An Environment for Cooperative Construction of Natural-Language Requirement Bases*

Vincenzo Ambriola Vincenzo Gervasi

Università di Pisa
Dipartimento di Informatica
E-mail: {ambriola,gervasi}@di.unipi.it

Abstract

To match the growing complexity of modern software systems, one needs to perform a fast, efficient, and complete requirement analysis. The purpose of this analysis is to produce a set of requirements acknowledged by all the participants in the definition process.

To make cooperation possible and productive, one needs adequate environments supporting the participants, capable of timely identification of conflicting, vague, or imprecise requirements, and able to offer different "views" on the requirement base to each participant.

To this end, we present the architecture of such an environment that we are building, together with the findings from some experiments on the use of a prototype.

The prototype accepts simple natural-language requirements (in Italian) and produces graphical representations of certain classes of information. Natural-language requirements are interpreted using a domain dictionary and a set of fuzzy-logic rules; each rule is composed of a pattern (matching a natural-language construction) and a generic action (in our experiments, actions build a data flow diagram and an entity-relationship representation of the requirements).

1. Requirement extraction

1.1. The birth of a new software system

Any industrial entity planning to adopt a new software system invariably has to face expenses, risks, and operating problems. The only justification to these problems lies in the promise of the advantages of the new system. We can distinguish three classes of advantages:

1. running costs reduction;
2. performance enhancement;
3. availability of new functionalities.

These advantages implicitly refer to a *pre-existing system*, to which the benefits of the new system are compared. In the great majority of cases, one wishes the new system to be at least as powerful (with regard to the advantages listed above) as the preceding one.

It is not necessary for the old system to be exclusively a software system; indeed, the most common case consists in the substitution of a (complex) bureaucratic apparatus with a software system (e.g., when networks and e-mail substitute the traditional Royal Mail document transmission system). In the same vein, it is not necessary for the new system to be exclusively a software system, either. A system (*tout court*) can well include new procedures for interested personnel, new technical tools, and so on.

1.2. Analysis of the pre-existing system

In all these cases, the pre-existing system analysis is a *key step* to perform in order to build the new system. This analysis is quite a different process from the requirement analysis of the new system and requires peculiar competences. Indeed, a successful pre-existing system analysis carefully describes the operations of a concrete system, that often cannot be found in a codified form altogether. Common practices, cultural background, and plain experience are the fundamental ingredients to a complete analysis. On the contrary, requirement writing benefits from a certain degree of ignorance of the system described therein (which, by the way, does not exist yet): only with this premise one can be reasonably sure that all the knowledge needed to define the new system is actually contained in the requirements.

From the above considerations, we can conclude that the old system analysis can only be performed by someone already familiar with it (agents, users, participants to various degrees), but that they are not well suited to writing

*This work was partially founded by MURST, Progetto V&V.

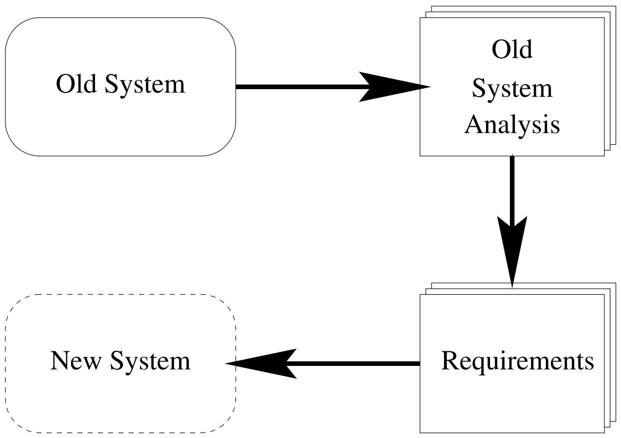

Figure 1. The introduction of a new software system.

requirements for the new system, as too much information is kept on their part "on the background", and not explicitly written down in the requirements. One can easily figure out the consequences that this scenario can bring to the design and coding phases of the development process, when the system goes into the hands of people with no knowledge whatsoever of the domain, apart from that coming from the requirements: unspecified behaviours, arbitrary decisions, anomalous reactions to exceptions are but a few of the possible outcomes.

