Chatbots: On Demand Creation of Conversational Agents

Luís Filipe Amaral Pinhanços dos Santos
luis.pinhancos@gmail.com

Instituto Superior Técnico, Lisboa, Portugal

Outubro 2015

Abstract

The Instituto Superior Técnico's Spoken Language Systems Laboratory (L2F) group has a variety of dialogue systems that would be best explored if fit together in a single software engineering project. This new project should be the foundation of a system that creates a conversational agent on demand, unifying several L2F tools.

Previous to any architecture definition or implementation, a study of the system was required. We needed to identify control points, understand the structure and have a global vision of how each system works. We also wanted to observe some systems on the market, trying to acquire know-how as to what to do in our own system.

A first approach to a possible architecture was designed, trying to find a solution that would meet our demands in terms of modularity, customization and integration. Afterwards, we started the implementation process and went over an iterative process of the architecture until we reached our current state.

Ten user configuration scenarios were defined, trying to understand the level of changes necessary to satisfy each specification and evaluate the flexibility of our system. We also did user evaluation on the system, as this is a system designed for end users and not only for programmers of the L2F group.
Keywords: conversational agent, chatbot, dialogue system

1. Introduction

Who would not wish to create his own chatbot, giving it a pronounced personality or a knowledge base that would turn it into an expert in a specific subject? However, creating virtual agents is not as simple as a click of a button.

Although many tools are at our disposal, the ability to personalize a chatbot is not as easy as one would wish. The configuration of features like the looks of the chatbot, knowledge base or expertise is a complex task, not to be taken lightly. Although many available tools allow users to create and develop a chatbot, with or without visual components, the level of configuration is focused mainly on the domain level. We felt that there was not enough configuration manipulation, within these tools, that would allow the user to have a more complex control on the decision process of the answer. We could define, either with Artificial Intelligence Markup Language (AIML)[1] or direct trigger-answer modules, what would be the domain of the chatbot but there is not an ability to define the control points of the answer retrieval method. Each approach has its own way to get an answer but there is not any system that conjugates different tools into a single one.

The main focus of our project is how to approach this complexity, but taking into account dialogue systems tools developed in the L2F/INESC-ID group[2]. For this purpose, there is a need to study, understand and play with the tools that are intended to be integrated, building the structure that will allow to create a conversational agent that agglomerates the existing tools in a single system. Our goal is to guarantee that there is a foundation to integrate the studied systems, creating an answer retrieval method that will consider all the plugged in modules. As the configuration issue is one of our highest concerns, we will try to create an architectural approach that gives the user most of the control on how the system will work.

This project has to be a system with a stable main module, in which all others will connect. With this step concluded, we need to create an answer retrieval method that will follow the methodologies implemented in the studied L2F tools, keeping a decision path that will be flexible to a variety of configuration combinations that can be defined by the user.

As the study and adaptation of the code in-

[1] http://www.alicebot.org/aiml.html

[2] https://www.l2f.inesc-id.pt

creases, so does the complexity of the changes that have to be done to the tools that were intended to be integrated. As these tools are not prepared to work in a single environment, most of the functional code has to be deeply understood so we can know what to change without totally changing the paradigm, methodology or purpose for which each tool was created.

2. Background

2.1. Edgar

One of the tools in the L2F group is Edgar, the butler of the Monserrate Palace in Sintra [3]. This is a conversational agent with tutoring goals, built to answer the questions of the ever inquiring visitors of the palace. Edgar may receive written or spoken inputs, outputting his answer in both forms. Received input can be perceived and answered in Portuguese, English or Spanish.

Edgar has an in-domain knowledge base, created thoroughly by hand in an XML format, that is capable of answering to direct triggers in each of the mentioned languages. This is possible by having multilingual pairs, of different paraphrases, of question and its possible answer.

This format is already contemplating attributes such as emotion and intensity, so that when a GUI interface is built, the graphical behavioural features of the chatbot can be set in motion.

These knowledge base can be extended with equivalences between words so that different triggers can be contextualized and it could lead to the same answer.

Although Edgar is a chatbot created to mainly answer questions about the Monserrate Palace, it has also some small talk and facts in his knowledge base. But this was not enough and Edgar has the ability to use a tool created to work with one of the most popular chatbot programming languages, Artificial Intelligence Markup Language (AIML)[3].

2.1.1 ProgramD

The AIML interpreter integrated in Edgar was ProgramD, an open source AIML bot system created for Alicebot[4]. This system is used to deal with different subjects such as slang, cinema or compliments, but it can be extended to deal with any kind of subject, as long as it is defined with the AIML set of rules.

2.2. Say Something Smart

Say Something Smart (SSS) is a tool created to deal with Out-Of-Domain (OOD) interactions, that are out of the domain of the chatbot knowledge base, using movie subtitles as corpora (latin plural of corpus) [4] [1]. For a large knowledge base this can lead to a huge number of interactions, so SSS was integrated with Lucene[5], an information retrieval library, that is capable of text indexing and fast searching among a huge amount of data.

