Using Toolsets and Architectures To Build NLP Systems

Proceedings of the COLING-2000 Workshop on
Using Toolsets and Architectures To Build NLP Systems

Centre Universitaire, Luxembourg
5 August 2000

Using Toolsets and Architectures To Build NLP Systems

Centre Universitaire, Luxembourg
5 August 2000

Table of Content

Author Index

Program Committee

Using Toolsets and Architectures To Build NLP Systems

Centre Universitaire, Luxembourg
5 August 2000

Many toolsets have been developed to support the implementation of single NLP components (taggers, parsers, generators, dictionaries) or complete Natural Language Processing applications (Information Extraction systems, Machine Translation systems). A source for available toolkits is the Natural Language Software Registry, an initiative of the Association for Computational Linguistics hosted by DFKI at `http://registry.dfki.de`. These tools aim at facilitating and lowering the cost of building NLP systems. Since the tools themselves are often complex pieces of software, they require a significant amount of effort to be developed and maintained in the first place. Is this effort worth the trouble? It is to be noted that NLP toolsets have often been originally developed for implementing a single component or application. In this case, why not build the NLP system using a general programming language such as Lisp or Prolog? There can be at least two answers. First, for pure efficiency issues (speed and space), it is often preferable to build a parameterized algorithm operating on a uniform data structure (e.g., a phrase-structure parser). Second, it is harder, and often impossible, to develop, debug and maintain a large NLP system directly written in a general programming language.

It has been the experience of many users that a given toolset is quite often unusable outside its environment: the toolset can be too restricted in its purpose (e.g. an MT toolset that cannot be used for building a grammar checker), too complex to use, or even too difficult to install. There have been, in particular in the US under the Tipster program, efforts to promote instead common architectures for a given set of applications (primarily IR and IE in Tipster; see also the Galaxy architecture of the DARPA Communicator project). Several software environments have been built around this flexible concept, which is closer to current trends in main stream software engineering.

The workshop aims at providing a picture of the current problems faced by developers and users of toolsets, and future directions for the development and use of NLP toolsets. It includes reports of actual experiences in the use of toolsets as well as presentation of toolsets and application development.

Rémi Zajac, Computing Research Laboratory, New Mexico State University
`zajac@crl.nmsu.edu`

Experience of using GATE for NLP R&D.

Hamish Cunningham, Diana Maynard,
Kalina Bontcheva, **Valentin Tablan** and **Yorick Wilks**
Department of Computer Science and
Institute for LAnguage, Speech and Hearing,
University of Sheffield, UK
{hamish,diana,kalina,valyt,yorick}@dcs.shef.ac.uk

Abstract

GATE, a General Architecture for Text Engineering, aims to provide a software infrastructure for researchers and developers working in NLP. GATE has now been widely available for four years. In this paper we review the objectives which motivated the creation of GATE and the functionality and design of the current system. We discuss the strengths and weaknesses of the current system, identify areas for improvement.

1 Introduction

This paper relates experiences in projects that have used GATE (General Architecture for Text Engineering) over the four years since its initial release in 1996.

We begin in section 2 with some of the motivation behind this type of system, and go on to give a definition of *architecture* in this context (section 3). Section 4 briefly describes GATE; section 5 covers a range of projects that have used the system. These experiences form the input to section 6 which discusses the system's strengths and weaknesses.

2 Motivation

If you're researching human language processing you should probably not be writing code to:

- store data on disk;
- display data;
- load processor modules and data stores into processes;
- initiate and administer processes;
- divide computation between client and server;
- pass data between processes and machines.

A *Software Architecture* for language processing should do all this for you. You will have to parameterise it, and sometimes deployment of your work into applications software will require some low-level fiddling for optimisation purposes, but in the main these activities should be carried out by infrastructure for the language sciences, not by each researcher in the field.

We can go further and say that you shouldn't have to reinvent components and resources outside of your specialism if there is already something that could do the job. A statistician doesn't need to know the details of the IEEE Floating Point computation standard; a discourse processing specialist doesn't need to understand all the ins and outs of part-of-speech tagging (or worse still how to install a particular POS tagger on a particular machine).

If you're a professional mathematician, you probably regard a tool like SPSS or Mathematica as necessary infrastructure for your work. If you're a computational linguist or a language engineer, the chances are that large parts of your work have no such infrastructural support. Where there is infrastructure, it tends to be specific to restricted areas. GATE, a General Architecture for Text Engineering (Cunningham et al., 1997), represents an attempt to fill this gap, and is a software architecture for language processing R&D.

We now have four years of experience with GATE, work on which began in 1995, with a first widespread release late in 1996. The system is currently at a pivotal point in its development, with a new version in development.

3 Infrastructure for Language Processing R&D

What does infrastructure mean for Natural Language Processing (NLP)? What sorts of tasks

should be delegated to a general tool, and which should be left to individual projects? The position we took in designing GATE is to focus on the *common elements of NLP systems.*

There are many useful tools around for performing specific tasks such as developing feature structure grammars for evaluation under unification, or collecting statistical measures across corpora. To varying extents, they entail the adoption of particular theories. The only common factor of NLP systems, alas, seems to be that they very often create information about text. Developers of such systems create modules and data resources that handle text, and they store this data, exchange it between various modules, compare results of test runs, and generally spend inordinate amounts of time pouring over samples of it when they really should be enjoying a slurp of something relaxing instead.

The types of data structure typically involved are large and complex, and without good tools to manage and allow succinct viewing of the data we work below our potential. At this stage in the progress of our field, no one should really have to write a tree viewing program for the output of a syntax analyser, for example, or even have to do significant work to get an existing viewing tool to process their data.

In addition, many common language processing tasks have been solved to an acceptable degree by previous work and should be reused. Instead of writing a new part of speech tagger, or sentence splitter, or list of common nominal compounds, we should have available a store of reusable tools and data that can be plugged into our new systems with minimal effort. Such reuse is much less common than it should be, often because of installation and integration problems that have to be solved afresh in each case (Cunningham et al., 1994).

In sum, we defined our infrastructure as an architecture, framework and development environment, where an architecture is a macro-level organisational pattern for the components and data resources that make up a language processing system; a framework is a class library implementing the architecture; a development environment adds graphical tools to access the services provided by the architecture.

4 GATE

GATE version 1.n does three things:

- manages textual data storage and exchange;
- supports visual assembly and execution of modular NLP systems plus visualisation of data structures associated with text;
- provides plug-in modularity of text processing components.

The architecture does this using three subsystems:

- GDM, the GATE Document Manager;
- GGI, the GATE Graphical Interface;
- CREOLE, a Collection of REusable Objects for Language Engineering.

GDM manages the information about texts produced and consumed by NLP processes; GGI provides visual access to this data and manages control flow; CREOLE is the set of resources so far integrated. Developers working with GATE begin with a subset of CREOLE that does some basic tasks, perhaps tokenisation, sentence and paragraph identification and part-of-speech tagging. They then add or modify modules for their specific tasks. They use a single API for accessing the data and for storing their data back into the central database. With a few lines of configuration information they allow the system to display their data in friendly graphical form, including tree diagrams where appropriate. The system takes care of data storage and module loading, and can be used to deliver embeddable subsystems by stripping the graphical interface. It supports modules in any language including Prolog, Lisp, Perl, Java, C++ and Tcl.

5 Projects that used GATE

5.1 ECRAN

Goal: ECRAN (Extraction of Content: Research at Near-market) (Basili et al., 1997) was a 3-year EU funded research

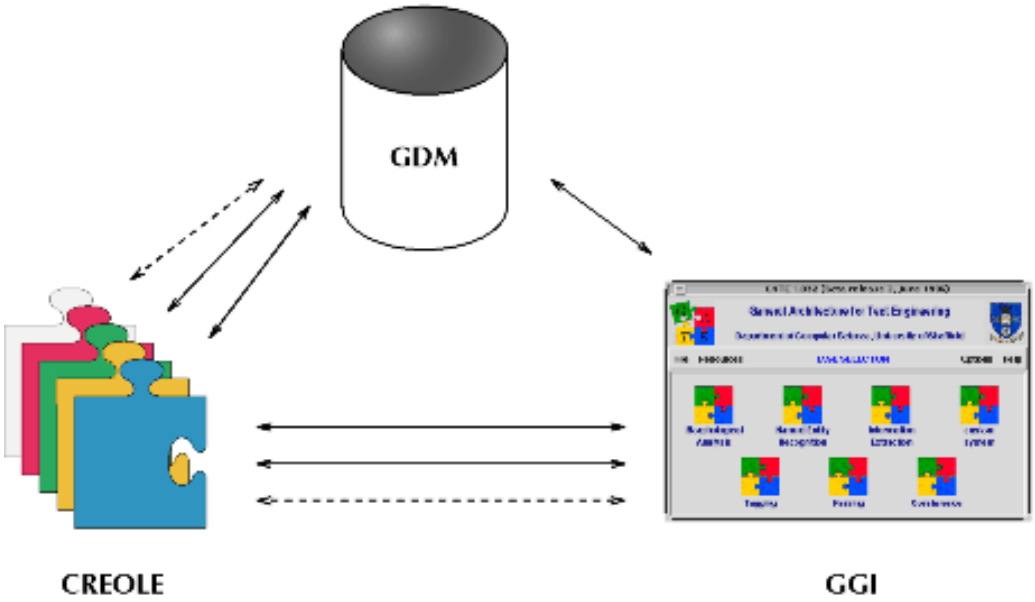

Figure 1: Gate Architecture

project with the main aim of carrying out information extraction using adapted lexicons.
Participants: Thomson-CSF (Paris) (project co-ordinators), SIS (Smart Information Systems, Germany), University of Sheffield, University of Rome La Sapienza, University of Geneva, NCSR "Demokritos" (Athens)
Description: GATE was mainly used in this project to implement a general word sense disambiguation engine based on a combination of classifiers.
Benefits: The modular architecture of GATE allowed this to be carried out very rapidly.
Drawbacks: Two main disadvantages were found with GATE. (1) The architecture was under development at the same time as the word sense disambiguation engine. (2) The speed of database access for the Tipster database was found to be slow for large amounts of lexical data. The solution used was to store large amounts of lexical data separately from GATE as gdbm hash tables.

5.2 Cass-SWE

Goal: The aim of the Cass-SWE project (A Cascaded Finite-State Parser for Syntactic Analysis of Swedish) (Kokkinakis and Johansson-Kokkinakis, 1999) was to create a parsing system for fast and accurate analysis of large volumes of written Swedish.
Participants: Språkdata/Göteborg University, Sweden.
Description: Cass-SWE implements the grammar as a modular set of 6 small grammars. GATE is used to integrate all the required software components into one system prior to parsing, and to enable the results to be visualised in a user-friendly environment.
Benefits: GATE allows the tagging process to be carried out sequentially, and enables modification of individual elements without disruption to others. Using GATE as a visualisation environment also enables the results of Cass-SWE to be further used in applications such as information extraction tasks and additional semantic processing.
Drawbacks: There were a few initial difficulties understanding the workings of the GATE system, but problems originally thought to be caused by GATE were later traced to the CASS parser.

5.3 GIE

Goal: The aim of the GIE (Greek Information Extraction) project (Petasis et al., 1999) was to develop a prototype named entity recognition model for Greek.
Participants: NCSR "Demokritos" (Athens), University of Sheffield
Description: The GIE system is based on the VIE system provided with GATE, but requires different language-specific resources such as gazetteers and grammars. Using GATE enables non-language specific resources to be reused from the English version, thereby saving time and effort.
Benefits: GATE facilitated significantly the integration of existing and new modules in GIE, as well as the validation of the final demonstrator. It was generally found to be fast, easy to use and powerful.
Drawbacks: GATE's demand for system resources as document size increases can become a serious limitation. Complex compilation processes made the embedding of static modules difficult. GATE also has some difficulties supporting non-Latin languages. mostly relating to the GUI. Many minor possible improvements to the GUI and to GATE in general (such as the addition of new features) were identified during this project.

5.4 LaSIE

Goal: LaSIE (Wilks and Gaizauskas, 1999)is an advanced large-scale IE system, performing named entity recognition, coreference resolution, template element filling and scenario template filling.
Participants: University of Sheffield
Description: LaSIE was designed specifically to work within the GATE architecture, and led to the free distribution of its counterpart, VIE, a base-line IE system. LaSIE modules within GATE have also formed part of other customised projects within the EC Fourth Framework (AVENTINUS and ECRAN).

5.5 EMPathIE

Goal: EMPathIE (Enzyme and Metabolic Path Information Extraction) was an 18-month research project aimed at applying Information Extraction technology to bioinformatics tasks.
Participants: Dept. of Information Studies & Dept. of Computer Science (University of Sheffield), Glaxo Wellcome plc., Elsevier Science.
Description: EMPathIE aims to extract details of enzyme reactions from articles in biomedical journals. The IE system is derived from LaSIE and was developed within the GATE architecture.
Benefits: The embedding of EMPathIE within the GATE environment means that many modules can be reused. EMPathIE thus makes use of many of the LaSIE modules, and itself produces modules which have been used for other related projects. Using GATE therefore enables much of the low-level work in moving IE systems to new domains to be carried out effortlessly.

5.6 SVENSK

Goal: SVENSK (Olsson, 1997; Olsson et al., 1998; Gambäck and Olsson, 2000) was a 4-year project aimed at developing an integrated toolbox of language processing components and resources for Swedish.
Participants: SICS (Swedish Institute of Computer Science), NUTEK, Uppsala University, Göteborg University, PipeBeach AB., Telia Research AB, IBM Svenska AB
Description: The toolbox is based on the GATE language engineering platform and incorporates language processing tools developed at SICS or contributed by external sources.
Benefits: Each component has a standardised interface, so users have the choice of working within GATE or selecting and combining supplied components for integration into a user application. GATE is useful in that it is not committed to any particular type of data or task. The emphasis on modularity was also found to be particularly appealing.

Drawbacks: GATE was at the time still in its early phases and had some problems with very large-scale resources. Specification of byte offset and I/O requirements for different modules was also difficult.

5.7 LOTTIE

Goal: LOTTIE (Low Overhead Triage from Text using Information Extraction) was a demonstrator project for the GATE infrastructure. It aimed to provide proof-of-concept by implementing demonstration software dealing with the major technological problems involved in computer-assisted triage.
Participants: University of Sheffield
Description: LOTTIE did not itself use GATE, but formed a basis on which to build it. Parts of it were real, based on a project in a different domain, and parts of it served as a test case for GATE development and as a demonstration of future possibilities.

5.8 AVENTINUS

Goal: AVENTINUS (Advanced Information System for Multinational Drug Enforcement) is an EU funded research and development programme set up to build an information system for multinational drug enforcement.
Participants: SIETEC (Germany), ADB (France), Amt für Auslandsfragen (Germany), Bundeskriminalamt (Germany), Sprakdata Gothenburg (Sweden), Institute for Language and Speech Processing (Greece), INCYTA (Spain), University of Sheffield.
Description: AVENTINUS aims to collect information from distributed international sources, using advanced linguistic techniques to improve IE, involving multimedia resources and supporting multilinguality.

5.9 TRESTLE

Goal: TRESTLE (Text Reuse, Extraction and Summarisation for Large Enterprises) (TRESTLE, 2000) is a 2-year project involving IE from electronic alerting bulletins distributed daily throughout the pharmaceutical industry.
Participants: Glaxo-Wellcome plc, University of Sheffield Dept. of Computer Science and Dept. of Information Studies.
Description: TRESTLE is based on the LaSIE IE system, but requires different domain-specific resources, such as gazetteers and ontology, and substantial modification of the discourse interpreter and template writer.
Benefits: GATE provides domain independent linguistic components for TRESTLE, the most important of which is the semantic parser. Named Entity recognition requires only the installation of domain specific gazetteers.
Drawbacks: It is very difficult to make even minor modifications to existing components of GATE. Current documentation is inadequate, and very strong computing skills are necessary in order to make the most of it.

5.10 PASTA

Goal: PASTA (Protein Active Site Template Acquisition) (K. Humphreys and Gaizauskas, 2000) extracts information about protein structures directly from scientific journal papers, and stores them in a template.
Participants: Depts. of Computer Science, Molecular Biology & Biotechnology, and Information Studies (University of Sheffield).
Description:The system has been adapted to the molecular biology domain from pre-existing IE technology such as LaSIE. The progress so far demonstrates the feasibility of developing intelligent systems for IE from text-based sources in the pursuit of knowledge in the biological domain.
Benefits: The use of a common database for storing intermediate results offers several advantages. GATE allows simple integration of heterogeneous system components and algorithms. The user interface is also attractive.
Drawbacks: GATE is slow and memory hungry, even for medium-sized documents, and is not very robust, particularly when upgrading is carried out.

5.11 EUDICO

Goal: The aim of Eudico was a distributed multimedia infrastructure supporting annotation of speech and video corpora (Brughman et al., 1998).
Participants: Max Planck Institute for Psycholinguistics (Nijmegen, Netherlands), University of Sheffield
Description: Eudico enables transcriptions of utterances to be time-aligned with speech and video data, so that dynamic and simultaneous viewing and editing is possible. Integration with GATE was carried out in order to benefit from GATE's ability to represent, store and visualise linguistic data.
Benefits: The flexibility of GATE's data model enabled the seamless integration between EUDICO's time-based data and GATE's offset-based annotations. This enabled the representation, manipulation and display of time-aligned transcriptions into GATE's viewers, allowing the user to manipulate the different types of data simultaneously in a uniform environment.
Drawbacks: There is a certain lack of support for distributed/remote access to the document manager. Therefore in a client-server environment, the entire data has to be sent over the network instead of just the parts that are needed.

5.12 German Named Entity Recognition

Goal: German Named Entity Recognition (Mitchell, 1997) was an MSc project to adapt part of the LaSIE system to deal with German, and to test whether the architecture was suitable for processing a language other than English.
Participants: Dept. of Computer Science, Sheffield University
Description: The system followed the same general architecture as LaSIE, but with modifications to various modules such as the grammar and tokeniser.
Benefits: Using the GATE architecture meant that only fairly minor modifications to individual modules were necessary, and rule adaptation was easy. The evaluation of LaSIE as a tool for processing other languages was very positive (as borne out by the later development of M-LaSIE (a multilingual IE system).
Drawbacks: The GATE API was large, complex and difficult to understand and modify, The ability to group modules into blocks for processing would be a useful addition, as would an easier method of inserting new modules in the correct place.

6 Strengths and Weaknesses

GATE has proved successful in a number of contexts, with users reporting a variety of work with the system, for example:

- Teaching undergraduates and postgraduates. Our colleagues at UMIST and the Universities of Edinburgh, and Sussex have reported using the system for teaching, as have the Universities of Stuttgart and Saarburcken.
- Information Extraction in English, Swedish, French, Spanish and Greek. Our colleagues in Fribourg University collaborated with us on a French IE system; both ILSP and NKSR Demokritus in Athens are developing a Greek IE system; the University of Gothenburg has a Swedish system; the University of Catalonia in Barcelona are working on Spanish.
- Integrating information extraction with Information Retrieval. The Naval Office of R&D (NRaD) in San Diego is using GATE for research on text summarisation and IE/IR integration.
- Integrating a national collection of NLP tools for Swedish. See http://www.sics.se/humle/projects/svensk/
- ESTEAM Inc., of Gothenburg and Athens are using the system for adding name recognition to their MT systems (for 26 language pairs) to improve performance on unknowns.
- The Speech and Hearing group at Sheffield are modelling out-of-

vocabulary language using VIE and GATE (Gotoh et al., 1998).

- Numerous postgraduates in locations as diverse as Israel, Copenhagen and Surrey are using the system to avoid having to write simple things like sentence splitters from scratch, and to enable visualisation and management of data.

Abstracting from their experiences and that of users at Sheffield, GATE's strengths can be summarised as:

- facilitating reuse of NLP components by reducing the overheads of integration, documentation and data visualisation;
- facilitating multi-site collaboration on IE research by providing a modular base-line system (VIE) with which others can experiment;
- facilitating comparative evaluation of different methods by making it easy to interchange modules;
- facilitating task-based evaluation, both of "internal" components such as taggers and parsers, and of whole systems, e.g. by using materials from the ARPA MUC programme (Grishman and Sundheim, 1996) (whose scoring software is available in GATE, as is the Parseval tree scoring tool (Harrison, 1991), and a generic annotation scoring tool);
- contributing to the portability of NLP systems across problem domains by providing a markup tool for generating training data for example-based learning (it can also take input from the Alembic tool (Day et al., 1997) for this purpose, using Edinburgh's SGML processing library (McKelvie et al., 1997)).

There several weaknesses in the system, and some areas that are underdeveloped or lacking polish. In rough order of severity:

1. Version 1 is biased towards algorithmic components for language processing, and neglects resource components.
2. Version 1 is biased towards text analysis components, and neglects text generation components.
3. The visual interface is complex and somewhat non-standard.
4. Installing and supporting the system is a skilled job, and it runs better on some platforms than on others (UNIX vs. Windows).
5. Sharing of modules depends on sharing of annotation definitions (but isomorphic transformations are relatively easy to implement).
6. It only caters for textual documents, not for multi-media documents.
7. It only supports 8-bit character sets.

Points 1 and 2 compromise the generality of the system, and have limited take-up, as well as the number of CREOLE modules integrated with the system. For modules like taggers, parsers, discourse analysers (i.e. just about anything that performs an analysis task) the GATE integration model provides a convenient and powerful abstraction layer based on storing information in association with the text under analysis. For resources like lexicons or corpora, no such layer exists. Similarly, for modules that do generation-side tasks, since there is no text under analysis, the utility of a text-based model is limited.

For details of the means by which we intend to combat these problems and extend the range of the system, see (Cunningham, 2000). More details of requirements for this type of system, and how to evaluate them, are available in (Cunningham et al., 2000).

7 Conclusion

Based on the collective experiences of a sizeable user base across the EU and elsewhere, the system can claim to be a viable infrastructure for certain sections of the field. Given further development, we hope that it can take on this role for a wider variety of tasks.

8 Acknowledgements

This work was supported by EPSRC grants GR/K25267 and GR/M31699.

References

R. Basili, M. Pazienza, P. Velardi, R. Xatizone, R. COllier, M. Stevenson, Y. Wilks, O. Amsaldi, A. Luk, B. Vauthey, and J. Grandchamp. 1997. Extracting case relations from corpora. ECRAN Deliverable 2.4 version 1.

H. Brughman, A. Russel, P. Wittenburg, and R. Piepenbrock. 1998. Corpus-based research using the Internet. In *First International Conference on Language Resources and Evaluation (LREC) Workshop on Distributing and Accessing Linguistic Reseources*, Granada, Spain.

H. Cunningham, M. Freeman, and W.J. Black. 1994. Software Reuse, Object-Oriented Frameworks and Natural Language Processing. In *New Methods in Language Processing (NeMLaP-1), September 1994*, Manchester. (Re-published in book form 1997 by UCL Press).

H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. 1997. Software Infrastructure for Natural Language Processing. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97)*, March. `http://xxx.lanl.gov/abs/cs.CL/9702005`.

H. Cunningham, K. Bontcheva, V. Tablan, and Y. Wilks. 2000. Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC-2)*, Athens. `http://gate.ac.uk/`.

Hamish Cunningham. 2000. Software Architecture for Language Engineering. Forthcoming.

D. Day, J. Aberdeen, L. Hirschman, R. Kozierok, P. Robinson, and M. Vilain. 1997. Mixed-Initiative Development of Language Processing Systems. In *Proceedings of the 5th Conference on Applied NLP Systems (ANLP-97)*.

B. Gambäck and F. Olsson. 2000. Experiences of Language Engineering Algorithm Reuse. In *Second International Conference on Language Resources and Evaluation (LREC)*, pages 155–160, Athens, Greece.

Y. Gotoh, S. Renals, R. Gaizauskas, G. Williams, and H. Cunningham. 1998. Named entity tagged language models for lvcsr. Technical Report CS-98-05, Department of Computer Science, University of Sheffield.

R. Grishman and B. Sundheim. 1996. Message understanding conference - 6: A brief history. In *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, June.

P. Harrison. 1991. Evaluating Syntax Performance of Parsers/Grammars of English. In *Proceedings of the Workshop on Evaluating Natural Language Processing Systems, ACL*.

G. Demetriou K. Humphreys and R. Gaizauskas. 2000. Two applications of information extraction to biological science journal articles: Enzyme interactions and protein structures. In *Proc. of Pacific Symposium on Biocomputing (PSB-2000)*, Honolulu, Hawaii.

D. Kokkinakis and S. Johansson-Kokkinakis. 1999. Cascaded finite-state parser for syntactic analysis of swedish. Technical Report GU-ISS-99-2, Dept. of Swedish, Göteborg University. `http://svenska.gu.se/ svedk/-publications.html`.

D. McKelvie, C. Brew, and H. Thompson. 1997. Using SGML as a Basis for Data-Intensive NLP. In *Proceedings of the fifth Conference on Applied Natural Language Processing (ANLP-97)*, Washington, DC.

B. Mitchell. 1997. Named Entity Recognition in German: the identification and classification of certain proper names. Master's thesis, Dept. of Computer Science, University of Sheffield. `http://www.dcs.shef.ac.uk/-campus/dcscd/projects/bm.pdf`.

F. Olsson, B. Gambäck, and M. Eriksson. 1998. Reusing Swedish Language Processing Resources in SVENSK. In *Workshop on Minimising the Efforts for LR Acquisition*, Granada.

F. Olsson. 1997. Tagging and morphological processing in the svensk system. Master's thesis, University of Uppsala. `http://http://stp.ling.uu.se/ fredriko/-exjobb.ps`.

G. Petasis, G. Paliouras, V. Karkaletsis, C.D. Spyropoulos, and I. Androutsopoulos. 1999. Resolving part-of-speech ambiguity in the greek language using learning techniques. In *Proc. of the ECCAI Advanced Course on Artificial Intelligence (ACAI)*, Chania, Greece.

TRESTLE. 2000. The TRESTLE project. `http:/www.dcs.shef.ac.uk/research/-groups/nlp/trestle`.

Y. Wilks and R. Gaizauskas. 1999. Report on epsrc research grant on the large scale information extraction research project. Technical Report GR/K25267, University of Sheffield.

Composing a General-Purpose Toolbox for Swedish

Fredrik Olsson and **Björn Gambäck**
{fredriko,gamback}@sics.se

Information and Language Engineering Group
Swedish Institute of Computer Science
Box 1263, S-164 29 Kista, Sweden
http://www.sics.se/humle/ile

Abstract

The paper discusses the lessons we have learned from the work on building a reusable toolset for Swedish within the framework of GATE, the General Architecture for Text Engineering, from the University of Sheffield, UK.

We describe our toolbox SVENSK and the reasons behind the choices made in the design, as well as the overall conclusions for language processing toolbox design which can be drawn.

1 Introduction

Why is it desirable to have a general-purpose toolset for Language Engineering? In general, it is likely that the following items hold:

language diversity
: Research in Language Engineering tends to be expensive since the results may not always be shared across languages, e.g., a tagger or parser for German is not applicable to Swedish. This implies that much of the work carried out in one language has to be carried out in other languages as well.

evaluation
: Evaluation of language processing software is a cumbersome task, and it would ease up things if researchers could cooperate in constructing test-suits and measures that apply to those and then share data and methods within a common framework.

commercialisation
: If you want to go commercial, it is important that the prototyping and testing phases can be carried out without the overhead of having to construct a new framework each time a new kind of system is to be developed.

In addition, for small languages like Swedish (with about 9 million speakers), there are not that many researchers in Computational Linguistics, and thus not many at all in the various sub-fields of the area. To be able to share results between groups in the same research area is crucial for every-day research. Both to show off results and for teaching purposes.

In order to encompass this, a general framework for Language Engineering could, or *should*, be expected to:

- cut development time and cost by reusing what has been done before;
- ensure that systems are scalable to prevent unexpected draw-backs due to the "toy problem syndrome";
- provide, in the long run, a good setting for evaluation of language engineering tasks.

In this paper we will discuss the lessons we have learned from the work on building such a toolbox for Swedish; however, we set out by describing some of the reasons for why our project from the start was laid out as it was.

In particular, the section following concentrates on our own project and toolbox architecture together with University of Sheffield's underlying GATE system, while Section 3 draws on the experiences from some other, previous and current, work on designing large language processing systems. Section 4 then moves on to the central purpose of the paper, discussing the main insights gained during the course of the project. Finally, Section 5 supplies a short bullet list with some of the overall conclusions we have drawn.

2 A toolbox for Swedish: SVENSK

The SVENSK project (Eriksson and Gambäck, 1997; Olsson et al., 1998; Gambäck and Olsson, 2000) is a national effort funded by the Swedish National Board for Industrial and Technical Development (NUTEK) to encompass some of the difficulties outlined above. The aim of SVENSK has been to develop a multi-purpose language processing system for Swedish based, where possible, on existing components, and targeted at research and teaching. The SVENSK system as such is thus mainly the sum of a fairly large set of different reusable language resources.

2.1 Choice of platform

In 1995 when the SVENSK project started to take shape, there was a need for a platform flexible enough to act as a framework for the language processing programs intended to constitute the toolbox. At the end of the selection process, there were two platforms remaining; the European Commission initiative ALEP, Advanced Language Engineering Platform (Simpkins and Groenendijk, 1994) and GATE, General Architecture for Text Engineering, from the University of Sheffield, UK (Cunningham et al., 1996).

GATE was chosen since it was, among other things, freely available and did not impose its own linguistic theories on the modules to be integrated. Even though ALEP, at the time, turned out to be too slow to fit as a software framework for SVENSK, it was considered feasible to integrate external modules in it (Eriksson and Gambäck, 1997).

More general points about both these projects will be discussed below, ALEP in Section 3.3 and GATE in Section 3.5. We will right away describe the GATE system as such, though, from our perspective within the SVENSK project.

2.2 GATE

GATE consists of three different parts; a document manager, a graphical interface, and a set of language engineering objects. This section gives an overview of each one of them.

2.2.1 GATE Document Manager, GDM

The GDM, which is based on the TIPSTER database architecture (Grishman and others, 1997), serves as a communication center for the components in GATE. It stores all information about texts that language processing systems require to run, as well as the information they produce. The GDM stores annotations associated to sequences of byte offsets in the original text. Each annotation may have several attributes, which in turn may have zero or more values. The byte offsets are used as pointers into the original text in order to enable separate storage of the source text and the database holding information associated to it. Also, the GDM provides a well-defined application programming interface (API) for handling the data it stores.

2.2.2 GATE Graphical Interface, GGI

The GGI is a graphical launch-pad which enables interactive testing and building of language processing systems within GATE. Various tasks are supported, such as integrating new language processing modules, building systems, launching them, and viewing the results. The philosophy of the interface is to provide the user woth a rich set of tools. There are, for example, several generic viewers for displaying module results, ranging from raw annotations to complex parse trees via output from part-of-speech taggers.

2.2.3 Language engineering objects

At the very heart of building a GATE-based system system we find the so-called Collection of REusable Objects for Language Engineering, CREOLE. The CREOLE modules/objects in the GATE system should be thought of as interfaces to resources; data, algorithmical or a mixture of both. A CREOLE module may be a "wrapper" around an already existing piece of software or it may be an entire program developed explicitly for GATE compliance. It is the CREOLE modules that perform the real work of analysing texts in a GATE-based system.

The tasks for a CREOLE module involve setting up the environment for the language processing program it implements, or "wraps" (e.g., processing arguments given by the user via the GGI), as well as retrieving information from the GDM, invoking the program, and taking care of the output produced, that is, format it and record it in the GDM.

