Reengineering a domain-independent framework for Spoken Dialogue Systems

Filipe M. Martins, Ana Mendes, Márcio Viveiros, Joana Paulo Pardal, Pedro Arez, Nuno J. Mamede and João Paulo Neto
Spoken Language Systems Laboratory, L^{2}F – INESC-ID
Department of Computer Science and Engineering,
Instituto Superior Técnico, Technical University of Lisbon
R. Alves Redol, 9 - 2° – 1000-029 Lisboa, Portugal
{fmfm,acbm,mviveiros,joana,pedro,njm,jpn}@l2f.inesc-id.pt
http://www.l2f.inesc-id.pt

Abstract

Our work in this area started as a research project but when L^{2}F joined TecnoVoz, a Portuguese national consortium including Academia and Industry partners, our focus shifted to real-time professional solutions. The integration of our domain-independent Spoken Dialogue System (SDS) framework into commercial products led to a major reengineering process.

This paper describes the changes that the framework went through and that deeply affected its entire architecture. The communication core was enhanced, the modules interfaces were redefined for an easier integration, the SDS deployment process was optimized and the framework robustness was improved. The work was done according to software engineering guidelines and making use of design patterns.

1 Introduction

Our SDS framework was created back in 2000 (Mourão et al., 2004), as the result of three graduation theses (Cassaca and Maia, 2002; Mourão et al., 2002; Viveiros, 2004), one of which evolved into a masters thesis (Mourão, 2005). The framework is highly inspired on the TRIPS architecture (Allen et al., 2000): it is a frame-based domain-independent framework that can be used to build domain-specific dialogue systems. Every domain is described by a frame, composed by domain slots that are filled with user requests. When a set of domain slots is filled, a service is executed. In order to do so, the dialogue system interacts with the user until enough information is provided.

From the initial version of the framework two systems were created for two different domains: a bus ticket vending system, which provides an interface to access bus timetables; and a digital virtual butler named Ambrósio that controls home devices, such as TVs (volume and channel), acclimatization systems, and lights (switch on/off and intensity) through the X10 electrical protocol and the IrDA (Infrared Data Association) standard. Since 2003, Ambrósio is publicly available in the "House of the Future"[1], on the Portuguese Telecommunications Museum[2].

As proof of concept, we have also built a prototype system that helps the user while performing some task. This was tested for the cooking domain and the automobile reparation domain.

After the successful deployment of the mentioned systems, we began developing two new automatic telephone-based systems: a home banking system and a personal assistant. These are part of a project of the TecnoVoz[3] consortium technology migration to enterprises. To answer to the challenges that the creation of those new systems brought to light, the focus of the framework shifted from academic issues to interactive use, real-time response and real users. Since our goal was to integrate our SDS framework into enterprise products, we started the development of a commercial solution. Nevertheless, despite this

[1] http://www.casadofuturo.org/
[2] http://www.fpc.pt/
[3] http://www.tecnovoz.pt/

Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 68–76,
Columbus, Ohio, USA, June 2008. ©2008 Association for Computational Linguistics

new focus, we wanted to maintain the research features of the framework. This situation led to deep changes in the framework development process: as more robust techniques needed to be used to ensure that new systems could easily be created to respond to client requests. From this point of view, the goal of the reengineering process was to create a framework that provides means of rapid prototyping similar to those of Nuance[4], Loquendo[5] or Artificial Solutions[6].

Also, the new systems we wanted to built carried a significant change on the paradigm of the framework: while in the first systems the effects of users' actions were visible (as they could watch the lights turning on and off, for instance) and a virtual agent face provided feedback, in the new scenarios communication is established only through a phone and, being so, voice is the only feedback.

The new paradigm was the trigger to this process and whenever a new issue needed to be solved the best practices in similar successful systems were studied. Not all can be mentioned. The most relevant are described in what follows.

