Bringing Information Extraction out of the Labs: the *Pinocchio* Environment

Fabio Ciravegna[1] and **Alberto Lavelli**[1] and **Giorgio Satta**[2]

Abstract. *Pinocchio* is an environment for developing Information Extraction applications. New applications and languages can be covered by just writing declarative resources. Information is represented uniformly throughout the architecture: all the modules use the same input structure and the same type of declarative resources. Modules are implemented via the same basic processors and share a common environment for resource development and debugging. The result is an environment easy to use with limited training and skills.

1 INTRODUCTION

The exponential increase in the quantity of information held in digital archives has fueled growing interest in techniques for Information Extraction. Given that the vast majority of available information is textual (e.g., web pages, electronic newspapers, agency news), the role of Information Extraction from text (IE) is becoming more and more central. An IE system extracts pieces of information that are salient to the user's needs. The typical output is a set of filled templates containing the extracted information. This information can be used for different purposes, e.g. data base population, text indexing, information highlighting.

IE is one of the few areas of Natural Language Processing (NLP) where evaluation methodologies have been defined and shared by a large community. Extensive evaluations have been carried out in MUC conferences [12]. Applications are beginning to appear, mainly for IE subtasks with a high reliability on generic corpora (e.g., named entity recognition [11]). Nevertheless there seems to be a disproportion between the effort spent in IE in the last ten years and the number of real-world applications already implemented. We believe that one of the reasons for this is the fact that, despite the emphasis on evaluation, scarce attention has been paid to whether the currently available technology and systems are suitable to produce real-world applications. MUCs stressed specific aspects of building applications (i.e., effectiveness in results and limited development time), but other important aspects were left out. As a consequence many existing systems do not take them into account. Unfortunately such aspects are very relevant for industrial applications and in our opinion this is one of the factors that is currently limiting the market for IE.

The first aspect concerns the cost of new applications. Application cost was considered in MUC as a by-product of the limited porting time allowed for development, but this is a simplified perspective. A potential application developer will consider as cost also the person time needed for building applications and the skills required. Given the current technology, competence in computational linguistics (CL) is required for developing IE applications. Skills in CL are not very common in industries. Moreover, in research systems many modules adopt approaches directly derived from basic research activities, and hence based on formal methodologies difficult to understand for a layman. It is unlikely that commercial companies would hire a whole group of experts in specific NLP areas. They would be more inclined to hire a small group (ideally just one person) with generic competence in CL. In addition, many IE systems are based on patchwork architectures composed of modules studied and implemented separately by experts in different areas of NLP (e.g., parsing, discourse processing) and later integrated. There is therefore a lack of uniformity in the architecture because such modules generally use different formalisms and development environments. A potential user will then be forced to cope with a variety of formalisms and environments. From this point of view most of the current IE systems do not seem to be ready for applications.

A second aspect for evaluating IE technology is the availability of development environments that allow the definition of new applications by people different from the system developers. This requires the possibility of porting via definition of declarative resources only.

A third critical aspect is easy integrability in two directions:

- external: the final application should be easily integrable into the user environment;
- internal: the architecture should provide facilities for extensions to cope with application-specific needs (e.g., the recognition of complex technical terminology). The inclusion of external modules based on commercial technology (e.g., morphological analyzers) is an additional interesting feature.

A fourth fundamental aspect in the current application scenario concerns multilinguality: the more documents are electronically available in languages other than English, the more the need of coping with different languages arises. Multilingual interaction can be realized in two ways:

- extracting information written in different languages (multilinguality);
- extracting information written in one language and presenting it in a different one (cross-linguality).

The problem of defining the same application for the same domain but in different languages then arises: there is the need of reusing already developed domain-dependent resources, without starting from scratch for every new language [7]. Most of the current IE systems are monolingual and probably not enough attention has been so far paid to this aspect.

