Design and Development of a Cognitive Assistant for the Architecting of Earth Observing Satellites

by

Antoni Virós Martin

September 2017

Submitted to the faculty of the Barcelona School of Informatics (FIB)

of Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

in Partial Fulfillment of the Requirements for the

Bachelor Degree in Informatics Engineering - Computing Specialization

Under the guidance of

Daniel Selva Valero, Cornell University

Department of Mechanical and Aerospace Engineering

Cornell University, Ithaca, New York

“Artists transform the average into the extraordinary, engineers should do the opposite”

Abstract

The aim of this thesis is to develop a cognitive assistant for architecting earth observation satellite systems. The motivation for such a system comes from both the recent commercial success of different cognitive assistants as well as the existing challenges in system architecture in general, including earth observation satellite systems. This system has been developed using a layered architecture, where the first layer is a set of front-ends which are deployed on different client machines, including computers, virtual reality headsets, and physical embodiments. The next layer is a server which distributes the requests the user makes from all the front-ends to the different back-ends, which are the ones responsible for performing the basic functionalities of the system. A system of skills, pieces of software which provide the functionalities the end user uses, is built on top of this server. Finally, the back-ends and the skills use different data sources to perform their functionalities. This report describes the whole architecture of the system, with special emphasis on the "Historical Database" and the "Historian Skill", which I have developed by myself, unlike other parts of the system where the development effort was shared. The final result of this thesis is Daphne, the first open-source cognitive assistant to support the high-level design of earth observation satellite systems by helping reduce the cognitive load of the systems engineer.

Resum

L'objectiu d'aquest treball de fi de grau és desenvolupar un assistent cognitiu per a dissenyar sistemes de satèl·lits d'observació terrestre. La motivació per crear aquest sistema prové tant del recent èxit comercial de diversos assistents cognitius com dels reptes existents en el disseny de sistemes en general, en els que s'inclou el de sistemes de satèl·lits d'observació terrestre. El sistema s'ha dissenyat amb una arquitectura de capes, la primera de les quals és un conjunt d'interfícies implementades en diferents màquines client com ordinadors, cascs de realitat virtual i encarnacions físiques (robots). La següent capa és un servidor que distribueix les sol·licituds de l'usuari des de les interfícies cap als diferents backends (processadors dorsals), que són els responsables de dur a terme les funcionalitats bàsiques del sistema. Construït també sobre aquest servidor hi ha un sistema d'habilitats, peces de programari que proveeixen les funcionalitats que utilitza l'usuari. Finalment, els backends i les habilitats utilitzen diverses fonts d'informació per dur a terme les seves funcions. Aquest informe descriu l'arquitectura completa del sistema, amb un èmfasi especial en la "Base de Dades Històrica" i l'"Habilitat d'Historiador", que he desenvolupat personalment, a diferència d'altres parts del sistema, que s'han desenvolupat de manera compartida. El resultat final d'aquest projecte és Daphne, el primer assistent cognitiu de codi obert que dóna suport al disseny d'alt nivell de sistemes de satèl·lits d'observació terrestre tot ajudant a la reducció de la càrrega cognitiva de l'enginyer de sistemes.

Resumen

El objetivo de este trabajo de fin de grado es desarrollar un asistente cognitivo para diseñar sistemas de satélites de observación terrestre. La motivación para crear este sistema proviene tanto del reciente éxito comercial de distintos asistentes cognitivos como de los retos existentes en el diseño de sistemas en general, que incluye el de satélites de observación terrestre. El sistema se ha diseñado con una arquitectura de capas, la primera formada por un conjunto de interfaces que se ejecutan en diferentes máquinas cliente como ordenadores, cascos de realidad virtual y encarnaciones físicas (robots). La siguiente capa es un servidor que distribuye las solicitudes del usuario desde las interfaces hacia los diferentes backends, que son los responsables de llevar a cabo las funciones básicas del sistema. Sobre el servidor se ha construido un sistema de habilidades, piezas de software que proveen las funcionalidades que utiliza el usuario. Finalmente, los backends y las habilidades utilizan distintas fuentes de información para llevar a cabo sus funciones. Este informe describe la arquitectura completa, con un énfasis especial en la "Base de Datos Histórica" y la "Habilidad de Historiador", que he desarrollado personalmente, a diferencia de otras partes del sistema que se han desarrollado de forma compartida. El resultado de este proyecto es Daphne, el primer asistente cognitivo de código abierto que apoya al diseño de alto nivel de sistemas de satélites de observación terrestre ayudando a la reducción de la carga cognitiva del ingeniero de sistemas.

Acknowledgements

In the first place, my thanks go to my advisor, Daniel Selva, for helping me develop this project while at the same time helping me adapt to living in the US and being a great advisor altogether.

In the second place, I would like to thank all of the Daphne team: Hyunseung Bang, Arnau Prat, Amrit Kwatra, and Ria Singh for helping the project become a reality so fast.

I also have to thank all the different sources of funding which have allowed me to do this project in the USA: the TFG scholarship from CFIS and Fundació Cellex, the MOBINT scholarship from Generalitat de Catalunya, and Cornell University.

Finally, this project has been funded in part by the following National Science Foundation grant: NSF CMMI 1635253 "Improved human-computer interaction for the design of complex systems".

Contents

Abstract **ii**

Resum **iii**

Resumen **iv**

Acknowledgements **v**

Contents **vi**

List of Figures **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Background 4

1.3 Approach 8

2 Failed approaches **10**

2.1 Reusing an already existing CA system architecture 10

2.1.1 Lucida 11

2.1.2 Mycroft 12

2.1.3 Next steps 13

2.2 Reusing existing QA systems 13

2.2.1 YodaQA 14

2.2.2 Aqqu 15

2.2.3 Next steps 15

2.3 Using unsupervised learning for creating database queries . . 16
2.3.1 Neural programmers 16

3 Daphne: General architecture 18
3.1 Overview 18
3.2 Data sources 19
3.2.1 Design Solutions database 20
3.2.2 Expert Knowledge database 21
3.2.3 Historical database 22
3.2.3.1 Overview 22
3.2.3.2 Data scraping 23
3.2.3.3 Data processing 27
3.2.3.4 Data mining 28
3.2.3.5 Ontology 29
3.3 Back-ends 30
3.3.1 Architecture Evaluation 31
3.3.2 Data Mining 32
3.3.3 QA System 34
3.4 Daphne Brain 36
3.5 Skills 38
3.5.1 iFEED Skill 38
3.5.2 Critic Skill 40
3.6 Front-ends 42
3.6.1 Web & Voice Visual Interfaces 42
3.6.1.1 iFEED Interface 42
3.6.1.2 Critic Interface 43
3.6.1.3 Historian Interface 45
3.6.2 Daphne VR 46
3.6.3 Physical Embodiment 49

4 Historian Skill **52**

4.1 Skill overview 52

4.2 Text processing 54

4.3 Question classifier 55

4.4 Definition of a Question Type 58

4.5 Feature extraction 60

4.6 Data augmentation 62

4.7 Database querying 62

4.8 Answer construction 63

5 Limitations and Future work **64**

5.1 Limitations 64

5.2 Future work 65

6 Conclusion **67**

7 Personal thoughts **68**

A Source code of the project **70**

References **71**

List of Figures

3.1 Overview of Daphne architecture 18
3.2 Historical database schema 24
3.3 Decision tree for the data mining algorithm 29
3.4 Small portion of the ontology 30
3.5 Screen-shot of the iFEED Interface 43
3.6 Screen-shot of the Critic Interface 44
3.7 Historian temporal interface 46
3.8 Views of the VR interface 46
3.9 Design of the VR interface in planar shape 47
3.10 Design of the Daphne physical embodiment 49
3.11 Hardware of the Daphne physical embodiment 49
3.12 Software architecture of the Daphne physical embodiment . . 50

4.1 Example of a question type JSON file 60

To my parents and my sister

Chapter 1

Introduction

1.1 Motivation

Architecting Earth Observing Satellite Systems (EOSS) such as the NASA A-Train or the NOAA Polar weather system is a challenging task, and it will likely get harder as we demand missions that are more affordable, reliable, robust and which can generate more and better data products using less resources.

NASA's technology roadmap for technology area "TA 11: Modeling, Simulation, Information Technology, and Processing", recognizes this problem as it describes a need for improved "Analysis Tools for Mission Design". Specifically, the document states that current tools are thought for monolithic missions and only take into account small parts of the system at a time [1]. This limits their ability to consider new trends on mission design, which include distributed, fractionated, or heterogeneous systems [2], and creates lost opportunities on designs which are not even considered due to the partial analysis. [3]

This trend towards more distributed missions can be seen in the latest missions flown by major space agencies, such as ESA's Earth Explorers (e.g. SMOS, Cryosat) and NASA's Earth Science System Pathfinders (e.g. Calipso, CloudSat, and more recently CYGNSS and TROPICS). These missions have two things in common: the mass and the number of instruments of each satellite have been reduced compared to the larger monolithic missions of the early 2000's (e.g., Envisat, UARS, Terra) [4] and most of them involve some sort of coordination among multiple assets to achieve the desired functionality (e.g., constellations, clusters, trains) [2]. Two commonly cited advantages of such distributed architectures are the increase in reliability and robustness achieved by reducing the number of single points of failure in the system, and the improved affordability as individual units become smaller and less costly and leverage miniaturized commercial off-the-shelf components [5]. On the other hand, the need for coordination among assets in distributed systems may lead to increased complexity in terms of position and attitude control as well as inter-satellite communications. [6]

Thus, improved tools are needed to architect these complex constellations, clusters, and trains, as there is a need to account for the entire system as opposed to a single satellite during the early design process. There has been some research in this area [7, 8, 9], including tools curretly being developed by NASA like TAT-C [10]. One shortcoming of these tools is that they provide limited cognitive support to the users, who can suffer from information overload when analyzing large, high-dimensional design spaces (a typical formulation of an EOSS architecting problem defines billions of valid design alternatives).

Indeed, system architecting remains mostly an art rather than a science, even 20 years after the publication of the foundational work in the field by Rechtin and Maier [11]. This is due, mainly, to the fact that it is a task that requires creativity and adaptability and dealing with deep uncertainty and ambiguity, and these abilities are hard, if not impossible, to compartmentalize and standardize. This does not mean that parts of the process can not be automated; in fact quantitative tools such as simulations and optimization have their space in systems architecture to help select the preferred system configuration and alleviate some of the well-known cognitive biases and limitations of humans.

To counter this increase in cognitive difficulty, intelligent agents called Cognitive Assistants (CA) have been studied for the last 20 years. Their objective, as well as that of most decision support tools developed, is "augmenting human intellect", as told by D.C. Engelbart [12] in one of the first works in human-computer interaction (HCI) back in 1962. What makes CA different from other intelligent agents in use is the types and modes of interaction with the user, as well as the use of cognitive architectures. Most CA send and receive information to/from the user by means of natural language, either through a voice or text-based interface. Some of them also can take image inputs, like Lucida [13, 14], or Google Photos Assistant [15]. Another big difference with other intelligent agents is the fact that some CA can take the initiative and act on what they think the user wants, by using different cognitive architectures [16] like Belief-Desire-Intention (BDI) [17], Logic-based agents [18], Reactive agents [9] or Layered architectures combining different models. The explicit use of some model of the user's cognitive process is arguably the most distinctive trait of modern CA.

The last few years have seen an exponential increase of usage of CA, as commercial systems have been developed by many large software companies to help do mundane tasks faster [19, 20, 21, 22, 23]. IBM has also reused parts of IBM Watson [24] to create the Watson Cloud [25], a set of CA APIs. There are also some research-focused CA [13, 14, 26, 27, 28, 29], some of which are open-source.

Motivated by the challenges of system architecture in general and architecting EOSS in particular, and inspired by the success of these commercial CA, an opportunity was identified to explore a mixed-initiative approach to this task and thus develop Daphne, the first CA to support the high-level design of EOSS.

This report explains in detail how the Daphne system was built, including some failed approaches which ended up leading to the final design. It then goes on to explain the software architecture: all its different front-ends; the Brain, a piece of software that distributes the tasks between the different sub-systems; the different back-ends which do the "heavy work"; the data sources from which the back-ends obtain the data to work with; and the *skills* on top of the Brain which provide all the high-level functionalities that the end user sees. The report then continues with a detailed description of the Historian skill, as it was the one to which I devoted the most time. Finally, the limitations, future work and conclusions of the whole project are presented.

1.2 Background

Intelligent tools have been used to support the design of complex systems since the dawn of the computing era, like [30] or others as seen in [31, 32]. These tools can take different forms, which include intelligent Computer Aided Design (CAD) systems [33], knowledge databases [34], design assistants [8] and design critics [35]. Since Daphne is centered on the first stages of design, sometimes referred as conceptual design (or system architecture in the case of complex systems), we are going to focus this first part of the background review on this kind of tools.

Most tools for the first stages of engineering design are catered towards helping the human performing the design task instead of substituting him/her, and enhancing their cognitive abilities. They usually take the shape of interactive visualization and decision support tools which allow for the analysis of the different design alternatives, and which have the capacity to handle the thousands to billions of options which can exist for a design problem. Examples of such tools include [36, 37, 38, 39]. Some of the visualization tools allow the users to alternate between and compare different views of the data (e.g., decision space vs objective space, or 2D slices of the objective space), highlight architectures sharing certain features, and reduce the objective space search to a much more manageable one [36, 37, 40, 41]. Other tools utilize unsupervised machine learning algorithms such as manifold learning, feature selection, and clustering to help users visualize solutions in a high-dimensional space [39, 42]. To further reduce the cognitive load of system engineers, one of the main objectives of the tool in this paper, other tools combine visualization with data mining algorithms that extract useful knowledge or insights, often in the form of simple logical rules with an if-then structure, such as "IF any spacecraft in the architecture weighs more than 3,000kg, THEN the architecture is likely to have low cost-efficiency" [43, 44, 8]. The use of logical rules as data structure for these insights has a long tradition in artificial intelligence, and is motivated by evidence that not only are logical rules easy to understand by humans, but they may also be the way human experts actually solve complex problems [45]. But even with all the work in reducing the information overload [46, 47], often the amount of data is still so large that it is hard to manage. This means there is a need to further support the systems engineer to help him or her direct their attention to specific portions of the dataset or other aspects of the problem, depending on relevance and other factors, and this is where CA come into play.

CAs (also called intelligent personal assistants or IPA in some of the literature) have a long story: starting with NLS from Engelbart back in 1962 [12], there has been a continuous stream of them: DENDRAL [48], MYCIN [49], DARPA PAL [50], OAA [51], IRIS [52], RADAR [53] or CALO [17]. More recently, as voice recognition software and natural language processing have advanced to an almost usable level, commercial alternatives have appeared, such as IBM Watson [24, 25], Wolfram Alpha [54], Siri [19], Google Assistant [20], Microsoft Cortana [22], Amazon Alexa [21], Facebook M [23] or Mycroft [55]. There has also been research on them, resulting in software such as Lucida [13, 14, 56], Cougaar [27] or OpenCog [29]. All these systems share the fact that the interaction is done either by natural language or through pictures and that all of them are generalist: they try to answer as many queries as possible from the user using a plethora of data sources, and some of them help the user with personal organization by reading emails, setting appointments and, in general, making the use of the system where they are simpler. But generalist CAs are of no use when the task at hand is very specialized. Thus, specialized CAs in the fields of aerospace and design are described next.

Most CAs in aerospace are thought out to be used by humans who are piloting one of the many vehicles which use either air or space as a medium of navigation: be it planes, helicopters, UAVs or space vehicles. Examples of these assistants include CAMA, which is an intelligent assistant for ensuring a pilot's situational awareness during a flight [57]. The CA is capable of understanding the flight situation and combine that with the intent of the pilot to keep a human-like communication with him/her to ensure their situational awareness. In the case of traffic conflict, it signals warning signs and generates proposals on how to resolve the conflict. A similar type of assistant for multi-UAV guidance have also been developed [58]. This system also helps the pilot of multiple UAV systems by telling the pilot when something strange might be happening in one of the missions and trying to keep the pilot engaged in all of the missions without getting him to a point of information overload. COGAS is another intelligent assistant, and supports crew of a combat information center in a Navy ship [59]. COGAS combines multiple sources of information to perform tasks such as obtaining and displaying various track data from various sensors and radars, and identifying an unknown object around the Naval ship. One very recent project in this field is [60], which is a CA thought for astronauts who are on missions where real time communication with Earth is not feasible, meaning they need to have cognitive assistance with them to solve most of the problems which might arise without help from Earth.

On the other hand, CAs for design are closer to Daphne in the sense that all of them are thought out to help the systems engineer come up with a good design in the field where they are applied. TAC [61], for example, does a trade-space exploration by following the commands of different stakeholders. In [62], a rule-based system is used to provide recommendations on manufacturing technology designs. There is also [63], where an expert system is used to evaluate alternatives on highway building, having the user interact by choosing from all the proposed alternatives. A much more recent effort, PQE [64], tries to model the user curiosity and creativity so it can come up with design alternatives the user might not have thought of, with the goal of helping the user think out of the box. Another recent work is [65], where human and computer synergies are studied so the final designs are more innovative. There has also been work in making existing design tools adapt their own functionalities depending on their use [66, 67]: one result is an email classifier which learns from how the user classifies its emails, while the other result is a more intelligent version of the Optimization Toolbox of MATLAB which can learn what kind of optimization algorithm is needed for different design problems.