As it was previously mentioned, TRIPS was the main inspiration for this framework. It is a well known and stable architecture that has proven its merits in accommodating a range of different tasks (Allen et al., 2007; Jung et al., 2007). The main modules of the system interact through a Facilitator (Ferguson et al., 1996), similar to the Galaxy HUB[7] (Polifroni and Seneff, 2000) with KQML (Labrou and Finin, 1997) messages. However, in TRIPS, the routing task is decentralized since the sender modules decide where to send its messages. At the same time, any module can subscribe to selected messages through the Facilitator according to the sender, the type of message or its contents. This mechanism makes it easier to integrate new modules that subscribe the relevant messages without the senders' acknowledgment.

Like our framework, the CMU Olympus is a classical pipeline dialog system architecture (Bohus et al., 2007) where the modules are connected via a Galaxy HUB that uses a central hub and a set of rules for relaying messages from one component to the other. It has the three usual main blocks: Language Understanding, through Phoenix parser and Helios confidence-based annotation module, Dialogue Management, through RavenClaw (Raux et al., 2005; Bohus, 2004), and Language Generation, through Rosetta. Recognition is made with Sphinx and synthesis with Theta. The back-end applications are directly connected to the HUB through an included stub.

Some of our recent developments are also inspired in Voice XML[8], in an effort to simplify the framework parameterization and development, required in the enterprise context. Voice XML provides standard means of declarative configuration of new systems reducing the need of coding to the related devices implementation (Nyberg et al., 2002).

Our reengineering work aimed at: i) making the framework more robust and flexible, enhancing the creation of new systems for different domains; ii) simplifying the system's development, debug and deployment processes through common techniques from software engineering areas, such as design patterns (Gamma et al., 1994; Freeman et al., 2004).

By doing this, we are trying to promote the development and deployment of new dialogue systems with our framework.

This paper is organized as follows: Section 2 presents the initial version of the framework; Section 3 describes its problems and limitations, as well as the techniques we adopted to solve them; Section 4 describes a brief empirical evaluation of the reengineering work; finally, Section 5 closes the paper with conclusions and some remarks about future work directions.

2 Framework description

This section briefly presents our architecture, at its initial stage, before the reengineering process. We also introduce some problems of the initial architecture, as they will be later explained in the next section.

[4] `http://www.nuance.com/`

[5] `http://www.loquendo.com/`

[6] `http://www.artificial-solutions.com/`

[7] The Galaxy Hub maintains connections to modules (parser, speech recognizer, back-end, etc.), and routes messages among them. See `http://communicator.sourceforge.net/`

[8] `http://www.w3.org/Voice/`

2.1 Domain Model

The domain model that characterizes our framework is composed by the following entities:

Domain, which includes a frame realization and generalizes the information about several devices;

Frame, which states the subset of slots to fill for a given domain;

Device, which represents a real device with several states and services. Only one active state exists, at each time, for each device;

State, which includes a subset of services that are active when the state is active;

Service, which instantiates a defined frame and specifies a set of slots type of data and restrictions for that service.

When developing a new domain all these entities have to be defined and instantiated.

2.2 Framework architecture

Our initial framework came into existence as the result of the integration of three main modules:

Input/Output Manager, that controls an Automatic Speech Recognition (ASR) module (Meinedo, 2008), a Text-To-Speech (TTS) module (Paulo et al., 2008) and provides a virtual agent face (Viveiros, 2004);

Dialogue Manager, that interprets the user intentions and generates output messages (Mourão et al., 2002; Mourão, 2005);

Service Manager, that provides a dialogue manager interface to execute the requested services, and an external application interface through the device concept (Cassaca and Maia, 2002).

2.3 Input/Output Manager

The Input/Output Manager (IOManager) controls an ASR module and a TTS module. It also integrates a virtual agent face, providing a more realistic interaction with the user. The synchronization between the TTS output and the animated face is done by an audio–face synchronization manager, which generates the visemes[9] for the corresponding TTS phonemes information. The provided virtual agent face is based on a state machine that informs, among others, when the system is "thinking" or when what the user said was not understood.