2.3 CREOLE modules in SVENSK

From GATE's point of view, SVENSK is a set of CREOLE objects. The language processing software wrapped by the CREOLE objects are in-house modules, commercially available mod-

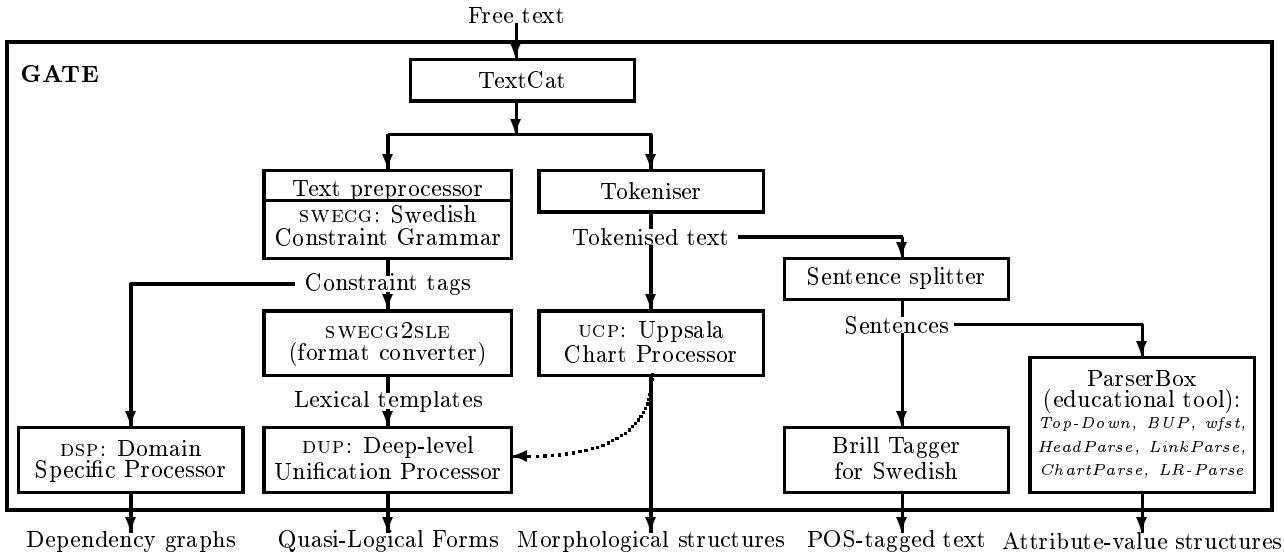

Figure 1: How the modules in SVENSK are interconnected to form different processing chains.

ules, and modules from Swedish academia. The modules integrated so far are shown in Figure 1.

As indicated in the figure, there are different ways the input texts can take through the system. At the top end of the picture, van Noord's freely available[1] language identifier TextCat constitutes the starting point for all processing chains in SVENSK. Here, it allows the user to restrain the input to the system to be in Swedish.

We then have two main options, either to pass the input through SWECG, the Swedish version of LingSoft oy's and Helsinki University's Constraint Grammar (Karlsson et al., 1995), or through a parallel sequence of tokenisation and sentence segmentation developed specifically for the SVENSK project (Olsson, 1998).

The processing chains then split further, and — from left to right in Figure 1 — end in the following modules;

- DSP, the Domain Specific Processor (Sunnehall, 1996) produces shallow dependency graphs intended for use in applications requiring a robust interface for a specific application, such as the Olga dialogue system (Beskow et al., 1997);
- DUP, the Deep-level Unification-based Processor, a component made up of a large-scale unification-based grammar for Swedish (Gambäck, 1997) and an LR-parser (Samuelsson, 1994). It yields a relatively 'deep' level of analysis but at the cost of robustness, and has previously been used for machine translation and database interfacing projects, including the SICS-SRI-Telia "Spoken Language Translator" (Rayner et al., 1993).
- the Uppsala Chart Processor (Sågvall-Hein, 1981) produces morphological analyses;
- a Swedish version (Prütz, 1997) of the Brill Tagger (Brill, 1992);
- the ParserBox, an educational tool consisting of seven parsers operating on a small grammar. At this end the different ways the parsers process the input is of main interest, rather than the output produced.

Each of the components has a standardised input/output interface, users will have the choice of working with the supplied development system — as may be appropriate for academic research on particular aspects of language use — or selecting and combining modules for integration into a user application. Since the I/O interfaces conform to the annotation model of the TIPSTER architecture (Grishman, 1995) developers (and users) can easily add components to the platform, and then link them together to form an application.

[1] At `www.let.rug.nl/~vannoord/TextCat`

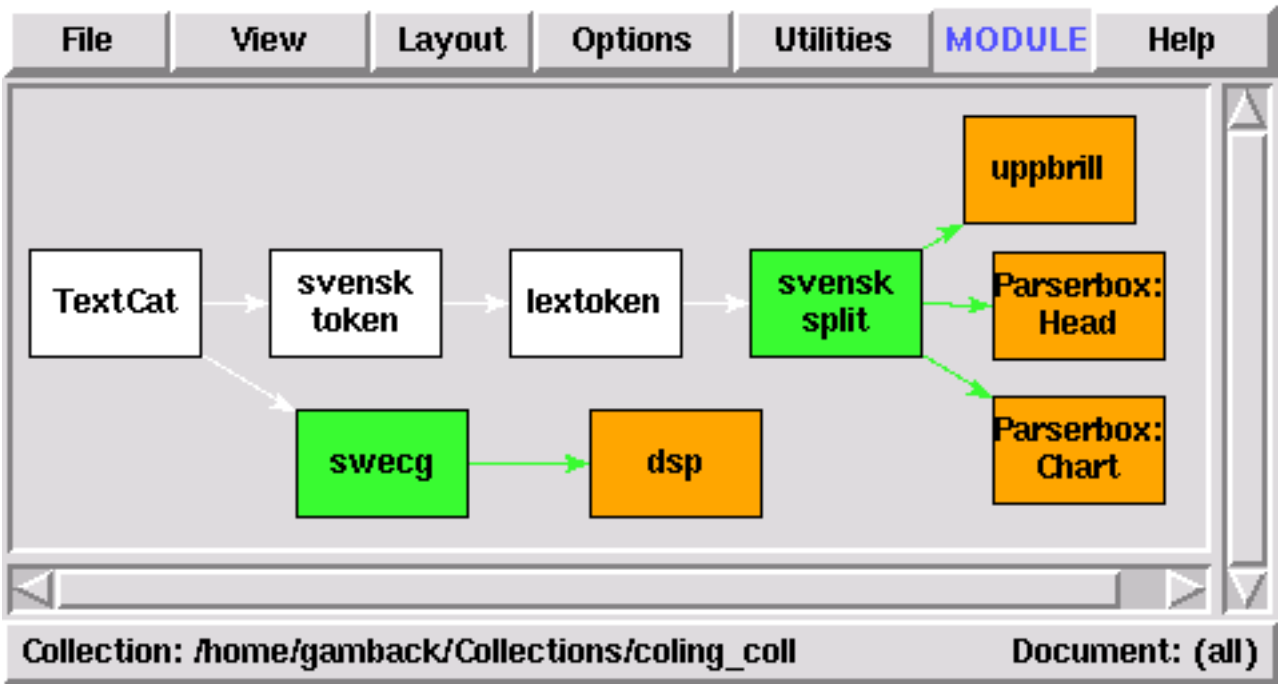

Figure 2: An example of a system built by some of the SVENSK modules.

Alternatively, two components with the same interfaces and functionality can be defined for the platform, and then evaluated in the same application. This allows students to experiment with different approaches to a linguistic problem (such as parsing, using the algorithms supplied on the ParserBox), or research experiments to use the most appropriate component for their purposes and performance criteria (such as speed, robustness, etc).

Figure 2 shows how some of the SVENSK modules can be linked within GATE to form four different processing chains, in this case with three possible output levels: dependency-based semantics (from DSP), POS-tagged text (from Brill), or syntactic analyses obtained from either a chart parser or a head-based parsing strategy.

3 Related work

Over the years there have been many efforts in the direction of creating large toolsets for language processing. Some have been built with one particular application — or class of applications — in mind, but mostly the more or less explicit aim has been to create reusable toolsets for a wide range of tasks. In this section we will look at some of the major stepping stones.

3.1 Setback 1: Eurotra

Back in 1977, the first steps were taken towards what would become the most ambitious effort in the field seen so far. The goal of the Eurotra Programme was to develop a machine translation system collaboratively in all the member-states of the (then) European Community. A working group tried to establish linguistic and software standards as the basis for the project, but the amount of work done in this group was intended to be small, while the main efforts were to be localised to centra in the different states, working on some of the (at the end) nine languages and 72 language pairs (King and Perschke, 1987; Bech and Nygaard, 1988).

After the project, "Eurotra bashing" has developed into something of a sport for European computational linguists, resulting in that the reasons for *why* the project failed (at least partially[2]) have in themselves not been dis-

[2] 'Failure' is a relative notion, since it was not originally a goal of the Eurotra Programme to build one production-quality system. The degrees of freedom left for the different groups had the positive effect of building up language processing competence and infrastructure in the EC countries and of producing some working, full-scale "spin-off" systems, such as PaTrans (Hansen, 1994).

cussed enough. Certainly, the lack of overall coordination soon became a liability. Inherent short-comings of the formalisms, and inefficiency of the implementation related to fundamental problems with the formalisms are another reason which have been pointed out (Crookston, 1990; Pulman et al., 1991).

However, the main problem with the framework was probably that it never in itself moved towards one system; indeed, Johnson and Rosner (1987) discussed a software environment for Eurotra building on tools for rapid implementation and evaluation of a variety of experimental theories. In the spirit of that several parallel systems and formalisms were used, and the formalisms changed rapidly over time. Still, no framework was developed which could accommodate these different types of modules, no clear interfaces were designed, no central instance under-took the integration task.

3.2 Success Story 1: CLE

In contrast to the multi-site Eurotra effort, 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 (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. 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]

3.3 Setback 2: ALEP

Following the Eurotra tradition, ALEP, the Advanced Language Engineering Platform (Simpkins and Groenendijk, 1994; Bredenkamp et al., 1997), introduced in Section 2.1, was another European Commission initiative to provide the European language research and engineering community with a general purpose research and development environment.

The ALEP platform supplied a range of processing resources and was particularly targeted at supporting multilinguality. However, it imposed its own formalisms (for grammars, etc.) on the developers and users. In addition, the initial implementations were as inefficient as Eurotra's and ALEP never became widely spread.

3.4 Success Story 2: Verbmobil

The real contrast to Eurotra came with Verbmobil, a multi-site German government effort which started in the early 90's (Kay et al., 1994). The main processing stream of the project was clearly defined (even though parallel and complementary modules were allowed) and the interfaces between different groups and modules were developed during the project in intense inter-group discussions.

A major reason why the project succeeded in producing an overall joint system was a concentrated effort by a central system administration group which incorporated components developed at several different sites and in many different programming paradigms into one platform (Bub and Schwinn, 1996). The Verbmobil architecture employs ICE, Intarc Communication Environment (Amtrup, 1997), a general communication package, but which of course has been primarily developed for the specific needs of the Verbmobil task, non-incremental multilingual spoken dialogue translation.

3.5 Success Story 3: GATE

In the mid 90's, the UK Engineering and Physical Sciences Research Council (EPSRC) started to fund a project at the University of Sheffield aimed at building a General Architecture for Text Engineering, GATE (Cunningham et al., 1996; Gaizauskas et al., 1996; Cunningham et al., 1997; Cunningham et al., 1999).

As described in Section 2.2, GATE 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. It presents users with an environment in which it is easy to use and integrate tools and databases, all accessi-

[3] "You don't give away a one million pound program" (SRI research manager). Contrast this with the strategy of, e.g., John McAfee to give away his antivirus software for free — and making millions on selling the updates!

Table 1: Language processing resources in SVENSK

Processing resource	Main task	Author(s)
TextCat	Language identification	[van Noord; U Groningen]
Tokeniser	Tokenisation	[Olsson; SICS & Uppsala U]
LexToken	Lexicalised phrase tokenisation	[Hassel, Johansson; SICS & Sthlm U]
Sentence splitter	Segmentation	[Olsson; SICS & Uppsala U]
Swedish Brill Tagger	Part-of-speech tagging	[Prütz; Uppsala U]
Uppsala Chart Processor	Morphology	[Sågvall-Hein; Uppsala U]
LP-Detect	Lexicalised phrase recognition	[J. Lindberg; Stockholm U]
Swedish Constraint Grammar	Morphosyntactic analysis	[LingSoft oy & Helsinki U]
SWECG2CLE	Format converter	[Eriksson; SICS]
Deep-level Unification-based Processor	Swedish grammar & LR-parser	[Gambäck; SICS], [Samuelsson; SICS]
Domain-specific Processor	Dependency structure semantics	[Sunnehall; SICS]
ParserBox	Educational tool	[Eineborg, Olsson et al; SICS]

ble through a friendly user interface. The platform is free for non-commercial and research purposes, and has so far been distributed to more than 250 different sites around the world.[4]

3.6 Meanwhile in the US... Galaxy

In the US, there have also been some efforts in the direction of open architectures that incorporates language processing resources, in particular within the research programmes sponsored by DARPA, the Defense Advanced Research Projects Agency. With TIPSTER (Grishman, 1995), the design of a general architecture was agreed upon; however, the full TIPSTER annotation scheme (Grishman and others, 1997) has not been implemented as such. Instead, the MITRE Cooperation is currently (under DARPA funding) developing Communicator, a testbed similar to the Verbmobil one.

The initial DARPA Communicator architecture builds on MIT's Galaxy system (Seneff et al., 1998). A central process, the Hub, is connected with a variety of server processes and governs the control flow between them. A wide range of component types are supported: language understanding and generation, speech recognition and synthesis, dialogue management, and context tracking (Goldschen and Loehr, 1999).

The goal of the Communicator — to provide an architecture used by everyone, easing the work of porting modules and system evaluation — seems decent in itself; however, the system has not been that well received within the US research community. ("We've spent most of our time the last year trying to make our stuff follow Communicator standards, rather than on doing research," anonymous US researcher, personal communication 2000).

4 Issues in composing a toolset

A result of the integration in SVENSK is that programs from many different sources and backgrounds, which originally were not built to communicate which each other are doing this now. Table 1 shows the main tasks of all the SVENSK modules, as well as the author(s) and sources behind the different units.

Collecting and distributing algorithmic resources and making different programs interoperate present a wide range of challenges, along several different dimensions; we will denote the key dimensions 'diplomatic', technical, and linguistic. In the rest of this section we will discuss some of the experiences we have drawn from the project with regards to these dimensions.

[4] In June 2000, according to the "incomplete" list of licensees given on the GATE web pages: `www.dcs.shef.ac.uk/research/groups/nlp/gate`

4.1 Diplomatic challenges

With 'diplomatic', we mean some of the conclusions which can be drawn from the examples of other systems in Section 3. Eurotra and ALEP both had the problem of linguists not wanting to agree on formalism standards while a framework supporting diversity was lacking. The CLE was successful as a system, but commercial shortsightedness destroyed its chances of more widespread popularity.

Commercial interests have also been a problem within SVENSK, but we have also seen that it is hard to get access to academic LE resources. The need for component reuse is often appreciated by everybody in the field. However, to put action behind words is not as easy. In particular, researches need to be convinced to invest the extra time and resources to package their components in an exportable and reusable form.

Table 2: Resource availability

Availability	Resource(s)
In-house and free	DSP, DUP, Tokeniser, LexToken, ParserBox, Sentence splitter
External and free	TextCat, GATE, LP-Detect
External, restricted	UCP, Swedish Brill
Commercial, closed	SWECG

A key aim of the project has been that the resources included in SVENSK should be freely available for non-commercial use, at least for Swedish institutions. As can been seen in Table 2 all current components except for SWECG meet this requirement.[5] Of course, processing resources included in the system in the future should preferably also match this free-for-all strategy.

Still, making language processing resources freely available and, in particular, reusability 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. Yes, this is the very core of what is accepted as good research at all. Strangely enough, this is rarely the case in Computer Science in general and definitely not within Computational Linguistics.

We believe that this will change and that reproducibility will be generally accepted as a criteria of good research even in Computational Linguistics. And to give other researchers the option of reproducing an experiment means giving them access to the language engineering resources used in the experiment. Convincing the members of the CL research community to both make their own processing resources freely available to the rest of the community and actually even to try to reuse somebody else's resources is indeed a tough 'diplomatic' challenge.

4.2 Technical/software challenges

From the technical point of view, one major conclusion is that the difficulties of integrating language processing software never can be overestimated. Even when using a liberal framework like 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.

To give the flavour of the problem, Table 3 singles out the underlying implementation languages of some of the SVENSK components, while diversity in software authors and sources was shown already in Table 1.

Table 3: Implementation languages

Language	Resource(s)
Prolog	DSP, DUP, ParserBox
C/C++	SWECG, Brill, Tokeniser, GATE (part)
Tcl/Tk	GATE
Perl	TextCat, Brill (part), LexToken, LP-Detect, Sentence splitter
LISP	UCP

The application programming interface thus moves to the centre of attention: No matter how

[5] SWECG, the Swedish Constraint Grammar, is available from LingSoft oy, Helsinki, unfortunately for a currently discouragingly high license fee, albeit reduced for academic SVENSK users.

linguistically adequate a piece of language processing software is, without a proper API it cannot be used in conjunction with other programs.

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. The trouble 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. This probably reflects a certain level of immaturity in the field when it comes to software development; the problem might solve itself when the level of competition between different companies increases.

Software portability is another issue: The components available often rely on a particular operating system or a particular software environment to work, something which may cause problems in settings when you wish to distribute your collected efforts to other parties, or when you wish to add a new component to your collection. If you are not careful when integrating components that are not first and foremost intended to function together, it is not likely that their combined performance will even level with the performance of the individual programs regarding, e.g., time and memory requirements.

In software industry in general, it is hard to recreate the situations where bugs occur, and it is even harder to correct them once you have found them. When collecting and integrating a set of heterogenous language processing components, the problem of localising a bug is even harder. As long as the source code of the software under consideration is available, you might be able to correct the bugs yourself; however, software delivered as a "black-box" (that is, if it is impossible to access the code inside, which is common for commercial software), allow no one to remedy even the smallest flaw.

4.3 Linguistic challenges

Of course, LE 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 done at various levels of proficiency. The trouble is that there is no quality control available to either the toolbox developer nor to the end-user. If a large number of LE 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 an LE resource is available only in the "black box" form (and thus relates to the issues of the previous subsection).

In general, the power required of a language processing system is affected by three main factors: the type of task involved, the needs of the specific application, and the application domain, including the vocabulary and the register (sublanguage) complexity. It seems impossible — or at least *very* hard! — to compose a really general toolkit. Toolkits will always have to focus on some classes of tasks and applications and/or on some language and operation domains.

A classification of the SVENSK components according to three different limitation dimensions is given in Table 4.

Table 4: Linguistic limitations

Limitations	Resource(s)
Language dependent	All except TextCat and Sentence splitter (some)
Domain dependent	DSP (most others more or less)
Sentence-based	DSP, DUP, ParserBox

5 Conclusions

- A toolset should not be *too* general. There has to be some focus on its end-usage, at least to some manageable set of classes of tasks and applications.
- The portability issues across operating systems as well as institutional borders depends on technical issues such as licenses and availability. The "*a chain is never stronger than its weakest link*"-metaphor is certainly applicable!
- The domain and coverage of language processing software is an additional obstacle; it is important to match pieces of software accordingly!

Acknowledgements

The SVENSK project has been funded by SICS and NUTEK under grants P5475 and P13338-1.

Several persons have been involved in the project since its start in 1995, in particular Mikael Eriksson who did the first work on the GATE interfacing. Mats Wirén and Barbro Atlestam were the key persons behind ensuring the project's funding. Scott McGlashan, Charlotta Berglund, Victoria Johansson and Kristina Hassel all worked directly on SVENSK at SICS, while Christer Samuelsson, Jussi Karlgren, Nikolaj Lindberg, Lena Santamarta and Ivan Bretan worked on other SICS projects which contributed indirectly to the system.

Klas Prütz, Anna Sågvall-Hein and Jan "Beb" Lindberg donated language processing resources to the project. Gertjan van Noord sat an example for all researchers by making his software freely available, as did Hamish Cunningham and the rest of the Sheffield GATE group. Thanks also to the members of SVENSK's scientific advisory board which have greatly influenced the project: Barbro, Anna, Mats, Robin Cooper, Lars Ahrenberg, and Calle Welin.

References

Hiyan Alshawi, editor, 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, Mass.

Jan W. Amtrup. 1997. ICE: A communication environment for Natural Language Processing. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada.

Annelise Bech and Anders Nygaard. 1988. The E-framework: A formalism for Natural Language Processing. In *Proc. 12th International Conference on Computational Linguistics*, volume 1, pages 36–45, Budapest, Hungary. ACL.

Jonas Beskow, Kjell Elenius, and Scott MacGlashan. 1997. Olga — a dialogue system with an animated talking agent. In G. Kokkinakis, N. Fakotakis, and E. Dermatas, editors, *Proc. 5th European Conference on Speech Communication and Technology*, volume 3, pages 1651–1654, Rhodes, Greece. ESCA.

Andrew Bredenkamp, Thierry Declerck, Frederik Fouvry, Bradley Music, and Axel Theofilidis. 1997. Linguistic engineering using ALEP. In (Mitkov and Nicolov, 1997), pages 92–97.

Eric Brill. 1992. A simple rule-based part of speech tagger. In *Proc. 3rd Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy. ACL.

Thomas Bub and Johannes Schwinn. 1996. Verbmobil: The evolution of a complex large speech-to-speech translation system. In *Proc. 4th International Conference on Spoken Language Processing*, Philadelphia, Pennsylvania.

Ian Crookston. 1990. The E-framework: Emerging problems. In H. Karlgren, editor, *Proc. 13th International Conference on Computational Linguistics*, volume 2, pages 66–71, Helsinki, Finland. ACL.

Hamish Cunningham, Yorick Wilks, and Robert J. Gaizauskas. 1996. GATE — a General Architecture for Text Engineering. In *Proc. 16th International Conference on Computational Linguistics*, volume 2, pages 1057–1060, København, Denmark. ACL.

Hamish Cunningham, Kevin Humphreys, Robert J. Gaizauskas, and Yorick Wilks. 1997. Software infrastructure for Natural Language Processing. In *Proc. 5th Conference on Applied Natural Language Processing*, Washington, DC. ACL.

Hamish Cunningham, Robert J. Gaizauskas, Kevin Humphreys, and Yorick Wilks. 1999. Experience with a Language Engineering architecture: Three years of GATE. In *Proc. Workshop on Reference Architectures and Data Standards for NLP*, Edinburgh, Scotland. AISB.

Mikael Eriksson and Björn Gambäck. 1997. SVENSK: A toolbox of Swedish language processing resources. In (Mitkov and Nicolov, 1997), pages 336–341.

Robert Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter Rodgers, and Kevin Humphreys. 1996. GATE: An environment to support research and development in Natural Language Engineering. In *Proc. 8th International Conference on Tools with AI*, Toulouse, France. IEEE.

Björn Gambäck. 1997. *Processing Swedish Sentences: A Unification-Based Grammar and*

some Applications. PhD Thesis, The Royal Institute of Technology, Dept. of Computer and Systems Sciences, Stockholm, Sweden.

Björn Gambäck and Fredrik Olsson. 2000. Experiences of Language Engineering algorithm reuse. In *Proc. 2nd International Conference on Language Resources and Evaluation*, volume 1, pages 161–166, Athens, Greece. 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. SISO.

Ralph Grishman et al., 1997. *TIPSTER Text Phase II Architecture Design. Version 2.3.* New York, NY.

Ralph Grishman, 1995. *TIPSTER Phase II Architecture Design Document (Tinman Architecture) Version 1.52.* New York, NY.

Viggo Hansen. 1994. PaTrans — a MT-system: Development and implementation of and experiences from a MT-system. In *Proc. 1st Conference of the Association for Machine Translation in the Americas*, pages 114–121, Columbia, Maryland. AMTA.

Rod Johnson and Mike Rosner. 1987. Machine translation and software tools. In (King, 1987), chapter 11, pages 154–167.

Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila, editors. 1995. *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter, Berlin, Germany.

Martin Kay, Jean Mark Gawron, and Peter Norvig. 1994. *Verbmobil: A Translation System for Face-to-Face Dialog.* Number 33 in Lecture Notes. CSLI, Stanford, California.

Maggie King, editor. 1987. *Machine Translation Today: the State of the Art.* Edinburgh University Press, Edinburgh, Scotland.

Maggie King and Sergei Perschke. 1987. EUROTRA. In (King, 1987), chapter 19, pages 373–391.

Ruslan Mitkov and Nicolas Nicolov, editors. 1997. *Proc. 2nd International Conference on Recent Advances in Natural Language Processing*, Tzigov Chark, Bulgaria.

Fredrik Olsson. 1998. Tagging and morphological processing in the SVENSK system. MA Thesis, Uppsala University, Sweden.

Fredrik Olsson, Björn Gambäck, and Mikael Eriksson. 1998. Reusing Swedish language processing resources in SVENSK. In *Proc. Workshop on Minimizing the Effort for Language Resource Acquisition (LREC98)*, pages 27–33, Granada, Spain. ELRA.

Klas Prütz. 1997. Preparing a training corpus in Swedish for training an automatic part of speech tagging system. In H. Kalverkämper and B. Svane, editors, *Übersetzen und Dolmetschen. Forschungsstand und Perspektive. Translation and Interpreting. State and Perspectives. Proc. Humboldt-Stockholm Symposium*, Stockholm University, Sweden.

S. G. Pulman, editor, H. Alshawi, D. J. Arnold, D. M. Carter, J. Lindop, K. Netter, J. Tsujii, and H. Uszkoreit. 1991. *Eurotra ET6/1: Rule Formalism and Virtual Machine Design Study.* CEC, Luxembourg.

Manny Rayner, Ivan Bretan, David Carter, Michael Collins, Vassilios Digalakis, Björn Gambäck, Jaan Kaja, Jussi Karlgren, Bertil Lyberg, Steve Pulman, Patti Price, and Christer Samuelsson. 1993. Spoken language translation with mid-90's technology: A case study. In *Proc. 3rd European Conference on Speech Communication and Technology*, volume 2, pages 1299–1302, Berlin, Germany. ESCA.

Anna Sågvall-Hein. 1981. An overview of the Uppsala Chart Parser version 1 (UCP-1). Technical report, Center for Computational Linguistics, Uppsala University, Sweden.

Christer Samuelsson. 1994. *Fast Natural-Language Parsing Using Explanation-Based Learning.* PhD Thesis, The Royal Institute of Technology, Dept. of Computer and Systems Sciences, Stockholm, Sweden.

Stephanieo Seneff, Ed Hurley, Raymond Lau, Christine Pao, Philipp Schmid, and Victor Zue. 1998. Galaxy–II: A reference architecture for conversational system development. In *Proc. 5th International Conference on Spoken Language Processing*, volume 3, pages 931–934, Sydney, Australia.

Neil Simpkins and Marius Groenendijk. 1994. The ALEP project. Technical report, Cray (now Anite) Systems / CEC, Luxembourg.

Joel Sunnehall. 1996. Robust parsing using dependency with constraints and preference. MA Thesis, Uppsala University, Sweden.

An Experiment in Unifying Audio-Visual and Textual Infrastructures for Language Processing Research and Development

Kalina Bontcheva[†], Hennie Brugman[‡], Albert Russel[‡], Peter Wittenburg[‡], Hamish Cunningham[†]

[†] Department of Computer Science, University of Sheffield, Sheffield, UK

<kalina,hamish>@dcs.shef.ac.uk

[‡] Max-Plank Institute for Psycholinguistics, Nijmegen, The Netherlands

<firstname.lastname>@mpi.nl

Abstract

This paper describes an experimental integration of two infrastructures (Eudico and GATE) which were developed independently of each other; for different media (video/speech vs. text) and applications. The integration resulted into gaining an in-depth understanding of the functionality and operation of each of the two systems in isolation, and the benefits of their combined use. It also highlighted some issues (e.g., distributed access) which need to be addressed in future work. The experiment also showed clearly the advantages of modularity and generality adopted in both systems.

1 Introduction

This paper describes the integration of two infrastructures (Eudico and GATE) which were developed independently of each other; and for different media (video/speech vs. text) and applications. Such integration was needed in order to have an application where the end users (linguists and language engineers) who annotate video and speech corpora with textual transcriptions in Eudico, can also benefit from language processing tools and viewers from GATE.

Eudico (European Distributed Corpora) is a distributed multimedia infrastructure supporting creation, presentation and analysis of annotations of speech and video corpora (Brugman et al., 1998). Annotations of all kinds of user-definable types can be time-aligned with the speech/video data so that dynamic and simultaneous viewing is possible. For example, when the user sets a new media time this time is reflected in all annotation viewers, and, vice versa, when the user selects an annotation this time selection is shown in all viewers, including the media player: viewing of media and transcription data is synchronised.

GATE (General Architecture for Text Engineering) (Cunningham et al., 1997; Cunningham et al., 1999) is an architecture, framework, and development environment, providing representation and storage of language data together with infrastructural support for building and deploying language engineering applications. Its plug-and-play support for processing modules and data viewers lowers the overhead of building such applications and facilitates code-reuse.

The idea behind this experiment is to combine and build on the strengths of these two architectures, thus bridging the gap between transcriptions of speech and video, and language engineering tools. From the user's perspective, this entails simultaneous manipulation of media, transcriptions and linguistic data in a uniform and synchronised way. The integration with GATE provides Eudico applications with ways to represent, display and manipulate linguistic data and, more importantly, to run GATE language processing modules on the text transcriptions (e.g., part-of-speech tagger, name-entity recogniser). In this way, linguists and language engineers developing speech/video corpora are assisted in the corpus annotation task by language processing tools and viewers.

The main question that had to be answered is whether it is possible to integrate into one application two separate architectures. The major difference between the two comes from the media structuring: speech/video annotation in Eudico is time-based while text annotation in GATE is offset-based. Therefore, we had to find a way of storing and accessing time information for offset-based linguistic objects. From an implementational viewpoint, this entailed:

1. Mapping objects from the GATE world to objects in the Eudico world in such a way that Eudico's views give a meaningful representation of GATE data, while keeping their dynamic and synchronised nature.
2. Embedding GATE viewers in Eudico viewers and making them time aware with minimal rewriting of existing code.

The next three sections are devoted to discussing the design and implementational aspects of these two problems. Section 2 describes how the actual mapping of objects between the architectures is done. Section 3 describes the synchronisation mech-

anism allowing GATE and Eudico viewers to operate and update in parallel with the media being played. Section 4 describes the specific GATE viewers used in this project. The pilot application is described in section 5. Section 6 concludes this report by summarising the outcome of this work and pointing to a set of open issues.

2 Integrating the Two Data Models

2.1 Eudico Data Model

The key concepts underlying Eudico's data model are:

Corpus - a collection of Transcriptions (or of subcorpora).

Transcription - all annotations that refer to one media file, or that describe one uninterrupted recorded event.

Tier - a collection of Tags that are strictly consecutive in time and that annotate one specific phenomenon. Tiers can also have meta information (e.g. name, transcriber).

Tag - a typed set of values applying to a time interval. A Tag is part of exactly one Tier. Tags on the same Tier may not overlap in time. Tags are either explicitly linked to media time or ordered in time.

These concepts are part of Eudico's Abstract Corpus Model (ACM). The ACM was designed to abstract the specifics of corpus annotation formats from the tools that work on the annotation data, thus making Eudico corpus format independent.

2.2 GATE Data Model

GATE's data model is based on the TIPSTER architecture (Grishman, 1996). The main classes are:

Collection - a set of Documents which can be loaded, stored, and processed together.

Document - consists of a document content (e.g. text) and a set of Annotation objects associated to the document content by means of spans (objects specifying the begin and end offset of the annotation).

Annotation - has a set of spans (specify the text parts covered by this annotation), type (e.g. part-of-speech (POS)), and a set of Attribute objects which hold further information about the annotation (e.g. time=12345, category="Noun").

Attribute - a feature-value pair which can hold any type of data as a value. Attributes can be associated to any of the above classes - collection (e.g., creator), document (e.g., language, media type), and annotation (see above).

2.3 Mapping between the two worlds

There is a two-way mapping between Eudico and GATE objects since GATE objects are constructed from Eudico ones (with a special 'Eudico-to-Gate'tool) that were initialised from another existing corpus, and, vice versa, Eudico objects are created from GATE ones when a stored Gate collection is loaded in Eudico for further processing. This mapping is realised by implementing the proper parts of the Eudico's ACM using GATE's API and data structures. We chose to specify the mapping only for a set of data that can be treated in a meaningful way in both environments, namely transcribed speech utterances.

Eudico corpora are mapped to GATE collections. All transcribed utterances on a given media file are ordered according to their place in the original file and form a *Document*. All the information in the document is accessed through time attributes and annotation spans. In this way, only the relevant document part(s) are manipulated at each given time point. All *Tag*s from the *Tier*s will are added as *Annotations* of type `utterance`. *Tier* information is encoded as an *Attribute* on the `utterance` annotations created for the *Tag*s. *Tier* objects can be re-created by selecting all `utterance` annotations which have the same value for the attribute `tier` (see the example below). In order to provide a two way link between *Tag*s and `utterance` annotations, all Tags have associated span information and all `utterance`s have associated time information.