In this paper we present *Pinocchio*, a tool for the quick delivery of multilingual and cross-lingual IE applications whose design takes

[1] ITC-irst Centro per la Ricerca Scientifica e Tecnologica, via Sommarive 18, 38050 Povo (TN), Italy, email: {cirave|lavelli}@irst.itc.it
[2] Dip. di Elettronica e Informatica, Università di Padova, via Gradenigo 6/A, 35131 Padova (PD), Italy, email: satta@dei.unipd.it

into account the needs mentioned above. In this respect we believe that *Pinocchio* is a step in the direction of bringing IE into real world-applications. In the paper we present the architecture and discuss how its organization meets application needs.

2 *Pinocchio*

Pinocchio is an environment that provides a uniform and powerful set of tools for IE applications. Its design aims to supply a single environment taking into account both the current trends in IE (evaluation methodology, reduced porting time) and the needs mentioned above (reduced cost, usability, integrability and multilinguality).

New applications can be developed and new languages can be covered by only modifying declarative resources. The declarative resources used by all the modules in the system employ the same formalism: all the modules operate on the same input structure and use the same type of rules (processed by the same basic machinery). The environment for developing and debugging resources is uniform across the whole architecture. This uniformity simplifies the application development process and reduces the need of manpower and the required skills, i.e. the cost of the application. A single person with a generic knowledge in CL can eventually do the entire work.

Pinocchio is available in two configurations: the development environment and the delivery configuration. The development environment provides a set of tools that support application development (e.g., editors, compilers, debugger, tracers). At the end of the development process, a delivery configuration is produced that can be inserted as a black-box into the user environment through an API. *Pinocchio* provides an open architecture. In case of applications with specific requirements, it is possible either to define and integrate new modules (using the internal processors) or to include pre-existing modules (e.g., commercial technologies).

Domain-dependent and language-dependent resources are sharply separated. For this reason, when building new applications in the same language, most of the language-dependent resources can be reused. Conversely, when porting the same application from one language to another, most domain-dependent resources may be reused.

The system output is compatible with the current IE evaluation methodologies, in particular with the MUC standards and tools [6].

Pinocchio has been used to develop applications and demonstrators, mainly in the financial domain. The system has been mainly used for Italian, but demonstrators exist for English and (partially) Russian. Coverage of English and Russian and their trial applications were defined by people different from the system developers.

In the next section the formalism used for information representation is introduced. In Section 4 the open architecture and the basic processors are described and the default application is presented. In Section 5 development and delivery configurations are outlined. Finally applications and experimental results are discussed.

3 INFORMATION REPRESENTATION

Before describing the *Pinocchio* architecture we introduce the formalism for representing information shared by all the modules.

Every lexical element a in a text t is abstractly represented by means of elementary objects, called **tokens**. A token T is associated with three structures:

- $[T]_{dep}$: a tree representing syntactic dependencies between a and other tokens in t.
- $[T]_{feat}$: a feature structure representing syntactic and semantic information needed to combine a with other tokens.

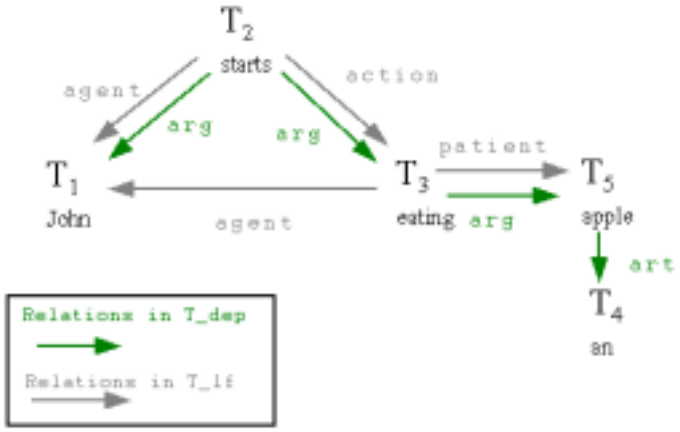

Figure 1. Relations in $[T_n]_{dep}$ and $[T_n]_{lf}$ for the sentence 'John starts eating an apple'

- $[T]_{lf}$: a Quasi Logical Form providing a semantic interpretation for the combination of a with other tokens.

