Università di Pisa
Dipartimento di Informatica

Technical Report: TR-01-25

The Cico Domain-Based Parser

Vincenzo Gervasi

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

The Cico Domain-Based Parser

Vincenzo Gervasi

Abstract

Domain-based parsing is a shallow parsing technique that exploits knowledge about domain-specific properties of terms in order to determine "optimal" parse trees for natural language sentences.

Cico is a simple parser using domain-based parsing. It is particularly well suited for parsing natural language sentences of technical nature (e.g., requirements documents for software systems), as in this case several simplifying assumptions hold, and has been used successfully in several experiments in the requirement engineering field. In the first part of this report, we describe formally the parsing algorithm implemented by Cico, and give some experimental data on its performance. In the second part, we provide a complete user manual for `cico3`, an implementation of the Cico algorithm, and for a number of associated tools. Finally, in the third part, we present some illustrative example taken from real applications.

1 Introduction

Cico is a shallow parser for natural language sentences implementing domain-based parsing [2]. It can also be described as an engine for the application of a fuzzy rewriting system to a text, using backtracking and heuristic optimization strategies to determine an optimal parse tree for a statement. We have used the parser in our requirements engineering environment Circe [7]. In Circe, the role of Cico is that of an input method: it parses natural language requirements documents input by the users, and populates the system's knowledge repository with the resulting parse trees (encoded as a set of tuples).

The choice of this particular application is not without reasons. In fact, collection, definition and analysis of the *requirements* for a software system are among the most difficult and error-prone activities in the software development process. On one hand, one would want requirements to be as formal and precise as possible, in order to facilitate the designers and implementors of the system in their work. This desire for preciseness has sometimes led people to resort to completely formal languages for expressing their requirements. On the other hand, the desire for ease of change, experimentation, and comunicability, calls for informal requirements — often expressed in (precise) natural language. As a matter of fact, despite the remarkable advancements in the design of user-acceptable formal languages, the vast majority of requirements documents for software projects are still written in plain English (or in other natural languages). Providing the ability to extract information from this kind of documents in order to automatically analyze and process them is thus an important goal in the field.

Many commercially available tools offer a good support to techniques like tracing the evolution of requirements, linking requirements to other documents, structuring, merging, and splitting requirements document, prioritizing requirements, assigning authorities over them to different individuals, etc. All these techniques, however, treat natural language requirements as black-box objects, without getting to their meaning.

It is thus not surprising that the extraction the semantic content from natural language requirements, in order to use the knowledge extracted to build, validate and measure models of the system described by the requirements, has attracted much attention. To this end, parsing techniques for natural language requirements are needed. Cico was designed to serve exactly this role.

While a long and honorable tradition of studies in the field of natural language processing exists (see e.g. [4] for a survey of the field), in developing Circe we had a particular set of conditions to comply with, and several simplification to take advantage of (both deriving from the peculiar characteristics of requirements and of requirements processes) that suggested the use of non-standard techniques. Table 1 summarizes some of the peculiarities that make parsing of natural language requirements simpler than parsing of general text. These simplifications come mainly from three factors:

- **Explicitness**. It is useful to reduce to a minimum the amount of unexpressed information in the requirements. What one could consider common sense could be a total surprise to 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 [13, 14] or ACAPULCO [11].

- **Interactivity**. Requirement writing is an interactive process. The user can repeatedly modify a requirement until it can be parsed 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 parsing a requirement is not fatal, differently from what happens in the case of

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

Table 1: Peculiarities of NLP for requirements.

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...). 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. As a consequence, domain knowledge can be used profitably in the parsing process, thus obtaining better parsing results with increased performance.

The characteristics of requirements processes, however, establish some constraint on the parsing process. For example, we do not want to embed a particular syntactic form nor a specific jargon into the parser, and this suggests that the dependency of the parsing process from the grammar rules of a given natural language should not be too tight. Moreover, the same interactivity aspect that makes parsing failures non-fatal, also imposes stringent constraints on the amount of time that can be spent in parsing a requirement.

Cico tries to satisfy these constraints, and to take advantage of the peculiarities in Table 1, by adopting a shallow parsing approach based on fuzzy matching of domain-specific rules to natural language statements. In particular:

- Shallow parsing is used since explicitness promotes the adherence of the surface structure of a statement to its semantic content. Shallow parsing also fosters efficiency.

- Fuzzy matching is used to improve the robustness of the parsing technique, that can succeed even in the presence of wrongly ordered or missing items in a statement. This allows the user of the system to benefit from the support offered by the tool, even when starting from incomplete or badly structured textual material. Imperfect matching is acceptable thanks to the interactive nature of the environment, since the results of the parsing process are presented back to the user for confirmation.

- Domain-specific rules are used to encourage the use of standard language and style, and to simplify the association of a meaning to a certain syntactic form. Domain-specific parsing improves efficiency, since domain-specific rules tend to combine much less freely than grammatical rules, and make translation in different languages easier, since a map can be established between domain-specific rules on the same domain, even if in different languages.

While our parsing technique supports both grammar-based tagging and domain-based tagging of language terms, alone or in combination, the rule sets currently uses in CIRCE only rely on domain-based tagging, thus simplifying the adaption of the tool to different natural languages. Grammar-based parsing was also used in different applications, again with positive results.

In this report, we describe CICO's parsing algorithms, and evaluate its performances. Section 2 describes the canonization process that is applied to input sentences before the parsing stage proper, which is described in detail in Section 3. A few more advanced features are discussed in Section 4, including primitives constructs supporting pronominal references resolution, and automatic tagging of terms based on the whole document context. A section on evaluation and some conclusions follow.

Sections 7 and 8 contain the user manual for `cico3`, a C language implementation of our parsing algorithm, and for a few associated utilities. Finally, Sections 9 and 10 present a few examples taken from real applications of CICO in different contexts.

Part I
Parsing Technique

2 Input format

Cico expects from its input stream a sequence of *free form statements*, obeying only a few lexical rules. In particular, it is assumed that the hash mark `#` introduces comments (extending to the end of the line), that a dot character `.` represents a full stop, and acts as a statement terminator, and that the single dash character `-`, when it appears in the first column, acts as an itemization marker for lists. Apart from that, valid text is essentially free-format, e.g.,

```
# Alarm handling.
# see DOD-STD such-and-such; last updated 22/3/99 BZ

The system can be in manual mode or in automatic mode. When the
alarm is active, if
- the system is in manual mode, or
- current time is greater than the last timestamp in the log,
the log daemon appends an entry for the alarm in the system log.
```

Before free text can be analyzed by the parser proper, however, a certain amount of normalization has to be performed in the text. This includes typographical adjustments, format handling, and glossary substitution, in order to transform free-form text into a sequence of *canonic statements*.

2.1 Typographical adjustments

Many human languages tend to have different typographical representations for certain alphabetic characters or punctuation signs. This is particularly relevant for electronic texts, that are often constrained by the historical limitations of the ASCII or ISO character sets.

For example, the German ß character can also be written `ss`; accented vowels in Italian can be written with an appended apostrophe, as in `e'` instead of è, quotation in many languages can be denoted by double quotes (`"..."`) but also by single or double backtick and apostrophe (`` `...' ``), or by guillemots (≪ ... ≫) — that can be themselves represented by << (double less-than) and >> (double greater-than) — and so on.

To handle these and other similar language-specific issues, Cico performs a typographical canonization step on the source text. For the Italian language, this step includes:

- substituting proper ISO-8859-1 accented vowels for *vowel*+`'` sequences, choosing acute or grave accents where appropriate (e.g., `e'` → `è`, but `poiche'` → `poiché`), and taking care of exceptions consisting in truncated words (e.g., `po'` is a truncated form of `poco` and thus must be left intact);
- converting decimal points (e.g., `2.1`) into decimal commas (`2,1`);
- substituting various quotation styles with the uniform use of double quotes.

2.2 Format handling

The second canonization step is concerned with structural features of the text, and is totally language-independent. This step includes:

- reducing runs of multiple white space characters to a single space;
- discarding comment lines;
- discarding blank lines;
- joining lines and splitting them on full stops, so that each statement consists of a single line of text;
- substituting itemization markers with special markup to identify item and list contexts (see Section 3.8);
- making a single term out of quoted excerpts;
- discarding remaining unneeded punctuation characters.

At the end of this stage, the input stream consists of a sequence of statements, one per line; each statement consists of a sequence of words and special markers, separated by spaces. Formally, a document $\mathcal{R}$ is defined as set of sequences of words:

$$\mathcal{R} = \{\langle \omega_1, \omega_2, \ldots, \omega_n \rangle\}$$

2.3 Glossary substitution

A glossary serves two purposes: it declares properties of the terms it contains, by annotating each term with a set of *tags*, and establish a map to the canonical form of a term from a set of synonyms. Formally, a glossary $\mathcal{G}$ is defined as a set of term-synonyms pairs:

$$\mathcal{G} = \{(\tau, \{\tau_i\})\}$$

where τ is the base term (i.e., the canonical form for the term) and τ_i are synonyms for τ. The syntax for a glossary is given by the following context-free grammar

$$
\begin{array}{rcl}
\text{GLOSS} & ::= & \epsilon \quad | \quad \text{ENTRY GLOSS} \\
\text{ENTRY} & ::= & \text{term TAGS SYNS .} \\
\text{TAGS} & ::= & \epsilon \quad | \quad / \text{ tag TAGS} \\
\text{SYNS} & ::= & \epsilon \quad | \quad : \text{ SYNL} \\
\text{SYNL} & ::= & \text{term} \quad | \quad \text{term , SYNL}
\end{array}
$$

Typical glossary entries for the example given above are

```
last timestamp/ATTR: time of the last update.
system log/INF: log, system journal, logbook.
system/HASMODE/SYS.
manual mode/MODE.
automatic mode/MODE: unsupervised.
log daemon/ELAB.
```

Given a term τ, we define the two functions $\mathsf{term}(\tau)$ and $\mathsf{tags}(\tau)$ to be, respectively, the base form of the term (denoted by "term" in the syntactic rule for ENTRY given above) and the set of its tags (denoted by TAGS in the same syntactic rule).

Glossary substitution is obtained by applying the $\mathsf{map}_{\mathcal{G}}()$ function described below to each ω_i in each statement in $\mathcal{R}$:

$$
\mathsf{map}_{\mathcal{G}}(\omega) = \begin{cases} \tau & \text{if } \exists(\tau,\sigma) \in \mathcal{G} \text{ s.t. } \omega \in \sigma \ \vee \ \omega = \mathsf{term}(\tau) \\ \omega & \text{otherwise} \end{cases}
$$

To guarantee uniqueness in the definition above, we assume that glossaries are always *well-formed*, i.e., there is at most one base term for each possible word. Formally, $\mathcal{G}$ is well-formed if the two conditions below hold:

$$
\forall(\tau,\sigma) \in \mathcal{G}, \forall \tau_i \in \sigma, \nexists(\tau',\sigma') \in \mathcal{G} \text{ s.t. } (\tau_i \in \sigma') \wedge (\tau' \neq \tau)
$$

$$
\forall(\tau,\sigma) \in \mathcal{G}, \forall(\tau',\sigma') \in \mathcal{G}, \mathsf{term}(\tau) = \mathsf{term}(\tau') \Rightarrow \tau = \tau'
$$

Given a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ we denote its glossary-substituted version with $s_{|\mathcal{G}}$:

$$
s_{|\mathcal{G}} = \langle \mathsf{map}_{\mathcal{G}}(\omega_1), \ldots, \mathsf{map}_{\mathcal{G}}(\omega_n) \rangle
$$

3 Parsing algorithm

Cico uses a parsing algorithm based on the fuzzy matching of sentence fragments to templates contained in a set of rules. Since generally several matchings are possible at each stage, Cico performs an heuristically-limited search in the space of all valid parsing trees to determine the overall best parsing tree.

