Participatory Design for Linguistic Engineering: the Case of the GEPPETTO Development Environment

Fabio Ciravegna, Alberto Lavelli, Daniela Petrelli, Fabio Pianesi
Istituto per la Ricerca Scientifica e Tecnologica
Loc. Panté di Povo
I-38050 Trento, Italy
e-mail: {cirave|lavelli|petrelli|pianesi}@irst.itc.it

Abstract

Current tools for Linguistic Engineering (LE) do not completely fit the requirements for scale development and deployment of real applications. What seems to lack in the available tools is a comprehensive study of user needs. This is a real limitation in a field where people with very different backgrounds (from computer scientists to linguists) are involved. To avoid such a shortcoming we adopted the Participatory Design (PD) methodology, i.e. a User Centered approach that favors the definition of tools suited to the real user needs. In this paper we show how such methodology was used in the design and implementation of a development environment for LE applications.

1 Introduction

The growing number of applications exploiting NLP techniques is bringing about a shift from an artisan attitude with respect to NLP towards the need for more sophisticated solutions and tools (Linguistic Engineering, LE). Consequently, interest has increased in the study and implementation of environments capable of supporting the users in the development, testing and debugging of Linguistic Engineering Applications (LEAs). In this respect, a major feature of a Linguistic Engineering Application Development System (LEADS) is to provide facilities for the development, reuse and integration of linguistic processors and data.

Despite the remarkable results that have been achieved (too many to be listed here), the general impression is that these systems do not completely fit the requirements for scale development and deployment of real applications. In this connection, one major point is the concern about human factors. In general, studies have conclusively shown that even small differences in the characteristics of the end users (e.g., computer experience, knowledge about the domain, use of the system) can heavily affect the suitability of the tools developed (Nielsen, 1993). The development of a LEA is a task involving different kinds of skills and expertise. For instance, it is conceivable that the development and maintenance of the needed linguistic resources (grammars and lexica) require different skills (and be therefore pursed by different people) than those involved in the construction and validation of the architecture of the final application. The involvement of users since the very beginning of the system design (i.e. the adoption of a User Centered approach) can greatly enhance the effectiveness of a LEADS: user needs can pervade the design and implementation of all the basic functionalities of the tool. Such an involvement has an impact on the design of the system: each typology of user should have available all the relevant tools for their work. Moreover, it is important that the system is friendly, and easy to get accustomed with. Since situations can be envisaged in which the user is not a full-time LEA developer, the system must also be perspicuous and intuitive enough to support her/him on a time-to-time basis. These results depend on the availability of a careful analysis of the development cycle of LEAs; only such an analysis permits to single out basic (even if abstract) typologies of users defined in terms of their skills, the tasks they accomplish, and the like.

Important support to these considerations comes from the field of human-computer interaction (Carmel et al., 1993). As a matter of fact, it is generally acknowledged that approximately 60-to-80% of all problems of information systems can be traced to poor or inadequate requirement specifications, including both lack of facilities and usability problems. What is needed is to involve day-to-day work experience early in the project, when the basic design choices are made. Positive effects of User Centered approaches for the design of information systems are not limited to usability: benefits were discovered (Nielsen, 1993; Carmel et al., 1993) to be connected to time and cost saving during development, completeness of system functionalities, repair effort savings as well as to user satisfaction.

The purpose of this paper is to describe a LEADS, called GEPPETTO, which has been developed with the aim of addressing these issues by adopting a User Centered design approach (Norman and Draper, 1986). Such a move resulted in an environment that: (A) supports the whole process of designing and developing LEAs; (B) provides the different typologies of users with dedicated facilities which are suited to their skills and backgrounds; (C) improves on the training phase.

2 Methodology

A User Centered (UC) approach was adopted for the design of GEPPETTO. Indeed, UC approach takes user needs into account from the very beginning of the design phase till the final evaluation of the system. This way, system design is changed from a mere technical and individual activity into an interdisciplinary and group activity. Importantly, UC approach secures the attainment of such goals as: appropriateness with respect to user needs and desiderata; flexibility with respect to different skills and different user typologies; and overall usability.

The engagement of users in the system design can occur at different levels: *consultative level* when users are considered as sources of information or as participants to final evaluations; *representative level* when they participate in structured design meetings; *consensus level* when users are part of the design team.