Tokens are organized in a basic data structure, called **token chart**. This is a directed graph whose vertices are tokens and whose arcs represent binary relations in either $[T]_{dep}$, $[T]_{feat}$ or $[T]_{lf}$. Initially, the arcs in the token chart represent lexical adjacency between tokens. During text processing such structure is processed and dynamically modified.

All *Pinocchio*'s modules operate on the token chart by means of rules organized in sequences of cascades (see [2] for a description of the rule formalism). These rules perform deterministic analysis and are implemented as Finite-State Transducers. Rules can access, incrementally build and update the token chart, i.e., they can access and modify all the three token structures mentioned above. Lexical, syntactic and semantic constraints can therefore be used in rules at any level [2]. An example will clarify how the token chart allows a rule to test semantic and syntactic constraints in a uniform way. In Figure 1 the token chart for "John starts eating an apple" is shown (only relations in $[T]_{dep}$ and $[T]_{lf}$ are displayed). A rule could easily test whether the "patient" of eating is a definite or an indefinite NP by starting from T_3, traversing the arc labeled "patient" (which is part of $[T_3]_{lf}$), reaching T_5 and then moving to T_4 by traversing the arc labeled "art" (in $[T_5]_{dep}$). $[T_4]_{feat}$ will then provide the information about the definiteness of the article (not shown).

4 ARCHITECTURE

An architecture in *Pinocchio* is composed of three main parts:

- The architecture manager that controls the information flow. It activates the other modules in order to perform IE.
- A set of modules, each devoted to a specific task (e.g., parsing). This set is open in principle.
- The kernel that provides the basic machinery used by the task-specific modules above.

Different architectures can be defined, given the set of task-specific modules available and the directions provided to the architecture manager.

In the rest of this section we focus on the kernel and the task-specific (internal and external) modules. Then we move to the description of the default architecture provided in *Pinocchio*.

4.1 The Kernel

The kernel is the set of basic processors and resources on which the task-specific modules rely (Figure 3). Its internal processors are:

- a **Rule Engine** that applies rules on the token chart;
- a **Knowledge Base Management System** (KBMS) that provides the formalism for defining the knowledge base and the basic machinery for testing consistency in $[T]_{lf}$;
- a **Feature Structure Management System** (FSMS) that provides the formalism for the typed feature system and the machinery for unification in $[T]_{feat}$.

Declarative resources available to the kernel are:

- **Knowledge Base** (KB): it defines the ontology for the application domain. Definitions and restrictions in the KB are used by the KBMS for computing consistency in $[T]_{lf}$.
- **Typed Feature System**: it is the basis for the (language-dependent) syntactic description. It provides the information used in the $[T]_{feat}$ and unified by the FSMS [4].
- **Lexicon**: it is divided into:
  - Foreground Lexicon (FL): for each word it provides the mapping with the ontology and a description of syntactic and semantic features (e.g., subcategorization frame). This information is copied into $[T]_{lf}$ and $[T]_{feat}$ during lexical lookup. The FL contains terms tightly bound to the domain and generally consists of few hundred words.
  - Background Lexicon (BL): it is a generic dictionary providing default generic information for words outside the FL.

4.2 Task-specific Modules

The architecture manager activates a specific set of modules in order to carry out IE tasks. Each module is devoted to a certain task. Modules are divided into internal and external. **External modules** are processors integrated in *Pinocchio* via an API. In general the role of such modules is limited to the connections with the application environment in which the final IE system will run, or to integrate commercially available software (e.g., morphological analyzers). **Internal modules**, on the other hand, are native and rely on the kernel for processing. They are composed of:

- a declarative resource representing cascades of rules;
- a processor (**wrapper**) whose aim is to:
  - accept the control from the main module;
  - select the token path, i.e. the portion of the token chart on which the module operates (for example, the token path for a parser is an ordered list of tokens representing a sentence);
  - ask the rule engine to apply the rules on the token path;
  - provide post-processing of results (if necessary);
  - return the control to the main module.

