QALL-ME
Question Answering Learning technologies in a multiLingual and Multimodal Environment

FP6 IST-033860

http://qallme.itc.it

The QALL-ME Architecture Design Issues and QA Framework

Authors: Günter Neumann, Christian Spurk, Bogdan Sacaleanu

Affiliation: DFKI

Keywords: *QALL-ME architecture, QA framework, QA components*

Abstract: *This deliverable describes the principles of the multilingual open-domain Question Answering framework used as the common Service Oriented Architecture (SOA). The QALL-ME framework consists of interacting Web services, leveraging increased flexibility by means of Metadata. The foreseen distributed QA architecture of the QALL-ME projects requires a high degree of modularity and a data and context driven major control flow. The major control flow is controlled by a flexible QA that is mainly data and context driven in the sense that the control flow is basically defined and triggered by the actual input, e.g., specific language and specific context restrictions.*

Date: 05/05/2007

Document Number: QALL-ME_D2.1_20070505

Status/Version: Final

Distribution Level: Public

Project Reference	FP6 IST-033860
Project Acronym	QALL-ME
Project Full Title	Question Answering Learning technologies in a multiLingual and Multimodal Environment
Distribution Level	Public
Contractual Date of Delivery	March, 2007
Actual Date of Delivery	May, 2007
Document Number	QALL-ME_D2.1_20070505
Type	Report
Status & Version	Final
Number of Pages	26
WP Contributing to the Deliverable	WP2: Architecture
WP Task responsible	DFKI
Authors	Günter Neumann, Christian Spurk, Bogdan Sacaleanu
Other Contributors	
Reviewer	Bonaventura Coppola (FBK-irst)
EC Project Officer	Erwin Valentini

Keywords: QA Architecture, QA framework, QA components

Abstract: This deliverable describes the principles of the multilingual open-domain Question Answering framework used as the common Service Oriented Architecture (SOA). The QALL-ME framework consists of interacting Web services, leveraging increased flexibility by means of Metadata. The foreseen distributed QA architecture of the QALL-ME projects requires a high degree of modularity and a data and context driven major control flow. The major control flow is controlled by a flexible QA that is mainly data and context driven in the sense that the control flow is basically defined and triggered by the actual input, e.g., specific language and specific context restrictions.

Summary

1 Introduction

QALL-ME aims at developing a shared infrastructure for multilingual and multimodal question answering (QA), which will include all the basic components that are required for providing the following capabilities:

- Automatically gathering, storing, and updating relevant information extracted from different (structured and non-structured) source data types;
- Dealing with complex multilingual questions, anchored to a spatial and temporal context;
- Dealing with both textual and spontaneous speech access modalities;
- Presenting users with correct, complete, and concise answers extracted from different multilingual source data types;
- Combining different output presentation formats (e.g., texts, maps, images).

This paper seeks to provide the basic architecture for such a QA infrastructure. Before we start, let's have a look at the central term *architecture* itself. The Open Group Architecture Forum (TOGAF) defines architecture as being the following:

> "The structure of components, their interrelationships, and the principles and guidelines governing their design and evolution over time."

Breaking it down even further, architecture is necessary to do the following:

- design and model at different levels of abstractions
- separate specification from implementation
- build flexible systems
- make sure requirements are addressed
- analyze the impact of a change in requirements

Figure 1 roughly shows the main modules of the distributed architecture which makes up the backbone of the QALL-ME service.

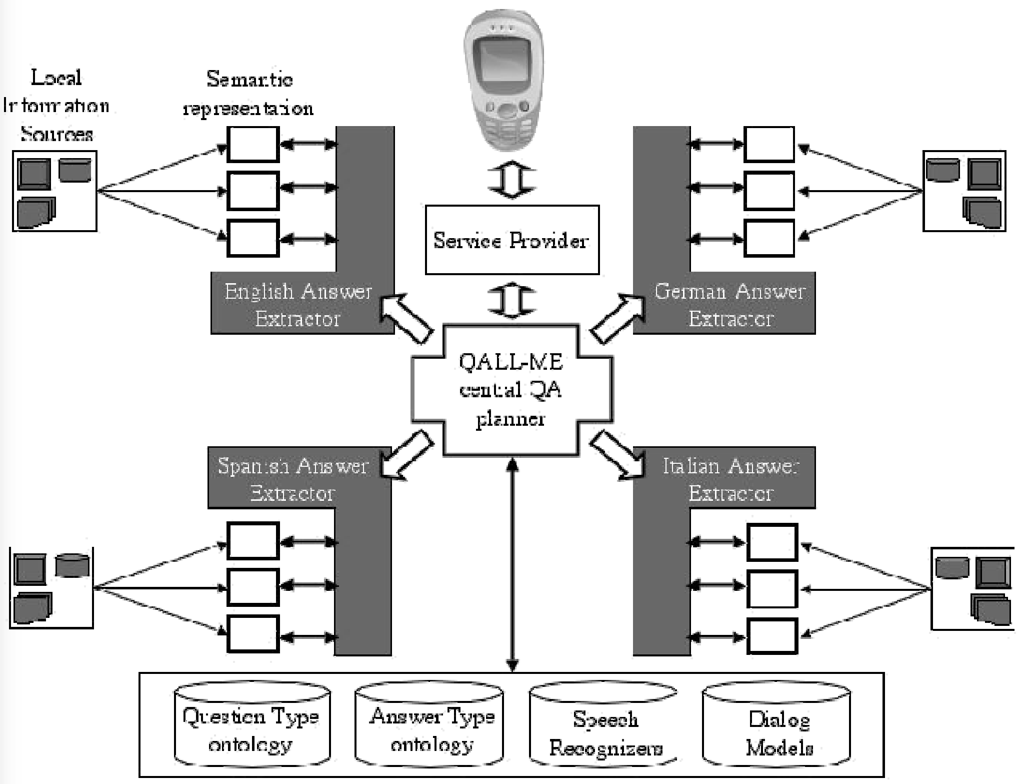

Figure 1: The QALL-ME distributed architecture.

Instead of developing different language specific end-to-end QA systems, QALL-ME aims at the development of a common QA architecture framework that supports a fine-grained integration of different language-specific components in a dynamic and data-driven manner. For example, if a German speaker is using the QALL-ME service in Trento to get information about some specific restaurant, then she can enter a natural language question in German. The QALL-ME system recognizes the language and passes the German question to the German question analysis tool (which is located on the German QA server in Saarbrücken) in order to get an internal meaning representation, which can be used to perform an information search using the local data provider in Trento. However, since it is quite likely that the data here is represented in the Italian language, the QALL-ME system first has to call a (probably externally available) machine translation (MT) service which translates the relevant part of the German question into Italian. Now, the QALL-ME system can call the local data provider in Trento to answer the question (using the Italian answering component located on the Italian QA server in Trento). In case the found answer is in Italian, the QALL-ME system first calls the MT service to translate the answer from Italian to German before it sends the answer back to the German speaker's mobile device.

1.1 Towards component-oriented QA architecture

In order to realize such a demanding QA scenario, a flexible dynamic information flow is needed. A closer look at the recent development of successful QA systems reveals a strong component-oriented perspective for the realization of the QA subtasks (commonly assumed major components are a question analysis component, a retrieval component, an extraction component, a selection component and a validation component). Based on our experience on large scale system development, the architectural framework will be specified from an abstract point of view using generic QA classes that define major input/output (IO) representations. This will also cover the specification of the major flow of interaction between components, ideally only on the level of the IO behavior between adjacent components. This might also involve the specification of alternative, competing components. For example, integration of Web-based search engines might be defined by a generic search engine class, which only defines basic means for mapping the internal representation of a Wh-question to a "syntax-free" IR-query. Through the definition of specific subclasses for different IR search engines, the specific syntactic representations are implemented.

