An Open Distributed Architecture for Reuse and Integration of Heterogeneous NLP Components

Rémi Zajac, Mark Casper and Nigel Sharples
Computing Research Laboratory
New-Mexico State University
{zajac,mcasper,nigel}@crl.nmsu.edu

Abstract

The shift from *Computational Linguistics* to *Language Engineering* is indicative of new trends in NLP. This paper reviews two NLP engineering problems: reuse and integration, while relating these concerns to the larger context of applied NLP. It presents a software architecture which is geared to support the development of a variety of large-scale NLP applications: Information Retrieval, Corpus Processing, Multilingual MT, and integration of Speech Components.

1 Introduction

The shift from *Computational Linguistics* to *Language Engineering*[1] is indicative of new trends in NLP. We believe that it is not simply a new fashion but that it is indicative of the growing maturation of the field, as also suggested by an emphasis on building large-scale systems, away from toy research systems. There is also an increasing awareness that real-size systems are not mere scaled-up toy systems but that they present an altogether qualitatively different set of problems that require new tools and new ideas, as clearly exemplified by recent projects and programs such as Pangloss (Frederking et al. 94), Tipster (ARPA 94), and Verbmobil (Görz et al. 96).

Natural language engineering addresses some traditional issues in software engineering: robustness, testing and evaluation, reuse, and development of large-scale applications (see e.g., (Sommerville 96) for an overview). These issues have been and are the topic of a number of NLP projects and programs: TSNLP, DECIDE, Tipster, MUC, TREC, Multext, Multilex, Genelex, Eagles, etc. This paper reviews two domains of problems in natural language engineering: reuse and integration in the context of software architectures for Natural Language Processing. The emphasis is put on reuse of NLP software, components and their integration in order to build large-scale applications. Also relevant to this presentation are topics such as integration of heterogeneous components for building hybrid systems or for integrating speech and other "higher-level" NLP components (section 2).

Section 3 presents the Corelli Document Processing Architecture, a new software architecture for NLP which is designed to support the development of a variety of large- scale NLP applications: Information Retrieval, Corpus Processing, Multilingual MT, and integration of Speech with other NLP components.

2 Reuse in NLP

There is an increasing amount of shared corpora and lexical resources that are being made available for NLP researchers through managed data repositories such as LDC, CLR, ELRA, etc. (see e.g., (Wilks et al. 92) for an overview of these repositories). These resources constitute the basic raw materials for building NLP software but not all of these resources can be readily used: they might be available in formats that require extensive pre-processing to transform them into resources that are tractable by NLP software. This pre-processing cannot usually be fully automated and is therefore costly.

Some projects have concentrated on developing lexical resources directly in a format suitable for further use in NLP software (e.g., Genelex, Multilex). These projects go beyond the definition of interchange formats to define a "neutral" linguistic representation in which all lexical knowledge is encoded and from which, by means of specialized compilers, application-specific dictionaries can be extracted. The lexical knowledge encoded in these systems can truly be called reusable since neither

[1] To use the name of two well-known NLP journals.

the format nor the content is application-dependent. The result of these projects is however not available to the research community.

Reuse of NLP software components remains much more limited (Cunningham et al. 96) since problems are compounded: the software components of an NLP system need not only to be able to exchange data using the same format (e.g., feature structures) and to share the same interpretation of the information they exchange (same linguistic theory, e.g., LFG), but they also need to communicate at the process level, either through direct API calls if they are written in the same programming language or through other means if, for example, they have to run on different platforms—a classical software integration problem. Thus, reuse of NLP software components can be defined as an integration problem. It is not of course the only approach to reuse in NLP (see for example (Biggerstaff & Perlis 89) for an overview of alternative approaches to software reuse) and some previous efforts have, for example, been directed at building Integrated Development Environments ((Boitet et al. 82; Simkins 94; Alshawi 92; Grover et al. 93) to mention but a few). Although Integrated Development Environments address some of the problems, they do not give a complete solution since one still has to develop rules and lexical entries using these systems.