The last part of this section is centered on Question Answering Systems such as the Historian Skill from Daphne. As long as there has been AI there have been QA systems, and most CAs have one, including all the commercial ones referenced before. Examples of current open-source research-focused QA systems and frameworks include OAQA [28], YodaQA [26] and Aqqu [68]. All these QA systems are built to answer simple questions on broad domains, which is something not needed for Daphne, which requires answering complex questions in a specific domain. This is the reason why the Historian Skill is implemented as a Natural Language Interface for a Database (NLIDB), a special kind of Restricted Domain QA system in which all the questions are translated into database queries. A few notable NLIDBs are PRECISE [69] or NALIX [70], but there are more recent efforts such as [71, 72, 73, 74, 75] as well as others as seen in [76, 77].

Finally, neural network models have been used for the text classification inside the skill. The model used is [78], with the parameter tuning recommended in [79] but there are a few more models which are even more up to date [80, 81, 82] but require higher computational power due to the fact that the training is much less prone to parallelization for a small performance benefit.

1.3 Approach

The objective of this project, as its title states, is to design and develop the software architecture for a CA for architecting Earth Observation Satellite Systems.

To achieve this objective, a team was formed, and the project has been developed as a collective effort. This means that, while most of the design of the software architecture ended up being on my shoulders, some parts of the development effort went into other people's. Although the chapters are organized following the project structure, who did what is accounted for in each section inside them.

The rest of the report will try to be both a general explanation of the whole project and at the same time a description of the work done by myself. It is organized as follows: Chapter 2 explains all the different approaches which were tried during the development of the project, why they were not chosen and what was learned from them. Chapter 3 is an overview of the whole system. Then the report gets to Chapter 4, where the Historian Skill, the one I devoted the most time to, is explained in detail. Next, Chapter 5 develops the limitations of the system and the future work which can be done on it. Finally, Chapter 6 wraps the whole project up, along with a few personal thoughts in Chapter 7.

Chapter 2

Failed approaches

Personally, I feel that putting this section in the report is a reminder that research is not always a straight path to success, and for every small step that tries to push forward the state of the art there are a lot of attempts that simply utterly fail. This is why I think letting other people know not only of what works but what does not is important to save time for future researchers who might think a certain approach has not been tested yet. I believe the fact that most research papers only explain the way forward without explaining all the found dead ends is a one-way street to lost time in research.

After this personal statement, here I detail the three failed approaches which ended up leading to the final design of Daphne. They are ordered as I tried them during development, so a trend of more general to more specific approaches can be seen.

2.1 Reusing an already existing CA system architecture

When beginning a project, and specially a software project, the main task during literature review is finding software which already fulfills the needs of the project and has a license that allows reusing it while expanding or limiting its capabilities to fit the exact needs of the users. Nobody wants to reinvent the wheel, as that is a waste of time and resources which can end with more buggy solutions than the already existing ones.

So I did what was expected: I started searching for cognitive and intelligent assistants which could be reused for Daphne and I found two viable alternatives, both open-source with permissive enough licenses. These two solutions were Lucida [13, 14, 56] and Mycroft [55]. The reasons to choose these two are that, apart from being open-source, they have a very similar architecture which in turn inspired that of Daphne, and both of them have been developed recently, which is usually a good thing when trying to run them on a computer, although nothing is always as easy as it seems.

2.1.1 Lucida

Lucida [13, 14, 56] is an IPA developed by the Clarity Lab at the University of Michigan. It is a complete CA solution, with Question Answering capabilities for factoid questions using OpenEphyra [83, 84], along with Speech Recognition using Kaldi [85], and even Image Recognition with OpenCV [86]. It also has the ability to add other services which leverage these three basic functionalities.

The main problem with this system is one which I consider very important: the ease of installation. The whole system can only be installed and run fully under an Ubuntu 14.04 system, and this means in less than 2 years it will not run in any non-obsolete system. It also makes permanent modifications to the system by creating certain symbolic links and installing some software in an unconventional way. Although some installation scripts have been improved and there have been some efforts to port it to Ubuntu 16.04, the system cannot (as of yet) run properly in that system, and one must switch the default GCC compiler version for an older one to be able to use the full system. This ended up meaning I could not even run the system in my computer after a few days of work, by which point I decided I would not use it.

This does not mean it has nothing of value to add to the final design: the idea to build the Daphne Brain came from Lucida's Command Center, and I also discovered the Thrift library for Inter Process Communication (IPC) while looking at the source code for Lucida. Thrift is now used as the communication layer between some sub-systems of Daphne.

2.1.2 Mycroft

Mycroft [55] is an IPA too, but in this case it is developed by Mycroft AI Inc., a for-profit company. It is also a pretty complete system, with support for Speech Recognition using either its own implementation or one of the implementations from the big data companies, namely IBM, Microsoft, Google and Amazon. It works in a similar fashion to most other commercial IPA systems: it provides a bare framework with a few working examples and lets the developers make their own "skills", which can add functionality to the system.

Using Mycroft was a big part of the project until the team realized that it was much more than what we needed in some areas, like the whole skills system, and it lacked in other ways, like in interacting with some web applications developed at the lab which needed to work together with Daphne or having no support to run as a library as needed for the physical embodiment and the VR interface. Also, the system is still in heavy development, which meant that some Application Programming Interfaces (APIs) changed a lot, creating a lot of headaches to the whole group when trying to keep up to date.

It got to a point where keeping up to date with the system was taking more time than actually developing the features needed for Daphne, so the team took the decision to abandon Mycroft and instead use our own web-based interface and server, which could be developed at a pace acceptable for the team.

Ideas taken from Mycroft include using a catch-word to start voice recognition, the way the skill system is implemented, the use of WebSockets for handling real-time communication between client and server and the idea to use voice recognition and voice speaking systems developed by the aforementioned big data companies, as other open-source, free solutions have underwhelming performance, as we found out when trying to use the systems inside Mycroft.

2.1.3 Next steps

As it became clearer that no complete solution in the ecosystem could be reused as the base for Daphne due to its special needs, a custom solution was developed, as seen in Chapter 3. This does not mean the search for already existing solutions stopped, but it was moved to only parts of the system instead of the whole of it. This is explained in the next section, where the search for a working QA system is described.

2.2 Reusing existing QA systems

One essential part of the system is the QA system, which has the job of recognizing the different questions asked by the users and answering them correctly if it can. Once again, before trying to build the whole subsystem from scratch I searched for already implemented solutions. Two QA systems were chosen as the best options due to their licenses and accuracy results: YodaQA [26] and Aqqu [68]. QA is an always evolving field, so every year there are more and more accurate systems, and this is why once again having recent implementations is usually a good sign for the system.

2.2.1 YodaQA

YodaQA [26] is an "open source Factoid Question Answering system that can produce answer both from databases and text corpora using on-the-fly information extraction". It is written in Java, which helps in integrating with the rest of the system, which is written in a mix of Java and Python.

The problems started very soon: the databases YodaQA can use are ontologies written in RDF format, and although one part of this project includes developing an ontology of the earth observation missions database in this format, adapting it for use in this system was deemed too time consuming for the scope of the project, as the system is thought out to answer questions on DBpedia [87] and Freebase [88], two generalist open RDF databases. Another big problem is the fact the system is slow and has a low accuracy of around 45% on the most common tests like the TREC benchmark [89] measured as the Mean Reciprocal Rank (MRR), which is a measure of the probability of getting the correct answer on a question, as defined in [90]. Finally, the system has not been maintained for the last year, meaning there is no improvement in sight for all those problems which have been identified.

All these problems prompted the decision to not use YodaQA and search for other alternatives, like Aqqu. Trying to use YodaQA made me realize users want fast and accurate systems, and that I need to use very well-maintained software solutions if I intend to not lose a lot of time on configuration and bug-fixing.

2.2.2 Aqqu

Aqqu [68] is an "end-to-end system that automatically translates a given natural-language question to the matching SPARQL query on a given knowledge base". It is written in Python, making it even easier to integrate than YodaQA.

The problems are very similar to those of YodaQA: the system is written to work on Freebase, with very few documentation on the code except for the paper, so this makes it hard to adapt to other databases. It is, though, much faster and more accurate than YodaQA, as it advanced the state of the art back in 2015.

When trying to integrate this system with Daphne, I realized three things: first of all, using general QA systems was a waste of resources, as they are catered towards answering easy questions on giant datasets of unstructured data, while what Daphne users need is complex answers to complex questions on a small dataset of very structured data. Also, ontologies need to be very well crafted to obtain good answers on them, which requires a knowledge and time I do not have. Finally, using software catered for a paper can easily turn into a very tedious task, as adjusting the program to produce results which are useful outside of paper publishing can be very hard.

2.2.3 Next steps

After these two attempts, it was clear that no complete QA system would help, as most of them are too generic to be useful for the task at hand without much adaptation work. It was clear the required system needed to be specific for the historical database. This brings us to the last failure in trying to use existing systems: using unsupervised deep learning to automatically translate natural language questions to database queries.

2.3 Using unsupervised learning for creating database queries

In the last two years there has been a resurgence [74, 73, 91] of an application of deep learning known as Neural Programmers: neural networks whose output is a set of instructions to be executed on the computer. There was some research on the topic more than 20 years ago [92], but it never took of, together with the rest of deep learning.

2.3.1 Neural programmers

One possible application of neural programmers is in QA over a database: a database query is nothing more than a set of instructions applied over a set of tables which give as a result either a text, a number or a list. This means that using such a system can be really helpful in saving development time when adding new questions to the QA system: instead of having to manually write a query for each question type, as seen in section 4.7, the system could train itself by providing access to the database, a lot of example inputs, and a lot of example outputs.

All this seems too good to be true, and for now it is. As seen in most of these publications, and especially [91], the accuracy for the task at hand is too low to be useful for a consumer-facing interface. What's more, the examples shown in the papers could be considered toy examples, which means bigger and harder questions like the ones found in Daphne would bring the system to its knees both in accuracy and computing power required to train the system. All these problems made me take the decision to not use such a solution until the state of the art in this field is furthered, instead relying on more classic techniques like text classification, feature extraction and manual query creation, as seen in Chapter 4.

Chapter 3

Daphne: General architecture

This chapter of the report consists of 5 different sections, and it is the main chapter of this report, as the bulk of the work done is explained here.

3.1 Overview

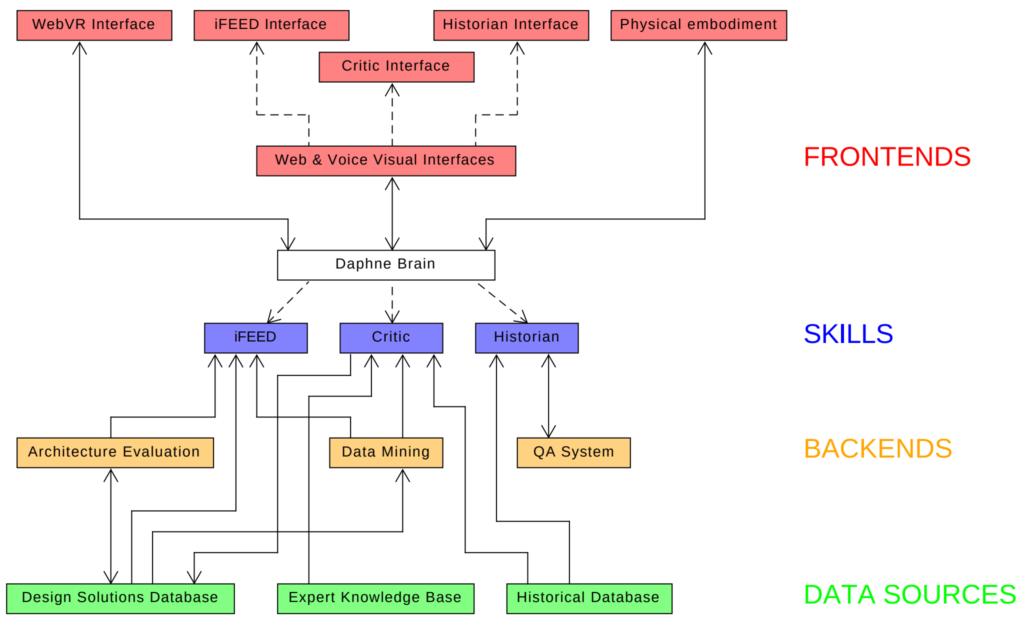


FIGURE 3.1: Overview of Daphne architecture

Daphne is structured as a 4-layer system: it has three kinds of front-ends which can interact with the user in different ways (section 3.6), a front-end server (the Daphne Brain in section 3.4) to direct all user requests to the appropriate back-end which also hosts the different *skills* (described in 3.5), three back-end services to resolve queries from the users (in section 3.3) and three data sources from which to obtain the data to answer some of the queries or save the progress of the user in using the application (explained in section 3.2).

It is easy to notice in figure 3.1 that all the skills also have an interface, and this is due to them being developed by different people, which meant everyone built their own interface to test the functionalities of their own skills. One important task left to do in the future is build a unified interface, as there was no time during the development of this first version to do so.

Finally, this Chapter is organized from the bottom up: it explains the Daphne sub-systems starting from the bottom of figure 3.1 and it ends with the ones at the top, as this helps in understanding the whole system better because each layer needs to refer to ideas and details from the layer below.

3.2 Data sources

There are three different data sources from which Daphne can get its information: a design solutions database, which contains information from a great variety of possible architectural solutions for the problem at hand; an expert knowledge database, with expert rules and recommendations on designs; and a historical database, with data from all past and present public and civilian earth observation missions.

The three of them help in obtaining a better final design on their own, but the real powers of the system are obtained when the three of them are combined to obtain a comprehensive set of recommendations for the system engineer developing the EOSS.

An overview of the three data sources is provided, but only the historical database will be extensively developed, as it is the one I have worked on.

3.2.1 Design Solutions database

The three main objectives of the design solutions database are to have an initial data-set of possible designs for the system being designed so the architect has something to compare its own designs to, to save the designs the engineer or the computer come up with so they are not lost after the work session is finished, and last but not least, to have a data-set over which the Data Mining back-end (subsection 3.3.2) can work and extract features.

The initial designs can be generated using various sampling methods such as random sampling and Latin Hypercube sampling, which can sample the feature space in an unbiased way, or optimization algorithms, which are more likely to generate high-quality solutions, thereby introducing a certain bias in terms of sampling of features.

The data saved for each system design contains which instruments are assigned to which satellites, the total cost and the science benefit as computed by the Architecture Evaluation back-end (subsection 3.3.1), and the set of rules triggered in JESS [93] to obtain both the cost and the science benefit using this back-end. All this data helps in the graphical representation of the designs, as the two axes which appear in the plot are always cost and science benefit, in other parts of the interface which need the exact design, and in the internal workings of the iFEED and Critic skills, as both of them need to reenact the execution of the Architecture Evaluation back-end to generate their expected responses.

This data is saved in JSON files, which are stored either as bare text files or inside a MongoDB database depending on the needs of each subsystem. The exact format is a bit-mask for the assignments of instruments to satellites, a double for the cost, a double for the science benefit, and a text array for the rules fired. The authors of the generation of information for this database are Hyunseung Bang and Daniel Selva.

3.2.2 Expert Knowledge database

The Expert Knowledge database is a little different to the other two in that it is not made of data but of logical rules: it is a set of if-then rules, to be more specific. The rules are made to encode domain-specific knowledge about how to architect EOSS.

One example is a rule that states that UV/VNIR chemistry spectrometers should be flown in afternoon sun-synchronous orbits rather than morning or dawn-dusk orbits. This is because the dawn-dusk orbit has suboptimal illumination conditions for this kind of instrument, and pollution levels typically peak in the afternoon as opposed to the morning [94]. Another example is that an active and a passive instrument that use the same frequency band should not be used in the same spacecraft, as they may interference with each other.

All of these rules can be either simple or complex, but all of them exist for the same purpose: having access to them without having to look at reference books or having to remember them during the design process may reduce the cognitive load of the system engineer, who has to consider many other design factors, and thus lead to improved performance and better designs.

The rules are written using of combination of raw JESS rules and a Java program which generates rules based on Excel files with a set structure. JESS is a superset of CLIPS, so as it happens with this language, rules are actually stored as raw text either in files or directly into RAM memory, and its format can be checked in both CLIPS and JESS documentation. The authors of the rules are Arnau Prat and Daniel Selva.

3.2.3 Historical database

The historical database has been sourced from a reliable source of information on Earth Observation Satellites: the Committee on Earth Observation Satellites (CEOS) Database [95], a joint effort by CEOS and the European Space Agency (ESA) to create a comprehensive database of all the Earth Observation Satellites which have been launched since the beginning of the Space Age, with detailed information on the orbits, launch and EOL dates, the agencies, the purpose, the instruments, the measurements and the data provided by each one of these missions. The exact number of missions in the database is 567, with 839 instruments also described. The whole development work on this database for this project has been done by me, but the authors of the information are CEOS and ESA.

3.2.3.1 Overview

The procedure to obtain a usable version of this database has been hard, as the only publicly available of it is the CEOS Database website, which contains tables and detailed pages on all missions, instruments and measurements. While this is good for reading about single missions and counting, it is completely insufficient for the advanced uses needed in Daphne, which require data analysis and data mining to get some answers out of it, as well as advanced SQL queries which are impossible to perform on the web interface.