We are assuming that orthogonal to the definition of the major QA classes, the major QA data objects are defined. By this we mean that the major data structure of a complete QA description is defined separately from the IO behavior of the QA components, but consistently defined with respect to them. It will now be possible to describe the QA system from two different sides (of the same coin) – the data and the process view. This means, for example, that a basic language independent representation of the internal question form is defined (capturing e.g., question type, focus, answer type etc.) that is considered as the necessary data exchange format, but not necessarily sufficient for a particular language. In that case, the corresponding language specific components have to provide the additional information, however, without affecting the generic standard. Note that this means, that we also consider additional (orthogonal) QA components, e.g., ontologies for question and answer types as abstract data types which have a language independent generic definition that might be specialized for the specific languages in use. However, major parts of the API are inherent and are thus treated identically.

1.2 Data-driven major control flow

Instead of hard-coding the QA information flow using programming language specific syntax, it will be specified in a declarative way in form of processing execution plans. An example of such execution plan might realize a standard QA-pipeline existing of a strict sequential ordering of standard QA components. However, there are other more complex execution plans possible, e.g., integration of feedback loops. Furthermore it might also be possible to define, for example, language-specific and even site-specific execution plans, each of them being aware of local (language- and resource-specific) control-flows without impacting the general architecture.

The major (and initial) control flow of the complete QALL-ME system is controlled by a central QA controller. Its main task is the evaluation of the major execution plan. We assume then that local execution plans are operated by local QA controllers, which might only run on specific sites. The QA controller is mainly data and context driven in the sense that the control flow is basically defined and triggered by the actual input, e.g., specific language and specific context restrictions.

As part of this framework, a QA specific episodic memory will be defined. It is very important that a QA system can acquire control information from past QA events in order to improve its future performance over new QA events (e.g., computed query-answer pairs). Two basic information sources are possible: either by obtaining situation-oriented information through interaction with the user (or from user-specific profiles), or obtaining that information by storing successfully computed query-answer pairs in an episodic memory. The former case requires query/answer clarification (query refinement, generation of paraphrases or query decomposition), and the latter involves acquisition of information concerning control strategies (selection between alternative processing components) and efficient data representation (Machine Learning of specific query sub-grammars and query-answer patterns). The episodic memory is also crucial for the development of QA-oriented discourse strategies on the basis of query/answer histories, and for statistical-based lexical and grammar induction.

The episodic memory will only store major input/output structures. For that reason it might not be detailed enough to perform a QA error analysis. Here, the definition of a specific monitoring component is vital. This component will maintain details reported by all QA components which have been affected when performing a QA cycle. This information covers the major input and output, but also details of intermediate results, error messages, parameter setting, space and time allocation, etc.

In the next sections we are describing the current state of the QALL-ME system architecture. We first begin by describing the basic QALL-ME system architecture, i.e., its components and the relations between them. We will then have a closer look at the proposed architecture model, the Service Oriented Architecture (SOA).

2 QALL-ME System Architecture Description

The major objectives concerning the development of the QALL-ME system architecture are the conceptualization, design, and implementation of a multilingual hybrid question answering system framework. It is multilingual in that the same core architecture is used for different languages (as those covered by the project participants) and it is hybrid in the sense that the same core architecture can be used in open-domain as well as domain-restricted QA applications. We use the term “core architecture” here in order to stress that we expect that for full-fledged end-to-end QA applications, language and/or task-specific adaptations and extensions of the existing QA framework might be necessary, but basically in a monotonic sense.

Since we aim at an open-source QA framework open-source standard NLP architectures (like IBM’s UIMA, cf. http://www.research.ibm.com/UIMA/) will be taken into account during the development phase of the QA framework. Furthermore, the descriptions of existing QA systems in the TREC, CLEF and the AQUANT conference proceedings reveal a continual improvement and refinement of existing QA components and the emerging of new QA components of different granularities. Thus, in order to be as flexible as possible for future developments we follow an *evolutionary* system design rather than a static one. Thus starting from a set of common basic QA components (largely based on the existing QA components already developed by the participating research groups), the QALL-ME QA system will be gradually adapted and refined during the project period.

2.1 System Component Breakdown and Basic Workflow

As a first step of outlining the QALL-ME system architecture we’ll have a look at the various components of the architecture. In section 2.1.1 we’ll have a top-down view of the system by naming and describing the components of the QA system and their respective tasks. After that we’ll have a look at the workflow between the system components. In section 2.1.2 we’ll see how these components interact and what the general information flow for a concrete example of an inquiry to the system looks like.

2.1.1 System Components

The QALL-ME system can be divided into several components. This breakdown can be more or less fine-grained. A first, very coarse-grained breakdown of the system is depicted in Figure 2.

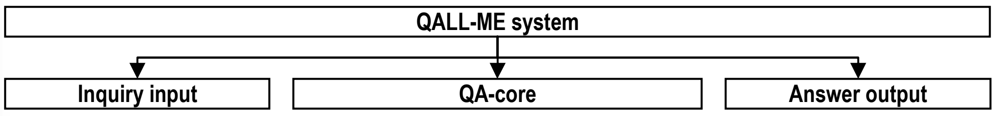

Figure 2: Coarse-grained component breakdown for the QALL-ME system.

Each of the three main components along with their subcomponents is described in more detail in the following sections. Note that the actual component breakdown of the QALL-ME system may be changing over time and that different subsystems may use other subcomponents than described in this basic component overview.

2.1.1.1 Inquiry Input Components

The inquiry input component and its subcomponents are responsible for receiving input for the whole QALL-ME system. Such input is not only the immediate user question but also the context of the inquiry. However, the task of the input component is not only to *receive* input, but also to *translate* the input into a suitable form which can be handled by the QA core system. Thus the inquiry input component along with its subcomponents constitutes a basic part of the system's interface to the end user and his environment.

Possible subcomponents of the inquiry input component are depicted in Figure 3 and described in more detail in the following.

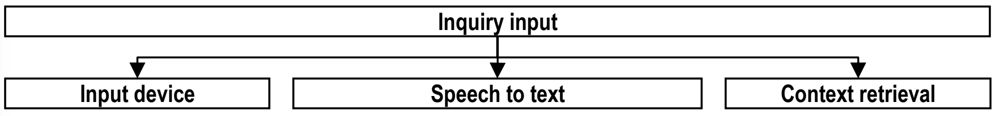

Figure 3: Possible subcomponents of the inquiry input component.

- *Input device component*: For QALL-ME this component might be some kind of mobile device or a web interface. The component handles the immediate receiving of the natural language user inquiry, e.g., in form of text messages (SMS), speech or keyboard input.
- *Speech to text component*: This component belongs to the group of quite specific components that are only used in conjunction with other specific components. A speech to text component translates speech signals to written text. So if an input medium component is used that receives written text input, a speech to text component is not required at all. For an input device with speech input, however, such a translation component is mandatory to supply the QA-core components with adequate text input.
- *Context retrieval component*: An inquiry is inherently dependent upon its context. Thus part of the inquiry input component needs to be some context retrieval component that introduces spatial and temporal data as well as information about the inquirer into the QA system.

2.1.1.2 QA-Core Components

The QA-core components, as the name suggests, constitute the core of the QALL-ME QA system. They assume to get adequate input from the inquiry input component and don't bother with answer presentation which is passed on to the answer output components. Nonetheless, the QA-core component and its subcomponents perform the principal work in the QALL-ME system. Their task is twofold: first of all the QA-core is responsible for *multilingual question interpretation*. Next they have to handle *crosslingual answer identification.*

Multilingual question interpretation component

The multilingual question interpretation component is responsible for extracting all relevant information from an inquiry which is useful and necessary to describe the answer of the inquiry as unambiguously and as precise as possible. This description has to have a certain format which can be understood by the answer identification components. The prevalent subcomponents of the multilingual question interpretation component are shown in Figure 4.