3.1 MAS rules

Cico's parsing algorithm is driven by a set of *MAS rules*. The MAS acronym stands for Model, Action, Substitution, and refers to the three parts in each rule. Formally, a MAS rule ρ is defined as

$$\rho = (\mu, \alpha, \sigma)$$

where

$$\begin{aligned} \mu &= \langle \tau_1, \ldots, \tau_n \rangle \text{ is the } \textit{model}, \\ \alpha &= \langle \beta_1, \ldots, \beta_p \rangle \text{ is the } \textit{action}, \text{ and} \\ \sigma &= \langle \gamma_1, \ldots, \gamma_q \rangle \text{ is the } \textit{substitution}. \end{aligned}$$

The syntactic denotation for a MAS rule is

$$\tau_1 \ \ldots \ \tau_n \ \texttt{::} \ \beta_1 \ \ldots \ \beta_p \ \texttt{>>} \ \gamma_1 \ \ldots \ \gamma_q$$

where the action or the substitution, but not both, can be empty (in the first case the :: separator can be omitted, as can the >> separator in the second case).

The intuitive meaning of a rule ρ applied to a statement r is "if the model μ of ρ matches a fragment of r, execute the action α and substitute the matching fragment with σ".

Terms in a rule are either constants or variables; the function $\mathsf{isvar}(\tau)$ is true if τ is a variable, and $\mathsf{isconst}(\tau)$ is true if τ is a constant (obviously, $\mathsf{isvar}(\tau) = \neg\mathsf{isconst}(\tau)$ always holds). We will use UPPER CASE to indicate constant terms and lower case to indicate variable names. When a rule is applied, the matching can bind a value to the variable terms in μ. This binding is then used when the rule is fired to assign values to the variables in α and σ.

3.2 Matchings

Given a term τ and a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$, we define a set of possible *targets* for τ in s as

$$\mathsf{target}(\tau, s) = \begin{cases} \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

A *matching map* between a rule $\rho = (\mu, \alpha, \sigma)$ (with $\mu = \langle \tau_1, \ldots, \tau_n \rangle$) and a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ is a sequence of targets for all the terms in the model of ρ, i.e.

$$\mathsf{match}(\rho, s) = \langle \mathsf{target}(\tau_1, s), \ldots, \mathsf{target}(\tau_n, s) \rangle$$

Given a matching map $M = \langle T_1, \dots, T_n \rangle$, we say that M is *complete* if $\forall i = 1..n, T_i \neq \emptyset$. A *valid match* m from a complete matching map M is defined as follows:

$$m = \langle t_1, \dots, t_n \rangle \text{ s.t. } \forall i = 1..n, (t_i \in T_i) \wedge (\forall j = 1..n, t_i = t_j \Rightarrow i = j)$$

The intuition behind the above definition is that a valid match associates each element in the model of a rule to a single, distinct term in a statement. In most cases, a single complete matching map will contain many different valid matches. To choose among them, CICO uses a scoring system.

3.3 Match scoring

Since models express "natural" templates for sentence fragments, the ideal match for a model would be a one-to-one, ordered correspondence between elements of the templates and terms in a statement. However, natural languages usually permit a certain degree of freedom in the placement of words. This is particular true of the Italian language, that — as its ancestor, Latin — permits largely unrestricted rearrangement in the order of words.

Moreover, words not accounted for in the template could be added to a statement without altering the original meaning, but rather specifying a concept in greater detail. This is typical, for example, of adjectives and adverbs, that could well be discarded when analyzing a text to obtain at least a rough understanding of its meaning.

CICO associates a score to each valid match taking into account the two factors mentioned above. The latter is characterized as follows: given a match $m = \langle t_1, \dots, t_n \rangle$, its *span* is given by the distance between the rightmost and leftmost target (inclusive), that is

$$\mathsf{span}(m) = \max_{i=1..n}(t_i) - \min_{i=1..n}(t_i) + 1$$

the number of unmatched terms in the statement fragment matched by m is thus

$$U_m = \mathsf{span}(m) - n$$

With respect to ordering, we can simply consider the distance, expressed in number of terms, between two terms that occupy consecutive positions in the model as an indicator of bad ordering. For each target $t_i, i > 1$, we define the distance D_i as

$$D_i = t_i - t_{i-1} - 1$$

(notice that this distance is 0 when t_{i-1} and t_i point to consecutive terms in the statement, and can be negative in case of an inversion).

Finally, the global matching score for a valid match $m = \langle t_1, \ldots, t_n \rangle$ is given by

$$\begin{aligned} \mathsf{mscore}(m) \quad = \quad & -K_{DROPPED} * U_m \\ & -K_{DIST} * \sum_{i=1..n|D_i>0} D_i \\ & -K_{INVERSION} * \sum_{i=1..n|D_i<0} -D_i \end{aligned}$$

where $K_{DROPPED}$, K_{DIST} and $K_{INVERSION}$ are parameters of the cost model. Notice that $\mathsf{mscore}(m)$ is always less or equal to 0, with 0 assigned to the "perfect" match in which strict ordering is preserved and no term is dropped.

3.4 Rule firing

When CICO determines that a rule should be fired, it applies the bindings established by the match to the action and the substitution part of the rule, executes the action, and replaces the matched fragment in the original statement with the substitution.

Formally, given a term sequence r, a model $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, a statement $s = \langle \omega_1, \ldots, \omega_m \rangle$ and a valid match $m = \langle t_1, \ldots, t_n \rangle$ between μ and s, we define[1]

$$\mathsf{subst}(\mu, m, r) = r \left[{}^{\tau_1}\!/_{\omega_{t_1}} \right] \cdots \left[{}^{\tau_n}\!/_{\omega_{t_n}} \right]$$

The replacement of the fragment of s matched by m with the substitution $\sigma' = \langle \gamma'_1, \ldots, \gamma'_q \rangle$ is defined by

$$\mathsf{repl}(s, \sigma', m) = \langle \omega_1, \ldots, \omega_{min-1}, \gamma'_1, \ldots, \gamma'_q, \omega_{max+1}, \ldots, \omega_m \rangle$$

where

$$\begin{aligned} max \quad &= \quad \max_{i=1..n} (t_i) \\ min \quad &= \quad \min_{i=1..n} (t_i) \end{aligned}$$

To characterize the application and firing of a rule $\rho = (\mu, \alpha, \sigma)$ to a statement s, we first define a *sa-structure* as a pair (s, A) where s is a statement and A is a sequence of actions. The function[2]

$$\mathsf{apply}(\rho, m, (s, A)) = (\mathsf{repl}(s, \mathsf{subst}(\mu, m, \sigma), m), A.\mathsf{subst}(\mu, m, \alpha))$$

returns a sa-structure in which the substitution of ρ has replaced the fragment of s matched by m, and the action of ρ has been appended to A.

[1] We use the notation $\alpha \left[{}^{b}\!/_{c} \right]$ to indicate the sequence α, in which every instance of b has been replaced by c.

[2] We use the notation $A.B$ to indicate the concatenation of the sequence B at the end of A (append).

3.5 Rule scoring

Beside the score due to the ordering of matches, a second score is computed depending on the *specificity* of a rule. Three factors influence specificity of a rule $\rho = (\mu, \alpha, \sigma)$ with $\mu = \langle \tau_1, \ldots, \tau_n \rangle$:

1. the length of the model, $ms = n$;
2. the total number of tags in the model, $ts = \sum_{i=1..n} \#\mathsf{tags}(\tau_i)$
3. the total number of constants in the model, $cs = \#\{\tau_i \mid \mathsf{isconst}(\tau_i)\}$

Moreover, we assign a fixed score to each rule to favor longer chains of rule applications (see Section 3.7). We define then

$$\begin{aligned} \mathsf{rscore}(\rho) &= ms * K_{TERMS} \\ &\quad + ts * K_{TAGS} \\ &\quad + cs * K_{CONSTS} \\ &\quad + K_{RULE} \end{aligned}$$

where K_{TERMS}, K_{TAGS}, K_{CONSTS} and K_{RULE} are parameters of the cost model.

3.6 Rests and rest scoring

A statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ that is not matched by any rule in a rule set R is called a *rest* (with respect to R). The name refers to the fact that the basic parsing algorithm (see Section 3.7) repeatedly applies rules from R to a statement until no rule can be applied. What is left is, indeed, a rest.

The trace of these repeated applications is called a *parsing sequence*. To encourage the selection of parsing sequences that leave as little unparsed material as possible, a score is given to each rest, computed as

$$\mathsf{lscore}(s) = -n * K_{REST}$$

where K_{REST} is a parameter of the cost model.

3.7 Basic parsing algorithm

The basic parsing algorithm implemented by Cico can be described informally as follows:

1. given a statement in input, repeatedly apply all known rules, in turn, trying for each rule all possible valid matches, until no rule can be applied;
2. evaluate the score of each possible parsing sequence;

Algorithm 1 BasicParse(s, R, A, k).

{s is a statement, R is a set of MAS rules, A is a sequence of actions and k is a score}
f := false
for all $\rho \in R$ **do**
  if $\mathsf{match}(\rho, s)$ is complete **then**
    for all m valid matches from $\mathsf{match}(\rho, s)$ **do**
      $f := \mathsf{true}$
      $(s', A') := \mathsf{apply}(\rho, m, (s, A))$
      $k' := k + \mathsf{mscore}(m) + \mathsf{rscore}(\rho)$
      BasicParse(s', R, A', k')
    end for
  end if
end for
if $\neg f$ **then** {no rule applies, we are on a leaf node}
  $k' = k + \mathsf{lscore}(s)$
  if $k' > B_k$ **then**
    $B_k := k'; B_A := A; B_s := s$
  end if
end if
{B_k, B_A, and B_s are respectively the score, the action sequence and the final rest of the best parse sequence, according to the scoring system}

Algorithm 2 SimpleParse($\mathcal{R}, \mathcal{G}, R$).

{$\mathcal{R}$ is a set of statements, $\mathcal{G}$ is a glossary, and R is a set of MAS rules}
for all $r \in \mathcal{R}$ **do**
  $B_k := \textsc{VeryBadScore}$
  BasicParse($r_{|\mathcal{G}}, R, \langle\rangle, 0$)
  if $B_k \geq \textsc{MinimumAcceptableScore}$ **then**
    execute B_A, final rest is B_s
  else
    final rest is $r_{|\mathcal{G}}$
  end if
end for

3. choose the best one and execute the relevant actions;

4. repeat from 1. until input exhausted.

This informal algorithm is described in more formal terms by Algorithms 1 and 2. Notice that Algorithm 1 selects internally the highest-ranking parsing sequence as the final result of the computation. This is used in Circe to ensure that each requirement is interpreted unambiguously by other components of the environment (the same interpretation is also presented back to the user for confirmation). However, it is possible and sensible in different contexts to output *all* the possible parsing sequences, together with their score, leaving to other components — possibly with a better knowledge of the domain — the task of selecting the "right" parsing sequence for a given context.

3.8 Parsing contexts

As introduced in Section 2.2, the usage of lists in the source text can introduce distinct *parsing contexts* in a statement. We will use $\triangleleft$ and $\triangleright$ markers to indicate a parsing context, like in

$$\langle a, b, \triangleleft, c, \triangleleft, d, e, \triangleright, f, g, \triangleright, h, \triangleleft, i, j, \triangleright, k\rangle$$

The format canonization process inserts context delimiters around every item in a list, and around the entire list as a whole. Thus, a statement like

```
a b
- c d
- e
f g
```

becomes

$$\langle \text{a}, \text{b}, \overbrace{\triangleleft, \underbrace{\triangleleft, \text{c}, \text{d}, \triangleright}_{\text{first item}}, \underbrace{\triangleleft, \text{e}, \triangleright}_{\text{second item}}, \triangleright}^{\text{entire list}}, \text{f}, \text{g}\rangle$$

Language-specific optimizers can also insert context in other cases. For example, the optimizers for Italian and English establish contexts around parenthetical text (e.g., "The system shall set the alarm (initially: FALSE) to TRUE"), and around any text between an "if" and a "then" ("se" and "allora" for Italian).

Cico handles a parsing context by computing the best parsing sequence for the context in insulation with respect to the rest of the statement, executing the relevant actions and replacing the entire context with the resulting rest. The process is then iterated until no more contexts are left in the statement, at which time a parsing sequence for the entire statement is computed.

This process is described formally by Algorithms 3 and 4, where the functions introduced below are used.