This is why the first task which was performed was to scrape all the data in the website and save it in a classical SQL database, as explained in sub-subsection 3.2.3.2. Having the same textual data in a classical database, while it enables a lot more queries than the ones available from the web, still misses on a lot of numeric data insights which would be available just by classifying the data in categories and putting those fields which are numeric as actually numeric. This is described in detail in sub-subsection 3.2.3.3. While this helps in answering most queries, sometimes there is information which is not really there in the database, and some data mining algorithms need to be run over it, as discussed in sub-subsection 3.2.3.4. Finally, as the database was originally thought to be used with already existing QA systems which expected ontologies, an ontology has also been built, as seen in sub-subsection 3.2.3.5.

The end result is the database in figure 3.2, with the schema of all the tables.

3.2.3.2 Data scraping

The data scraping has been performed using the Scrapy framework for Python, which is described as "A Fast and Powerful Scraping and Web Crawling Framework" [96]. Using this piece of software has helped save a lot of time by not having to research how to actually download and navigate a web page programatically, which could have been really tedious.

Then, the task of data scraping becomes two-fold: on one side, the structure of the data needs to be discovered and written down so the schema of the database as seen in figure 3.2 can be built. On the other hand, the "spider" program (the program which actually navigates the website) needs to be written. Both tasks complement each other, as the order of the web pages scraped is really important to maintain consistency in the database, and the

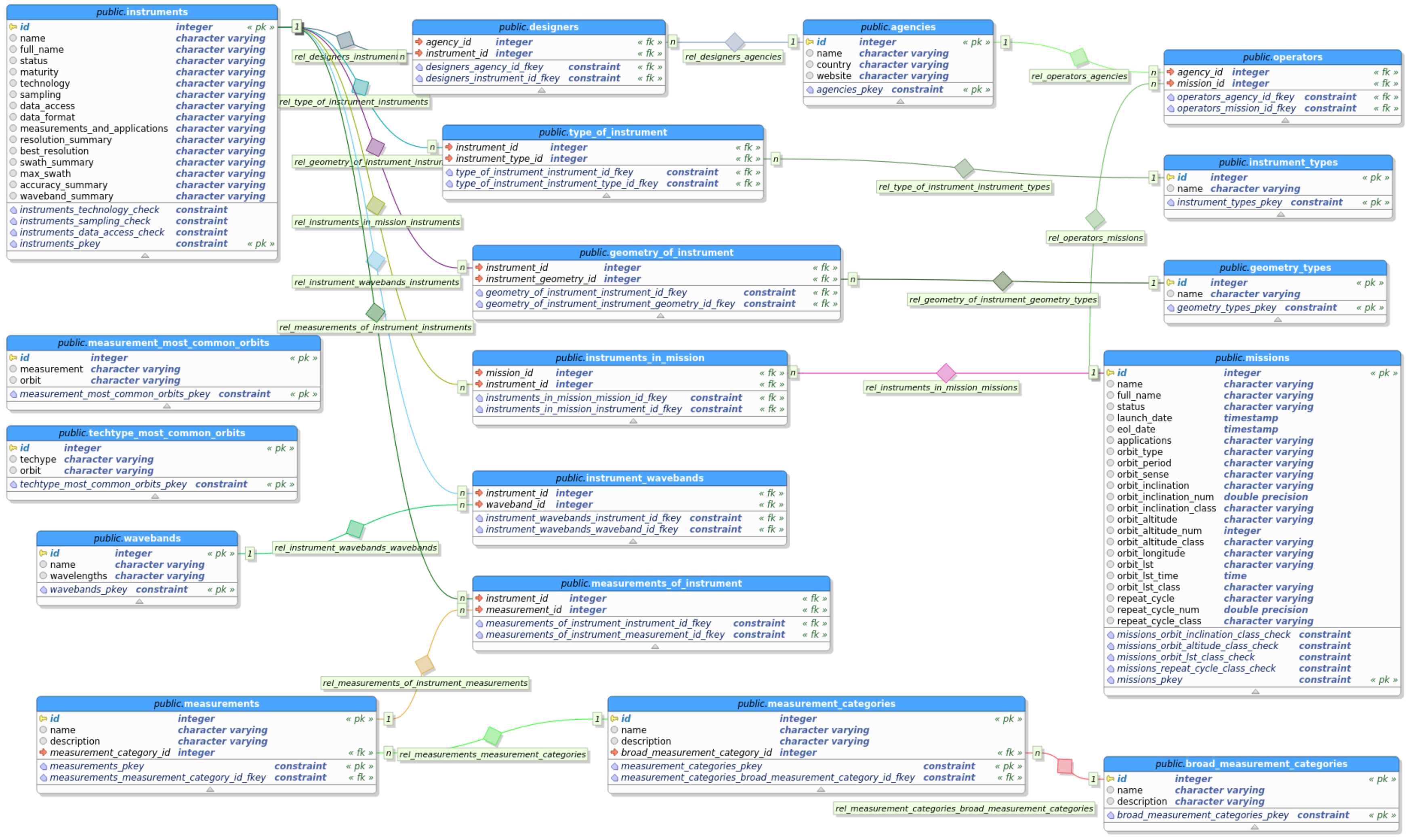

public.instruments
id integer « pk »
name character varying
full_name character varying
status character varying
maturity character varying
technology character varying
sampling character varying
data_access character varying
data_format character varying
measurements_and_applications character varying
resolution_summary character varying
best_resolution character varying
swath_summary character varying
max_swath character varying
accuracy_summary character varying
waveband_summary character varying
instruments_technology_check constraint
instruments_sampling_check constraint
instruments_data_access_check constraint
instruments_pkey constraint « pk »
public.designers
agency_id integer « fk »
instrument_id integer « fk »
designers_agency_id_fkey constraint « fk »
designers_instrument_id_fkey constraint « fk »
public.agencies
id integer « pk »
name character varying
country character varying
website character varying
agencies_pkey constraint « pk »
public.operators
agency_id integer « fk »
mission_id integer « fk »
operators_agency_id_fkey constraint « fk »
operators_mission_id_fkey constraint « fk »
public.type_of_instrument
instrument_id integer « fk »
instrument_type_id integer « fk »
type_of_instrument_instrument_id_fkey constraint « fk »
type_of_instrument_instrument_type_id_fkey constraint « fk »
public.instrument_types
id integer « pk »
name character varying
instrument_types_pkey constraint « pk »
public.geometry_of_instrument
instrument_id integer « fk »
instrument_geometry_id integer « fk »
geometry_of_instrument_instrument_id_fkey constraint « fk »
geometry_of_instrument_instrument_geometry_id_fkey constraint « fk »
public.geometry_types
id integer « pk »
name character varying
geometry_types_pkey constraint « pk »
public.measurement_most_common_orbits
id integer « pk »
measurement character varying
orbit character varying
measurement_most_common_orbits_pkey constraint « pk »
public.techtype_most_common_orbits
id integer « pk »
techype character varying
orbit character varying
techtype_most_common_orbits_pkey constraint « pk »
public.instruments_in_mission
mission_id integer « fk »
instrument_id integer « fk »
instruments_in_mission_mission_id_fkey constraint « fk »
instruments_in_mission_instrument_id_fkey constraint « fk »
public.missions
id integer « pk »
name character varying
full_name character varying
status character varying
launch_date timestamp
eol_date timestamp
applications character varying
orbit_type character varying
orbit_period character varying
orbit_sense character varying
orbit_inclination character varying
orbit_inclination_num double precision
orbit_inclination_class character varying
orbit_altitude character varying
orbit_altitude_num integer
orbit_altitude_class character varying
orbit_longitude character varying
orbit_lst character varying
orbit_lst_time time
orbit_lst_class character varying
repeat_cycle character varying
repeat_cycle_num double precision
repeat_cycle_class character varying
missions_orbit_inclination_class_check constraint
missions_orbit_altitude_class_check constraint
missions_orbit_lst_class_check constraint
missions_repeat_cycle_class_check constraint
missions_pkey constraint « pk »
public.instrument_wavebands
instrument_id integer « fk »
waveband_id integer « fk »
instrument_wavebands_instrument_id_fkey constraint « fk »
instrument_wavebands_waveband_id_fkey constraint « fk »
public.wavebands
id integer « pk »
name character varying
wavelengths character varying
wavebands_pkey constraint « pk »
public.measurements_of_instrument
instrument_id integer « fk »
measurement_id integer « fk »
measurements_of_instrument_instrument_id_fkey constraint « fk »
measurements_of_instrument_measurement_id_fkey constraint « fk »
public.measurements
id integer « pk »
name character varying
description character varying
measurement_category_id integer « fk »
measurements_pkey constraint « pk »
measurements_measurement_category_id_fkey constraint « fk »
public.measurement_categories
id integer « pk »
name character varying
description character varying
broad_measurement_category_id integer « fk »
measurement_categories_pkey constraint « pk »
measurement_categories_broad_measurement_category_id_fkey constraint « fk »
public.broad_measurement_categories
id integer « pk »
name character varying
description character varying
broad_measurement_categories_pkey constraint « pk »
rel_designers_instruments
rel_designers_agencies
rel_operators_agencies
rel_operators_missions
rel_type_of_instrument_instruments
rel_type_of_instrument_instrument_types
rel_geometry_of_instrument_instruments
rel_geometry_of_instrument_geometry_types
rel_instruments_in_mission_instruments
rel_instruments_in_mission_missions
rel_instrument_wavebands_instruments
rel_instrument_wavebands_wavebands
rel_measurements_of_instrument_instruments
rel_measurements_of_instrument_measurements
rel_measurements_measurement_categories
rel_measurement_categories_broad_measurement_categories
1
n

FIGURE 3.2: Historical database schema

structure is easier to discover and write if done at the same time the spider is being built.

The order in which everything has been scraped is the following: first, all the measurement types are downloaded, parsed and classified between broad categories, categories and actual types of measurements. This is done first as they have no dependencies on other data from the database while all instruments need to have all this information to be complete. Then the space agencies are downloaded and saved into the database, as they need to be referenced in all the missions and instruments. With all this auxiliary data in place inside the database all the missions are downloaded, parsed and saved, with all the relations to the agencies included as well. Finally, the instruments are added with all their relations to agencies, missions and measurements, completing the database.

The database is built as an SQL database, and the interface used between Python and the actual database (which is PostgreSQL but can be changed) is SQLAlchemy [97], which helps in both making the database easier to model by implementing an Object Oriented interface on top of the SQL schema and easier to communicate with thanks to its declarative syntax which is much easier to compartmentalize compared to pure SQL.

One design point to take into account is how to define the relationships between the different objects in the database. There are two different options when talking about relationships: one-to-many (and many-to-one) and many-to-many. One-to-many is usually used in hierarchical relations where it makes sense for a lot of elements to be linked to a single element. One example of this kind of relationship is that a lot of measurement types will be of a single category, but not two. This is implemented using foreign keys in one of the two related tables in SQL. Many-to-many relationships link any number of elements of one type with any other number form another type. For example, instruments can be used in different missions, and a mission can have more than one instrument. This is implemented in SQL through an auxiliary table that contains pairs of foreign keys, with each pair having a key of each of the related tables. A conservative approach has been taken when modelling these relationships: first, all relationships are considered one-to-many. If during the process of data extraction a case of a many-to-many relationship is discovered for that field then the structure is changed to allow for this more complex situation. This ensures extra space and complexity is only used when absolutely needed. In the end, most relationships in the database are many-to-many, but for the few which are not, this approach has been useful in saving space and complexity.

It is important to note as well that the database is reset every time the scraper is run, as in this way the information in the database is much easier to keep up to date and with no inconsistencies.

As technical notes, a few of the most difficult and bizarre problems encountered include the POST request crafting for ASP.NET forms as the ones used by the CEOS database website, which include a lot of hidden fields which need the exact values so the page returned is actually the requested one. This brought a lot of headaches. There is also the CSS vs XPath selectors decision. Initially, everything was coded using CSS selectors as they are easier to learn and seemed to work fine, but it got to a point where the non-ambiguity of XPath won over the easy use of CSS. This also helped in discovering that the web page has a completely different HTML structure depending on the browser used: the XPath selectors are different if using Chrome, Firefox or Scrapy, which has lead to really confusing bugs when extracting the information.

3.2.3.3 Data processing

While textual data already enables a lot of queries in a SQL database, when data is actually numerical it can be good to have it be that way in the database. This also enables the database to deal with certain queries which depend on numeric or date ordering or aggregation.

Examples of fields being processed into their real types of data are the ids of all the agencies, measurements, missions and instruments, as well as the launch and end of life dates for all missions, together with all the orbital parameters of each mission like the period, the repeat cycle, the inclination, the altitude, the longitude or the local sun time, for those missions which have them.

Another important data processing task in the database is knowing which fields are actually lists, which are required and which can be empty. The methodology to obtain this information has been to always consider all elements a single required field and then, when errors happen during the data extraction from the web page, change that into either a list or a non required field, ensuring the most stringent requirements are met in the database.

For those fields with set lists of possible values, constraint checks have been added to the database so there is no way a forbidden value ends up being used. If a field is both a list and has a constrained set of values it is implemented as a many-to-many relationship, which already ensures the values constraint.

Finally, some numerical fields are also post-processed into categories. For example, orbit altitudes are given classes such as Very Low, Low, Medium, High or Very High Altitude. This is useful for both sub-subsection 3.2.3.4 and for certain queries which are better expressed in these qualitative terms rather than crude numbers.

3.2.3.4 Data mining

At a certain point during development, a question was proposed for the Historian Skill: “Which is the most common orbit for taking <measurement>?”. It seems like a rather normal question a systems engineer might want to ask, but when trying to craft a single SQL query which obtained this information from the database I realized the information was simply not there and therefore had to be added. This is due to the fact this knowledge needs to be computed using data already in the database. This is what this section describes.

The only data mining being done in the database is to obtain the most common orbit where measurements are carried out and where instrument types and technologies are usually flied. The data mining algorithm works as follows:

1. For each measurement, instrument type and instrument technology, the set of missions with it is obtained. Let this set be TTS.
2. Then, the algorithm looks for the innermost node of a decision tree where a set of conditions related to both TTS and a new set defined on the node, called DTS, are fulfilled. Each level in the decision tree adds restrictions to the DTS obtained in the last level, making the conditions harder to fulfill. DTS is the set of missions whose common denominator is that all of them have the properties defined in the node and all its ancestors. The decision tree can be seen in Figure 3.3. The conditions which need to be fulfilled are: $|DTS \cap TTS| \geq 10$ and $\frac{|DTS \cap TTS|}{|TTS|} \geq 0.5$. If a node fulfills both conditions it is saved as the last valid node. This step is performed level by level on the decision tree following the descendants of the last valid node until no nodes fulfill the two conditions. The last node which does is considered to be the most common orbit for that specific measurement or instrument technology/type.

3. The most common orbit is then saved in a table in the database.

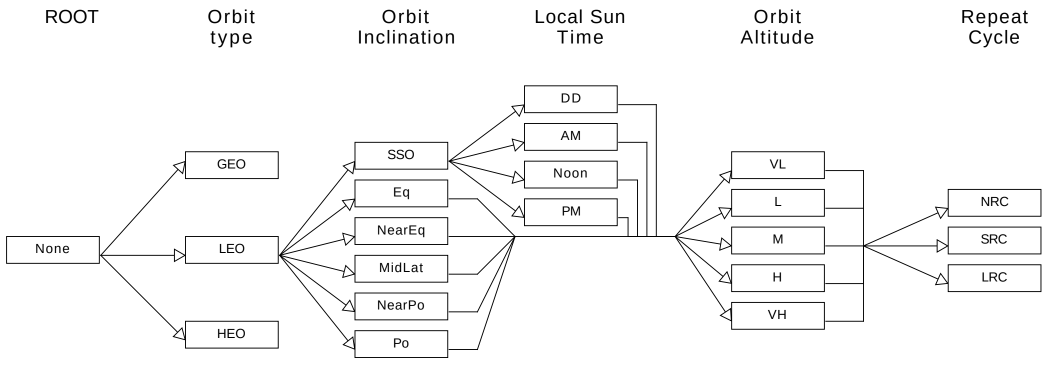


FIGURE 3.3: Decision tree for the data mining algorithm

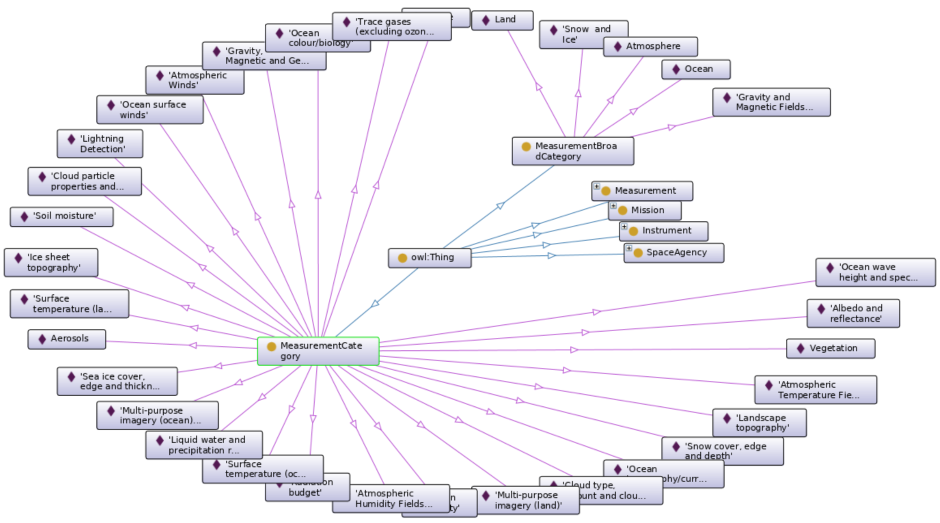
This is the only data mining process in the database for now, but if more were needed they could be added as well, using a similar structure to the one already described.

