Processing Natural Language Requirements

Vincenzo Ambriola   Vincenzo Gervasi
Dipartimento di Informatica, Università di Pisa

Abstract

The importance of requirements, which in practice often means natural language requirements, for a successful software project cannot be underestimated. Although requirement analysis has been traditionally reserved to the experience of professionals, there is no reason not to use various automatic techniques to the same end.

In this paper we present **Circe***, a Web-based environment for aiding in natural language requirements gathering, elicitation, selection, and validation and the tools it integrates. These tools have been used in several experiments both in academic and in industrial environments.*

Among other features, **Circe** *can extract abstractions from natural language texts, build various models of the system described by the requirements, check the validity of such models, and produce functional metric reports. The environment can be easily extended to enhance its natural language recognition power, or to add new models and views on them.*

Keywords: *Natural language, requirement engineering, requirement validation, tools, Web-based environments.*

1. Introduction

While formal methods have witnessed great progress, and their application has resulted in remarkable successes, in the vast majority of the applications it is the customer who has to validate the requirements. Formal methods, indeed, can be applied to models *already* formalized in some way; the very first step, from the wishes of the customer to the first (semi)formal and usually very abstract representation of a problem is often left to the experience, the sensibility and the knowledgeability of the analyst — but better not leave it to his *creativity*!

A number of systematic approaches to requirement gathering, elicitation, selection, and validation have been developed in recent years [28, 29, 33]. However, these approaches often fail to recognize the need for an *informed* validation by the customer who, usually, cannot decode the formal or semiformal models produced by the analyst using the above mentioned methods. Also, in common practice the analysis of a problem has often to start from interviews with the customer or from available user documentation, and both sources of information are heavily based on natural language. Even unorthodox techniques, like capturing the user's work with a videocamera for subsequent analysis [9], need to rely on natural language to assign meaning to the activities captured.

These observations lead us to enhance the role of natural language as a way of expressing requirements. However, natural language has many drawbacks that make it not well suited to the definition of a system; among others, its inherent ambiguity and the difficulty in proving properties of the system so described. In this paper, we consider some properties that a system for processing natural language requirements (NLR) should satisfy, and present a report on our work in building and experimenting with such a system. In this context, we use the term "requirements" to mean not only the final product of the requirement analysis (that we could call the polished requirements), but also early incarnations of the same information — in some case as far as to include parts of the original user's documentation. Our approach is thus to assist the requirement engineer during the entire life cycle of the requirements, proposing some methodology-independent technique to simplify his task.

Section 2 of this paper motivates the need and discusses the role of natural language processing in requirement engineering, and presents the rationale for some of our choices. Section 3 describes the environment we built and used in our experiments. An example, followed by Section 5 on typical applications of the techniques we outlined and by some final remarks, concludes the paper.

2. Using natural language in requirement engineering

2.1. Motivations

2.1.1. Requirement expression. Writing requirements is essentially a cooperative work [5, 14]; only the customer really knows the problem at hand[1], but only the analyst can help her in fully and correctly expressing it. Natural-language is — obviously — the most natural language the customer can use to express what she expects from the system, her perception of the problem, and a minimal model of the environment in which the system to build will work.

Almost always, the customer does not know any different, more formal, language, but should she know any, it will be seldom shared by both the customer and the analyst. On the contrary, natural language (surely the least common denominator in our societies) can be decoded both by the customer and by the analyst, and can assure the communication between them. Sadly, natural language is inherently ambiguous, imprecise and incomplete; often a natural language

[1] Although in most cases this knowledge is far from perfect [27].

document is redundant, and several classes of terminological problems (e.g., jargon or specialistic terms) can arise to make communication difficult. However, we still believe that natural language is the most useful and direct tool for expressing and communicating requirements, and several issues in requirement selection, validation, and conflict identification can be already tackled at this level.

2.1.2. Requirement validation. While the literature offers a plethora of methods and criteria for the validation of requirements expressed in the most varied formalisms [13], none of them can guarantee the actual equivalence of what expressed in the requirements with the (eventually unwritten) wishes of the customer. Only the customer can give the final approval on the requirements, especially in view of any legal implication stemming from this approval, and so she must be put in a position to understand, evaluate, and validate them in their *final* form. One cannot rely on the equivalence between a natural language version of the requirements and a *different*, (semi)formal one, since in this situation the equivalence itself should be validated by the customer (who cannot perform the validation since she does not know the (semi)formal language involved).

This need drives us to consider natural language requirements as the reference version, and to choose natural language itself as the preferred way to communicate with the customer. The equivalence between the natural language version and a (semi)formal one of the requirements can be assured either by unparsing the (semi)formal representation into natural language, or by directly producing the (semi)formal version from the natural language text. This latter approach is the only possible in the first stages of the requirement drafting process, when no formal model is available yet, and calls for some form of "understanding" of natural language requirements.

2.2. Current NLP

Current natural language processing techniques have grown quite sophisticated. From the first successful attempts [3], the field has evolved to handle conversations, viewpoints, beliefs, counterfactual information, and other subtle points. Natural language understanding (NLU) systems are currently in operation, and almost-automatic text translation has become feasible. However, this power has been obtained at the cost of considerable complexity. It is not uncommon for these systems to include AI components, vast amounts of semantic information, statistical data, knowledge bases, theorem provers, and so on, with the aim of inferring as much contextual information as possible from the (vague) source text.

Natural language generation (NLG) has grown to the point where current research issues are about how to plan the best rhetorical structure for a dialogue, and the output generated by these systems appears quite "natural". NLG is generally considered an easier task with respect to NLU, but in this field, too, systems can grow quite complex.