2.2.1 How does SSS work?

For each interaction received in the SSS dialog system, Lucene will tokenize it and compare it to the triggers that are on the corpus, giving different relevance to each, and creating an organized list with the best matched trigger-answer pairs. To filter this output given by Lucene, which consists of the best X matches, with X being a number configured by the developer, SSS can be configured to use an algorithm that combines both the Jaccard and the Overlap algorithm, or any of these separately, which will compute the similarities between the user input and the trigger-answer pairs returned by Lucene.

SSS will take the best matches from Lucene, trying to find out what is the best answer with its own algorithm. For this, SSS uses four different measures, with different weights, that combined are supposed to give the best possible answer. As this is based only in movie subtitle interactions, not every answer is appropriate or even makes sense.

2.2.2 The SSS measures

SSS applies four measures which are used to discover the best possible answer given by Lucene. Each measure has its different weight and can be combined to give an organized list of twenty answers with the best possible values combined. These weights can be configured in the config.xml file located in the "config" folder inside the "resources" directory. It should be noted that it is not mandatory to use the four measures at the same time, but at least one should be active. The sum of all four weights should always be 100 per cent.

The sentence with the higher score will be the one that will be given as an answer to the user interaction.

Answer Frequency is the first measure. This measure will give a higher value to the answer that is most common among the possible answers, in order to give some value to the redundancy found in the corpus. Due to giving a higher score to the most frequent answer, SSS assumes that if an answer is more frequent to a given question then it has a higher probability of being the right answer.

The second measure is Answer Similarity to the User Input. One of the many possible ways to face the problem of giving a viable answer is considering that if an answer has many similarities to the question, then it is highly probable that it will be a good answer.

The third SSS measure is Question Similarity to

[3] http://www.alicebot.org/aiml.html
[4] http://www.alicebot.org/
[5] http://lucene.apache.org/core/

the User Input. If it can be assumed that an answer could have some similarities to a question, then it is also correct to think that if there is a question, in the knowledge base, that is very similar to the user interaction, we can assume that any answer given to that question could also be a possible answer to the input given by the user.

The fourth, and final, measure will consider the time difference between the found trigger and a possible answer, so that when a match is made in the trigger, the closest following answer could be related to the given subject and maybe a possible match.

2.3. Talkpedia

Talkpedia was created with the purpose of returning appropriate[6] answers to Out-of-Domain (OOD) interactions. When Talkpedia receives an OOD interaction it will create a list of possible search terms, namely the nouns or expressions. The identification of the terms class is done by using a parametrized Portuguese TreeTagger[7]. After the list is created, the terms will be prioritised so that the articles retrieved with higher priority terms are more likely to contain an appropriate answer.

Then Talkpedia will retrieve articles from Wikipedia, based on queries created with the terms of the list. If any answer is found, the result is presented in sets of pairs article-terms. With the priorities previously defined, a selection process begins so that one final answer is selected. This final answer will still be modified, with a prefix being added, so that it generates a more realistic answer. For instance, if the user interaction is "How is the Pope doing?", Talkpedia's answer is "Unfortunately I cannot tell you, but the Pope is the Bishop of Rome and the leader of the worldwide Catholic Church" (example from [5]).

3. Architecture

As we first approached this project, there was an intention to build an architecture that could support future new features with a plug-and-play methodology. The design of such an architecture would need to be modular and with a well structured layered design. The relations between each layer would have to contemplate the interaction between modules, but keeping the layers and their purpose well defined.

As this project was intended to merge several systems into a single one, the architecture solution first designed was thought believing that these projects could support such an approach with standardized changes to the code, that could be replied between each new system. An utopic view of the solution this was. The longer we meddled around with the code, the harder this kind of implementation looked feasible.

3.1. Implemented Architecture

The goal was to have an architecture that would withstand all possible plugins with minimal to none effort on the programming side. But soon into the code exploration, we realized that this was something that would collide with our configurability feature, as it would be necessary for a programmer to guarantee that the WebServices would be established and the usage of each plugin, in terms of their own purpose.

As the implementation was progressing, the 4-tier layer architecture stopped following the approach that we were trying to achieve, as we chose Edgar to be our main brain. We were considering a central HUB to deal with the different messages that could come from the different plugins. Using this paradigm, we though that one of those systems could work as our central brain, our main agent, dealing with the communication with other plugins, and withstanding the function to receive a user interaction and returning its answer.

This choice was made because we though that the basis, already built in Edgar, would be a good starting point as to where our plugins would meet. This approach would allow us to take an already established and functional system, and build on it. The growth of our project started with Edgar as our nuclear system and we decided to maintain it, giving up on the 4-tier architecture and merging the Service Layer with the Business Logic Layer.