Each internal module operates on the token chart and eventually modifies it. It is important to stress that all the internal modules rely on the kernel for processing rules. They use the same rule engine, the same rule formalism and primitives and operate on the same token chart. This uniformity is of great help in building applications.

The definition of new internal modules requires the definition of both the sequence of the rule cascades and the wrapper. In our experience a module of average complexity can be defined in few hours.

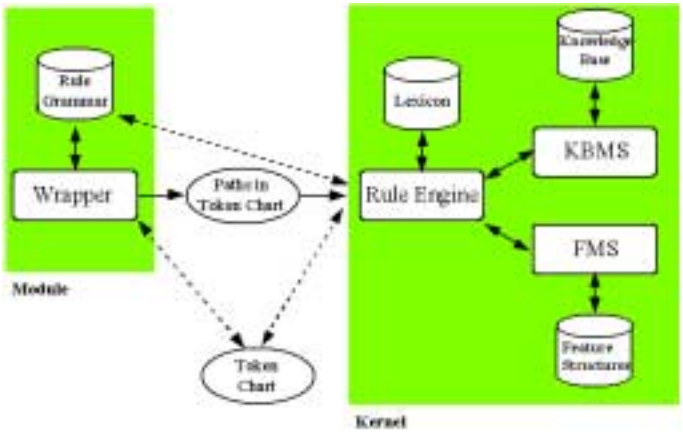

Figure 3. Communication between a task-specific module and the kernel

4.3 Default Architecture

In *Pinocchio* a default architecture is provided following the model presented in [9]. Such default architecture allows to define new applications or to cover new languages by just modifying declarative resources (lexicon, knowledge base, typed feature system and finite-state grammars). For applications with specific needs, new architectures can be defined and the set of task-specific modules can be extended. The current default architecture consists of internal modules implementing the following tasks:

- Named entity recognition: identification and classification of proper names, numbers, percentages, monetary quantities, etc.
- Preparsing: identification of sentence and clause boundaries.
- Parsing: production of a sufficient IE approximation (SIEA) of a complete parse tree for each sentence. A SIEA is a complete parse tree where all the relations relevant for template filling are explicitly represented, while other relations are underspecified or even left implicit. In other words, a SIEA is the minimal approximation of a complete parse tree providing all the relations useful for IE. Hence, differently from many current systems that perform just partial parsing [10, 8], the parser performs a kind of full text parsing [2]. Since parsing is deterministic, just one structure is produced for $[T_{sent}]_{dep}$, $[T_{sent}]_{feat}$ and $[T_{sent}]_{lf}$.
- Inference: derivation of additional information in $[T]_{lf}$ not explicitly mentioned in the text, that can be derived by reasoning on $[T]_{lf}$ itself (e.g., if a person working for company X is hired by company Y, s/he is no longer employee of X).
- Discourse processing: (pro)nominal references are resolved and implicit relations captured (e.g., "The Bank of Japan decided ... The president said ..."). The result of discourse processing is $[T_{ext}]_{lf}$, a logical form associated to the whole text.
- Template filling and merging: $[T_{ext}]_{lf}$ is mapped into application-specific templates. Template merging and recovery actions cope with missing information. Templates are internally represented using the language for knowledge representation.

