Experiences with Domain-Based Parsing of Natural Language Requirements

Vincenzo Ambriola — ambriola@di.unipi.it

Vincenzo Gervasi — gervasi@di.unipi.it

Dipartimento di Informatica — Pisa, Italy

Introduction

Requirement engineering is the part of software engineering that concerns the identification, expression, validation and analysis of the real-world goals and constraints for a software system. In particular, requirement engineering offers tools, methods and techniques to bridge the gap between (often vague) customer's desires and a precise specification of the system that can be used to design and build a concrete system satisfying the requirements stated.

Despite the wide array of formalisms devised to express requirements for software systems, natural language requirements are still the most common form of requirements found in practice. Natural language requirements have the undesirable property that it is often very difficult to prove properties on them (e.g.: non-redundancy, consistency, coverage of all possible cases) due in part to their inherent ambiguity, and in part to their low amenability to automatic processing. On the other hand, natural language plays a key role in communication with the customer, and is also the best expressive medium in the initial "brainstorming" part of a requirement process, so it cannot be simply discarded as inconvenient.

For these reasons, an environment supporting requirement engineering activities has to devote a special attention to the handling of natural language requirements. Most commercially available tools offer a good support to techniques like tracing the evolution of requirements, linking requirements to other documents, structuring, merging and splitting requirement documents, assigning authorities over requirements to different individuals, etc. All these techniques, however, treat natural language requirements as black-box objects, without getting to their meaning.

In this paper, we discuss a technique for shallow parsing of natural language that we have found to be particularly effective in the case of software requirements. The technique described here is implemented in a component named Cico[1] of our cooperative requirement engineering environment Circe [1], and has been used to perform partial validation of some of the requirements for the International Space Station [2].

[1] The name is a shameless play on the acronym GIGO, itself standing for "Garbage In, Garbage Out".

Peculiarities of requirement documents

We concentrate on the parsing of natural language requirements in the context of a requirement engineering supporting environment. The case we are interested in lends itself well to several simplifications, summarized in the table below, with respect to the general case.

NLP for Requirements	*NLP for arbitrary text*
Explicitness is desired	Elliptical, depends on context
Mostly **interactive** redaction	Often batch processing
Restricted domain	Arbitrary domain
Reduced vocabulary	Vast vocabulary
Well-defined terms	Mostly approximate definitions

In particular, *explicitness* ensures us that all the needed part of discourse will be present in the text; *interactivity* allows us to consider parsing failures as non-critical failures; *restrictions* on the domain and on the vocabulary make huge knowledge bases unnecessary, and the strict *definition* of terms allows us to assume a 1:1 correspondence between terms and concepts. The last two peculiarities call for the introduction of an explicit *glossary* document, defining the vocabulary, to go alongside the requirement document that uses the terms defined in it.

Cico: a parser for natural language requirements

In our system, a glossary consists of a list of *terms*; each term has an associated set of *tags*, describing the domain-specific properties of the term, and a set of *synonyms*, that are considered equivalent to the base term.[2] Beside the glossary explicitly specified by the writer of the requirements (the *user glossary)*, a number of *system glossaries* exist, defining common terms typical of a certain domain. These glossaries are used, together with a lexical normalizer, to put requirements in *canonical form*: in a canonic requirement, all synonyms are substituted by the corresponding base term, and all tags of each term are appended to each occurrence of the term in the text. As an example, a statement like "*The terminal sends the password to the server*" could be canonized as "*terminal/IN/OUT sends password/INF to server/IN/OUT/ELAB*", where the /IN, /OUT, /ELAB, and /INF tags are domain-specific (in this case, the domain is that of data flows).

After canonization, Cico applies a domain-specific set of *MAS-rules*. Each rule consists of a *model*, which is a tag-based template of a statement, of an *action*, which is executed when a match occurs, and of a *substitution*, that replaces the matched fragment in the text. A matching can be imperfect in several ways (ordering can be different, some term may be missing, extra terms may be present, etc.); the degree of correspondence, determined according to a fuzzy metric, is recorded and used in the end to select the overall

[2] Currently, our parser lacks a morphological component, so synonyms are also used to encode morphological variations.

best matching tree. Cico can thus be seen as an engine for the application of a fuzzy rewriting system using backtracking and minimax strategies to determine an optimal parse tree for a statement.
It is important to notice that this approach *does not* build syntax trees. On the contrary, the parse tree records information on the semantics of the requirement, as implicitly defined by the domain-specific rule sets and glossaries.