It is hence necessary to develop, verify, and adopt new methodologies that can simplify (and thus make more controllable) the migration process from a simple analysis (of the old system) to a set of requirements (of the new system). It goes without saying that the requirements must fulfill all the needs of completeness, adherence, consistence, precision, and correctness that are usually desired in such documents. One could perceive here a certain overlapping between the role given to the requirements and the one typically assigned to the system *specification*. This overlapping is not due to a careless confusion of roles, but rather to the practical observation that most often the customer cannot really understand and validate a formal-language specification, which is the only way to assure, if not the correctness, at least an unambiguous interpretation. Due to this fact, a formal-language specification can be considered more an internal process document (we could say, a sort of contract between the author of the requirements and the designer) than a real opportunity for the customer to check and verify the new system behaviour in the first stage of its life cycle.

Given that we will ask someone who is not necessarily technically educated to participate in writing the requirements, we should consider the problem of identifying a suitable language. Obviously enough, in the greatest majority of the cases the most natural expressive means is the *natural language* itself (free text). Indeed, we cannot ask everybody to learn and make profitable use of a formal or even semi-formal language, although there have been several attempts in this direction[1].

On the other hand, one cannot postulate that a requirement engineer may become expert in every possible domain and, even if we consider a kind of single-domain specialist, most certainly he cannot become an expert of every problem posed by a particular customer's system. This is true even if the classical methodology (study of the user's documentation, interviews with the customer...) exactly prescribes this approach.

1.3. Cooperation and its problems

It appears clearly the need for a *cooperative* approach to requirement drafting: on one hand, only the customer has the *knowledge* of her current environment, her problem, and its domain; on the other, only a trained requirement engineer has a *methodology* for successfully expressing, validating, and formalizing these requirements.

Our target of maximal efficiency and swiftness in the construction of a set of requirements must also keep into account several problems related to the cooperative nature of the drafting process:

- It is often the case that the system to produce, or its domain, are of such complexity as to require splitting in sub-domains, each covered by the competence of different individuals. As a consequence, each of them will have to participate in the drafting process. The dual consideration beholds with regard to the production of the new system, that can be split in sub-systems entrusted to different people (even outsourced to different organizations altogether). Again, each of the participants will want to have a word on the requirements, at least to extract from the set those of their own interest. The drafting process is clearly a *multiple-actor* one, and we cannot limit ourselves to a single customer – single designer model.

- Different people have different agendas, obligations, and working habits; the drafting process can be far more efficient if we make it possible for each of them to work on the requirements at different times, based on their own preferences. Thus, while some actual meeting will be needed from time to time, we should head towards an *asynchronous* drafting process.

[1] A typical example is the Shlaer-Mellor method [10], that adopted a graphical representation aiming at immediate comprehensibility. Sadly, practice and experience have shown that it really requires taking specialistic courses to become proficient in its correct use.

- For reasons similar to the ones given above, it is quite common that participants in the drafting process are spatially separated. The actual distance can go from a couple of desks or a few floor of a building to a cross-oceanic gap, and therefore we should allow for a *distributed* drafting process to accommodate all cases.
- As we have seen in section 1.2, one can assume that the generic customer will find it more comfortable to express her ideas on the requirements using natural language. On the other hand, a requirement engineer will want to use some more formal representation, and different "experts" will have each their own specialized languages, too. Many of these "languages" will not be traditional languages altogether, but rather graphical representations of some concept, not even necessarily shared with anyone else in the process (e.g., one could use a particular representation only to make up his mind). Having to support such different representations, our process should allow for a *multiple-view* model, and let different participants exchange their ideas using each one his preferred "language".

The analysis of a system, with the purpose of building a set of requirements for a (prevalently) software system that should substitute it, is thus confronted with a wide range of difficulties and must satisfy several, often contrasting needs. This process needs the widest available support through the use of adequate *environments* and *tools*. In the rest of this paper we will discuss the architecture and the expected effectiveness of an environment of this kind we are building.

2. A cooperative environment

Some of the difficulties we have mentioned can be solved quite efficiently with existing tools and techniques; we would not render ourselves a good service should we choose not to profit from these.

To allow for a multi-actor environment, we will build a *client-server* architecture, with an unspecified number of *peer clients* and a single *administrator*, empowered to perform the usual mundane tasks. This will also allow the environment to scale easily to drafting processes of varying sizes.

Asynchronous cooperation can be achieved by the use of tools already in use, in similar conditions, for the coding phase. We are planning to use *CVS*, the Concurrent Versions System by Brian Berliner [2], a tool that seems adequate to our purposes. CVS allows each participant to work (experiment) on a local copy of the requirements, and commit the changes only when satisfied with them. CVS also manages conflicting changes, asking interested parts to resolve them, and can manage several versions and configurations of a requirement base. This can be useful both for historical and statistical purposes, and for managing the requirements for different configurations of a software system (e.g., the requirements for similar, but slightly different, power plants of an electricity supplier).