Given a statement $s = \langle \omega_1, \ldots, \omega_n \rangle$ with embedded, correctly-nested context markers $\triangleleft$ and $\triangleright$, we define

$$\mathsf{hascontext}(s) = \exists i \in [1, n] \text{ s.t. } \omega_i = \text{“}\triangleleft\text{”}$$

A *context pointer* is a pair of indexes pointing to two matching context markers in s. In particular, we define the leftmost innermost context in s by

$$\mathsf{firstcontext}(s) = (p, q)$$

where

$$\begin{aligned} p &= \max_{i=1..n} i \text{ s.t. } \omega_i = \text{“}\triangleleft\text{”} \;\wedge\; \forall h = 1..i-1, \omega_h \neq \text{“}\triangleright\text{”} \\ q &= \min_{j=p+1..n} j \text{ s.t. } \omega_j = \text{“}\triangleright\text{”} \end{aligned}$$

Given a statement s as above, a context c pointed by a context pointer (p, q), and a second statement $r = \langle \zeta_1, \ldots, \zeta_m \rangle$, we also define the following two operations to extract c from s and to replace c in s by r:

$$\begin{aligned} \mathsf{extractcontext}(s, c) &= \langle \omega_{p+1}, \ldots, \omega_{q-1} \rangle \\ \mathsf{replacecontext}(s, c, r) &= \langle \omega_1, \ldots, \omega_{p-1}, \zeta_1, \ldots, \zeta_m, \omega_{q+1}, \ldots, \omega_n \rangle \end{aligned}$$

These operations are used in the full parsing algorithm discussed below.

3.9 Full parsing algorithm

The full parsing algorithm described by Algorithms 3 and 4 works by parsing deep contexts in insulation, accumulating resulting actions and substituting final rests in place of the original context.

It is worthwhile to notice that unparseable contexts are simply substituted back in the enclosing context, and that parsing is attempted again on the resulting block. Thus, generally context insertion does not change the resulting action sequence for a statement.

Although rare, however, there are cases in which the insertion of a context can change the parsing sequence. This can happen when multiple rules with the same score can be applied to the text inside a context, and a third rule acts as a discriminator when applied outside the context. For example, consider the rule set containing the following three rules:

$$\begin{aligned} \rho_1 &= (\langle A, B \rangle, \alpha_1, \langle S \rangle) \\ \rho_2 &= (\langle A, B \rangle, \alpha_2, \langle T \rangle) \\ \rho_3 &= (\langle T, C \rangle, \alpha_3, \langle \rangle) \end{aligned}$$

Algorithm 3 ContextParse(s, R).

{s is a statement, R is a set of MAS rules}
while $\mathsf{hascontext}(s)$ **do**
  $c := \mathsf{firstcontext}(s)$
  $r := \mathsf{extractcontext}(s, c)$
  $B_k :=$ VeryBadScore
  BasicParse($r, R, \langle\rangle, 0$)
  if $B_k \geq$ MinimumAcceptableScore **then**
    execute B_A
    $s := \mathsf{replacecontext}(s, c, B_s)$
  else
    $s := \mathsf{replacecontext}(s, c, r)$
  end if
end while
$B_k :=$ VeryBadScore
BasicParse($s, R, \langle\rangle, 0$)
if $B_k \geq$ MinimumAcceptableScore **then**
  execute B_A, final rest is B_s
else
  final rest is s
end if

Algorithm 4 FullParse($\mathcal{R}, \mathcal{G}, R$).

{$\mathcal{R}$ is a set of statements, $\mathcal{G}$ is a glossary, and R is a set of MAS rules}
for all $r \in \mathcal{R}$ **do**
  ContextParse($r_{|\mathcal{G}}, R$)
end for

and the two statements, differing only in the presence of context markers around “A B”,

$$\begin{aligned} s_1 &= \langle A, B, C \rangle \\ s_2 &= \langle \triangleleft, A, B, \triangleright, C \rangle \end{aligned}$$

The application of FullParse() to s_1 finds that the best parsing sequence is given by $\langle \rho_2, \rho_3 \rangle$ (with the obvious matches), and causes the execution of α_2 and α_3, as one would expect. When applied to s_2, however, FullParse() would first call BasicParse() on $\langle A, B \rangle$, that could non-deterministically result either in the application of ρ_1, leading to a rest $\langle S, C \rangle$, or in the application of ρ_2, leading to $\langle T, C \rangle$. Both parsing sequences, in fact, have the same score. If the latter is chosen, parsing proceeds by firing ρ_3, thus causing the execution of α_2 and α_3, as in the previous case. But if the former is chosen, no rule matching $\langle S, C \rangle$ would be found, thus terminating with an action sequence $\langle \alpha_1 \rangle$ and a rest $\langle S, C \rangle$.

In practical applications, this kind of ambiguity is extremely rare, and if present could be resolved by avoiding the use of context-introducing markup (e.g., lists).

4 Special forms of tagging

Three special forms of tagging exists, to support higher-level features of the Circe toolset. *Speculative tagging* is a means to leave the tag set for a term unspecified, and instruct the parser to compute a set of tags for the term that maximizes overall parsing score. Backward and forward *lookup references* support the implementation of pronouns, and specify that a term should be considered equivalent to a previous (resp. following) term in the statement, choosing the one that maximizes parsing score. Finally, *wildcard tagging* supports model terms that match any tagged term in the statement — regardless of the specific tags it has.

4.1 Speculative tagging

Speculative tagging is syntactically denoted by the special tag `/?`. Its effect is as follows: when testing for containment among tag sets in the definition of $\mathsf{target}()$ (see page 10), if $\texttt{/?} \in \mathsf{tags}(\omega)$ the test is always considered passed, and when the corresponding rule is applied, a special action is executed to record the tag set that is being speculatively associated with the given term. Formally, the definition of $\mathsf{target}()$ is changed to

$$\mathsf{target}(\tau, s) = \begin{cases} \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} \cup \{i \mid \texttt{/?} \in \mathsf{tags}(\omega_i)\} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

Next, we introduce the special action $\alpha^?(\omega, T)$ whose purpose is to record that term ω is being speculatively assigned all the tags in the set T. Given a model $\mu = \langle \mu_1, \ldots, \mu_n \rangle$, a match $m = \langle t_1, \ldots, t_n \rangle$, and a statement $s = \langle \omega_1, \ldots, \omega_p \rangle$, we also define the *reverse match map*

$$\mathsf{rmap}(i, m) = \begin{cases} j & \text{if } \exists j \text{ s.t. } t_j = i \\ 0 & \text{otherwise} \end{cases}$$

and the auxiliary function

$$\mathsf{rst}(\mu, m, s, i) = \begin{cases} \langle \alpha^?(\omega_i, \mathsf{tags}(\mu_{\mathsf{rmap}(i,m)})) \rangle & \text{if } \mathsf{/?} \in \mathsf{tags}(\omega) \wedge \mathsf{rmap}(i, m) \geq 1 \\ \langle \rangle & \text{otherwise} \end{cases}$$

The definition of $\mathsf{apply}()$ thus becomes

$$\mathsf{apply}(\rho, m, (s, A)) = (\mathsf{repl}(s, \mathsf{subst}(\mu, m, \sigma), m), A.\mathsf{rec}_?(\mu, m, s).\mathsf{subst}(\mu, m, \alpha))$$

where

$$\mathsf{rec}_?(\mu, m, s) = \mathsf{rst}(\mu, m, s, 1). \cdots .\mathsf{rst}(\mu, m, s, p)$$

As an example of application of speculative tagging, consider the rule $\rho = (\langle x, y/\mathrm{TAG}_1, C \rangle, \langle \alpha_0 \rangle, \langle S \rangle)$[3] and the statement $s = \langle A, B/?, C \rangle$. By using the $\mathsf{target}()$ definition above, the matching function would yield

$$\begin{aligned} \mathsf{match}(\rho, s) &= \langle \mathsf{target}(x, s), \mathsf{target}(y/\mathrm{TAG}_1, s), \mathsf{target}(C, s) \rangle \\ &= \langle \{1, 2, 3\}, \{2\}, \{3\} \rangle \end{aligned}$$

This matching map is complete; of all the possible matches from the matching map, only the match $m = \langle 1, 2, 3 \rangle$ is valid (and has an acceptable score in the sense of algorithm 1), so the rule can be applied. In particular,

$$\mathsf{rec}_?(\langle x, y/\mathrm{TAG}_1, C \rangle, m, s) = \langle \rangle.\langle \alpha^?(B, \{\mathrm{TAG}_1\}) \rangle.\langle \rangle$$

so that the $\mathsf{apply}()$ function results in

$$\mathsf{apply}(\rho, m, (s, \langle \rangle)) = (\langle S \rangle, \langle \alpha^?(B, \{\mathrm{TAG}_1\}), \alpha_0 \rangle)$$

as expected. In other words, given the glossary declaration `B/?` meaning "what are the best tags for `B`?", the parsing process provides the answer "`B/TAG`$_1$ is the best declaration", where 'best' is always meant with respect to the scoring system and to the rule set in use, for the purpose of maximizing parsing score.

[3]Recall that lower case indicates variable terms, while upper case indicates constant terms.

4.2 Lookup references

Lookup references, denoted by `/<` (resp. `/>`), let a term inherit the properties (tags) of another term to its left (resp. right) in a statement. Whereas speculative tagging permits a term to assume *any* tag if that improves parsing score, lookup references only permit a term to assume the tags of another term in the same statement. Moreover, the assignment is not recorded as a special action, but rather the referred term is directly substituted for the reference in the action and in the substitution of the rule. Formally, the definition of $\mathsf{target}()$ is extended again to account for backward (`/<`) and forward (`/>`) references:

$$\mathsf{target}(\tau, s) = \begin{cases} \mathit{Std} \cup \mathit{Spec} \cup \mathit{BRef} \cup \mathit{FRef} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

where

$$\begin{aligned} \mathit{Std} &= \{i \mid \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_i)\} \\ \mathit{Spec} &= \{i \mid \texttt{/?} \in \mathsf{tags}(\omega_i)\} \\ \mathit{BRef} &= \{i \mid (\texttt{/<} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j < i \wedge \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_j))\} \\ \mathit{FRef} &= \{i \mid (\texttt{/>} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j > i \wedge \mathsf{tags}(\tau) \subseteq \mathsf{tags}(\omega_j))\} \end{aligned}$$

In the same spirit, the $\mathsf{subst}()$ function must be refined to transparently substitute referred terms for matched references. Given a term sequence r, a model $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, a statement $s = \langle \omega_1, \ldots, \omega_m \rangle$, and a valid match $m = \langle t_1, \ldots, t_n \rangle$ between μ and s, we define

$$\mathsf{subst}(\mu, m, r) = r \left[{}^{\tau_1}\!\big/_{\mathsf{ref}(\tau_1, s, m, t_1)} \right] \cdots \left[{}^{\tau_n}\!\big/_{\mathsf{ref}(\tau_n, s, m, t_n)} \right]$$

where

$$\mathsf{ref}(\tau, s, m, l) = \begin{cases} \omega_{\mathsf{bref}_<(\tau, s, m, l)} & \text{if } \texttt{/<} \in \mathsf{tags}(\omega_l) \\ \omega_{\mathsf{bref}_>(\tau, s, m, l)} & \text{if } \texttt{/>} \in \mathsf{tags}(\omega_l) \\ \omega_l & \text{otherwise} \end{cases}$$

and the best referred term among possibly many is determined by the following functions:

$$\begin{aligned} \mathsf{bref}_<(\tau, s, m, l) &= \begin{cases} \max\{i \in \mathit{Std} \mid i < \min m\} & \text{if } \exists i \in \mathit{Std} \text{ s.t. } i < \min m \\ \min\{i \in \mathit{Std} \mid i < l\} & \text{otherwise} \end{cases} \\ \mathsf{bref}_>(\tau, s, m, l) &= \begin{cases} \min\{i \in \mathit{Std} \mid i > \max m\} & \text{if } \exists i \in \mathit{Std} \text{ s.t. } i > \max m \\ \max\{i \in \mathit{Std} \mid i > l\} & \text{otherwise} \end{cases} \end{aligned}$$