2.3. Peculiarities of the requirements case

Luckily, the particular case of requirements understanding lends itself to several simplifications with respect to the general case of NLU. These simplifications come mainly from three factors:

- **Explicitness**. It is useful in itself to reduce to a minimum the amount of unexpressed information in the requirements. What one could consider common sense could be a real surprise for someone else, especially when they come from different cultural backgrounds. Since our target is not to obtain the most likely interpretation of incomplete or vague requirements, but rather to force the expression of requirements as explicit as possible (to minimize the probability of misunderstandings), an application geared to requirement engineering could dispense with huge knowledge bases like the Cyc ontology [22, 23] or ACAPULCO [19]. In this case, ignorance is a blessing [6].
- **Interactivity**. Requirement drafting is an interactive process. The user can repeatedly modify a requirement until it can be understood by a tool; in our experience, this effort always makes the requirement clearer and often exposes real problems in the original version. Given the interactive nature of the drafting process, a failure in understanding a requirement is not fatal, differently from what happens in the case of non-interactive processes (when there is no way to obtain a different formulation of the text).
- **Repetitiveness**. The application domain, while largely variable, usually falls in a small number of well-know paradigms (reactive systems, data bases, information systems…) [20]. Each of these paradigms can be analyzed independently, and a great part of the experience coming from one such project can be easily ported to a different project using the same paradigm.

Thanks to these simplifications (no need to infer information, non-fatal failures, small number of paradigms), we can use a simpler approach based on the identification of common linguistic structures in the text, avoiding the need for complex NLU systems and unusually large knowledge bases.

2.4. Supporting natural language requirement processing

Table 1 shows a list of requirements that a NLR-supporting environment should satisfy[2]. Most of them stem from the cooperative nature of the requirement drafting process: since both the customer and the requirement engineer must participate, we stress *ease of access* (related to the wide applicability of the tools composing the environment, along several dimensions), *ease of use* (that means having both the customer and the engineer comfortable in their work environment), and *ease of interpretation* of the results.

[2]The third column in this table refers to subsequent sections in which our implementation is discussed.

To allow	we need	provided by Circe as
Using URD, interviews etc.; validation by the customer; customer–engineer cooperation.	Natural language	NLR understanding by **Cico**; NL-like output by the views. (3.1)
Wide applicability (on *working environment* dimension); engineer's freedom.	Methodology-neutrality	Flat structure of views. (3)
Wide applicability (on *language* dimension); multiple language support.	Language-neutrality	MAS rules, predefined glossaries. (3.1.3, 3.4.3)
Wide applicability (on *problem* dimension).	Domains	MAS rules; predefined glossaries; view modules. (3.2)
Wide applicability (on *hardware platform* dimension).	Portability	Client/server architecture; Web interface; Java. (3)
Early applicability (on *time* dimension) in the requirement life cycle.	Accepting incomplete or inconsistent input	Fuzzy matching, abstraction finding. (3.1.3, 3.3.3)
Informed requirement selection	Metric support	MFeP view. (3.3.3)
Requirement validation.	Model diagnosis	Validation views and modules. (3.3.2)
Ease of use.	Simple, fast interaction	Web interface, fast and automatic view building, interactive diagrams. (3.3)
Ease of interpretation.	Transparency	No information hidden by the system, no information injected by the system.
Distributed, cooperative work.	Team work support	Work in progress.

Table 1. Requirements for a NLR-supporting environment.

We found no example in the literature of a system satisfying all these requirements to a reasonable extent, so we decided to build our own. Our aim is to support most typical requirement engineering activities *without* prescribing a specific methodology. Indeed, this aspect is better determined by the engineer's habits, the kind of engineer-customer relationships, and the needs of the whole development process. We make only the very general assumption that requirements undergo a number of refinement steps from their very first form, in some case coinciding with user's documentation, to their final version, that is input to the specification phase. At each step, we assume that the requirement engineer can examine a number of *views*, based on different *models* of the system described in the requirements, graphically or textually representing a model or the results of its validation.

In the next section we present the tools and techniques we have developed to support the requirement drafting process satisfying the requirements in Table 1.

3. A system for NLR engineering

Figure 1 shows the main architecture of the system we present here, called **Circe**. **Circe** offers to the user a complete environment integrating a number of tools discussed in the following sections. Currently, the user can exploit three different interfaces to the system: command-line-based, mail-based and Web-based. We will focus on the latter, since it is the only one allowing both distributed and cooperative work on the requirements [5].

The main tool, called **Cico**, acts as a front-end for the other components. It performs the recognition of natural language sentences, extracting from them some *facts* that it hands (in a suitable encoding we call *abstract requirements*) to the remaining tools (back-ends) for graphical representation, metrication, and analysis.

3.1. Cico: a simple engine for natural language recognition

The peculiarities we discussed in Section 2.3 let us handle the task of recognizing natural language sentences in the requirements via relatively simple techniques, whose combination turns out to be powerful enough for our goals. **Cico** works on the text on a requirement-by-requirement basis. We will not give here a detailed description of the algorithms used by **Cico**[3], limiting ourselves to a general description of its workings.

3.1.1. Preparing the input: glossary and requirements. Central to our work is the idea that requirements are supplemented by a *glossary* describing and classifying all the domain- and system-specific terms used in the requirements. From the methodological point of view, this glossary can be built manually, after studying the domain, or semi-automatically, using an abstraction finding tool [2, 18, 24] on the text of the requirements (see Section 3.3.3). In any case, the user must produce a list of significant terms used in the requirements, marking each term with a (maybe empty) set of *tags* to classify it. Each term can also have a list of *synonyms*, i.e., different terms or whole expressions that can be substituted for the term being defined. **Cico** ignores the actual human-readable *definition* of the term, treating it as a remark, but its inclusion in the same document enhances the ease of maintenance and the evolvibility of the requirements.