Besides, a Graphical User Interface (GUI) exists for text interactions between the user and the system. Although this input interface is usually only used for test and debug proposes (as it skips the ASR module), it could be used in combination with speech, if requested by any specific multi-modal system implementation.

The IOManager provides an interface to the Dialogue Manager that only includes text input and output functions. However, the Dialogue Manager needs to rely on other information, such as the instant the user starts to speak or the moment a synthesized sentence ends. These events are useful, for instance, to set and trigger for user input timeouts.

2.4 Dialogue Manager

The architecture of the Dialogue Manager (Figure 1) has seven main modules: a Parser, an Interpretation Manager, a Task Manager, a Behavior Agent, a Generation Manager, a Surface Generation and a Discourse Context.

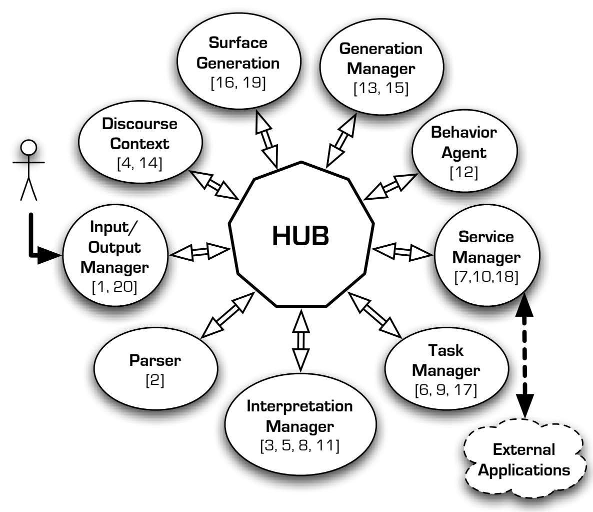

Figure 1: Dialogue Manager architecture through the central HUB. Numbers show the execution sequence.

[9]A *viseme* is the visual representation of a phoneme and is usually associated with muscles positioned near the region of the mouth (Neto et al., 2006).

These modules have specific code from the implementations of the two first systems (the bus ticket vending system and the butler). When building a generic dialogue framework, this situation turns out to be a problem since domain-dependent code was being used that was not appropriate in new systems. Also, the modules have many code for HUB messaging, which makes debug and development harder.

2.5 Service Manager

The Service Manager (Figure 2) was initially developed to handle all domain specific information. It has the following components:

Service Manager Galaxy Server, that works like a HUB stub, managing the interface with the devices and the Dialogue Manager;

Device Manager, that stores information related to all devices. This information is used by the Dialogue Manager to find the service that should be executed after an interaction;

Access Manager, that controls the user access to some devices registered in the system;

Domain Manager, that stores all the information about the domains. This information is used to build interpretations and for the language generation process;

Object Recognition Manager, that recognizes the discourse objects associated with a device;

Device Proxy, abstracts all communication with the Device Core and device specific information protocol. This is done through the *Virtual Proxy* design pattern

Device Core, that implements the other part of the communication protocol with the Service Manager and the Dialogue Manager.

Since the Service Manager interface is shared by the Dialogue Manager and all devices, a device can execute a service that belongs to another device or even access to internal Dialogue Manager information.

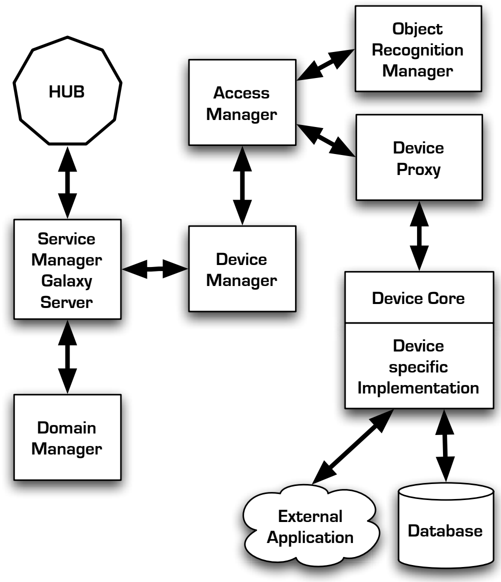

Figure 2: Service Manager architecture.

