A requirement analysis for an open set of human language technology tasks

Fredrik Olsson

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
fredrik.olsson@sics.se

Abstract

This work presents a requirement analysis and a design proposal for a general architecture for a specified, yet open set of human language technology (HLT) tasks — the set chosen is dubbed *information refinement*. Apart from using information refinement as a means to focus the requirement analysis and accompanying design proposal, the analysis and proposal are based on a survey of a number of projects that have had great impact on the realisation of today's HLT architectures, as well as on the experiences gained from a long-term case study aiming at composing a general purpose tool-kit for Swedish. The analysis and design are currently used in an ongoing effort at SICS to implement an open and general architecture for information refinement.

1. Introduction

During the last few years, the need for general, reusable software for computational linguistics and human language technology (HLT) has become widely acknowledged by the research community as well as by the industry. Usually, the overall motivation of striving for reusable software is to shorten the way from the origin of an idea to its implementation in a prototype system. Utilising reusable software also means that the effort spent in building an HLT system is reduced, and thus, that personal labour can be focused on more important issues.

The aim of this paper is to present a requirement analysis and design proposal for a specified, yet open set of human language technology tasks — *information refinement* is introduced as constituting a set of related tasks intended to serve as a target for developing a general and open architecture, Kaba. The requirement analysis and design proposal presented in sections 7. and 8. are based on three parts: the notion of information refinement (Section 2.); a survey of a number of projects and software that have had great impact on how HLT software is constructed today (TIPSTER, CLE, ALEP, GATE, DARPA Communicator, and ATLAS presented in Section 3.); and on the experience gained from a case study on constructing a language processing tool-set for Swedish in a national project called SVENSK (Section 4.). See (Olsson, 2002) for an elaboration on the requirements specification and design of an open architecture for information refinement.

2. The notion of information refinement

By the term information refinement, the *process* is referred to in which text is handled with the aim of *accessing* the pieces of content that are relevant from a certain *perspective* (Olsson et al., 2001).

Information access is about providing people with different tools and methods for granting reliable and simple access to the information they need, ideally with awareness of task and context of the access situation. A system for information access is intended to see to an expressed information need. Such a need is not always static — the *process* of searching for information is a dynamic one in which the information need, sources of information, characteristics of the task, and the type of text involved may change during a search session.

Since different readers have different information needs, prerequisites, and attitudes, they also have different *perspectives* when reading one and the same text. When considering that there are different perspectives, it is natural to think of information access and refinement systems as something that should not (only) deliver texts in their entirety, but rather in some sense understand the contents of the text and tailor the information according to the reader's perspective.

Information exctraction, information retrieval and automatic summarisation are all examples of human language techniques that fall under the information refinement category. Current work concerning information refinement at SICS include protein name tagging (Eriksson et al., 2002), information access using mobile services (Hulth et al., 2001), and support of professionals in information seeking (Hansen and Järvelin, 2000).

3. Some important HLT projects

This section introduces some of the software and projects that have, or have had, impacts on the ways today's software for HLT is designed and implemented. The survey of the literature in the area is not exhaustive, but merely provides an overview of the aspects and features of some important projects.

3.1. The TIPSTER architecture

The TIPSTER project (Grishman et al., 1997) was a joint effort between a number of U.S. government agencies led by DARPA and funded by CIA, DARPA, and DoD in collaboration with NIST and SPAWAR. The project started in 1991 and ended due to lack of funding in 1998.

The main focus of TIPSTER was to improve document processing efficiency and cost effectiveness, and in doing that, technologies such as information retrieval, information extraction, and automatic text summarisation were of great interest. There were two primary goals of the TIPSTER project, the first of which was to provide developers and users with an architecture that allowed for information retrieval in several gigabytes of texts, and the second goal was to provide an environment for research in document

detection and data extraction. However, by the time the project was discontinued, no fully implemented version of the TIPSTER architecture was produced.

3.2. CLE

SRI International's Cambridge Research Centre and Cambridge University's Computer Laboratory in 1985 suggested a UK-internal project developing a Core Language Engine (CLE), a domain independent system for translating English sentences into formal representations (Moore and Jones, 1985; Alshawi et al., 1992).

SRI's CLE built on a modular-staged design in which explicit intermediate levels of linguistic representation were used as an interface between successive phases of analysis. The CLE has been applied to a range of tasks, including machine translation and interfacing to a reasoning engine. Smith (1992) gives two examples of such systems; the LF-Prolog Query Evaluator and the Order Processing Exemplar (OPEX). The modular design also proved well suited for porting to other languages and the implementation was quite efficient. Thus, the project proved its purpose. However, even though the CLE system received considerable attention, it failed to spread in the community, the main reason being that it simply was too expensive to obtain it.