For GEPPETTO we chose the *Participatory Design* methodology (henceforth PD, (of the ACM, 1993)) which falls in the third class. In PD, users act as fully empowered participants to the design process, sharing decisions together with system designers. Such a working style promotes mutual learning between users and designers, and facilitates the identification of user needs and of possible misunderstandings. Hence PD is particularly suitable for complex domains where it might be difficult for designers alone to get a knowledge sufficient for proposing meaningful solutions. This is certainly true of LE, because of the complexity of the domain and of the different skills involved in the development of a LEA. Moreover, certain peculiar techniques of PD, such as participatory prototyping, offer a natural way to reduce errors otherwise not detectable until the final system is put to use.

PD employs a wide range of techniques (Muller et al., 1993) whose applicability depends on such factors as design goals, group size, availability of users for long periods, and the like.

Concerning GEPPETTO design, PD was implemented by establishing a working group (WG) of five people consisting of system developers (2), users (2), and an interface design expert (1). Different techniques were applied at different stages of the design process:

- **Envisioning future solutions**: in the early phases, informal discussions for clarifying global statements and stimulating the users' creative thinking took place. Outcomes of these meetings concerned the awareness of the different roles, skills and knowledge involved in the development of LEAs, and the identification of a number of basic user typologies. Moreover, it seemed unlikely to the WG that a single person could cover the whole process alone. That is, the development of a LEA is a multidisciplinary work which can benefit from some kind of support in its cooperative evolution.
- **Participatory requirement specifications**: the discussion focussed on users desiderata and system capabilities and resulted in a list of the required functionalities. Such a list was then divided into subsets, each one corresponding to one of the user typologies. The discussion then centered on how each typology acts in isolation, and how it interacts with the others, during the development of a LEA. Thus, different levels for single and cooperative work were identified.[1]
- **Collaborative low-fi prototyping**: during collaborative prototyping workshops, paper mock-ups (also called low-fi prototypes) were designed and evaluated by the WG. This activity was extremely useful to move from ideas to concrete interface design, to detect and correct misunderstandings, and to elicit original solutions to unforeseen problems. The outcome was the complete definition of the system.
- **Cooperative evaluations**: cooperative evaluations of the first implementation supported the final refinements of the implemented environment. At this stage, feedbacks from the users were discussed and taken into account for further improvements.
- **Experimental sessions**: even if not required by PD, empirical evaluations with users not involved in PD have been conducted to verify the effectiveness of the design. Method and results are discussed in Section 7.

In the next section we will focus on the results of the first two steps of the PD methodology, as applied to GEPPETTO design.

3 Users, Tasks and LE Systems

The discussion inside working group was organized around three main (sequential) topics:

- the construction process of LEAs: development cycle, involved skills and tasks, etc.;
- user desiderata: rapid prototyping, graphical interfaces, openness of the architecture, deliv-

[1]The present version of GEPPETTO does not provide features for advanced cooperative work.

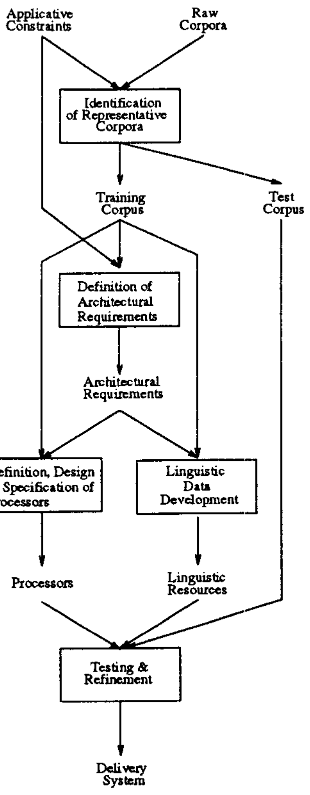
Applicative Constraints
Raw Corpora
Identification of Representative Corpora
Training Corpus
Test Corpus
Definition of Architectural Requirements
Architectural Requirements
Definition, Design & Specification of Processors
Linguistic Data Development
Processors
Linguistic Resources
Testing & Refinement
Delivery System

Figure 1: Development cycle of LE applications

ery systems, modular approach to linguistic development, etc.;

- specifications of system facilities: tools for browsing and editing linguistic data, API for integrating external resources, etc.

3.1 Building LE Applications

The working group focused on the abstract definition of the development cycle of LEAs and of the typologies of the involved users. As a matter of fact this is a requirement of an LE approach to NLP systems.

