Automatic Generation of Virtual Assistants from Databases using Active Ontologies

Martin Blersch and Sebastian Weigelt and Walter F. Tichy
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
Karlsruhe, Germany
blersch@kit.edu, weigelt@kit.edu, tichy@kit.edu

Kevin Angele
Semantic Technology Institute
University of Innsbruck, Innsbruck, Austria
and *ONLIM GmbH*, Telfs, Austria
kevin.angele@sti2.at

***Abstract*—Virtual assistants such as Siri or Google Assistant are omnipresent. However, their development remains costly. One must either manually model the problem domain or provide thousands of labeled samples.**

We propose to automatically create virtual assistants based on Active Ontologies for interacting with databases. Our approach generates Active Ontologies; we use the database structure to derive a concept hierarchy and database values together with synonyms to extract information from user queries. Our approach also learns common phrases from samples, e.g. from existing Dialogflow agents. We extract pre- and postfixes and attach them to concepts, e.g. *at* to detect a succeeding location. The generated Active Ontologies reply to previously unseen and composed requests. The approach is not limited to virtual assistants but can be applied to any system with a textual or voice-based conversational interface such as chatbots.

We evaluate our approach in three domains: tourism, hotel, and web cams. The study shows that automatically generated Active Ontologies extract relevant information from user utterances with a precision of 58%. The precision increases to 79% (recall 46%, F_1 58%) when we use sample utterances. Our approach successfully transfers between domains, e.g. we learn phrases from the tourism domain and use them to reply to hotel requests without any adjustments.

I. Introduction

Virtual assistants and other systems with conversational interfaces (CI) are used by an ever-growing number of people. Users ask Siri for their next meeting, talk to chatbots, or tell Alexa to turn on the lights. While these systems are a blessing for casual users, they remain a curse for developers. They are complicated to build and hard to maintain. Today's virtual assistants are either trained on thousands or even millions of (manually) labeled samples, or their linguistic competence is modeled manually, i.e. they are built by domain experts. Both require serious manual effort to make the system appear human-like to the user. However, users will soon expect CIs for all kinds of applications. Thus, the efficient creation of CIs (i.e. developer assistance, transferability, automation, etc.) will become one of the major challenges in software engineering.

We propose to generate systems with CIs largely automatically. As underlying technology we use Active Ontologies [1], [2] (AO) that originally were at the core of Apple's Siri. AOs are hierarchical domain models equipped with processing rules. In terms of structure they are trees, where the leaves react to words in user utterances. The inner nodes join information and the root creates a services call, e.g. a restaurant reservation.

As a prerequisite for our generation process, we assume that a service provider stores information in a database, e.g. the addresses, ratings, and names of restaurants. Having this information, our AOs are supposed to answer requests such as, "Show me restaurants in Lisbon." We automatically generate AOs from the databases in two steps. First, we use the structure of the database to infer the concept hierarchy of the AO. Second, we add leaf nodes to provide the AO with linguistic competence. We generate those from database values and add synonyms obtained from Wiktionary[1]. Optionally, if a data set of sample user utterances is present, we can further improve the linguistic competence. We extract related phrases for each concept. For example, we can learn the phrase *where can I find* for the concept `location` from examples like, "Where can I find an Italian restaurant." Our generated AOs reply to previously unseen request, e.g. finding a hotel in a city (which was learned from *finding* web cams and listing tourist attractions *in a city*). Moreover, they correctly respond to complex requests composed of two or more simple requests, e.g. asking for both, the location and opening hours of a restaurant, at the same time. The developer reviews the result of the generation process and adapts the AO. Usually, this involves altering the type of inner nodes and adding common phrases to leaf nodes.

The remainder is structured as follows. First we introduce the basic elements of Active Ontologies in section II. Then, we discuss related work in section III. In section IV we present our approach and the generative process in detail together with a discussion about its inherent limitations. Afterwards, we evaluate our approach in section V. We conclude our work and discuss further improvements in section VI.