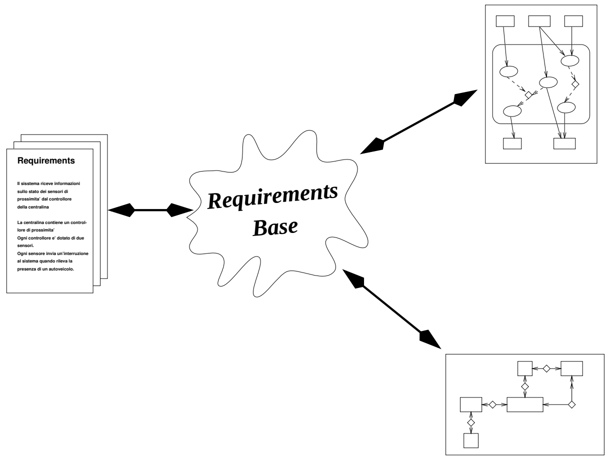

Figure 2. Different views on the requirements.

The distribution problem is actually a non-problem: network evolution has made physical distance almost ininfluent. To allow for the widest possible access to the clients, we choose to use hardware- and platform-independent tools, languages and protocols; in a word *Java* [6] and *WWW*. These choices bring several other advantages we do not need to pinpoint here.

Unfortunately, there is no shrink-wrapped solution to the problem of giving multiple views on the requirement base (fig. 2). We are developing *ad hoc* tools in support of our environment; our target is to provide, behind the natural language representation, at least diagrammatic representations of data-flow, entity-relationship, temporal ordering and dependency, with possibly more specialized representations to come. The system will maintain the consistency between these different representations, and will signal inconsistent, incomplete, or ambiguous requirements.

2.1. Environment architecture

We can now show the architecture of the environment resulting from the above decisions (see fig. 3).

On the server side, we use CVS to maintain and distribute (parts of) the requirement base, transported via HTML/HTTP (for simple browsing) or via custom protocols sitting directly on TCP/IP (for specialized clients). On the client side, we distinguish three classes of clients:

– *extractors* simply get information stored in the requirement base and show them to their users via one of several views, possibly performing some analysis (e.g., metric extraction);

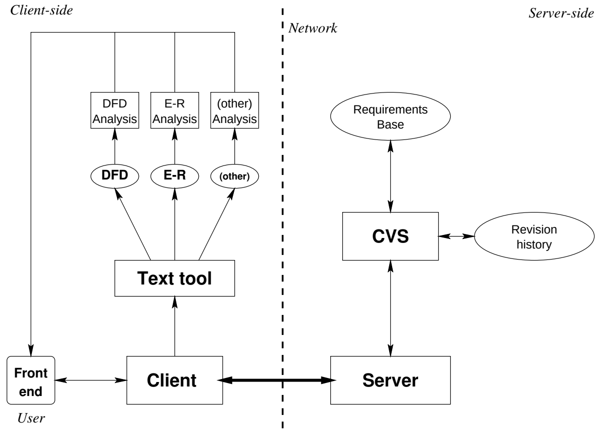

Figure 3. Environment architecture.

– *injectors* only add requirements to the requirement base, may be importing them from other sources;
– *interactors* let the user edit a local copy of the requirements, getting immediate feedback via some of the alternate representations, and possibly performing some analysis on the modified base.

In any case, a text processing engine (text tool) extracts relevant information from the natural-language requirements and transforms them into more formal representations. Each representation is then forwarded to a specialized analysis engine that checks the specific representation for formal correctness, sending any warning or suggestion, together with the representation itself, to the user's front-end. This last component uses the Java Abstract Window Toolkit for its UI, thus assuring platform-neutrality and wide accessibility.

2.2. A prototype

We built a prototype tool to verify the feasibility and usefulness of the environment we designed. Since the most difficult task we were faced with was the multiple-view paradigm, we started tackling this issue. The current prototype accepts simple natural-language requirements[2], can report a number of interpretation problems and extracts from the requirements information as data-flow and entity-relationship diagrams. Requirements are interpreted via a *domain dictionary*, composed of a *glossary*, containing all terms identifying particular domain entities, and of a *verb list* identifying the actions we are interested in. In addition, the tool uses a number of sets of simple fuzzy *rules*, each matching a natural-language *template* with a generic *action*. In the prototype, these actions build a series of tuples describing the DF and E-R diagrams. Ancillary tools translate the tuples to actual graphical representations and report any problem to the user.