3.3. ALEP

The origin of the Advanced Language Engineering Platform (ALEP), the work on which started in 1991 and ended in 1995, was the issue of the lack of a general platform for research and development of large scale natural language processing systems (Simpkins, 1995; Bredenkamp et al., 1997). ALEP was an initiative of the Commission of the European Community (CEC) based on the experiences from the Eurotra and CLE projects.

ALEP was intended to function as a catalyst for speeding up the process of going from a research prototype of a system to a ready-to-ship product. The kind of users that ALEP first and foremost was targeted at were advanced experts, i.e., researchers in computational linguistics, possibly in conjunction with application developers. Simpkins (1995) expected that the openness of ALEP would attract users for research and development. Later, it turned out that this was not the case and ALEP never became widely spread.

3.4. GATE

Since the mid 90's, the General Architecture for Text Engineering (GATE) platform as reported on by, e.g., Cunningham (2000) is being developed at the University of Sheffield and funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC). GATE provides a communication and control infrastructure for linking together language engineering software. It does not adhere to a particular linguistic theory, but is rather an architecture and a development environment designed to fit the needs of researchers and application developers. GATE, currently available as version 2.0, is free for non-commercial and research purposes.

GATE supports reuse of resources, data as well as algorithms, since it provides for well-defined application programmers interfaces (APIs). Once a module has been integrated in the system, it is very easy to combine it with already existing modules to form new systems. Each component integrated into GATE has a standard I/O interface, which conforms to a subset of the TIPSTER annotation model. The infrastructure of GATE provides several levels of integration, reflecting how closely a new module should be connected to the core system.

3.5. The DARPA Communicator

Currently, the MITRE Corporation is (under DARPA funding) developing the DARPA Communicator. The goal of the DARPA Communicator is to set the scene for the next generation of conversational, multi-modal, interfaces to distributed information to be used in, e.g., travel planning, that require information from different sources to be combined.

The reference DARPA Communicator architecture builds on MIT's Galaxy-II system (Polifroni and Seneff, 2000; Seneff et al., 1999; Seneff et al., 1998). Among its key features, the authors list the ability to control system integration using a scripting language: each script includes information about the active servers, a set of operations supported by the server, as well as a set of programs. An in-depth explanation of the program control is given by Seneff et al. (1999). Essentially, the Galaxy-II system builds on a central process, the Hub, which mediates information between a number of different servers. The Galaxy-II system supports a wide range of component types, e.g., language understanding and generation, speech recognition and synthesis, dialogue management, and context tracking (Goldschen and Loehr, 1999).

There is a freely available, public version of the core DARPA Communicator.

3.6. ATLAS

The Architecture and Tools for Linguistic Analysis Systems (ATLAS) project is conducted by NIST, MITRE and LDC (Bird et al., 2000). The main goal is to develop a general architecture for annotation of linguistic data, including a formal/logical data format, a set of APIs, a tool-set, and persistent storage.

Within the ATLAS project, the participants are mainly interested in creating a formal framework for constructing, maintaining, and searching in linguistic annotations. In some aspects, the ATLAS annotation set model seems very similar to the TIPSTER annotation scheme. Bird and Liberman (2000) say that there are several ways of translating a TIPSTER-style annotation to a corresponding ATLAS one. In the end, the ATLAS working group concludes that TIPSTER-like annotations are not appropriate for audio transcriptions, except for "cases where such transcriptions are immutable in principle", (Bird and Liberman, 2000).

4. A case study — SVENSK

The SVENSK project was a national effort funded by the former Swedish National Board for Industrial and Techni-

cal Development (Nutek) and SICS addressing the problem of reusing language engineering software, see e.g., (Eriksson and Gambäck, 1997; Gambäck and Olsson, 2000). The SVENSK project was divided into three phases, spanning the spring of 1996 to the end of 1999. The aim has been to develop a multi-purpose language processing system for Swedish based, where possible, on existing components. Rather than building a monolithic system attempting to meet the needs of both academia and industry, the project has created a general tool-box of reusable language processing components and resources, primarily targeted at teaching and research.

The re-usability of the language processing components in SVENSK system arises from having each component integrated into GATE.

Collecting and distributing algorithmic resources and making different programs inter-operate present a wide range of challenges, along several different dimensions outlined next.

4.1. Diplomatic challenges

Making language processing resources freely available and, in particular, re-usability of resources is really a very uncommon concept in the computational linguistic community. Possibly this also reflects another uncommon concept, that of experiment reproducibility. In most research areas the possibility for other researchers to reproduce an experiment is taken for granted. It is even considered as the very core of what is accepted as good research at all. Strangely enough, this is seldom the case in computer science in general and even more rare within computational linguistics, perhaps because of tradition or lack of interest.