II. Active Ontologies

Active Ontologies were first proposed by Guzzoni et al. [1], [2]. Originally, they were used to build virtual assistants. However, they can be used as a generic CI. We present

DOI reference number: 10.18293/SEKE2019-077

[1]Wiktionary: https://www.wiktionary.org/

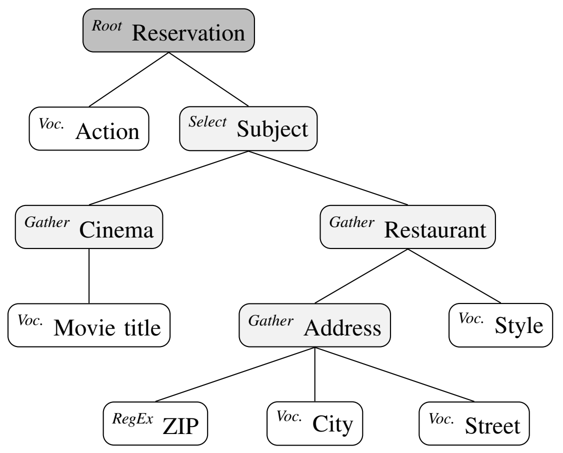

Fig. 1. An AO for the problem domain "reservations". The AO is can create service calls to cinema and restaurant reservation systems. The root node is colored in gray, the inner nodes in light gray, and leaf nodes are white.

the fundamental elements and basic processing of AOs. All upcoming examples refer to the AO depicted in Figure 1. Detailed descriptions may be found in the above reference.

AOs combine concept hierarchies with processing rules. On the one hand, they model the domain as an ontology. On the other hand, AOs process natural language requests and turn them into service calls. The concepts and relations form trees, i.e. an AO is a concept hierarchy of a problem domain. Information is processed bottom-up. The leaf nodes react to words in the user utterances. The concept nodes (i.e. the inner nodes) join information and gradually create an abstract representation of the utterance. Finally, the root node gathers all information and initiates a service call. Basically, all nodes react to input as follows. When they receive a message (i.e. a piece of information), they decide whether they send information upwards or not. For the creation of new messages, nodes often use parts of the received information. Additionally, they attach a confidence to the fired message. This may support the decision-making processes of nodes at higher levels. The decision whether, what, and with which confidence they fire depends on the particular usage. All of that is implemented by rule sets. In the following we describe common node types.

Basic leaf nodes (called *vocabulary nodes*) simply match a predefined set of keywords, e.g. city names. Often, *pre-* and *postfix nodes* are used. They define a pre-/succeeding word or phrase, e.g. the prefix "from" to match the departure airport for flight booking systems. Another common type of leaf node is the *regex node* that matches input with a regular expression, e.g. to detect ZIP codes. Usually, there are two types of inner nodes: *gather nodes* and *selection nodes*. Gather nodes simply collect information sent by their child nodes; e.g. an *address* gather node might collect *ZIP*, *city*, and *street* facts. Selection nodes decide which information will be passed on; e.g. a *subject* selection node might decide whether the user talks about restaurants or cinemas. Most commonly, selection nodes decide on the basis of the confidences of the respective inputs. However, other strategies are possible. The most common type of root node is the *call node*. It creates a service call and passes it to the *service broker*. The service broker is a sub-system that selects the most suitable service provider(s) for the call. A language generation module post-processes the response to the service call. The result is presented to the user.

III. Related Work

In this section, we first review proprietary virtual assistants (see subsection III-A). Then, we present platforms for developers to create systems with CIs, e.g. chatbots and the like (see subsection III-B). Finally, we discuss work from the research area natural language interfaces to databases (see subsection III-C).

A. Proprietary Virtual Assistants

Virtual assistants both for home environments (e.g., Amazon Echo, Google Home) and mobile use (e.g., Apple Siri, Google Assistant, Microsoft Cortana) are omnipresent. However, little is known about the technology behind these assistants. US patent no. 8,677,377 [3] suggests that Apple's Siri makes use of Active Ontologies. Amazon's Alexa interacts with third-party services through so-called "skills". Google uses a knowledge graph to answer user queries[2] and Amazon states that they use "deep learning technologies"[3] to develop Alexa.

B. Platforms for Conversational Interfaces

Besides proprietary assistant systems, all major companies provide platforms for developers to create CIs, e.g. IBM Watson[4] , Microsoft LUIS[5] , Facebook's WIT[6] , Amazon Lex[7] , and Google's Dialogflow[8] .

Dialogflow provides an API for developers to add natural language processing capabilities to applications. One can build CIs to create chatbots and the like. Developers build so-called *agents*. Agents determine the user's intent from an utterance. Each agent deals with one or more *intents*, e.g. requests for weather forecasts or web cams. When an agent grasps an intent, it extracts relevant information and passes it to a connected service. Since users may express intents in different ways, it is necessary to provide Dialogflow agents with a variety of different phrases for each intent. The developer does not only have to provide the phrases but also annotate actions and parameters in all phrases. One must also specify the parameter mapping, i.e. which word must be translated to which parameter in the service call.