The development cycle of LEAs was defined as in figure 1.

As a first step, *applicative constraints* must be considered. In fact, the working context of a LEA determines not only the global behavior of the LEA, but also the way the different modules interact to produce the desired behavior. Another preliminary step is the collection of *raw corpora.*

After such a preparatory work, the following development cycle typically takes place:

- *identification of representative corpora.* In this step the aforementioned raw corpora are classified and filtered to find a set of examples that is representative of the characteristics of the whole corpus. The resulting corpus is then split in two parts: one to be used during the system development (*training corpus*), the other during the testing phase (*test corpus*);
- *definition of the architectural requirements.* Given the applicative constraints and the characteristics of the corpus, the specific requirements of the LEA are defined;
- *definition, design and implementation of the processors,* according to the requirements of the previous point;
- *development of the linguistic resources,* according to the requirements arising from the previous analysis;
- *testing and refinement* of both the processors and the data collection.

Once all these steps have been gone through, the resulting architecture is delivered (*delivery system*) and customization can start.

The working group singled out three different user typologies which can play a role in the tasks above. Each of them corresponds to different backgrounds, knowledge and skills:[2]

- Linguistic EngineeR (LER): expert on architectures for LE. Background: computer science; knowledge of computational linguistics;
- Computational Linguist (CL): expert on linguistic data development. Background: computational linguistics; little knowledge of computer science;
- Processor Manager (PM): expert on processors for language processing. Background: computer science; knowledge of computational linguistics.

Accordingly, the development cycle has been refined as follows:

- identification of representative corpora: LER interacts with CL to provide a representative corpus for the application;
- definition of architectural requirements: given the corpus and the requirements for processors and linguistic data, LER interacts with PM and CL to define the correct architecture;
- definition, design and implementation of the processors: PM chooses (or designs and implements) them;
- development of linguistic resources: CL chooses (or designs and implements) them;

[2]Actually the working group added also an Application Manager, i.e. an expert of the domains and of the users of the LEA. Such a profile is not discussed in this paper.

- test and refinement: LER checks the correspondence between the current implementation and the architectural requirements; the processors are tested by PM and the data collection by CL.

In the end, the working group had effectively specified the actions, the tasks, and the skills required to create LEAs. The following step was the identification of the user needs.

3.2 User Needs and Desiderata

The working group discussed some of the desirable features of a LEADS, from the point of view of the users. Results can be summarized as follows:

- facilities for the rapid prototyping of LEAs via graphical interfaces;
- facilities for choosing among resources (e.g. lexica and grammars) provided by libraries of linguistic data;
- specialized graphical browsers and editors for linguistic data;
- facilities for interactively testing and debugging processors and data;
- facilities for testing and debugging the whole architecture against test suites;
- aids for providing the delivery system;
- facilities for integrating processors and data different from those already provided by the environment;
- facilities for integrating knowledge stored in external modules (e.g. Knowledge Bases).

One of the main outcomes of PD discussions was that the different users would benefit from a single, common tool capable of facilitating and supporting their mutual interactions (even when performing their tasks independently) as well as the integration of resources developed independently.[3]

On the other hand, given the different profiles and skills involved, each of the three user typologies needs different facilities and might prefer different interaction modalities. For example CLs tend to favor graphical interfaces that hide as much as possible low-level details (e.g. internal data representation). On the other hand, PMs have to cope with low level details. As it turns out, the ideal environment should both address the differing interaction styles of each user, and, at the same time, provide a uniform environment where their contributions can be easily integrated. These results can be obtained if, at any time, the user can select all and only the functionalities he/she actually needs.

A similar tension involves also linguistic data and processors. LERs want to see them as units that can be assembled to build the final architecture. PMs are inclined to consider the linguistic data as a unit, but see the processors as complex modules to manipulate. Finally, CLs obviously must be able to single out pieces of linguistic data and organize them in a significant way, while using the processors as black boxes.

Before discussing how user needs have been implemented in GEPPETTO, we briefly introduce the formalism for linguistic data as it was developed by the CLs of the working group.