For example, consider a statement s and a rule ρ defined by

$$\begin{aligned} s &= \langle \texttt{A/T}_1\texttt{/T}_2, \texttt{B}, \texttt{C/<} \rangle \\ \rho &= (\langle x/T_1, B, y/T_2 \rangle, \langle S, x, y \rangle, \langle U, y \rangle) \end{aligned}$$

The statement term `C/<` could stand either for $\mathtt{A/T_1/T_2}$ or for `B`. In fact, the matching map

$$\mathsf{match}(\rho, s) = \langle\{1,3\},\{2\},\{1,3\}\rangle$$

is complete, and produces the two valid matches $m_1 = \langle 1,2,3\rangle$ and $m_2 = \langle 3,2,1\rangle$. Of these, the cost model selects the first as the best one. The action to execute is computed as follows:

$$\begin{aligned}
&\mathsf{subst}(\mu, m, \langle S, x, y\rangle) = \\
&\quad= \ \langle S,x,y\rangle \left[{}^{x}\middle/\mathsf{ref}(x/T_1,s,1)\right] \left[{}^{B}\middle/\mathsf{ref}(B,s,2)\right] \left[{}^{y}\middle/\mathsf{ref}(y/T_2,s,3)\right] \\
&\quad= \ \langle S,x,y\rangle \left[{}^{x}/\omega_1\right] \left[{}^{B}/\omega_2\right] \left[{}^{y}\middle/\omega_{\mathsf{bref}_<(y/T_2,s,3)}\right] \\
&\quad= \ \langle S,x,y\rangle \left[{}^{x}/\omega_1\right] \left[{}^{B}/\omega_2\right] \left[{}^{y}/\omega_1\right] \\
&\quad= \ \langle S,x,y\rangle \left[{}^{x}\middle/A/T_1/T_2\right] \left[{}^{B}/B\right] \left[{}^{y}\middle/A/T_1/T_2\right] \\
&\quad= \ \langle S,\ A/T_1/T_2,\ A/T_1/T_2\rangle
\end{aligned}$$

Similarly, the computation of the substitution yields

$$\mathsf{subst}(\mu, m, \langle U, y\rangle) \quad = \quad \langle U,\ A/T_1/T_2\rangle$$

4.3 Wildcard tagging

Wildcard tagging, denoted by a `/*` tag, is a way to specify that a term in the model part of a rule should match any term in the statement that has at least a tag. This is different from simply using in the model a variable term with no tag: in that case, *any* term would be matched, by definition of $\mathsf{target}()$, including terms with an empty tag set. Formally, the definition of $\mathsf{target}()$ is extended once more:

$$\mathsf{target}(\tau, s) = \begin{cases} \mathit{Std} \cup \mathit{Spec} \cup \mathit{BRef} \cup \mathit{FRef} \cup \mathit{Wild} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

where *Std*, *Spec*, *BRef*, and *FRef* are the same as before, and *Wild* is defined as follows:

$$\mathit{Wild} = \begin{cases} \{i \mid \#\mathsf{tags}(\omega_i) \geq 1\} & \text{if } \mathtt{/*} \in \mathsf{tags}(\tau) \\ \{\} & \text{otherwise} \end{cases}$$

To reflect the intuition that precise matches are more relevant than wildcard matches, we also update the definition of $\mathsf{rscore}()$:

$$\begin{aligned}
\mathsf{rscore}(\rho) \quad = \quad & ms * K_{TERMS} \\
& + ts * K_{TAGS} \\
& + cs * K_{CONSTS} \\
& + K_{RULE} \\
& + ws * K_{WILDS}
\end{aligned}$$

where $ws = \#\{i \mid \texttt{/*} \in \mathsf{tags}(\tau_i)\}$ and K_{WILDS} is a parameter of the cost model.

As an example of application of wildcard tagging, consider a statement $s = \langle \texttt{A/T}_1, \texttt{B/T}_2, \texttt{C/<} \rangle$ and a rule $\rho = \langle x/T_1, y/* \rangle$. The matching map would be

$$\mathsf{match}(\rho, s) = \langle \{1,3\}, \{1,2,3\} \rangle$$

with valid matches $m_1 = \langle 1,2 \rangle$, $m_2 = \langle 1,3 \rangle$, $m_3 = \langle 3,1 \rangle$, $m_4 = \langle 3,2 \rangle$ (of which m_1 has the best score).

4.4 Optional terms

Optionality of a term in the model part of a rule is indicated by a /O tag. When a term in a model has this tag, the rule can be fired even if no match for the term can be found. To formally define this mechanism, we introduce a special term representing "no match", that we denote with $\perp$. The target for an optional term can thus be any standard term, or $\perp$:

$$\mathsf{target}(\tau, s) = \begin{cases} \mathit{Std} \cup \mathit{Spec} \cup \mathit{BRef} \cup \mathit{FRef} \cup \mathit{Wild} \cup \mathit{Opt} & \text{if } \mathsf{isvar}(\tau) \\ \{i \mid \mathsf{term}(\tau) = \mathsf{term}(\omega_i)\} & \text{if } \mathsf{isconst}(\tau) \end{cases}$$

where *Opt* is defined as follows:

$$\mathit{Opt} = \begin{cases} \{\perp\} & \text{if } \texttt{/O} \in \mathsf{tags}(\tau) \\ \{\} & \text{otherwise} \end{cases}$$

Moreover, we must refine *Std*, *BRef*, and *FRef*, in order to ignore the /O tag in the model term when checking for tag inclusion:

$$\begin{aligned} \mathit{Std} &= \{i \mid (\mathsf{tags}(\tau) \sqsubseteq \mathsf{tags}(\omega_i)\} \\ \mathit{BRef} &= \{i \mid (\texttt{/<} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j < i \wedge \mathsf{tags}(\tau) \sqsubseteq \mathsf{tags}(\omega_j))\} \\ \mathit{FRef} &= \{i \mid (\texttt{/>} \in \mathsf{tags}(\omega_i)) \wedge (\exists j \text{ s.t. } j > i \wedge \mathsf{tags}(\tau) \sqsubseteq \mathsf{tags}(\omega_j))\} \end{aligned}$$

where the new operator $\sqsubseteq$ on tag sets is defined as

$$t_1 \sqsubseteq t_2 \quad \overset{def}{\Longleftrightarrow} \quad (t_1 \setminus \{\texttt{/O}\}) \subseteq t_2$$

Finally, we must specify how $\perp$'s should be handled in substitutions. To this end, it is sufficient to refine the ref() function that we introduced in Section 4.2 so that the empty string[4] ϵ is substituted when an optional term has no match:

$$\mathsf{ref}(\tau, s, m, l) = \begin{cases} \omega_{\mathsf{bref}_<(\tau,s,m,l)} & \text{if } \texttt{/<} \in \mathsf{tags}(\omega_l) \\ \omega_{\mathsf{bref}_>(\tau,s,m,l)} & \text{if } \texttt{/>} \in \mathsf{tags}(\omega_l) \\ \epsilon & \text{if } l = \perp \\ \omega_l & \text{otherwise} \end{cases}$$

[4]In the implementation, the empty string is represented by an underscore character in action parts. An explicit representation is needed to ensure that the arguments' positions in the output do not vary depending on whether an optional term was matched or not. Notice however that in substitution parts, empty strings are used for ϵ instead.

Valid matches (from Section 3.2) are also naturally extended to accommodate multiple $\perp$'s. Given a complete matching map $M = \langle T_1, \ldots, T_n \rangle$, a *valid match* m from M is defined as follows:

$$\begin{array}{rl} m = \langle t_1, \ldots, t_n \rangle \text{ s.t.} \quad \forall i = 1..n, & (t_i \in T_i) \quad \wedge \\ & (t_i = \perp \quad \vee \quad \forall j = 1..n, t_i = t_j \Rightarrow i = j) \end{array}$$

To reflect the intuition that having a match is generally preferable than having no match at all, even for optional terms, we also update the definition of $\mathsf{mscore}()$:

$$\begin{aligned} \mathsf{mscore}(m) \quad = \quad & -K_{DROPPED} * U_m \\ & -K_{DIST} * \sum_{i=1..n|D_i>0} D_i \\ & -K_{INVERSION} * \sum_{i=1..n|D_i<0} -D_i \\ & -K_{OPT} * O_m \end{aligned}$$

where O_m is the number of $\perp$'s in m, and K_{OPT} is a parameter of the cost model.

4.5 Negative tags

When performing a substitution, the $r\left[{}^{a}\!/_{b}\right]$ operator substitutes every instance of a in r with b, where a and b are terms, and r is a sequence of terms. To be more precise, a is replaced by the base term of b, with tags obtained by merging the tags of a and b. Formally,

$$\begin{aligned} \langle\rangle\left[{}^{a}\!/_{b}\right] &= \langle\rangle \\ (\langle x\rangle.r)\left[{}^{a}\!/_{b}\right] &= \langle x\rangle.(r\left[{}^{a}\!/_{b}\right]) \text{ if } x \neq a \\ (\langle a\rangle.r)\left[{}^{a}\!/_{b}\right] &= \langle c\rangle.(r\left[{}^{a}\!/_{b}\right]) \end{aligned}$$

where c is defined by $\mathsf{term}(c) = \mathsf{term}(b)$ and $\mathsf{tags}(c) = \mathsf{tags}(a) \cup \mathsf{tags}(b)$. While this is generally the desired behavior, in some case (especially when simply rewriting fragments of text), it can be preferable to alter the set of tags of a substitution in a more flexible way. To this end, we define *negative tags*, as a way to remove specific tags from the resulting substitution.

Negative tags are denoted by inserting a minus sign between the slash and the name of the tag. Thus, the negative form of a tag `/TAG` is `/-TAG`. Negative tags are allowed only in the action and substitution part of a MAS rule, thus matching and scoring do not need any modification to support this kind of tags. Actually, only the basic replacement operator defined above needs to be refined. In particular, the replaced instance c is now defined by

$$\begin{aligned} \mathsf{term}(c) &= \mathsf{term}(b) \\ \mathsf{tags}(c) &= (\mathsf{tags}(a) \setminus \mathsf{tags}^{-}(b)) \cup \mathsf{tags}^{+}(b) \end{aligned}$$

where $\mathsf{tags}^-(b)$ is the set of the negative tags among the tags of b, and $\mathsf{tags}^+(b)$ is the set of the positive (normal) tags among the tags of b. Obviously, $\mathsf{tags}(b) = \mathsf{tags}^+(b) \cup \mathsf{tags}^-(b)$.

It is worthwhile to remark that negative tags can only be used in actions and substitutions: in particular, they cannot be used in the source text, nor in the model parts of MAS rules.

As an example of application of this kind of tagging, consider the statement s and the rule ρ defined by

$$\begin{aligned} s &= \langle \mathtt{A/T_1/T_3, B, C/T_2} \rangle \\ \rho &= (\langle x/T_1, B, y\rangle, \langle S, x/T_4/{-T_3}, y{-T_4}\rangle, \langle U, y/{-T_2}\rangle) \end{aligned}$$

The model part of the rule will match the text s, and assuming that no other rule in the current rule set can be applied, ρ is fired. In particular, x is bound to $A/T_1/T_3$, and y to C/T_2. In the action part, $/T_4$ is added to x, and T_3 is removed, thus obtaining $A/T_1/T_4$. Analogously, $/T_4$ is removed from y, but since the term bound to y did not have $/T_4$ anyway, the result is simply C/T_2. The final action is thus $\langle S, A/T_1/T_4, C/T_2\rangle$. The same process is applied to the substitution part, producing $\langle U, C\rangle$.

5 Implementation and evaluation

5.1 Implementation issues and optimizations

In implementing the algorithms described in the previous section, particular care has been taken with respect to efficiency. In fact, a number of optimization techniques and heuristics have been introduced in the parsing algorithm.

Among the code-level optimizations, the following are particularly relevant:

- tokenization is applied to all input text. Each string is represented by an integer hash code (a token); tagged terms are represented by a token for the base term and by a sorted, variable-length, array of tokens for the tags. By using this representation, the function $\mathsf{target}(\tau, s)$ can be implemented as a comparison between two integers (if τ is a constant) or as a linear[5] comparison among (small) arrays of integers (if τ is a variable).