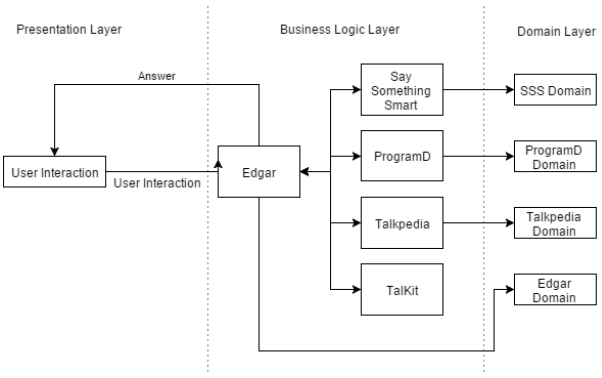

Figure 1: Implemented Architecture

With Edgar being the central module of our system, the user interaction no longer flows directly on the classifier to know its way. Instead, it receives the interaction from Edgar and reports directly to it. The decision method is implemented in Edgar, so that even if the classifier is not working, the chatbot will still be able to find an answer.

The employment of Edgar as our core module, with the singularity that its module receives a user

[6] An appropriate answers is not always the correct answer, but instead an answer that is coherent with a dialogue

[7] http://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/

interaction and returns a final answer, allows us to perceive that the implementation of the Presentation Layer would be similar to the one Edgar already uses, allowing future work to build on this approach and create a new interface, independent of the Business Logic Layer. Every other Business Logic Module, that we implemented as WebService plugins, are directly connected to Edgar. This will keep our intention of achieving modularity because whatever the GUI that could be used, it will only need to send its information to a single module and wait a response from it.

The previous architecture was expected to have incorporated a direct match methodology between user interaction (or Trigger) and answer. As this functionality is already implemented in Say Something Smart, we found no need to create a tool that would replicate one possible behavior of the said system. Instead, the user has the free will to define the weights of the four possible measures.

The Domain Layer is kept, as it has direct connection with each respective module, meaning that the Domain of each module can only be changed inside said module. As we were integrating the plugins, the thought of separating each domain into a single unified layer would imply major changes to every system without a real gain to it, because if we would want to permanently remove a system, two operations would be required: eliminating the system and then eliminate by hand all its Domain. By keeping the domains and their original locations would be a higher modular approach. If a new system is to be integrated, as long as the Domain is in its structure, there is lesser risk to the changes made to the code. If a single unified layer would be used, the risk of breaking the dependencies would be higher.

Not all the tools were integrated in our project, as we chose Edgar to be our basis module, our main brain, deciding to complement it with ProgramD, SSS and Talkpedia, as they are of different natures when it comes to the answer domain. We also integrated TalkIt, not to be used as a dialogue system but opted to use its classifier. We can integrate JustChat as a filter module, taking advantage of its three filters. A user interaction would enter, the module would filter it based on what the user wants, and would return a treated interaction to the Answer Retrieval Algorithm. JustAsk can also be integrated, as its functionality resembles Talkpedia. We would need to create the WebService relation, and insert the module in our algorithm.

4. Implementation

Our main feature is the integration of several L2F systems, maintaining a fully functional base and a more complex ability to configure the systems integrated. The most challenging feature of this thesis project is how to harmoniously conjugate all different systems in one. Each was built to work on its own, with the exception of Edgar and ProgramD that were already working together, and they were never thought as systems that one day would work in a synergetic environment.

The logical choice for the programming language would always be the one that would allow continuation of previous works, instead of utterly changing the paradigm and base work of the systems that would be incorporated. As all system were developed in Java, there was no question in trying to explore different options.

Exploring the different systems would give us an idea on how to conjugate them all, with an effort to make its architecture as modular as possible. Our initial idea was to focus on the modularity of our project, so that its scalability could be guaranteed with the addition of other functionalities. This would also allow us to separate the core of the project from its graphic component. As we went to a deeper level of the study of our applications, and how they could be integrated with each other, we started realizing that this modularity would not be as functional as desired.

Every system was designed, and built, to work in a single architecture, with the exception being Edgar and ProgramD. Even though TalKit also integrated SSS and Talkpedia, using a Java wrapper, the approach used by Edgar guarantees a higher tolerance to faults. If the code from any of the integrated systems should stop working, we have to assure the chatbot would keep functioning and giving believable answers to the users, until the system can be put back online. With the Java wrapper approach, if any unexpected exception or any code sequence that would break the execution happened, our whole system would stop working, causing a major malfunction. Also, if any bug is reported and there is a need to update the malfunctioning code, we would have to change it, compile it, create the wrapper and copy it to our systems folder.

If we can use a main system, integrating the other systems with it, but using a Plugin approach with WebServices, like the one exampled with Edgar and ProgramD, even if one of our systems would break, the rest of the system would keep functioning. The only issue is that we had to guarantee, programatically, that our code would be prepared to respond accurately to any plugin instantly breaking. This was implemented in our Answer Retrieval Method, by stating that if no response was returned from the evoked plugin, the system would keep trying other plugins, until one had an answer that could be retrieved to the user. But we still had to choose one system to be able to receive these plugins, and to

gather a response if every other system would fail. As Edgar is the most complex system, and the one that already implements an example of this plugin methodology, it was our logical choice to use him as our main brain. If any bug is reported in this methodology, we can correct it while the system is still working, exception made to bugs in Edgar, and put it immediately back online so that our system is able to again use that system.

