Semi-automatic Generation of Active Ontologies from Web Forms for Intelligent Assistants

Martin Blersch
Karlsruhe Institute of Technology
Karlsruhe, Germany
blersch@kit.edu

Mathias Landhäußer
thingsTHINKING GmbH
Karlsruhe, Germany
mathias@thingsTHINKING.net

Thomas Mayer
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

Intelligent assistants are becoming widespread. A popular method for creating intelligent assistants is modeling the domain (and thus the assistant's capabilities) as Active Ontology. Adding new functionality requires extending the ontology or building new ones; as of today, this process is manual.

We describe an automated method for creating Active Ontologies for arbitrary web forms. Our approach leverages methods from natural language processing and data mining to synthesize the ontologies. Furthermore, our tool generates the code needed to process user input.

We evaluate the generated Active Ontologies in three case studies using web forms from the UIUC Web Integration Repository, namely from the domains airfare, automobile, and book search. First, we examine how much of the generation process can be automated and how well the approach identifies domain concepts and their relations. Second, we test how well the generated Active Ontologies handle end-user input to perform the desired actions. In our evaluation, Easier automatically generates 65% of the Active Ontologies' sensor nodes; the generated ontology for airfare search correctly answers 70% of the queries.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces**; • **Computing methodologies** → *Ontology engineering*; Natural language processing;

ACM Reference Format:
Martin Blersch, Mathias Landhäußer, and Thomas Mayer. 2018. Semi-automatic Generation of Active Ontologies from Web Forms for Intelligent Assistants. In *RAISE'18: IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194104.3194108

1 INTRODUCTION

Intelligent assistants such as Amazon's Alexa and Apple's Siri are omnipresent. They cover the basic functions of the computers they run on and use external services to answer questions such as "How's the weather going to be in Gothenburg tomorrow?" Yet adding new capabilities to such assistants is a time-consuming manual task.

Apple's Siri uses Active Ontologies (AOs) to process the end-user's requests [1]. When building AOs, developers must explicitly model the domain concepts as nodes and their relationships as directed edges in an hierarchical graph. In the weather example above, we'd need (at least) the following nodes: the action (i.e., *deliver the weather report*), the *place*, and the *date* for the forecast. Specialized leaf nodes, so called sensor nodes, are responsible to extract relevant information from the end-users input, e.g., the *place* node would extract city names. If a sensor node detects relevant information in the input, it sends a message to its parent node. Inner nodes, in contrast to sensor nodes, receive messages from their children and forward them to their parents, e.g., the *date* node could have a child that interprets words such as "tomorrow" and another child that detects dates such as "May 27, 2018"; in doing so, inner nodes can refine, combine, or ignore the information that they receive. The root node models a distinct operation or function of the assistant and calls, for example, a web service. Because of this design, bottom-up input processing is efficient and developers can understand the domain model easily. The downside of this approach is that every time the capabilities of the intelligent assistant must change (e.g., supporting additional options), developers must explicitly model the operation and their parameters. This is a time consuming task and limits the applicability of AO-based assistants.

Easier is an AO server and AO building framework for form-based services. Many service providers that target end-users provide HTML forms for accessing them but do not provide standardized web services that could be invoked in the root node. Easier simplifies building AOs for such forms by guiding the developer through the creation process. First, Easier crawls the Web for HTML forms and categorizes them (e.g., as airfare, automobile, or book search services). In previous work, we explored how clustering techniques can be adapted to this task [3]. This paper concentrates on the generation of an AO for a specific service category. Given multiple web forms providing the same service, Easier generates an unified AO that captures all characteristics of the different forms (e.g., similar forms from different airlines). In addition, it generates the necessary sensor nodes; if information for fully automatic AO generation is missing, the developer is asked. Last but not least, it generates a service directory that is used in production to actually call the external services and collect the results.

Section 2 explains AOs and the Easier Active Server and Section 3 reviews related work. The following sections describe how we automatically derive AOs from a collection of web forms and how well these AOs can respond to actual end-user input. The final section presents our conclusions and discusses future work.