- the optimized code for the parser is less than 16Kb in size, so that it can reside entirely in most contemporary microprocessors' cache. Also, most data structures are aligned and arranged so that the access

[5]Although better comparison strategies exists for ordered arrays, the typical size of a tag list is rather small — one to four elements —, making linear comparison faster in practice.

patterns of the parsing algorithm exhibits a high degree of locality, thus minimizing cache misses.

Such code-level optimizations, however, can only improve performance by a constant factor. More relevant improvements are obtained by algorithm-level optimizations, like the following:

- matches in which consecutive terms in the model point to terms in the statement that are more than MaxDistance terms apart each other are called *thin matches*, and are not considered valid (in the sense of Section 3.2). If a matching map does not contain at least a valid match that is not thin, the entire matching map is not considered complete (and thus the corresponding rule is not applicable).

- *early abandon* of a parse sequence that is being explored is activated when any of the following three conditions is met:

  1. the current score is worse than BadThreshold;
  2. the current score is worse than LevelMult times the depth of the current parse sequence, plus BadThreshold;
  3. the difference between the current score and the previous score (i.e., the score before the last parsing step) is worse than BadImprovement.

Each of the four heuristic optimizations cited above can be enabled or disabled independently, according to the needs of a particular configuration. In the standard configuration, thin matches refusal is disabled, while all three early abandon clauses are enabled.

5.2 Coverage

The particular applications at which Cico is directed make it difficult to formally define a measure for coverage. In classical usage, a parser for natural language is applied to a known corpus of text, and coverage is defined as the percentage of sentences from the corpus that have been parsed correctly. This measure is relevant under the assumption that the parser should parse correctly *any* sentence in the given language. This is a reasonable assumption for grammar-based parsing, but not for domain-based parsing. In fact, Cico is used in two contexts:

1. during requirements writing, with a fixed rule set. In this case there is no reference corpus, and instead the requirements writer is encouraged to rewrite unparsed or wrongly parsed requirements *that convey information in one of the known domains* in order to make them parseable. The assumption in this case is that Cico's rule sets constitute a style

manual for the domain. Notice that CICO's inability to parse requirements that are not relevant for one of the known domain is not to be counted as a parsing failure. Rather, it should be considered either as an indication that the set of models in use is not sufficient with respect to the domain being addressed (e.g., there is no user interface model defined while writing usability requirements for a system), or that the unparsed requirement is not significant for the analysis (e.g., a requirement prescribes voltage constraints on certain output lines, whereas the requirements engineer is more interested in studying the system's time behavior).

2. during requirements analysis, with a fixed requirements document. In this case, there is no reference rule set, and instead a rule set is defined to cover the language used in a document, *thus defining a new domain* that CICO should be able to parse. [8] gives an account of a case study in which this application was relevant.

While no exact measure can be given of coverage, there is empirical evidence that CICO's coverage is adequate for the kind of documents at which it is aimed. Indeed, several dozens projects, both industrial and academic, have succeeded in obtaining complete parsing of requirements document describing a wide array of systems.

5.3 Performance

CICO's basic parsing algorithm does not guarantee convergence. In fact, any rule whose model matches its substitution, if fired once, would cause the algorithm to run forever.[6] The obvious extension to cases where multiple rules are involved in a loop also applies.

However, sufficient conditions for convergence can be given. In particular, the basic parsing algorithm terminates on any input if the following condition holds on the rule set R:

$$\forall(\mu, \alpha, \sigma) \in R, \mathsf{len}(\mu) > \mathsf{len}(\sigma)$$

where the function $\mathsf{len}()$ is defined as follows:

$$\mathsf{len}(\langle a_1, \ldots, a_n \rangle) = n$$

This is indeed the case for the applications CICO is geared to. When building parse trees, the substitution is simply a 1-term reference to the parse tree for the substituted fragment, so that convergence is usually reached in only a few iterations (corresponding to a number of nested recursive calls

[6]Naturally, CICO provides configurable timeouts — both in terms of parse tree depth and of wall clock time — to avoid getting stuck in similar cases.

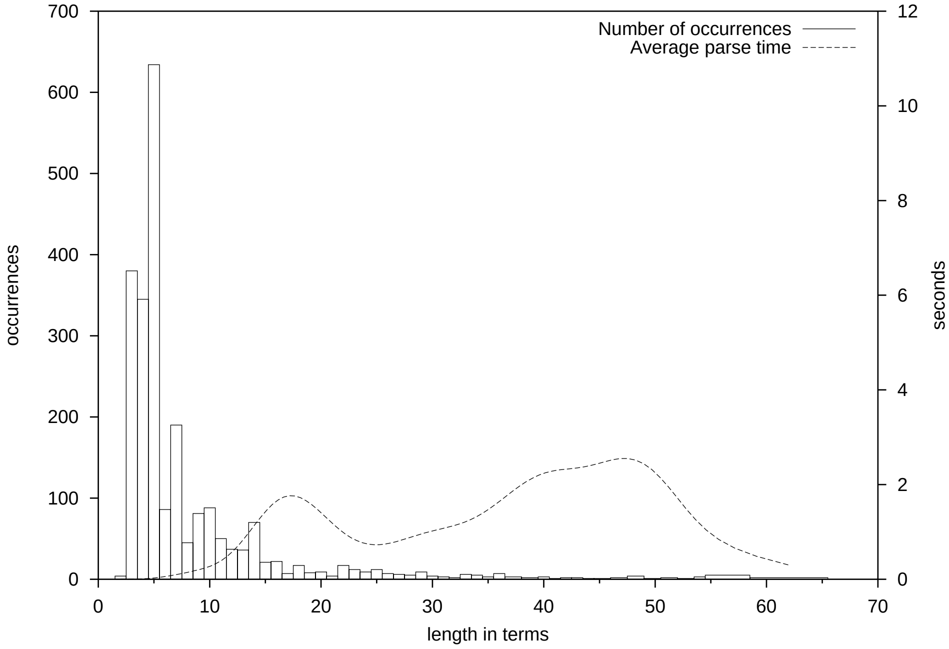

Figure 1: Frequency and parse times at different sentence length.

of BasicParse() in the order of 10). Actually, heuristics, contexts and a restricted rule set make CICO's parse algorithm run almost in linear time in most cases.

To study CICO's performance in real cases, we analyzed a set of 2276 requirements (statements) from 32 projects, representative of CICO's normal usage, with a rule set consisting of 146 rules in various domains. The source text for those statements contained 26467 words, that were reduced to 18037 terms by input canonization and glossary substitution (that included stopwords removal). After canonization, statements ranged in length from 2 terms to a rather exceptional 62 terms. The histogram in Figure 1 shows how statements were distributed according to their length. As can be seen, most statements were only 3 to 7 terms long, corresponding to simple, atomic statements (e.g., “The server receives transactions from a terminal”). The second data series in Figure 1 refers to the average parse time for statements of a given length. Its distinctive features can be explained by considering changes that occur in input data when statements gets too long:

- the first peak, at statement length 17, corresponds to the introduction of context-delimiting structures in the text. The effect of the introduction of progressively more pervasive contexts can be observed up to length 25, when the increase in length once again offsets the gain obtained by the use of contexts;

- the second peak, at statement length 47, corresponds to a sort of self-selective effects. In fact, as statements increase in length, the person writing them tends to split them in smaller pieces, with only the simplest (and thus, faster to parse) long statements remaining intact.

In both cases, experimental evidence indicate that a psychological threshold exists around 2-3 seconds of parse time, prompting changes in the statements when parse time for a statement grows above that limit.

A few observations can be made on the empirical data in Figure 1 to understand the scale of performance in a real-life situation:

- 80% of all requirements were 10 terms long or shorter; average parse time for those requirements was 0.048 seconds;

- 90% of all requirements were 15 terms long or shorter; average parse time for those requirements was 0.061 seconds;

- the longest requirement in our data set was 62 terms long. Average parse time for *all* requirements was 0.17 seconds

- average parse time for an entire requirement *document* was 12.58 seconds;

- all times were taken on a "slow" machine (powered by a Pentium 133MHz processor from Intel). Since CICO's performances depend almost exclusively on its CPU power, contemporary hardware exhibits significant improvements (e.g., more than 20:1 on a dual-processor machine with 600MHz Pentium IIIs from Intel, that would have corresponded to an average parse time for all requirements of 0.008 seconds).

While less than 13 seconds to parse a requirements document can be considered acceptable in batch processing, this time is excessive for interactive processing. One technique to overcome this limitation consists in caching the results of a parse run and only re-parse changed or added statements when a subsequent version for the same document is submitted for parsing. Experimental results on this *differential parsing* approach, based on the analysis of almost 200 submissions of a single, quite complex requirements document during its evolution, indicate typical parse times for a new version of approximatively 1.2 seconds, versus an average 40 seconds for a complete re-parse — a speedup factor of 25.1, and well below the acceptability limit for interactive use.

5.4 Cost model parameters

Table 2 lists the actual numeric values for the parameters used by CICO's scoring system. The specific values shown have proved effective in practical

Parameter	Value	Used in
$K_{DROPPED}$	1.00	
K_{DIST}	0.10	mscore()
$K_{INVERSION}$	0.60	
K_{OPT}	0.03	
K_{TERMS}	0.6	
K_{TAGS}	0.2	
K_{WILDS}	-0.1	rscore()
K_{CONSTS}	0.3	
K_{RULE}	1.0	
K_{REST}	0.6	lscore()
VeryBadScore	-100.0	algorithms
MinimumAcceptableScore	-4.0	
BadThreshold	-4.0	
LevelMult	0.3	optimizations
BadImprovement	-2.0	

Table 2: Parameter values for the scoring system.

use with the Italian language, but most of the parameters are non-critical, and can be adjusted (at single sentence level) according to the needs of a particular language or application. For example, "rule score" in English is a very different concept from "score rule", and the value for $K_{INVERSION}$ should be adjusted accordingly. Also, the various thresholds for optimizations may be adjusted depending on the performance of the host machine on which Cico is run.

By tuning these parameters, Cico can be adapted to different tasks, from classic information extraction / template filling (by choosing liberal values) to enforcing a strict adherence to some controlled natural language in an editing environment (by choosing restrictive values). Moreover, all parameters can be changed dynamically and on a single-sentence grain. This feature has proven important in technical writing courses, in which at first adherence to precise standards is checked, and if the check is failed, more liberal information extraction is used to identify the important features of the text and suggest alternative formulations to the student.

6 Conclusions

While the application of natural language processing techniques to the analysis of software requirements is by no means novel (e.g., [1, 5, 6, 12, 15, 16, 18]), our experiences indicate that the domain-based parsing technique used by Cico if particularly well suited to the analysis of technical documents. In fact, not only domain-based parsing is highly efficient in terms of parsing times (due to the higher specificity of domain-based rules with respect to classical grammar rules), but it also rejects sentences that are grammatically correct, but not semantically well-formed, already at the parsing stage. As an example, consider the statement below:

> *Whenever the system receives a command from a remote terminal, it shall append it to the log file.*

where *system* is tagged as `/ELAB` (i.e., a computational device), *command* and *log file* are tagged `/INF` (i.e., information packets), and *remote terminal* is tagged `/IN/OUT` (i.e., an I/O device). While from a purely grammatical perspective both occurrences of "it" could refer either to the system, to the command or to the terminal, semantic-based rules have no problems in associating the first "it" to the system, and the second one to the command (see Section 4.2), since the relevant MAS rule would have a model similar to

```
agent/ELAB APPENDS data/INF TO collection/INF
```

and thus the *it* matching `data/INF` can only associate with the previous[7] occurrence of *command*, while the *it* matching `agent/ELAB` can only refer to *system*. In the same spirit,

> *the command appends the terminal to the log file*

would be rejected altogether, since no `/ELAB` term can be found to match the `agent/ELAB` in the model.

For all the advantages it may offer, domain-based parsing still requires a careful study of the domain, to identify the right abstractions (semantic classes) to use as tags in the glossary, and demands accurate crafting of MAS rules, to precisely define the kind and style of language that is to be parsed. While the fuzziness of Cico's algorithm simplifies this task, by reducing the number of rules that have to be defined, creating a rule set for a specific domain still requires a thorough understanding of the domain.