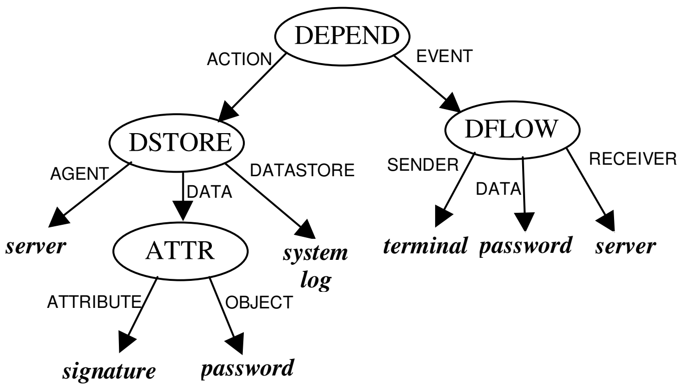
As an example, consider the following requirement:

> *When the server receives from the terminal the password, the server stores the signature of the password in the system log*

Let us assume that the user glossary contains *server/IN/OUT/ELAB*, *terminal/IN/OUT*, *password/INF*, *signature/ATTR*, *system log/STORE*, and that the following MAS-rules are defined, among others, in the system:[3]

Model	***Action***	***Substitution***
WHEN event/EVT action/ACT	Out DEPEND $action $event	
Receiver/IN RECEIVES data/INF FROM sender/OUT	Out DFLOW $sender $data $receiver	$ID/EVT
Agent/ELAB STORES data/INF IN datastore/STORE	Out DSTORE $agent $data $datastore	$ID/ACT
Attribute/ATTR OF object	Out ATTR $attribute $object	$ID/INF

Then, the parsing tree corresponding to our sample requirement would be the following:



From a forest of similar parsing trees, other tools in the Circe environment extract a number of different representations, including data flow diagrams, entity-relationships diagrams and event charts. Also, consistency checking and metrication is performed on these alternative representations. All these chores are made considerably easier by the fact that domain-specific knowledge is exploited already in the parsing process.

---

[3] In our syntax, lower case parts are variables, while upper case parts are constants. $ is the usual variable substitution symbol, with $ID being a special variable holding a unique identifier that is used to reference a node in the parsing tree. “Out” is a system action to store a tuple in Circe’s repository.

Evaluation and limitations

Despite the apparent combinatorial factor implicit in the backtracking strategy, Cico has proven quite satisfactory performance-wise. It can parse a document containing 150 requirements, applying a set of 169 MAS-rules and producing over 700 tuples to inject in the system repository, in slightly more than 3 seconds on a standard-class PC. This is largely due to the fact that domain-based MAS rules are much more specific, and thus selective, than usual syntax-based rules. The fuzzy matching technique also makes Cico quite robust with respect to errors or perturbations in the text of the requirements. Cico's coverage, on the other hand, is very restrictive: domain-specific rules tend to combine much less freely than syntactic rules. This, however, is not entirely undesirable in the case of requirements, as discussed above, and also encourages the adoption of a defined *style* in requirement documents.

A similar point of view can be taken on the need for an explicit user glossary: the effort spent in identifying relevant terms, and in giving them appropriate domain-specific tags, is an important part of every requirement elicitation process. Performing it with the help of a supporting system[4] can actually simplify life.

All the factors that make domain-specific parsing palatable for use in requirement engineering also have a positive effect on the size of the parsing engine. Cico only extends for less than 3,000 lines of codes, and its knowledge base (including MAS rules and system glossaries for a handful of different domains) takes currently less than 20 Kbytes of textually encoded information. Moreover, MAS rules are not difficult to define and evolve. In particular, adaptation to a different language may be obtained to a great extent simply by translating the rules model parts.

Within the limitations of a restricted and well-defined application field, in our case that of requirement engineering, we believe that domain-specific parsing is an adequate and efficient technique for extracting semantic information from natural language sentences. Experiences have also shown that documents accepted by a domain-based parser tend to be more precise and far less ambiguous than uncontrolled natural language texts.

References

[1] V. Ambriola, V. Gervasi. *Processing Natural Language Requirements*. Proc. of the 13th IEEE Conference on Automated Software Engineering. IEEE Press, 1997.

[2] V. Gervasi, B. Nuseibeh. *On the Lightweight Validation of Natural Language Requirements*. Submitted to the First International Workshop on the Requirement Engineering Process, 1999.

[4] Cico supports so-called *speculative tagging*: if a term is given a special "unknown" tag, the parser can suggest a set of tags that would be appropriate for it in order to maximize the matching score in the contexts where it appears.