RAISE'18 , May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5723-4/18/05…$15.00
https://doi.org/10.1145/3194104.3194108

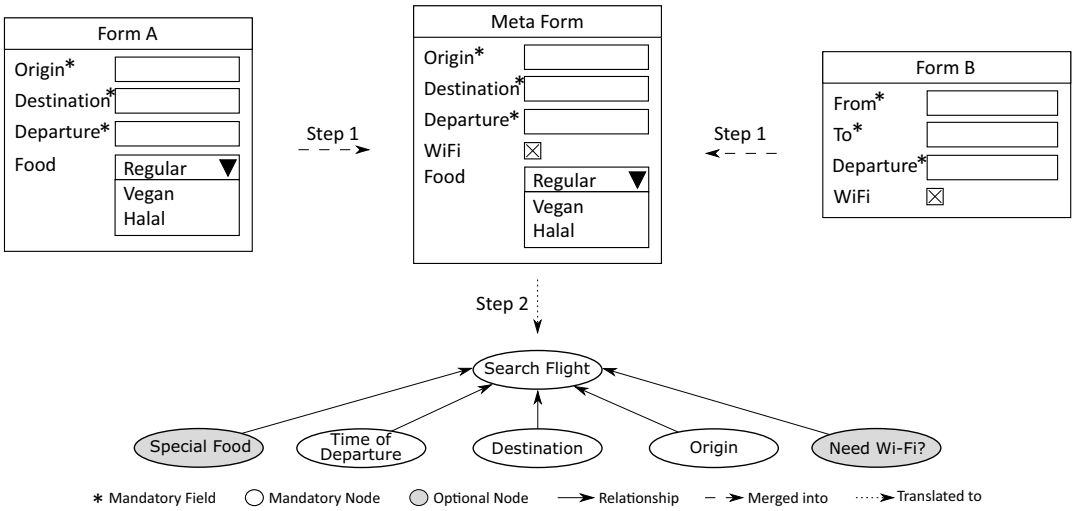

Figure 1: An Active Ontology modeling the input data for the airfare domain. First, Easier combines the forms to a meta form; semantically equivalent fields from different forms are merged. Then the meta form is translated into an AO. Optional nodes represent form-specific fields.

2 ACTIVE ONTOLOGIES AND THE EASIER ACTIVE SERVER

This section introduces AOs and reviews the Easier Active Server architecture. An in-depth description of both can be found in [3, 6, 7]. Here, we repeat the most important concepts for the sake of self-containedness.

AOs are used to build intelligent assistants [6, 7]. They a) are used to model the domain and b) include a processing environment specifically designed for natural language processing (NLP) and understanding (NLU). The information an intelligent assistant needs to perform a specific operation is modeled bottom-up in the AO: As one can see in Figure 1, the assistant needs the origin, the destination, the time of departure and the travel class to book a flight; other information, such as Wi-Fi or the need for special food, is optional.

NLP starts in the leaf nodes: The so called sensor nodes contain code or use word lists to extract their part of the information from the incoming utterances of the end-user. If a node detects relevant information, it *fires* and sends the information to its parent(s). Inner nodes can either select one of or combine several messages sent by their children and forward them to their parents; they can also ignore messages, e.g., if they seem unreliable. If the root node receives information from all of its mandatory child nodes, its operation can be carried out. If mandatory information is missing at any point, the AO asks the user specifically to provide it.

There are four different types of sensor nodes in an AO (see reference [3]):

- *Vocabulary list leaves* compare incoming tokens with a list of words.
- *Prefix and postfix leaves* consider several tokens following or preceding one or more keywords.
- *Regular expression leaves* use a regular expression to identify well-formed tokens, e.g., ZIP codes or email addresses.
- *Specialized leaves or groups*, for example, date and time leaves identify exact dates and to determine relative dates such as "tomorrow" or "now".

The Easier Active Server is a runtime environment for AOs and is based on Guzzoni's Active Server [6]. Figure 2 gives a short overview of the architecture (see reference [3] for details). It provides all AOs with a shared memory, the fact store, and an evaluation engine that executes the code stored in the AOs. The Dialog Manager (DM in Figure 2) communicates with the user; it uses the dialog interface to deliver results to the end-user and to ask for additional information. The Broker AO calls external services that actually perform the desired operation, e.g., booking a flight. Therefore it knows how to call the external services and their capabilities.

Usually, adding new functionality to an AO-based assistant requires the developer to model the domain and to provide the code for processing the end-users' utterances. To reduce the time needed for these complicated tasks, Easier automatically derives AOs from web forms: Given a set of web forms fulfilling a common operation (e.g., booking forms of two different airlines as in Figure 1), it produces one AO to gather all needed information and provides the broker with the service descriptions.

