Research Article

Matúš Sulír* and Jaroslav Porubän

Natural mapping between voice commands and APIs

https://doi.org/10.1515/comp-2020-0125
Received Mar 31, 2020; accepted Jun 10, 2020

Abstract: After a voice control system transforms audio input into a natural language sentence, its main purpose is to map this sentence to a specific action in the API (application programming interface) that should be performed. This mapping is usually specified after the API is already designed. In this paper, we show how an API can be designed with voice control in mind, which makes this mapping natural. The classes, methods, and parameters in the source code are named and typed according to the terms expected in the natural language commands. When this is insufficient, annotations (attribute-oriented programming) are used to define synonyms, string-to-object maps, or other properties. We also describe the mapping process and present a preliminary implementation called VCMapper. In its evaluation on a third-party dataset, it was successfully used to map all the sentences, while a large portion of the mapping was performed using only naming and typing conventions.

Keywords: voice control, natural language, application programming interface (API), classes, methods

1 Introduction

The ability to control applications by voice is becoming more prevalent. From the developer's point of view, the implementation of a voice control system consists of two main tasks: the transformation of a voice signal into its textual form and the selection of an appropriate action to perform [19]. In this paper, we focus solely on the second part, i.e., the mapping of a natural language sentence to a method in the API (application programming interface) which should be executed as a result of this input.

Currently, there exist multiple ways to map a sentence to a method in the API. Manual approaches usually require the combination of configuration files and imperative source code to specify the mapping. Automated approaches utilize probabilistic grammars and heuristics [5], translation of keywords [11], or translation based on machine learning [15].

All of these approaches have one disadvantage – they perceive voice control only as an afterthought. The mapping from voice commands to API calls is performed after the API is already implemented, possibly by other developers. In contrast to them, we will show how an API can be tailored to be voice-controllable by leveraging the similarities between the natural language sentence on one hand; and the names, types and class structure on the other hand. For example, the command "lock the screen after 5 seconds" could lead to a call to `screen.lock(5)` if there is a method `void lockAfter(int seconds)` in the class `Screen`. If such similarities are absent or insufficient, we can fine-tune the mapping using annotations (attribute-oriented programming) over classes, methods, and parameters.

This article presents an extended version of our conference paper [22]. In section 2, we not only review and elaborate the core ideas behind the mapping, but also define many new possibilities including string parameter predicates, expansion of non-primitive parameters into members, command composition, voice response, and dialogs. In section 3, we describe how the mapping patterns can be applied in the implementation of a command-to-API mapping framework. Our implementation, called VCMapper, is available online[1]. In the newly added section 4, we qualitatively evaluate our approach. Finally, we review related work (section 5) and conclude the paper (section 6).

***Corresponding Author: Matúš Sulír:** Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00, Košice, Slovakia; Email: matus.sulir@tuke.sk
Jaroslav Porubän: Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00, Košice, Slovakia; Email: jaroslav.poruban@tuke.sk

1 https://github.com/sulir/vcmapper

2 Mapping patterns

Now we will gradually describe the individual patterns of mapping between natural language sentences and API calls. We will use examples mainly from two domains – the control of hardware devices and smartphone voice assistant commands – but the approach is not limited to them. The language used in these examples is Java. However, the approach is applicable to any object-oriented language supporting reflection and attribute-oriented programming (annotations); static typing is also helpful for some of the patterns.

2.1 Classes and methods

Suppose we want to turn on the light in the room after saying "turn on light". The most primitive and natural way to achieve this is to name the corresponding method `turnOnLight` and annotate it with the marker annotation `VoiceControllable`. Thanks to this, the mapping framework can recognize this method as voice-controllable:

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

The framework would then split the name of the method into individual words with respect to the camel-case convention and transform them to lowercase. The resulting string, "turn on light", exactly matches the input sentence. Therefore, this matching method would be automatically executed.