Almond [4], developed by the Stanford University, is an open and crowd-sourced platform to build virtual assistants. Almond is composed of three modules: a virtual assistant, the knowledge base *Thingpedia*, and the runtime environment

[2]Andreas Blumauer, Semantic Web Company: https://semantic-web.com/2018/08/23/knowledge-graphs-connecting-dots-increasingly-complex-world/
[3]Amazon Alexa: https://developer.amazon.com/de/alexa/science/
[4]Watson Assistant: https://www.ibm.com/cloud/watson-assistant/
[5]LUIS: https://www.luis.ai/
[6]wit.ai: https://wit.ai/
[7]Amazon Lex – Build Conversational Bots: https://aws.amazon.com/lex/
[8]Dialogflow: https://dialogflow.com/

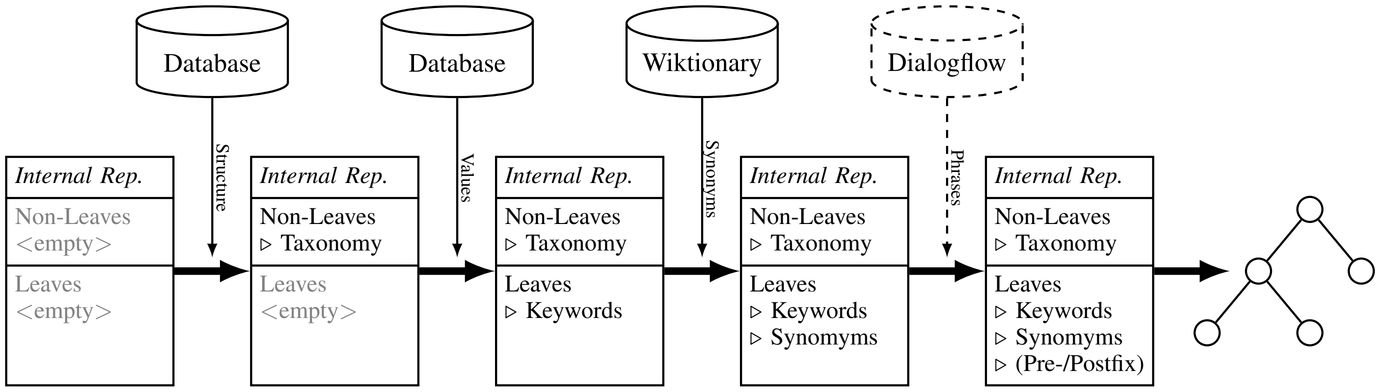

Fig. 2. The process to automatically create AOs. We use an (initially empty) internal representation that is successively populated from different sources: database structures and values, synonyms from Wiktionary, and – optionally – sample utterances from Dialogflow or similar sources.

ThingSystem. The virtual assistant translates natural language queries to the specialized programming language *ThingTalk*. The code is executed on ThingSystem, which creates service calls to the respective providers. Developers contribute to Thingpedia, a crowd-sourced public knowledge base of open APIs and their natural language interfaces. They specify trigger-actions such as If-This-Than-That (IFTTT) structures. The virtual assistant uses trigger-actions during code generation. The evaluation of the prototype shows that about 40% of tasks provided by a user familiar with the system are understood by Almond.

C. Natural Language Interfaces to Databases

Natural language interfaces to databases (NLIDB) have been studied since the late sixties; the associated conference (NLDB) is in the 24th iteration. Androutsopoulos et al. [5] give an introduction to the research area. Pazos R. et al. [6] review the state of the art. In the following, we briefly present representatives for common approaches.

Many NLIDB systems use specialized grammars. Rao et al. [7] use a *semantic grammar* to translate natural language to SQL queries. With customized production rules SQL queries are directly derived from the words in the utterances. Some NILDB systems employ intermediate representations to grasp the intent of the natural language query. C-Phrase [8] uses an intermediate representation based on first order logic supplemented by additional higher-order predicates. It uses synchronous context-free grammars and lambda calculus expressions to convert natural language queries into the intermediate representation. Then, the intermediate representation is converted to an SQL query. More recent NILDB approaches use machine learning techniques. Neelakantan et al. [9] use neural networks to map from language to SQL. Zhong et al. [10] employ reinforcement learning to improve quality.

All the above need developer-generated information: sample sentences, specialized grammars, or rules sets. We derive AOs directly from the database with minimal human effort.

IV. Automatic Generation of Active Ontologies

We aim to create CIs for virtual assistants, chatbots, and the like (semi-)automatically. This way, we lower the effort for service providers to make their data accessible through a natural language interface. The sole precondition for our approach is that the service provider stores all information it wants to provide through the CI in databases. For example, if a service provider wants to publish information about touristic attractions it must store addresses, ratings, and the like of restaurants, museums, or galleries in accessible databases.

