Hasso Plattner Institute for Digital Engineering

Bachelor's Thesis

Supporting Iterative Development of Voice Interfaces using a Domain Specific Language

Unterstützen von Iterativer Entwicklung von Sprachassistenten durch einen Domänenspezifische Sprache

Arne Zerndt

arne.zerndt@student.hpi.de

Supervised by Prof. Dr. Holger Giese
System Analysis and Modeling Group

Potsdam, June 5, 2019

Contents

1 | Introduction

REWRITE BELOW TEXT

In this bachelor thesis, I plan to provide a way to save a voice interface in Git alongside the code it corresponds to. To approach this goal, I will attempt to create a Domain Specific Language (DSL) to describe the configuration of a voice interface in a text-based but still abstract and intuitive way.
I will focus on development using Google Dialogflow as it is by far the more widely used tool it is also what my bachelor team is using in our project.

I propose that by designing a DSL that can be used to configure a Dialogflow agent in a way that is text based and can be managed by version control systems like Git - but that still feels seamless to a user and is intuitive to use for a developer familiar with the Dialogflow web interface - development on voice interfaces can be simplified and development teams in the future will be able to work on voice assistants in the same way that they are used to working with code. This opens up the entirety of code based tools which exist for making development more streamlined for the development of voice interfaces, while keeping the robustness of code which can be saved in version control.

I will initially attempt to use Xtext to create a grammar and code generator for configuring Google Dialogflow agents. The grammar will be used to create a plugin for eclipse and the language server protocol that provides syntax highlighting and validation. The output of the code generator should be a valid voice interface configuration in its JSON representation, ready to be imported into Google Dialogflow.

The DSL I am planning to create will need to compete with two other solutions.
Firstly, there is the web interface that is the common way of editing an agent. This solution is very easy to get started with and lets anyone, even someone with no development experience, configure a voice interface. Secondly, there is the JSON representation of the agent, that can - despite this not being intended - be directly edited. For the DSL to provide a tangible benefit compared to both of these solutions it has to reach the following criteria:

- The DSL has to be easy to use for a developer who is used to the Dialogflow web interface.
- The DSL has to be saved as a text representation (code) that can be saved in source control alongside application code, but that is more readable and less cluttered than the JSON representation.

There are additional goals that are worth trying for but not strictly mandatory:

- Object oriented inheritance - specifically for "intents" and "entities" - would allow significant benefits over configuring the agent on the web interface through reducing duplication.

- IDE support for the DSL would provide developers with automatic checking for the validity of their agent configuration.

I am planning to validate the goals mentioned above through a user study - using the cognitive walkthrough method - in which I aim to have a group of developers use my solution to edit an existing Dialogflow configuration and collect feedback on the above criteria.
A specific risk that I have to mitigate is the potential impact of changes Google could make to their V2 API which is currently under development or to the format of their JSON configuration.

2 | Status Quo

2.1 Voice Interfaces

Voice assistants and voice interfaces are increasingly popular technologies that are experimented with and used by every major player in the technology market [citation needed]. To keep up with this trend, developers either need to be able to build their own voice interfaces or integrate with an existing system of which Google Assistant and Amazon Alexa are the most relevant. [citation needed]Currently, depending on whether you are developing for Google Assistant or Alexa, designing a voice interface for one of these systems usually entails working with either Google Dialogflow or Amazon Lex. These are powerful tools which enable developers or domain specialists to quickly and easily design a voice interface. These tools are interacted with through a website which provides a graphical editor for the configuration of the voice interface. Although this makes the initial development of a voice interface easier, it comes with a severe drawback.

2.2 Version Control

While the web interface makes initial setup of the voice interface easier for a single domain expert or developer, new difficulties arise when a team of developers is working on a voice interface in an iterative fashion. It becomes crucial for the team to manage versions and track changes to the interface along with the changes to the application the interface is supporting. [citation needed]

The state of the art solution for managing iterative work in development teams is Git. [citation needed]Git provides a functionality for saving snapshots of specific iteration in your project and handling the problems that come up in iterative work. These problems are: • merging work done by multiple users • ensuring that a stable version of a project is saved while developers are working on more experimental changes • giving the team the ability to easily track and revert changes that have been made. The issue that arises is that the above mentioned technologies are not compatible and the configuration of a voice interface can easily get out of sync with the changes made to the application and managed in Git. [citation needed]

2.3 Previous Work

Previously, similar issues have been solved for other technologies [citation needed](cite this, evtl the Evolution control paper or with smt. from here `https://scholar.google.de/scholar?`

hl=de&as_sdt=0%2C5&q=version+control+xml&btnG=) by designing Version control systems that specialize in managing the artifacts those technologies rely on.