We are currently working in the area of automatic synthesis of MAS rules from unannotated corpora, in the spirit of [10] and [3]. Preliminary results have been applied to the extraction of information from financial bulletins (in Italian). In our initial experiments, this approach was able to discover the concepts of "thing with a variable value" (i.e., shares and currencies), of "marketplace" (i.e., NYSE and Piazza Affari) and of "market index" (i.e., Dow Jones, Nasdaq, and MIB30), proposing rules like[8]

[7] Actually, preference is given to a precedent occurrence of an /INF term, resorting to subsequent occurrences if no suitable precedent occurrence is found.

[8] The labels (tags) assigned to the semantic classes identified by the tools are, naturally, our own.

`index/INDEX LOST howmuch/NUM`

and

`share/VALUED SUSPENDED AT exchange/MARKET`

However, automatically distinguishing between domain-based and grammar-based rules seems extremely difficult, and in our experiments we obtained many more grammatical patterns (e.g., `a/ADJECTIVE n/NOUN`) than rules with a specific semantics in the domain; we are currently working to address these shortcomings.

Part II
Cico User Manual

7 The parser

In this section, we provide detailed information on the usage of `cico3`, a program that implements the Cico parsing algorithm that we have discussed so far. This information is current as of the date of writing, and corresponds to version 3.5 of the software. The executables and associated files can be found on the World Wide Web at the following URL:

`http://circe.di.unipi.it/Cico`

7.1 Invocation syntax

The executable command for Cico is called `cico3`. The invocation syntax is

```
cico3 [MAS files...]
```

the command `cico3` can be followed by any number of file specifications. The files given as arguments contain the MAS rules that will be used for parsing the text read from *stdin*. Currently, `cico3` does not understand any option (not even `--help`).

After reading all the MAS rules from the files specified, the parser reads sentences to parse from *stdin*, one per line, and applies the MAS rules in order to produce a parse tree, according to the algorithm described in Part I. The *actions* from the rules that provide the best parse tree are output, in the format specified by the `USE_FRAMES` option, to *stdout*. The user can set a maximum for the amount of time that `cico3` can devote to the parsing of a single requirement. If this maximum is reached, `cico3` stops parsing the current line, outputs the best parse tree found so far, preceded by a special `CICOTIMEOUT` tuple, and proceeds to the next line.

When the input is exhausted, `cico3` emits a special `CICOSRFINISHED` tuple, so that any other program observing the output can know that the parsing process completed successfully.

To ease synchronization with other programs, `cico3` flushes its output after each write. Thus, the output always accurately reflect the progresses of the parsing process.

7.2 Input file format

The input to `cico3` must obey a number of rules on content and format. The input is interpreted as a series of *lines*, separated according to the local operating system rules for *newline* (e.g., LF on UNIX, CR on MacOS, CR/LF on MS-DOS, etc.).

Lines can contain *directives* that influence the parsing or, more commonly, *text* to be parsed. A number of tools and filters are provided in the `bin/` directory of the distribution package to prepare input files for `cico3`; these tools are more thoroughly discussed in Section 8.

7.2.1 Text lines

Text lines consist in a sequence of *terms*, with each term optionally followed by a sequence of tags. Terms are separated by spaces, while tags are separated by slashes. A typical text is thus

```
terminal/IN/OUT send password/INF to system/IN/OUT/ELAB
```

Case is not significant in text lines, but it is customary to write terms in lower case, and tags in upper case. The first occurrence of any given term or tag (either on a MAS file or in the text) determines the way in which `cico3` will write the term or tag in output.

`cico3` is 8-bit clean; input files can be written in any 8-bit character set (e.g., ISO-8859-1 for western countries), as long as the MAS rules provided use the same character set, and the system is correctly set up to handle that particular language. On UNIX, this typically means that the environment for `cico3` must assign an appropriate value to the `LANG` or `LC_CTYPE` variables.

7.2.2 Directives

Currently, `cico3` accepts two kind of directives: `#line` and `%config`. Each directive must be on a line on its own. The first directive has the following syntax:

`#line` *src_no tuple_no*

where *src_no* is the source line number that will be associated to the next text line to be read, and *tuple_no* is the number of the first tuple (or frame) to be emitted upon parsing the next text line. The `#line` directive is often used in support of *differential parsing*, when only parts of a larger document are being parsed, and `cico3` needs to be instructed on the position of the parsed blocks in the original document.

The `%config` directive allows the user to change almost all the parameters in CICO's parsing algorithm. The syntax of the directive is:

Item	Type	Default	Description
RULE_BONUS	float	+1.0	K_{RULE}
TERMS_SPECIFICITY_BONUS	float	+0.6	K_{TERMS}
CONSTS_SPECIFICITY_BONUS	float	+0.3	K_{CONSTS}
TAGS_SPECIFICITY_BONUS	float	+0.2	K_{TAGS}
NOSOL_SCORE	float	-100.0	VeryBadScore
COST_FINAL_DROPPED	float	-0.6	K_{REST}
COST_BAD_THRESHOLD	float	-4.0	BadThreshold
COST_BAD_IMPROVEMENT	float	-2.0	BadImprovement
COST_LEVEL_MULT	float	+0.3	LevelMult
REQ_TIMEOUT	int	30	Time in seconds before aborting a parse in progress
COST_INVERSION	float	-0.6	$K_{INVERSION}$
COST_DROPPED	float	-1.0	$K_{DROPPED}$
COST_DIST	float	-0.1	K_{DIST}
COST_OPTEMPTY	float	+0.03	K_{OPT}
MAX_DISTANCE	int	4	MaxDistance
USE_FRAMES	int	0	Output tuples (0) or frames (1)

Table 3: Configuration items for the `%config` directive.

`%config` *item_name item_value*

where *item_name* is the symbolic name of the configuration option that is being defined, and *item_value* is the value that is being assigned to the option. The available configuration items, their types, their default values, and a brief description, are listed in Table 3. As can be observed, most items corresponds directly to the parameters for the algorithm described in Section 3.

7.3 MAS file format

Files containing sets of MAS rules are commonly called *MAS files*, and conventionally given the extension `.mas`. These files contains rules that drive the parsing process performed by Cico. In these files, empty lines or lines beginning with `#` are ignored; the latter can be used to insert remarks in MAS files.

"Normal" lines contain the rules, one per line. Each rule has the form

```
Model ::  Action >> Substitution
```

The Action and Substitution parts can be omitted, together with the preceding `::` or `>>`, respectively. Thus, the following variants are also legal:

Model :: *Action*

Model >> *Substitution*

Model

As described in Section 3.1, the Model part can contain *fixed terms* and *variable terms*. Fixed terms are written entirely in upper case, and must match literally in the text. Variable terms are written in lower case, and can match any term in the text as long as the text term has a superset[9] of the tags of the model term. For example, the model

```
THE adj1/JJ noun/NN IS adj2/JJ
```

matches text fragments like

```
the lazy/JJ dog/NN/ANIMAL is fat/JJ/WEIGHT
```

The Action part also consists of a list of terms. When the rule is fired, the action part undergoes variables expansion (see below), and is output to `cico3`'s standard output in tuple or frame format, depending on the value of the `USE_FRAMES` configuration option. In any case, it is preceded by a unique numeric identifier (called a *tuple ID*) that is monotonically incremented for each tuple or frame output, and may be followed by a number of system tuples describing the match that caused the firing. The output format is described in more detail in Section 7.4.

Variables expansion is performed by substituting any term starting with the special variable dereferencing symbol `$` with the text term matched by any term with a similar name (minus the `$`) in the Model part. In the result, tags from the matched text and from the variable term are merged. Thus, when the MAS rule

```
THE adj1/JJ noun/NN IS adj2/JJ ::  ATT $noun/OBJ $adj1 $adj2
```

matches the text above, the output of the parser will be (apart from the numbering)

```
31 ATT dog/NN/ANIMAL/OBJ lazy/JJ fat/JJ/WEIGHT
```

If a variable term does not have a corresponding term in the model, its expanded value coincides with its name (including the `$`). The special variable `$ID` (in upper case) expands to the unique identifier of the tuple, preceded by the tuple reference marker `@`, e.g. `@31` in the case above.

The Substitution part, as the other parts, consists of a list of terms. When a rule is fired, its substitution part undergoes variables expansion, and the result is substituted in the text, replacing the matched fragment

[9]With due exceptions for special tags like `/0`, `/<`, or `/*`.

(see Section 3.4). The process is repeated for each node in the optimal parse tree, as determined by CICO's scoring system.

If the `USE_FRAMES` configuration option has value 1, variables expansion in Substitution parts works in a slightly different way. In particular, the name of the variable becomes part of the substitution, followed by an equal sign (`=`), followed by the value of the substitution as in the previous case. If `USE_FRAMES` is in effect at the time the previous rule is fired, the output would be

```
31 ATT noun=dog/NN/ANIMAL/OBJ adj1=lazy/JJ adj2=fat/JJ/WEIGHT
```

7.4 CICO's output

The output from `cico3` is a list of tuples (or frames, depending on the value of the `USE_FRAMES` configuration option), one per line, that are either generated by the system, or obtained from the Action parts of fired MAS rules, as described in the previous section.

For each tuple with ID *tid* coming from a rule action, `cico3` produces a `CICOFRAG` and a `CICOUNPARSE` tuple detailing, respectively, the fragment of the original text that has been matched by the current rule, according to the definition in Section 3.4, and the canonic form for the match, obtaining by substituting each term in the model with the corresponding matched term in the text. The signatures of these tuples are

```
id CICOFRAG @ref_id fragment...
id CICOUNPARSE @ref_id unparsed_model...
```

where `@ref-id` is the tuple ID of the tuple describing the action fired by the present rule. Moreover, other system tuples can appear, providing detailed information on the parsing process. Rests, as defined in Section 3.6, are recorded by tuples like

```
id CICOREST @ref_id begin-end rest...
id CICOFINALREST @ref_id final_rest...
```

`CICOREST` tuples record the terms that were discarded by the matching that fired the action *ref_id*. The position of the rest is given by *begin* and *end*, that are 1-based term positions relative to the source text; the terms that constitute the rest follows. A final rest, i.e., what is left after all the possible rules have been applied, is instead recorded by `CICOFINALREST` tuples.

After an entire text line has been parsed, CICO emits a `CICOSRCREQ` tuple, with signature

```
id CICOSRCREQ src_line_no @first_id @last_id text...
```

where *src_line_no* is the 1-based line number of the text that is being parsed (the line counter can be reset with the `#line` directive, see Section 7.2.2), *first_id* and *last_id* record which tuples have been produced from the text line, and *text* records the parsed text, in its original form.

If the parsing process lasts more than the current value of the `REQ_TIMEOUT` configuration option, `cico3` emits a `CICOTIMEOUT` tuple, with signature

```
id CICOTIMEOUT src_line_no
```

before any other tuple related to the source text indicated by *src_line_no*. In any case, after parsing a line of source text, `cico3` emits a tuple giving a number of statistics about the parsing process:

```
id CICOSTATS src_line_no nr nm ns d secs score
```

where *src_line_no* has the usual meaning, and the other values are defined as follows:

nr	number of MAS rules known
nm	total number of matches
ns	number of parse trees analyzed
d	max depth reached during analysis
secs	time elapsed, in seconds
score	score of the best parse tree

The last system tuple is used in conjunction with speculative tagging (see Section 4.1). Every time a speculative tag `/?` is used in a match, the actual tags that are speculatively assigned to the term are recorded in a `CICOSPECTAG` tuple:

```
id CICOSPECTAG term/TAGS
```

where *term* is the term that was originally marked `/?`, and */TAGS* is the set of tags that Cico has speculatively assigned to the term for the purpose of maximizing the matching score.

Finally, we recall that a successful run of the parser is marked by emitting a `CICOSRFINISHED` tuple at the end of the parsing process, and that timeouts while parsing a single text line are marked by `CICOTIMEOUT` tuples. Both types of tuples have already been described in Section 7.1.