As underlying technology we use Active Ontologies. Usually, AOs are built manually. However, we have shown that it is possible to create AOs (semi-)automatically from web forms [11], [12]. In this work, we leverage the information provided by databases.

In contrast to web forms, databases always provide values, i.e. instances of concepts. We use the database values and synonyms to raise the linguistic competence of our AOs. Moreover, if sample utterances are present, we are able to add even more linguistic competence through an automatic extraction of common phrases. Our AOs are able to react to previously unseen user requests. Furthermore, they reply to composed requests.

Figure 2 shows an overview of our approach. We use an internal representation that is populated step by step. First, we extract the database structure and infer the concept hierarchy. Then, we create basic leaf nodes from the values in the database tables. Next, we add synonyms from Wiktionary, where applicable. If utterance samples are present, we add common phrases to the respective concepts. Finally, we generate the Active Ontology from the internal representation.

In the upcoming subsections we first describe the extraction of the concept hierarchy of the AOs from database schemes (subsection IV-A). Then we show how we increase the linguistic competence, i.e how we add the leaf nodes to the AOs (subsection IV-B). Finally, we discuss the limitations of the approach, i.e. what a developer must review or add to the automatically generated AOs (subsection IV-C).

A. Taxonomy Extraction from Database Structures

AOs are hierarchic domain models. Inner nodes typically join information (gather nodes) to create a more complete view to the user's request. We observed that database schemes follow the same intuition. For example, a database table *restaurant* may store information about *addresses*. The *addresses* again may be composed of a *ZIP*, a *city*, and a *street*. The *restaurant* table itself maybe used in different contexts. Thus, the *restaurant* table defines a concept that includes the subconcepts *address*, *ZIP*, *city*, and *street*. For our prototype we use deductive databases. [9]. However, our approach can also be applied to relational databases as both database types store data in a structured way.

The structure is used to create the concept hierarchy of the Active Ontology, i.e the hierarchy of inner nodes. The database type only affects the way the database structure is extracted. Deductive databases contain triples that consist of an internal ID, the name of the property, and the property value. A property value may refer to another internal ID. For example, information about a restaurant is represented as follows:

```
[id01, @type, restaurant]
[id01, name, The Golden Eagle]
[id01, address, id01.address]
[id01.address, city, Karlsruhe]
[id01.address, ZIP, 76131]
```

We collect all property names and create a concept for each. Through the references in property values we infer hierarchies. For the above example the hierarchy in Figure 3 arises. Concept nodes can either be converted into gather or selection nodes. For our prototype we decided to only create gather nodes, because this is the best choice in most cases. Even though we found that for some concepts selection nodes would be the better choice, we were not able to come up with a generic rule to make this decision.

B. Leaf Node Generation

Now that we have created the hierarchy of concepts we need to extend the AO with linguistic competence. Up to now, the AO can join information only (and create a service call). However, it can not gather any information from a natural language request at all. Therefore, we create leaf nodes that match certain words or phrases in the utterance. We connect these leaf nodes to the respective inner nodes. For example, to create a leaf node that recognizes restaurants by their names we can use the list of restaurants obtained from the database and attach it to the inner node *name*.

[9]A deductive database is a database equiped with a rule set. The rules are written in dialog, a simplified variant of logic programming [13], [14]. The deductive component of the database deviates additional knowledge from the data via rule executions. Queries are also composed in datalog. Ramakrishnana and Ullman describe deductive database systems as, "[...] database management systems whose query language and (usually) storage structure are designed around a logical model of data. As relations are naturally thought of as the 'value' of a logical predicate, and relational languages such as SQL are syntactic sugarings of a limited form of logical expression, it is easy to see deductive database systems as an advanced form of relational systems. [15]"

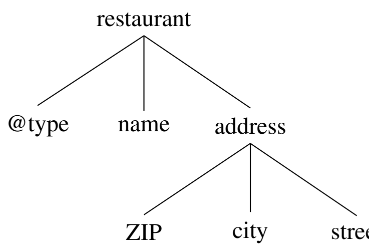

Fig. 3. A hierarchy extracted from database property names.

Besides vocabulary nodes, we generate all kinds of common node types (see section II), e.g. pre- and postfix nodes and specialized nodes such as date nodes. We use information from different sources to create leaf nodes. Next, we describe the sources, which kind of information we extract, the strategies to create leaf nodes, and which type of leaf nodes we create respectively.

1) Database Values: We create vocabulary, regex, and date nodes from database (property) values. For all string-valued properties we create vocabulary nodes. We join all values that belong to the same property (name). Thus, all names of restaurants are joined in a single vocabulary node. For numeric database values, e.g. phone numbers, ZIP codes, lengths, and the like, we use regex nodes. This way, we are able to extract any numeric information from the natural language input. Finally, to cope with specific phrases that contain date or time information such as "the day after tomorrow" we use date nodes. Date nodes recognize such phrases in the input and transform them into a machine-readable format.