4.2. Technical challenges

From the technical point of view, one major conclusion is that the difficulties of integrating language processing software never can be over-estimated. Even when using a liberal architecture such as GATE it is hard work making different pieces of software from different sources and built according to different programming traditions meet any kind of interface standard.

In a way, it is understandable that academia does not always put much effort in packaging and documenting their software, since their main purpose is not to sell and widely distribute it. More surprising and discouraging, however, is that some of the actors on the commercial scene do not document their systems in a proper manner, either. Far too often this has resulted in inconsistencies with the input and output of other modules.

4.3. Linguistic challenges

Of course, language engineering components differ with respect to such things as language coverage, processing accuracy and the types of tasks addressed. It is also the case that tasks can be carried out at various levels of proficiency. The trouble is that there is no quality control available neither to the tool-box developer nor to the end-user. If a large number of language processing components are to be integrated, they should first be categorised so that components with a great difference in, say, lexical coverage are not combined.

A familiar problem for all builders of language processing systems relates to the adaptation to new domains. When reusing resources built by others this becomes even more accentuated, especially if a language engineering resource is available only in the black-box form (and thus relates to the issues of the previous subsection).

5. General observations and experiences

Below are some broad conclusions — focal points — drawn from the previous and present chapters, of what should be considered when creating a general HLT architecture:

1. **An architecture should be general with respect to a class of tasks, not to an entire field of research** The issue of *how* general an architecture should be needs to be considered since a too general one tends to be hard to handle.

2. **Keep the software open** There are various dimensions along which software could be considered open: distributing and licensing it; keeping its source open and inviting other people to participate in developing it; and to achieve software that are easily adaptable to new domains and types of information.

3. **Allow for use of existing programs as well as for the creation of system-specific ones** The potential drawback in using existing, externally produced software concerns issues such as, e.g., maintenance, fixing bugs, and extending/updating resources such as lexica and ontologies. All these things rely on the external program being supported by its producer.

4. **Support maintenance of systems and the components making them up** Develop tools and methods to support maintenance of components and systems, both on the linguistic level, e.g., integrated machine learning methods for lexical acquisition and grammar induction, and on the software level, e.g., new file formats and operating systems.

6. Motivation for a new architecture

The motivation for building a new architecture is primarily due to the fact that when information refinement emerged as a research area at SICS, there was no single architecture which fulfilled the demands that SICS's projects made at the time. In particular, no one of the existing platforms granted us full access to the source code and full distributional rights of the code, something which would be of great interest to us since we wanted to be able to distribute the source code of future information refinement systems freely, and since the functionality of the tools used for information refinement will have to be tuned to each new information refinement task. The latter may include changes to, e.g., the way the tools interact with each other and with the user, as well as the kind of data they produce — such changes may be difficult to achieve unless the software architecture hosting the tools is accessible at the source code level.

The work on a new HLT architecture called Kaba is an ongoing effort which was initiated in 1999 by Kristofer Franzén and Jussi Karlgren at SICS. At first, Kaba was intended to constitute an information extraction system for Swedish. An attempt at porting an existing information extraction system from English to Swedish turned out to be cumbersome (Franzén, 1999). Along the above lines, the conclusion was reached that future research in information refinement at SICS would benefit from a research vehicle having been built on site. Since 1999, the research focus has shifted slightly from information extraction to the more general goal of information refinement, which makes the need for an open and general architecture even clearer.

7. Requirement analysis

Deciding on what requirements are relevant for a given project tends to be a top-down process, going from broad issues such as, e.g., that the software under development should be portable to new operating systems, to splitting the portability into more specific sub-requirements. Requirements analysis always asks the *what*-questions regarding the software, e.g.: *what* equipment constraints exist, and *what* functions are to be incorporated. The *how*-questions are issued in the design phase described in Section 8..

Kaba is intended to function as a tool for developers of information refinement systems, first and foremost for research systems, but also for prototypes for testing ideas within information refinement. Kaba will *not* be a fixed set of tools for creating ready-to-ship products.

A typical Kaba user is a computational linguist with programming skills. This person's role is to use Kaba for the creation of information refinement systems to be used further in research and prototyping.