External modules in the default architecture are:

- a text zoner for recognizing text parts (e.g., title, body). This is an application-dependent module to be defined for each application;
- a preprocessor that provides tokenization, morphological analysis and PoS tagging. *Pinocchio* does not include any direct facilities

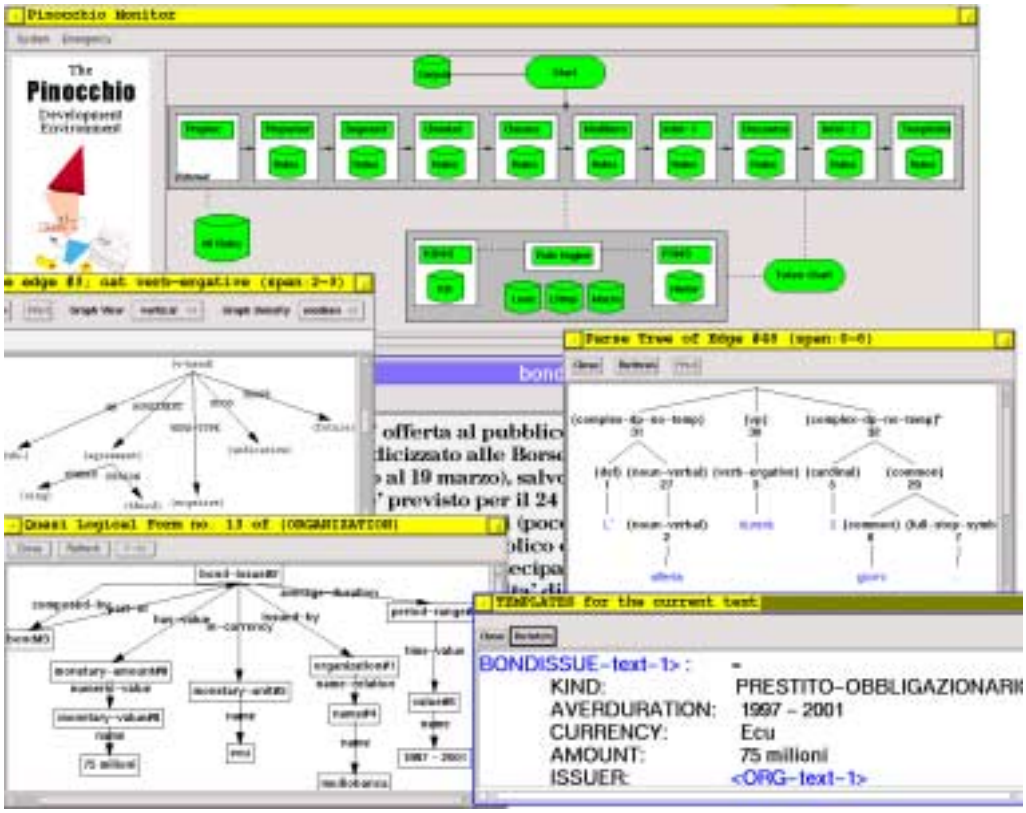

Figure 2. A snapshot of the browsers for system results showing a $[T]_{dep}$ (center right), a $[T]_{lf}$ (bottom left), a $[T]_{feat}$ (middle left). The template browser is shown at the right bottom. The panel for controlling the architecture is shown at the top. It is used for tracing, inspecting partial results, etc.

for preprocessing. The API allows to interface existing preprocessors. A built-in interface is currently provided for the LinguistX tools produced by InXight^{TM}.

- a postprocessor translating the filled templates from the internal format into a user-defined format. *Pinocchio* provides a default module producing a format fully compatible with the MUC evaluation methodology to be used during development for comparing system results against user-tagged corpora.

5 DEVELOPMENT ENVIRONMENT AND DELIVERY CONFIGURATION

Pinocchio provides an environment for supporting the users in developing applications. It includes facilities for the following tasks:

- resource development:
  - editors, compilers and graphers for kernel resources (KB, lexicon and type hierarchy);
  - editor and compiler for rule cascades;
- resource test:
  - a panel for running the system on texts and/or corpora;
  - a set of tracers to monitor:
    - the effect of each rule on the token chart;
    - the rule cascade application;
    - the KBMS and FSMS consistency checks;
  - a browser for the token chart;
  - specialized browsers for $[T]_{dep}$, $[T]_{feat}$, $[T]_{lf}$, and IE templates (Figure 2);
- application test: comparison of system results with the templates provided by a human being for user-defined corpora via the MUC scorer [6].