3 Reengineering a framework

When the challenge of building two new SDSs on our framework appeared, some of the mentioned architectural problems were highlighted. A reengineering process was critical. A starting point for the reengineering process was needed, even though that decision was not clear.

By observing the framework's data and control flow, we noticed that part of the code in the different modules was related with HUB messaging, namely the creation of messages to send, and the conversion of received messages into internal structures (*marshalling*). A considerable amount of time was spent in this task that was repeated across the framework.

Based on that, we decided that the first step should be the analysis of the Galaxy HUB communication flow and the XML structures used to encode those messages, replacing them with more appropriate and efficient protocols.

3.1 Galaxy HUB and XML

The Galaxy HUB protocol is based in generic XML messages. That allows new modules to be easily plugged into the framework, written in any programming language, without modifying any line of code. However, we needed to improve the development and debugging processes of the existing modules, and having a time consuming task that was repeated whenever two modules needed to communicate was a serious drawback.

Considering this, we decided to remove the Galaxy HUB. This decision was enforced by the fact that all the framework modules were written in the Java programming language, which already provides direct invocations and objects serialization through Java Remote Method Invocation (RMI).

The major advantage associated with the use of this protocol, was the possibility of removing all the XML-based messaging that repeatedly forced the creation and interpretation of generic messages in execution time. With the use of RMI, these structures were replaced by Java objects that are interchanged between modules transparently. Not only RMI is native to Java.

This was not a simple task, as the team that was responsible for this process was not the team who originally developed the framework. Because of this, the new team lacked familiarity with the overall code structure. In order to reduce the complexity of the process, it was necessary to create a proper interface for each module removing the several entry points that each one had. To better understand the real flow and to minimize the introduction of new bugs while refactoring the code we made the information flow temporarily synchronous.

The internal structure of each module was redesigned and every block of code with unknown functionality was commented out.

This substitution improved the code quality and both the development and the debugging processes. We believe that it also improved the runtime efficiency of the system, even though no evaluation of the performance was made. Empirically, we can say that in the new version of the system less time is needed to complete a task since no explicit conversion of the objects into generic messages is made.

3.2 Domain dependent code

The code of the Parser, the Interpretation Manager and the Surface Generation modules had domain dependent code and it was necessary to clean it out. Since we were modifying the Galaxy HUB code, we took the opportunity and redesigned that code in the aforementioned modules to make it more generic (and, consequently less domain dependent). Being so, the code cleaning process took place while the Galaxy HUB was being replaced.

We were unable to redesign the domain dependent code. Cases like hard-coded word replacement, used both to provide a richer interpretation of the user utterances and to allow giving a natural response to the user. In such cases, we either isolated the domain specific portions of the code or deleted them, even if the interpretation or generation processes were degraded. It can be recovered in the future by including the domain specific knowledge in the dynamic configuration of the Interpretation and Generation managers as suggested by Paulo Pardal (2007)

An example of this process is the splitting of the parser specific code into several parsers: some domain-dependent, some domain-independent, while creating a mechanism to combine them in a configurable chain (through a pipes and filters architecture). This allows the building of smaller data-type specific parsers that the Interpretation Manager selects to achieve the best parsing result, according to the expectations of the system (Martins et al., 2008). These expectations are created according to the assumption that the user will follow the mixed-initiative dialogue flow that the system "suggests" during its turn in the interaction. The strategy also handles those cases were the user does not keep up with those expectations.