7.1. Project constraints and external factors

To accomplish the portability of Kaba on the software level, a widely supported programming language, such as Java, has to be used throughout the development process to implement all parts of the architecture. Further, Kaba will require (and presuppose) a linguistic processor that performs basic linguistic analysis of the texts to be processed, e.g., part-of-speech tagging and some fundamental grammatical analysis. Most likely the processor will be the Swedish and the English Functional Dependency Grammars (FDG) from Conexor Oy, Helsinki, Finland (Tapanainen and Järvinen, 1997).

Kaba must be implemented using a technology and an environment that facilitates easy integration of in-house or third party software for linguistic analysis as well as basic computational facilities, e.g., for reading and writing various file formats.

7.2. The scope of the work

Figure 1 shows three different ways that an information refinement system based on Kaba can interface with its environment, and thus gives some notion of what a developer of such a system has to deal with. What differs between the three constellations is the kind of user the system is intended for. In Figure 1 A, the system interacts with an information provider of some sort, e.g., a web site, a database,

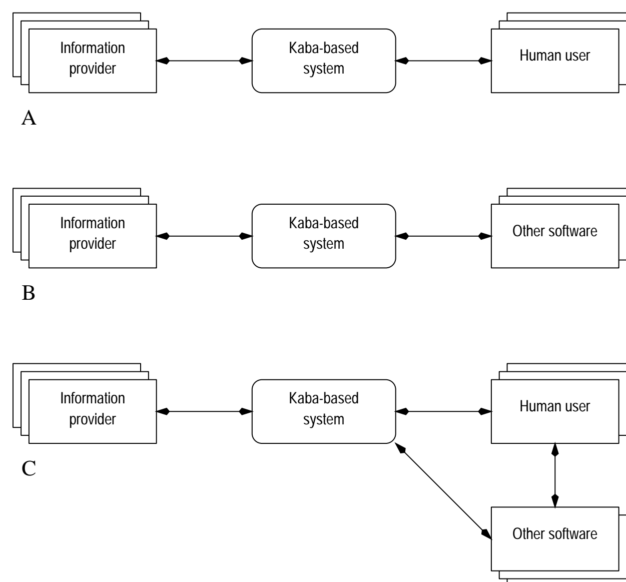

Figure 1: Characteristics of the environment of a Kaba-based system.

or a mobile service, on the one hand, and a human user on the other.

In Figure 1 B, the Kaba-based system communicates with the same kind of information provider as in Figure 1 A, but with another machine as counterpart instead of a human. The setup illustrates the case when a Kaba-based system is part of a larger system.

Finally, Figure 1 C, shows a configuration in which the system interacts with a human user as well as another machine.

7.3. The scope of the architecture

When starting to look at what a user may want to do with Kaba, it seems as a good idea to structure the requirements into what is commonly known as use cases (UC). Cockburn (1997) gives an overview of a method that deals with the identification and structuring of UCs. He defines a use case as being what happens when *actors* interact with a system to achieve a desired goal. An actor is an external entity (human or other software) that uses the system. In effect, UCs hold the functional requirements of a system in an easy-to-read format, and they represent the goal of an interaction between an actor and the system.

In total, 30 use cases have been identified for Kaba and seven of these constitute the top level of the use case hierarchy (Olsson, 2002):

UC 1: Develop an information refinement research and development prototype system.

UC 2: Evaluate an information refinement research and development prototype system.

UC 3: Port an existing system to a new domain or language.

UC 4: Document system.

UC 5: Maintain system.

UC 6: Create learning material or tutorial.

UC 7: Manage LR and PR components.

Use cases 1 and 7 each have several sub-goals which are illustrated in Figure 2 and Figure 3, respectively.

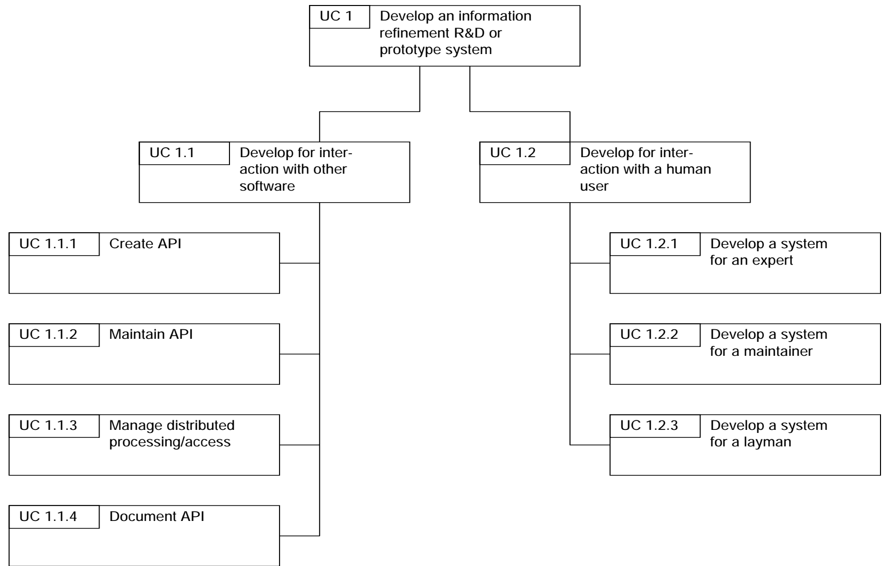

Figure 2: Schematic view of use case 1 and its sub-goals.