2.4 Lack of Solution

A solution like this has not been built for Dialogflow, most likely because voice interface tools like Amazon Lex or Google Dialogflow are a newer development, with Dialogflow having only existed in its current fashion since October 2017 [citation needed]and the Dialogflow V2 API that I am using, and that was necessary for the success of this project is currently in Beta stage. The V2 API is generally available since 17 April 2018 (https://blog.dialogflow.com/post/v2-and-enterprise-edition-generally-available/ 4.6.2019) And "V1 of Dialogflow´s API will be deprecated on October 23, 2019". (https://dialogflow.com/docs/reference/v1-v2-migration-guide , 4.6.2019). Dialogflow is currently still transitioning tot he use oft he new V2 API. It is the V2 API that makes this project possible because it includes the agent management APIs used for exporting, importing, and updating of Dialogflow "agents" (this is what Google calls a specific voice interface) using the JSON format.

3 | Problem Statement

As stated before, currently there is no version control system for Dialogflow agents. This is problematic because when designing a voice interface, it is necessary to make iterative changes, so the voice interface can evolve alongside the application it supports. This can lead to issues in the following workflows that are common in software development: (Reword this to prepare for ALL points below)

- When starting work on a new feature a new branch is created. This is done so that changes, which might not work right away, are contained to this branch and can be merged into the main application at a later time. If this feature requires changes to the voice interface, a problem may arise, because the voice interface has no mechanism for branching.

- Once experimental changes on a feature branch advance to a point where they are meant to be included in the master branch, code can simply be merged from one branch to another. The changes to a voice interface configuration cannot be included in a pull request.

- When working on a product it is generally considered to be "best practice" to have code that should be merged into the master branch from a feature branch reviewed by at least one or two other developers to make sure that it is working as expected and does not have any obvious flaws. This cannot be done for changes to a voice interface.

- When working in a team, it is not always possible to remain aware of all the changes team members have made. Source code that is managed in Git automatically creates a history of all changes, which is highly valuable to the developers. This type of history does not exist for the voice interface configuration.

- Open source development is an important part of today's development landscape. Since voice interfaces are not usually checked into source control alongside application code, open source development of voice interfaces or applications that use voice interfaces is stifled. In addition, open source projects can be forked by other developers and can be improved by many developers in an iterative fashion. This is another advantage that voice interfaces are currently lacking.
  When working on a Dialogflow agent, one can save a version of the configuration and then continue as a draft, but Dialogflow assumes that you will only ever have one draft at a time. Versions are also designed in a linear fashion with no way to merge changes from multiple versions. This is obviously nowhere near the depth of features that are necessary in version control and all of these features are already provided for normal code by using the current state of the art version control system, Git.

In summary the problem is that there is no sufficient version control system that is compatible with Dialogflow.

4 | Approach

4.1 Choosing a Direction

4.1.1 Version Control System

Since there is no version control system that is compatible with Dialogflow the obvious solution might be to build a new version control system that is compatible with Dialogflow. This has been done before for other technologies (cite this, evtl the Evolution control paper or with smt. from here `https://scholar.google.de/scholar?hl=de&as_sdt=0%2C5&q=version+control+xml&btnG=`) but I decided against this approach for multiple reasons.

Firstly, I believe that it will be hard to get developers to move from an established tool like Git; I assume that developers would not choose to give up feature rich support that exists for Git (Github, Gitlab, Bitbucket to only name a few). Secondly, using a specific version control system that is developed to allow compatibility with Dialogflow, it would be hard to also maintain compatibility with other tools.

Therefore, instead of trying and failing to develop a competing standard to Git, I decided to make use of a workaround that is possible with Dialogflow.

4.1.2 Git with exported JSONS

It is possible to export a folder with a JSON representation of an agent and save this in Git. This solves some of the issues mentioned in chapter X (...), but an agent's JSON representation is not intended for readability by humans, and neither is it meant for direct editing. This makes certain aspects of the workflows I described above much harder, e.g. conducting a code review, since it is difficult to read the changes to the JSON files describing the agent's configuration.