Based on the `utterance` annotations, which provide a link to Eudico's time-based world, new linguistic data in the form of other annotations can be added now to the media corpus.

We experimented with two types of such data:

- `POS` - part of speech annotations which are currently obtained from hand-annotation.
- `syntaxTree` - syntax trees obtained by manual annotation.

2.4 Example

The following example of the mapping is based on an example of a time-aligned file from the CHILDES Corpus (MacWhinney, 1999).

```
@Begin
@Filename: boys73.cha
@Participants
ROS Ross Child, MAR Mark Child,
FAT Brian Father, MOT Mary Mother
*ROS: yahoo.
%snd: "boys73a.aiff" 7349 8338
*FAT:   you got a lot more to do # don't you?
%snd: "boys73a.aiff" 8607 9999
*MAR: yeah.
%snd: "boys73a.aiff" 10482 10839
*MAR:   because I'm not ready to
        go to <the bathroom>[>] +/.
%snd: "boys73a.aiff" 11621 13784
```

This file is first parsed and Eudico Tiers and Tags are created as follows:

```
Tiers: ROS, MAR, FAT, MOT.
Tags for ROS: {(7349, 8338, "yahoo")}
Tags for FAT: {(8607, 9999,
     "you got a lot more to do# don't you?"}
Tags for MAR: {(10482, 10839, "yeah"),
               (11621, 13784, "because..+/.")}
```

Speech utterances are combined into a GATE document:

```
yahoo. You got a lot more to do# don't you?
0...|5...|10..|15..|20..|25..|30..|35..|40..

yeah. because I'm not ready to go to...
45..|50..|55..|60..|65..|70..|75..|80..
```

Utterance annotations are created for each Tag:

Id	Type	Span start	Span end	Attribute
1	utterance	0	5	Tier=ROS, Time=(7349,8338)
2	utterance	7	43	Tier=FAT, Time=(8607,9999)
3	utterance	45	49	Tier=MAR, Time=(10482,10839)
4	utterance	51	102	Tier=MAR, Time=(11621,13784)

3 Time-based Synchronisation

3.1 Encoding Time Information for Linguistic Objects

Eudico objects can be time-aligned which provides a way of synchronising all Eudico viewers by always showing the information relevant to the current point of time in the media. Therefore it is desirable for GATE linguistic objects to be time-aligned as well when such information is available.

This is achieved by encoding the time in milliseconds as a value of an *Attribute* object, which is then associated with linguistic *Annotation* objects (e.g. `POS` annotations). In this way, annotations with such an attribute are linked to the media time and can be manipulated and displayed in the same way as Eudico objects.

For example, given a transcription tier for one speaker, `POS` annotations are added using GATE. Each `POS` annotation has to be associated with both a time interval and a text span.

The procedure is as follows:

- Use GATE to add `POS` Annotations
- Display the `POS` annotations in a GATE viewer
- Select an Annotation in this viewer (for multi-span annotations, select one of the spans)
- Associate the span of this annotation with a time interval that is taken from a time selection set with the Eudico media player.

3.2 Using Time Information to Synchronise GATE Viewers

Eudico has a time-handling mechanism where, at the time of creation of a new viewer, all relevant time points are registered with the media player. These time points are derived from the time information of the annotation objects that are to be displayed in the viewer. During media playback an event is generated for each timepoint and based on these events, the viewers update themselves to reflect the current time in their own specific way. This mechanism was extended to the domain of GATE viewers in a non-trivial way. Extra timepoints are taken from the attributes of time-aligned annotations and also registered with the media player. Additional events are generated at play back time and passed on by the Eudico viewer to the proper embedded GATE viewer. These events can then be used by the GATE viewer to show/highlight annotations at the appropriate time, resulting in time synchronisation between all Eudico and GATE viewers. Examples are given below.

4 Re-using GATE Viewers inside the Eudico Interface

4.1 Disguising GATE Viewers as "Native" Eudico GUI classes

In order to provide creation, editing, and visualisation of linguistic data, we embedded GATE GUI modules into classes implementing Eudico's interface for a panel displaying a single Tag. In this way, the various Eudico viewers can manipulate them in the same way as the "native" Eudico ones. The wrapper classes also provide the time synchronisation functionality described in the previous section. We experimented with two GATE viewers - POS and SyntaxTree viewers - corresponding to the chosen linguistic annotations.

The POS viewer works at the level of orthographic transcription because that is a type of text that can be handled meaningfully in both the GATE and Eudico domains. Utterances at this level are almost all already time-aligned which means that the POS viewer can operate sensibly even if no finer time alignment exists. Words in the utterances can be selected for manual annotation with part of speech data (the annotation dialog is shown in Figure 1). These annotations are then visualised by textually aligning them with the proper span of the speech utterance.

The TreeViewer displays an utterance together with one or more (partial) syntax trees of sentences within this utterance (see Figure 2). The annota-

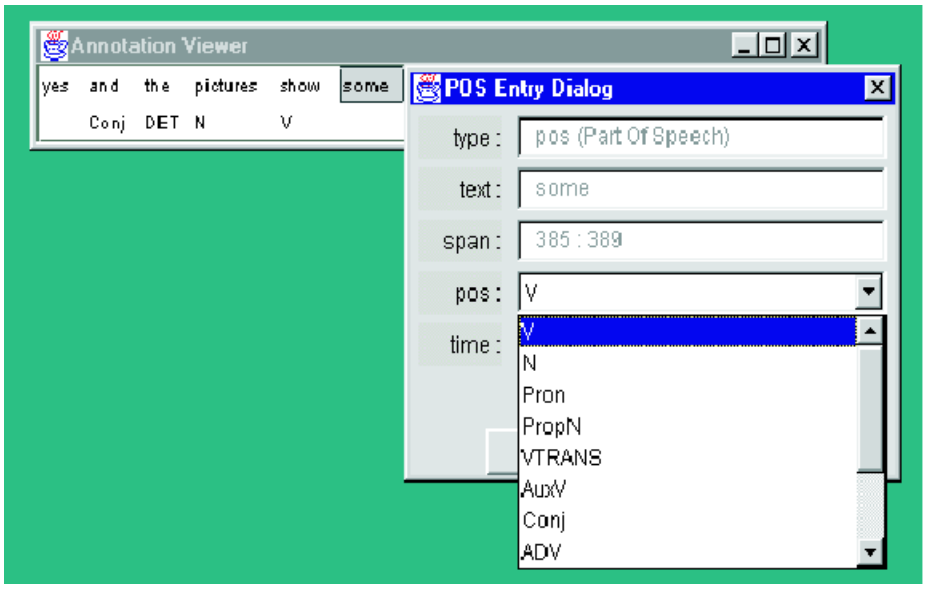

Figure 1: The Part-Of-Speech (POS) viewer

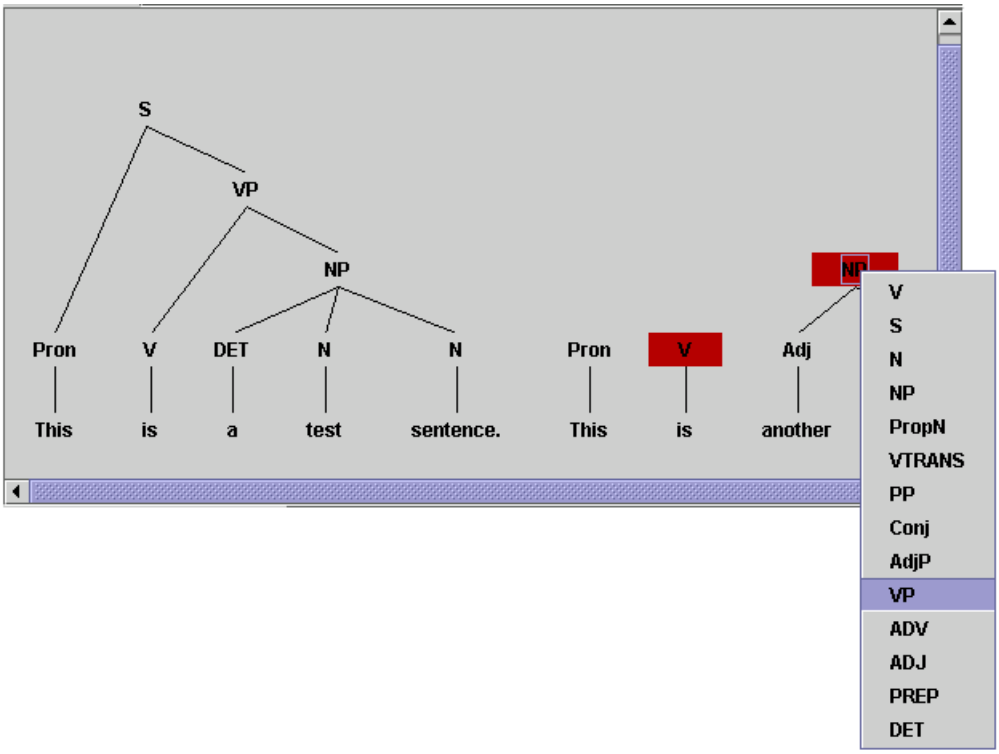

Figure 2: The TreeViewer

tion process starts from the words and proceeds upwards. The appropriate category can be selected from a menu of availabe categories which is constructed from the corresponding stereotype (see following section). Non-terminal nodes can be combined into higher-level non-terminals (e.g. V and NP into a VP) by selecting all relevant nodes for the new category and selecting the category itself from the menu. Deletion of selected nodes is avaialble too.

Similar to all GATE viewers, both the POS and Syntax Tree viewers are implemented as Java Beans which enables their easy reuse and config-

uration. On-line demonstrations of GATE applications embedding these viewers are available at `http://www.gate.ac.uk`[1].

4.2 Configuring Viewers for Types of Annotations

Annotations in GATE are flexible and unconstrained data structures. This has the advantage that the architecture is theory-neutral, and can represent a wide range of data. The disadvantage is that the application using them needs meta-information about annotations in order to allow proper creation, editing, and visualisation. Therefore, annotation stereotypes are used for encoding this information, which is then used for configuring the behaviour of the GATE viewers and editors.

These stereotypes (AnnotationStereotype class) have

- annotation type (e.g. POS, SyntTree)
- annotation structure type (single span, multiple span, tree or graph)
- a set of attribute stereotypes specifying the name of the attribute feature, the type of permissible values (e.g., String), and a list of permissible values where applicable.

For example, the stereotype of POS annotations is given in Table 4.1.

Viewers use stereotypes to define what annotation types they can visualise. For example, the syntax tree viewer can register itself as supporting annotations with structure type tree, or even more specifically, annotations of type SyntaxTree. Creation/editing of annotations also uses stereotypes to control what attributes and values can be entered. For instance, in the POSEntryDialog in Figure 1 the list of permissible categories is taken from the permissible values for the attribute `cat` as defined in the POS stereotype (see Table 4.1 for some example values)[2].

5 The Showcase Application

The publicly-available showcase application[3] currently demonstrates the time syncronisation and viewing of linguistic data provided by the integration of the two architectures. The application uses sound media annotated with utterances for each speaker[4]. The data is taken from the ESF corpus[5] and imported via Eudico into a GATE document with annotations. Afterwards some utterances were manually annotated with part-of-speech information using the GATE POS Viewer (see the figure above). The result is stored using GATE's persistence mechanism and is read every time the application runs.

The showcase application allows playback and viewing of media and associated linguistic data. It demonstrates the synchronysed operation of GATE's linguistic viewers embedded into Eudico's media and annotation viewers. The screen shot below shows two types of Eudico viewers used - subtitle and tag list viewer[6]. The subtitle viewer (the window with liela11p.wav caption) shows the current utterance for each speaker (in this case, INN and SLA) and controls the media playback. The other two Eudico viewers (the windows with GATE List View captions) are tag list viewers, one for each speaker, and integrate many instances of the GATE's POS viewer. Each instance displays a particular utterance and all POS annotations related to it.

The embedding Eudico viewers take care of layout, scrolling, update and time synchronisation of the embedded GATE components. They obtain the relevant time data from each GATE POS viewer's annotation data and register it with Eudico's time synchronisation mechanism. They also receive all time events and direct them to the appropriate GATE viewer which then highlights the word which is currently played as a sound by the media player.

In the example screen shot in Figure 3, the current word is **day**. A few milliseconds later, another time event would cause the highlight to be moved to the word **and**, which is the next word to be spoken. When no POS data is available (as for the word **this**), the last word remains highlighted until the next time-aligned annotation becomes current or the end of the utterance is reached. The Eudico viewers also take care of displaying a moving red bar in front of the currently playing utterance.

In addition to this synchronous playing behaviour, time synchronisation is exploited in some other ways: it is possible to select an utterance in one of the tag list viewers. This (time) selection is then reflected in each of the other viewers (by means of a blue bar in front of the overlapping utterances). The media time is automatically reset to the begin time of the selection. Manipulating the media time by dragging the media player's slider is reflected by the red bar and text highlight in the other viewers.

6 Conclusion

This paper described the integration of two infrastructures (Eudico and GATE) which were developed independently of each other; for different me-

[1] The demos require a browser (e.g., Netscape, Internet Explorer) with enabled Java and Java applets.

[2] Eudico's Abstract Corpus Model includes comparable concepts, but no effort to integrate at this level was made in the scope of the pilot project.

[3] Available for download under demos at `http://www.gate.ac.uk`.

[4] Video data could have been used just as easily, but at the cost of having the user install the Java Media Framework (JMF) on her system

[5] http://www.mpi.nl/world/tg/lapp/esf/esf.html

[6] For examples and a discussion of all Eudico viewers see http://www.mpi.nl/world/tg/lapp/eudico/eudico.html

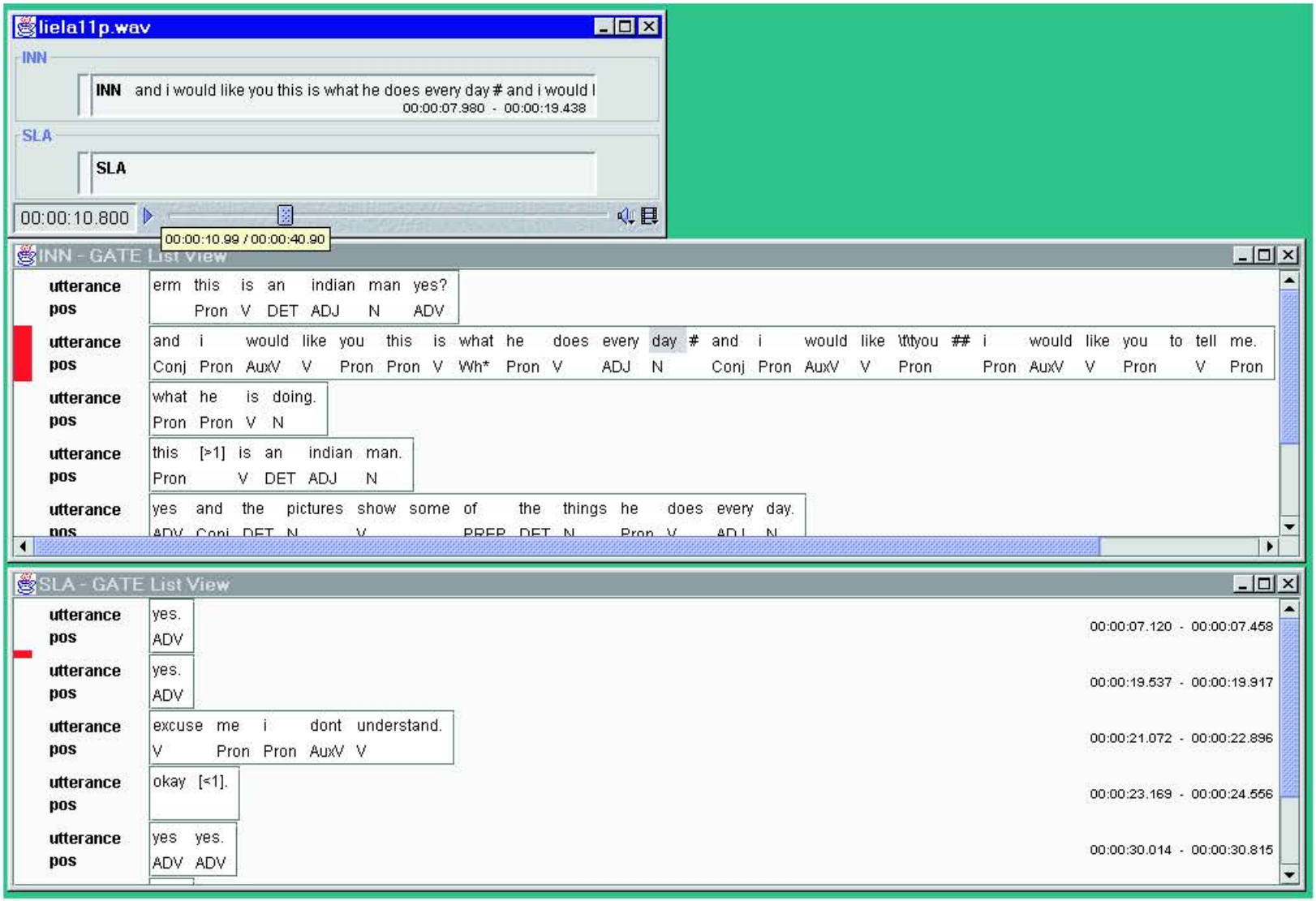

Figure 3: A screen shot of the demo

annotationType	POS		
AnnotationStructureType	Single span		
AttributeStereotypes	cat	String	det, n, adj, v, prep, conj, aux
	Time	Long	

Table 1: Stereotype for POS annotations

dia (video/speech vs. text) and applications. Such integration was needed in order to have an application where the end users (linguists and language engineers) who annotate video and speech corpora with textual transcriptions in Eudico, can also benefit from language processing tools and viewers from GATE.

The experiment lead to gaining an in-depth understanding of the functionality and operation of each of the two frameworks in isolation, and the benefits of their combined use. The showcase application exemplifies how the strengths of each framework have been combined to achieve seamless integration between speech, transcribed utterances, and linguistic data. The novel aspect of this integration is the resulting close inter-operation of the two infrastructures, which allows bi-directional data exchange and embedding of GUI components. Technically this is much more difficult to achieve than the usual case where one system uses the other through wrapper code. The application also proved the feasibility of the data- and GUI-reuse emphasis in GATE, as well as the extendibility of Eudico's dynamic viewers.

The inter-operation was made possible by the openness and flexibility of the underlying data models. Eudico's abstract corpus model showed its generality by the straightforward implementation of GATE/TIPSTER support. In turn, the generality of the GATE/TIPSTER model allowed efficient encoding and manipulation of transcribed speech data and the associated media time information.

From end-user perspective, the Eudico/GATE integrated application has introduced language engineering to the domain of spoken language research. In this way, linguists collecting and annotating speech/video corpora can also encode, manipulate and view linguistic data. From GATE programming perspective, the application highlighted the need for supporting different media and distributed processing. GATE version 2 which is currently under development, is aiming to address these issues.

References

H. Brugman, A. Russel, P. Wittenburg, and R. Piepenbrock. 1998. Corpus-based Research using the Internet. In *Workshop on Distributing and Accessing Linguistic Resources*, pages 8–15, Granada. http://www.dcs.shef.ac.uk/~hamish/dalr/.

H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. 1997. Software Infrastructure for Natural Language Processing. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97)*, March. http://xxx.lanl.gov/abs/cs.CL/9702005.

H. Cunningham, R.G. Gaizauskas, K. Humphreys, and Y. Wilks. 1999. Experience with a Language Engineering Architecture: Three Years of GATE. In *Proceedings of the AISB'99 Workshop on Reference Architectures and Data Standards for NLP*, Edinburgh, U.K., April. The Society for the Study of Artificial Intelligence and Simulation of Behaviour.

R. Grishman. 1996. The TIPSTER Text Phase II Architecture Design. Document Version 2.2. Technical report, Department of Computer Science, New York University, September. http://www.cs.nyu.edu/pub/nlp/tipster/152.ps .

B. MacWhinney. 1999. *The CHILDES Project: Tools for Analysing Talk (second ed.)*. Lawrence Arlbaum, Hillsdale, N.J.

A Modular Toolkit for Machine Translation Based on Layered Charts

Jan W. Amtrup and Rémi Zajac
Computing Research Lab, New Mexico State University
{jamtrup,zajac}@crl.nmsu.edu

Abstract

We present a freely available toolkit for building machine translation systems for a large variety of languages. The toolkit uses standard linguistic data representation based on charts and typed feature structures; A modular open architecture based on standardized interfaces and processing architecture, enabling the addition of external language processing components and the configuration of new applications (plug-and-play); An open library of basic parameterizable language processing components including a morphological finite-state processor, dictionary components, an island chart parser, chart generator, and chart-based transfer engine (for MT systems). It is open-source: the C++ source code is available, and portable: targeted systems are Unix and Windows systems.

1 Introduction

The MEAT[1] machine translation toolkit was developed in order to significantly shorten the development cycle for machine translation prototypes. In addition, systems developed using the toolkit should be robust and their performance (both qualitative and quantitative) should be predictable. Finally, basic components should be easily reconfigured or modified to adapt to new applications or languages.

The toolkit is geared towards multilingual processing and offers a well-founded uniform representation of all processing steps. Based on modern computational linguistic concepts, the aim is to incorporate best practice in language engineering.

The toolkit uses throughout a standard linguistic data representation based on charts to represent processing results and typed feature structures to represent linguistic structures. The toolkit is based on a modular open architecture that uses standardized interfaces for processing components and a single simple processing architecture. The architecture enables the addition of external NLP processing components and the configuration of new applications (plug-and-play).

[1]Multilingual Architecture for Advanced Translation

The system includes an open library of basic parameterizable NLP components that include a morphological finite-state processor, dictionary components, an island chart parser, a generator, and a transfer component. Complex components such as the parser or the morphological analyzer are parameterized by using high-level declarative languages for the linguist. The system has been implemented in C++ and the source code is available. The system is portable and currently exists in a Unix version and a Windows version.

2 Representation

The architecture is derived from previous work on NLP architectures within the Tipster framework (Zajac et al., 1997; Zajac, 1998; Steven Bird, 1999) and combines ideas from early modular NLP systems such as Q-systems (Colmerauer, 1971) and tree-transducers such as GRADE (Nakamura, 1984) or ROBRA (Vauquois and Boitet, 1985), which provide the linguist which very flexible ways of decomposing a complex system into small building blocks which can be developed, tested and executed one by one. It uses a uniform central data structure which is shared by all components of the system, much like in blackboard systems (Boitet and Seligman, 1994), and incorporating ideas on chart-based NLP (Kay, 1973; Kay, 1996; Amtrup, 1995; Amtrup, 1997; Amtrup and Weber, 1998; Amtrup, 1999; Zajac et al., 1999). All linguistic structures are encoded as Typed Features Structure and the association of linguistic structures to the text is maintained through the use of a Chart. The Chart itself is the main processing data structure.

2.1 Typed Feature Structures

A declarative, efficient and theoretically well-founded formalism to describe linguistic objects is an essential ingredient in any natural language processing system. A uniform data structure that is used by all components of a system offers several advantages over the use of multiple description systems. In particular, it simplifies enormously communication between NLP modules. All linguistic information in the system is encoded using Typed Feature Struc-

tures which is a versatile standard for representing linguistic structures. Typed Feature Structures are an extension of the traditional notion of linguistic features (Kay, 1979; Aït-Kaci, 1986; Pollard and Sag, 1987) and are used in all modern computational linguistic frameworks (LFG, HPSG, etc.). The TFS formalism also unifies object-oriented concepts and theorem proving techniques. TFSs are declarative with a sound logical semantics; they are associated to a small set of logical operators and can benefit of efficient implementations.

In the toolkit, the Typed Feature Structure system uses a version where types define their appropriate features (and type of their values), see e.g. Carpenter (1992). To improve the runtime behavior of the system, no complex constraints are associated to types as for example in the formalism presented in Zajac (1992). Feature structures provide a simple, versatile and uniform way of describing linguistic objects while a type system with appropriateness ensures the validity of feature descriptions and increases efficiency. Descriptions of words, syntactic structures, as well as rules for the various components can uniformly be coded as feature structures. The use of types enforces a type discipline for linguistic data: all legal linguistic structures are specified as a set of type definitions. Therefore, one of the initial task of the linguist building a system using the toolkit is to build and inventory of kinds of linguistic structures built during processing and formalize this inventory as a set of types and type definitions. The type definitions will then be used (1) by various compilers to compile (and type-check) linguistic resources such as dictionaries or grammar rules, and (2) at run-time by the various components accessing and manipulating feature structures to ensure that all feature structures created in the system are legal (i.e., conform to the type definitions). The type definitions themselves are compiled and the binary file is used as a runtime parameter by the TFS C++ library.

The formalism we developed uses a consecutive memory model for feature structures. Feature structures are stored as arrays of memory words rather than having a representation relying on the use of pointers. This is mainly done to reduce the processing needed for input/output operations and also targets at the distributed employment of a formalism. Similar representations are used for implementations of formalisms oriented towards abstract machine operations (Carpenter and Qu, 1995; Wintner and Francez, 1995). The formalism itself is implemented as a set of C++ classes representing types and feature structures. Apart from the usual operations for feature structures (subsumption and unification), the system also provides an API to destructively manipulate feature structures, a property that has to be used with care, but is extremely useful at times. The efficiency of the implementation is satisfactory and currently, we reach 4500 unifications per second in a translation application.

```
MainVerb[
 exp : "sufrir",
 infl : InflMorph[
        number : Singular,
        tense : Past,
        gender:Masculine,
        mood : Participle],
 lex : LexMorph[
        subcat : Transitive],
 trans : <:
        LSign[exp : "suffer",
              lex : LexMorph[
              regular : True]]:>]
```

2.2 Charts

Charts are a standard for representing sets of embedded linguistic structures (Hockett, 1958; Kay, 1973). They are also a versatile computational data structure for parsing (text and speech), generation and transfer (MT). A chart represents partial results independently of processing strategies and processing peculiarities (Kay, 1973; Sheil, 1976; Haruno et al., 1993; Kay, 1999). Formally a directed, acyclic, rooted graph, a chart can be viewed as a generalization of a *well-formed substring table*, capable not only of representing complete constituents ('*inactive edges*'), but also storing partial results ('*active edges*') (Sheil, 1976). The basic functions that operate on a chart are very simple. Since chart-based algorithms are almost always designed to be monotonic[2], a chart parser for example uses two main rules to add edges to the chart:

- The *Hypothesize* rule takes an edge of the chart and consults a grammar to propose new promising hypotheses that should be pursued;
- The *Combination* rule takes two edges, one of them active, the other inactive, and tries to combine them. If this combination succeeds, a new edge is created and eventually inserted into the chart.

The main advantage of formulating a natural language processing task as a chart-based process is the division of describing *what* has to be computed from *how* the individual operations have to be carried out. Kay (1980) calls the specification of a task that does not specify search and processing strategies an *algorithm schema*. In practice, one can experiment with various dimensions of strategies, e.g. top down vs. bottom up, left-to-right vs. right-to-left vs.

[2]See Wirén (1992) for a notable exception.

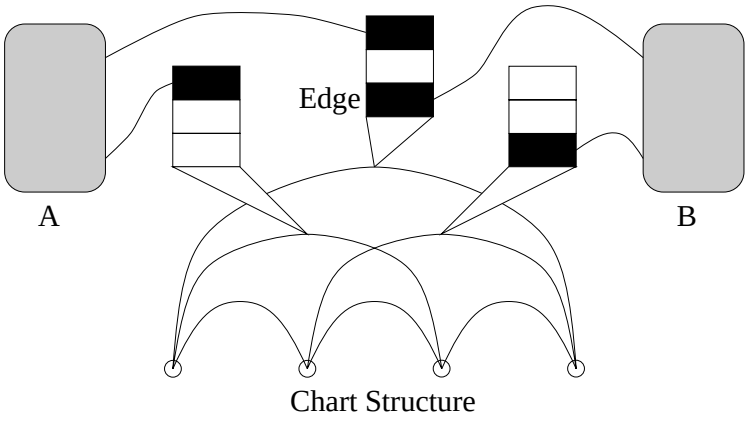

Figure 1: A layered chart.

mixed strategies or depth first search vs. breadth first search.

In our system, charts are *layered*. A layered chart is modular and declarative representation of the data manipulated by multiple processes. The traditional chart structure, which stores linguistic information on edges and where nodes represent a time-point in the input stream, are augmented, following Tipster ideas on 'annotations', with tags which define the kind of content an edge bears, and with spans (pairs of integers) pointing to a segment of the input stream covered by the edge. Spans are used for example in debugging and displaying a chart with edges positioned relative to the input text they cover. Tags identify for example edges built by a tokenizer, morphological analyzer, or a syntactic parser, and define sub-graphs of the whole chart that are input to some component. The chart is implemented as a C++ class which provides a set of methods to traverse the graph and manipulate edges and their content.

By attaching *tags* to edges that define what kind of content an edge bears, charts can be used to store information for more than one component. In this layered chart[3] each component sees only the fraction of information that it needs to operate on. Therefore, the content of the chart gives a precise view of the current state of operations within the system, and interfaces between two modules become extremely easy to implement, as the exchange of information rests on a common concept, that of a chart edge.

At runtime, the chart is kept in memory and the various components of an application work on the same chart. Each component processes only a subset of layers, typically only two: the input layer and the output layer. For example, a parser will look at morphological edges and produce syntactic edges.

[3]The type of chart we use here is a weaker version of the layered charts defined in Amtrup and Weber (1998), as we do not distribute the chart, and we don't use parallel processing on the component side.

The chart is actually implemented as a lattice (directed rooted acyclic graph) where nodes can be time-aligned but where two time-aligned nodes are not necessarily identical. In the general case, nodes are partially ordered (with respect to time), and not completely ordered as in the traditional chart. This enables, the implementation of processes that create a sequence of edges covering a single input edge:[4]

- Normalization of contraction and elision phenomena: English contraction *don't* expanded as *do not*, or French elision *du* expanded as *de le* for example.
- POS disambiguation: sequences such as French *la porte* are ambiguous between determiner/pronoun and noun/verb. A disambiguation process would eliminate the incorrect sequences determiner+verb and pronoun+noun, leaving only two valid sequences determiner+noun and pronoun+verb, creating 2 additional distinct intermediary nodes in between the two words *la porte*.
- Chart generation, where an input edge results in a sequence of sub-edges covered by the input edge, but unrelated to other edges in the graph.

3 Architecture

The toolkit is architectured around the following three notions:

1. A module performs a complete elementary processing step.
2. An application is defined as a sequence of modules.
3. A module is an instance of a component from the component library.

[4]This also allows to represent directly the output of a speech recognizer, a word lattice.

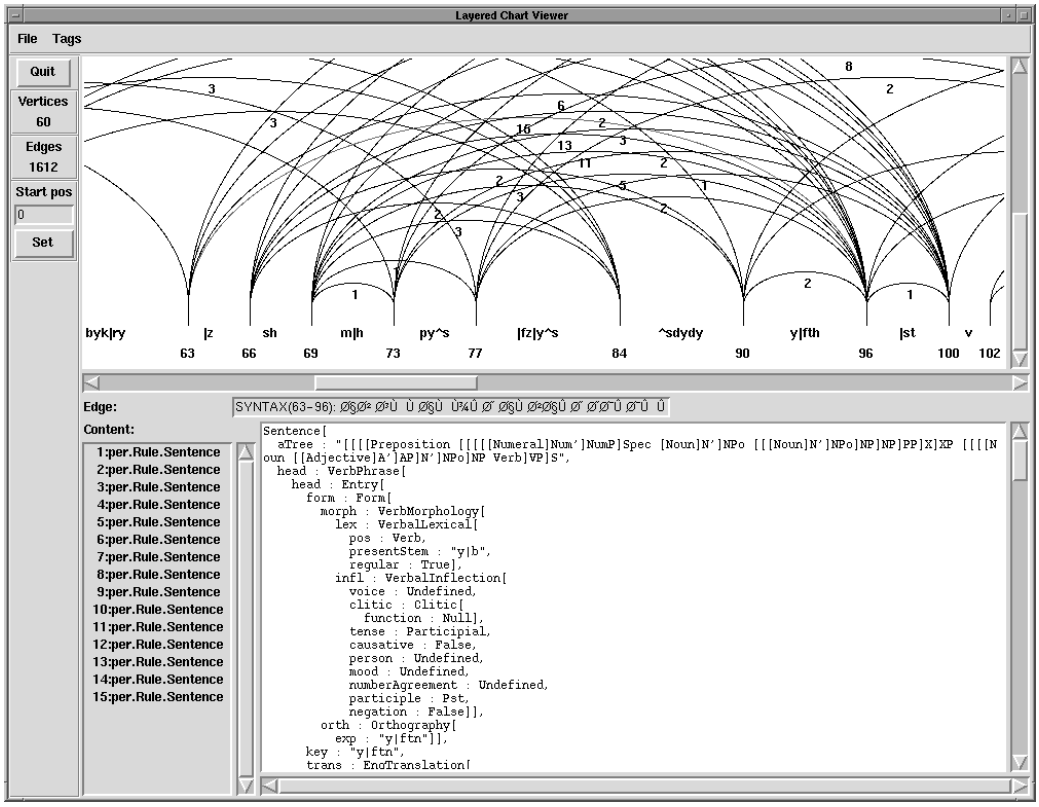

Figure 2: Viewing a complex analysis with the Chart Viewer.