2) Lexical Databases and Dictionaries: To amplify the linguistic competence of Active Ontologies one can add synonyms from lexical databases or dictionaries such as WordNet [16], [17], Wiktionary, or OpenThesaurus[10]. For our prototype we used Wiktionary for two reasons. First, as the values in our test databases are from Austrian service providers we need to recognize German utterances; Wiktionary provides the most extensive collection of German synonyms. Second, synonyms listed in Wiktionary are of high quality.

To retrieve synonyms, we first query Wiktionary for synonyms of the respective word. For each synonym we again look for synonyms. If no synonyms are available, we extract the so-called *similar words*. For each similar word we also search for synonyms. The list of synonyms and similar words is stored in a leaf node. Note that the leaf node does not pass the synonym to its parent nodes. Instead, the value of the associated property is used. For example, the leaf node that is supposed to recognize the word "Gaststätte" (German for "restaurant") and all its (German and English) synonyms, e.g. "inn", "restaurant", or "eatery" passes the value *Gaststätte* to its parent nodes even if it recognizes the word "restaurant"

[10]OpenThesaurus: https://www.openthesaurus.de

in an utterance. This also reliefs us from the issue that some values are German and others are English.

3) Utterance Samples: So far the generated AOs are capable to match keywords (or phrases) and their synonyms; they additionally detect dates and other regex-specified values. However, some user queries do not mention keywords directly. For example, a user might ask, "Where can I get pasta and tiramisu?" The utterance neither mentions restaurants directly nor the keyword "address". However, obviously the user is interested in the address of an italian restaurant. To overcome this limitation we automatically add common phrases that hint at a concept. Technically, such phrases form pre- and postfix nodes. We extract the phrases from sample utterances.

We are aware that sample utterances are not available for all domains/services. Thus, this step is optional. Alternatively, a developer can manualy add pre- and postfix nodes to the AO.

For our prototype we use sample utterances from Dialogflow agents. Therefore, we extract all sample utterances from an agent and use the *intent* and *entities* as labels. With the help of the intent we are able to discover the appropriate AO part, e.g. restaurant requests. The entities have been linked to the respective database properties by the developer of the Dialogflow agent. Thus, we can determine the according concept in the AO. Additionally, often synonyms for entities are given. We add these to the synonym vocabulary nodes attached to the respective concept.

To extract pre- and postfixes we consider all entities in an utterance. We use all words preceding an entity as prefix and all succeeding as postfix. Of course, we stop discovery at the next entity. We consolidate all pre- and postfixes from all utterances for the same entity. Then we create one pre- and one postfix node per entity and attach them to the respective concept node. Given the Dialogflow sample utterance, "[Find me a [place to sleep]$_{\text{ent:type}}$ in [Lisbon]$_{\text{ent:city}}$]$_{\text{int:hotel}}$", we can identify the synonym *place to sleep* for the concept *@type* of the hotel AO part. Additionally, we extract the prefixes *find me a* (concept *@type*) and *in* (concept *city*) as well as the postfix *in* (concept *@type*).

C. Limitations

To automatically generate AOs, we have to make design decisions. For example, since we cannot automatically decide between selection and gather nodes, we create gather nodes for all concepts. However, in rare cases selection nodes are more appropriate. A developer can determine which node type is most suitable and select it accordingly. Another design decision concerns database values. We create word lists (for vocabulary nodes) for all string-valued properties. However, regex nodes might be more suitable to extract particular words or phrases, e.g. sub-strings.

In some cases, the extracted pre- and postfixes are either to specific or not specific enough. Modifying the phrases could improve matching with utterances. Entirely missing prefixes cause false positive matches and consequential conflicting hypotheses. Therefore, we expect that adding missing pre- and postfixes improves the accuracy of the AOs considerably. The same applies to synonyms; a manual review improves accuracy since an automatic extraction of synonyms is error-prone.

V. Evaluation

We evaluate our approach in three domains: tourism, hotels, and webcams. For each we have a deductive database and Dialogflow agents. The tourism dataset comprises information about local restaurants, ski rentals, events and the like. The hotel and web cam datasets contain information about the names and locations of hotels and web cams. In total, we included 3,861 data elements: 1,873 elements from the tourism, 1,303 from hotel, and 685 from web cam dataset.

We generate AOs for each domain in different variants. The basic AOs are created from the databases and synonyms only. All other configurations use sample utterances from Dialogflow agents to add pre- and postfix nodes. We use phrases from individual domains and all possible combinations of domain. For each domain we use all available samples provided by the agents for phrase extraction.

