Applications of Semantics*, pages 997–1013. Springer-Verlag, 2003.

[41] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *International Joint Conferences on Artificial Intelligence*, 1997.

[42] Alberto H. F. Laender, Berthier Ribeiro-Neto, and Altigran S. da Silva. Debye - date extraction by example. *Data & Knowledge Engineering*, 40:121–154, February 2002.

[43] Juliano Palmieri Lage, Altigran S. da Silva, Paulo B. Golgher, and Alberto H. F. Laender. Automatic generation of agents for collecting hidden web pages for data extraction. *Data and Knowledge Engineering*, 49(2):177–196, May 2004.

[44] Jens Lehmann, Jörg Schüppel, and Sören Auer. Discovering unknown connections - the dbpedia relationship finder. In *Proceedings of the 1st SABRE Conference on Social Semantic Web*, 2007.

[45] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[46] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[47] Zehua Liu, Feifei Li, and Wee Keong Ng. Wiccap data model: Mapping physical websites to logical views. In *Proceedings of the 21st International Conference on Conceptual Modeling*, ER '02, pages 120–134, London, UK, UK, 2002. Springer-Verlag.

[48] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's Deep Web Crawl. In *The Proceedings of the VLDB Endowment*, PVLDB, pages 1241–1252, 2008.

[49] Anirban Maiti, Arjun Dasgupta, Nan Zhang, and Gautam Das. Hdsampler: revealing data behind web form interfaces. In *Proceedings of the 35th SIGMOD international conference on Management of Data*, SIGMOD '09, pages 1131–1134, New York, NY, USA, 2009. ACM.

[50] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the 3rd Annual Conference on Autonomous Agents*, AGENTS '99, pages 190–197, New York, NY, USA, 1999. ACM.

[51] Isabel Navarrete and Guido Sciavicco. Spatial reasoning with rectangular cardinal direction relations. In *Proceedings of the 17th European Conference on Artificial Intelligence*, ECAI '06, pages 1–9, 2006.

[52] Hoa Nguyen, Thanh Nguyen, and Juliana Freire. Learning to Extract Form Labels. In *The Proceedings of the VLDB Endowment*, PVLDB, pages 684–694, 2008.

[53] Thanh Hoang Nguyen, Hoa Nguyen, and Juliana Freire. Prusm: a prudent schema matching approach for web forms. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 1385–1388, New York, NY, USA, 2010. ACM.

[54] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. Wordnet::similarity: measuring the relatedness of concepts. In *Proceedings of Demonstration Papers at HLT-NAACL 2004*, HLT-NAACL–Demonstrations '04, pages 38–41, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[55] Jin Pei, Jun Hong, and David Bell. A robust approach to schema matching over web query interfaces. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, pages 46–, Washington, DC, USA, 2006. IEEE Computer Society.

[56] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *The Proceedings of the VLDB Endowment*, PVLDB, pages 129–138, 2001.

[57] Arnaud Sahuguet and Fabien Azavant. Web ecology: Recycling html pages as xml documents using w4f. In *In ACM SIGMOD Workshop on the Web and Databases*, WebDB '99.

[58] D. Shestakov, S.S. Bhowmick, and E.P. Lim. Deque: querying the deep web. *Data & Knowledge Engineering*, 52(3):273–311, 2005.

[59] Weifeng Su, Jiying Wang, and Frederick Lochovsky. Holistic query interface matching using parallel schema matching. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 122–, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Weifeng Su, Jiying Wang, and Frederick H. Lochovsky. ODE: Ontology-Assisted Data Extraction. *Transactions on Database Systems*, 34(2), 2009.

[61] Guilherme A. Toda, Eli Cortez, Altigran S. da Silva, and Edleno de Moura. A probabilistic approach for automatically filling form-based web interfaces. *The Proceedings of the VLDB Endowment*, 4(3):151–160, December 2010.