3.3 Dialogue Manager Interface

The enhancements introduced at the IOManager level augmented the amount of the information interchanged between this module and the Dialogue Manager, as it could deal with more data coming from the ASR, TTS and the virtual agent face.

However, the Dialogue Manager Interface was continuously evolving and changing. This lack of stability made it harder to maintain the successive versions completely functional during the process.

Following the software engineering practices, and using the *Template Method* design pattern, we started with the definition of modules interfaces and only after that the implementation code of the methods was written. This allows the simultaneous development of different modules that interact. Only when some conflict is reported, the parallel development processes need to be synchronized resulting in the possible revision of the interfaces. Even when an interface was not fully supported by the Dialogue Manager, it was useful since it lead the IOManager continuous improvements and allowed simultaneous developments in the Dialogue Manager.

In order to ease the creation of this interface, an Input/Output adapter was created. This adapter makes the conversion of the information sent by the IOManager to the Dialogue Manager specific format. Having this, when the information exchanged with the Dialogue Manager changes, the Dialogue Manager Interface does not need any transformation. In addition, the Dialogue Manager is able to interact with other Input/Output platforms without the need of internal changes.

This solution for the interfaces follows the *Facade* design pattern, which provides an unique interface for several internal modules.

3.4 File system reorganization

When the different dialogue systems were fully implemented in the new version of the framework, we wanted to keep providing simultaneous access to the several available domains during the same execution of the system.

In fact, in our initial framework it was already possible to have several different domains running in parallel. When an interaction is domain ambiguous, the system tries to solve the ambiguity by asking the user which domain is being referred.

User:	*Ligar*
System:	*O que deseja fazer:* *ligar um electrodoméstico* *ou fazer um telefonema?*

Figure 3: Example of a domain ambiguous interaction while running with two different running domains. In Portuguese “*ligar*” means “*switch on*” and “*call*”

Consider the example on Figure 3: an user interaction with two different running domains, the butler and the personal digital assistant. In Portuguese, the verb “*ligar*” means “*to switch something on*” or “*to make a phone call*”. Since there are two running domains, and the user utterance is domain ambiguous, the systems requests for a disambiguation in its next turn (*O que deseja fazer*), by asking if the user wants to switch on a home device (*ligar um electrodoméstico*) or make a phone call (*fazer um telefonema*).

While using this feature, it came to our attention that it was necessary to reorganize the file system: the system folder held the code of all domains, and every time we needed to change a specific domain property, we had hundreds of properties files to look at. This situation was even harder for novice framework developers, since it was difficult to find exactly which files needed to be modified in that dense file system structure. Moreover, the ASR, TTS and virtual agent configurations were shared by all domains.

To solve this problem we applied the concept of *system–instance*. A *system–instance* has one or more domains. When the system starts, it receives a parameter that specifies which instance we want to run. The configuration of the existing instances is split across different folders. A library folder was created and organized in external libraries (libraries from an external source), internal libraries (library developed internally at our laboratory) and instance specific libraries (specific libraries of a *system–instance*).

With this organization we improved the versioning management and updates. The conflicting configuration was removed since each *system–instance* has now its own configuration. The configuration files are organized and whenever we need to deliver a new version of a *system–instance*, we simply need to select the files related with it.

3.5 Service Manager redesign

The Service Manager code had too many dependencies with different modules. The Service Manager design was based on the *Virtual Proxy* design pattern. However, it was not possible to develop new devices without creating dependencies on all of the Service Manager code, as the Device Core code relied heavily on some classes of the Service Manager.

This situation created difficulties in the SDSs development process and affected new developments since the Service Manager code needed to be copied whenever a Device Core was running in another computer or in a web container. This is a known bad practice in software engineering, since the code is scattered, making it harder to maintain updated code in all the relevant locations.

It was necessary to split the Service Manager code for the communication protocol between communication itself and the device specific code.