3 RELATED WORK

To derive an AO from two or more web forms that provide essentially the same service, Easier has to unify them. Therefore it identifies semantically equivalent form elements and then generates proper AO nodes that model these elements. To identify equivalent form elements, Easier employs linguistic and structural element matching; it uses the elements' names, labels, and attributes for linguistic matching and the form elements' positions and the forms' hierarchies for structural matching, respectively. In this section, we review related approaches that merge forms or ontologies and match the respective elements to each other.

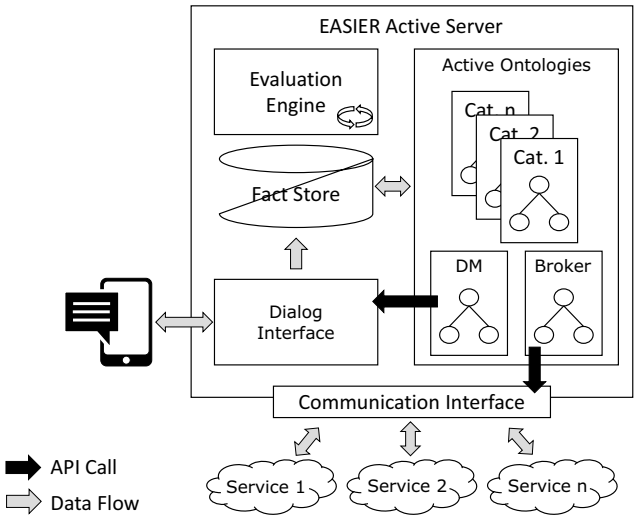

Figure 2: The Easier Active Server architecture, as shown in reference [3]. AOs use a shared fact store to communicate with each other. Special-purpose AOs, the dialog manager and the broker respectively, handle interaction with the end-user and with the service providers.

OntoBuilder extracts ontologies from web forms and then creates a mapping between similar ontology elements [5, 12]. It considers the form elements' labels, data types, and value constraints. In addition, application semantics, such as the ordering of the form elements, are used for similarity analysis. To identify matching elements, OntoBuilder first calculates pairwise similarity measures for all elements; then it selects the subset with the highest similarities. Element similarity is based on term matching and value matching: Term matching considers the elements' labels and names and the similarity is calculated using the number of common words and the length of the longest common substring. Value matching considers the possible values of select options, radio button groups, and check boxes; in essence, the more values two elements share, the higher their similarity. Easier uses a similar approach during similarity analysis.

He et al. developed WISE-Integrator to generate meta search engines for E-Commerce [8, 9]. First, WISE-Integrator groups semantically related labels and form elements and derives their meta data (e.g., their data types). A two-step clustering technique uses these so-called logic attributes to identify matching form elements of different forms. Then, it merges matching (the so-called local) attributes and thereby generates so-called global attributes. The global attributes are then used as elements of the meta search form. When a global attribute is formed from local attributes, a new global domain type and its values must be derived. There are four domain types: finite, infinite, range, and hybrid. The authors use rules to merge local attributes with different domain types and describe how to merge different value ranges to obtain a common set of values. Easier also uses a Meta Form to represent similar form elements and we use similar rules for combining different domain types and value ranges.

Wu et al. use a clustering-based approach to match interfaces for querying on-line data sources [13]. They model the forms' hierarchies as ordered trees and use them to identify semantically related elements. Their approach can produce complex mappings where one element of one form semantically relates to two or more elements of a second form. For example, some forms use a single text field to ask for a date of birth whereas others use three separate fields for day, month, and year. To calculate the similarity of two form elements, they compare the linguistic (i.e., names and labels) and domain similarity (i.e., data types such as int or string). Their matching algorithm first generates one cluster for each form element and then iteratively merges the two most similar clusters. The approach relies on user interaction to reduce the matching error rate. Easier employs a similar hierarchical clustering process.

Cupid is a generic schema matching algorithm that discovers mappings between arbitrary schema elements [11]. It can be used to match any hierarchical schema, e.g., database schemas and XML documents. First, Cupid creates a schema tree that contains all elements of a schema. Then, it computes the similarity between the elements of two trees, again using linguistic and structural matching. Similar to the other approaches, Cupid's linguistic matching uses the names, data types etc. but normalizes the terms beforehand: Cupid removes unnecessary tokens and expands abbreviations and acronyms. Then, similar pairs are determined also considering the environment of the elements (i.e., their ancestors and siblings in the schema tree).