During the interpretation, each requirement undergoes a number of processing steps, including:

1. structure simplification, linguistic variety reduction, and stopword (and similar expressions) elimination;

2. identification, categorization, and synonym substitution of the relevant entities, using the glossary;

3. identification, categorization, and synonym substitution of the actions, using the verb list;

4. matching between the requirement and the rule templates, and firing of the corresponding action when a match is found. This step is repeated for every set of rules, thus extracting different kind of information from a single requirement.

In addition, the tool reports to the user "syntactic" problems like missing terms in a matched rule[3].

Obviously, this tool cannot afford *real* natural language recognition; to this end, it would reveal itself totally inadequate. However, his role is that of a place-holder for more complex systems [3, 7], with the added advantage that the rules can be easily modified and adapted to different uses, allowing us to experiment freely and effortlessly. Moreover, *all* the knowledge about both the natural language and the other representations is contained in the rules, with no hard-coded information. This makes the tool easily adaptable to different, more sophisticated uses by simply writing a new set of rules.

2.3. Some experiment

To assess the tool usefulness, we have conducted some experiment.

The first one was conducted with the help of 30 students of the Software Engineering course held at the University of Pisa by one of the authors (V. Ambriola). Each group of 5 students independently produced a set of requirements for a common problem[4], guided only by the recommendation of "keeping them as simple as possible"; user's documentation consisted in almost 200 pages from disparate sources and of varying importance. The suggested methodology prescribed studying user's material, building a glossary "encapsulating" the domain and then writing the requirements using only terms defined in the glossary. Students were given the opportunity to interview a representative of the user to clarify some aspects of the problem.

[2] Actually, the prototype handles the Italian language; however, the tool has no built-in knowledge at all about a particular tongue.

[3] This can happen, since the matching is fuzzy.

[4] The problem concerned a control system for a railway maintenance plant near Milan [4].

We expected the six resulting documents to be reasonably accurate (in preceding iterations of the same process [1, 5] we measured some 500 to 600 hrs/man spent by the students on each document)[5]. To our surprise, applying the prototype tool to these requirements revealed at least some problem in *all* of them. The most common defect was by far the absence of significant requirements, suggesting that the intensive study of the domain had placed the students in the same "mental context" of the customer, turning them into domain specialists with the consequences described in section 1.2. Once the defects were revealed on the graphical representations, corrective actions appeared immediate.

A second experiment was performed on a case study often found in the literature, the so-called "elevator problem". In this problem, one is asked to design a system controlling a certain number of elevators in a building, correctly handling all the usual interfaces (summon buttons, display panels, motors…) and optimally scheduling the elevators stops.

The original statement of the problem, taken from [11], consisted in three pages of narrative description. It was used by one of the authors (V. Gervasi) as a base to obtain a set of requirements for the desired system, together with the associated dictionary. At the same time, the rules were extended to allow for language constructs never found in the first experiment. The effort required to accomplish this task was found to be quite small, and the natural language engine flexible enough to handle almost any new construct needed. As before, the tool revealed a few flaws in the requirements, easily removed by revising them.

The third experiment used the same elevator problem and the rules developed in the second one. The original text was given to two people not involved with the project, but rather experienced in the craft of requirement writing. They were briefly instructed on the particular methodology (glossary, simple requirements), and were asked to write the requirements for the elevator problem. We found that applying the very same rules and verb list developed in the previous experiment to the new requirements, we were able to correctly interpret most of them. We found a substantial agreement between the world-description part of the three analysis, that showed itself very clearly in the E-R diagrams. On the contrary, the data flow parts were quite different, reflecting the different modeling choices of the participants. As found in the second experiment, the first-run requirements were lacking some important information (especially on the scheduling part) that were readily added.

[5] Another, perhaps more convincing evidence that any effort was made to obtain accurate documents comes by noting that the quality of the requirements documents directly influenced the final score given to the students…

2.4. Experiments findings

Our prototype was used in different ways in every experiment.

In the first one, we used it as an *inspection aiding tool*; the students were not informed of the existence of the tool and thus it was used in a *black box* fashion, with no explicit competence required both to the requirement writer and to the inspector. The number of problems exposed by the tool in the students' analysis make us confident that it can be indeed a valuable tool in conducting requirement inspections.