Calling many static methods of one class is not a good programming practice, of course. Let us say we have separate classes to control lights, speakers, monitors, and other devices. For simplicity, suppose they are named `LightService`, `SpeakerService`, `MonitorService`, etc. In modern application frameworks, it is not necessary to call the constructors of such classes manually. The application framework initializes the objects of these classes for us using techniques such as dependency injection. Therefore, we will assume we have an instance of each necessary class readily available and we will focus on calling appropriate methods on these objects. We can transform the previous example into this code:

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

The `VoiceControllable` annotation is now placed over the class, which means all methods of this class are voice-controllable. Our mapping framework removes the word `Service` from the class name, since it represents an implementation detail irrelevant to the problem domain. After appending the word `Light` to the words from the method name `turnOn`, we get "turn on light", which is equal to the input sentence.

Of course, such exact equality is rare – we should accept some variation in the input sentences.

First, we should consider only stemmed versions of the words during their comparison. This allows us to use various forms of the words – e.g., the plural "turn on the lights" instead of the singular.

Next, the permutations of the words should be allowed. For example, "turn on lights" can be expressed also as "turn lights on".

Finally, extraneous words can be present in the input sentence. This includes, for instance, stop words such as "the" ("turn on the lights") and courtesy phrases ("please turn on the lights"). This should work also vice versa: Some words contained in the method/class name can be omitted from the command – e.g., "lights on".

What variations are allowed in the sentences – and to what degree – is controlled by a specific implementation of the mapping framework. In section 3, we describe an implementation accepting all of these variations to a large degree. Note, however, that alternative implementations are possible.

2.2 Parameters

Now we will describe the mapping patterns for methods with parameters.

2.2.1 Primitive and enumeration types

Suppose we would like to set the volume level using the command such as "set volume level to 80". This corresponds to the code:

```java
@VoiceControllable
public class VolumeService {
    public void setLevel(int percent) { ... }
}
```

Suppose this method was selected as the only one matching the sentence. Then we can surely declare the number "80" is the value of the parameter `percent`, since it is the only number in the input sentence – and the parameter `percent` is mandatory. The system thus executes the method `setLevel` with the argument value of 80.

A similar logic can be applied to floating-point numbers. If we would like to recognize also the command "set volume level to 81.5", changing the parameter type to `double` is sufficient:

```
public void setLevel(double percent) { ... }
```

An enumeration-typed parameter is another straightforward case. For example, a speaker (sound channel) can be defined as:

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

In the class `VolumeService`, we can then have a method to turn on the given channel:

```
public void turnOn(Speaker speaker) { ... }
```

In the sentence such as "turn on the left speaker", we easily can match the word "left" with the enumeration constant `LEFT`.

2.2.2 Strings

Often we need to select one of multiple possible options, but a complete list of these options is not available at compile time. Suppose we need to open the given application in the smartphone after saying a sentence such as "open Gmail". We create a method in the class `ApplicationService`:

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

To enumerate all possible applications at runtime, we first create a class implementing the method which returns a set of these values:

```
public class AppNames implements StringEnumerator {
    public Set<String> getValidValues() {
        return Apps.getAll(); // {"Gmail", "Twitter",...}
    }
}
```

Then we annotate the parameter `name` of the method `open` with this class (using the refers-to-element idiom [16] on the class since annotations in Java cannot refer directly to methods):

```
public void open(
    @StringEnumeration(AppNames.class) String name
) { ... }
```

Thanks to this, our framework can easily determine which part of an input sentence corresponds to the value of the parameter. Additionally, it helps the system to match the sentence with this method since the sentence contains "Gmail" and this method accepts such a value as a parameter.

Sometimes it is impractical to enumerate all possible values. Instead, we can ask whether a given string is a possible value of the given parameter (i.e., to specify a precondition of the argument):

```
public class AppNames implements StringTester {
    public boolean isValid(String value) {
        return Apps.contain(value);
    }
}

@VoiceControllable
public class ApplicationService {
    public void open(
        @StringTesting(AppNames.class) String name
    ) { ... }
}
```

Preconditions about parameter values are useful mainly in situations when the set of possible values is too long or even infinite. The drawback is that instead of searching the input sentence for a given term, we should execute the precondition on all n-grams of the input sequence to find whether there is a match. Performing this on all methods with a `@StringTesting`-annotated parameter can be time-intensive in case of a large number of such methods.