3.2.3.5 Ontology

An ontology, in this context, is nothing more than another kind of database, one where the basic data unit is a triple of Subject-Relation-Object. There is a whole field of philosophy dedicated to it, which tries to classify all the entities of the world into different categories and then relate them to others, be it through hierarchies or simple relations. Back to Computer Science, they can be encoded in many different formats, but one of the most common is the Resource Description Format, or RDF, and this is the one used in this project, along with RDFlib [98] for Python to handle it more easily.

In the beginning of the project, I thought I would use one of the multiple available open-source QA systems which work on RDF ontologies by querying them with SPARQL (SPARQL Protocol and RDF Query Language). This is the reason why a whole ontology which tries to replicate as much as possible the SQL database is being extracted together with the main database. At a certain point I realized that these systems are not yet powerful enough for the task at hand, so the ontology effort was halted, and this is the reason why not all the processed fields are in it, although all the textual ones along with all the relationships are present. An overall view of the ontology as is can be seen in Figure 3.4.

The data is obtained in the same way as in sub-subsection 3.2.3.2, with the only difference being that a whole other schema has been created with the classes of each object in the database and its relations.



'Trace gases (excluding ozon...
Land
'Snow and Ice'
Atmosphere
Ocean
'Gravity and Magnetic Fields...
'Ocean colour/biology'
'Gravity, Magnetic and Ge...
'Atmospheric Winds'
'Ocean surface winds'
'Lightning Detection'
'Cloud particle properties and...
'Soil moisture'
'Ice sheet topography'
'Surface temperature (la...
Aerosols
MeasurementBroadCategory
Measurement
Mission
Instrument
SpaceAgency
owl:Thing
MeasurementCategory
'Ocean wave height and spec...
'Albedo and reflectance'
Vegetation
'Atmospheric Temperature Fie...
'Landscape topography'
'Snow cover, edge and depth'
'Ocean ...phy/curr...
'Cloud type, ...unt and clou...
'Multi-purpose imagery (land)'
'Atmospheric Humidity Fields...
...budget'
'Surface temperature (oc...
'Liquid water and precipitation r...
'Multi-purpose imagery (ocean)...
'Sea ice cover, edge and thickn...

Figure 3.4: Small portion of the ontology

3.3 Back-ends

The back-ends in Daphne provide different functionalities which different *skills* may use to provide useful kinds of support to the systems engineer. There are three back-ends: the Architecture Evaluation back-end, which can be given designs and returns the value of their objective functions; the Data

Mining back-end, which can run machine learning algorithms on the dataset of designs for insight generation; and the QA System back-end, which processes questions, sends them to other back-ends or *skills* and gives the answer back in a human readable format, be it text, audio or images.

3.3.1 Architecture Evaluation

Architecture evaluation is one of those tasks in systems engineering which is very hard to systematize: it is always specific to the task each system must accomplish, and it is usually comprised of both objective and subjective measurements. This back-end uses the VASSAR methodology [7, 99], which is a systematic framework that allows automating the architecture evaluation process by combining both objective and subjective information.

There is no theoretical limit on how many numerical results can be obtained for the evaluation, but one has to keep in mind that humans are limited by us only being able to see 3 dimensions, and the whole point of the system is to reduce the cognitive load, so creating hard to understand plots is clearly something to avoid. 3D plots and 4D plots which are done by the use of colors can be understood by humans, as well as others such as parallel coordinates plots, scatter plot matrices, and radar charts. Daphne can handle any number of results with all these different plots, but in the design problem being handled right now only two results are needed: the science benefit and the cost, which makes the use of a simple 2D scatter plot enough.

Daphne is a CA made to help design EOSS, so it makes sense that there is a way to evaluate how good a satellite system design is. This evaluation system is described in [94], and it uses the VASSAR methodology to compute both the science benefit and the cost of each system design being studied. It works this way: starting with a given assignment of instruments to orbits, a set of expert rules is executed on a rule engine (JESS). These rules are used to first compute the measurement capabilities of a given architecture of satellite systems. Then the capabilities are compared with the measurement objectives set by the stakeholders. Estimates of the science benefit for each design are produced by computing how many objectives are being satisfied either fully or partially (at different levels of performance degradation). Estimates of lifecycle cost including payload, bus, launch, operations, and overhead cost are obtained by using cost estimating relationships and a simple spacecraft design algorithm providing rough mass, volume and power budgets for each spacecraft.

Thus, this back-end works by receiving a system design and returning a pair of floating point numbers which represent the science benefit and the cost for that design. VASSAR was developed by Daniel Selva, my advisor, and the communication layer between VASSAR and Daphne has been coded by Hyunseung Bang.

VASSAR is coded in Java and JESS, so a communication layer is needed to connect to the Daphne Brain. This layer is Thrift [100], a software developed at Facebook and then at the Apache Software Foundation to provide inter-language and process communication.

3.3.2 Data Mining

The main task of Daphne is to help the systems engineer perform better at creating good designs, and one really insightful way to do so is to give him/her a set of features which are present on most of the best designs found by algorithms so he/she can use all these to construct their own designs.

The way Daphne does it is using the Data Mining back-end, which runs machine learning algorithms on the Design Solutions database to obtain a set of logical rules which describe “good” features, meaning the ones which appear frequently in the Pareto front of the dataset. One example can be that perhaps most solutions currently on the Pareto front have two instruments together on the same orbit, or the other way around, never together in the same orbit.

The data mining method used is called association rule mining [101], and the algorithm used to extract the rules is called the Apriori algorithm [102]. Association rule mining extracts knowledge by creating logical rules which describe parts of a dataset. The major advantage of creating categories based on logical rules is that they are very similar to how humans structure knowledge and are therefore easy to understand [45]. These logical rules can also be called driving features, and in Daphne they are defined as combinations of design variables through logical operators such as AND, OR or NOT that appear to drive designs to a desired (target) region of the tradespace [8]. The whole idea is that if a design has some of these driving features, the probability of it being in the desired region of objective values (cost and science benefit) is also higher.

The back-end works by receiving a target region of the dataset, where that region can take any shape. Then the data mining algorithm is run on that region, returning a list of the most significant driving features in that subspace. The significance of a feature is measured by two confidences. Confidence is a measure of the statistical relation between a feature and the target region. One of the two confidences is computed as the ratio between the designs with a certain feature inside the target region and the whole target region, while the other is computed as the ratio between the designs with a certain feature inside the target region and the whole set of designs with the feature. The first confidence measures is the feature is general enough in the target region, while the seconds measures if it is specific enough to that region. Only if both confidences are high enough a feature is considered important enough to be significant for that region. This whole back-end has been developed by Hyunseung Bang in Java. A port of the back-end to Python is being developed by Amrit Kwatra.

3.3.3 QA System

The QA System is one of the key differences of Daphne compared to other decision support tools available to system engineers. Its goal is to answer diverse questions from the user. It helps in creating a more intuitive interface for Daphne, by giving the users a natural language interface, which is cited as being a better interface with a computer compared to creating code to develop charts, making database queries, or just using a mouse, which are the most common ways to interact with a decision support tool [103, 104]. This back-end has been developed by me.

There is usually a trade-off between accuracy and the number of questions answered in a QA system: some systems choose to answer only a small set of questions with great accuracy while others decide to answer as much questions as possible, with the result of lowered accuracy. This can be seen both in open-domain solutions like Aqqu [68] and YodaQA [26], which have already been discussed in chapter 2. The first has a lower recall (number of questions answered) and a high accuracy (number of correctly answered questions) and the second exhibits the opposite behavior over general datasets such as WebQuestions [105]. Domain-specific systems like PRECISE [106], which is basically a translator from natural language to SQL queries, work over a much smaller set of questions. As a result, it shows a much higher accuracy for those questions it can answer, but get a low recall from ignoring the rest of them. Because Daphne has a well-defined application domain, it can be assumed that the types of questions that the user will ask are limited as well. Therefore, the objective is achieving high accuracy rather than high recall by implementing the QA System in a similar fashion to PRECISE for generating database queries from natural language questions.

This back-end receives natural language sentences as input. For now, the language of these sentences is English. The output can be varied, as it can range from another sentence to some action happening in the screen or the physical embodiment, as well as images, plots and videos.

The sentences are classified according to their intent, which represents the objective the user wants the system to accomplish by speaking that sentence. To classify the intent of the natural language queries, a deep learning model based on a Convolutional Neural Network (CNN) [78] has been chosen over simpler methods such as regular expressions because of its greater tolerance to input variability and the lower difficulty of adding new intents to the set. This model, trained with different parameters each time, is used to classify the input utterance (another word for a natural language sentence, and the most common in the literature) into different categories at two different points in the pipeline: first to decide the intent of the sentence and later, if the sentence is a question, to classify the type of the question. There is also a spelling corrector, which uses Sellers' algorithm [107] to search for close words, correcting wrongly spelled commands and questions so they make sense to the different back-ends and skills which need to act based on the input. A longer discussion on the choices of algorithms and the pipeline can be seen throughout chapter 4, which explains everything in detail coupled with an actual example of a skill using the system.

The functional flow of the QA System is as follows:

1. The input question or command is obtained as either a typed or spoken

sentence. The system assumes that the recognized sentence by the voice Speech To Text (STT) system is correct and will try to find errors later.

2. The CNN model is used to classify questions and commands into different categories based on what module is responsible for the response.
   (a) If the input sentence is a command for any of the back-ends, it is executed on that back-end and the results are sent back to the user. These results can be from any of the back-ends already explained, and they can take different forms: be it JSON files, text messages, audio clips, plots, etc.
   (b) If it is a question, the QA System tries to find a skill which can give an answer to it, and in case there is a match it is sent there. The question goes through a full QA pipeline as that in Chapter 4 which results in an answer which can, again, take a variety of forms: text, audio or multimedia content.

3.4 Daphne Brain

The Daphne Brain is the key piece of the whole architecture that ties everything together: its job is to forward the user requests in all the different front-ends to the required back-end, be it voice commands, a click on a button or just some textual instructions. Its job also includes returning the response from the back-end to all active front-ends. This job is performed through the *skills* system, as will be detailed below. It has been developed by me.

User requests are translated in the front-ends to either pure, classical client-server HTTP requests like GET, POST, PUT or DELETE; or as a real-time connection implemented over WebSockets. This is usually a job for a web server, and this is exactly what the Daphne Brain is. It has been implemented using Django [108], a popular Python framework for creating web and REST servers, using different extensions to it like Django REST Framework [109] for easy API creation and Channels to leverage advanced features used in some front-ends like WebSockets and HTTP/2. REST APIs are the standard way to program APIs for web applications, and they consist on a set of what are called end-points, which are URLs which receive an input and produce and output. It is, in a way, similar to how a normal library API works, but the big difference is each end-point can be on different machines, making the system much more easily scalable.

As a critical piece of software, it is very important for it not to fail, for it can be considered a crucial point of failure: the system becomes unresponsive under a Brain error. This is why it is implemented as 4 different processes which are clones and will share the incoming requests between themselves, effectively eliminating the threat of a single critical process failing.

The Brain has a basic API implemented: the Daphne Commands API, which consists of two end-points: `GET /api/daphne/commands`, which returns all the voice and textual commands accepted by Daphne; and `POST /api/daphne/command`, which receives a command and either defers it to one of the skills or executes it directly if it is a general Daphne command. This API acts as a central receiver for all the voice and natural language communications, and it uses an instance of the QA System (subsection 3.3.3) to classify the commands between those of a skill and those which are general.

The rest of the requests received by Daphne are handled by the *skills*. Each skill is nothing more than a set of REST and WebSockets end-points which live inside the Daphne Brain and has access to all the back-ends and data sources. This gives a lot of flexibility when implementing them, as anything can be done inside the end-point handler: back-ends can be called, data sources can be accessed, and computations can be performed. At the same time, having this flexibility means it is even more important to have the Brain be resilient to failures, as they are bound to happen. Some skills are described in the next section.

3.5 Skills

Skills, as called by most CA, are no more than computer programs which leverage all the functionalities available to the programmer to perform the actual useful tasks to the end user, which in this case is the systems engineer.

In the case of Daphne, as already explained, they are implemented as independent sets of APIs inside the Daphne Brain so they can access all the back-ends and data sources and communicate back to the front-ends. They also have a unique Graphical Interface, although this is something which will not be happening in future versions of Daphne, where all the skills will be integrated into a Unified UI.

The next few subsections describe the two skills already present in the system which have not been developed by me: iFEED, which provides all the functionalities of [8] and the Critic, which given a design can give back suggestions on how to improve it. Chapter 4 describes the Historian skill, which has been developed by me. All the skills receive and send data in the JSON format.

3.5.1 iFEED Skill

iFEED is an interactive tool to support a systems engineer in the task of mining the dataset of design solutions for features describing those designs which are in regions of interest, for example the Pareto front. The job of this skill is to provide access to all those functionalities inside Daphne which are required to perform this mining task. Everything in this skill has been developed by Hyunseung Bang.

This Skill has a lot of end-points, separated on different groups depending on which back-end they are accessing. There are end-points for the Architecture Evaluation back-end, the Data Mining back-end and some for generic iFEED functionalities, as iFEED needs to access all of them to be able to perform its functionalities.

There are four end-points to access the Architecture Evaluation back-end:

- `POST /api/vassar/get-orbit-list`: This endpoint returns a list of the different orbits in the design space.
- `POST /api/vassar/get-instrument-list`: This endpoint returns a list of the different instruments which can be assigned to the different orbits of the architecture.
- `POST /api/vassar/evaluate-architecture`: This endpoint returns a pair of doubles with the values of both the cost and the science benefit of an architecture it receives through the POST body.
- `POST /api/vassar/initialize-jess`: This API needs to run once in the beginning before all the others so the system is prepared to evaluate all the different architectures.

A single API endpoint is available for the Data Mining back-end, `POST /api/data-mining/get-driving-features`, which receives a set of bounds to the dataset being studied which are sent to the Data Mining back-end so the driving features of that subset can be extracted using the algorithms described in [8].

Finally, there are five different generic iFEED end-points, but three of them are already deprecated. The two useful end-points are:

- `POST /api/ifeed/venn-diagram-distance`: This endpoint receives three areas: two of circles and one of the intersection between them. It returns the distance between the centers of the circles so that the intersection area is the one requested. It is used in the iFEED front-end for data analysis.
- `POST /api/ifeed/apply-feature-expression`: This endpoint applies the feature it receives as input (as described in subsection 3.3.2) to the iFEED GUI.

3.5.2 Critic Skill

Criticizing a design is a very specialized task in systems engineering: for a criticism to be useful it has to provide constructive ideas on how to improve the design, and these once again depend on the definition of a good design, which is unique to every system. This means a lot of work is needed to create a good critic, and Daphne is no exception. The bulk of the work in this skill has been carried by Arnau Prat, so this section will be a summary of his work.

The Critic skill receives a system design as an input, and its output is feedback to the user about the strengths and weaknesses of that design, along with specific suggestions to the user about how to improve a given architecture. It only has one single end-point, called `criticize`, and it runs as a WebSockets endpoint, which receives an architecture and returns the list of criticisms for it.

The Critic includes four different agents which create different kinds of feedback and suggestions: the Expert, the Historian, the Analyst and the Explorer. All of these are explained in the following paragraphs.

The Expert agent uses known design rules written by experts in earth observing satellites system design. The rules can be considered as basic design principles or heuristics that domain experts use for designing good systems. One example in Daphne use case is two instruments using the same frequency for radio transmission should not be used in the same spacecraft if at least one of them is active. All these rules are stored in the Expert Knowledge database, described in subsection 3.2.2.

The Historian agent (not to be confused with the Historian Skill) uses the historical database (subsection 3.2.3) to come out with a similarity score of the current design compared to past, successful, missions. The similarity is computed by checking how close the instrument configurations and the orbits are. The agent will then return the closest missions or say that none match. The motivation for this approach is similar to that of the case-based reasoning [110], which is one of the popular reasoning methods used in artificial intelligence. If a mission proposed by the user is very similar to many missions in the database, chances are that this a good or at least safe (well-known) option. On the other hand, if there are no similar missions in the database, there is a higher risk, as the mission appears to be a one-of-a-kind mission with little heritage.

The Analyst agent uses the last dataset available, the Design Solutions database (subsection 3.2.1), to notify the user if a given design shares some of the driving features usually found among good designs so far. A driving feature can be any combination of design variables, such as having an instrument in an orbit or having two instruments together. Having good features is no assurance of success, but it can be helpful. Daphne can also use this information to suggest changes to the systems engineer, based on known to be good features.

Finally, the Explorer agent runs a few optimization passes on the user design and tells the engineer if it has found some better design in the close neighborhood of the trade-space.