Using Edgar as our main brain for the answer retrieval method, we integrated it with the TalkIt classifier as well as with other answer retrieval systems like Say Something Smart and Talkpedia. The result is a more resourceful Edgar, specially with Out Of Domain interactions, with diverse methodologies incorporated to its answer retrieval ability. If we want a clean basis, Edgar is a system that is capable of functioning on minimal services, and from which we can build on our knowledge base through its XML files.

The central focus point was how to connect all the different systems, maintaining a stable environment without destroying all the previous work and still being able to give the user a high-level configuration permission to decide how the new system may work. Two major conflicting issues were then discovered, as we wanted to maintain the systems specific configuration in their own modules, but would also wanted to create an architecture that would easily allow a plug-and-play methodology for new plugins. This revealed to be a task bigger than the initial purpose, as plugins may have different purposes, and so they would affect different sections of our system. For example, we integrated the TalKit classifier so it could return the module that is more appropriate to answer to the perceived user interaction. This is completely different than using answer retrieval systems like Say Something Smart or Talkpedia, that return a single answer to each user interaction.

Although some modularity is allowed, every new plugin still requires programming skills. To guarantee the functioning of the base system, even if some plugins would break its execution, a WebService methodology was implemented. As the new systems were not built to work in this methodology, it required that the plugins code needed to be changed, trying always to maintain its level of authenticity and main purpose, but having to make some structural adjustments. For example, in the TalkIt classifier, there was not a method that would return the system to use and we could not use the answer retrieved by the classifier. Instead, we created a method that, following the answer retrieval algorithm already used [2], would return the system to use. With this result we can directly invoke the system and retrieve an answer to the user interaction.

For the answer retrieval method, two different methodologies are used, depending if the TalkIt classifier is active or not. If the TalKit classifier is active, the answer retrieval method follows the algorithm previously described, and when inputting a new plugin the programmer needs to guarantee several steps, namely that the WebService is well established, with a defined communication port and the existence of a method that returns an answer to a given utterance and that the new plugin is incorporated in the system decision algorithm of the classifier. If the classifier is turned off, the programmer needs to guarantee that the WebService is well established, with all the work that it is linked to it, but the invocation of the new plugin does not need to be treated so thoroughly.

4.1. Plugins

Several L2F in-house systems were studied but only Edgar and ProgramD were prepared to be integrated in a single system. We had to make adaptations to the core code of each system, following a step by step implementation to make sure the WebService was well implemented in each one. We started by defining the new plugin's attribute in Edgar's main configuration file with a hostname, port and the attributes for the Answer Retrieval Algorithm. In the system's own configuration file, the same hostname and port also needed to be specified.

But the XML representations needs to be interpreted, so we had to add the code to read the new attributes, both in Edgar's and the plugin's Configuration parser.

In Edgar's package l2f.webservice, a new set of files was necessary to materialize the new plugin WebService Client. In the plugin side the same was needed to implement the WebService, creating a package l2f.webservice but with the implementation of the service instead of the client.

This process was completed with the respective binding class in the package l2f.dialog.plugin of Edgar.

The new system now needed a listener to keep the channel open and trying to receive the calls that were made. For our system to be able to retrieve an answer, it was necessary the existence of a method that would receive a string, our user interaction, and would return another string, our possible answer. Some of the integrated systems did not have this kind of method and so, we had to manually create it with the core functionality that would permit us to retrieve an answer.

4.2. Answer Retrieval Algorithm

With the level of configuration combinations that we allow the user to define, our answer retrieval

method would have to be refined to contemplate all the possibilities. But it is not only the configurations that we had to be concerned, when dealing with multiple plugin systems, as their execution breaking from a malfunction would also be of some concern. We had to have a system that would keep its ability to respond to user interactions, even without being able to connect to all of the integrated plugins.

Because of this issue, there is a lot of code that can be reused but with some subtle particularities, depending on the configurations defined or if the plugins are offline.

Our algorithm begins by receiving a user interaction, evaluating whether or not the classifier is active. If the classifier is inactive, the main agent, Edgar, will try to retrieve an answer from its knowledge base. After this step, every active plugin, defined by the user configuration, will be evoked and a list of valid answers, from the plugins, will be created. The answer retrieved from the main agent will be compared with the list of answers retrieved from the plugins, and the system with the highest confidence, defined by the user, will be the answer given to the user. If no answer is retrieved, the algorithm will retry this approach one last time before returning a failed answer. If still no answer can be found the system will inform the user that no agent or plugin was able to answer. This is a rare happening, since Edgar has implemented a system of diversion, with the invocation of an interaction _REPEAT_, that is prepared to avoid this situation with a believable answer.