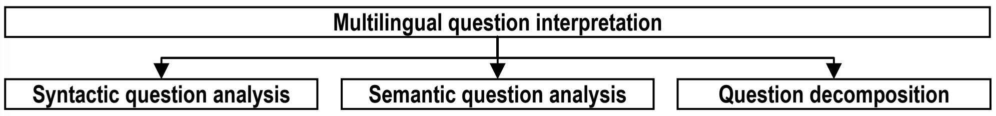
Multilingual question interpretation
Syntactic question analysis
Semantic question analysis
Question decomposition

Figure 4: Subcomponents of the multilingual question interpretation component.

- *Syntactic question analysis component*: Most QA systems will probably contain one or more components that perform a syntactic analysis of the input question. The depth of the analysis depends on the component itself as well as the other parts of the complete system that the component cooperates with.
- *Semantic question analysis component*: The principal goal of semantic question analysis components is the finding of the properties of the expected answer, such as type, quantity etc. Usually this does not involve a complete analysis of the question meaning.
- *Question decomposition component*: Questions may become almost arbitrarily complex, i.e., many questions are actually a bunch of multiple simpler questions. To handle such cases more easily, complex questions are often broken up or decomposed into simpler questions. This is done by the question decomposition components.

Other subcomponents of the multilingual question interpretation component might include some *discourse tracking component* that interprets an inquiry relative to its (discourse) context.

All the subcomponents mentioned so far are probably language specific and might exist in several versions, i.e., for several languages. Thus in a multilingual system and as part of a multilingual question interpretation component there might also be the need for a *language identification component* which analyses the raw input and assigns it the correct language flag. This flag might then be used by the language dependent components to determine whether they're responsible for some input or not.

Crosslingual answer identification component

The task of the crosslingual answer identification component is to retrieve answers that match the description of the question interpretation component and from these answers select the ones that fit best. The prevalent subcomponents of the crosslingual answer identification component are depicted in Figure 5.

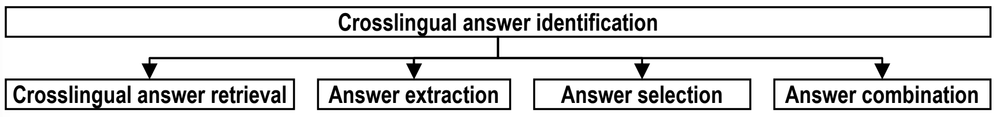
Crosslingual answer identification
Crosslingual answer retrieval
Answer extraction
Answer selection
Answer combination

Figure 5: Subcomponents of the crosslingual answer identification component.

- *Crosslingual answer retrieval component*: For each inquiry language there has to be an answer retrieval component which at first only retrieves documents that potentially contain an appropriate answer to the inquiry. Such a component might be a search engine on the web for example.
- *Answer extraction component*: A whole document as an answer is mostly not what is expected, so the output of the answer retrieval components has to be processed by another component. This answer extraction component takes an answer document and extracts only the relevant answer string, for example.
- *Answer selection component*: Mostly question interpretation, answer retrieval and answer extraction won't be hundred per cent correct and so it may be advantageous to have multiple answer candidates. The task of the answer selection component is now to rank these answer candidates and select the best of them.
- *Answer combination component*: Analogous to the question decomposition component, there may be an answer combination component which takes the answers for the parts of a decomposed question and combines them into an answer for the original, complex question.

As with the multilingual question interpretation component, there may be different versions of the answer identification subcomponents for different languages in the QALL-ME system.

2.1.1.3 Answer Output Components

Showing the results of the QA core component to the user is the responsibility of the answer output components. This involves preparing the raw answer for being understandable as well as translating the answer into a suitable format for the targeted output device. Finally, the output device itself is part of the answer output component. Possible subcomponents of the answer output component are depicted in Figure 6.

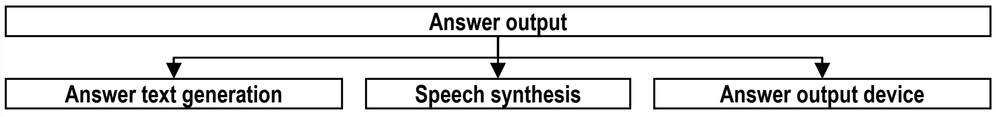

Figure 6: Possible subcomponents of the answer output component.

- *Answer text generation component*: In some cases the answer as returned by the QA core system might not be suited for directly being handed on to the user. In such cases it might be necessary to reformulate the answer, e.g., create a full sentence from only a bunch of words. Such work is performed by the answer text generation component.
- *Speech synthesis component*: Just like the speech to text component the speech synthesis component is only needed for certain output scenarios. If the answer output device is a phone line, then any answer text has to be converted to speech first in the speech synthesis component.

- *Answer output device component*: The answer output device component is comparable to the input device component and it is in many cases even the same device, actually. This component may be a phone line or a web interface for example and thus is another part of the interface to the end user of the QALL-ME system.

2.1.1.4 Other components

At some points in the previous sections we have already noted that the presented component overview will probably not be the final set of components for the QALL-ME system. These were somewhat rather the most prevalent components in a generic QA system. Implementation as well as the final focus will yield other components and make it necessary to remove or specialize some of the components we have seen so far.

One component that has not been considered so far but which definitely deserves a closer look in the QALL-ME project is a component for *mapping and geocoding*. In QALL-ME we want to take the situational context of the inquirer into account in several ways: on the one hand we want to be able to analyze questions in relation to their spatial context and on the other hand we'd like to provide personalized maps and location plans as answers. For such applications we need some specialized component that is not found in a classic QA system, the mapping and geocoding component.

Another component which is often needed in a crosslingual QA system is a *translation component* (that may be externally available already). A component that might prove useful in a QA some system might also be some kind of *episodic memory* (cf. section 2.2.2).

2.1.2 Component Workflow

In the previous section we have seen the principal components of the QALL-ME system. In the following we'll have a look at how these components interact. The intelligence that actually organizes this interaction is out of scope for this section; it's kind of an implementation problem rather than part of an architectural description.

The diagram in Figure 7 depicts the general workflow between the (meta) components of the QALL-ME system.[1] The detailed flow of information between the subcomponents follows in the upcoming subsections.

[1] This diagram – as well as all diagrams in the following sections – adhere to the following conventions: The (meta) component which is in the focus of the current figure is pictured at the top as a dark-gray box; the subcomponents of the focused component are represented as light-gray boxes below. Data flowing between the components is depicted as arrows which are sometimes labelled with exemplary data. In some diagrams on the left and/or on the right we have the previous/next (meta) component that passes on/receives data to/from (sub)components of the currently focused component; such components are depicted as dashed boxes. All data flows in the diagrams may potentially parallel; refer to the respective text for more precise information.

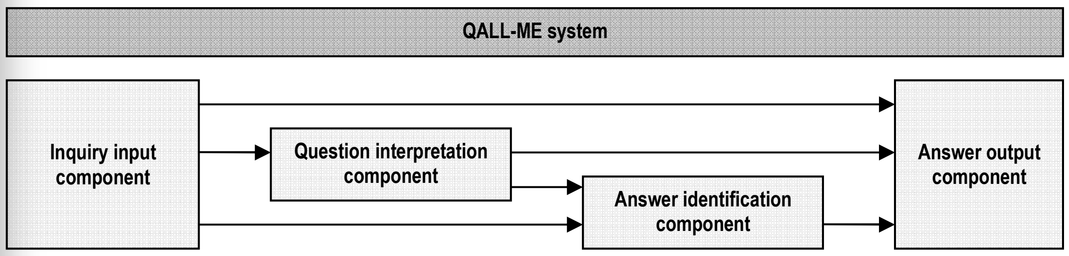
QALL-ME system
Inquiry input component
Question interpretation component
Answer identification component
Answer output component

Figure 7: General workflow in the QALL-ME system for the principal (meta) components.