Direct reuse of NLP software components, e.g., using an existing morphological analyzer as a component of a larger system, is still very limited but is nevertheless increasingly attractive since the development of large-scale NLP applications, a focus of current NLP research, is prohibitive for many research groups. The Tipster architecture for example is directed towards the development of information retrieval and extraction systems (ARPA 94; Grishman 95) and provides a modular approach to component integration. The GATES architecture builds upon the Tipster architecture and provides a graphical development environment to test integrated applications (Cunningham et al. 96). Speech machine-translation architectures need also to solve difficult integration problems and original solutions have been developed in the Verbmobil project (Görz et al. 96), and by researchers at ATR (e.g., (Boitet & Seligman 94)) for example. A generic NLP architecture needs to address component communication and integration at three distinct levels:

1. The *process or communication layer* involves, for example, communication between different components that could be written in different programming languages and could be running as different processes on a distributed network.
2. The *data layer* involves exchange and translation of data structures between components.
3. At the linguistic level, components need to share the same interpretation of the data they exchange.

A particular NLP architecture embodies design choices related to how components can talk to each other. A variety of solutions are possible as illustrated below.

- Each component can talk directly to each other and thus all components need to incorporate some knowledge about each other at all three levels mentioned above. This is the solution adopted in the Verbmobil architecture which makes use of a special communication software package (written in C and imposing the use of C and Unix) at the process level and uses a chart annotated with feature structures at the data-structure level. At the linguistic level, a variant of HPSG is used (Kesseler 94; Amtrup 95; Turk & Geibler 95; Görz et al. 96).
- A central coordinator can incorporate knowledge about each component but the component themselves don't have any knowledge about each other, or even about the coordinator. Filters are needed to transform data back and forth between the central data-structure managed by the coordinator (a lattice would be appropriate) and the data processed by each component. Communication between the coordinator and the components can be asynchronous and the coordinator needs then to serialize the actions of each component. This solution, a variant of the blackboard architecture (Erman & Lesser 80) is used in the Kasuga speech translation prototype described in (Boitet & Seligman 94). This architecture imposes no constraints on the components (programming language or software architecture) since communication is based on the SMTP protocol.
- The Tipster Document Architecture makes no assumption about the solution used either at the process level or at the linguistic level. At the data structure level, NLP components exchange data by reading and writing "annotations" associated with some segment of a document (Grishman 95). This solution also forms the basis of the GATES system (Cunningham et al. 96). Various versions of this architecture

have been developed (in C, C++ and Lisp) but no support is defined for integration of heterogeneous components. However, in the Tipster Phase III program, a CORBA version of the Tipster architecture will be developed to support distributed processing.

3 The Corelli Document Processing Architecture

The Corelli Document Processing Architecture is an attempt to address the various problems mentioned above and also some other software-level engineering issues such as robustness, portability, scalability and inter-language communication (for integrating components written in Lisp, C or other languages). Also of interest are some ergonomic issues such as tractability, understandability and ease of use of the architecture (the programmer being the user in this case). The architecture provides support for component communication and for data exchange. No constraint is placed on the type of linguistic processing but a small library of data-structures for NLP is provided to ease data-conversion problems. The data layer implements the Tipster Document Architecture and enables the integration of Tipster-compliant components. This architecture is geared to support the development of large-scale NLP applications such as Information Retrieval systems, multilingual MT systems (Vanni & Zajac 96), hybrid or multi-engine MT systems (Wilks et al. 92; Frederking et al. 94; Sumita & Iida 95), speech-based systems (Boitet & Seligman 94; Görz et al. 96) and also systems for the exploration and exploitation of large corpora (Ballim 95; Thompson 95).

Basic software engineering requirements

- A *modular* and *scalable* architecture enables the development of small and simple applications using a file-based implementation such as a grammar checker, as well as large and resource-intensive applications (information retrieval, machine translation) using a database back-end (with two levels of functionality allowing for a single-user persistent store and a full-size commercial database).

- A *portable* implementation allows the development of small stand-alone PC applications as well as large distributed Unix applications. Portability is ensured through the use of the Java programming language.

- A *simple* and *small* API which can be easily learned and does not make any presupposition about the type of application. The API is defined using the IDL language and structured according to CORBA standards and the CORBA services architecture (OMG 95).

- A *dynamic Plug'n Play architecture* enabling easier integration of components written in different programming languages (C, C++, Lisp, Java, etc), where components are "wrapped" as tools supporting a common interface.