7.5 Error messages and diagnostics

In a normal run, `cico3` outputs the tuples or frames describing the parse trees to *stdout*, and is completely silent otherwise. However, if operational errors are found, or other exceptional events happen, `cico3` emits error or warning messages to *stderr*, without disturbing *stdout*, and tries to continue if possible.

The various messages, together with a short explanation, are listed in the following.

7.5.1 Warning messages

Warning messages have the form

> *file*:*line* -W- *text*

where *file* and *line* identify the module and the exact spot inside `cico3` where the messages was generated, the `W` classifies the message as a warning, and the *text* describes the error. In any case, parsing continues after a warning message, albeit reduced functionality may be obtained as a consequence of the error described by the message.

Cannot open rules file *file*. The MAS file *file*, as given on the command line, could not be opened. Parsing will continue, but rules in *file* were not read. If *file* was the only argument, parsing proceeds with an empty set of MAS rules — the output may well be discarded in this case.

7.5.2 Error messages

Error messages have the form

> *file*:*line* -E- *text*

where the various field have the same meaning as for warning messages, and the `E` indicates an error message. Parsing is stopped upon encountering the error described by *text*, and `cico3` exits with a return code of 20.

matching array too small.
termstring array too small.
out of token space.
out of rules space.
out of tag space.
solution stack depth exceeded. `cico3` uses a number of fixed-length structures, that are usually exaggeratedly large for the kind of applications at which Cico is geared. If any of these compiled-in limits is reached, `cico3` emits one of the errors above and aborts the compilation. The only possible solutions are reducing the size of some element of the input (e.g., number of MAS rules, length of a single text line, etc.), or recompiling `cico3` with larger values.

out of memory. `cico3` was unable to obtain from the system some dynamically allocated memory that is needed for the parsing process. This typically means that the system has run out of (physical and virtual) memory, or that the operating system is imposing stricter limits on the process that runs `cico3`[10], and that such limits have been reached.

unknown mt type.
looking for a pronoun reference, but not a pronoun.
asserted pronoun reference not found.
invalid token. These messages indicate that something really bad has happened, and that sanity checks inside `cico3` have caught the error. Most probably, occurrences of these messages indicate bugs in `cico3`, and the author would be delighted to hear about them.

8 Tools and utilities

The distribution of CICO is complemented by a small number of utilities, that help in preparing correctly-formatted files to provide as input to `cico3`, and other administrative tasks. These utilities — that can be found in executable forms where `cico3` itself is found — are described in the following.

8.1 `gloss`

The `gloss` utility takes as input a *glossary*, i.e., a document listing a set of words with associated tags and synonyms, and a *source document*, essentially consisting of free-form text. The tool looks for occurrences of the terms defined in the glossary in the source document, and substitutes every occurrence (of the base term or of any synonym) with a properly tagged version of the base term. The resulting document is then output, and can be fed to `cico3` to build a parse tree for the source document. Thus, `gloss` implements the glossary substitution process described in Section 2.3.

The invocation syntax for the tool is

```
gloss [glossary]
```

if the *glossary* argument is not given, `gloss` tries to open a glossary file named `glossario.txt` in the current directory. It then reads *stdin* and applies the *glossary* to each line, outputting the results to *stdout*; any error message is output to *stderr*, as usual.

A glossary file is constituted of a series of lines. Blank lines and lines starting with `#` are ignored; the remaining lines have the format

```
term [ /TAG...  ]  [:  synonym [, synonym...]  ]  .
```

a BNF grammar for glossary files and some example of their format are given in Section 2.3.

[10] This can be obtained, on UNIX, by using the `ulimit` and `limit` shell commands, or by calling the `ulimit(3)`, `setrlimit(2)` or equivalent system functions.

For historical reasons, `gloss` will also read a few other files with special names (containing a “system” glossary) if these are found in the current directory. The use of this feature, however, is discouraged, and support for it will be removed in a future release in favor of a more complete command line syntax.

8.2 glossdump

The `glossdump` utility transforms a glossary file into a set of tuples in the same format as those produced by `cico3`. This can be useful if glossary entries are to be processed together with the parse trees by some other program.

The invocation syntax for `glossdump` is

```
glossdump glossary baseid type
```

where *glossary* is the name of a glossary file, as for the `gloss` command, *baseid* is the numeric *id* for the first tuple to be output, and *type* is a single `u` or `s` letter. A type of `u` specifies the “user” glossary, read from *glossary*, while a type of `s` specifies the “system” glossary, read from an implicit set of files.

The command does not read any input from *stdin*. It writes to *stdout* a set of tuples, describing the user or system glossary, according to the *type*. These tuples have the signature

id *x*`GLOSSENTRY` *b/TAGs* *b* s_1 ... s_n

where *id* is the monotonously increasing identifier of the tuple, starting with the id provided on the command line, and x is either `U` or `S`, depending on whether the user or system glossary is being output (according to the *type*). Following that, the base term b is output, together with any tag, followed in turn by the whole set of alternative synonyms for the term. For example, a glossary entry like

```
vt100/IN/OUT: terminal, user terminal.
```

would result in a tuple like

```
32 UGLOSSENTRY vt100/IN/OUT vt100 terminal user_terminal
```

8.3 rulesgloss

The `rulesgloss` utility supports the extraction of an “implicit” glossary from a set of MAS rules. Its syntax is

```
rulesgloss [MAS files...]
```

The tool reads a set of MAS rules from the *MAS files*, and outputs to *stdout* a list of all the constant terms found in the model parts of the rules. The list is unsorted and may include repetitions. To obtain a cleaner output, the following UNIX command line could be used:

```
rulesgloss MAS files | sort | uniq
```

8.4 destruct

`destruct` (as in de-structure) converts an input file with embedded list markers to a source file suitable for `cico3` in which explicit list markers have been substituted by context delimiters markers (denoted by `<` and `>`), according to the process outlined in Section 3.8.

List items are introduced by one or more dashes at the beginning of a line; progressively deeply nested lists are identified by an increasing number of dashes before each element. `destruct` introduces a context marker around each item, and a marker around an entire (sub-)list. Thus, an input like

```
a
b
- c
- d
-- e
-- f
g
```

becomes

```
a b < < c > < d > < < e > < f > > > g
```

The output of `destruct` can then be fed to `cico3`, which will correctly interpret the markers introduced as context delimiters.

Part III
Sample Rule Sets

9 Domain specific rules: NASA requirements

In this section we present the outcome of the parsing process performed by Cico on a typical requirement for a software system. The requirement below is taken from [17], and is part of the Requirements Specification document for the on-board software of the International Space Station (further details are provided in [8] and [9]). The document analyzed was written under very precise rules and assumptions. This made it possible to write a very compact domain-based grammar for parsing it. Also, the vast majority of the sentences had a fixed subject ("NCS", the Node Control Software), and a standard structure:

> If *some combination of events*, the NCS shall *perform some action, under certain constraints.*

Thus, most of the rules in the rule set are devoted to describing the structure of events , actions, and constraints, while only few rules deal with properly grammatical issues.

Only a few semantic classes were needed for this rule set. In particular, the `/INF` tag denotes items that have some information content (i.e., data items), `/RELOP` indicates relational operators (e.g., "greater than"), and `/MSGTYPE` serves to indicate the various kind of messages ("warning", "error", "panic", etc.). `/NUM` and `/UNIT` denote numerical quantities (typically, numerals) and measure units (e.g., "psi" for "per square inch"). The NCS itself was tagged `/ELAB`, to indicate its ability to perform computations and to take decisions.

Tagging was performed automatically by the `gloss` tool (see Section 8.1), after a glossary was built from the tables in the original document. The fragment of the glossary needed for parsing the requirement below is shown in Figure 3.

Figure 5 shows in graphical form the parse tree obtained by applying the rules in Figure 4 to the source text in Figure 2. The tree was obtained by removing from `cico3`'s output the various system tuples listed in Section 7.4, and by drawing an edge for each referent of the form `@`n from the tuple containing the reference to the tuple with id n.

3.2.2.1.3 Cabin Pressure Monitoring Processing

Processing Startup and Termination

...

d. If the ACS N1S2 Cabin Pressure FDIR State is equal to ENABLED, then upon receipt of ACS N1S2 Cabin Pressure HW Data less than the Node 1 Cabin Pressure Lower Limit (initial: 13.9 psia) for 3 consecutive acquisitions, the NCS shall within 1.1 seconds:

   (1) Set the ACS N1S2 Cabin Pressure Lower Limit Warning State (initial value: FALSE) to TRUE, and

   (2) Issue a warning level alarm (message: "Node 1 Cabin Pressure Lower Limit Warning Violation") in accordance with the section on "annunciate alarms".

Figure 2: A sample requirement from [17].

Terms from the data dictionary in [17]	
ACS N1S2 Cabin Pressure FDIR State	/INF
ACS N1S2 Cabin Pressure HW Data	/INF
Node 1 Cabin Pressure Lower Limit	/INF
ACS N1S2 Cabin Pressure Lower Limit Warning State	/INF
Other terms from the domain	
NCS	/ELAB/SYS
warning level	/MSGTYPE
ENABLED	/INF/CONST
psia	/UNIT

Figure 3: Glossary for the requirement above.

Model	*Action*	*Substitution*
Operations		
SET var/INF TO val/INF	SET $var $val	$ID/OPERATION
ISSUE type/MSGTYPE MESSAGE text	ISSUEMESG $type $text	$ID/OPERATION
op1/OPERATION AND op2/OPERATION	CONJ $op1 $op2	$ID/OPERATION/CONJ
op1/OPERATION op2/OPERATION/CONJ	CONJ $op1 $op2	$ID/OPERATION/CONJ
References to other documents		
operation/OPERATION IN ACCORDANCE WITH standard/STANDARD	STANDARDREF $operation $standard	$ID/OPERATION
SECTION ON title	SECTIONREF $title	$ID/STANDARD
Actions, events, and timing		
operation/OPERATION WITHIN time/TSPAN	WITHIN $time $operation	$ID/OPERATION
operator/ELAB SHALL operation/OPERATION	DOOP $operator $operation	$ID/ACT
val1/INF op/RELOP val2/INF	RELOP $val1 $op $val2	$ID/EVT
RECEIPT OF val1/INF op/RELOP val2/INF	CONDRECEIPT $val1 $op $val2	$ID/EVT
IF cond/EVT act/ACT	DEPC $cond $act	$ID/ACT
UPON event/EVT act/ACT	DEPT $cond $act	$ID/ACT
n/NUM CONSECUTIVE ACQUISITIONS	TSPAN $n sampling	$ID/TSPAN
n/NUM SECONDS	TSPAN $n secs	$ID/TSPAN
event/EVT FOR span/TSPAN	EVTFORSPAN $event $span	$ID/EVT
Default values and numeric quantities		
item/INF INITIAL value/INF	DEFAULTVAL $item $value	$item
number/NUM unit/UNIT	UNITVAL $number $unit	$ID/INF