There exist cases when all possible values cannot be enumerated at all. Then we should assume the sentence starts with the given key words and continues with the value of the `String` parameter until the end. For example, the input "add note: buy coffee and spinach" can be mapped to the following method in the class `NoteService`:

```
public void add(String note) { ... }
```

The words "add" (the method name) and "note" (from the class name) are used to find the corresponding method. The rest of the sentence is mapped to the value of the `String` parameter.

If the programmer encounters problems during the design of an API with non-enumerable string values, there is always a fallback mechanism available: to exactly specify a regular expression how the sentence should look (see section 2.5 for more details).

2.2.3 Collections

Suppose we would like to turn off multiple speakers at once. In the sentence "turn off the left and right speaker", the part “left and right” is mapped to the value of the collection-typed parameter, such as a list or a set:

```
public void turnOff(Set<Speaker> speakers) { ... }
```

The special keyword “all” can be used to denote all possible values of an enumeration type, e.g., “turn off all speakers”.

In the case of integers, we can use also ranges, e.g., “turn off speakers 3 to 7”. Again, we declare the given parameter as a collection type:

```
public void turnOff(Set<Integer> speakers) { ... }
```

2.2.4 Classes composed of members

Sometimes classes consist of member variables that are distinguishable in voice commands. For example, this simple date consists of a month (an enumeration) and a day:

```
public class Date {
    private Month month;
    private int day;

    public Month getMonth() { ... }
    ...
}
```

Suppose we would like to note the date of our birthday by saying, e.g., “set my birthday to January 1”. This sentence is mapped to the following voice-controllable method, which has one parameter:

```
public void setMyBirthday(Date date);
```

In the voice control framework, such a parameter is expanded to the member variables of the given class – i.e., in this case, the method behaves as `setMyBirthday(Month month, int day)`.

2.2.5 Other classes

Finally, let us consider parameters of any other class. For example, the class `Color` has three member variables along with the corresponding getters and setters:

```
public class Color {
    private int red, green, blue;
    public int getRed() { ... }
    ...
}
```

This class is used as a parameter of the method setting a solid wallpaper on a phone:

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

When we use the color in a command, such as “set wallpaper to yellow color”, we do not explicitly mention any members of the class (the values of red, green, or blue). Therefore, we need a special form of mapping from the string representation to the object instance. One option is to define a constructor taking a string – or a static factory method such as `fromString(String string)` which would create the given object. However, this does not solve the problem of the enumeration of all possible values. This can be solved by specifying the string-to-object map:

```
public class Color implements Mapper<Color> {
    private int red, green, blue;
    private int getRed();
    ...

    public Map<String, Color> getMap() {
        return Map.of(
            "black", new Color(0, 0, 0),
            "yellow", new Color(255, 255, 0),
            ...
        );
    }
}
```

We must sometimes deal with third-party classes that cannot be modified. In such cases, we can specify the mapping in a separate class and annotate the parameter with this created class:

```
public class ColorMapper implements Mapper<Color> {
    public Map<String, Color> getMap() { ... }
}

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

Instead of specifying the whole map as a run-time constant, we can implement the `Converter` interface by specifying a method that dynamically tests whether the given term is acceptable as a value of the parameter (i.e., if the precondition for the argument holds) and a method that eventually returns the converted value. For instance, a rational number can be recognized either as the word “half” or as a pair consisting of a cardinal and ordinal number (“one third”).

```
public class Rational implements Converter {
    public Rational(int numerator, int denominator) {...}
    ...

    public boolean isValid(String term) {
        return representsFraction(term);
    }

    public Rational convert(String term) {
        return new Rational(..., ...);
    }
}
```

Voice-controlled methods can then have parameters of type `Rational` to represent, in combination with other parameters, terms such as “half a minute”.

2.2.6 Handling multiple parameters

Methods with multiple parameters do not pose any problems at all, as long as the sets of the possible values of each parameter are disjoint. Consider the modified screen-locking example from the introduction:

```
public void lockAfter(int number, TimeUnit unit) { ... }

enum TimeUnit {
    SECONDS, MINUTES, HOURS
}
```

For the sentence “lock the screen after 30 seconds”, the number “30” is passed as a value of the first parameter. The word “seconds” is mapped to the enumeration constant `TimeUnit.SECONDS` and passed as a value of the parameter `unit`.

Things start to complicate a bit when the possible values of the parameters are not disjoint. For example, suppose we would like to set the volume of a speaker, where the speakers are labeled by numbers:

```
public void setVolume(int speaker, int percent) { ... }
```

Here we have two possibilities. First, the parameters can be matched by their position in the sentence: For instance, in the sentence “set speaker 1 volume to 80”, the first parameter is “1” and the second is “80”.

The second possibility is matching by parameter names. For example, in the sentence “set volume to 80 percent for speaker 1”, the two numbers (80, 1) are matched by their proximity to the parameter names (`percent`, `speaker`) in the sentence.

2.3 Synonyms

Instead of using the exact words contained in the identifier names, the users often utter their synonyms. Consider the method `open(String name)` in the class `ApplicationService`. A user would like to say “run WhatsApp” instead of “open WhatsApp”. Note that “run” is not a synonym of the word “open” in general – it is limited only to this particular case. Therefore, our framework supports the notion of a method-local synonym, denoted by the annotation `Synonym` over a method:

```
@Synonym(of="open", is="run")
public void open(String name);
```

Such a synonym is applied only during the matching process of this particular method. In a similar way, we can support package-local synonyms (by annotating the package in the special file `package-info.java`), class-local and parameter-local synonyms. Multiple synonyms per element can be specified too:

```
@Synonym(of="application", is="app")
@Synonym(of="open", is="start")
public class ApplicationService { ... }
```

If the number of synonyms was too high or a standard set of general synonyms was appropriate, we would suggest loading the set of synonym pairs from an external file, such as a dictionary. This could be accomplished with an annotation such as `@SynonymFile("general.csv")`.

2.4 Part of speech recognition

Although allowing permutations in input sentences is useful, sometimes it can cause ambiguity. For example, suppose we have these two voice-controllable classes:

```
class MonitorService {
    public void switch_() { ... }
}

class SwitchService {
    public void monitor() { ... }
}
```

The sentences “switch the monitor” (change the current monitor to another one) and “monitor the switch” (start monitoring the network switch) are ambiguous with respect to the given code. Therefore, we need to tag the words in the input sentence using a part-of-speech tagger.

The words in the source code identifiers are tagged too. According to usual code conventions, the class name should represent the object being manipulated, and the method represents the action performed with this object. Thus, by default, the words in the class names are tagged as nouns, and the words in the method names are tagged as verbs. However, the programmer can override this by using the annotations `@Noun("word")` and `@Verb("word")`.

The part-of-speech tags both in the sentence and the methods can be then used for potentially unambiguous matching. For instance, "monitor the switch" will be mapped to `SwitchService::monitor`.

2.5 Fallback

There are situations when we encounter difficulties with the application of the mapping patterns. For example, we would like to say "play Four Seasons by Vivaldi". In this sentence, there are two strings – the song name and the artist – which are very difficult to enumerate (they can contain almost any value). Furthermore, the name of the song can contain words from other method names/commands: "turn on the lights" vs. "play Turn on the lights". Therefore, we can annotate a method with a regular expression denoting how exactly should a sentence look.

```
@VoiceCommand("play (.*) by (.*)")
public void play(String song, String artist)
```

The specific values of the groups (parentheses) in the regular expression will be supplied as values of the parameters during the execution of the method.

2.6 Composition

Except for the simple actions, the user may sometimes want to use compound commands, such as "turn on the lights and close the door" or "if wifi is available, disable mobile data". The implementation of such commands consists of two steps. First, using the fallback regular expression as previously described, we annotate a method with the generic syntax of the compound command. Then we specify how the compound command should be executed in the body of the method.

```
@VoiceCommand("(.*) and (.*)")
public void and(Runnable action1, Runnable action2) {
    action1.run();
    action2.run();
}
```

In the example above, two individual commands separated by "and" are matched – i.e., the corresponding voice-controllable methods are found. An object which is an instance of `Runnable` (a functional interface used to implement lambda functions in Java) is created for each of the matched methods. The body of the method is then executed, which means the matched methods are run sequentially in this case.

Another example is the "if-then" compound command, which can be implemented as follows.