4.1.3 Domain Specific Language

While trying to solve the problems mentioned so far, it became obvious that a solution to these problems would require configuring an agent in a format that is compatible with text based tools like Git but that also maintains or even enhances upon the maintainability and readability of Google Dialogflow configuration in the web interface. For this purpose, I designed a domain specific language (DSL) in order to create a text-based notation (DSL code) for the configuration of an agent.

4.1.4 Requirenments for the DSL

This DSL must fulfill a number of requirements: • It needs to be significantly shorter than the JSON representation of an agent. • It needs to be more readable than the JSON representation. • It needs to create smaller diffs (sum of changed lines) than the JSON representation when making changes. • A user needs to be able to automatically compile and update an agent on the web from DSL code.

During the process of my work it became clear that solving the above mentioned problems made it necessary to find solutions for every one of these requirements.

4.2 DSL Engineering

What I built is a so called Dom. Spec. Lang.. It consists of 2 main parts. Firstly, there´s a grammar which defines what you can write in the DSL. The grammar contains all possible combinations of words that you can use to write in this language

4.2.1 Grammar

Design Decisions

I wanted the DSL code to read naturally for a person used to the Df interface.

I tried to – in all the language design – mimic the interface on the Df website. E.g.: when entering an intent, the way it starts is: (Sample code here with highlighting)

1. key word “intent” 2. the name of the intent – 3. option to enter contexts and/or the option to enter events 4. Enter parameters 5. Enter training phrases 6. actions 7. enter responses 8. fulfillment settings

The cool things is: On the Df website when creating a new intent, the 1st thing you do is to klick “create intent”. Then you enter an intent name, and are presented with a page where you can enter in this exact order: contexts, events, training phrases, actions and parameters, responses, and fulfillment settings; As you can see, the DSL closely mirrors the web interface.

There is one exception: the parameters are entered before training phrases to allow for validation of training phrases that use these parameters.

Another example: Entity type : (Sample code here with highlighting)

In Df you create entities which represent a type for which you supply all possible values. There are numerous built-in types like dates or numbers. In building an agent, oftentimes you will want to use custom types and in the DSL this is possible through the following: - key word “Type” - name of the type Next it is possible to enter either the keyword “dynamic” or the keyword “values” followed by a comma separated list of values that are possible for this type. - additional settings

This allows the user to either enter a type with it’s associated values, or have the type exist in the DSL and be valid for validation - but not have any JSON files generated for it, so as not to overwrite an entity type that is dynamically set on Df via the API. This is relevant in cases like the team selection in our Ask your Repository project. In that case a list of teams

was kept as an entity type, and dynamically updated whenever new teams were created. This allowed the user to say "Select Team" followed by their team's name and have the voice interface understand their team name.

Specifying this grammar allows me to have live IDE support including syntax validation while writing DSL code. Which Markus Voelter says is a vital part of DSL design. (cite this using Markus Voelter: dsl engineering)

Concessions and Drawbacks

MORE HERE LATER

4.2.2 Code generation

The other part I built is a code generator. What this does is it takes input in the form of code that is written in a DSL in accordance with the grammar, and output files containing generated code, In this case containing the full JSON representation of a Dialogflow agent.

On a higher level, let´s have a look at this graphic: ...

4.2.3 Usage

When you use the tool/compiler that I built, you run this compiler with the input being the DSL code; the output will be a full JSON representation of a Dialogflow agent which can be sent to the Dialogflow V2API. The V2API allows you to update the online version of your agent using the JSON files.

4.3 Circumstances under which using the DSL is beneficial

Under which circumstances will the DSL provide a benefit for a project?

5 | Summary and further Work

It was my job for this Bacherlor thesis to build a prototype for Df to find out if you can configure Df agents as code in a DSL.

For further projects one could consider

• build a second compiler for amazon lex • my compiler is not ready for the market (minor bugs and flaws), so there is more work here • build a generator for DSL code from the JSON representation of an agent (the other way round) so you would want a “reverse compiler” to be able to input . . . and get, as an put, the DSL – no more manual . . . !

6 | Declaration of Independence