The uniformity of the editing and debugging environment is important in the application development process. It allows to avoid the problem of many current IE systems that are composed of different modules, each based on its own formalism and with its own debugging environment. In our experience, using our framework a single person is able to build a whole application.

The possibility of automatically comparing system results with user-provided results allows to continuously monitor resource development. It also provides the customer with an objective evaluation of the quality of results in the final IE application.

At the end of the application development process, a delivery configuration is produced with reduced hardware and software requirements. Such configuration does not contain the editing, compiling and debugging facilities or the graphical interface. Resources are compiled into Lisp code and no longer require the rule engine. The architecture is frozen. The final application can then be inserted into the user environment as a black-box.

6 APPLICATIONS AND EXPERIMENTAL RESULTS

Applications and demonstrators in different languages have been developed using *Pinocchio*, mainly in the financial domain [3]. For the

Italian language, one application about bond issues has been fully developed and two others have reached the level of demonstration (management succession and company financial results). Demonstrators were developed for English (economic indicators) and partially for Russian (bond issues). Concerning the Italian application to bond issues, the system reached P=80, R=72, F(1)=76 on 95 texts used for development (33 ANSA agency news, 20 "Il Sole 24 ore" newspaper articles, 42 Radiocor agency news; 10,472 words in all). Figure 4 shows the adopted template. Effectiveness was automatically calculated by comparing the system results against user-provided results via the MUC scorer. The development cycle of the template application was organized as follows: resources were developed by inspecting the first 33 texts of the corpus. Then the system was compared against the whole corpus (95 texts), reaching R=51, P=74, F(1)=60. Finally resources were tuned on the whole corpus mainly by focusing on the texts that did not reach sufficient results in terms of R&P. The system analyzed 1,125 word/minute on a Sparc Ultra 5, 128M RAM. The Italian Named Entity Recogniser was also tested as a stand alone module on generic corpora in the financial domain. It reached P=86 R=92 in a blind test on a 5,700 word corpus after training on a 9,200 word corpus.

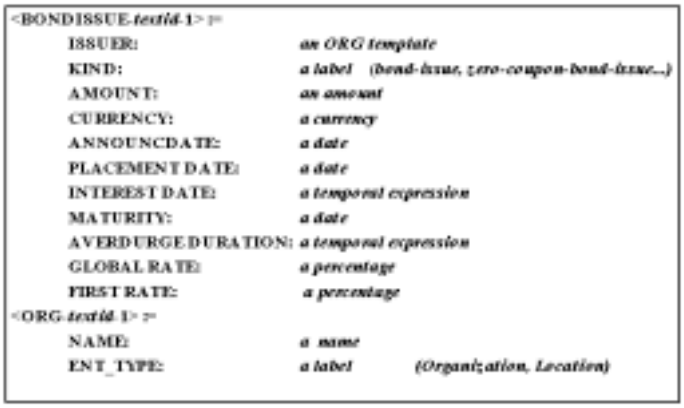

Figure 4. The template to be filled for bond issues.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented *Pinocchio*, an environment for developing and running IE applications. The most appealing facilities of this environment are:

- It can be adapted to new languages and applications with a limited manpower by developing declarative resources only.
- The architecture is open to the insertion of new internal and external modules.
- It provides a uniform architecture where all the internal modules are implemented using the same basic processors (the kernel). Improvements in efficiency can then be obtained by modifying the implementation of the kernel. Last year the rule engine was reimplemented in two weeks and the system performances doubled.
- It provides a uniform common environment for resource development and debugging. The user is required to learn just the formalism for rules, feature structures and knowledge representation.
- Rules are organized in sequences of cascades. Cascades are easy to control and debug. A methodology is suggested for sequence and cascade organization that maximizes the separation of linguistic knowledge from domain knowledge and simplifies porting [5].
- It is possible to use the environment with only a limited knowledge of computational linguistics; a first-year PhD student was able to define the resources covering a reasonable subset of English and to develop a demonstrator for an application.