4 The Formalism for Linguistic Data

CLs participating in the working group suggested a Typed Feature Logic oriented (Carpenter, 1992) formalism. The reasons were as follows:

- TFL formalisms provide a way for breaking down the structure of linguistic data, allowing for a clear separation between the description of abstract linguistic types and that of grammatical rules and lexical entries.[4] This facilitates knowledge encapsulation as well as a modular architecture of linguistic data. Such a modularity can play an important role in the reuse of existing data;
- typing secures to a high degree the consistency of the linguistic data. This speeds up the process of data editing and debugging;
- the formalism is well known and many basic unification algorithms are available;
- it meets the demands of many current linguistic theories, e.g. LFG, GPSG, HPSG, etc.

TFL specifications are compiled into a graph format, where each node represents a Typed Feature Structure (TFS). Types and the type hierarchy have been implemented by adapting the encoding schema proposed by (Ait-Kaci et al., 1989) to the TFL format. This permits to efficiently handle very large type hierarchies as well as to account in a straightforward way for type disjunction. The standard TFL formalism has been modified to accommodate:

- *Declaration statements* specifying, for instance, that a certain object is not an ordinary TFS. In case its properties must be assessed by other,

[3]In this paper we focus on the interactions among users belonging to the different typologies and on the integration of their work. We will not address the important question of how to support the interactions and integration involving users of the same typology. For instance, we will not discuss here the issue of how the development of large grammars by different CLs can be properly supported by a LEADS.

[4]In this respect, CLs strongly suggested that some phrase-structure-like skeleton should be provided. This seems to better address their expectations and ways of thinking than a system in which grammar rules are absent, as it is normally possible in type-oriented linguistic formalism (e.g. HPSG).

possibly external, modules; such a fact can be specified by means of *external constraints*;

- *External constraints* providing explicit links to external modules, e.g. morphological processors, independent KBs, etc.;
- *Directives* for the unifier. For instance, it is possible to force the unifier to consider in the first place the paths that have been observed to cause more frequent failures (Uszkoreit, 1991).
- *Macros.*

Declaration statements and external constraints greatly enhance the modularity and portability of the LEAs developed by means of GEPPETTO, by allowing the reuse of existing processors and/or data.

5 The GEPPETTO Environment

In this section some of the characteristics of GEPPETTO are outlined, focusing on those aspects that specifically meet user needs. A more detailed description of GEPPETTO is contained in (Ciravegna et al., 1996).

In GEPPETTO an application consists of two main parts: a (set of) processor(s) and a *Linguistic System*. The latter is the collection of all the sets of linguistic descriptions relevant for the characterization of a given corpus. Given the kind of formalism adopted, namely TFL, a Linguistic System consists of: a type hierarchy, a grammar, a lexicon, and a set of macros. The concept of linguistic system is not simply conceived as a set of the four components just mentioned but it is a complex object with a central role in GEPPETTO: much of the development of LEAs is centered around linguistic systems. CLs edit, browse, and update linguistic systems. They can reuse existing linguistic systems, or parts thereof, to produce new ones.

GEPPETTO maintains a conceptual distinction between browsing/editing and testing/debugging. Actually, browsing/editing can be performed independently by different users, whereas testing/debugging can require a strict cooperation between different typology of users. This is so whenever an error is due to unexpected interactions between data and processors. These observations emphasize the advantage of a single environment for the whole development cycle: different users have dedicated facilities for development, but a common environment for integrating and testing.

We now turn to a discussion of the facilities and tools provided to the different users.

5.1 Supporting the Linguistic EngineeR

LER main task is the study of architectural requirements (together with PM). He/she also controls the compliance of the LEA with the initial requirements. To this end, GEPPETTO provides support for: (a) the rapid prototyping of architectures by assembling already existing processors and linguistic systems, and (b) tests against a test corpus. Both data and processors are seen by the LER as black boxes that can be combined by means of a graphical interface.

When the architecture meets the requirements, a *delivery system* can be produced. It contains the selected linguistic system and processor(s), and excludes the GEPPETTO development environment.

5.2 Supporting the Processor Manager

PM task is to identify the processors that can satisfy the architectural requirements. She/he can choose among the processors made available by GEPPETTO[5] or link external ones to the environment. In the latter case, an API is provided to connect the external processor to the GEPPETTO world. Once a new processor has been properly linked, it is completely identical to the other default processors: it can be selected via the graphical interface, it can take advantage of the debugging/testing facilities, and so on.

Via API, it is also possible to interface LEAs with other kinds of external modules, e.g. modules which make available functionalities not provided by the environment (e.g. Knowledge Bases or morphological analyzers).