Also, the Service Manager class[10] interface was shared by the Dialogue Manager and all devices. Being so, it was possible that a device requested the execution of a service in other device, as well as to access the internal information exchanged between the Service Manager and the Dialogue Manager.

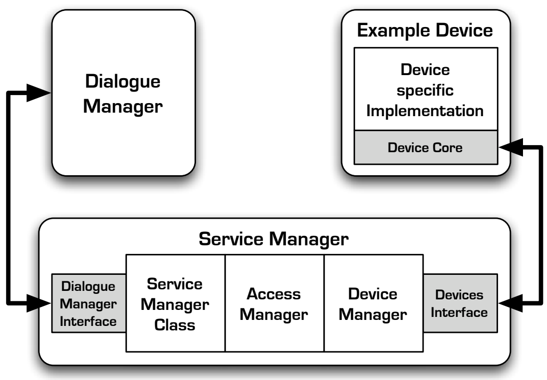

Figure 4: Service Manager architecture.

Like we did with the Dialogue Manager, we specified a coherent interface for the different Service Manager modules, removing the unwanted entry points. The Service Manager class interface was split and the Device Manager is now the interface between the Service Manager and the devices (Figure 4). Also, the Service Manager class interface is only accessed by the Dialogue Manager. The classes between the Service Manager and the Device implementation were organized in a small library, containing the classes and the Device Core code. This library is all what is needed to create a new device and to connect it to both the Service Manager and the Dialogue Manager.

Finally, we changed the Access Manager to control not only user access to registered devices, but also the registry of devices in the system. This prevents a device which is running on a specific *system–instance* to be registered in some other running *system–instance*. This module changed its position in the framework architecture: now it is between the Service Manager class and the Device Manager.

[10]The Service Manager Galaxy Server was renamed to Service Manager. However, we decided to call it here by Service Manager class so it will not be mistaken with the Service Manager module.

3.6 Event Manager

In the initial stage, when the Galaxy HUB was removed, all the communication was made synchronous. After that, to enhance the framework and allow mixed initiative interactions, a mechanism that provides asynchronous communication was needed. Also, it was necessary to propagate information between the ASR, TTS, GUI and the Dialogue System, crucial for the error handling and recovery tasks.

We came to the conclusion that most of the frameworks deal with these problems by using event management dedicated modules. Although TRIPS, the framework that initially inspired ours, has an Event Manager, that was not available in ours. The ASR and TTS modules provided already an event-based information propagation, and we needed to implement a dedicated module to make the access to this sort of information simpler. This decision was enforced by the existence of a requirement on handling events originated by an external Private Branch eXchange (PBX) system, like incoming call and closed call events. The PBX system was integrated with the personal assistant that is available through a phone connection. SDS.

We decided to create an Event Manager in the IOManager. The Dialogue Manager implements an event handler that receives events from the Event Manager and knows where to deliver them. Quickly we understood that the event handler needed to be dependent of the *system–instance* since the events and their handling are different across systems (like a telephone system and kiosk system). With this in mind, we implemented the event handler module, following the *Simple Factory* design pattern, by delegating the events handling to the specific system-instance handler. If this specific *system–instance* event handler is not specified, the system will use a default event handler with "generic" behavior.

This developments were responsible for the continuous developments in the IOManager, referred in section 3.3, and occurred at the same time.

With this approach, we can propagate and handle all the ASR events, the TTS events, GUI events and external applications events.

The Event Manager has evolved to a decentralized HUB. Through this, the sender can set identifiers in some events. These identifiers are used by other modules to identify messages relevant to them. In TRIPS a similar service is provided by the Facilitator, that routes messages according to the recipients specified by the sender, and following the subscriptions that modules can do by informing the Facilitator. This approach eases the integration of new modules without changing the existing ones, just by subscribing the relevant type of messages.