Since the glossary contains structured information, it must obey some (rather simple) syntactic conventions, exemplified in Section 4. On the contrary, requirements are just free-text, with the only constraints of auto-completeness (each requirement, individually taken, must be fully understandable) and of the absence of implicit references (pronouns, understood terms etc.). Both the use of synonyms to account for inflectional variants and the missing treatment of anaphora are tied

[3] A technical report on the subject is being prepared [4].

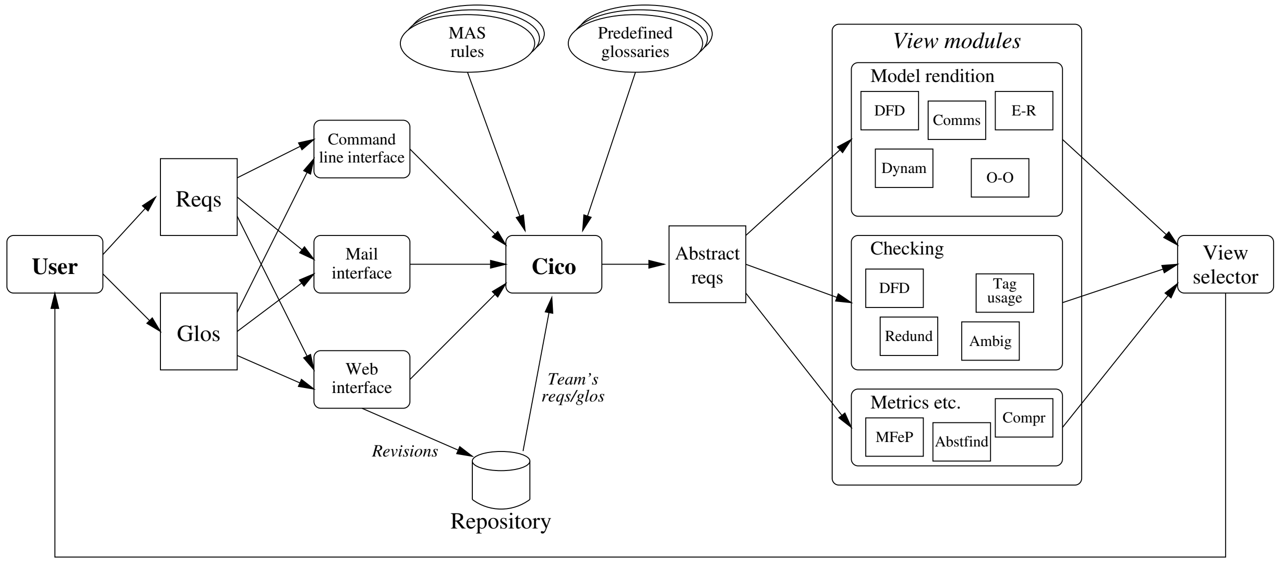

Figure 1. Architecture of the Circe environment.

to the desired property of language-independence, and could be removed by specializing **Cico** for a particular human language.

The requirement part can contain, beside actual system requirements, also a minimal domain description needed to understand the system requirements. As long as the two parts are kept distinct, this helps both human- and machine-analysability of the description without losing focus on the system to design.

3.1.2. Canonization and tokenization. Each requirement undergoes initially a series of simple textual transformations aimed at making the subsequent steps easier. These transformations include, in addition to some typographical operation (accents, white space, letter case, etc.), also the substitution of synonyms defined in the glossary and the simplification of some complex, tied words. In Italian, we handle articles, compound articles, euphonic forms of conjunctions and disjunctions and a small number of other exceptional cases. In this operations, as in the other tools comprising **Circe**, language-specific elements are defined via user-editable configuration files, to allow for easy adaption to other languages. Also during this phase, requirements are tokenized; free-form text is organized into a list of words. In this context, terms coming from a glossary (either the user's one or any of the system-defined ones) are treated as "words", even if composed of several words in the grammatical sense. Words coming from a glossary also get marked with the relevant tags.

3.1.3. Fuzzy matching. The actual recognition is performed by an arbitrary number of sets of *MAS rules*, defined at user level. The MAS acronym refers to the three components of a rule: Model, Action, and Substitution, usually denoted with

$$\frac{\dfrac{\text{model}}{\text{action}}}{\text{substitution}}$$

The semantics of a MAS-rule $r = \langle m, a, s \rangle$ applied to a requirement t is the following: *if a fragment of t matches m, a is executed and the matching fragment of t is replaced by s*. Here is an example of a typical rule:

```
source/OUT SENDS data/INF TO target/IN
---------------------------------------
echo $ID DATAFLOW $source $target $data
=======================================
[output $ID from $source/ACT/EVT]
```

Lower-case words in the model are *variables*; they can match any term in the text that share their same tags or, with a much lower score, any other word. In this example, the tags `/OUT` and `/IN` tell **Cico** that the *source* and *target* must be able to perform output and input, respectively; `/INF` marks terms that have an information content, `/ACT` means that the term is an action and `/EVT` that it is an event. It can be noted in this example that this kind of tagging is strictly semantic. It shares nothing with classical syntax-based approaches, and is quite different from common-sense based ones: one cannot tell from common sense whether "controller" can request interrupts or not, since it is strictly a problem-related property.

The value of a variable (i.e., the actual term that it matched) can be referred to in the action and in the substitution by using the familiar symbol `$`, used also to obtain the value of other system-defined variables like `ID`.

In our example, a matching with the model will cause the execution of the system command `echo` and the substitution of the matching fragment with the text "`output...`".

Model matching is a fuzzy algorithm; it associates to every possible matching a score of similarity with the model. This allows for uncomplete matchings, weighted independence from the order of the words, multiple matchings, and several other kind of imperfect correspondance, greatly enhancing the recognition power of the tool. The added flexibility obtained in this way is more evident when comparing the fuzzy matching algorithm used by **Cico** with classical ones based on formal syntax (even if using a natural language-like vocabulary), like the one used in [15].