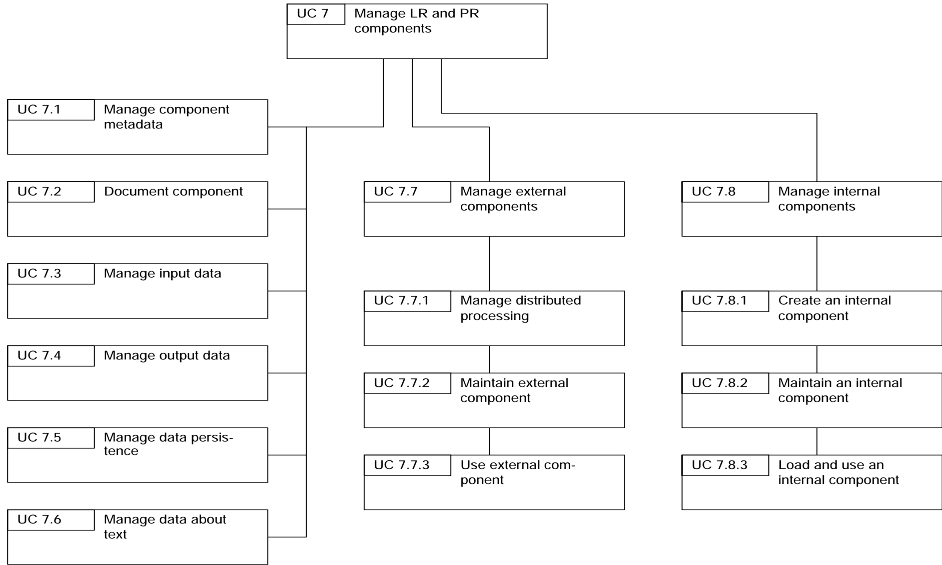

Figure 3: Schematic view of use case 7 and its sub-goals.

8. Design proposal

The design proposal is intended to give a hint as to how the requirement analysis could be realised.

8.1. Component metadata

This section covers use case 7.1 (*Manage component metadata*). Metadata about both language resources (LR) and processing resources (PR) is needed for several reasons, the first of which is to allow the developer (and the future users of the system) to browse a collection of components to see what components there are in order to build an information refinement system utilising existing components. In the same manner, metadata can be used to identify shortcomings of existing components and act as a basis for requirements analysis and specification when new language processing components need to be constructed or when new language resources need to be developed.

There are several means by which metadata can be expressed, and it seems natural to convey such data in the same format as the components themselves are annotated or produce annotations about text. Thus, the system internal format of metadata should correspond to the internal format of the data about text as described in Section 8.3., while the external format of metadata should agree with the format for data persistence described in Section 8.4..

8.2. Input and output

This section deals with use cases 7.3 (*Manage input data*) and 7.4 (*Manage output data*). The Kaba information refinement development platform presupposes that some sort of linguistic analysis has been performed on the text to be processed by a Kaba-based system. Currently, the FDG for English and Swedish are intended to be used, but it should also be possible to use any TIPSTER compliant linguistic processing component.

On the output side, a Kaba-based system should be able

to generate representations of the text it has processed in a format suitable to the user, regardless of whether the user is another computer program or a human.

8.3. System internal representation of annotated text

This section covers use case 7.6 (*Manage data about text*). Data about text can be expressed in various ways and the crucial point in all data representation is that it should facilitate rapid access to arbitrary pieces of information about the text. The representation formalism should allow for scaling up without causing the system's performance to drop.

While the format of the external and persistent data is like XML (see Section 8.4.), the internal representation is based on the TIPSTER annotation scheme. Although the two schemes are conceptually different the conversion between TIPSTER-style annotations and XML-based representations is quite straightforward.

8.4. Data persistence

This section deals with use case 7.5 (*Manage data persistence*). Data persistence is needed in order to provide Kaba with multiple-session capabilities, that is, to allow a user to work with the same source of information during several sessions and, in each session, having access to the results from the previous ones. The need for working in multiple sessions may occur, e.g., due to a system crash, for saving intermediate results, or simply because the user needs to interrupt the refinement process for other reasons.