For the workflow description we'll utilize an example scenario where some inquirer uses his mobile phone to call the QALL-ME system and ask the question: "Where can I eat pizza tonight?"

2.1.2.1 From the First System Interaction to a Complete Input Object

So what we have at the beginning of a user interacting with the QALL-ME system is a realization of the *input device component*, namely a mobile phone. This device receives the spoken question "Where can I eat pizza tonight?" and passes the audio recording on to the *speech to text component* for English. The latter translates the audio recording into written text and passes the result to the *QA core component*. At the same time, the *context retrieval component* starts retrieving the current time, saves the user ID for later reference and asks the input device for spatial information of the inquirer. All this information is passed to the QA core as well.

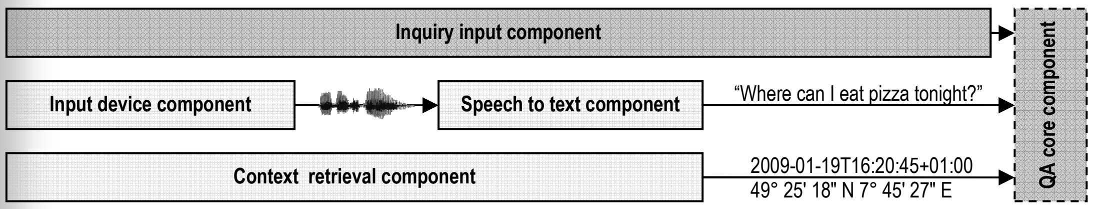
Inquiry input component
Input device component
Speech to text component
"Where can I eat pizza tonight?"
Context retrieval component
2009-01-19T16:20:45+01:00
49° 25' 18" N 7° 45' 27" E
QA core component

Figure 8: Exemplified workflow in the inquiry input component.

2.1.2.2 From an Input Object to a Complete Analysis of the Inquiry

So far, the QA system has converted the actual inquiry into a raw text written question along with information about the context. All of this data is used to analyze the inquiry syntactically now. The *syntactic question analysis component* might involve part of speech tagging, named entity recognition, parsing etc. Anyway, in the end there is an English syntactic analysis of the question which is passed on to the English *semantic question analysis*. The latter resolves spatial and temporal restrictions ("tonight" in our example) and retrieves the type of the expected answer (e.g., "restaurant" in our example), the number of required answers (in the example: one or more answers) etc. This information is saved as some kind of description of the expected answer for the next processing steps. In our example we have a simple question that does not need to be decomposed (cf. *question decomposition component*) and so the next step is the first part of the *answer identification component* for English. All collected and calculated information so far is passed on to this component.

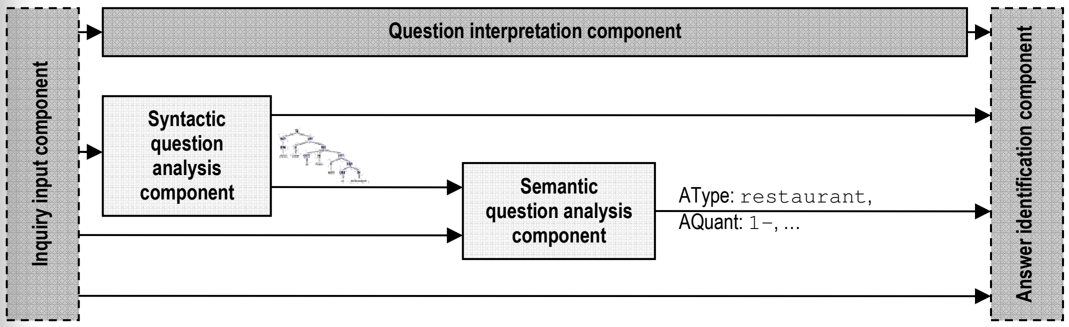
Question interpretation component
Inquiry input component
Syntactic question analysis component
Semantic question analysis component
AType: restaurant,
AQuant: 1-, ...
Answer identification component

Figure 9: Exemplified workflow in the question interpretation component.

2.1.2.3 *From an Inquiry Analysis to a Raw Answer*

In the answer identification component we have a detailed description of the answer from the previous steps and – if needed – a syntactic analysis of the inquiry as well as information about the inquiry context. On the way to the final answer the next step to go is using an *answer retrieval component* to get documents that potentially contain the answer. Using parts of the syntax analysis and the answer description a search query is built that leads to a bunch of documents. These documents are fed into the *answer extraction component* which extracts answer candidate strings from the document collection. The *answer selection component* again matches the answer description with the answer candidate strings and selects the best answers. This raw answer collection is now passed on to the *answer output component*.

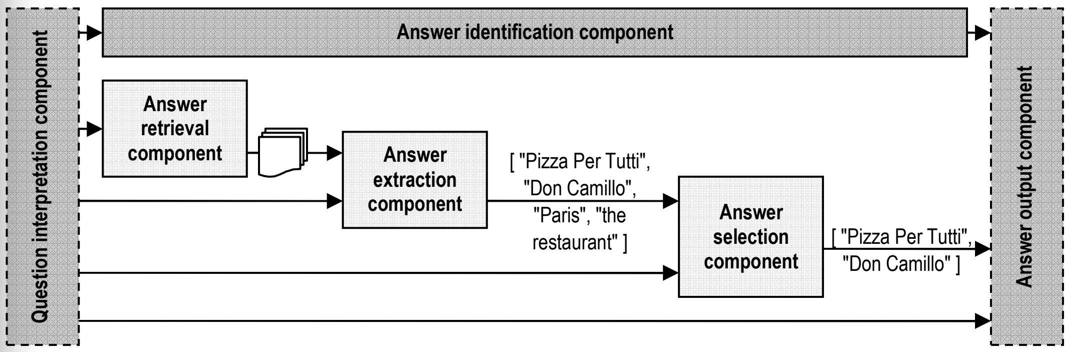
Answer identification component
Question interpretation component
Answer retrieval component
Answer extraction component
["Pizza Per Tutti", "Don Camillo", "Paris", "the restaurant"]
Answer selection component
["Pizza Per Tutti", "Don Camillo"]
Answer output component

Figure 10: Exemplified workflow in the answer identification component.

2.1.2.4 *From the Raw Answer to the Final Answer Presentation*

From the steps so far we have received a collection of raw answer strings; for our example, this might be `["Pizza Per Tutti", "Don Camillo"]`. Simply throwing this answer collection at the user is not what we want, so the *answer text generation component* builds a nice answer sentence from it, e.g., “Tonight you can eat pizza at Pizza Per Tutti and at Don Camillo.” We now almost have the final answer. Depending on the user’s profile that we got from the context retrieval component at the beginning, the output device is selected. If the user has set his preferred answer device to `text message` (SMS), then the answer text is directly passed on to a suitable *answer output device component*. Otherwise there might be yet another component involved, e.g., the *speech synthesis component* for converting the written answer text to speech for a mobile output device.

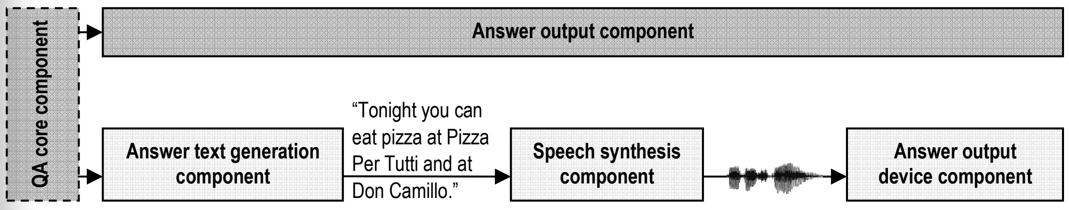
QA core component
Answer output component
Answer text generation component
"Tonight you can eat pizza at Pizza Per Tutti and at Don Camillo."
Speech synthesis component
Answer output device component

Figure 11: Exemplified workflow in the answer output component.