3.1 Applications

An application is basically a sequence of modules. The standard I/O for a module is a layered chart, which is a global parameter for an application (special modules can also deal with files). Another global parameter for an application is the set of type definitions specifying the set of legal linguistic structures that can be stored on chart edges.

A working system (an application) can easily be assembled from a set of components by writing a resource file, called an application definition file. This file uses a simple scripting language to describe instantiations of components, the calling sequence of components and various global variables and parameters. Assembling components together is done using a composition operator which behaves much like the Unix pipe. When, in a Unix command, data between programs is transmitted using files (stdin/stdout) and programs are combined using the pipe '|' command, in M, the data transmitted between components is a chart, and syntactically the sequence of component is combined using the ':' composition operator. In effect, the MEAT system is a specialized shell for building NLP systems. The implementation language is C++, but external components can be integrated in the system by writing wrappers (as done for several morphological analyzers previously built or used at CRL).

An application definition file consists of three sections. (1) variable definitions reduce typing and enhance the transparency of application definition files. (2) Application definitions specify the sequence of modules that compose a given application. (3) Module definitions define named building blocks for applications and the parameters that they receive during a system run.

Variables provide symbolic names for long path names and make it easy to switch configuration values that pertain to several modules. Once defined, the variable name can be used instead of its value throughout the application definition file. We support both variables defined in the application definition as well as environment variables. Variable definitions in an application definition file can refer to other variables for their values. Typically, this is used in contexts like this:

```
$ROOT = /home/user/M
$SYNGRAM = $ROOT/per/SynGram.cbolero
$MORPHGRAM = $ROOT/per/Morph.samba
```

Aside from variables that are defined inside an application definition file and environment variables, we also support command line variables which are passed to the application.

Applications are defined as sequences of Modules that have to be executed to achieve a certain task. Each application is defined by its name together with the names of modules that have to be processed in turn:

```
application lookup =
  Tok($ifile=$1):Morph:Dictionary:ChartSaver
```

This application would first perform tokenization. The variable equation in the definition for the tokenizer specifies that the variable `$ifile` is set to the value of the first command line parameter. Any reference to that variable for this particular execution of the Tokenizer module would use the value given by the user on the command line. This binding is strictly local to the module for which it is defined. After Tokenization, a number of other modules are executed, including the morphological analyzer that we described earlier.

Currently, we restrict the model of operation to a sequence of modules without alternatives. We do not support graphs of modules as a model for an application. Thus, we do not support multi-threading or otherwise concurrently executed modules. Applications can be executed using a shell command or through a graphical interface (see below).

3.2 Modules

Conceptually, a module performs a single linguistic task on the data currently present in the chart. Thus, a module would take some edges of the chart as input data and provide new edges as output. In some cases, however, a module may be executed for its side effects. For instance, an input component might read a file and produce edges.

A sample module definition (performing morphological analysis of Persian text) looks like this:

```
module MorphAnalyzer {
  class = MorphAnalyzer
  grammar = $RES/morph.samba
  rule = Morphology
  type = chart
  sourceTag = TOKEN
  targetTag = MATOKEN
}
```

A module is an instance of a component from the MEAT component library (a set of C++ classes). Every module definition must specify at least one parameter, the name of the component (C++ class) of which the module is an instance (parameter `class`). By parameterizing the class representing the module within the main program, the same component can be used several times within one application. For instance, there could be several parsers within one application.

Additional parameters can be provided according to the specifications of the module in question. In the example above, the morphological grammar and initial rule need to be specified, as well as the tags that define the input and output sub-graphs of the chart.

Parameters can also be defined as global and used outside the scope of a module definition. In this case, they are global and inherited by all modules of an application. However, the local definition of a parameter overrides the global behavior. Thus, if one would define `verbose = false` on the global level, and define it as being `true` for only a subset of the components, then only those components would issue logging messages.

In the current implementation, all modules are linked in the main executable at compile time and the model does not support distributed processing. Although we have experimented with distributed architectures (Zajac et al., 1997) in the past, the overhead can be significant and the architecture must be carefully designed to support the needs for distributed components while minimizing overhead. In particular, a distributed architecture can be designed to support either collaborative research (with remote execution of components) or parallel processing on the same text. The requirements are fairly different and could be difficult to reconciliate within a single model.

3.3 Library

The toolkit includes libraries of approximately 30 processing components. Most of these components perform simple tasks and are parameterized directly in the application file. Some more complex component have external parameters such as a unification-based grammar or a dictionary file. The core library include components that have a general use:

- Utilities: Unicode tokenizer; Store/Load charts (for debugging)
- Dictionary: compiler; indexer; lookup (single words, compounds).
- Morphology: wrappers; parameterized analyzer/generator.
- Parsing: modular bi-directional island parser.
- Generation: linearizer.
- Transfer: lexical transfer; morphological feature transfer.

Each component is a C++ class that implements a pre-defined interface. The core library can easily be extended by creating a new class in the user library and linking to the other libraries at compile time. User-defined components can be used in applications as if there were native components. At

runtime, the MEAT interpreter instantiates modules defined in the application file by creating an instance of the corresponding C++ class with the appropriate parameters as specified in the module definition (in particular, an obligatory parameter is the set of types definitions defining legal feature structures). The module is executed by calling the `run()` method (which is implemented as part of the component interface).

3.4 External Components

It is possible to integrate external software modules via special components that act as wrappers. For example, the current implementation includes a morphological wrapper component that reads a file of tokens in a standard format to build a chart which is then used for further processing. This wrapper has been used to integrate several morphological analyzers (Prolog for a Spanish morphological analyzer, Lisp for a Russian one, Java for a Serbo-Croatian, and C for a Japanese and Korean).

We are currently working on extending this mechanism to provide a more general wrapping mechanism that can work on any kind of input chart, and not only a linear sequence of (possibly ambiguous) tokens. Note that it is also relatively easy to develop C++ components that implement wrappers communicating with some software module with its API if available.

4 Linguistic Knowledge

All linguistic knowledge used by the components of the core library (morphological analyzer and generator, parser, generator, dictionary lookup, transfer) is defined in external resource files that parameterize the runtime components. For example, a unification grammar used by the parser component is stored in a text file that contains the set of rules for that grammar. During the initialization phase at runtime, an instance of the parser component reads the file containing the rules that will drive the parsing algorithm. Since both input and output of the parser are charts, it is possible to to create several parser instances with different grammars and apply them in sequence on a chart (Zajac and Amtrup, 2000).

A rule is specified in the feature structure notation and each syntactic element follows the general feature structure syntax. Although this makes it sometimes a little bit awkward, it allows to compile rules as feature structures which are themselves compiled as compact arrays of integers[5] and enable very fast access of the rule at runtime (see for example (Wintner, 1997)).

[5] The unification algorithm operates on this data structure. Arrays of integers can also be written or loaded from a file very efficiently.

```
NounBarNoEzafe = per.Rule[
 lhs: per.NounBar[
      head: #head,
      boundary: per.NPtrue],
 rhs: <:
      #head=per.NounOrNounCompound[
            infl:
              [ezafe: per.EzFalse,
               indefEncl: False,
               clitic.function: per.Null]]
      :>
 island: #head
 ];
```

Most of the language resource files are compiled before runtime and components load compact binary files instead of text source files. The toolkit provides compilers for the various formalisms and for dictionaries; dictionaries are compiled as one data file and one or more index files (tries). Since all resources files use typed feature structures as the basic representation formalism, all resource files include a set of type definitions which is used by the compilers to create binary instances of feature structures (linear arrays of integers), and by runtime components to create in memory instances (using the same array layout) or to print feature structure in a text format. The type definitions themselves are stored in a separate text file which must be itself compiled before compilation of any other resource file. The type definition file specifies the set of types used in a given application (type definitions are global to an application and are an obligory parameter to each of the components). A type definition defines super-types or sub-types (inheritance hierarchy), and the set of appropriate features for that type (and the types of their values), not but complex type constraints as in (Zajac, 1992) for example.

5 Development Environment

The development environment consist currently of two tools: the Chart Viewer and the application Runner. The chart built by some application can be saved in a file at any point during processing for further inspection. The chart can then be displayed using the Chart Viewer which allows the selective display of chart layers (by tags), and the selective display of feature structures.

The MEAT Application Runner can be run from the command line by passing to the MEAT interpreter the application file, the name of the application to be executed and the application parameters:

```
% meat -v app lookup test/doc1.txt
```

There is also a graphical tool that allows to execute applications defined in an application file using a graphical interface. This tool basically provides a

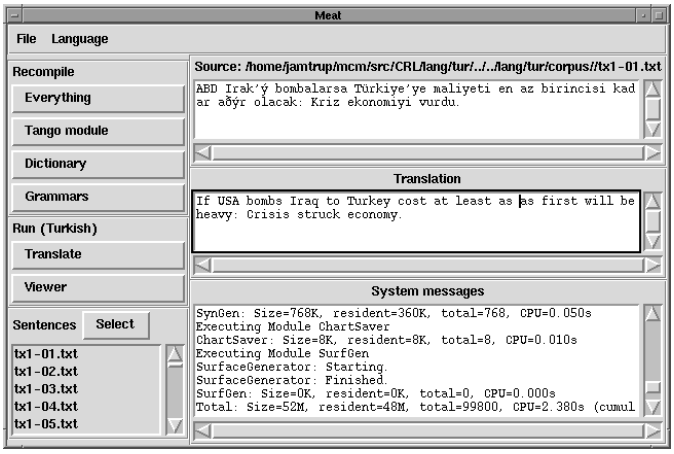

Figure 3: Executing applications from the Runner.

graphical view of the application file and allows execution of applications. (NB: this tool is still under development).

6 Conclusion

This toolkit has been used to develop a Persian-English MT system; to port previously developed glossary-based MT systems and to develop a Turkish-English MT system; and as the Machine Translation infrastructure of an elicitation-based MT system. The architecture and the core library is also used in new projects on multilingual information extraction and multilingual question-answering systems. Documentation, sources and binaries (Unix and Windows) available at `http://crl.nmsu.edu/meat`.

The toolkit is still under development as new components are added to the core library and previous components are enhanced or corrected. In the near future, we plan to enhance the library with new components for machine translation, including better transfer and generation components.

Acknowledgments

The MEAT system has been implemented by Jan Amtrup and Mike Freider with contributions from many other people at CRL.

This research has been funded in part by DoD, Maryland Procurement Office, MDA904-96-C-1040.

References

Hassan Aït-Kaci. 1986. An Algebraic Semantics Approach to the Effective Resolution of Type Equations. *Theoretical Computer Science*, 54:293–351.

Jan W. Amtrup and Volker Weber. 1998. Time Mapping with Hypergraphs. In *Proc. of the 17 th COLING*, Montreal, Canada.

Jan W. Amtrup. 1995. Chart-based Incremental Transfer in Machine Translation. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation, TMI '95*, pages 188–195, Leuven, Belgium, July.

Jan W. Amtrup. 1997. Layered Charts for Speech Translation. In *Proceedings of the Seventh International Conference on Theoretical and Methodological Issues in Machine Translation, TMI '97*, Santa Fe, NM, July.

Jan W. Amtrup. 1999. *Incremental Speech Translation.* Number 1735 in Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New York.

Christian Boitet and Mark Seligman. 1994. The "Whiteboard" Architecture: A Way to Integrate Heterogeneous Components of NLP systems. In *COLING-94: The 15th International Conference on Computational Linguistics*, Kyoto, Japan.

Bob Carpenter and Yan Qu. 1995. An Abstract Machine for Attribute-Value Logics. In *Proceedings of the 4th International Workshop on Parsing Technologies (IWPT95)*, pages 59–70, Prague. Charles University.

Bob Carpenter. 1992. *The Logic of Typed Feature Structures.* Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

Alain Colmerauer. 1971. Les systemes-q: un formalisme pour analyser et synthetiser des phrases sur ordinateur. Technical report, Groupe TAUM, Universite de Montreal.

Masahiko Haruno, Yasuharu Den, Yuji Mastumoto, and Makato Nagao. 1993. Bidirectional chart generation of natural language texts. In *Proc. of AAAI-93*, pages 350–356.

C. F. Hockett. 1958. *A course in modern linguistics.* Macmillan, New-York.

Martin Kay. 1973. The MIND System. In R. Rustin, editor, *Natural Language Processing*, pages 155–188. Algorithmic Press, New York.

Martin Kay. 1979. Functional grammar. In C. Chiarelloet *et al.*, editor, *Proc. 5th Annual Meeting of the Berekeley Linguistic Society*, pages 142–158, Berkeley, CA.

Martin Kay. 1980. Algorithmic Schemata and Data Structures in Syntactic Processing. Technical Report CSL-80-12, Xerox Palo Alto Research Center, Palo Alto, CA.

Martin Kay. 1996. Chart generation. In *Proc. of the 34 nd ACL*, pages 200–204, Santa Cruz, CA, June.

Martin Kay. 1999. Chart Translation. In *Machine Translation Summit VII*, pages 9–14, Singapore.

Makoto Nagao Nakamura, J. Juni-Ichi Tsujii. 1984. Grammar Writing System (GRADE) of Mu-Machine Translation Projects and its Characteristics. In *Proc. of the 10 th COLING*, Stanford, CA.

Carl Pollard and Ivan A. Sag. 1987. *Information-based Syntax and Semantics. Vol 1: Fundamentals.* CSLI Lecture Notes 13, Stanford, CA.

B. A. Sheil. 1976. Observations on Context-Free Parsing. *Statistical Methods in Linguistics*, 6:71–109.

Mark Liberman Steven Bird. 1999. A Formal Framework for Linguistic Annotation. Technical Report MS-CIS-99-01, Dept of Computer and Information Science, University of Pennsylvania.

Bernard Vauquois and Christian Boitet. 1985. Automated Translation at Grenoble University . *Computational Linguistics*, 11(1):28–36.

Shuly Wintner and Nissim Francez. 1995. Abstract Machine for Typed Feature Structures. In *Proceedings of the 5th Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Spain.

Shuly Wintner. 1997. *An Abstract Machine for Unification Grammars.* Ph.D. thesis, Technion - Israel Institute of Technology, Haifa, Israel, January.

Mats Wirén. 1992. *Studies in Incremental Natural-Language Analysis.* Ph.D. thesis, Linköping University, Linköping, Sweden.

Rémi Zajac and Jan W. Amtrup. 2000. Modular Unification-Based Parsers. In *Proc. Sixth International Workshop on Parsing Technologies*, trento, Italy, February.

Rémi Zajac, Marc Casper, and Nigel Sharples. 1997. An Open Distributed Architecture for Reuse and Integration of Heterogeneous NLP Components. In *Proc. of the 5 th Conference on Applied Natural Language Processing*, Washington, D.C.

Rémi Zajac, Malek Boualem, and Jan W. Amtrup. 1999. Specification and Implementation of Input Methods Using Finite-State Transducers. In *Fourteenth International Unicode Conference*, Boston, MA, March.

Rémi Zajac. 1992. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182.

Rémi Zajac. 1998. Annotation Management for Large-Scale NLP. In *ESSLLI-98 Workshop on Recent Advances in Corpus Annotation*, Saarbruken, Germany.

Finite State Tools for Natural Language Processing

Jan Daciuk
Alfa Informatica, Rijksuniversiteit Groningen
Oude Kijk in 't Jatstraat 26, Postbus 716
9700 AS Groningen, the Netherlands
e-mail: *j.daciuk@let.rug.nl*

Abstract

We describe a set of tools using deterministic, acyclic, finite-state automata for natural language processing applications. The core of the tool set consists of two programs constructing finite-state automata (using two different, but related algorithms). Other programs from the set interpret the contents of those automata. Preprocessing scripts and user interfaces complete the set. The tools are available for research purposes in source form in the Internet.

1 Introduction

Finite-state automata (both acceptors and transducers) play increasingly important role in natural language processing. Their main advantages are their small size as compared with the data they hold (see e.g. (Kowaltowski et al., 1993)), and the very fast lookup of strings in an automaton – proportional to the length of the string.

Deterministic, acyclic, finite-state automata (DAFSA) are used in a variety of applications, including DNA sequencing, computer virus detection, and VLSA design. In natural language processing, they are used for tasks like spelling correction, restoration of diacritics, morphological analysis, perfect hashing, and acquisition of morphological descriptions for morphological dictionaries. DAFSA hold a finite set of strings of finite length, so they can be perceived as a kind of dictionaries. Depending on the application, the contents of an automaton may differ considerably, and so do programs that interpret it. However, the basic data structure remains the same. And so do the programs that produce automata. It is relatively easy to import data from other systems, as the basic unit in the system is just a string.

2 System Architecture

The architecture of the system is shown on figure 1. The key data structure in the system is a string of characters. The core of the system consists of two programs for construction of DAFSA. They both produce the same results, but they have different memory requirements and run at different speeds. The algorithms are taken from (Daciuk et al., 1998) (newer version has just appeared in (Daciuk et al., 2000)). The input data for both of them is a set of strings. It may be prepared using a variety of preprocessing scripts. The output of the programs is a DAFSA interpreted by other application programs. The first construction program – `fsa_build` – constructs an automaton from a lexicographically sorted list of strings. It is very fast, and it needs very little memory (see (Daciuk et al., 2000)). The other construction program – `fsa_ubuild` – constructs an automaton from a set of strings in arbitrary order. Its speed is much lower, and it may need much more memory (depending on the order of strings). It can be used in situations where we are short of disk space for sorting, and we have much core memory. Both programs accept various run-time options. They can also use two modules: one for adding information necessary for perfect hashing, the other one for producing guessing automata. The modules are switched on by run-time options.

Different kinds of applications require different information to be stored in automata. The following sections describe that in detail.

The application programs use a command line interface, but an emacs interface for GNU emacs 19 is also available for tasks like spelling correction or restoration of diacritics. Recently, a Tcl/Tk interface has been added for a task of acquisition of descriptions for a morphological dictionary.

The automata are represented as vectors of transitions. States are represented only implicitly. Various compression methods are provided as compile time options. Their influence on the speed of interpretation is small. However, some of them may significantly lengthen the construction time. By using combinations of compile options one can obtain automata that differ in size by about 40%. It is also possible to use language-specific features, like coding of prefixes and infixes, to get more compression.

A software package containing the system consists of 9 programs, 3 shell scripts, 11 awk scripts, 12 perl scripts, one emacs lisp module, and one Tcl script. The documentation consists of 11 man pages, on-

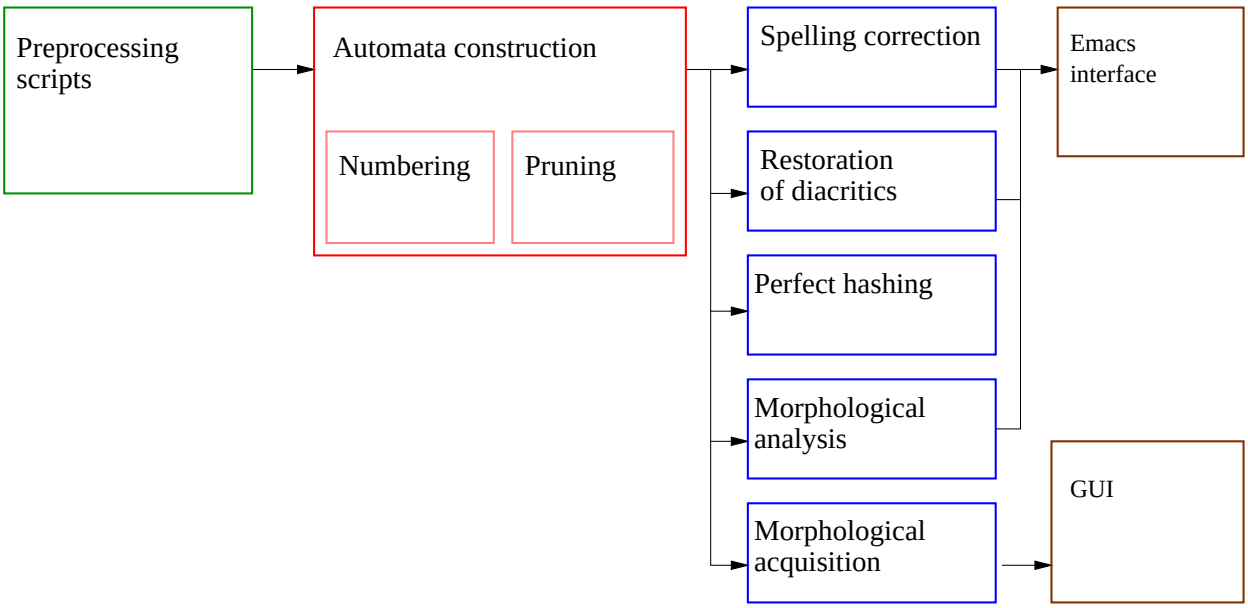

Figure 1: System architecture

line help file for a Tcl/Tk interface for morphological data acquisition, and 3 additional text files.

3 Spelling Correction and Restoration of Diacritics

Crude spelling correction requires only a word list. Such a list can be obtained from various sources. The system does not provide any scripts for that, as the sources may differ widely, and so do the methods of getting the words. However, a conversion from mmorph format to a 3-column format used by tools from the University of Aix-en-Provence is provided. The first column in that format is the inflected word form. Mmorph (see (Petitpierre and Russell, 1995)) is Multext morphology tool from ISSCO, Geneva.

The spelling correction tool uses an algorithm by Kemal Oflazer ((Oflazer and Güzey, 1994), (Oflazer, 1996)). Restoration of diacritics is implemented as a simple search with relaxed comparison. An emacs 19 interface can be used to correct words from within that editor. The interface is based on ispell.el and offers similar options. It is relatively easy to develop interface for other programs, as the program reads standard input and produces results on standard output.

4 Perfect Hashing

Perfect hashing (see (Lucchiesi and Kowaltowski, 1993), (Roche, 1995)), like spelling correction, also requires a list of words. However, the words in the automaton must be numbered. This is done by a special module in the programs that construct automata. The module stores additional information in the automaton structure. For each state, the number of different strings (including the empty string ϵ) recognized by a part of the automaton beginning in that state is stored. The order of words in the automaton (and thus the mapping between the words and their numbers) depends on various factors, e.g. various compression methods in use. Therefore, a program that lists the words in the dictionary in the order they are stored is provided.

The program that converts numbers to words and vice versa is a stand-alone tool, not a library. However, since it reads the standard input and produces results on the standard output, it can be used by other programs.

5 Morphological Analysis

Two kinds of morphological analysis are possible using the tool set. The first one is lexicon-based. The outcome is the canonical form, or the categories (features), or both of them. The strings stored in the automaton consist of two parts. One is the inflected word form to be analyzed, the other - the outcome of the analysis. They are separated with a special character – an annotation separator. This can be seen as an implementation of a p-subsequential transducer. The outcome of the analysis must be coded (see e.g. (Kowaltowski et al., 1998)), because otherwise the automaton would grow to enormous size. Basically, the coding is used to avoid storing the stem more than once in the same string. To help in constructing the automaton, several scripts are provided. There is one script for languages that have no flectional prefixes or infixes, a different one

for those that have only flectional prefixes and no infixes, and another one for languages that have both inflectional prefixes and infixes. The user must know which script to choose. It is also up to the user to choose appropriate run-time options of `fsa_morph` – the program that performs morphological analysis. However, the user does not need to separate the prefixes or infixes from the stems in the entries. It is done automatically by the scripts.

The morphological analysis program `fsa_morph` searches for the inflected form in the automaton, and then decodes and outputs the annotation, i.e. the outcome of the analysis. In the basic case, the canonical form is coded so that one letter says how many characters to strip from the end of the inflected form, and it is followed by the ending of the canonical form. In case of flectional prefixes, the code is supplemented by an additional letter that says how many characters are to be deleted from the beginning of the inflected form before turning it into the canonical form. The version that handles infixes as well has one more letter that says how far from the beginning of the word the characters to be deleted are.

It is also possible to analyze words not present in the dictionary. This is done by analyzing the endings, and sometimes the prefixes and infixes (e.g. in case of German). An automaton for approximate morphological analysis (a guessing automaton) associates endings, and sometimes prefixes and infixes as well, with appropriate outcomes of the analysis. But first, those associations need to be created. The system contains several scripts to aid in that process. They invert the inflected form, look for endings, prefixes and infixes, and code them appropriately. The association between an ending and the corresponding analysis is created by inverting the inflected form and appending the analysis as an annotation (similar to the lexicon-based analysis). If prefixes and infixes are present, they are moved from the inflected form to annotations. The coding of prefixes and infixes is very similar to that used by `fsa_morph`. However, the prefixes, and infixes when needed, must be specified in the string, so that not only the beginning, but also the end of the analyzed word can be compared to the strings stored in a guessing automaton. The resulting strings are data for a guessing automaton.

Automata are created in the usual way, and then a specialized module in automata creation programs prunes the structure. If from a given state all paths lead to the same set of annotations, then all states between that state and the annotations can be removed with all their transitions. This significantly reduces the size of the automaton. Further heuristics can be used to improve either recall or precision of the predictions made with such tool. During the analysis, the analyzed word is inverted, and the consecutive letters are looked up in the automaton. When no more letters can be recognized, all annotations reachable from the state where the recognition process stopped are decoded as the result of the analysis. The program that performs the approximative morphological analysis – `fsa_guess` – has options that turn on recognition of prefixes and infixes.

6 Acquisition of Data for Morphological Dictionary

Morphological dictionaries are usually constructed using morphology tools, e.g. two-level morphology. In many advanced tools, a lexeme description is a line containing the base form, categories (or features) including the flectional paradigm, and often the canonical form. It is possible to associate endings, prefixes and infixes with that sort of information in a similar manner to that used in approximate morphological analysis. So the same program – `fsa_guess` – that performs the approximate morphological analysis is also used (with an appropriate option) for guessing the morphological description of an inflected form. A user runs the program on a list of new words, and the results can be processed using a graphical user interface, where the user can select descriptions, compare them, and see what they produce.

This part of the system is still under development. The version available in the Internet does not contain the Tcl/Tk interface, and it has no scripts to help building data for guessing automata for morphological data acquisition. Although the system works well for French, efforts are under way to make it work for German. The main problem is the use of archiphonemes. If not treated properly, they can inflate the automaton, and in the process some generalizations might be lost as well.

7 Auxiliary Programs

In the system, there are two additional programs that perform auxiliary tasks. The first one – `fsa_prefix` – was briefly mention is section 4, page 2. It can be used for listing the contents of a dictionary (an automaton). However, this is a specific instance of a more general task, i.e. listing all words (or strings) in the automaton that have a specified prefix. In order to get the whole contents of the automaton one simply specifies a null string.

Another program – `fsa_visual` – produces data for a graph visualization software `vcg`. It can be used for didactic purposes, or for debugging on tiny data samples. Larger samples make the graphs too large to be readable.

8 Conclusions

We have presented a set of tools based on a simple observation, that DAFSA can be useful in variety of

natural language applications. The main data type is an automaton representing a set of strings. For the automata construction programs, the strings are just sequences of symbols or characters. This makes it easy to use data from other tools. The meaning is attributed to the strings by application programs that interpret them.

The tools are available in the Internet and can freely be used for research purposes. They can handle large data, e.g. they have been used to build a morphological dictionary of German with 3,977,448 inflected forms. It took 20 minutes on a pentium 350MHz computer. They are also very fast. For example, morphological analysis using the same German dictionary is 7.5 times faster than that done by mmorph. Depending on compile options, an automaton holding the German morphological dictionary can take approximately 0.5 MB.

A page describing the software package, with pointers to downloadable software and relevant information accessible through the Internet is available at:

`http://www.pg.gda.pl/~jandac/fsa.html`

The package contains source code in C++, man pages, and a few accompanying documentation files (README, CHANGES, and INSTALL). HTML versions of man pages are available either directly from the same page, or as a tar archive. Another (rather dated) software package using Mealy's automata (transducers) is also available from the same address. That package is no longer developed, as almost all its features are also available in the package described in this paper.

References

Jan Daciuk, Richard E. Watson, and Bruce W. Watson. 1998. Incremental construction of acyclic finite-state automata and transducers. In *Finite State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey, June – July.

Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. 2000. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April.

Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. 1993. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil.

Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. 1998. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, January.

Claudio Lucchiesi and Tomasz Kowaltowski. 1993. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan.

Kemal Oflazer and Cemalettin Güzey. 1994. Spelling correction in agglutinative languages. In *4th Conference on Applied Natural Language Processing*, pages 194–195, Stuttgart, Germany, October.

Kemal Oflazer. 1996. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, March.

Dominique Petitpierre and Graham Russell, 1995. *MMORPH - The Multext Morphology Program.* ISSCO, 54 route des Acacias, CH-1227, Carouge, Switzerland, version 2.3 edition, October.

Emmanuel Roche. 1995. Finite-state tools for language processing. In *ACL'95*. Association for Computational Linguistics. Tutorial.

The XML Framework and Its Implications for the Development of Natural Language Processing Tools

Nancy IDE
Department of Computer Science
Vassar College
Poughkeepsie, New York, USA 12604-0520
ide@cs.vassar.edu

Abstract

The eXtensible Markup Language (XML) (Bray, et al., 1998) is the emerging standard for data representation and exchange on the World Wide Web. The XML Framework includes very powerful mechanisms for accessing and manipulating XML documents that are likely to significantly impact the development of tools for processing natural language and annotated corpora.

Introduction

All language processing applications, including machine translation, information retrieval and extraction, text summarization, user/machine dialogue systems, and speech understanding and synthesis, manipulate language data represented in some electronic format. Some applications (e.g., machine translation, summarization, speech understanding) process streams of data more or less sequentially, while others (e.g., retrieval and extraction) rely more heavily on search and access over large bodies of data. In either case, processing exploits the markup in the data to assist in the analysis. For example, in textual data, markup for logical structure (e.g., section, paragraph, and sentence boundaries, etc.) provides essential information for any language processing task. In addition, markup identifying terms, foreign words, names, dates, etc. can be exploited for tasks such as machine translation and information retrieval, while identification of titles, footnotes, and other extra-textual matter can be used to limit the data to be searched. Because the data that will be analyzed by language processing applications in the future will consist largely of documents delivered over the World Wide Web, the markup format these applications process will be XML.

The language processing community also *creates* text and speech data for training statistical language processing algorithms. The cost of creating annotated data can be very high, both in direct financial terms and in terms of the cost of allocating skilled labor. So funders, whether public or commercial, have come to expect that the cost of resource creation will be amortized over multiple research and development efforts. Such reusability demands the use of standardized, non-proprietary encoding formats for data interchange and to enable easy human-readable display and access to data. For the applications we are now beginning to develop, these formats must support multi-lingual, multi-media, and multi-modal data, as well as linkage among them.

As an international standard, the eXtensible Markup Language (XML) (Bray, *et al.*, 1998) is the obvious basis for a standardized encoding format, and is or will be used by several language processing projects (e.g., LT XML[1], ATLAS[2], XCES[3], ANC[4]). At its most basic level

[1] McKelvie, Brew, and Thompson, 1998.

[2] Bird, *et al.*, 2000.

XML is a document markup language directly derived from SGML (i.e., allowing tagged text (elements), element nesting, and element references). However, various features and extensions of XML make it a far more powerful tool for data representation and access than SGML, including means for complex linkage within and between documents, easy data transformations using the XML Transformation Language (XSLT) (Clark, 1999), constraint and validation of markup using XML Schemas (Thompson, *et al.*, 2000; Biron & Malhotra, 2000), and display, manipulation, and search of data via the World Wide Web.

This paper provides an overview of the most important XML mechanisms and suggests how they may impact the design of language processing tools. The focus here is on the use of XML for the creation and annotation of text and speech data; however, we also consider some of the capabilities for search and retrieval from XML-encoded documents.