The most suitable format is likely to be some instance of XML, partially because of the fact that it is becoming increasingly widespread in language engineering applications, and partially because there exist tools for manipulating and converting between different instantiations of XML.

8.5. Interacting with others

There are several aspects of interaction which have to be taken into account when designing an information refinement architecture like Kaba: (1) when a Kaba-based system is used by other software as a part of a larger system, (2) when a Kaba-based system utilises external components, both processing and data, as a part of an information refinement system, and (3) when a Kaba-based system needs to interact with human users.

Case (1) is reflected in use case 1.1.1 (*Create API*). In effect, what is required for a Kaba-based system to function in the context of a larger system, is a means for the developer of the larger system to have access to a restricted and well-defined set of the functionality in the Kaba-based system. Such access can be provided by means of a Java API.

Case (2) is addressed in use case 7.7.3 (*Use external component*) which concerns how to allow a Kaba-based system to use external components, i.e., components not primarily implemented for use within Kaba such as, for instance, part-of-speech taggers and ontologies. To allow Kaba to interact with external components, it is important that the components all look the same from Kaba's point of view. This means that the APIs that Kaba has to use to achieve this interaction have to be well defined and consistent.

Case (3) is addressed in use cases 1.2.1 (*Develop a system for an expert*), 1.2.2 (*Develop a system for a maintainer*), 1.2.3 (*Develop a system for layman*), all of which aim at facilitating interaction between different kinds of end-users and a Kaba-based system. Case (3) boils down to creating a connection between a tool or library for constructing GUIs, such as the Java Swing Classes (Topley, 1998), and Kaba.

8.6. Distributed processing

This section addresses use cases 1.1.3 (*Manage distributed processing/access*) and 7.7.1 (*Manage distributed processing*). In various settings, the parts making up a Kaba-based system need to be situated on different machines, connected by a network. One such setting occurs when some component, for example the one providing the initial linguistic analysis of input text, is available only for a particular operating system, while the rest of the system runs on another machine in the network. The different parts of the system then have to communicate using some protocol, e.g., SOAP.

8.7. Documentation and tutorials

This section addresses use cases 1.1.4 (*Document API*), 4 (*Document system*), 6 (*Create learning material or tutorial*), and 7.2 (*Document component*).

Kaba should come with incentives for developers, both of the Kaba architecture itself and of Kaba-based systems, to document their efforts. Such stimulus should be in the form of guide-lines and examples. There is a range of possible formats for documenting software systems, e.g., HTML and plain ASCII. It is also important that the guidelines are tied as little as possible to the chosen format. As for documenting the source code, existing tools such as Javadoc should be used.

Examples and tutorials should be encouraged by providing templates, example examples and tutorials to Kaba users and system developers.

8.8. Creating internal components

This section deals with use case 7.8.1 (*Create an internal component*). In Kaba, an internal component is one that is under the control of the developer in that it provides him with a more elaborate API than external components do. Typically, an internal component is created explicitly for use within a Kaba-based system.

A variant of the Common Pattern Specification Language (CPSL) called Kaba Pattern Specification Language (KPSL) will form the base formalism in which the functionality of the internal components will be expressed. CPSL is an effort by the TIPSTER working group that, unfortunately, has not been officially released. However, Appelt (1999) as well as Cunningham et al. (2000) present implementations of annotation engines based on CPSL. Essentially, a CPSL rule describes a finite state transducer for TIPSTER annotations.

It should be possible to construct internal components in several ways, for instance by hand-crafting rules using

a graphical rule editor, or by breeding them using machine learning methods.

8.9. Loading and using internal components

This section deals with use case 7.8.3 (*Load and use an internal component*). Once the KPSL rules making up a component have been developed, they are turned into Java code by a KPSL rule compiler. Along with the compiler come Java classes that facilitate dynamic loading of compiled sets of rules. Thus, as long as the KPSL rules have been compiled to Java and the Kaba-based system knows where to find the components, there are means by which they can be dynamically loaded into the system at run time.

8.10. Maintenance

The fundamental question when it comes to maintenance of any software is *When is maintenance necessary for this piece of software in this particular setting?* and, in the context of information refinement systems, this calls for well-defined criteria that can be used to probe the system's performance with respect to the task it is supposed to accomplish, or the system's affordance with respect to the users' expectations as to what the system is really supposed to do.

8.10.1. Maintenance of external components

This section addresses use cases 1.1.2 (*Maintain API*) and 7.7.2 (*Maintain external component*). The cases are closely related in that communication between a Kaba-based system and other software will always take place via some kind of API. Thus, maintaining an external component is in many cases the same as maintaining the API that Kaba uses for communicating with that component.