2.1.2.5 *Summary*

To conclude the description of the general workflow in the QALL-ME system, we summarize all subcomponents that were used in our example in Figure 12.

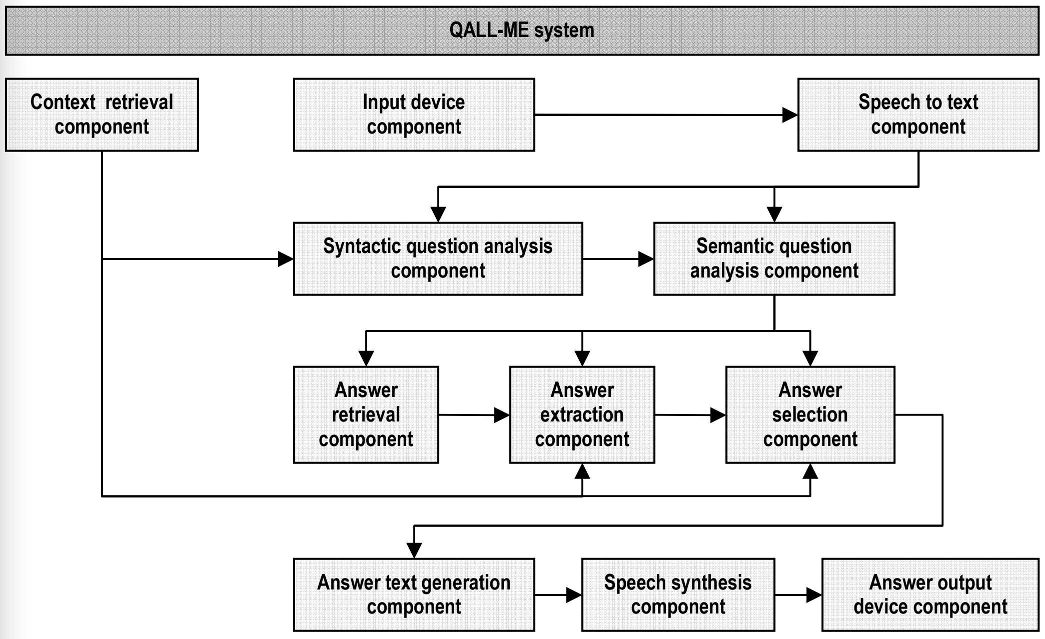
QALL-ME system
Context retrieval component
Input device component
Speech to text component
Syntactic question analysis component
Semantic question analysis component
Answer retrieval component
Answer extraction component
Answer selection component
Answer text generation component
Speech synthesis component
Answer output device component

Figure 12: Summarized workflow in the QALL-ME system for all described subcomponents.

2.2 Data-Driven View

The basic components that might be needed for the QALL-ME system architecture were described in the previous section in a relatively concrete manner. In the following section, we'll look at the architecture in a more data-driven, abstract manner. Figure 13 gives such a top-down perspective on the QALL-ME QA architecture.

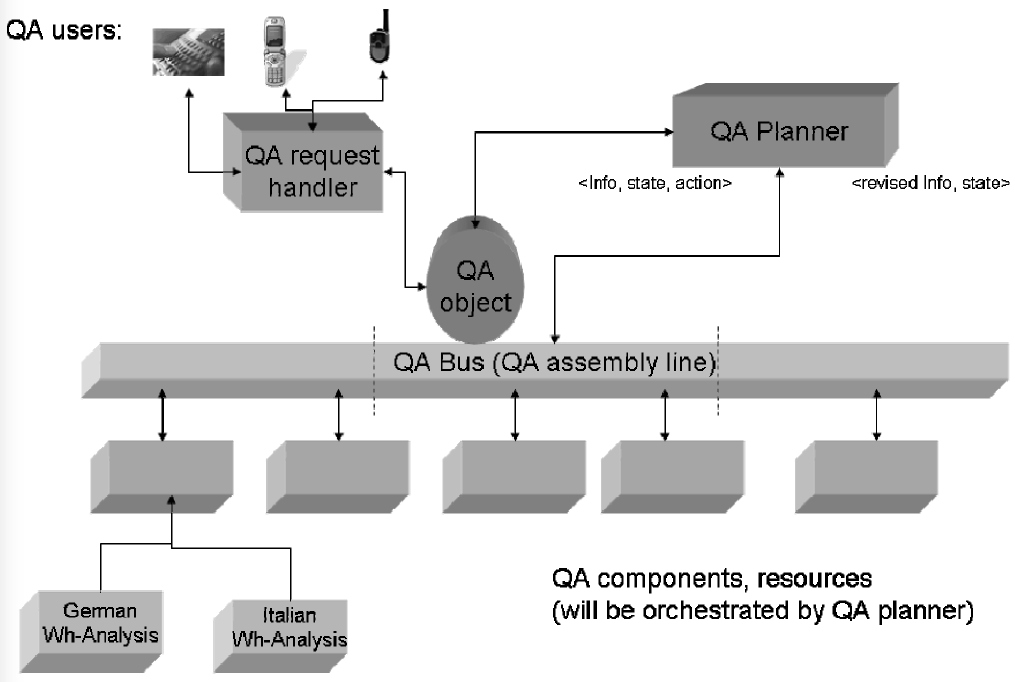

Figure 13: A top-down perspective onto the QALL-ME architecture.

At the core of the QALL-ME architecture is the `QA object` which is the basic interface for the user, the QA planner and the QA components. It represents the relevant information of a complete question-answering cycle in form of a quintuple

`<Context, Inquiry, IA-query, Answer, AnswerSource>`,

whereby:

- `Context`: is a substructure representing the identification of the user, the current time of point and the user's geographical location. The `Context` element is used for setting up the space-time and the position of the question-answering result in the overall structure of an interactive QA discourse.
- `Inquiry`: represents the Natural Language question input (speech and/or text), the source language and the relevant question Meta information, i.e., question type (e.g., definition/factoid/list question), expected answer type and the determined meaning representation of the NL question in form of a logical form. The `Inquiry` will be the basic means for initializing the QA planner and for selecting question-specific QA plans. Hence, a highly precise analysis of the `Inquiry` is needed for initializing successful answer search and extraction.
- `IA-query`: represents the query expression for the underlying search engines. The concrete form depends on the specific search engine and the representation of the information access (IA) units, which can range from a simple bag of words to a complex semantic search query, see below for some more details.
- `Answer`: the identified answer. Its concrete form depends of the question and expected answer type and ranges from a small text string (e.g., the textual representation of a person name), a list of text snippets (e.g., a list of textual descriptions for a definition question) to a multimedia object (e.g., a

picture or a map). Each individual answer will come with a score, so that it is also possible that a ranked list of answer candidates is delivered.

- AnswerSource: the sources from which the Answer elements have been extracted. The answer source will also encode its media type.

When the user enters a natural language question, the Context and Inquiry elements are instantiated first. We assume that the question type and the expected answer type are used to trigger specific answer extraction strategies or specific discourse strategies, e.g., the decomposition of the question or the activation of a clarification dialog. Furthermore, strategies must be developed that integrates the information from the Context element with the Inquiry element. For example, if the NL question is "How far is it from here?" and the current location is the DFKI building in Saarbrücken, then the question should be expanded to something like "How far is it from DFKI, Saarbrücken?".

Note that the IA-query is used as input to the data retrieval engine from which the answers to a NL question will be determined. As such the concrete form will be computed on basis of all information available in the Context and Inquiry slots of a QA object. Its concrete realization depends on the type of the initial access media, e.g., documents, data-base entries, XML annotated text files, etc. Thus possible realization forms are: standard IR query, SQL like expressions or XML queries, see Neumann and Sacaleanu (2005) for a possible realization of such an approach.