1 XML Links

The recommended practice in encoding annotated corpora is to maintain all or most annotations in separate documents, each of which references appropriate locations in the document containing the original data (Ide & Brew, 2000). This strategy yields, in essence, a finely linked hypertext format where the links specify a semantic role rather than navigational options. That is, links signify the location(s) where markup contained in a given annotation document *would appear* in the document to which it is linked. As such, annotation information comprises remote or "stand-off" markup that is virtually added to the document to which it is linked. In principle, the original data may contain no markup at all (or, more likely, markup for gross logical structure only); all markup can be retained in separate documents with links into the original based on offsets.

The standoff scheme, then, requires addressing elements within the original document, as well as characters and chains of characters within those elements. It also requires that elements and characters can be addressed both within the same document and other documents. XML provides the following linking mechanisms, which satisfy these requirements:

- *XLink* (DeRose, et al., 2000), a mechanism for specifying a link (uni-directional or more complex linking structures) between two or more resources or portions of resources;
- the *XML Path Language* (XPath) (Clark & DeRose, 1999), an extended addressing syntax that defines a concise notation for element localization in the document tree (as defined by the nesting of elements in the document itself), and allows addressing fragments within a particular element by providing predicates for manipulating chains of characters;
- *XPointer* (DeRose, Daniel, & Maler, 1999), which extends XPath syntax to allow addressing points and ranges as well as nodes, locating information by string matching, and use of addressing expressions in URI-references as fragment identifiers.

For example, the Xpath expression `/div/p[2]/s[3]` specifies the third `<s>` (sentence) element within the second `<p>` (paragraph) element within each `<div>` (text division) element; `/descendant::p` specifies all `<p>` elements in the document. In addition, Xpath allows addressing text fragments within a particular element by providing predicates for manipulating chains of characters. The expression

```
substring(/p/s[2]/text(),6)
```

selects the string "one would expect that the whole sky would be as bright as the sun, even at night." from the following text:

```
<p><s id="d3p13s4">The difficulty
is that in an infinite static
universe nearly every line of
sight would end on the surface of
```

[3] Ide, Bonhomme, and Romary, 2000.

[4] Macleod, Ide, and Grishman, 2000.

```
a star.</s><s id="d3p13s5">Thus
one would expect that the whole
sky would be as bright as the sun,
even at night.</s></p>
```

The Xlink mechanism can be used to link corresponding segments of two or more primary documents (as for alignment of text or speech), to link annotation documents to a base document containing the primary data, or, more generally, to link resources in any medium (audio, video, etc.). This allows for linking speech, external images, video, applets, form-processing programs, style sheets, etc.

In addition to specifying the target location for information in the same or external documents, XLink attributes can be used to specify the role of the link, i.e., how the link should be activated (by hand, or automatically by the browser) and what to do with the target fragment (replace it or insert it into the source document).

2 XML transformations

The Extensible Style Language (XSL) is a part of the XML framework, consisting of two parts: the XSL formatting or "style sheet" language, and a powerful tree-traversal language, XSLT (Clark, 1999), that can be used to convert any XML document or documents into another document in any form (e.g., XML, well-formed HTML, plain text, etc.) by selecting, rearranging, and/or adding information to it. The transformed documents may or may not be intended for rendering data on a computer screen, but may be used to move data from one computer system or program to another (e.g., to transduce between encoding and/or annotation formats, etc.).

XSLT supports the following kinds of document manipulation:

- selection of elements or portions of element content using the XPath syntax, from one or more XML documents;
- rearrangement or transformation of extracted information (including not only text content but also element names, etc.) in the target document;
- addition of information in the target document.

A suite of documents representing a base document (or documents) and its annotations can be manipulated to serve any application that relies on part or all of its contents. Thus, XSLT is likely to have the most impact on the design of language processing tools.

Several projects have developed and implemented language processing tools and tool architectures intended to facilitate flexibility and reusability: for example, MULTEXT (Ide & Véronis, 1994), LT XML (McKelvie, Brew, & Thompson, 1998), GATE (Cunningham, Wilks, & Gaiauskas, 1996), ATLAS (Bird, *et al.*, 2000). While each of these systems is slightly different, they all implement a modular, "plug-and-play" tool architecture based on a three-layered design: one for physical storage representation; one to translate to and from the physical storage representation to one or more internal formats, and an API to enable application development. In addition, all assume SGML or (in the more recently developed systems) XML as the physical representation, together with the use of the stand-off strategy for annotation. The SGML or XML documents containing the data and its annotations are typically transduced into some internal format used by the tools; at any stage in the processing, the results may be transduced back into SGML or XML As a powerful language for selecting from one or several documents and transducing the data into other formats, XSLT provides the means to enable the import and export of data from and to XML.

Of course, XSLT can be used with the documents resulting from processing by tools to deliver the data in any desired format. Although space prevents a full description of XSLT, which is relatively complex, a short example can provide some idea of the possibilities. Using as input a document containing morpho-syntactic information (e.g., a document containing the fragment in Figure 1[5]), the XSLT document in

[5] Note that this document, encoded according to the

Figure 2 can be used to create an HTML document that displays a text in "word | lemma | pos" form. When the resulting HTML document is loaded into a browser, it will display the following:

```
It|it|PPER3 was|be|PAST3 a|a|DINT
bright|bright|ADJE cold|cold|ADJE
day|day|NN…
```

```
<?xml version="1.0">
<chunk type="BODY" lang="en"
 xml:base=
"http://www.cs.vassar.edu/~ME/Oen.xcesDoc#">
 <par xlink:href="xptr(substring(//p[1]">
  <s xlink:href="xptr(substring(//p/s[1]">
   <tok type="WORD"
     xlink:href=
     "xptr(substring(//p/s[1]/text(),1,2">
      <orth>It</orth>
      <disamb>
        <base>it</base>
        <msd>Pp3ns</msd>
        <ctag>PPER3</ctag></lex>
      <lex>
        <base>it</base>
        <msd>Pp3ns</msd>
        <ctag>PPER3</ctag></lex></tok>
    <tok type="WORD"
      xlink:href=
      "xptr(substring(//p/s[1]/text(),4,2">
      <orth>was</orth>
      <disamb>
        <base>be</base>
        <msd>Vmis3s</msd>
        <ctag>PAST3</ctag></lex>
      <lex>
        <base>be</base>
        <msd>Vais1s</msd>
        <ctag>AUX1</ctag></lex>
      <lex>
    <base>be</base>
    <msd>Vais3s</msd>
    <ctag>AUX3</ctag></lex>
  <lex>
    <base>be</base>
    <msd>Vmis1s</msd>
    <ctag>PAST1</ctag></lex>
  <lex>
    <base>be</base>
    <msd>Vmis3s</msd>
    <ctag>PAST3</ctag></lex></tok>…
```