8.10.2. Maintenance of internal components

This section deals with use case 7.8.2 (*Maintain an internal component*). Maintenance of internal components should be facilitated by a graphical interface for inspecting, editing, loading, executing, and evaluating KPSL rules with respect to some success criteria set up for the component. It should be possible to do all this using the same, or a similar, graphical interface as when creating internal components.

8.10.3. Maintenance of systems

This section deals with use case 5 (*Maintain system*) which involves all other kind of maintenance mentioned previously in this section, i.e., maintenance of component APIs and external components (Section 8.10.1.), as well as of internal components (Section 8.10.2.). In addition, maintenance of systems also involves taking care of the whole formed by the pieces, e.g., seeing to it that the documentation is up to date, installing new software when needed, and monitoring the system's performance on a regular basis. This should be supported in the same way as maintenance of external components is, e.g., by giving guidelines for how to integrate the documentation of the parts into a central repository, and collect information about availability of new components.

8.11. Providing support for porting systems to new domains

This section deals with use case 3 (*Port an existing system to a new domain or language*). While maintenance may accommodate correction of minor changes to a system, there will also be occasions when the shift of domain or information need is so different from that captured by an existing system that maintenance of the system or one of its components is not enough to compensate for it. In these cases, the question of whether to use an existing system or to create a new one from scratch arises. One of the issues of providing support for porting systems to new domains and needs should be to supply the developer with clues for deciding the answer to that question. If the answer is that an existing system could probably be altered (ported) to meet the new needs, then the follow-up question should be: *What parts of the existing system can be re-used, and to what degree do they need to be modified?* Again, Kaba should provide methods that makes answering this question easier.

8.12. Providing support for evaluation

This section addresses use case 2 (*Evaluate an information refinement R&D or prototype system*). Evaluation of information refinement systems is a crucial issue in several aspects. The basic support for evaluation of information refinement systems can be of two kinds: by providing linguistically annotated data that act as a key to the questions for which a system is to be evaluated, or by providing tools that presuppose the presence of an answer-key for comparing data structures and calculating measurements of performance. Both kinds of support are necessary. In the former case, machine learning methods are often used as an aid in obtaining the correctly annotated corpora constituting the answer-key. In the latter case, the comparison of data structures should yield values in an appropriate metric, e.g., precision and recall, depending on the features that are evaluated.

9. Conclusions

When developing a general tool or architecture, it is possible to focus the technical and linguistic efforts in several ways. The most obvious one is to formulate and maintain an explicit goal regarding the kind of tasks that programs developed within the general architecture at hand should cope with. By obtaining and focusing on the goal at an early stage in the development of the open architecture, one can avoid ending up with a definition and design of a far too general system: when it comes to generality for language engineering, it should be with respect to a class of tasks, rather than to the field as such.

Acknowledgements

Lars Borin, Björn Gambäck, Kristofer Franzén, Jussi Karlgren, Preben Hansen, Gunnar Eriksson, Mikael Eriksson, Anette Hulth, Mark Tierney, Anna Jonsson.

10. References

Hiyan Alshawi, David Carter, Jan van Eijck, Björn Gambäck, Robert C. Moore, Douglas B. Moran, Fernando

C. N. Pereira, Stephen G. Pulman, Manny Rayner, and Arnold G. Smith. 1992. *The Core Language Engine*. MIT Press, Cambridge, Massachusetts, March.

Douglas E. Appelt, 1999. *The Complete TextPro Reference Manual*, June.

Steven Bird and Mark Liberman. 2000. A Formal Framework for Linguistic Annotation. *Speech Communication*, 33(1,2):23–60.

Steven Bird, David Day, John Garofolo, John Henderson, Christophe Laprun, and Mark Liberman. 2000. ATLAS: A flexible and Extensible Architecture for Linguistic Annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 1699–1706, Athens, Greece, June.

Andrew Bredenkamp, Thierry Declerck, Frederik Fouvry, Bradley Music, and Axel Theofilidis. 1997. Linguistic Engineering using ALEP. In R. Mitkov and N. Nicolov, editors, *Proceedings of the 2nd International Conference on Recent Advances in Natural Language Processing*, pages 92–97, Tzigov Chark, Bulgaria, September.

Alistair Cockburn. 1997. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, Sep-Oct and Nov-Dec.

Hamish Cunningham, Diana Maynard, and Valentin Tablan. 2000. JAPE: A Java Annotation Patterns Engine. Technical Report CS–00–10, University of Sheffield, Department of Computer Science, Sheffield, UK. Second Edition.