```
@VoiceCommand("if (.*) then (.*)")
public void ifThen(BooleanSupplier condition,
        Runnable action) {
    if (condition.getAsBoolean())
        action.run();
}
```

Since the first parameter is a `BooleanSupplier`, a method returning `boolean` must match the first simple command (the part between "if" and "then"). It is executed and if it returns true, the second simple command is performed.

2.7 Response

So far we considered mainly commands which perform some external, visible actions such as control a hardware device or run an application. However, the purpose of many voice commands is to provide a response to a question. For example, after saying "what time is it?", we would like to hear a response like "it is 12:34" from the voice assistant. The most natural way to accomplish this is to create a method returning a `String`:

```
@VoiceControllable
public class TimeService {
    public String whatTimeIsIt() {
        return "It is " + ...;
    }
}
```

The voice command system then reads this string response as-is to the user. If the creation of such a method is not desirable, we can return an object of any type (e.g., `Time`) and the system will read its string representation. By default, we can use the standard Java method `toString` which is available for all objects. In cases when this representation is not suitable for direct communication with humans, we can implement the similarly working method `toResponse` from the `VoiceResponse` interface. When we cannot modify the given class, it is possible to annotate the method returning the object with, e.g., `@ResponseConversion(TimeResponse.class)` and implement the conversion separately:

```
public class TimeResponse
        implements ResponseConverter<Time> {
    public String toResponse(Time time) {
        return "It is" + ...;
    }
}
```

When a user's request cannot be processed properly, the executed method may throw an exception. For example, if we request to open a window in a home automation system and the window is already open, the method may throw `IllegalStateException`. By default, the system will read the human-readable message of the exception, obtained by calling `getMessage()` on the exception object. If this is not desirable, we can specify the message using the annotation `OnException`:

```
@OnException(of = IllegalStateException.class,
             say = "The window is already open.")
public void open() {
    ...
}
```

2.8 Dialog

The patterns presented so far represented one input sentence paired with one final response – the execution of an action and/or a voice answer. Sometimes the user would like to communicate with the system using a dialog, though. One of the frequently used kinds of dialog is the so-called form-filling flow pattern [6]. Here the user asks the system to perform something, supplying only a minimum amount of information. Then the system gradually asks the user to fill in all the missing data. We will illustrate this with a very simple example where the user would like to send a message. Suppose there is the method `send` in the voice-controllable class `MessageService` (the enumeration of possible values of `contact` is omitted for brevity):

```
public void send(String contact, String text) { ... }
```

The user says “send a message to Joe”. This sentence contains all necessary information except the text itself. Thus the system asks the user “What is the text?”; the user then responds by dictating the text of the message.

By default, the system generates the questions using the names of the parameters – e.g., “text” in our example. If there is a better alternative, we can manually specify the question for the given parameter:

```
public void send(@IfAbsent("To whom?")
                 String contact,
                 @IfAbsent("Please tell me the message.")
                 String text) { ... }
```

3 Mapping process

In this section, we will describe how the mentioned mapping patterns can be integrated to form a pattern-based voice control framework. We will discuss the process of the transformation of an input sentence to the corresponding action.

First, the input sentence is matched against all regular expressions specified using the fallback mechanism (`@VoiceCommand("regex")`). If a match is found and the parameters of the method are string-typed (or none), this method is executed and the process stops here. If the parameters are functional interfaces (practically, lambda functions), this is a compound command – individual subcommands are extracted and recursively processed using the usual rules.

Then the type-based matching of parameters is performed. The premise is that, for example, if a method has an integer parameter and an enum-typed parameter, the sentence must contain an integer and a word matching one of the enumeration constants, otherwise it does not match (unless we want to initiate a dialog, which is deferred to a later phase). During this phase, we consider not only the types themselves, but also preconditions specified using annotations, string-to-object mappings, and other mechanisms described in section 2.2. The results of this phase is a list of potentially matching methods, along with the mapping of the terms in the sentence to the corresponding parameters.

For every method in this list, a score of similarity with the sentence is computed: Let W_M be the set of words contained in the class and method name. Let W_S be the set of words in the input sentence, excluding the parameter values matched in the previous step. The score is computed as the Jaccard index [7] of these two sets:

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

In cases when the classes or methods are annotated with synonyms, multiple variants of the set W_M are constructed, containing words from the methods and classes gradually replaced with their specified synonyms. The resulting score is the maximum of the scores calculated for individual variants.

Methods with a score lower than a certain threshold (e.g., 0.2) are excluded from the set, since they represent poor matches. This threshold limits the degree to which the user can omit or add words to the sentence so that it differs from the words used in the source code.

Finally, a method with the highest score is selected. Ideally, there is exactly one method with the highest score. Such a method is executed, and if it has a non-void return value or throws an exception, a voice response is also provided.

If there are multiple methods with the same highest score, the input sentence is tagged using a part-of-speech tagger. Words that are present in all matching methods are tagged according to the occurrence types (nouns in class names, verbs in method names) or manually defined annotations. Matches with at least one word tagged differently in the sentence and the identifiers are excluded. Then if only one method is left, it is executed. Otherwise, the system should tell the user the available options and ask him to select one of them.

If there is no matching method, the process is repeated from the type-based matching step, but this time without strictly requiring the sentence to contain all parameter values. Thanks to this, it is possible to identify the matching methods even if some information is missing and initiate the dialog to fill in these data.

An implementation of the majority of parts of the described process, along with numerous examples and tests, is available at https://github.com/sulir/vcmapper.

4 Evaluation

Since we do not map sentences to existing APIs but manually create a new API, it would not be appropriate to perform a fully quantitative evaluation of the approach using metrics such as precision and recall. Instead, we decided to perform a predominantly qualitative evaluation of its strengths, weaknesses, and expressiveness. We used our implementation of the framework, which is called VCMapper[2].

Our task was to design a voice-controllable API in the smart home domain. The API should support commands from The Fluent Speech Commands dataset [13] by Fluent.ai[3]. Among other data, this dataset contains English sentences, such as "Put on the music". Each of these commands is mapped to an intent, e.g., {action: "activate", object: "music", location: "none"}. We created a subset of the dataset by randomly selecting 50 sentences, which were associated with 22 unique intents.

One of the researchers was reading these sentences and trying to design an API in a most straightforward way which would enable VCMapper to successfully map the sentences to the API methods. At the same time, he was writing mock-based tests to assess the correctness of matching. In each test, VCMapper was asked to recognize a sentence, and the mocking framework was asked to verify whether the correct method was called (and with the given arguments).

Each time there was a problem, the researcher noted it and remedied it if possible. This allowed us to answer the following research questions:

- **RQ1**: What are the strengths, weaknesses, and problems of our natural command-to-API mapping?
- **RQ2**: To what degree can our approach express the mapping of basic commands, such as the ones in the smart home domain?

4.1 Strengths and weaknesses

To answer **RQ1**, the main advantage of our approach is the ease and speed of the mapping definition. Except for a few problematic situations, which we will describe later, adding a new API method or the recognition support for a new sentence was often a matter of seconds.