If the classifier is active, the decision to which path to follow will reflect the configuration defined by the user of the attribute forcePriority. This attribute gives the user the ability to try to force the retrieval of an answer in the agent or plugin with the highest confidence. If such an attribute is active, the algorithm will firstly try to get an answer of the chosen agent or plugin. If an answer is retrieved, this will be the output given to the user. In case the agent or plugin with the highest confidence is Edgar, the algorithm will still run the user interaction with ProgramD. We chose to implement it this way as ProgramD is mainly an addition to the main agent Edgar.

In case that the chosen system is not able to retrieve any answer, the classifier routine will be used and the classifier will define the taxonomy category of the user interaction. If no category is found, the classifier is not able to retrieve which system to use and the algorithm will try to retrieve an answer as if the classifier was not active at all, just considering which systems are online and retrieving the answer from the system with the highest confidence.

If the classifier is able to determine a taxonomy to the user interaction, the answer retrieval algorithm will be similar to the one used in TalKit. Edgar will deal with most of the on-domain interactions, relegating to the other integrated plugins the task to deal with out-of-domain interactions, namely Talkpedia will deal with factoid questions and SSS will deal with non personal interactions. If the chosen plugin is SSS but it is not active, for definition, execution error or even if it can not retrieve an answer, Edgar will be the system chosen to deal with the interaction. If Talkpedia is chosen by TalKit but it is not active, for any of the already mentioned reasons, the algorithm will secondly try to use SSS, as it is an out-of-domain interaction, and only if SSS also fails, then the algorithm will call upon Edgar to try and retrieve an answer.

The algorithm path to retrieve an answer, if the classifier is active but the attribute forcePriority is inactive, is similar to the one when the attribute forcePriority is active but the agent or plugin with the highest confidence can not retrieve an answer.

4.3. Towards a flexible configuration

Making all systems flow together in a single environment was the main goal of our project. This was the most time consuming goal to achieve, always trying to understand how the systems would better fit together, but staying true to the purpose for what they were built. But this goal would not be completed with one single step. We wanted to give the user the power to structure and define this environment to his will, without losing focus of what our project final goal is, retrieving a believable answer to a user interaction.

We wanted the user to feel the power to choose what he thinks is the better conjugation of the participant systems. If the user feels that his chatbot should be focused on answering a majority of factoid questions, for example, for a fun-fact intended chatbot as his own personal knowledge base, then the user has the ability to give a higher confidence to Talkpedia. Determining what plugins are active or not, and the respective confidence given, will influence the answer retrieval algorithm.

4.3.1 The Brain

Although all plugged in systems may be deactivated, the main agent, what we defined as Edgar and worked as our brain, can not be deactivated and also has a confidence property attached to it. Imagining that all systems would fail, and only Edgar, our main agent, would still be online, our system would always try to determine a believable answer, with his current knowledge base. Only if all corpora would be removed, would Edgar stop being able to try to determine a valid answer.

4.4. Plugins

The range of confidence is not limited purposely to ensure that, no matter how many systems may be connected, there is always the possibility to rank them in a different unitary system, where each has its own value.

The TalkIt classifier is the first decision paradigm that the answer retrieval method faces. Whether it is active or not, the agent will choose different paths to follow. For example, if the classifier is deactivated, the agent will order its retrieved answers to satisfy the user's configuration of the plugins confidence, without overlooking the main agent's confidence. The system that retrieves a valid answer and has the highest confidence will be the one chosen as the final answer. If the TalkIt classifier is activated, one of two paths may occur, depending on the configuration of the property forcePriority. This priority, when active as the boolean true, will always force the plugin with the highest confidence to retrieve an answer. If such answer does not exist, then, the answer retrieval method will invoke the classifier and follow the path that TalKit has defined.

If this forcePriority property is false, the TalkIt classifier will run the same path as if no answer was retrieved from the highest confidence system. This implementation was made to give more freedom of decision to the user. For example, if he has most confidence that Talkpedia will be able to give the most valid answers, but wishes to ensure that, if Talkpedia fails, the chatbot will be able to follow the logical and trained path defined by the classifier, without leaving the answer retrieval method to be solely based on the confidence the user has on the systems.

4.5. Other Systems Configurations

Despite having this main configuration possibilities, every system has its own configuration, although not easily modifiable by a user without previous knowledge of the system or without programming skills. These were purposely left inside each system, so it better fits our modular approach to the problem. Our main brain, Edgar, can be used with a different corpora or even with its own knowledge base enlarged by the addition of XML files with interactions that he could directly answer. ProgramD, as previously mentioned, allows us to create a set of rules and equivalences that Edgar is able to detect and answer in conformity. TalkIt is a system that is able to retrieve an answer to a user interaction but, in our project, we decided to use only its most eloquent tool, the classifier. To retrain the classifier, a new training set and corpora would be needed. In Say Something Smart, there are 4 measures to be accounted for, as previously described in Section 2. The user can configure the weight of these measures as he assumes they would better work together, by instinct, by trial-and-error or any other way the user sees fit. Talkpedia is the less configurable system, as all its knowledge is based on a URL connection to the Wikipedia server, retrieving the necessary information and trimming it. The user can only define the possible template phrases where this information will be used.