3.1 Data Layer: Document Services

The data layer of the Corelli Architecture is derived from the Tipster Architecture and implements the requirements listed above. In this architecture, components do not talk directly to each other but communicate through information (so-called 'annotations') attached to a document. This model reduces inter-dependencies between components, promoting the design of modular applications (Figure 1) and enabling the development of blackboard-type applications such as the one described in (Boitet & Seligman 94). The architecture provides solutions for

- Representing information about a document,

- Storing and retrieving this information in an efficient way,

- Exchanging this information among all components of an application.

It does not however provide a solution for translating linguistic structures (e.g., mapping a dependency tree to a constituent structure). These problems are application-dependent and need to be resolved on a case-by-case basis; such integration is feasible, as demonstrated by the various Tipster demonstration systems, and use of the architecture reduces significantly the load of integrating a component into the application.

Documents, Annotations and Attributes

The data layer of the Corelli Document Processing Architecture follows the Tipster Architecture. The basic data object is the document. Documents can have attributes and annotations, and can be grouped into collections. Annotations are used to store information about a particular segment of the document (identified by a span, i.e., start-end byte offsets in the document content) while the document itself remains unchanged. This contrasts with the SGML solution used in the Multext project where information about a piece of text is stored as additional SGML mark-up in the document itself (Ballim 95;

Thompson 95). This architecture supports read-only data (e.g., data stored in a CD-ROM) as well as writable data. Annotations are attributed objects that contain application objects. They can be used, for example, to store morphological tags produced by some tagger, to represent the HTML structure of an HTML document or to store partial results of a chart-parser.

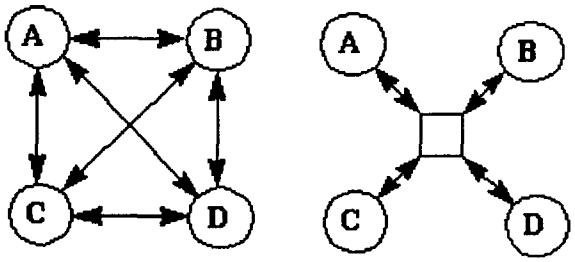

Figure 1: Document annotations as a centralized data-structure enable modular architectures and reduce the number of interfaces from the order of n^2 to the order of n.

Document Annotations

Corelli document annotations are essentially the same as Tipster document annotations and a similar generic interface is provided. However, considering the requirements of NLP applications such as parsers or documents browsers, two additional interfaces are provided:

- Since a set of annotations can be quite naturally interpreted as a chart, a chart interface provides efficient access to annotations viewed as a directed graph following the classical model of the chart first presented in (Kay 73).
- An interval-tree interface provides efficient access for efficient implementation of display functionalities.

Application Objects

An application manipulating only basic data types (strings, numbers,...) need not define application objects. However, some applications may want to store complex data structures as document annotations, for example, trees, graphs, feature structures, etc. The architecture provides a top application-object class that can be sub-classed to define specific application objects. To support persistency in the file-based version, an application object needs to implement the `read-persistent` and `write-persistent` interfaces (this is provided transparently by the persistent versions). A small library of application objects is provided with the architecture.

Accessing Documents

Documents are accessible via a Document Server which maintains persistent collections, documents and their attributes and annotations. An application can define its own classes for documents and collections. In the basic document class provided in the architecture, a document is identified by its name (URL to the location of the document's content). In this distributed data model, accessing a document via a Document Server gives access to a document's contents and to attributes and annotations of a document.

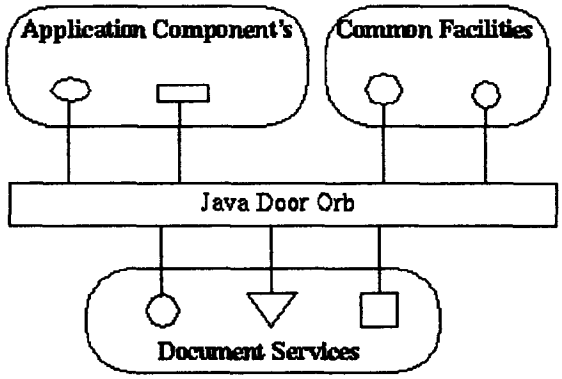

Figure 2: NLP components access Document Services and other facilities (e.g., codeset converters) through JavaSoft's Java Door Orb.