Figure 1 : Fragment of an xcesAna document

The XSLT script in Figure 2 could be modified to produce output in any desired form, or to produce another XML document containing the merged data and annotation documents (see [www.cs.vassar.edu/XCES] for some more complex examples). Similarly, XSLT can be used to produce concordances, paired sentences or words from a parallel text, or even a web document that displays the orthographic representation of a text and provides the audio rendition when the word is clicked on, etc. XSLT can also be used to implement an inheritance mechanism over the document tree[6]; for example, Ide, Kilgarriff, & Romary (2000) show how XSLT can implement inheritance mechanism for lexical information.

```
<xsl:stylesheet version="1.0"
   xmnls:xsl=
   "http://www.w3.org/1999/XSL/Transform">

<xsl:template match= "/">
     <html>
       <body>
           <xsl:apply-templates/>
       </body>
     </html>
</xsl:template>

<xsl:template match="//par"/>

  <xsl:for-each select="//tok"/>
     <xsl:value-of select="orth"/>
     <xsl:text>|</xsl:text>
     <xsl:value-of select="disamb/base"/>
     <xsl:text>|</xsl:text>
     <xsl:value-of select="disamb/ctag"/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Figure 2 : XSLT document to create HTML output

3 XML Schemas

The XML Schema definition language (Thompson, et al., 2000; Biron & Malhotra, 2000) enables document creators to constrain and document the meaning, usage and relationships of the constituent parts of XML documents: datatypes, elements and their content, and attributes and their values. Schemas can also be used to provide default values for attributes and elements. As such, XML schemas provide means to define an abstract data model for a class of documents. While duplicating (or making explicit) some of the capabilities provided by XML DTDs, they significantly extend their power and provide for much tighter validation of document form and content.

xcesAna specifications (Ide, Bonhomme, & Romary, 2000), contains full segmentation and annotation information, including full morpho-syntactic specifications for all potential annotations and the results of automatic disambiguation.

[6] See also Erjavec *et al.* (2000)

XML schemas have considerable implications for the creation of annotated data. The following lists only a few possibilities for the application of XML schemas:

- different attribute declarations and/or content models can apply to elements with the same name in different contexts, building on definitions using XML Namespaces (Bray, Hollander, and Layman, 1999). This allows for more tightly constrained content models than possible with DTDs. For example, names in headers (names of authors, etc., consisting of the usual "first name", "last name" elements) and names in the text ("named entities") should have different content models and attributes in order to provide for tight validation of form in each context.
- equivalence classes can be defined for groups of elements and/or attributes, indicating that they may be used in the same ways as defined for a particular named element ("the exemplar").
- attribute or element values, or combinations of attribute and element values, can be constrained to be unique. That is, it is possible to indicate in a computational lexicon that only one entry can be defined with the value of a given word form as its content (or the content of one of its child elements), only one paragraph can have an attribute indicating that it is the 23rd, only one disambiguated form is given for each token in an annotation document, or only one correspondence for a given item in an alignment document. Obviously, this is useful for error detection and prevention.

dependencies can be established based on values of elements or attributes. This has similar benefits for error detection in creating annotation documents: nouns can be prevented from being assigned a tense, tokens whose *type* attribute has the value PUNCT can be specified to include only `<orth>` elements containing specific characters, etc. In addition, annotation labels (e.g., POS indicators) used in an annotation document can be specified elsewhere, and element content can be constrained to these values only.

Conclusion

This paper outlines some of the potential uses of the mechanisms provided within the XML framework for the creation and use of annotated text and speech data. Because XML is an international standard that is becoming the base of information exchange and access over the World Wide Web, high-end language processing applications intended to extract and manipulate information from diverse sources will necessarily handle XML. It is to our advantage to exploit the XML framework to our greatest advantage, and to ensure compatibility of the data we create with the emerging standard.

Acknowledgements

The author gratefully acknowledges the contributions of Laurent Romary, Patrice Bonhomme, and Chris Brew to the ideas and examples in this paper.

References

Bird, S., Day, D., Garofolo, J., Henderson, J., Laprun, C. Liberman, M., 2000. ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation. In *Proceedings of the Second International Language Resources and Evaluation Conference*. Paris: European Language Resources Association, 1699-1706.

Biron, P. & Malhotra, A., 2000. XML Schema Part 2: Datatypes. W3C Working Draft, 25 February 2000. http://www.w3.org/TR/xmlschema-2/.

Bray, T., Paoli, J., Sperberg-McQueen, C.M. (eds.), 1998. Extensible Markup Language (XML) Version 1.0. W3C Recommendation. http://www.w3.org:TR/1998/REC-xml-19980210.

Bray, T., Hollander, D., Layman, M., 1999. Namespaces in XML. World Wide Web Consortium Recommendation, 14 January 1999. http://www.w3.org/TR/REC-xml-names/.

Clark, J. (ed.), 1999. XSL Transformations (XSLT). Version 1.0. W3C Recommendation. http://www.w3.org/TR/xslt.

Clark, J. and DeRose, S., 1999. XML Path Language (XPath). Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.

Cunningham, H., Wilks, Y., Gaizauskas, R., 1996. GATE -- a General Architecture for Text Engineering. In *Proceedings of the 16th International Conference on Computational Linguistics, COLING-96*, Copenhagen, Denmark, 1057-1060.

DeRose, S, Maler, E., Orchard, D., Trafford, B. (eds.), 2000. XML Linking Language (XLink). W3C Working Draft, 21 February 2000. http://www.w3.org/TR/xlink.

DeRose, S., Daniel, R., & Maler, E., 1999. XML Pointer Language (XPointer). W3C Working Draft, 6 December 1999. http://www.w3.org/TR/xptr.

Ide, N. & Brew, C., 2000. Requirements, Tools, and Architectures for Annotated Corpora. In *Proceedings of the EAGLES/ISLE Workshop on Meta-Descriptions and Annotation Schemas for Multimodal/Multimedia Language Resources and Data Architectures and Software Support for Large Corpora*. Paris: European Language Resources Association, 1-6.

Ide, N., & Véronis, J., 1994. MULTEXT: Multilingual Text Tools and Corpora. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING'94*, Kyoto, Japan, 588-92.

Ide, N., Bonhomme, P., & Romary, L., 2000. XCES: An XML-based Encoding Standard for Linguistic Corpora. In *Proceedings of the Second International Language Resources and Evaluation Conference*. Paris: European Language Resources Association, 825-30.

Ide, N., Kilgarriff, A., Romary, L., 2000. A Formal Model of Dictionary Structure and Content. In *Proceedings of EURALEX'00* (to appear).

Macleod, C., Ide, N., Grishman, R., 2000. Progress Report on the American National Corpus. In *Proceedings of the Second International Language Resources and Evaluation Conference* (to appear). Paris: European Language Resources Association, 831-35.

McKelvie, D., Brew, C., & Thompson, H. 1998. Using SGML as a Basis for Data-Intensive Natural Language Processing. *Computers and the Humanities* 31:5, 367-388.

Thompson, H., Beech, D., Maloney, M. Mendelsohn, N., 2000. XML Schema Part 1: Structures. W3C Working Draft, 25 February 2000. http://www.w3.org/TR/xmlschema-1/.

Benefits of Modularity in an Automated Essay Scoring System

Jill BURSTEIN
ETS Technologies
Rosedale Road
Princeton, NJ 08541
jburstein@ets.org,
jburstein@etstechnologies.com

Daniel MARCU
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6601
marcu@isi.edu

Abstract

E-rater is an operational automated essay scoring application. The system combines several NLP tools that identify linguistic features in essays for the purpose of evaluating the quality of essay text. The application currently identifies a variety of syntactic, discourse, and topical analysis features. We have maintained two clear visions of *e-rater's* development. First, new linguistically-based features would be added to strengthen connections between human scoring guide criteria and *e-rater* scores. Secondly, *e-rater* would be adapted to automatically provide explanatory feedback about writing quality. This paper provides two examples of the flexibility of *e-rater's* modular architecture for continued application development toward these goals. Specifically, we discuss a) how additional features from rhetorical parse trees were integrated into *e-rater*, and b) how the salience of automatically generated discourse-based essay summaries was evaluated for use as instructional feedback through the re-use of *e-rater*'s topical analysis module.

1 Introduction

E-rater is an operational automated essay scoring system that was designed to score essays based on holistic scoring guide criteria (Burstein, et al 1998), specifically for the Graduate Management Admissions Test (GMAT). Holistic scoring guides instruct the human reader to assign an essay score based on the quality of writing characteristics in an essay. For instance, the reader is to assess the overall quality of the writer's use of *syntactic variety*, the *organization of ideas*, and appropriate *vocabulary use*. *E-rater* combines several NLP tools to identify syntactic, discourse, and vocabulary-based features.

In developing this automated essay scoring application, we have two primary goals. We are continually experimenting with *e-rater* to enrich its current feature sets to represent additional scoring guide criteria. Furthermore, we are adapting the system to provide test-takers with feedback about the quality of their writing, so that they may use it to improve their overall writing competency.

In light of the application development goals, this paper discusses the *e-rater* application components and the benefits of its modular design. Using specific studies to exemplify, the paper points out the importance of the application's modularity with regard to: a) experiments that evaluate the integration of new features, and b) the re-use of modules for evaluations that contribute to the adaption of the system toward the generation of feedback.

2 *E-rater* System Modules & Design

The *e-rater* application currently has five main independent modules. The application is designed to identify features in the text that reflect writing qualities specified in human reader scoring criteria. The system has three independent modules for identifying scoring guide relevant features from the following categories: syntax, discourse, and topic. Each of the feature recognition modules described below identifies features that correspond to scoring guide criteria features which can be correlated to essay score, namely, *syntactic variety*, *organization of ideas*, and *vocabulary usage*. *E-rater* uses a fourth independent model building module to select and weight predictive features for essay scoring. The model building module reconfigures the feature selections and associated regression weightings given a sample of human reader scored essays for a particular test question. A fifth module is used for final score assignment.

All modules are called from a main driver program. Each independent module can be run as a stand-alone program. There are interactions between the modules, and these are described throughout the paper.

The modules and their subcomponents are written in either Perl or C programming languages. The model building module is implemented in SAS, a statistical programming language. *E-rater* can be run on both Unix or PC platforms.

2.1 Syntactic Module

E-rater's syntactic analyzer (parser) works in the following way to identify syntactic features constructions in essay text. *E-rater* tags each word for part-of-speech (Brill, 1997), uses a syntactic "chunker" (Abney, 1996) to find phrases, and assembles the phrases into trees based on subcategorization information for verbs (Grishman, et al, 1994). The parser identifies various clauses, including infinitive, complement, and subordinate clauses. The ability to identify such clause types allows *e-rater* to capture *syntactic variety* in an essay.

2.2 Discourse Module

E-rater identifies discourse cue words, terms, and syntactic structures, and these are used to annotate each essay according to a discourse classification schema (Quirk, et al, 1985). The syntactic structures, such as complement clauses, are outputs from the syntactic module described earlier. Such syntactic structures are used to identify, for example, the beginning of a new argument based on their position within a sentence and within a paragraph.

Generally, *e-rater's* discourse annotations denote the beginnings of arguments (the main points of discussion), or argument development within a text, as well as the classification of discourse relations associated with the argument type (e.g., *parallel relation*). Discourse features based on the annotations have been shown to predict the holistic scores that human readers assign to essays, and can be associated with *organization of ideas* in an essay.

E-rater uses the discourse annotations to partition essays into separate arguments. These argument partitioned versions of essays are used by the topical analysis module to evaluate the content individual arguments (Burstein, et al, 1998; Burstein & Chodorow, 1999). *E-rater's* discourse analysis produces a flat, linear sequence of units. For instance, in the essay text *e-rater*'s discourse annotation indicates that a contrast relationship exists, based on discourse cue words, such as *however*. Discourse-based relationships across sentences in text are not defined by this module.

2.3 Topical Analysis Module

Vocabulary usage is another criterion listed in human reader scoring guides. To capture use of vocabulary, or identification of topic *e-rater* includes a topical analysis module. The

procedures in this module are based on the vector-space model, commonly found in information retrieval applications (Salton, 1989). These analyses are done at the level of the essay (big bag of words) or the argument.

For both levels of analysis, training essays are converted into vectors of word frequencies, and the frequencies are then transformed into word weights. These weight vectors populate the training space. To score a test essay, it is converted into a weight vector, and a search is conducted to find the training vectors most similar to it, as measured by the cosine between the test and training vectors. The closest matches among the training set are used to assign a score to the test essay.

As already mentioned, *e-rater* uses two different forms of the general procedure sketched above. For looking at **topical analysis at the essay level**, each of the training essays (also used for training *e-rater*) is represented by a separate vector in the training space. The score assigned to the test essay is a weighted mean of the scores for the 6 training essays whose vectors are closest to the vector of the test essay.

In the method used to analyze **topical analysis at the argument level**, all of the training essays are combined for each score category to populate the training space with just 6 "supervectors", one each for scores 1-6. The argument partitioned version of the essays generated from the discourse module are used in the set of test essays. Each test essay is evaluated one argument at a time. Each argument is converted into a vector of word weights and compared to the 6 vectors in the training space. The closest vector is found and its score is assigned to the argument. This process continues until all the arguments have been assigned a score. The overall score for the test essay is an adjusted mean of the argument scores.

2.4 Model Building and Scoring

The syntactic, discourse, and topical analysis modules each yield numerical outputs that can be used for model building, and scoring. Specifically, counts of identified syntactic and discourse features are computed. The counts of features in each essay are stored in vectors for each essay (test candidate). Similarly, for each essay, the scores from the topical analysis by-essay, and topical analysis by-argument procedures are stored in vectors. The vectors generated from each module are stored in independent output files. The values in the vectors for each feature category are then used to build scoring models for each test question as described below.

To build models, a training set of human scored sample essays is collected that is representative of the range of scores in the scoring guide. For the type of essay generally scored by *e-rater*, the scoring guides typically have a 6-point scale, where a “6” indicates the score assigned to the most competent writer, and a score of “0” indicates the score assigned to the least competent writer. Optimal training set samples contain 265 essays that have been scored by two human readers. The data sample is distributed in the following way with respect to score points: 15 1’s, and 50 in each of the score points 2 through 6.[1]

The model building module is a program that runs a forward-entry stepwise regression. Feature values stored in the syntactic, discourse, and topical analysis vector files are the input to the regression program. This regression program automatically selects the features which are predictive for a given set of training data (from one test question). The program outputs the predictive features and their associated regression weightings. This output composes the model that is then used for scoring.

In an independent scoring module, a linear equation is used to compute final essay score. To compute the final score for each essay, the

sum of the product of each regression weighting and its associated feature integer is calculated.

2.4.1 *Advantages of Modularity for Model Building & Scoring*

In the model building program, one can choose to use all the features for a particular run, or some feature subset. This flexibility makes it relatively easy to introduce new sets of features into the model building procedure for research and development purposes. The model building module can be run independently. Therefore, once *e-rater* has generated feature vector files for training samples, the model building module can be revised accordingly, so that numerous runs can be performed on data sets, using various feature combinations for model building, without rerunning the entire application.[2]

Once new models have been built, they can be easily cross-validated on an independent data set. Specifically, once the feature vector information has been generated for the independent data set, it can be scored quickly using any model desired to test the performance of the model. For each new model, the vector information, (e.g., counts of syntactic clauses) is recombined in the linear equation using the model-specific predictive features and regression weightings. Therefore, given the same set of test data, performance may vary across models.

The design of an independent scoring module is also useful for tracking down changes in performance that occur when making revisions to the code. Code changes can have unexpected affects on feature assignment which can alter vector counts. If vector counts are affected for a feature used in the model, then this may affect the final essay score. Simple comparisons can be made between the scoring equation variables in a previous version of the code, and the revised version. Such comparisons are often useful to trouble-shoot the unanticipated affects of code changes on specific feature variables, and final scores.

3 Benefits of Modularity for Application Development

As discussed earlier, a goal in *e-rater* application development is to enhance the current feature set by adding new features that correspond to characteristics of writing defined in the scoring guide criteria. Currently, *e-rater* features represent these scoring guide criteria: *syntactic variety*, *organization of ideas*, and *vocabulary usage*. *E-rater* discourse features capture the criterion, *organization of ideas*, at a high level. However, the existing discourse features are linear, and do not express relationships across a text. Hierarchical discourse relations can be expressed with rhetorical structure theory (RST) features (Mann and Thompson, 1989).

In an experiment, we evaluated the potential use of RST features in *e-rater*. An existing rhetorical parser (Marcu, 1997) was used to generate parse trees for essay samples from 20 test questions to the GMAT. A program was written to identify the RST features in essays, compute counts of tokens, types and ratios of the features, and to store the three categories of feature counts in vectors for each essay. For the RST vector files, separate files were output for each type of feature count (tokens, types, and ratios). The model building program was modified to introduce the new RST variables. In this way, the RST feature variables could be evaluated either individually or in combination during model building -- as specified in the model building program.

E-rater had been run on these 20 essay samples previously, so all of the standard vector information that *e-rater* outputs already existed. The model building component in *e-rater* can easily be run independently once all vector information exists, so the process of building new models after RST feature variables had been integrated was quickly and easily done.

Accordingly, the evaluation of experimental models on independent test sets is also conveniently done with the *e-rater* scoring module. Specifically, the predictive features and their associated regression weightings from the new models that include RST features are introduced into the linear equation used in scoring.

So, in experimental runs (of which we do many!), only the additional pieces, in this case the rhetorical parser, and RST feature extraction program, were required for feature generation, and extraction, and creation of formatted vector files used as input to the model building and scoring programs. This particular experiment provided strong evidence that the RST features would serve to enhance the current application.

Running model building and scoring independently on an essay sample (training and cross-validation[3] sets) for a single prompt takes approximately 5 seconds. To build a model and score the same essay sample would take up to an hour. The independence of the model building and scoring programs allows unlimited flexibility for continued research and development of the application with regard to the addition of new features.

4 Re-Using *E-rater's* Topical Analysis Module

A strong motivation behind *e-rater* application development is to adapt the system so that it generates feedback along with an essay score. In a recent experiment, we re-used the *e-rater* topical analysis module, and the essay data to evaluate the saliency of text in automated essay summaries (Burstein and Marcu, 2000). The score from the topical analysis by-argument module is amongst *e-rater's* strongest predictors of essay score. That is, it is almost always selected in the model building process. Furthermore, by itself, the topical analysis by-argument score agrees with human reader scores approximately 85% of the time, on average.[4]

Within the context of adapting *e-rater* to generate feedback, we hypothesized that summaries could be used to determine the most important points of essays. We envisioned at least two possible uses of essay summaries. First, for any essay question, one can, for example, build individual summaries of all essays of score 6 (the most competent essay); use sentence-based similarity measures to determine the topics that occur frequently in these essays; and present these topics to a test-taker. Test-takers would then be able to assess what topics they might have included in order to be given a high score. Second, for any given essay, one can build a summary and present it to the test-taker in a format that makes explicit whether the main points in the summary cover the topics that are considered important for the test question. One way of doing this might be to present to test-takers, summaries of other essays that received a high score. Test-takers would be able to assess whether the rhetorical organization of their essays makes the important topics salient.

For the experiment, the training and cross-validation sets from the 20 GMAT essay samples were run through an existing discourse-based automatic text summarizer (Marcu, 1999). Summaries were generated at different compression rates: 20%, 40% and 60%. For each of the 20 samples, the topical analysis module was run on training and cross-validation sets. We evaluated the performance of the topical analysis by-argument score on all summaries.[5] The performance of the topical analysis by-argument measure was higher for 40% and 60% summaries than using the full text of essays. The re-use of this *e-rater* module for evaluating the saliency of essay summaries proved to be informative.

5 Discussion and Conclusions

In this paper, we have discussed the importance of modularity in an automated essay scoring system for research and development. Modularity, especially with regard to the model building and scoring functionality, is critical to

application development. Unlike other NLP tools, such as part-of-speech taggers and syntactic parsers, for which there is a reasonably well-defined and standard feature set, the feature set that will become part of *e-rater* will be determined by continued experimentation. Though *e-rater* currently contains linguistic features that have been shown to be highly predictive of essay score, the interests and queries from the writing community require further experimentation with new features (such as RST features).

As was discussed in the paper, the new types of features that could become used in the system reflect qualities of writing that appear in scoring guide criteria. These criteria are "fuzzy" in some sense, in that they describe general qualities of writing (e.g., organization of ideas), but do not state specifically what form of linguistic feature will reflect a particular quality. Therefore, repeated experimentation with new features is critical in order to discover how to represent these criteria computationally.

From a purely linguistic perspective we must first ask: *What linguistic features map to the concept, organization of ideas, for instance?* But, in addition, from the computational linguistic view we must also ask: *What are the linguistic features that map to a scoring guide criteria that can be reliably captured by NLP-based tools?* To further develop *e-rater*, we must be able to handle both points-of-view; hence, a modular system is required in which we can easily test the use of new features (or, hypotheses about new features) toward further application development. The ability to easily modify *e-rater's* model building module, so that models can be easily reconfigured with new feature combinations allows us to conveniently evaluate the performance of new features. This is shown in the experiment in which RST features were introduced into *e-rater* models. This approach also allows us to quickly evaluate feature performance within the linear regression modeling technique. What we have also learned through our continued research is that alternative measures outside of the linear regression may also be useful to characterize the competency of an essay with regard to its rhetorical structure. Similar research is on-going that employs alternative methods of evaluating the relevance of essay vocabulary using measures independent of the regression. It is critical to have the ability to evaluate the reliability of different approaches for representing and evaluating features of writing as they relate to writing competency.

A second argument for the modularity of the system is to be able to re-use independent *e-rater* tools and data for related applications (e.g., automated scoring of short answers). Alternatively, in the summarization experiment, we were able to re-use the essay data for the purpose of generating summaries, and also to re-use the topical analysis tool to evaluate the performance of the tool on the summaries. Since the topical analysis component is an independent module, no modifications were required to run the experiment.

Acknowledgements

We thank Martin Chodorow and Dennis Quardt for helpful discussions and insights. We are also grateful to Magdalena Wolska for writing the programs that perform the model building and scoring functionality for our research.

References

Abney, S. (1996) Part-of-speech tagging and partial parsing. In Young, S. and Bloothooft, G. (eds), Corpus-based Methods in Language and Speech. Dordrecht: Kluwer, 118-136.

Brill, E. (to appear). Unsupervised Learning of Disambiguation Rules for Part of Speech Tagging, Natural Language Processing Using Very Large Corpora. Dordrecht: Kluwer Academic Press.

Burstein, J. and D. Marcu (2000). Toward Using Text Summarization for Essay-Based Feedback. In the *Proceedings of TALN 2000, Swiss Federal Institute of Technology*, Lausanne, Switzerland, October, 2000.

Burstein, J., Kukich, K., Wolff, S., Lu, C., Chodorow, M., Braden-Harder, L., and Dee Harris, M. (1998). Automated Scoring Using A Hybrid Feature Identification Technique. In the *Proceedings of the Annual Meeting of the Association of Computational Linguistics*, Montreal, Canada.

Grishman, R., Macleod, C., and Meyers, A. (1994). "COMLEX Syntax: Building a Computational Lexicon", Proceedings of Coling, Kyoto, Japan. (available for download at:http://cs.nyu.edu/cs/projects/proteus/comlex/)

Mann, W.C. and Thompson, S.A. (1988). Rhetorical Structure Theory: Toward a Functional Theory of Text Organization. *Text* 8(3), 243–281.

Marcu, D. (1999). Discourse trees are good indicators of importance of text. In I. Mani and M. Maybury eds., *Advances in Automatic Text Summarization*, pp. 123-136.The MIT Press.

Marcu D. (1997). The Rhetorical Parsing of Natural Language Texts. *The Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pp. 96-103.

Quirk, R., Greenbaum, S., Leech, S., and Svartik, J. (1985). A Comprehensive Grammar of the English Language. Longman, New York.

Salton G. (1989). *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley Publishing Co.

[1] Essays at score point 0 are not required as these tend to contain no text at all, or to be off-task in some way.

[2] In practice, we wrote a program that performs the functionality of the model building and scoring modules. It is in this program where code revision actually occurs, not in the application code.

[3] Cross-validation samples usually contain about 500 essays.

[4] Agreement statistics are for the 20 GMAT essay samples discussed. The agreement indicates that the human reader and topical analysis scores are within 1-point. This is a standard measure of agreement between 2 human readers. Additionally, two human readers agree within 1 point of each other approximately 92% of the time.

[5] The performance of the topical analysis by-argument scores is approximately 5% higher than the scores from the topical analysis by-essay procedure.

An Integrated Development Environment for Spoken Dialogue Systems

Matthias Denecke
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
denecke@cs.cmu.edu

Abstract

Development environments for spoken dialogue processing systems are of particular interest because the turn-around time for a dialogue system is high while at the same time a considerable amount of components can be reused with little or no modifications. We describe an Integrated Development Environment (IDE) for spoken dialogue systems. The IDE allows application designers to interactively specify reusable building blocks called *dialogue packages* for dialogue systems. Each dialogue package consists of an assembly of data sources, including an object-oriented domain model, a task model and grammars. We show how the dialogue packages can be specified through a graphical user interface with the help of a wizard.

1 Introduction

The specification and design of interactive spoken language systems has become the focus of research recently. Partly fueled by the increasing demand of spoken language applications and telephony-based services, the deployment of development environments has increased. At the time of writing, at least three main types of dialogue tools can be distinguished. One approach to development environments consists of graphical editors for Finite State Automata (FSA) [Sutton et al, 1996], [Cole, 1999]. These systems equate a dialogue with a possible path from the start state to one of the accepting states. Possible actions of the application are specified by annotations on states or arcs or both. Besides relying on a dialogue model that has been considered as problematic in the past, finite-state automata based dialogue editors do not exploit the desirable characteristics of software engineering, such as reusability and orthogonality of the components. For example, recovery strategies need to be duplicated for each state in which they should be applied. Moreover, they require a system designer to anticipate every single possible path through the system, a fact that leads to an explosion of dialogue states.

Another approach to development environments emphasizes reusability of the domain model over graphical design interfaces. Here, object-oriented features of the underlying programming language such as JAVA or C++ are used to design a class hierarchy of *speech objects* or *dialogue modules* that can be assembled and re-assembled for new applications. These modules are often used for basic data types, such as *date, time, credit card numbers*, etc. This approach has proven its practicability in numerous commercial applications. Since the modules can be reused, this is an improvement over finite-state based dialogue machines. However, fine tuning of recovery strategies requires separate fine-tuning in each module. Moreover, the dialogue flow is partly defined by an FSA whose nodes consist of the dialogue modules. When a node is reached, the dialogue module determines the dialogue control until it gives up control and an adjacent arc is traversed.

A third approach consists in designing a library of reusable dialogue strategies based on the observation that the behavior of a dialogue manager should be predictable in similar situations across several domains. Araki et al [Araki et al, 1999] proposed a library of dialogue strategies to be reused. Koelzer [Koelzer, 1999] proposed a reusable dialogue system architecture based on specifications of knowledge sources for the different components.

In this paper, we identify knowledge sources such as grammars, task models and database conversion rules, that characterize our dialogue manager for a given application. Each of the knowledge sources can be composed of smaller, modular knowledge sources. A collection of these knowledge source modules, called a *dialogue package*, specifies a subdomain of a dialogue application. We borrow techniques known from object oriented programming languages to combine partial specifications of knowledge sources to form the knowledge sources for a new application. The specifications are mostly declarative rather than procedural, leaving to the dialogue manager the decision how best to interpret them in the context of the dialogue. We describe the implementation of a wizard-based integrated development environment called Chapeau Clac that allows the specification of the knowledge sources, their in-

tegration and testing.

2 The Architecture of the IDE and the Dialogue System

2.1 The Architecture of the Dialogue System

The dialogue manager makes use of different knowledge sources. First, it contains a set of task descriptions or task models. A task description can be considered as a form to be filled in through the dialogue, together with constraints stating the minimum amount of information necessary to execute the task. The dialogue strategy is specified in a declarative programming language similar to PROLOG that can be easily adapted to the task at hand should the need arise.

The state of the dialogue system at any given time is determined implicitly by the relations of the forms with the information available in the discourse at that time. For example, a task description whose constraints are inconsistent with information in the discourse can not be a description of the intent of the user. The elements the forms can be populated with are descriptions of objects, actions and properties of objects and actions drawn from a domain model. The domain model can loosely be compared to a class hierarchy in object oriented programming languages. In addition to task model and domain model, the dialogue manager uses data base conversion rules to generate SQL queries and to transform the result sets. As the domain model is dependent on the particular speech application, it belongs to the knowledge sources to be specified through the wizard.

As the semantics of the utterances are expressed in terms of the domain model, we need to provide a mechanism to translate the text input from the speech recognizer into a canonical representation. Attributed grammar rules provide transformation between text input and semantic representations.

The place of the dialogue manager in the system is similar in spirit to, but different in functionality from, the design of a Graphical User Interface for a back-end application. In the case of the GUI, the design of windows, dialog boxes and menus is independent from the design of the back-end application that uses these graphical display elements. Similarly, in our approach, the design of dialogue grammars, dialogue goals and domain models is independent of the design of the back-end application. As in GUIs, the back-end application is notified of manipulations through events and callback functions. This approach separates clearly the speech user interface from the back-end application. The callbacks and events constitute one integration point between speech user interface and back-end application whose form and content needs to be specified for each new speech application.

It should be noted, however, that the analogy between graphical and speech user interfaces ends here. Reference in GUIs is extensional. For example, the click of the button or a menu, together with the state of the application and the focus, determines the intended action. In spoken dialogue systems, the need to resolve reference of noun phrases or ellipsis forces us to provide one more integration point with the back-end application in order to allow database retrieval.

Consequently, we argue that a dialogue manager for a given speech application can be characterized by the specification of four knowledge sources, namely (i) the domain model to characterize the semantic content of the utterances, (ii) the conversion from the text input into a canonical semantic representation and vice versa, (iii) the task model to describe the event stream from the speech user interface to the back-end application, and (iv) the conversion from semantic representation into database retrieval requests. Figure 1 shows the place of the knowledge sources in the dialogue manager. As can be seen, the knowledge sources (ii) to (iv) encapsulate entirely the dialogue manager from the remaining components of the system.

Note that we make no assumptions as to how the dialogue manager might make use of these knowledge sources. In particular, we do not make any assumptions as to how the dialogue strategy might determine the actions of the dialogue manager. As long as the provided knowledge sources are sufficient for the dialogue manager to determine its actions, the dialogue manager could implement a simple information seeking dialogue system or a more sophisticated system based on speech act or discourse theories.

All four knowledge sources can be modularized more or less straightforwardly. The domain model can be composed of different subdomain [Denecke and Waibel, 1999]; new concepts may use multiple inheritance of abstract base types. Grammar rules containing generic semantic information can be specialized and adapted to the given domain. Database conversion and dialogue goal specification modules may simply be joined; but see section 6 for potential problems. It is the task of the wizard to help the user in specifying and reusing these knowledge sources.

2.2 Requirements for the IDE

The requirements for the IDE's functionality comprise three main items. First, it should guide the application designer to specify and modify the spoken language part of an entire application through a GUI. The data sources relevant for the spoken language interface currently include grammars, domain model, data bases, task model and input/output channels. Moreover, conversions back and forth

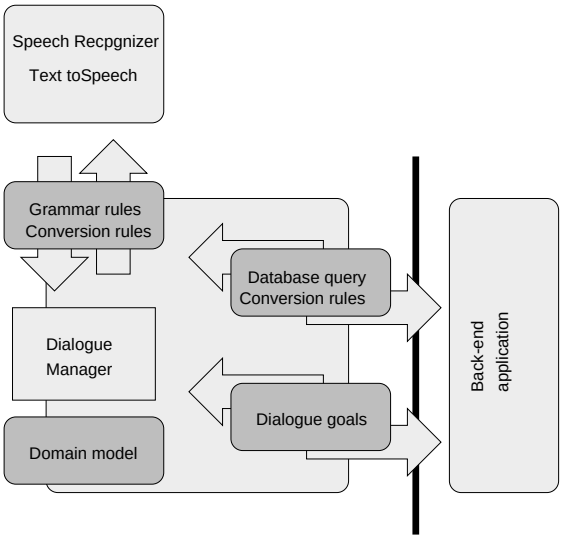

Figure 1: The place of the knowledge sources in the system architecture.

between semantic representation of utterances and database queries and results on one hand and text on the other need to be specified. The object model of the data sources used in the dialogue system is shown in figure 2. Second, the IDE should support a developer by adapting and modifying the existing dialogue strategy through the usual debugging tools such as tracer, walk through, call stacks, breakpoints and variable dumps. Third, it should support an application designer in testing the final application using batch tests and single utterance tests.

In addition, since experienced users may obtain results faster using a keyboard rather than a wizard interface, the system designer should be able switch between a standard text editor and the wizard interface at any time in the design process. Surprisingly, this design requirement had a more thorough impact on the layout of the system implementation than anticipated. For each data source to be specified, we need two classes that implement the data source: one class implementing the data source itself, and a second class implementing a description of the data source. The second class consists only of primitive data types such as strings and integers that can easily be manipulated by a wizard interface and can also be easily parsed from a file. When the final data sources are instantiated, the constructor of the data source, taking a description as its only argument, creates the data source according to the specification.

In addition to the decoupling of the GUI with the dialogue system itself, the description objects also introduce an additional level of abstraction that allows the replacement of similar data source implementations (such as different grammar formalisms

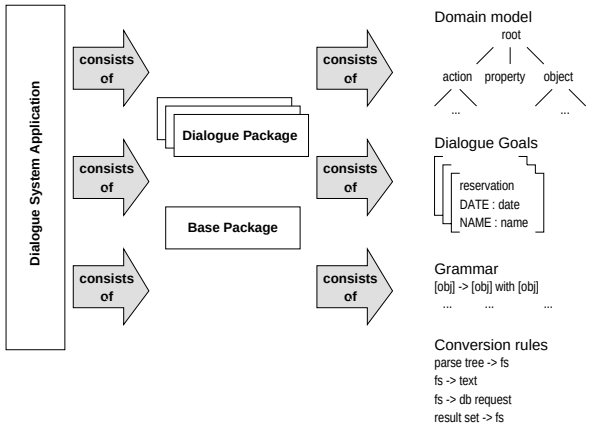

Figure 2: The object model for the dialogue system.

as required by JSapi and Sapi). Figure 3 depicts the relationship between the different entities.

The data source specifications are organized in a modular fashion in dialogue packages. Each dialogue package consists of at least one and possibly all the mentioned data sources. A final application is then composed of several dialogue packages. In order to avoid naming conflicts, each dialogue package introduces its own namespace. As an example of a dialogue package, consider the task of a hotel reservation. The implementation of the hotel reservation package may contain several tasks, such as calculating the price of a stay or displaying the hotel's location on a map. The interface between the implementation of the package and the dialogue system is regulated by the knowledge sources in the package description. For example, the hotel reservation package may consist of several concepts such as hotel, room, reservation, and all possible actions that go with it. The dialogue system notifies the dialogue package in case an event related to the package occurs. It is then the responsibility of the dialogue package to process the event properly.

Similar to class libraries in object-oriented programming languages, the dialogue packages may be reused in different applications. The hotel reservation package may be reused in an information booth application (which uses another dialogue package concurrently offering services related to current events) and in a travel agency setting (which, in turn, allows the user to book flights through the use of a third dialogue package). The intention of this level of granularity is it to have each package cover all aspects of an entire subdomain.

3 The Specifications

The IDE offers a wizard-style GUI to specify the data sources described above. The wizard guides the user through the process of specifying a dialogue package. In this section, we describe the steps the

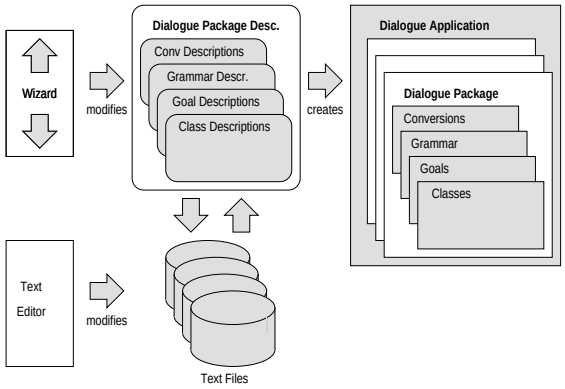

Figure 3: Descriptions specify the content of the data sources to be used in the application. Descriptions of data sources can be created and modified using the IDE or a plain text editor.

wizard guides a user through for each dialogue package. In each step, one of the four knowledge sources described above is interactively specified.

3.1 The Domain Model Specification