3.7 Dialogue Manager distribution

Currently, there are some clients interested in our framework to create their own SDS. However, since the code is completely written in Java, distributions are made available through `jar` files that can be easily decoded, giving access to the source of our code. To avoid this we need to obfuscate the code.

Even though obfuscation is an interesting solution, our code used Java's *reflexion* in several points. This technique enables dynamic retrieval of classes and data structures by name. By doing so, it needs to know the specific name of the classes being reflected so that the Java class loader knows where to find them. Obfuscation, among other things, changes class names and locations, preventing the Java class loader from finding them.

To cope with this additional challenge, the code that makes use of *reflexion* was replaced using the *Simple Factory* design pattern. This change allows the translation of the hard-coded names to the new obfuscated names in obfuscation time. After that, when some class needs to instantiate one of those classes that used reflection, that instance can be created through the proper factory.

4 Evaluation

Although a SDS was successfully deployed in our initial framework, which is publicly available at a Museum since 2003, no formal evaluation was made at that initial time. Due to this, effective or numeric comparison between the framework as it was before the reengineering work and as it is now, is not possible. Previous performance parameters are not available. However, some empirical evaluation is possible, based on generic principles of Software (re) Engineering.

In the baseline framework, each improvement, like modifications in the dialogue flow or at the parser level, was a process that took more than two weeks of work, of two software engineers. With the new version, similar changes are done in less than one week, by the same team. This includes internal improvements, and external developments made by entities using the system. The system is more stable and reliable now: in the beginning, the system had an incorrect behavior after some hours of running time; currently with a similar load, it runs for more than one month without needing to be restarted.

This is one great step for the adoption of our framework. This stability, reliability and development speed convinced our partners to create their Spoken Dialogue Systems with our framework.

5 Conclusions and Future Work

Currently, our efforts are concentrated on interpretation improvement and on error handling and recovery (Harris et al., 2004).

Currently, we are working on representing emotions within the SDS framework. We want to test the integration, and how people will react to a system with desires and moods.

The next big step will be the inclusion of an efficient morpho-syntactic parser which generates and provides more information (based on speech acts) to the Interpretation Manager.

Another step we have in mind is to investigate how the events and probabilistic information that the ASR module injects in the system can be used to recover recognition errors.

The integration of a Question-Answering (QA) system (Mendes et al., 2007) in this framework is also in our horizon. This might require architectural changes in order to bring together the interpretation and disambiguation features from the SDS with the Information Retrieval (IR) features of QA systems. This would provide information-providing systems through voice interaction (Mendes, 2008).

Another ongoing work is the study of whether ontologies can enrich a SDS. Namely, if they can be used to abstract knowledge sources allowing the system to focus only on dialogue phenomena rather than architecture adaptation, when including new domains (Paulo Pardal, 2007).

Acknowledgments

This work was partially funded by TECNOVOZ, PRIME National Project number 03/165.

It was also partially funded by DIGA, project POSI/PLP/14319/2001 of Fundação para a Ciência e Tecnologia (FCT).

Joana Paulo Pardal is supported by a PhD fellowship from FCT (SFRH/BD/30791/2006).

References

James Allen, Donna Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. 2000. An architecture for a generic dialogue shell. *Natural Language Engineering, Cambridge University Press*, 6.

James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. Plow: A collaborative task learning agent. In *Proc. 22th AAAI Conf.* AAAI Press.

Dan Bohus, Antoine Raux, Thomas Harris, Maxine Eskenazi, and Alexander Rudnicky. 2007. Olympus: an open-source framework for conversational spoken language interface research. In *Workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technology*, HLT-NAACL.

Dan Bohus. 2004. Building spoken dialog systems with the RavenClaw/Communicator architecture. Presentation at Sphinx Lunch Talk, CMU, Fall.

Renato Cassaca and Rui Maia. 2002. Assistente electrónica. Instituto Superior Técnico (IST), Universidade Técnica de Lisboa (UTL), Graduation Thesis.