Berlanga et. al present a tool for semi-automatically building ontologies from a set of forms [2]. FAETON extracts a logical model of the form's basic components (i.e., the form's controls and their corresponding labels), their visual and geometric features (e.g., absolute position) and their data types. External sources such as domain ontologies and lexicons are used to semantically annotate form elements. Finally, FAETON combines structural information and semantic annotations to build the ontology.

4 GENERATION OF AOS FROM WEB FORMS

To create an AO for a set of forms, Easier identifies form elements from different forms that semantically take the same input. Then the matching elements are combined to a Meta Form. The Meta Form captures all individual forms and is then transformed into an AO. This section presents our four-step approach in detail: A) Semantically related form elements of different web forms are identified using linguistic and structural analyses. B) Hierarchical clustering groups related form elements. C) Using the element groups, we generate a Meta Form that captures all forms of the service category. D) From the Meta Form, we automatically derive the AO and load it in the Easier Active Server. The following paragraphs explain these four steps in detail.

4.1 Identification of Semantically Related Form Elements

Web forms provide query interfaces for humans. Using such query interfaces requires humans to know – for every form element – which values are permitted. Therefore, the forms provide additional information about the semantics of the form elements to the user.

We use this information to match similar form elements of different forms to each other.

Easier's similarity analysis comprises two parts: a linguistic analysis and a structural analysis. The linguistic analysis compares the strings associated with the form elements and their data types; the structural analysis considers the layout and structure of the forms. We calculate the similarity between two form elements e_1 and e_2 as follows:

$$sim(e_1, e_2) = w_l Ling(e_1, e_2) + w_s Struc(e_1, e_2)$$

Ling is the linguistic similarity whereas *Struc* is the result of the structural analysis. The final similarity is a weighted sum of the two components.

The linguistic analysis comprises three components:

$$Ling(e_1, e_2) = w_1 V(e_1, e_2) + w_2 LCS(e_1, e_2) + w_3 D(e_1, e_2)$$

V is a vector similarity considering the words that describe the form elements. *LCS* is a measure based on the longest common substring. D considers the domain (i.e., the data type) of the fields.

The structural analysis considers the vicinity of two elements:

$$Struc(e_1, e_2) = \frac{Vic(Pred(e_1), Pred(e_2))}{2} + \frac{Vic(Succ(e_1), Succ(e_2))}{2}$$

where $Pred(e_i)$ is the set of predecessors and $Succ(e_i)$ the set of successors of element e_i respectively. In the following, we describe how we calculate the components of the formula in detail.

To determine the semantics of a form element, Easier analyzes its label and the values of the HTML attributes `id`, `name`, `placeholder` and `value`. Labels and the values of the `placeholder` and `value` attributes (iff set) are visible elements used to describe a form element to a human. Labels appear close to the corresponding form element whereas the value of the placeholder attribute is displayed inside the corresponding input field. Both give users hints which kind of value to enter. The values of the attributes `id` and `name` are not (directly) visible to them; they are part of the HTML code but often hint the form elements' semantics, as many form designers use semantically related words as their values.

We create so called local objects[1] for each input field of a web form. A local object comprises six properties: the values of an element's `label` and its attributes `id`, `name`, `placeholder`, and `value`[2]. We normalize all strings contained in these properties and tokenize them using uppercase letters, non-numeric characters and numbers as separators. In addition, we remove stop words and transform the strings to lowercase. The sixth property of a local object is the element's data type.

4.1.1 Vector Similarity. As proposed by Wu et al. in [13], our linguistic analysis calculates a cosine similarity between the form elements e_1 and e_2 considering the first five properties of their respective local objects as follows:

$$V(e_1, e_2) = \max_{\alpha,\beta}\{\cos(l_{1,\alpha}, l_{2,\beta})\}, \quad \alpha, \beta \in \{\text{label, id, name, placeholder, value}\}$$

$l_{i,j}$ is a binary bag of words vector for the property j of local object l that belongs to form element e_i. We use stemming and determine the dimensions of the vectors for each pair separately.

4.1.2 Substring Matching. We use substring matching as used in reference [5] to calculate the substring similarity LCS:

$$LCS(e_1, e_2) = \max_{\alpha,\beta}\{\frac{\text{lcs}(s_{1,\alpha}, s_{2,\beta})}{\max_{\alpha,\beta}\{\text{strlen}(s_{1,\alpha}), \text{strlen}(s_{2,\beta})\}}\}, \quad \alpha, \beta \in \{\text{label, id, name, placeholder, value}\}$$

To calculate the longest common substring (lcs), we use the original string $s_{i,j}$ (not tokenized but in lower case and with whitespace removed) of the local object's property j that belongs to form element e_i. To get a proper similarity, we normalize the result with the length of the longer string.