The domain model employed in the dialogue system uses a simple *class hierarchy*. A class hierarchy is a type hierarchy [Carpenter, 1992] extended by method descriptions. Class specifications may contain variables (whose type is a class from the ontology) and methods (whose arguments are classes from the ontology). In addition, class specifications may be related through multiple inheritance. While in conventional object-oriented design, objects in the domain correspond to classes, actions of the objects correspond to methods, and properties correspond to variables, we chose to model these elements by classes. First, this allows us to uniformly express mappings from noun phrases, verbal phrases and adjuncts to classes (see section 3.3). Second, any constituent of a spoken utterance may be underspecified. Our approach allows us to select through the inheritance mechanism the most specific class from the ontology whose informational content can be warranted in the absence of complete information.

A method specification does not implement any particular behavior of the class it belongs to. Rather, it can be seen as a constraint specification that generates an event to the back-end application as soon as it is satisfied. It is then the task of the back-end application to carry out the functionality associated with the method. Consider a class obj_displayable with an associated method display() and the constraint string $<$ obj_displayable.name,int $<$ obj_displayable.x,int $<$ obj_displayable.y (read: the variable obj_displayable.name contains more information than the fact that it is a string,i.e. it is instantiated). As soon as the position and the name of the object become known to the dialogue system (e.g. through database retrieval), an event is generated and sent to the implementation of the dialogue package, providing class information as well as the values of the three variables expressed in the constraint. Should a description of an object refer ambiguously, an event is generated for each retrieved object that verifies the constraint. Not only does this approach provide a declarative way of specifying behavior and abstract over the form of the dialogue, it also decouples the natural language understanding component from the application itself in a natural way.

This form of method invocation interacts nicely with another characteristic of our approach to object-oriented design. While traditionally an instance of a class is an object, in dialogue processing an instance of a class can only be a (possibly incomplete) description of an object. Necessary information for object instantiation may be missing and can only be acquired through dialogue. Since descriptions of objects do not need to refer uniquely to objects, procedural method invocations become more complicated. For this reason, we chose the declarative approach to method invocation over a procedural one.

The domain model is the backbone of the specification process. Not only does the dialogue manager use the domain model for inferences at runtime, but other knowledge sources such as grammar and database access specifications can partly be derived from the domain model. Moreover, the type information helps to restrict choices and to verify the consistency of the specification at the design stage. Consequently, the design of the domain model is the first step in the design process. This is in contrast to many other design tools whose first step is to design the information flow of the dialogue.

Example

By way of an example, we describe the design of a fast food order service. The service offers pizza with different toppings and different pasta with different sauces. Pizzas and pasta come in different sizes. The price of the items varies as a function of the size and the toppings or the sauce, respectively. The user should be able to query properties of the items, such as price, add and remove items from a virtual shopping list, and finalize the purchase. We introduce one abstract base type *obj_priceable* with the real valued feature BASEPRICE and a feature SIZE, the value being one of *small, medium, large*. As toppings and sauces may not be purchased separately, a second abstract base type *obj_buyable*, inheriting from *obj_priceable*, allows to distinguish the dishes from its ingredients. *obj_buyable* serves then as a base type for *obj_pizza* and *obj_pasta* while *obj_topping* and *obj_sauce* are derived from *obj_priceable*. As the calculation of the price is a task specific to the back-

end application, we introduce a method

$$obj\_buyable.calcprice : real \times set(real) \times real$$

with the constraints

$$\begin{array}{lcl} obj\_buyable.baseprice & > & real, \\ obj\_buyable.ingredients.\{baseprice\} & > & real, \\ obj\_buyable.price & \geq & real \end{array}$$

As soon as an *obj_buyable* whose values of the BASEPRICE features is defined appears in the discourse, all values are passed on to the back-end application. It is the task of the back-end application to determine the price of the dish and to return the result in the third argument of the method description. Since the third argument is described by the constraint $obj\_buyable.price \geq real$ (a constraint that is always satisfied due to the feature definition), the dialogue manager places the result returned from the back-end application at the appropriate place in the feature structure.

3.2 The Dialogue Goal Specifications

The application designer needs to design a description of a dialogue goal for each task the back-end system can execute. A dialogue goal can be considered as the description of a form that is filled out through the spoken dialogue with the system [Papineni et al, 1999]. The goal description consists of a typed feature structure [Carpenter, 1992] whose types are drawn from the class hierarchy designed in step 3.1. It serves as an informational lower bound, guaranteeing that the back-end application is notified if and only if the information acquired through the dialogue is at least as specific as the specification in the dialogue goal.

Note that the dialogue goal specification does not make any assumptions as to how this information is acquired, nor as to how the acquired information is to be processed. Thus, the dialogue goals form the specification of a task model that is orthogonal to any dialogue strategy specification and independent from the implementation of the back-end system. Furthermore, it should be noted that the specification of dialogue goals in typed feature structures does not restrict the dialogue strategy to be a simple form filling strategy. Rather, the dialogue goal specification is an encapsulation of a method invocation which, when triggered, causes the back-end application to do what the user intended the system to do. The assumptions made here are similar to those in the general PARADISE framework [Walker et al, 1997] for dialogue evaluation where the task model for dialogue managers is equally described in attribute value matrices.

Example (continued)

We continue the fast food service example. We concentrate on the dialogue goals relevant to the pizza and pasta objects, as we assume that we have recourse to a dialogue package *Shopping Cart* that defines the knowledge sources relevant to the virtual shopping list. We thus need to introduce only one dialogue goal, namely the one allowing the user to seek information on the buyable objects.

3.3 The Grammar Specification

It is the task of the grammar specification to map an utterance onto a feature structure. We use the robust spoken language parser described in [Gavalda and Waibel, 1998] for context free parsing. In addition to the grammar rule specification, a set of conversion rules needs to be created to declare the way a parse tree is mapped onto a semantic representation. A parse tree generated by this parser contains semantic concepts as nonterminal symbols.

Grammar rules can be either lexical rules, i.e. rules whose right hand side consists entirely of lexical entries, or phrasal rules, i.e. rules whose right hand side consists entirely of nonterminal symbols. A grammar nonterminal symbol consists of three part $\langle sem, syn_{maj}, syn_{min} \rangle$ where sem is a type drawn from the type hierarchy, syn_{maj} is the name of the major syntactic category, currently one of N, V, A or their phrasal projections NP, AP, VP, and syn_{min} is the name of their minor syntactic category. Minor categories depend on the major categories. For example, minor categories for adjectives are *predicative*, *comparative* and *superlative*. The purpose of separating syntactic and semantic information in the nonterminal symbols is threefold. First, it allows the technique of multiple inheritance to be applied during grammar design and parsing. For example, a nonterminal symbol $\langle sem, syn_{maj}, syn_{min} \rangle$ might be expanded by a rule with a left hand symbol $\langle sem', syn_{maj}, syn_{min} \rangle$, provided that sem subsumes sem' in the type hierarchy. Second, it provides more information to compare nonterminal symbols during parsing than plain slot names. Third, the semantic information is helpful in ensuring the semantic constructions associated with the grammar rules is well-typed. Please refer to [Denecke, 2000] for more information on the first two points. In this paper, we will concentrate on the third point as it is relevant to the design of the wizard interface.

As the syntactic structure of the input sentences might vary, it is not sufficient to rely on the names of the concept to extract the meaning of the utterances. Rather, we pursue an approach that is resembles the one found in *attributed grammars* used in compiler construction or *Montague grammars* in that the grammar rules contain an annotation describing how to construct the semantics. Consider a

rule

$$\langle sem, syn_{maj}, syn_{min}\rangle \rightarrow \langle sem^1, syn^1_{maj}, syn^1_{min}\rangle \ldots \langle sem^n, syn^n_{maj}, syn^n_{min}\rangle$$

for an expression describing an object of type sem. We assume by induction that the constituents described by $\langle sem^i, syn^i_{maj}, syn^i_{min}\rangle$ are expressions describing objects of type sem^i. As the semantic representation of the phrases covered by $\langle sem, syn_{maj}, syn_{min}\rangle$ needs to be a feature structure of type sem, all that remains to be done is to define n feature paths $\pi^i = f^i_1 \ldots f^i_{m_i}$ for each of the right hand symbols such that $sem.\pi^i$ is allowable according to the type hierarchy specification and $sem.\pi^i$ takes a value that is compatible with sem^i. This sort of type information restricts the number of possible feature paths. Only allowable feature paths are offered through the wizard interface so as to ensure that the resulting structures correspond to the domain model.

As an application designer sets out to develop a new application, he can take recourse to a base ontology and a base grammar. We make the assumptions that the base grammar and the base ontology already cover a wide variety of surface forms of the input sentences. The application designer simply needs to provide the lexical rules and to specialize existing generic rules. The nonterminals in the base grammar do not contain any domain-specific semantic information, but only rather general information such as object, or location. It is then only necessary for the application designer to specialize the predefined rules and to provide the "ontological" part of the grammar.

Robust Parsing

The fact that syntactic and semantic information are represented separately in the nonterminal symbols enables a more fine grained comparison of nonterminal symbols. This can be exploited for robust parsing. For example, two symbols differing only in their minor syntactic category could be matched, with an appropriate penalty, to allow for robust parsing. At the time of writing, a standard context free grammar to be used in the parser is created from the rule specifications. Additional rules covering close matches are created for robustness.

The well-typed constraint imposed on the rules by the conversion information does not render the parsing more brittle as robustness is achieved by loosely matching the input and by a fuzzy matching of nonterminal symbols. The form of the rules can be expected to be unaltered.

Clarification Questions

The need to generate a clarification question arises in the case of ambiguous reference. The dialogue manager determines discriminating information of a set of representations using a technique described in [Denecke and Waibel, 1997]. As the grammar rules contain syntactic and semantic information, they are reversible to a limited extent. Thus, the rules can be used to generate phrases describing the discriminating information.

Example (continued)

In the fast food application, phrases such as *a pizza with salami* or *tortellini with cream sauce* need to be covered. The generic grammar provides an abstract rule of the form $\langle obj, N\rangle \rightarrow \langle obj, N\rangle\langle p, with\rangle\langle obj, N\rangle$ which is specialized to

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle\langle p, with\rangle\langle obj\_topping, N\rangle$ and
$\langle obj\_pasta, N\rangle \rightarrow$
$\langle obj\_pasta, N\rangle\langle p, with\rangle\langle obj\_sauce, N\rangle$

respectively (minor categories are omitted for clarity). Each nonterminal symbol on the right hand side is assigned a part of the resulting semantic representation. The first right hand symbol gets assigned an empty feature path, since its relation to the left hand symbol relation needs to be an $\mathsf{is-a}$ relation. The semantics of the second nonterminal symbol is ignored. We concentrate on the third nonterminal symbol in both rules. In this example, TOPPINGS and SAUCE, respectively, are the only feature paths that express an $\mathsf{is-part-of}$ relation between *obj_pizza* and *obj_toppings*, and *obj_pasta* and *obj_sauce*, respectively. This yields the following annotated rules.

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle$ [*obj_pizza*]
$\langle p, with\rangle$
$\langle obj\_topping, N\rangle$ [*obj_pizza* TOP'S *obj_topping*]
and
$\langle obj\_pasta, N\rangle \rightarrow$
$\langle obj\_pasta, N\rangle$ [*obj_pasta*]
$\langle p, with\rangle$
$\langle obj\_sauce, N\rangle$ [*obj_pasta* SAUCE *obj_sauce*]

The type information serves to restrict the number of admissible feature paths for the semantic construction. Only admissible feature paths are offered as choices in the wizard, thus reducing the burden on the grammar designer. Had the designer erroneously specialized the abstract rule to

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle\langle p, with\rangle\langle obj\_sauce, N\rangle$

the wizard would not be able to offer any consistent semantic interpretation, thus uncovering inconsistencies in the specification early in the design process.

3.4 The Database Access Conversion Rules

The IDE provides an interface to SQL databases. The tables of SQL databases are self-describing in

that the form, the datatypes and the relations between the tables can be determined at run-time. If the user wishes to create a new database for some of the objects specified in step 3.1, then the corresponding SQL data definition query is generated from the domain model automatically. In this case, there is a one-to-one relationship between a type description and a table, and conversion rules are created automatically. However, it is more probable that application designers are faced with the design requirement that existing databases be reused. In this case, the wizard interface allows the user to establish a conversion between features and entries in tables. Please note that in this case there is not necessarily a one-to-one correspondence between type descriptions and tables. Here, the databases consist typically of multiple tables T that are linked via primary keys E.

The dialogue strategy executes database requests at appropriate times during the dialogue with the goal being to fill in missing feature values. It is then the responsibility of the database manager to determine the database that needs to be queried and to generate the query itself based on the information available. This is done in the following manner. First, by examining the partly filled form and scanning the conversion rules, the set of tables $t_1^1, \ldots, t_n^1 \in T_1$ for which keys are given are determined. Then, we need to obtain all pairs of primary keys that establish the links between the tables in T_1. However, a link between two tables can be given through a chain of tables not all of which need to be in T_1. Thus, we need to determine the set $t_1^2, \ldots, t_m^2 \in T_2$ of all tables involved in the query by calculating a minimal subtree $\langle T_2, E_2 \rangle$ of the graph $\langle T, E \rangle$ that spans over all tables from T_1. The information in T_1, T_2 and E_2 together with the partially filled form is then sufficient to arrive at a query of the form

SELECT
$t_1^2.e_1, \quad \cdots, \quad t_1^2.e_{n_1}$
$\vdots \qquad\qquad \vdots$
$t_m^2.e_1, \quad \cdots, \quad t_m^2.e_{n_m}$
FROM $t_1^2, \ldots, t_m^2$
WHERE
$t_1^1.e_1 \quad = \quad v_1$ AND
$\vdots \qquad\qquad \vdots$
$t_1^n.e_p \quad = \quad v_p$ AND
$t_i^2.e_k \quad = \quad t_j^2.e_l$ AND $\quad \forall \langle t_i^2.e_k, t_j^2.e_l \rangle \in E_2$

where the v_i are the values provided by the partly filled form. The result set returned from the query engine is then converted back to feature structures corresponding to the domain model. There exist additional constraints on the size of the result set that are verified before converting in order to avoid time consuming conversion operations in the case of large result sets.

Example (continued)

In the fast food application, the data is stored in four tables, namely *pizza*, *pasta*, *sauces* and *toppings*. The tables are assigned to the types *obj_pizza*, *obj_pasta*, *obj_sauces* and *obj_toppings* in the same order; additional assignments exist between feature names and table entries. The tables *pizza* and *toppings*, and *pasta* and *sauces*, respectively, are linked in the database through unique IDs. As the relationships between the tables is is-part-of, the links are assigned the path prefixes SAUCES and TOPPINGS. A feature structure

$$\begin{bmatrix} obj\_pasta & \\ \text{SAUCE} & obj\_cheesesauce \end{bmatrix}$$

is then converted to the query

SELECT
pasta.name, *pasta.baseprice*, *pasta.size*,
sauces.name, *sauces.baseprice*
FROM *pasta*, *sauces*
WHERE
sauces = *cheesesauce* AND
pasta.ID = *sauce.ID*

Using the same conversion rules backwards, underspecified feature structures are constructed from the resulting table. Note that the database as a relational database cannot express inheritance relationships. This means that although *tortellini*, *greennoodles* and *spaghetti* all are derived from *pasta*, a query containing the constraint *pasta.name* = "*pasta*" would return the empty set, as the database does not know about the inheritance relationships. For this reason, the conversion rules associated with the table entries also contain a type restricting the constraint generation. Only types that are more specific than the restriction are taken into consideration for query generation. In this example, the types taken into consideration for query generation would need to be more specific than *obj_pasta*. This is to ensure extensionality for database access. Alternatively, one could employ extensional feature structures as described by Carpenter [Carpenter, 1992] and make sure that only extensional types are used for queries.

3.5 Interfacing the Wizard with the Knowledge Sources

A wizard-style GUI guides the application designer through the design process of the dialogue package. The knowledge sources are introduced in the order in which they are described in this section. The result of the process is a prototypical system that needs to be refined interactively using test sets. Figure 4 shows a screenshot of the wizard in step 1 at the point of specifying the domain model.

In order to abstract over different input and output modalities, the dialogue system contains an entity to maintain input and output channels. For each channel, there is a channel specification that allows

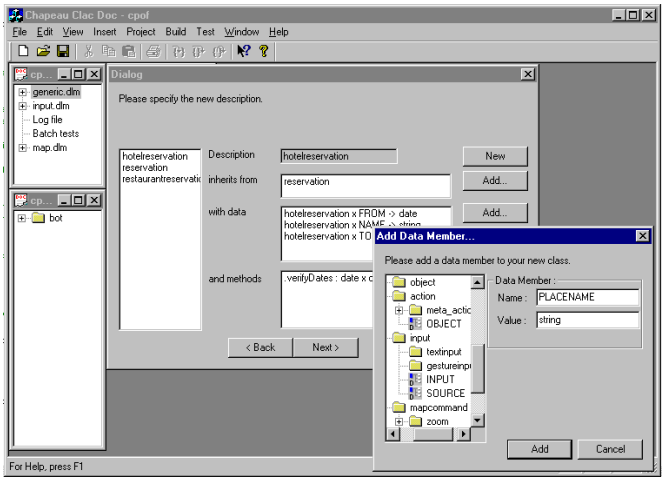

Figure 4: The wizard in action. Currently, a class hotelreservation is being specified. The list boxes in the larger dialog box display the base class, the member variables and the methods associated with the class. A new member variable is being added through the smaller dialog box in the foreground. The tree-shaped interface item provides a view on the domain model.

to transform an array of strings into a feature structure (for an input channel) or a feature structure into an array of strings (for an output channel). Input and output devices communicate with the dialogue system only through these channels. The intention of this approach is it to abstract away the particular form of input and output events, thus achieving modularity and extensibility.

4 Debugging of the Dialogue Strategy

The dialogue manager is driven by a PROLOG style program which contains the dialogue strategy. As long as a user is engaged with the system in a dialogue, it is then the task of the dialogue system

1. to determine if the user intends to have the system perform one of the tasks known to the system, and if so,
2. to interactively acquire all the information that is needed for the system to uniquely determine the task to be executed and all its parameters, and
3. finally to notify and pass control to the subsystem responsible for the task execution once this state has been reached.

For that purpose, each task description has an internal state that can take one of the following values: NEUTRAL, SELECTED, DESELECTED, DETERMINED and FINALIZED. The state transitions are as shown in figure 5. Each state transition is passed on to the implementation of the dialogue package in the

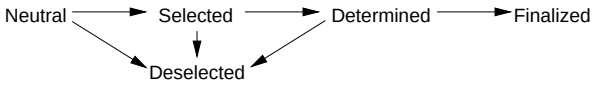

Figure 5: State transitions of the dialogue goals

back-end application which may or may not choose to make use of this information.

The state of the dialogue system is implicitly represented by the vector of dialogue goal states. The states of the dialogue goals are updated by a set of rules that compare the representations of the utterances with the representations in the dialogue goals. The state of a goal incompatible with the current representation becomes DESELECTED. A goal in the state SELECTED becomes DETERMINED as soon as it is the only goal in the SELECTED state. A DETERMINED goal becomes FINALIZED as soon as the information acquired in the dialogue is at least as specific as it is required by the goal.

There is a generic dialogue strategy that serves as a starting point for system development. As possible domains may be very distinct, it becomes necessary to adapt the strategy to the domain at hand. For this reason, the IDE offers an interactive debugger interface to the rule program. It allows for single step execution, display of call stacks and variable substitutions as well as a direct query interface to evaluate the effects of single rules.

5 Testing

The cycle of grammar maintenance, testing and evaluation is a tedious and time consuming part of the development of a new application. The IDE offers a set of utilities simplifying the task.

5.1 Batch Testing

Grammar Testing

The IDE offers a tool for batch testing of grammar coverage. Here, a text string is passed through the semantic parser and conversion routine. The resulting feature structure is then presented graphically to the user. The designer is then prompted to evaluate the semantic representation of the utterance. Current choices are those defined by the partial order of feature structures. In other words, the system's designer can specify if the semantic content contains information that is equal to, less specific than, more specific than or inconsistent with the information the sentence conveys. The text string, the feature structure and the evaluation are then automatically entered to the batch test set. The system designer can then run this batch test set later in the development process and receive notification should the resulting feature structures differ in informational content. This procedure assumes, however, that the domain model is not changed between the tests. Alternatively, the system designer can enter the desired

feature structure directly.

Testing for Goal State Transitions

In addition to the grammar coverage batch test, there is a dialogue goal batch test. As mentioned above, the state of the dialogue manager is implicitly described by the vector of goal states. Each utterance is assumed to represent a speech act that performs a state transition in some of the dialogue goals. Here, we store together with the utterance two vectors of dialogue goal states: before the utterance has been processed and after the utterance has been processed. During batch testing, the dialogue goals are set to the states specified in the first vector. Subsequently, the utterance is passed through the dialogue system. Then the actual goal states after processing of the utterance are compared with those in the batch test and differences are prompted to the application designer.

Testing for Orthogonality between Modules

Testing for dialogue goal state transitions requires the configuration of dialogue packages to be constant between tests. However, there are several utterances whose meaning can unambiguously be attributed to one dialogue package. For this reason, the IDE offers an additional batch test. Here, the utterances are assigned a dialogue package as well as vectors of goal states. In contrast to the state transition test, we only represent goal states from dialogue goals in the package in question. As above, the application developer is notified if the desired goal configuration in the package differs from the calculated one. Moreover, any goals not in the assigned dialogue package whose state differs from DESELECTED are displayed to the user.

5.2 Dialogue Goal Activation and WOZ

Since the IDE contains a detailed description of the dialogue goals, it is possible to present the dialogue goals to the application designer in form that needs to be filled in through the standard graphical user interface rather than through speech. Once the back-end application is in place, the application designer may proceed to test the interface of the dialogue system with the back-end application. Another possibility would be to use this feature as a poor man's Wizard of Oz interface, in which case only the domain model and the task model need to be in place (although additional support from the database would be desirable). This feature is currently under development.

6 Discussion

We are currently using the described system to prototype two spoken language applications. While it is too early to arrive at any conclusive results, our preliminary experience shows that a substantial amount of time is saved simply by using the wizard to avoid formatting errors and typographic errors in the several specification files. Moreover, as the wizard displays the options available for the user to choose from, it is easier to arrive at consistent specifications. This is particularly true in the instances where type information from the domain model can be used to reduce the number of options.

Another characteristic of the system is its integrated architecture. The entire system runs as a single thread in a single process. Comparing to an earlier version of the system in which a client/server architecture was employed, we find debugging and testing easier.

From a domain model perspective, the dialogue packages as a primary building block offer a coarse granularity compared to dialogue states, speech objects or dialogue libraries. We feel it is for this reason more comprehensive. Whether this characteristic is of benefit and whether the specifications in the different packages are sufficiently orthogonal to not interact when building the final system remains to be seen.

Although the specifications of knowledge sources in separate modules can be independent of each other, undesired interaction may not be excluded. In particular, the informational content of the dialogue goal specifications need pairwise inconsistent. The reason is that the dialogue manager bases its decision on the compatibility of the dialogue goals with the information in the discourse. If one dialogue goal were less specific than another, the second dialogue goal could never be reached as the first is satisfied first. For this reason, the dialogue manager checks for pairwise inconsistency of the goals at runtime.

Future work includes the integration of a speech recognizer directly into the development environment and improvements of the graphical user interface to speed up the design process. These improvements can only be made by experiences gained through continuous use of the wizard.

Acknowledgements

This research is supported by the Defense Advanced Research Projects Agency under contract number DAAD17-99-C-0061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the DARPA, or any other party. I would like to thank the members of the dialogue working group in the DARPA *Command Post of the Future* project for valuable discussions.

References

M. Araki, K. Komatani, T. Hirata and S. Doshita. *A Dialogue Library for Task-Oriented Spoken Dialogue Systems* Workshop on Knowledge and Reasoning in Practical Dialogue Sys-

tems. Stockholm, Sweden, 1999. Available from http://www.ida.liu.se/ext/etai.

B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, 1992.

R. Cole. *Tools for Research and Education in Speech Science*. Proceedings of the International Conference of Phonetic Sciences, San Francisco, USA, 1999.

M. Denecke and A. Waibel, *Dialogue Strategies Guiding Users to Their Communicative Goals*. Proceedings of Eurospeech, Rhodos, Greece,1997. Available from http://www.is.cs.cmu.edu.

M. Denecke and A.H. Waibel, *Integrating Knowledge Sources for a Task-Oriented Dialogue System*. Workshop on Knowledge and Reasoning in Practical Dialogue Systems, Stockholm, Sweden,1999 Available from http://www.is.cs.cmu.edu.

M. Denecke. *Modularity in Grammar and Ontology Specification*. Proceedings of the MSC 2000 Workshop, Kyoto, 2000. Available from http://www.is.cs.cmu.edu.

M. Gavalda and A. Waibel. *Growing Semantic Grammars*. Proceedings of the COLING/ACL, Montreal, Canada. Available from http://www.is.cs.cmu.edu.

Anke Koelzer. *Universal Dialogue Specification for Conversational Systems* Workshop on Knowledge and Reasoning in Practical Dialogue Systems. Stockholm, Sweden, 1999. Available from http://www.ida.liu.se/ext/etai.

K.A. Papineni, S. Roukos and R.T. Ward. *Free-Flow Dialogue Management Using Forms*. Proceedings of EUROSPEECH 99, Budapest, Ungarn, 1999.

S. Sutton, D. G. Novick, R. A. Cole, and M. Fanty. *Building 10,000 spoken-dialogue systems*. Proceedings of the International Conference on Spoken Language Processing, Philadelphia, PA, October 1996.

Walker, M.A. and Litman, D.J. and Kamm, C.A. and Abella, A. *PARADISE: A Framework for Evaluating Spoken Dialogue Agents* Proceedings of the 35th Annual Meeting of the Association of Computational Linguistics, 1997. Available from http://www.research.att.com/ walker.

A Rational Agent for the Construction of a Semantic Model*

PAUTRET Vincent
Université de Rennes I, ENSSAT
6 rue de Kérampont, BP 447
Lannion, France, 22300
Vincent.Pautret@enssat.fr

Abstract

This paper presents a methodology that aims at building knowledge models from a natural language description of a domain. Our methodology is based on the establishment of a dialogue with the knowledge engineer of an application. This dialogue is motivated by the Semantic Differentiation Process, which solves problems related to acquisition and modelling.

Moreover, the dialogue can be naturally formalised within a theory of communicating rational agents. We can thus consider a more complete automation of the process of modelling and show how to integrate our methodology into this type of theory.

Introduction

Knowledge Based Systems separate the semantic model - which handles the system knowledge - from the reasoning process - which uses this knowledge. The main advantage of this approach is that only the semantic model has to be changed to handle a different application domain. However, the creation of a semantic model for a given application is a manual process, which is difficult to automate (Paris and Vander Linden (1996)).

Tools (Heijst et al. (1997)) or workbenches ((Mikheev and Finch (1995), (Delisle (1996)) already exist that aim at building semantic representations at the domain level (using the vocabulary of KADS (Wielinga et al. (1992)). With these tools and workbenches, conceptual knowledge models (like ontologies) independent of the application domain are built. However, the knowledge engineer task remains fastidious. One of the difficulties in completely automating the acquisition and modelling process comes from a lack of interaction with the knowledge engineer.

In order to improve these interactions (and thus to facilitate modelling), we propose a methodology based on a natural language dialogue with the knowledge engineer. This methodology can be implemented into a rational agent. In this way, this agent is given capabilities of modelling by means of conceptual diagrams defined in our methodology. We show how to make this integration within the formal theory of communicating rational agents of Sadek (Sadek (1991), (Sadek et al. (1997))).

Section 1 introduces the bases of the methodology. Section 2 explains how to integrate it into a theory of rational agents for its effective implementation. The last section presents the guidelines to implement our methodology into a rational agent.

1 Bases of the methodology

The methodology aims at building a semantic model of a domain from a natural language description. It is based on three successive stages: the acquisition stage, the modelling stage, and the transfer stage.

The acquisition stage consists of the analysis of each domain description utterance. A morpho-syntactic analysis is followed by a semantic analysis in order to build a semantic representation of each utterance. In an iterative way, these representations are integrated into a general model: the Construction Model (CM).

The modelling stage consists of an interactive reorganisation of the CM once the description process is completed.

The transfer stage extracts the relevant information from the CM and builds the semantic domain model.

* This work was realised within the framework of a PhD in France Télécom R&D.

1.1 Construction Model

On the one hand, Construction Model must have a sufficient expressiveness, which makes it possible to represent domain knowledge and knowledge related to its own structure at the same time. On the other hand, it must have a flexible enough structure, which can be handled simply and efficiently.
We use a formal language based on KL-ONE-like description logic. The central part of the model is a semantic network whose nodes are concepts and whose arcs are semantic domain or modelling relations (for example *subconcept*, *composition*, *property*, etc.) The representation language also offers the possibility to express *abstract* concepts (as a composition of concepts and relations of the network), as well as constraints and negative knowledge related to the concepts and relations of the network.

1.2 Basic tools for knowledge acquisition

During the modelling process, which is based on dialogue, the knowledge engineer utterances are analysed and the relevant information has to be extracted from them. For this purpose, we use two tools to acquire knowledge from texts.
The first tool is a robust morpho-syntactic analyser, which produces a syntagmatic graph (Giguet (1998)) where each node is a syntagm and each relation is a syntactic relation. A syntagmatic graph is produced for each utterance of the description.
The semantic tool makes use of the results of the morpho-syntactic tool to produce a semantic representation of each utterance. Thanks to four basic operations, it integrates this representation into CM. The first operation identifies the concepts that are already known. The second one is related to generalisation and organises the concepts into hierarchies. The third one calculates the common characteristics to the concepts. Finally, the last one places the semantic relations resulting from the semantic analysis into CM.

1.3 Semantic Differentiation Process

The Semantic Differentiation Process is based on a set of generic conceptual diagrams, whose role is to modify the CM structure. We follow an empirical process to exhibit modelling problems and to define a conceptual diagram as a solution to each one.

An initial situation and several final situations define a conceptual diagram. Situations are expressed in terms of the language of CM representation, namely as sets of first order logic formulae. A situation corresponds to a particular structuring of generic concepts and generic relationships between these concepts. A condition is associated with each final situation. The conceptual diagrams are represented in the following form (we will use more readily a chart of the initial and final situations of a diagram as on the example of figure 1).

Name_of_the_diagram
<*Initial_situation*>
<*condition_1*> <*Final_situation_1*>
…
<*condition_n*> <*Final_situation_n*>

where *Initial_situation* and *Final_situation_k* ($k \in \{1,\ldots,n\}$) refer to the initial situation and the *n* final situations associated with the diagram, and *condition_k* ($k \in \{1,\ldots,n\}$) refer to the condition associated with *Final_situation_k*.

Diagrams are divided into three main families. The first family (two diagrams) is dedicated to the integration problems. The second one (seven diagrams) allows model simplifications while the third one (eleven diagrams) allows modifications of the model structure. The diagrams constituting the first family are applied during the acquisition stage while those of the two other families are applied during the modelling stage. The diagrams are ordered according to the importance of the modifications they produce on the model. For example, the simplification diagrams are applied before the modelling ones.
A diagram can only be used once the model under development has validated the initial situation. When the initial situation has been instantiated, a dialogue begins with the knowledge engineer until one of the conditions associated with each final situation is validated. CM is then restructured to resemble the final situation, which corresponds to the condition. The role of this dialogue is to determine the best transformation of the model by the considered conceptual diagram for the problem under consideration. An algorithm of processing of graphs carries out the passage from the initial situation to the selected final

situation, which directly removes assertions from the model or adds some to it.

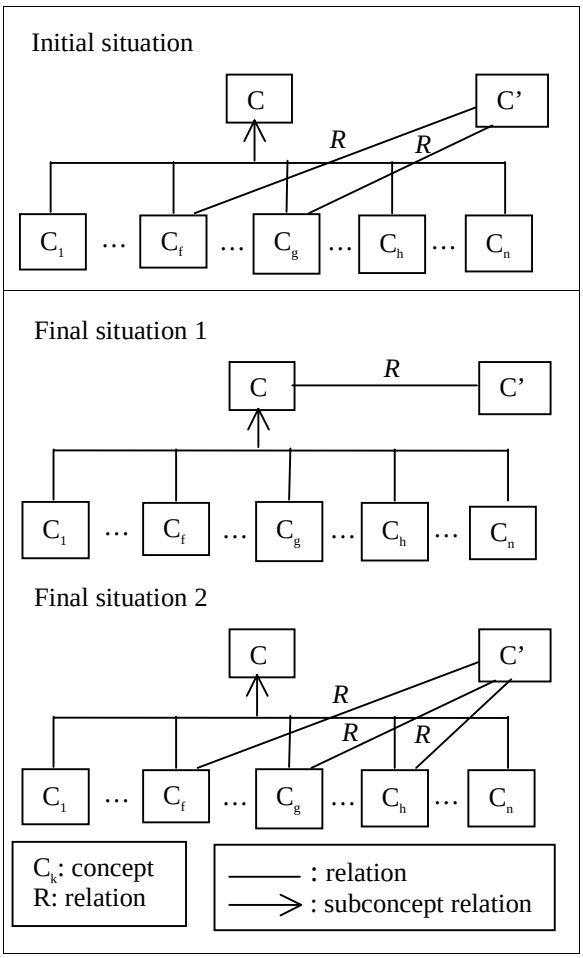

Figure 1: Factorisation diagram

Figure 1 shows the *Factorisation diagram*, which belongs to the second family of diagrams. The role of this diagram is to factorise a relation from the subconcepts to their supconcept (final situation 1) or to add relations, which would have been forgotten by the knowledge engineer (final situation 2). The first case makes it possible to reduce the number of relations and thus the complexity of the model. With the second one, supplements can be added to the model after missing information has been detected.

The initial situation shows several subconcepts C_f, …, C_g of concept *C*, which have the same relation *R* on the same concept *C'*.

In final situation 1, relation *R* is placed on concept *C*, whereas in final situation 2, relation *R* is extended between *C'* and some of the subconcepts of *C* (concepts C_f, …, C_g, …, C_h).

When the initial situation is detected in the model, the evolution of the model is determined thanks to the following dialogue with the knowledge engineer:

Q1 – Subconcepts C_f, …, C_g of C have the same relation R with C'. Have all subconcepts of C this relation with C'?

With a positive answer, the model is transformed like final situation 1 (by adding relation *R* on *C* and by removing *R* between subconcepts of *C* and *C'*). With a negative answer the dialogue proceeds as follows:

Q2 – What are the different concepts that have this relation R with concept C'?

The model then evolves to situation 2.

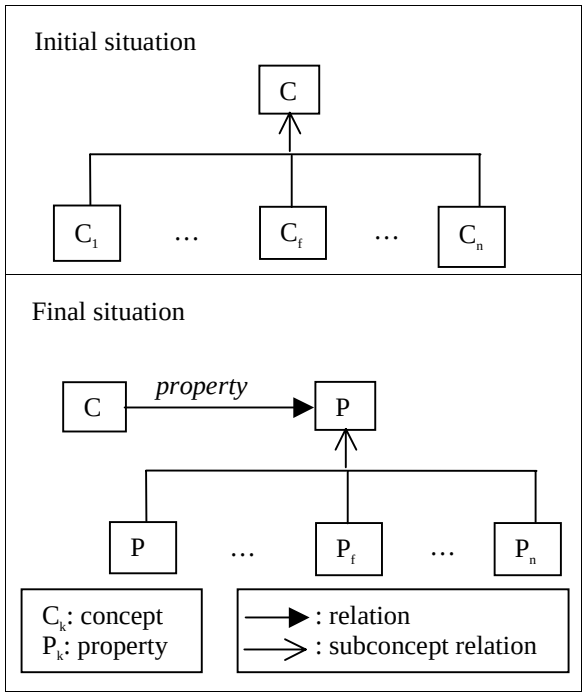

Figure 2: Property Extraction diagram

Figure 2 shows the *Property Extraction diagram*, which belongs to the third family of diagrams (modelling stage). It is intended to extract a particular property of a concept from its structure. From the structural viewpoint, a definition of a concept could be compared to the complex terms extracted by LEXTER (Bourigault (1994)). A concept comprises a word in head position (the headword) and a list of qualifications. Each qualification is recursively composed of a word and a list of qualifications. For example, *a specialised server for weather forecast* has the word *server* as headword and *weather forecast* and *specialised* as qualifications. In the context of

the diagram of figure 2, *weather forecast* is seen as a property of the concept.
Hence, if the system detects different qualifications of the concepts that have the same headword, the system asks the knowledge engineer whether it is possible to gather the qualifications under a same property. The dialogue associated to this diagram is very simple. The system only needs to ask the knowledge engineer if s/he wants to apply the diagram. If s/he answers positively, the system asks the knowledge engineer for the name of the property. Otherwise, the system does not modify the model.

1.4 A short example

Here is an example of dialogue in the AGS application, which is a vocal query, over the telephone, of the shared-revenue information voice services directory (applied to the areas of "employment" and "weather forecast") hosted by France Télécom. *K* stands for the knowledge engineer, *S* for the system, and N for note on the current modelling state.

K: *There exist servers for weather forecast, servers for employment and mail order servers.*
N: The system creates the concept "server" and places the three types of servers as subconcepts of this abstraction (by using the second basic operation, cf. section 1.2).

K: *The servers for weather forecast and the servers for employment have telephone numbers.*
K: *That is all.*
S: *O.K.*

S: *The servers for weather forecast and the servers for employment own a telephone number. Do mail-order servers own a telephone number as well?*
K: *Yes.*
S: *O.K.*
N: The system factorizes the relation on the concept "server" by applying the Factorization diagram.

S: *I detect some properties of the servers (weather forecast, employment, and mail order). Is it possible to group together these properties under the same concept?*
N: The system applies here the Property Extraction diagram.

K: *Yes.*
S: *Could you give the name of the concept?*
K: *Theme.*
S: *O.K.*
N: Then the system builds the final model.

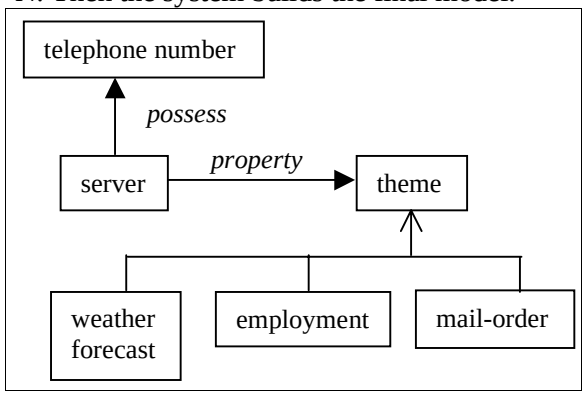

Figure 3: Example of a semantic model

2 Integration of the methodology into a rational agent

The core of our methodology is the dialogue with the knowledge engineer. The communicative rational agents provide a theoretical framework that is particularly adapted to the formalisation of this dialogue. In this way, we extend the theory of rational agents proposed by Sadek (Sadek (1991)), thus giving the agents the ability to build a semantic model of an application while following the interactive principles of our methodology.

2.1 The theory of rational agency

The whole theory of rational agency is expressed in a homogeneous multimodal logic of mental attitudes and actions (or events). Modal operator of belief B_i satisfies a *KD45*-model. The resulting agents are fully introspective and have consistent beliefs. Formula B_ip is read "property p is a logical consequence of the beliefs of agent *i*". The mental attitude of intention is defined as a complex combination of primitive mental attitudes like belief and choice[1], as a relatively similar way as Cohen and Levesque (1990). Formula I_ip is read "agent *i* intends to bring about proposition p".
In order to reason about action, two modal operators are introduced, *a* being an action expression and ϕ a formula: *Feasible*(a,ϕ) means that *a* can take place and if it does, ϕ

[1] For sake of simplicity, we only focus on belief and intention. For more details, see Sadek (1991).

will be true after that, and *Done*(a,ϕ) means that a has just taken place and ϕ was true before that.
The formal theory provides a set of axioms that specify rational agent behaviour in multi-agent environment and flexible behaviours according to the type of desired agent. For example, the first ones allow infer chains of actions corresponding to the agent intentions and the second ones generate cooperative reactions.

2.2 Primitive action of modelling

We propose to formalise conceptual diagrams with primitive actions of the theory of rational agency. Conceptual diagrams are then directly usable and can be planned by the agents like the other actions.

2.2.1 Action model

For each action to be planned, the classical components preconditions and effects are defined with the following meaning. Preconditions refer to the statements that must be true for the action to be performed. Effects refer to the statements that are intended to hold in the world following the performance of the action. Actions are represented in actions schemata:

<*actor*, Action(*parameters*)>
P: ϕ_1
E: ϕ_2

where *actor* refers to the agent of the action, *parameters* refers to eventual parameters of the action, ϕ_1 refers to the preconditions, and ϕ_2 refers to the effects.