Pinocchio has already demonstrated its capabilities in coping with different languages. Although we have not yet tested it in a cross-lingual application, the underlying approach is compatible with [7].

Some applications and demonstrators have been developed so far and others are under study. In our experience new applications required from two to four person/months, while porting to new languages required four to six person/months. We are currently discussing a pilot project with a big American corporate for applications in the field of pharmacology on English texts. We are also continuing the work in the financial domain.

Future work on *Pinocchio* will concern the application of machine learning techniques to IE, in order to simplify rule writing and lexicon development [1]. The current application development cycle requires some months of person time; we intend to reduce it to some weeks.

REFERENCES

[1] Fabio Ciravegna, 'Learning to tag for information extraction', in *Proc. of the ECAI workshop on Machine Learning for Information Extraction*, eds., F. Ciravegna, R. Basili, and R. Gaizauskas, Berlin, (2000).

[2] Fabio Ciravegna and Alberto Lavelli, 'Full text parsing using cascades of rules: An information extraction perspective', in *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics*, Bergen, Norway, (1999).

[3] Fabio Ciravegna, Alberto Lavelli, Nadia Mana, Luca Gilardoni, Silvia Mazza, Massimo Ferraro, Johannes Matiasek, William J. Black, Fabio Rinaldi, and David Mowatt, 'FACILE: Classifying texts integrating pattern matching and information extraction', in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, (1999).

[4] Fabio Ciravegna, Alberto Lavelli, Daniela Petrelli, and Fabio Pianesi, 'Developing language resources and applications with GEPPETTO', in *Proceedings of First International Conference on Language Resources & Evaluation*, Granada, Spain, (1998).

[5] Fabio Ciravegna, Alberto Lavelli, and Giorgio Satta, 'Full parsing approximation, finite-state cascades and grammar organization for information extraction', Technical Report 9911-02, ITC-irst, (November 1999).

[6] Aaron Douthat, 'The message understanding conference scoring software user's manual', in *Proceedings of MUC-7*, `http://www.muc.saic.com/`, (1998).

[7] Robert Gaizauskas, Kevin Humpreys, Saliha Azzam, and Yorick Wilks, 'Concepticons vs. lexicons: An architecture for multilingual information extraction', in *Information Extraction: A multidisciplinary approach to an emerging information technology*, ed., Maria Teresa Pazienza, 28–43, Springer Verlag, (1997).

[8] Ralph Grishman, 'Information extraction: Techniques and challenges', in *Information Extraction: a multidisciplinary approach to an emerging technology*, ed., M. T. Pazienza, Springer Verlag, (1997).

[9] Jerry R. Hobbs, 'The generic information extraction system', in *Fifth Message Understanding Conference (MUC-5)*, ed., B. Sundheim. Morgan Kaufmann Publishers, (August 1993).

[10] Jerry R. Hobbs, Douglas E. Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tison, 'FASTUS a cascaded finite-state transducer for extracting information from natural language text', in *Finite State Language Processing*, eds., Emmanuel Roche and Yves Schabes, MIT Press, (1997).

[11] G. R. Krupka and K. Hausman, 'IsoQuest Inc.: Description of the NetOwl extractor system as used for MUC-7', in *Proceedings of the Seventh Message Understanding Conference (MUC-7)*, `http://www.muc.saic.com/`, (1998).

[12] MUC7, *Proceedings of the Seventh Message Understanding Conference (MUC-7)*, SAIC, `http://www.muc.saic.com/`, 1998.