Further note that the use of our term QA component not only covers standard QA components, like question analysis, answer extraction or answer selection. We also consider complex QA components which might consist of a composition of "smaller" QA components. For example, a language-specific QA system can be considered as a QA component from our abstract architectural perspective. The main reason for this perspective is that we can not and we should not pre-define a certain QA granularity of QA components, because actually this is at least at the beginning of the project not possible.

2.2.1 The QA Bus (QA Assembly Line)

The QA Bus mirrors the structure of the OA object. It is the domain for all QA components, the execution manager of the central QA planner and the QA machine learning engines. From the perspective of a service oriented architecture (SOA), the QA Bus realizes the major implementation decisions for the integration of QA components, the registration of new QA components into the QALL-ME architecture and the data-oriented communication between QA components. For the integration of a QA component, the following features are required:

- Component identifier
- NL language
- Input and output types
- Description of its main functionality
- Server coordinates on which the QA component is running

For the integration of a QA component into the QALL-ME architecture, the SOA framework will provide the main web service specification and wrapper class definition, so that the new QA component can easily be integrated by providing a properly defined subclass, whose concrete implementation is the responsibility of the QA component developer. Thus, a QA component is defined and integrated into the QALL-ME architecture by means of:

- Component interface (the abstract data type of the component)
- Web service functionality (e.g., SOAP) which is automatically added to the component interface
- Definition of the main wrapper classes which implements the interface.

For example, the question analysis component might be defined by the following simple java interface

```
Interface QuestionAnalysis {
String process();
}
```

used to implement the following standard question analysis class

```
public class StandardQuestionAnalysis
        implements QuestionAnalysis {
        public QuestionAnalysis() {};
        public String process (String query)
        {return query;}
}
```

which is actually used for providing the interface for a language specific question analysis component:

```
public class GermanQuestionAnalysis
        extends StandardQuestionAnalysis {
        "German specific question analysis"
}
```

The concrete implementations are part of the SOA specification and mainly defined via the Web Service Description Language (WSDL), which is a declarative formalism for defining the QA component interfaces.

2.2.2 QA Planner and QA Episodic Memory

The QA planner manages the overall QALL-ME system integration and flow. It defines necessary conditions that have to be fulfilled to select and activate a basic QA plan, which executes step by step the call of all necessary QA components that are needed to answer a question. After each step it validates the utility of the returned partial result and either refines the current QA planning strategy or continues with the selected strategy. More precisely, the QA planner starts with an initial QA object that contains instances of the Context and Inquiry elements. The major goal of the QA planner is to answer a NL question as soon as possible. Thus, the initial strategy is:

1. Check whether the QA Episodic Memory already contains an answer for the question and return the "cached" answer or

2. Either call a question-type specific QA plan: This is one that has been automatically learned by the QALL-ME system to be successful for answering questions of a certain type (e.g., specific plans for answering factoid questions about persons).
3. Or start answering the question using a standard QA pipeline.

Step 1 means that the QA planner has found that for this question (or a paraphrase of this question), it already had found an answer in the past, and decides to return this answer again. This is a sort of QA caching, and hence, similar problems as known from web page caching have to be considered here as well, cf. (Wessels, 2001). Step 2 is actually an open research problem, and we are not aware of any existing proposal of a solution of it in the existing QA literature. Step 3 means that the QA planner uses as a default strategy a standard QA pipeline, which is assumed to be the most effective one to find an answer as soon as possible. Of course, it might be that during the execution phase new situations emerge which require the call of more complex strategies, but this is not known in advance. So seen by selecting a standard QA pipeline, the QA planner follows a *principle of parsimony*.

The QA Episodic Memory stores past successful QA results and selected plans. It mainly interacts with the QA planner, but is accessible also for other QA components and the QA Machine Learning Engine. However, rather than being a passive caching component, we consider it as being an *active part* of the QALL-ME system. It disposes of complex memory access functions, which support exact and partial matching for analogical reasoning (in order to support question paraphrases). The fact that the memory also caches selected plans means that strategies can be developed that realize an experience-based parameterization of QA planner decisions.

We assume that a QA plan consists of a sequence of calls of a state transition function of the form <current state, action, new state, weight>. Here a state is characterized by the QA object element and by additional parameters. The transition function specifies the actions that can be performed to reach a new state. We consider one call of the transition function as one atomic operation of a QA plan. For example, <Inquiry, create_IA_query, IA-query, 0.8> means that the current node is the Inquiry element of a QA object and when the function create_IA_query is called then the IA-query element is instantiated. As an alternative <Inquiry, expand_Inquiry, Inquiry, 0.2> will lead to an expansion of the Inquiry. The specified weights here mean that the activation of the first transition is more likely than the second one.

Thus, the main task of the QA planner is to determine and select the most "useful" sequence of atomic operations. Its main task is the overall controlling, selecting and invoking of QA components to maximize the expected utility of the information produced. The basic execution of a QA plan is performed by the QA execution manager. It performs one action of the current plan (selected by the QA planner), i.e., it realizes the transition from the current state of the plan to the next state. The QA planner tells the QA execution manager what component to call and what data to use. Plans are selected on the basis of the query analysis and the context information. In each step, the QA planner is able to determine the utility of the results computed by the QA execution manager. The QA planner can decide to depart from the initially selected QA pipeline in order to call additional "intermediate" components, such as error handling or feedback callers eventually causing complex user interactions.

A QA plan is not a fixed sequence of operations but more likely is a probabilistic state model, in the sense of a Partially Observable Markov Decision Process (POMDP, cf. Kaelbling et al. 1998). A major obstacle currently is how such an approach can be combined with the web service composition operation of a SOA-based approach. Here, a promising approach for defining complex control flow is the Business Process Execution Language (BPEL), which supports the declarative specification of composing web services. Unfortunately, BPEL does not seem to support probabilistic transitions, and hence it would not easily be possible to embed utility functions.

3 SOA – an Architecture Model for QALL-ME

The architectural style defining a Service Oriented Architecture (SOA) describes a set of patterns and guidelines for developing loosely-coupled, highly-reusable services that, because of separation of concern between description, implementation and binding, provide both increased interoperability and flexibility in responsiveness to changes and extensions.

SOA introduces a new abstract layer in terms of analysis, design and development: the service layer. Built upon existing technologies like object-oriented and component-based the service layer brings in a new solution that provides more coarse-grained implementations composed of reusable services, with well-defined, published and standards-compliant interfaces. (Figure 14)

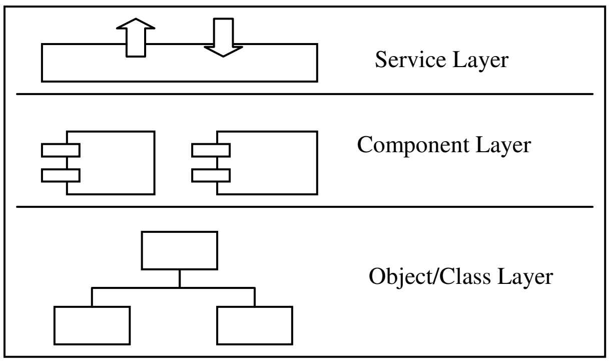

Figure 14: SOA layers of abstraction

The new artifact introduced, the service, represents another step forward in the evolution of software packaging beside functions and packages, objects and classes, and components. Services can provide an abstraction of specific component models, allowing users of these components to think only in terms of these new concepts and ignore specific details of the component model and how it is implemented. They separate the logic of a processing unit from the flow of control and routing, and the data and protocol transformation. This approach provides loose-coupling, making it much more reusable and flexible for integration compared to older technologies.

The key characteristics of a service could be defined as:

- A service is available at a particular endpoint in the network, and it receives and sends messages.
- The service has specific functionality specified through an interface.
- Interfaces and policies are published so that potential users of the service can discover and be given all the information they need to bind (perhaps dynamically) to that service.