All these agents are coded in a combination of Java and Python, and the communication between themselves and with the Daphne Brain is either done directly for the Python modules or through JPype [111] for the Java ones. Thrift cannot be used here as everything is run in the same process, while Thrift demands everything to be run under different processes.

3.6 Front-ends

Daphne has 3 different kinds of front-ends: the Web & Voice Visual Interfaces, which serve as the main interfaces of the system as they provide access to all the different skills; Daphne VR, a VR interface which is used as both a demonstration of what could be done and as a part of an experiment; and the Physical Embodiment, which is a robot with a screen and camera which provides a different kind of interaction with the system.

3.6.1 Web & Voice Visual Interfaces

There are 3 different web & voice visual interfaces, one for each skill that has been developed: the iFEED Interface, the Critic Interface and the Historian interface.

3.6.1.1 iFEED Interface

The iFEED interface is thoroughly described in [8], as it is part of the research being done by Hyunseung Bang for his PhD thesis.

It allows the user to work on a data-set of design solutions by finding features which describe regions of interest of the data-set, as these features can be hints of good designs if the region of interest contains interesting designs for the system engineer. Thus, it provides tools to select arbitrary regions of interest, along with a design inspector which gives information on a certain design using the functionalities from the back-end in subsection 3.3.1, a feature (filter) creator, and a data mining interface which allows for features to be built by the computer by using the back-end described in 3.3.2. All the designs for the scatter plot are obtained from the Design Solutions Database (subsection 3.2.1).

A screen-shot of this interface can be seen in figure 3.5, with the whole data-set being shown along with a region of interest and the mined features below.

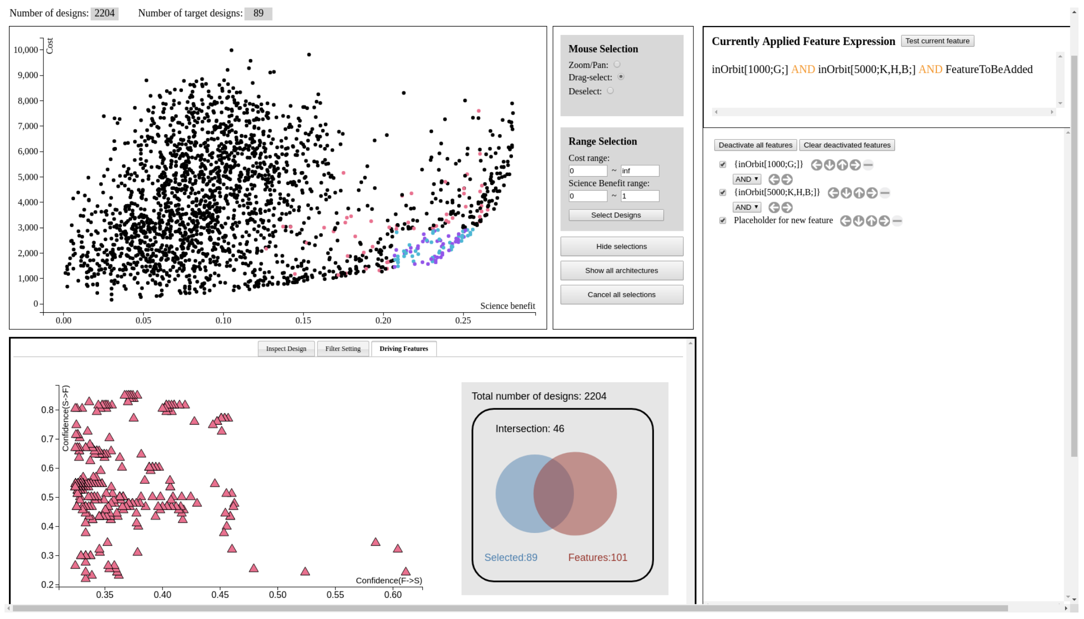
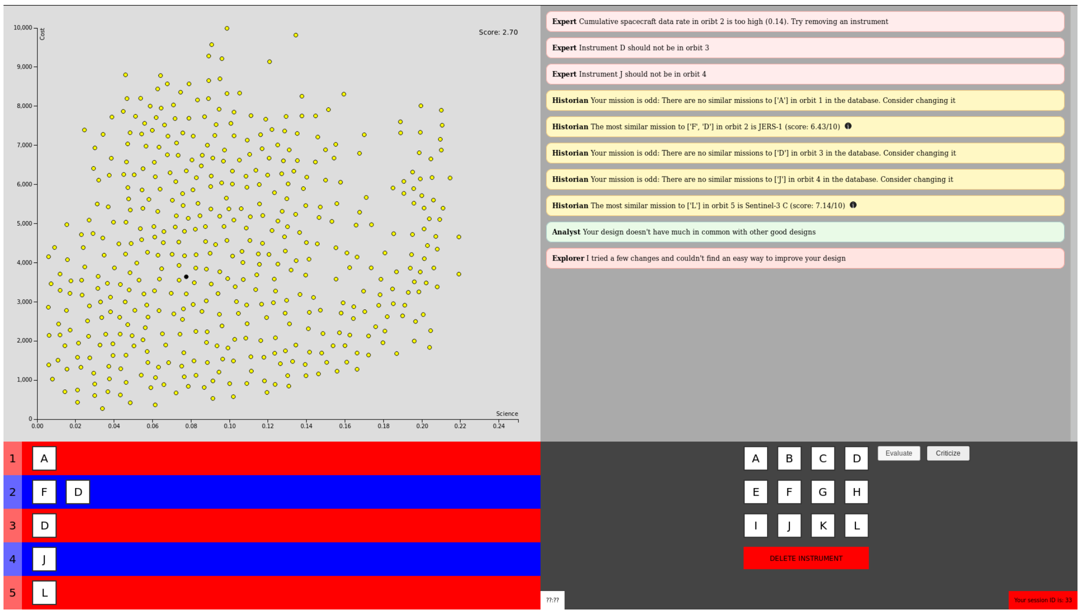


FIGURE 3.5: Screen-shot of the iFEED Interface

3.6.1.2 Critic Interface

The Critic Interface is designed to allow the user to create, evaluate and get feedback about any design (i.e. satellite constellation). It has been developed by Arnau Prat.



Expert Cumulative spacecraft data rate in oribt 2 is too high (0.14). Try removing an instrument

Expert Instrument D should not be in orbit 3

Expert Instrument J should not be in orbit 4

Historian Your mission is odd: There are no similar missions to ['A'] in orbit 1 in the database. Consider changing it

Historian The most similar mission to ['F', 'D'] in orbit 2 is JERS-1 (score: 6.43/10)

Historian Your mission is odd: There are no similar missions to ['D'] in orbit 3 in the database. Consider changing it

Historian Your mission is odd: There are no similar missions to ['J'] in orbit 4 in the database. Consider changing it

Historian The most similar mission to ['L'] in orbit 5 is Sentinel-3 C (score: 7.14/10)

Analyst Your design doesn't have much in common with other good designs

Explorer I tried a few changes and couldn't find an easy way to improve your design

Figure 3.6: Screen-shot of the Critic Interface

As it can be seen in figure 3.6, the interface consists of 4 different spaces, which take the whole screen space: the top left corner is taken by a scatter plot representation of the design trade-space (meaning a graphical representation of different designs along with their cost and perceived value or science benefit). The top right corner has information produced by the Critic skill. The bottom left one is the design building area, with a blocks in rows representation of the design being studied at the moment. Finally, the bottom right corner contains all the different building blocks to create a design: a creator of each type of instrument and a button to delete any of them. This area also contains buttons to call the Critic skill and the Architecture Evaluation back-end.

The user interaction flow for this interface is as follows: the user first creates a design by dragging blocks to the 5 different rows and/or removing them if needed. When the user is happy with the design, s/he can evaluate it to see where it lays in the trade-space. Finally, s/he can ask the Critic Skill to suggest improvements or point out weaknesses of the current design, which are printed in the space reserved for them.

The interface also includes a tutorial that fires up the first time it is used to explain all this work-flow to potential users. It also supports the running of experiments with human subjects by showing and hiding different parts of the interface so different treatment groups have access to different versions of the interface.

This interface is written in HTML, CSS and Javascript, using Tether [112] to position the elements in the web, Shepherd [113] to create the interactive tutorial for the application, D3.js [114] to draw the scatter plot and jQuery [115] for miscellaneous tasks like Ajax.

3.6.1.3 Historian Interface

The Historian interface has been developed by me as a placeholder until the unified Daphne interface is ready.

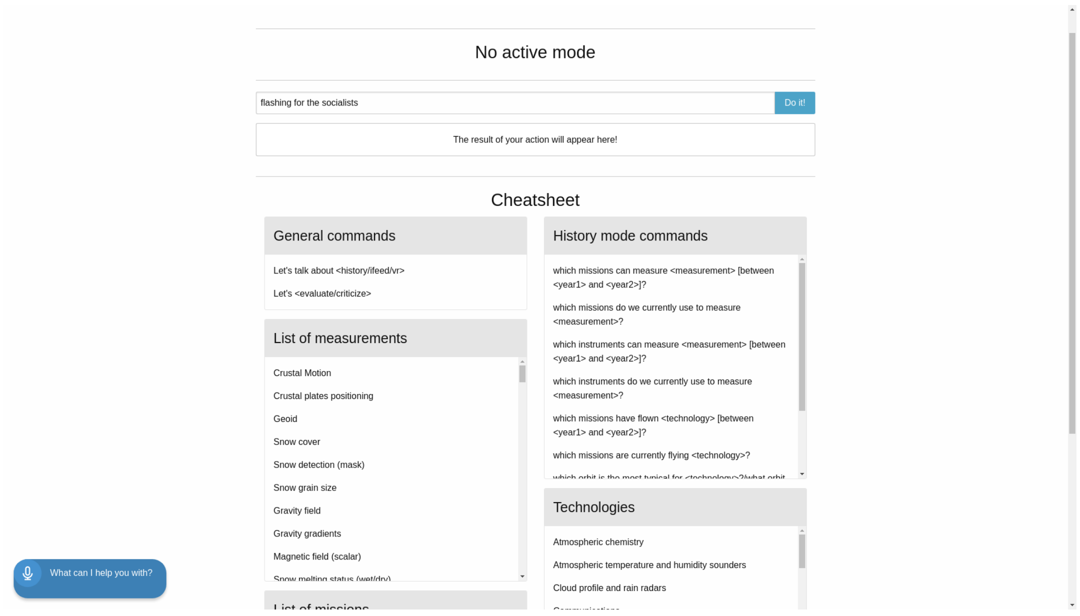
It has a simple text field to type the historical questions the user can ask, along with a block of white space where the response can be read.

Below all of that there are different cards with information on the questions which can be asked from the system and different lists of values for stuff such as missions, measurements or instrument technologies and types.

This interface also supports voice input which is mirrored in the text field automatically, and it has voice output as well. Both features are implemented by the annyang! [116] and the ResponsiveVoice.JS [117] libraries, respectively.

The rest of the interface is built with HTML, CSS and Javascript, with the help of Foundation 6 [118] and jQuery.

A screen-shot of it is shown in figure 3.7.



No active mode

flashing for the socialists | Do it!

The result of your action will appear here!

Cheatsheet

General commands

Let's talk about <history/ifeed/vr>

Let's <evaluate/criticize>

List of measurements

Crustal Motion

Crustal plates positioning

Geoid

Snow cover

Snow detection (mask)

Snow grain size

Gravity field

Gravity gradients

Magnetic field (scalar)

Snow melting status (wet/dry)

List of missions

History mode commands

which missions can measure <measurement> [between <year1> and <year2>]?

which missions do we currently use to measure <measurement>?

which instruments can measure <measurement> [between <year1> and <year2>]?

which instruments do we currently use to measure <measurement>?

which missions have flown <technology> [between <year1> and <year2>]?

which missions are currently flying <technology>?

which orbit is the most typical for <technology>?/what orbit

Technologies

Atmospheric chemistry

Atmospheric temperature and humidity sounders

Cloud profile and rain radars

Communications

What can I help you with?

Figure 3.7: Historian temporal interface

3.6.2 Daphne VR

The Daphne VR interface aims to be an alternative to the Web & Voice Visual Interface and its objective is to explore the benefits of working in a VR environment compared to the traditional desktop experience. It has been developed by Arnau Prat.

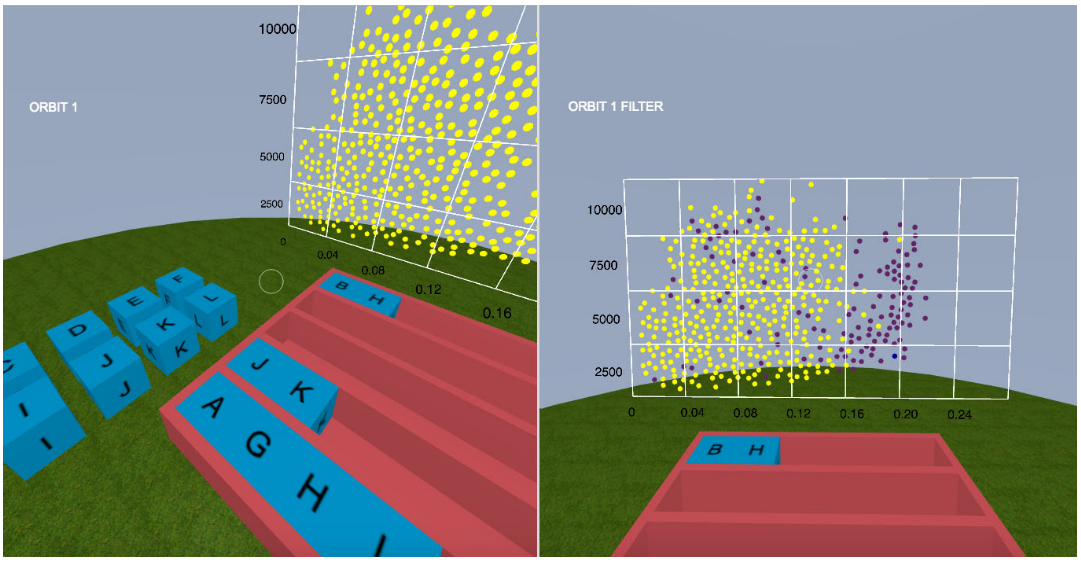


Figure 3.8: Views of the VR interface

As can be seen in figure 3.8, the interface resembles a screen, which is used to show the same graphic that appears in the Critic interface, with a few shelves below, which are the physical representation of orbits in this case. The blocks inside each shelf represent instruments which are assigned to the spacecraft which goes in that orbit. The other component of the interface that can be seen is a collection of all the block representation of instruments.

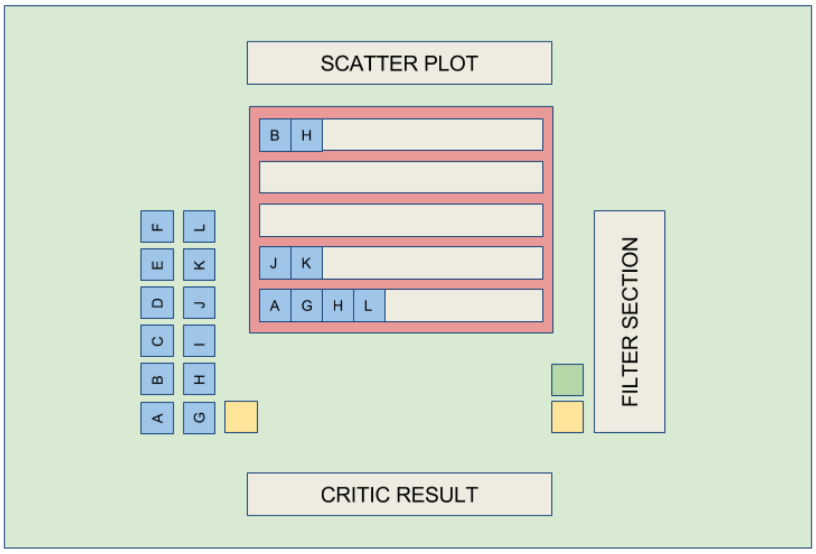


FIGURE 3.9: Design of the VR interface in planar shape

The interface has two more interactive elements, as seen in figure 3.9: the filter selection area, which is used to emulate some of the filters which can be used on the dataset in the web interface, and the Critic result space, in which the results from a call to the Critic Skill are shown so the user can improve their design.

The user interacts with this interface by looking around, centering the view on an interactive portion and clicking on it, either by using a mouse, a keyboard or action buttons on VR headsets. A few examples of these interactions are detailed in the following paragraphs.

To add an instrument to an orbit, 2 steps need to happen. First, the orbit in which the user wants to place the instrument needs to be selected by looking at the chosen shelf and then selecting it. Finally, the user needs to look at the instrument he/she wants to add in the section of the interface with all of them and select it. This will effectively add the instrument to the wanted orbit.

To delete an instrument, the user has to select the bin block close to all the instruments and then keep selecting the instruments s/he wants to remove.

To update the plot shown in the screen after changing the instrument assignation, the user must select the “Update” prompt which appears when changing anything in the shelves.

To get the information from the Critic Skill, the user must select the “Criticize” prompt on the back of its head inside the interface.