5. Evaluation

With most of the focus on this project to its architectural structure and system integration, we could not overlook the purpose for what the project was being built, becoming a conversational agent system. There are some ways to test a software architecture but more from an applicative and technical point of view, usually being done at an early stage of the development of the project.

In an ideal project, we would start to develop and plan several architectural approaches to understand what would best fit our needs. Then, we would simulate the effort to keep the dependencies between possible modules, dividing them in layers. An iterative methodology would be used to understand flaws and rectify them until a possible solution is reached. Only then would the implementation phase start.

As much as this project was intended to result in a specific architecture, the limitations of working with already developed systems resulted in an adaptation of the architecture and in an inadequate timing for this type of testing. When a project has the kind of artifacts that this one had, already produced, it is more appropriate to perform more of a functional testing.

With this in mind, two types of functional testing were applied. Ten different scenarios were defined, by the Thesis' supervisor Professora Luísa Coheur, to understand how limited or not our project is. These scenarios identify possible uses, that a user with any kind of interests could realize, and we will try to see if such demands are possible or not. But this is, in fact, a system built for the end user to interact with. Although the quality of the answers is not entirely related to the developed work, it is still a goal to produce believable answers. A script, briefly explaining the project, is handed to a user and he will have to perform two different tasks that involve configuration and interaction with the system. In the end, the user will fill a form so that we can understand how satisfied he is with the retrieved answers and how adequate they were.

5.1. Scenarios

Our objectives always revolved around how configurable this system could be to the end user. We intended to give more power and control to the users,

while integrating systems that already have their own configurations. Not only this is not an easy and straight forward task, as it is not simple to evaluate. We can not evaluate the system simply on the answers retrieved because the dialogue systems that have that functionality were already created and we did not have the task to improve them.

Through the definition of ten possible user configuration scenarios, we were trying to simulate needs and desires of an end user. We did not test them with users but we tried to respond to the needs, in a configuration point of view, of each scenario, trying to figure out if such is possible or not, and how we could satisfy those specifications.

Scenarios:

1. **How to define an agent where all the possible interactions are previously defined by hand and that if it can not answer, it does not?**

   Both Edgar and ProgramD are a viable and valid solution to a user that wishes to define each interaction that the conversational agent can answer. While a user interface is not available, the user would need to have some knowledge of XML[8] or AIML[9]. With a GUI implemented it would be an interesting addition to have a tool that could automatically create the code necessary to expand the corpora by hand.

2. **How to define an agent that can answer to factoid questions and some personal questions?**

   By activating Talkpedia, our system is automatically able to respond to factoid questions[10]. To define some personality traces to which our agent could answer, there would be a need to, as in Listing **??**, define the interactions by hand through an XML corpora file.

3. **How to define an agent that has some hand defined personality traits but only answers to small talk.**

   Every aspect of the personality of a conversational agent can be defined in an XML file located in the ProgramD plugin[11]. The corpora in Edgar defines the domain to which the agent can answer, so if we want the agent to only be able to interact with small talk, the corpora should be decreased to the files that have that specifications[12].

4. **How to define an agent that can answer factoid questions, has hand defined personality traits and answers to small talk.**

   We already covered the steps necessary to satisfy each condition, so we only need to conjugate them in a single configuration. Activate Talkpedia, have a small talk file in our corpora and define personality traits in ProgramD.

5. **How to hand define an agent that can answer in Spanish.**

   The integration of multiple systems did not contemplate the multi language possibility. Some systems were created to work in Portuguese and some in English. Only Edgar has the ability to answer in Spanish but ProgramD, with its AIML, can be hand defined to understand Spanish. The other integrated systems would require more effort. SSS would need a knowledge base in Spanish, Talkpedia would need its templates to be written in Spanish and to change the Wikipedia used, from the Portuguese to the Spanish, and TalKit would need to be trained with a Spanish Corpora before being used.

6. **How to define an agent that can integrate a new Question & Answer engine to the specifications of item 4.**

   We do not have an implemented Plug & Play methodology for our external systems, since most systems were developed without the contemplation to work in an integrated environment and it involves a lot of effort to adapt the code, but an WebService methodology was implemented that can, with less effort, combine any new QA system. As the WebService classes would need to be created and adapted, the system would also need to be contemplated in the main configuration file and in our Answer Retrieval Algorithm.

7. **How to define an agent that can answer to interactions like Yes/No Questions or Rhetorical Questions.**

   The TalKit system as a whole is prepared to answer to that type of interactions, but we decided only to integrate the classifier, leaving the answer functionality to other systems. With some effort, specially in the aspect of the changes necessary to the existing TalKit code, our system would be able do include that feature. In this moment, we can not guarantee an agent that would be completely able to do this kind of interactions, with our best possibility being the configuration of SSS measures to focus in user input similarity or the direct

[8] /edgar/resources/qa/corpusXML/edgar-2013-05-23/*.xml

[9] /ProgramD/aiml/edgar/*.aiml

[10] edgar/resources/qa/config/dialogConfig_pt.xml

[11] /ProgramD/conf/edgar/properties.xml

[12] /edgar/resources/qa/corpusXML/edgar-2013-05-23/small-talk.xml

definition of that kind of trigger in Edgar's corpora.