The second experiment was conducted in a *white box* style, since the requirement writer had a detailed knowledge of the tool and of the rules. Its purpose was to test the adaptability of the tool to new domains (i.e., real-time or business systems), and the small effort required let us think that a wide applicability can be attained with a relatively small effort. Even in a white box approach, the multiple-view capability can prove useful, so that the writer can be confident that the requirements indeed reflect his intentions (it was not uncommon for the writer to be surprised at his own words).

In the last experiment, the prototype was used as a tool to *enforce a methodology*; both the rules and the verbs were fixed, and the writers adapted their requirements to the rules. This adaption proved feasible in almost any case, only leaving out those requirements expressing properties not mappable in the target domains (DFD and E-R), and thus not considered by the tool.

Our experiences with the prototype have been quite satisfactory in all three cases. We have tested three important features:

1. the tool's ability to extract information from the text, that was found remarkable even in texts non explicitly produced for the analysis, as long as written to be "simple";

2. the expressive power of the requirements produced with this methodology, i.e., their ability to describe complex systems in a form that is both human-readable and tool-interpretable. We found the expressive power to be rather good, but still not fully satisfying;

3. the tool's effectiveness in finding defects in a set of requirements. Though we are certainly not even near to a complete requirement validation (a feat that, actually, can only be accomplished by the customer), the prototype has proven quite useful in identifying and reporting some class of problems, especially about missing or contradictory requirements.

3 An example[6]

We will show in this section a small example of prototype use. We start with a (greatly) simplified version of the elevator problem in [11], described by the following 13 requirements:

```
# The elevator problem, simplified version.

# The building
The building has 40 floors.
The building has 4 elevators.

# The elevator
Each elevator is furnished with a control panel.
A panel contains 40 destination buttons, one for each
floor.
When a passenger pushes a button, the panel circuitry
sends the number of the required floor to the
controller.
The system maintains a requested stop list based on
the floors required by the passengers.

# The floor
Each floor has a floor panel.
Each floor panel contains a single summon button.
When a user pushes a summon button, the floor panel
sends a call signal to the controller.
Using the call signals, the system defines a call
list.

# Motor handling
To each elevator corresponds a motor controlling its
movements.
The controller sends movement commands to the
elevator motors.

# Scheduling
The controller obtains the schedule from the call
list and the stop list.
```

Feeding the requirements to the tool, we obtain the E-R and data-flow diagrams depicted in figure 4 and 5, respectively. By just looking at figure 5, one could easily figure out the reason of the warnings produced by the tool:

```
* Warning:  "schedule" computed but never used.
* Warning:  "movement commands" has no source.
```

Obviously, we forgot to express the following requirement:

```
The system obtains from the current schedule the
movement commands for the elevators.
```

Adding this requirement makes both the tool and the designer happy.

4. Future directions

We plan to carry on the development of the proposed environment in several directions, that we will briefly discuss here.

[6]All the original texts shown boxed in this section were in Italian, and are translated here into English for reader's convenience.

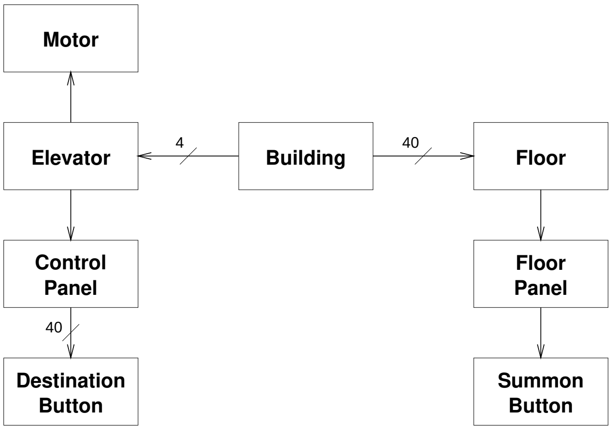

Figure 4. E-R diagram for the sample system.

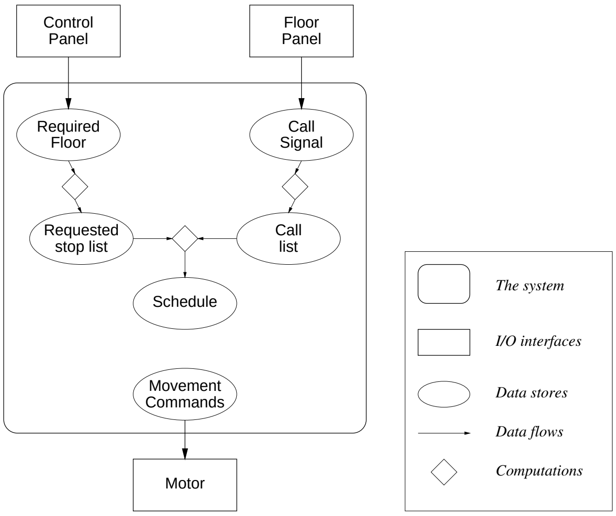

Figure 5. Data-flow diagram for the sample system.