Since the firing of a MAS-rule can lead to more higher-level matchings, **Cico** explores for each requirement the most likely part of the matching tree, as determined by several heuristics, trying to maximize the number of firing and the likelihood of each matching, and to minimize the amount of unparsed text left at the end of the process.

3.1.4. Actions and substitutions. After the matching phase, **Cico** will have determined a set of rules candidate for firing, and will select one of them for actual firing. Usually, this selection merely involves choosing the highest-ranking rule, but other strategies are possible as well (e.g., keeping into account the semantics of the rules, or their compositional properties).

Once the rule to fire has been selected, **Cico** will start its action. The action can be arbitrarily complex, given that it can also be a Shell script or even a generic executable, but usually a simple `echo` command to output some encoded form of the information captured by the rule is adequate. The output from the actions accumulate and becomes **Cico**'s own output.

The evaluation of the substitution proceeds in the same vein. The fragment to replace is defined as the substring delimited by the first and last matching term[4]. The requirement, after the substitution, is scanned again to check for the applicability of other rules, thus allowing the extraction of more information.

3.2. Modelling domains

One of the main concepts underlying the design of our system is that of *domain*. In this context, a domain is the field of applicability of a *model*[5] which is used to represent and organize the information extracted from the requirements. Typically, each domain corresponds to a set of rules that map the (natural language) linguistic expression of properties of entities belonging to the domain to some appropriate encoding, designed to simplify subsequent automatic processing. The separation of domains and MAS-rules from each other let us tackle each problem separately; moreover, each domain (and set of rules) can evolve separately, thus making for easy enrichment or editing of any single domain.

Our predefined rule sets offer several basic domains. While surely far from completeness, these domains already allow some useful modelling.

3.2.1. Data flow. This rule set covers a data flow model, allowing linguistic forms for sending and receiving informations, client/server interactions, queries and interrupt requests, for a total of 10 rules (we met one of them as an example in Section 3.1.3). The tags used by these rules concern the ability of performing inputs and outputs, to request interrupts and receive such requests, to participate in client/server interactions and the identification of a term as "information". Implicitly, this model defines the physical and logical data path between components of the system, and induces the creation of a *data dictionary* built from all the terms defined as "information".

3.2.2. Computations and policies. This domain, strictly coupled with the preceding one, covers several type of computations on the information (transformation, composition, selection, etc.) and some type of information-oriented policy (maximization, minimization, etc.). The 18 rules of this domain use only a new tag, to declare terms as "capable of performing computations".

3.2.3. Entity-relationship. Three rules, highly parametric, recognize many linguistic forms for entity-relationship models. Relationships can be defined by the user via the glossary (with a specific tag), both in active and passive form, where applicable. The rules recognize 1–1, 1–n and m–n relationships, where the multiplicity is expressed as an element of the numeric domain in the next paragraph. Some common relationships are already defined in the system glossary: among others, composition, provision, connection, and control.

3.2.4. Numeric amounts. This domain, whose model is simply the set of naturals enriched with an "indeterminate" element (corresponding to expressions like "some", "a number of", etc.), allows for natural numbers written out both in digits and as words, and for other terms like "a dozen", "a couple", and the like. The user can further enrich the set of terms by simply defining new ones in the glossary, with the corresponding `/NUM` tag.

3.2.5. Causal relationships. This domain describes causal relationships between events and actions of interest to the system described by the requirements. The underlying model is just a set of pairs (cause, effect), describing a part of the control flow of the system. The user can define events or actions by introducing them in the glossary; computations and input/output activities are automatically recognized as events and/or actions[6], as applicable.

3.3. The views

Circe offers several views on the requirements and on the system described therein. We briefly present them here, and discuss in Section 5 their typical use in support of various requirement engineering chores.

3.3.1. Model renditions. Models are usually rendered through graphical, interactive diagrams (see Figures 4 and 5)[7]. The *Data Flow* diagram shows, both at system- and at subsystem-level, the flows of data described in the requirements, including inputs, outputs, computations, client/server interactions and so on. Only data flowing in, out or through the system are shown, and indeed requirements specifying data flows related to non-system components are probably

[4]This definition can cause the loss of relevant material, extraneous to the rule but included in the fragment; on the other hand, removing only matching words could leave some "garbage" that could obscure the meaning of other parts of the requirement.

[5]We are not speaking here of MAS-rules models, but of models in their traditional meaning: abstract representations of a concrete system.

[6]This is obtained by giving `/EVT` and `/ACT` tags to the relevant substitutions.

[7]The current implementation is written in Java to maintain platform-independence.

redundant. The *Communication* diagram gives a very abstract view of the system, only showing the communication paths between the system (modules) and external entities. The *Entity-Relationship* diagram simply shows the E-R model built from the requirements, and the *Dependence* diagram graphically shows temporal and causal relationships between internal and external events and actions happening in the system.

A fifth view, called the *Object-Oriented Paraphrase*, represents textually an object model of the system; for each identified object, this view collects its relationships with other objects, all the data it handles and all the actions it can perform – the final result being very close to an OMT [28] object model. In our experience, an inspection performed on this representation is useful to reveal defects that could be easily skipped in inspecting the original format of the requirements, due to a different linguistic form or to the scattering of relevant facts among several, textually apart requirements.

3.3.2. Reporting views. These modules perform several checks on the models discussed above and on the requirements and the glossary themselves. The output from these modules tends to be as natural language-like as possible, trying to avoid any form of encoding that would defeat the main aim of the whole environment.