4.1.3 Domain Analysis. Finally, we compare the data types of two local objects and their ranges of valid values (see also reference [13]). We distinguish five different groups of data types: time, choice, numerical, pattern, and text. Two objects from different groups are not similar by definition ($D = 0$); if the objects come from the same group, we take a closer look.

Two time input fields are considered equal ($D = 1$) iff they have the same type (e.g., date, week, or time) and not equal otherwise ($D = 0$).

Choices (e.g., check boxes, radio buttons, and selections) provide valid values. Again, we use the cosine distance (see 4.1.1) to determine the similarity between two options of form elements e_1 and e_2. We count the number c of similar options and calculate the similarity as $D_{\text{choice}} = \frac{2c}{O}$ where O is the number of options from both form elements.

The pattern group consists of `text` fields (where validation patterns are provided), `url`, `tel`, and `email`. Two elements of that group are considered equal ($D = 1$) iff they have the same type (or the same pattern, respectively) and not equal otherwise ($D = 0$). Regular text fields (`textarea`, `search`, and `text` without validation patterns) are considered not equal by definition.

To calculate the similarity of two numerical fields n_i (`number` and `range`), we use their `min` and `max` attributes to calculate the overlap:

$$D = \frac{\min_{\alpha,\beta}\{n_{1,\max}, n_{2,\max}\} - \max\{n_{1,\min}, n_{2,\min}\}}{\max\{n_{1,\max}, n_{2,\max}\} - \min\{n_{1,\min}, n_{2,\min}\}}$$

4.1.4 Structural Analysis. If two fields are semantically related, there is a high probability that the fields in their vicinity are pairwise related as well [5, 11, 13]. We analyze the form's structure (i.e., the positions of form elements inside forms) and calculate the similarity of two elements' vicinity, i.e. their predecessor and successor terms. Our approach takes into account that the influence of the terms' vicinity decreases with the distance. The similarity between an

[1] Gal et. al [5] use *terms* to store combinations of form elements' `label` and `name` pairs. We extend their terms by additional HTML attributes (i.e., `value`, `id`, and `placeholder`) and the data type.

[2] For all control elements providing a finite number of options (`select`, `datalist`, `checkboxes` and `radio buttons`) we use the values of the `option` attributes.

element and its vicinity (Vic) is calculated as follows:

$$Vic(E_1, E_2) = \sum_{i=0}^{j-1} \frac{1}{4 * 2^i} * sim_\tau(e_{1i}, e_{2i}),$$

where E_1, E_2 are sets of predecessors or successors of elements from form 1 and 2, $j = min\{|E_1|, |E_2|\}$, $e_{1i} \in E_1$ and $e_{2i} \in E_2$ where i indicates the distance between an element and its respective sibling (the direct neighbors have distance of 0). sim_τ indicates whether the linguistic similarity of two elements exceeds a certain threshold τ:

$$sim_\tau(e_1, e_2) = \begin{cases} 1, & Ling(e_1, e_2) > \tau. \\ 0, & \text{otherwise.} \end{cases}$$

4.2 Clustering Similar Elements

To obtain sets of similar elements from different forms we employ the clustering technique introduced by Wu et al. [13]. We start with a set of minimal clusters – one for each element; then we merge the clusters until the stopping criterion is met:

(1) Place each element in a cluster by itself.
(2) As long as there are two or more clusters with a cluster similarity above the threshold do:
   (a) Find the clusters c_i and c_j with the highest cluster similarity $sim_c(c_i, c_j)$.
   (b) Merge the clusters to generate cluster c_m iff c_m does not contain two elements from the same form.

The cluster similarity is defined as the maximum pairwise similarity of two elements from the two clusters: $sim_c(c_i, c_j) = \max\{sim(e_{i,a}, e_{i,b})\}$.

4.3 Creation of a Meta Form

One global object – ideally – corresponds to one set of semantically equivalent form fields and is used to derive one sensor node. Thus, we create one global object for each cluster. Each global object has a type, a value range/value set and attributes such as *name* or *ID*.

The type of the respective AO leaf node depends on the global object's type. First, we determine the global object's type following WISE-Integrator's approach [8, 9]. We classify value ranges and value sets in three categories[3]:

- **finite** types have a defined set of values (e.g., from checkboxes, radio buttons, and select tags)
- **infinite** types have no known value range or value set (e.g., a text input field without a pattern attribute)
- **range** limits the value range to a given interval (e.g., the input field types "number", "date", "text" with the pattern attribute set).

