Designing Voice-Controllable APIs

Matúš Sulír, Jaroslav Porubän
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Košice, Slovakia
{matus.sulir, jaroslav.poruban}@tuke.sk

***Abstract*—The main purpose of a voice command system is to process a sentence in natural language and perform the corresponding action. Although there exist many approaches to map sentences to API (application programming interface) calls, this mapping is usually performed after the API is already implemented, possibly by other programmers. In this paper, we describe how the API developer can use patterns to map sentences to API calls by utilizing the similarities between names and types in the sentences and the API. In the cases when the mapping is not straightforward, we suggest the usage of suitable annotations (attribute-oriented programming).**

***Index Terms*—application programming interface (API), methods, parameters, voice control, natural language commands**

I. Introduction

Voice control is slowly becoming a mainstream form of human-computer interaction. For this reason, developers often integrate voice control into the applications being developed. In this article, we will focus on simple, one-sentence commands after which we expect the computer to perform the desired action. Such a process consists of two main parts [1]: the translation of the voice input to a natural language sentence and the selection of an appropriate action according to the meaning of the sentence. In this paper, we suppose the voice recognition is implemented by a third-party library or service. Therefore, we will focus on the second part: the mapping of a sentence to the desired action.

Since we are interested in the implementation of the voice control from the developer's view, by an action we mean a call to an appropriate method in an API (application programming interface) with correct parameters. There already exist many approaches to map a natural language sentence to API calls. Some of them utilize probabilistic grammars and heuristics [2], others perceive phrases only as lists of keywords [3]. An increasingly popular approach is the application of machine learning to translate natural language sentences to API snippets [4].

The common sign of these approaches is that they perceive voice control only as an afterthought – when the API is already implemented, probably by other developers. In contrast to them, we will show how the API can be tailored to be voice-controllable, predominantly by exploiting the similarity of names and data types used in the sentences and the API. For example, a sentence "open the door for 5 seconds" could be interpreted as a call to `door.open(5)` during the execution of a program, provided there is a method `void open(int seconds)` in the class `Door`. When such similarities are not observed, we suggest utilizing annotations (attribute-oriented programming) to specify the necessary details of the sentence-to-API mapping.

Throughout the paper, we also discuss how such APIs could be analyzed by a voice command recognition framework, which would then execute an appropriate action given a natural language sentence. A prototype implementation of such a framework, supporting many of the described features, is available online[1].

II. Mapping Patterns

In this section, we will gradually describe individual patterns of mapping between a natural language sentence and an API call. Our examples will be related mainly to the hardware device control domain, but it is not restricted to it. Although we will use Java, the approach is generally applicable to any object-oriented programming language with sufficient metaprogramming features, static typing is also helpful for some patterns.

A. Verbatim Name Mapping

Suppose we would like to turn on light in a room after saying "turn on light". The most primitive way to map this voice command to a method in the API would be to name the method `turnOnLight` and annotate it with a marker annotation `VoiceControllable`, so our framework could recognize it as a voice-controllable function:

```
@VoiceControllable
public static void turnOnLight() {
    // implementation turning on the light...
}
```

The name of the method would be split into individual words and transformed to lowercase. Using exact string matching, the input "turn on light" could be then mapped to this method, which would be executed without any parameters.

Of course, having one class with many static methods is a bad programming practice. Therefore, suppose we have separate classes to control lights, monitors, speakers, and other devices. For simplicity, let us say these classes are named `LightService`, `MonitorService`, etc. In modern application frameworks, service constructors are usually not called

[1] https://github.com/sulir/voice-control-demo

 This is the accepted version of: M. Sulír, J. Porubän. Designing voice-controllable APIs. 2019 IEEE 15th International Scientific Conference on Informatics, pp. 393–398. http://doi.org/10.1109/Informatics47936.2019.9119302

manually – the framework initializes them using techniques such as dependency injection. Thus we will assume we have an instance of every necessary class readily available and we will focus on calling methods of this object. We can transform the previous example to this code:

```
@VoiceControllable
public class LightService {
    public void turnOn() {
        // implementation turning on the light...
    }
}
```

The annotation over a class means all its methods are now "voice-controllable". Our framework searches for all such classes and their methods, building potential command-to-method mappings. Since `Service` is an implementation-oriented name irrelevant to the problem domain [5], our framework strips it from the class name when scanning it. The rest of the class name is appended to the method names. Therefore, the `turnOn` method of class `LightService` will be called when the command "turn on light" is issued.