One of the most useful views is the *DFD consistency* check, that can report on dead (unused) data, data coming from nowhere or plainly badly-defined (e.g., declared built-in and then inconsistently input from the outside). This check is important to force the author to express a consistent data flow, albeit abstract. In this way, the specification phase will have to refine inputs, outputs, and computations, but will not need to "invent" anything from scratch. The *Tag usage* check verifies the consistency between the tags assigned to terms in the glossary and their actual use, reporting unused tags, incompatible declarations (e.g., a data object cannot perform I/O), etc. This usage check is important to fight the tendency, which we found in practice, to unneedingly enrich the terms with tags to force **Cico** to (wrongly!) understand a requirement. The double, opposing push to add tags (so that **Cico** can understand a requirement) and to remove them (to shut up the warnings) helps in exactly defining only the needed properties of all entities represented by the various terms, thus obtaining a greater adherence of the requirements to the problem.

The *Ambiguity* check verifies that abstract requirements are completely specified (due to the fuzzy nature of the parsing process, some of them could miss important elements), and the *Redundancy* check assures that no fact is unneedingly expressed twice in the requirements (see Section 4).

3.3.3. Metric and other views. Currently, only a functional metric derived from the Feature Point one [21] is implemented. This view, called *MFeP* for Modified Feature Point, shows the details of the calculation and the cost-of-presence of each requirement (see Section 4). Two more views present the *Understanding report*, by which **Cico** shows the requirements it has not "understood" in whole or in part, and the *Residual abstractions*, i.e., the abstractions found by the Goldin–Berry algorithm [18] in the textual material not understood by **Cico**.

These three views are tied in their evolution: initially, when **Cico** is presented with unrefined requirements and almost empty glossary, the MFeP metric is very low, very little text is understood and many abstractions are reported. As soon as requirements are brought in a form understandable by **Cico**, the MFeP metric grows progressively, most text is understood and very few residual abstractions are left unparsed.

3.4. Customizability

During the design of the whole environment great care has been taken to assure the *flexibility* of the resulting tools. This has been obtained, whenever possible, by lifting to user-level as much system logic as possible, thus making the individual tools almost programmable.

3.4.1. Languages and language constructs. **Cico** is essentially independent of the language used in requirements and rules; all the relevant information is contained in system-level glossaries and rule sets.

All recognized linguistic constructs are defined via MAS rules, that the user can modify or extend at will (they are encoded in textual configuration files), while the tags are implicitly defined by their use in the rules. Extension or adaption to a new language typically boils down to simply defining a new set of MAS-rules models capturing the linguistic constructs used to express relevant properties in the new language, and to translating any relevant predefined glossary.

3.4.2. Actions and substitutions. Thanks to the full generality of the possible actions, **Cico** can be usefully applied even in contexts quite different from the one it was designed to work in. It is really a generic, albeit elementary, system for the execution of commands in correspondence of natural language constructs, or the translation from natural language to a different (usually more abstract) one. Among possible uses, we could note NL database queries, filtering of textual news-like flows, or automatic markup of NL documents. Naturally, in some of the above cases our simplifications in Section 2.3 do not hold, and **Cico** would reveal the weakness of its matching algorithm; however, this wide range of applicability gives evidence of the flexibility of the tool.

3.4.3. Predefined glossaries. **Cico**'s predefined glossaries *do not* include ontological elements, as discussed in Section 2.3; however, they include a number of *relations* taken from common sense, generally expressed as verbs ("to send","to receive", "to compute"...) whose semantics and correspondence with the domain's underlying model is assumed to be intuitive. As already done for the rules, all glossaries are stored as text files, and can be modified and extended at user-level as needed.

3.4.4. View modules. A view module, be it for model rendition, checking or other, is essentially a filter that is fed a set of abstract requirements (but can also access the original requirements and glossary or the output of other modules, if desired) and must output HTML code representing the view[8]. Inside this very loose specification, a view module can do

[8] Remember that we are considering here only the Web-based interface.

almost everything, can be written in any language and can produce outputs of arbitrary complexity.

3.4.5. Extending Circe. Given the various level of customization we outlined above, extending **Circe** to handle new domains is rather easy. One needs to add only a new set of MAS-rules describing natural language constructs related to entities or properties of the model, possibly with some predefined term in a system glossary, and then one or more view modules to represent, check, or measure the model so built. Using script languages for the modules, a new domain typically requires between 5 and 15 Kb of MAS-rules and code, an effort that is surely worth the advantages.

4. An example[9]

We use as an example a very minimal description of **Circe** itself. We could have started from the description in Section 3, using the abstraction and understanding views to obtain by stepwise refinement the actual requirements, but for the sake of simplicity we start instead from the NLR list in Figure 2. This set of requirements is consistent and unambigous, as far as **Circe** is concerned, and none of the checking views report any error. The OMT-style object diagram from the object-oriented paraphrase is shown (graphically) in Figure 3, while the screenshot in Figure 4 shows the dataflow diagram built by the DFD view and Figure 5 depicts the communication structure of the system. Compare these with the architecture shown in Figure 1. Now, let us pretend we forgot Requirement 2. We would have received the following warnings:

```
** How is 'requirements' obtained?
** How is 'glossary' obtained?
* 'Web interface' is defined as /IN but not
used as such.
```

meaning that **Circe** (or better, the DFD checking module) cannot understand how "requirements" and "glossary" are introduced in the system, and moreover (from the tag usage checking module) that "Web interface" is declared in the glossary as capable of performing input, but actually does not input anything. A simple glance at the module-level data flow diagram would be enough to spot the problem. On the other hand, should we add a requirement

```
13. The repository receives from the Web
    interface requirements and glossary.
```

we would obtain the following warning:

```
* Requirement 13 is redundant with respect to
requirement 5.
```

Assuming we are satisfied with our requirements, we could be interested in knowing the MFeP count for **Circe** , as described at this level of abstraction. The MFeP view in Figure 6 informs us that the total count is 136 MFeP (not too much), and that the requirement most costly to implement, accounting for almost a third of the entire cost, will probably be requirement number 6 — not too surprising!