Hamish Cunningham. 2000. *Software Architecture for Language Engineering*. Ph.D. thesis, University of Sheffield, UK.

Mikael Eriksson and Björn Gambäck. 1997. SVENSK: A Toolbox of Swedish Language Processing Resources. In R. Mitkov and N. Nicolov, editors, *Proceedings of the 2nd International Conference on Recent Advances in Natural Language Processing*, pages 336–341, Tzigov Chark, Bulgaria, September.

Gunnar Eriksson, Kristofer Franzén, Fredrik Olsson, Lars Asker, and Per Lidén. 2002. Exploiting Syntax when Detecting Protein Names in Text. In *Proceedings of Workshop on Natural Language Processing in Biomedical Applications*, Nicosia, Cyprus, March.

Kristofer Franzén. 1999. Adapting an English Information Extraction System to Swedish. In *Proceedings of the 12th Nordic Conference of Computational Linguistics*, pages 57–65, Norwegian University of Science and Technology, Trondheim, Norway, December.

Björn Gambäck and Fredrik Olsson. 2000. Experiences of Language Engineering Algorithm Reuse. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation*, volume 1, pages 161–166, Athens, Greece, May. ELRA.

Alan Goldschen and Dan Loehr. 1999. The role of the DARPA Communicator Architecture as a Human Computer Interface for Distributed Simulations. In *Spring Simulation Interoperability Workshop*, Orlando, Florida, USA, March. Simulation Interoperability Standards Organization (SISO).

Ralph Grishman, Ted Dunning, Jamie Callan, Bill Caid, Jim Cowie, Louise Guthrie, Jerry Hobbs, Paul Jacobs, Matt Mettler, Bill Ogden, Bev Schwartz, Ira Sider, and Ralph Weischedel, 1997. *TIPSTER Text Phase II Architecture Design. Version 2.3*. New York, New York, January.

Preben Hansen and Kalervo Järvelin. 2000. The Information Seeking and Retrieval Process at the Swedish Patent and Registration Office. Moving from Lab-based to Real Life Work-task Environment. In *Proceedings of the ACM-SIGIR 2000 Workshop on Patent Retrieval*, pages pp. 43–53, Athens, Greece, July 28.

Anette Hulth, Fredrik Olsson, and Mark Tierney. 2001. Exploring Key Phrases for Browsing an Online News Feed in a Mobile Context. In *Proceedings of Management of uncertainty and imprecision in multimedia information systems*, Toulouse, France, September. A workshop held in conjunction with the Sixth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-2001).

Robert C. Moore and Karen Sparck Jones. 1985. A research programme in natural language processing. CRC technical report, SRI International, Cambridge, England.

Fredrik Olsson, Preben Hansen, Kristofer Franzén, and Jussi Karlgren. 2001. Information Access and Refinement — A Research Theme. *ERCIM News*, 46, July.

Fredrik Olsson. 2002. *Requirements and Design Considerations for an Open and General Architecture for Information Refinement*. Licentiate of philosophy thesis, Department of Linguistics, Uppsala University, Uppsala, March. Available at `http://www.sics.se/~fredriko/lic`.

Joseph Polifroni and Stephanie Seneff. 2000. Galaxy-II as an Architecture for Spoken Dialogue Evaluation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, Athens, Greece, May. ELRA.

Stephanie Seneff, Ed Hurley, Raymond Lau, Christine Pao, Philipp Schmid, and Victor Zue. 1998. Galaxy–II: A Reference Architecture for Conversational System Development. In *Proceedings of the 5th International Conference on Spoken Language Processing*, volume 3, pages 931–934, Sydney, Australia, December.

Stephanie Seneff, Raymond Lau, and Joseph Polifroni. 1999. Organization, Communication, and Control in the Galaxy-II Conversational System. In *Proceedings of Eurospeech 99*, Budapest, Hungary, September.

Neil K. Simpkins. 1995. ALEP — An Open Architecture for Language Engineering. Technical report, Cray Systems, 151 rue des Muguets, L-2167 Luxembourg.

Arnold Smith. 1992. The CLE in Application Development. In Hiyan Alshawi, editor, *The Core Language Engine*, chapter 12, pages 235–250. MIT Press, Cambridge, Massachusetts, USA, March.

Pasi Tapanainen and Timo Järvinen. 1997. A Non-Projective Dependency Parser. In *Proceedings of the 5th Conference of Applied Natural Language Processing*, Washington, D.C. USA, April. ACL.

Kim Topley. 1998. *Core — Java Foundation Classes*. Prentice Hall PTR Core Series. Prentice Hall.