Figure 4: MAS rules used in parsing the requirement in Fig. 2

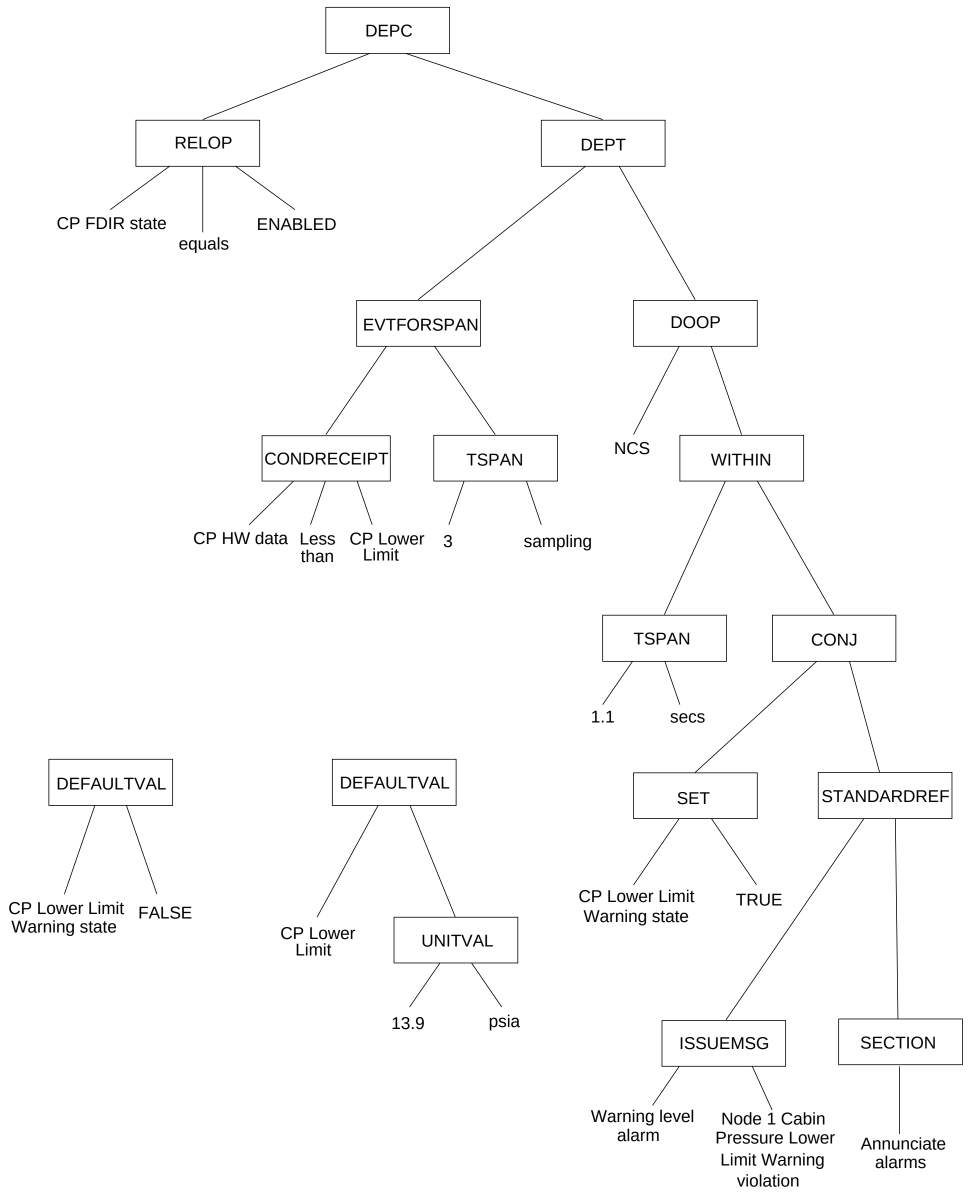

Figure 5: Parse trees for the sample requirement.

10 Grammar based rules: the CARL case

In this section, we present a number of sample rules taken from the CARL project [19]. These rules expect a source text in English, with terms tagged according to their part-of-speech role. The tags used are essentially those from the Penn Treebank set, with minor modifications. The rules are geared towards parsing of NL requirements and translating them into logic form for further processing.

As can be noted, the rule set makes extensive use of optional terms, denoted by the special tag `/O` (see, for example, among the basic predicates in Figure 6). In particular, negative forms (`/NOT`) and complements introduced by prepositions (`/PREP`) are treated as optional components in a sentence (`/SENT`).

Also of interest is the use of rewriting rules to transform the input into a simpler form. For example, consider the rule

```
a/TERM vbing/VB/G >> $a AND $a $vbing/-G
```

This rule matches a term (i.e., an optional article, followed by one or more optional adjectives, followed by a noun — see Figure 6), followed by a gerund. If a match is found, the fragment is rewritten as the same term, conjuncted with itself followed by the verb in its present form. Thus, a statement like

> If an operator receives **a phone call concerning** a medical emergency...

is rewritten as

> If an operator receives **a phone call and the phone call concerns** a medical emergency...

that can be more easily parsed with the remaining rules.

Finally, the last rule in Figure 7 deserves some explanation. The rule is essentially saying that a verb in the past participle (that is often used adjectivally) *can* be considered as an adjective. It is important to notice that Cico can apply this rule if the transformation improves the overall scoring of the parsing tree, but that it is also free to ignore it — and to maintain the verb as such — if this strategy yields a better parsing score. So, for example, in

> The operator should only dispatch serviced ambulances.

"serviced" is treated as an adjective, while in

> If the technician has serviced the ambulance...

Model	*Action*	*Substitution*
Basic predicates		
a/TERM BE/VB neg/NOT/0 b/TERM compl/PREP/0	SENT $neg ISA $a $b $compl	$ID/SENT
a/TERM BE/VB neg/NOT/0 b/JJ compl/PREP/0	SENT $neg $b $a $compl	$ID/SENT
a/TERM BE/VB EITHER b/TERM OR c/TERM	XOR2 $b $a $c $a	$ID/SENT
a/TERM BE/VB EITHER b/JJ OR c/JJ	XOR2 $b $a $c $a	$ID/SENT
a/TERM v/VB EITHER b/TERM OR c/TERM	XOR3 $v $a $b $v $a $c	$ID/SENT
subj/TERM neg/NOT/0 verb/VB obj/TERM/0 compl/PREP/0	SENT $neg $verb $subj $obj $compl	$ID/SENT
existential/EX BE/VB a/TERM	EXIST $a	$ID/SENT
Prepositions and complements		
to/TO term/TERM	PREP $to $term	$ID/PREP
prep/IN term/TERM	PREP $prep $term	$ID/PREP
prep1/PREP prep2/PREP	PCONJ $prep1 $prep2	$ID/PREP/CONJ
Implication, equivalence, and such		
IF a/SENT THEN/0 b/SENT	IMP $a $b	$ID/SENT
a/SENT IF b/SENT	IMP $b $a	$ID/SENT
a/SENT ONLY IF b/SENT	EQV $b $a	$ID/SENT
a/SENT WHEN b/SENT	IMP $b $a	$ID/SENT
WHEN a/SENT b/SENT	IMP $a $b	$ID/SENT
a/SENT ONLY WHEN b/SENT	EQV $b $a	$ID/SENT
EITHER a/SENT OR b/SENT	XOR $a $b	$ID/SENT
a/SENT UNLESS b/SENT	XOR $a $b	$ID/SENT
Conjunction, disjunction		
a/SENT AND b/SENT	AND $a $b	$ID/SENT/AND
a/SENT OR b/SENT	OR $a $b	$ID/SENT/OR
a/SENT b/SENT/AND	AND $a $b	$ID/SENT/AND
a/SENT b/SENT/OR	OR $a $b	$ID/SENT/OR
a1/JJ a2/JJ	CONJ $a1 $a2	$ID/JJ/CONJ
Back to grammar school!		
a/DT/0 neg/NOT/0 adj/JJ/0 n/NN	TERM $a $neg $adj $n	$ID/TERM
DO v/VB	MOOD absolute	$v
DO neg/NOT v/VB	MOOD absolute	$neg $v
aux/MD v/VB	MOOD $aux	$v
aux/MD neg/NOT v/VB	MOOD $aux	$neg $v

Figure 6: MAS rules used in the CARL project.

Model	*Action*	*Substitution*
Pronouns		
`it/PP`	`TERM _ _ _ $it`	`$ID/PPREF/TERM`
A few rewriting tricks		
`a/TERM vbing/VB/G`	`$a AND $a $vbing/-G`	
`a/TERM neg/NOT vbing/VB/G`	`$a AND $a $neg $vbing/-G`	
`a/TERM THAT vb/VB`	`$a AND $a $vb`	
`a/TERM THAT $neg/NOT vb/VB`	`$a AND $a $neg $vb`	
`a/TERM THAT IS/VB $neg/NOT adj/JJ`	`$a AND $a IS $neg $adj`	
`a/TERM THAT IS/VB adj/JJ`	`$a AND $a IS $adj`	
`vb/VB/N`	`$vb/-VB/-N/JJ`	

Figure 7: MAS rules used in the CARL project (continued).

the rule is not applied, and "serviced" is treated as a verb. Notice that while the different roles of "serviced" in the two sentences above could have been discovered during parts-of-speech tagging, the fuzziness of the parsing process in CICO makes it possible for the tool to maintain a certain robustness even when tagging errors are present.

References

[1] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the Twelfth IEEE Conference on Automated Software Engineering*, pages 36–45. IEEE Press, Nov. 1997.

[2] V. Ambriola and V. Gervasi. Experiences with domain-based parsing of natural language requirements. In G. Fliedl and H. C. Mayr, editors, *Proceedings of the Fourth International Conference on Applications of Natural Language to Information Systems*, number 129 in OCG Schriftenreihe (Lecture Notes), pages 145–148, 1999.

[3] S. Argamon, I. Dagan, and Y. Krymolowski. A memory-based approach to learning shallow natural language patterns. In C. Boitet and P. Whitelock, editors, *Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics*, pages 67–73, San Francisco, California, 1998. Association for Computational Linguistics, Morgan Kaufmann Publishers.

[4] R. Cole, J. Mariani, H. Uszkoreit, G. B. Varile, A. Zaenen, A. Zampolli, and V. Zue, editors. *Survey of the State of the Art in Human Language Technology.* Studies in Natural Language Processing. Cambridge University Press, Mar. 1998.

[5] 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.

[6] 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.

[7] V. Gervasi. *Environment Support for Requirements Writing and Analysis.* PhD thesis, University of Pisa, February 2000.

[8] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements: a case study. In *Proc. of the 4th International Conference on Requirements Engineering*, pages 140–148, June 2000.

[9] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements. *Software Practice & Experience*, 2001. (to be published).

[10] U. Hahn and K. Schnattinger. Knowledge mining from textual sources. In F. Golshani and K. Makki, editors, *Proceedings of the 6th International Conference on Information and Knowledge Management (CIKM-97)*, pages 83–90, New York, Nov. 10–14 1997. ACM Press.

[11] 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.

[12] N. Juristo, A. M. Moreno, and M. López. How to use linguistic instruments for object-oriented analysis. *IEEE Software*, 17(3):80–89, May/June 2000.

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

[14] 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.

[15] B. Macias and S. G. Pulman. *Natural Language Processing for Requirements Specification*, pages 57–89. Chapman and Hall, 1993.

[16] L. Mich. NL-OOPS: From natural language requirements to object oriented requirements using the natural language processing system LOLITA. *Journal of Natural Language Engineering*, 2(2):161–187, 1996.

[17] NASA/Boeing. Software requirements specification for the NCS MDM CSCI. International Space Station Document S684-10174, Apr. 1997.

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

[19] D. Zowghi, V. Gervasi, and A. McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Proceedings of the Australia-Pacific Software Engineering Conference*, Dec. 2001.

Contents

1 Introduction **3**

I Parsing Technique **7**

2 Input format **7**
- 2.1 Typographical adjustments 7
- 2.2 Format handling 8
- 2.3 Glossary substitution 8

3 Parsing algorithm **9**
- 3.1 MAS rules 10
- 3.2 Matchings 10
- 3.3 Match scoring 11
- 3.4 Rule firing 12
- 3.5 Rule scoring 13
- 3.6 Rests and rest scoring 13
- 3.7 Basic parsing algorithm 13
- 3.8 Parsing contexts 15
- 3.9 Full parsing algorithm 16

4 Special forms of tagging **18**
- 4.1 Speculative tagging 18
- 4.2 Lookup references 20
- 4.3 Wildcard tagging 21
- 4.4 Optional terms 22
- 4.5 Negative tags 23

5 Implementation and evaluation **24**
- 5.1 Implementation issues and optimizations 24
- 5.2 Coverage 25
- 5.3 Performance 26
- 5.4 Cost model parameters 28

6 Conclusions **29**

II Cico User Manual **32**

7 The parser **32**
- 7.1 Invocation syntax 32
- 7.2 Input file format 33
  - 7.2.1 Text lines 33

7.2.2 Directives 33
7.3 MAS file format 34
7.4 Cico's output 36
7.5 Error messages and diagnostics 37
7.5.1 Warning messages 38
7.5.2 Error messages 38

8 Tools and utilities **39**
8.1 `gloss` 39
8.2 `glossdump` 40
8.3 `rulesgloss` 40
8.4 `destruct` 41

III Sample Rule Sets **42**

9 Domain specific rules: NASA requirements **42**

10 Grammar based rules: the CARL case **46**