B. Minor Variations

Since natural language is not restricted to exact phrases, we must count with some variations in the commands. First, we should stem all words in both the command and the class/method names using an appropriate stemmer. This way, we allow for variation in individual words since only their root forms are compared. For example, the command "turn on lights" can now execute the aforementioned method, even if "Light" is singular in the class name.

Next, we should allow permutations of the words. For example, "turn on lights" can be expressed also as "turn lights on". In general, we need to find a compromise between allowing all permutations (which allows diverse sentences at the price of higher ambiguity) and supporting only certain kinds of permutations.

We can also allow extraneous words to be present in the command. Typical examples include stop words ("the" in "turn on the light") or courtesy phrases ("please turn on the light"). An extreme example is the sentence "Please be so kind and turn on the light, dear." As long as the extraneous words are not contained in the names in our API, we can safely remove them from the sentence without any ambiguity. To enable more precise decisions, we can annotate the method with words we expect to encounter in the command. For example, we would like to execute the method `turnOff` by the command "turn off all lights":

```
@ExtraWord("all")
public void turnOff() { ... }
```

A similar case is the non-presence of certain words from API names in the natural language command. For instance, the command "light off" does not contain the word "turn" from the method `LightService::turnOff`. To help the mapping system to make a correct decision, we can annotate the method with `@OptionalWord("turn")`.

By allowing all of the mentioned variations at once, we can achieve a system which is highly flexible and accepts a wide range of different sentences – at the price of potential ambiguity. Essentially, the framework should find a method in the API which has the highest number of words common with the input. When there are ties or the number of common words is too low, we can start taking the `ExtraWord` and `OptionalWord` annotations into account. If the situation is still indecisive, the voice assistant can simply list the most probable options and ask the person to select one of them.

C. Parameters

Now we will take into account methods with parameters. The simplest case is a method with one numeric parameter:

```
public void turnOn(int number) { ... }
```

Typical commands to execute this method are: "turn on light number 1" and "turn on light 1". In the first sentence, the argument value follows the parameter name "number", which means the argument can be matched by its name. In the second case, the parameter is matched only by its type – the input sentence contains a number and the API method has a numerical parameter.

Mapping enum-typed parameters is straightforward too. Suppose `Position` is defined as an enumeration:

```
public enum Position {
    LEFT, MIDDLE, RIGHT
}
```

We also have a method with a `Position` parameter:

```
public void turnOn(Position position) { ... }
```

Then this method can be called by the command "turn on the left light". On the other hand, mapping of string parameters is more problematic:

```
public void turnOn(String name) { ... }
```

If the user says "turn on the light named 'front'", the system can use the matching based on parameter names. However, mapping the sentence "turn on the front light" to this method is questionable in general, since we do not have any way to enumerate all possible valid values of a `String` parameter. A potentially dangerous way to determine if this method matches the command is trying to execute it and observing if it does not throw an exception. There exist better ideas, though. If the list of valid values is fixed at a compile time, it can be supplied directly:

```
public void turnOn(
    @Values({"front", "back"}) String name
) { ... }
```

In such cases, however, it is much better to use enumerations instead. Most often, the set of valid values can be enumerated only at runtime. Then we can create a class with a method returning these values:

```
public class LightNames implements ValueSet<String> {
    public Set<String> getValues() {
        return configuration.getValidLightNames(...);
```

```
    }
}
```

Subsequently, we connect this list with a parameter using an annotation:

```
public void turnOn(
    @ValidValues(LightNames.class) String name
) { ... }
```

Now we will consider parameters of any class in general, e.g., a parameter of type `Color`:

```
public void setColor(Color color) { ... }
```

In the command "set light color to green", we do not know how to map "green" to `Color(0, 255, 0)`. It is possible that `Color` has a constructor accepting a string, in which case we can utilize it. However, this still does not solve the problem of the enumeration of all valid values described in the previous example. Therefore, we suggest using a string-to-object mapping via annotations. Each parameter of a custom class should be annotated with a mapper:

```
public void setColor(
    @StringMapping(ColorMapper.class) Color color
) { ... }
```

Then the `ColorMapper` class can look like this:

```
public class ColorMapper
implements StringMapper<Color> {
    public Map<String, Color> getMap() {
        return Map.of(
            "red", new Color(255, 255, 0),
            "green", new Color(0, 255, 0),
            ...
        );
    }
}
```

This way, we can both enumerate all possible values of a `Color`-typed parameter and map a string such as "green" to a `Color` object. The approach is not limited to a constant associative array of string–color pairs: the mapping could be also dynamically generated by arbitrarily complicated code. Note that since the mapping is specified in a separate class, we did not modify the original `Color` class in any way – it can be defined in a third-party library without problems.