[9]All the text shown boxed in this section were originally in Italian, and are translated here for reader's convenience.

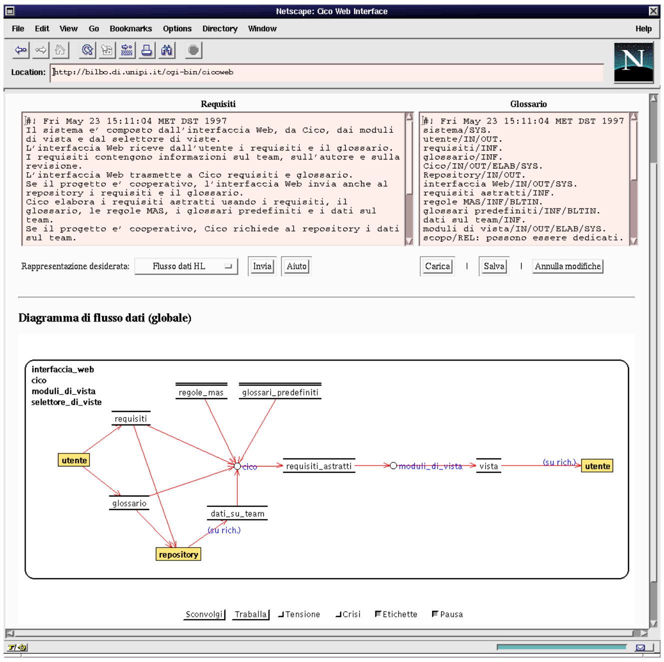

Figure 4. The Web interface showing a DFD of the example (in Italian).

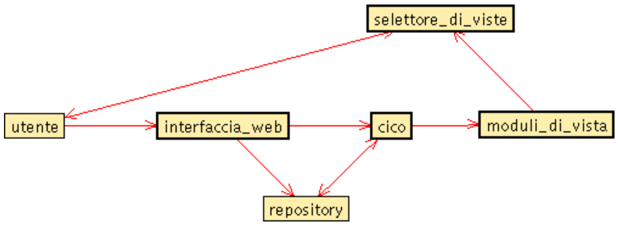

Figure 5. The communication diagram built from the sample requirements (in Italian).

```
MFeP computation:                   Cost breakdown:
 # of algorithms:  4 x 3 =  12+      req.  2:   8 MFePs
 # of inputs:      6 x 4 =  24+      req.  4:  18 MFePs
 # of outputs:     6 x 5 =  30+      req.  5:  17 MFePs
 # of data files:  8 x 7 =  56+      req.  6:  44 MFePs
 # of interfaces:  2 x 7 =  14+      req.  8:   9 MFePs
-----------------------------        req. 10:   3 MFePs
                  TOTAL = 136        req. 11:   9 MFePs
```

Figure 6. The MFeP counting report.

5. Some typical application

In this section we discuss how some typical requirement engineering problems can be solved or made somewhat easier by NLR processing. We stress again the fact that these techniques are methodology-neutral, and that the results are intended as an aid to the engineer, rather than a complete solution.

```
 1. The system is made of the Web interface, of Cico, of the
    view modules and of the view selector.
 2. The Web interface receives from the user requirements and
    glossary.
 3. Requirements contain data on the team, on the author and on
    the revision.
 4. The Web interface transmits to Cico requirements and
    glossary.
 5. If the project is cooperative, the Web interface sends
    requirements and glossary to the repository, too.
 6. Cico computes abstract requirements using requirements,
    glossary, MAS-rules, predefined glossary and team data.
 7. If the project is cooperative, Cico requests team data to
    the repository.
 8. The view modules receive abstract requirements from Cico.
 9. The view modules can be dedicated to modelling, validation
    or metrication.
10. From abstract requirements, view modules compute a view.
11. The view modules send the view to the view selector.
12. The user requests a view to the view selector.
```

```
system/SYS.
user/IN/OUT.
requirements/INF.
glossary/INF.
Cico/IN/OUT/ELAB/SYS.
Repository/IN/OUT.
Web interface/IN/OUT/SYS.
abstract requirements/INF.
MAS-rules/INF/BLTIN.
predefined glossaries/INF/BLTIN.
team data/INF.
view modules/IN/OUT/ELAB/SYS.
purpose/REL: can be dedicated.
modelling. validation. metrication.
view/INF: views.
view selector/IN/OUT/SYS.
project is cooperative/EVT.
team. author. revision.
containment/REL: contain.
```

Figure 2. Some simple requirement (left) and glossary (right) for Circe.