Note that any AO is enriched with pre-/postfixes where applicable. In other words, if we extract phrases from the web cam domain only but the other domains share concepts (e.g. locations) the respective phrases are added to all AOs. With that, we asses the impact of phrase extraction (the more the better?) and synergy effects (can we employ phrases from other domains?).

The database values are a mixture of English and German words. However, the sample utterances are in German. During AO generation we translate database values in the synonym mapping step (see subsubsection IV-B2). The generated leaf nodes are defective in some cases due to incorrect translations. However, we cannot measure the effect (if existing).

To asses the quality of our generated AOs we compare them to the Dialogflow agents. Therefore, we match the replies for a request returned by the AOs against Dialogflow. We assume that replies given by the Dialogflow agents are always correct, since all sample utterances have been manually annotated with intent and entity labels by developers. The Dialogflow agents contain 1,652 sample utterances (tourism: 1,140, hotels: 378, web cams: 134) with 4,597 entities (tourism: 2,430, hotels: 727, web cams: 1,440). To limit the effort, we analyzed a randomly sampled subset. We used 100 test utterances per domain for the configuration with no sample utterances, i.e. AOs built from databases and synonyms, and for the configuration with samples from *all* domains. For the other configurations we used 50 samples each.

The results of our study are depicted in Table I; the highest values for each configuration and per measure are highlighted. The first column shows the configuration for the phrase extraction, i.e from which domain we took samples during AO generation: (T)ourism, (H)otel, and (W)eb cam. For all configurations we determine accuracy, recall, precision, and F_1 for the test utterances for individual domains (T, H, and W) and all domains ($\forall$). Note that we determine accuracy on a per-reply level, i.e. we count only answers that exactly match the Dialogflow result as correct. For precision, recall, and F_1

TABLE I

EVALUATION RESULTS FOR DIFFERENT PHRASE EXTRACTION SETTINGS. WE EITHER USED NONE, PHRASES FROM TOURISM (T), HOTEL (H), AND WEBCAM (W) AGENTS, OR VARIOUS COMBINATIONS.

Phrase	**Accuracy**				**Recall**				**Precision**				**F$_1$**			
Extr.	**T**	**H**	**W**	∀	**T**	**H**	**W**	∀	**T**	**H**	**W**	∀	**T**	**H**	**W**	∀
none	0.11	0.00	**0.16**	0.09	0.12	0.06	**0.29**	0.18	0.34	0.60	**0.79**	0.58	0.18	0.11	**0.42**	0.27
T	**0.30**	0.02	0.22	0.18	**0.50**	0.22	0.29	0.35	**0.72**	0.45	0.42	0.55	**0.59**	0.30	0.34	0.40
H	**0.14**	0.00	0.06	0.06	0.04	0.10	**0.26**	0.17	0.13	0.40	**0.75**	0.48	0.06	0.16	**0.39**	0.25
W	0.14	0.02	**0.24**	0.13	0.19	0.31	**0.51**	0.34	0.42	0.63	**0.84**	0.66	0.26	0.42	**0.63**	0.45
T+H	**0.26**	0.04	0.22	0.17	0.43	0.25	**0.55**	0.28	**0.71**	0.47	0.45	0.52	**0.55**	0.35	0.37	0.36
T+W	0.32	0.02	**0.40**	0.24	0.44	0.33	**0.55**	0.46	0.78	0.52	**0.96**	0.79	0.57	0.40	**0.70**	0.58
H+W	0.02	0.14	**0.34**	0.16	0.18	0.24	**0.55**	0.33	0.26	0.50	**0.78**	0.51	0.21	0.32	**0.65**	0.43
T+H+W	**0.30**	0.08	0.28	0.22	0.45	0.35	**0.50**	0.45	0.74	0.62	**0.79**	0.78	0.54	0.43	**0.61**	0.57

we compare each element (i.e. *entity* in Dialogflow, *concept* in the AO) of the service call separately.

If an element is identified correctly, it is considered a true positive. Elements that were extracted mistakenly, i.e. they do not match an element extracted by the Dialogflow agent, are false positives. Any missing elements account for false negatives. Assuming that the service call elements for the exemplary request, "Find me a hotel in Lisbon," are:

- Dialogflow: [hotel]$_{\text{ent:type}}$, [Lisbon]$_{\text{ent:location}}$
- Active Ontology: [hotel]$_{\text{ent:type}}$, [sauna]$_{\text{ent:feature}}$

In this example, *hotel* is a true positive, *Lisbon* a false negative, and the *sauna* accounts for a false positive.