Next, we consider a method with multiple parameters:

```
public void setColor(int number, Color color) { ... }
```

As long as the parameters are of distinct types and the sets of valid values of all types are disjunctive, we can map various commands to a method execution without too much ambiguity: e.g., "set light 3 to blue", "I would like yellow color for light 4". However, methods with multiple parameters of same or similar type are more problematic:

```
public void setBrightness(int light, double brightness)
{ ... }
```

Here, "set light 1 to brightness 50" can be mapped to a correct execution using the parameter names. However, the sentence "set brightness of light number 2 to 30" could be probably mapped only by the position of arguments, which would not work correctly if the parameters were switched in the API.

Note that parameters have an important role in the process of the API method selection. Since it is not valid to call a method without all mandatory (non-null) arguments filled, they must be all present in the input sentence. For example, if the command does not contain any number, we can assume it does not match any method with a numeric parameter.

D. Collections

Suppose we want to dim multiple lights at once. A collection, such as a set, is an ideal candidate for this:

```
public void dim(Set<Integer> lights) { ... }
```

The content of the parameter `lights` can be expressed by enumerating all values or by specifying a range: "dim lights 1, 7 and 9" or "dim lights 6 to 10". In case the expected range can be very large, we recommend using a lazy collection denoted by an interface, such as `Iterable<Integer>`.

Similarly to numeric collections, we accept collections of enumerations or other classes:

```
public void dim(Set<Position> lights) { ... }
```

Naturally, ranges are not supported in such cases. We can still fill this set with values by naming them all, though: "dim the left and middle light". Alternatively, commands such as "dim all lights" should work if there is a way to enumerate all valid values of the parameter programmatically.

E. Synonyms

Many words have synonyms which can a user utter instead of the words used in the API names. A natural way to cope with such situations is the usage of thesauri and lexical databases such as WordNet [6]. Any word in the API can be then replaced by its synonym to perform a successful match with the natural language command. However, consider the following example of a method in class `ScreenService` (controlling a multi-monitor ambient user interface [7]), where `State` is an enumeration with values `ON` and `OFF`:

```
public void set(int screen, State state) { ... }
```

We would certainly like to say not only "set screen 1 to 'on'" but also "turn screen 1 on". However, "turn" is not a synonym of "set" in general – only in this specific case. Therefore, we devised a way to specify method-local synonyms via annotations:

```
@Synonym(of = "set", is = "turn")
public void set(int screen, State state) { ... }
```

This synonym can be applied only during the matching of this particular method. Analogously, we support package-local, class-local and parameter-local synonyms using annotations over packages (in a special file `package-info.java`), classes, and parameters, respectively. It is also allowed to specify multiple synonyms per element, e.g.:

```
@Synonym(of = "screen", is = {"display", "monitor"})
@Synonym(of = "turn", is = "switch")
public class Screen { ... }
```

F. Fallback

If we encounter difficulties during the application of the aforementioned mapping patterns, we can always specify the voice commands manually. For example, suppose we have a class `SpeechService` with a method `pronounce(String sentence)` which says out loud the given sentence. Since the sentence can be completely arbitrary, matching will likely fail. Therefore, we specify the voice command as a regular expression:

```
@VoiceCommand("say (.*)")
public void pronounce(String sentence) { ... }
```

The content of the group in the parentheses will be supplied as a parameter value. Because a manually specified command has a high priority, we can successfully match commands such as "say 'I like turning off the screens'" even when they contain words present in the API.

III. Mapping Process

In our prototype implementation[2], we decided to use a simple sentence recognition algorithm designed to allow for relatively large deviations of the input sentences from the prescribed forms. Now we will briefly describe it.

First, the input sentence is matched against all fallback regular expressions (section II-F). If a match is found, the process is stopped and the annotated method is executed.

Next, for each voice-controllable method, we try to type-match all its parameters with the sentence. For example, if a method has a numeric parameter and a `Color` enumeration parameter, the sentence is searched for a numeral and a word denoting a color. The result is a list of potentially matching methods, along with word-to-parameter mappings.