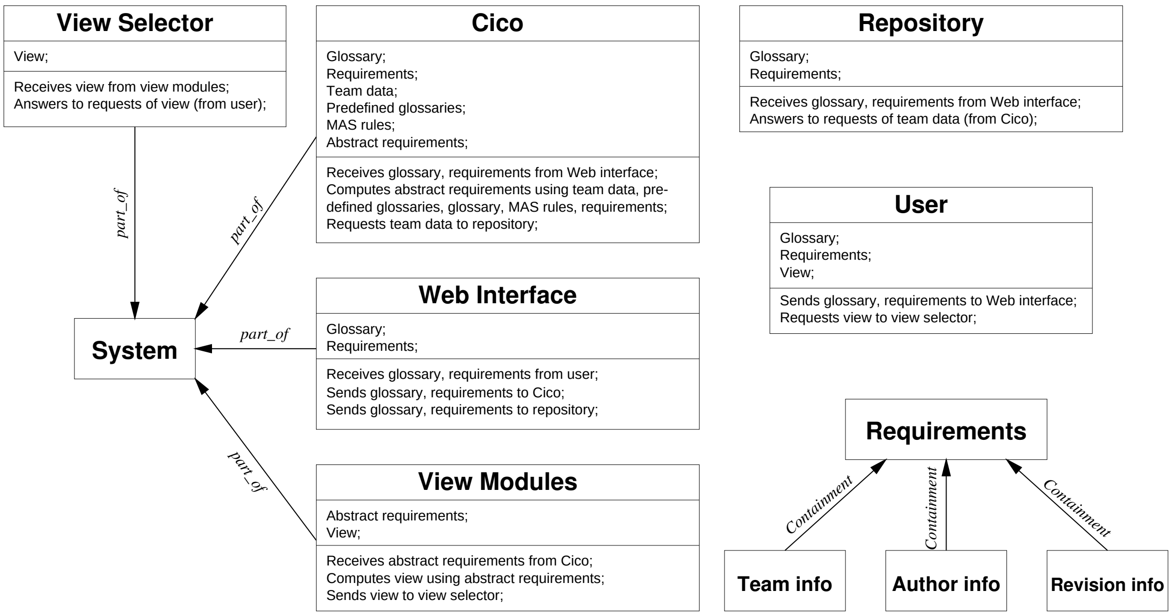

Figure 3. A partial OMT-like model built from the object-oriented paraphrase.

5.1. Requirement elicitation

When starting the elicitation from a NL user's documentation (e.g., available manuals or interview transcripts), the usage of the residual abstractions view can help in quickly identifying the main concepts in the problem domain. After initially populating the glossary with relevant abstractions, **Cico** usually can already "understand" parts of the original documents, that can then become requirements (or domain description statements) as applicable.

5.2. Requirement selection

When it is needed to perform a cost-based selection to reach a balance between allocated budget and the features of the system to design, the cost-of-presence report in the MFeP view can aid in quickly estimating what requirement or group of requirements can be dropped to reduce the projected complexity of the system. Obviously, this kind of selection is based on the acceptability of a function-point based metric. Although not formally defined, function points seem to be often accepted by the industry as a reliable hint, if not as a real measure of the implementation cost, so the early availability of this information can help in taking decisions at this stage.

5.3. Conflict identification

Circe manages conflicts differently based on their source. If a conflict arises between two requirements from the same requirement list, it is reported as an *inconsistency* in the relevant checking view (i.e., DFD or tag usage). If the conflict arises between a requirement from a certain user and a requirement from a different user from the same *team*[10], **Circe** reports a *conflict*. In both cases, the semantic knowledge leading to the discovery of a conflict is contained in a checking module. Currently, **Circe** discovers only limited classes of conflicts in the DFD model, in the tags assigned to a term, and in the policies model.

Conflict *resolution* is left to interested people, in keeping with our choice of methodology-neutrality.

5.4. Enforcing good style in requirements

Circe can be profitably adopted as a means to induce the use of a defined *style* in the requirements. In fact, while the matching algorithm used by **Cico** leaves ample freedom on the linguistic form of the fact one wants to express in the requirements, the need to explicitly identify (via the glossary) and categorize (via the tags) the entities and their relationships that, together, form the fact, pushes the user to express as clearly as possible the requirements.

While a well-expressed requirement, concerning a fact covered by one of the available domains, has a well-defined semantics given by the action/substitution pair of the corresponding MAS-rule, a requirement that is incomplete or ambiguous, or contains terms not defined in the glossary, is easily spotted (being refused by **Cico**).

During the recognition phase, the tool will also report any requirement with *missing* parts, sometimes symptom of the presence of some understood term, but not infrequently spy of an embarassing confusion (often not recognized at a conscious level) in the mind of the user. Our experience is that this last case is more common than one could think; for this very effect, an inspection on a document performed by a third party is invariably more effective than an inspection performed by the author himself.

6. Related works

The need to bridge the linguistic gap between problem professionals and information professionals has already been identified in the literature [32]; this is an important factor in the information gathering problem that [16] judges "the most difficult task in requirement engineering". In the last few years, solutions have appeared of three kinds:

- full-fledged natural language processing and understanding systems, usually endowed with huge knowledge bases, countless syntactic rules, statistical records, and semantic nets [19, 26];
- formal language-based systems, made more palatable to the user by using a natural language like (but usually strict) concrete syntax [15];
- text-based approaches, that dispense with semantic hassles by limiting themselves to strictly textual and statistical algorithms [2, 18].

In our opinion, all these solutions are missing some key element: the first case lacks a useful underlying model for the knowledge extracted from the text[11] and does not allow the assignment of richer, context-specific meaning to common terms; the second case lacks desirable flexibility in the use of natural language; the third one relies entirely on the (sometime fortunate) equivalence between concrete syntax (the text) and semantics (its meaning).

Database literature offers a number of proposals similar to our one, e.g. [7, 10, 26], but almost invariably these proposals limit themselves to the extraction of a conceptual (E-R) schema from a descriptive text, with no validation or measurement.

The need for an informed validation of a formal model by a customer was felt since a long time. This need has led to interest in the field of natural language generation [1] and to *explaining* systems capable of producing a NL version of a formal model [12, 30, 31]. As already said, these techniques are not applicable in the early stages of requirement gathering and elicitation, that usually start from NL material. Their full strenght is best seen when they are *integrated* with the dual approach, allowing editing both on the NL text and on the formal models without losing consistency between the two representations.

On the other hand, a growing number of modelling techniques have been and are being developed: SADT [25], Yourdon [33], Shlaer-Mellor [29], OMT [28], UML [8] and many others. All these techniques, however, fall short on the field of customer interaction — effectively *encoding too early* the requirements.