[62] Guilherme A. Toda, Eli Cortez, Filipe Mesquita, Altigran S. da Silva, Edleno Moura, and Marden Neubert. Automatically filling form-based web interfaces with free text inputs. In *Proceedings of the 18th International Conference on World wide web*, WWW '09, pages 1163–1164, New York, NY, USA, 2009. ACM.

[63] Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 187–196, New York, NY, USA, 2003. ACM.

[64] Jiying Wang, Ji-Rong Wen, Fred Lochovsky, and Wei-Ying Ma. Instance-based schema matching for web databases by domain-specific query probing. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB '04, pages 408–419. VLDB Endowment, 2004.

[65] Wensheng Wu, AnHai Doan, and Clement Yu. Webiq: Learning from the web to match deep-web query interfaces. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 44–, Washington, DC, USA, 2006. IEEE Computer Society.

[66] Wensheng Wu, AnHai Doan, Clement Yu, and Weiyi Meng. Modeling and Extracting Deep-Web Query Interfaces. In *Advances in Information & Intelligent Systems*, pages 65–90, 2009.

[67] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the Deep Web. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, 2004.

[68] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Understanding Web Query Interfaces: Best-Effort Parsing with Hidden Syntax. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, 2004.

Appendix A

Prototype Evaluation Results on UK Real Estate Domain

Table A.1 summarizes the prototype evaluation result for each of the websites in the UK real estate domain. The website name is given in the first column. The rest of the table is partitioned into two sections: the first presents the result of analysis with only field and segment heuristics, and the second shows the result after integrating domain knowledge. Both sections are constructed the same with the five columns representing in sequence: total number of fields, the percentage of fields correctly retrieved, the percentage of fields correctly labeled, total number of segments, and whether the prototype performed a correct segmentation (✓) otherwise the number of missed segments. We compute the correctness for the segmentation task as the percentage of found segments out of the total number.