For each method in this list, a score of similarity with the input sentence is calculated: Let W_M be the set of words contained in the class and method name. Let W_S be the set of words contained in the input sentence, excluding the parameter values matched in the previous steps. The score is computed as the Jaccard index [8] of these two sets:

$$\frac{|W_M \cap W_S|}{|W_M \cup W_S|}$$

If the class or method is annotated by synonyms, multiple variants of the set W_M are constructed, with the words in the identifiers gradually replaced by their corresponding synonyms. The resulting score is a maximum of the scores computed for individual variants.

Finally, the method with the highest score is executed. If multiple methods have the same score or if no method receives a score higher than a certain threshold (e.g., 0.2), the user should be asked to reformulate the command.

It is important to note that the described algorithm is not the only one possible. The ideas described in the previous section are compatible, for instance, with strict regex-based solutions based on the precise matching of sentences with regular expressions generated from the identifier names and annotations. Another possibility is a grammar-based approach, which would distinguish parts of sentences: Some of them could be derived automatically if possible, the rest will be determined by manually written annotations.

[2]https://github.com/sulir/voice-control-demo

IV. Related Work

Many approaches mapping sentences to API calls are grouped under the umbrella term of program synthesis, particularly program synthesis using natural language input [9]. Desai et al. [10] synthesize source code written in various domain-specific languages, thanks to a dataset of sentence–code pairs used for training. Gvero and Kuncak [2] synthesize Java expressions using probabilistic grammars and heuristics. In T2API, Nguyen et al. [4] perceive program synthesis as a statistical translation process from natural language to a programming language. Little and Miller [11] generate Java code from a set of brief keywords. Some synthesis approaches are focused on particular domains or technologies: e.g., SQL query generation [12], smartphone automation script synthesis [13], bot API invocations [14]. All of the mentioned approaches perceive APIs as black boxes, which are already designed. In contrast to them, our idea is to engage the API designers in the process of natural language command specification.

Landhäußer et al. [15] designed NLCI (natural language command interpreter), which has a goal similar to ours – to perform an action in the API, given a natural language sentence as an input. In contrast to us, they first transform the API into an ontology by analyzing the relationships between elements in the code and combining it with a general-purpose ontology. Furthermore, they do not support simple mapping customization via annotations.

The command execution approach by Little and Miller [3] is based on the similarity of names used in the sentence and the API. They also perceive sentences as lists of keywords, allowing for variations such as extraneous words. However, they do not allow any customization using annotations, since they consider APIs to be developed by a third party and thus not modifiable.

Naturalistic programming [16] is a paradigm aiming to make the source code look more like natural language. For example, Knöll et al. [17] discuss naturalistic types which include the mapping of natural language quantities such as "nearly all" to exact numeric intervals. Compared to them, we aim to integrate voice control with existing, traditional programming languages instead of designing new ones.

There exist guidelines on how to design APIs in general [18] and an overview of design decisions to be made when creating an API [19]. Nevertheless, none of these works take voice-controllability into account.

Hirzel et al. [20] describe an idea of grammars for dialog systems, including virtual voice assistants. However, they do not try to solve the problem of the mapping of sentences to API calls.

Commercial system APIs, such as Google Voice Actions for Android[3] or SiriKit [4] are often limited to certain domains and action types. Furthermore, they require some effort to integrate, such as the creation of configuration files or implementation of non-trivial interfaces.

YAJCo [21] is a parser generator utilizing the similarity between the relations of program elements in Java source files and production rules of computer language grammars. The core ideas behind this article stemmed from YAJCo, however, this time they are applied to natural languages.

V. Conclusion and Future Work

In this paper, we described patterns of mapping between a natural language command and an API method call. These patterns are based on:

- class, method, and parameter names,
- parameter types and positions.

The mapping can be enabled simply by placing the annotation `@VoiceControllable` over a class or a method. When these natural mapping patterns are not sufficient, the programmer can adjust the mapping by using annotations, such as:

- `@StringMapping(Mapper.class)` to specify the string-to-object mapping for a parameter,
- `@ValidValues(ValueSet.class)` to enumerate possible values of a parameter at runtime,
- `@Synonym(of="word1", is="word2")` over packages, classes, methods, and parameters to specify local synonyms,
- `@VoiceCommand("regex (param)")` to specify an exact regular expression whose match will execute the given method.