The filter system can be turned on and off by a block in the filtering system of the interface. The different filters available in this interface include the “Instrument in any orbit”, “Instrument in orbit N”, “Instruments together”, and “Instruments separated”. All of them work in a similar fashion to the “Add instrument to orbit” filter: the user first selects the filter s/he wants to use and then keeps selecting the instruments to put in that filter.

This interface is written in Three.js [119], a 3D library for Javascript which is used as a high level abstraction over all 3D web technologies like WebGL [120] and WebVR [121]. The system can only work in Google Chrome in VR mode, as it is the only web browser which supports this feature as of September 2017. That being said, the 3D scene can be seen in any modern web browser. It is important to note that the end user needs to have access to a Google Cardboard, Samsung GearVR or a similar headset in order to enjoy the full Virtual Reality experience. Apart from a mobile VR headset the user does not require any external controllers or equipment.

3.6.3 Physical Embodiment

The design of Daphne's physical embodiment is presented in figure 3.10. This robot is built using 3D printing technology and its design is open-source (a link to the source files is provided in the appendix chapter A), meaning anyone is able to build their own version of it. The design has also been done by Arnau Prat.

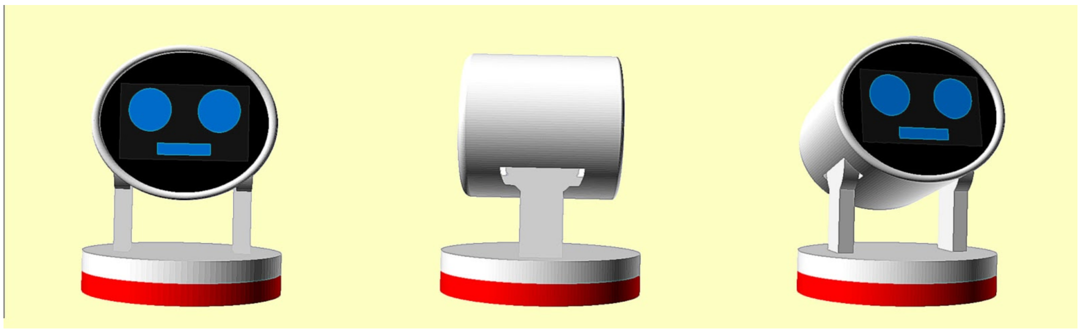


FIGURE 3.10: Design of the Daphne physical embodiment

The hardware architecture of the robot is presented in figure 3.11. Its main components are a Raspberry Pi, an Arduino, a 7" LCD screen, a camera, a microphone, a speaker, two servo motors and a normal rotation motor.

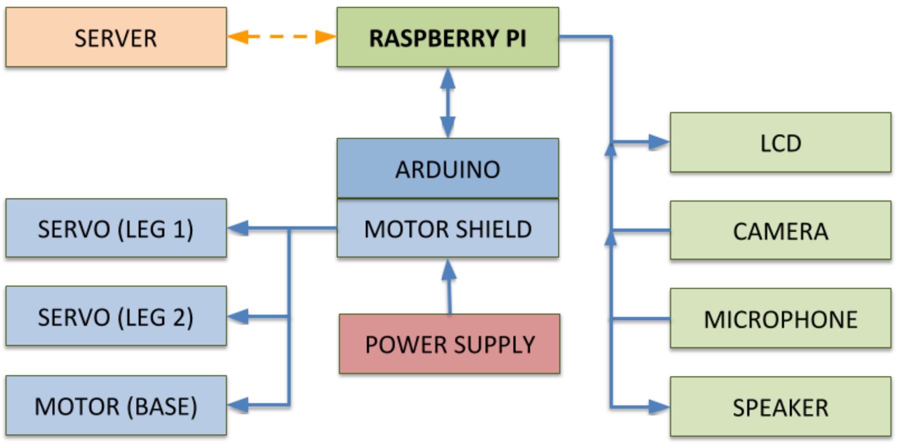


FIGURE 3.11: Hardware of the Daphne physical embodiment

The LCD screen is used to either show facial expressions or display any type of data, as for example images or graphs. The robot also has smooth, variable speed, pan and tilt moves. These features, together with the camera, are used to track the user, but also to increase its expressiveness, by giving it human gestures like nodding. The microphone and the speaker are used as the physical input and output for Daphne voice interaction. These kind of features might help improve the interaction and/or human performance in the task at hand. Finally, the Raspberry and the Arduino act as the brain and nerves of the robot, by driving the remaining functions.

To perform all the tasks a software architecture is needed, and it is described in figure 3.12.

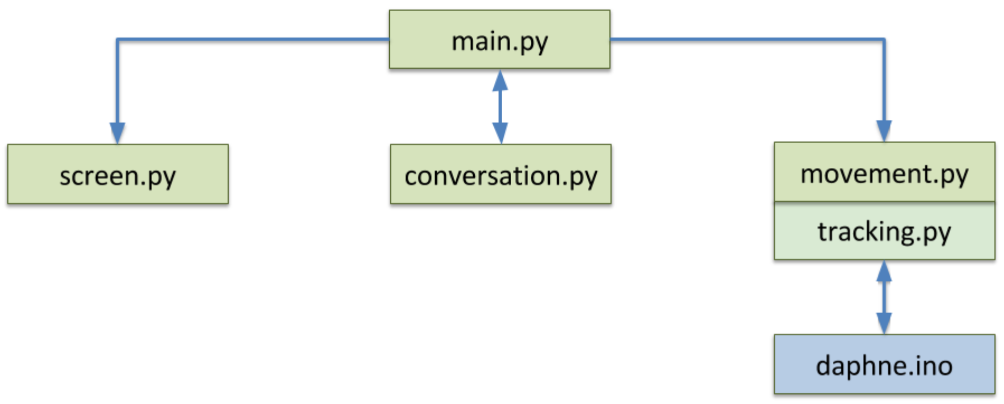


FIGURE 3.12: Software architecture of the Daphne physical embodiment

There is one module for moving the robot, which interacts with the Arduino, one for controlling the screen and one for the audio input and output. All these processes are controlled by the main process, which is also in charge of sending and receiving the results and responses from the Daphne Brain over the network. It is important to note that all these processes are running concurrently and are as non-blocking as possible, as this helps in believing the robot is more human-like by not having awkward pauses in the middle of interactions.

Chapter 4

Historian Skill

The Historian Skill has been developed by me, and because of this reason I can detail much more of its design philosophy, and this is also why it has a special chapter dedicated to it.

4.1 Skill overview

In the first days of Daphne, when the system was being designed, the team agreed on building a few demonstration skills to check if the system was being designed correctly by checking if the design included all the needed parts for each skill to work properly.

One of these skills was the Historian Skill, whose job was one which seemed really simple at the time: it had to answer questions about the historical database, which had to be scraped from the internet. Some example questions are: "which missions can measure <measurement>", "when was mission <mission> launched" or "which missions are currently flying <technology>". Both the scraping and the QA system were deemed as easy tasks, as there seemed to be readily available systems for both tasks which could do the job by simply adapting them.

I was wrong. No open-source software could do the whole job with simple adaptations, so in the end the development of the skill took almost all of my development time for the whole duration of the project, as can be seen in chapter 2 with most of the failed approaches related to the systems needed for this skill and this whole chapter dedicated to the development of the skill.

The final design of the historian skill is the following: it works as a Restricted Domain Question Answering (RDQA) system, meaning it can only answer a limited set of questions about a limited set of knowledge, which is fine for the use case of answering questions about Earth Observing Missions, as the set of questions which can be asked is pretty limited and the priority is getting correct answers over a lot of answers. It is implemented as a Natural Language Interface for a Database (NLIDB), as it basically parses the questions into database queries, but using state of the art techniques instead of more classical approaches such as regular expressions, syntax based systems or semantic grammar systems, for which examples can be seen at [76, 75, 77].

The skill receives a raw text as input and returns another raw text as output, to keep the interface of it as simple as possible. The processing includes preprocessing the raw text to obtain relevant information from it, then classifying the question into one of the answerable classes by using a CNN, and continuing with the extraction of the relevant data from the question using a custom algorithm based on approximate string matching. After all this, the database is queried and a response is constructed based on a template. This response is then sent back to the user.

With the story behind the skill out of the way, this chapter of the report explains in detail the final architecture of the Historian Skill: it begins by explaining how the natural language questions are processed so all the different subsystems inside the skill can use the text in a more useful way (section 4.2). It continues, in section 4.3, discussing how the question classifier has been programmed. Right after that, section 4.4 describes the configuration files format for a question type. Section 4.5 explains how the relevant features are extracted from each question. Next, section 4.6 explains a process done to increase the range of questions that can be asked by adding more information from the user context. Then, section 4.7 describes the process to create the database query based on the question type. Finally, section 4.8 discusses how a natural language response is created from all the gathered data and the database query results.

4.2 Text processing

Text processing of the raw input, which is the voice input as processed by the Speech To Text system or text from the input field of the interface, is done by leveraging the models in the spaCy Python library, whose mission is "to make cutting-edge NLP (Natural Language Processing) practical and commonly available" [122].

The decision to use spaCy instead of more complex and complete solutions like NLTK or OpenNLP [123, 124] or even creating a custom solution in-house was taken after doing a comparison run between the three where it was seen that the needed functionality existed in all three solutions and spaCy was the only system that provided a fast enough computation to make the system work in almost real-time. It must be noted that one of spaCy goals is to provide fast algorithms, so it makes sense that it is actually faster than the other two solutions, whose main objectives are completeness and reconfigurability rather than raw speed.

The very simple interface of spaCy had the potential of turning into a pain point as Daphne might have needed more than it could deliver in terms of customization, but these fears have been dissipated with the complete system, for which spaCy provides more than enough functionality.

This means the raw string is processed by a single call to a function provided by the spaCy API, and this turns the raw text into a spaCy document, with all the words tokenized, with part-of-speech tagging, sentence segmentation, dependency parsing, entity recognition and integrated word vectors.

For every token (word/number/space/punctuation) a lot of information is provided, such as its lemma (the word you would find in a dictionary), a lower-case version of the word, all its prefixes and suffixes, if it is a number or a real word, if it looks like common computer related strings (emails and urls), its part-of-speech tag, its syntactic dependencies and even the perceived sentiment, among other features.

Not all this information is needed for the Historian skill, but some of it, like the lemmas, the classifications and the lower-case versions come in handy for most of the pipeline.

4.3 Question classifier

With all the information about the question already obtained, the next step in the skill pipeline is classifying the question into a type of those which can be answered by the skill.

Classifying a text into a set of different classes is one of the classic Natural Language Processing problems, due to its various uses, including news classifiers, sentiment analysis, spam detectors, language identification and others, such as the use case of our Historian skill, where every question is assigned a type which defines the query to the database and which data will need to be extracted from the question in order to successfully perform the query.

Text (sometimes called document) classification has been studied since the dawn of writing: every library in the world has a class (or topic) ordering of all its books, and the main problem for most of history has been to decide which is the "correct" set of classes. It has been, until the last 50 years, a mostly human task, performed by people who specialize in classifying texts. In the last 50 years, the types of texts being classified have changed a lot, and the content being produced has grown exponentially, prompting researchers and companies to invest money and time researching algorithms and AI techniques which classify sets of texts into sets of categories or classes, as it is no longer feasible to have humans do most of these tasks.

As this is a recurrent problem when dealing with text, a lot of algorithms and techniques have been developed since the dawn of computing [125]: rule-based systems, naïve bayes, k-nearest neighbors, support vector machines, decision trees, logistic regressions and, for the last 3-4 years, neural networks, of the convolutional and recurrent varieties. It is easy to see that almost all AI techniques have been applied to the problem, all with the goal of improving the accuracy of the last method used while striving to maintain or reduce the amount of work needed to train the algorithm.

Focusing on neural networks, the state of the art evolved from using the relatively simple CNNs [126], to augmenting them [78] with word embeddings such as word2vec [127] or GloVe [128] to using Recurrent Neural Networks (RNNs) with LSTM [129] and attention mechanisms [81].

When deciding what algorithm to use, I tried to strive for a balance between accuracy, training performance and evaluation performance, as well as implementation ease, while being as close to the state of the art as possible. As seen in [81], the accuracy of the neural networks is usually higher than all other algorithms before them. But training RNNs takes much longer than training CNNs, due to the more sequential nature of RNNs over CNNs, which makes

RNN much harder to parallelize. This difference in training time combined with a less acute loss of accuracy ended up tipping the balance in favor of the CNN with word embeddings model in [78]. Its implementation has been relatively fast to train and for the task at hand its accuracy has been beyond my expectations, as it is very hard to find sentences which are misclassified. The accuracy for the question classifier for 10 question types is around 95% using cross-validation, which is pretty high compared to other systems in the literature. This might be due to overfitting or the problem being easier than most, but there is not enough evidence to confirm any of the two hypothesis.

The implementation of the algorithm is based on [130], with all the due changes to adapt the system to a multi-class classifier as needed by the historian skill, for example by changing the input loader and adding support for a multi-class output. The whole subsystem is implemented in Tensorflow [131], a Python library developed by Google to simplify the development of different machine learning workflows. The training component lives in the historical_db repository and the actual classifier is running inside the Historian Skill API on the Daphne Brain. It is also important to note the hyper-parameter tuning done to the model, with the size of the word filters set to 3, 4 and 5 words and the number of filters per word size set to 100. According to [79], these values give consistently good results on different datasets.

Returning to the input, it is known that machine learning algorithms, and specially neural networks, need big datasets to be able to perform with great accuracy. The problem is in the span of this project there was no time to collect a big enough dataset of questions which could be asked, so a compromise solution was found: use a question generator which can add random variations of questions of the same type, with even random words put in the middle which make no sense so the algorithm can be trained on a quasi-human input.

For the actual training, a set of 2190 examples is generated, with different numbers for each question depending on how many variations of the question there can be. Then, before they are fed into the neural network, they are cleaned by lemmatizing each word and then removing the so-called stop words: words which do not help in NLP tasks, and which are selected from a list already included in spaCy with some exceptions such as the wh- question words, which are really important in this classification task. Finally, the questions are fed into the neural network as a vector of word vectors computed with word2vec.

This approach has given good results, as the system testers have a hard time finding questions that can fool the neural network into misclassifying a question, while at the same time keeping the time to add a new function relatively low, as the only two steps needed are generating a new set of examples with the random generator and retraining the network, which is already programmed to get the number of output classes from the number of files with examples which is provided to it. Training the network takes around 30 minutes on a Nvidia GTX 1050.

4.4 Definition of a Question Type

When programming the analyst back-end and the Historian skill, I realized that the system needed to be easy to expand, in the sense that adding new questions to the pool of answerable questions should be a process as easy as possible so anyone with some database knowledge could do it. This meant creating a system based on files which could be created and edited without any intervention from me. This software architecture is sometimes referred as data-driven design, as software systems are programmed to read information from data files, meaning they only need to be programmed once and can have totally different behaviors depending on the files read.

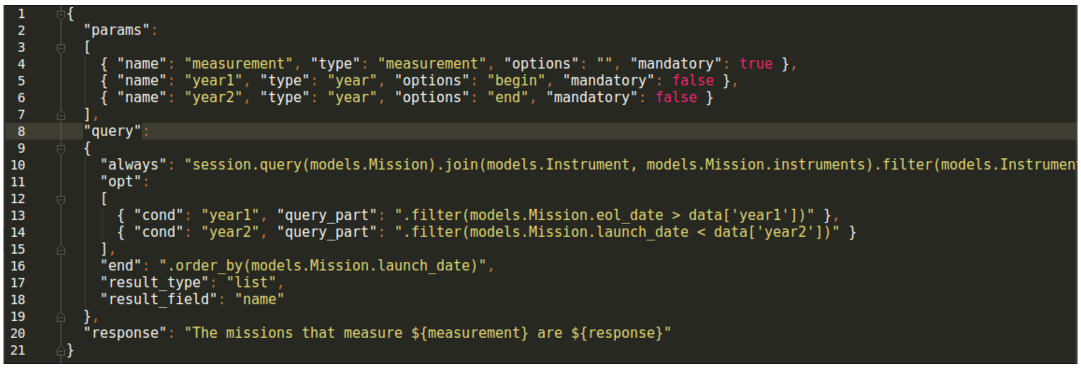
In Daphne, and the Historian skill in particular, these files are the definitions of the different question types, meaning each file describes a question type completely.

The information contained in each of these files is:

- **Parameters**: The first field of the file is an array containing all of the parameters that can be found in a question type, including a name for each parameter to refer to it later, the type of the parameter to know how to extract it later on, some extra data to be passed to the extract function, and whether the parameter is mandatory for the question type or not, as some question types allow for optional parameters.
- **Query**: The second field of the file contains the different building blocks (or templates) of the database query. It includes the "always" section, which is executed all the time, followed by an array of optional filters and sub-queries which depend on some of the optional parameters being present. Then there is the "end" portion, which is always appended at the end of the query. The last two sections are the "result_type" and the "result_field", which define if the result is a number, a text or a list, for example, and tell the system which field of all those returned by the database is the needed result. All the sections are written using the SQLAlchemy declarative syntax, which makes it much easier to add new parts to the query compared to the regular SQL syntax.
- **Response**: The last field of the file contains a Python template [132] on how to build the response which is going to be sent back to the user. This template allows to put any HTML on it, but as images or audio cannot be generated as a response for now, only text is usually written

here. All parameters along with the query result can be used in building the response.