If semantically related form elements with different value ranges or value sets and types are merged, a suitable type must be chosen for the global object. With focus on natural language understanding, our AOs must understand a wide range of user utterances. It is important that our AOs understand any information that is valid in any of the forms. Therefore, we prefer broader value ranges to small ranges. We defined rules for merging different input types. We prefer range types over infinite types and infinite types over finite types. Within range types, we also prefer general to specific types. For example, when merging input fields of type `tel` and `number`, we create a global object with the range type `number`, because numbers are more general and subsume telephone numbers.

To determine the global objects' attributes `id`, `label`, `name`, and `placeholder`, we use the majority strategy [10] and choose the element that occurs most often. Different `options` (value sets) are merged, regular expressions from `pattern` attributes are combined. If the attribute `required` is set in one of the form elements, we keep the value to indicate the global elements' necessity. When merging two numerical fields with the attributes *min* and *max* we use the smallest *min* and the largest *max* value to achieve the widest range of possible values.

[3] In [9], the authors also used a Boolean type for classification. We do not distinguish between *Boolean* and *finite* types.

4.4 Generation of Active Ontologies

From the Meta Form, we automatically generate the AO and the service directory for the Broker. The service directory contains the mapping from the Meta Form Elements to the elements of the category's forms. For every form, we also store the information about the service provider, such as its name, the web form's name or ID, the URL, its service category, and the number of form steps (important for multi-stepped web forms). The broker component can look up all usable service providers and map the global information it gets from the AO to the service provider's form elements. For example, in a service directory for the *Book Flight* AO in Figure 1, two local objects represent the form elements *FormA.From* and *FormB.Origin* which refer to the global object *Origin*. When a user provides "Gothenburg" as the flight's origin for the object *Origin*, the broker can set the values for *FormA.From* and *FormB.Origin* to "Gothenburg".

Then the AO generator derives the sensor nodes from the Meta Form Elements (see Section 2 for the node types). Each element becomes one building block, depending on its type and value range. For example, if the element offers a list of possible values, a vocabulary list node is created. Table 1 shows an excerpt of our derivation rules. For input fields of type *text* which do not provide a *pattern* attribute, we cannot automatically derive its value ranges or value sets. In this case, we ask the developer to enter a list with valid values and to chose the desired node type. For prefix or postfix leaf nodes, we provide possible pre-/postfixes, i.e., the label before or after the input field.

After all sensors have been derived, the generator connects them to a newly created root node. Then the AO can be loaded in the Easier Active Server together with the service directory.

5 EVALUATION

The evaluation of our approach is threefold: 1) We examine how well Easier matches corresponding form fields (i.e., how good the clustering is), 2) how well the generated meta form can be translated into AO definitions (i.e., how much of the process can be automated), and 3) how well the generated AO answers queries of actual users (i.e., how good the overall process performs).

To test Easier we used 58 web forms from the three service categories airfare, automobile, and book search services (20, 20, and 18 forms respectively); all forms have been taken from the *ICQ*

Table 1: Derivation rules for AO leaves from form fields (excerpt). Types: (Pr)efix leaf nodes, (Po)stfix leaf nodes, (V)ocabulary list leaf, (Re)egular expression leaf.

Input Type	Type	Values/Value Range
Text, search (w/o pattern)	V/Pr/ Po/Re	Ask User
Text, search (w/ pattern)	Re	Regular expression from pattern attribute
Color, date, datetime, email, month, tel, url, and week	Re	Regular expression for this type
Checkbox	V	Check boxes' values
Radio	V	Radio buttons' values
Select	V	Options

Table 2: Field matching performance for three service categories.

Category	Precision	Recall	$F_{0.5}$ Measure
Airfare	90.6	21.0	54.4
Automobile	90.6	37.3	70.5
Book	98.4	46.4	80.4

Query Interfaces dataset from the UIUC Web Integration Repository [4]. The web forms were fed to Easier without any modifications.

5.1 Field Matching

To examine the field matching's performance, we manually created a gold standard with the correct field mappings. We measure the performance of field matching with precision, recall, and the $F_{0.5}$-measure. Precision p is the percentage of correct mappings over all generated mappings and recall r is the percentage of correct mappings with respect to all mappings in the gold standard. Overall we favor precision over recall for field matching: An unmatched field (i.e., a field that forms its own cluster) results in an additional meta form element which is translated to a separate sensor node. A field that is matched incorrectly results in a sensor node with one semantic meaning but listening for (at least) two semantically different inputs. The $F_{0.5}$-measure is defined as

$$F_{0.5} = (1 + 0.5^2)\frac{p \times r}{(0.5^2 \times p) + r}$$

and favors precision over recall. We report the measures for all three categories separately.