Services

The Corelli Architecture incorporates standards such as CORBA for defining inter-operable interfaces, and HTTP for data transport. Following the CORBA model, the Architecture is structured as a set of services with well- defined interfaces:

- A Document Management Service (DMS) provides functions for manipulating collections, documents, annotations and attributes.
- A Life-Cycle Service provides creation, copying, moving and deletion of objects.
- A Naming Service provides access to documents and collections via their names. Named collections and documents are persistent.

Figure 2 gives an overview of the Corelli Document Architecture: an NLP component accesses a Document Service provided by a Document Server using the Corelli Document Architecture API. Client-side application component API calls on remote object references (requested from the Orb)

are transparently 'transferred' by the Orb to a Document Services implementation object for invocation.

Figure 3 describes the Java IDL compiler and Java Door Orb interaction. The Corelli Document Architecture API is specified using the Interface Definition Language (IDL), a standard defined by the Object Management Group (OMG 95). The IDL-to-Java compiler essentially produces three significant files: one containing a Java interface corresponding to the IDL operational interface itself, a second containing client-side 'stub' methods to invoke on remote object references (along with code to handle Orb communication overhead), and a third containing server-side 'skeleton' methods to handle implementation object references. What remains is for the server code, implementing the IDL operational interface to be developed.

When the server implementing the IDL specification is launched, it creates skeleton object references for implemented services/objects and publishes them on the Orb. A client wishing to invoke methods on those remote objects creates stub object references and accesses the orb to resolve them with the implementation references on the server side. Any client API call made on a resolved object reference is then transparently (to the client) invoked on the corresponding server-side object.

The Document Management Service, the Life-Cycle Service and the Naming Service are included in the three versions of the architecture which implement increasingly sophisticated support of database functionalities:

1. The basic file-based version of the architecture uses the local file system to store persistent data (collections, attributes and annotations); the contents of a document can however be located anywhere on the Internet.

2. A persistent store version uses a persistent-store back-end for storing and retrieving collections, attributes and annotations: this version supports the Persistent Object Service which provides greater efficiency for storing and accessing persistent objects as well as enhanced support for defining persistent application objects.

3. A database version uses a commercial database management system to store and retrieve collections, attributes and annotations and also documents (through an import/export mechanism). This version provides a Concurrency Control Service and a Transaction Service.

Communication Layer

To support integration and communication at the process level, the current version of the Corelli Architecture provides component inter-communication via the Corelli Plug'n Play architecture (see below) and the Java Door Orb.

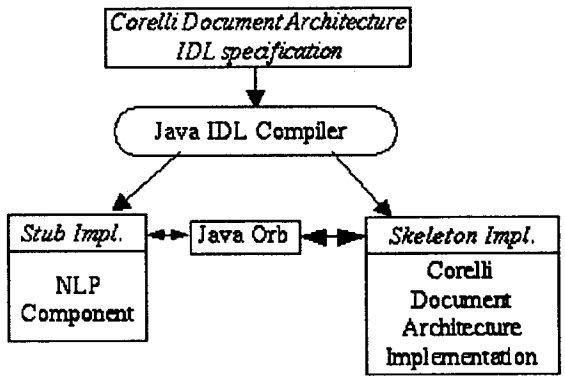

Figure 3: Java IDL Compiler Java Door Orb Interaction.

3.2 Plug'n Play Architecture

The data layer of the Corelli Document Architecture, as described above, provides a static model for component integration through a common data framework. This data model does not provide any support for communication between components, i.e., for executing and controlling the interaction of a set of components, nor for rapid tool integration. The Corelli Plug'n Play layer aims at filling this gap by providing a dynamic model for component integration: this framework provides a high-level of plug-and-play, allowing for component interchangeability without modification of the application code, thus facilitating the evolution and upgrade of individual components.

In the preliminary version of the Corelli Plug'n Play layer, the choice was made to develop the most general version of the architecture to ensure that any tool can be integrated using this framework. In this model, all components run as servers and the application code which implements the logic of the application runs as a client of the component servers. To be integrated, a component needs to support synchronous or asynchronous versions of one or several of four basic operations: `execute`, `query`, `convert` and `exchange` (in addition to standard initialization ad termination operations). Client-server communication is supported by the Java Door Orb.