- You can create new services from existing ones (orchestration) without leaving the service world (*programming in the large).*

Conceptually, we distinguish between two major units of programming within the service layer:

- *Services*: operations that reflect the functionality of lower layer units (components); they can be of two types:
    - *Atomic Services*: directly comparable to object-oriented methods, they have a specific interface and return structured responses.
    - *Services (composite)*: represent logical groupings of operations.
- *Business Processes*: consist of a series of operations which are executed in an ordered sequence (*choreography*) according to a set of predefined rules.

One distinguishing characteristic between these two concepts can be found in intended usage: processes are defined once and used ideally within a single context; services are defined once and reused many times over within diverse contexts such as different business processes, domains, and applications.

An abstract view of SOA could be presented as a layered architecture of composite services that align with the project's goals and builds upon existing components or create new artifacts. (Figure 15).

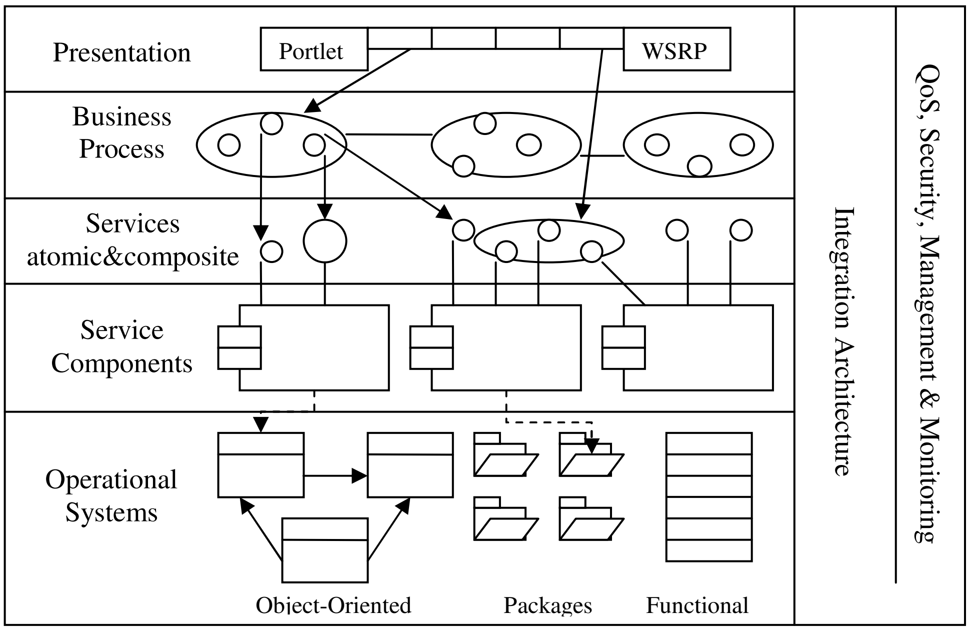

Figure 15: The layers of a SOA.

The relationship between services and components is that components (large-grained units of programming) implement the services and are in charge of delivering their functionality and maintaining their quality of service. Business process flows can be supported by choreography of those exposed services into composite applications. An integration layer supports the routing, mediation and translation of these services, components and flows using an Enterprise Service Bus (ESB). The deployed services must be monitored and managed for quality of service and adherence to non-functional requirements.

For each of the above depicted layers decisions in terms of design and architecture must be made. Therefore, we will approach each of the layers separately and present a list of items that should be considered toward a full SOA solution for a Question Answering system.

3.1 Scope

There are two underlying themes behind building an open source Question Answering framework: heterogeneity and change. They are relevant both to the developer of such a framework and to the user of it. The framework should allow for the integration of a range of different QA-systems, applications and architectures contributed by the partners of the consortium and based on best practice methods, as well as for a smooth integration of and full interoperability with user-developed components for adapting the framework to the user's needs. Moreover, the architecture of such a framework should sustain the cyclic development of the project with changing and updateable component requirements as the system evolves.

3.2 Operational Systems Layer

This layer consists of existing custom built applications or technology baseline contributed by the members of the Consortium that should be leveraged as much as possible. The applications cover all specific areas of the project including:

- Linguistic analysis
    - sentence splitters
    - tokenizers
    - PoS taggers
    - Shallow parsers
- Semantic analysis
    - Anaphora resolution
    - Word sense disambiguation
    - Temporal reasoning
    - Named Entity Recognition
- Linguistic resources
    - Lexical databases
    - Ontologies
- Machine learning tools
- Speech technologies
    - Automatic speech recognition
    - Dialogue systems
    - Human digital assistant
- Telephone infrastructure
    - Telephone platform
    - Multimodal interfaces
- Mapping and geocoding
    - Route calculation packages
    - Proximity search
    - Geocoding and reverse geocoding
    - Location based applications

3.3 Service Components Layer

This layer realizes the functionality of services, either by using one or more applications in the operational systems layer or providing new components. It typically uses container-based technologies such as application servers to implement the components, workload management, high-availability and load balancing.

This components either will be implemented along the development of the system or exist as custom built units in the available QA applications at each partner's site. Architectural decision concerning each of the coarse-grained enterprise components will be made according to the modeling technology chosen, be it object-oriented, functional or other.

Components in this layer will directly map to functionalities provided in the existing subsystems for multilingual question interpretation, data access, multilingual answer extraction, and multimodal interaction or they will implement new behavior according to the section 2.1.

3.4 Services Layer

This layer exposes to the user the underlying functionality of the system as either atomic or composite services. It also provides for the mechanism to take enterprise scale components and externalizes a subset of their interfaces in the form of service descriptions. The design strategy for this service layer consists in two steps: service identification and service specification.

The identification step consists of a combination of top-down, bottom-up and middle-out techniques of domain decomposition, existing applications analysis and goal-related modeling. The top-down process, known as domain-decomposition, consists of the break-down of the targeted domain (i.e., Question Answering) into its functional areas and subsystems, including its flow or process decomposition. In the bottom-up process, known as existing application analysis, existing systems are analyzed and selected as candidates for implementing underlying service functionality. The middle-out process, known as goal-related modeling, is covering still not captured services by either of the previous methods.

The service specification step aims to identify and specify components that will be required to realize services. One of the most important activities at this stage is to determine which services should be exposed and specify their interfaces and descriptions. The main characteristics of a service that should be considered for externalization are:

- Traceability: the service can be traced back to goals and objectives of the project.
- Stateless: the service should minimize the amount of information or state required between requests.
- Discoverable: the service should be exposed externally to the project and have well defined interface and description.
- Reusability: the service should serve the interest of other processes and be reused to this extent.

Figure 16 shows a breakdown of possible services and processes that can be defined in the context of a Question Answering system. The brighter components represent the services and the darker ones the processes.

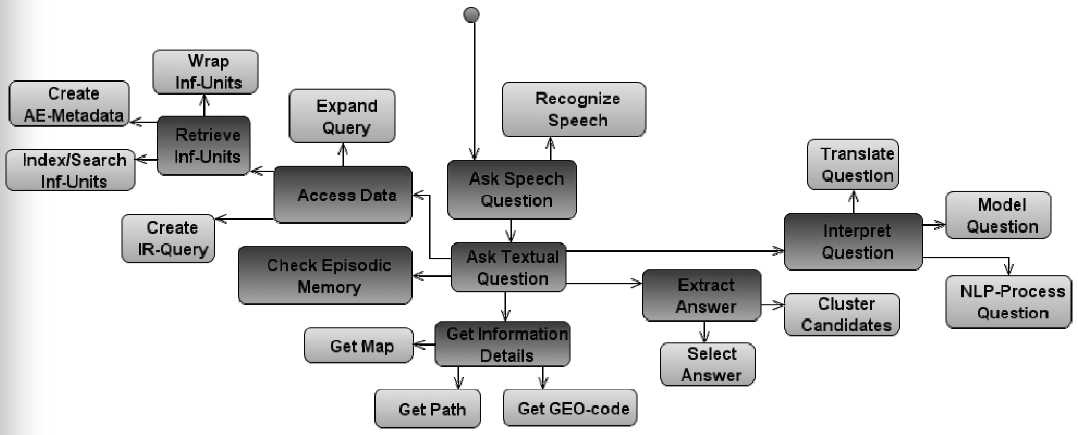

Figure 16: Services & Processes Breakdown.