website	field, segment					field, segment, domain				
	form fields			form segments		form fields			form segments	
	total	found	labeled	total	found	total	found	labeled	total	found
rodgersestates	8	100.00	100.00	1	✓	8	100.00	100.00	1	✓
bspokeproperty	4	100.00	100.00	0	✓	4	100.00	100.00	0	✓
dreweryandwheeldon	5	100.00	100.00	0	✓	5	100.00	100.00	0	✓
nicholasestates	6	100.00	100.00	0	✓	6	100.00	100.00	0	✓
wilkie/main.htm	7	100.00	100.00	0	✓	7	100.00	100.00	0	✓
harveyrobinson	6	83.33	83.33	0	✓	6	100.00	100.00	0	✓
henrygeorgeestates	9	100.00	100.00	3	✓	9	100.00	100.00	3	✓
dawsonsproperty	23	100.00	100.00	6	✓	23	100.00	100.00	6	✓
knightfrank	2	100.00	100.00	0	✓	2	100.00	100.00	0	✓
all-about-homes	5	100.00	100.00	0	✓	5	100.00	100.00	0	✓
carlisleandborder	6	100.00	100.00	0	✓	6	100.00	100.00	0	✓
bernadetteharris	7	100.00	100.00	2	✓	7	100.00	100.00	2	✓
harmony-homes	6	100.00	100.00	0	✓	6	100.00	100.00	0	✓
kippencampbell	3	100.00	100.00	0	✓	3	100.00	100.00	0	✓
jimmcmillan	10	100.00	100.00	5	✓	10	100.00	100.00	5	✓
stokesestateagents	5	100.00	20.00	0	✓	5	100.00	60.00	0	✓
wrightschurchstretton	3	100.00	100.00	0	✓	3	100.00	100.00	0	✓
huwtudor	17	100.00	100.00	2	✓	17	100.00	100.00	2	✓
tspc	8	100.00	100.00	0	✓	8	100.00	100.00	0	✓
stewartwatson	24	100.00	75.00	5	2 missed	24	100.00	75.00	5	✓
morganyork	2	100.00	100.00	0	✓	2	100.00	100.00	0	✓
robsoncarter	3	100.00	100.00	0	✓	3	100.00	100.00	0	✓
clearwateruk	9	100.00	100.00	0	✓	9	100.00	100.00	0	✓
johnhoole	8	100.00	100.00	1	✓	8	100.00	100.00	1	✓
hi-m	6	100.00	66.67	1	1 missed	6	100.00	66.67	1	1 missed
qualityhomes	13	100.00	100.00	5	✓	13	100.00	100.00	5	✓
bychoice	7	100.00	100.00	0	✓	7	100.00	100.00	0	✓
rowelluk	9	100.00	77.78	4	2 missed	9	100.00	77.78	4	2 missed
nicktart	17	100.00	100.00	4	✓	17	100.00	100.00	4	✓
lawsonsestateagents	5	100.00	100.00	1	✓	5	100.00	100.00	1	✓
christopherbice	7	100.00	100.00	1	✓	7	100.00	100.00	1	✓
finders	8	100.00	100.00	2	✓	8	100.00	100.00	2	✓
andrewsonline	7	100.00	100.00	0	✓	7	100.00	100.00	0	✓
vebra	6	100.00	100.00	1	✓	6	100.00	100.00	1	✓
ankerandpartners	4	75.00	75.00	0	✓	4	100.00	100.00	0	✓
babingtons	5	100.00	100.00	0	✓	5	100.00	100.00	0	✓
bairstoweves	2	50.00	50.00	0	✓	2	100.00	100.00	0	✓
cjhole	7	100.00	100.00	3	✓	7	100.00	100.00	3	✓
heritage4homes	11	100.00	100.00	1	✓	11	100.00	100.00	1	✓
besleyhill	5	100.00	100.00	1	✓	5	100.00	100.00	1	✓
countryproperty	8	100.00	100.00	0	✓	8	100.00	100.00	0	✓
chestertonhumberts	15	100.00	100.00	3	✓	15	100.00	100.00	3	✓
edisonfordproperty	5	100.00	100.00	0	✓	5	100.00	100.00	0	✓
edwards-online	7	100.00	100.00	1	✓	7	100.00	100.00	1	✓
bruntandfussell	5	100.00	100.00	1	✓	5	100.00	100.00	1	✓
geoffreysmith	5	80.00	80.00	0	✓	5	100.00	100.00	0	✓
sequencehome	7	100.00	100.00	1	✓	7	100.00	100.00	1	✓
hootons	14	100.00	100.00	3	✓	14	100.00	100.00	3	✓
lettingzed	7	100.00	100.00	0	✓	7	100.00	100.00	0	✓
houseandco	13	92.31	84.62	6	✓	13	100.00	91.67	6	✓
		97.61%	94.91%	92.19%			100.00%	97.42%	95.31%	
		average precision		segmentation			average precision		segmentation	

Table A.1: Prototype Evaluation Results on UK Real Estate Domain

Appendix B

List of Concepts Evaluated in Domain Concept Mapping

1	Property ID Input Field
2	Address/Postcode Input Field
3	Area Selector
4	Branch Selector
5	Address Container
6	Radius Selector
7	Min Price Input Field
8	Min Price Selector
9	Max Price Input Field
10	Max Price Selector
11	Currency Selector
12	Type-Style Mix Selector
13	Type Selector
14	Style Selector
15	Age/Period Selector
16	Garden Selector
17	Parking Selector
18	Min Bedroom Input Field
19	Min Bedroom Selector
20	Max Bedroom Input Field
21	Max Bedroom Selector
22	Min Bathroom Input Field
23	Min Bathroom Selector
24	Max Bathroom Input Field
25	Max Bathroom Selector
26	Min Receptionroom Input Field
27	Min Receptionroom Selector
28	Max Receptionroom Input Field
29	Max Receptionroom Selector
30	STC/Sold Selector

31 Available-only Selector
32 Under Offer Selector
33 Add-In-Time Selector
34 New/Resale Selector
35 Display Method Selector
36 Orderby Mix Selector
37 Alphabetic Order Selector
38 Price Order Selector
39 Bedroom Order Selector
40 Add-In-Time Order Selector