As one can see in Table 2 the precision is highest for the book search. This is because matching the book-related field names is easier because the forms are more consistent and more standardized, e.g., they all use an *ISBN* field. The consistently low values for recall indicate that we have to improve our clustering approach.

5.2 Degree of Automation

The degree of automation indicates whether Easier can derive the sensor nodes without asking the developer for additional information. Deriving the sensor nodes entails determining and selecting a suitable node type and the value range of a sensor node. If the value range or the node type cannot be derived automatically, Easier lets the developer choose.

Table 3: Degree of automation.

Category	Nodes			
	Total	Manual	Autom.	Autom.
Airfare	126	29	97	77%
Automobile	41	23	18	44%
Book	49	24	25	51%
Total	216	76	140	65%

We determined how many sensor nodes were generated automatically and for how many nodes Easier needed help from the developer. Table 3 shows the results for all three categories. The degree of automation differs among the categories: We can automatically generate 77% of the airfare-related elements. Many forms of this category provide select elements with a predefined set of options. On the other side, the book and automobile search forms contain many text fields without hints about their value range, e.g., for specifying the author of a book or the car brand. Easier cannot transform such form elements without the help of the developer. But in total, Easier needed developer support only 35% of the time. Considering that the web forms from the UIUC repository are quite old, these results are encouraging. In the future, we will exploit more modern HTML5 elements, such as the ARIA tags, to improve the results even further.

5.3 AO Performance

To examine the overall AO performance, we asked subjects to interact with our system actually querying for flights. In total, we had 40 subjects, 20 of which were English native speakers. We introduced Easier to the subjects as "yet another digital assistant" similar to Apple's Siri that is able to perform flight reservations. Then we asked them to formulate queries and to write them down. In a post-processing step, we ensured that the actual values in the queries (e.g., the airport names) were written down correctly. We did not alter the queries in any other way and left the query formulation entirely to the subjects.

To measure the AO's coverage of the queries, we determined how many query elements were correctly recognized by the sensor nodes. Given the query "Book a flight from Frankfurt to Gothenburg." and the ontology in Figure 1, we expect Easier to extract the origin and departure airports. Since the query does not contain the time of departure, we expect Easier to ask the user for this information. The nodes for food and Wi-Fi should not fire because the user did not specify any preferences; since the nodes are not mandatory, Easier should not ask for them.

In total, we sent 61 queries to Easier. The sensor nodes produce a recall of 75% which indicates a broad coverage of the queries. Unfortunately, only 35.5% of the queries were completely recognized without errors (i.e., without missing or wrong form fields, including optional fields). Analyzing the results we learned that the automatically generated AO has a hard time distinguishing two (or more) form fields with identical value ranges. This is especially hard on the airline AO as all of the forms contain two fields for airports (*origin* and *destination*) and often two date/time fields (date/time

Table 4: Evaluation of the sensor nodes for ten complete queries.

Field	correct	wrong	missing
origin	191	9	0
destination	185	15	0
date of travel	124	24	52
Total	500	48	52

of *departure* and of *arrival*). Asking the developer, e.g., for prefixes or postfixes to distinguish between these fields would counteract this issue. In this evaluation we did not improve the automatically generated parts of the AO.

Easier provides a dialog system which can be used to ask for missing information. When users do not supply sufficient information to execute the root node (i.e., if not all mandatory nodes fire), the AO is supposed to start a basic clarification dialog. The generated AO contains many optional nodes for which usually no questions are triggered. For the mandatory nodes, Easier correctly asked the questions 64% of the time; in the remaining 36% the AO misinterpreted the users' input.

Next, we wanted to know whether Easier correctly answers queries that contain all necessary information. To perform a flight search, we expect the queries to supply at least origin, destination, and the date of departure; these three fields are supported by all 20 service providers. 26 queries contain all required information and the results are shown in Table 4. To limit the effort, we only analyzed a subset of these queries in detail. Of the ten queries we have sampled, seven were performed correctly. In two cases Easier did not recognize the supplied information and asked the user for the missing values. Only one query was recognized incorrectly and led to erroneous calls to the service providers.