The results indicate that our approach is feasible. Using the databases and synonyms for values only we achieve a precision of 58% (recall 18% and F$_1$ 27%). However, the accuracy (9%) shows that there is still much room for improvement. The evaluation also shows that enriching the AOs with sample utterances improves the quality in almost all cases. The best accuracy (40%), recall (55%), precision (96%), and F$_1$ (70%) are achieved for web cam requests when we use sample utterances from both the tourism and web cam domains. This clearly shows that phrases from other domains improves the linguistic competence of AOs. The effect can be further assessed with the results from the tourism domain with phrases extracted from the web cam domain (row *H*). Accuracy, recall, and precision improve in comparison to the configuration without any phrase extraction. This is due to shared concepts in both domains and similar sample utterances. For example, our approach extracts the prefixes *are there* for the concept `@type` and *in* for `location` from the web cam sample: "Are there [live pictures]$_{\text{ent:type}}$ in [Salzburg]$_{\text{ent:location}}$?"[11] The prefixes can be applied to requests from the tourism domain such as, "Are there ski schools in Seeberg?"[12]

However, extracting sample utterances from the hotel domain often degrades the results as one can see, e.g. in the rows *H* and *H+W*. This is due to requests with many enumerations that occur frequently such as, "I am looking for a designer hotel with free parking, sauna, whirlpool, wifi, tennis court, and a restaurant voucher."[13] Pre- and postfixes extracted from such samples produce both false positives and false negatives. For example, if the sample does not have an entity annotation for *free parking* our approach extracts the prefix *with free parking* for the concept `feature` (instance *sauna*) that causes false positives. Vice versa, if *free parking* has an annotation our approach misses the (correct) prefix *with* as we discontinue phrase extraction at the next annotation.

The best results for the hotel domain are achieved if we extract phrases from the web cam domain (see rows *W*, *T+W*, *T+H+W*). The results for tourism and web cams both improve when phrase samples from their own domain are used.

Accuracy values are low. However, using any kind of sample utterances increases the accuracy in all domains in almost all cases and reaches 40% in the best case.

We identified three errors classes that primarily affect the results. Rare errors are false positive elements due to missing pre-/postfixes. Concepts never mentioned in sample utterances are still part of the AO. These concept nodes have vocabulary, regex, or date nodes attached. Therefore, they might match requests even though the request have different intents. For example, a hotel might have a laundry but no sample mentioning it. The hotel AO then has a vocabulary node that reacts to any "laundry" in user requests. This may create false positives from requests such as, "I'm looking for laundries."

Another error class is the incorrect selection of service calls. Our approach consolidates all concepts from each domain in a single AO. This way, we assure that only one service call is created from all hypothesis (combinations of elements). In some cases correct hypothesis have lower confidences than incorrect ones. If so, the root node chooses the wrong hypothesis and creates an incorrect service call.

Missing or incorrect synonyms form the third error class. Missing synonyms cause false negatives, while incorrect synonyms may produce false positives. An example for the latter is the German synonym *Haufen* (engl. *pile/bunch*) for the word *Berg* (engl. *mountain*) that is often used in colloquial idioms. This causes the AO part responsible for information on ski tours (to particular mountains) to react to phrases such as "ein Haufen Leute" (engl. "a bunch of people").

[11]Original: "Gibt es [Live Bilder]$_{\text{ent:type}}$ in [Salzburg]$_{\text{ent:location}}$?"

[12]Original: "Gibt es Skischulen in Seeberg?"

[13]Original: "Ich suche ein Designerhotel mit gratis Parkplatz, Sauna, Whirlpool, WLan, Tennisplatz und einem Restaurantgutschein."

VI. Conclusion and Future Work

We have presented an approach to automatically create virtual assistants from databases. Our virtual assistants use Active Ontologies to model the problem domain and process user utterances.

To generate AOs we first infer a concept hierarchy from database schemes and create word lists and regular expressions from extracted database values. We extend the word lists with synonyms from Wiktionary. Additionally, if sample utterances are present, we build pre- and postfix nodes to enhance the linguistic competence of our AOs.

Our evaluation in the domains tourism, hotel, and web cams shows that our approach successfully generates AOs that extract relevant information from user utterances. When we use sample utterances, precision and recall increases. Our approach successfully learns phrases from a domain (e.g. tourism) and uses it for queries from another (e.g. hotel).

We plan to improve the AO generation process. For the time being, we create one AO per domain. We experiment with smaller AOs, that may extract intents with higher confidences. This also evades the duplication of subtrees (e.g. for addresses of restaurants and cinemas).