The rationale for this architecture is that many NLP tools are themselves rather large software com-

ponents, and embedding them in servers helps to reduce the computation load. For example, some morphological analyzers load their dictionary in the process memory, and on small documents, simply starting the process could take more time than actual execution. In such cases, it is more efficient to run the morphological analyzer as a server that can be accessed by various client processes. This architecture also allows the processing load of an application to be distributed by running the components on several machines accessible over the Internet, thereby enabling the integration of components running on widely different architectures. This model also provides adequate support for the integration of static knowledge sources (such as dictionaries) and of ancillary tools (such as codeset converters).

Figure 4 gives a picture of one possible integration solution. In this example, each component of the application is embedded in a server which is accessed through the Corelli Component Integration API as described above. A component server translates an incoming request into a component action. The server also acts as a filter by translating the document data structures stored in the Document Server in a format appropriate as input for the component and conversely for the component output. Each component server acts as a wrapper and several solutions are possible:

- If the component has a Java API, it can be encapsulated directly in the server.
- If the component has an API written in one of the languages supported by the Java Native Interface (currently C and C++), it can be dynamically loaded into the server at runtime and accessed via a Java front end.
- If the component is an executable, the server must issue a system call for running the program and data communication usually occurs through files.

The Document Server itself is accessed via its API and is running as a Java Door Orb supporting requests from the component's servers.

This framework does not provide a model for controlling the interaction between the components of an application : the designer of an NLP application can use a simple sequential model or more sophisticated blackboard models : since this distributed model supports both the synchronous and the asynchronous types of communication between components, it supports a large variety of control models.

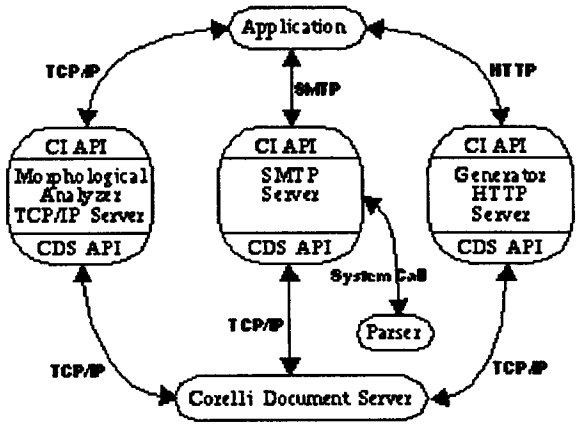

Figure 4: Some possible integration paths for heterogeneous components.

4 Implementation

4.1 Document Server Implementation

The Document Server consists of three major modules: Document Management Service, Naming Service, and Life-Cycle Service. The modules are defined in IDL, and implemented in Java. The Sun Java IDL system, with its Door Orb implementation, is used to interface client programs to the Document Server implementation.

The Document Management Service module provides methods to access and manipulate the components of objects (e.g., attributes, annotations and content of a document object).

The Life-Cycle Service is responsible for creating and copying objects.

The Naming Service binds a name to an object. The Naming Service supports a limited form of persistency for storing bindings.

For example, to create a new document, the client program creates it through the Life-Cycle Service, bind a name to it using the Naming Service, and add attributes and annotations to it through the Document Management Service.

4.2 Porting of the Temple Machine Translation System

To bootstrap the Corelli Machine Translation System and test the implementation of the architecture, we are currently porting the CRL's Temple machine-translation system prototype (Vanni & Zajac 96) to the Corelli architecture. This task will be aided by two features: first, the Temple system already utilizes the Tipster Document Architecture for data exchange between components, and second, the Temple system has a pipelined architecture which will

allow modular encapsulation of translation stages (e.g., dictionary lookup) as Corelli Plug'n Play tools.

The Temple morphological analyzers and the English morphological generator all function as standalone executables and will be easily converted to Corelli Plug'n Play tools. Lexical resources (e.g., dictionaries and glossaries), on the other hand, are currently maintained in a database and are accessed via calls to a C library API. Each lexical resource is wrapped as a Plug'n Play tool implementing the query interface: in order to interface with the databases, the Java Native Interface is used to wrap the C database library. Finally, we will have to reengineer a portion of the top-level application control code (in C) in Java.