6 CONCLUSION AND FUTURE WORK

In this paper we detailed how Easier can be used to construct AOs for arbitrary web services that can be accessed with HTML web forms. Easier assembles a meta form for every service category and semi-automatically constructs the AO nodes that are needed to process textual input. Overall, Easier automates 65% of the developers grunt work and in doing so increases the applicability of AO-based intelligent assistants.

Our evaluation shows that Easier successfully generates AOs but also that we have to improve input disambiguation. In the evaluation we used 58 web forms of three service categories to assess the degree of automation. With 61 flight searches formulated by 40 participants in natural language we assessed how well the automatically generated AO handles user input. The results are promising but leave room for improvement: Overall, the airline AO covers 75% of the queries but only 36% of the queries are fully covered. Many issues result from ambiguous fields that cannot be distinguished without additional information. Despite this fact, a detailed analysis of ten complete queries showed that Easier was able to answer 70% of the queries.

In the future, we want to improve the field matching performance using additional information such as HTML5 labels for Accessible Rich Internet Applications (ARIA). To increase the field mapping performance further we will integrate complex field mappings (e.g., when one form uses a single element to ask for a date of birth and another form uses three elements to ask for day, month and year separately). When generating sensor nodes, Easier asks the developer to provide valid data ranges or a set of valid values if it cannot determine them automatically; we plan on increasing the degree of automation by integrating background and/or domain knowledge (e.g., from Wikipedia and ontologies such as Cyc). Today, Easier supports textual input only. In the future, we will integrate automatic speech recognition and synthesis to facilitate a truly natural dialog. We also plan on strengthening the evaluation by increasing the number of service categories and web forms respectively and conduct an additional survey with more participants to increase the variety in the gold standard.

REFERENCES

[1] Jerome R. Bellegarda. 2014. Spoken Language Understanding for Natural Interaction: The Siri Experience. In *Natural Interaction with Robots, Knowbots and Smartphones*. Springer, New York, NY, 3–14.

[2] Rafael Berlanga, Ernesto Jimenez-Ruiz, Victoria Nebot, and Ismael Sanz. 2010. FAETON: Form Analysis and Extraction Tool for Ontology Construction. *International Journal of Computer Applications in Technology* 39, 4 (2010), 224–233.

[3] Martin Blersch and Mathias Landhäußer. 2016. Easier: An Approach to Automatically Generate Active Ontologies for Intelligent Assistants. In *Proceedings of the 20th World Multiconference on Systemics, Cybernetics and Informatics (WMSCI 2016)*. Orlando, FL, USA.

[4] Computer Science Department, University of Illinois at Urbana-Champaign. 2003. The UIUC Web Integration Repository. (2003). http://metaquerier.cs.uiuc.edu/repository

[5] Avigdor Gal, Giovanni Modica, Hasan Jamil, and Ami Eyal. 2005. Automatic Ontology Matching Using Application Semantics. *AI magazine* 26, 1 (2005), 21.

[6] Didier Guzzoni. 2008. *Active: A Unified Platform for Building Intelligent Applications*. PhD Thesis. École Polytechnique Fédérale De Lausanne.

[7] Didier Guzzoni, Charles Baur, and Adam Cheyer. 2006. Active: A Unified Platform for Building Intelligent Web Interaction Assistants. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology - Workshops, Hong Kong, China, 18-22 December 2006*. IEEE Computer Society, 417–420. https://doi.org/10.1109/WI-IATW.2006.27

[8] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. 2003. Wise-Integrator: An Automatic Integrator of Web Search Interfaces for e-Commerce. In *Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29*. VLDB Endowment, 357–368.

[9] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. 2004. Automatic Integration of Web Search Interfaces with WISE-Integrator. *The VLDB Journal* 13, 3 (Sept. 2004), 256–273. https://doi.org/10.1007/s00778-004-0126-4

[10] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. 2005. WISE-Integrator: A System for Extracting and Integrating Complex Web Search Interfaces of the Deep Web. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, Trondheim, Norway, 1314–1317.

[11] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. *Generic Schema Matching with Cupid*. Technical Report MSR-TR-2001-58. Microsoft Research. 49–58 pages.

[12] Haggai Roitman and Avigdor Gal. 2006. Ontobuilder: Fully Automatic Extraction and Consolidation of Ontologies from Web Sources Using Sequence Semantics. In *Current Trends in Database Technology–EDBT 2006*. Springer, 573–576.

[13] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. 2004. 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*. ACM, 95–106.