For example, the communicative act of an agent *i informing* an agent *j* that a proposition ϕ holds is defined by the actions schema:

<*i*, Inform(*j*, ϕ)>
P: $B_i\phi \wedge \neg B_i(B_j\phi \vee B_j\neg\phi)$
E: $B_j\phi$

Thus an agent who achieves the act to inform of ϕ aims that *j* believes that ϕ is true. It cannot make it if it thinks itself ϕ true and does not think that *j* already has a belief on ϕ.

Among the axioms, which define the characteristics of an action in the theory, we exhibit the one, which refers to the preconditions[2]:

$$B_i(Feasible(a) \Leftrightarrow \phi_1) \quad (1)$$

where ϕ_1 refers to preconditions of *a*.

2.2.2 Actions and conceptual diagrams

In order to express conceptual diagrams of our methodology into the theory, we propose to associate with them the primitive actions, which make the transformation from their initial situations to each one of their final situations. In particular, this is made possible thanks to the fact that the situations can be described in a logical language. In this way, each one of these primitive actions corresponds to a modification of the mental state of the agent. We define actions associated with conceptual diagrams in the following way:

<*i*, Action_1>
P: $B_i(initial\_situation \wedge condition\_1)$
E: $B_i(final\_situation\_1)$

<*i*, Action_2>
P: $B_i(initial\_situation \wedge condition\_2)$
E: $B_i(final\_situation\_2)$

...

In this schemata, *i* refers to the agent (the modeller agent), which applies the methodology, *initial_situation*, *final_situation_1*, … refer to logical formulae associated with situations of a conceptual diagram, and *condition_1*, *condition_2*, … refer to logical formulae, which describe conditions for each final situation of the conceptual diagram.

2.2.3 Axioms for conceptual diagrams

Unlike actions defined in the theory of Sadek, the primitive actions that we associate with each conceptual diagram are not planned according to their effects. The conceptual diagrams are applied as long as possible. Therefore, their planning by the modeller agent does not depend on the goals it seeks, but rather on the current situation of its mental state, i.e. on its knowledge.
The following axioms schema corresponds to this strategy. As soon as an action can be

[2] *Feasible(a)* is the syntactic abbreviation of *Feasible(a,True)*.

applied, it must be done. The calculation of feasibility results from the axiom (1) of the theory.

$$B_i(Feasible(a)) \Rightarrow I_i Done(a) \qquad (2)$$

where *a* refers to a primitive action associated with a conceptual diagram.

Conditions associated with each alternative of a conceptual diagram are the source of the dialogue to apply the diagram (see section 1). We have to supplement the model of communicative behaviour of modeller agent in order to convey to it the capacity to initiate the dialogue with the knowledge engineer. This dialogue increases its knowledge until it can determine which alternative of the diagram (i.e. which primitive action) to apply. For each primitive action, we introduce an axioms schema of the following form:

$$B_i(initial\_situation) \wedge \phi \Rightarrow I_i \psi \qquad (3)$$

where *initial_situation* is associated with the corresponding conceptual diagram. ϕ (condition of release of the primitive action) and ψ (goal starting the corresponding dialogue) are expressed according to *condition_k* (condition associated with the primitive action).
For example, ϕ can be defined by $\neg Bif_i(condition\_k)$ and ψ by $Bif_i(condition\_k)$. $Bif_i\phi$ is a syntactic abbreviation defined in the theory and means that agent *i* knows if ϕ is true or not.
When several actions (resulting from the intentions derived by the axioms) are applicable in the same state, logic does not make it possible to choose the order in which they are applied. In an implementation, we need to be careful to follow the order defined by the methodology. This is achieved by a process of control of the inferences.

2.3 Example

We show the reasoning process related to the application of the factorisation diagram presented in section 1. The primitive actions associated to the diagram are:

<*i*, Facto_1>
P: $B_i(initial\_situation \wedge relation(R,C,C'))$
E: $B_i(final\_situation\_1)$

<*i*,Facto_2>
P: $B_i(initial\_situation \wedge \neg relation(R,C,C')) \wedge \wedge_{k\in\{1,\ldots,n\}} Bif_i(relation(R,C_k,C'))$
E: $B_i(final\_situation\_2)$

where *R*, *C*, C_1, …, C_n, and *C'* are the relations and the concepts identified in the conceptual diagram (cf. figure 1).

The two axioms schemata associated to the primitive actions are respectively[3]:

$$B_i(initial\_situation) \wedge \neg Bif_i(relation(R,C,C')) \Rightarrow I_i Bif_i(relation(R,C,C')) \qquad (4)$$

$$B_i(initial\_situation) \wedge B_i(\neg relation(R,C,C')) \wedge \neg[\wedge_{k\in\{1,\ldots,n\}} Bif_i(relation(R,C_k,C'))] \Rightarrow I_i Bref_i(\iota x\ relation(R,x,C') \wedge subconcept(x,C)) \qquad (5)$$

where *R*, *C*, and C' are the relations and concepts identified in the conceptual diagram (cf. figure 1).

Let us suppose that the modeller agent is configured in such a way that the conceptual diagram is applicable, i.e. $B_i(initial\_situation)$ can be derived from its mental state. Moreover, let us suppose that it can infer no knowledge in connection with *relation*(*R*, *C*, *C'*), i.e. it can only conclude ($Bif_i(relation(R, C, C'))$).
The instantiation of the axiom (4) thus generates the intention of the agent to know if the concepts *C* and *C'* are or not linked by *R*. Then the traditional mechanisms of planning of the theory take over (cf. Sadek (1991)) and produce an act of dialogue aiming at requiring missing information to the knowledge engineer (cf. dialogue *Q1* of section 1.3).
If the answer to this question is positive, the axioms of rational behaviour of the theory involve $B_i(relation(R, C, C'))$. All the preconditions of the primitive action Facto_1 are then checked and the axioms (1) and (2) induce the execution of the first alternative of the conceptual diagram (Facto_1). The resulting mental state of the agent conforms to the final situation 1.
If, on the contrary, the answer is negative, the agent acquires the knowledge: $B_i(\neg relation(R, C, C'))$. The axiom (5) then applies as long as the agent does not have a knowledge

[3] $Bref_i(\iota x\ \phi(x))$ means that agent *i* knows the objects, which check the property ϕ.

supplements on *relation*(R, C_k, C') for all $k \in \{1,..., n\}$. These produces the intention at the origin of the act of dialogue aiming at requiring of the knowledge engineer the whole of the subconcepts of C in relation R with C' (cf. dialogue *Q2* of section 1.3). The primitive action Facto_2 can then be carried out and leads to a mental state that conforms to the final situation 2.

3 Integration into an operational system

3.1 The Artimis technology

The Artimis technology of France Télécom R&D provides a generic framework to instantiate intelligent dialoguing agents. Such agents can interact cooperatively in natural language with human users.

Artimis software is composed of four main modules: a rational unit (which is the kernel of the system), a natural language interpretation unit, a natural language generation unit, and a domain knowledge management unit (Sadek et al. (1997), Sadek (1999)).
The rational unit conveys the agent the ability to dialogue and to reason about knowledge and action.
The natural language interpretation unit uses island-driven parsing and semantic completion (Sadek et al. (1997)). Island-driven parsing means that small syntactic structures in the text are spotted, with as few range dependencies as possible. The semantic completion builds a well-formed logical formula with the result of the parse.
The natural language generation unit verbalises dialogue acts produced by the rational unit. Finally, the domain knowledge management unit contains a representation of the domain knowledge. It provides several functions (like concepts identification) to have access to the knowledge.

Artimis software works in lab versions on several real applications like the AGS one. It is written in Quintus Prolog.

3.2 Guidelines for the integration into an Artimis dialogue agent

We present, in this section, the needed modifications in order to integrate our methodology into an Artimis dialogue agent.

3.2.1 Modification of the two natural language components

We have to modify the natural language interpretation unit at two levels.
Firstly, the system must take into account all the words of the utterance in order to detect the new concepts. We make the assumption that the sentences are syntactically and semantically correct.
Secondly, a robust syntactic analysis, based on an approach such as (Giguet (1998)) must be implemented to get as much information as possible on the relations between the concepts. In order to solve syntaxico-semantic ambiguities we introduce two particular relations: *unknown* and *context*. The *unknown* relation means that the analyser detects a relation but can not determine its exact nature. The relation *context* means that two concepts are present in the same utterance without any other information.
Example:
Input sentences:
There are servers and telephone numbers.
Results:
concept([server]) concept([telephone, number])
relation(context, [server], [telephone, number])

The natural language generation unit recovers the vocabulary necessary to the generation of sentences related to the domain thanks to the interpreter, which keeps the link between the concepts of CM and the vocabulary of the description.

3.2.2 Modification of the rational unit

In order to increase the reasoning capabilities of the rational unit so that it can direct the construction of the semantic model as well as the dialogue with the knowledge engineer, we add the logical axioms and the primitive actions that we defined in section 2. The rational unit should then not be rebuilt but rather updated.

3.2.3 Modification of the knowledge management unit

The main modifications concern this module.
We extend the language of representation of the model with the primitives of the CM.
In order to be able to insert a new knowledge in the model, we add the four basic operations.

The original identification function is modified to take into account the modification of the knowledge representation language.
The second function builds the hierarchies by using the structure of the concepts. For example, " server for employment " and "server for weather forecast" belong to the same hierarchy since they are two "servers". They are generalised by the concept "server".
The third function places relations between two concepts and their qualifying common part. For example, "server for employment" and "employment theme" have the common part "employment".
The last one is a transfer function between the result of interpretation and CM.

The organisation algorithm of the model tries to instantiate the initial situations of the diagrams in a definite order. This order depends of the priority associated to each diagram. The priority is given according to the transformations carried out by the diagram: the more significant the transformations, the weaker the priority. When a situation is validated, the corresponding formulae are injected into the rational unit. This one then takes over to calculate a question. Following the answer of the knowledge engineer, a new knowledge is asserted and the process starts again from the beginning. When no diagram is applicable, the algorithm stops and the knowledge engineer is provided with the model.

4 Conclusion

We define a methodology of semantic modelling of a domain. It is based on conceptual diagrams that formalise the incremental evolutions of the structure of the semantic model during its construction. A dialogue with the knowledge engineer directs the application of these diagrams.
We also formalise the use of our methodology within a theory of communicating rational agents. This specification provides the rational agent with new reasoning capabilities, which aim at building a semantic model by questioning the knowledge engineer and by applying the conceptual diagrams according to the principles of our methodology.
We thus open prospects for automation since effective agents implementing this type of theory are already operational like, for example, those resulting from Artimis technology (Sadek et al. (1997), Sadek (1999)).

References

Bourigault D. (1994) *Lexter, un logiciel d'Extraction de Terminologie. Application à l'acquisition à partir de textes*. École des Hautes Études en Sciences Sociales, Paris.

Cohen P.R. and Levesque H.J. (1990) Intention is choice with commitment, *Artificial Intelligence* 42(2-3):213-262.

Delisle S. (1996) Le Traitement Automatique du Langage Naturel au Service de l'Ingénieur de la Connaissance : le Système READER. *International Conference on Natural Language Processing and Industrial Applications*, Moncton (New-Brunswick, Canada).

Giguet E. (1998) *Méthode pour l'analyse automatique de structures formelles sur documents multilingues*. Thèse de Doctorat Informatique, Université de Caen, France.

Mikheev A. and Finch S. (1995) Towards a Workbench for Acquisition of Domain Knowledge from Natural Language. *Proceedings of EACL'95*, Dublin, Ireland.

Paris C. and Vander Linden K. (1996) Building Knowledge Bases for the Generation of Software Documentation, *COLING'96*, University of Copenhagen, Denmark.

Sadek M.D. (1991) *Attitudes mentales et interaction rationnelle : vers une théorie formelle de la communication*, Thèse de Doctorat Informatique, Université de Rennes I, France.

Sadek M.D., Bretier P., and Panaget F. (1997) ARTIMIS: Natural Dialogue Meets Rational Agency, *IJCAI-97*, Japan.

Sadek D. (1999) Design Considerations on Dialogue Systems: From Theory to Technology – The Case of Artimis- *Proceedings of the ESCA TR Workshop on Interactive Dialogue for Multimodal Systems (IDS'99)*, Klöster Irsee, Germany.

Van Heijst G., Schreiber A.TH., and Wielinga B.J (1997) *Using Explicit Ontologies in KBS Development*, International Journal of Human-Computer Studies, 42(2/3) : 183-292.

Wielinga B., Schreiber A., and Breuker J. (1992) *KADS: A Modelling Approach to Knowledge Engineering*. Knowledge Acquisition 4(1):5-53.

Diamod - a Tool for Modeling Dialogue Applications

Anke Kölzer

Speech Understanding Systems (FT3/AV)
DaimlerChrysler AG – Research and Technology
P.O.Box 2360
D–89013 Ulm (Germany)
e–mail: anke.koelzer@daimlerchrysler.com

Abstract

Speech dialogue systems are currently becoming state–of–the–art for different kinds of applications, but they are still weak in the support of spontaneous speech and correct interpretation of what was said. One reason for the lack of good interactive dialogue systems is their complexity. To develop a system which is able to handle more than simple commands and phrases requires a lot of experience and time. To be able to accelerate and improve this process we are currently working on methods and tools which support this development. A new method called *Dialogue Statecharts* was defined for the graphical specification of complex dialogues. It is capable of representing parallel dialogue steps which is e.g. necessary for mixed–initiative dialogues. Our tool system named *Diamod* provides editors for different dialogue concepts, such as dialogue structures, grammars and parameters. The modeling is supported by graphical editors for *Dialogue Statecharts* and *Task Hierarchies*. *Diamod* is able to check for the completeness and consistency of dialogue models. One goal when developing *Diamod* was to provide specification models which are universal enough to be interpreted within different dialogue systems, i.e. different implementations of generic conversational systems.[1] With the help of a uniform representation of data a transformation between different models and different dialogue description languages (DDL) such as VoiceXML (AT&T et al., 2000) and some in–house–DDLs, such as Temic–DDL and Dialogue-Prolog, will be possible.

[1] By this we mean systems which are implemented application independently and are easily adapted to different applications.

1 Introduction

You find different dialogue system approaches on the market place and in research. One has been developed by the DaimlerChrysler research and is able to understand spontaneous speech speaker–independently and carry on dialogues on special topics. The structure and algorithms used are based on concepts developed in the Sundial project (Peckham, 1993). Most applications are made for telephony domains. Thus, up to now we gathered experience in applications like train time–table information, call centers for insurances and telematic systems for traffic data (see (Brietzmann et al., 1994), (Heisterkamp and McGlashan, 1996), (Ehrlich et al., 1997), (Boros et al., 1998) for further information).

We made the experience that developing new applications is very expensive concerning time and staff and needed tools to accelerate the process. Another goal was to make dialogue application modeling possible even for non–experts and help the expert to achieve consistent reusable applications. As there are different dialogue systems all over the world and many steps of application development are similar or even the same for all of them we decided to create tools which are system independent resp. easily adaptable to different needs and different dialogue systems. Our focus was on modeling dialogue structure for information–extracting resp. –processing systems of the slot–filling kind (Bilange, 1991).

In order to find out what functionality a tool must provide to be helpful we analyzed the way we construct dialogue applications and the different knowledge bases that are needed. Similar operation steps have to be executed for every new application in order to obtain a structured and maintainable dialogue. Typical tasks are:

- modeling of the dialogue structure: i.e. divide the dialogue into subdialogues to handle a special part of the interaction like the identification of a caller
- definition of the application parameters, i.e. the parameters necessary to give information to the caller or access a database like the name of the caller
- attachment of system prompts to dialogue situations like what the system has to say when asking the name of the caller
- definition of the appropriate vocabulary (pronunciation) and training of the language models
- definition of linguistic structures (lexicon, grammar, semantics)
- definition of the interface to the application system (e.g. an SQL–interface to a data base)

Diamod supports the specification of all of these dialogue application concepts (some are still under construction) and generates code which is interpreted by the target dialogue system.

2 Requirements

Dialogue systems which allow for spontaneous speech are much more difficult to handle than those which are only capable of processing single commands. *Diamod* has to support different ways of modeling dialogue structure and to transform one into another regarding special consistency requirements.

Thus the knowledge – i.e. the dialogue concepts – has to be represented in a universal way so that different aspects of dialogue can be modeled and code for different dialogue systems can be generated. A transformation from a spontaneous speech dialogue model to a rather restrictive command–and–control one and vice versa should be possible or a transformation from a state–based dialogue flow model to a rule–based one which is organized in tasks (as will be described in section 3.1). The approach must be extensible with little effort for specifying the additional knowledge bases, necessary for conversational systems, such as grammar models.

All the concepts necessary for dialogue flow modeling are to be integrated in the dialogue flow tool. Thus the dialogue flow tool must provide concepts such as application parameters, system prompts, state and task modeling. The state logic has to be described in a rather abstract way so that an automatic transformation for different dialogue systems is possible. Therefore it is not sufficient to use the widely employed state machines with which the specialties of spontaneous speech cannot be described adequately. Instead we use a design method based on Harel's statecharts (Harel, 1987) which are capable of describing concurrency and provide special event mechanisms and called it *Dialogue Statecharts*.

2.1 Properties of *Diamod*

Diamod is a CASE–tool (Computer Aided Software Engineering) specialized for language engineering which provides the concepts necessary for dialogue specification. To be able to develop new and modify old knowledge bases easily, the tool supports the language engineer with the following functionality:

Graphical editors for visual languages such as *Dialogue Statecharts* for the specification of structured dialogue data. The graphical interface shall enable the user to specify his models in a rather easy and intuitive way.

Data representation of all relevant information and the dependences between them.

Consistency checking by a formalism for defining constraints on the models and informing the user of violations of these constraints.

Code–generation (Prolog, VoiceXML, standardized speech API–code, ...) that can be interpreted by the currently preferred generic dialogue system.

Reuse support of formerly developed application models.

Two-phase modeling in order to be able to specify generic data independently of application specific data.

Easy adaptability to further dialogue systems and needs.

The principles of working with *Diamod* are described in the following sections.

3 The Tool System *Diamod*

Figure 1 shows the workflow in *Diamod*. The central unit is the tool system which provides methods for specifying knowledge, keeps the data and models, and does consistency checks. The user modifies the models with the help of a graphical user interface. A second possibility in future editions will be a textual interface for off–line specification where the user can model the dialogue with the help of a dialogue description language. The tool system represents data in a uniform graph representation and is able to generate code in different dialogue description languages such as Prolog[2] or VoiceXML dependent on the generic dialogue system currently in use. This code output (commonly spoken textual files) is read and interpreted by the corresponding generic dialogue system at runtime.

3.1 Dialogue flow models

With *Diamod* the application developer models what the system has to do in a given situation. As this must work for different generic dialogue systems, *Diamod* must also consider the generic features of the system (because they can be different for different dialogue systems). Therefore a two–phase approach is supported where in the first phase a dialogue expert (usually the developer of the generic dialogue system) models application independent data. In a second phase an application developer models application dependent data using the data which was modeled by the expert (Kölzer, 1999).

Another feature of *Diamod* is the support of different dialogue structure models. Our research system is a rule–based system (Ehrlich, 1999) which can be modeled in *Diamod* using tasks and task–hierarchy–diagrams. A rather state–based system can be modeled using the *Dialogue–Statecharts*–editor.

The following listing sums up the most important steps which have to be done by the application developer in order to specify the dialogue flow of a new application:

- definition of the components of the dialogue; e.g. a subdialogue for handling the identification of the caller and finding out why he calls, a subdialogue for reservation of a ticket, and one for callers who only want information.
- definition of the dialogue structure i.e. what the system has to handle first and what comes next. This is done by defining a start dialogue and the successors of each dialogue.
- attachment of application parameters to the dialogues; e.g. in the identification dialogue the system must request the caller's name and password.
- attachment of system prompts to the states where the system has to say something such as *confirm the parameter "Source"* in the reservation dialogue.

The following sections describe how *Diamod* supports the modeling for different approaches. With *Diamod* the user can model every concept by entering a name, a comment and information on the specific structure of the concept.

3.1.1 Task–Based Approach

The DaimlerChrysler research system is organized in tasks. Every task represents a subdialogue, e.g. a caller identification or a hotel reservation. The task structure is organized in a task hierarchy as shown in figure 2, which can be modeled with *Diamod* using the task–hierarchy–editor. At runtime the dialogue system can only activate the direct daughter– or mother–task of a currently active task in this hierarchy. This is used to make dialogue handling easier and more consistent. It is not necessary to model exactly the system states and their sequence as it often has to be done for other dialogue systems. The dialogue system uses a set of dialogue acts (Gazdar, 1981), (Heisterkamp et al., 1992) such as `confirm`, `request` and `inform` in order to distinguish between different dialogue situations. Every task has different application concepts attached to it. Among others these are:

Task (application) parameters: These are the concepts which model what values must be found out in order to reach the goals of the task, e.g. to be able to make a database access. This is usually what you

[2] A predefined sublanguage of Prolog is used to model applications for the DaimlerChrysler research system.

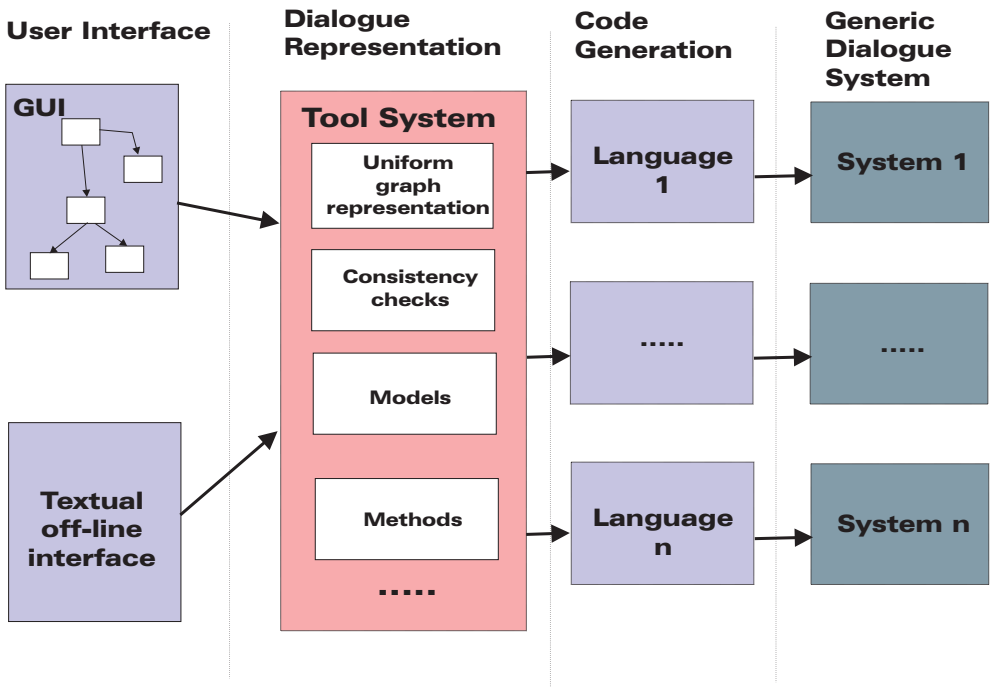

Figure 1: Specification of dialogues with *Diamod*. The central unit is the tool system which provides methods for the dialogue specification, keeps the data and is capable of checking the consistency. Data are modeled by the user on–line with the help of a graphical user interface or offline with textual dialogue description languages. When the specification is complete, the tool system generates the code necessary for the dialogue system in use.

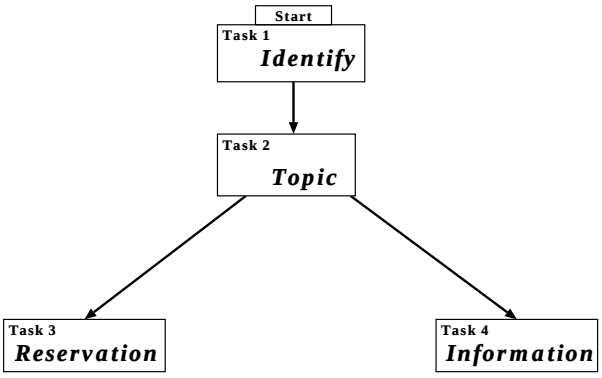

Figure 2: Task hierarchy diagram. Each rectangle models a task i.e. a subdialogue. The edges between the tasks show how tasks can follow each other.

have to request from the user such as a caller name or address. Task parameters can have attributes such as if they are optional or obligatory and if the system may repeat them or not (like passwords). *Diamod* supports the specification of such parameters with user definable types such as records and lists. The user can enter defaults and set the mentioned task parameter specific attributes using masks as shown in figure 3.

Databases and database parameters: If the dialogue system uses databases every task can declare a set of databases and database parameters that it wants to access. Task parameters can be mapped to database parameters. E.g. if the user speaks of *tomorrow*, this must be mapped to a concrete database date like *03.03.2000*. This is supported by *Diamod* with special masks.

System prompts: Given a dialogue act and an application parameter this concept models what the system has to say in that situation. With *Diamod* dialogue acts can be modeled and combined with task parameters in order to model the appropriate prompt. References to the values of task parameters can be used in a prompt such as in "'Your name is <*value name*>?"'. This

prompt is an example for confirming (dialogue act `confirm`) a task parameter `name`. *Diamod* is able to check if a used parameter value reference is feasible. This is the case when the appropriate task parameter was declared for this task. Prompts can be entered for different languages and *Diamod* can check if there is a prompt for every situation in every language. Figure 4 shows the prompt table mask of *Diamod*.

The prompt table can be calculated automatically. I.e. all combinations of dialogue acts and application parameter values are generated in order to gain all those system states, where a system prompt is needed. The result of such a generation is shown in figure 4. The user only has to fill in the prompts or delete table entries which are not needed.

Language models, grammars and lexicons: They can be declared for a task in order to switch between different ones and improve speech recognition this way. This is still under development (see section 4.1).

Actions: The application developer can model typed actions which should be performed on entering, resp. exiting the task. They can be related to task parameters using *Diamod*–masks which offer the user a list of accessable parameters and functions.

The transitions between tasks are realized using rules and conditions which are generic. This means that they are implemented in the dialogue system and do not have to be modeled by the application developer. Such a rule is for example that a task can only be exited successfully if all obligatory task parameters are known. In order to determine the next task to be activated the user's utterance is interpreted, like if he wants a hotel reservation. This together with preconditions for entering possible successor tasks is considered to control the dialogue flow.

When the developer has finished the specification he or she starts the code generation. The code produced can then be interpreted by the dialogue system. For our research system this is Prolog–code specifying the application knowledge bases.

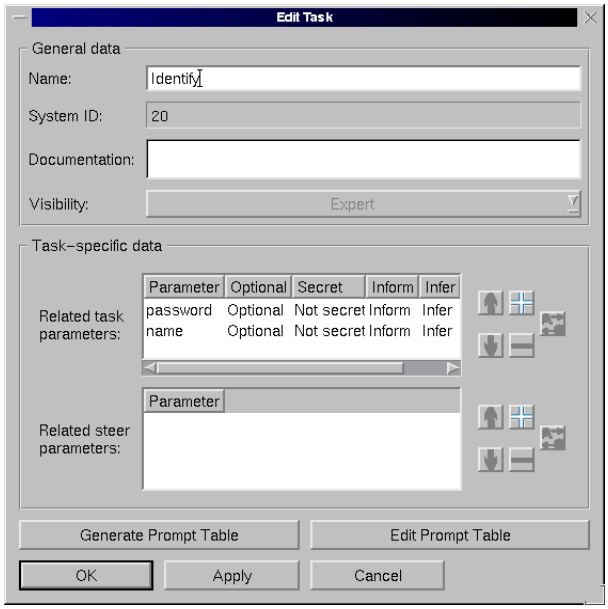

Figure 3: A mask for the description of a dialogue for the DaimlerChrysler research system. In some dialogue systems parameters can have attributes like if they are obligatory or optional. Therefore the masks have to be configured for the dialogue system. Clicking the button `Generate Prompt Table` will generate the possible prompts. Clicking the button `Edit Prompt Table` will open the mask shown in figure 4.

3.1.2 State–Based Approach

Many dialogue systems use a state based approach where dialogue flow is described in detail using state–transition–models combined with events (Failenschmid and Thornton, 1998), (Cole, 1999). Simple state–transition–models are adequate for very simple dialogue systems such as command–and–control systems.[3] As conversational systems have a high complexity of states, the expressiveness of state–transition–models is too small to be a good means for dialogue flow modeling. The number of states is usually too big to be handled by a human.

A good alternative for complex state modeling are statecharts as described by Harel (1987). They provide different means of abstraction such as concurrent states, state refinement, special event handling and action triggers.

[3]These are systems where a speaker may only say special commands like "radio louder" and not speak spontaneously.

Thus modeling of complex dialogue flow can be done in a rather intuitive way. Figure 6 is an example of modeling the task data shown in figure 2 in a state–based way. The dialogues are represented as complex states that are refined top–down to basic states where actions to be triggered are defined. Thus the state *DoDialogue* is represented as an XOR–State. This indicates that the system can only be in one of the states *Identify*, *PossibleTopics* or *End* at the same time. In simple cases a dialogue is represented by a basic state (*End*) which need not be refined any more. The *Reservation*–state must be refined into substates, one for each dialogue act. These are refined again as shown in figure 7. The developer defines entry and exit actions for basic states, i.e. actions to be triggered when entering and when leaving the state. The preconditions for changing the state taking an outgoing transition are described by events and conditions which have to occur. It is possible to describe actions and conditions common for several states or transitions by special means. E.g. any exit from the states *Reservation* and *Information* will lead to the state *End*.

There were already some state–based approaches for graphical dialogue representation. They were never used for complex systems such

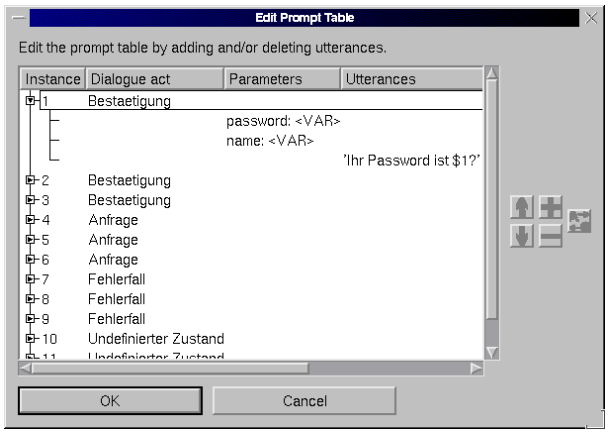

Figure 4: Defining the prompts for a dialogue. The application parameters that are talked about in this dialogue have to be declared for it before. For every dialogue act and every application parameter there must be a system prompt defined. The table here is calculated automatically by *Diamod* using the generic parameters (in this case the dialogue acts) defined by the expert and the application parameters defined here. The application developer only has to fill in the system prompts.

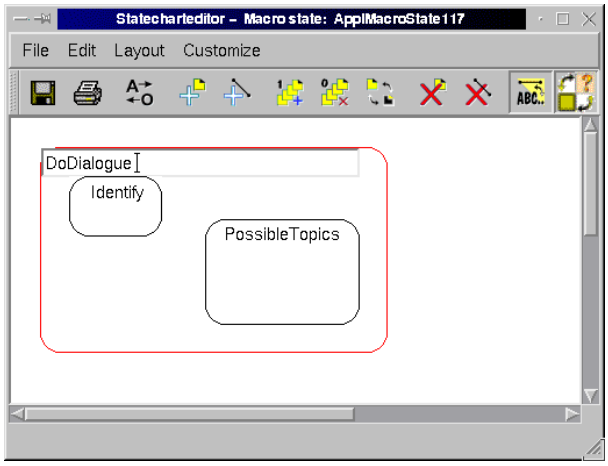

Figure 5: The Dialogue–Statecharts–editor.

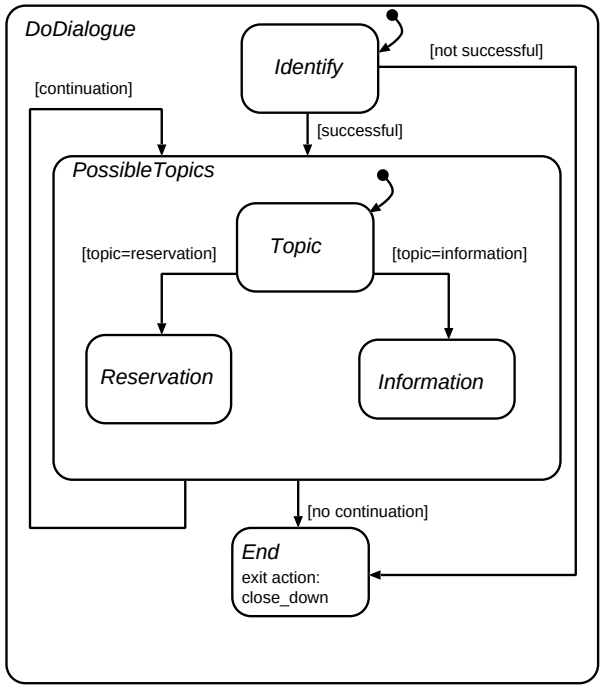

Figure 6: Describing dialogue flow in a statechart based manner. States are represented by rectangles with rounded corners and can be structured. Thus the state *DoDialogue* is an XOR–State. This indicates that the system can only be in one of the states lying graphically inside. The small rounded arrow at the state *Identify* means that this is the default entry state for *DoDialogue*. The transitions are labeled with conditions indicating when this transition is to be taken.

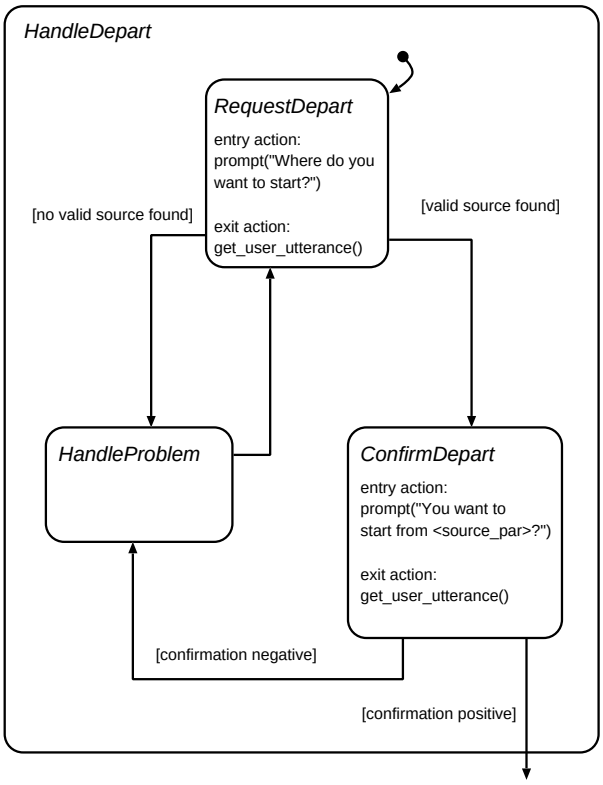

Figure 7: Refining states: the confirmation subdialogue in the reservation dialogue. The dialogue developer can add to the basic states actions to be triggered. Entry actions are executed when entering the state, exit actions when leaving the state.

as mixed– and user–initiative dialogues because there expressiveness was too small. With *Dialogue Statecharts* we think that we found a way to handle even such complex structures using the concurrency concepts of Harel's statecharts. Figure 8 shows an example for the representation of a mixed initiative dialogue. All the topics that a speaker may talk about in one sentence are represented as parallel slots of a concurrent dialogue state. All the slots represent parallel[4] substates of the dialogue system. If the speaker can tell the departure city, the destination city and the departure time in one sentence in a train time table information, there will be one slot for every parameter. The action which the dialogue system has to perform are described inside these slots. E.g. if the utter-

[4]This does not mean, that they really have to be processed in parallel, but that they are independent of one another.

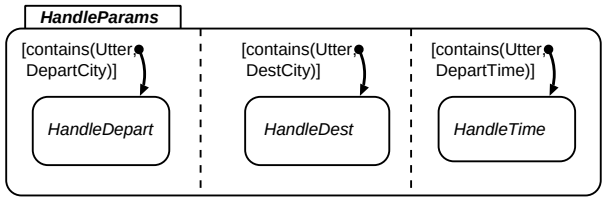

Figure 8: Concurrent states: dialogue parameters which the speaker can talk about in one sentence are handled in parallel. The picture represents e.g. for a train time table information that the speaker can tell the departure city, the destination city and the departure time at one time. The statechart in figure 7 is a possible refinement of the state *HandleDepart*.

ance contains a value for the departure time the value of a dialogue parameter concerned with the departure time must be set and analogous for the other parameters.

Diamod supports the state–based modeling with graphical editors which can check consistency concerning the depth of the state–hierarchy, unwanted cycles, completeness of the system prompts etc. The rules which indicate when the model is consistent must be entered for every dialogue system, as they can be different according to the given system. States can be described in detail using masks similar to the ones used for task modeling (see figure 3). Here also prompts, conditions, actions and so on can be related to the state.

This is only a short description of what can be done with statecharts. The figures are simplified for reasons of clarity. Statecharts offer many features of abstraction which makes them capable of complex state modeling.

The models specified by the user of *Diamod* are internally represented as graphs which are also used as the basis for the model transformation. In order to do this, rules have to be specified how one graph can be automatically transformed into another. As different dialogue systems work with different concepts this transformation cannot be completely automatic. The approach here is to use defaults where possible and ask the user to make some additional editing, where needed. Some information can be lost by such a transformation. *Diamod* must warn the user about this.

3.1.3 Rule–based approach

Advanced dialogue systems are often not modeled using states and transitions but rules and conditions. *Diamod* can support this, too, as states can be used as abstract dialogue units. Thus states can represent subdialogues and dialogue steps. Every state can be modeled by a set of preconditions which indicate when the state may be entered and postconditions which represent when the state can be exited successfully. Rules can be specified to model how the next state to be activated has to be selected. There is a default order on the states which supports this selection. Some of these concepts are used for the application modeling of the DaimlerChrysler research dialogue system.

The benefits of *Diamod* in this context has not been investigated yet as one needs a well defined dialogue description language as interface to such a rule–based dialogue system.[5] Thus this is work for the future.

3.1.4 Concistency checking

An important point is that the tool is capable of checking the completeness of the models and their consistency. This is done using an object–oriented graph structure which represents all required concepts and the dependences between them. Consistency checks can be executed by formulating constraints on the graph using path expressions and having them examined by a special path interpreter (Ebert et al., 1996). Thus, it is possible to guarantee that for example

- there are no problematic cycles in the model
- there is a system prompt defined for every system initiative state (i.e. states where the system has to speak an utterance) and every parameter, so that the system never runs in a situation where it is 'speechless'.
- domains are defined properly for all parameters
- there is a following state in every situation (or the end of the dialogue)

4 Summary

The paper introduces the tool system *Diamod* which implements a universal approach for the specification of dialogue applications with a focus on task–oriented dialogue systems of the slot–filling kind. The tool system supports dialogue flow modeling in terms of tasks and states which can be specified in detail by describing parameters, actions, prompts and other typical concepts of dialogue models. The most important features of *Diamod* are

- a uniform knowledge representation which allows for automatic transformation of data for different generic dialogue systems
- the possibility of modeling different aspects of dialogue with different views on the data
- the capability of checking the consistency of the models automatically
- the support of the reuse of models
- the easy adaptability to additional knowledge bases and different dialogue systems.

4.1 State of work – technical realization

The task and statechart modeling are completely implemented as described in section 3. The following summary gives an overview over what *Diamod* contains up to now:

- task structure modeling as shown in figure 2
- *Dialogue Statecharts* modeling as shown in figure 5; this includes relating prompts, conditions, actions and events etc. to the dialogues. These are described in masks as shown in figure 3
- automatic prompt table generation
- system parameters and application dependent application parameters which represent the dialogue state
- mapping from application parameters to data base parameters; e.g. if the caller talks about "tomorrow" this has to be mapped into the actual date in a form that can be handled by the database such as 03.02.99
- attaching multilingual system prompts to the modeled dialogues.

The system is implemented in C++ using graphs and one set of constraints per dialogue

[5] This would be a quite interesting project and we would be grateful for suggestions of collaboration here.

system, which represents the consistency rules for this system.

We are currently working on adapting the system to the needs of Temic–DDL (a dialogue description language developed by Temic) and VoiceXML (AT&T et al., 2000) and on the automatic transformation of models. The integration of a grammar specification tool (work in progress) is planned for the end of the year. This module will provide different grammar formalisms such as UCG (Zeevat, 1988), PSG (Boros, 1997) and Java Speech API (Sun microsystems, 2000). The conversion between these grammar types will be supported.

The implementation of the system has just been finished so far that it can be used by application developers. But as it is completely new and the graphical user interface is still being improved in order to make it more intuitive, we have not made any experience yet how much the win of using *Diamod* will be for realistic dialogues. We are currently starting the evaluation and we are optimistic after the first tests.

4.2 Outlook

The dialogue systems we aimed at when we developed *Diamod* were mainly task–oriented systems, i.e. systems giving information on special topics or modifying databases. The benefits of *Diamod* in another context like translation systems (e.g. Verbmobil (Wahlster et al., 2000)) has not been investigated so far, but this is one of our goals in the future.

Another interesting topic would be the adaptation of *Diamod* to dialogue systems using dialogue grammars (Reichman, 1981) or plan–based systems (Cohen and Levesque, 1980).

Further plans include the integration of a prototyper into the tool system to be able to immediately check the consequences of the modifications made. With these different means it will be possible even for an untrained user to specify new applications for his or her own requirements.

References

AT&T et al. 2000. *VoiceXML*. World Wide Web, http://www.voicexml.org/.

Eric Bilange. 1991. A task independent oral dialogue model. In *Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics*, pages 83–88, Congress Hall, Alexanderplatz, Berlin, Germany.

Manuela Boros, Ute Ehrlich, Paul Heisterkamp, and Heinrich Niemann. 1998. An evaluation framework for spoken language processing. In *Proceedings of the International Workshop Speech and Computer 1998*, Russian Academy of Sciences, St.Petersburg, Russia, October.

Manuela Boros. 1997. Gepard - dokumentation des parsers f"ur phrasenstrukturgrammatiken. Projektbericht, FORWISS, Juni.

Astrid Brietzmann, Fritz Class, Ute Ehrlich, Paul Heisterkamp, Alfred Kaltenmeier, Klaus Mecklenburg, Peter Regel-Brietzmann, Gerhard Hanrieder, and Waltraud Hiltl. 1994. Robust speech understanding. In *International Conference on Spoken Language Processing*, pages 967–970, Yokohama.

Philip R. Cohen and Hector J. Levesque. 1980. Speech acts and the recognition of shared plans. In *Proceedings of the Third Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 263–271.

Ron Cole. 1999. Tools for research and education in speech science. In *Proceedings of the International Conference of Phonetic Sciences*, San Francisco, USA, August.

Jürgen Ebert, Angelika Franzke, Peter Dahm, Andreas Winter, and Roger Sttenbach. 1996. Graph based modeling and implementation with eer/gral. In B. Thalheim, editor, *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, number 1157 in LNCS, pages 163–178, Berlin. Springer.

Ute Ehrlich, Gerhard Hanrieder, Ludwig Hitzenberger, Paul Heisterkamp, Klaus Mecklenburg, and Peter Regel-Brietzmann. 1997. ACCeSS - automated call center through speech understanding system. In *Proc. Eurospeech '97*, pages 1819–1822, Rhodes, Greece, September.

Ute Ehrlich. 1999. Task hierarchies - representing sub-dialogs in speech dialog systems. In *6th European Conference on Speech Communication and Technology (EUROSPEECH)*, Budapest, Hungary, September.

Klaus Failenschmid and J.H. Simon Thornton. 1998. End-user driven dialogue system design: The reward experience. In *Proceedings*

of the International Conference on Spoken Language Processing (ICSLP) 1998, Sydney, Australia, November.

Gerald Gazdar. 1981. Speech act assignment. In Aravind K. Joshi, Bonnie Lynn Webber, and Ivan Sag, editors, *Elements of Discourse Understanding*, pages 63–83. Cambridge University Press, Cambridge.

David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Paul Heisterkamp and Scott McGlashan. 1996. Units of dialogue management: An example. In *Proc. ICSLP '96*, volume 1, pages 200–203, Philadelphia, PA, October.

Paul Heisterkamp, Scott McGlashan, and N. Youd. 1992. Dialogue semantics for an oral dialogue system. In *International Conference on Spoken Language Processing (ICSLP), Volume1*, pages 643–646, Banff, Alberta, Canada.

Anke Kölzer. 1999. Universal dialogue specification for conversational systems. In *Proceedings of the International Workshop: Knowledge and Reasoning in Practical Dialogue Systems, IJCAI 1999*, pages 65–72, Stockholm, Sweden, August.

Jeremy Peckham. 1993. A new generation of spoken dialogue systems: Results and lessons from the sundial project. In *3rd European Conference on Speech Communication and Technology (EUROSPEECH'93); Vol.1*, pages 33–40, Berlin, September.

Rachel Reichman. 1981. *Plain-speaking: A theory and grammar of spontaneous discourse.* Ph.D. thesis, Department of Computer Science, Harvard University, Cambridge, Massachusetts.

Sun microsystems. 2000. *Java Speech API.* World Wide Web, http://java.sun.com/products/java-media/speech/index.html.

Wahlster et al. 2000. *Project Verbmobil.* World Wide Web, http://www.coli.uni-sb.de/~vm/.

Henk Zeevat. 1988. Combining categorial grammar and unification. In *Reyle, Rohrer: Natural Language Parsing and Linguistic Theories*, pages 202–229, Dordrecht. D. Reidel Publishing Company.