5 Conclusion

The Corelli Document Architecture is currently used as the integration layer for the Corelli Machine-Translation System. This multilingual machine-translation system is built out of heterogeneous components, such as an English generator written in Lisp, a Spanish morphological analyzer written in Prolog, a Glossary-Based Machine-Translation engine written in C, etc. This architecture will also be used to support integration of various machine translation systems in a multi-engine machine translation project (building on ideas first developed in the Pangloss project, see (Frederking et al. 94)).

The Corelli project has started collaborating with the University of Sheffield with the aim to merge the Corelli Document Architecture and the GATE architecture.[2] More specifically, the current GATE document manager will be replaced with the Corelli document manager and the Plug'n Play layer will be added to support distributed processing.

The file-based version of the Corelli Document Processing Architecture will be made freely available for research purposes. It will also be available as part of the GATE system distribution.

Acknowledgments. An initial version of this architecture has been developed by Vani Mahesh.

Research reported in this paper is supported by the DoD, contract MDA904-96-C-1040.

References

Hiyan Alshawi. 1992. *The Core Language Engine.* MIT Press.

ARPA - Advanced Research Projects Agency. 1993. Proceedings of the *TIPSTER Text Program – Phase 1.* Morgan-Kaufmann.

Jan W. Amtrup. 1995. "Chart-based Incremental Transfer in Machine Translation". Proceedings of the *6th International Conference on Theoretical and Methodological Issues in Machine Translation – TIM'95*, 5-7 July 1995, Leuven, Belgium. pp188-195.

A. Ballim. 1995. "Abstract Data Types for Multext Tool I/O". LRE 62-05 Deliverable 1.2.1.

Ted J. Biggerstaff, Alan J. Perlis, eds. 1989. *Software Reusability,* 2 volumes. ACM Press, Addison-Wesley.

Christian Boitet, Pierre Guillaume, Maurice Quézel-Ambrunaz. 1982. "Implementation of the conversational environment of ARIANE 78.4, an integrated system for automated translation and human revision". Proceedings of the *9th International Conference on Computational Linguistics – COLING'82.*

Christian Boitet and Mark Seligman. 1994. "The Whiteboard Architecture: a Way to Integrate Heterogeneous Components of NLP Systems". Proceedings of the *15th International Conference on Computational Linguistics – COLING'94*, August 5-9 1994, Kyoto, Japan. pp426-430.

H. Cunningham, M. Freeman, W.J. Black. 1994. "Software Reuse, Object-Oriented Frameworks and Natural Language Processing". Proceedings of the *1st Conference on New Methods in Natural Language Processing – NEMLAP-1*, Manchester.

H. Cunningham, Y. Wilks, R. Gaizauskas. 1996. "New Methods, Current Trends and Software Infrastructure for NLP". Proceedings of the *2nd Conference on New Methods in Natural Language Processing – NEMLAP-2*, Ankara, Turkey.

L.D. Erman, V.R. Lesser. 1980. "The Hearsay-II speech understanding system". In W.A. Lea (ed.), Trends in Speech Recognition, Prentice-Hall. pp361-381.

Robert Frederking, Sergei Nirenburg, David Farwell, Stephen Helmreich, Eduard Hovy, Kevin Knight, Stephen Beale, Constantine Domashnev, Donalee Attardo, Dean Grannes, Ralf Brown. 1994. "Integrating Translations from Multiple Sources within the Pangloss Mark III Machine Translation System". Proceedings of the *1st Conference of the Association for Machine Translation in the Americas – AMTA'94*, 5-8 October 1994, Columbia, Maryland. pp73- 80.

[2]The GATE system already uses a previous version (written in C) of a Tipster document manager developed at CRL.

Günther Görz, Marcus Kesseler, Jörg Spilker, Hans Weber. 1996. "Research on Architectures for Integrated Speech/ Language Systems in Verbmobil". Verbmobil Report 126, Universität Erlangen-Nürnberg, May 1996.

Claire Grover, John Caroll and Ted Briscoe. 1992. *The Alvey Natural Language Tools.* Computer Laboratory, University of Cambridge, UK.