PM can also choose among different unification algorithms that have been designed to:

- carefully control and minimize the amount of copying needed with non-deterministic parsing schemata (Wroblewski, 1987) (Kogure, 1990);
- provide a better match between the characteristics of the unifiers and those of the linguistic processors. Indeed, different linguistic processors may profit of different unification algorithms. The availability of different unification algorithms allows the user to choose the one which best fits the needs of the particular linguistic processor at hand.

5.3 Supporting the Computational Linguist

A considerable amount of effort has been devoted to create suitable (specialized) graphical tools for CL. Recall that CL main task is to build a linguistic system satisfying the application requirements. The graphical environment must allow CL to ignore low-level details as much as possible, and concentrate on the linguistic aspects of data description.

CLs can build a linguistic system both by pasting already existing components (and modifying them

[5]At present, GEPPETTO features two chart-based parsers (a bidirectional Head-Driven Bottom-Up (Satta and Stock, 1989) and a CYK-like) and a Head-Driven Bottom-Up non-deterministic generator (Pianesi, 1993). We plan to make available a wider array of processors in the near future.

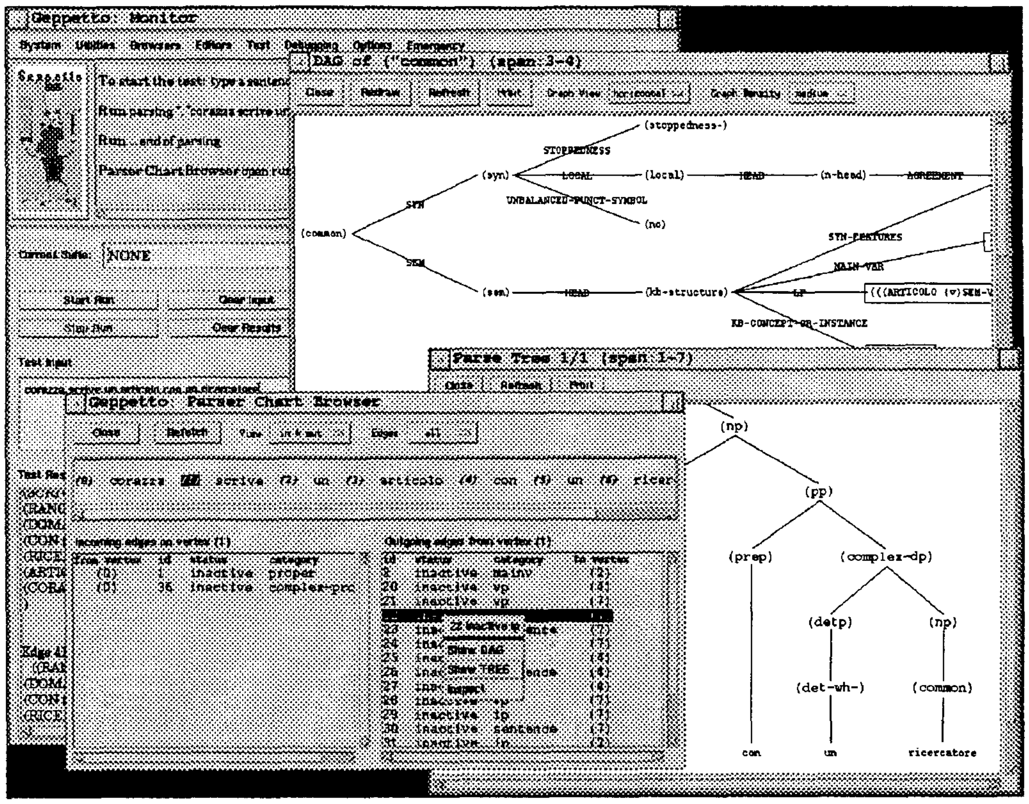

Figure 2: GEPPETTO during a debugging session.

when necessary) and by building it from scratch.[6]

As the data forming the parts of a linguistic system differ in attributes, global organization and functions, specialized graphical tools have been designed for browsing/editing the type hierarchy, the grammar, the lexicon and the macros.

The main tools for the CL are:

- a grapher for browsing and editing the type inheritance hierarchy. It displays mouse sensible nodes and allows to add/delete/modify nodes, as well as to modify the hierarchy itself;
- browsers for data sets such as lexicon, grammar and macros. They allow to add/delete/modify/copy elements in the data sets, as well as to undertake actions on the data set as a whole (e.g. compiling it);
- editors for editing and modifying properties of single lexical entries, grammar rules, macros and type hierarchy nodes. They include editors for TFL descriptions, feature appropriateness statements, etc. TFL-syntax error checking, TFL description compilation and TFS visualization are supported. Documentation and comment notes can be attached to each item;
- interactive and post processing debugging tools (at now mainly a sophisticated chart browser).