Another improvement involves the general structure of the AOs. In some caeses multiple pre- and postfix nodes fire at the same time, which makes intent extraction ambiguous. An additional layer of selection nodes that selects the most reasonable pre-/postfix may be beneficial.

We observed that some phrase segments are more meaningful than others. Therefore, we have implemented an alternative to pre-/postix nodes. Instead of using full pre-/postfix phrases we extract single words and create a vocabulary node for each. We attach all of them to an additional layer of gather nodes that represent the original pre-/postfixes. This way, we create subtrees that capture bags of words. We calculate the TFIDF for each word per intent (each intent is a document and all intents correspond to the document set). The TFIDF value is used as confidence for the new vocabulary nodes.

A first case study shows the potential of this approach. We used 239 sample utterances from Dialogflow agents for the intent "request opening hours". Additionally, we created 13 synthetic requests for addresses (e.g., *What is the address of the restaurant The Toothless Shark in Cologne?*).

We created bag-of-words subtrees for both intents and integrated them into the AO. On the test set the AOs answered opening hours request with an accuracy of 65% and addresses with an accuracy of 23%. The low accuracy for the latter is due to the small number of sample utterances.

With the bag-of-words subtrees the AOs can extract multiple intents from single user utterance. For the time being, virtual assistants such as Dialogflow cannot combine different intents. Thus, our approach improves the state of the art. Composed requests such as, "Give me the address and the opening hours of the hotel Tivoli Oriente in Lisbon," achieve an accuracy of 40%. In the future, we plan to further investigate the bag-of-words-approach and evaluate it on a larger data set.

VII. Acknowledgement

The evaluation of our approach was partially supported by Onlim GmbH[14] that provided the data for testing. We thank Dr. Ioan Toma (Onlim GmbH) who greatly assisted our research.

References

[1] D. Guzzoni, C. Baur, and A. Cheyer, "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, Dec. 2006, pp. 417–420.

[2] D. Guzzoni, "Active: A unified platform for building intelligent applications," PhD Thesis, École Polytechnique Fédérale De Lausanne, Jan. 2008.

[3] A. Cheyer and D. Guzzoni, "United States Patent: 8677377 - Method and apparatus for building an intelligent automated assistant," USA Patent 8 677 377, Sep., 2005.

[4] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam, "Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 341–350.

[5] I. Androutsopoulos, G. Ritchie, and P. Thanisch, "Natural language interfaces to databases – an introduction," *Natural Language Engineering*, vol. 1, no. 01, pp. 29–81, Mar. 1995.

[6] R. A. Pazos R., J. J. González B., M. A. Aguirre L., J. A. Martinez F., and H. J. Fraire H., "Natural Language Interfaces to Databases: An Analysis of the State of the Art," in *Recent Advances on Hybrid Intelligent Systems*, ser. Studies in Computational Intelligence, O. Castillo, P. Melin, and J. Kacprzyk, Eds. Springer Berlin Heidelberg, 2013, no. 451, pp. 463–480.

[7] G. Rao, C. Agarwal, S. Chaudhry, N. Kulkarni, and D. S. Patil, "Natural language query processing using semantic grammar," *International journal on computer science and engineering*, vol. 2, no. 2, pp. 219–223, 2010.

[8] M. Minock, "C-Phrase: A system for building robust natural language interfaces to databases," *Data & Knowledge Engineering*, vol. 69, no. 3, pp. 290–302, 2010, special Issue: 13th International Conference on Natural Language and Information Systems (NLDB 2008) – Five selected and extended papers.

[9] A. Neelakantan, Q. V. Le, M. Abadi, A. McCallum, and D. Amodei, "Learning a natural language interface with neural programmer," *arXiv preprint arXiv:1611.08945*, 2016.

[10] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," *CoRR*, vol. abs/1709.00103, 2017.

[11] M. Blersch and M. Landhäußer, "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, Jul. 2016.

[12] M. Blersch, M. Landhäußer, and T. Mayer, "Semi-automatic Generation of Active Ontologies from Web Forms for Intelligent Assistants," in *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE '18. Gothenburg, Sweden: ACM, 2018, pp. 28–34.

[13] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, Mar. 1989.

[14] M. Krötzsch, S. Rudolph, and P. H. Schmitt, "A closer look at the semantic relationship between Datalog and description logics," *Semantic Web*, vol. 6, no. 1, pp. 63–79, 2015.

[15] R. Ramakrishnan and J. D. Ullman, "A survey of deductive database systems," *The Journal of Logic Programming*, vol. 23, no. 2, pp. 125–149, 1995.

[16] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219748

[17] C. Fellbaum, *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[14]Onlim GmbH: https://www.onlim.com