Our approach, while surely inferior to most of the alternatives cited under at least some respect, is novel in that it integrates natural language processing, modeling and validation aspects, in our hope offering a useful tool to assist in requirement elicitation and validation.

7. Conclusions and future work

The creation of complete, precise and adherent requirements is a task that cannot leave out of consideration an involvment of the customer, both in her traditional role of supplier of information and in the less widely recognized one of final judge on the suitability of the requirements produced — the latter implying a greater responsibility on her part.

To allow the customer to succesfully play both roles, a customer-understandable (and thus, typically natural language) form of the *final* requirements must be provided.

In this paper we discussed the requirements for a system supporting creation and evolution of high-quality natural language requirements, and described several tools we built to this end and associated techniques. These tools, easily customizable, can build (semi)formal models in an almost automatic fashion extracting information from the natural language text of the requirements, and can measure and check

[10] Teams are defined by a declaration in the requirement list by which a user states which other users are working on different parts of the same project. Obviously, the users interested must have dual declarations in their requirements.

[11] Usually, these systems use semantic nets to store this knowledge: a representation barely useful to a software engineer.

the consistence of these models. Our experiences with these tools [5], currently covering five differents projects in both academic and industrial environments, make us confident on the value of the approach, that can also be used to obtain an initial OMT model whose equivalence with the natural language requirements is implicitly certified.

The remarkable flexibility of the tools we built allows for easy evolution and maintenance, and makes it rather easy to adapt them to different contexts, or adopting modelling techniques different from the ones we used in our example. More work is needed to enrich the set of domains to which **Circe** can be applied; we will add new domains (e.g., temporal) and views on them in the near future. We also plan to further investigate several issues related to distributed cooperative work on the requirements, e.g., how to split, join, and verify the consistency of distinct sets of requirements, maybe reflecting different viewpoints, about the same system [17].

We believe that in allowing requirements validation both by the customer and by tools through all their evolution, thus strictly tying requirements production and validation [16], we have made a step forward to higher-quality requirements and higher user's satisfaction.

References

[1] G. Adorni and M. Zock, editors. *Trends in Natural Language Generation: An Artificial Intelligence Perspective*. Number 1036 in LNCS. Springer, 1993.

[2] C. S. Aguilera and D. M. Berry. The use of a repeated phrase finder in requirements extraction. *Journal of Systems and Software*, 13(9), 1990.

[3] J. Allen. *Natural Language Understanding*. Addison-Wesley, 1987.

[4] V. Ambriola and V. Gervasi. Cico: A tool for natural language requirement processing. Technical report, Dipartimento di Informatica, Pisa, Italy, 1997. (in preparation).

[5] V. Ambriola and V. Gervasi. An environment for cooperative construction of natural-language requirement bases. In *Proceedings of the Eighth Conference on Software Engineering Environments*. IEEE Computer Society Press, 1997.

[6] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, Feb. 1995.

[7] W. J. Black. Acquisition of conceptual data models from natural language descriptions. In *Proc. of the 3rd Conference of the European Chapter of the ACM*, Danemark, 1987.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling language, version 1.0*. Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, 1997.

[9] F. Brum-Cottan and P. Wall. Using video to re-present the user. *Communications of the ACM*, 38(5), May 1995.

[10] C. P. C. Rolland. A natural language approach for requirements engineering. In P. Loucopoulos, editor, *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag, 1992.

[11] 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.

[12] H. Dalianis. A method for validating a conceptual model by natural language discourse generation. In *Advanced Information Systems Engineering*, number 593 in LNCS. Springer-Verlag, 1992.

[13] R. A. DeMillo et al. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company, Inc., 1987.

[14] K. El Emam, S. Quintin, and N. H. Madhavji. User participation in the requirements engineering process: an empirical study. *Requirements Engineering Journal*, 1(1), 1996.

[15] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3), 1994.

[16] A. Finkelstein. Requirements engineering: a review and research agenda. In *Proceedings of the First Asian & Pacific Software Engineering Conference*, pages 10–19. IEEE CS Press, 1994.

[17] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8), Aug. 1994.

[18] L. Goldin and D. M. Berry. `AbstFinder`, a prototype natural language text abstraction finder for use in requirements elicitation. In *Automated Software Engineering*, 1997. (To appear).

[19] A. Hars. Advancing CASE productivity by using natural language processing and computerized ontologies: The ACAPULCO system. In *Proceedings of the Eleventh Automated Software Engineering Conference*, 1996.

[20] M. Jackson. *Software Requirements and Specification*. Addison-Wesley, 1995.

[21] C. Jones. *A Short History of Function Points and Feature Points*. Software Productivity Research, Inc., Burlington, Mass., June 1986.

[22] D. B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11), Nov. 1995.

[23] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc: Toward programs with common sense. *Communications of the ACM*, 33(8), Aug. 1990.

[24] Y. Maarek and D. M. Berry. The use of lexical affinities in requirements extraction. Technical report, Faculty of Computer Science, Technion, Haifa, Israel, 1988.

[25] D. A. Marca and C. L. McGowan. *SADT : Structured Analysis and Design Techniques*. McGraw-Hill, New York, 1988.

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

[27] B. Nuseibeh. Conflicting requirements: When the customer is not always right. *Requirements Engineering Journal*, 1(1), 1996.

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

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

[30] W. R. Swartout. GIST english generator. In D. Waltz, editor, *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982. AAAI Press.

[31] W. R. Swartout. The GIST behavior explainer. In M. A. Genesereth, editor, *Proceedings of the National Conference on Artificial Intelligence*, Washington, DC, 1983. AAAI Press.

[32] J. L. Whitten, L. D. Bentley, and V. M. Barlow. *System analysis and design*. Burr Ridge, 1994.

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