Ralph Grishman, editor. 1995. "Tipster Phase II Architecture Design Document". New-York University, NY, July 1995.

Bill Janssen, Mike Spreitzer. 1996. "ILU 2.0 Reference Manual". Xerox PARC.

Martin Kay. 1973. "The MIND system". In R. Rustin (ed.), *Courant Computer Science Symposium 8: Natural Language Processing.* Algorithmics Press, New-York, NY. pp155-188.

Martin Kay. 1996. "Chart Generation". Proceedings of the *34th Meeting of the Association for Computational Linguistics* ACL'96. pp200-204.

M. Kesseler. 1994. "Distributed Control in Verbmobil". Verbmobil Report 24, Universität Erlangen-Nürnberg, August 1994.

Sergei Nirenburg. 1994. "The Workstation Substrate of the Pangloss Project". Proceedings of the *Conference on Future Generation of Natural Language Processing Systems - FGNLP-2.*

Sergei Nirenburg and Robert Frederking. 1994. "Towards Multi-Engine Machine Translation". Proceedings of the *ARPA Human Language Technology Workshop*, March 8-11 1994, Plainsboro, NJ. pp147-151.

Sergei Nirenburg, David Farwell, Robert Frederking, Yorick Wilks. 1994. "Two types of adaptative MT environments". Proceedings of the *15th International Conference on Computational Linguistics - COLING'94*, August 5-9 1994, Kyoto, Japan. pp125-128.

OMG. 1995. "The Common Object Request Broker: Architecture and Specification, Version 2.0". OMG Technical Document PTC/96-03-0.

N.K. Simkins. 1994. "An Open Architecture for Language Engineering". Proceedings of the *1st Language Engineering Convention*, Paris.

Ian Sommerville. 1996. *Software Engineering* (5th Edition). Addison-Wesley.

Eiichiro Sumita and Hitoshi Iida. 1995. "Heterogeneous Computing for Example-based Translation of Spoken Language". Proceedings of the *6th International Conference on Theoretical and Methodological Issues in Machine Translation - TIM'95*, 5-7 July 1995, Leuven, Belgium. pp273-286.

Henry Thompson and Graeme Ritchie. 1984. "Implementing Natural Language Parsers". In T. O'Shea and E. Eisenstadt (eds.), *Artificial Intelligence.* Harper & Row, New-York. pp245-300.

Henry Thompson. 1995. "Multext Workpackage 2, Milestone B, Deliverable Overview". LRE 62-050 Deliverable 2.

Andrea Turk and Stefan Geibler. 1995. "Integration alternativer Komponenten für die Sparchverarbeitung im Verbmobil Demonstrator". Verbmobil Report 67, IBM Informationssysteme GmBH, April 1995.

Michelle Vanni and Rémi Zajac. 1996. "Glossary-Based MT Engines in a Multilingual Analyst's Workstation for Information Processing". To appear in *Machine Translation*, Special Issue on New Tools for Human Translators.

Yorick Wilks, Louise Guthrie, Joe Guthrie and Jim Cowie. 1992. "Combining Weak Methods in Large-Scale Text Processing". In Paul S. Jacob (ed.), *Text-Based Intelligent Systems*, Lawrence Erlbaum Associates, pp35-58.

Rémi Zajac. 1992. "Towards Computer-Aided Linguistic Engineering". Proc. of the *14th International Conference on Computational Linguistics - COLING'92*, 23-28 August 1992, Nantes, France. pp827-834.

Rémi Zajac. 1996. "A Multilingual Translator's Workstation for Information Access", Proceedings of the *International Conference on Natural Language Processing and Industrial Applications - NLP+IA 96*, Moncton, New-Brunswick, Canada, June 4-6, 1996.

Rémi Zajac. 1996. "Towards a Multilingual Analyst's Workstation: Temple". In *Expanding MT Horizons - Proceedings of the 2nd Conference of the Association for Machine Translation in the Americas, AMTA-96.* 2-5 October 1996, Montréal, Canada. pp280-284.

Rémi Zajac and Mark Casper. "The Temple Web Translator". Proc. of the *1997 AAAI Spring Symposium on Natural Language Processing for the World Wide Web*, March 24-26, 1997, Stanford University.