[6]Currently GEPPETTO provides some standard resources for Italian: a type hierarchy, two lexica and two grammars.

Facilities are also supplied for computing statistics about performances on test suites. In particular, it is possible to detect points where unification failures arise. Such results can be exploited either to hand-tune the linguistic systems to the corpus needs, or by feeding them into a module which forces unification algorithms to consider unification failure hypothesis first, this way speeding up the whole processing.

6 PD at Work: the Debugging Tools

The PD working group suggested to divide the tools for testing and debugging into interactive facilities (such as tracers and steppers to follow the application of grammar rules during processing), and "post-processing" facilities. In the working group it was agreed that naive interactive tools can be quite difficult to be fully exploited given the great number of rule applications and unifications that happen during parsing. In order to reduce the number of rule applications and unifications traced, it is necessary to have a very powerful (and complex) language which makes the user able to program the tracer; but usually tracer's expressive power is quite difficult to be fully taken advantage of. Moreover, it is important that the tools are (relatively) easy to use, so that they can be usefully exploit also by users not necessarily expert of that particular tool or by time-to-time users. Given these considerations and also the fact that all the processors currently available are chart-based (and hence all the results produced during processing are still available at the end of the processing itself), the discussion focused on the post-processing tools.

Within such tools, the chart browser plays a central role. To better discuss its characteristics, paper mockups were jointly created and evaluated. Such an effort produced a highly detailed description of the tool functionalities and of its layout; in particular, the kind of information and actions (showing parse/generation trees, TFS descriptions associated with edges) to be made available to the user, the different viewpoints on edges and vertices, etc.

As it turned out, the chart browser window is the starting point for the exploration of the structures produced during processing. The tool (cf. figure 2) allows the user

- to see the edges either in a strictly sequential way or as organized around the objects connecting them (i.e. vertices for the parser and constraints for the generator);
- to filter edges according to their type (active/inactive edges), to their categories, etc.;
- to browse through the wide and complex data structures produced;
- to activate auxiliary tools.

The chart browser is a fairly standard debugging tool; in GEPPETTO the adoption of a User Centered approach permitted to design a flexible and extendible tool, which is the central structure for browsing through the elements built during processing.

Besides the chart browser facilities described above (and already implemented), the working group faced the problem of how to single out the failures happened during parsing and to understand their causes. Browsing edges in the chart it is possible to identify (guess) possible error points and to concentrate the focus on them: it was envisaged the possibility of selecting some edges in the chart and run the parser on them in a special mode. During this special running mode GEPPETTO reports diagnostic messages on the causes of the failure: missing grammar rules/lexical items, failure during unification, etc. If the failure is due to unification, the involved paths are reported.

7 Evaluations with Users

The implemented system was assessed by means of a *formative evaluation* (Nielsen, 1993), to test its general usability and the quality of the proposed solutions.

Participants The testing group consisted of eight people from our department. Participants had different degrees of expertise in NLP, though none of them had ever used GEPPETTO before, nor had participated in the PD process. Participants were not required to have any previous knowledge of the TFS formalism.

Procedure Users were given the manual in advance but they were not required to read it before the test, nor any training occurred before the testing phase. During the experiment, users were allowed to freely consult the manual. Each participant was asked to perform 4 tasks:

1. architecture definition and composition: the participant was required to create her/his personal LEA by composing existing linguistic resources and processors to form a new architecture;
2. lexicon update: the participant had to insert lexical entries in the lexicon, and to run the parser over a sentence containing the new terms;
3. hierarchy manipulation and grammar update: the participant was asked to modify the type hierarchy by acting on its graph. Furthermore, she/he had to modify the grammar. Finally, by browsing the resulting parse tree, the subject was asked to verify the correctness of the changes;
4. test suite run: lastly users had to load an existing test suite (a file), to add the sentence of task 2 and to run their architecture over it; results of the running had to be saved in a log file.