These files are written in JSON format so they are lightweight and easy to understand, and an example is provided in figure 4.1.



```
{
  "params":
  [
    { "name": "measurement", "type": "measurement", "options": "", "mandatory": true },
    { "name": "year1", "type": "year", "options": "begin", "mandatory": false },
    { "name": "year2", "type": "year", "options": "end", "mandatory": false }
  ],
  "query":
  {
    "always": "session.query(models.Mission).join(models.Instrument, models.Mission.instruments).filter(models.Instrumen
    "opt":
    [
      { "cond": "year1", "query_part": ".filter(models.Mission.eol_date > data['year1'])" },
      { "cond": "year2", "query_part": ".filter(models.Mission.launch_date < data['year2'])" }
    ],
    "end": ".order_by(models.Mission.launch_date)",
    "result_type": "list",
    "result_field": "name"
  },
  "response": "The missions that measure ${measurement} are ${response}"
}
```

FIGURE 4.1: Example of a question type JSON file

4.5 Feature extraction

Once the skill and the back-end know which parameters need to be present in the question being processed, the next step is to actually extract them. The algorithm works as follows:

1. For each parameter, an extract function is called depending on the type of the parameter. These functions can be of three types: it can use the named entity recognition of spaCy to obtain the required features from the question if spaCy already implements the algorithm for that type of feature; it can be a sub-string matcher based on the lists of possible values for each parameter type being extracted from the database; or it can be just a sub-string search to see if there are sub-strings in the question which comply with certain conditions. For example, the years are extracted by checking if there is any sub-string that looks like a year according to spaCy and then saving all of them in order in case more

than one is needed. For missions, measurements and technologies the process is more complex, and is described in the following steps:

(a) First, a list with all the possible values is obtained from the database.

(b) Then, all the elements in the list are compared to the whole question using Sellers' algorithm [107], and the result is the same list, but with the elements ordered by the maximum similarity of each of them to the question. The maximum similarity is computed by checking the edit distance for each element against all the substrings with the same size in the question string and then computing a ratio with the following formula: $max\_similarity = max\left(\forall substrings \frac{length\_element - edit\_distance(element, substring)}{length\_element}\right)$. The list is then cropped to only the elements with a high enough max similarity (0.75 for now).

(c) The list is cropped to only the number of required elements of the same type if it is longer than that.

(d) The list is then reordered by the appearing position of each element in the question and then saved in that order for the rest of the pipeline.

2. Once the extraction is done, the list of features is passed through a process function which applies certain modifications to each feature depending on its type. For example, years are converted into specific dates depending on an optional parameter which is written in the JSON. Most other features remain roughly the same, with some minor adjustments like the elimination of spaces at the beginning and at the end of the string or some changes in the capitalization.

3. All the extracted and processed features are sent back to the main pipeline so they can continue their journey to the answer.

4.6 Data augmentation

Even with all the data collected from the question, sometimes people consider some information to be common knowledge and so it does not appear in the question. One clear example is the meaning of "now". While people know which is the date of today, the Historian skill will not unless someone tells him so. This is the reason why this step exists in the pipeline. Its job is to simply add some of this implicit contextual information humans have and expect the computer to have when interacting with it.

For now the system only adds the current date as contextual information, as this is the only extra information needed to answer all the programmed questions.

4.7 Database querying

The next step in the pipeline is querying the database to obtain the required information from it. The data needed in this step is all the augmented data from the question along with all the query templates already mentioned in section 4.4. There are a few sub-steps to it, which are described in the following list:

1. First, the "always" template is run through the Python template engine to obtain the first part of the query which is going to be run.
2. Then, for each "optional" template, the condition of activation is checked against all the available data, and if it evaluates to true then

the template is run through the engine and appended to the end of the query.

3. Finally, the "end" portion of the query is run through the engine and it is appended to the end of the query.

With the query fully constructed, it is run in the database and its results are obtained.

With the results in hand, the portion of the answer which depends on them is built. Responses can be of completely different types, and each type has a completely different process to build itself.

For example, if the response is a list, all the query results are appended in a string with separating commas. In case it is a date, the date is written in a human readable form, as the database stores it as a UNIX timestamp. The last implemented case is for orbits, as the orbit related questions have their answers stored in a format which can encode all the information from the decision tree in section 3.2.3.4, and this means a parser needed to be built in order to decode that information into a human readable format.

4.8 Answer construction

Finally, the last step in the pipeline is actually building the answer the user will see in his/her screen. The answer is built running the answer template corresponding with the question type on the template engine with all the augmented data from the question together with the response from the last step.

The result can sometimes sound awkward to a human, and future improvements are discussed in Chapter 5.

Chapter 5

Limitations and Future work

While developing the whole system, a lot of ideas were postponed as they were considered not basic for the project to actually be in a working state. Some difficult to solve problems were also identified, which are listed as limitations of the project.

5.1 Limitations

The three main limitations of the system as of now are the following:

- **Not testing the system**: There has been no formal test with humans to see if the system really accomplishes its intended benefits. There is anecdotal evidence that it does, but it cannot be considered hard evidence. This limitation is going to be solved soon after this report is finished, as some tests are planned as part of a journal paper to be published in the next months.
- **Scaling**: Adding a new question type or a command to the system implies retraining the whole statistical model for classification of the input, which takes time. Also, the process of actually adding and programming the question is hard, as it requires knowledge on the internal structure of the database, SQL and SQLAlchemy declarative syntax,

which is out of reach for most non-programmers. An improved workflow would help much in this, but it is difficult to develop and thus is really far away in the list of tasks to do.

- **Generalization**: The system is not thought out to be reused in other domains, even inside system design. A lot of the work done here has been specific to the task at hand, and it is not clear how much of the project can be generalized without almost starting form scratch. One idea is to make the whole QA pipeline a standalone project, with a command line interface which allows the end-user to only run a single command to obtain a completely functional QA server, but this requires time and more thought, which I do not have right now.

5.2 Future work

On a brighter note, these are features which can actually be developed in a short time frame and for which I have time and a plan:

- **Unified web interface**: Right now Daphne is running on three different web interfaces: iFEED, Critic and Historian. A new, unified interface with the functionality of all the other interfaces coupled together has already been planned and designed on paper, which means only the actual coding of it is left to do.
- **Better command classification**: The Daphne Voice and Text Command Classification is now implemented as a system of ifs and elses which changes a state machine so the commands are actually routed to the correct skill. The original idea was to reuse parts of the Historian skill so the commands are routed automatically using a statistical model instead of the solution implemented now, which is just a placeholder.

- **Dialog system**: The system should be able to maintain a conversation with the user, keeping the context of what has already been said and being able to answer follow-up questions from the user on questions asked before. One of the main steps forward for Daphne is to add some kind of cognitive architecture like those presented in section 1.1 behind it, so this project is already starting.
- **Visual answers**: The system only answers questions with textual responses, but there are no limitations on the kinds of output the system can emit. And some questions are better answered with plots, images or videos. Actually adding these kinds of responses is then something which can be worked out in the future.
- **More human answers**: No one wants to hear "Mission X was launched in 2030" when it is still 2017, as it might make you feel old. Building a more robust answer system which can actually output the correct verb tense when talking about dates among other common human details is a project which might be worth pursuing.

Chapter 6

Conclusion

The aim of this final degree thesis was to develop a fully functional CA for helping system engineers design EOSSs. This main objective has been fully completed with the development of the whole Daphne architecture, including all the front-ends, the Daphne Brain, the back-ends, the data sources, and, most importantly, the three demo skills: the Historian Skill, the iFEED Skill and the Critic Skill.

Until now, most CA have been developed for commercial general usage. But there has been a recent trend of specializing these tools, as the problem of cognitive overload of humans is becoming common in a lot of fields, with design and aerospace being two of them. While there have been CA developed for both aerospace and design, as seen in section 1.2, Daphne is the first CA to support the task of designing EOSS. It also tries to set a precedent by making the system completely open, which is something sorely missed in aerospace, where most technologies are completely locked down.

The work done on the QA pipeline for the Historian Skill is also interesting as it has combined the latest trends on NLP using Deep Learning to create a NLIDB which is more flexible and easier to configure than most existing systems, but can still be made easier if there is some more time investment.

Chapter 7

Personal thoughts

Through the development of the whole Daphne architecture along with the Historian skill I have had the chance of explaining in detail how the research and software building processes work with the failed approaches chapter and the two development chapters. I am specially happy with Chapter 2, as it tells a story which is usually hidden from the majority of people, who can only judge a project by its success or failure. I feel this has been said a lot of times, but every success is built on a mountain of failures, and this project is no exception.

I did not expect to have to work with a team in this project, but having had to do so has been a good experience at tackling what people sometimes call soft skills such as time management, scheduling, and communication with other people in a team: these are important for any career but are almost never taught, which is strange considering most career advancements depend on them.

As a final note, finishing this project and thesis brings a sense of closure for me, as, after more than five years of hard work doing two bachelor degrees at the same time, this is the end result of the whole effort. I could not be prouder of it, as I have been able to use a lot of knowledge I learned along the way in both my degrees and I feel this is like putting the icing on the very sweet cake that has been my experience for the years.

Appendix A

Source code of the project

The source code for the whole project can be found under different repositories in Github, under the `seakers` group: https://www.github.com/seakers.

- **Daphne Brain**: https://github.com/seakers/daphne_brain
- **iFEED**: https://github.com/seakers/iFEED
- **VASSAR**: https://github.com/seakers/VASSAR
- **Data Mining**: https://github.com/seakers/data-mining
- **VR Interface**: https://github.com/seakers/daphne-VR
- **Physical Embodiment**: https://github.com/seakers/daphne-robot
- **Historical DB**: https://github.com/seakers/historical_db
- **Daphne Current Interface**: https://github.com/seakers/daphne-visual
- **Daphne Unified Interface**: https://github.com/seakers/daphne-interface

References

[1] National Aeronautics and Space Administration. *NASA Technology Roadmaps: Introduction, Crosscutting Technologies, and Index*. Tech. rep. July. 2015.

[2] Daniel Selva et al. “Distributed and Federated Satellite Systems: What is Needed to Move Forward?” In: *Journal of Aerospace Information Systems* 14.August (2017), pp. 412–438. DOI: 10.2514/1.I010497.

[3] National Aeronautics and Space Administration. *NASA Technology Roadmaps TA 11: Modeling, Simulation, Information Technology, and Processing*. Tech. rep. May. 2015.

[4] Daniel Selva and Edward F. Crawley. “Integrated assessment of packaging architectures in earth observing programs”. In: *IEEE Aerospace Conference Proceedings*. 2010. DOI: 10.1109/AERO.2010.5446885.

[5] T. Spear and NASA. *NASA faster, better, cheaper task final report*. Tech. rep. Washington: NASA Headquarters, 2000, pp. 1–18.

[6] R. W. Kingsbury, D. O. Caplan, and K. L. Cahoy. “Implementation and validation of a CubeSat laser transmitter”. In: *Proceedings of SPIE*. Vol. 9739. 2016, pp. 1–9. DOI: 10.1117/12.2217990.

[7] Daniel Selva and Edward F. Crawley. “VASSAR: Value assessment of system architectures using rules”. In: *Aerospace Conference, 2013 IEEE*. IEEE, 2013, pp. 1–21. DOI: 10.1109/AERO.2013.6496936.

[8] Hyunseung Bang and Daniel Selva. “iFEED: Interactive Feature Extraction for Engineering Design”. In: *ASME 2016 International Design*

Engineering Technical Conferences and Computers and Information in Engineering Conference. 2016, pp. 1–11.

[9] Robert E Thompson et al. "Disaggregated Space System Concept Optimization: Model-Based Conceptual Design Methods". In: *Systems Engineering* 18.6 (2015), pp. 549–567. DOI: 10.1002/sys.21310.

[10] Sreeja Nag, Steven P Hughes, and Jacqueline Le Moigne. "Streamlining the Design Tradespace for Earth Imaging Constellations". In: *AIAA Space 2016*. 2016, pp. 1–17. DOI: 10.2514/6.2016-5561.

[11] Mark W Maier and Eberhardt Rechtin. *The Art of Systems Architecting*. 3rd ed. Boca Raton, FL, USA: CRC Press, 2009.

[12] Douglas C. Engelbart. *Augmenting human intellect: a conceptual framework*. Tech. rep. Washington D.C.: Stanford Research Institute, 1962, p. 144.

[13] Johann Hauswald et al. "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers". In: *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*. 2015, pp. 223–238. DOI: 10.1145/2694344.2694347.

[14] Johann Hauswald et al. "DjiNN and Tonic". In: *ACM SIGARCH Computer Architecture News* 43.3 (2015), pp. 27–40. DOI: 10.1145/2872887.2749472.

[15] Google. *Photos | Google Blog*. 2017. URL: https://www.blog.google/products/photos/ (visited on 09/10/2017).

[16] Gerhard Weiss. *Multiagent Systems*. 2nd ed. Cambridge, Massachusetts, USA: The MIT Press, 2013, p. 920.

[17] Karen Myers et al. "An intelligent personal assistant for task and time management". In: *AI Magazine* 28.2 (2007), pp. 47–62. DOI: 10.1609/aimag.v28i2.2039.

[18] John Grant, Sarit Kraus, and Donald Perlis. “A logic-based model of intention formation and action for multi-agent subcontracting”. In: *Artificial Intelligence* 163.2 (2005), pp. 163–201. DOI: 10.1016/j.artint.2004.11.003.

[19] Apple. *iOS - Siri - Apple*. 2017. URL: https://www.apple.com/ios/siri/ (visited on 09/10/2017).

[20] Google. *Google Assistant - Your own personal Google*. 2017. URL: https://assistant.google.com/ (visited on 09/10/2017).

[21] Amazon. *Alexa*. 2017. URL: https://developer.amazon.com/alexa (visited on 09/10/2017).

[22] Microsoft. *Cortana | Your Intelligent Virtual & Personal Assistant | Microsoft*. 2017. URL: https://www.microsoft.com/en-us/windows/cortana (visited on 09/10/2017).

[23] MIT Technology Review. *Facebook's Perfect, Impossible Chatbot - MIT Technology Review*. 2017. URL: https://www.technologyreview.com/s/604117/facebooks-perfect-impossible-chatbot/ (visited on 09/10/2017).

[24] D. A. Ferrucci. “Introduction to “This is Watson””. In: *IBM Journal of Research and Development* 56.3.4 (2012), pp. 1–15. DOI: 10.1147/JRD.2012.2184356.

[25] IBM. *Watson Products and Services - IBM Watson*. 2017. URL: https://www.ibm.com/watson/products-services/ (visited on 09/10/2017).

[26] Petr Baudiš. “YodaQA : A Modular Question Answering System Pipeline”. In: *POSTER 2015-19th International Student Conference on Electrical Engineering*. 2015, pp. 1156–1165.

[27] Aaron Helsinger, Michael Thome, and Todd Wright. “Cougaar: a scalable, distributed multi-agent architecture”. In: *2004 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 2. 2004, pp. 1910–1917. DOI: 10.1109/ICSMC.2004.1399959.

[28] Di Wang and Eric Nyberg. "CMU OAQA at TREC 2016 LiveQA : An Attentional Neural Encoder-Decoder Approach for Answer Ranking". In: *Text REtrieval Conference (TREC) 2016*. 2016, pp. 1–6.

[29] Ben Goertzel and Gino Yu. "A cognitive API and its application to AGI intelligence assessment". In: *International Conference on Artificial General Intelligence*. 2014, pp. 242–245. DOI: 10.1007/978-3-319-09274-4_25.

[30] John McDermott. "R1: A rule-based configurer of computer systems". In: *Artificial intelligence* 19.1 (1982), pp. 39–88.

[31] Caroline C. Hayes et al. "Intelligent Support for Product Design: Looking Backward, Looking Forward". In: *Journal of Computing and Information Science in Engineering* 11 (2011), pp. 1–9. DOI: 10.1115/1.3593410.

[32] Ashok K. Goel et al. "Cognitive, collaborative, conceptual and creative - Four characteristics of the next generation of knowledge-based CAD systems: A study in biologically inspired design". In: *CAD Computer Aided Design* 44.10 (2012), pp. 879–900. DOI: 10.1016/j.cad.2011.03.010.

[33] William C Regli, Simon Szykman, and Ram D Sriam. "The role of knowledge in next-generation product development systems". In: *Journal of computing and information Science in Engineering* 1 (2001), pp. 3–11.

[34] Yoshinobu Kitamura et al. "Deployment of an ontological framework of functional design knowledge". In: *Advanced Engineering Informatics* 18.2 (2004), pp. 115–127.

[35] Stephanie A Guerlain et al. "Interactive critiquing as a form of decision support: An empirical evaluation". In: *Human Factors* 41.1 (1999), pp. 72–89.

[36] John Eddy and Kemper E Lewis. "Visualization of Multidimensional Design and Optimization using Cloud Visualization". In: *Proceedings of DETC'02* (2002), pp. 899–908. DOI: 10.1115/DETC2002/DAC-34130.

[37] Gary M Stump et al. “Design Space Visualization and Its Application to a Design by Shopping Paradigm”. In: *Proceedings of DETC'03*. 2003, pp. 795–804.