George Ferguson, James Allen, Brad Miller, and Eric Ringger. 1996. The design and implementation of the TRAINS-96 system: A prototype mixed-initiative planning assistant. Technical Report TN96-5.

Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. 2004. *Head First Design Patterns*. O'Reilly.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Thomas Harris, Satanjeev Banerjee, Alexander Rudnicky, June Sison, Kerry Bodine, and Alan Black. 2004. A research platform for multi-agent dialogue dynamics. In *13th IEEE Intl. Workshop on Robot and Human Interactive Communication (ROMAN)*.

Hyuckchul Jung, James Allen, Nathanael Chambers, Lucian Galescu, Mary Swift, and William Taysom. 2007. Utilizing natural language for one-shot task learning. *Journal of Logic and Computation*.

Yannis Labrou and Tim Finin. 1997. A proposal for a new KQML specification. Technical Report CS-97-03, Computer Science and Electrical Engineering Department, Univ. of Maryland Baltimore County.

Filipe M. Martins, Ana Mendes, Joana Paulo Pardal, Nuno J. Mamede, and João Paulo Neto. 2008. Using system expectations to manage user interactions. In *Proc. PROPOR 2008 (to appear)*, LNCS. Springer.

Hugo Meinedo. 2008. *Audio Pre-processing and Speech Recognition for Broadcast News*. Ph.D. thesis, IST, UTL.

Ana Mendes, Luísa Coheur, Nuno J. Mamede, Luís Romão, João Loureiro, Ricardo Daniel Ribeiro, Fernando Batista, and David Martins de Matos. 2007. QA@L2F@QA@CLEF. In *Cross Language Evaluation Forum: Working Notes - CLEF 2007 Workshop*.

Ana Mendes. 2008. Introducing dialogue in a QA system. In *Doctoral Symposium of 13th Intl. Conf. Apps. Nat. Lang. to Information Systems, NLDB (to appear)*.

Márcio Mourão, Pedro Madeira, and Miguel Rodrigues. 2002. Dialog manager. IST, UTL, Graduation Thesis.

Márcio Mourão, Renato Cassaca, and Nuno J. Mamede. 2004. An independent domain dialogue system through a service manager. In *EsTAL*, volume 3230 of *LNCS*. Springer.

Márcio Mourão. 2005. Gestão e representação de domínios em sistemas de diálogo. Master's thesis, IST, UTL.

João Paulo Neto, Renato Cassaca, Márcio Viveiros, and Márcio Mourão. 2006. Design of a Multimodal Input Interface for a Dialogue System. In *Proc. PROPOR 2006*, volume 3960 of *LNCS*. Springer.

Eric Nyberg, Teruko Mitamura, and Nobuo Hataoka. 2002. DialogXML: extending Voice XML for dynamic dialog management. In *Proc. 2th Int. Conf. on Human Language Technology Research*. Morgan Kaufmann Publishers Inc.

Sérgio Paulo, Luís C. Oliveira, Carlos Mendes, Luís Figueira, Renato Cassaca, Céu Viana, and Helena Moniz. 2008. DIXI - A Generic Text-to-Speech System for European Portuguese. In *Proc. PROPOR 2008 (to appear)*, LNCS. Springer.

Joana Paulo Pardal. 2007. Dynamic use of ontologies in dialogue systems. In *NAACL-HLT Doctoral Consortium*.

Joseph Polifroni and Stephanie Seneff. 2000. GALAXY-II as an architecture for spoken dialogue evaluation. In *Proc. 2nd Int. Conf. Language Resources and Evaluation (LREC)*.

Antoine Raux, Brian Langner, Dan Bohus, Alan Black, and Maxine Eskenazi. 2005. Let's go public! taking a spoken dialog system to the real world. In *Proc. INTERSPEECH*.

Márcio Viveiros. 2004. Cara falante – uma interface visual para um sistema de diálogo falado. IST, UTL, Graduation Thesis.