Since we did not use a general synonymic dictionary, such as WordNet [14], we had to manually specify synonyms when we wanted to recognize multiple variants of the same command. This was probably the most prominent weakness we encountered. In many cases, however, it was not necessary to add synonyms, since the mapping process used in VCMapper is robust with regard to small changes in the sentences.

During our task, we encountered a few small problems, some of which were already resolved. First, we needed to support both "bathroom" and "washroom" as the name of the room which was specified as an `enum` constant. We could add the `@Synonym` annotation over all parameters of type `Room`, but this would cause unnecessary code duplication. We decided to add support for the `@Synonym` annotation over the enumeration type definition itself.

Next, the annotation `@Synonym(of="...", is="...")` caused confusion about its directionality: Does the word in the parameter "`of`" represent a word in the sentence and "`is`" in the identifier names or vice versa? We decided to consider both directions when performing the matching using synonyms.

Some words have synonyms consisting of multiple words, e.g., "decrease" and "turn down". Therefore, we needed to implement the multi-word synonym feature.

Multiple times, we encountered a situation when an intent (action) was associated with a sentence that could be naturally transformed to a method name, but also with another, completely different sentence. For instance, both "Decrease temperature" and "Make it cooler" had to be mapped to `decreaseTemperature()`. This was resolved either by using the fallback mechanism (`@VoiceCommand("regex")`)

2 https://github.com/sulir/vcmapper
3 https://www.fluent.ai/research/fluent-speech-commands/

or by specifying synonyms which are not synonyms in the strict sense (e.g., "decrease" and "cooler").

A few methods would benefit from some kind of intermediate option between the fuzzy matching based on identifier names and the strict, fully manual regular expression fallback. In these cases, we usually opted for the regular expression, such as "`.*\btoo loud\b.*`" matching "That's too loud" and similar utterances.

Some sentences contained irrelevant or even confusing parts, such as the first part of the command "It's too loud, turn the volume down". A mechanism to remove such parts could be helpful, but it was not necessary and these sentences were successfully recognized.

Some regular expressions in the `VoiceCommand` annotation contained groups for purely syntactical reasons, not to denote potential parameter values, e.g., "`.*\b(too quiet|can't hear)\b.*`". We ignored superfluous groups, but named groups (after parameter names) would represent a better long-term solution.

4.2 Expressiveness

Answering **RQ2**, we were able to successfully express the mapping from all 50 natural language sentences in our dataset to the corresponding API methods. This means that in the end, all 50 tests passed.

Now we will inspect how often the individual features of our mapping approach were utilized. In the ideal scenario, only mapping based on identifier names would be used, since it requires minimum effort from the programmer. Less ideal, but still relatively natural mapping features are the annotations of synonyms, possible valid values, etc. Fallback annotations (regular expressions) are the least desirable ones.

To measure this, we first disabled all annotations except `@VoiceControllable` (purely name-based mapping) and determined the proportion of passing tests. Then we enabled also synonyms (`@Synonym`), the enumeration of valid values for string arguments (`@StringEnumeration`), and finally also the definitions of commands via regular expressions (`@VoiceCommand`). After enabling each additional annotation type, we measured the portion of the passing tests.

The results are in Table 1. We can see the mapping for a relatively large portion of commands (40%) was expressed in the most natural way, i.e., only by conveniently naming the classes, methods, and enumeration constants. Since multiple different utterances were often mapped to the same methods, the `Synonym` annotation was used a lot.

Table 1: Portions of tests passed with the given annotations enabled

Enabled annotations	Tests passing
only @VoiceControllable	40%
also @Synonym	92%
also @StringEnumeration	94%
also @VoiceCommand	100%

The mapping based on regular expressions was utilized sporadically.

Of course, this small-scale study has its threats to validity. First, the used dataset is limited to one domain, and the structure of the intents is relatively simple. Some more advanced features of our mapping approach were therefore not used in this evaluation. Second, the API was designed by one person, which could lead to the subjectivity of the results.

5 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 [4]. Desai *et al.* [2] synthesize source code written in various domain-specific languages, thanks to a dataset of sentence–code pairs used for training. Gvero and Kuncak [5] synthesize Java expressions using probabilistic grammars and heuristics. In T2API, Nguyen *et al.* [15] perceive program synthesis as a statistical translation process from natural language to a programming language. Little and Miller [12] generate Java code from a set of brief keywords. Some synthesis approaches are focused on particular domains or technologies, e.g., SQL query generation [23], smartphone automation script synthesis [10], bot API invocations [24]. 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.* [9] 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 [11] 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 [18] is a paradigm aiming to make the source code look more like natural language. For example, Knöll *et al.* [8] 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.