3.5 Business Process Layer

This layer defines the operational artifacts that implement the business processes as choreographies of services. Services are bundled into a flow through orchestration or choreography and thus act together as a single application, which can be also exposed as a service at its turn. Figure 16 depicts some possible business processes that could be defined in a Question Answering system.

The QA Planner mentioned in section 2.2.2 will be defined in this layer as the aggregation of the major QA processes.

4 Web Services and SOA

Web Services technology is a collection of standards that can be used to implement an SOA. They provide a distributed computing approach for integrating heterogeneous applications over the Internet. The Web Services specifications are completely independent of the programming language, operating system and hardware to promote loose coupling between the service consumer and provider.

Web Services are self-contained, self-describing, modular applications that can be published, located and invoked over networks. They encapsulate functionality ranging from simple request-reply to full process interactions and can be new defined or wrap around existing applications.

4.1 SOA – Basic Components

At the most basic level, an SOA consists of the following three components (Figure 17):

- Service provider (Service)
- Service consumer (Requestor)
- Service registry (Discovery facility)

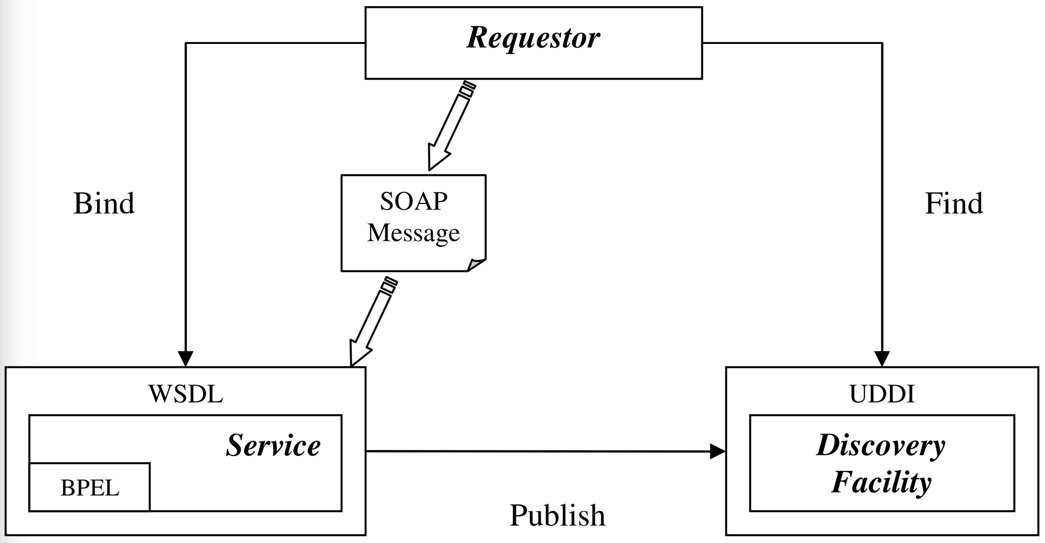

Figure 17: SOA components and operations.

The service provider creates a service and publishes its interface and access information (WSDL) to a service registry (UDDI). The service registry is responsible for making the service interface and implementation access information available to service consumers. The service consumer locates (finds) entries in the service registry and then binds to the service provider in order to invoke (SOAP) the defined service.

Following are the core technologies used for Web Services and their short description.

4.2 SOAP

Simple Object Access Protocol (SOAP) is a specification for the exchange of structured XML based messages between the service consumer, service provider and service registry. SOAP provides four main capabilities:

- A standardized message structure based on the XML Infoset
- A processing model that describes how a service should process the messages
- A mechanism to bind SOAP messages to different network transport protocols
- A way to attach non-XML encoded information to SOAP messages

4.3 WSDL

Web Services Description Language (WSDL) is an XML-based interface and implementation description language. It allows service authors to provide crucial information about the service so that others can use it. WSDL is what everyone uses to tell others what they can do with the service.

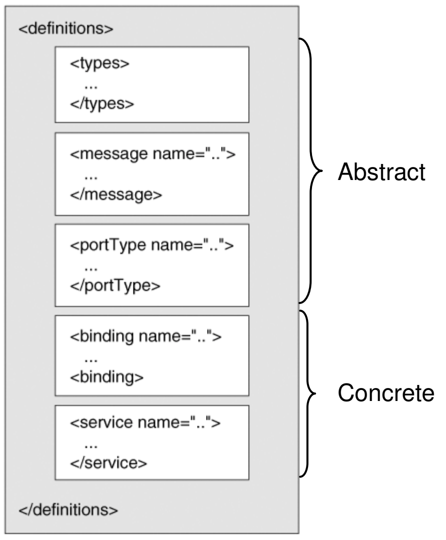
<definitions>
<types> ... </types>
<message name=".."> ... </message>
<portType name=".."> ... </portType>
<binding name=".."> ... <binding>
<service name=".."> ... </service>
</definitions>
Abstract
Concrete

Figure 18: Structure of WSDL document.

A WSDL document consists of two parts (Figure 18): a reusable abstract part and a concrete part. The abstract part of WSDL describes the operational behavior of Web services by describing the messages that go in and out from services. The concrete part of WSDL allows you to depict how and where to access a service implementation.

4.4 UDDI

Universal Description, Discovery and Integration (UDDI) is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web services. It plays a central role in the "service composition" process of creating new functionality by assembling existing services. UDDI provides a single well-known place that can be searched for services and provide pointers to more detailed information about the services, including the WSDL description.

UDDI is itself made up of three different elements:

- **A "white pages"** This contains the basic contact information for each Web service entry. It generally includes basic information about the company, as well as how to make contact.
- **A "yellow pages"** This has more details about the company. It uses commonly accepted industrial categorization schemes, industry codes, product codes, business identification codes and the like to make it easier for companies to search through the entries and find exactly what they want.
- **A "green pages"** This is what allows someone to bind to a Web service after it's been found. It includes the various interfaces, URL locations, discovery information and similar data required to find and run the Web service.

Entries are created using the Web Services Description Language (WSDL), and then send to a UDDI registry. UDDI allows registries to exchange entries with each other, so that if an entry is sent to one UDDI registry, it can be replicated to other registries.

4.5 BPEL

A BPEL process specifies the exact order in which participating web services should be invoked. This can be done sequentially or in parallel. With BPEL, we can express conditional behavior, for example, a web service invocation can depend on the value of a previous invocation. We can also construct loops, declare variables, copy and assign values, define fault handlers, and so on. By combining all these constructs, we can define complex business processes in an algorithmic manner.

BPEL is thus comparable to general purpose programming languages, but it is not as powerful as they are. On the other hand it is simpler and better suited for business process definition. Therefore BPEL is not a replacement but rather a supplement to modern programming languages.

References

- Kaelbling, L. P., M. L. Littman and A. R. Cassandra (1998): *Planning and acting in partially observable stochastic domains.* Artificial Intelligence, Volume 101, pp. 99–134.
- Neumann, G. and B. Sacaleanu (2005): *Experiments on Robust NL Question Interpretation and Multi-layered Document Annotation for a Cross-Language Question/Answering System.* In: C. Peters and others (eds.): Lecture Notes in Computer Science, Multilingual Information Access for Text, Speech and Images, CLEF 2004,volume 3491, p. 411–422. Berlin, Heidelberg: Springer.
- Wessels, D. (2001): *Web Caching*. O'Reilly and Associates (ISBN 1-56592-536-X).