8. **How to define an agent with a new knowledge base of factoid answers, for example, based on Answers.com[13], and using it with high priority.**

   Although Talkpedia is a system used to retrieve answers from a given URL, it is only prepared to deal with the schematics and structure of Wikipedia, not being able to withstand other factoid answering websites, like Answers.com. To perform this feature a whole new system would be necessary .

9. **How to define a new classifier, new QA Module and how to adjust the new classifier.**

   To replace the current classifier TalKit, we would need to create a new method that relates the classification with the integrated modules in our project. Afterwards, we would need to relate the existing WebService methodology with the new classifier. Based on the new taxonomy and evaluation, we would have to specify which systems would be best fit to answer each type of interactions. The same process would need to be applied to a new QA module to be integrated in our project, but the answer retrieval algorithm would need to be revised to include this new module in its consideration and decision methodology.

10. **How to define different priorities to different systems.**

    Any dialogue system plugged in our system can have a numerical confidence. The higher the confidence, the higher the priority given to an answer coming from that module. Our project is prepared to work with the expected and integrated systems all at the same time or even with only Edgar active, as Edgar can not be deactivated. If Edgar corpora is reduced to a minimum or, let us say, non existing, then our agent would not be able to retrieve any answer as it would not have any knowledge base to work with.

5.2. User Evaluation

Even though we do not want to judge the success of our project based on the quality of the retrieved answer, as it is a process that little has to do with the work developed, we still want to gather some information about the adequacy of the interactions. The evaluation is performed in our hardware, where the project is already installed and the paths configured. As our project include other tools, with big corpora, knowledge bases or even training files for the classifier, it has a memory space requisition that we did not want to put our users through. Doing the whole user evaluation process in our computers saves everybody time and effort.

A script was defined, explaining what our system consists of and briefly explaining the tools integrated with it. As the user is reading the script, we open the configuration files in a text editor so the user can handle them while performing the tasks. In a completed system, the project would have a simple GUI that would spare the user from the configuration code, with simple buttons and text to alternate between all possible configurations. Since we do not yet have this implemented, and the user does not have to know where the configuration files are, we decided to open them in an editor but all interaction with the files would be of the user's responsibility. Then, we launch the system in the command line but delegate all textual interaction to the user, also. We want to evaluate the interactions and how well the user interacts with the configuration possibilities. Each task will end with the registration of ten user interactions, the user evaluation of the answers adequacy and the possibility given to the user to write what he thought of the system.

In the first task, where a more factoid agent was defined, the agent answered all factoid questions with a high level of satisfaction, where all answers were categorized with the maximum grade. It did not behave as well when asked personal questions or interactions that would go out of factoid domain, with the adequacy ranging from the lowest grade to medium. Since the forcePriority attribute was active and focused on Talkpedia, it is normal that a high level of adequacy would be verified. Some adjustments should be made to the Answer Retrieval Algorithm, as the agent returns very inadequate answers to small talk or personal questions, even with all systems active. The answers returned from out-of-factoid questions would only reach the medium grade if a topic or main word could be retrieved from the user interaction. For example, "do you like sushi?" would not get a personal answer but instead it would retrieve the Wikipedia definition of sushi. As it is not entirely incorrect, because the agent is more focused on factoid interactions, it does not completely satisfy the question because of its personal tone.

The second task defined a more embracing agent, without a specific domain, and with SSS as the system with the highest confidence, with the TalKit classifier active but without the forcePriority attribute. This agent did not perform as expected, with most of the answers being graded average or below. The interactions that performed worst were the ones that TalKit would classify as Personal, and

[13] http://www.answers.com/

which the classifier would delegate the task of answer retrieval to SSS. This approach followed the algorithm that was already implemented in TalKit that, derived from our user evaluation, we can now perceive as inadequate to a system with such a large spectrum of implemented tools. TalKit was concluded as a Dialogue System, able to return answers to the user interactions, based on SSS, Talkpedia and some predefined templates. We opted to maintain the usage of SSS and Talkpedia in the same taxonomies, Personal and Impersonal respectively, and replace the templates with Edgar combined with ProgramD.

This approach is not the best when dealing with our systems, as the answers did not satisfy the user evaluation. A better approach for the TalKit classifier would be to delegate to Edgar and ProgramD our Personal Interactions, as they can be built around personal specifications, keep Talkpedia with Impersonal questions, where most of them are factoid, and assign SSS to other taxonomies. Other possible approach would need to relate our answer retrieval algorithm to the TalKit domain so that we can also retrieve the templates. This was initially considered but, as our goal was to use TalKit only as a classifier and not a Dialogue System, we decided to only use the other dialogue systems as valid answer retrieving tools.