During the experiment, participants were requested to verbalize their thoughts. This method, known as *thinking-aloud*, permits an easy detection of the problematic parts of the human-computer interaction as well as to understand how users perceive the system (Nielsen, 1993). An experimenter sat near the participant, taking notes on occurring problems and stimulating the subject to express her/his thoughts. After the test phase, the experimenter interviewed each participant, discussing the problems she/he run into, gathering suggestions on possible improvements, and investigating user satisfaction. All performances were videotaped to allow successive analysis.

Results The choices done and implemented into GEPPETTO supported naive users in moving around and acting in a complex and unfamiliar environment. Even participants who had not read the manual and had only a little experience in NLP were able to complete the tasks in less then one hour.[7] Through

[7]This time is definitely low considering that users were required to comment their actions, were allowed

observations and interviews it could be verified that participants reached a good understanding of the system and judged it positively.

Some weaknesses in the interface design were identified: problems mainly limited to common graphical user interfaces mistakes, e.g. lack of feedback in resource status, and to the understanding of the terminology developed during PD (*naming problem*) emerged. Identified problems may be solved with a limited revision of the graphical interface.

Experiments demonstrated that the adoption of PD can bring intuitiveness also in the design of a complex LEADS: even users without any experience with GEPPETTO and limited knowledge in NLP were able to easily understand the system organization and to effectively use its tools to accomplish non trivial tasks.

8 Conclusions and Future Work

In this paper we have discussed the importance of user involvement in the design of a LEADS and exemplified it by discussing our experience with GEPPETTO.

The PD methodology enabled users to express their desires and needs while participating to the design phase. This permitted to create an environment whose facilities are suited for each of the users/tasks involved in the development of a LEA. The design work started from very general issues (e.g. the definition of the development cycle) and went into very specific details (e.g. the functionalities associated with the buttons of each window).

It must be stressed that a crucial role was played by the interface design expert, who organized the many different ideas in a concrete and coherent interface layout, provided the necessary insights to analyze user-machine interactions, and continuously stimulated the active cooperation within the working group.

GEPPETTO has been implemented under Allegro Common Lisp and runs on SUN SPARCstations. The graphical facilities have been implemented by means of CLIM and Grasper.

GEPPETTO has been used in the development of a number of applicative projects, in different application domains, including multi-lingual text generation (LRE-GIST), information extraction from agency news (LE-FACILE), and Natural Language information query (LE-TAMIC-P); all these projects have been funded by the European Union.

Future work on GEPPETTO will address a number of important pending issues. Among them it is worth mentioning: the full implementation of the debugging tools suggested by the user group and the implementation of a number of facilities to improve GEPPETTO's capability of supporting the design of LEA architectures.

to consult the manual and were stimulated in exploring GEPPETTO.

References

Hassan Ait-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. 1989. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146.

Erran Carmel, Randall Whitaker, and Joey George. 1993. PD and Joint Application Design: A transatlantic comparison. *Communication of the ACM*, 36(4):40–48, June.

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

Fabio Ciravegna, Alberto Lavelli, Daniela Petrelli, and Fabio Pianesi. 1996. The Geppetto Development Environment. Version 2.0.b. User Manual. Technical Report 9608-10, IRST, August.

Kiyoshi Kogure. 1990. Strategic lazy incremental copy graph unification. In *Proceedings of the International Conference on Computational Linguistics*, pages 223–228, Helsinki, Finland.

Michael Muller, Daniel Wildman, and Ellen White. 1993. Taxonomy of PD practices: A brief practitioner's guide. *Communications of the ACM*, 36(4):26–28, June.

Jakob Nielsen. 1993. *Usability Engineering.* Academic Press.

Donald A. Norman and Stephen W. Draper. 1986. *User Centered System Design: new Perspectives on Human-Computer Interaction.* Lawrance Erlbaum Associates.

Communications of the ACM. 1993. Special Issue on Participatory Design, June.

Fabio Pianesi. 1993. Head-driven bottom-up generation and Government and Binding: a unified perspective. In Helmut Horacek and Michael Zock, editors, *New Concepts in Natural Language Generation: Planning, Realization and Systems*, pages 187 – 214. Pinter Publishers, London.

Giorgio Satta and Oliviero Stock. 1989. Formal properties and implementation of bidirectional charts. In *proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI.

Hans Uszkoreit. 1991. Strategies for adding control information to declarative grammars. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 237–245, Berkeley, California, USA.

David A. Wroblewski. 1987. Nondestructive graph unification. In *Proceedings of AAAI-87*, pages 582–587, Seattle, WA.