NLP enhancement

The fuzzy text-vs-rule matching algorithm used in the prototype, albeit working finely, is still rather ingenuous. We plan to extend it in several ways (higher-order rules, negative constructions, better handling of ambiguity and non-atomicity).

Analysis and defects reporting

In all experiments, we were able to identify defects in the requirements with a glance at the graphical representations. However, more complex problems will probably require support from the environment to identify problems, report them and suggest corrective actions. Among the defects easiest to identify, we could cite the lack of sender, data or receiver in a data flow, the presence of "dead" (unused) data, the use of "ghost" data that come from nowhere, and the lack of connection between inputs and outputs.

Inverse translation: diagrams → text

While the ability to see graphically the information contained in the requirements is already greatly useful, even more useful would be the inverse translation, from diagrams to text. One could then edit the diagrams and find one's change propagated to the natural-language text of the requirements. This would also make amending the requirements from defects a more natural step. In the current prototype, correcting a problem requires identifying the source requirement and editing its textual representation. With the inverse translation, one could directly edit the graphical representation.

At a first analysis, this appears feasible only on a limited class of corrections, and presents great linguistic difficulties. It will take further work to determine if this translation is implementable at all.

Use of richer representations

The tool usefulness is currently limited by the relative poorness of the domains we are using to express the knowledge extracted from the requirements (DFD and E-R). We need to identify and use richer representations, while maintaining automatic-analysability.

As first candidates, we plan to investigate the use of Petri nets-based representations [8], that can collect information on the control flow and the casual dependency between events, and of OMT [9], that would allow a more profitable use of the information collected on the entities of the system in development.

Knowledge representation

As soon as the system will be able to extract more (and more complex) knowledge from the requirements, a simple textual representation of the requirements (such as the one used in the prototype) will no longer be capable of fully making use of such knowledge. The environment will need to be able to correlate different information, coming from different representations of the requirements, *deducing* from them further properties of the system described by the requirements. To this end, we plan to use a deductive data base; the relevant design choices, however, have not yet been finalized.

Metrics

We would not make good use of the information extracted by the tool without an automatic support to the collection of *metric* data both on the requirements and on the system described by them. Regarding the comparative use of the tool, and only relating to the textual representation of the requirements, the problem has been already tackled in [5], but we expect that more interesting measures can be extracted from the non-textual representations.

Real-world tests

We are preparing a real-world test for the prototype, in connection with an Italian firm working in the database field. We expect to learn much on the industrial effectiveness of our methodology and tools from this cooperation.

References

[1] V. Ambriola, R. Di Meglio, V. Gervasi, and B. Mercurio. Applying a metric framework to the software process: an experiment. In B. C. Warboys, editor, *Proceedings of the Third European Workshop on Software Process Technology*, pages 207–226. Lecture Notes in Computer Science Nr. 772, Springer–Verlag, Feb. 1994.

[2] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Association Conference*, Washington, D.C., 1990. (also available electronically as ftp://ftp.gnu.ai.mit.edu/pub/gnu/cvs-1.9.tar.gz).

[3] M. Costantino, R. J. Collingham, and R. G. Morgan. Natural language processing in finance. *The Magazine of Artificial Intelligence in Finance*, 2(4), Jan. 1996.

[4] Ferrovie dello Stato S.p.A. — Area Trasporto — Settore materiale rotabile e trazione — Strategie, pianificazione e controllo, Firenze. *Maintenance Organization in the IDP Milano Fiorenza and IDP Milano Martesana Plant*, Dec. 1995. In Italian.

[5] V. Gervasi and B. Mercurio. Observing and measuring the software development process. Master's thesis, Università degli studi di Pisa, Corso di laurea in Scienze dell'Informazione, Dec. 1993. In Italian.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[7] L. Mich and R. Garigliano. Design of an object extraction algorithm from natural language requirements. In *Proceedings of AICA 95*, Cagliari, Sept. 1995.

[8] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice–Hall, 1981.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[10] S. Shlaer and S. J. Mellor. *Object–Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[11] E. Yourdon. *Modern Structured Analysis*. Prentice–Hall, Englewood Cliffs, 1989.