6. Conclusions

More than a scientific research assignment, this Master's Thesis was a hands-on project. We were not aiming to reach new scientific achievements but, instead, we were aiming to build the foundation of a system that could integrate and support most of the L2F Dialogue Systems. New L2F projects are created every semester and this is a project that does not have a definite ending, as it should be the main system where all these new projects should be incorporated. Instead, this is a project that represents the end of a cycle and the beginning of a new one.

We started studying what were the systems we had to work with. What was their purpose, how they worked and what were the inputs and outputs that each was prepared to use. We could not look at these systems in a black box methodology. We needed to understand where the dots would connect and what was the bigger picture. Where we were heading was as important as how we would get there.

But we could not look only to systems developed in-house. We needed to broaden our horizons, step outside and see what was out there. We were trying to understand where our project could be unique and where it could distinguish from others. Our biggest goal would have to explore the lack of configuration possibilities that other systems allow. They will let you define the Domain, but how many would allow the user to configure how he wants the system to act? Others may have a more flexible architecture, but we built an architecture that, with some level of modularity and layer distinction, can integrate our own systems. Systems who were never designed, or even expected, to work in a single symbiotic environment.

We needed to define an architecture and we were ambitious. We aimed high and envisioned a design that would integrate all systems with a Plug & Play methodology, in a 4-tier layer architecture. We wanted high modularity combined with high functionality. Although ambitious, our modularity would diminish the quality of our functionality and we had to adapt our initial architecture to one that could conjugate goals of modularity, functionality and customization. As we learned in latter stages, the creation and planning of such a project, where our artifacts are already designed and implemented, is a process that involves several iterations. And that was exactly how things ended up, with iterations over iterations where we were refining our design, so that it would meet our needs.

This architectural iteration process, as it was not planned initially, ended up meddling with the implementation phase of the project. As we were facing integration obstacles, created by the systems we defined would make the foundation of our system, we were adapting our architecture so that we would still have a layer definition, but with the difference that we started using one system, Edgar, as our central module, where all others would connect. This way, the handling of the communication would be exclusive of Edgar and the Edgar module would be responsible for delegating tasks and withstanding our Answer Retrieval Algorithm.

As we were integrating the systems, we had to guarantee that our objective of giving the user more power over the system, than in other observed systems, was being accomplished. We defined confidences so that the user could define what are his most trusted answer retrieving tools and how the system should order them. The inclusion of the TalKit classifier opened more decision paths in our algorithm, as well as the ability to, no matter what is the configuration, try to force one of the systems to return an answer. This all complements the already possible configurations of each system, that we decided not to include in our main configuration file, as that would create dependencies that would break the modular aspect of the architecture.

To evaluate the created system, a set of constraints and specifications were defined. We wanted to understand and explain how our system would adapt to new possibilities and the amount of effort

that each specification would need to be integrated. We also performed user oriented evaluation to see the performance of our system in a user interaction environment, although the adequacy of answers was not one of our main goals, as it has dependencies that were not implemented by us. From this user evaluation we were able to understand what else we need to improve in our own Answer Retrieval Algorithm.

In conclusion, we believe that the basis of our structure is successfully implemented and that we can continue to build on the developed system.

6.1. Future Work

Our system is a continuous project. There are new systems being developed in-house and it will always be an objective to integrate new features in our system. It would be interesting to add a Graphical User Interface (GUI), specially to facilitate the usage of the configuration aspect by the end user. For example, an interface where the XML files would be translated into buttons to activate or deactivate properties of our system, to add new set of rules to the AIML of ProgramD or even to be able to define the weights of SSS in a graphical environment.

We worked on this project with an approach to its logical side, but a conversational agent can have more features than just the ability to return answers. It would be interesting to add a context driven algorithm so that the answers become more believable and more related to the user input.

The refinement of the Answer Retrieval Algorithm should always be a work in progress. As more systems are integrated, the broaden is the spectrum of paths from which to choose, to retrieve the best possible answer.

A more extent user evaluation would be useful, to determine where we can improve the effectiveness of our system. The definition of new tasks would also help us to prevent a higher range of different configurations.

New features of customization could be implemented, always with the goal to give the user more power of his own conversational agent. As the system gets bigger, with the addition of new plugins, there may be a way to create relations and volatile dependencies that could be defined by the user.

References

[1] David Ameixa. Sss: Say something smart. Master's thesis, Instituto Superior Tecnico, October 2013.

[2] Cátia Andreia Gomes Dias. Talkit: Desenvolvimento de um sistema de diálogo para português. Master's thesis, Instituto Superior Tecnico, May 2015.

[3] Pedro Fialho, Luisa Coheur, Sergio Curto, Pedro Claudio, Angela Costa, Alberto Abad, Hugo Meinedo, and Isabel Trancoso. Meet edgar, a tutoring agent at monserrate. *Association for Computer Linguistics*, August 2013.

[4] Daniel Filipe Nunes Magarreiro, Luisa Coheur, and Francisco S. Melo. Using subtitles to deal with out-of-domain interactions. In *SemDial 2014 - DialWatt*, August 2014.

[5] Pedro Mota. Addressing out-of-domain interactions in dialogue systems. *Instituto Superior Tecnico*.