[38] Po Wen Chiu and Christina L. Bloebaum. “Hyper-Radial Visualization (HRV) method with range-based preferences for multi-objective decision making”. In: *Structural and Multidisciplinary Optimization* 40.1-6 (2010), pp. 97–115. DOI: 10.1007/s00158-009-0361-9.

[39] Nathan Knerr and Daniel Selva. “Cityplot: Visualization of High-Dimensional Design Spaces with Multiple Criteria”. In: *Journal of Mechanical Design* 138.9 (2016), pp. 1–53. DOI: 10.1115/1.4033987.

[40] PW Chiu and CL Bloebaum. “Visual Steering for Design Generation in Multi-objective Optimization Problems”. In: *47th AIAA Aerospace Sciences Meeting*. January. 2009, pp. 1–14. DOI: 10.2514/6.2009-1167.

[41] Gary Stump et al. “Visual Steering Commands for Trade Space Exploration: User-Guided Sampling With Example”. In: *Journal of Computing and Information Science in Engineering* 9.4 (2009), pp. 1–10. DOI: 10.1115/1.3243633.

[42] Xin Yan et al. “Work-Centered Visual Analytics to Support Multidisciplinary Design Analysis and Optimization”. In: *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSM*. 2012. DOI: 10.2514/6.2012-5662.

[43] S Watanabe, Y Chiba, and M Kanazaki. “A proposal on analysis support system based on association rule analysis for non-dominated solutions”. In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. 2014, pp. 880–887. DOI: 10.1109/CEC.2014.6900650.

[44] Guido Cervone, Pasquale Franzese, and Allen P.K. Keesee. “Algorithm quasi-optimal (AQ) learning”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.2 (2010), pp. 218–236. DOI: 10.1002/wics.78.

[45] Allen Newell. *Human Problem Solving*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1972.

[46] Daniel Keim et al. *Mastering The Information Age – Solving Problems with Visual Analytics*. Jan. 2010.

[47] N W Hirschi and D D Frey. "Cognition and complexity: An experiment on the effect of coupling in parameter design". In: *Research in Engineering Design* 13 (2002), pp. 123–131. DOI: 10.1007/s00163-002-0011-3.

[48] Edward A Feigenbaum, Bruce G Buchanan, and Joshua Lederberg. *On generality and problem solving: A case study using the DENDRAL program*. Tech. rep. Stanford University CS Department, 1970.

[49] Edward H Shortliffe and Bruce G Buchanan. "A model of inexact reasoning in medicine". In: *Mathematical Biosciences* 23.3 (1975), pp. 351–379. DOI: https://doi.org/10.1016/0025-5564(75)90047-4.

[50] DARPA. *PAL - The PAL Framework*. 2011. URL: https://pal.sri.com/ (visited on 09/10/2017).

[51] Adam Cheyer and David Martin. "The Open Agent Architecture". In: *Autonomous Agents and Multi-Agent Systems* 4.1-2 (2001), pp. 143–148. DOI: 10.1023/A:1010091302035.

[52] Adam Cheyer, Jack Park, and Richard Giuli. "IRIS: Integrate. Relate. Infer. Share". In: *Proceedings of the 2005 International Conference on Semantic Desktop Workshop*. Aachen, Germany, Germany, 2005, pp. 59–73.

[53] Michael Freed et al. "RADAR : A Personal Assistant that Learns to Reduce Email Overload". In: *Twenty-Third AAAI Conference on Artificial Intelligence*. 2008, pp. 1287–1293. DOI: 10.1093/acprof:oso/9780199606375.003.0001.

[54] Wolphram Alpha LLC. *Wolfram|Alpha: Computational Knowledge Engine*. 2017. URL: http://www.wolframalpha.com/ (visited on 09/10/2017).

[55] Mycroft AI Inc. *Mycroft AI - Open Source Artificial Intelligence Voice Assistant*. 2017. URL: `https://mycroft.ai/` (visited on 08/31/2017).

[56] Clarity Lab. *Lucida*. 2017. URL: `http://lucida.ai/` (visited on 08/31/2017).

[57] Reiner Onken and Anton Walsdorf. “Assistant systems for aircraft guidance: Cognitive man-machine cooperation”. In: *Aerospace Science and Technology* 5.8 (2001), pp. 511–520. DOI: `10.1016/S1270-9638(01)01137-3`.

[58] Diana Donath, Andreas Rauschert, and Axel Schulte. “Cognitive assistant system concept for multi-UAV guidance using human operator behaviour models”. In: *Humous'10*. 2010.

[59] E. Özyurt and B. Döring. “A Cognitive Assistant for Supporting Air Target Identification on Navy Ships”. In: *IFAC Proceedings Volumes*. Vol. 45. 2. IFAC, 2012, pp. 469–474. DOI: `10.3182/20120215-3-AT-3016.00082`.

[60] G. Tokadli. *Cognitive Assistant | ACSL*. 2017. URL: `http://www.imse.iastate.edu/acsl/cognitive-assistant/` (visited on 09/10/2017).

[61] Kimberle Koile. “An Intelligent Assistant for Conceptual Design”. In: *Design Computing and Cognition '04*. Ed. by John S Gero. Dordrecht: Springer Netherlands, 2004, pp. 3–22. DOI: `10.1007/978-1-4020-2393-4_1`.

[62] P. Floss and J. Talavage. “A knowledge-based design assistant for intelligent manufacturing systems”. In: *Journal of Manufacturing Systems* 9.2 (1990), pp. 87–102. DOI: `10.1016/0278-6125(90)90024-C`.

[63] Lawrence Mandow and Jose Luis Perez-De-La-Cruz. “Sindi: An intelligent assistant for highway design”. In: *Expert Systems with Applications* 27.4 (2004), pp. 635–644. DOI: `10.1016/j.eswa.2004.06.005`.

[64] Kazjon Grace et al. "Personalised Specific Curiosity for Computational Design Systems". In: *Design Computing and Cognition '16*. 2017, pp. 593–610. DOI: `10.1007/978-94-017-9112-0`. arXiv: `1011.1669v3`.

[65] Tony McCaffrey and Lee Spector. "An approach to human–machine collaboration in innovation". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 2012 (2017), pp. 1–15. DOI: `10.1017/S0890060416000524`.

[66] J.S. Gero and W. Peng. "A situated agent-based design assistant". In: *Computer-Aided Architectural Design Research in Asia Conference*. 2004, pp. 145–157.

[67] W Peng and JS Gero. "Computer-aided design tools that adapt". In: *Computer-Aided Architectural Design Futures*. 2007, pp. 417–430.

[68] Hannah Bast and Elmar Haussmann. "More Accurate Question Answering on Freebase". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 2015, pp. 1431–1440. DOI: `10.1145/2806416.2806472`.

[69] Ana-Maria Popescu et al. "Modern natural language interfaces to databases: Composing Statistical Parsing with Semantic Tractability Ana-Maria". In: *Proceedings of the 20th international conference on Computational Linguistics - COLING '04*. 2004, pp. 1–7. DOI: `10.3115/1220355.1220376`.

[70] Yunyao Li, Huahai Yang, and H. V. Jagadish. "NaLIX: an interactive natural language interface for querying XML". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data - SIGMOD '05*. 2005, p. 900. DOI: `10.1145/1066157.1066281`.

[71] Yongjun Zhu, Erjia Yan, and Il Yeol Song. "A natural language interface to a graph-based bibliographic information retrieval system". In: *Data and Knowledge Engineering* July 2016 (2017), pp. 1–17. DOI: `10.1016/j.datak.2017.06.006`.

[72] B Sujatha and S Vishwanadha Raju. "Natural Language Query Processing for Relational Database using EFFCN Algorithm". In: *International Journal of Computer Sciences and Engineering* 4.02 (2016), pp. 49–53.

[73] Scott Reed and Nando de Freitas. "Neural Programmer-Interpreters". In: *International Conference on Learning Representations*. 2016, pp. 1–13. arXiv: `1511.06279`.

[74] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. "Neural Programmer: Inducing Latent Programs with Gradient Descent". In: *International Conference on Learning Representations 2015*. 2015, pp. 1–18. DOI: `10.1016/j.physa.2015.05.013`. arXiv: `1511.04834`.

[75] Manju Mony et al. "An Overview of NLIDB Approaches and Implementation for Airline Reservation System". In: *International Journal of Computer Applications* 107.5 (2014), pp. 36–41. DOI: `http://dx.doi.org/10.5120/18750-0006`.

[76] Neelu Nihalani, Sanjay Silakari, and Mahesh Motwani. "Natural language Interface for Database: A Brief review". In: *International Journal of Scientific and Computational Intelligence* 8.2 (2011), pp. 600–608.

[77] Yunyao Li and Davood Rafiei. "Natural Language Data Management and Interfaces". In: *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*. 2017, pp. 1765–1770. DOI: `10.1145/3035918.3054783`.

[78] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *Conference on Empirical Methods in Natural Language Processing*. Aug. 2014. arXiv: `1408.5882`.

[79] Ye Zhang and Byron Wallace. "A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification". In: *arXiv* (2015). arXiv: `1510.03820`.

[80] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. "A Hierarchical Neural Autoencoder for Paragraphs and Documents". In: *arXiv* (2015). DOI: 10.3115/v1/P15-1107. arXiv: 1506.01057.

[81] Zichao Yang et al. "Hierarchical Attention Networks for Document Classification". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2016, pp. 1480–1489. DOI: 10.18653/v1/N16-1174. arXiv: 1606.02393.

[82] Franck Dernoncourt, Ji Young Lee, and Peter Szolovits. "Neural Networks for Joint Sentence Classification in Medical Paper Abstracts". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. 2016. DOI: 10.18653/v1/E17-2110. arXiv: 1612.05251.

[83] Nico Schlaefer et al. "Semantic Extensions of the Ephyra QA System for TREC 2007". In: *Sixteenth Text REtrieval Conference (TREC)*. 2007.

[84] Nico Schlaefer. *OpenEphyra Code Repository*. 2014. URL: https://github.com/TScottJ/OpenEphyra (visited on 08/31/2017).

[85] D. Povey et al. "The Kaldi speech recognition toolkit". In: *IEEE Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, 2011.

[86] OpenCV Team. *OpenCV library*. 2017. URL: http://opencv.org/ (visited on 08/31/2017).

[87] Jens Lehmann et al. "DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia". In: *Semantic Web* 6.2 (2015), pp. 167–195. DOI: 10.3233/SW-140134.

[88] Kurt Bollacker et al. "Freebase: a collaboratively created graph database for structuring human knowledge". In: *SIGMOD 08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1247–1250. DOI: 10.1145/1376616.1376746.

[89] Text REtrieval Conference (TREC). *Question Answering Collections*. 2002. URL: http://trec.nist.gov/data/qa.html (visited on 09/05/2017).

[90] Ellen M Voorhees. "The TREC-8 Question Answering Track Report". In: *Natural Language Engineering* 7.04 (1999), pp. 77–82. DOI: 10.1017/S1351324901002789.

[91] Arvind Neelakantan et al. "Learning a natural language interface with neural programmer". In: *arXiv* (2016). arXiv: 1611.08945.

[92] Astro Teller and Manuela Veloso. "Neural programming and an internal reinforcement policy". In: *Late breaking papers at the genetic programming 1996 conference*. Stanford University Bookstore. 1996, pp. 186–192.

[93] Sandia National Laboratories. *Jess, the Rule Engine for the Java Platform*. 2013. URL: https://herzberg.ca.sandia.gov/ (visited on 09/15/2017).

[94] Daniel Selva. "Knowledge-intensive global optimization of Earth observing system architectures: a climate-centric case study". In: *SPIE Remote Sensing*. Vol. 9241. 2014, pp. 1–22. DOI: doi:10.1117/12.2067558.

[95] CEOS. *THE CEOS DATABASE: MISSION, INSTRUMENTS AND MEASUREMENTS*. 2017. URL: http://database.eohandbook.com/ (visited on 09/16/2017).

[96] Scrapinghub. *Scrapy | A Fast and Powerful Scraping and Web Crawling Framework*. 2017. URL: https://scrapy.org/ (visited on 09/16/2017).

[97] SQLAlchemy. *SQLAlchemy*. 2017. URL: https://www.sqlalchemy.org/ (visited on 09/16/2017).

[98] RDFLib Team. *RDFLib is a Python library for working with RDF*. 2013. URL: https://github.com/RDFLib/rdflib (visited on 09/16/2017).

[99] Daniel Selva, Bruce Cameron, and Edward F. Crawley. "A rule-based method for scalable and traceable evaluation of system architectures".

In: *Research in Engineering Design* 25.4 (2014), pp. 325–349. DOI: 10.1007/s00163-014-0180-x.

[100] Apache Software Foundation. *Apache Thrift - Home*. 2017. URL: https://thrift.apache.org/ (visited on 09/15/2017).

[101] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases". In: *ACM SIGMOD Record* 22.2 (1993), pp. 207–216. DOI: 10.1145/170036.170072.

[102] Rakesh Agrawal and Ramakrishnan Srikant. "Fast algorithms for mining association rules". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Vol. 1215. 1994, pp. 487–499.

[103] Gary G Hendrix et al. "Developing a natural language interface to complex data". In: *ACM Transactions on Database Systems (TODS)* 3.2 (1978), pp. 105–147.

[104] Gary G Hendrix. "Natural-language interface". In: *Computational Linguistics* 8.2 (1982), pp. 56–61.

[105] Jonathan Berant et al. "Semantic Parsing on Freebase from Question-Answer Pairs". In: *Proceedings of EMNLP*. October. 2013, pp. 1533–1544.

[106] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. "Towards a theory of natural language interfaces to databases". In: *Proceedings of the 8th international conference on Intelligent user interfaces - IUI '03*. 2003, p. 327. DOI: 10.1145/604045.604120.

[107] Peter H Sellers. "The theory and computation of evolutionary distances: Pattern recognition". In: *Journal of Algorithms* 1.4 (1980), pp. 359–373. DOI: 10.1016/0196-6774(80)90016-4.

[108] Django Software Foundation. *The Web framework for perfectionists with deadlines | Django*. 2017. URL: https://www.djangoproject.com/ (visited on 09/15/2017).

[109] Tom Christie. *Home - Django REST framework*. 2017. URL: `http://www.django-rest-framework.org/` (visited on 09/15/2017).

[110] L D Xu. "Case based reasoning". In: *IEEE Potentials* 13.5 (1994), pp. 10–13. DOI: `10.1109/45.464654`.

[111] Steve Menard and Luis Nell. *JPype documentation - JPype 0.6.2 documentation*. 2014. URL: `http://jpype.readthedocs.io/en/latest/` (visited on 09/15/2017).

[112] HubSpot. *Tether*. 2017. URL: `http://tether.io/` (visited on 09/15/2017).

[113] HubSpot. *shepherd*. 2017. URL: `http://github.hubspot.com/shepherd/` (visited on 09/15/2017).

[114] Mike Bostock. *D3.js - Data-Driven Documents*. 2017. URL: `https://d3js.org/` (visited on 09/15/2017).

[115] The jQuery Foundation. *jQuery*. 2017. URL: `http://jquery.com/` (visited on 09/15/2017).

[116] Tal Ater. *annyang! Easily add speech recognition to your site*. 2017. URL: `https://www.talater.com/annyang/` (visited on 09/16/2017).

[117] LearnBrite. *ResponsiveVoice.JS*. 2017. URL: `https://responsivevoice.org/` (visited on 09/16/2017).

[118] ZURB. *The most advanced responsive front-end framework in the world. | Foundation*. 2017. URL: `http://foundation.zurb.com/` (visited on 09/16/2017).

[119] Ricardo Cabello. *three.js - Javascript 3D library*. 2017. URL: `https://threejs.org/` (visited on 09/15/2017).

[120] Khronos Group. *WebGL Overview - The Khronos Group Inc*. 2017. URL: `https://www.khronos.org/webgl/` (visited on 09/15/2017).

[121] W3C. *WebVR Spec Version List*. 2017. URL: `https://w3c.github.io/webvr/` (visited on 09/15/2017).

[122] Explosion AI. *spaCy - Indusrtial-strength Natural Language Processing in Python*. 2017. URL: https://spacy.io/ (visited on 09/16/2017).

[123] NLTK Project. *Natural Language Toolkit*. 2017. URL: http://www.nltk.org/ (visited on 09/16/2017).

[124] Apache Software Foundation. *Apache OpenNLP*. 2017. URL: https://opennlp.apache.org/ (visited on 09/16/2017).

[125] Aurangzeb Khan et al. "A review of machine learning algorithms for text-documents classification". In: *Journal of advances in information technology* 1.1 (2010), pp. 4–20.

[126] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. "A convolutional neural network for modelling sentences". In: *arXiv* (2014). arXiv: 1404.2188.

[127] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv* (2013). arXiv: 1301.3781.

[128] Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[129] Kai Sheng Tai, Richard Socher, and Christopher D Manning. "Improved semantic representations from tree-structured long short-term memory networks". In: *arXiv* (2015). arXiv: 1503.00075.

[130] Denny Britz. *Implementing a CNN for Text Classification in TensorFlow - WildML*. 2015. URL: http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow (visited on 09/16/2017).

[131] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv* (2016). arXiv: 1603.04467.

[132] Python Software Foundation. *6.1. string - Common string operations - Python 3.6.2 documentation*. 2017. URL: `https://docs.python.org/3/library/string.html#template-strings` (visited on 09/16/2017).