There are many limitations of the described work. First of all, we did not yet perform full validation of our ideas. We should create a golden standard of sentence-to-API mappings (or utilize an existing one). Then we need to validate our approach by measuring the accuracy of the algorithms based on our ideas when compared to the golden standard. We hypothesize our simple approach based on word similarities would work well for small or medium-sized APIs, but it could be problematic for larger ones.

Next, we described only a small portion of all useful patterns. In the future, we could devise more elaborate ways to express numerical ranges, binary operators, various collection types, exceptions, etc. String-to-object mappers could be improved too. In addition to the manual definition of synonyms, domain ontologies could be used too.

The examples mentioned in this article are very simple – in order to show the point of our approach. We should inspect larger APIs from multiple domains and assess the applicability of our patterns to them.

Finally, in this article, we were interested only in simple, one-sentence inputs being mapped to single API calls. The approach could be extended to support nested API calls or more complex natural language dialogs in the future.

[3]https://developers.google.com/voice-actions/custom-actions

[4]https://developer.apple.com/documentation/sirikit

Acknowledgment

This work was supported by Project VEGA No. 1/0762/19 Interactive pattern-driven language development. This work was also supported by FEI TUKE Grant no. FEI-2018-57 “Representation of object states in a program facilitating its comprehension”.

References

[1] A. Rogowski, “Industrially oriented voice control system,” *Robot. Comput.-Integr. Manuf.*, vol. 28, no. 3, pp. 303–315, Jun. 2012.

[2] T. Gvero and V. Kuncak, “Synthesizing Java expressions from free-form queries,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. ACM, 2015, pp. 416–432.

[3] G. Little and R. C. Miller, “Translating keyword commands into executable code,” in *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’06. ACM, 2006, pp. 135–144.

[4] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, “T2API: Synthesizing API code usage templates from english texts with statistical translation,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 1013–1017.

[5] J. Kollár, E. Pietriková, and S. Chodarev, “Abstraction in programming languages according to domain-specific patterns,” *Acta Electrotechnica et Informatica*, vol. 12, no. 2, pp. 9–15, 2012.

[6] G. A. Miller, “WordNet: A lexical database for English,” *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[7] L. Galko and J. Porubän, “Tools used in ambient user interfaces,” *Acta Electrotechnica et Informatica*, vol. 16, no. 3, pp. 32–40, 2016.

[8] P. Jaccard, “The distribution of the flora in the alpine zone,” *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[9] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ser. PPDP ’10. ACM, 2010, pp. 13–24.

[10] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. ACM, 2016, pp. 345–356.

[11] G. Little and R. C. Miller, “Keyword programming in Java,” in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’07. ACM, 2007, pp. 84–93.

[12] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, “SQLizer: Query synthesis from natural language,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 63:1–63:26, Oct. 2017.

[13] V. Le, S. Gulwani, and Z. Su, “SmartSynth: Synthesizing smartphone automation scripts from natural language,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’13. ACM, 2013, pp. 193–206.

[14] S. Zamanirad, B. Benatallah, M. Chai Barukh, F. Casati, and C. Rodriguez, “Programming bots by synthesizing natural language expressions into API invocations,” in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, pp. 832–837.

[15] M. Landhäußer, S. Weigelt, and W. F. Tichy, “NLCI: A natural language command interpreter,” *Automated Software Engg.*, vol. 24, no. 4, pp. 839–861, Dec. 2017.

[16] O. Pulido-Prieto and U. Juárez-Martínez, “A survey of naturalistic programming technologies,” *ACM Comput. Surv.*, vol. 50, no. 5, pp. 70:1–70:35, Sep. 2017.

[17] R. Knöll, V. Gasiunas, and M. Mezini, “Naturalistic types,” in *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2011. ACM, 2011, pp. 33–48.

[18] J. Bloch, “How to design a good API and why it matters,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’06. ACM, 2006, pp. 506–507.

[19] J. Stylos and B. Myers, "Mapping the space of API design decisions," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '07. IEEE Computer Society, 2007, pp. 50–60.

[20] M. Hirzel, L. Mandel, A. Shinnar, J. Simeon, and M. Vaziri, "I can parse you: Grammars for dialogs," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, ser. LIPIcs, vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 6:1–6:15.

[21] J. Porubän, M. Forgáč, and M. Sabo, "Annotation based parser generator," in *2009 International Multiconference on Computer Science and Information Technology*, Oct. 2009, pp. 707–714.