The idea of the specification of possible argument values resembles the idea of refinement types [3]. Both their approach and ours narrow the set of acceptable values of types, e.g., from a general string to a string meeting certain requirements. While the main purpose of refinement types is to catch more errors at compile time, our approach aims to ease the matching of a voice command at runtime.

There exist guidelines on how to design APIs in general [1] and an overview of design decisions to be made when creating an API [20]. None of these works take voice-controllability into account.

YAJCo [17] 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.

While our approach raises the level of abstraction during the mapping specification, adding specific examples of input sentences matching the given method could help the programmer. Visual augmentation of a source code editor [21] would be a suitable approach to display these examples alongside the source code.

Commercial system APIs, such as Google Assistant integration[4] or SiriKit [5] are often limited to certain domains and action types. Furthermore, they require considerable effort to integrate, such as the creation of configuration files or implementation of non-trivial interfaces.

Finally, Hirzel *et al.* [6] 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.

4 https://developers.google.com/assistant

5 https://developer.apple.com/documentation/sirikit

6 Conclusion

In this paper, which extends our previous work [22], we described a natural way to map natural language sentences (voice commands) to API calls. In ideal cases, when the classes/methods are conveniently named and the parameters appropriately typed, using the marker annotation `@VoiceControllable` is sufficient. Otherwise, the programmer has a variety of alternative mechanisms available – from synonym definitions to object–string mappings and recognition based on regular expressions. The definition of compound commands is possible too.

We not only defined how a single command can cause the execution of an action, but also briefly outlined how the system can respond with another sentence or communicate with the user via a dialog.

Many of the presented ideas were implemented in a framework called VCMapper. Note, however, that the described mapping process is not the only one possible – we could use the patterns (conventions and annotations) in a framework based on completely different algorithms.

During the evaluation, we were able to successfully map 50 sentences from a third-party dataset to their corresponding API calls. Almost half of them were mapped using only naming conventions, the rest relied mainly on the specification of synonyms.

In the future, we could perform a more thorough evaluation with a larger and more diverse dataset. We can also expand the idea of dialog communication with more advanced patterns.

Acknowledgement: This work was supported by Project VEGA No. 1/0762/19 Interactive pattern-driven language development.

References

[1] Bloch J. 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*, OOPSLA ’06, pages 506–507. ACM, 2006.

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

[3] Freeman T. and Pfenning F. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI ’91, page 268–277, New York, NY, USA, 1991. Association for Computing Machinery.

[4] Gulwani S. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles*

and Practice of Declarative Programming, PPDP '10, pages 13–24. ACM, 2010.

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

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

[7] Jaccard P. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, 1912.

[8] Knöll R., Gasiunas V., and Mezini M. Naturalistic types. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 33–48. ACM, 2011.

[9] Landhäußer M., Weigelt S., and Tichy W. F. NLCI: A natural language command interpreter. *Automated Software Engineering*, 24(4):839–861, Dec. 2017.

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

[11] Little G. and Miller R. C. Translating keyword commands into executable code. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 135–144. ACM, 2006.

[12] Little G. and Miller R. C. Keyword programming in Java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 84–93. ACM, 2007.

[13] Lugosch L., Ravanelli M., Ignoto P., Tomar V. S., and Bengio Y. Speech model pre-training for end-to-end spoken language understanding. In *Proceedings of Interspeech 2019*, pages 814–818. ISCA, 2019.

[14] Miller G. A. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, Nov. 1995.

[15] Nguyen T., Rigby P. C., Nguyen A. T., Karanfil M., and Nguyen T. N. 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*, FSE 2016, pages 1013–1017. ACM, 2016.

[16] Nosáľ M., Sulír M., and Juhár J. Source code annotations as formal languages. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 953–964, 2015.

[17] Porubän J., Forgáč M., and Sabo M. Annotation based parser generator. In *2009 International Multiconference on Computer Science and Information Technology*, pages 707–714, Oct. 2009.

[18] Pulido-Prieto O. and Juárez-Martínez U. A survey of naturalistic programming technologies. *ACM Computing Surveys*, 50(5):70:1–70:35, Sept. 2017.

[19] Rogowski A. Industrially oriented voice control system. *Robotics and Computer-Integrated Manufacturing*, 28(3):303–315, June 2012.

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

[21] Sulír M., Bačíková M., Chodarev S., and Porubän J. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing*, 49:46–59, Dec. 2018.

[22] Sulír M. and Porubän J. Designing voice-controllable APIs. In *Proceedings of the IEEE 15th International Scientific Conference on Informatics*, pages 393–398. IEEE, 2019.

[23] Yaghmazadeh N., Wang Y., Dillig I., and Dillig T. SQLizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63:1–63:26, Oct. 2017.

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